The Tiger Compiler Project

Edition April 16, 2018

Akim Demaille, Roland Levillain and Etienne Renault

This document presents the EPITA version of the Tiger project. This revision, , was last updated April 16, 2018.

Nul n'est censé ignorer la loi.

Everything exposed in this document is expected to be known.

This document, revision of April 16, 2018, details the various tasks EPITA students must complete. It is available under various forms:

- Assignments in a single HTML file[1].
- Assignments in several HTML files[2].
- Assignments in PDF[3].
- Assignments in text[4].
- Assignments in Info[5].

[1] `https://www.lrde.epita.fr/~tiger/assignments.html`.
[2] `https://www.lrde.epita.fr/~tiger/assignments.split`.
[3] `https://www.lrde.epita.fr/~tiger/assignments.pdf`.
[4] `https://www.lrde.epita.fr/~tiger/assignments.txt`.
[5] `https://www.lrde.epita.fr/~tiger/assignments.info`.

Table of Contents

1 Introduction

This document presents the Tiger Project as part of the EPITA[1] curriculum. It aims at the implementation of a Tiger compiler (see Section 5.2 [Modern Compiler Implementation], page 233) in C++.

1.1 How to Read this Document

If you are a newcomer, you might be afraid by its sheer size. Don't worry, but in any case, do not give up: as stated in the very beginning of this document,

Nul n'est censé ignorer la loi.

That is to say everything exposed in this document is considered to be known. If it is written but you didn't know, you are wrong. If it is not written *and* was not clearly reported in the news, we are wrong.

Basically this document contains three kinds of information:

Initial and Permanent
: What you must read and know since the very beginning of the project. This includes most the following chapters: Chapter 1 [Introduction], page 3, (except the Section 1.4 [History], page 6, section), Chapter 2 [Instructions], page 29, and Section 2.7 [Evaluation], page 51.

Incremental
: You should read these parts as and when needed. This includes mostly Chapter 4 [Compiler Stages], page 71.

Auxiliary
: This information is provided to help you: just go there when you feel the need, Chapter 5 [Tools], page 233, and Chapter 3 [Source Code], page 55. If you want to have a better understanding of the project, if you are about to criticize something, be sure to read Section 1.4 [History], page 6, beforehand.

There is additional material on the Internet:

- The Wiki page for the Tiger Compiler Project[2] is the official home page of the project. It holds related material (e.g., links).
- The packages of the tools that we use (Bison, Autoconf etc.) can be found in the Tiger download area[3].
- The developer documentation of the Tiger Compiler[4].
- Most of the provided material (lecture notes, older exams, current tarballs etc.) is in the Tiger area[5].

1.2 Why the Tiger Project

This project is quite different from most other EPITA projects, and has aims at several different goals, in different areas:

Several iterations
: This project is about the only one with which you will live for 4 months (6 months for the brave ones), with the constant needs to fix errors found in earlier stages.

[1] `http://www.epita.fr/`.
[2] `http://tiger.lrde.epita.fr/`.
[3] `http://www.lrde.epita.fr/~tiger/download`.
[4] `https://www.lrde.epita.fr/~tiger/tc-doc/`.
[5] `https://www.lrde.epita.fr/~tiger/`.

Complete Project

While the evaluation of most student projects is based on the code, this project restores the deserved emphasis on *documentation* and *testing*. Because of the duration of the project, you will value the importance of a good (developer's) documentation (why did we write this 4 months ago?), and of a good test suite (why does TC-2 fails now that we implemented TC-4? When did we break it?).

This also means that you have to design a test suite, and maintain it through out the project. *The test suite is an integral part of the project.*

Team Management

The Tiger Compiler is a long project, running from February to May (and optionally further). Each three person team is likely to experience nasty "human problems". This is explicitly a part of the project: the team management is a task you have to address. That may well include exclusion of lazy members.

C++ C++ is by no means an adequate language to *study* compilers (C would be even worse). Languages such as Haskell[6], Ocaml[7], Stratego[8] are much better suited (actually the latter is even designed to this end). But, as already said, the primary goal is not to learn how to write a compiler: for an EPITA student, learning C++, Design Patterns, and Object Oriented Design is much more important.

Note, however, that implementing an industrial strength compiler in C++ makes a lot of sense[9]. Bjarne Stroustrup's list of C++ Applications[10] mentions Section 5.5 [GCC], page 249, Section 5.6 [Clang], page 250, and LLVM, Metrowerks (CodeWarrior), HP, Sun, Intel, M$ as examples.

Understanding Computers

Too many students still have a very fuzzy mental picture of what a computer is, and how a program runs. Studying compilers helps understanding how it works, and therefore *how to perform a good job*. Although most students will never be asked to write a single line of assembly during their whole lives, knowing assembly is also of help. See [Bjarne Stroustrup], page 237, for instance, says:

> Q: What is your opinion, is knowing assembly language useful for programmers nowadays?
>
> BS: It is useful to understand how machines work and knowing assembler is almost essential for that.

English English is *the* language for this project, starting with this very document, written by a French person, for French students. You cannot be a good computer scientist with absolutely no fluency in English. The following quote is from Bjarne Stroustrup, who is danish ([The Design and Evolution of C++], page 245, 6.5.3.2 Extended Character Sets):

> English has an important role as a common language for programmers, and I suspect that it would be unwise to abandon that without serious consideration.

---

[6] `http://www.haskell.org`.

[7] `http://caml.inria.fr`.

[8] `http://www.stratego-language.org`.

[9] The fact that the compiler compiles C++ is virtually irrelevant.

[10] `http://www.stroustrup.com/applications.html`.

Any attempt to break the importance of English is wrong. For instance, *do not translate this document nor any other*. Ask support to the Yakas, or to the English team. By the past, some oral and written examinations were made in English. It may well be back some day. Some books will help you to improve your English, see [The Elements of Style], page 246.

Compiler The project aims at the implementation of a compiler, but *this is a minor issue*. The field of compilers is a wonderful place where most of computer science is concentrated, that's why this topic is extremely convenient as long term project. But *it is not the major goal*, the full list of all these items is.

The Tiger project is not unique in these regards, see [Cool - The Classroom Object-Oriented Compiler], page 239, for instance, with many strikingly similar goals, and some profound differences. See also [Making Compiler Design Relevant for Students who will (Most Likely) Never Design a Compiler], page 243, for an explanation of why compilation techniques have a broader influence than they seem.

1.3 What the Tiger Project is not

This section could have been named "What Akim did not say", or "Common misinterpretations".

The first and foremost misinterpretation would be "Akim says C sucks and is useless". Wrong. C sucks, definitely, but let's face it: C is **mandatory** in your education. The fact that C++ is studied afterward does not mean that learning C is a loss of time, it means that since C is basically a subset of C++ it makes sense to learn it first, it also means that (let it be only because it is a superset) C++ provides additional services so it is often a better choice, but even more often *you don't have the choice*.

C++ is becoming a common requirement for programmers, so you also have to learn it, although it "features" many defects (but heredity was not in its favor...). It's an industrial standard, so learn it, and learn it well: know its strengths and weaknesses.

And by the way, of course C++ sucks++.

Another common rumor in EPITA has it that "C/Unix programming does not deserve attention after the first period". Wrong again. First of all its words are wrong: it is a legacy belief that C and Unix require each other: *you can implement advanced system features using other languages than C* (starting with C++, of course), and of course *C can be used for other tasks than just system programming*. For instance Bjarne Stroustrup's list of C++ Applications[11] includes:

Apple OS X is written in a mix of language, but a few important parts are C++. The two most interesting are:

- Finder
- IOKit device drivers. (IOKit is the only place where we use C++ in the kernel, though.)[...]

Ericsson

- TelORB - Distributed operating system with object oriented

Microsoft Literally everything at Microsoft is built using recent flavors of Visual C++. The list would include major products like:

- Windows XP
- Windows NT (NT4 and 2000)

[11] `http://www.stroustrup.com/applications.html`.

- Windows 9x (95, 98, Me)
- Microsoft Office (Word, Excel, Access, PowerPoint, Outlook)[...]
- Visual Studio

CDE
: The CDE desktop (the standard desktop on many UNIX systems) is written in C++.

Mozilla
- Firefox
- Thunderbird

Adobe Systems
: All major applications are developed in C++:
- Photoshop
- Illustrator
- Acrobat

Know C. Learn when it is adequate, and why you need it.

Know C++. Learn when it is adequate, and why you need it.

Know other languages. Learn when they are adequate, and why you need them.

And then, if you are asked to choose, make an educated choice. If there is no choice to be made, just deal with Real Life.

1.4 History

The Tiger Compiler Project evolves every year, so as to improve its infrastructure, to demonstrate more instructional material and so forth. This section tries to keep a list of these changes, together with the most constructive criticisms from students (or ourselves).

If you have information, including criticisms, that should be mentioned here, please send it to us.

The years correspond to the class, e.g., Tiger 2005 refers to EPITA class 2005, i.e., the project ran from October 2002 to July (previously September) 2003.

1.4.1 Fair Criticism

Before diving into the history of the Tiger Compiler Project in EPITA, a whole project in itself for ourselves, with experimental tries and failures, it might be good to review some constraints that can explain why things are the way they are. Understanding these constraints will make it easier to criticize actual flaws, instead of focusing on issues that are mandated by other factors.

Bear in mind that Tiger is an instructional project, the purpose of which is detailed above, see Section 1.2 [Why the Tiger Project], page 3. Because the input is a stream of students with virtually no knowledge whatsoever in C++, and our target is a stream of students with good fluency in many constructs and understanding of complex matters, we have to gradually transform them via intermediate forms with increasing skills. In particular this means that by the end of the project, evolved techniques can and should be used, but at the beginning only introductory knowledge should be needed. As an example of a consequence, we cannot have a nice and high-tech AST.

Because the insight of compilers is not the primary goal, when a choice is to be made between (i) more interesting work on compiler internals with little C++ novelty, and (ii)

providing most of this work and focusing on something else, then we are most likely to select the second option. This means that the Tiger Project is doomed to be a low-tech featureless compiler, with no call graph, no default optimization, no debugging support, no bells, no whistles, and even no etc. Hence, most interested students will sometimes feel we "stole" the pleasure to write nice pieces of code from them; understand that we actually provided code to the *other* students: you are free to rewrite everything if you wish.

1.4.2 Tiger 2002

This is not standard C++
: We used to run the standard compiler from NetBSD: `egcs` 1.1.2. This was not standard C++ (e.g., we used to include '`<iostream.h>`', we could use members of the `std` name space unqualified etc.). In addition, we were using `hash_map` which is an SGI extension that is not available in standard C++. It was therefore decided to upgrade the compiler in 2003, and to upgrade the programming style.

Wrapping a tarball is impossible
: During the first edition of the Tiger Compiler project, students had to write their own Makefiles — after all, knowing Make is considered mandatory for an Epitean. This had the most dramatic effects, with a wide range of creative and imaginative ways to have your project fail; for instance:

  - Forget to ship some files
  - Ship object files, or even the executable itself. Needless to say that NetBSD executables did not run properly on Akim's GNU/Linux box.
  - Ship temporary files (`*~`, `#*#`, etc.).
  - Ship core dumps ("Wow! This *is* the heck of an heavy tarball...").
  - Ship tarballs in the tarball.
  - Ship *tarballs of other groups* in the tarball. It was then hard to demonstrate they were not cheating :)
  - Have incorrect dependencies that cause magic failures.
  - Have completely lost confidence in dependencies and Make, and therefore define the `all` target as first running `clean` and then the actual build.

  As a result Akim grew tired of fixing the tarballs, and in order to have a robust, efficient (albeit some piece of pain in the neck sometimes) distribution[12] we moved to using Automake, and hence Autoconf.

  There are reasons not to be happy with it, agreed. But there are many more reasons to be sad without it. So Autoconf and Automake are here to stay.

  Note, however, that you are free to use another system if you wish. Just obey the standard package interface (see Section 2.6 [Submission], page 51).

The `SemantVisitor` is a nightmare to maintain
: The `SemantVisitor`, which performs both the type checking and the translation to intermediate code, was near to impossible to deliver in pieces to the students: because type checking and translation were so much intertwined, it was not possible to deliver as a first step the type checking machinery template, and then the translation pieces. Students had to fight with non applicable patches. This was fixed in Tiger 2003 by splitting the `SemantVisitor`

[12] See the shift of language? From tarball to distribution.

into `TypeVisitor` and `TranslationVisitor`. The negative impact, of course, is a performance loss.

Akim is tired during the student defenses
: Seeing every single group for each compiler stage is a nightmare. Sometimes Akim was not enough aware.

1.4.3 Tiger 2003

During this year, Akim was helped by:

Comaintainers
: Alexandre Duret-Lutz, Thierry Géraud.

Submission dates were:

Stage	Submission
TC-1	Monday, December 18th 2000 at noon
TC-2	Friday, February 23th 2001 at noon
TC-3	Friday, March 30th 2001 at noon
TC-4	Tuesday, June 12th 2001 at noon
TC-5	Monday, September 17th 2001 at noon

Some groups have reached TC-6.

Criticisms include:

The C++ compiler is broken
: Akim had to install an updated version of the C++ compiler since the system team did not want non standard software. Unfortunately, NetBSD turned out to be seriously incompatible with this version of the C++ compiler (its `crt1.o` dumped core on the standard stream constructors, way before calling `main`). We had to revert to using the bad native C++ compiler.

  It is to be noted that some funny guy once replaced the `g++` executable from Akim's account into '`rm -rf ~`'. Some students and Akim himself have been bitten. The funny thing is that this is when the system administration realized the teacher accounts were not backed up.

  Fortunately, since that time, decent compilers have been made available, and the Tiger Compiler is now written in strictly standard C++.

The AST is rigid
: Because the members of the AST objects were references, it was impossible to implement any change on it: simplifications, optimization etc. This is fixed in Tiger 2004 where all the members are now pointers, but the interface to these classes still uses references.

Akim is even more tired during the student defenses
: Just as the previous year, see Section 1.4.2 [Tiger 2002], page 7, but with more groups and more stages. But now there are enough competent students to create a group of assistants, the Yakas, to help the students, and to share the load of defenses.

Upgrading is not easy
: Only tarballs were submitted, making upgrades delicate, error prone, and time consuming. The systematic use of patches between tarballs since the 2004 edition solves this issue.

Upgraded tarballs don't compile
: Students would like at least to be able to compile a tarball with its holes. To this end, much of the removed code is now inside functions, leaving just what it needed to satisfy the prototype. Unfortunately this is not very easy to do, and conflicts with the next complaint:

Filling holes is not interesting
: In order to scale down the amount of code students have to write, in order to have them focus on instructional material, more parts are submitted almost complete except for a few interesting places. Unfortunately, some students decided to answer the question completely mechanically (copy, paste, tweak until it compiles), instead of focusing of completing their own education. There is not much we can do about this. Some parts will therefore grow; typically some files will be left empty instead of having most of the skeleton ready (prototypes and so forth). This means more work, but more interesting I (Akim) guess. But it conflicts with the previous item...

1.4.4 Tiger 2004

During this year, Akim was helped by:

Comaintainers
: Alexandre Duret-Lutz, Raphaël Poss, Robert Anisko, Yann Régis-Gianas,

Assistants
: Arnaud Dumont, Pascal Guedon, Samuel Plessis-Fraissard,

Students
: Cédric Bail, Sébastien Broussaud (Darks Bob), Stéphane Molina (Kain), William Fink.

Submission dates were:

Stage	**Submission**
TC-2	Tuesday, March 4th 2002 at noon
TC-3	Friday, March 15th 2002 at noon
TC-4	Friday, April 12th 2002 at noon
TC-5	Friday, June 14th 2002, at noon
TC-6	Monday, July 15th 2002 at noon

Criticisms include:

The driver is not maintainable
: The compiler driver was a nightmare to maintain, extend etc. when delivering additional modules etc. This was fixed in 2005 by the introduction of the `Task` model.

No sane documentation
: This was addressed by the use of Doxygen in 2005.

No UML documentation
: The solution is yet to be found.

Too many visitors
: It seems that some students think there were too many visitors to implement. I (Akim) do not subscribe to this view (after all, why not complain that "there are too many programs to implement", or, in a more C++ vocabulary "there are too many classes to implement"), nevertheless in Tiger 2005 this was addressed by making the `EscapeVisitor` "optional" (actually it became a rush).

Too many memory leaks
: The only memory properly reclaimed is that of the AST. No better answer for the rest of the compiler. This is the most severe flaw in this project, and definitely the worst thing to remember of: what we showed is not what student should learn to do.

  Though a garbage collector is tempting and well suited for our tasks, its pedagogical content is less interesting: students should be taught how to properly manage the memory.

Upgraded tarballs don't compile

Filling holes is not interesting
: Cannot be solved, see Section 1.4.3 [Tiger 2003], page 8.

Ending on TC-6 is frustrating
: Several students were frustrated by the fact we had to stop at TC-6: the reference compiler did not have any back-end. Continuing onto TC-7 was offered to several groups, and some of them actually finished the compiler. We took their work, adjusted it, and it became the base of the reference compiler of 2005. The most significant effort was made by Daniel Gazard.

Double submission is intractable
: Students were allowed to deliver twice their project — with a small penalty — if they failed to meet the so-called "first submission deadline", or if they wanted to improve their score. But it was impossible to organize, and led to too much sloppiness from some students. These problems were addressed with the introduction of "uploads" in Tiger 2005.

1.4.5 Tiger 2005

A lot of the following material is the result of discussion with several people, including, but not limited to[13]:

Comaintainers
: Benoît Perrot, Raphaël Poss,

Assistants
: Alexis Brouard, Sébastien Broussaud (Darks Bob), Stéphane Molina (Kain), William Fink,

Students
: Claire Calméjane, David Mancel, Fabrice Hesling, Michel Loiseleur.

I (Akim) here thank all the people who participated to this edition of this project. It has been a wonderful vintage, thanks to the students, the assistants, and the members of the LRDE.

Deliveries were:

Stage	**Submission**
TC-0	Friday, January 24th 2003 12:00
TC-1	Friday, February 14th 2003 12:00
TC-2	Friday, March 14th 2003 12:00
TC-4	Friday, April 25th 2003 12:00
TC-3	Rush from Saturday, May 24th at 18:00 to Sunday 12:00
TC-56	Friday, June 20th 2003, 12:00
TC-7	Friday, July 4th 2003 12:00

[13] Please, let us know whom we forgot!

TC-78 Friday, July 18th 2003 12:00

TC-9 Monday, September 8th 2003 12:00

Criticisms about Tiger 2005 include:

Too many memory leaks
: See Section 1.4.4 [Tiger 2004], page 9. This is the most significant failure of Tiger as an instructional project: we ought to demonstrate the proper memory management in big project, and instead we demonstrate laziness. Please, criticize us, denounce us, but do not reproduce the same errors.

  The factors that had pushed to a weak memory management is mainly a lack of coordination between developers: we should have written more things. So don't do as we did: define the memory management policy for each module, and write it.

  The 2006 edition pays strict attention to memory allocation.

Too long to compile
: Too much code was in `*.hh` files. Since then the policy wrt file contents was defined (see Section 2.4.3 [File Conventions], page 32), and in Tiger 2006 was adjusted to obey these conventions. Unfortunately, although the improvement was significant, it was not measured precisely.

  The interfaces between modules have also been cleaned to avoid excessive inter dependencies. Also, when possible, opaque types are used to avoid additional includes. Each module exports forward declarations in a `fwd.hh` file to promote this. For instance, `ast/tasks.hh` today includes:

```
// Forward declarations of ast:: items.
#include "ast/fwd.hh"
// ...
    /// Global root node of abstract syntax tree.
    extern ast::Exp* the_program;
// ...
```

  where it used to include all the AST headers to define exactly the type `ast::Exp`.

Upgraded tarballs don't compile

Filling holes is not interesting
: Cannot be solved, see Section 1.4.3 [Tiger 2003], page 8.

No written conventions
: Since its inception, the Tiger Compiler Project lacked this very section (see Section 1.4 [History], page 6) and that dedicated to coding style (see Section 2.4 [Coding Style], page 32) until the debriefing of 2005. As a result, some students or even so co-developers of our own `tc` reproduced errors of the past, changed something for lack of understanding, slightly broke the homogeneity of the coding style etc. Do not make the same mistake: write down your policy.

The AST is too poor
: One would like to insert annotations in the AST, say whether a variable is escaping (to know whether it cannot be in a register, see Section 4.5 [TC-3], page 98, and Section 4.14 [TC-5], page 132), or whether the left hand side of an assignment in `Void` (in which case the translation must not issue an actual assignment), or whether '`a < b`' is about strings (in which case the translation

will issue a hidden call to `strcmp`), or the type of a variable (needed when implementing object oriented Tiger), etc., etc.

As you can see, the list is virtually infinite. So we would need an extensible system of annotation of the AST. As of September 2003 no solution has been chosen. But we must be cautious not to complicate TC-2 too much (it is already a very steep step).

People don't learn enough C++
: It seems that the goal of learning object oriented programming and C++ is sometimes hidden behind the difficult understanding of the Tiger compiler itself. Sometimes students just fill the holes.

  To avoid this:

  - The holes will be bigger (conflicting with the ease to compile something, of course) to avoid any mechanical answering.
  - Each stage is now labeled with its "goals" (e.g., Section 4.4.1 [TC-2 Goals], page 89) that should help students to understand what is expected from them, and examiners to ask the appropriate questions.

The computation of the escapes is too hard
The computation of the escapes is too easy
: If you understood what it means that a variable escapes, then the implementation is so straightforward that it's almost boring. If you didn't understand it, you're dead. Because the understanding of escapes needs a good understanding of the stack management (explained more in details way afterward, during TC-5), many students are deadly lost.

  We are considering splitting TC-5 into two: TC-5- which would be limited to programs without escaping variables, and TC-5+ with escaping variables *and* the computation of the escapes.

The static-link optimization pass is improperly documented
: Todo.

The use of references is confusing
: We used to utilize references instead of pointers when the arity of the relation is one; in other words, we used pointers iff 0 was a valid value, and references otherwise. This is nice and clean, but unfortunately it caused great confusion amongst students (who were puzzled before '`*new`', and, worse yet, ended believing that's the only way to instantiate objects, even automatic!), and also confused some of the maintainers (for whom a reference does not propagate the responsibility wrt memory allocation/deallocation).

  Since Tiger 2006, the coding style enforces a more conventional style.

Not enough freedom
: The fact that the modelisation is already settled, together with the extensive skeletons, results in too tight a space for a programmer to experiment alternatives. We try to break these bounds for those who want by providing a generic interface: if you comply with it, you may interchange with your full re-implementation. We also (now explicitly) allow the use of a different tool set. Hints at possible extensions are provided, and finally, alternative implementation are suggested for each stage, for instance see Section 4.4.6 [TC-2 Improvements], page 98.

1.4.6 Tiger 2006

Akim has been helped by:

Assistants Claire Calméjane, Fabrice Hesling, Marco Tessari, Tristan Lanfrey

Deliveries:

Stage	Kind	Submission	Supervisor
TC-0		Wednesday, 2004-02-04 12:00	Anne-Lise Brourhant
TC-1		Sunday, 2004-02-08 12:00	Tristan Lanfrey
TC-2		Sunday, 2004-03-07 12:00	Anne-Lise Brourhant, Tristan Lanfrey
TC-3	Rush	Fr., 2004-03-19 18:30 to Sun., 2004-03-21 19:00	Fabrice Hesling
TC-4		Sunday, 2004-04-11 19:00	Tristan Lanfrey
TC-5		Sunday, 2004-06-06 12:00	Fabrice Hesling
TC-6		Sunday, 2004-06-27 12:00	Marco Tessari
TC-7	Opt	Sunday, 2004-07-11 12:00	Marco Tessari or Fabrice Hesling
TC-89	Opt	Thursday, 2004-07-29 12:00	Marco Tessari

Criticisms about Tiger 2006 include:

The interface of `symbol::Table` should be provided
: On the one hand side, we meant to have students implement it from scratch so we shouldn't provide the header, and on the other hand, the rest of the (provided) code expects a well defined interface, so we should publish it! The result was confusion and loss of time.

  The problem actually disappeared: Tiger 2007 no longer depends so heavily on scoped symbol tables.

Some examples are incorrectly rejected by the reference compiler
: The Tiger reference manual does not exclude sick examples such as:

```
let
  type rec = {}
in
  rec {}
end
```

  where the type `rec` escapes its scope since the type checker will assign the type `rec` to the `let` construct. Given the suggested implementation, which reclaims memory allocated by the declarations when closing the scope, the compiler dumps core.

  The new implementation, tested with 2005b, copes with this gracefully: types are destroyed when the AST is. This does not cure the example, which should be invalid IMHO. The following example, from Arnaud Fabre, amplifies the problem.

```
let
  var box :=
    let
      type box = {val: string}
      var box := box {val = "42\n"}
    in
       box
    end
in
```

```
  print(box.val)
end
```

TC-5 is too hard a stage
: This is a recurrent complaint. We tried to make it easier by moving more material into earlier stages (e.g., scopes are no longer dealt with by the `TranslateVisitor`: the `Binder` did it all).

Multiple inheritance is not demonstrated
: There are several nice opportunities of factoring the AST using multiple inheritance. Tiger 2007 uses them (e.g., `Escapable`, `Bindable` etc.).

The coding style for types is inconsistent
: The sources are ambivalent wrt to pointer and reference types. Sometimes '`type *var`', sometimes '`type* var`'. Obviously the latter is the more "logical": the space separates the type from the variable name. Unfortunately the declaration semantics in C/C++ introduces pitfalls: '`int* ip, i`' is equivalent to '`int* ip; int i;`'. That is why I, Akim, was using the '`type *var`' style, and resisted to expressing *the* coding style on this regard. The resulting mix of styles was becoming chronic: defining a rule was needed... In favor of '`type* var`', with the provision that multiple variable declarations are forbidden.

More enthusiasm from the assistants
: It has been suggested that assistants should show more motivation for the Tiger Project. It was suggested that they were not enough involved in the process. For Tiger 2007, there are no less than 10 Tiger assistants (as opposed to 4), and two of them are co-maintaining the reference compiler. Assistants will also be kept more informed of code changes than before.

More technical lectures
: Some regret when programming techniques (e.g., object functions, '`#include <functional>`') are not taught. My (Akim's) personal opinion is that students should learn to learn by themselves. It was decided to more emphasize these goals. Also, oral examinations should be *ahead* the code submission, and that should ensure that students have understood what is expected from them.

Formal definition of Booleans
: The Tiger language enjoys well defined semantics: a given program has a single defined behavior... except if the value of '`a & b`' or '`a | b`' is used. To fix this issue, in Tiger 2007 they return either 0 or 1.

Amongst other noteworthy changes, after five years of peaceful existence, the stages of the compiler were renamed from T1, T4 etc. to TC-1, TC-4... EPITA moved from "periods" (P1, P2...) to "trimesters" and they stole T1 and so forth from Tiger.

1.4.7 Tiger 2005b

Akim has been helped by:

Comaintainers
: Arnaud Fabre, Gilles Walbrou, Roland Levillain

Assistants
: Charles Rathouis, Claire Calméjane, Fabrice Hesling, Marco Tessari, Tristan Carel, Tristan Lanfrey,

Deliveries:

Stage	**Kind**	**Submission**
TC-1		Sun 2004-10-10 12:00

TC-2	Sun 2004-10-24 12:00
TC-3	Sun 2004-11-7 12:00
TC-4	Sun 2004-11-28 12:00

Criticisms about Tiger 2006 include:

Use of `misc::ident`
: Some examples would be most welcome. Well, there is `misc/test-indent.cc`, and now the `PrintVisitor` code includes a few examples.

`test-ref.cc`
: This file is used only in TC-5, yet it is submitted at TC-1, so students want to fix it, which is too soon. Tarballs will be adjusted to avoid this.

1.4.8 Tiger 2007

Akim has been helped by:

Comaintainers
: Arnaud Fabre, Roland Levillain, Gilles Walbrou

Assistants
: Arnaud Fabre, Bastien Gueguen, Benoît Monin, Chloé Boivin, Fanny Ricour, Gilles Walbrou, Julien Nesme, Philippe Kajmar, Tristan Carel

Deliveries:

Stage	**Kind**	**Launch**	**Submission**	**Supervisor**
TC-0		Wed 2005-03-09	Tue 2005-03-15 23:42	Bastien Gueguen
TC-1	Rush	Fri 2005-03-18	Sun 2005-03-19 9:00	Guillaume Bousquet
TC-2		Mon 2005-03-21	Sun 2005-04-03	Nicolas Rateau
TC-3	Rush	Fri 2005-04-08 20:00	Sun 2005-04-10 12:00	Fanny Ricour
TC-4		Mon 2005-04-18	Sun 2005-05-01	Julien Nesme
TC-5		Mon 2005-05-09	Sun 2005-06-05	Benoît Monin
TC-6		Mon 2005-06-06	Sun 2005-06-12	Philippe Kajmar
TC-7		Mon 2005-06-13	Sun 2005-06-19	Gilles Walbrou
TC-8		Mon 2005-06-20	Mon 2005-06-27	Arnaud Fabre
TC-9		Mon 2005-06-20	Sun 2005-07-03	Arnaud Fabre
Final submission			Wed 2005-07-06	

Criticisms about Tiger 2007 include:

Cheating
: Too much cheating during TC-5. Some would like more repression; that's fair enough. We will also be stricter during the exams.

Debriefing
: After a submission, there should be longer debriefings, including details about common errors. Some of the mysterious test cases should be explained (but not given in full). Maybe some bits of C++ code too.

Design of the compiler
: More justification of the overall design is demanded. Some selected parts, typically TC-5, should have a UML presentation.

Tarball
: Keep the tarball simple to use. We have to improve the case of tcsh. Also: give the tarball before the presentation by the assistants.

Oral examinations
: Assistants should be given a map of where to look at. The test suite should be evaluated at each submission. The use of version control too.

Optional parts
: They want more of them! We have more: see Section 4.6 [TC-R], page 106, Section 4.9 [TC-D], page 116, and Section 4.10 [TC-I], page 119.

`misc::` tools
: There should be a presentation of them.

TC-3 is too long
: TC-3, a rush, took several groups by surprise.

  Some groups would have liked to have the files earlier: in the future we will publish them on the Wednesday, instead of the last minute.

  Some groups have found it very difficult to be several working together on the same file (`binder.cc` of course). This is also a problem in the group management, and use of version control: when tasks are properly assigned, and using a tool such as Subversion, such problems should be minimal. In particular, merges resulting from updates should not be troublesome! Difficult updates result from disordered edition of the files. Dropping the use of a version control manager is not an answer: you will be bitten one day if two people edit concurrently the same file. One option is to split the file, say `binder-exp.cc` and `binder-dec.cc` for instance. I (Akim) leave this to students.

The template method template is too hard
: Some students would have preferred not to have the declaration of `Binder::decs_visit`, but the majority prefers: we will stay on this version, but we will emphasize that students are free not to follow our suggestions.

TC-5
: Several people would like more time to do it. But let's face it: the time most student spend on the project is independent of the amount of available time. Rather, early oral exams about TC-5 should suffice to prompt students to start earlier.

  People agree it is harder, and mainly because of compiler construction issues, not C++ issues. But many students prefer to keep it this way, rather than completely giving away the answers to compiler construction related problems.

1.4.9 Tiger 2008

We have been helped by:

Comaintainers
: Christophe Duong, Fabien Ouy

Assistants

Deliveries:

Stage	**Kind**	**Launch**	**Submission**	**Supervisor**
TC-0		Tue 01-03	Fri 01-13 23:42	Christophe Duong
TC-1	Rush	Fri 03-17	Sun 03-19 12:12	Renaud Lienhart
TC-2		Mon 03-20	Thu 03-30 23:42	David Doukhan
TC-3	Rush	Fri 03-31	Sun 04-02 12:12	Frederick Mousnier-Lompre
TC-4		Tue 04-04	Mon 04-24 23:42	Guillaume Deslandes
TC-5		Mon 05-01	Sun 05-28 23:42	Alexis Sebbane
TC-6		Mon 05-29	Sun 06-11 23:42	Christophe Duong
TC-7		Wed 06-14	Wed 06-21 12:00	
TC-8		Wed 06-21	Sun 07-2 12:00	

TC-9 Final	Mon 07-03	Sun 07-16 12:00

Some of the noteworthy changes compared to Section 1.4.8 [Tiger 2007], page 15:

Simplification of the parser
: The parser is simplified in a number of ways. First the old syntax for imported files, `let <decs> end`, is simplified into `<decs>`. We also use GLR starting at TC-2. `&`, `|` and the unary minus operator are desugared using concrete syntax transformations.

See Section 4.6 [TC-R], page 106, Unique identifiers
: This new optional part should be done during TC-3. Leave TC-E for later (with TC-5 or maybe TC-4).

Concrete syntax
: Transformations can now be written using Tiger concrete syntax rather than explicit AST construction in C++. This applies to the `DesugarVisitor`, `BoundsCheckingVisitor` and `InlineVisitor`.

1.4.10 Leopard 2009

We have been helped by:

Comaintainers
: Benoît Tailhades, Alain Vongsouvanh, Razik Yousfi, Benoît Perrot, Benoît Sigoure

Assistants

Deliveries:

Stage	**Kind**	**Launch**	**Submission**	**Supervisor**
LC-0		Mon 03-05	Fri 03-16 12:00	
LC-1	Rush	Fri 03-23	Sun 03-25 12:00	
LC-2		Mon 03-26	Fri 04-06 12:00	
LC-3 & LC-R	Rush	Fri 04-06	Sun 04-08 12:00	
LC-4		Mon 04-23	Sun 05-06 12:00	
LC-5		Mon 05-15	Sun 06-03 12:00	
LC-6		Mon 06-04	Sun 06-10 12:00	
LC-7		Mon 06-11	Wed 06-20 12:00	
LC-8		Thu 06-21	Sun 07-01 12:00	
LC-9		Mon 07-02	Sun 07-15 12:00	

Some of the noteworthy changes compared to Section 1.4.9 [Tiger 2008], page 16:

Object-Oriented Programming
: The language is extended with object-oriented features, as described by Andrew Appel in chapter 14 of Section 5.2 [Modern Compiler Implementation], page 233. The syntax is close to Appel's, with small modifications, see See Section "Syntactic Specifications" in *Tiger Compiler Reference Manual*.

Leopard
: To reflect this major addition, the language (and thus the project) is given a new name, *Leopard*. These changes was announced at TC-2, (renamed LC-2).

LC-R
: LC-R is a mandatory part of the LC-3 assignment.

1.4.11 Tiger 2010

We have been helped by:

Comaintainers
: Benoît Perrot, Benoît Sigoure, Guillaume Duhamel, Yann Grandmaître, Nicolas Teck

Assistants

Deliveries:

Stage	Kind	Launch	Submission	Supervisor
TC-0		Mon Nov 05, 2007	Sun Nov 25, 2007 12:00	
TC-1		Mon Dec 10, 2007	Sun Dec 16, 2007 12:00	
TC-2		Mon Feb 25, 2008	Wed Mar 05, 2008 12:00	
TC-3 & TC-R	Rush	Fri Mar 07, 2008	Sun Mar 09, 2008 12:00	
TC-4		Mon Mar 10, 2008	Sun Mar 23, 2008 12:00	
TC-5		Mon Mar 24, 2008	Sun Apr 06, 2008 12:00	
TC-6		Mon Apr 14, 2008	Sun Apr 20, 2008 12:00	
TC-7		Mon Apr 21, 2008	Sun May 04, 2008 12:00	
TC-8		Mon May 05, 2008	Sun May 18, 2008 12:00	
TC-9		Mon May 19, 2008	Sun Jun 01, 2008 12:00	

Some of the noteworthy changes compared to Section 1.4.10 [Leopard 2009], page 17:

The Tiger is back
: The project is renamed back to its original name.

1.4.12 Tiger 2011

This is the tenth year of the Tiger Project.

We have been helped by:

Assistants
: Adrien Biarnes, Medhi Ellaffet, Vincent Nguyen-Huu, Yann Grandmaître, Nicolas Teck

Deliveries:

Stage	Kind	Launch	Submission	Supervisor
.tig	Rush	Dec 20, 2008	Dec 21, 2008	
TC-0		Jan 05, 2009	Jan 16, 2009 at 12:00	
TC-1	Rush	Jan 16, 2009	Jan 18, 2009 at 12:00	
TC-2		Feb 16, 2009	Feb 25, 2009 at 23:42	
TC-3 & TC-R	Rush	Feb 27, 2009	Mar 01, 2009 at 11:42	
TC-4 & TC-E		Mar 02, 2009	Mar 15, 2009 at 11:42	
TC-5		Mar 16, 2009	Mar 25, 2009 at 23:42	
TC-6		Apr 23, 2009	May 03, 2009 at 12:00	
TC-7		May 04, 2009	May 17, 2009	
TC-8		May 18, 2009	May 31, 2009	
TC-9		Jun 29, 2009	Jul 12, 2009	

Some of the noteworthy changes compared to Section 1.4.11 [Tiger 2010], page 18:

The Bistromatig
: A new assignment is given for the `.tig` project: The Bistromatig. It consists in implementing an arbitrary-radix infinite-precision calculator. The project

is an adaptation of the famous Bistromathic project, that used to be one of the first C assignments at EPITA in the Old Days. The name was borrowed from Douglas Adams[14]'s invention[15] from *Life, the Universe and Everything*[16].

TC-E TC-E is a mandatory part of the TC-4 assignment.

1.4.13 Tiger 2012

This is the eleventh year of the Tiger Project.

We have been helped by:

Assistants Adrien Biarnes, Rémi Chaintron, Julien Delhommeau, Thomas Joly, Alexandre Laurent, Vincent Lechemin, Matthieu Martin

Deliveries:

Stage	**Kind**	**Launch**	**Submission**	**Supervisor**
.tig	Rush	Dec 02, 2009	Dec 04, 2009	
TC-0		Dec 11, 2009	Dec 20, 2009	
TC-1		Jan 11, 2010	Jan 17, 2010	
TC-2		Feb 01, 2010	Feb 17, 2010	
TC-3 & TC-R	Rush	Feb 19, 2010	Feb 26, 2010	
TC-4 & TC-E		Feb 22, 2010	Mar 07, 2010	
TC-5		Mar 11, 2010	Mar 22, 2010	
TC-6		Apr 19, 2010	May 02, 2010	
TC-7		May 12, 2010	May 25, 2010	
TC-8		May 25, 2010	Jun 06, 2010	
TC-9		Jun 07, 2010	Jun 12, 2010	

Some of the noteworthy changes compared to Section 1.4.12 [Tiger 2011], page 18:

Shorter mandatory assignment
By decision of the department of studies, the mandatory assignment ends after TC-3.

1.4.14 Tiger 2013

This is the twelfth year of the Tiger Project.

We have been helped by:

Assistants Rémi Chaintron, Julien Grall

Deliveries:

Stage	**Kind**	**Launch**	**Submission**	**Supervisor**
.tig	Rush			
TC-0				
TC-1				
TC-2				
TC-3 & TC-R	Rush			
TC-4 & TC-E				
TC-5				
TC-6				

[14] `http://en.wikipedia.org/wiki/Douglas_Adams`.

[15] `http://en.wikipedia.org/wiki/Bistromathic_drive#Bistromathic_drive`.

[16] `http://en.wikipedia.org/wiki/Life%2C_the_Universe_and_Everything`.

TC-7
TC-8
TC-9

Some of the noteworthy changes compared to Section 1.4.13 [Tiger 2012], page 19:

Build overhaul
: Silent rules, fewer Makefiles.

Bison Variant
: The parser is storing objects on its stacks, not only pointers. Other recent Bison features are also used.

1.4.15 Tiger 2014

This is the thirteenth year of the Tiger Project.

We have been helped by:

Assistants Jonathan Aigrain, Jules Bovet, Hugo Damme, Michael Denoun, Julien Grall, Christophe Pierre, Paul Similowski

CSI students
: Félix Abecassis

Deliveries for Ing1 students:

Stage	**Kind**	**Launch**	**Submission**	**Supervisor**
.tig	Lab	Nov 16, 2011	Nov 16, 2011	
TC-0		Dec 05, 2011	Dec 18, 2011 at 23:42	
TC-1	Rush	Jan 30, 2012 at 19:00	Feb 02, 2012 at 18:42	
TC-2		Feb 02, 2012 at 19:00	Feb 10, 2012 at 18:42	
TC-3 & TC-R	Rush	Feb 10, 2012 at 19:00	Feb 12, 2012 at 11:42	
TC-4 & TC-E		Feb 20, 2012 at 19:00	Mar 04, 2012 at 11:42	
TC-5		Mar 05, 2012 at 19:00	Mar 18, 2012 at 11:42	
TC-6		Apr 23, 2012 at 19:00	May 06, 2012 at 11:42	
TC-7		May 21, 2012 at 19:00	Jun 03, 2012 at 11:42	
TC-8		Jun 04, 2012 at 19:00	Jun 17, 2012 at 11:42	
TC-9		Jul 02, 2012 at 19:00	Jul 15, 2012 at 11:42	

Deliveries for AppIng1 students:

Stage	**Kind**	**Launch**	**Submission**	**Supervisor**
.tig	Lab	Nov 19, 2011	Nov 19, 2011	
TC-0		Dec 05, 2011	Dec 18, 2011 at 23:42	
TC-1		Jan 28, 2012 at 10:00	Feb 05, 2012 at 11:42	
TC-2		Feb 08, 2012 at 19:00	Feb 17, 2012 at 18:42	
TC-3 & TC-R	Rush	Feb 17, 2012 at 19:00	Feb 19, 2012 at 11:42	

Some of the noteworthy changes compared to Section 1.4.14 [Tiger 2013], page 19:

The Logomatig
: Due to time constraints, the Bistromatig assignment that has been previously used in the past three years for the `.tig` rush has been replaced by a 4-hour lab assignment: The Logomatig. This assignment is about implementing a small interpreter in Tiger for a subset of the Logo language[17]. The name of this project is a tribute to Logo, Tiger and the Bistromathic (though there are very few calculations in it).

[17] `http://en.wikipedia.org/wiki/Logo_%28programming_language%29`.

Introduction of C++ 2011 features
: Since a new C++ standard has been released this year (September 11, 2011), we are introducing some of its features in the Tiger project, namely range-based `for`-loops, `auto`-typed variables, use of the `nullptr` literal constant, use of explicitly defaulted and deleted functions, template metaprogramming traits provided by the standard library, and use of consecutive right angle brackets in templates. This set of features has been chosen for it is supported both by GCC 4.6 and Clang 3.0.

Git
: Git has replaced Subversion as version control system at EPITA. As of this year, we also provide the code with gaps through a public Git repository[18]. This method makes the integration of the code provided at the beginning of each stage easier (with the exception of TC-0, which is still to be done from scratch).

1.4.16 Tiger 2015

This is the fourteenth year of the Tiger Project.

We have been helped by:

Assistants
: Laurent Gourvénec, Xavier Grand, Frédéric Lefort, Théophile Ranquet, Robin Wils

Deliveries for Ing1 students:

Stage	Kind	Launch	Submission	Supervisor
.tig	Rush	Nov 23, 2012 at 18:42	Nov 25, 2012 at 11:42	
PTHL (TC-0)		Dec 10, 2012 at 18:42	Dec 23, 2012 at 11:42	
TC-1	Rush	Feb 11, 2013 at 18:42	Feb 13, 2013 at 23:42	
TC-2		Feb 14, 2013 at 18:42	Feb 24, 2013 at 11:42	
TC-3 & TC-R		Mar 4, 2013 at 18:42	Mar 10, 2013 at 11:42	
TC-4 & TC-E		Mar 11, 2013 at 18:42	Mar 24, 2013 at 11:42	
TC-5		Apr 22, 2013 at 18:42	May 5, 2013 at 11:42	
TC-6		May 20, 2013 at 19:00	Jun 2, 2013 at 11:42	
TC-7		Jun 2, 2013 at 19:00	Jun 16, 2013 at 11:42	
TC-8		Jun 28, 2013 at 19:00	Jul 11, 2013 at 11:42	
TC-9		Jul 12, 2013 at 19:00	Jul 21, 2013 at 11:42	

Deliveries for AppIng1 students:

Stage	Kind	Launch	Submission	Supervisor
.tig	Rush	Nov 23, 2012 at 18:42	Nov 25, 2012 at 11:42	
PTHL (TC-0)		Dec 10, 2012 at 18:42	Dec 23, 2012 at 11:42	
TC-1		Feb 11, 2013 at 18:42	Feb 17, 2013 at 11:42	
TC-2		Feb 18, 2013 at 18:42	Feb 28, 2013 at 11:42	
TC-3 & TC-R		Mar 11, 2013 at 18:42	Mar 20, 2013 at 23:42	

Some of the noteworthy changes compared to Section 1.4.15 [Tiger 2014], page 20:

TC-0 renamed as PTHL
: In an effort to emphasize the link between the THL (Formal Languages) lecture and the first stage of the Tiger project, the latter has been renamed as PTHL ("THL Project").

[18] `https://gitlab.lrde.epita.fr/tiger/tc-base.git`.

TC-3 is no longer a rush
: TC-3 has not been a successful step among many students for several years now. It has been deemed by many of them as too complex to be understood and implemented in a couple of days. Therefore we decided to extend the time allotted to this stage so as to give students more chance to pass TC-3.

Extension of the mandatory assignment to TC-5
: By decision of the department of studies, all Ing1 are required to work on the Tiger project up to TC-5. Subsequent steps remain optional.

Use of more C++ 2011 features
: This year, explicit template instantiation declarations (`extern template` clauses) are introduced in the project to control template instantiations in lieu of `*.hcc` files. The set of C++ features used in the Tiger compiler is still supported by both GCC 4.6 and Clang 3.0.

1.4.17 Tiger 2016

This is the fifteenth year of the Tiger Project.

We have been helped by:

Beta testers
: Anthony Seure, Rémi Weng

Assistants
: Aurélien Baud, Alexis Chotard, Baptiste Covolato, Arnaud Farbos, Laurent Gourvénec, Frédéric Lefort, Vincent Mirzaian-Dehkordi

Deliveries for Ing1 students:

Stage	**Kind**	**Launch**	**Submission**
.tig	Rush	Nov 22, 2013 at 21:00	Nov 24, 2013 at 11:42
PTHL (TC-0)		Dec 9, 2013 at 18:42	Dec 22, 2013 at 11:42
TC-1	Rush	Feb 17, 2014 at 14:00	Feb 19, 2014 at 23:42
TC-2		Feb 20, 2014 at 09:00	Mar 2, 2014 at 11:42
TC-3 & TC-R		Mar 3, 2014 at 19:00	Mar 16, 2014 at 11:42
TC-4 & TC-E		Mar 14, 2014 at 19:00	May 4, 2014 at 11:42
TC-5		May 5, 2014 at 19:00	May 24, 2014 at 23:42
TC-6		May 23, 2014 at 19:00	Jun 8, 2014 at 11:42
TC-7		Jun 9, 2014 at 19:00	Jun 22, 2014 at 11:42
TC-8		Jul 7, 2014 at 19:00	Jul 13, 2014 at 11:42
TC-9		Jul 15, 2014 at 10:00	Jul 20, 2014 at 11:42

Deliveries for AppIng1 students:

Stage	**Kind**	**Launch**	**Submission**
.tig	Rush	Nov 22, 2013 at 21:00	Nov 24, 2013 at 11:42
PTHL (TC-0)		Dec 9, 2013 at 18:42	Dec 22, 2013 at 11:42

Some of the noteworthy changes compared to Section 1.4.16 [Tiger 2015], page 21:

Use of even more C++ 2011 features
: The compiler introduces the following C++ 2011 features:
  - (standard) smart pointers (`std::unique_ptr`, `std::shared_ptr`);
  - general-purpose initializer lists;
  - lambda expressions;
  - explicit `override`s;

- template aliases;
- new function declarator syntax;
- delegating constructors;
- non-static data member initializers;
- inherited constructors.

The whole set of C++ features used in the Tiger compiler is supported by both GCC 4.8 and Clang 3.3.

C++ scanner
: We introduce a C++ scanner this year, still generated by Flex, but implemented as classes. The management of the scanner's inputs has been improved and responsibilities shared between the scanner and the driver (`parse::TigerParser`).

More Git Usage
: Starting this year, we deliver code with gaps exclusively through the tc-base public Git repository[19]. We no longer provide tarballs nor patches as a means to update students' code bases.

Changes in the language regarding object-oriented constructs
: The `nil` keyword has been made compatible with objects.

Style
: Many stylistics changes have been performed, mainly to match the EPITA Coding Style.

1.4.18 Tiger 2017

This is the sixteenth year of the Tiger Project.

We have been helped by:

Assistants
: Aurélien Baud, Baptiste Covolato, Pierre De Abreu, Léo Ercolanelli, Arnaud Farbos, Axel Manuel, Vincent Mirzaian-Dehkordi, Matthieu Simon, Jérémie Simon

Deliveries for Ing1 students:

Stage	**Kind**	**Launch**	**Submission**
.tig	Rush	Nov 21, 2014 at 21:00	Nov 23, 2014 at 11:42
PTHL (TC-0)		Dec 8, 2014 at 18:42	Dec 21, 2014 at 11:42
TC-1	Rush	Feb 4, 2015 at 22:00	Feb 8, 2015 at 11:42
TC-2		Feb 13, 2015 at 22:00	Feb 22, 2015 at 11:42
TC-3 & TC-R		Feb 23, 2015 at 22:00	Mar 1, 2015 at 11:42
TC-4 & TC-E		Mar 9, 2015 at 19:00	Mar 22, 2015 at 11:42
TC-5		Avr 20, 2015 at 19:00	May 3, 2015 at 11:42
TC-6		May 25, 2015 at 19:00	May 31, 2015 at 11:42
TC-7		Jun 1, 2015 at 19:00	Jun 7, 2015 at 11:42
TC-8		Jun 8, 2015 at 19:00	Jun 14, 2015 at 11:42
TC-9		Jul 6, 2015 at 10:00	Jul 19, 2015 at 11:42

Deliveries for AppIng1 students:

Stage	**Kind**	**Launch**	**Submission**
.tig	Rush	Nov 21, 2014 at 21:00	Nov 23, 2014 at 11:42

[19] `https://gitlab.lrde.epita.fr/tiger/tc-base.git`.

PTHL (TC-0)	Dec 8, 2014 at 18:42	Dec 21, 2014 at 11:42

Some of the noteworthy changes compared to Section 1.4.17 [Tiger 2016], page 22:

Use of even more C++ 2011 features
: The compiler introduces the following C++ 2011 features:
  - use `using` instead of `typedef`;
  - variadic templates (`misc::variant`).

  The C++ features used in the Tiger compiler are supported by both GCC 4.8 and Clang 3.3.

Style
: Many stylistics changes have been performed.

TC-Y
: An ARM back end has been added.

All the given code compiles
: Code given to students compiles even with the `// FIXME` chunks.

1.4.19 Tiger 2018

This is the seventeenth year of the Tiger Project.

We have been helped by:

Assistants
: Rémi Billon, Pierre-Louis Dagues, Pierre De Abreu, Léo Ercolanelli, Arnaud Gaillard, Axel Manuel, Sébastien Piat, Matthieu Simon, Jérémie Simon, Francis Visoiu Mistrih

Deliveries for Ing1 students:

Stage	**Kind**	**Launch**	**Submission**
.tig	Rush	Nov 20, 2015 at 20:00	Nov 22, 2015 at 11:42
PTHL (TC-0)		Dec 7, 2015 at 20:00	Dec 20, 2015 at 11:42
TC-1	Rush	Feb 15, 2016 at 20:00	Feb 19, 2016 at 11:42
TC-2		Feb 19, 2016 at 20:00	Feb 28, 2016 at 11:42
TC-3 & TC-R		Mar 7, 2016 at 20:00	Mar 20, 2016 at 11:42
TC-4 & TC-E		Apr 18, 2016 at 20:00	May 1, 2016 at 11:42
TC-5		May 2, 2016 at 20:00	May 15, 2016 at 11:42
TC-6		May 23, 2016 at 20:00	May 29, 2016 at 11:42
TC-7		May 30, 2016 at 20:00	Jun 5, 2016 at 11:42
TC-8		Jun 6, 2016 at 20:00	Jun 12, 2016 at 11:42
TC-9		Jun 27, 2016 at 20:00	Jul 10, 2016 at 11:42

Some of the noteworthy changes compared to Section 1.4.18 [Tiger 2017], page 23:

`type::Type` visitor
: Make the `type::Type` class visitable.

`#pragma once`
: Remove the `cpp` guards and replace them with `#pragma once` directives.

Use of C++14
: Move the standard from C++11 to C++14 since it is fully supported by both GCC 5.0 and Clang 3.4.

LLVM translator
: Add TC-L, a stage for LLVM IR generation. After TC-4, students have two choices:
  - Continue with Section 4.21 [TC-L], page 212, and stop.

– Continue with Section 4.14 [TC-5], page 132, and choose to do TC-backend.

Dementors
: Allow students to fix and push previous stages of TC more often after the final submission.

Overfun-object
: Add support for programs with overload and object.

  Usable through the new options:

  - `--overfun-object-bindings-compute`
  - `--overfun-object-types-compute`
  - `--overfun-object-object-desugar`

1.4.20 Tiger 2019

This is the eighteenth year of the Tiger Project.

We have been helped by:

Assistants
: Loïc Banet, Moray Baruh, Rémi Billon, Pierre-Louis Dagues, Arnaud Gaillard, Ashkan Kiaie-Sandjie, Guillaume Marques, Sarasvati Moutoucomarapoule, Cyprien Orfila, Sébastien Piat, Francis Visoiu Mistrih

Deliveries for Ing1 students:

Stage	**Kind**	**Launch**	**Submission**
.tig	Rush	Nov 4, 2016 at 19:00	Nov 6, 2016 at 11:42
PTHL (TC-0)		Dec 5, 2016 at 20:00	Dec 18, 2016 at 11:42
TC-1	Rush	Jan 30, 2017 at 20:00	Feb 3, 2017 at 11:42
TC-2		Feb 3, 2017 at 20:00	Feb 12, 2017 at 11:42
TC-3 & TC-R		Feb 13, 2017 at 20:00	Feb 26, 2017 at 11:42
TC-4 & TC-E		Mar 13, 2017 at 20:00	Mar 26, 2017 at 11:42
TC-5		Apr 17, 2017 at 20:00	Apr 30, 2017 at 11:42
TC-6		May 15, 2017 at 20:00	May 21, 2017 at 11:42
TC-7		May 29, 2017 at 20:00	Jun 4, 2017 at 11:42
TC-8		Jun 5, 2017 at 20:00	Jun 11, 2017 at 11:42
TC-9		Jun 26, 2017 at 20:00	Jul 9, 2017 at 11:42

Deliveries for AppIng1 students:

Stage	**Kind**	**Launch**	**Submission**
.tig	Rush	Nov 4, 2016 at 19:00	Nov 6, 2016 at 11:42
PTHL (TC-0)		Dec 5, 2016 at 20:00	Dec 18, 2016 at 11:42

Some of the noteworthy changes compared to Section 1.4.19 [Tiger 2018], page 24:

ast::tasks::the_program
: Make the_program a smart pointer, removing the `--ast-delete` option.

Use of even more C++14 features
: - use helper type for type_traits
  - use smart pointers' make function

Style
: Many stylistics changes have been performed.

Debug info
: Adding support debug information for the Tiger language using LLVM.

C++17 Notify future C++17's changes in comments.

Continuous integration
: Provide a CI for the students.

1.4.21 Tiger 2020

This is the nineteenth year of the Tiger Project.

We have been helped by:

Assistants Loïc Banet, Moray Baruh, Meven Courouble, Maxime Joubert, Ashkan Kiaie-Sandjie, Steven Lariau, Guillaume Marques, Sarasvati Moutoucomarapoule, Cyprien Orfila, Nicolas Poitoux, Loic Reyreaud, Andreas Touly

Deliveries for Ing1 students:

Stage	**Kind**	**Launch**	**Submission**
.tig	Rush	Jan 29, 2018 at 09:00	Jan 31, 2018 at 11:42
TC-0 (PTHL)		Jan 29, 2018 at 14:00	Feb 1, 2018 at 19:42
TC-1	Rush	Feb 1, 2018 at 20:00	Feb 4, 2018 at 11:42
TC-2		Feb 5, 2018 at 20:00	Feb 25, 2018 at 11:42
TC-3 & TC-R		Feb 12, 2018 at 20:00	Mar 11, 2018 at 11:42
TC-4 & TC-E		Mar 12, 2018 at 20:00	Mar 25, 2018 at 11:42
TC-5		Apr 16, 2018 at 20:00	Apr 29, 2018 at 11:42
TC-6		May 14, 2018 at 20:00	May 20, 2018 at 11:42
TC-7		May 21, 2018 at 20:00	May 27, 2018 at 11:42
TC-8		Jun 4, 2018 at 20:00	Jun 10, 2018 at 11:42
TC-9		Jun 25, 2018 at 20:00	Jul 8, 2018 at 11:42

Deliveries for AppIng1 students:

Stage	**Kind**	**Launch**	**Submission**
.tig	Rush	Jan 29, 2018 at 09:00	Jan 31, 2018 at 11:42
TC-0 (PTHL)		Jan 29, 2018 at 14:00	Feb 1, 2018 at 19:42

Some of the noteworthy changes compared to Section 1.4.20 [Tiger 2019], page 25:

C++17
- use structured bindings
- use std::variant instead of boost::variant
- use class template argument deduction
- use if(init; condition)

callee/caller-save
: Swap callee-save and caller-save order

Desugar Add desugar implementation for ArrayExp during TC-O

enum class
: Replace enums with enum classes

_main Ensure _main existence and correct prototype in the AST

Metavar Remove MetavarExp and Metavariable AST nodes

Namespaces
: Use nested namespaces

Smart pointers
: Replace some raw pointers with unique_ptr or shared_ptr

target::*::rewrite_program
: Add alternative rewrite_program implementation

Setup a debian-sid Dockerfile
: Provide a docker with requirements to build tc

2 Instructions

2.1 Interactions

Bear in mind that if you are writing, it is to be read, so pay attention to your reader.

The right place
: Using mails is almost always wrong: first ask around you, then try to find the assistants in their lab, and finally post into `assistants.tiger`. You need to have a very good reason to send a message to the assistants or to Akim and Etienne, as it usually annoys us, which is not in your interest.

  The newsgroup `assistants.tiger` is dedicated to the Compiler Construction lecture, the Tiger project, and related matters (e.g. assignments in Tiger itself). Any other material is off topic.

A meaningful title
: Find a meaningful subject.

Don't do that	**Do this**
Problem in TC-1	Cannot generate location.hh
make check	make check fails on test-ref

A legal content
: Pieces of critical code (e.g., precedence section in the parser, the string handling in the scanner, or whatever *you* are supposed to find by yourself) are not to be published.

  This includes the test cases. While posting a simple test case is tolerated, sending many of them, or simply one that addresses a specific common failure (e.g., some obscure cases for escapes) is strictly forbidden.

A complete content
: If you experience a problem that you fail to solve, make a report as complete as possible: include pieces of code (**unless the code is critical and shall not be published**) and the full error message from the compiler/tool. The following text by Simon Tatham is enlightening; its scope goes way beyond the Tiger Project: How to Report Bugs Effectively[1]. See also [How not to go about a programming assignment], page 242, item "Be clever when using electronic mail".

A legible content
: Use French or English. Epitean is definitely not a language.

A pertinent content
: Trolls are not welcome.

2.2 Rules of the Game

As any other assignment, the Tiger Project comes with its rules to follow.

`Thou Shalt Not Copy Code` [Rule]
`Thou Shalt Not Possess Thy Neighbor's Code` [Rule]
: It is *strictly* forbidden to possess code that is not yours. You are encouraged to work with others, but don't get a copy of their code. See [How not to go about a programming assignment], page 242, for more hints on what will *not* be accepted.

[1] `http://www.chiark.greenend.org.uk/~sgtatham/bugs.html`.

`Tests are part of the project` [Rule]
`Do not copy tests or test frame works` [Rule]
Test cases and test engines development are parts of the Tiger Project. As such the same rules apply as for code.

`If something is fishy, say it` [Rule]
If something illegal happened in the course of a stage, let us know, arrangements *might* be possible. If *we* find out, the rules will be strictly applied. It already happened that third year students have had to redo the Tiger Project because their code was found in another group: -42/20 is seldom benign.

`Don't hesitate working with other groups` [Rule]
Don't bother everybody instead of trying first. Conversely, once you did your best, don't hesitate working with others.

2.3 Groups

Starting with TC-1, assignments are to be done by groups of three.

The first cause of failures to the Tiger project is human problems within the groups. We cannot stress too much the importance of constituting a good group of four people. The Tiger project starts way before your first line of code: it begins with the selection of your partners.

Here are a few tips, collected wisdom from the previous failures.

You work for yourself, not for grades
Yes, we know, when you're a student grades are what matters. But close your eyes, make a step backwards, and look at yourself for a minute, from behind. You see a student, some sort of a larva, which will turn into a grownup. The larva stage lasts 3 to 4 years, while the hard working social insect is there for 40+ years: a 5% ratio without the internships. Three minutes out of an hour. These years are made to prepare you to the rest of your life, to provide you with what it takes to enjoy a lifelong success in jobs. So don't waste these three minutes by just cheating, paying little attention to what you are given, or by just waiting for this to end. The opportunity to learn is a unique moment in life: treasure it, even if it hurts, if it's hard, because you may well regret these three minutes for much of your life.

Start recruiting early
Making a team is not easy. Take the time to know the people, talk with them, and prepare your group way before beginning the project. The whole TC-0 is a test bed for you to find good partners.

Don't recruit good lazy friends
If s/he's lazy, you'll have to scold her/him. If s/he's a friend, that will be hard. Plus it will be even harder to report your problems to us.

Recruit people you can depend on
Trust should be your first criterion.

Members should have similar programming skills
Weak programmers should run away from skilled programmers
The worst "good idea" is "I'm a poor programmer, I should be in a group of skilled programmers: I will learn a lot from them". Experience shows this is wrong. What actually happens is as follows.

At the first stage, the leader assigns you a task. You try and fail, for weeks. In the meanwhile, the other members teach you lots of facts, but (i) you can't

memorize everything and end up saying "hum hum" without having understood, and (ii) because they don't understand what you don't understand, they are often poor teachers. The day before the submission, the leader does your assignments to save the group. You learned nothing, or quite. Second stage: same beginning, you are left with your assignment, but the other members are now bothered by your asking questions: why should they answer, since you don't understand what they say (remember: they are poor teachers because they don't understand your problems), and you don't seem to remember anything! The day before the submission, they do your work. From now on, they won't even ask you for anything: "fixing" you is much more time consuming than just doing it by themselves. Oral examinations reveal you neither understand nor do anything, hence your grades are bad, and you win another round of first year...

Take our advice: if you have difficulties with programming, be with other people like you. Your chances are better together, and anyway you are allowed to ask for assistance from other groups.

Don't mix repeaters with first year students

Repeaters have a much better understanding of the project than they think: they know its history, some parts of the code, etc. This will introduce a difference of skills from the beginning, which will remain till the end. It will result in the first year students having not participated enough to learn what was to be learned.

Don't pick up old code

This item is especially intended to repeaters: you might be tempted to keep the code from last year, believing this will spare you some work. It may not be so. Indeed, every year the specifications and the provided code change, sometimes with dramatic impact on the whole project. Struggling with an old code base to meet the new standard is a long, error prone, and uninteresting work. You might spend more time trying to preserve your old code than what is actually needed to implement the project from scratch. Not to mention that of course the latter has a much stronger educational impact.

Diagnose and cure drifts

When a dysfunction appears, fix it, don't let it grow. For instance, if a member never works in spite of the warnings, don't cover him: he will have the whole group drown. It usually starts with one member making more work on Tiger, less on the rest of the curriculum, and then he gets tired all the time, with bad mood etc. Don't walk that way: denounce the problems, send ultimatums to this person, and finally, warn the assistants you need to reconfigure your group.

Reconfigure groups when needed

Members can leave a group for many reasons: dropped EPITA, dropped Tiger, joined one of the schools' laboratories, etc. If your group is seriously unbalanced (two skilled people is OK, otherwise be three), ask for a reconfiguration in the news.

Tiger is a part of your curriculum

Tiger should neither be 0 nor 100% of your curriculum: find the balance. It is not easy to find it, but that's precisely one thing EPITA teaches: balancing overloads.

2.4 Coding Style

This section could have been named "Strong and Weak Requirements", as it includes not only mandatory features from your compiler (memory management), but also tips and advice. As the captain Barbossa would put it, "actually, it's more of a guideline than a rule."

2.4.1 No Draft Allowed

The code you deliver *must* be clean. In particular, when some code is provided, and you have to fill in the blanks denoted by '`FIXME: Some code has been deleted.`'. Sometimes you will have to write the code from scratch.

In any case, *dead code and dead comments must be removed*. You are free to leave comments spotting places where you fixed a '`FIXME:`', but never leave a fixed '`FIXME:`' in your code. Nor any irrelevant comment.

The official compiler for this project, is GNU C++ Compiler, 5.0 or higher (see Section 5.5 [GCC], page 249).

2.4.2 Use of Foreign Features

If, and only if, you already have enough fluency in C++ to be willing to try something wilder, then the following exception is made for you. Be warned: along the years the Tiger project was polished to best fit the typical epitean learning curve, trying to escape this curve is also taking a major risk. By the past, some students tried different approaches, and ended with unmaintainable pieces of code.

If you *and your group* are sure you can afford some additional difficulty (for additional benefits), then you may use the following extra tools. *You have to warn the examiners* that you use these tools. You also have to take care of harnessing `configure.ac` to make sure that what you need is available on the testing environment. Be also aware that you are likely to obtain less help from us if you use tools that we don't master: You are on your own, but, hey!, that's what you're looking for, ain't it?

The Loki Library
: See [Modern C++ Design], page 243, for more information about Loki.

The Boost Library
: As provided by the unstable Debian packages `libboost-*`. See [Boost.org], page 237.

Any Other Parser or Scanner Generator
: If you dislike Flex and/or Bison *but you already know how to use them*, then you are welcome to use other technologies.

If you think about something not listed here, please send us your proposal; acceptance is required to use them.

2.4.3 File Conventions

There are some strict conventions to obey wrt the files and their contents.

`One class LikeThis per files like-this.*` [Rule]
: Each class `LikeThis` is implemented in a single set of file named `like-this.*`. Note that the mixed case class names are mapped onto lower case words separated by dashes.

  There can be exceptions, for instance auxiliary classes used in a single place do not need a dedicated set of files.

`*.hh: Declarations` [Rule]

The `*.hh` should contain only declarations, i.e., prototypes, `extern` for variables etc. Inlined short methods are accepted when there are few of them, otherwise, create an `*.hxx` file. The documentation should be here too.

There is no good reason for huge objects to be defined here.

As much as possible, avoid including useless headers (GotW007[2], GotW034[3]):

- when detailed knowledge of a class is not needed, instead of

  ```
  #include <foo.hh>
  ```

  write

  ```
  // Fwd decl.
  class Foo;
  ```

  or better yet: use the appropriate `fwd.hh` file (read below).
- if you need output streams, then include `ostream`, not `iostream`. Actually, if you merely need to declare the existence of streams, you might want to include `iosfwd`.

`*.hxx: Inlined definitions` [Rule]

Some definitions should be loaded in different places: templates, inline functions etc. Declare and document them in the `*.hh` file, and implement them in the `*.hxx` file. The `*.hh` file *last* includes the `*.hxx` file, conversely `*.hxx` *first* includes `*.hh`. Read below.

`*.cc: Definitions of functions and variables` [Rule]

Big objects should be defined in the `*.cc` file corresponding to the declaration/documentation file `*.hh`.

There are less clear cut cases between `*.hxx` and `*.cc`. For instance short but time consuming functions should stay in the `*.cc` files, since inlining is not expected to speed up significantly. As another example features that require massive header inclusions are better defined in the `*.cc` file.

As a concrete example, consider the `accept` methods of the AST classes. They are short enough to be eligible for an `*.hxx` file:

```
void
LetExp::accept(Visitor& v)
{
  v(*this);
}
```

We will leave them in the `*.cc` file though, since this way only the `*.cc` file needs to load `ast/visitor.hh`; the `*.hh` is kept short, both directly (its contents) and indirectly (its includes).

`Explicit template instantiation` [Rule]

There are several strategies to compile templates. The most common strategy consists in leaving the code in a `*.hxx` file, and letting every user of the class template instantiate the code. While correct, this approach has several drawbacks:

- Because the `*.hh` file includes the `*.hxx` file, each time a simple declaration of a template is needed, the full implementation comes with it. And if the implementation requires other declarations such as `std::iostream`, you force all the client code to parse the `iostream` header!

[2] `http://www.gotw.ca/gotw/007.htm`.

[3] `http://www.gotw.ca/gotw/034.htm`.

– The instantiation is performed several times, which is time and space consuming.
– The dependencies are tight: the clients of the template depend upon its implementation.

To circumvent these problems, we may control template instantiations using *explicit template instantiation definitions* (available since C++ 1998) and *declarations* (introduced by C++ 2011).

This mechanism is compatible with the way templates are usually handled in the Tiger compiler, i.e., where both template declarations *and* definitions are accessible from the included header, though often indirectly (see above). We use the following two-fold strategy:

– First, we add an explicit template *definition* in the implementation file of the template's client (for instance `temp/temp.cc`) to instruct the compiler that we want to instantiate a template (e.g. `misc::endo_map<T>`) for a given (set of) parameter(s) (e.g. `temp::Temp`) in this compilation unit (`temp/temp.o`). This explicit template definition is performed using a `template` clause.

```
/**
 ** \file temp/temp.cc
 ** \brief temp::Temp.
 */

#include <temp/temp.hh>

// ...

namespace misc
{
  // Explicit template instantiation definition to generate the code.
  template class endo_map<temp::Temp>;
}
```

File 2.1: `temp.cc`

– Then, we *block* the automatic (implicit) instantiation of the template for this (set of) parameter(s), which would otherwise be triggered by default by the compiler when the implementation of the template is made available to it—which is the case in our example, since the header of the template (`misc/endomap.hh`) also includes its implementation (`misc/endomap.hxx`). To do so, we add an explicit template instantiation *declaration* matching the previous explicit template definition, using an `extern template` clause.

```
/**
 ** \file temp/temp.hh
 ** \brief Fresh temps.
 */

#pragma once

#include <misc/endomap.hh>

namespace temp
{
  struct Temp { /* ... */ };
```

```
}

// ...

namespace misc
{
  // Explicit template instantiation declaration.
  extern template class endo_map<temp::Temp>;
}
```

File 2.2: `temp.hh`

Any translation unit containing this explicit declaration will not generate this very template instantiation, unless an explicit definition is seen (in our case, this will happen within `temp/temp.cc` only).

You will notice that both the approach and the syntax used here recall the ones used to declare and define global variables in C and C++.

We can further improve the previous design by factoring explicit instantiation code using the preprocessor.

```
/**
 ** \file temp/temp.hh
 ** \brief Fresh temps.
 */

#pragma once

#include <misc/endomap.hh>

#ifndef MAYBE_EXTERN
# define MAYBE_EXTERN extern
#endif

namespace temp
{
  struct Temp { /* ... */ };
}

// ...

namespace misc
{
  // Explicit template instantiation declaration.
  MAYBE_EXTERN template class endo_map<temp::Temp>;
}
```

File 2.3: `temp-factored.hh`

```
/**
 ** \file temp/temp.cc
 ** \brief temp::Temp.
```

```
 */

#define MAYBE_EXTERN
#include <temp/temp.hh>
#undef MAYBE_EXTERN

// ...
```

File 2.4: `temp-factored.cc`

Explicit template instantiation declarations (not definitions) are only available since C++ 2011. Before that, we used to introduce a fourth type of file, `*.hcc`: files that had to be compiled once for each concrete template parameter.

`Guard included files (*.hh & *.hxx)` [Rule]

Use the '`#pragma once`' directive to ensure the contents of a file is read only once. This is critical for `*.hh` and `*.hxx` files that include one another.

One typically has:

```
/**
 ** \file sample/sample.hh
 ** \brief Declaration of sample::Sample.
 **/

#pragma once

// ...

#include <sample/sample.hxx>
```

File 2.5: `sample/sample.hh`

```
/**
 ** \file sample/sample.hxx
 ** \brief Inlined definition of sample::Sample.
 **/

#pragma once

#include <sample/sample.hh>

// ...
```

File 2.6: `sample/sample.hxx`

`fwd.hh: forward declarations` [Rule]

Dependencies can be a major problem during big project developments. It is not acceptable to "recompile the world" when a single file changes. To fight this problem, you are encouraged to use `fwd.hh` files that contain simple forward declarations. Everything that defeat the interest of `fwd.hh` file must be avoided, e.g., including actual header files. These forward files should be included by the `*.hh` instead of more complete headers.

The expected benefit is manifold:

- A forward declaration is much shorter.
- Usually actual definitions rely on other classes, so other '`#include`'s etc. Forward declarations need nothing.
- While it is not uncommon to change the interface of a class, changing its name is infrequent.

Consider for example `ast/visitor.hh`, which is included directly or indirectly by many other files. Since it needs a declaration of each AST node one could be tempted to use `ast/all.hh` which includes virtually all the headers of the `ast` module. Hence all the files including `ast/visitor.hh` will bring in the whole `ast` module, where the much shorter and much simpler `ast/fwd.hh` would suffice.

Of course, usually the `*.cc` files need actual definitions.

`Module, namespace, and directory likethis` [Rule]
The compiler is composed of several modules that are dedicated to a set of coherent specific tasks (e.g., parsing, AST handling, register allocation etc.). A module name is composed of lower case letters exclusively, `likethis`, not `like_this` nor `like-this`. This module's files are stored in the directory with the same name, which is also that of the namespace in which all the symbols are defined.

Contrary to file names, we do not use dashes to avoid clashes with Swig and `namespace`.

`lib`*`module`*`.*: Pure interface` [Rule]
The interface of the *module* module contains only *pure* functions: these functions should not depend upon globals, nor have side effects of global objects. Global variables are forbidden here.

`tasks.*: Impure interface` [Rule]
Tasks are *the* place for side effects. That's where globals such as the current AST, the current assembly program, etc., are defined and modified.

2.4.4 Name Conventions

`Stay out of reserved names` [Rule]
The standard reserves a number of identifier classes, most notably '`_*`' [17.4.3.1.2]:

> Each name that begins with an underscore is reserved to the implementation for use as a name in the global namespace.

Using '`_*`' is commonly used for CPP guards ('`_FOO_HH_`'), private members ('`_foo`'), and internal functions ('`_foo ()`'): don't.

`Name your classes LikeThis` [Rule]
Class should be named in mixed case; for instance `Exp`, `StringExp`, `TempMap`, `InterferenceGraph` etc. This applies to class templates. See [CStupidClassName], page 239.

`Name public members like_this` [Rule]
No upper case letters, and words are separated by an underscore.

`Name private/protected members like_this_` [Rule]
It is extremely convenient to have a special convention for private and protected members: you make it clear to the reader, you avoid gratuitous warnings about conflicts in constructors, you leave the "beautiful" name available for public members etc. We used to write `_like_this`, but this goes against the standard, see [Stay out of reserved names], page 37.

For instance, write:

```
class IntPair
{
public:
  IntPair(int first, int second)
    : first_(first)
    , second_(second)
  {
  }
protected:
  int first_, second_;
}
```

See [CStupidClassName], page 239.

`Name your using type alias foo_type` [Rule]

When declaring a `using` type alias, name the type *`foo`*`_type` (where *foo* is obviously the part that changes). For instance:

```
using map_type = std::map<const symbol, Entry_T>;
using symtab_type = std::list<map_type>;
```

We used to use *`foo`*`_t`, unfortunately this (pseudo) name space is reserved by POSIX.

`Name the parent class super_type` [Rule]

It is often handy to define the type of "the" super class (when there is a single one); use the name `super_type` in that case. For instance most Visitors of the AST start with:

```
class TypeChecker: public ast::DefaultVisitor
{
public:
  using super_type = ast::DefaultVisitor;
  using super_type::operator();
  // ...
```

(Such `using` clauses are subject to the current visibility modifier, hence the `public` beforehand.)

`Hide auxiliary classes` [Rule]

Hide auxiliary/helper classes (i.e., classes private to a single compilation unit, not declared in a header) in functions, or in an anonymous namespace. Instead of:

```
struct Helper { ... };

void
doit()
{
  Helper h;
  ...
}
```

write:

```
namespace { struct Helper { ... }; }

void
doit()
{
  Helper h;
```

```
  ...
}
```

or

```
void
doit()
{
  struct Helper { ... } h;
  ...
}
```

The risk otherwise is to declare two classes with the same name: the linker will ignore one of the two silently. The resulting bugs are often difficult to understand.

2.4.5 Use of C++ Features

`Hunt Leaks` [Rule]

Use every possible means to release the resources you consume, especially memory. Valgrind can be a nice assistant to track memory leaks (see Section 5.8 [Valgrind], page 250). To demonstrate different memory management styles, you are invited to use different features in the course of your development: proper use of destructors for the AST, use of a factory for `symbol`, `Temp` etc., use of `std::unique_ptr` starting with the `Translate` module, and finally use of reference counting via smart pointers for the intermediate representation.

`Hunt code duplication` [Rule]

Code duplication is your enemy: the code is less exercised (if there are two routines instead of one, then the code is run half of the time only), and whenever an update is required, you are likely to forget to update all the other places. Strive to prevent code duplication from sneaking into your code. Every C++ feature is good to prevent code duplication: inheritance, templates etc.

`Prefer dynamic_cast of references` [Rule]

Of the following two snippets, the first is preferred:

```
const IntExp& ie = dynamic_cast<const IntExp&>(exp);
int val = ie.value_get();
```

```
const IntExp* iep = dynamic_cast<const IntExp*>(&exp);
assert(iep);
int val = iep->value_get();
```

While upon type mismatch the second `abort`s, the first throws a `std::bad_cast`: they are equally safe.

`Use virtual methods, not type cases` [Rule]

Do not use type cases: if you want to dispatch by hand to different routines depending upon the actual class of objects, you probably have missed some use of virtual functions. For instance, instead of

```
bool
compatible_with(const Type& lhs, const Type& rhs)
{
  if (&lhs == &rhs)
    return true;
  if (dynamic_cast<Record*>(&lhs))
    if (dynamic_cast<Nil*>(&rhs))
       return true;
```

```
  if (dynamic_cast<Record*>(&rhs))
    if (dynamic_cast<Nil*>(&lhs))
       return true;
  return false;
}
```

write

```
bool
Record::compatible_with(const Type& rhs)
{
  return &rhs == this || dynamic_cast<const Nil*>(&rhs);
}

bool
Nil::compatible_with(const Type& rhs)
{
  return dynamic_cast<const Record*>(&rhs);
}
```

`Use dynamic_cast for type cases` [Rule]

Did you read the previous item, "Use virtual methods, not type cases"? If not, do it now.

If you really *need* to write type dispatching, carefully chose between `typeid` and `dynamic_cast`. In the case of `tc`, where we sometimes need to down cast an object or to check its membership to a specific subclass, we don't need `typeid`, so use `dynamic_cast` only.

They address different needs:

`dynamic_cast` for (sub-)membership, `typeid` for exact type

The semantics of testing a `dynamic_cast` vs. a comparison of a `typeid` are not the same. For instance, think of a class `A` with subclass `B` with subclass `C`; then compare the meaning of the following two snippets:

```
// Is 'a' containing an object of exactly the type B?
bool test1 = typeid(a) == typeid(B);
// Is 'a' containing an object of type B, or a subclass of B?
bool test2 = dynamic_cast<B*>(&a);
```

Non polymorphic entities

`typeid` works on hierarchies without `vtable`, or even builtin types (`int` etc.). `dynamic_cast` requires a dynamic hierarchy. Beware of `typeid` on static hierarchies; for instance consider the following code, courtesy from Alexandre Duret-Lutz:

```
#include <iostream>

struct A
{
  // virtual ~A() {};
};

struct B: A
{
};
```

```
int
main()
{
  A* a = new B;
  std::cout << typeid(*a).name() << std::endl;
}
```

it will "answer" that the `typeid` of '`*a`' is `A`(!). Using `dynamic_cast` here will simply not compile[4]. If you provide `A` with a virtual function table (e.g., uncomment the destructor), then the `typeid` of '`*a`' is `B`.

Compromising the future for the sake of speed

Because the job performed by `dynamic_cast` is more complex, it is also significantly slower that `typeid`, but hey! better slow and safe than fast and furious.

You might consider that today, a strict equality test of the object's class is enough and faster, but can you guarantee there will never be new subclasses in the future? If there will be, code based `dynamic_cast` will probably behave as expected, while code based `typeid` will probably not.

More material can be found the chapter 8 of see [Thinking in C++ Volume 2], page 246: Run-time type identification[5].

`Use const references in arguments to save copies (`EC22`)` [Rule]

We use const references in arguments (and return value) where otherwise a passing by value would have been adequate, but expensive because of the copy. As a typical example, accessors ought to return members by const reference:

```
const Exp&
OpExp::lhs_get() const
{
  return lhs_;
}
```

Small entities can be passed/returned by value.

`Use references for aliasing` [Rule]

When you need to have several names for a single entity (this is the definition of *aliasing*), use references to create aliases. Note that passing an argument to a function for side effects is a form of aliasing. For instance:

```
template <typename T>
void
swap(T& a, T& b)
{
  T c = a;
  a = b;
  b = c;
}
```

`Use pointers when passing an object together with its management` [Rule]

When an object is created, or when an object is *given* (i.e., when its owner leaves the management of the object's memory to another entity), use pointers. This is consistent

[4] For instance, `g++` reports an '`error: cannot dynamic_cast 'a' (of type 'struct A*') to type 'struct B*' (source type is not polymorphic)`'.

[5] `http://www.smart2help.com/e-books/ticpp-2nd-ed-vol-two/#_Toc53985808`.

with C++: `new` creates an object, returns it together with the responsibility to call `delete`: it uses pointers. For instance, note the three pointers below, one for the return value, and two for the arguments:

```
OpExp*
opexp_builder(OpExp::Oper oper, Exp* lhs, Exp* rhs)
{
  return new OpExp(oper, lhs, rhs);
}
```

`Avoid static class members` (EC47) [Rule]

More generally, "Ensure that non-local static objects are initialized before they're used", as reads the title of EC47.

Non local static objects (such as `std::cout` etc.) are initialized by the C++ system even before `main` is called. Unfortunately there is no guarantee on the order of their initialization, so if you happen to have a static object which initialization depends on that of another object, expect the worst. Fortunately this limitation is easy to circumvent: just use a simple Singleton implementation, that relies on a *local* static variable.

This is covered extensively in EC47.

`Use foo_get, not get_foo` [Rule]

Accessors have standardized names: `foo_get` and `foo_set`.

There is an alternative attractive standard, which we don't follow:

```
class Class
{
public:
  int foo();
  void foo(int foo);
private:
  int foo_;
}
```

or even

```
class Class
{
public:
  int foo();
  Class& foo(int foo);  // Return *this.
private:
  int foo_;
}
```

which enables idioms such as:

```
{
  Class obj;
  obj.foo(12)
     .bar(34)
     .baz(56)
     .qux(78)
     .quux(90);
}
```

`Use dump as a member function returning a stream` [Rule]

You should always have a means to print a class instance, at least to ease debugging. Use the regular `operator<<` for standalone printing functions, but `dump` as a member function. Use this kind of prototype:

```
std::ostream& Tree::dump(std::ostream& ostr [, ...]) const
```

where the ellipsis denote optional additional arguments. `dump` returns the stream.

2.4.6 Use of STL

`Specify comparison types for associative containers of pointers (ES20)` [Rule]

For instance, instead of declaring

```
using temp_set_type = std::set<const Temp*>;
```

declare

```
/// Object function to compare two Temp*.
struct temp_compare
{
  bool
  operator()(const Temp* s1, const Temp* s2) const
  {
    return *s1 < *s2;
  }
};

using temp_set_type = std::set<const Temp* , temp_compare>;

temp_set_type my_set;
```

Or, using C++11 lambdas:

```
/// Lambda to compare two Temp*.
auto temp_compare = [](const Temp* s1, const Temp* s2)
                    {
                      return *s1 < *s2;
                    };

using temp_set_type = std::set<const Temp* , decltype(temp_compare)>;

temp_set_type my_set{temp_compare};
```

Scott Meyers mentions several good reasons, but leaves implicit a very important one: if you don't, since the outputs will be based on the order of the pointers in memory, and since (i) this order may change if your allocation pattern changes and (ii) this order depends of the environment you run, then *you cannot compare outputs (including traces)*. Needless to say that, at least during development, this is a serious misfeature.

`Prefer standard algorithms to hand-written loops (ES43)` [Rule]

Using `for_each`, `find`, `find_if`, `transform` etc. is preferred over explicit loops. This is for (i) efficiency, (ii) correctness, and (iii) maintainability. Knowing these algorithms is mandatory for who claims to be a C++ programmer.

`Prefer member functions to algorithms with the same names` [Rule]
(ES44)

For instance, prefer '`my_set.find(my_item)`' to '`find (my_item, my_set.begin(), my_set.end())`'. This is for efficiency: the former has a logarithmic complexity, versus... linear for the latter! You may find the Item 44 of Effective STL[6] on the Internet.

2.4.7 Matters of Style

The following items are more a matter of style than the others. Nevertheless, you are asked to follow this style.

`80 columns maximum` [Rule]

Stick to 80 column programming. As a matter of fact, stick to 76 or 78 columns most of the time, as it makes it easier to keep the diffs within the limits. And if you post/mail these diffs, people are likely to reply to the message, hence the suggestion of 76 columns, as for emails.

`Order class members by visibility first` [Rule]

When declaring a class, start with public members, then protected, and last private members. Inside these groups, you are invited to group by category, i.e., methods, types, and members that are related should be grouped together. The motivation is that private members should not even be visible in the class declaration (but of course, it is mandatory that they be there for the compiler), and therefore they should be "hidden" from the reader.

This is an example of what should **not** be done:

```
class Foo
{
public:
  Foo(std::string, int);
  virtual ~Foo();

private:
  using string_type = std::string;
public:
  std::string bar_get() const;
  void bar_set(std::string);
private:
  string_type bar_;

public:
  int baz_get() const;
  void baz_set(int);
private:
  int baz_;
}
```

rather, write:

```
class Foo
{
public:
  Foo(std::string, int);
  virtual ~Foo();
```

[6] http://www.informit.com/articles/article.aspx?p=21851.

```
  std::string bar_get() const;
  void bar_set(std::string);

  int baz_get() const;
  void baz_set(int);

private:
  using string_type; = std::string
  string_type bar_;
  int baz_;
}
```

and add useful Doxygen comments.

`Keep superclasses on the class declaration line` [Rule]

When declaring a derived class, try to keep its list of superclasses on the same line. Leave a space at least on the right hand side of the colon. If there is not enough room to do so, leave the colon on the class declaration line (the opposite applies for constructor, see [Put initializations below the constructor declaration], page 47).

```
class Derived: public Base
{
  // ...
};

/// Object function to compare two Temp*.
struct temp_ptr_less
{
  bool operator()(const Temp* s1, const Temp* s2) const;
};
```

`Don't use inline in declarations` [Rule]

Use `inline` in implementations (i.e., `*.hxx`, possibly `*.cc`)), not during declarations (`*.hh` files).

`Use override` [Rule]

If a method was once declared `virtual`, it remains virtual, there is no need to repeat it. However, be sure to explicitly mark it as `override` so that your compiler can verify it.

```
class Base
{
public:
  // ...
  virtual void foo() = 0;
};

class Derived: public Base
{
public:
  // ...
  void foo() override;
};
```

`Pointers and references are part of the type` [Rule]

Pointers and references are part of the type, and should be put near the type, not near the variable.

```
int* p;        // not ‘int *p;’
list& l;       // not ‘list &l;’
void* magic(); // not ‘void *magic();’
```

`Do not declare many variables on one line` [Rule]

Use

```
int* p;
int* q;
```

instead of

```
int *p, *q;
```

The former declarations also allow you to describe each variable.

`Leave no space between template name and effective parameters` [Rule]

Write

```
std::list<int> l;
std::pair<std::list<int>, int> p;
```

with a space after the comma. There is no need for a space between two closing ‘`>`’ (since C++ 2011):

```
std::list<std::list<int>> ls;
```

These rules apply for casts:

```
// Come on baby, light my fire.
int* p = static_cast<int*>(42);
```

`Leave one space between` TEMPLATE `and formal parameters` [Rule]

Write

```
template <class T1, class T2>
struct pair;
```

with one space separating the keyword `template` from the list of formal parameters.

`Leave no space between a function name and its argument(s), either formal or actual` [Rule]

```
int
foo(int n)
{
  return bar(n);
}
```

The ‘`()`’ operator is not a list of arguments.

```
class Foo
{
public:
  Foo();
  virtual ~Foo();
  bool operator()(int n);
};
```

`Put initializations below the constructor declaration` [Rule]

Don't put or initializations or constructor invocations on the same line as you declare the constructor. As a matter of fact, don't even leave the colon on that line. Instead of '`A::A(): B(), C()`', write either:

```
A::A()
  : B()
  , C()
{
}
```

or

```
A::A()
  : B(), C()
{
}
```

The rationale is that the initialization belongs more to the body of the constructor than its signature. And when dealing with exceptions leaving the colon above would yield a result even worse than the following.

```
A::A()
try
  : B()
  , C()
{
}
catch (...)
{
}
```

2.4.8 Documentation Style

`Write correct English` [Rule]

Nowadays most editors provide interactive spell checking, including for sources (strings and comments). For instance, see `flyspell-mode` in Emacs, and in particular the `flyspell-prog-mode`. To trigger this automatically, install the following in your `~/.emacs.el`:

```
(add-hook 'c-mode-hook         'flyspell-prog-mode 1)
(add-hook 'c++-mode-hook       'flyspell-prog-mode 1)
(add-hook 'cperl-mode-hook     'flyspell-prog-mode 1)
(add-hook 'makefile-mode-hook 'flyspell-prog-mode 1)
(add-hook 'python-mode-hook    'flyspell-prog-mode 1)
(add-hook 'sh-mode-hook        'flyspell-prog-mode 1)
```

and so forth.

End comments with a period.

`Be concise` [Rule]

For documentation as for any other kind of writing, the shorter, the better: hunt useless words. See [The Elements of Style], page 246, for an excellent set of writing guidelines.

Here are a few samples of things to avoid:

Don't document the definition instead of its object

Don't write:

```
/// Declaration of the Foo class.
```

```
class Foo
{
  ...
};
```

Of course you're documenting the definition of the entities! "Declaration of the" is totally useless, just use '`/// Foo class`'. But read bellow.

Don't qualify obvious entity kinds

Don't write:

```
/// Foo class.
class Foo
{
public:
  /// Construct a Foo object.
  Foo(Bar& bar)
  ...
};
```

It is so obvious that you're documenting the class and the constructor that you should not write it down. Instead of documenting the *kind* of an entity (class, function, namespace, destructor...), document its *goal*.

```
/// Wrapper around Bar objects.
class Foo
{
public:
  /// Bind to \a bar.
  Foo(Bar& bar)
  ...
};
```

`Use the Imperative` [Rule]

Use the imperative when documenting, as if you were giving order to the function or entity you are describing. When describing a function, there is no need to repeat "function" in the documentation; the same applies obviously to any syntactic category. For instance, instead of:

```
/// \brief Swap the reference with another.
/// The method swaps the two references and returns the first.
ref& swap(ref& other);
```

write:

```
/// \brief Swap the reference with another.
/// Swap the two references and return the first.
ref& swap(ref& other);
```

The same rules apply to ChangeLogs.

`Use rebox.el to mark up paragraphs` [Rule]

Often one wants to leave a clear markup to separate different matters. For declarations, this is typically done using the Doxygen '`\name ... \{ ... \}`' sequence; for implementation files use `rebox.el` (see [rebox.el], page 56).

`Write Documentation in Doxygen` [Rule]

Documentation is a genuine part of programming, just as testing. We use Doxygen (see Section 5.16 [Doxygen], page 255) to maintain the developer documentation of the Tiger Compiler. The quality of this documentation can change the grade.

Beware that Doxygen puts the first letter of documentation in upper case. As a result,

```
/// \file  ast/arrayexp.hh
/// \brief ast::ArrayExp declaration.
```

will not work properly, since Doxygen will transform `ast::ArrayExp` into '`Ast::ArrayExp`', which will not be recognized as an entity name. As a workaround, write the slightly longer:

```
/// \file  ast/arrayexp.hh
/// \brief Declaration of ast::ArrayExp.
```

Of course, Doxygen documentation is not appropriate everywhere.

`Document namespaces in lib*.hh files` [Rule]
`Document classes in their *.hh file` [Rule]
There must be a single location, that's our standard.

`Use '\directive'` [Rule]
Prefer backslash ('\') to the commercial at ('@') to specify directives.

`Prefer C Comments for Long Comments` [Rule]
Prefer C comments ('`/** ... */`') to C++ comments ('`/// ...`'). This is to ensure consistency with the style we use.

`Prefer C++ Comments for One Line Comments` [Rule]
Because it is lighter, instead of

```
/** \brief Name of this program. */
extern const char* program_name;
```

prefer

```
/// Name of this program.
extern const char* program_name;
```

For instance, instead of

```
/* Construct an InterferenceGraph. */
InterferenceGraph(const std::string& name,
                  const assem::instrs_t& instrs, bool trace = false);
```

or

```
/** @brief Construct an InterferenceGraph.
 ** @param name    its name, hopefully based on the function name
 ** @param instrs  the code snippet to study
 ** @param trace   trace flag
 **/
InterferenceGraph(const std::string& name,
                  const assem::instrs_t& instrs, bool trace = false);
```

or

```
/// \brief Construct an InterferenceGraph.
/// \param name    its name, hopefully based on the function name
/// \param instrs  the code snippet to study
/// \param trace   trace flag
InterferenceGraph(const std::string& name,
                  const assem::instrs_t& instrs, bool trace = false);
```

write

```
/** \brief Construct an InterferenceGraph.
    \param name    its name, hopefully based on the function name
```

```
    \param instrs  the code snippet to study
    \param trace   trace flag
  */
InterferenceGraph(const std::string& name,
                  const assem::instrs_t& instrs, bool trace = false);
```

2.5 Tests

As stated in Section 2.2 [Rules of the Game], page 29, writing a test framework and tests is part of the exercise.

As a starting point, we provide a tarball containing a few Tiger files, see Section 3.3 [Given Test Cases], page 70. **They are not enough**: your test suite should be continually expanding.

2.5.1 Writing Tests

In three occasions tests are "easy" to write:

- The specifications of the language are a fine source for many tests. For instance the specification of integer literals show several cases to exercise.
- If your compiler crashes or fails, before even trying to fix it, include the test case in your test suite.
- If you are developing a component for the compiler, you can certainly feel the weak points. Immediately write a test for these.

See [Testing student-made compilers], page 245, for many hints on what tests you need to write.

2.5.2 Generating the Test Driver

Unless your whole test infrastructure is embedded in a single file (which is not a good idea), we advise you to generate any script used to run your tests so that they can be run from a directory other than the source directory where they reside. This is especially useful to maintain several builds (e.g. with different compilers or compiler flags) in parallel (see the section on VPATH Builds[7] in Automake's manual) and when running '`make distcheck`' (see the section on Checking the Distribution[8]), as source and build directories are distinct in these circumstances.

The simplest way to generate a script is to rely on `configure`. For instance, the following line in `configure.ac` generates a script `tests/testsuite` from the input `tests/testsuite.in`, while performing variables substitutions (in particular '`@srcdir@`' and similar variables):

```
AC_CONFIG_FILES([tests/testsuite], [chmod a=rx tests/testsuite])
```

The template file `tests/testsuite.in` can then leverage this information to find data in the source directory. E.g., if tests are located in the `tests/` subdirectory of the top source directory, the beginning of `tests/testsuite.in` might look like this:

```
#! /bin/sh
# @configure_input@

# Where the tests can be found.
testdir="@abs_top_srcdir@/tests"
```

[7] `http://www.gnu.org/software/automake/manual/html_node/VPATH-Builds.html`.

[8] `http://www.gnu.org/software/automake/manual/html_node/Checking-the-Distribution.html`.

```
# ...
```

Another strategy to generate scripts is to use `make`, as suggested by Autoconf's manual (see the section on Installation Directory Variables[9]).

2.6 Submission

We use two kinds of project submissions in the project.

- For PTHL (see Section 4.2 [PTHL (TC-0)], page 72), your *sources* must be pushed through the '`master`' branch to the central Git repository at submission time. Follow the instructions given by the teaching assistants.
- From TC-1 on, your code must still be submitted through git, following indications given on the assistants's intranet. However, the assistants "moulinette" will use the *tarball* built by the command '`make distcheck`'. Be sure that the created tarball has a correct name. If `bardec_f` is the head of your group, the tarball must be `bardec_f-tc-n.tar.bz2` where *n* is the number of the "release" (see Section 5.4.1 [Package Name and Version], page 247). The following commands must work properly:

```
$ bunzip2 -cd bardec_f-tc-n.tar.bz2 | tar xvf -
$ cd bardec_f-tc-n
$ export CC=gcc+-5.0 CXX=g++-5.0
$ mkdir _build
$ cd _build
$ ../configure
$ make
$ src/tc /tmp/test.tig
$ make distcheck
```

  For more information on the tools, see Section 5.4 [The GNU Build System], page 247, Section 5.5 [GCC], page 249.

2.7 Evaluation

Some stages are evaluated only by a program, and others are evaluated both by humans, and a program.

2.7.1 Automated Evaluation

Each stage of the compiler will be evaluated by an automatic corrector. Soon after your work is submitted, the logs are available on the assistants' intranet.

Automated evaluation enforces the requirements: you *must* stick to what is being asked. For instance, for TC-E it is explicitly asked to display something like:

```
var /* escaping */ i : int := 2
```

so if you display any of the following outputs

```
var i : int /* escaping */ := 2
var i /* escaping */ : int := 2
var /* Escapes */ i : int := 2
```

be sure to fail all the tests, even if the computation is correct.

[9] `http://www.gnu.org/software/autoconf/manual/html_node/Installation-Directory-Variables.html`.

2.7.2 During the Examination

When you are defending your projects, here are a few rules to follow:

Don't talk Don't talk unless you are asked to: when a person is asked a question, s/he is the only one to answer. You must not talk to each other either: often, when one cannot answer a question, the question is asked to another member. It is then obvious why the members of the group shall not talk.

Don't touch the screen
Don't touch my display! You have nice fingers, but I don't need their prints on my screen.

Tell the truth
If there is something the examiner must know (someone did not work on the project at all, some files are coming from another group etc.), *say it immediately*, for, if we discover that by ourselves, you will be severely sanctioned.

Learn It is explicitly stated that you *can* not have worked on a stage *provided this was an agreement with the group*. But it is also explicitly stated that *you must have learned what was to be learned from that compiler stage*, which includes C++ techniques, Bison and Flex mastering, object oriented concepts, design patterns and so forth.

Complain now!
If you don't agree with the notation, say it immediately. Private messages about "this is unfair: I worked much more than bardec_f but his grade is better than mine" are *thrown away*.

Conversely, there is something we wish to make clear: examiners will probably be harsh (maybe even very harsh), but this does not mean they disrespect you, or judge you badly.

You are here to defend your project and knowledge, they are here to stress them, to make sure they are right. Learning to be strong under pressure is part of the exercise. Don't burst into tears, react! Don't be shy, that's not the proper time: you are selling them something, and they will never buy something from someone who cries when they are criticizing his product.

You should also understand that human examination is the moment where we try to evaluate who, or what group, needs help. We are here to diagnose your project and provide solutions to your problems. If you know there is a problem in your project, but you failed to fix it, tell it to the examiner! *Work with her/him* to fix your project.

2.7.3 Human Evaluation

The point of this evaluation is to measure, among other things:

the quality of the code
How clean it is, amount of code duplication, bad hacks, standards violations (e.g., '`stderr`' is forbidden in proper C++ code) and so forth. It also aims at detecting cheaters, who will be severely punished (mark = -42).

the knowledge each member acquired
While we do not require that each member worked on a stage, we do require that each member (i) knows how the stage works and (ii) has perfectly understood the (C++, Bison etc.) techniques needed to implement the stage. Each stage comes with a set of goals (see Section 4.2.1 [PTHL Goals], page 72, for instance) on which you will be interrogated.

Examiners: the human grade.

The examiner should not take (too much) the automated tests into account to decide the mark: the mark is computed later, taking this into account, so don't do it twice.

Examiners: broken tarballs.

If you fixed the tarball or made whatever modification, *run* '`make distcheck`' again, and update the delivered tarball. Do not keep old tarballs, do not install them in a special place: just replace the first tarball with it, but say so in the '`eval`' file.

The rationale is simple: only tarballs pass the tests, and every tarball must be able to pass the tests. If you don't do that, then someone else will have to do it again.

2.7.4 Marks Computation

Because the Tiger Compiler is a project with stages, the computation of the marks depends on the stages too. To spell it out explicitly:

A stage is penalized by bad results on tests performed for previous stages.

It means, for instance, that a TC-3 compiler will be exercised on TC-1, TC-2, and TC-3. If there are still errors on TC-1 and TC-2 tests, they will pessimize the result of TC-3 tests. The older the errors are, the more expensive they are.

3 Source Code

3.1 Given Code

Starting with TC-1, code with gaps is provided through the tc-base public Git repository[1]. We used to provide code through tarballs and patches before, but we only rely on Git now. This approach is the best one, as `git merge` is arguably simpler than `patch` and has other advantages (like preserving the execution bit of scripts, identifying the origin of every line of code using `git blame`, etc.). Each commit containing the contents of a new stage is labeled with a '*class*`-tc-base-`*x.y*' tag.

Here is the recommended strategy to use this repository.

1. At TC-1, subscribe to the repository, fetch its contents and integrate the given code using `git merge` with the commit labeled '`2020-tc-base-1.0`' into your '`master`' branch:

   ```
   $ git remote add tc-base https://gitlab.lrde.epita.fr/tiger/tc-base.git
   $ git fetch tc-base
   $ git merge 2020-tc-base-1.0
   ```

   Fix the conflicts and record the merge commit:

   ```
   $ git add src/tc.cc ...
   $ git commit
   ```

2. For any subsequent stage `m`, all you will need to do is fetch the new commits from the '`tc-base`' repository and merge the code given at stage `m` into yours (and of course, fix the conflicts). E.g.:

   ```
   $ git fetch tc-base
   $ git merge 2020-tc-base-m.0
   ```

3.2 Project Layout

This section describes the *mandatory* layout of the package.

3.2.1 The Top Level

`AUTHORS.txt`
: In the top level of the distribution, there must be a file `AUTHORS.txt` which contents is as follows:

  ```
  Fabrice Bardèche     <bardec_f@epita.fr>
  Jean-Paul Sartre     <sartre_j@epita.fr>
  Jean-Paul Deux       <deux_j@epita.fr>
  Jean-Paul Belmondo   <belmon_j@epita.fr>
  ```

  The group leader is first. Do not include emails other than those of EPITA. We repeat: give the '`login@epita.fr`' address. Starting from TC-1, the file `AUTHORS.txt` is distributed thanks to the `EXTRA_DIST` variable in the top-level `Makefile.am`, but pay attention to the spelling.

`ChangeLog`
: Optional. The list of the changes made in the compiler, with the dates and names of the people who worked on it. See the Emacs key binding '`C-x 4 a`'.

`README.txt`
: Various free information.

[1] `https://gitlab.lrde.epita.fr/tiger/tc-base.git`.

`NEWS.txt` Optional. Summary of changes introduced by each release.

`lib/` This directory contains helping tools, that are not specific to the project.

`src/` All the sources are in this directory.

`tests/` Your own test suite. You should make it part of the project, and ship it like the rest of the package. Actually, it is abnormal not to have a test suite here.

3.2.2 The `build-aux` Directory

`bison++.in` (*build-aux/bin*) [File]
This is a wrapper around Bison, tailored to produce C++ parsers. Compared to `bison`, `bison++` updates the output files only if changed. For a file such as `location.hh`, virtually included by the whole front-end, this is a big win.

Also, `bison` outputs '`\file location.hh`' in Doxygen documentation, which clashes with `ast/location.hh`. `bison++` changes this into '`\file parse/location.hh`'.

`flex++.in` (*build-aux/bin*) [File]
A wrapper around Flex, to simplify and improve the generation of C++ scanners.

`monoburg++.in` (*build-aux/bin*) [File]
Likewise for MonoBURG.

`rebox.el` (*build-aux/*) [File]
This file provides two new Emacs functions, '`M-x rebox-comment`' and '`M-x rebox-region`'. They build and maintain nice looking boxed comments in most languages. Once installed (read it for instructions), write a simple comment such as:

```
// Comments end with a period.
```

then move your cursor into this comment and press '`C-u 2 2 3 M-q`' to get:

```
/*-----------------------------.
| Comments end with a period.  |
`-----------------------------*/
```

'`2 2 3`' specifies the style of the comment you want to build. Once the comment built, '`M-q`' suffices to refill it. Run '`C-u - M-q`' for an interactive interface.

`tiger.el` (*build-aux*) [File]
`panther.el` (*build-aux*) [File]
Theses files provide Emacs major modes for Tiger programs (`*.tig`) and Panther ("object-less" Tiger) programs (`*.pan` files). Read them to get installation instructions.

`tiger-ftdetect.vim` (*build-aux*) [File]
`tiger-syntax.vim` (*build-aux*) [File]
Vim scripts to detect and enable syntax hilighting for Tiger files.

3.2.3 The `lib` Directory

3.2.4 The `lib/misc` Directory

Convenient C++ tools.

`contract.*` (*lib/misc/*) [File]
A useful improvement over `cassert`.

`error.*` (*lib/misc/*) [File]

The class `misc::error` implements an error register. Because libraries are expected to be pure, they cannot issue error messages to the error output, nor exit with failure. One could pass call-backs (as functions or as objects) to set up error handling. Instead, we chose to register the errors in an object, and have the library functions return this register: it is up to the caller to decide what to do with these errors. Note also that direct calls to `std::exit` bypass stack unwinding. In other words, with `std::exit` (instead of `throw`) your application leaks memory.

An instance of `misc::error` can be used as if it were a stream to output error messages. It also keeps the current exit status until it is "triggered", i.e., until it is thrown. Each module has its own error handler. For instance, the `Binder` has an `error_` attribute, and uses it to report errors:

```
void
Binder::error(const ast::Ast& loc, const std::string& msg)
{
  error_ << misc::error::bind
         << loc.location_get() << ": " << msg << std::endl;
}
```

Then the task system fetches the local error handler, and merges it into the global error handler `error` (see `common.*`). Some tasks trigger the error handler: if errors were registered, an exception is raised to exit the program cleanly. The following code demonstrates both aspects.

```
void
bindings_compute()
{
  // bind::bind returns the local error handler.
  error << ::bind::bind(*ast::tasks::the_program);
  error.exit_on_error();
}
```

`escape.*` (*lib/misc/*) [File]

This file implements a means to output string while escaping non printable characters. An example:

```
std::cout << "escape(\"\111\") = " << escape("\"\111\"") << std::endl;
```

Understanding how `escape` works is required starting from TC-2.

`flex-lexer.hh` (*lib/misc/*) [File]

The skeleton of the C++ scanner. Adapted from Flex's `FlexLexer.h` and used as a replacement, thanks to `flex++` (see [flex++.in], page 56).

`graph.*` (*lib/misc/*) [File]

This file contains a generic implementation of oriented and undirected graphs.

Understanding how `graph` works is required starting from TC-8.

`indent.*` (*lib/misc/*) [File]

Exploiting regular `std::ostream` to produce indented output.

`ref.*` (*lib/misc/*) [File]

Smart pointers implementing reference counting.

`set.*` (*lib/misc/*) [File]

A wrapper around `std::set` that introduce convenient operators (`operator+` and so forth).

`scoped-map.*` (*lib/misc/*) [File]
The handling of `misc::scoped_map<Key, Data>`, generic scoped map, serving as a basis for symbol tables used by the `Binder`. `misc::scoped_map` maps a `Key` to a `Data` (that should ring a bell...). You are encouraged to implement something simple, based on stacks (see `std::stack`, or better yet, `std::vector`) and maps (see `std::map`).

It must provide this interface:

`put` (*const Key& key, const Data& value*) [void]
Associate *value* to *key* in the current scope.

`get` (*const Key& key*) *const* [Data]
If *key* was associated to some `Data` in the open scopes, return the most recent insertion. Otherwise, if `Data` is a pointer type, then return the empty pointer, else throw a `std::range_error`. To implement this feature, see `<type_traits>`[2]

`dump` (*std::ostream& ostr*) *const* [std::ostream&]
Send the content of this table on *ostr in a human-readable manner*, and return the stream.

`scope_begin` () [void]
Open a new scope.

`scope_end` () [void]
Close the last scope, forgetting everything since the latest `scope_begin()`.

`symbol.*` (*lib/misc/*) [File]
In a program, the rule for identifiers is to be used many times: at least once for its definition, and once for each use. Just think about the number of occurrences of `size_t` in a C program for instance.

To save space one keeps a single copy of each identifier. This provides additional benefits: the address of this single copy can be used as a key: comparisons (equality or order) are much faster.

The class `misc::symbol` is an implementation of this idea. See the lecture notes, `scanner.pdf`[3]. `misc::symbol` is based on `misc::unique`.

`timer.*` (*lib/misc/*) [File]
A class that makes it possible to have timings of processes, similarly to `gcc`'s `--time-report`, or `bison`'s `--report=time`. It is used in the `Task` machinery, but can be used to provide better timings (e.g., separating the scanner from the parser).

`unique.*` (*lib/misc/*) [File]
A generic class implementing the Flyweight design pattern. It maps identical objects to a unique reference.

`variant.*` (*lib/misc/*) [File]
A wrapper over `std::variant` supporting conversion operators.

[2] `http://en.cppreference.com/w/cpp/header/type_traits`.
[3] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/scanner.pdf`.

3.2.5 The `src` Directory

`common.hh` (*src/*) [File]
Used throughout the project.

`tc` (*src/*) [File]
Your compiler.

`tc.cc` (*src/*) [File]
Main entry. Called, the *driver*.

3.2.6 The `src/task` Directory

No namespace for the time being, but it should be `task`. Delivered for TC-1. A generic scheme to handle the components of our compiler, and their dependencies.

3.2.7 The `src/parse` Directory

Namespace '`parse`'. Delivered during TC-1.

`scantiger.ll` (*src/parse/*) [File]
The scanner.

`parsetiger.yy` (*src/parse/*) [File]
The parser.

`position.hh` (*src/ast/*) [File]
Keeping track of a point (cursor) in a file.

`location.hh` (*src/ast/*) [File]
Keeping track of a range (two cursors) in a (or two) file.

`libparse.hh` (*src/ast/*) [File]
which prototypes what `tc.cc` needs to know about the module '`parse`'.

3.2.8 The `src/ast` Directory

Namespace '`ast`', delivered for TC-2. Implementation of the abstract syntax tree. The file `ast/README` gives an overview of the involved class hierarchy.

`location.hh` (*src/ast/*) [File]
Imports Bison's `parse::location`.

`visitor.hh` (*src/ast/*) [File]
Abstract base class of the compiler's visitor hierarchy. Actually, it defines a class template `GenVisitor`, which expects an argument which can be either `misc::constify_traits` or `misc::id_traits`. This allows to define two parallel hierarchies: `ConstVisitor` and `Visitor`, similar to `iterator` and `const_iterator`.

The understanding of the template programming used *is not required at this stage* as it is quite delicate, and goes far beyond your (average) current understanding of templates.

`default-visitor.*` (*src/ast/*) [File]
Implementation of the `GenDefaultVisitor` class template, which walks the abstract syntax tree, doing nothing. This visitor does not define visit methods for nodes related to object-oriented constructs (classes, methods, etc.); thus it is an abstract class, and is solely used as a basis for deriving other visitors. It is instantiated twice: `GenDefaultVisitor<misc::constify_traits>` and `GenDefaultVisitor<misc::id_traits>`.

`non-object-visitor.*` (*src/ast/*) [File]
Implementation of the `GenNonObjectVisitor` class template, which walks the abstract syntax tree, doing nothing, but aborting on nodes related to object-oriented constructs (classes, methods, etc.). This visitor is abstract and is solely used as a basis for deriving other visitors (see Section 4.4.5 [TC-2 FAQ], page 96). It is instantiated twice: `GenNonObjectVisitor<misc::constify_traits>` and `GenNonObjectVisitor<misc::id_traits>`.

`object-visitor.*` (*src/ast/*) [File]
Implementation of the `GenObjectVisitor` class template, which walks object-related nodes of an abstract syntax tree, doing nothing. This visitor is abstract and is solely used as a basis for deriving other visitors. It is instantiated twice: `GenObjectVisitor<misc::constify_traits>` and `GenObjectVisitor<misc::id_traits>`.

`pretty-printer.*` (*src/ast/*) [File]
The `PrettyPrinter` class, which pretty-prints an AST back into Tiger concrete syntax.

`typable.*` (*src/ast/*) [File]
This class is not needed before TC-4 (see Section 4.8 [TC-4], page 109).

Auxiliary class from which typable AST node classes should derive. It has a simple interface made to manage a pointer to the type of the node:

`type_set` (*const type::Type**) [void]
`type::Type*` *type_get* () *const* [const]
Accessors to the type of this node.

`accept` (*ConstVisitor*`&` *v*) *const* [void]
`accept` (*Visitor*`&` *v*) [void]
These methods are abstract, as in `ast::Ast`.

`type-constructor.*` (*src/ast/*) [File]
This class is not needed before TC-4 (see Section 4.8 [TC-4], page 109).

Auxiliary class from which should derive AST nodes that construct a type (e.g., `ast::ArrayTy`). Its interface is similar to that of `ast::Typable` with one big difference: `ast::TypeConstructor` is responsible for de-allocating that type.

`created_type_set` (*const type::Type**) [void]
`type::Type*` *created_type_get* () *const* [const]
Accessors to the *created* type of this node.

`accept` (*ConstVisitor*`&` *v*) *const* [void]
`accept` (*Visitor*`&` *v*) [void]
It is convenient to be able to visit these, but it is not needed.

`escapable.*` (*src/ast/*) [File]
This class is needed only for TC-E (see Section 4.7 [TC-E], page 107).

Auxiliary class from which AST node classes that denote the declaration of variables and formal arguments should derive. Its role is to encode a single Boolean value: whether the variable escapes or not. The natural interface includes `escape_get` and `escape_set` methods.

3.2.9 The `src/bind` Directory

Namespace '`bind`'. Binding uses to definitions.

`binder.*` (*src/bind/*) [File]
The `bind::Binder` visitor. Binds uses to definitions (works on syntax *without* object).

`renamer.*` (*src/bind/*) [File]
The `bind::Renamer` visitor. Renames every identifier to a unique name (works on syntax *without* object).

3.2.10 The `src/escapes` Directory

Namespace '`escapes`'. Compute the escaping variables.

`escapes-visitor.*` (*src/escapes/*) [File]
The `escapes::EscapesVisitor`.

3.2.11 The `src/type` Directory

Namespace '`type`'. Type checking.

`libtype.*` (*src/type/*) [File]
The interface of the Type module. It exports a single procedure, `types_check`.

`types.hh` (*src/type/*) [File]
`type.*` (*src/type/*) [File]
`array.*` (*src/type/*) [File]
`attribute.*` (*src/type/*) [File]
`builtin-types.*` (*src/type/*) [File]
`class.*` (*src/type/*) [File]
`field.*` (*src/type/*) [File]
`function.*` (*src/type/*) [File]
`method.*` (*src/type/*) [File]
`named.*` (*src/type/*) [File]
`record.*` (*src/type/*) [File]
The definitions of all the types. Built-in types (`Int`, `String` and `Void`) are defined in `src/type/builtin-types.*`.

`nil.*` (*src/type/*) [File]
The `Nil` type is holding information about the real record type that it's hiding. The `record_type` represents the actual type that the `nil` was meant to be used with.

The `record_type` is set during the type-checker in the parent nodes of the node holding a `Nil` type.

`type-checker.*` (*src/type/*) [File]
The `type::TypeChecker` visitor. Computes the types of an AST and adds type labels to the corresponding nodes (works on syntax *without* object).

`pretty-printer.*` (*src/type/*) [File]
The `type::PrettyPrinter` visitor which pretty-prints `type::Types` in a human-readable way. Used to output nice type errors.

3.2.12 The `src/object` Directory

`binder.*` (*src/object/*) [File]
The `object::Binder` visitor. Binds uses to definitions (works on syntax with objects). Inherits from `bind::Binder`.

`type-checker.*` (*src/object/*) [File]
The `object::TypeChecker` visitor. Computes the types of an AST and adds type labels to the corresponding nodes (works on syntax with objects). Inherits from `type::TypeChecker`.

`renamer.*` (*src/object/*) [File]
The `object::Renamer` visitor. Renames every identifier to a unique name (works on syntax with objects), and keeps a record of the names of the renamed classes. Inherits from `bind::Renamer`.

`desugar-visitor.*` (*src/object/*) [File]
The `object::DesugarVisitor` visitor. Transforms an AST with objects into an AST without objects.

3.2.13 The `src/overload` Directory

Namespace '`overload`'. Overloading function support.

3.2.14 The `src/astclone` Directory

`cloner.*` (*src/astclone*) [File]
The `astclone::Cloner` visitor. Duplicate an AST. This copy is purely structural: the clone is similar to the original tree, but any existing binding or type information is *not* preserved.

3.2.15 The `src/desugar` Directory

`desugar-visitor.*` (*src/desugar*) [File]
The `desugar::DesugarVisitor` visitor. Remove constructs that can be considered as syntactic sugar using other language constructs. For instance, turn `for` loops into `while` loops, string comparisons into function calls. Inherits from `astclone::Cloner`, so the desugared AST is a modified copy of the initial tree.

`bounds-checking-visitor.*` (*src/desugar*) [File]
The `desugar::BoundsCheckingVisitor` visitor. Add dynamic array bounds checks while duplicating an `ast`. Inherits from `astclone::Cloner`, so the result is a modified copy of the input AST.

3.2.16 The `src/inlining` Directory

`inliner.*` (*src/inlining*) [File]
The `desugar::Inliner` visitor. Perform inline expansion of functions.

`pruner.*` (*src/inlining*) [File]
The `desugar::Pruner` visitor. Prune useless function declarations within an `ast`.

3.2.17 The `src/temp` Directory

Namespace `temp`, delivered for TC-5.

`identifier.*` (*src/temp/*) [File]
Provides the class template `Identifier` built upon `misc::variant` and used to implement `temp::Temp` and `temp::Label`. Also contains the generic `IdentifierCompareVisitor`, used to compare two identifiers.

`Identifier` handles maps of `Identifiers`. For instance, the `Temp t5` might be allocated the register `$t2`, in which case, when outputting `t5`, we should print `$t2`. Maps stored in the xalloc'd slot `Identifier::map` of streams implements such a correspondence. In addition, the `operator<<` of the `Identifier` class template itself "knows" when such a mapping is active, and uses it.

`label.*` (*src/temp/*) [File]
We need labels for `jumps`, for functions, strings etc. Implemented as an instantiation of the `temp::Identifier` scheme.

`temp.*` (*src/temp/*) [File]
So called *temporaries* are pseudo-registers: we may allocate as many temporaries as we want. Eventually the register allocator will map those temporaries to either an actual register, or it will allocate a slot in the activation block (aka frame) of the current function. Implemented as an instantiation of the `temp::Identifier` scheme.

`temp-set.*` (*src/temp/*) [File]
A set of temporaries, along with its `operator<<`.

3.2.18 The `src/tree` Directory

Namespace `tree`, delivered for TC-5. The implementation of the intermediate representation. The file `tree/README` should give enough explanations to understand how it works.

Reading the corresponding explanations in Appel's book is mandatory.

It is worth noting that contrary to A. Appel, just as we did for `ast`, we use n-ary structures. For instance, where Appel uses a binary `seq`, we have an n-ary `seq` which allows us to put as many statements as we want.

To avoid gratuitous name clashes, what Appel denotes `exp` is denoted `sxp` (Statement Expression), implemented in `translate::Sxp`.

Please, pay extra attention to the fact that there are `temp::Temp` used to create unique temporaries (similar to `misc::symbol`), and `tree::Temp` which is the intermediate representation instruction denoting a temporary (hence a `tree::Temp` needs a `temp::Temp`). Similarly, on the one hand, there is `temp::Label` which is used to create unique labels, and on the other hand there are `tree::Label` which is the IR statement to *define* to a label, and `tree::Name` used to *refer* to a label (typically, a `tree::Jump` needs a `tree::Name` which in turn needs a `temp::Label`).

`fragment.*` (*src/tree/*) [File]
It implements `tree::Fragment`, an abstract class, `tree::DataFrag` to store the literal strings, and `tree::ProcFrag` to store the routines.

`fragments.*` (*src/tree/*) [File]
Lists of `tree::Fragment`.

`visitor.*` (*src/tree/*) [File]
Implementation of `tree::Visitor` and `tree::ConstVisitor` to implement function objects on `tree::Fragments`. In other words, these visitors implement polymorphic operations on `tree::Fragment`.

3.2.19 The `src/frame` Directory

Namespace '`frame`', delivered for TC-5.

`access.*` (*src/frame/*) [File]
An `Access` is a location of a variable: on the stack, or in a temporary.

`frame.*` (*src/frame/*) [File]
A `Frame` knows only what are the "variables" it contains.

3.2.20 The `src/translate` Directory

Namespace '`translate`'. Translation to intermediate code translation. It includes:

`libtranslate.*` (*src/translate/*) [File]
The interface.

`access.*` (*src/translate/*) [File]
Static link aware versions of `level::Access`.

`level.*` (*src/translate/*) [File]
`translate::Level` are wrappers `frame::Frame` that support the static links, so that we can find an access to the variables of the "parent function".

`exp.hh` (*src/translate/*) [File]
Implementation of `translate::Ex` (expressions), `Nx` (instructions), `Cx` (conditions), and `Ix` (`if`) shells. They wrap `tree::Tree` to delay their translation until the actual use is known.

`translation.hh` (*src/translate/*) [File]
functions used by the `translate::Translator` to translate the AST into HIR. For instance, it contains '`Exp* simpleVar(const Access& access, const Level& level)`', '`Exp* callExp(const temp::Label& label, std::list<Exp*> args)`' etc. which are routines that produce some '`Tree::Exp`'. They handle all the `unCx` etc. magic.

`translator.hh` (*src/translate/*) [File]
Implements the class '`Translator`' which performs the IR generation thanks to `translation.hh`. It must not be polluted with translation details: it is only coordinating the AST traversal with the invocation of translation routines. For instance, here is the translation of an '`ast::SimpleVar`':

```
virtual void operator()(const SimpleVar& e)
{
  exp_ = simpleVar(*var_access_[e.def_get()], *level_);
}
```

3.2.21 The `src/canon` Directory

Namespace `canon`.

3.2.22 The `src/assem` Directory

Namespace `assem`, delivered for TC-7.

This directory contains the implementation of the Assem language: yet another intermediate representation that aims at encoding an assembly language, plus a few needed features so that register allocation can be performed afterward. Given in full.

`instr.*` (*src/assem/*) [File]
`move.*` (*src/assem/*) [File]
`oper.*` (*src/assem/*) [File]
`label.*` (*src/assem/*) [File]
`comment.*` (*src/assem/*) [File]
Implementation of the basic types of assembly instructions.

`fragment.*` (*src/assem/*) [File]
Implementation of `assem::Fragment`, `assem::ProcFrag`, and `assem::DataFrag`. They are very similar to `tree::Fragment`: aggregate some information that must remain together, such as a `frame::Frame` and the instructions (a list of `assem::Instr`).

`visitor.hh` (*src/assem/*) [File]
The root of assembler visitors.

`layout.hh` (*src/assem/*) [File]
A pretty printing visitor for `assem::Fragment`.

`libassem.*` (*src/assem/*) [File]
The interface of the module, and its implementation.

3.2.23 The `src/target` Directory

Namespace `target`, delivered for TC-7. Some data on the back end.

`cpu.*` (*src/target/*) [File]
Description of a CPU: everything about its registers, and its word size.

`target.*` (*src/target/*) [File]
Description of a target (language): its CPU, its assembly (`target::Assembly`), and it translator (`target::Codegen`).

`assembly.*` (*src/target/*) [File]
The abstract class `target::Assembly`, the interface for elementary assembly instructions generation.

`codegen.*` (*src/target/*) [File]
The abstract class `target::Codegen`, the interface for all our back ends.

`mips` (*src/target/*) [Directory]
`ia32` (*src/target/*) [Directory]
`arm` (*src/target/*) [Directory]
The instruction selection per se split into a generic part, and a target specific (MIPS, IA-32 and ARM) part. See Section 3.2.24 [src/target/mips], page 66, Section 3.2.25 [src/target/ia32], page 67, and Section 3.2.26 [src/target/arm], page 68.

`libtarget.*` (*src/target/*) [File]
Converting `tree::Fragment`s into `assem::Fragment`s.

`tiger-runtime.c` (*src/target/*) [File]
This is the Tiger runtime, written in C, based on Andrew Appel's `runtime.c`[4]. The actual `runtime.s` file for MIPS was written by hand, but the IA-32 was a compiled version of this file. It should be noted that:

Strings — Strings are implemented as 4 bytes to encode the length, and then a 0-terminated à la C string. The length part is due to conformance to the

[4] `http://www.cs.princeton.edu/~appel/modern/java/chap12/runtime.c`.

Tiger Reference Manual, which specifies that 0 is a regular character that can be part of the strings, but it is nevertheless terminated by 0 to be compliant with SPIM/Nolimips' `print` syscall. This might change in the future.

Special Strings
: There are some special strings: 0 and 1 character long strings are all implemented via a singleton. That is to say there is only one allocated string '`""`', a single '`"1"`' etc. These singletons are allocated by `main`. It is essential to preserve this invariant/convention in the whole runtime.

`strcmp` vs. `stringEqual`
: We don't know how Appel wants to support '`"bar" < "foo"`' since he doesn't provide `strcmp`. We do. His implementation of equality is more efficient than ours though, since he can decide just be looking at the lengths. That could be improved in the future...

`main`
: The runtime has some initializations to make, such as strings singletons, and then calls the compiled program. This is why the runtime provides `main`, and calls `tc_main`, which is the "main" that your compiler should provide.

3.2.24 The `src/target/mips` Directory

Namespace `target::mips`, delivered for TC-7. Code generation for MIPS R2000.

`cpu.*` (*src/target/mips/*) [File]
: The description of the MIPS (actually, SPIM/Nolimips) CPU.

`spim-assembly.*` (*src/target/mips/*) [File]
: Our assembly language (syntax, opcodes and layout); it abstracts the generation of MIPS R2000 instructions. `target::mips::SpimAssembly` derives from `target::Assembly`.

`spim-layout.*` (*src/target/mips/*) [File]
: How MIPS (and SPIM/Nolimips) fragments are to be displayed. In other words, that's where the (global) syntax of the target assembly file is selected.

`codegen.*` (*src/target/mips/*) [File]
`tree.brg` (*src/target/mips/*) [File]
`exp.brg` (*src/target/mips/*) [File]
`binop.brg` (*src/target/mips/*) [File]
`call.brg` (*src/target/mips/*) [File]
`temp.brg` (*src/target/mips/*) [File]
`mem.brg` (*src/target/mips/*) [File]
`stm.brg` (*src/target/mips/*) [File]
`move.brg` (*src/target/mips/*) [File]
`move_load.brg` (*src/target/mips/*) [File]
`move_store.brg` (*src/target/mips/*) [File]
`cjump.brg` (*src/target/mips/*) [File]
`prologue.hh` (*src/target/mips/*) [File]
`epilogue.cc` (*src/target/mips/*) [File]
: A translator from LIR to ASSEM using the MIPS R2000 instruction set defined by `target::mips::SpimAssembly`. It is implemented as a dynamic programming algorithm generated by MonoBURG from a set of `brg` files. `target::mips::Codegen` derives from `target::Codegen`.

`target.*` (*src/target/mips/*) [File]
The main back end, based on a MIPS CPU and a MIPS code generator.

`runtime.s` (*src/target/mips/*) [File]
`runtime.cc` (*src/target/mips/*) [File]
The Tiger runtime in MIPS assembly language: `print` etc. The C++ file `runtime.cc` is built from `runtime.s`: do not edit the former. See Section 3.2.23 [src/target], page 65, `tiger-runtime.c`.

3.2.25 The `src/target/ia32` Directory

Namespace `target::ia32`, delivered for TC-7. Code generation for IA-32. This is not part of the student project, but it is left to satisfy their curiosity. In addition its presence is a sane invitation to respect the constraints of a multi-back-end compiler.

`cpu.*` (*src/target/ia32*) [File]
Description of the i386 CPU.

`gas-assembly.*` (*src/target/ia32/*) [File]
The IA-32 assembly language (syntax, opcodes and layout); it abstracts the generation of IA-32 instructions using the GNU Assembler (Gas) syntax. `target::ia32::GasAssembly` derives from `target::Assembly`.

`gas-layout.*` (*src/target/ia32/*) [File]
How IA-32 fragments are to be displayed. In other words, that's where the (global) syntax of the target assembly file is selected.

`codegen.*` (*src/target/ia32/*) [File]
`tree.brg` (*src/target/ia32/*) [File]
`exp.brg` (*src/target/ia32/*) [File]
`binop.brg` (*src/target/ia32/*) [File]
`call.brg` (*src/target/ia32/*) [File]
`temp.brg` (*src/target/ia32/*) [File]
`mem.brg` (*src/target/ia32/*) [File]
`stm.brg` (*src/target/ia32/*) [File]
`move.brg` (*src/target/ia32/*) [File]
`move_load.brg` (*src/target/ia32/*) [File]
`move_store.brg` (*src/target/ia32/*) [File]
`cjump.brg` (*src/target/ia32/*) [File]
`prologue.hh` (*src/target/ia32/*) [File]
`epilogue.cc` (*src/target/ia32/*) [File]
A translator from LIR to ASSEM using the IA-32 instruction set defined by `target::ia32::GasAssembly`. It is implemented as a dynamic programming algorithm generated by MonoBURG from a set of `brg` files. `target::ia32::Codegen` derives from `target::Codegen`.

`target.*` (*src/target/ia32/*) [File]
The IA-32 back-end, based on an IA-32 CPU and an IA-32 code generator.

`runtime-gnu-linux.s` (*src/target/ia32/*) [File]
`runtime-freebsd.s` (*src/target/ia32/*) [File]
The GNU/Linux and FreeBSD Tiger runtimes in IA-32 assembly language: `print` etc. The C++ files `runtime-gnu-linux.cc` and `runtime-freebsd.cc` are built from `runtime-gnu-linux.s` and `runtime-freebsd.s`: do not edit the former. See Section 3.2.23 [src/target], page 65, `tiger-runtime.c`.

3.2.26 The `src/target/arm` Directory

Namespace `target::arm`, delivered for TC-7. Code generation for ARM. This is not part of the student project, but it is left to satisfy their curiosity. In addition its presence is a sane invitation to respect the constraints of a multi-back-end compiler.

`cpu.*` (*src/target/arm*) [File]
Description of the ARMV7 CPU.

`arm-assembly.*` (*src/target/arm/*) [File]
The ARM assembly language (syntax, opcodes and layout); it abstracts the generation of ARM instructions. `target::arm::ArmAssembly` derives from `target::Assembly`.

`arm-layout.*` (*src/target/arm/*) [File]
How ARM fragments are to be displayed. In other words, that's where the (global) syntax of the target assembly file is selected.

`arm-codegen.*` (*src/target/arm/*) [File]
`tree.brg` (*src/target/arm/*) [File]
`exp.brg` (*src/target/arm/*) [File]
`binop.brg` (*src/target/arm/*) [File]
`call.brg` (*src/target/arm/*) [File]
`temp.brg` (*src/target/arm/*) [File]
`mem.brg` (*src/target/arm/*) [File]
`stm.brg` (*src/target/arm/*) [File]
`move.brg` (*src/target/arm/*) [File]
`move_load.brg` (*src/target/arm/*) [File]
`move_store.brg` (*src/target/arm/*) [File]
`cjump.brg` (*src/target/arm/*) [File]
`prologue.hh` (*src/target/arm/*) [File]
`epilogue.cc` (*src/target/arm/*) [File]
A translator from LIR to ASSEM using the ARM instruction set defined by `target::arm::ArmAssembly`. It is implemented as a dynamic programming algorithm generated by MonoBURG from a set of `brg` files. `target::arm::Codegen` derives from `target::Codegen`.

`target.*` (*src/target/arm/*) [File]
The ARM back-end, based on an ARM CPU and an ARM code generator.

`runtime.s` (*src/target/arm/*) [File]
The Tiger runtime in ARM assembly language: `print` etc.

3.2.27 The `src/liveness` Directory

Namespace `liveness`, delivered for TC-8.

`flowgraph.*` (*src/liveness/*) [File]
`FlowGraph` implementation.

`test-flowgraph.cc` (*src/liveness/*) [File]
`FlowGraph` test.

`liveness.*` (*src/liveness/*) [File]
Computing the live-in and live-out information from the `FlowGraph`.

`interference-graph.*` (*src/liveness/*) [File]
Computing the `InterferenceGraph` from the live-in/live-out information.

3.2.28 The `src/llvmtranslate` Directory

Namespace `llvmtranslate`, delivered for TC-5. Translate the AST to LLVM intermediate code using the LLVM libraries.

`escapes-collector.*` (*src/llvmtranslate/*) [File]
The `FrameBuilder` and the `EscapesCollector`.

LLVM IR doesn't support static link and nested functions. In order to translate those functions to LLVM IR, we use Lambda Lifting, which consists in passing a pointer to the escaped variables to the nested function using that variable.

In order to do that, we need a visitor to collect these kind of variables and associate them to each function.

This visitor is the `EscapesCollector`.

In order for the `EscapesCollector` to work properly, the variables located in the function's frame have to be excluded. The `FrameBuilder` is building a frame for the `EscapesCollector` to use.

`libllvmtranslate.*` (*src/llvmtranslate/*) [File]
The interface.

`llvm-type-visitor.*` (*src/llvmtranslate/*) [File]
The LLVM IR is a typed language. In order to ensure type safety, the Tiger types (`type::Type`) have to be translated to LLVM types (`llvm::Type`). In order to do that, this visitor defined in Section 3.2.28 [src/llvmtranslate], page 69, is used to traverse the type hierarchy and translate it to LLVM types.

`translator.hh` (*src/llvmtranslate/*) [File]
Implements the class '`Translator`' which performs the LLVM IR generation using the LLVM API.

For instance, here is the translation of a '`ast::SimpleVar`':

```
virtual void operator()(const SimpleVar& e)
{
  value_ = builder_.CreateLoad(access_var(e), e.name_get().get());
}
```

`tiger-runtime.c` (*src/llvmtranslate/*) [File]
This is the specific runtime for Section 4.21 [TC-L], page 212. It is based on the original runtime, with some adaptations for LLVM.

It is compiled to LLVM IR in `$(build_dir)/src/llvmtranslate/runtime.ll`, then a function `llvmtranslate::runtime_string()` is generated in `$(build_dir)/src/llvmtranslate/runtime.cc`.

This function is used by the task `--llvm-runtime-display` to print the runtime along the LLVM IR.

Strings — Strings are implemented as `char*` 0-terminated buffers, like C strings.

Functions — Most of the built-ins are just calls to the C standard library functions.

Characters

Since the type `char` doesn't exist in TC, a `char` is nothing more than a `string` of length 1.

In order to avoid allocations every time a character is asked for, an array containing all the characters followed by a `\0` is initialized at the beginning of the program.

`main` The runtime initializes the one-character strings, then calls `tc_main`, which is the `main` that your compiler should have provided.

3.2.29 The `src/regalloc` Directory

Namespace `regalloc`, register allocation, delivered for TC-9.

`color.*` (*src/regalloc/*) [File]
Coloring an interference graph.

`regallocator.*` (*src/regalloc/*) [File]
Repeating the coloration until it succeeds (no spills).

`libregalloc.*` (*src/regalloc/*) [File]
Removing useless `move`s once the register allocation performed, and allocating the register for fragments.

`test-regalloc.cc` (*src/regalloc/*) [File]
Exercising this.

3.3 Given Test Cases

We provide a few test cases: *you must write your own tests. Writing tests is part of the project.* Do not just copy test cases from other groups, as you will not understand why they were written.

The initial test suite is available for download at `tests.tgz`[5]. It contains the following directories:

`good` These programs are correct.

`bind` These programs have bind mismatches.

`syntax` These programs have syntactial errors.

`type` These programs contain type mismatches.

[5] `https://www.lrde.epita.fr/~tiger//tc/tests.tgz`.

4 Compiler Stages

The compiler will be written in several steps, described below.

4.1 Stage Presentation

The following sections adhere to a standard layout in order to present each stage *n*:

Introduction
: The first few lines specify the last time the section was updated, the class for which it is written, and the submission dates. It also briefly describes the stage.

T*n* Goals, What this stage teaches
: This section details the goals of the stage as a teaching exercise. Be sure that examiners will make sure you understood these points. They also have instructions to ask questions about previous stages.

T*n* Samples, See T*n* work
: Actual examples generated from the reference compilers are exhibited to present and "specify" the stage.

T*n* Given Code, Explanation on the provided code
: This subsection points to the on line material we provide, introduces its components, quickly presents their designs and so forth. Check out the developer documentation of the Tiger Compiler[1] for more information, as the code is (hopefully) properly documented.

T*n* Code to Write, Explanation on what you have to write
: But of course, this code is not complete; this subsection provides hints on what is expected, and where.

T*n* Options, Want some more?
: During some stages, those who find the main task too easy can implement more features. These sections suggest possible additional features.

T*n* FAQ, Questions not to ask
: Each stage sees a blossom of new questions, some of which being extremely pertinent. We selected the most important ones, those that you should be aware of, contrary to many more questions that you ought to find and ask yourselves. These sections answer this few questions. And since they are already answered, you should not ask them...

T*n* Improvements, Other Designs
: The Tiger Compiler is an instructional project the audience of which is *learning* C++. Therefore, although by the end of the development, in the latter stages, we can expect able C++ programmers, most of the time we have to refrain from using advanced designs, or intricate C++ techniques. These sections provide hints on what could have been done to improve the stage. You can think of these sections as material you ought to read once the project is over and you are a grown-up C++ programmer.

[1] `https://www.lrde.epita.fr/~tiger/tc-doc/`.

4.2 PTHL (TC-0), Naive Scanner and Parser

2020-PTHL submission is Wednesday, February 1st 2018 at 19:42.

This section has been updated for EPITA-2020 on 2015-11-16.

TC-0 is a weak form of TC-1: the scanner and the parser are written, but the framework is simplified (see Section 4.3.4 [TC-1 Code to Write], page 86). The grammar is also simpler: object-related productions are not to be supported at this stage (see Section 4.2.5 [PTHL Improvements], page 79). No command line option is supported.

4.2.1 PTHL Goals

Things to learn during this stage that you should remember:

- Writing/debugging a scanner with Flex.
- Using start conditions to handle non-regular issues within the scanner.
- Using '`yy::parser::make_`*`SYMBOL`*' to build whole symbols (containing a token type, a location, and possibly a semantic value) and pass them to the parser.
- Writing/debugging a parser with Bison.
- Resolving simple conflicts due to precedences and associativities thanks to directives (e.g., '`%left`' etc.).
- Resolving hard conflicts with loop unrolling. The case of lvalue vs. array instantiation is of first importance.

4.2.2 PTHL Samples

First, please note that all the samples, including in this section, are generated with a TC-1+ compliant compiler: its behavior differs from that of a TC-0 compiler. In particular, for the time being, forget about the options (`-X` and `--parse`).

Running TC-0 basically consists in looking at exit values:

```
print("Hello, World!\n")
```

File 4.1: `simple.tig`

```
$ tc simple.tig
```

Example 4.1: *`tc simple.tig`*

The following example demonstrates the scanner and parser tracing. The glyphs "error" and "⇒" are typographic conventions to specify respectively the standard error stream and the exit status. *They are not part of the output per se.*

```
$ SCAN=1 PARSE=1 tc -X --parse simple.tig
error Parsing file: "simple.tig"
error Starting parse
error Entering state 0
error Reading a token: --(end of buffer or a NUL)
error --accepting rule at line 196("print")
error Next token is token "identifier" (simple.tig:1.1-5: print)
error Shifting token "identifier" (simple.tig:1.1-5: print)
error Entering state 2
error Reading a token: --accepting rule at line 138("(")
error Next token is token "(" (simple.tig:1.6: )
error Reducing stack 0 by rule 100 (line 626):
error    $1 = token "identifier" (simple.tig:1.1-5: print)
error -> $$ = nterm funid (simple.tig:1.1-5: print)
```

```
error Entering state 36
error Next token is token "(" (simple.tig:1.6: )
error Shifting token "(" (simple.tig:1.6: )
error Entering state 85
error Reading a token: --accepting rule at line 197("""")
error --accepting rule at line 266("Hello, World!")
error --accepting rule at line 253("\n")
error --accepting rule at line 228("""")
error Next token is token "string" (simple.tig:1.7-23: Hello, World!
error )
error Shifting token "string" (simple.tig:1.7-23: Hello, World!
error )
error Entering state 1
error Reducing stack 0 by rule 4 (line 296):
error    $1 = token "string" (simple.tig:1.7-23: Hello, World!
error )
error -> $$ = nterm exp (simple.tig:1.7-23: "Hello, World!\n")
error Entering state 131
error Reading a token: --accepting rule at line 139(")")
error Next token is token ")" (simple.tig:1.24: )
error Reducing stack 0 by rule 45 (line 417):
error    $1 = nterm exp (simple.tig:1.7-23: "Hello, World!\n")
error -> $$ = nterm args.1 (simple.tig:1.7-23: "Hello, World!\n")
error Entering state 133
error Next token is token ")" (simple.tig:1.24: )
error Reducing stack 0 by rule 44 (line 412):
error    $1 = nterm args.1 (simple.tig:1.7-23: "Hello, World!\n")
error -> $$ = nterm args (simple.tig:1.7-23: "Hello, World!\n")
error Entering state 132
error Next token is token ")" (simple.tig:1.24: )
error Shifting token ")" (simple.tig:1.24: )
error Entering state 174
error Reducing stack 0 by rule 6 (line 304):
error    $1 = nterm funid (simple.tig:1.1-5: print)
error    $2 = token "(" (simple.tig:1.6: )
error    $3 = nterm args (simple.tig:1.7-23: "Hello, World!\n")
error    $4 = token ")" (simple.tig:1.24: )
error -> $$ = nterm exp (simple.tig:1.1-24: print("Hello, World!\n"))
error Entering state 25
error Reading a token: --(end of buffer or a NUL)
error --accepting rule at line 134("
error ")
error --(end of buffer or a NUL)
error --EOF (start condition 0)
error Now at end of input.
error Reducing stack 0 by rule 1 (line 287):
error    $1 = nterm exp (simple.tig:1.1-24: print("Hello, World!\n"))
error -> $$ = nterm program (simple.tig:1.1-24: )
error Entering state 24
error Now at end of input.
error Shifting token "end of file" (simple.tig:2.1: )
error Entering state 63
```

```
error Cleanup: popping token "end of file" (simple.tig:2.1: )
error Cleanup: popping nterm program (simple.tig:1.1-24: )
error Parsing string: function _main() = (_exp(0); ())
error Starting parse
error Entering state 0
error Reading a token: --(end of buffer or a NUL)
error --accepting rule at line 164("function")
error Next token is token "function" (:1.1-8: )
error Shifting token "function" (:1.1-8: )
error Entering state 8
error Reading a token: --accepting rule at line 133(" ")
error --accepting rule at line 195("_main")
error Next token is token "identifier" (:1.10-14: _main)
error Shifting token "identifier" (:1.10-14: _main)
error Entering state 43
error Reading a token: --accepting rule at line 138("(")
error Next token is token "(" (:1.15: )
error Shifting token "(" (:1.15: )
error Entering state 93
error Reading a token: --accepting rule at line 139(")")
error Next token is token ")" (:1.16: )
error Reducing stack 0 by rule 95 (line 605):
error -> $$ = nterm funargs (:1.16: )
error Entering state 144
error Next token is token ")" (:1.16: )
error Shifting token ")" (:1.16: )
error Entering state 186
error Reading a token: --accepting rule at line 133(" ")
error --accepting rule at line 152("=")
error Next token is token "=" (:1.18: )
error Reducing stack 0 by rule 86 (line 567):
error -> $$ = nterm typeid.opt (:1.17: )
error Entering state 215
error Next token is token "=" (:1.18: )
error Shifting token "=" (:1.18: )
error Entering state 231
error Reading a token: --accepting rule at line 133(" ")
error --accepting rule at line 138("(")
error Next token is token "(" (:1.20: )
error Shifting token "(" (:1.20: )
error Entering state 12
error Reading a token: --accepting rule at line 191("_exp")
error Next token is token "_exp" (:1.21-24: )
error Shifting token "_exp" (:1.21-24: )
error Entering state 21
error Reading a token: --accepting rule at line 138("(")
error Next token is token "(" (:1.25: )
error Shifting token "(" (:1.25: )
error Entering state 60
error Reading a token: --accepting rule at line 113("0")
error Next token is token "integer" (:1.26: 0)
error Shifting token "integer" (:1.26: 0)
```

```
error Entering state 106
error Reading a token: --accepting rule at line 139(")")
error Next token is token ")" (:1.27: )
error Shifting token ")" (:1.27: )
error Entering state 164
error Reducing stack 0 by rule 37 (line 397):
error    $1 = token "_exp" (:1.21-24: )
error    $2 = token "(" (:1.25: )
error    $3 = token "integer" (:1.26: 0)
error    $4 = token ")" (:1.27: )
error -> $$ = nterm exp (:1.21-27: print("Hello, World!\n"))
error Entering state 48
error Reading a token: --accepting rule at line 148(";")
error Next token is token ";" (:1.28: )
error Reducing stack 0 by rule 48 (line 424):
error    $1 = nterm exp (:1.21-27: print("Hello, World!\n"))
error -> $$ = nterm exps.1 (:1.21-27: print("Hello, World!\n"))
error Entering state 49
error Next token is token ";" (:1.28: )
error Shifting token ";" (:1.28: )
error Entering state 99
error Reading a token: --accepting rule at line 133(" ")
error --accepting rule at line 138("(")
error Next token is token "(" (:1.30: )
error Shifting token "(" (:1.30: )
error Entering state 12
error Reading a token: --accepting rule at line 139(")")
error Next token is token ")" (:1.31: )
error Reducing stack 0 by rule 52 (line 436):
error -> $$ = nterm exps.0.2 (:1.31: )
error Entering state 51
error Next token is token ")" (:1.31: )
error Shifting token ")" (:1.31: )
error Entering state 100
error Reducing stack 0 by rule 11 (line 321):
error    $1 = token "(" (:1.30: )
error    $2 = nterm exps.0.2 (:1.31: )
error    $3 = token ")" (:1.31: )
error -> $$ = nterm exp (:1.30-31: ())
error Entering state 153
error Reading a token: --(end of buffer or a NUL)
error --accepting rule at line 139(")")
error Next token is token ")" (:1.32: )
error Reducing stack 0 by rule 51 (line 431):
error    $1 = nterm exps.1 (:1.21-27: print("Hello, World!\n"))
error    $2 = token ";" (:1.28: )
error    $3 = nterm exp (:1.30-31: ())
error -> $$ = nterm exps.2 (:1.21-31: print("Hello, World!\n"), ())
error Entering state 50
error Reducing stack 0 by rule 53 (line 437):
error    $1 = nterm exps.2 (:1.21-31: print("Hello, World!\n"), ())
error -> $$ = nterm exps.0.2 (:1.21-31: print("Hello, World!\n"), ())
```

```
error Entering state 51
error Next token is token ")" (:1.32: )
error Shifting token ")" (:1.32: )
error Entering state 100
error Reducing stack 0 by rule 11 (line 321):
error    $1 = token "(" (:1.20: )
error    $2 = nterm exps.0.2 (:1.21-31: print("Hello, World!\n"), ())
error    $3 = token ")" (:1.32: )
error -> $$ = nterm exp (:1.20-32: (
error   print("Hello, World!\n");
error   ()
error ))
error Entering state 239
error Reading a token: --(end of buffer or a NUL)
error --EOF (start condition 0)
error Now at end of input.
error Reducing stack 0 by rule 93 (line 598):
error    $1 = token "function" (:1.1-8: )
error    $2 = token "identifier" (:1.10-14: _main)
error    $3 = token "(" (:1.15: )
error    $4 = nterm funargs (:1.16: )
error    $5 = token ")" (:1.16: )
error    $6 = nterm typeid.opt (:1.17: )
error    $7 = token "=" (:1.18: )
error    $8 = nterm exp (:1.20-32: (
error   print("Hello, World!\n");
error   ()
error ))
error -> $$ = nterm fundec (:1.1-32:
error function _main() =
error   (
error     print("Hello, World!\n");
error     ()
error   ))
error Entering state 35
error Now at end of input.
error Reducing stack 0 by rule 91 (line 593):
error    $1 = nterm fundec (:1.1-32:
error function _main() =
error   (
error     print("Hello, World!\n");
error     ()
error   ))
error -> $$ = nterm fundecs (:1.1-32:
error function _main() =
error   (
error     print("Hello, World!\n");
error     ()
error   ))
error Entering state 34
error Now at end of input.
error Reducing stack 0 by rule 54 (line 447):
```

```
error -> $$ = nterm decs (:1.33: )
error Entering state 83
error Reducing stack 0 by rule 57 (line 451):
error    $1 = nterm fundecs (:1.1-32:
error function _main() =
error   (
error     print("Hello, World!\n");
error     ()
error   ))
error    $2 = nterm decs (:1.33: )
error -> $$ = nterm decs (:1.1-32:
error function _main() =
error   (
error     print("Hello, World!\n");
error     ()
error   ))
error Entering state 27
error Reducing stack 0 by rule 2 (line 289):
error    $1 = nterm decs (:1.1-32:
error function _main() =
error   (
error     print("Hello, World!\n");
error     ()
error   ))
error -> $$ = nterm program (:1.1-32: )
error Entering state 24
error Now at end of input.
error Shifting token "end of file" (:1.33: )
error Entering state 63
error Cleanup: popping token "end of file" (:1.33: )
error Cleanup: popping nterm program (:1.1-32: )
```

Example 4.2: *SCAN=1 PARSE=1 tc -X --parse simple.tig*

A lexical error must be properly diagnosed *and reported*. The following (generated) examples display the location: *this is not required for TC-0*; nevertheless, an error message on the standard error output is required.

```
"\z does not exist."
```

File 4.2: back-zee.tig

```
$ tc -X --parse back-zee.tig
error back-zee.tig:1.1-3: unrecognized escape: \z
⇒2
```

Example 4.3: *tc -X --parse back-zee.tig*

Similarly for syntactical errors.

```
a++
```

File 4.3: postinc.tig

```
$ tc -X --parse postinc.tig
error postinc.tig:1.3: syntax error, unexpected +
```

```
⇒3
```

Example 4.4: `tc -X --parse postinc.tig`

4.2.3 PTHL Code to Write

We don't need several directories, you can program in the top level of the package.

You must write:

`src/scantiger.ll`
: The scanner.

  `lval` supports strings, integers and even symbols. Nevertheless, symbols (i.e., identifiers) are returned as plain C++ strings for the time being: the class `misc::symbol` is introduced in TC-1.

  If the environment variable `SCAN` is defined (to whatever value) Flex scanner debugging traces are enabled, i.e., set the variable `yy_flex_debug` to 1.

`src/parsetiger.yy`
: The parser, and maybe `main` if you wish. Bison advanced features will be used in TC-1.

  - Use C++ (e.g., C++ I/O streams, strings, etc.)
  - Use C++ features of Bison.
  - Use locations.
  - Use '`%expect 0`' to have Bison report conflicts are genuine errors.
  - Use the '`%require "3.0"`' directive to prevent any problem due to old versions of Bison.
  - Use the '`%define api.value.type variant`' directive to ask Bison for C++ object support in the semantic values. Without this, Bison uses `union`, which can be used to store objects (just Plain Old Data), hence pointers and dynamic allocation must be used.
  - Use the '`%define api.token.constructor`' directive to request that symbols be handled as a whole (token type, location, and possibly semantic value) in the scanner through '`parse::parser::make_`*`SYMBOL`*' routines.
  - Use the environment variable `PARSE` to enable parser traces, i.e., to set `yydebug` to 1, run:

    ```
    PARSE=1 tc foo.tig
    ```

  - Use `%printer` to improve the tracing of semantic values. For instance,

    ```
    %define api.value.type variant
    %token <int> INT "integer"
    %printer { yyo << $$; } <int>
    ```

`Makefile` This file is mandatory. Running `make` *must build an executable* `tc` *in the root directory.* The GNU Build System is not mandatory: TC-1 introduces Autoconf, Automake etc. You may use it, in which case we will run `configure` before `make`.

4.2.4 PTHL FAQ

Translating escapes in the scanner (or not)
: Escapes in string can be translated at the scanning stage, or kept as is. That is, the string `"\n"` can produce a token `STRING` with the semantic value `\n`

(translation) or `\\n` (no translation). You are free to choose your favorite implementation, but keep in mind that if you translate, you'll have to "untranslate" later (i.e., convert `\n` back to `\\n`).

We encourage you to do this translation, but the other solution is also correct, as long as the next steps of your compiler follow the same conventions as your input.

You must check for bad escapes whatever solution you choose.

Must lexical and syntactic extensions be implemented?

No. Language extensions (see Section "Language Extensions" in *Tiger Compiler Reference Manual*) such as metavariables keywords ('`_decs`', '`_exp`', '`_lvalue`', '`_namety`') and casts ('`_cast`') are not required for PTHL.

Handling metavariables constructs becomes mandatory at TC-2 (see Section 4.4.4 [TC-2 Code to Write], page 95) where they are used within TWEASTS (Text With Embedded AST, see `ast.pdf`[2]), while casts are only needed for the optional bounds checking assignment (see Section 4.11 [TC-B], page 119).

What values can be represented by an `int`?

The set of valid integer values is the set of signed 32-bit integers in 2's complement, that is the integer interval $[-2^{31}, 2^{31} - 1]$.

What values can be represented by an integer literal?

Although an integer *value* can be any number in $[-2^{31}, 2^{31} - 1]$, it is however not possible to represent the *literal* $-2^{31}(= -2147483648)$ for technical reasons. It is however possible to create an integer value representing this number.

To put it in nutshell, the following declaration is not valid:

```
var i := -2147483648
```

whereas this one is:

```
var i := -2147483647 - 1
```

4.2.5 PTHL Improvements

Possible improvements include:

Using `%destructor`

You may use `%destructor` to reclaim the memory lost during the error recovery. It is mandated in TC-2, see Section 4.4.5 [TC-2 FAQ], page 96.

Parser driver

You may implement a parser driver to handle the parsing context (flags, open files, etc.). Note that a driver class will be (partially) provided at TC-1.

Handling object-related constructs from PTHL

Your scanner and parser are not required to support OO constructs at PTHL, but you can implement them in your LALR(1) parser if you want. (Fully supporting them at TC-2 is highly recommended though, during the conversion of your LALR(1) parser to a GLR one.)

Object-related productions from the Tiger grammar[3] are:

```
# Class definition (canonical form).
ty ::= 'class' [ 'extends' type-id ] '{ ' classfields '} '
```

[2] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/ast.pdf`.

[3] `https://www.lrde.epita.fr/~tiger/tiger.split/Syntactic-Specifications.html`.

```
# Class definition (alternative form).
dec ::= ‘class’ id [ ‘extends’ type-id ] ‘{ ’ classfields ‘} ’

classfields ::= { classfield }
# Class fields.
classfield ::=
  # Attribute declaration.
    vardec
  # Method declaration.
  | ‘method’ id ‘(’ tyfields ‘)’ [ ‘:’ type-id ] ‘=’ exp

# Object creation.
exp ::= ‘new’ type-id

# Method call.
exp ::= lvalue ‘.’ id ‘(’ [ exp { ‘,’ exp }] ‘)’
```

4.3 TC-1, Scanner and Parser

2020-TC-1 submission for Ing1 students is Sunday, February 4th 2018 at 11:42.

This section has been updated for EPITA-2020 on 2016-01-27.

Scanner and parser are properly running, but the abstract syntax tree is not built yet. Differences with PTHL (TC-0) include:

GNU Build System
: Autoconf, Automake are used.

Options, Tasks
: The compiler supports basic options via in the Task module. See Section “Invoking tc” in *Tiger Compiler Reference Manual*, for the list of options to support.

Locations
: The locations are properly computed and reported in the error messages.

Relevant lecture notes include `dev-tools.pdf`[4] and `scanner.pdf`[5].

4.3.1 TC-1 Goals

Things to learn during this stage that you should remember:

Basic use of the GNU Build System
: Autoconf, Automake. The initial set up of the project will best be done via ‘`autoreconf -fvim`’, but once the project initiated (i.e., `configure` and the `Makefile.in`s exist) you should depend on `make` only. See Section 5.4 [The GNU Build System], page 247.

Integration into an existing framework
: Putting your own code into the provided code base.

Basic C++ classes
: The classes `Location` and `Position` provide a good start to study foreign C++ classes. Your understanding them will be controlled, including the ‘`operator`’s.

[4] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/dev-tools.pdf`.

[5] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/scanner.pdf`.

Location Tracking
: Issues within the scanner and the parser.

Implementation of a few simple C++ classes
: The code for `misc::symbol` and `misc::unique` is incomplete.

A first standard container: `std::set`
: The implementation of the `misc::unique` class relies on `std::set`.

The Flyweight design pattern
: The `misc::unique` class is an implementation of the Flyweight design pattern.

Version Control System
: Using the Git version control system is mandatory. Your understanding of it will be checked.

4.3.2 TC-1 Samples

The only information the compiler provides is about lexical and syntax errors. If there are no errors, the compiler shuts up, and exits successfully:

```
/* An array type and an array variable. */
let
  type  arrtype = array of int
  var arr1 : arrtype := arrtype [10] of 0
in
  arr1[2]
end
```

File 4.4: `test01.tig`

```
$ tc -X --parse test01.tig
```

Example 4.5: *`tc -X --parse test01.tig`*

If there are lexical errors, the exit status is 2, and an error message is output on the standard error output. Its format is standard and mandatory: file, (precise) location, and then the message (see Section "Errors" in *Tiger Compiler Reference Manual*).

```
1
 /* This comments starts at /* 2.2 */
```

File 4.5: `unterminated-comment.tig`

```
$ tc -X --parse unterminated-comment.tig
error unterminated-comment.tig:2.2-3.0: unexpected end of file in a comment
⇒2
```

Example 4.6: *`tc -X --parse unterminated-comment.tig`*

If there are syntax errors, the exit status is set to 3:

```
let var a : nil := ()
in
  1
end
```

File 4.6: `type-nil.tig`

```
$ tc -X --parse type-nil.tig
```

```
error type-nil.tig:1.13-15: syntax error, unexpected nil, expecting iden-
tifier or _namety
⇒3
```

Example 4.7: *tc -X --parse type-nil.tig*

If there are errors which are non lexical, nor syntactic (Windows will not pass by me):

```
$ tc C:/TIGER/SAMPLE.TIG
error tc: cannot open ‘C:/TIGER/SAMPLE.TIG’: No such file or directory
⇒1
```

Example 4.8: *tc C:/TIGER/SAMPLE.TIG*

The option `--parse-trace`, which relies on Bison's `%debug` and `%printer` directives, must work properly[6]:

```
a + "a"
```

File 4.7: `a+a.tig`

```
$ tc -X --parse-trace --parse a+a.tig
error Parsing file: "a+a.tig"
error Starting parse
error Entering state 0
error Reading a token: Next token is token "identifier" (a+a.tig:1.1: a)
error Shifting token "identifier" (a+a.tig:1.1: a)
error Entering state 2
error Reading a token: Next token is token "+" (a+a.tig:1.3: )
error Reducing stack 0 by rule 90 (line 585):
error    $1 = token "identifier" (a+a.tig:1.1: a)
error -> $$ = nterm varid (a+a.tig:1.1: a)
error Entering state 33
error Reducing stack 0 by rule 38 (line 402):
error    $1 = nterm varid (a+a.tig:1.1: a)
error -> $$ = nterm lvalue (a+a.tig:1.1: a)
error Entering state 26
error Next token is token "+" (a+a.tig:1.3: )
error Reducing stack 0 by rule 35 (line 395):
error    $1 = nterm lvalue (a+a.tig:1.1: a)
error -> $$ = nterm exp (a+a.tig:1.1: a)
error Entering state 25
error Next token is token "+" (a+a.tig:1.3: )
error Shifting token "+" (a+a.tig:1.3: )
error Entering state 74
error Reading a token: Next token is token "string" (a+a.tig:1.5-7: a)
error Shifting token "string" (a+a.tig:1.5-7: a)
error Entering state 1
error Reducing stack 0 by rule 4 (line 296):
error    $1 = token "string" (a+a.tig:1.5-7: a)
error -> $$ = nterm exp (a+a.tig:1.5-7: "a")
error Entering state 119
error Reading a token: Now at end of input.
```

[6] For the time being, forget about `-X`.

```
error Reducing stack 0 by rule 29 (line 376):
error    $1 = nterm exp (a+a.tig:1.1: a)
error    $2 = token "+" (a+a.tig:1.3: )
error    $3 = nterm exp (a+a.tig:1.5-7: "a")
error -> $$ = nterm exp (a+a.tig:1.1-7: (a + "a"))
error Entering state 25
error Now at end of input.
error Reducing stack 0 by rule 1 (line 287):
error    $1 = nterm exp (a+a.tig:1.1-7: (a + "a"))
error -> $$ = nterm program (a+a.tig:1.1-7: )
error Entering state 24
error Now at end of input.
error Shifting token "end of file" (a+a.tig:2.1: )
error Entering state 63
error Cleanup: popping token "end of file" (a+a.tig:2.1: )
error Cleanup: popping nterm program (a+a.tig:1.1-7: )
error Parsing string: function _main() = (_exp(0); ())
error Starting parse
error Entering state 0
error Reading a token: Next token is token "function" (:1.1-8: )
error Shifting token "function" (:1.1-8: )
error Entering state 8
error Reading a token: Next token is token "identifier" (:1.10-14: _main)
error Shifting token "identifier" (:1.10-14: _main)
error Entering state 43
error Reading a token: Next token is token "(" (:1.15: )
error Shifting token "(" (:1.15: )
error Entering state 93
error Reading a token: Next token is token ")" (:1.16: )
error Reducing stack 0 by rule 95 (line 605):
error -> $$ = nterm funargs (:1.16: )
error Entering state 144
error Next token is token ")" (:1.16: )
error Shifting token ")" (:1.16: )
error Entering state 186
error Reading a token: Next token is token "=" (:1.18: )
error Reducing stack 0 by rule 86 (line 567):
error -> $$ = nterm typeid.opt (:1.17: )
error Entering state 215
error Next token is token "=" (:1.18: )
error Shifting token "=" (:1.18: )
error Entering state 231
error Reading a token: Next token is token "(" (:1.20: )
error Shifting token "(" (:1.20: )
error Entering state 12
error Reading a token: Next token is token "_exp" (:1.21-24: )
error Shifting token "_exp" (:1.21-24: )
error Entering state 21
error Reading a token: Next token is token "(" (:1.25: )
error Shifting token "(" (:1.25: )
error Entering state 60
error Reading a token: Next token is token "integer" (:1.26: 0)
```

```
error Shifting token "integer" (:1.26: 0)
error Entering state 106
error Reading a token: Next token is token ")" (:1.27: )
error Shifting token ")" (:1.27: )
error Entering state 164
error Reducing stack 0 by rule 37 (line 397):
error    $1 = token "_exp" (:1.21-24: )
error    $2 = token "(" (:1.25: )
error    $3 = token "integer" (:1.26: 0)
error    $4 = token ")" (:1.27: )
error -> $$ = nterm exp (:1.21-27: (a + "a"))
error Entering state 48
error Reading a token: Next token is token ";" (:1.28: )
error Reducing stack 0 by rule 48 (line 424):
error    $1 = nterm exp (:1.21-27: (a + "a"))
error -> $$ = nterm exps.1 (:1.21-27: (a + "a"))
error Entering state 49
error Next token is token ";" (:1.28: )
error Shifting token ";" (:1.28: )
error Entering state 99
error Reading a token: Next token is token "(" (:1.30: )
error Shifting token "(" (:1.30: )
error Entering state 12
error Reading a token: Next token is token ")" (:1.31: )
error Reducing stack 0 by rule 52 (line 436):
error -> $$ = nterm exps.0.2 (:1.31: )
error Entering state 51
error Next token is token ")" (:1.31: )
error Shifting token ")" (:1.31: )
error Entering state 100
error Reducing stack 0 by rule 11 (line 321):
error    $1 = token "(" (:1.30: )
error    $2 = nterm exps.0.2 (:1.31: )
error    $3 = token ")" (:1.31: )
error -> $$ = nterm exp (:1.30-31: ())
error Entering state 153
error Reading a token: Next token is token ")" (:1.32: )
error Reducing stack 0 by rule 51 (line 431):
error    $1 = nterm exps.1 (:1.21-27: (a + "a"))
error    $2 = token ";" (:1.28: )
error    $3 = nterm exp (:1.30-31: ())
error -> $$ = nterm exps.2 (:1.21-31: (a + "a"), ())
error Entering state 50
error Reducing stack 0 by rule 53 (line 437):
error    $1 = nterm exps.2 (:1.21-31: (a + "a"), ())
error -> $$ = nterm exps.0.2 (:1.21-31: (a + "a"), ())
error Entering state 51
error Next token is token ")" (:1.32: )
error Shifting token ")" (:1.32: )
error Entering state 100
error Reducing stack 0 by rule 11 (line 321):
error    $1 = token "(" (:1.20: )
```

```
error    $2 = nterm exps.0.2 (:1.21-31: (a + "a"), ())
error    $3 = token ")" (:1.32: )
error -> $$ = nterm exp (:1.20-32: (
error   (a + "a");
error   ()
error ))
error Entering state 239
error Reading a token: Now at end of input.
error Reducing stack 0 by rule 93 (line 598):
error    $1 = token "function" (:1.1-8: )
error    $2 = token "identifier" (:1.10-14: _main)
error    $3 = token "(" (:1.15: )
error    $4 = nterm funargs (:1.16: )
error    $5 = token ")" (:1.16: )
error    $6 = nterm typeid.opt (:1.17: )
error    $7 = token "=" (:1.18: )
error    $8 = nterm exp (:1.20-32: (
error   (a + "a");
error   ()
error ))
error -> $$ = nterm fundec (:1.1-32:
error function _main() =
error   (
error     (a + "a");
error     ()
error   ))
error Entering state 35
error Now at end of input.
error Reducing stack 0 by rule 91 (line 593):
error    $1 = nterm fundec (:1.1-32:
error function _main() =
error   (
error     (a + "a");
error     ()
error   ))
error -> $$ = nterm fundecs (:1.1-32:
error function _main() =
error   (
error     (a + "a");
error     ()
error   ))
error Entering state 34
error Now at end of input.
error Reducing stack 0 by rule 54 (line 447):
error -> $$ = nterm decs (:1.33: )
error Entering state 83
error Reducing stack 0 by rule 57 (line 451):
error    $1 = nterm fundecs (:1.1-32:
error function _main() =
error   (
error     (a + "a");
error     ()
```

```
error   ))
error    $2 = nterm decs (:1.33: )
error -> $$ = nterm decs (:1.1-32:
error function _main() =
error   (
error     (a + "a");
error     ()
error   ))
error Entering state 27
error Reducing stack 0 by rule 2 (line 289):
error    $1 = nterm decs (:1.1-32:
error function _main() =
error   (
error     (a + "a");
error     ()
error   ))
error -> $$ = nterm program (:1.1-32: )
error Entering state 24
error Now at end of input.
error Shifting token "end of file" (:1.33: )
error Entering state 63
error Cleanup: popping token "end of file" (:1.33: )
error Cleanup: popping nterm program (:1.1-32: )
```

Example 4.9: *`tc -X --parse-trace --parse a+a.tig`*

Note that (i), `--parse` is needed, (ii), it cannot see that the variable is not declared nor that there is a type checking error, since type checking... is not implemented, and (iii), the output might be slightly different, depending upon the version of Bison you use. But what matters is that one can see the items: '`"identifier" a`', '`"string" a`'.

4.3.3 TC-1 Given Code

Some code is provided through the '`tc-base`' repository; use tags '`2020-tc-base-1.0`' to integrate it with your existing code base. See Section 3.1 [Given Code], page 55, for more information on using the '`tc-base`' Git repository.

See Section 3.2.1 [The Top Level], page 55, Section 3.2.5 [src], page 59, Section 3.2.7 [src/parse], page 59, Section 3.2.4 [lib/misc], page 56.

4.3.4 TC-1 Code to Write

Be sure to read Flex and Bison documentations and tutorials, see Section 5.9 [Flex & Bison], page 251.

`configure.ac`
`Makefile.am`
: Include your own test suite in the `tests` directory, and hook it to `make check`.

`src/parse/scantiger.ll`
: The scanner must be completed to read strings, identifiers etc. and track locations.

  – Strings will be stored as C++ `std::string`. See the following code for the basics.

```
...
\"          grown_string.clear(); BEGIN SC_STRING;
```

```
<SC_STRING>{ /* Handling of the strings.  Initial " is eaten. */
     \" {
       BEGIN INITIAL;
       return TOKEN_VAL(STRING, grown_string);
     }
...
     \\x[0-9a-fA-F]{2}  {
       grown_string.append(1, strtol(yytext + 2, 0, 16));
     }
...
}
```

- Symbols (i.e., identifiers) must be returned as `misc::symbol` objects, not strings.
- The locations are tracked. The class `Location` to use is produced by Bison: `src/parse/location.hh`.

  To track of locations, adjust your scanner, use `YY_USER_ACTION` and the `yylex` prologue:

```
...
%%
%{
  // Everything here is run each time yylex is invoked.
%}
"if"    return TOKEN(IF);
...
%%
...
```

  See the lecture notes, and read the C++ chapter of[7] GNU Bison's documentation. Note that the version being used for the Tiger project may differ from the latest public release, thus students should build their own documentation by running '`make html`' in the provided Bison tarball.

  Pay special attention to its "Complete C++ Example[8]" which is *very much* like our set up.

`src/parse/parsetiger.yy`

- The grammar must be complete but without actions.
- Use `%printer` to implement `--parse-trace` support for terminals (see Section 4.3.2 [TC-1 Samples], page 81)

`src/parse/tiger-parser.cc`

The class `TigerParser` drives the lexing and parsing of input file. Its implementation in `src/parse/tiger-parser.cc` is incomplete.

`lib/misc/symbol.*`
`lib/misc/unique.*`

The class `misc::symbol` keeps a single copy of identifiers, see Section 3.2.4 [lib/misc], page 56. Its implementation in `lib/misc/symbol.hxx` and `lib/misc/symbol.cc` is incomplete. Note that running '`make check`' in `lib/misc` exercises `lib/misc/test-symbol.cc`: having this unit test pass

[7] `http://www.gnu.org/software/bison/manual/bison.html`.

[8] `http://www.gnu.org/software/bison/manual/bison.html#A-Complete-C_002b_002b-Example`.

should be a goal by itself. As a matter of fact, unit tests were left to help you: once they pass successfully you may proceed to the rest of the compiler. `misc::symbol`'s implementation is based on `misc::unique`, a generic class implementing the Flyweight design pattern. The definition of this class, `lib/misc/unique.hxx`, is also to be completed.

`lib/misc/variant.*`
: The implementation of the class template `misc::variant<T0, Ts...>` lacks a couple of conversion operators that you have to supply.

4.3.5 TC-1 FAQ

Bison reports type clashes
: Bison may report type clashes for some actions. For instance, if you have given a type to `"string"`, but none to `exp`, then it will choke on:

```
exp: "string";
```

because, unless you used '`%define variant`', it actually means

```
exp: "string" { $$ = $1; };
```

which is not type consistent. So write this instead:

```
exp: "string" {};
```

Where is `ast::Exp`?
: Its real definition will be provided with TC-2, so meanwhile you have to provide a fake. We recommend for a forward declaration of '`ast::Exp`' in `libparse.hh`.

Finding `prelude.tih`
: When run, the compiler needs the file `prelude.tih` that includes the signature of all the primitives. But the executable `tc` is typically run in two very different contexts:

installed
: An installed binary will look for an installed `prelude.tih`, typically in `/usr/local/share/tc/`. The `cpp` macro `PKGDATADIR` is set to this directory. Its value depends on the use of `configure`'s option `--prefix`, defaulting to `/usr/local`.

compiled, not installed
: When compiled, the binary will look for the installed `prelude.tih`, and of course will fail if it has never been installed. There are two means to address this issue:

The environment variable `TC_PKGDATADIR`
: If set, it overrides the value of `PKGDATADIR`.

The option `--library-prepend/-p`
: Using this option you may set the library file search path to visit the given directory *before* the built-in default value. For instance '`tc -p /tmp foo.tig`' will first look for `prelude.tih` in `/tmp`.

Must import be functional?
: Yes. Read the previous item.

4.3.6 TC-1 Improvements

Possible improvements include:

4.4 TC-2, Building the Abstract Syntax Tree

2020-TC-2 submission for Ing1 students is Sunday, February 25th 2018 at 11:42.

This section has been updated for EPITA-2020 on 2016-01-27.

At the end of this stage, the compiler can build abstract syntax trees of Tiger programs and pretty-print them. The parser is now a GLR parser and equipped with error recovery. The memory is properly deallocated on demand.

The code must follow our coding style and be documented, see Section 2.4 [Coding Style], page 32, and Section 5.16 [Doxygen], page 255.

Relevant lecture notes include `dev-tools.pdf`[9], `ast.pdf`[10].

4.4.1 TC-2 Goals

Things to learn during this stage that you should remember:

Strict Coding Style
: Following a strict coding style is an essential part of collaborative work. Understanding the rationales behind rules is even better. See Section 2.4 [Coding Style], page 32.

Memory Leak Trackers
: Using tools such as Valgrind (see Section 5.8 [Valgrind], page 250) to track memory leaks.

Understanding the use of a GLR Parser
: The parser should now use all the possibilities of a GLR parser.

Error recovery with Bison
: Using the `error` token, and building usable ASTs in spite of lexical/syntax errors.

Using STL containers
: The AST uses `std::vector`, `misc::symbol` uses `std::set`.

Inheritance
: The AST hierarchy is typical example of a proper use of inheritance, together with. . .

Inclusion polymorphism
: An intense use of inclusion polymorphism for `accept`.

Use of constructors and destructors
: In particular using the destructors to reclaim memory bound to components.

`virtual` Dynamic and static bindings.

`misc::indent`
: `misc::indent` extends `std::ostream` with indentation features. Use it in the `PrettyPrinter` to pretty-print. Understanding how `misc::indent` works will be checked later, see Section 4.5.1 [TC-3 Goals], page 99.

The Composite design pattern
: The AST hierarchy is an implementation of the Composite pattern.

The Visitor design pattern
: The `PrettyPrinter` is an implementation of the Visitor pattern.

Writing good developer documentation (using Doxygen)
: The AST must be properly documented.

[9] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/dev-tools.pdf`.

[10] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/ast.pdf`.

4.4.2 TC-2 Samples

Here are a few samples of the expected features.

4.4.2.1 TC-2 Pretty-Printing Samples

The parser builds abstract syntax trees that can be output by a pretty-printing module:

```
/* Define a recursive function. */
let
  /* Calculate n!. */
  function fact (n : int) : int =
    if  n = 0
      then 1
      else n * fact (n - 1)
in
  fact (10)
end
```

File 4.8: `simple-fact.tig`

```
$ tc -XA simple-fact.tig
/* == Abstract Syntax Tree. == */

function _main() =
  (
    let
      function fact(n : int) : int =
        (if (n = 0)
          then 1
          else (n * fact((n - 1))))
    in
      fact(10)
    end;
    ()
  )
```

Example 4.10: *`tc -XA simple-fact.tig`*

The pretty-printed output must be *valid* and *equivalent*.

Valid means that any Tiger compiler must be able to parse with success your output. Pay attention to the banners such as '`== Abstract...`': you should use comments: '`/* == Abstract... */`'. Pay attention to special characters too.

```
print("\"\x45\x50ITA\"\n")
```

File 4.9: `string-escapes.tig`

```
$ tc -XA string-escapes.tig
/* == Abstract Syntax Tree. == */

function _main() =
  (
    print("\"EPITA\"\n");
    ()
```

```
  )
```

Example 4.11: *tc -XA string-escapes.tig*

Equivalent means that, except for syntactic sugar, the output and the input are equal. Syntactic sugar refers to '`&`', '`|`', unary '`-`', etc.

```
1 = 1 & 2 = 2
```

File 4.10: `1s-and-2s.tig`

```
$ tc -XA 1s-and-2s.tig
/* == Abstract Syntax Tree. == */

function _main() =
  (
    (if (1 = 1)
      then ((2 = 2) <> 0)
      else 0);
    ()
  )
```

Example 4.12: *tc -XA 1s-and-2s.tig*

```
$ tc -XA 1s-and-2s.tig >output.tig
```

Example 4.13: *tc -XA 1s-and-2s.tig >output.tig*

```
$ tc -XA output.tig
/* == Abstract Syntax Tree. == */

function _main() =
  (
    (if (1 = 1)
      then ((2 = 2) <> 0)
      else 0);
    ()
  )
```

Example 4.14: *tc -XA output.tig*

Beware that `for` loops are encoded using a `ast::VarDec`: do not display the '`var`':

```
for i := 0 to 100 do
   (print_int (i))
```

File 4.11: `for-loop.tig`

```
$ tc -XA for-loop.tig
/* == Abstract Syntax Tree. == */

function _main() =
  (
    (for i := 0 to 100 do
      print_int(i));
    ()
  )
```

Example 4.15: *tc -XA for-loop.tig*

Parentheses must not stack for free; you must even remove them as the following example demonstrates.

```
((((((((((0))))))))))
```

File 4.12: parens.tig

```
$ tc -XA parens.tig
/* == Abstract Syntax Tree. == */

function _main() =
  (
    0;
    ()
  )
```

Example 4.16: *tc -XA parens.tig*

This is not a pretty-printer trick: the ASTs of this program and that of '0' are exactly the same: a single `ast::IntExp`.

As a result, *anything output by 'tc -A' is equal to what 'tc -A | tc -XA -' displays!*

4.4.2.2 TC-2 Chunks

The type checking rules of Tiger, or rather its binding rules, justify the contrived parsing of declarations. This is why this section uses `-b`/`--bindings-compute`, implemented later (see Section 4.5 [TC-3], page 98).

In Tiger, to support recursive types and functions, continuous declarations of functions and continuous declarations of types are considered "simultaneously". For instance in the following program, `foo` and `bar` are visible in each other's scope, and therefore the following program is correct wrt type checking.

```
let function foo() : int = bar()
    function bar() : int = foo()
in
  0
end
```

File 4.13: foo-bar.tig

```
$ tc -b foo-bar.tig
```

Example 4.17: *tc -b foo-bar.tig*

In the following sample, because `bar` is not declared in the same bunch of declarations, it is not visible during the declaration of `foo`. The program is invalid.

```
let function foo() : int = bar()
    var stop := 0
    function bar() : int = foo()
in
  0
end
```

File 4.14: `foo-stop-bar.tig`

```
$ tc -b foo-stop-bar.tig
error foo-stop-bar.tig:1.28-32: undeclared function: bar
⇒4
```

Example 4.18: *tc -b foo-stop-bar.tig*

The same applies to types.

We shall name *chunk* a continuous series of type (or function) declaration.

A single name cannot be defined more than once in a chunk.

```
let function foo() : int = 0
    function bar() : int = 1
    function foo() : int = 2
    var stop := 0
    function bar() : int = 3
in
  0
end
```

File 4.15: `fbfsb.tig`

```
$ tc -b fbfsb.tig
error fbfsb.tig:3.5-28: redefinition: foo
error fbfsb.tig:1.5-28: first definition
⇒4
```

Example 4.19: *tc -b fbfsb.tig*

It behaves exactly as if chunks were part of embedded `let in end`, i.e., as if the previous program was syntactic sugar for the following one (in fact, in 2006-tc used to desugar it that way).

```
let
  function foo() : int = 0
  function bar() : int = 1
in
  let
     function foo() : int = 2
  in
    let
      var stop := 0
    in
      let
        function bar() : int = 3
      in
        0
      end
    end
  end
end
```

File 4.16: `fbfsb-desugared.tig`

Given the type checking rules for variables, whose definitions cannot be recursive, chunks of variable declarations are reduced to a single variable.

4.4.2.3 TC-2 Error Recovery

Your parser must be robust to (some) syntactic errors. Observe that on the following input several parse errors are reported, not merely the first one:

```
(
  1;
  (2, 3);
  (4, 5);
  6
)
```

File 4.17: `multiple-parse-errors.tig`

```
$ tc multiple-parse-errors.tig
error multiple-parse-errors.tig:3.5: syntax error, unexpected ",", ex-
pecting ;
error multiple-parse-errors.tig:4.5: syntax error, unexpected ",", ex-
pecting ;
⇒3
```

Example 4.20: *tc multiple-parse-errors.tig*

Of course, the exit status still reveals the parse error. Error recovery must not break the rest of the compiler.

```
$ tc -XA multiple-parse-errors.tig
error multiple-parse-errors.tig:3.5: syntax error, unexpected ",", ex-
pecting ;
error multiple-parse-errors.tig:4.5: syntax error, unexpected ",", ex-
pecting ;
/* == Abstract Syntax Tree. == */

function _main() =
  (
    (
      1;
      ();
      ();
      6
    );
    ()
  )
⇒3
```

Example 4.21: *tc -XA multiple-parse-errors.tig*

4.4.3 TC-2 Given Code

Code is provided through the 'tc-base' repository, using tag '2020-tc-base-2.0'.

For a description of the new modules, see Section 3.2.4 [lib/misc], page 56, and Section 3.2.8 [src/ast], page 59.

4.4.4 TC-2 Code to Write

What is to be done:

`src/parse/parsetiger.yy`

Build the AST
: Complete actions to instantiate AST nodes.

Support object-related syntax
: Supporting object constructs, an improvement suggested for TC-0 (see Section 4.2.5 [PTHL Improvements], page 79), is highly recommended.

Support metavariable constructs
: Augment your scanner and your parser to support the (reserved) keywords '`_decs`', '`_exp`', '`_lvalue`' and '`_namety`' and implement the corresponding grammar rules (see Section "Language Extensions" in *Tiger Compiler Reference Manual*). The semantic actions of these productions shall use the '`metavar`' function template to fetch the right AST subtree from the `parse::Tweast` object attached to the parsing context (`parse::TigerParser` instance).

Implement error recovery.
: There should be at least three uses of the token `error`. Read the Bison documentation about it.

Use `%printer`
: Extend the use of `%printer` to display non-terminals.

Use `%destructor`
: Use `%destructor` to reclaim the memory bound to semantic values thrown away during error recovery.

GLR
: Change your skeleton to `glr.cc`, use the `%glr-parser` directive. Thanks to GLR, conflicts (S/R and/or R/R) can be accepted. Use `%expect` and `%expect-rr` to specify their number. For information, we have no R/R conflicts, and two S/R: one related to the "big lvalue" issue, and the other to the implementation of the two `_cast` operators (see Section "Additional Syntactic Specifications" in *Tiger Compiler Reference Manual*).

Chunks
: In order to implement easily the type checking of declarations and to simplify following modules, adjust your grammar *to parse declarations by chunks*. The implementations of these chunks are in `ast::FunctionDecs`, `ast::MethodDecs`, `ast::VarDecs`, and `ast::TypeDecs`; they are implemented thanks to `ast::AnyDecs`). Note that an `ast::VarDecs` node appearing in a declaration list shall contain exactly one `ast::VarDec` object (see Section 4.4.2.2 [TC-2 Chunks], page 92); however, an `ast::VarDecs` used to implement a function's formal arguments may of course contain several `ast::VarDec` (one per formal).

`src/ast`
: Complete the abstract syntax tree module: no '`FIXME:`' should be left. Several files are missing in full. See `src/ast/README` for additional information on the missing classes.

`src/ast/default-visitor.hxx`
: Complete the `GenDefaultVisitor` class template. It is the basis for following visitors in the Tiger compiler.

`src/ast/object-visitor.hxx`
: Likewise, complete `GenObjectVisitor`. This class template is used to instantiate visitors factoring common code (default traversals of object-related nodes) and serves as a base class of `ast::PrettyPrinter` (and later `bind::Binder`).

`src/ast/pretty-printer.hh`
`src/ast/pretty-printer.cc`
: The `PrettyPrinter` class must be written entirely. It must use the `misc::xalloc` features to support indentation.

4.4.5 TC-2 FAQ

A `NameTy`, or a `symbol`
: At some places, you may use one or the other. Just ask yourself which is the most appropriate given the context. Appel is not always right.

Bison
: Be sure to read its dedicated section: Section 5.9 [Flex & Bison], page 251.

Memory leaks in the parser during error recovery
: To reclaim the memory during error recovery, use the `%destructor` directive:

```
%type <ast::Exp*> exp
%type <ast::Var*> lvalue
%destructor { delete $$; } <ast::Exp*> <ast::Var*> /* ... */;
```

Memory leaks in the standard containers
: See Section 5.8 [Valgrind], page 250, for a pointer to the explanation and solution.

How do I use `misc::error`
: See [misc/error], page 57, for a description of this component. In the case of the parse module, `TigerParser` aggregates the local error handler. From `scan_open`, for instance, your code should look like:

```
if(!yyin)
  error_ << misc::error::failure
         << program_name << ": cannot open `" << name << "': "
         << strerror(errno) << std::endl
         << &misc::error::exit;
```

`ast::fields_type` vs. `ast::VarDecs`
Record definition vs. Function declaration
: The grammar of the Tiger language (see Section "Syntactic Specifications" in *Tiger Compiler Reference Manual*) includes:

```
# Function, primitive and method declarations.
<dec> ::=
    "function"  <id> "(" <tyfields> ")" [ ":" <type-id> ] "=" <exp>
  | "primitive" <id> "(" <tyfields> ")" [ ":" <type-id> ]
<classfield> ::=
    "method"    <id> "(" <tyfields> ")" [ ":" <type-id> ] "=" <exp>

# Record type declaration.
<ty> ::= "{" <tyfields>  "}"
```

```
# List of ``id : type''.
<tyfields> ::= [ <id> ":" <type-id> { "," <id> ":" <type-id> } ]
```

This grammar snippet shows that we used `tyfields` several times, in two very *different* contexts: a list of formal arguments of a function, primitive or method; and a list of record fields. The fact that the syntax is similar in both cases is an "accident": it is by no means required by the language. A. Appel could have chosen to make them different, but what would have been the point then? It does make sense, sometimes, to make two different things look alike, that's a form of economy — a sane engineering principle.

If the concrete syntaxes were chosen to be identical, should it be the case for abstract too? We would say it depends: the inert data is definitely the same, but the behaviors (i.e., the handling in the various visitors) are very different. So if your language features "inert data", say C or ML, then keeping the same abstract syntax makes sense; if your language features "active data" — let's call this... objects — then *it is a mistake*. Sadly enough, the first edition of Red Tiger book made this mistake, and we also did it for years.

The second edition of the Tiger in Java introduces a dedicated abstract syntax for formal arguments; we made a different choice: there is little difference between formal arguments and local variables, so we use a `VarDecs`, which fits nicely with the semantics of chunks.

Regarding the abstract syntax of a record type declaration, we use a list of `Field`s (aka `fields_type`).

Of course this means that you will have to *duplicate* your parsing of the `tyfields` non-terminal in your parser.

`ast::DefaultVisitor` and `ast::NonObjectVisitor`

The existence of `ast::NonObjectVisitor` is the result of a reasonable compromise between (relative) safety and complexity.

The problem is: as object-aware programs are to be desugared into object-free ones, (a part of) our front-end infrastructure must support two kinds of traversals:

- Traversals dealing with AST with objects: `ast::PrettyPrinter`, `object::Binder`, `object::TypeChecker`, `object::DesugarVisitor`.
- Traversals dealing with AST without objects `bind::Binder`, `type::TypeChecker`, and all other AST visitors.

The first category has visit methods for all type of nodes of our (object-oriented) AST, so they raise no issue. On the other hand, the second category of visitors knows nothing about objects, and should either be unable to visit AST w/ objects (static solution) or raise an error if they encounter objects (dynamic solution).

Which led us to several solutions:

1. Consider that we have two kinds of visitors, and thus two *hierarchies* of visitors. Two hierarchies might confuse the students, and make the maintenance harder. Hooks in the AST nodes (`accept` methods) must be duplicated, too.
2. Have a single hierarchy of visitors, but equip all concrete visitors traversing ASTs w/o objects with methods visiting object-related node aborting at run time.
3. Likewise, but factor the aborting methods in a single place, namely `ast::NonObjectVisitor`. That is the solution we chose.

Solutions 2 and 3 let us provide a default visitor for ASTs without objects, but it's harder to have a meaningful default visitor for ASTs *with* objects: indeed, concrete visitors on ASTs w/ objects inherit from their non-object counterparts, where methods visiting object nodes are *already* defined! (Though they abort at run time.)

We have found that having two visitors (`ast::DefaultVisitor` and `ast::NonObjectVisitor`) to solve this problem was more elegant, rather than merging both of them in `ast::DefaultVisitor`. The pros are that `ast::DefaultVisitor` remains a default visitor; the cons are that this visitor is now abstract, since object-related nodes have no visit implementation. Therefore, we also introduced an `ast::ObjectVisitor` performing default visits of the remaining node types; the combined inheritance of both `ast::DefaultVisitor` and `ast::ObjectVisitor` provides a complete default visitor.

4.4.6 TC-2 Improvements

Possible improvements include:

Desugar Boolean operators and unary minus in concrete syntax
: In the original version of the exercise, the `|` and `&` operators and the unary minus operator are *desugared* in abstract syntax (i.e., using explicit instantiations of AST nodes). Using `TigerInput`, you can desugar using Tiger's concrete syntax instead. This second solution is advised.

Introduce an `Error` class
: When syntactic errors are caught, a valid AST must be built anyway, hence a critical question is: what value should be given to the missing bits? If your error recovery is not compatible with what the user meant, you are likely to create artificial type errors with your invented value.

  While this behavior is compliant with the assignment, you may improve this by introducing an `Error` class (one?), which will never trigger type checking errors.

Using Generic Visitors
: Andrei Alexandrescu has done a very interesting work on generic implementation of Visitors, see [Modern C++ Design], page 243. It does require advanced C++ skills, since it is based on type lists, which requires heavy use of templates.

Using Visitor Combinators
: Going even further that Andrei Alexandrescu, Nicolas Tisserand proposes an implementation of Visitor combinators, see [Generic Visitors in C++], page 242.

4.5 TC-3, Bindings

2020-TC-3 submission for Ing1 students is Sunday, March 11th 2018 at 11:42.
Section 4.6 [TC-R], page 106, is part of the mandatory assignment of 2020-TC-3.

This section has been updated for EPITA-2020 on 2016-01-27.

At the end of this stage, the compiler must be able to compute and display the bindings. These features are triggered by the options `-b`/`--bindings-compute`, `--object-bindings-compute` and `-B`/`--bindings-display`.

Relevant lecture notes include: `names.pdf`[11].

[11] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/names.pdf`.

4.5.1 TC-3 Goals

Things to learn during this stage that you should remember:

The Command design pattern
: The `Task` module is based on the Command design pattern.

Writing a Container Class Template
: Class template are most useful to implement containers such as `misc::scoped_map`.

Using methods from parents classes
: `super_type` and qualified method invocation to factor common code.

Traits
: Traits are a useful technique that allows to write (compile time) functions ranging over types. See Section A.1 [Glossary], page 257. The implementation of both hierarchies of visitors (const or not) relies on traits. You are expected to understand the code.

Streams' internal extensible arrays
: C++ streams allows users to dynamically store information within themselves thanks to `std::ios::xalloc`, `std::stream::iword`, and `std::stream::pword` (see `ios_base` documentation by Cplusplus Ressources[12]). Indented output can use it directly in `operator<<`, see `lib/misc/indent.*` and `lib/misc/test-indent.cc`. More generally, if you have to resort to using `print` because you need additional arguments than the sole stream, consider using this feature instead.

  Use this feature so that the `PrettyPrinter` can be told *from the* `std::ostream` whether escapes and bindings should be displayed.

4.5.2 TC-3 Samples

Binding is relating a name use to its definition.

```
let
  var me := 0
in
  me
end
```

File 4.18: `me.tig`

```
$ tc -XbBA me.tig
/* == Abstract Syntax Tree. == */

function _main /* 0x563048f78b00 */() =
  (
    let
      var me /* 0x563048f7b5b0 */ := 0
    in
      me /* 0x563048f7b5b0 */
    end;
    ()
  )
```

Example 4.22: *`tc -XbBA me.tig`*

[12] `http://www.cplusplus.com/ref/iostream/ios_base/`.

This is harder when there are several occurrences of the same name. Note that primitive types are accepted, but have no pre-declaration, contrary to primitive functions.

```
let
  var me := 0
  function id(me : int) : int = me
in
  me
end
```

File 4.19: `meme.tig`

```
$ tc -XbBA meme.tig
/* == Abstract Syntax Tree. == */

function _main /* 0x5566cd725b00 */() =
  (
    let
      var me /* 0x5566cd7285b0 */ := 0
      function id /* 0x5566cd7272d0 */(me /* 0x5566cd7261e0 */ : int /* 0 */) : int
        me /* 0x5566cd7261e0 */
    in
      me /* 0x5566cd7285b0 */
    end;
    ()
  )
```

Example 4.23: *tc -XbBA meme.tig*

TC-3 is in charge of incorrect uses of the names, such as undefined names,

```
me
```

File 4.20: `nome.tig`

```
$ tc -bBA nome.tig
error nome.tig:1.1-2: undeclared variable: me
⇒4
```

Example 4.24: *tc -bBA nome.tig*

or redefined names.

```
let
  type me = {}
  type me = {}
  function twice(a: int, a: int) : int = a + a
in
  me {} = me {}
end
```

File 4.21: `tome.tig`

```
$ tc -bBA tome.tig
error tome.tig:3.3-14: redefinition: me
error tome.tig:2.3-14: first definition
error tome.tig:4.25-31: redefinition: a
```

```
error tome.tig:4.18-23: first definition
⇒4
```

Example 4.25: *tc -bBA tome.tig*

In addition to binding names, `--bindings-compute` is also in charge of binding the `break` to their corresponding loop construct.

```
let var x := 0 in
  while 1 do
  (
    for i := 0 to 10 do
    (
      x := x + i;
      if x >= 42 then
        break
    );
    if x >= 51 then
      break
  )
end
```

File 4.22: `breaks-in-embedded-loops.tig`

```
$ tc -XbBA breaks-in-embedded-loops.tig
/* == Abstract Syntax Tree. == */

function _main /* 0x55a197e92b00 */() =
  (
    let
      var x /* 0x55a197e955e0 */ := 0
    in
      (while /* 0x55a197e960a0 */ 1 do
        (
          (for /* 0x55a197e94ae0 */ i /* 0x55a197e94280 */ := 0 to 10 do
            (
              (x /* 0x55a197e955e0 */ := (x /* 0x55a197e955e0 */ + i /* 0x55a197e942
              (if (x /* 0x55a197e955e0 */ >= 42)
                then break /* 0x55a197e94ae0 */
                else ())
            ));
          (if (x /* 0x55a197e955e0 */ >= 51)
            then break /* 0x55a197e960a0 */
            else ())
        ))
    end;
    ()
  )
```

Example 4.26: *tc -XbBA breaks-in-embedded-loops.tig*

```
break
```

File 4.23: `break.tig`

```
$ tc -b break.tig
error break.tig:1.1-5: 'break' outside any loop
⇒4
```

Example 4.27: *tc -b break.tig*

Embedded loops show that there is scoping for `break`s. Beware that there are places, apparently inside loops, where `break`s make no sense too.

Although it is a matter of definitions and uses of names, record members are not bound here, because it is easier to implement during type checking. Likewise, duplicate fields are to be reported during type checking.

```
let
  type     box = { value : int }
  type     dup = { value : int, value : string }
  var      box := box { value = 51 }
in
  box.head
end
```

File 4.24: `box.tig`

```
$ tc -XbBA box.tig
/* == Abstract Syntax Tree. == */

function _main /* 0x55c8f9b45ed0 */() =
  (
    let
      type box /* 0x55c8f9b44db0 */ = { value : int /* 0 */ }
      type dup /* 0x55c8f9b440a0 */ = {
        value : int /* 0 */,
        value : string /* 0 */
      }
      var box /* 0x55c8f9b44750 */ := box /* 0x55c8f9b44db0 */ { value = 51 }
    in
      box /* 0x55c8f9b44750 */.head
    end;
    ()
  )
```

Example 4.28: *tc -XbBA box.tig*

```
$ tc -T box.tig
error box.tig:3.33-46: identifier multiply defined: value
error box.tig:6.3-10: invalid field: head
⇒5
```

Example 4.29: *tc -T box.tig*

But apart from these field-specific checks delayed at TC-4, TC-3 should report other name-related errors. In particular, a field with an invalid type name *is* a binding error (related to the field's type, not the field itself), to be reported at TC-3.

```
let
  type rec = { a : unknown }
in
  rec { a = 42 }
end
```

File 4.25: `unknown-field-type.tig`

```
$ tc -XbBA unknown-field-type.tig
error unknown-field-type.tig:2.20-26: undeclared type: unknown
⇒4
```

Example 4.30: *`tc -XbBA unknown-field-type.tig`*

Likewise, class members (both attributes and methods) are not to be bound at Section 4.5 [TC-3], page 98, but at the type-checking stage (see Section 4.8 [TC-4], page 109). Therefore, no bindings are to be displayed in regards to object at Section 4.5 [TC-3], page 98.

```
let
  type C = class {}
  var c := new C
in
  c.missing_method();
  c.missing_attribute
end
```

File 4.26: `bad-member-bindings.tig`

```
$ tc -X --object-bindings-compute -BA bad-member-bindings.tig
/* == Abstract Syntax Tree. == */

function _main /* 0x55914f1e3b00 */() =
  (
    let
      type C /* 0x55914f1e4610 */ =
      class extends Object /* 0 */
      {
      }
      var c /* 0x55914f1e3bd0 */ := new C /* 0x55914f1e4610 */
    in
      (
        c /* 0x55914f1e3bd0 */.missing_method();
        c /* 0x55914f1e3bd0 */.missing_attribute
      )
    end;
    ()
  )
```

Example 4.31: *`tc -X --object-bindings-compute -BA bad-member-bindings.tig`*

```
$ tc --object-types-compute bad-member-bindings.tig
error bad-member-bindings.tig:5.3-20: unknown method: missing_method
error bad-member-bindings.tig:6.3-21: unknown attribute: missing_attribute
⇒5
```

Example 4.32: `tc --object-types-compute bad-member-bindings.tig`

Concerning the super class type, the compiler should just check that this type exists in the environment at Section 4.5 [TC-3], page 98. Other checks are left to TC-4 (see Section 4.8.2 [TC-4 Samples], page 110).

```
let
  /* Super class doesn't exist.  */
  class Z extends Ghost {}
in
end
```

File 4.27: `missing-super-class.tig`

```
$ tc -X --object-bindings-compute -BA missing-super-class.tig
error missing-super-class.tig:3.19-23: undeclared type: Ghost
⇒4
```

Example 4.33: `tc -X --object-bindings-compute -BA missing-super-class.tig`

4.5.3 TC-3 Given Code

Code is provided through the '`tc-base`' repository, using tag '`2020-tc-base-3.0`'. For a description of the new module, see Section 3.2.9 [src/bind], page 61.

4.5.4 TC-3 Code to Write

`misc::scoped_map<Key, Data>`
: Complete the class template `misc::scoped_map` in `lib/misc/scoped-map.hh` and `lib/misc/scoped-map.hxx`. See Section 3.2.4 [lib/misc], page 56, See [scoped_map], page 58, for more details.

Equip `ast`
: Augment constructs "using" an identifier, such as `CallExp`, with `def_`, `def_get`, and `def_set` to be able to set a reference to their definition, here a `FunctionDec`.

`ast::PrettyPrinter`
: Implement `--bindings-display` support in the `PrettyPrinter`. Be sure to display the addresses exactly as displayed in this document: immediately after the identifier.

Complete the `bind::Binder`
: Most of the assignment is here...

Complete the `object::Binder`
: ...and here. `object::Binder` inherits from `bind::Binder` so as to factor common parts.

Implement renaming to unique identifiers.
: TC-R is a mandatory assignment. Once TC-3 completed, implementing TC-R is straightforward, see Section 4.6 [TC-R], page 106. Note that `--rename` is helpful to write a test suite for TC-3.

Complete auxiliary code
: Write the tasks, `libbind.*` etc.

4.5.5 TC-3 FAQ

Ambiguous resolution of `operator<<` for `ast::VarDec`
: Starting from TC-3, `ast::VarDec` inherits both from `ast::VarDec` and `ast::Escapable`. Printing an `ast::VarDec` using `operator<<` can be troublesome as this operator may be overloaded for both `ast::VarDec`'s base classes, but not for `ast::VarDec` itself, resulting in an ambiguous overload resolution. The simplest way to get rid of this ambiguity is to convert the `ast::VarDec` object to the type of one of its base classes ("upcast") before printing it, either by creating a alias or (more simply) by using the `static_cast` operator:

```
const ast::VarDec& vardec = ...

// Printing VARDEC as an ast::Dec using an intermediate
// variable (alias).
const ast::Dec& dec = vardec;
ostr << dec;

// Printing VARDEC as an ast::Escapable using an
// on-the-fly conversion.
ostr << static_cast<const ast::Escapable&>(vardec);
```

What is the purpose of the '`bound`' task?
: The computation of name bindings can be carried out in different ways, depending on the input language: Tiger without object constructs ("Panther"), Tiger with object constructs and Tiger with support for function overloading. These different flavors of the binding computation are performed by options `--bindings-compute`, `--object-bindings-compute` and `--overfun-bindings-compute` respectively (see Section "Invoking tc" in *Tiger Compiler Reference Manual*).

  However, some subsequent task may later just require that an AST is annotated with bindings ("bound") regardless of the technique used to compute these bindings. The purpose of the '`bound`' task is to address this need: ensuring that one of the bindings task has been executed. This task can be considered as a disjunction (logical "or") of the '`bindings-compute`', '`object-bindings-compute`' and '`overfun-bindings-compute`' tasks, the first one being the default binding strategy.

4.5.6 TC-3 Improvements

Possible improvements include:

Factoring the binding interface
: In the `ast` module, several classes need to be changed to be "bindable", i.e., to have new data and function members to set, store, and retrieve their associated definition. Instead of changing several classes in a very similar fashion, introduce a `Bindable` template class and derive from its instantiation.

Hash tables
: How about using true hash tables (aka "unordered associative containers" in Boost parlance) instead of trees? You might also want to try Google's Sparse Hash Tables[13].

[13] `http://code.google.com/p/sparsehash/`.

Escaping Variables Computation
: Once TC-3 completed, you might consider the TC-E option now, see Section 4.7 [TC-E], page 107. It takes about 100 lines to make it.

4.6 TC-R, Unique Identifiers

2020-TC-3 submission for Ing1 students is Sunday, February 25th 2018 at 11:42.
Section 4.6 [TC-R], page 106, is part of the mandatory assignment of 2020-TC-3.

This section has been updated for EPITA-2020 on 2016-01-27.

At the end of this stage, when given the option `--rename`, the compiler produces an AST such that no identifier is defined twice.

Relevant lecture notes include: `names.pdf`[14].

4.6.1 TC-R Samples

Note that the transformation does not apply to field names.

```
let
  type a = { a: int }
  function a(a: a): a = a{ a = a + a }
  var a : a := a(1, 2)
in
  a.a
end
```

File 4.28: `as.tig`

```
$ tc -X --rename -A as.tig
/* == Abstract Syntax Tree. == */

function _main() =
  (
    let
      type a_0 = { a : int }
      function a_2(a_1 : a_0) : a_0 =
        a_0 { a = (a_1 + a_1) }
      var a_3 : a_0 := a_2(1, 2)
    in
      a_3.a
    end;
    ()
  )
```

Example 4.34: *`tc -X --rename -A as.tig`*

4.6.2 TC-R Given Code

No additional code is provided, see Section 4.5.3 [TC-3 Given Code], page 104.

4.6.3 TC-R Code to Write

`bind::Renamer`
: Write it from scratch.

[14] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/names.pdf`.

Complete auxiliary code
: Write the tasks, `libbind.*` etc.

4.6.4 TC-R FAQ

Should I rename primitives (builtins) or `_main`?
: No, you shall not rename them; you have to keep the interface of the Tiger runtime. Likewise for `_main`.

4.7 TC-E, Computing the Escaping Variables

2020-TC-E submission is Sunday, March 25th 2018 at 11:42.
Section 4.7 [TC-E], page 107, is part of the mandatory assignment of 2020-TC-4.

This section has been updated for EPITA-2020 on 2015-01-27.

At the end of this stage, the compiler must be able to compute and display the escaping variables. These features are triggered by the options `--escapes-compute`/`-e` and `--escapes-display`/`-E`.

Relevant lecture notes include: `names.pdf`[15] and `intermediate.pdf`[16].

4.7.1 TC-E Goals

Things to learn during this stage that you should remember:

Understanding escaping variables
: In TC-E, we consider the case of non-local variables, i.e., variables that are defined in a function, but used (at least once) in *another* function, nested in the first one. This possibility for an inner function to use variables declared in outer functions is called *block structure*. Because such variables are used outside of their host function, they are qualified as "escaping". This information will be necessary during the translation to the intermediate representation (see Section 4.14 [TC-5], page 132) when variables (named temporaries a that stage) are assigned a location (in the stack or in a register). Escaping variables shall indeed be stored in memory, so that non-local uses of such variables can actually have a means to access them.

Writing a Visitor from scratch
: The `escapes::EscapesVisitor` provided is almost empty. A goal of TC-E is to write a complete visitor (though a small one). Do not forget to use `ast::DefaultVisitor` to factor as much code as possible.

4.7.2 TC-E Samples

This example demonstrates the computation and display of escaping variables (and formal arguments). By default, all the variables must be considered as escaping, since it is safe to put a non escaping variable onto the stack, while the converse is unsafe.

```
let
  var one := 1
  var two := 2
  function incr(x: int) : int = x + one
in
  incr(two)
end
```

[15] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/names.pdf`.
[16] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/intermediate.pdf`.

File 4.29: `variable-escapes.tig`

```
$ tc -XEAeEA variable-escapes.tig
/* == Abstract Syntax Tree. == */

function _main() =
  (
    let
      var /* escaping */ one := 1
      var /* escaping */ two := 2
      function incr(/* escaping */ x : int) : int =
        (x + one)
    in
      incr(two)
    end;
    ()
  )
/* == Abstract Syntax Tree. == */

function _main() =
  (
    let
      var /* escaping */ one := 1
      var two := 2
      function incr(x : int) : int =
        (x + one)
    in
      incr(two)
    end;
    ()
  )
```

Example 4.35: `tc -XEAeEA variable-escapes.tig`

Compute the escapes after binding, so that the AST is known to be sane enough (type checking is irrelevant): the `EscapeVisitor` should not bother with undeclared entities.

```
undeclared
```

File 4.30: `undefined-variable.tig`

```
$ tc -e undefined-variable.tig
error undefined-variable.tig:1.1-10: undeclared variable: undeclared
⇒4
```

Example 4.36: `tc -e undefined-variable.tig`

Run your compiler on `merge.tig` and to study its output. There is a number of silly mistakes that people usually make on TC-E: they are all easy to defeat when you do have a reasonable test suite, and once you understood that *torturing your project is a good thing to do.*

4.7.3 TC-E Given Code

No additional code is provided, see Section 4.5.3 [TC-3 Given Code], page 104.

4.7.4 TC-E Code to Write

See Section 3.2.8 [src/ast], page 59, and Section 3.2.10 [src/escapes], page 61.

`ast::PrettyPrinter`
: Implement `--escapes-display` support in the `PrettyPrinter`. Follow strictly the output format, since we parse your output to check it. Display the '`/* escaping */`' flag where needed, and *only* where needed: each definition of an escaping variable/formal is *preceded* by the comment '`/* escaping */`'. Do not display meaningless flags due to implementation details. How this pretty-printing is implemented is left to you, but factor common code.

`escapes::EscapesVisitor`
: Write the class `escapes::EscapesVisitor` in `src/escapes/escapes-visitor.hh` and `src/escapes/escapes-visitor.cc`.

Introduce `ast::Escapable`
: Ensure `ast::VarDec` inherits from `ast::Escapable`. See [Escapable], page 60.

4.7.5 TC-E FAQ

4.7.6 TC-E Improvements

Possible improvements include:

4.8 TC-4, Type Checking

2020-TC-4 submission is Sunday, May 25th 2018 at 11:42.
Section 4.7 [TC-E], page 107, is part of the mandatory assignment of 2020-TC-4.

This section has been updated for EPITA-2020 on 2016-01-27.

At the end of this stage, the compiler type checks Tiger programs, and annotates the AST. Clear error messages are required.

Relevant lecture notes include `names.pdf`[17], `type-checking.pdf`[18].

4.8.1 TC-4 Goals

Things to learn during this stage that you should remember:

Function template and member function templates
: Functions template are quite convenient to factor code that looks alike but differs by the nature of its arguments. Member function templates are used to factor error handling the `TypeChecker`.

Virtual member function templates
: You will be asked why there can be no such thing in C++.

Template specialization
: Although quite different in nature, types and functions are processed in a similar fashion in a Tiger compiler: first one needs to visit the headers (to introduce the names in the scope, and to check that names are only defined once), and then to visit the bodies (to bind the names to actual values). We use templates and template specialization to factor this. See also the Template Method.

[17] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/names.pdf`.
[18] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/type-checking.pdf`.

The Template Method design pattern
: The Template Method allows to factor a generic algorithm, the steps of which are specific. This is what we use to type check function and type declarations. Do not confuse Template Method with member function template, the order matters. Remember that in English the noun is usually last, preceded by qualifier.

Type-checking
: What it is, how to implement it.

Stack unwinding
: What it means, and when the C++ standard requires it from the compiler.

4.8.2 TC-4 Samples

Type checking is optional, invoked by `--types-compute`. As for the computation of bindings, this option only handles programs with no object construct. To perform the type-checking of programs with objects, use `--object-types-compute`.

Implementing overloaded functions in Tiger is an option, which requires the implementation of a different type checker, triggered by `--overfun-types-compute` (see Section 4.12 [TC-A], page 123). The option `--typed`/`-T` makes sure one of them was run.

```
1 + "2"
```

File 4.31: `int-plus-string.tig`

```
$ tc int-plus-string.tig
```

Example 4.37: *`tc int-plus-string.tig`*

```
$ tc -T int-plus-string.tig
error int-plus-string.tig:1.5-7: type mismatch
error   right operand type: string
error   expected type: int
⇒5
```

Example 4.38: *`tc -T int-plus-string.tig`*

The type checker shall ensure loop index variables are read-only.

```
/* error: index variable erroneously assigned to.  */
for i := 10 to 1 do
  i := i - 1
```

File 4.32: `assign-loop-var.tig`

```
$ tc -T assign-loop-var.tig
error assign-loop-var.tig:3.3-12: variable is read only
⇒5
```

Example 4.39: *`tc -T assign-loop-var.tig`*

When there are several type errors, it is admitted that some remain hidden by others.

```
unknown_function(unknown_variable)
```

File 4.33: `unknowns.tig`

```
$ tc -T unknowns.tig
```

```
error unknowns.tig:1.1-34: undeclared function: unknown_function
⇒4
```

Example 4.40: *tc -T unknowns.tig*

Be sure to check the type of all the constructs.

```
if 1 then 2
```

File 4.34: bad-if.tig

```
$ tc -T bad-if.tig
error bad-if.tig:1.1-11: type mismatch
error   then clause type: int
error   else clause type: void
⇒5
```

Example 4.41: *tc -T bad-if.tig*

Be aware that type and function declarations are recursive by chunks. For instance:

```
let
  type one = { hd : int, tail : two }
  type two = { hd : int, tail : one }
  function one(hd : int, tail : two) : one
     = one { hd = hd, tail = tail }
  function two(hd : int, tail : one) : two
     = two { hd = hd, tail = tail }
  var one := one(11, two(22, nil))
in
  print_int(one.tail.hd); print("\n")
end
```

File 4.35: mutuals.tig

```
$ tc -T mutuals.tig
```

Example 4.42: *tc -T mutuals.tig*

In case you are interested, the result is:

```
$ tc -H mutuals.tig >mutuals.hir
```

Example 4.43: *tc -H mutuals.tig >mutuals.hir*

```
$ havm mutuals.hir
22
```

Example 4.44: *havm mutuals.hir*

The type-checker must catch erroneous inheritance relations.

```
let
  /* Mutually recursive inheritance.  */
  type A = class extends A {}

  /* Mutually recursive inheritance.  */
  type B = class extends C {}
  type C = class extends B {}
```

```
  /* Class inherits from a non-class type.  */
  type E = class extends int {}
in
end
```

File 4.36: `bad-super-type.tig`

```
$ tc --object-types-compute bad-super-type.tig
error bad-super-type.tig:3.12-29: recursive inheritance: A
error bad-super-type.tig:6.12-29: recursive inheritance: C
error bad-super-type.tig:10.26-28: class type expected, got: int
⇒5
```

Example 4.45: *`tc --object-types-compute bad-super-type.tig`*

Handle the type-checking of `TypeDecs` with care in `object::TypeChecker`: they are processed in three steps, while other declarations use a two-step visit. The `object::TypeChecker` visitor proceeds as follows when it encounters a `TypeDecs`:

1. Visit the headers of all types in the block.
2. Visit the bodies of all types in the block, but ignore members for each type being a class.
3. For each type of the block being a class, visit its members.

This three-pass visit allows class members to make forward references to other types defined in the same block of types, for instance, instantiate a class `B` from a class `A` (defined in the same block), even if `B` is defined *after* `A`.

```
let
  /* A block of types.  */
  class A
  {
    /* Valid forward reference to B, defined in the same block
       as the class enclosing this member.  */
    var b := new B
  }
  type t = int
  class B
  {
  }
in
end
```

File 4.37: `forward-reference-to-class.tig`

```
$ tc --object-types-compute forward-reference-to-class.tig
```

Example 4.46: *`tc --object-types-compute forward-reference-to-class.tig`*

(See `object::TypeChecker::operator()(ast::TypeDecs&)` for more details.

4.8.3 TC-4 Given Code

Some code is provided through the '`tc-base`' repository, using tag '`2020-tc-base-4.0`'. For a description of the new module, see Section 3.2.11 [src/type], page 61.

4.8.4 TC-4 Code to Write

What is to be done.

`ast::Typable`
`ast::TypeConstructor`
: Because many AST nodes will be annotated with their type, the feature is factored by these two classes. See [Typable], page 60, and [TypeConstructor], page 60, for details.

`ast::Exp`, `ast::Dec`, `ast::Ty`
: These are typable.

`ast::FunctionDec`, `ast::TypeDec`, `ast::Ty`
: These build types.

`src/type/type.*`,
`src/type/array.*`,
`src/type/builtin-types.*`,
`src/type/class.*`,
`src/type/function.*`,
`src/type/method.*`,
`src/type/named.*`,
`src/type/nil.*`,
`src/type/record.*`
: Implement the Singletons `type::String`, `type::Int`, and `type::Void`. Using templates would be particularly appreciated to factor the code between the three singleton classes, see Section 4.8.5 [TC-4 Options], page 114.

  The remaining classes are incomplete.

  Pay extra attention to `type::operator==(const Type& a, const Type& b)` and `type::Type::compatible_with`.

`type::TypeChecker`
`object::TypeChecker`
: Of course this is the most tricky part. We hope there are enough comments in there so that you understand what is to be done. Please, post your questions and help us improve it.

  It is also the `type::TypeChecker`'s job to set the `record_type` in the `type::Nil` class. `record_type` is holding some information about the `type::Record` type associated to the `type::Nil` type. We choose to handle the `record_type` only when no error occured in the type-checking process.

`type::GenVisitor`
`type::GenDefaultVisitor`
: `type::Type`s are visitable. You must implement the default visitor class template, which walks through the tree of types doing nothing. It's used as a base class for the type visitors.

`type::PrettyPrinter`
: In order to output nice error messages, the types need to be printed. You must implement a visitor that prints the types, similar to `ast::PrettyPrinter`.

Computing the Escaping Variables
: The implementation of Section 4.7 [TC-E], page 107, suggested at Section 4.5 [TC-3], page 98, becomes a mandatory assignment at Section 4.8 [TC-4], page 109.

4.8.5 TC-4 Options

These are features that you might want to implement in addition to the core features.

`type::Error`
: One problem is that type error recovery can generate false errors. For instance our compiler usually considers that the type for incorrect constructs is `Int`, which can create cascades of errors:

```
"666" = if 000 then 333 else "666"
```

File 4.38: `is_devil.tig`

```
$ tc -T is_devil.tig
error is_devil.tig:1.9-34: type mismatch
error   then clause type: int
error   else clause type: string
error is_devil.tig:1.1-34: type mismatch
error   left operand type: string
error   right operand type: int
⇒5
```

Example 4.47: *tc -T is_devil.tig*

One means to avoid this issue consists in introducing a new type, `type::Error`, that the type checker would never complain about. This can be a nice complement to `ast::Error`.

Various Desugaring
: See Section 4.9 [TC-D], page 116, for more details. This is quite an easy option, and a very interesting one. Note that implementing desugaring makes TC-5 easier.

Bounds Checking
: If you felt TC-D was easy, then implementing bounds checking should be easy too. See Section 4.11 [TC-B], page 119.

Overloaded Tiger
: See Section 4.12 [TC-A], page 123, for a description of this ambitious option.

Renaming object-oriented constructs
: Like TC-R, this task consists in writing a visitor renaming AST nodes holding names (either defined or used), this time with support for object-oriented constructs (option `--object-rename`). This visitor, `object::Renamer`, shall also update named types (`type::Named`) and collect the names of all (renamed) classes. This option is essentially a preliminary step of TC-O (see the next item).

Desugaring Tiger to Panther
: If your compiler is complete w.r.t. object constructs (in particular, the type-checking and the renaming of objects is a requirement), then you can implement this very ambitious option, whose goal is to convert a Tiger program with object constructs into a program with none of them (i.e., in the subset of Tiger called *Panther*). This work consists in completing the `object::DesugarVisitor` and implementing the `--object-desugar` option. See Section 4.13 [TC-O], page 126.

4.8.6 TC-4 FAQ

Stupid Types

One can legitimately wonder whether the following program is correct:

```
let type weirdo = array of weirdo
in
  print("I'm a creep.\n")
end
```

the answer is "yes", as nothing prevents this in the Tiger specifications. This type is not usable though.

Is `type::Field` useful?

Using `std::pair` in `type::Record` is probably enough, and simpler.

Is `nil` compatible with objects?

For instance, is the following example valid?

```
var a : Object := nil
```

The answer is yes: `nil` is both compatible with records *and* objects.

Can one redefine the built-in class `Object`?

Yes, if the rules of the Tiger Compiler Reference Manual are honored, notably:

- Every class has a super class, defaulting to the built-in class `Object` (syntactic sugar of `class` without an `extends` clause).
- Recursive inheritance (within the same block of types) is forbidden.

For example,

```
let class Object {} in end
```

is invalid, since it is similar to

```
let class Object extends Object {} in end
```

and recursive inheritance is invalid.

One can try and introduce a `Dummy` type as a workaround

```
let
  class Dummy {}
  class Object extends Dummy {}
in
end
```

but this is just postponing the problem, since the code above is the same as the following:

```
let
  class Dummy  extends Object {}
  class Object extends Dummy  {}
in
end
```

where there is still a recursive inheritance.

The one solution is to define our `Dummy` type beforehand (i.e., in its own block of type declarations), then to redefine `Object`.

```
/* Valid.  */
let
  class Dummy {}
in
  let
```

```
    class Object extends Dummy {}
  in
  end
end
```

Take care: this new `Object` type is *different* from the built-in one. The code below gives an example of an invalid mix of these two types.

```
let
  class Dummy {}
  function get_builtin_object() : Object = new Object /* builtin */
in
  let
    class Object extends Dummy {} /* custom */

    /* Invalid assignment, since an instance of the builtin Object
       is *not* an instance of the custom Object.  */
    var o : Object /* custom */ := get_builtin_object() /* builtin */
  in
  end
end
```

4.8.7 TC-4 Improvements

Possible improvements include:

A Singleton template
: Implementations of the Singleton design pattern are frequently needed; the `type` module alone requires three instances! Therefore a template to generate such singletons is desirable. There are two ways to address this issue: tailored to `type` (directly in `src/type/builtin-types.*`), or in a completely generic way (in `lib/misc/singleton.*`). See [Modern C++ Design], page 243, for a topnotch implementation.

A more verbose type display
: When reporting a type, one must be careful with recursive definitions that could produce never ending outputs. The suggested simple implementation ensure this by limiting the `Named`-depth (i.e., the number of `Named` objects traversed) to one. Another, nicer possibility, would be to limit the expansion to once *per* `Named`.

A Graphical User Interface
: `tcsh` is up and running. You might want to use it to implement a GUI using Python's Tkinter[19].

4.9 TC-D, Removing the syntactic sugar from the Abstract Syntax Tree

TC-D is an optional assignment.

This section has been updated for EPITA-2009 on 2007-04-26.

At the end of this stage, the compiler must be able to remove syntactic sugar from a type-checked AST. These features are triggered by the options `--desugar` and `--overfun-desugar`.

[19] `http://docs.python.org/2/library/tkinter.html`.

4.9.1 TC-D Samples

String comparisons can be translated to an equivalent AST using function calls, before the translation to HIR.

```
"foo" = "bar"
```

File 4.39: `string-equality.tig`

```
$ tc --desugar-string-cmp --desugar -A string-equality.tig
/* == Abstract Syntax Tree. == */

primitive print(string_0 : string)
primitive print_err(string_1 : string)
primitive print_int(int_2 : int)
primitive flush()
primitive getchar() : string
primitive ord(string_3 : string) : int
primitive chr(code_4 : int) : string
primitive size(string_5 : string) : int
primitive streq(s1_6 : string, s2_7 : string) : int
primitive strcmp(s1_8 : string, s2_9 : string) : int
primitive substring(string_10 : string, start_11 : int, length_12 : int) : string
primitive concat(fst_13 : string, snd_14 : string) : string
primitive not(boolean_15 : int) : int
primitive exit(status_16 : int)
function _main() =
  (
    streq("foo", "bar");
    ()
  )
```

Example 4.48: *`tc --desugar-string-cmp --desugar -A string-equality.tig`*

```
"foo" < "bar"
```

File 4.40: `string-less.tig`

```
$ tc --desugar-string-cmp --desugar -A string-less.tig
/* == Abstract Syntax Tree. == */

primitive print(string_0 : string)
primitive print_err(string_1 : string)
primitive print_int(int_2 : int)
primitive flush()
primitive getchar() : string
primitive ord(string_3 : string) : int
primitive chr(code_4 : int) : string
primitive size(string_5 : string) : int
primitive streq(s1_6 : string, s2_7 : string) : int
primitive strcmp(s1_8 : string, s2_9 : string) : int
primitive substring(string_10 : string, start_11 : int, length_12 : int) : string
primitive concat(fst_13 : string, snd_14 : string) : string
primitive not(boolean_15 : int) : int
primitive exit(status_16 : int)
```

```
function _main() =
  (
    (strcmp("foo", "bar") < 0);
    ()
  )
```

Example 4.49: *tc --desugar-string-cmp --desugar -A string-less.tig*

`for` loops can be seen as sugared `while` loops, and be transformed as such.

```
for i := 0 to 10 do print_int(i)
```

File 4.41: `simple-for-loop.tig`

```
$ tc --desugar-for --desugar -A simple-for-loop.tig
/* == Abstract Syntax Tree. == */

primitive print(string_0 : string)
primitive print_err(string_1 : string)
primitive print_int(int_2 : int)
primitive flush()
primitive getchar() : string
primitive ord(string_3 : string) : int
primitive chr(code_4 : int) : string
primitive size(string_5 : string) : int
primitive streq(s1_6 : string, s2_7 : string) : int
primitive strcmp(s1_8 : string, s2_9 : string) : int
primitive substring(string_10 : string, start_11 : int, length_12 : int) : string
primitive concat(fst_13 : string, snd_14 : string) : string
primitive not(boolean_15 : int) : int
primitive exit(status_16 : int)
function _main() =
  (
    let
      var _lo := 0
      var _hi := 10
      var i_17 := _lo
    in
      (if (_lo <= _hi)
        then (while 1 do
            (
              print_int(i_17);
              (if (i_17 = _hi)
                then break
                else ());
              (i_17 := (i_17 + 1))
            ))
        else ())
    end;
    ()
  )
```

Example 4.50: *tc --desugar-for --desugar -A simple-for-loop.tig*

4.10 TC-I, Function inlining

TC-I is an optional assignment.

This section has been updated for EPITA-2009 on 2007-04-26.

At the end of this stage, the compiler inlines function bodies where functions are called. In a later pass, useless functions can be pruned from the AST. These features are triggered by the options `--inline` and `--prune`. If you also implemented function overloading (see Section 4.12 [TC-A], page 123), use the options `--overfun-inline` and `--overfun-prune`.

4.10.1 TC-I Samples

```
let
  function sub(i: int, j: int) :int = i + j
in
  sub(1, 2)
end
```

File 4.42: `sub.tig`

```
$ tc -X --inline -A sub.tig
/* == Abstract Syntax Tree. == */

function _main() =
  (
    let
      function sub_2(i_0 : int, j_1 : int) : int =
        (i_0 + j_1)
    in
      let
        var i_0 : int := 1
        var j_1 : int := 2
        var res : int := (i_0 + j_1)
      in
        res
      end
    end;
    ()
  )
```

Example 4.51: *`tc -X --inline -A sub.tig`*

Recursive functions cannot be inlined.

4.11 TC-B, Array bounds checking

TC-B is an optional assignment.

This section has been updated for EPITA-2020 on 2015-01-31.

At the end of this stage, the compiler adds dynamic checks of the bounds of arrays to the AST. Every access (either on read or write) is checked, and the program should stops with the runtime exit code (120) on out-of-bounds access. This feature is triggered by the options `--bounds-checks-add` and `--overfun-bounds-checks-add`.

4.11.1 TC-B Samples

Here is an example with an out-of-bounds array subscript, run with HAVM.

```
let
  type int_array = array of int
  var foo := int_array [10] of 3
in
  /* Out-of-bounds access.  */
  foo[20]
end
```

File 4.43: `subscript-read.tig`

```
$ tc --bounds-checks-add -A subscript-read.tig
/* == Abstract Syntax Tree. == */

primitive print(string_0 : string)
primitive print_err(string_1 : string)
primitive print_int(int_2 : int)
primitive flush()
primitive getchar() : string
primitive ord(string_3 : string) : int
primitive chr(code_4 : int) : string
primitive size(string_5 : string) : int
primitive streq(s1_6 : string, s2_7 : string) : int
primitive strcmp(s1_8 : string, s2_9 : string) : int
primitive substring(string_10 : string, start_11 : int, length_12 : int) : string
primitive concat(fst_13 : string, snd_14 : string) : string
primitive not(boolean_15 : int) : int
primitive exit(status_16 : int)
function _main() =
  let
    type __int_array = array of int
    type _int_array = {
      arr : __int_array,
      size : int
    }
    function _check_bounds(a : _int_array, index : int, location : string) : int =
      (
        (if (if (index < 0)
              then 1
              else ((index >= a.size) <> 0))
          then (
              print_err(location);
              print_err(": array index out of bounds.\n");
              exit(120)
            )
          else ());
        index
      )
  in
    (
      let
        type _box_int_array_17 = {
          arr : int_array_17,
```

```
            size : int
          }
          type int_array_17 = array of int
          var foo_18 := let
              var _size := 10
            in
              _box_int_array_17 {
                arr = int_array_17 [_size] of 3,
                size = _size
              }
            end
        in
          foo_18.arr[_check_bounds(_cast(foo_18, _int_array), 20, "1.1")]
        end;
        ()
      )
    end
```

Example 4.52: *tc --bounds-checks-add -A subscript-read.tig*

```
$ tc --bounds-checks-add -L subscript-read.tig >subscript-read.lir
```

Example 4.53: *tc --bounds-checks-add -L subscript-read.tig >subscript-read.lir*

```
$ havm subscript-read.lir
error 1.1: array index out of bounds.
⇒120
```

Example 4.54: *havm subscript-read.lir*

And here is an example with an out-of-bounds assignment to an array cell, tested with Nolimips.

```
let
  type int_array = array of int
  var foo := int_array [10] of 3
in
  /* Out-of-bounds assignment.  */
  foo[42] := 51
end
```

File 4.44: subscript-write.tig

```
$ tc --bounds-checks-add -A subscript-write.tig
/* == Abstract Syntax Tree. == */

primitive print(string_0 : string)
primitive print_err(string_1 : string)
primitive print_int(int_2 : int)
primitive flush()
primitive getchar() : string
primitive ord(string_3 : string) : int
primitive chr(code_4 : int) : string
primitive size(string_5 : string) : int
```

```
primitive streq(s1_6 : string, s2_7 : string) : int
primitive strcmp(s1_8 : string, s2_9 : string) : int
primitive substring(string_10 : string, start_11 : int, length_12 : int) : string
primitive concat(fst_13 : string, snd_14 : string) : string
primitive not(boolean_15 : int) : int
primitive exit(status_16 : int)
function _main() =
  let
    type __int_array = array of int
    type _int_array = {
      arr : __int_array,
      size : int
    }
    function _check_bounds(a : _int_array, index : int, location : string) : int =
      (
        (if (if (index < 0)
              then 1
              else ((index >= a.size) <> 0))
          then (
              print_err(location);
              print_err(": array index out of bounds.\n");
              exit(120)
            )
          else ());
        index
      )
  in
    (
      let
        type _box_int_array_17 = {
          arr : int_array_17,
          size : int
        }
        type int_array_17 = array of int
        var foo_18 := let
            var _size := 10
          in
            _box_int_array_17 {
              arr = int_array_17 [_size] of 3,
              size = _size
            }
          end
      in
        (foo_18.arr[_check_bounds(_cast(foo_18, _int_array), 42, "1.1")] := 51)
      end;
      ()
    )
  end
```

Example 4.55: *tc --bounds-checks-add -A subscript-write.tig*

```
$ tc --bounds-checks-add -S subscript-write.tig >subscript-write.s
```

Example 4.56: *tc --bounds-checks-add -S subscript-write.tig >subscript-write.s*

```
$ nolimips -l nolimips -Nue subscript-write.s
error 1.1: array index out of bounds.
⇒120
```

Example 4.57: *nolimips -l nolimips -Nue subscript-write.s*

4.11.2 TC-B FAQ

The bounds checking extension relies on the use of casts (see Section 4.11.1 [TC-B Samples], page 119), see See Section "Language Extensions" in *Tiger Compiler Reference Manual*. However, a simplistic implementation of casts introduces ambiguities in the grammar that even a GLR parser cannot resolve dynamically.

Consider the following example, where `foo` is an l-value :

```
_cast(foo, string)
```

This piece of code can be parsed in two different ways:

1. `exp -> cast-exp -> exp -> lvalue (foo)`
2. `exp -> lvalue -> cast-lvalue -> lvalue (foo)`

As the cast must preserve the l-value nature of `foo`, it must itself produce an l-value. Hence we want the latter interpretation. This is a true ambiguity, not a local ambiguity that GLR can resolve simply by "waiting for enough look-ahead".

To help it take the right decision, you can favor the right path by assigning *dynamic* priorities to relevant rules, using Bison's `%dprec` keyword. See Bison's manual (see Section 5.9 [Flex & Bison], page 251) for more information on this feature.

4.12 TC-A, Ad Hoc Polymorphism (Function Overloading)

TC-A is an optional assignment.

This section has been updated for EPITA-2009 on 2007-04-26.

At the end of this stage, the compiler must be able to resolve overloaded function calls. These features are triggered by the options `--overfun-bindings-compute` and `--overfun-types-compute/-O`.

Relevant lecture notes include: `names.pdf`[20].

4.12.1 TC-A Samples

Overloaded functions are not supported in regular Tiger.

```
let
  function null(i: int) : int    = i = 0
  function null(s: string) : int = s = ""
in
  null("123") = null(123)
end
```

File 4.45: `sizes.tig`

```
$ tc -Xb sizes.tig
error sizes.tig:3.3-41: redefinition: null
```

[20] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/names.pdf`.

```
error sizes.tig:2.3-40: first definition
⇒4
```

Example 4.58: *tc -Xb sizes.tig*

Instead of regular binding, overloaded binding binds each function call to the *set* of active function definitions. Unfortunately displaying this set is not implemented, so we cannot see them in the following example:

```
$ tc -X --overfun-bindings-compute -BA sizes.tig
/* == Abstract Syntax Tree. == */

function _main /* 0x55940eca7890 */() =
  (
    let
      function null /* 0x55940eca7c20 */(i /* 0x55940ecaa700 */ : int /* 0 */) : int
        (i /* 0x55940ecaa700 */ = 0)
      function null /* 0x55940eca8610 */(s /* 0x55940eca9ce0 */ : string /* 0 */) :
        (s /* 0x55940eca9ce0 */ = "")
    in
      (null /* 0 */("123") = null /* 0 */(123))
    end;
    ()
  )
```

Example 4.59: *tc -X --overfun-bindings-compute -BA sizes.tig*

The selection of the right binding cannot be done before type-checking, since precisely overloading relies on types to distinguish the actual function called. Therefore it is the type checker that "finishes" the binding.

```
$ tc -XOBA sizes.tig
/* == Abstract Syntax Tree. == */

function _main /* 0x55ce2d384890 */() =
  (
    let
      function null /* 0x55ce2d386ed0 */(i /* 0x55ce2d385e00 */ : int /* 0 */) : int
        (i /* 0x55ce2d385e00 */ = 0)
      function null /* 0x55ce2d384b00 */(s /* 0x55ce2d3850a0 */ : string /* 0 */) :
        (s /* 0x55ce2d3850a0 */ = "")
    in
      (null /* 0x55ce2d384b00 */("123") = null /* 0x55ce2d386ed0 */(123))
    end;
    ()
  )
```

Example 4.60: *tc -XOBA sizes.tig*

There can be ambiguous (overloaded) calls.

```
let
  type foo = {}
  function empty(f: foo) : int = f = nil
  type bar = {}
```

```
  function empty(b: bar) : int = b = nil
in
  empty(foo {});
  empty(bar {});
  empty(nil)
end
```

File 4.46: `over-amb.tig`

```
$ tc -XO over-amb.tig
error over-amb.tig:9.3-12: nil ambiguity calling ‘empty’
error matching declarations:
error     empty @
error     {
error       f : foo =
error       {
error       }
error     }
error     empty @
error     {
error       b : bar =
error       {
error       }
error     }
⇒5
```

Example 4.61: *tc -XO over-amb.tig*

The spirit of plain Tiger is kept: a “chunk” is not allowed to redefine a function with the same signature:

```
let
  function foo(i: int) = ()
  function foo(i: int) = ()
in
  foo(42)
end
```

File 4.47: `over-duplicate.tig`

```
$ tc -XO over-duplicate.tig
error over-duplicate.tig:3.3-27: function complete redefinition: foo
error over-duplicate.tig:2.3-27: first definition
⇒5
```

Example 4.62: *tc -XO over-duplicate.tig*

but a signature can be defined twice in different blocks of function definitions, in which case the last defined function respecting the calling signature is used..

```
let
  function foo(i: int) = ()
in
  let
    function foo(i: int) = ()
  in
```

```
    foo(51)
  end
end
```

File 4.48: `over-scoped.tig`

```
$ tc -XOBA over-scoped.tig
/* == Abstract Syntax Tree. == */

function _main /* 0x55818121a110 */() =
  (
    let
      function foo /* 0x55818121bed0 */(i /* 0x55818121ae00 */ : int /* 0 */) =
        ()
    in
      let
        function foo /* 0x55818121a230 */(i /* 0x55818121a0a0 */ : int /* 0 */) =
          ()
      in
        foo /* 0x55818121a230 */(51)
      end
    end;
    ()
  )
```

Example 4.63: *`tc -XOBA over-scoped.tig`*

4.12.2 TC-A Given Code

No additional code is provided.

4.12.3 TC-A Code to Write

See Section 3.2.8 [src/ast], page 59, and Section 3.2.13 [src/overload], page 62.

4.13 TC-O, Desugaring object constructs

TC-O is an optional assignment.

This section has been updated for EPITA-2012 on 2015-01-21.

At the end of this stage, the compiler must be able to desugar object constructs into plain Tiger without objects, a.k.a. Panther. This feature is triggered by the option `--object-desugar`. Do not forget that you need to complete and write all missing parts of the object support (parser, ast, binder, type-checker, etc...). Make sure that all of these are correctly working before starting this bonus.

This a very hard assignment. If you plan to work on it, start with very simple programs, and progressively add new desugaring patterns. Be sure to keep a complete test suite to cover all cases and avoid regressions.

Achieving a faithful and complete translation from Tiger to Panther requires a lot of work. Even the reference implementation of the object-desugar pass (about 1,000 lines of code) is not perfect, as some inputs may generate invalid Tiger code after desugaring objects (in particular when playing with scopes).

4.13.1 TC-O Samples

Be warned: even Small object-oriented Tiger programs may generate complicated desugared outputs.

```
let
  class A {}
in
end
```

File 4.49: `empty-class.tig`

```
$ tc -X --object-desugar -A empty-class.tig
/* == Abstract Syntax Tree. == */

function _main() =
  let
    type _variant_Object = { exact_type : int }
    type _variant_A_0 = { exact_type : int }
    var _id_Object := 0
    var _id_A_0 := 1
    function _new_Object() : _variant_Object =
      _variant_Object { exact_type = _id_Object }
  in
    (
      let
        function _new_A_0() : _variant_A_0 =
          let
          in
            _variant_A_0 { exact_type = _id_A_0 }
          end
        function _upcast_A_0_to_Object(source : _variant_A_0) : _vari-
ant_Object =
          _variant_Object { exact_type = _id_A_0 }
      in
        ()
      end;
      ()
    )
  end
```

Example 4.64: *`tc -X --object-desugar -A empty-class.tig`*

```
let
  class B
  {
    var a := 42
    method m() : int = self.a
  }
  var b := new B
in
  b.a := 51
end
```

File 4.50: simple-class.tig

```
$ tc -X --object-desugar -A simple-class.tig
/* == Abstract Syntax Tree. == */

function _main() =
  let
    type _variant_Object = {
      exact_type : int,
      field_B_1 : _contents_B_1
    }
    type _contents_B_1 = { a : int }
    type _variant_B_1 = {
      exact_type : int,
      field_B_1 : _contents_B_1
    }
    var _id_Object := 0
    var _id_B_1 := 1
    function _new_Object() : _variant_Object =
      _variant_Object {
        exact_type = _id_Object,
        field_B_1 = nil
      }
  in
    (
      let
        function _new_B_1() : _variant_B_1 =
          let
            var contents_B_1 := _contents_B_1 { a = 42 }
          in
            _variant_B_1 {
              exact_type = _id_B_1,
              field_B_1 = contents_B_1
            }
          end
        function _upcast_B_1_to_Object(source : _variant_B_1) : _vari-
ant_Object =
          _variant_Object {
            exact_type = _id_B_1,
            field_B_1 = source.field_B_1
          }
        function _method_B_1_m(self : _variant_B_1) : int =
          self.field_B_1.a
        function _dispatch_B_1_m(self : _variant_B_1) : int =
          _method_B_1_m(self)
        var b_2 := _new_B_1()
      in
        (b_2.field_B_1.a := 51)
      end;
      ()
    )
```

```
  end
```

Example 4.65: *tc -X --object-desugar -A simple-class.tig*

```
let
  class C
  {
    var a := 0
    method m() : int = self.a
  }
  class D extends C
  {
    var b := 9
    /* Override C.m().  */
    method m() : int = self.a + self.b
  }
  var d : D := new D
  /* Valid upcast due to inclusion polymorphism.  */
  var c : C := d
in
  c.a := 42;
  /* Note that accessing ‘c.b’ is not allowed, since ‘c’ is
     statically known as a ‘C’, even though it is actually a ‘D’
     at run time.  */
  let
    /* Polymorphic call.  */
    var res := c.m()
  in
    print_int(res);
    print("\n")
  end
end
```

File 4.51: `override.tig`

```
$ tc --object-desugar -A override.tig
/* == Abstract Syntax Tree. == */

primitive print(string_0 : string)
primitive print_err(string_1 : string)
primitive print_int(int_2 : int)
primitive flush()
primitive getchar() : string
primitive ord(string_3 : string) : int
primitive chr(code_4 : int) : string
primitive size(string_5 : string) : int
primitive streq(s1_6 : string, s2_7 : string) : int
primitive strcmp(s1_8 : string, s2_9 : string) : int
primitive substring(string_10 : string, start_11 : int, length_12 : int) : string
primitive concat(fst_13 : string, snd_14 : string) : string
primitive not(boolean_15 : int) : int
primitive exit(status_16 : int)
function _main() =
```

```
  let
    type _variant_Object = {
      exact_type : int,
      field_C_18 : _contents_C_18,
      field_D_20 : _contents_D_20
    }
    type _contents_C_18 = { a : int }
    type _variant_C_18 = {
      exact_type : int,
      field_C_18 : _contents_C_18,
      field_D_20 : _contents_D_20
    }
    type _contents_D_20 = { b : int }
    type _variant_D_20 = {
      exact_type : int,
      field_D_20 : _contents_D_20,
      field_C_18 : _contents_C_18
    }
    var _id_Object := 0
    var _id_C_18 := 1
    var _id_D_20 := 2
    function _new_Object() : _variant_Object =
      _variant_Object {
        exact_type = _id_Object,
        field_C_18 = nil,
        field_D_20 = nil
      }
  in
    (
      let
        function _new_C_18() : _variant_C_18 =
          let
            var contents_C_18 := _contents_C_18 { a = 0 }
          in
            _variant_C_18 {
              exact_type = _id_C_18,
              field_C_18 = contents_C_18,
              field_D_20 = nil
            }
          end
        function _upcast_C_18_to_Object(source : _variant_C_18) : _vari-
ant_Object =
          _variant_Object {
            exact_type = _id_C_18,
            field_C_18 = source.field_C_18,
            field_D_20 = source.field_D_20
          }
        function _downcast_C_18_to_D_20(source : _variant_C_18) : _vari-
ant_D_20 =
          _variant_D_20 {
            exact_type = _id_D_20,
            field_D_20 = source.field_D_20,
```

```
            field_C_18 = source.field_C_18
          }
        function _method_C_18_m(self : _variant_C_18) : int =
          self.field_C_18.a
        function _dispatch_C_18_m(self : _variant_C_18) : int =
          (if (self.exact_type = _id_D_20)
            then _method_D_20_m(_downcast_C_18_to_D_20(self))
            else _method_C_18_m(self))
        function _new_D_20() : _variant_D_20 =
          let
            var contents_D_20 := _contents_D_20 { b = 9 }
            var contents_C_18 := _contents_C_18 { a = 0 }
          in
            _variant_D_20 {
              exact_type = _id_D_20,
              field_D_20 = contents_D_20,
              field_C_18 = contents_C_18
            }
          end
        function _upcast_D_20_to_C_18(source : _variant_D_20) : _vari-
ant_C_18 =
          _variant_C_18 {
            exact_type = _id_D_20,
            field_C_18 = source.field_C_18,
            field_D_20 = source.field_D_20
          }
        function _upcast_D_20_to_Object(source : _variant_D_20) : _vari-
ant_Object =
          _variant_Object {
            exact_type = _id_D_20,
            field_C_18 = source.field_C_18,
            field_D_20 = source.field_D_20
          }
        function _method_D_20_m(self : _variant_D_20) : int =
          (self.field_C_18.a + self.field_D_20.b)
        function _dispatch_D_20_m(self : _variant_D_20) : int =
          _method_D_20_m(self)
        var d_21 : _variant_D_20 := _new_D_20()
        var c_22 : _variant_C_18 := _upcast_D_20_to_C_18(d_21)
      in
        (
          (c_22.field_C_18.a := 42);
          let
            var res_23 := _dispatch_C_18_m(c_22)
          in
            (
              print_int(res_23);
              print("\n")
            )
          end
        )
      end;
```

```
      ()
    )
  end
```

Example 4.66: *tc --object-desugar -A override.tig*

```
$ tc --object-desugar -L override.tig >override.lir
```

Example 4.67: *tc --object-desugar -L override.tig >override.lir*

```
$ havm override.lir
51
```

Example 4.68: *havm override.lir*

4.14 TC-5, Translating to the High Level Intermediate Representation

2020-TC-5 submission is Saturday, April 29th 2018 at 11:42

This section has been updated for EPITA-2020 on 2016-01-27.

At the end of this stage the compiler translates the AST into the high level intermediate representation, HIR for short.

Relevant lecture notes include `intermediate.pdf`[21].

4.14.1 TC-5 Goals

Things to learn during this stage that you should remember:

Smart pointers
: The techniques used to implement reference counting via the redefinition of `operator->` and `operator*`. `std::unique_ptr`s are also smart pointers.

`std::unique_ptr`
: The intermediate translation is stored in an `unique_ptr` to guarantee it is released (`delete`) at the end of the run.

Reference counting
: The class template `misc::ref` provides reference counting smart pointers to ease the memory management. It is used to handle nodes of the intermediate representation, especially because during TC-6 some rewriting might transform this tree into an DAG, in which case memory deallocation is complex.

Variants
: C++ features the `union` keyword, inherited from C. Not only is `union` not type safe, it also forbids class members. Some people have worked hard to implement `union` à la C++, i.e., with type safety, polymorphism etc. These union are called "discriminated unions" or "variants" to follow the vocabulary introduced by Caml. See the papers from Andrei Alexandrescu: Discriminated Unions (i)[22], Discriminated Unions (ii)[23], Generic: Discriminated Unions (iii)[24] for an introduction to the techniques. We use `misc::variant` in `temp`.

[21] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/intermediate.pdf`.

[22] `http://www.drdobbs.com/cpp/discriminated-unions-i/184403821?queryText=alexandrescu%2Bdiscriminated%2Bunions`.

[23] `http://www.drdobbs.com/cpp/discriminated-unions-ii/184403828?queryText=alexandrescu%2Bdiscriminated%2Bunions`.

[24] `http://www.drdobbs.com/generic-discriminated-unions-iii/184403834?queryText=alexandrescu%2Bdiscriminated%2Bunions`.

I (Akim) strongly encourage you to read these enlightening articles.

Default copy constructor, default assignment operator
: The C++ standard specifies that unless specified, default implementations of the copy constructor and assignment operator must be provided by the compiler. There are some pitfalls though, clearly exhibited in the implementation of `misc::ref`. You must be able to explain these pitfalls.

Template template parameters
: C++ allows several kinds of entities to be used as template parameters. The most well known kind is "type": you frequently parameterize class templates with types via '`template <typename T>`' or '`template <class T>`'. But you may also parameterize with a class template. The `temp` module heavily uses this feature: understand it, and be ready to write similar code.

Explicit template instantiations
: You must be able to explain how templates are "compiled". In addition, you know how to explicitly instantiate templates, and explain what it can be used for. The implementation of `temp::Identifier` (and `temp::Temp` and `temp::Label`) is based on these ideas. See the corresponding rule in Section 2.4.3 [File Conventions], page 32, for some explanations on this topic.

Covariant return
: C++ supports covariance of the method return type. This feature is crucial to implement methods such as `clone`, as in `frame::Access::clone()`. Understand return type covariance.

Lazy/delayed computation
: The '`Ix`', '`Cx`', '`Nx`', and '`Ex`' classes delay computation to address context-depend issues in a context independent way.

Intermediate Representations

A different approach of hierarchies
: In this project, the AST is composed of different classes related by inheritance (as if the kinds of the nodes were *class* members). Here, the nodes are members of a single class, but their nature is specified by the object itself (as if the kinds of the nodes were *object* members).

Stack Frame, Activation Record
: The implementation of recursion and automatic variables.

Inner functions and their impact on memory management at runtime
: Reaching non local variables.

4.14.2 TC-5 Samples

TC-5 can be started (and should be started if you don't want to finish it in a hurry) by first making sure your compiler can handle code that uses no variables. Then, you can complete your compiler to support more and more Tiger features.

4.14.2.1 TC-5 Primitive Samples

This example is probably the simplest Tiger program.

```
0
```

File 4.52: `0.tig`

```
$ tc --hir-display 0.tig
```

```
/* == High Level Intermediate representation. == */
# Routine: _main
label main
# Prologue
# Body
seq
  sxp
    const 0
  sxp
    const 0
seq end
# Epilogue
label end
```

Example 4.69: *tc --hir-display 0.tig*

You should then probably try to make more difficult programs with literals only. Arithmetics is one of the easiest tasks.

```
1 + 2 * 3
```

File 4.53: arith.tig

```
$ tc -H arith.tig
/* == High Level Intermediate representation. == */
# Routine: _main
label main
# Prologue
# Body
seq
  sxp
    binop add
      const 1
      binop mul
        const 2
        const 3
  sxp
    const 0
seq end
# Epilogue
label end
```

Example 4.70: *tc -H arith.tig*

Use havm to exercise your output.

```
$ tc -H arith.tig >arith.hir
```

Example 4.71: *tc -H arith.tig >arith.hir*

```
$ havm arith.hir
```

Example 4.72: *havm arith.hir*

Unfortunately, without actually printing something, you won't see the final result, which means you need to implement function calls. Fortunately, you can ask `havm` for a verbose execution:

```
$ havm --trace arith.hir
error checkingLow
error plaining
error unparsing
error checking
error evaling
error  call ( name main ) []
error 9.6-9.13:  const 1
error 11.8-11.15:  const 2
error 12.8-12.15:  const 3
error 10.6-12.15:  binop mul 2 3
error 8.4-12.15:  binop add 1 6
error 7.2-12.15:  sxp 7
error 14.4-14.11:  const 0
error 13.2-14.11:  sxp 0
error  end call ( name main ) [] = 0
```

Example 4.73: *havm --trace arith.hir*

If you look carefully, you will find an '`sxp 7`' in there...

Then you are encouraged to implement control structures.

```
if 101 then 102 else 103
```

File 4.54: `if-101.tig`

```
$ tc -H if-101.tig
/* == High Level Intermediate representation. == */
# Routine: _main
label main
# Prologue
# Body
seq
  seq
    cjump ne
      const 101
      const 0
      name l0
      name l1
    label l0
    sxp
      const 102
    jump
      name l2
    label l1
    sxp
      const 103
    label l2
  seq end
```

```
    sxp
      const 0
  seq end
  # Epilogue
  label end
```

Example 4.74: *tc -H if-101.tig*

And even more difficult control structure uses:

```
while 101
  do (if 102 then break)
```

File 4.55: while-101.tig

```
$ tc -H while-101.tig
/* == High Level Intermediate representation. == */
# Routine: _main
label main
# Prologue
# Body
seq
  seq
    label l1
    cjump ne
      const 101
      const 0
      name l2
      name l0
    label l2
    seq
      cjump ne
        const 102
        const 0
        name l3
        name l4
      label l3
      jump
        name l0
      jump
        name l5
      label l4
      sxp
        const 0
      label l5
    seq end
    jump
      name l1
    label l0
  seq end
  sxp
    const 0
seq end
# Epilogue
```

```
label end
```

Example 4.75: *tc -H while-101.tig*

Beware that HAVM *has some known bugs* with its handling of `break`, see [HAVM Bugs], page 252.

4.14.2.2 TC-5 Optimizing Cascading If

Optimize the number of jumps needed to compute nested `if`, using '`translate::Ix`'. A plain use of '`translate::Cx`' is possible, but less efficient.

Consider the following sample:

```
if if 11 < 22 then 33 < 44 else 55 < 66 then print("OK\n")
```

File 4.56: `boolean.tig`

a naive implementation will probably produce too many `cjump` instructions[25]:

```
$ tc --hir-naive -H boolean.tig
/* == High Level Intermediate representation. == */
label l7
        "OK\n"
# Routine: _main
label main
# Prologue
# Body
seq
  seq
    cjump ne
      eseq
      seq
        cjump lt
          const 11
          const 22
          name l0
          name l1
        label l0
        move
          temp t0
          eseq
          seq
            move
              temp t1
              const 1
            cjump lt
              const 33
              const 44
              name l3
              name l4
            label l4
            move
              temp t1
```

[25] The option `--hir-naive` is not to be implemented.

```
              const 0
            label l3
          seq end
            temp t1
        jump
          name l2
        label l1
        move
          temp t0
          eseq
          seq
            move
              temp t2
              const 1
            cjump lt
              const 55
              const 66
              name l5
              name l6
            label l6
            move
              temp t2
              const 0
            label l5
          seq end
            temp t2
        jump
          name l2
        label l2
      seq end
        temp t0
      const 0
      name l8
      name l9
    label l8
    sxp
      call
        name print
        name l7
      call end
    jump
      name l10
    label l9
    sxp
      const 0
    jump
      name l10
    label l10
  seq end
  sxp
    const 0
seq end
```

```
# Epilogue
label end
```

Example 4.76: *tc --hir-naive -H boolean.tig*

```
$ tc --hir-naive -H boolean.tig >boolean-1.hir
```

Example 4.77: *tc --hir-naive -H boolean.tig >boolean-1.hir*

```
$ havm --profile boolean-1.hir
error /* Profiling.  */
error fetches from temporary : 2
error fetches from memory    : 0
error binary operations      : 0
error function calls         : 1
error stores to temporary    : 2
error stores to memory       : 0
error jumps                  : 2
error conditional jumps      : 3
error /* Execution time.  */
error number of cycles : 19
OK
```

Example 4.78: *havm --profile boolean-1.hir*

An analysis of this pessimization reveals that it is related to the computation of an intermediate expression (the value of '`if 11 < 22 then 33 < 44 else 55 < 66`') later decoded as a condition. A better implementation will produce:

```
$ tc -H boolean.tig
/* == High Level Intermediate representation. == */
label l0
        "OK\n"
# Routine: _main
label main
# Prologue
# Body
seq
  seq
    seq
      cjump lt
        const 11
        const 22
        name l4
        name l5
      label l4
      cjump lt
        const 33
        const 44
        name l1
        name l2
      label l5
      cjump lt
        const 55
```

```
          const 66
          name l1
          name l2
      seq end
      label l1
      sxp
        call
          name print
          name l0
        call end
      jump
        name l3
      label l2
      sxp
        const 0
      label l3
    seq end
    sxp
      const 0
  seq end
  # Epilogue
  label end
```

Example 4.79: *tc -H boolean.tig*

```
$ tc -H boolean.tig >boolean-2.hir
```

Example 4.80: *tc -H boolean.tig >boolean-2.hir*

```
$ havm --profile boolean-2.hir
error /* Profiling.  */
error fetches from temporary : 0
error fetches from memory    : 0
error binary operations      : 0
error function calls         : 1
error stores to temporary    : 0
error stores to memory       : 0
error jumps                  : 1
error conditional jumps      : 2
error /* Execution time.  */
error number of cycles : 13
OK
```

Example 4.81: *havm --profile boolean-2.hir*

4.14.2.3 TC-5 Builtin Calls Samples

The game becomes more interesting with primitive calls (which are easier to compile than function definitions and function calls).

```
(print_int(101); print("\n"))
```

File 4.57: `print-101.tig`

```
$ tc -H print-101.tig >print-101.hir
```

Example 4.82: *tc -H print-101.tig >print-101.hir*

```
$ havm print-101.hir
101
```

Example 4.83: *havm print-101.hir*

Complex values, arrays and records, also need calls to the runtime system:

```
let
  type ints = array of int
  var  ints := ints [51] of 42
in
  print_int(ints[ints[0]]); print("\n")
end
```

File 4.58: print-array.tig

```
$ tc -H print-array.tig
/* == High Level Intermediate representation. == */
label l0
        "\n"
# Routine: _main
label main
# Prologue
move
  temp t1
  temp fp
move
  temp fp
  temp sp
move
  temp sp
  binop sub
    temp sp
    const 4
# Body
seq
  seq
    move
      mem
        temp fp
      eseq
        move
          temp t0
          call
            name init_array
            const 51
            const 42
          call end
        temp t0
    seq
      sxp
```

```
          call
            name print_int
            mem
              binop add
                mem
                  temp fp
                binop mul
                  mem
                    binop add
                      mem
                        temp fp
                      binop mul
                        const 0
                        const 4
                  const 4
          call end
        sxp
          call
            name print
            name l0
          call end
      seq end
    seq end
    sxp
      const 0
  seq end
  # Epilogue
  move
    temp sp
    temp fp
  move
    temp fp
    temp t1
  label end
```

Example 4.84: *tc -H print-array.tig*

```
$ tc -H print-array.tig >print-array.hir
```

Example 4.85: *tc -H print-array.tig >print-array.hir*

```
$ havm print-array.hir
42
```

Example 4.86: *havm print-array.hir*

The case of record is more subtle. Think carefully about the following example

```
let
  type list = { h: int, t: list }
  var list := list { h = 1,
                     t = list { h = 2,
                                t = nil } }
in
```

```
  print_int(list.t.h); print("\n")
end
```

File 4.59: `print-record.tig`

4.14.2.4 TC-5 Samples with Variables

The following example demonstrates the usefulness of information about escapes: when it is not computed, all the variables are stored on the stack.

```
let
  var a := 1
  var b := 2
  var c := 3
in
  a := 2;
  c := a + b + c;
  print_int(c);
  print("\n")
end
```

File 4.60: `vars.tig`

```
$ tc -H vars.tig
/* == High Level Intermediate representation. == */
label l0
        "\n"
# Routine: _main
label main
# Prologue
move
  temp t0
  temp fp
move
  temp fp
  temp sp
move
  temp sp
  binop sub
    temp sp
    const 12
# Body
seq
  seq
    move
      mem
        temp fp
      const 1
    move
      mem
        binop add
          temp fp
          const -4
      const 2
```

```
    move
      mem
        binop add
          temp fp
          const -8
      const 3
    seq
      move
        mem
          temp fp
        const 2
      move
        mem
          binop add
            temp fp
            const -8
        binop add
          binop add
            mem
              temp fp
            mem
              binop add
                temp fp
                const -4
          mem
            binop add
              temp fp
              const -8
      sxp
        call
          name print_int
          mem
            binop add
              temp fp
              const -8
        call end
      sxp
        call
          name print
          name l0
        call end
    seq end
  seq end
  sxp
    const 0
seq end
# Epilogue
move
  temp sp
  temp fp
move
  temp fp
```

```
  temp t0
label end
```

Example 4.87: *tc -H vars.tig*

Once escaping variable computation implemented, we *know* none escape in this example, hence they can be stored in temporaries:

```
$ tc -eH vars.tig
/* == High Level Intermediate representation. == */
label l0
        "\n"
# Routine: _main
label main
# Prologue
# Body
seq
  seq
    move
      temp t0
      const 1
    move
      temp t1
      const 2
    move
      temp t2
      const 3
    seq
      move
        temp t0
        const 2
      move
        temp t2
        binop add
          binop add
            temp t0
            temp t1
          temp t2
      sxp
        call
          name print_int
          temp t2
        call end
      sxp
        call
          name print
          name l0
        call end
    seq end
  seq end
  sxp
    const 0
seq end
```

```
# Epilogue
label end
```

Example 4.88: *tc -eH vars.tig*

```
$ tc -eH vars.tig >vars.hir
```

Example 4.89: *tc -eH vars.tig >vars.hir*

```
$ havm vars.hir
7
```

Example 4.90: *havm vars.hir*

Then, you should implement the declaration of functions:

```
let
  function fact(i: int) : int =
    if i = 0 then 1
             else i * fact(i - 1)
in
  print_int(fact(15));
  print("\n")
end
```

File 4.61: fact15.tig

```
$ tc -H fact15.tig
/* == High Level Intermediate representation. == */
# Routine: fact
label l0
# Prologue
move
  temp t1
  temp fp
move
  temp fp
  temp sp
move
  temp sp
  binop sub
    temp sp
    const 8
move
  mem
    temp fp
  temp i0
move
  mem
    binop add
      temp fp
      const -4
  temp i1
# Body
move
```

```
  temp rv
  eseq
  seq
    cjump eq
      mem
        binop add
          temp fp
          const -4
      const 0
      name l1
      name l2
    label l1
    move
      temp t0
      const 1
    jump
      name l3
    label l2
    move
      temp t0
      binop mul
        mem
          binop add
            temp fp
            const -4
        call
          name l0
          mem
            temp fp
          binop sub
            mem
              binop add
                temp fp
                const -4
            const 1
        call end
    label l3
  seq end
    temp t0
# Epilogue
move
  temp sp
  temp fp
move
  temp fp
  temp t1
label end

label l4
        "\n"
# Routine: _main
label main
```

```
# Prologue
# Body
seq
  seq
    sxp
      call
        name print_int
        call
          name l0
          temp fp
          const 15
        call end
      call end
    sxp
      call
        name print
        name l4
      call end
  seq end
  sxp
    const 0
seq end
# Epilogue
label end
```

Example 4.91: *tc -H fact15.tig*

```
$ tc -H fact15.tig >fact15.hir
```

Example 4.92: *tc -H fact15.tig >fact15.hir*

```
$ havm fact15.hir
2004310016
```

Example 4.93: *havm fact15.hir*

Note that the result of 15! (1307674368000) does not fit on a signed 32-bit integer, and is therefore wrapped (to 2004310016).

And finally, you should support escaping variables (see File 4.29).

```
$ tc -eH variable-escapes.tig
/* == High Level Intermediate representation. == */
# Routine: incr
label l0
# Prologue
move
  temp t2
  temp fp
move
  temp fp
  temp sp
move
  temp sp
  binop sub
```

```
    temp sp
    const 4
move
  mem
    temp fp
  temp i0
move
  temp t1
  temp i1
# Body
move
  temp rv
  binop add
    temp t1
    mem
      mem
        temp fp
# Epilogue
move
  temp sp
  temp fp
move
  temp fp
  temp t2
label end

# Routine: _main
label main
# Prologue
move
  temp t3
  temp fp
move
  temp fp
  temp sp
move
  temp sp
  binop sub
    temp sp
    const 4
# Body
seq
  sxp
    eseq
    seq
      move
        mem
          temp fp
        const 1
      move
        temp t0
        const 2
```

```
        seq end
          call
            name l0
            temp fp
            temp t0
          call end
      sxp
        const 0
    seq end
    # Epilogue
    move
      temp sp
      temp fp
    move
      temp fp
      temp t3
    label end
```

Example 4.94: *tc -eH variable-escapes.tig*

4.14.3 TC-5 Given Code

Some code is provided through the '`tc-base`' repository, using tag '`2020-tc-base-5.0`'. For a description of the new modules, see Section 3.2.17 [src/temp], page 62, Section 3.2.18 [src/tree], page 63, Section 3.2.19 [src/frame], page 64, Section 3.2.20 [src/translate], page 64.

4.14.4 TC-5 Code to Write

You are encouraged to first try very simple examples: '`nil`', '`1 + 2`', '`"foo" < "bar"`' etc. Then consider supporting variables, and finally handle the case of the functions.

`temp::Identifier`
: Their implementations are to be finished. This task is independent of others. Passing `test-temp.cc` is probably the sign you completed correctly the implementation.

  You are invited to follow the best practices for variants, in particular, avoid "type switching" by hand, rather use variant visitors. For instance the `IdentifierEqualVisitor` can be used this way:

```
template <template <typename Tag_> class Traits_>
bool
Identifier<Traits_>::operator==(const Identifier<Traits_>& rhs) const
{
  return
    rank_get() == rhs.rank_get()
    && std::visit(IdentifierEqualToVisitor(),
                  static_cast<std::variant<unsigned, misc::symbol>>(value_
                  static_cast<std::variant<unsigned,
                  misc::symbol>>(rhs.value_));
}
```

`tree::Fragment`
: There remains to implement `tree::ProcFrag::dump` that outputs the routine themselves *plus* the glue code (allocating the frame etc.).

`translate/translation.*`
`translate::Translator`
: There are holes to fill.

4.14.5 TC-5 Options

This section documents possible extensions you could implement in TC-5.

4.14.5.1 TC-5 Bounds Checking

The implementation of the bounds checking can be done when generating the IR. Requirements are the same than for the see Section 4.11 [TC-B], page 119, option. You can use HAVM to test the success of your bounds checking.

4.14.5.2 TC-5 Optimizing Static Links

Warning: this optimization is *difficult* to do perfectly, and therefore, expect a *big* bonus.

In a first and conservative extension, the compiler considers that all the functions (but the builtins!) need a static link. This is correct, but inefficient: for instance, the traditional `fact` function will spend almost as much time handling the static link, than its real argument.

Some functions need a static link, but don't need to save it on the stack. For instance, in the following example:

```
let
  var foo := 1
  function foo() : int = foo
in
  foo()
end
```

the function `foo` does need a static link to access the variable `foo`, but does not need to store its static link on the stack.

It is suggested to address these problems in the following order:

1. Implement the detection of functions that do not *need* a static link (see exercise 6.5 in Section 5.2 [Modern Compiler Implementation], page 233), but still consider any static link escapes.
2. Adjust the output of `--escapes-display` to display '`/* escaping sl */`' *before* the first formal argument of the functions (declarations) that need the static link:

```
$ cat fact.tig
let
   function fact(n : int) : int =
      if (n = 0)
         then 1
         else n * fact((n - 1))
in
   fact(10)
end
$ tc -XEA fact.tig
/* == Abstract Syntax Tree. == */

function _main() =
  (
    let
      function fact(/* escaping sl *//* escaping */ n : int) : int =
```

```
        (if (n = 0)
          then 1
          else (n * fact((n - 1))))
    in
      fact(10)
    end;
    ()
  )
$ tc -XeEA fact.tig
/* == Abstract Syntax Tree. == */

function _main() =
  (
    let
      function fact(n : int) : int =
        (if (n = 0)
          then 1
          else (n * fact((n - 1))))
    in
      fact(10)
    end;
    ()
  )
```

3. Adjust your `call` and `progFrag` prologues.
4. Improve your computation so that non escaping static links are detected:

```
$ cat escaping-sl.tig
let
   var toto := 1
   function outer() : int =
     let function inner() : int = toto
     in inner() end
in
  outer()
end
$ tc -XeEA escaping-sl.tig
/* == Abstract Syntax Tree. == */

function _main() =
  (
    let
      var /* escaping */ toto := 1
      function outer(/* escaping sl */ ) : int =
        let
          function inner(/* sl */ ) : int =
            toto
        in
          inner()
        end
    in
      outer()
    end;
```

```
    ()
  )
```

Here, both `outer` and `inner` need their static link (so that `inner` can access `toto`. However, `outer`'s static link escapes, while `inner`'s does not.

Watch out, it is not trivial to find the minimum. What do you think about the static link of the function `sister` below?

```
let
  var v := 1
  function outer() : int =
    let
      function inner() : int = v
    in
      inner()
    end
  function sister() : int = outer()
in
  sister()
end
```

4.14.6 TC-5 FAQ

'`$fp`' or '`fp`'?

Andrew Appel clearly has his HIR/LIR depend on the target in three different ways: the names of the frame pointer and result registers[26], and the machine word size.

That would mean that the `target` module (see Section 3.2.23 [src/target], page 65) would be given during TC-5, which seemed too difficult and anti-pedagogical, so we used `fp` and `rv` where he uses `$fp` and `$v0`. While this does make TC-5 more target independent and TC-5 code base lighter, it slightly complicates the rest of the compiler.

There remains one target dependent information wired in hard: the word size is set to 4.

'`$x13`' or '`t13`'?

Anonymous temporaries should be output as '`t13`' for HAVM at stages 5 and 6, and as '`$x13`' for Nolimips, stage 7. The code provided does not support (yet) this double standard, so it always outputs '`t13`', although the samples provided here use '`$x13`'. Fortunately HAVM supports both standards[27], so this does not matter for TC-5 and TC-6. We recommend '`t13`' though, contrary to our samples, generated with a `tc` that needs more work.

How to perform the allocation of the static link in a level?

The constructor of `translate::Level` reads:

```
// Install a slot for the static link if needed.
Level::Level(const misc::symbol& name,
             const Level* parent,
             frame::bool_list_type formal_escapes)
  : parent_(parent)
  , frame_(new frame::Frame(name))
```

[26] The case of the stack pointer register is different because it is not used in the actual function body: it is referred to by the "fake" prologue/epilogue output by the `ProcFrag`.

[27] Actually temporaries in HAVM may have any name, you might use '`He110W0r1d13`' as well.

```
  {
// FIXME: Some code was deleted here (Allocate a formal for the static lin

    // Install translate::Accesses for all the formals.
    for (const bool b : formal_escapes)
      formal_alloc(b);
  }
```

To allocate a formal for the static link, look at how other formals are allocated, and take these into account:

- there is *always* a formal attribute allocated for the static link in `translate::Level`;
- this formal *always* escapes.

Obviously, this won't hold if you plan to optimize the static links (see Section 4.14.5.2 [TC-5 Optimizing Static Links], page 151); you'll have to tweak `translate::Level`'s constructor.

Why `var i := 0 function _main() = (i, ())` won't compile?

If you try to compute the intermediate representation for a single variable declaration, you'll probably run into a `SIGSEGV` or a failed assertion. For instance, the following command probably won't work: `echo 'var i := 0 function _main() = (i, ())' | tc --hir-compute -`.

Variables must be allocated in a level (see `translate::Translator::operator()(const ast::VarDec&)`). However, there is no level for global variable declarations (outside `_main`). The current language specification does not address this case, so you are free to handle it as you wish, though an assertion on the presence of an enclosing level is probably the easiest solution.

4.14.7 TC-5 Improvements

Possible improvements include:

Maximal node sharing

The proposed implementation of `Tree` creates new nodes for equal expressions; for instance two uses of the variable `foo` lead to two equal instantiations of `tree::Temp`. The same applies to more complex constructs such as the same translation if `foo` is actually a frame resident variable etc. Because memory consumption may have a negative impact on performances, it is desirable to implement maximal sharing: whenever a `Tree` is needed, we first check whether it already exists and then reuse it. This must be done recursively: the translation of '`(x + x) * (x + x)`' should have a single instantiation of '`x + x`' instead of two, but also a single instantiation of '`x`' instead of four.

Node sharing makes some algorithms, such as rewriting, more complex, especially wrt memory management. Garbage collection is almost required, but fortunately the node of `Tree` are reference counted! Therefore, almost everything is ready to implement maximal node sharing. See [spot], page 245, for an explanation on how this approach was successfully implemented. See The ATerm library[28] for a general implementation of maximally shared trees.

4.15 TC-6, Translating to the Low Level Intermediate Representation

2020-TC-6 is a part of the TC Back End option.

[28] `http://www.meta-environment.org/Meta-Environment/ATerms`.

2020-TC-6 submission is Sunday, May 20st 2018 at 11:42.

This section has been updated for EPITA-2020 on 2016-01-27.

At the end of this stage, the compiler produces low level intermediate representation: LIR. LIR is a subset of the HIR: some patterns are forbidden. This is why it is also named *canonicalization*.

Relevant lecture notes include `intermediate.pdf`[29].

4.15.1 TC-6 Goals

Things to learn during this stage that you should remember:

Term Rewriting System
: Term rewriting system are a whole topic of research in itself. If you need to be convinced, just look for "term rewriting system" on Google[30].

"Functional" Programming in C++
: A lot of TC-6 is devoted to looking for specific nodes in lists of nodes, and splitting, and splicing lists at these places. This could be done by hand, with many hand-written iterations, or using functors and STL algorithms. You are expected to do the latter, and to discover things such as `std::splice`, `std::find_if`, lambda functions, etc.

4.15.2 TC-6 Samples

There are several stages in TC-6.

4.15.2.1 TC-6 Canonicalization Samples

The first task in TC-6 is getting rid of all the `eseq`. To do this, you have to move the statement part of an `eseq` at the end of the current *sequence point*, and keeping the expression part in place.

Compare for instance the HIR to the LIR in the following case:

```
let function print_ints(a: int, b: int) =
    (print_int(a); print(", "); print_int(b); print("\n"))
    var a := 0
in
  print_ints(1, (a := a + 1; a))
end
```

File 4.62: `preincr-1.tig`

One possible HIR translation is:

```
$ tc -eH preincr-1.tig
/* == High Level Intermediate representation. == */
label l1
        ", "
label l2
        "\n"
# Routine: print_ints
label l0
# Prologue
move
```

[29] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/intermediate.pdf`.

[30] `http://www.google.com/search?q=term+rewriting+system`.

```
  temp t2
  temp fp
move
  temp fp
  temp sp
move
  temp sp
  binop sub
    temp sp
    const 4
move
  mem
    temp fp
  temp i0
move
  temp t0
  temp i1
move
  temp t1
  temp i2
# Body
seq
  sxp
    call
      name print_int
      temp t0
    call end
  sxp
    call
      name print
      name l1
    call end
  sxp
    call
      name print_int
      temp t1
    call end
  sxp
    call
      name print
      name l2
    call end
seq end
# Epilogue
move
  temp sp
  temp fp
move
  temp fp
  temp t2
label end
```

```
# Routine: _main
label main
# Prologue
# Body
seq
  seq
    move
      temp t3
      const 0
    sxp
      call
        name l0
        temp fp
        const 1
        eseq
          move
            temp t3
            binop add
              temp t3
              const 1
          temp t3
      call end
  seq end
  sxp
    const 0
seq end
# Epilogue
label end
```

Example 4.95: *tc -eH preincr-1.tig*

A possible canonicalization is then:

```
$ tc -eL preincr-1.tig
/* == Low Level Intermediate representation. == */
label l1
        ", "
label l2
        "\n"
# Routine: print_ints
label l0
# Prologue
move
  temp t2
  temp fp
move
  temp fp
  temp sp
move
  temp sp
  binop sub
    temp sp
    const 4
```

```
move
  mem
    temp fp
  temp i0
move
  temp t0
  temp i1
move
  temp t1
  temp i2
# Body
seq
  label l3
  sxp
    call
      name print_int
      temp t0
    call end
  sxp
    call
      name print
      name l1
    call end
  sxp
    call
      name print_int
      temp t1
    call end
  sxp
    call
      name print
      name l2
    call end
  label l4
seq end
# Epilogue
move
  temp sp
  temp fp
move
  temp fp
  temp t2
label end

# Routine: _main
label main
# Prologue
# Body
seq
  label l5
  move
    temp t3
```

```
    const 0
  move
    temp t5
    temp fp
  move
    temp t3
    binop add
      temp t3
      const 1
  sxp
    call
      name l0
      temp t5
      const 1
      temp t3
    call end
  label l6
seq end
# Epilogue
label end
```

Example 4.96: *tc -eL preincr-1.tig*

The example above is simple because '`1`' *commutes* with '`(a := a + 1; a)`': the order does not matter. But if you change the '`1`' into '`a`', then you cannot exchange '`a`' and '`(a := a + 1; a)`', so the translation is different. Compare the previous LIR with the following, and pay attention to

```
let function print_ints(a: int, b: int) =
    (print_int(a); print(", "); print_int(b); print("\n"))
    var a := 0
in
  print_ints(a, (a := a + 1; a))
end
```

File 4.63: `preincr-2.tig`

```
$ tc -eL preincr-2.tig
/* == Low Level Intermediate representation. == */
label l1
        ", "
label l2
        "\n"
# Routine: print_ints
label l0
# Prologue
move
  temp t2
  temp fp
move
  temp fp
  temp sp
move
  temp sp
```

```
  binop sub
    temp sp
    const 4
move
  mem
    temp fp
  temp i0
move
  temp t0
  temp i1
move
  temp t1
  temp i2
# Body
seq
  label l3
  sxp
    call
      name print_int
      temp t0
    call end
  sxp
    call
      name print
      name l1
    call end
  sxp
    call
      name print_int
      temp t1
    call end
  sxp
    call
      name print
      name l2
    call end
  label l4
seq end
# Epilogue
move
  temp sp
  temp fp
move
  temp fp
  temp t2
label end

# Routine: _main
label main
# Prologue
# Body
seq
```

```
    label l5
    move
      temp t3
      const 0
    move
      temp t5
      temp fp
    move
      temp t6
      temp t3
    move
      temp t3
      binop add
        temp t3
        const 1
    sxp
      call
        name l0
        temp t5
        temp t6
        temp t3
      call end
    label l6
  seq end
  # Epilogue
  label end
```

Example 4.97: *tc -eL preincr-2.tig*

As you can see, the output is the same for the HIR and the LIR:

```
$ tc -eH preincr-2.tig >preincr-2.hir
```

Example 4.98: *tc -eH preincr-2.tig >preincr-2.hir*

```
$ havm preincr-2.hir
0, 1
```

Example 4.99: *havm preincr-2.hir*

```
$ tc -eL preincr-2.tig >preincr-2.lir
```

Example 4.100: *tc -eL preincr-2.tig >preincr-2.lir*

```
$ havm preincr-2.lir
0, 1
```

Example 4.101: *havm preincr-2.lir*

Be very careful when dealing with `mem`. For instance, rewriting something like:

```
call(foo, eseq(move(temp t, const 51), temp t))
```

into

```
move temp t1, temp t
move temp t, const 51
```

```
call(foo, temp t)
```

is wrong: '`temp t`' is not a subexpression, rather it is being *defined* here. You should produce:

```
move temp t, const 51
call(foo, temp t)
```

Another danger is the handling of '`move(mem, )`'. For instance:

```
move(mem foo, x)
```

must be rewritten into:

```
move(temp t, foo)
move(mem(temp t), x)
```

not as:

```
move(temp t, mem(foo))
move(temp t, x)
```

In other words, the first subexpression of '`move(mem(foo), )`' is '`foo`', not '`mem(foo)`'. The following example is a good crash test against this problem:

```
let type int_array = array of int
    var tab := int_array [2] of 51
in
  tab[0] := 100;
  tab[1] := 200;
  print_int(tab[0]); print("\n");
  print_int(tab[1]); print("\n")
end
```

File 4.64: `move-mem.tig`

```
$ tc -eL move-mem.tig >move-mem.lir
```

Example 4.102: *tc -eL move-mem.tig >move-mem.lir*

```
$ havm move-mem.lir
100
200
```

Example 4.103: *havm move-mem.lir*

You also ought to get rid of nested calls:

```
print(chr(ord("\n")))
```

File 4.65: `nested-calls.tig`

```
$ tc -L nested-calls.tig
/* == Low Level Intermediate representation. == */
label l0
        "\n"
# Routine: _main
label main
# Prologue
# Body
seq
  label l1
```

```
    move
      temp t1
      call
        name ord
        name l0
      call end
    move
      temp t2
      call
        name chr
        temp t1
      call end
    sxp
      call
        name print
        temp t2
      call end
    label l2
  seq end
  # Epilogue
  label end
```

Example 4.104: *tc -L nested-calls.tig*

There are only two valid call forms: '`sxp(call(...))`', and '`move(temp(...), call(...))`'.

Contrary to C, the HIR and LIR always denote the same value. For instance the following Tiger code:

```
let
  var a := 1
  function a(t: int) : int =
     (a := a + 1;
      print_int(t); print(" -> "); print_int(a); print("\n");
      a)
  var b := a(1) + a(2) * a(3)
in
  print_int(b); print("\n")
end
```

File 4.66: `seq-point.tig`

should always produce:

```
$ tc -L seq-point.tig >seq-point.lir
```

Example 4.105: *tc -L seq-point.tig >seq-point.lir*

```
$ havm seq-point.lir
1 -> 2
2 -> 3
3 -> 4
14
```

Example 4.106: *havm seq-point.lir*

independently of the what IR you ran. *It has nothing to do with operator precedence!*

In C, you have no such guarantee: the following program can give different results with different compilers and/or on different architectures.

```
#include <stdio.h>

int a_ = 1;
int
a(int t)
{
  ++a_;
  printf("%d -> %d\n", t, a_);
  return a_;
}

int
main(void)
{
  int b = a(1) + a(2) * a(3);
  printf("%d\n", b);
  return 0;
}
```

4.15.2.2 TC-6 Scheduling Samples

Once your `eseq` and `call` canonicalized, normalize `cjumps`: they must be followed by their "false" label. This goes in two steps:

1. Split in *basic blocks*.

   A basic block is a sequence of code starting with a label, ending with a jump (conditional or not), and with no jumps, no labels inside.

2. Build the traces.

   Now put all the basic blocks into a single sequence.

The following example highlights the need for new labels: at least one for the entry point, and one for the exit point:

```
1 & 2
```

File 4.67: `1-and-2.tig`

```
$ tc -L 1-and-2.tig
/* == Low Level Intermediate representation. == */
# Routine: _main
label main
# Prologue
# Body
seq
  label l3
  cjump ne
    const 1
    const 0
```

```
      name l0
      name l1
    label l1
    label l2
    jump
      name l4
    label l0
    jump
      name l2
    label l4
  seq end
  # Epilogue
  label end
```

Example 4.107: *tc -L 1-and-2.tig*

The following example contains many jumps. Compare the HIR to the LIR:

```
while 10 | 20 do if 30 | 40 then break else break
```

File 4.68: broken-while.tig

```
$ tc -H broken-while.tig
/* == High Level Intermediate representation. == */
# Routine: _main
label main
# Prologue
# Body
seq
  seq
    label l1
    seq
      cjump ne
        const 10
        const 0
        name l3
        name l4
      label l3
      cjump ne
        const 1
        const 0
        name l2
        name l0
      label l4
      cjump ne
        const 20
        const 0
        name l2
        name l0
    seq end
    label l2
    seq
      seq
```

```
          cjump ne
            const 30
            const 0
            name l8
            name l9
          label l8
          cjump ne
            const 1
            const 0
            name l5
            name l6
          label l9
          cjump ne
            const 40
            const 0
            name l5
            name l6
        seq end
        label l5
        jump
          name l0
        jump
          name l7
        label l6
        jump
          name l0
        label l7
      seq end
      jump
        name l1
      label l0
    seq end
    sxp
      const 0
  seq end
  # Epilogue
  label end
```

Example 4.108: *tc -H broken-while.tig*

```
$ tc -L broken-while.tig
/* == Low Level Intermediate representation. == */
# Routine: _main
label main
# Prologue
# Body
seq
  label l10
  label l1
  cjump ne
    const 10
    const 0
```

```
      name l3
      name l4
    label l4
    cjump ne
      const 20
      const 0
      name l2
      name l0
    label l0
    jump
      name l11
    label l2
    cjump ne
      const 30
      const 0
      name l8
      name l9
    label l9
    cjump ne
      const 40
      const 0
      name l5
      name l6
    label l6
    jump
      name l0
    label l5
    jump
      name l0
    label l8
    cjump ne
      const 1
      const 0
      name l5
      name l13
    label l13
    jump
      name l6
    label l3
    cjump ne
      const 1
      const 0
      name l2
      name l14
    label l14
    jump
      name l0
    label l11
  seq end
  # Epilogue
  label end
```

Example 4.109: *tc -L broken-while.tig*

4.15.3 TC-6 Given Code

Some code is provided through the 'tc-base' repository, using tag '2020-tc-base-6.0'. For a description of the new module, see Section 3.2.21 [src/canon], page 64.

It includes most of the canonicalization.

4.15.4 TC-6 Code to Write

Everything you need.

4.15.5 TC-6 Improvements

Possible improvements include:

4.16 TC-7, Instruction Selection

2020-TC-7 is a part of the TC Back End option.
2020-TC-7 submission is Sunday, May 27th 2018 at 11:42.

This section has been updated for EPITA-2020 on 2016-01-27.

At the end of this stage, the compiler produces the very low level intermediate representation: ASSEM. This language is basically the target assembly, enhanced with arbitrarily many registers (`$x666`). This output is obviously target dependent: we aim at MIPS, as we use Nolimips to run it.

Relevant lecture notes include `instr-selection.pdf`[31].

4.16.1 TC-7 Goals

Things to learn during this stage that you should remember:

RISC vs. CISC etc.
: Different kinds of microprocessors, different spirits in assembly.

Assembly
: Understanding how computer actually run.

Memory hierarchy/management at runtime
: Recursive languages need memory management to implement automatic variables.

Tree matching, rewriting
: Writing/debugging a code generator with MonoBURG.

Use of `ios::xalloc`
: `Instr` are contained in `Instrs`, itself in `Fragment`, itself in `Fragments`. Suppose you mean to add a debugging flag to print an `Instr`, what shall you do? Add another argument to all the `dump` methods in these four hierarchies? The problem with `Temp` is even worse: they are scattered everywhere, yet we would like to specify how to output them thanks to a `std::map`. Should we pass this map in each and every single call?

  Using `ios::xalloc`, `ostream::pword`, and `ostream::iword` saves the day.

[31] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/instr-selection.pdf`.

4.16.2 TC-7 Samples

The goal of TC-7 is straightforward: starting from LIR, generate the MIPS instructions, except that you don't have actual registers: we still heavily use `Temp`s. Register allocation will be done in a later stage, Section 4.18 [TC-9], page 189.

```
let
  var answer := 42
in
  answer := 51
end
```

File 4.69: `the-answer.tig`

```
$ tc --inst-display the-answer.tig
# == Final assembler ouput. == #
# Routine: _main
tc_main:
# Allocate frame
        move    $x11, $ra
        move    $x3, $s0
        move    $x4, $s1
        move    $x5, $s2
        move    $x6, $s3
        move    $x7, $s4
        move    $x8, $s5
        move    $x9, $s6
        move    $x10, $s7
l0:
        li      $x1, 42
        sw      $x1, ($fp)
        li      $x2, 51
        sw      $x2, ($fp)
l1:
        move    $s0, $x3
        move    $s1, $x4
        move    $s2, $x5
        move    $s3, $x6
        move    $s4, $x7
        move    $s5, $x8
        move    $s6, $x9
        move    $s7, $x10
        move    $ra, $x11
# Deallocate frame
        jr      $ra
```

Example 4.110: `tc --inst-display the-answer.tig`

At this stage the compiler cannot know what registers are used; that's why in the previous output it saves "uselessly" all the callee-save registers on `main` entry. For the same reason, the frame is not allocated.

While Nolimips accepts the lack of register allocation, it does require the frame to be allocated. That is the purpose of `--nolimips-display`:

```
$ tc --nolimips-display the-answer.tig
```

```
# == Final assembler ouput. == #
# Routine: _main
tc_main:
        sw      $fp, -4 ($sp)
        move    $fp, $sp
        sub     $sp, $sp, 8
        move    $x11, $ra
        move    $x3, $s0
        move    $x4, $s1
        move    $x5, $s2
        move    $x6, $s3
        move    $x7, $s4
        move    $x8, $s5
        move    $x9, $s6
        move    $x10, $s7
l0:
        li      $x1, 42
        sw      $x1, ($fp)
        li      $x2, 51
        sw      $x2, ($fp)
l1:
        move    $s0, $x3
        move    $s1, $x4
        move    $s2, $x5
        move    $s3, $x6
        move    $s4, $x7
        move    $s5, $x8
        move    $s6, $x9
        move    $s7, $x10
        move    $ra, $x11
        move    $sp, $fp
        lw      $fp, -4 ($fp)
        jr      $ra
```

Example 4.111: *tc --nolimips-display the-answer.tig*

The final stage, register allocation, addresses both issues. For your information, it results in:

```
$ tc -sI the-answer.tig
# == Final assembler ouput. == #
# Routine: _main
tc_main:
        sw      $fp, -4 ($sp)
        move    $fp, $sp
        sub     $sp, $sp, 8
l0:
        li      $t0, 42
        sw      $t0, ($fp)
        li      $t0, 51
        sw      $t0, ($fp)
l1:
        move    $sp, $fp
```

```
        lw      $fp, -4 ($fp)
        jr      $ra
```

Example 4.112: *tc -sI the-answer.tig*

A delicate part of this exercise is handling the function calls:

```
let function add(x: int, y: int) : int = x + y
in
  print_int(add(1,(add(2, 3)))); print("\n")
end
```

File 4.70: add.tig

```
$ tc -e --inst-display add.tig
# == Final assembler ouput. == #
# Routine: add
tc_l0:
# Allocate frame
        move    $x15, $ra
        sw      $a0, ($fp)
        move    $x0, $a1
        move    $x1, $a2
        move    $x7, $s0
        move    $x8, $s1
        move    $x9, $s2
        move    $x10, $s3
        move    $x11, $s4
        move    $x12, $s5
        move    $x13, $s6
        move    $x14, $s7
l2:
        add     $x6, $x0, $x1
        move    $v0, $x6
l3:
        move    $s0, $x7
        move    $s1, $x8
        move    $s2, $x9
        move    $s3, $x10
        move    $s4, $x11
        move    $s5, $x12
        move    $s6, $x13
        move    $s7, $x14
        move    $ra, $x15
# Deallocate frame
        jr      $ra

.data
l1:
        .word 1
        .asciiz "\n"
.text
```

```
# Routine: _main
tc_main:
# Allocate frame
        move    $x28, $ra
        move    $x20, $s0
        move    $x21, $s1
        move    $x22, $s2
        move    $x23, $s3
        move    $x24, $s4
        move    $x25, $s5
        move    $x26, $s6
        move    $x27, $s7
l4:
        move    $a0, $fp
        li      $x16, 2
        move    $a1, $x16
        li      $x17, 3
        move    $a2, $x17
        jal     tc_l0
        move    $x4, $v0
        move    $a0, $fp
        li      $x18, 1
        move    $a1, $x18
        move    $a2, $x4
        jal     tc_l0
        move    $x5, $v0
        move    $a0, $x5
        jal     tc_print_int
        la      $x19, l1
        move    $a0, $x19
        jal     tc_print
l5:
        move    $s0, $x20
        move    $s1, $x21
        move    $s2, $x22
        move    $s3, $x23
        move    $s4, $x24
        move    $s5, $x25
        move    $s6, $x26
        move    $s7, $x27
        move    $ra, $x28
# Deallocate frame
        jr      $ra
```

Example 4.113: *tc -e --inst-display add.tig*

Once your function calls work properly, you can start using Nolimips (using options `--nop-after-branch --unlimited-registers --execute`) to check the behavior of your compiler.

```
$ tc -eR --nolimips-display add.tig >add.nolimips
```

Example 4.114: *tc -eR --nolimips-display add.tig >add.nolimips*

```
$ nolimips -l nolimips -Nue add.nolimips
6
```

Example 4.115: *nolimips -l nolimips -Nue add.nolimips*

You must also complete the runtime. No difference must be observable between a run with HAVM and another with Nolimips:

```
substring("", 1, 1)
```

File 4.71: substring-0-1-1.tig

```
$ tc -e --nolimips-display substring-0-1-1.tig
# == Final assembler ouput. == #
.data
l0:
        .word 0
        .asciiz ""
.text

# Routine: _main
tc_main:
# Allocate frame
        move    $x12, $ra
        move    $x4, $s0
        move    $x5, $s1
        move    $x6, $s2
        move    $x7, $s3
        move    $x8, $s4
        move    $x9, $s5
        move    $x10, $s6
        move    $x11, $s7
l1:
        la      $x1, l0
        move    $a0, $x1
        li      $x2, 1
        move    $a1, $x2
        li      $x3, 1
        move    $a2, $x3
        jal     tc_substring
l2:
        move    $s0, $x4
        move    $s1, $x5
        move    $s2, $x6
        move    $s3, $x7
        move    $s4, $x8
        move    $s5, $x9
        move    $s6, $x10
        move    $s7, $x11
        move    $ra, $x12
```

```
# Deallocate frame
        jr      $ra
```

Example 4.116: *tc -e --nolimips-display substring-0-1-1.tig*

```
$ tc -eR --nolimips-display substring-0-1-1.tig >substring-0-1-1.nolimips
```

Example 4.117: *tc -eR --nolimips-display substring-0-1-1.tig >substring-0-1-1.nolimips*

```
$ nolimips -l nolimips -Nue substring-0-1-1.nolimips
error substring: arguments out of bounds
⇒120
```

Example 4.118: *nolimips -l nolimips -Nue substring-0-1-1.nolimips*

4.16.3 TC-7 Given Code

Some code is provided through the 'tc-base' repository, using tag '2020-tc-base-7.0'. For more information about the TC-7 code delivered see Section 3.2.23 [src/target], page 65, Section 3.2.22 [src/assem], page 64.

4.16.4 TC-7 Code to Write

There is not much code to write:

- `Codegen` (`src/target/mips/call.brg`, `src/target/mips/move.brg`): complete some rules in the grammar of the code generator produced by MonoBURG.
- `SpimAssembly::move_build` (`src/target/mips/spim-assembly.cc`): build a move instruction using MIPS R2000 standard instruction set.
- `SpimAssembly::binop_inst`, `SpimAssembly::binop_build` (`src/target/mips/spim-assembly.cc`): build arithmetic binary operations (addition, multiplication, etc.) using MIPS R2000 standard instruction set.
- `SpimAssembly::load_build`, `SpimAssembly::store_build` (`src/target/mips/spim-assembly.cc`): build a load (respectively a store) instruction using MIPS R2000 standard instruction set. Here, the indirect addressing mode is used.
- `SpimAssembly::cjump_build` (`src/target/mips/spim-assembly.cc`): translate conditional branch instructions (branch if equal, if lower than, etc.) into MIPS R2000 assembly.
- You have to complete the implementation of the runtime in `src/target/mips/runtime.s`:

  `strcmp`

  `streq`

  `print_int`
  `substring`
  `concat`

  Information on MIPS R2000 assembly instructions may be found in SPIM manual.

Completing the following routines will be needed during register allocation only (see Section 4.18 [TC-9], page 189):

- `Codegen::rewrite_program` (`src/target/mips/epilogue.cc`)

4.16.5 TC-7 FAQ

Nolimips 'Precondition 'has_unlimited(reg.get_index())' failed'
: This lovely error message is the sign you're using an obsolete version of Nolimips. Update.

4.16.6 TC-7 Improvements

Possible improvements include:

4.17 TC-8, Liveness Analysis

2020-TC-8 is a part of the TC Back End option.
2020-TC-8 submission is Sunday, June 10th 2018 at 11:42.

This section has been updated for EPITA-2020 on 2016-01-27.

At the end of this stage, the compiler computes the input of TC-9: the *interference graph* (or *conflict graph*). The options `-N` and `--interference-dump` allow the user to see these graphs, one per function. To compute the interference graph, the compiler first computes the *liveness* of each temporary, i.e., a graph whose nodes are the instructions, and labeled with *live temporaries*. The options `-V`, `--liveness-dump` dumps these graphs. Finally, the structure of the liveness graph is the *flow graph*: its nodes are the instructions, and edges correspond to control flow. Use options `-F`, `--flowgraph-dump` to dump them.

All dumped graphs use the DOT format. You can display them using `dotty` or convert them to other formats (such as PDF or PNG) using `dot`, both part of the GraphViz package.

Relevant lecture notes include `liveness.pdf`[32].

4.17.1 TC-8 Goals

Things to learn during this stage that you should remember:

Graph handling, using the Boost Graph Library
: We use the Boost Graph Library[33] to implement graphs in the Tiger Compiler. You must be able to manipulate Boost Graphs, and understand some aspects of their design.

Flow graph
Liveness

Interference graph/conflict graph

4.17.2 TC-8 Samples

First consider simple examples, without any branching:

```
10 + 20 * 30
```

File 4.72: `tens.tig`

```
$ tc -I tens.tig
# == Final assembler ouput. == #
# Routine: _main
tc_main:
# Allocate frame
        move    $x13, $ra
```

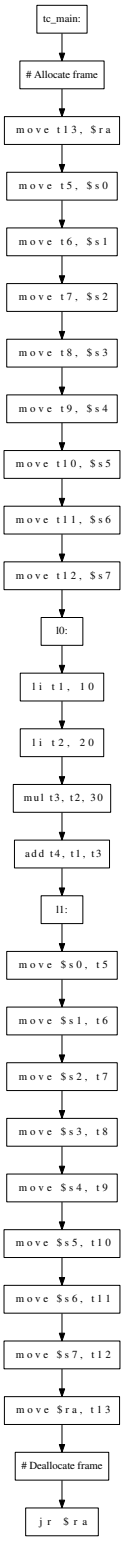
[32] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/liveness.pdf`.
[33] `http://www.boost.org/libs/graph/doc/`.

```
        move    $x5, $s0
        move    $x6, $s1
        move    $x7, $s2
        move    $x8, $s3
        move    $x9, $s4
        move    $x10, $s5
        move    $x11, $s6
        move    $x12, $s7
l0:
        li      $x1, 10
        li      $x2, 20
        mul     $x3, $x2, 30
        add     $x4, $x1, $x3
l1:
        move    $s0, $x5
        move    $s1, $x6
        move    $s2, $x7
        move    $s3, $x8
        move    $s4, $x9
        move    $s5, $x10
        move    $s6, $x11
        move    $s7, $x12
        move    $ra, $x13
# Deallocate frame
        jr      $ra
```

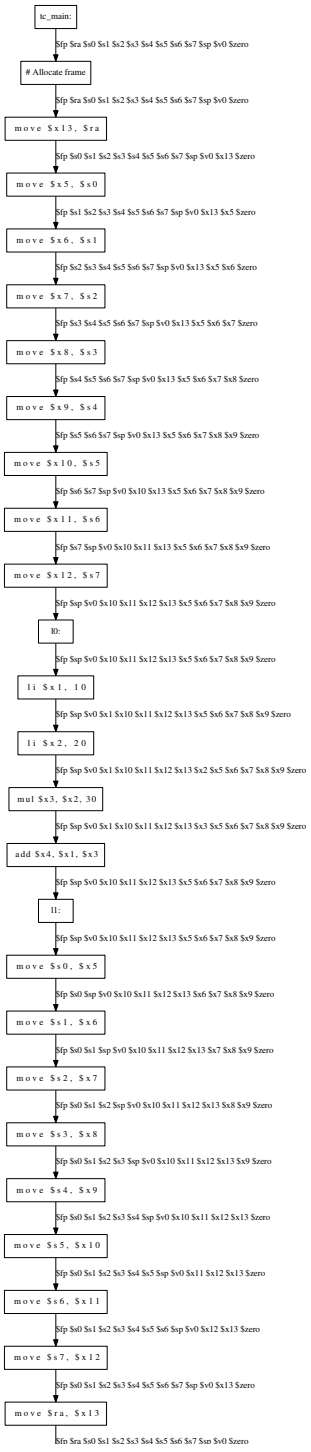
Example 4.119: *tc -I tens.tig*

```
$ tc -FVN tens.tig
```

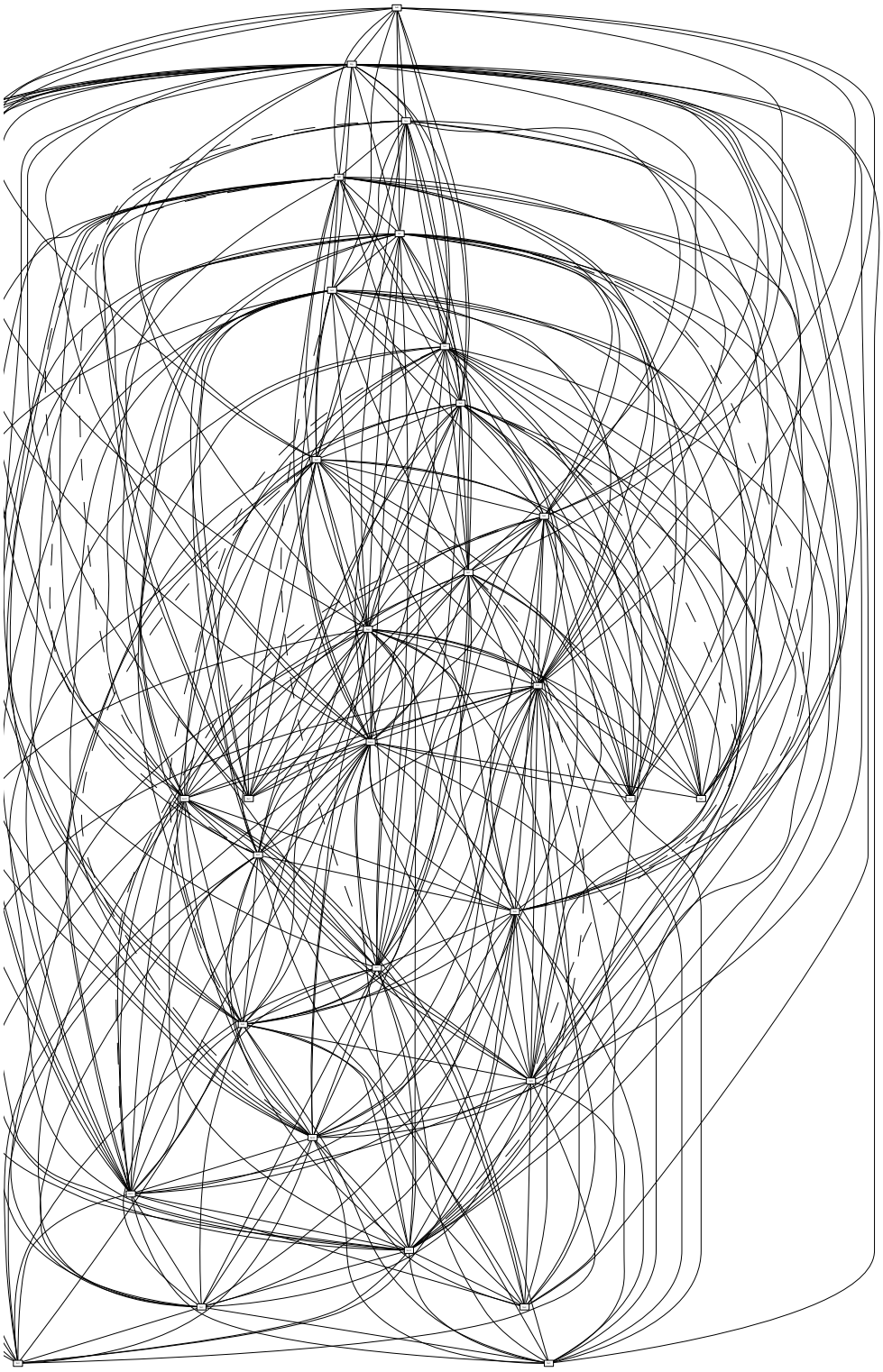
Example 4.120: *tc -FVN tens.tig*



File 4.73: `tens.main._main.flow.gv`



File 4.74: `tens.main._main.liveness.gv`



File 4.75: `tens.main._main.interference.gv`

But as you can see, the result is quite hairy, and unreadable, especially for interference graphs:

– the callee save registers ('`$s0`' to '`$s7`' on Mips) collide with every other temporary.
– the callee save registers have to be... saved, which doubles the number of `Temp`.

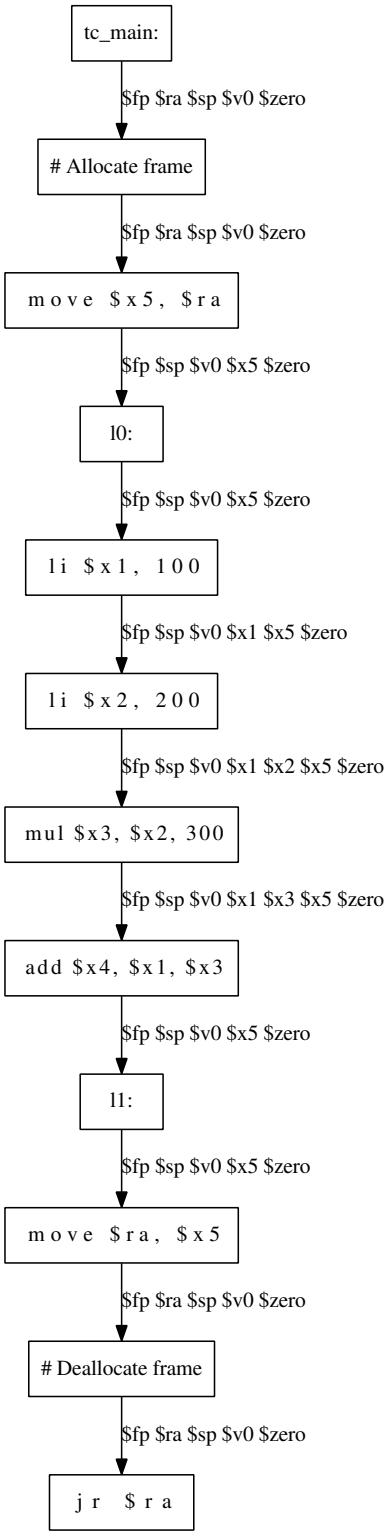
To circumvent this problem, use `--callee-save` to limit the number of such registers:

```
100 + 200 * 300
```

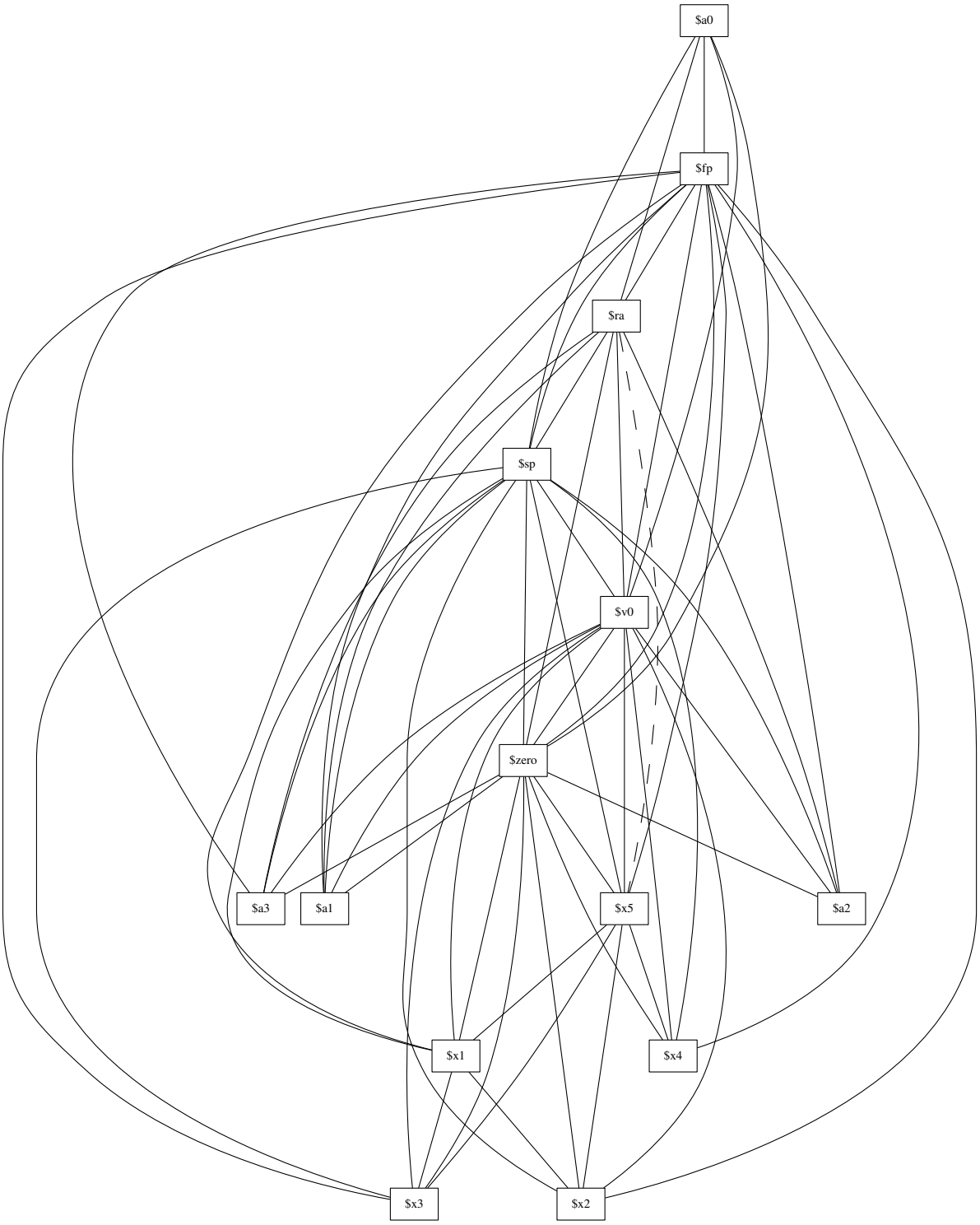
File 4.76: `hundreds.tig`

```
$ tc --callee-save=0 -VN hundreds.tig
```

Example 4.121: *`tc --callee-save=0 -VN hundreds.tig`*



File 4.77: `hundreds.main._main.liveness.gv`



File 4.78: `hundreds.main._main.interference.gv`

Branching is of course a most interesting feature to exercise:

```
1 | 2 | 3
```

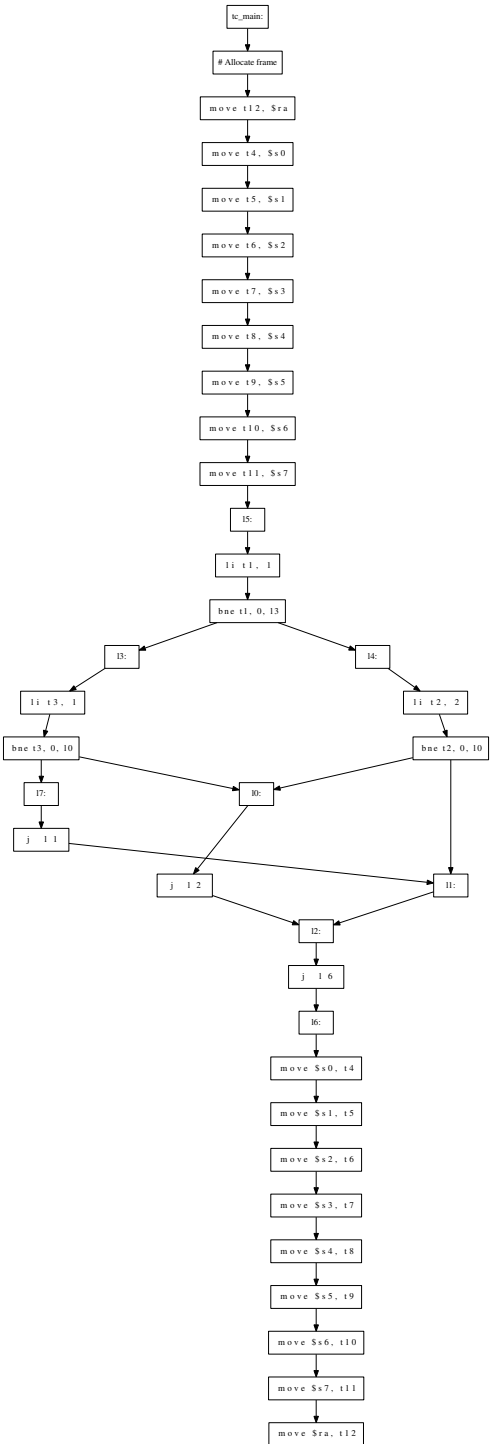
File 4.79: `ors.tig`

```
$ tc --callee-save=0 -I ors.tig
# == Final assembler ouput. == #
# Routine: _main
tc_main:
# Allocate frame
        move    $x4, $ra
l5:
        li      $x1, 1
        bne     $x1, 0, l3
l4:
        li      $x2, 2
        bne     $x2, 0, l0
l1:
l2:
        j       l6
l0:
        j       l2
l3:
        li      $x3, 1
        bne     $x3, 0, l0
l7:
        j       l1
l6:
        move    $ra, $x4
# Deallocate frame
        jr      $ra
```

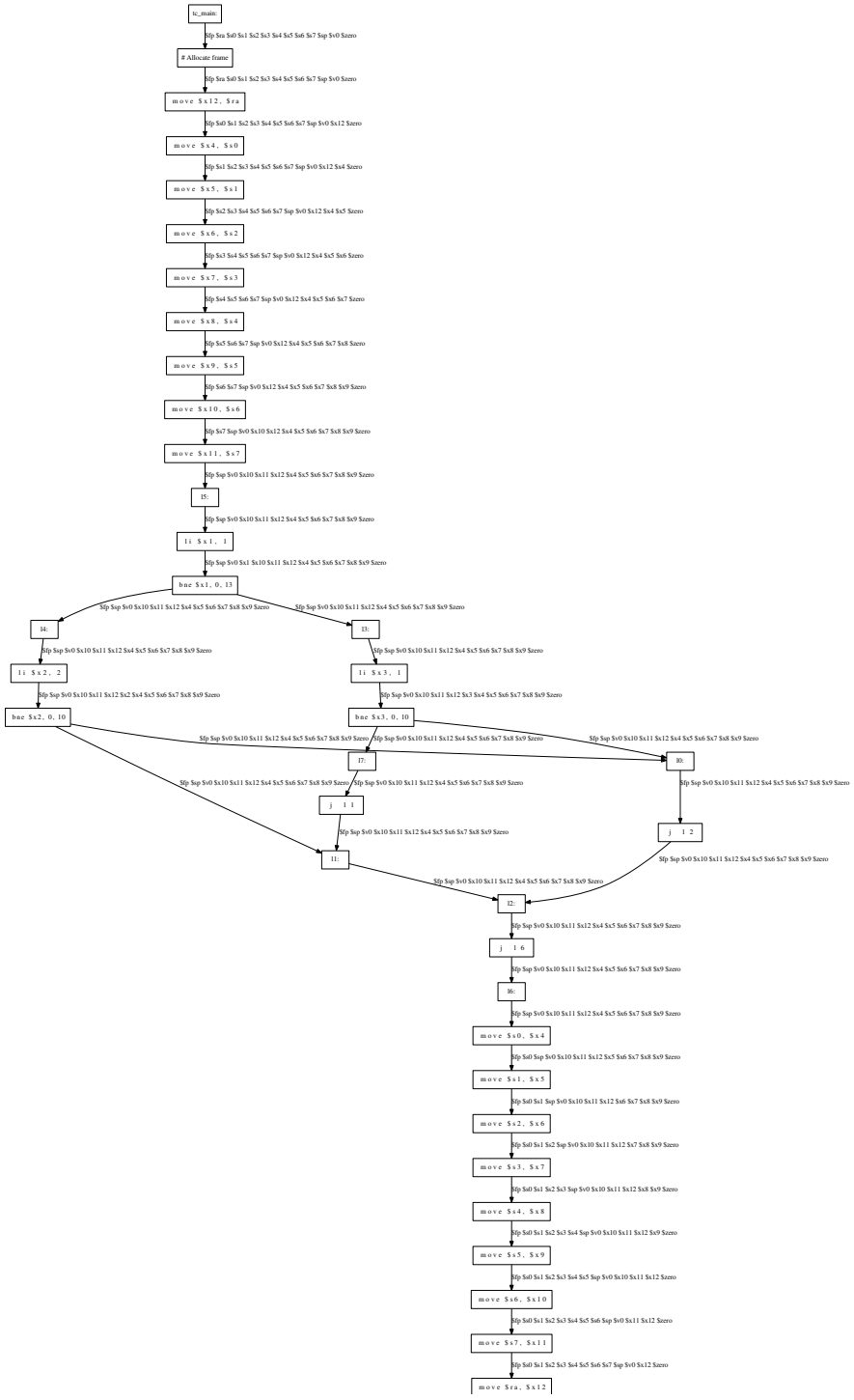
Example 4.122: *tc --callee-save=0 -I ors.tig*

```
$ tc -FVN ors.tig
```

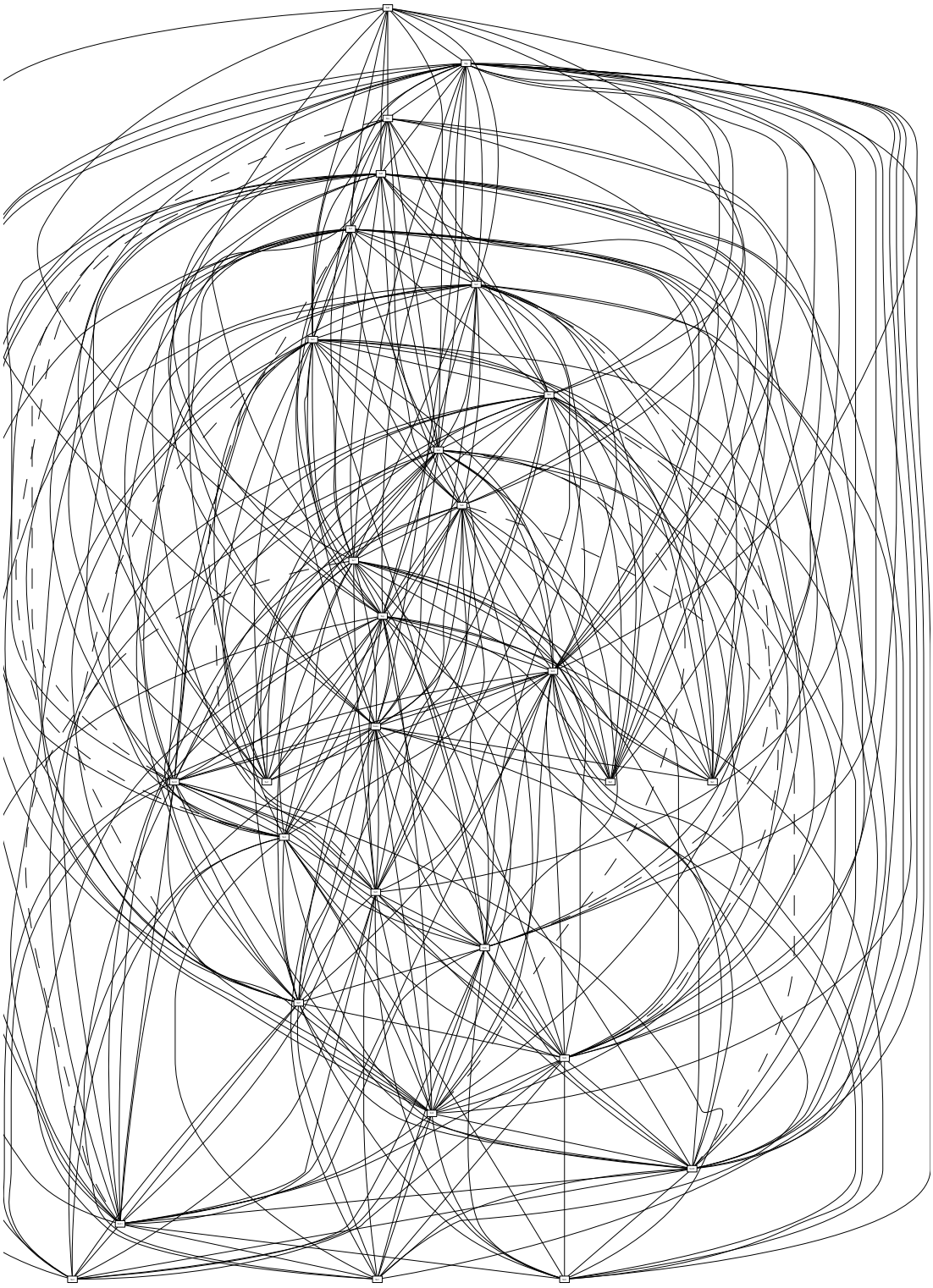
Example 4.123: *tc -FVN ors.tig*



File 4.80: `ors.main._main.flow.gv`



File 4.81: `ors.main._main.liveness.gv`



File 4.82: `ors.main._main.interference.gv`

4.17.3 TC-8 Given Code

Some code is provided through the '`tc-base`' repository, using tag '`2020-tc-base-8.0`'. To read the description of the new modules, see Section 3.2.4 [lib/misc], page 56, Section 3.2.27 [src/liveness], page 68.

4.17.4 TC-8 Code to Write

`lib/misc/graph.*`
: Implement the topological sort.

`src/liveness/flowgraph.*`
: Write the constructor, which is where the `FlowGraph` is actually constructed from the assembly fragments.

`src/liveness/liveness.*`
: Write the constructor, which is where the `Liveness` (a decorated `FlowGraph`) is built from assembly instructions.

`src/liveness/interference-graph.*`
: In `InterferenceGraph::compute_liveness`, build the graph.

4.17.5 TC-8 FAQ

Why do we have a `TempMap`, and not Appel?
: See [$fp or fp], page 153, for all the details. Pay special attention to converting the temporaries where needed:

  - the flow graph is independent of the temporaries
  - the liveness graph, when computing live-in and live-out sets, must of course convert the "def" and "use" sets
  - the interference graph, when attributing a node *number* for each temporary (`InterferenceGraph::node_of`), must allocate the same number to corresponding temporaries (e.g., '`$fp`' and '`fp`' must bear the same number).

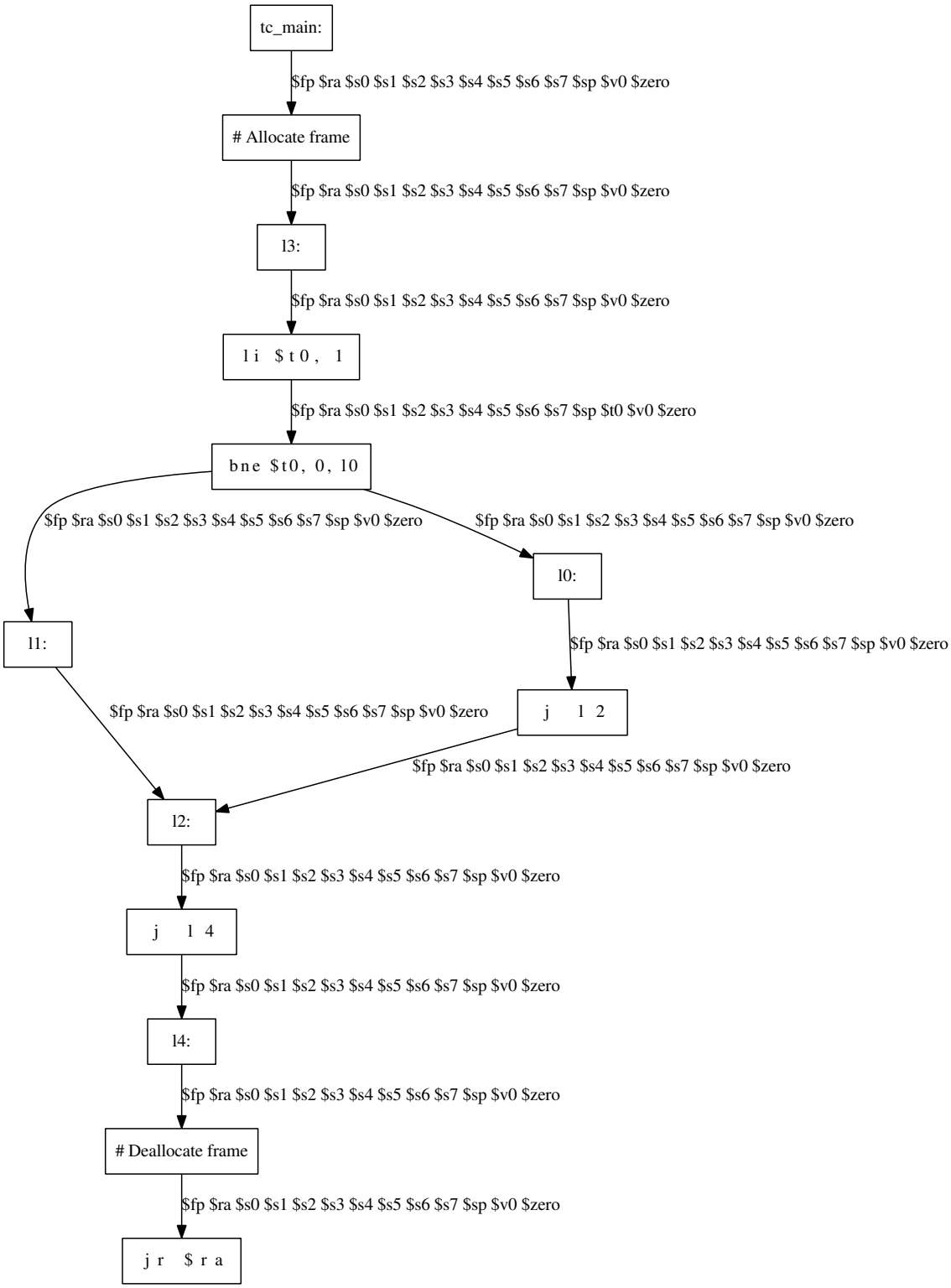
  There is another reason to use a `TempMap` here: to build the liveness graph *after* register allocation, to check the compiler.

```
1 & 2
```

File 4.83: `and.tig`

```
$ tc -sV and.tig
```

Example 4.124: *`tc -sV and.tig`*



File 4.84: `and.main._main.liveness.gv`

4.17.6 TC-8 Improvements

Possible improvements include:

4.18 TC-9, Register Allocation

2020-TC-9 is a part of the TC Back End option.
2020-TC-9 submission is Sunday, July 8th 2018 at 11:42.

This section has been updated for EPITA-2020 on 2016-01-27.

At the end of this stage, the compiler produces code that is runnable using Nolimips.

Relevant lecture notes include `regalloc.pdf`[34].

4.18.1 TC-9 Goals

Things to learn during this stage that you should remember:

- Use of work lists for efficiency
- Attacking NP complete problems
- Register allocation as graph coloring

4.18.2 TC-9 Samples

This section will not demonstrate the output of the option `-S`, `--asm-display`, since it outputs the long Tiger runtime. *Once the registers allocated* (i.e., once `-s`, `--asm-compute` executed) the option `-I`, `--instr-display` produces the code without the runtime. In short: we use `-sI` instead of `-S` to save place.

Allocating registers in the main function, when there is no register pressure is easy, as, in particular, there are no spills. A direct consequence is that many `move` are now useless, and have disappeared. For instance (File 4.85, see Example 4.125):

```
1 + 2 * 3
```

File 4.85: `seven.tig`

```
$ tc -sI seven.tig
# == Final assembler ouput. == #
# Routine: _main
tc_main:
# Allocate frame
l0:
        li      $t1, 1
        li      $t0, 2
        mul     $t0, $t0, 3
        add     $t0, $t1, $t0
l1:
# Deallocate frame
        jr      $ra
```

Example 4.125: *`tc -sI seven.tig`*

```
$ tc -S seven.tig >seven.s
```

Example 4.126: *`tc -S seven.tig >seven.s`*

```
$ nolimips -l nolimips -Ne seven.s
```

Example 4.127: *`nolimips -l nolimips -Ne seven.s`*

[34] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/regalloc.pdf`.

Another means to display the result of register allocation consists in reporting the mapping from `temp`s to actual registers:

```
$ tc -s --tempmap-display seven.tig
/* Temporary map. */
fp -> $fp
rv -> $v0
t1 -> $t1
t2 -> $t0
t3 -> $t0
t4 -> $t0
t5 -> $s0
t6 -> $s1
t7 -> $s2
t8 -> $s3
t9 -> $s4
t10 -> $s5
t11 -> $s6
t12 -> $s7
t13 -> $ra
```

Example 4.128: *tc -s --tempmap-display seven.tig*

Of course it is much better to *see* what is going on:

```
(print_int(1 + 2 * 3); print("\n"))
```

File 4.86: `print-seven.tig`

```
$ tc -sI print-seven.tig
# == Final assembler ouput. == #
.data
l0:
        .word 1
        .asciiz "\n"
.text

# Routine: _main
tc_main:
        sw      $fp, -4 ($sp)
        move    $fp, $sp
        sub     $sp, $sp, 8
        sw      $ra, ($fp)
l1:
        li      $t0, 1
        li      $ra, 2
        mul     $ra, $ra, 3
        add     $a0, $t0, $ra
        jal     tc_print_int
        la      $a0, l0
        jal     tc_print
l2:
        lw      $ra, ($fp)
        move    $sp, $fp
```

```
        lw      $fp, -4 ($fp)
        jr      $ra
```

Example 4.129: *tc -sI print-seven.tig*

```
$ tc -S print-seven.tig >print-seven.s
```

Example 4.130: *tc -S print-seven.tig >print-seven.s*

```
$ nolimips -l nolimips -Ne print-seven.s
7
```

Example 4.131: *nolimips -l nolimips -Ne print-seven.s*

To torture your compiler, you ought to use many temporaries. To be honest, ours is quite slow, it spends way too much time in register allocation.

```
let
  var a00 := 00      var a55 := 55
  var a11 := 11      var a66 := 66
  var a22 := 22      var a77 := 77
  var a33 := 33      var a88 := 88
  var a44 := 44      var a99 := 99
in
  print_int(0
            +  a00 + a00 + a55 + a55
            +  a11 + a11 + a66 + a66
            +  a22 + a22 + a77 + a77
            +  a33 + a33 + a88 + a88
            +  a44 + a44 + a99 + a99);
  print("\n")
end
```

File 4.87: print-many.tig

```
$ tc -eIs --tempmap-display -I --time-report print-many.tig
error Execution times (seconds)
error  1: parse                : 0.01   (   50%)   0      (    0%)   0.01   (  100%)
error  9: asm-compute          : 0.01   (   50%)   0      (    0%)   0      (    0%)
error  rest                    : 0.02   (  100%)   0      (    0%)   0.01   (  100%)
error Cumulated times (seconds)
error  1: parse                : 0.01   (   50%)   0      (    0%)   0.01   (  100%)
error  rest                    : 0.02   (  100%)   0      (    0%)   0.01   (  100%)
error  TOTAL (seconds)         : 0.02   user,      0      system,    0.01   wall
# == Final assembler ouput. == #
.data
l0:
        .word 1
        .asciiz "\n"
.text

# Routine: _main
tc_main:
# Allocate frame
```

```
        move    $x41, $ra
        move    $x33, $s0
        move    $x34, $s1
        move    $x35, $s2
        move    $x36, $s3
        move    $x37, $s4
        move    $x38, $s5
        move    $x39, $s6
        move    $x40, $s7
l1:
        li      $x0, 0
        li      $x1, 55
        li      $x2, 11
        li      $x3, 66
        li      $x4, 22
        li      $x5, 77
        li      $x6, 33
        li      $x7, 88
        li      $x8, 44
        li      $x9, 99
        li      $x11, 0
        add     $x12, $x11, $x0
        add     $x13, $x12, $x0
        add     $x14, $x13, $x1
        add     $x15, $x14, $x1
        add     $x16, $x15, $x2
        add     $x17, $x16, $x2
        add     $x18, $x17, $x3
        add     $x19, $x18, $x3
        add     $x20, $x19, $x4
        add     $x21, $x20, $x4
        add     $x22, $x21, $x5
        add     $x23, $x22, $x5
        add     $x24, $x23, $x6
        add     $x25, $x24, $x6
        add     $x26, $x25, $x7
        add     $x27, $x26, $x7
        add     $x28, $x27, $x8
        add     $x29, $x28, $x8
        add     $x30, $x29, $x9
        add     $x31, $x30, $x9
        move    $a0, $x31
        jal     tc_print_int
        la      $x32, l0
        move    $a0, $x32
        jal     tc_print
l2:
        move    $s0, $x33
        move    $s1, $x34
        move    $s2, $x35
        move    $s3, $x36
        move    $s4, $x37
```

```
        move    $s5, $x38
        move    $s6, $x39
        move    $s7, $x40
        move    $ra, $x41
# Deallocate frame
        jr      $ra
/* Temporary map. */
fp -> $fp
rv -> $v0
t0 -> $t9
t1 -> $t8
t2 -> $t7
t3 -> $t6
t4 -> $t5
t5 -> $t4
t6 -> $t3
t7 -> $t2
t8 -> $t1
t9 -> $t0
t11 -> $ra
t12 -> $ra
t13 -> $ra
t14 -> $ra
t15 -> $ra
t16 -> $ra
t17 -> $ra
t18 -> $ra
t19 -> $ra
t20 -> $ra
t21 -> $ra
t22 -> $ra
t23 -> $ra
t24 -> $ra
t25 -> $ra
t26 -> $ra
t27 -> $ra
t28 -> $ra
t29 -> $ra
t30 -> $ra
t31 -> $a0
t32 -> $a0
t33 -> $s0
t34 -> $s1
t35 -> $s2
t36 -> $s3
t37 -> $s4
t38 -> $s5
t39 -> $s6
t40 -> $s7
t110 -> $ra
t111 -> $ra
```

```
# == Final assembler ouput. == #
.data
l0:
        .word 1
        .asciiz "\n"
.text

# Routine: _main
tc_main:
        sw      $fp, -4 ($sp)
        move    $fp, $sp
        sub     $sp, $sp, 8
        sw      $ra, ($fp)
l1:
        li      $t9, 0
        li      $t8, 55
        li      $t7, 11
        li      $t6, 66
        li      $t5, 22
        li      $t4, 77
        li      $t3, 33
        li      $t2, 88
        li      $t1, 44
        li      $t0, 99
        li      $ra, 0
        add     $ra, $ra, $t9
        add     $ra, $ra, $t9
        add     $ra, $ra, $t8
        add     $ra, $ra, $t8
        add     $ra, $ra, $t7
        add     $ra, $ra, $t7
        add     $ra, $ra, $t6
        add     $ra, $ra, $t6
        add     $ra, $ra, $t5
        add     $ra, $ra, $t5
        add     $ra, $ra, $t4
        add     $ra, $ra, $t4
        add     $ra, $ra, $t3
        add     $ra, $ra, $t3
        add     $ra, $ra, $t2
        add     $ra, $ra, $t2
        add     $ra, $ra, $t1
        add     $ra, $ra, $t1
        add     $ra, $ra, $t0
        add     $a0, $ra, $t0
        jal     tc_print_int
        la      $a0, l0
        jal     tc_print
l2:
        lw      $ra, ($fp)
        move    $sp, $fp
        lw      $fp, -4 ($fp)
```

```
        jr      $ra
```

Example 4.132: *tc -eIs --tempmap-display -I --time-report print-many.tig*

4.18.3 TC-9 Given Code

Some code is provided through the 'tc-base' repository, using tag '2020-tc-base-9.0'. To read the description of the new module, see Section 3.2.29 [src/regalloc], page 70.

4.18.4 TC-9 Code to Write

`src/regalloc/color.hh`
: Implement the graph coloring. The skeleton we provided is an exact copy of the implementation of the code suggest by Andrew Appel in the section 11.4 "Graph Coloring Implementation" of his book. A lot of comments that are verbatim copies of his comments are left in the code. Unfortunately, the books have several nasty mistakes on the algorithm, they reported on his web page (see Section 5.2 [Modern Compiler Implementation], page 233); be sure to fix your books.

  Pay attention to `misc::set`: there is a lot of syntactic sugar provided to implement set operations. The code of `Color` can range from ugly and obfuscated to readable and very close to its specification.

`src/regalloc/regallocator.cc`
: Run the register allocation on each code fragment. Remove the useless moves.

`src/target/mips/epilogue.cc`
: If your compiler supports spills, implement `Codegen::rewrite_program`.

4.18.5 TC-9 FAQ

4.18.6 TC-9 Improvements

Possible improvements include:

4.19 TC-X, ia-32 Back End

TC-X is an optional assignment.

This section has been updated for EPITA-2015 on 2013-07-19.

At the end of this stage, the compiler produces IA-32 code (possibly with infinite registers). Basically, this stage is Section 4.16 [TC-7], page 168, with the IA-32 assembly language instead of MIPS.

The IA-32 architecture is the 32-bit Intel Architecture defined for the Intel 80306 (i386) processors, an extension of the original 16-bit 8086 (x86) architecture. IA-32 may also be referenced as x86, i386 and sometimes x86-32 or even x32, to distinguish it from the original 16-bit ("x86-16") or the 64-bit (x86-64 or x64) variants of the x86 family.

Relevant lecture notes include `instr-selection.pdf`[35].

4.19.1 TC-X Goals

Things to learn during this stage that you should remember:

CISC vs. RISC (again)
: MIPS (see Section 4.16 [TC-7], page 168) has shown you an example of RISC architecture. Targeting IA-32 shows you an example of the CISC family of processors.

[35] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/instr-selection.pdf`.

Compiler toolchain

At the end of the compiler (when register allocation is functional), the IA-32 back end generates code in IA-32 assembly language, which can be assembled and linked to produce a genuine executable program.

4.19.2 TC-X Samples

The goal of TC-X is straightforward: starting from LIR, generate the IA-32 instructions, except that you don't have actual registers: we still heavily use `Temp`s. Register allocation has been (or will be) done in a another stage, Section 4.18 [TC-9], page 189.

```
let
  var answer := 42
in
  answer := 51
end
```

File 4.88: `the-answer-ia32.tig`

```
$ tc --target-ia32 --inst-display the-answer-ia32.tig
/** Tiger final assembler ouput. */

/** Routine: _main */
        .text
        .globl  tc_main
        .type   tc_main,@function
tc_main:
# Allocate frame
        movl    %ebx, %t3
        movl    %edi, %t4
        movl    %esi, %t5
l0:
        movl    $42, %t1
        movl    %t1, (%ebp)
        movl    $51, %t2
        movl    %t2, (%ebp)
l1:
        movl    %t3, %ebx
        movl    %t4, %edi
        movl    %t5, %esi
# Deallocate frame
        ret     $0
l2:
        .size   tc_main,l2-tc_main
        .ident  "LRDE Tiger Compiler"
```

Example 4.133: *tc --target-ia32 --inst-display the-answer-ia32.tig*

At this stage the compiler cannot know what registers are used; the frame is not allocated. The final stage, register allocation, addresses this issue. For your information, it results in:

```
$ tc --target-ia32 -sI the-answer-ia32.tig
/** Tiger final assembler ouput. */
```

```
/** Routine: _main */
        .text
        .globl  tc_main
        .type   tc_main,@function
tc_main:
        pushl   %ebp
        subl    $4, %esp
        movl    %esp, %ebp
        subl    $4, %esp
l0:
        movl    $42, %ecx
        movl    %ecx, (%ebp)
        movl    $51, %ecx
        movl    %ecx, (%ebp)
l1:
        addl    $4, %ebp
        leave
        ret     $0
l2:
        .size   tc_main,l2-tc_main
        .ident  "LRDE Tiger Compiler"
```

Example 4.134: *tc --target-ia32 -sI the-answer-ia32.tig*

A delicate part of this exercise is handling the function calls:

```
let function add(x: int, y: int) : int = x + y
in
  print_int(add(1,(add(2, 3)))); print("\n")
end
```

File 4.89: `add-ia32.tig`

```
$ tc -e --target-ia32 --inst-display add-ia32.tig
/** Tiger final assembler ouput. */

/** Routine: add */
        .text
        .globl  tc_l0
        .type   tc_l0,@function
tc_l0:
# Allocate frame
        movl    12(%ebp), %t10
        movl    %t10, (%ebp)
        movl    16(%ebp), %t0
        movl    20(%ebp), %t1
        movl    %ebx, %t7
        movl    %edi, %t8
        movl    %esi, %t9
l2:
        movl    %t0, %t6
        addl    %t1, %t6
        movl    %t6, %eax
```

```
l3:
        movl    %t7, %ebx
        movl    %t8, %edi
        movl    %t9, %esi
# Deallocate frame
        ret     $12
l6:
        .size   tc_l0,l6-tc_l0

        .section        .rodata
l1:
        .long 1
        .asciz "\n"

/** Routine: _main */
        .text
        .globl  tc_main
        .type   tc_main,@function
tc_main:
# Allocate frame
        movl    %ebx, %t15
        movl    %edi, %t16
        movl    %esi, %t17
l4:
        movl    $3, %t11
        pushl   %t11
        movl    $2, %t12
        pushl   %t12
        pushl   %ebp
        call    tc_l0
        movl    %eax, %t4
        pushl   %t4
        movl    $1, %t13
        pushl   %t13
        pushl   %ebp
        call    tc_l0
        movl    %eax, %t5
        pushl   %t5
        call    tc_print_int
        lea     l1, %t14
        pushl   %t14
        call    tc_print
l5:
        movl    %t15, %ebx
        movl    %t16, %edi
        movl    %t17, %esi
# Deallocate frame
        ret     $0
l7:
        .size   tc_main,l7-tc_main
        .ident  "LRDE Tiger Compiler"
```

Example 4.135: *tc -e --target-ia32 --inst-display add-ia32.tig*

Once your compiler is complete, you can produce an actual IA-32 output, assemble it and link it with `gcc` to produce a real executable program:

```
$ tc -e --target-ia32 --asm-compute --inst-display add-ia32.tig
/** Tiger final assembler ouput. */

/** Routine: add */
        .text
        .globl  tc_l0
        .type   tc_l0,@function
tc_l0:
        pushl   %ebp
        subl    $4, %esp
        movl    %esp, %ebp
        subl    $4, %esp
        movl    12(%ebp), %ecx
        movl    %ecx, (%ebp)
        movl    16(%ebp), %eax
        movl    20(%ebp), %ecx
l2:
        addl    %ecx, %eax
l3:
        addl    $4, %ebp
        leave
        ret     $12
l6:
        .size   tc_l0,l6-tc_l0

        .section        .rodata
l1:
        .long 1
        .asciz "\n"

/** Routine: _main */
        .text
        .globl  tc_main
        .type   tc_main,@function
tc_main:
        pushl   %ebp
        subl    $4, %esp
        movl    %esp, %ebp
        subl    $0, %esp
l4:
        movl    $3, %ecx
        pushl   %ecx
        movl    $2, %ecx
        pushl   %ecx
        pushl   %ebp
        call    tc_l0
```

```
        pushl   %eax
        movl    $1, %ecx
        pushl   %ecx
        pushl   %ebp
        call    tc_l0
        pushl   %eax
        call    tc_print_int
        lea     l1, %ecx
        pushl   %ecx
        call    tc_print
l5:
        addl    $4, %ebp
        leave
        ret     $0
l7:
        .size   tc_main,l7-tc_main
        .ident  "LRDE Tiger Compiler"
```

Example 4.136: *tc -e --target-ia32 --asm-compute --inst-display add-ia32.tig*

```
$ tc -e --target-ia32 --asm-display add-ia32.tig >add-ia32.s
```

Example 4.137: *tc -e --target-ia32 --asm-display add-ia32.tig >add-ia32.s*

```
$ gcc -m32 -oadd-ia32 add-ia32.s
```

Example 4.138: *gcc -m32 -oadd-ia32 add-ia32.s*

```
$ ./add-ia32
6
```

Example 4.139: *./add-ia32*

The runtime must be functional. No difference must be observable in comparison with a run with HAVM:

```
substring("", 1, 1)
```

File 4.90: substring-0-1-1-ia32.tig

```
$ tc -e --target-ia32 --inst-display substring-0-1-1-ia32.tig
/** Tiger final assembler ouput. */

        .section        .rodata
l0:
        .long 0
        .asciz ""

/** Routine: _main */
        .text
        .globl  tc_main
        .type   tc_main,@function
tc_main:
# Allocate frame
```

```
        movl    %ebx, %t4
        movl    %edi, %t5
        movl    %esi, %t6
l1:
        movl    $1, %t1
        pushl   %t1
        movl    $1, %t2
        pushl   %t2
        lea     l0, %t3
        pushl   %t3
        call    tc_substring
l2:
        movl    %t4, %ebx
        movl    %t5, %edi
        movl    %t6, %esi
# Deallocate frame
        ret     $0
l3:
        .size   tc_main,l3-tc_main
        .ident  "LRDE Tiger Compiler"
```

Example 4.140: *tc -e --target-ia32 --inst-display substring-0-1-1-ia32.tig*

```
$ tc -e --target-ia32 --asm-compute --inst-display substring-0-1-1-ia32.tig
/** Tiger final assembler ouput. */

        .section        .rodata
l0:
        .long 0
        .asciz ""

/** Routine: _main */
        .text
        .globl  tc_main
        .type   tc_main,@function
tc_main:
        pushl   %ebp
        subl    $4, %esp
        movl    %esp, %ebp
        subl    $0, %esp
l1:
        movl    $1, %ecx
        pushl   %ecx
        movl    $1, %ecx
        pushl   %ecx
        lea     l0, %ecx
        pushl   %ecx
        call    tc_substring
l2:
        addl    $4, %ebp
        leave
        ret     $0
```

```
l3:
        .size   tc_main,l3-tc_main
        .ident  "LRDE Tiger Compiler"
```

Example 4.141: *tc -e --target-ia32 --asm-compute --inst-display substring-0-1-1-ia32.tig*

```
$ tc -e --target-ia32 --asm-display substring-0-1-1-ia32.tig >substring-0-1-1-ia32.s
```

Example 4.142: *tc -e --target-ia32 --asm-display substring-0-1-1-ia32.tig >substring-0-1-1-ia32.s*

```
$ gcc -m32 -osubstring-0-1-1-ia32 substring-0-1-1-ia32.s
```

Example 4.143: *gcc -m32 -osubstring-0-1-1-ia32 substring-0-1-1-ia32.s*

```
$ ./substring-0-1-1-ia32
error substring: arguments out of bounds
⇒120
```

Example 4.144: *./substring-0-1-1-ia32*

The following example illustrates conditional jumps.

```
if 42 > 51 then "forty-two" else "fifty-one"
```

File 4.91: `condjump-ia32.tig`

```
$ tc -e --target-ia32 --inst-display condjump-ia32.tig
/** Tiger final assembler ouput. */

        .section        .rodata
l0:
        .long 9
        .asciz "forty-two"

        .section        .rodata
l1:
        .long 9
        .asciz "fifty-one"

/** Routine: _main */
        .text
        .globl  tc_main
        .type   tc_main,@function
tc_main:
# Allocate frame
        movl    %ebx, %t4
        movl    %edi, %t5
        movl    %esi, %t6
l5:
        movl    $42, %t1
        cmp     $51, %t1
```

```
        jg      l2
l3:
        lea     l1, %t2
l4:
        jmp     l6
l2:
        lea     l0, %t3
        jmp     l4
l6:
        movl    %t4, %ebx
        movl    %t5, %edi
        movl    %t6, %esi
# Deallocate frame
        ret     $0
l7:
        .size   tc_main,l7-tc_main
        .ident  "LRDE Tiger Compiler"
```

Example 4.145: *tc -e --target-ia32 --inst-display condjump-ia32.tig*

```
$ tc -e --target-ia32 --asm-compute --inst-display condjump-ia32.tig
/** Tiger final assembler ouput. */

        .section        .rodata
l0:
        .long 9
        .asciz "forty-two"

        .section        .rodata
l1:
        .long 9
        .asciz "fifty-one"

/** Routine: _main */
        .text
        .globl  tc_main
        .type   tc_main,@function
tc_main:
        pushl   %ebp
        subl    $4, %esp
        movl    %esp, %ebp
        subl    $0, %esp
l5:
        movl    $42, %ecx
        cmp     $51, %ecx
        jg      l2
l3:
        lea     l1, %ecx
l4:
        jmp     l6
l2:
        lea     l0, %ecx
```

```
        jmp     l4
l6:
        addl    $4, %ebp
        leave
        ret     $0
l7:
        .size   tc_main,l7-tc_main
        .ident  "LRDE Tiger Compiler"
```

Example 4.146: *tc -e --target-ia32 --asm-compute --inst-display condjump-ia32.tig*

4.19.3 TC-X Given Code

Some code is provided along with the code given at TC-7 (see Section 4.16.3 [TC-7 Given Code], page 174). See Section 3.2.25 [src/target/ia32], page 67.

4.19.4 TC-X Code to Write

There is not much code to write:

- `Codegen` (`src/target/ia32/call.brg`, `src/target/ia32/move.brg`): complete some rules in the grammar of the code generator produced by MonoBURG.
- `GasAssembly::cjump_build` (`src/target/ia32/gas-assembly.cc`): translate conditional branch instructions (branch if equal, if lower than, etc.) into IA-32 assembly.

Information on IA-32 assembly instructions may be found in the Intel® 64 and IA-32 Architectures Software Developer Manuals[36] or in this much shorter IA32 Instruction List form[37]. The documentation of the GNU Assembler (GAS)[38] is also a recommended reading.

Completing the following routines is needed for register allocation only (see Section 4.18 [TC-9], page 189):

- `Codegen::rewrite_program` (`src/target/ia32/epilogue.cc`)

4.19.5 TC-X FAQ

4.19.6 TC-X Improvements

Possible improvements include:

- Support OS X Assembler The IA-32 back end supports only the ELF file format.

  OS X doesn't support ELF files, but has its own file format, Mach-O. Check out a discussion about the difference between OS X and Linux assembly[39].

  You can start by taking a look at the OS X Assembler reference[40].

---

[36] `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[37] `http://www.eti.pg.gda.pl/katedry/kask/pracownicy/Jaroslaw.Kuchta/AKO/IA32%20Instruction%20Set.pdf`.

[38] `http://sourceware.org/binutils/docs-2.23.1/as/`.

[39] `http://stackoverflow.com/questions/19720084/what-is-the-difference-between-assembly-on-mac-and-assembly-o 19725269#19725269`.

[40] `https://developer.apple.com/library/mac/documentation/DeveloperTools/Reference/Assembler/Assembler.pdf`.

4.20 TC-Y, arm Back End

TC-Y is an optional assignment.

This section has been updated for EPITA-2018 on 2015-10-28.

At the end of this stage, the compiler produces ARM code (possibly with infinite registers). Basically, this stage is Section 4.16 [TC-7], page 168, with the ARM assembly language instead of MIPS.

The ARM architecture is a family of RISC instruction set architectures for computer processors.

Relevant lecture notes include `instr-selection.pdf`[41].

4.20.1 TC-Y Goals

Things to learn during this stage that you should remember:

Discover ARMv7
: Discover the ARMv7 architecture and run programs on Raspberry Pi.

4.20.2 TC-Y Samples

The goal of TC-Y is straightforward: starting from LIR, generate the ARM instructions, except that you don't have actual registers: we still heavily use `Temp`s. Register allocation has been (or will be) done in a another stage, Section 4.18 [TC-9], page 189.

```
let
  var answer := 42
in
  answer := 51
end
```

File 4.92: `the-answer-arm.tig`

```
$ tc --target-arm --inst-display the-answer-arm.tig
# Tiger final assembler ouput.

# Routine: _main
.global tc_main
.text
tc_main:
# Allocate frame
        mov     t3, r10
        mov     t4, r4
        mov     t5, r5
        mov     t6, r6
        mov     t7, r7
        mov     t8, r8
        mov     t9, r9
l0:
        ldr     t1, =42
        str     t1, [fp, #0]
        ldr     t2, =51
        str     t2, [fp, #0]
l1:
```

[41] `https://www.lrde.epita.fr/~tiger//lecture-notes/slides/ccmp/instr-selection.pdf`.

```
        mov     r10, t3
        mov     r4, t4
        mov     r5, t5
        mov     r6, t6
        mov     r7, t7
        mov     r8, t8
        mov     r9, t9
# Deallocate frame
        pop     {fp, pc}

.ltorg
```

Example 4.147: *tc --target-arm --inst-display the-answer-arm.tig*

At this stage the compiler cannot know what registers are used; the frame is not allocated. The final stage, register allocation, addresses this issue. For your information, it results in:

```
$ tc --target-arm -sI the-answer-arm.tig
# Tiger final assembler ouput.

# Routine: _main
.global tc_main
.text
tc_main:
        push    {fp, lr}
        sub     fp, sp, #4
        sub     sp, sp, #4
l0:
        ldr     r1, =42
        str     r1, [fp, #0]
        ldr     r1, =51
        str     r1, [fp, #0]
l1:
        add     sp, sp, #4
        pop     {fp, pc}

.ltorg
```

Example 4.148: *tc --target-arm -sI the-answer-arm.tig*

```
let function add(x: int, y: int) : int = x + y
in
  print_int(add(1,(add(2, 3)))); print("\n")
end
```

File 4.93: `add-arm.tig`

```
$ tc -e --target-arm --inst-display add-arm.tig
# Tiger final assembler ouput.

# Routine: add
.global tc_l0
```

```
.text
tc_l0:
# Allocate frame
        str     r1, [fp, #0]
        mov     t0, r2
        mov     t1, r3
        mov     t7, r10
        mov     t8, r4
        mov     t9, r5
        mov     t10, r6
        mov     t11, r7
        mov     t12, r8
        mov     t13, r9
l2:
        add     t6, t0, t1
        mov     r0, t6
l3:
        mov     r10, t7
        mov     r4, t8
        mov     r5, t9
        mov     r6, t10
        mov     r7, t11
        mov     r8, t12
        mov     r9, t13
# Deallocate frame
        pop     {fp, pc}

.ltorg

.data
l1:
        .word 1
        .asciz "\n"

# Routine: _main
.global tc_main
.text
tc_main:
# Allocate frame
        mov     t18, r10
        mov     t19, r4
        mov     t20, r5
        mov     t21, r6
        mov     t22, r7
        mov     t23, r8
        mov     t24, r9
l4:
        mov     r1, fp
        ldr     t14, =2
        mov     r2, t14
        ldr     t15, =3
        mov     r3, t15
```

```
        bl      tc_l0
        mov     t4, r0
        mov     r1, fp
        ldr     t16, =1
        mov     r2, t16
        mov     r3, t4
        bl      tc_l0
        mov     t5, r0
        mov     r1, t5
        bl      tc_print_int
        ldr     t17, =l1
        mov     r1, t17
        bl      tc_print
l5:
        mov     r10, t18
        mov     r4, t19
        mov     r5, t20
        mov     r6, t21
        mov     r7, t22
        mov     r8, t23
        mov     r9, t24
# Deallocate frame
        pop     {fp, pc}

.ltorg
```

Example 4.149: *tc -e --target-arm --inst-display add-arm.tig*

The runtime must be functional. No difference must be observable in comparison with a run with HAVM:

```
substring("", 1, 1)
```

File 4.94: substring-0-1-1-arm.tig

```
$ tc -e --target-arm --inst-display substring-0-1-1-arm.tig
# Tiger final assembler ouput.

.data
l0:
        .word 0
        .asciz ""

# Routine: _main
.global tc_main
.text
tc_main:
# Allocate frame
        mov     t4, r10
        mov     t5, r4
        mov     t6, r5
        mov     t7, r6
        mov     t8, r7
```

```
        mov     t9, r8
        mov     t10, r9
l1:
        ldr     t1, =l0
        mov     r1, t1
        ldr     t2, =1
        mov     r2, t2
        ldr     t3, =1
        mov     r3, t3
        bl      tc_substring
l2:
        mov     r10, t4
        mov     r4, t5
        mov     r5, t6
        mov     r6, t7
        mov     r7, t8
        mov     r8, t9
        mov     r9, t10
# Deallocate frame
        pop     {fp, pc}

.ltorg
```

Example 4.150: *tc -e --target-arm --inst-display substring-0-1-1-arm.tig*

```
$ tc -e --target-arm --asm-compute --inst-display substring-0-1-1-arm.tig
# Tiger final assembler ouput.

.data
l0:
        .word 0
        .asciz ""

# Routine: _main
.global tc_main
.text
tc_main:
        push    {fp, lr}
        sub     fp, sp, #4
        sub     sp, sp, #0
l1:
        ldr     r1, =l0
        ldr     r2, =1
        ldr     r3, =1
        bl      tc_substring
l2:
        add     sp, sp, #0
        pop     {fp, pc}

.ltorg
```

Example 4.151: *tc -e --target-arm --asm-compute --inst-display substring-0-1-1-arm.tig*

The following example illustrates conditional jumps.

```
if 42 > 51 then "forty-two" else "fifty-one"
```

File 4.95: condjump-arm.tig

```
$ tc -e --target-arm --inst-display condjump-arm.tig
# Tiger final assembler ouput.

.data
l0:
        .word 9
        .asciz "forty-two"

.data
l1:
        .word 9
        .asciz "fifty-one"

# Routine: _main
.global tc_main
.text
tc_main:
# Allocate frame
        mov     t4, r10
        mov     t5, r4
        mov     t6, r5
        mov     t7, r6
        mov     t8, r7
        mov     t9, r8
        mov     t10, r9
l5:
        ldr     t1, =42
        cmp     t1, #51
        bgt     l2
l3:
        ldr     t2, =l1
l4:
        b       l6
l2:
        ldr     t3, =l0
        b       l4
l6:
        mov     r10, t4
        mov     r4, t5
        mov     r5, t6
        mov     r6, t7
        mov     r7, t8
        mov     r8, t9
```

```
        mov     r9, t10
# Deallocate frame
        pop     {fp, pc}

.ltorg
```

Example 4.152: *tc -e --target-arm --inst-display condjump-arm.tig*

```
$ tc -e --target-arm --asm-compute --inst-display condjump-arm.tig
# Tiger final assembler ouput.

.data
l0:
        .word 9
        .asciz "forty-two"

.data
l1:
        .word 9
        .asciz "fifty-one"

# Routine: _main
.global tc_main
.text
tc_main:
        push    {fp, lr}
        sub     fp, sp, #4
        sub     sp, sp, #0
l5:
        ldr     r1, =42
        cmp     r1, #51
        bgt     l2
l3:
        ldr     r1, =l1
l4:
        b       l6
l2:
        ldr     r1, =l0
        b       l4
l6:
        add     sp, sp, #0
        pop     {fp, pc}

.ltorg
```

Example 4.153: *tc -e --target-arm --asm-compute --inst-display condjump-arm.tig*

4.20.3 TC-Y Given Code

Some code is provided along with the code given at TC-7 (see Section 4.16.3 [TC-7 Given Code], page 174). See Section 3.2.26 [src/target/arm], page 68.

4.20.4 TC-Y Code to Write

There is not much code to write:

- `Codegen` (`src/target/arm/call.brg`, `src/target/arm/move.brg`): complete some rules in the grammar of the code generator produced by MonoBURG.
- `ArmAssembly::cjump_build` (`src/target/arm/arm-assembly.cc`): translate conditional branch instructions (branch if equal, if lower than, etc.) into ARM assembly.

  Information on ARM may be found in the ARM Architecture Reference Manual[42].

Completing the following routines is needed for register allocation only (see Section 4.18 [TC-9], page 189):

- `Codegen::rewrite_program` (`src/target/arm/epilogue.cc`)

4.20.5 TC-Y FAQ

How to compile and test?

To generate a binary from an ARM assembly file:

```
(print_int(42); print("\n"))
```

File 4.96: `print-int-arm.tig`

```
$ tc --target-arm -S print-int-arm.tig >print-int-arm.s
```

Example 4.154: *`tc --target-arm -S print-int-arm.tig >print-int-arm.s`*

```
$ arm-linux-gnueabihf-gcc-7 -march=armv7-a -oprint-int print-int-arm.s
```

Example 4.155: *`arm-linux-gnueabihf-gcc-7 -march=armv7-a -oprint-int print-int-arm.s`*

To run your code, use QEMU:

```
$ qemu-arm -L /usr/arm-linux-gnueabihf ./print-int
42
```

Example 4.156: *`qemu-arm -L /usr/arm-linux-gnueabihf ./print-int`*

QEMU (Quick Emulator) is a machine emulator and virtualizer. It can emulate a full system, including processor and peripherals. We are using it to emulate an ARM processor.

4.20.6 TC-Y Improvements

Possible improvements include:

- Take a quick look at the calling convention for both Section 4.19 [TC-X], page 195, and Section 4.20 [TC-Y], page 205, you might find some work to do.

4.21 TC-L, llvm ir

TC-L is an optional assignment.

This section has been updated for EPITA-2018 on 2015-10-06.

[42] `http://www.club.cc.cmu.edu/~mjrosenb/ARM%20v7%20Architecture%20Reference%20Manual.pdf`.

At the end of this stage, the compiler procudes LLVM IR code. This stage produces an intermediate representation like Section 4.14 [TC-5], page 132.

The LLVM IR is a Static Single Assignment (SSA) based representation, that provides type safety, low-level operations, and is capable of representing most of high-level languages cleanly. It is the intermediate representation used by Section 5.6 [Clang], page 250.

Compared to the HIR, LLVM IR is typed. Providing type information can help the LLVM back end to optimize even more.

You can find more information about the language in the LLVM Language Reference Manual[43].

For more documentation on LLVM, use the LLVM Documentation[44].

A relevant tutorial is available here: Kaleidoscope: Implementing a Language with LLVM[45]. It may be useful if you want to go further.

The dependency for this stage is Section 5.6 [Clang], page 250. You can install it either of a part of the `llvm-dev` package, or by visiting LLVM Download Page[46].

This stage makes use of multiple previous stages:

Section 4.6 [TC-R], page 106
: All the identifiers have to be unique, in order to translate them to LLVM identifiers (for debug purposes).

Section 4.9 [TC-D], page 116
: The desugar visitor is used to translate `for` loops and comparison between strings.

4.21.1 TC-L Goals

Things to learn during this stage that you should remember:

Smart pointers
: Usage of `std::unique_ptr`.

Move semantics
: How move semantics make `std::unique_ptr` a powerful tool.

Basic blocks
: Why do we need them, and how LLVM uses them in control-flow handling.

Inner functions and their impact on memory management at runtime
: Reaching non local variables.

Properties of LLVM IR
- SSA
- Type safe
- Target independent

The LLVM compiler infrastructure
- LLVM IR
- The optimizer, `opt`

  `opt` takes LLVM IR and applies optimization passes on it. This allows you to select several optimization passes to apply on the LLVM IR and observe the resulting LLVM IR.

[43] `http://llvm.org/docs/LangRef.html`.
[44] `http://llvm.org/docs/index.html`.
[45] `http://llvm.org/docs/tutorial/index.html`.
[46] `http://llvm.org/releases/download.html`.

- The LLVM Core libraries
- The LLVM Tools: `llvm-as`, `llvm-dis`, `llvm-link`, etc.

Using an external runtime
: Using a C runtime interacting with the Tiger code.

4.21.2 TC-L Samples

Starting from a typed AST, generate the LLVM IR instructions using the LLVM framework.

```
let
  var answer := 42
in
  answer := 51
end
```

File 4.97: `the-answer-llvm.tig`

```
$ tc --llvm-display the-answer-llvm.tig
; ModuleID = 'tc'
source_filename = "tc"
target triple = "i386-pc-linux-gnu"

; Function Attrs: inlinehint nounwind
declare void @tc_print(i8*) #0

; Function Attrs: inlinehint nounwind
declare void @tc_print_err(i8*) #0

; Function Attrs: inlinehint nounwind
declare void @tc_print_int(i32) #0

; Function Attrs: inlinehint nounwind
declare void @tc_flush() #0

; Function Attrs: inlinehint nounwind
declare i8* @tc_getchar() #0

; Function Attrs: inlinehint nounwind
declare i32 @tc_ord(i8*) #0

; Function Attrs: inlinehint nounwind
declare i8* @tc_chr(i32) #0

; Function Attrs: inlinehint nounwind
declare i32 @tc_size(i8*) #0

; Function Attrs: inlinehint nounwind
declare i32 @tc_streq(i8*, i8*) #0

; Function Attrs: inlinehint nounwind
declare i32 @tc_strcmp(i8*, i8*) #0

; Function Attrs: inlinehint nounwind
declare i8* @tc_substring(i8*, i32, i32) #0
```

```
; Function Attrs: inlinehint nounwind
declare i8* @tc_concat(i8*, i8*) #0

; Function Attrs: inlinehint nounwind
declare i32 @tc_not(i32) #0

; Function Attrs: inlinehint nounwind
declare void @tc_exit(i32) #0

; Function Attrs: nounwind
define void @tc_main() #1 {
entry__main:
  %answer_17 = alloca i32
  store i32 42, i32* %answer_17
  store i32 51, i32* %answer_17
  ret void
}

attributes #0 = { inlinehint nounwind }
attributes #1 = { nounwind }
```

Example 4.157: *tc --llvm-display the-answer-llvm.tig*

```
let function add(x: int, y: int) : int = x + y
in
  print_int(add(1,(add(2, 3)))); print("\n")
end
```

File 4.98: add-llvm.tig

```
$ tc --llvm-display add-llvm.tig
; ModuleID = 'tc'
source_filename = "tc"
target triple = "i386-pc-linux-gnu"

@string = private unnamed_addr constant [2 x i8] c"\0A\00"

; Function Attrs: inlinehint nounwind
declare void @tc_print(i8*) #0

; Function Attrs: inlinehint nounwind
declare void @tc_print_err(i8*) #0

; Function Attrs: inlinehint nounwind
declare void @tc_print_int(i32) #0

; Function Attrs: inlinehint nounwind
declare void @tc_flush() #0

; Function Attrs: inlinehint nounwind
declare i8* @tc_getchar() #0
```

```
; Function Attrs: inlinehint nounwind
declare i32 @tc_ord(i8*) #0

; Function Attrs: inlinehint nounwind
declare i8* @tc_chr(i32) #0

; Function Attrs: inlinehint nounwind
declare i32 @tc_size(i8*) #0

; Function Attrs: inlinehint nounwind
declare i32 @tc_streq(i8*, i8*) #0

; Function Attrs: inlinehint nounwind
declare i32 @tc_strcmp(i8*, i8*) #0

; Function Attrs: inlinehint nounwind
declare i8* @tc_substring(i8*, i32, i32) #0

; Function Attrs: inlinehint nounwind
declare i8* @tc_concat(i8*, i8*) #0

; Function Attrs: inlinehint nounwind
declare i32 @tc_not(i32) #0

; Function Attrs: inlinehint nounwind
declare void @tc_exit(i32) #0

; Function Attrs: nounwind
define void @tc_main() #1 {
entry__main:
  %call_add_19 = call i32 @add_19(i32 2, i32 3)
  %call_add_191 = call i32 @add_19(i32 1, i32 %call_add_19)
  call void @tc_print_int(i32 %call_add_191)
  call void @tc_print(i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string, i32 0
  ret void
}

; Function Attrs: nounwind
define internal i32 @add_19(i32 %x_17, i32 %y_18) #1 {
entry_add_19:
  %y_182 = alloca i32
  %x_171 = alloca i32
  store i32 %x_17, i32* %x_171
  store i32 %y_18, i32* %y_182
  %x_173 = load i32, i32* %x_171
  %y_184 = load i32, i32* %y_182
  %addtmp = add i32 %x_173, %y_184
  ret i32 %addtmp
}

attributes #0 = { inlinehint nounwind }
attributes #1 = { nounwind }
```

Example 4.158: *tc --llvm-display add-llvm.tig*

Once your compiler is complete, you can produce an actual LLVM IR output and compile it with `clang` to produce a real executable program.

```
$ tc --llvm-runtime-display --llvm-display add-llvm.tig
; ModuleID = 'tc'
source_filename = "tc"
target datalayout = "e-m:e-p:32:32-f64:32:64-f80:32-n8:16:32-S128"
target triple = "i386-pc-linux-gnu"

%struct._IO_FILE = type { i32, i8*, i8*, i8*, i8*, i8*, i8*, i8*, i8*, i8*, i8*, i8*
%struct._IO_marker = type { %struct._IO_marker*, %struct._IO_FILE*, i32 }

@string = private unnamed_addr constant [2 x i8] c"\0A\00"
@stderr = external global %struct._IO_FILE*, align 4
@.str = private unnamed_addr constant [29 x i8] c"chr: character out of range\0A\00"
@consts = internal global [512 x i8] zeroinitializer, align 1
@.str.1 = private unnamed_addr constant [36 x i8] c"substring: arguments out of boun
@stdin = external global %struct._IO_FILE*, align 4
@.str.2 = private unnamed_addr constant [1 x i8] zeroinitializer, align 1
@.str.3 = private unnamed_addr constant [3 x i8] c"%s\00", align 1
@.str.4 = private unnamed_addr constant [3 x i8] c"%d\00", align 1
@stdout = external global %struct._IO_FILE*, align 4

; Function Attrs: nounwind
define void @tc_main() #0 {
entry__main:
  %call_add_19 = call i32 @add_19(i32 2, i32 3)
  %call_add_191 = call i32 @add_19(i32 1, i32 %call_add_19)
  call void @tc_print_int(i32 %call_add_191)
  call void @tc_print(i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string, i32 0
  ret void
}

; Function Attrs: nounwind
define internal i32 @add_19(i32 %x_17, i32 %y_18) #0 {
entry_add_19:
  %y_182 = alloca i32
  %x_171 = alloca i32
  store i32 %x_17, i32* %x_171
  store i32 %y_18, i32* %y_182
  %x_173 = load i32, i32* %x_171
  %y_184 = load i32, i32* %y_182
  %addtmp = add i32 %x_173, %y_184
  ret i32 %addtmp
}

; Function Attrs: noinline nounwind optnone
define i32* @tc_init_array(i32, i32) #1 {
  %3 = alloca i32, align 4
```

```
  %4 = alloca i32, align 4
  %5 = alloca i32*, align 4
  %6 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %7 = load i32, i32* %3, align 4
  %8 = mul i32 %7, 4
  %9 = call noalias i8* @malloc(i32 %8) #0
  %10 = bitcast i8* %9 to i32*
  store i32* %10, i32** %5, align 4
  store i32 0, i32* %6, align 4
  br label %11

; <label>:11:                                     ; preds = %20, %2
  %12 = load i32, i32* %6, align 4
  %13 = load i32, i32* %3, align 4
  %14 = icmp ult i32 %12, %13
  br i1 %14, label %15, label %23

; <label>:15:                                     ; preds = %11
  %16 = load i32, i32* %4, align 4
  %17 = load i32*, i32** %5, align 4
  %18 = load i32, i32* %6, align 4
  %19 = getelementptr inbounds i32, i32* %17, i32 %18
  store i32 %16, i32* %19, align 4
  br label %20

; <label>:20:                                     ; preds = %15
  %21 = load i32, i32* %6, align 4
  %22 = add i32 %21, 1
  store i32 %22, i32* %6, align 4
  br label %11

; <label>:23:                                     ; preds = %11
  %24 = load i32*, i32** %5, align 4
  ret i32* %24
}

; Function Attrs: nounwind
declare noalias i8* @malloc(i32) #2

; Function Attrs: noinline nounwind optnone
define i32 @tc_not(i32) #1 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = icmp ne i32 %3, 0
  %5 = xor i1 %4, true
  %6 = zext i1 %5 to i32
  ret i32 %6
}
```

```
; Function Attrs: noinline nounwind optnone
define void @tc_exit(i32) #1 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  call void @exit(i32 %3) #6
  unreachable
                                                  ; No predecessors!
  ret void
}

; Function Attrs: noreturn nounwind
declare void @exit(i32) #3

; Function Attrs: noinline nounwind optnone
define i8* @tc_chr(i32) #1 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = icmp sle i32 0, %3
  br i1 %4, label %5, label %8

; <label>:5:                                      ; preds = %1
  %6 = load i32, i32* %2, align 4
  %7 = icmp sle i32 %6, 255
  br i1 %7, label %11, label %8

; <label>:8:                                      ; preds = %5, %1
  %9 = load %struct._IO_FILE*, %struct._IO_FILE** @stderr, align 4
  %10 = call i32 @fputs(i8* getelementptr inbounds ([29 x i8], [29 x i8]* @.str, i32
  call void @exit(i32 120) #6
  unreachable

; <label>:11:                                     ; preds = %5
  %12 = load i32, i32* %2, align 4
  %13 = mul nsw i32 %12, 2
  %14 = getelementptr inbounds i8, i8* getelementptr inbounds ([512 x i8], [512 x i8
sts, i32 0, i32 0), i32 %13
  ret i8* %14
}

declare i32 @fputs(i8*, %struct._IO_FILE*) #4

; Function Attrs: noinline nounwind optnone
define i8* @tc_concat(i8*, i8*) #1 {
  %3 = alloca i8*, align 4
  %4 = alloca i8*, align 4
  %5 = alloca i8*, align 4
  %6 = alloca i32, align 4
  %7 = alloca i32, align 4
  %8 = alloca i32, align 4
  %9 = alloca i32, align 4
```

```
  %10 = alloca i8*, align 4
  store i8* %0, i8** %4, align 4
  store i8* %1, i8** %5, align 4
  %11 = load i8*, i8** %4, align 4
  %12 = call i32 @strlen(i8* %11) #7
  store i32 %12, i32* %6, align 4
  %13 = load i8*, i8** %5, align 4
  %14 = call i32 @strlen(i8* %13) #7
  store i32 %14, i32* %7, align 4
  %15 = load i32, i32* %6, align 4
  %16 = icmp eq i32 %15, 0
  br i1 %16, label %17, label %19

; <label>:17:                                     ; preds = %2
  %18 = load i8*, i8** %5, align 4
  store i8* %18, i8** %3, align 4
  br label %69

; <label>:19:                                     ; preds = %2
  %20 = load i32, i32* %7, align 4
  %21 = icmp eq i32 %20, 0
  br i1 %21, label %22, label %24

; <label>:22:                                     ; preds = %19
  %23 = load i8*, i8** %4, align 4
  store i8* %23, i8** %3, align 4
  br label %69

; <label>:24:                                     ; preds = %19
  store i32 0, i32* %8, align 4
  %25 = load i32, i32* %6, align 4
  %26 = load i32, i32* %7, align 4
  %27 = add i32 %25, %26
  store i32 %27, i32* %9, align 4
  %28 = load i32, i32* %9, align 4
  %29 = add nsw i32 %28, 1
  %30 = call noalias i8* @malloc(i32 %29) #0
  store i8* %30, i8** %10, align 4
  store i32 0, i32* %8, align 4
  br label %31

; <label>:31:                                     ; preds = %43, %24
  %32 = load i32, i32* %8, align 4
  %33 = load i32, i32* %6, align 4
  %34 = icmp ult i32 %32, %33
  br i1 %34, label %35, label %46

; <label>:35:                                     ; preds = %31
  %36 = load i8*, i8** %4, align 4
  %37 = load i32, i32* %8, align 4
  %38 = getelementptr inbounds i8, i8* %36, i32 %37
  %39 = load i8, i8* %38, align 1
```

```
  %40 = load i8*, i8** %10, align 4
  %41 = load i32, i32* %8, align 4
  %42 = getelementptr inbounds i8, i8* %40, i32 %41
  store i8 %39, i8* %42, align 1
  br label %43

; <label>:43:                                     ; preds = %35
  %44 = load i32, i32* %8, align 4
  %45 = add nsw i32 %44, 1
  store i32 %45, i32* %8, align 4
  br label %31

; <label>:46:                                     ; preds = %31
  store i32 0, i32* %8, align 4
  br label %47

; <label>:47:                                     ; preds = %61, %46
  %48 = load i32, i32* %8, align 4
  %49 = load i32, i32* %7, align 4
  %50 = icmp ult i32 %48, %49
  br i1 %50, label %51, label %64

; <label>:51:                                     ; preds = %47
  %52 = load i8*, i8** %5, align 4
  %53 = load i32, i32* %8, align 4
  %54 = getelementptr inbounds i8, i8* %52, i32 %53
  %55 = load i8, i8* %54, align 1
  %56 = load i8*, i8** %10, align 4
  %57 = load i32, i32* %8, align 4
  %58 = load i32, i32* %6, align 4
  %59 = add i32 %57, %58
  %60 = getelementptr inbounds i8, i8* %56, i32 %59
  store i8 %55, i8* %60, align 1
  br label %61

; <label>:61:                                     ; preds = %51
  %62 = load i32, i32* %8, align 4
  %63 = add nsw i32 %62, 1
  store i32 %63, i32* %8, align 4
  br label %47

; <label>:64:                                     ; preds = %47
  %65 = load i8*, i8** %10, align 4
  %66 = load i32, i32* %9, align 4
  %67 = getelementptr inbounds i8, i8* %65, i32 %66
  store i8 0, i8* %67, align 1
  %68 = load i8*, i8** %10, align 4
  store i8* %68, i8** %3, align 4
  br label %69

; <label>:69:                                     ; preds = %64, %22, %17
  %70 = load i8*, i8** %3, align 4
```

```
  ret i8* %70
}

; Function Attrs: nounwind readonly
declare i32 @strlen(i8*) #5

; Function Attrs: noinline nounwind optnone
define i32 @tc_ord(i8*) #1 {
  %2 = alloca i32, align 4
  %3 = alloca i8*, align 4
  %4 = alloca i32, align 4
  store i8* %0, i8** %3, align 4
  %5 = load i8*, i8** %3, align 4
  %6 = call i32 @strlen(i8* %5) #7
  store i32 %6, i32* %4, align 4
  %7 = load i32, i32* %4, align 4
  %8 = icmp eq i32 %7, 0
  br i1 %8, label %9, label %10

; <label>:9:                                        ; preds = %1
  store i32 -1, i32* %2, align 4
  br label %15

; <label>:10:                                       ; preds = %1
  %11 = load i8*, i8** %3, align 4
  %12 = getelementptr inbounds i8, i8* %11, i32 0
  %13 = load i8, i8* %12, align 1
  %14 = sext i8 %13 to i32
  store i32 %14, i32* %2, align 4
  br label %15

; <label>:15:                                       ; preds = %10, %9
  %16 = load i32, i32* %2, align 4
  ret i32 %16
}

; Function Attrs: noinline nounwind optnone
define i32 @tc_size(i8*) #1 {
  %2 = alloca i8*, align 4
  store i8* %0, i8** %2, align 4
  %3 = load i8*, i8** %2, align 4
  %4 = call i32 @strlen(i8* %3) #7
  ret i32 %4
}

; Function Attrs: noinline nounwind optnone
define i8* @tc_substring(i8*, i32, i32) #1 {
  %4 = alloca i8*, align 4
  %5 = alloca i8*, align 4
  %6 = alloca i32, align 4
  %7 = alloca i32, align 4
  %8 = alloca i32, align 4
```

```
  %9 = alloca i8*, align 4
  %10 = alloca i32, align 4
  store i8* %0, i8** %5, align 4
  store i32 %1, i32* %6, align 4
  store i32 %2, i32* %7, align 4
  %11 = load i8*, i8** %5, align 4
  %12 = call i32 @strlen(i8* %11) #7
  store i32 %12, i32* %8, align 4
  %13 = load i32, i32* %6, align 4
  %14 = icmp sle i32 0, %13
  br i1 %14, label %15, label %24

; <label>:15:                                     ; preds = %3
  %16 = load i32, i32* %7, align 4
  %17 = icmp sle i32 0, %16
  br i1 %17, label %18, label %24

; <label>:18:                                     ; preds = %15
  %19 = load i32, i32* %6, align 4
  %20 = load i32, i32* %7, align 4
  %21 = add nsw i32 %19, %20
  %22 = load i32, i32* %8, align 4
  %23 = icmp ule i32 %21, %22
  br i1 %23, label %27, label %24

; <label>:24:                                     ; preds = %18, %15, %3
  %25 = load %struct._IO_FILE*, %struct._IO_FILE** @stderr, align 4
  %26 = call i32 @fputs(i8* getelementptr inbounds ([36 x i8], [36 x i8]* @.str.1, i
  call void @exit(i32 120) #6
  unreachable

; <label>:27:                                     ; preds = %18
  %28 = load i32, i32* %7, align 4
  %29 = icmp eq i32 %28, 1
  br i1 %29, label %30, label %38

; <label>:30:                                     ; preds = %27
  %31 = load i8*, i8** %5, align 4
  %32 = load i32, i32* %6, align 4
  %33 = getelementptr inbounds i8, i8* %31, i32 %32
  %34 = load i8, i8* %33, align 1
  %35 = sext i8 %34 to i32
  %36 = mul nsw i32 %35, 2
  %37 = getelementptr inbounds i8, i8* getelementptr inbounds ([512 x i8], [512 x i8
sts, i32 0, i32 0), i32 %36
  store i8* %37, i8** %4, align 4
  br label %64

; <label>:38:                                     ; preds = %27
  %39 = load i32, i32* %7, align 4
  %40 = add nsw i32 %39, 1
  %41 = call noalias i8* @malloc(i32 %40) #0
```

```
  store i8* %41, i8** %9, align 4
  store i32 0, i32* %10, align 4
  br label %42

; <label>:42:                                     ; preds = %56, %38
  %43 = load i32, i32* %10, align 4
  %44 = load i32, i32* %7, align 4
  %45 = icmp slt i32 %43, %44
  br i1 %45, label %46, label %59

; <label>:46:                                     ; preds = %42
  %47 = load i8*, i8** %5, align 4
  %48 = load i32, i32* %6, align 4
  %49 = load i32, i32* %10, align 4
  %50 = add nsw i32 %48, %49
  %51 = getelementptr inbounds i8, i8* %47, i32 %50
  %52 = load i8, i8* %51, align 1
  %53 = load i8*, i8** %9, align 4
  %54 = load i32, i32* %10, align 4
  %55 = getelementptr inbounds i8, i8* %53, i32 %54
  store i8 %52, i8* %55, align 1
  br label %56

; <label>:56:                                     ; preds = %46
  %57 = load i32, i32* %10, align 4
  %58 = add nsw i32 %57, 1
  store i32 %58, i32* %10, align 4
  br label %42

; <label>:59:                                     ; preds = %42
  %60 = load i8*, i8** %9, align 4
  %61 = load i32, i32* %7, align 4
  %62 = getelementptr inbounds i8, i8* %60, i32 %61
  store i8 0, i8* %62, align 1
  %63 = load i8*, i8** %9, align 4
  store i8* %63, i8** %4, align 4
  br label %64

; <label>:64:                                     ; preds = %59, %30
  %65 = load i8*, i8** %4, align 4
  ret i8* %65
}

; Function Attrs: noinline nounwind optnone
define i32 @tc_strcmp(i8*, i8*) #1 {
  %3 = alloca i8*, align 4
  %4 = alloca i8*, align 4
  store i8* %0, i8** %3, align 4
  store i8* %1, i8** %4, align 4
  %5 = load i8*, i8** %3, align 4
  %6 = load i8*, i8** %4, align 4
  %7 = call i32 @strcmp(i8* %5, i8* %6) #7
```

```
  ret i32 %7
}

; Function Attrs: nounwind readonly
declare i32 @strcmp(i8*, i8*) #5

; Function Attrs: noinline nounwind optnone
define i32 @tc_streq(i8*, i8*) #1 {
  %3 = alloca i8*, align 4
  %4 = alloca i8*, align 4
  store i8* %0, i8** %3, align 4
  store i8* %1, i8** %4, align 4
  %5 = load i8*, i8** %3, align 4
  %6 = load i8*, i8** %4, align 4
  %7 = call i32 @strcmp(i8* %5, i8* %6) #7
  %8 = icmp eq i32 %7, 0
  %9 = zext i1 %8 to i32
  ret i32 %9
}

; Function Attrs: noinline nounwind optnone
define i8* @tc_getchar() #1 {
  %1 = alloca i8*, align 4
  %2 = alloca i32, align 4
  %3 = load %struct._IO_FILE*, %struct._IO_FILE** @stdin, align 4
  %4 = call i32 @_IO_getc(%struct._IO_FILE* %3)
  store i32 %4, i32* %2, align 4
  %5 = load i32, i32* %2, align 4
  %6 = icmp eq i32 %5, -1
  br i1 %6, label %7, label %8

; <label>:7:                                        ; preds = %0
  store i8* getelementptr inbounds ([1 x i8], [1 x i8]* @.str.2, i32 0, i32 0), i8**
  br label %12

; <label>:8:                                        ; preds = %0
  %9 = load i32, i32* %2, align 4
  %10 = mul nsw i32 %9, 2
  %11 = getelementptr inbounds i8, i8* getelementptr inbounds ([512 x i8], [512 x i8
sts, i32 0, i32 0), i32 %10
  store i8* %11, i8** %1, align 4
  br label %12

; <label>:12:                                       ; preds = %8, %7
  %13 = load i8*, i8** %1, align 4
  ret i8* %13
}

declare i32 @_IO_getc(%struct._IO_FILE*) #4

; Function Attrs: noinline nounwind optnone
define void @tc_print(i8*) #1 {
```

```
  %2 = alloca i8*, align 4
  store i8* %0, i8** %2, align 4
  %3 = load i8*, i8** %2, align 4
  %4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @
  ret void
}

declare i32 @printf(i8*, ...) #4

; Function Attrs: noinline nounwind optnone
define void @tc_print_err(i8*) #1 {
  %2 = alloca i8*, align 4
  store i8* %0, i8** %2, align 4
  %3 = load %struct._IO_FILE*, %struct._IO_FILE** @stderr, align 4
  %4 = load i8*, i8** %2, align 4
  %5 = call i32 (%struct._IO_FILE*, i8*, ...) @fprintf(%struct._IO_FILE* %3, i8* get
mentptr inbounds ([3 x i8], [3 x i8]* @.str.3, i32 0, i32 0), i8* %4)
  ret void
}

declare i32 @fprintf(%struct._IO_FILE*, i8*, ...) #4

; Function Attrs: noinline nounwind optnone
define void @tc_print_int(i32) #1 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @
  ret void
}

; Function Attrs: noinline nounwind optnone
define void @tc_flush() #1 {
  %1 = load %struct._IO_FILE*, %struct._IO_FILE** @stdout, align 4
  %2 = call i32 @fflush(%struct._IO_FILE* %1)
  ret void
}

declare i32 @fflush(%struct._IO_FILE*) #4

; Function Attrs: noinline nounwind optnone
define i32 @main() #1 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  br label %3

; <label>:3:                                      ; preds = %15, %0
  %4 = load i32, i32* %2, align 4
  %5 = icmp slt i32 %4, 512
  br i1 %5, label %6, label %18
```

```
; <label>:6:                                     ; preds = %3
  %7 = load i32, i32* %2, align 4
  %8 = sdiv i32 %7, 2
  %9 = trunc i32 %8 to i8
  %10 = load i32, i32* %2, align 4
  %11 = getelementptr inbounds [512 x i8], [512 x i8]* @consts, i32 0, i32 %10
  store i8 %9, i8* %11, align 1
  %12 = load i32, i32* %2, align 4
  %13 = add nsw i32 %12, 1
  %14 = getelementptr inbounds [512 x i8], [512 x i8]* @consts, i32 0, i32 %13
  store i8 0, i8* %14, align 1
  br label %15

; <label>:15:                                    ; preds = %6
  %16 = load i32, i32* %2, align 4
  %17 = add nsw i32 %16, 2
  store i32 %17, i32* %2, align 4
  br label %3

; <label>:18:                                    ; preds = %3
  call void bitcast (void ()* @tc_main to void (i32)*)(i32 0)
  ret i32 0
}

attributes #0 = { nounwind }
attributes #1 = { noinline nounwind optnone "correctly-rounded-divide-
sqrt-fp-math"="false" "disable-tail-calls"="false" "less-precise-fpmad"="false" "no-
frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-
math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-
zeros-fp-math"="false" "no-trapping-math"="false" "stack-protector-buffer-
size"="8" "target-cpu"="pentium4" "target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "un
fp-math"="false" "use-soft-float"="false" }
attributes #2 = { nounwind "correctly-rounded-divide-sqrt-fp-math"="false" "disable-
tail-calls"="false" "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-
frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "no
signed-zeros-fp-math"="false" "no-trapping-math"="false" "stack-protector-
buffer-size"="8" "target-cpu"="pentium4" "target-features"="+fxsr,+mmx,+sse,+sse2,+x
fp-math"="false" "use-soft-float"="false" }
attributes #3 = { noreturn nounwind "correctly-rounded-divide-sqrt-fp-
math"="false" "disable-tail-calls"="false" "less-precise-fpmad"="false" "no-
frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-
math"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-
trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="pentium4" "ta
features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-
float"="false" }
attributes #4 = { "correctly-rounded-divide-sqrt-fp-math"="false" "disable-
tail-calls"="false" "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-
frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "no
signed-zeros-fp-math"="false" "no-trapping-math"="false" "stack-protector-
buffer-size"="8" "target-cpu"="pentium4" "target-features"="+fxsr,+mmx,+sse,+sse2,+x
fp-math"="false" "use-soft-float"="false" }
```

```
attributes #5 = { nounwind readonly "correctly-rounded-divide-sqrt-fp-
math"="false" "disable-tail-calls"="false" "less-precise-fpmad"="false" "no-
frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-
math"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-
trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="pentium4" "ta
features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-
float"="false" }
attributes #6 = { noreturn nounwind }
attributes #7 = { nounwind readonly }

!llvm.ident = !{!0}
!llvm.module.flags = !{!1, !2}

!0 = !{!"clang version 5.0.1-2 (tags/RELEASE_501/final)"}
!1 = !{i32 1, !"NumRegisterParameters", i32 0}
!2 = !{i32 1, !"wchar_size", i32 4}
```

Example 4.159: *tc --llvm-runtime-display --llvm-display add-llvm.tig*

```
$ tc --llvm-runtime-display --llvm-display add-llvm.tig >add-llvm.ll
```

Example 4.160: *tc --llvm-runtime-display --llvm-display add-llvm.tig >add-llvm.ll*

```
$ clang -m32 -oadd-llvm add-llvm.ll
```

Example 4.161: *clang -m32 -oadd-llvm add-llvm.ll*

```
$ ./add-llvm
6
```

Example 4.162: *./add-llvm*

4.21.3 TC-L Given Code

Some code is provided along with the code given at TC-5 See Section 3.2.28 [src/llvmtranslate], page 69.

4.21.4 TC-L Code to Write

`src/llvmtranslate/escapes-collector.cc`

- `build_frame` Collect all the local variables used in a function. Used in the escape collector.
- `collect_escapes` Collect escapes for every function in the ast, and store them in a map. This is used for Lambda Lifting

Both functions are based on an internal visitor.

`src/llvmtranslate/translator.*`

This is where all the translation logic goes.

The translation to LLVM IR is using the `llvm::IRBuilder`.

`src/llvmtranslate/llvm-type-visitor.*`

Translate `type::Type` objects into `llvm::Type` objects.

4.21.5 TC-L FAQ

The build failed on my machine

If the following error occurs:

```
  CXXLD    src/tc
src/.libs/libtc.a(lt14-translator.o):(.rodata._ZTIN4llvm17GetElementPtrIns
src/.libs/libtc.a(lt14-translator.o):(.rodata._ZTIN4llvm8ICmpInstE[_ZTIN4l
src/.libs/libtc.a(lt14-translator.o):(.rodata._ZTIN4llvm7PHINodeE[_ZTIN4ll
collect2: error: ld returned 1 exit status
Makefile:2992: recipe for target 'src/tc' failed
make: *** [src/tc] Error 1
```

then you are using an old version of LLVM. The version required is 3.8 or more.

If you still want to use LLVM 3.7, then the LLVM build you are using is compiled without RTTI.

In order to make it work, you have two choices:

- If you are building LLVM 3.7 from the source code, apply this[47] patch to fix the compilation.
- Build LLVM with RTTI enabled. In order to build LLVM with RTTI enabled, follow these steps, assuming the current directory is the root of LLVM:
  - `mkdir _build`
  - `cd _build`
  - `cmake .. -DLLVM_REQUIRES_RTTI=ON -DCMAKE_BUILD_TYPE=Release`
  - `make install`

LLVM builds with RTTI disabled by default. They use their own RTTI-like system. Tiger is compiled using RTTI, and actually uses it quite a lot (`dynamic_cast`). In order to make them work together, LLVM has to emit the `vtables` of its classes in their own translation unit.

This regression appeared in LLVM 3.7 when a virtual destructor was inlined, so the `vtables` were emitted in every translation unit. It was the following classes: `llvm::GetElementPtrInst`, `llvm::ICmpInst` and `llvm::PHINode`.

In order to solve the problem, LLVM uses a dedicated member function called `anchor`, that is going to force the emission to happen in its own translation unit.

As of today, here are some packages of LLVM 3.7 that work/don't work:

- ArchLinux (pacman) - RTTI enabled.
- OS X (brew) - RTTI enabled.
- OS X (macports) - RTTI disabled. Does not compile.
- Ubuntu (apt) - RTTI enabled.

What is a PHI node?

LLVM instructions are represented in the SSA (Static Single Assignment) form.

Let's take an example:

```
let
  var v := 10
  var a := 1
```

[47] `https://github.com/llvm-mirror/llvm/commit/fecd8a332e245e2535faf04de564bba8f7e068d8.patch`.

```
  var b := 0
in
  if (v < 10) then
    a := 2;
  b := a
end
```

The whole point of SSA is to forbid re-assignments, so we cannot assign `2` to `a`.

In that case, LLVM is going to create two `a`'s, and the assignment has to pick the desired version.

Using a PHI node, the assignment will depend on the original path of the code, and using that information, it can decide which version of `a` should be picked.

You can use the `opt` tool in order to display the control-flow graph.

```
opt -dot-cfg fact.ll
```

This generates two files: `cfg.tc_main.dot` and `cfg.fact_18.dot`, corresponding to the `main` function and the `fact` function.

I don't understand all the acronyms used in LLVM.

Where can I find their meaning? You can find it in The LLVM Lexicon[48].

Can I output the LLVM IR of a C/C++ program?

Yes, you can. Section 5.6 [Clang], page 250, allows you to do it using the flags `-S -emit-llvm`.

```
int main(void)
{
  int a = 1 + 2 * 3;
  return a;
}
```

File 4.99: `clang-example.c`

```
$ clang -m32 -S -emit-llvm -o - clang-example.c
; ModuleID = 'clang-example.c'
source_filename = "clang-example.c"
target datalayout = "e-m:e-p:32:32-f64:32:64-f80:32-n8:16:32-
S128"
target triple = "i386-pc-linux-gnu"

; Function Attrs: noinline nounwind
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 7, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  ret i32 %3
}

attributes #0 = { noinline nounwind "correctly-rounded-divide-
sqrt-fp-math"="false" "disable-tail-calls"="false" "less-precise-
```

[48] `http://llvm.org/docs/Lexicon.html`.

```
fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-
elim-non-leaf" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-
nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-
trapping-math"="false" "stack-protector-buffer-size"="8" "target-
cpu"="pentium4" "target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-
fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"NumRegisterParameters", i32 0}
!1 = !{!"clang version 4.0.1-8 (tags/RELEASE_401/final)"}
```

Example 4.163: *clang -m32 -S -emit-llvm -o - clang-example.c*

Your compiler crashes when `llvm::Linker::linkModules` is called

When using `--llvm-runtime-display`, this behavior can occur when the linker is asked to link two LLVM IR modules that may have been compiled with two different LLVM IR versions.

Since the runtime is compiled with Section 5.6 [Clang], page 250, from C to LLVM IR, you have to make sure that the Section 5.6 [Clang], page 250, version and the LLVM version are exactly the same.

This crash currently occurs with Section 5.6 [Clang], page 250, 3.6 and LLVM 3.8.

4.21.6 TC-L Improvements

Possible improvements include:

Debug information

LLVM has support for debug information. If you want to generate debug information, have a look at Adding Debug Information[49].

LLVM generates DWARF code.

Start by emitting the locations of your nodes first, then go further with scopes and variables.

Global variables

As you noticed at Section 4.14 [TC-5], page 132, you can't have global variables. LLVM has support for global variables, using the `GlobalVariable` class.

[49] `http://llvm.org/docs/tutorial/LangImpl8.html`.

5 Tools

This chapter aims at providing some helpful information about the various tools that you are likely to use to implement `tc`. It does not replace the reading of the genuine documentation, nevertheless, helpful tips are given. Feel free to contribute additional information.

5.1 Programming Environment

This section lists the tools you need to work in good conditions.

Tool	Version	Comment
GCC	5.0	See Section 5.5 [GCC], page 249.
Clang	3.8	Optional for TC < 5: See Section 5.6 [Clang], page 250.
Autoconf	2.64	See Section 5.4 [The GNU Build System], page 247.
Automake	1.14.1	See Section 5.4 [The GNU Build System], page 247.
Libtool	2.2.6	See Section 5.4 [The GNU Build System], page 247.
GNU Make	3.81	
Boost	1.53	TC >= 5, See [Boost.org], page 237.
Doxygen	1.5.1	See Section 5.16 [Doxygen], page 255.
Python	2.5	See Section 5.15 [Python], page 254.
SWIG	2.0	Optional: See Section 5.14 [SWIG], page 254.
Flex	2.5.35	See Section 5.9 [Flex & Bison], page 251.
Bison	3.0.4.19-fbaf	See Section 5.9 [Flex & Bison], page 251.
HAVM	0.27	TC >= 5, See Section 5.10 [HAVM], page 252.
MonoBURG	1.0.6a	TC >= 7, See Section 5.11 [MonoBURG], page 252.
Nolimips	0.10	TC >= 7, See Section 5.12 [Nolimips], page 253.
GDB	6.6	See Section 5.7 [GDB], page 250.
Valgrind	3.6	See Section 5.8 [Valgrind], page 250.
Git	1.7	
GraphViz	2.26.3	Optional: display DOT graphs.

5.2 Modern Compiler Implementation

The Tiger Bible exists in two profoundly different versions.

5.2.1 First Editions

The single most important tool for implementing the Tiger Project is the original book, Modern Compiler Implementation in C/Java/ML[1], by Andrew W. Appel[2], published by Cambridge University Press (New York, Cambridge). ISBN 0-521-58388-8/.

It is not possible to finish this project without having at least one copy per group. We provide a convenient mini Tiger Compiler Reference Manual[3] that contains some information about the language but it does not cover all the details, and sometimes digging into the original book is required. This is on purpose, by virtue of due respect to the author of this valuable book.

Several copies are available at the EPITA library.

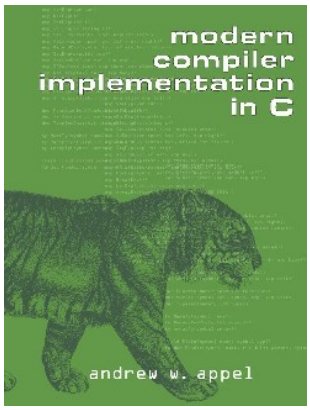
[1] `http://www.cs.princeton.edu/~appel/modern/`.

[2] `http://www.cs.princeton.edu/~appel/`.

[3] `https://www.lrde.epita.fr/~tiger/tiger.html`.

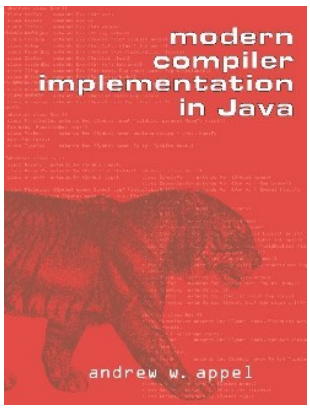
There are three flavors of this book:

C



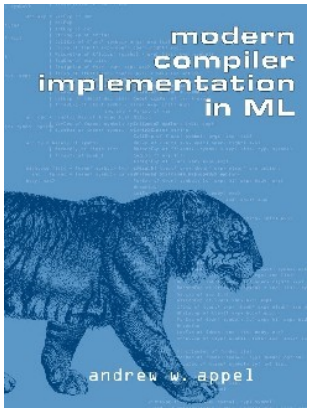
The code samples are written in C. Avoid this edition, as C is not appropriate to describe the elaborate algorithms involved: most of the time, the simple ideas are destroyed with longuish unpleasant lines of code.

Java, First edition



The samples are written in Java. This book is the closest to the EPITA Tiger Project, since it is written in an object oriented language. Nevertheless, the modelisation is very poor, and therefore, don't be surprised if the EPITA project is significantly different. For a start, there is no Visitors at all. Of course the main purpose of the book is compilers, but it is not a reason for such a poor modelisation.

ML



This book, which is the "original", provides code samples in ML, which is a very adequate language to write compilers. Therefore it is very readable, even if you are not fluent in ML. We recommend this edition, unless you have severe problems with functional programming.

This book addresses many more issues than the sole Tiger Project as we implement it. In other words, it is an extremely interesting book whose provides insights on garbage collection, object oriented and functional languages etc.

There is a dozen copies at the EPITA library, but buying it is a good idea.

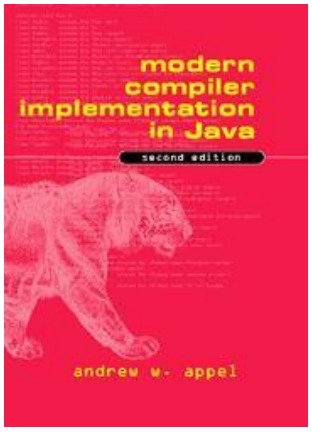
Pay extra attention: there are several errors in the books, some of which are reported on Andrew Appel's pages (C[4] Java[5], and ML[6]), and others are not.

Because these pages no longer seem to be maintained, additional errors are reported below. "p. C.245" means page 245 in the C book. Please send us additions.

11.3. Example with Precolored Nodes (p. C.245)
The first interference graph presented for this example lacks the interference between `r1` and `c`.

11.4 Graph Coloring Implementation (p. C.248)
In the first sentence, s/inteference/interference/.

5.2.2 In Java - Second Edition



The Second Edition of Modern Compiler Implementation in Java[7], by Andrew W. Appel[8] and Jens Palsberg[9], published by Cambridge University Press (New York, Cambridge), ISBN 052182060X, is a very different book from the rest of the series.

While, finally, the design *is* much better, starting with the introduction of the Visitors, there are many shortcoming for us:

– The language is no longer Tiger, in spite of the cover, but MiniJava, a subset of Java. It should be noted that, although dressed in OO fashion, the core language addressed in the first part of the book is no more OO than Tiger. Just as in the first edition, OO is addressed in Chapter 14 (a good thing IMHO).

[4] `http://www.cs.princeton.edu/~appel/modern/c/errata.html`.
[5] `http://www.cs.princeton.edu/~appel/modern/java/errata.html`.
[6] `http://www.cs.princeton.edu/~appel/modern/ml/errata.html`.
[7] `www.cambridge.org/gb/knowledge/isbn/item1170327/`.
[8] `http://www.cs.princeton.edu/~appel/`.
[9] `http://www.cs.ucla.edu/~palsberg/`.

- This language seems, at first sight, to have a simpler syntax. In particular, it does not include the "l-value vs. array instantiation" ambiguity, which is a pity, since that's a nice grammar massage exercise.
- The appendix no longer contains the Tiger Language Reference Manual, but the MiniJava Language Reference Manual. This is a real problem for EPITA students who have to produce a compiler for Tiger. This is why our Section "Tiger Language Reference Manual" in *Tiger Compiler Reference Manual* is now much more detailed: so that students can buy the recent version of this book, and still have an access to the definition of the Tiger language.
- MiniJava, as Java, does not need static links. Although this book does mention static links (and uses an example in... Tiger!), it contains much less material than the original edition. This is unfortunate: try to find another version of the book.
- Sometimes the sentence are convoluted because... it would be nice to illustrate using Tiger... For instance page 151 "Record and Array Creation" begins with "Imagine a language construct {e1, e2, ..., en} that creates an n-element record...".

Nevertheless, because we don't encourage book copying, we now provide a complete definition of the Tiger language in Section "Tiger Language Reference Manual" in *Tiger Compiler Reference Manual*.

5.3 Bibliography

Below is presented a selection of books, papers and web sites that are pertinent to the Tiger project. Of course, you are not requested to read them all, except Section 5.2 [Modern Compiler Implementation], page 233. A suggested ordered small selection of books is:

1. Section 5.2 [Modern Compiler Implementation], page 233,
2. [C++ Primer], page 238,
3. [Design Patterns - Elements of Reusable Object-Oriented Software], page 240,
4. [Effective Modern C++], page 240,
5. [Effective C++], page 241,
6. [Effective STL], page 241,

The books are available at the EPITA Library: you are encouraged to borrow them there. If some of these books are missing, please suggest them to the library's manager. To buy these books, we recommend Le Monde en "tique"[10], a bookshop that has demonstrated several times its dedication to its job, and its kindness to EPITA students/members.

`Autotools Tutorial` – *Alexandre Duret-Lutz* [Web Site]

The Autotools Tutorial[11] is the best introduction to Autoconf, Automake, and Libtool, that we know. It covers also other components of the GNU Build System. You should read this before diving into the documentation.

Other resources include:

- the Autoconf documentation[12]
- the Automake documentation[13]
- the Libtool documentation[14]

[10] `http://www.lmet.fr`.

[11] `http://www.lrde.epita.fr/~adl/autotools.html`.

[12] `http://www.gnu.org/software/autoconf/manual/index.html`.

[13] `http://www.gnu.org/software/automake/manual/index.html`.

[14] `http://www.gnu.org/software/libtool/manual/index.html`.

– the Goat Book[15] covers the whole GNU Build System: Autoconf, Automake and Libtool.

`Bjarne Stroustrup` [Web Site]

Bjarne Stroustrup[16] is the author of C++, which he describes as (The C++ Programming Language[17]):

> C++ is a general purpose programming language with a bias towards systems programming that
>
> – is a better C
> – supports data abstraction
> – supports object-oriented programming
> – supports generic programming.

His web page contains interesting material on C++, including many interviews. The interview by Aleksey V. Dolya for the Linux Journal[18] contains thoughts about C and C++. For instance:

> I think that the current mess of C/C++ incompatibilities is a most unfortunate accident of history, without a fundamental technical or philosophical basis. Ideally the languages should be merged, and I think that a merger is barely technically possible by making convergent changes to both languages. It seems, however, that because there is an unwillingness to make changes it is likely that the languages will continue to drift apart–to the detriment of almost every C and C++ programmer. [...] However, there are entrenched interests keeping convergence from happening, and I'm not seeing much interest in actually doing anything from the majority that, in my opinion, would benefit most from compatibility.

His list of C++ Applications[19] is worth the browsing.

`Boost.org` [Web Site]

The Boost.org web site[20] reads:

> The Boost web site provides free peer-reviewed portable C++ source libraries. The emphasis is on libraries that work well with the C++ Standard Library. One goal is to establish "existing practice" and provide reference implementations so that the Boost libraries are suitable for eventual standardization. Some of the libraries have already been proposed for inclusion in the C++ Standards Committee's upcoming C++ Standard Library Technical Report.

[15] `http://www.sourceware.org/autobook/`.
[16] `http://www.stroustrup.com`.
[17] `http://www.stroustrup.com/C++.html`.
[18] `http://www.linuxjournal.com/article/7099`.
[19] `http://www.stroustrup.com/applications.html`.
[20] `http://www.boost.org`.

In addition to actual code, a lot of good documentation is available. Amongst libraries, you ought to have a look at the Spirit object-oriented recursive-descent parser generator framework[21], the Boost Graph Library[22], the Boost Variant Library[23] etc.

BURG: `Fast Optimal Instruction Selection and Tree Parsing` [Paper]
– *Christopher W. Fraser, Robert R. Henry, Todd A. Proebsting*
SIGPLAN Notices 24(4), 68-76. 1992.

This paper[24] is a description of BURG and an introduction to the concept of code generator generators.

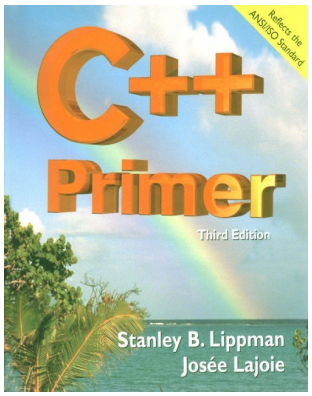
`Compilers and Compiler Generators, an introduction with C++` [Book]
– *P.D. Terry*

Its site reads:

> This site[25] provides an on-line edition of the text and other material from the book "Compilers and Compiler Generators - an introduction with C++", published in 1997 by International Thomson Computer Press. The original edition is now out of print, and the copyright has reverted to the author.

This book is not very interesting for us: it depends upon tools we don't use, its C++ is antique, and its approach to compilation is significantly different from Appel's.

`C++ Primer` – *Stanley B. Lippman, Josée Lajoie* [Book]



Published by Addison-Wesley; ISBN 0-201-82470-1.

This book teaches C++ for programmers. It is quite extensive and easy to read. Unfortunately it is not 100% standard compliant, in particular many `std::` are missing. Weirdly enough, the authors seems to promote `using` declarations instead of explicit qualifiers; the page 441 reads:

> In this book, to keep the code examples Short, and because many of the examples were compiled with implementations not supporting `namespace`, we have not explicitly listed the `using` declarations needed to properly compile the examples. It is assumed that `using` declarations are provided for the members of namespace `std` used in the code examples.

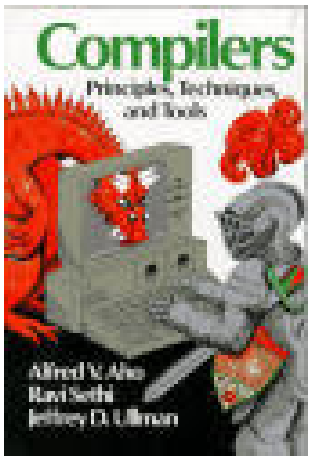
It should not be too much of a problem though. This is the book we recommend to learn C++. See the Addison-Wesley C++ Primer Page[26].

[21] `http://www.boost.org/libs/spirit/index.html`.
[22] `http://www.boost.org/libs/graph/doc/table_of_contents.html`.
[23] `http://www.boost.org/libs/variant/index.html`.
[24] `http://www-inst.eecs.berkeley.edu/~graham/papers/burg-doc.ps`.
[25] `http://www.scifac.ru.ac.za/compilers/`.
[26] `http://www.informit.com/store/c-plus-plus-primer-9780201824704`.

Warning: The French translation is *L'Essentiel du C++*, which is extremely stupid since *Essential C++* is another book from Stanley B. Lippman (but not with Josée Lajoie).

`Compilers: Principles, Techniques and Tools` – *Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman* [Book]

`The Dragon Book` [Book]



Published by Addison-Wesley 1986; ISBN 0-201-10088-6.

This book is *the* bible in compiler design. It has extensive insight on the whole architecture of compilers, provides a rigorous treatment for theoretical material etc. Nevertheless I (Akim) would not recommend this book to EPITA students, because

it is getting old
: It doesn't mention RISC, object orientation, functional, modern optimization techniques such as SSA, register allocation by graph coloring[27] etc.

it is fairly technical
: The book can be hard to read for the beginner, contrary to Section 5.2 [Modern Compiler Implementation], page 233.

Nevertheless, curious readers will find valuable information about historically important compilers, people, papers etc. Reading the last section of each chapter (Bibliographical Notes) is a real pleasure for whom is interested.

It should be noted that the French edition, "Compilateurs: Principes, techniques et outils", was brilliantly translated by Pierre Boullier, Philippe Deschamp, Martin Jourdan, Bernard Lorho and Monique Lazaud: the pleasure is as good in French as it is in English.

`Cool: The Classroom Object-Oriented Compiler` [Web Site]

The Classroom Object-Oriented Compiler[28], from the University of California, Berkeley, is very similar in its goals to the Tiger project as described here. Unfortunately it seems dead: there are no updates since 1996. Nevertheless, if you enjoy the Tiger project, you might want to see its older siblings.

`CStupidClassName` – *Dejan Jelović* [Paper]

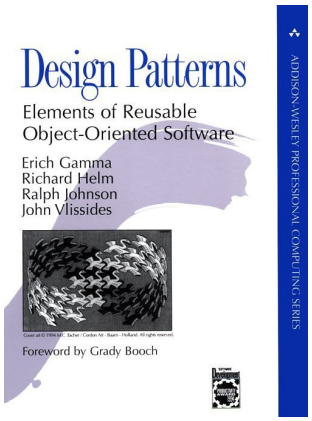
This short paper, CStupidClassName[29], explains why naming classes `CLikeThis` is stupid, but why lexical conventions are nevertheless very useful. It turns out we follow the same scheme that is emphasized there.

[27] To be fair, the Dragon Book leaves a single page (not sheet) to graph coloring.

[28] `http://theory.stanford.edu/~aiken/software/cool/cool.html`.

[29] `http://www.jelovic.com/articles/stupid_naming.htm`.

`Design Patterns: Elements of Reusable Object-Oriented Software` – *Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides* [Book]

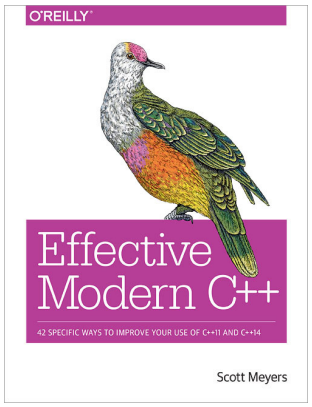


Published by Addison-Wesley; ISBN: 0-201-63361-2.

A book you must have read, or at least, you must know it. In a few words, let's say it details nice programming idioms, some of them you should know: the VISITOR, the FLYWEIGHT, the SINGLETON etc. See the Design Patterns Addison-Wesley Page[30]. A pre-version of this book is available on the Internet as a paper: Design Patterns: Abstraction and Reuse of Object-Oriented Design[31]. Surprisingly, The full version of Design Pattern CD[32] is available on the net.

You may find additional information about Design Patterns on the Portland Pattern Repository[33].

`Effective Modern C++` – *Scott Meyers* [Book]



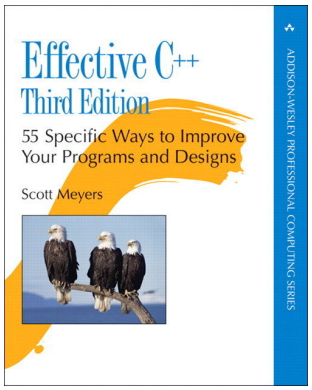
336 pages; Publisher: O'Reilly Media; 1st edition (November 2014); ISBN: 1-491-90399-6

An amazingly practical book when using C++11 and C++14 (modern C++). These days, it should be the first book that every new C++ programmer should read. It follows the same format as [Effective C++], page 241. Effective Modern C++ O'Reilly Page[34].

In this document, EMC*n* refers to item *n* in Effective Modern C++.

[30] `http://www.informit.com/store/design-patterns-elements-of-reusable-object-oriented-9780201633610`.
[31] `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.136.2555`.
[32] `http://www.saeedsh.com/resources/Design%20Patterns.pdf`.
[33] `http://c2.com/cgi/wiki?PortlandPatternRepository`.
[34] `http://shop.oreilly.com/product/0636920033707.do`.

`Effective C++` – *Scott Meyers* [Book]

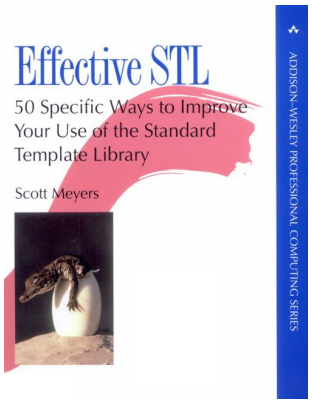


320 pages; Publisher: Addison-Wesley Pub Co; 3rd edition (May 2005); ISBN: 0-321-33487-6

An excellent book that might serve as a C++ lecture for programmers. Every C++ programmer should have read it at least once, as it treasures C++ recommended practices as a list of simple commandments. Be sure to buy the second edition, as the first predates the C++ standard. See the Effective C++ Addison-Wesley Page[35].

In this document, EC*n* refers to item *n* in Effective C++.

`Effective` STL – *Scott Meyers* [Book]



Published by Addison-Wesley; ISBN: 0-201-74962-9

A remarkable book that provides deep insight on the best practice with STL. Not only does it teach what's to be done, but it clearly shows why. A book that any C++ programmer should have read. See the Effective STL Addison-Wesley Page[36].

In this document, ES*n* refers to item *n* in Effective STL.

`Engineering a simple, efficient code generator generator` – *Christopher W. Fraser, David R. Hanson, Todd A. Proebsting* [Paper]

ACM Letters on Programming Languages and Systems 1, 3 (Sep. 1992), 213-226.

This paper[37] describes iburg, a BURG clone that delay dynamic programming at compile time (BURG-like programs use dynamic programming to select the optimum tree tiling during a bottom-up walk).

[35] `http://www.informit.com/store/effective-c-plus-plus-55-specific-ways-to-improve-your-9780321334879`.
[36] `http://www.informit.com/store/effective-stl-50-specific-ways-to-improve-your-use-9780201749625`.
[37] `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.140.1823&rep=rep1&type=pdf`.

`Generic Visitors in C++` – *Nicolas Tisserand* [Technical Report]
This report is available on line from Visitors Page[38]: Generic Visitors in C++[39]. Its abstract reads:

> The Visitor design pattern is a well-known software engineering technique that solves the double dispatch problem and allows decoupling of two interdependent hierarchies. Unfortunately, when used on hierarchies of Composites, such as abstract syntax trees, it presents two major drawbacks: target hierarchy dependence and mixing of traversal and behavioral code.
>
> CWI's visitor combinators are a seducing solution to these problems. However, their use is limited to specific "combinators aware" hierarchies.
>
> We present here Visitors, our attempt to build a generic, efficient C++ visitor combinators library that can be used on any standard "visitable" target hierarchies, without being intrusive on their codes.

This report is in the spirit of [Modern C++ Design], page 243, and should probably be read afterward.

`Guru of the Week` [News]
Written by various authors, compiled by Herb Sutter

Guru of the Week (GotW) is a regular series of C++ programming problems created and written by Herb Sutter. Since 1997, it has been a regular feature of the Internet newsgroup `comp.lang.c++.moderated`, where you can find each issue's questions and answers (and a lot of interesting discussion).

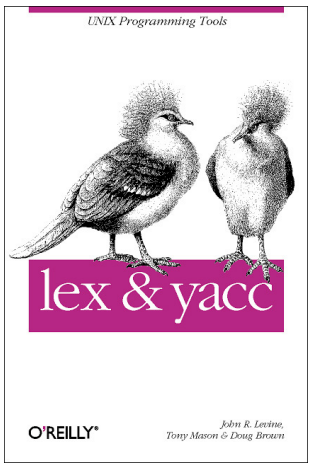
The Guru of the Week Archive[40] (the famous GotW) is freely available. In this document, GotW*n* refers to the item number *n*.

`How not to go about a programming assignment` – *Agustín Cernuda del Río* [Article]
This paper provides excellent advice on how to succeed an assignment by showing the converse: how *not* to go about a programming assignment[41]:

- All about programming, in the strictest sense of the word
  - Ignore messages
  - Don't stop to think
  - I don't want any trouble
- If only I could find the words
  - Reading
  - Writing
- Your relationship with your lecturer
  - Don't ask for help
  - Challenge your lecturer
  - Be clever using electronic mail
- And, of course...
  - Leave it all for the last minute
  - Cheat with your assignment

[38] `http://www.lrde.epita.fr/cgi-bin/twiki/view/Projects/Visitors`.
[39] `http://www.lrde.epita.fr/cgi-bin/twiki/view/Publications/20030528-Seminar-Tisserand-Report`.
[40] `http://www.gotw.ca/gotw/`.
[41] `http://www.di.uniovi.es/~cernuda/noprog_ENG.html`.

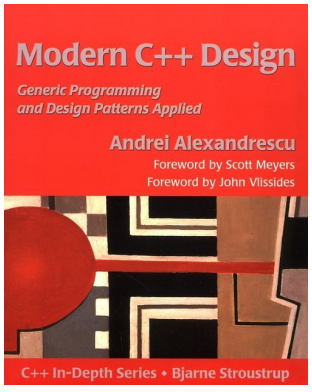
`Lex & Yacc` – *John R. Levine, Tony Mason, Doug Brown* [Book]
Published by O'Reilly & Associates; 2nd edition (October 1992); ISBN: 1-565-92000-7.



Because the books aims at a complete treatment of Lex and Yacc on a wide range of platforms, it provides too many details on material with little interest for us (e.g., we don't care about portability to other Lexes and Yacces), and too few details on material with big interest for us (more about exclusive start condition (Flex only), more about Bison only stuff, interaction with C++ etc.).

`Making Compiler Design Relevant for Students who will (Most Likely) Never Design a Compiler` – *Saumya K. Debray* [Article]
This paper about teaching compilers[42] justifies this lecture. This paper is addressing compiler construction **lectures**, not compiler construction **projects**, and therefore it misses quite a few motivations we have for the Tiger *project*.

`Modern C++ Design -- Generic Programming and Design Patterns Applied` – *Andrei Alexandrescu* [Book]



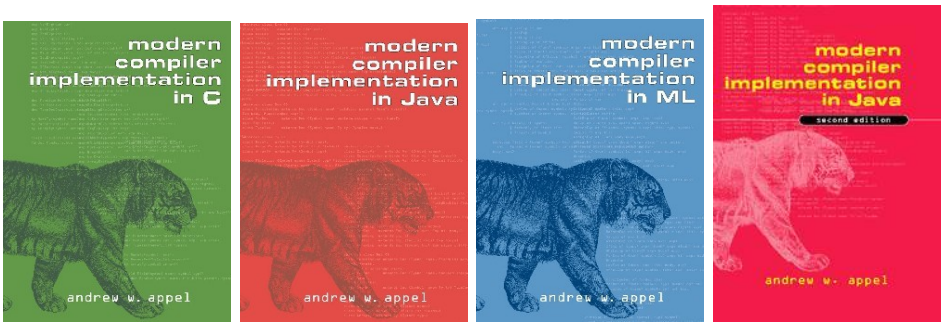
Published by Addison-Wesley in 2001; ISBN: 0-52201-70431-5
A wonderful book on very advanced C++ programming with a heavy use of templates to achieve beautiful and useful designs (including the classical design patterns, see [Design Patterns - Elements of Reusable Object-Oriented Software], page 240). The code is available in the form of the Loki Library[43]. The Modern C++ Design Web Site[44] includes pointers to excerpts such as the Smart Pointers[45] chapter.

[42] `https://cs.arizona.edu/~debray/Publications/teaching_compilers.pdf`.
[43] `http://sourceforge.net/projects/loki-lib/`.
[44] `http://www.moderncppdesign.com/book/main.html`.
[45] `http://www.aw.com/samplechapter/0201704315.pdf`.

Read this book only once you have gained good understanding of the C++ core language, and after having read the "Effective C++/STL" books.

`Modern Compiler Implementation in C, Java,` ML – *Andrew W. Appel* [Book]

Published by Cambridge University Press; ISBN: 0-521-58390-X



See Section 5.2 [Modern Compiler Implementation], page 233. In our humble opinion, most books give way too much emphasis to scanning and parsing, leaving little material to the rest of the compiler, or even nothing for advanced material. This book does not suffer these flaws.

`Object Management Group` [Web Site]

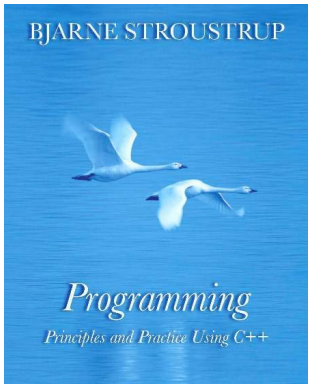
OMG's Home Page[46], with a lot of ressources for object-oriented software engineering, particularly on the Unified Modeling Language[47] (UML).

`Parsing Techniques -- A Practical Guide` – *Dick Grune and Ceriel J. Jacob* [Book]

Published by the authors; ISBN: 0-13-651431-6

A remarkable review of all the parsing techniques. Because the book is out of print, its authors made it freely available: Parsing Techniques – A Practical Guide[48].

`Programming: Principles and Practice Using C++` – *Bjarne Stroustrup* [Book]



This book targets the "advanced beginner" in C++ and covers a wide range of topics including non-core C++ subjects such as GUI programming. A recommended lecture for modern C++ learning.

Published by Addison-Wesley Professional, 2008; ISBN-13: 978-0321543721.

[46] `http://www.omg.org/`.

[47] `http://www.uml.org/`.

[48] `http://dickgrune.com/Books/PTAPG_1st_Edition/`.

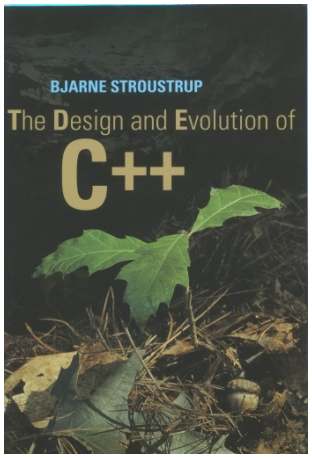
SPOT : une bibliothèque de vérification de propriétés de logique temporelle à temps linéaire [Report]
– *Alexandre Duret-Lutz & Rachid Rebiha*

This report presents SPOT, a model checking library written in C++ and Python. Parts were inspired by the Tiger project, and reciprocally, parts inspired modifications in the Tiger project. For instance, earlier versions of SPOT made use of a visitor hierarchy. You are encouraged to read the sections about the visitor hierarchy and its implementation. Another useful source of inspiration was the use of Python and Swig to write the command line interface.

Testing student-made compilers – *José de Oliveira Guimarães* [Paper]
ACM SIGCSE Bulletin archive Volume 26, Issue 3 (September 1994).

This paper[49] gives a classified list of test cases for a small Pascal compiler. It is a good source of inspiration for any other language.

The Design and Evolution of C++ – *Bjarne Stroustrup* [Book]



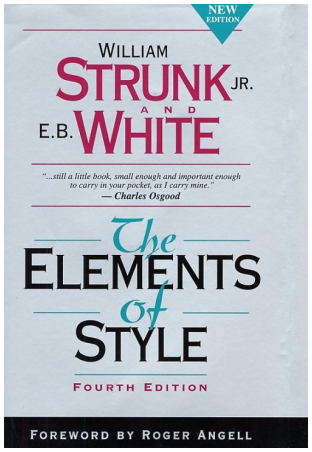
Published by Addison-Wesley, ISBN 0-201-54330-3.

This book is definitely worth reading for curious C++ programmers. I (Roland) find it an excellent companion to reference C++ books, or even to the C++ standard. Many aspects of the language that are often criticized find a justification in this book. Moreover, the book not only tells the history of C++ (up to 1994), but it also explains the design choices and reflexions of its authors (and Bjarne Stroustrup's in the first place), which go far beyond the scope of C++.

However, the book only describes the first 15 years of C++ or so. Recent work on C++ (and especially on the C++0x effort that eventually led to C++ 2011) can be found in Stroustrup's papers, available online.

[49] http://cyan-lang.org/jose/green/articles/icse1.pdf.

`The Elements of Style` – *William Strunk Jr., E.B. White* [Book]



Published by Pearson Allyn & Bacon; 4th edition (January 15, 2000); ISBN: 020530902X.

This little book (105 pages) is perfect for people who want to improve their English prose. It is quite famous, and, in addition to providing useful writing thumb rules, it features rules that are interesting as pieces of writing themselves! For instance "The writer must, however, be certain that the emphasis is warranted, lest a clipped sentence seem merely a blunder in syntax or in punctuation".

You may find the much shorter (43 pages) First Edition of *The Elements of Style*[50] on line.

`Thinking in C++ Volume 1` – *Bruce Eckel* [Book]

Published by Prentice Hall; ISBN: 0-13-979809-9

Available on the Internet on many Book Download Sites[51]. For instance, Thinking in C++ Volume 1 Zipped[52].

`Thinking in C++ Volume 2` – *Bruce Eckel and Chuck Allison* [Book]

Available on the Internet on many Book Download Sites[53]. For instance, Thinking in C++ Volume 2 Zipped[54].

`Traits: a new and useful template technique` – *Nathan C. Myers* [Article]

The first presentation of the traits technique is from this paper, Traits: a new and useful template technique[55]. It is now a common C++ programming idiom, which is even used in the C++ standard.

`Writing Compilers and Interpreters -- An Applied Approach Using C++` – *Ronald Mak* [Book]

Published by Wiley; Second Edition, ISBN: 0-471-11353-0

This book is not very interesting for us: the compiler material is not very advanced (no real AST, not a single line on optimization, register allocation is naive as the translation is stack based etc.), and the C++ material is not convincing (for a start, it is not

[50] `http://www.bartleby.com/141/`.
[51] `http://mindview.net/Books/DownloadSites`.
[52] `http://www.babeuk.net/mirror/book/TICPP-2nd-ed-Vol-one.zip`.
[53] `http://mindview.net/Books/DownloadSites`.
[54] `http://www.babeuk.net/mirror/book/TICPP-2nd-ed-Vol-two.zip`.
[55] `http://www.cantrip.org/traits.html`.

standard C++ as it still uses '`#include <iostream.h>`' and the like, there is no use of STL etc.).

STL `Home` [Web site]
SGI's STL Home Page[56], which includes the complete documentation on line.

5.4 The gnu Build System

Automake is used to facilitate the writing of power `Makefile`. Libtool eases the creation of libraries, especially dynamic ones. Autoconf is required by Automake: we do not address portability issues for this project. See [Autotools Tutorial], page 236, for documentation.

Using `info` is pleasant, for instance '`info autoconf`' on any properly set up system.

5.4.1 Package Name and Version

To set the name and version of your package, change the `AC_INIT` invocation. For instance, TC-4 for the `bardec_f` group gives:

```
AC_INIT([Bardeche Group Tiger Compiler], 4, [bardec_f@epita.fr],
        [bardec_f-tc])
```

5.4.2 Bootstrapping the Package

If something goes wrong, or if it is simply the first time you create `configure.ac` or a `Makefile.am`, you need to set up the GNU Build System. That's the goal of the simple script `bootstrap`, which most important action is invoking:

```
$ autoreconf -fvi
```

The various files (`configure`, `Makefile.in`, etc.) are created. There is no need to run '`make distclean`', or `aclocal` or whatever, before running `autoreconf`: it knows what to do.

Then invoke `configure` and `make` (see Section 5.5 [GCC], page 249):

```
$ mkdir _build
$ cd _build
$ ../configure CXX=g++-5.0
$ make
```

Alternatively you may set `CC` and `CXX` in your environment:

```
$ export CXX=g++-5.0
$ mkdir _build
$ cd _build
$ ../configure && make
```

This solution is preferred since the value of `CC` etc. will be used by the `configure` invocation from '`make distcheck`' (see Section 5.4.3 [Making a Tarball], page 247).

5.4.3 Making a Tarball

Once the package correctly autotool'ed and configured (see Section 5.4.2 [Bootstrapping the Package], page 247), run '`make distcheck`' to build the tarball. Contrary to a simple '`dist`', '`distcheck`' makes sure everything will work properly. In particular it:

1. performs some simple checks. For instance, it checks that the `NEWS` file is about the current version, i.e., it checks that the second argument given to `AC_INIT` is in the top of `NEWS`, otherwise it fails with '`NEWS not updated; not releasing`'.
2. creates the tarball (via '`make dist`')

[56] `http://www.sgi.com/tech/stl/index.html`.

3. untars the tarball
4. configures the tarball in a separate directory `_build` (to avoid cluttering the source files with the built files).

   Arguments passed to the top level `configure` (e.g., '`CXX=g++-5.0`') *will not be taken into account here*. Running '`export CXX=g++-5.0`' is a better way to require these compilers. Alternatively use `DISTCHECK_CONFIGURE_FLAGS` to specify the arguments of the embedded `configure`:

   ```
   $ make distcheck DISTCHECK_CONFIGURE_FLAGS='--without-swig CXX=g++-4.0'
   ```
5. runs '`make`' (and following targets) in paranoid mode. This mode consists in forbidding any change in the source tree, because if, when you run '`make`' something must be changed in the sources, then it means something is broken in the tarball. If, for instance, for some reason it wants to run `autoconf` to recreate `configure`, or if it complains that `autom4te.cache` cannot be created, then it means the tarball is broken! So track down the reason of the failure.
6. runs '`make check`'
7. runs '`make dist`' again.

If you just run '`make dist`' instead of '`make distcheck`', then you might not notice some files are missing in the distribution. If you don't even run '`make dist`', the tarball might not compile elsewhere (not to mention that we don't care about object files etc.).

Running '`make distcheck`' is the only means for you to check that the project will properly compile on our side. Not running `distcheck` is like turning off the type checking of your compiler: you hide instead of solving.

At this stage, if running '`make distcheck`' does not create `bardec_f-tc-4.tar.bz2`, something is wrong in your package. Do not rename it, do not create the tarball by hand: something is rotten and be sure it will break on the examiner's machine.

5.4.4 Setting site defaults using `CONFIG_SITE`

Another way to pass options to `configure` is to use a site configuration file. This file will be "sourced" by configure to set some values and options, and will save you some bytes on your command line when you'll invoke `configure`.

First, write a `config.site` file:

```
# -*- shell-script -*-

echo "Loading config.site for $PACKAGE_TARNAME"
echo "(srcdir: $srcdir)"
echo

package=$PACKAGE_TARNAME

echo "config.site: $package"
echo

# Configuration specific to EPITA KB machines (GNU/Linux on x86-64).
case $package in
  tc)
    # Turn off optimization when building with debugging information
```

```
      # (the build dir must have ``debug'' in its name).
      case `pwd` in
        *debug*) :
          : ${CFLAGS="-ggdb -O0"}
          : ${CXXFLAGS="-ggdb -O0 -D_GLIBCXX_DEBUG"}
        ;;
      esac
      # Help configure to find the Boost libraries on NetBSD.
      if test -f /usr/pkg/include/boost/config.hpp; then
        with_boost=/usr/pkg/include
      fi

      # Set CC, CXX, BISON, MONOBURG, and other programs as well.
      : ${CC=/u/prof/acu/pub/NetBSD/bin/gcc}
      : ${CXX=/u/prof/acu/pub/NetBSD/bin/g++}
      : ${BISON=/u/prof/yaka/bin/bison}
      : ${MONOBURG=/u/prof/yaka/bin/monoburg}
      # ...
    ;;
  esac

  set +vx
```

Then, set the environment variable `CONFIG_SITE` to the path to this file, and run `configure`:

```
$ export CONFIG_SITE="$HOME/src/config.site"
$ ../configure
```

or if you use a C-shell:

```
$ setenv CONFIG_SITE "$HOME/src/config.site"
$ ../configure
```

This is useful when invoking `make distcheck`: you don't need to pollute your environment, nor use Automake's `DISTCHECK_CONFIGURE_FLAGS` (see Section 5.4.3 [Making a Tarball], page 247).

Of course, you can have several `config.site` files, one for each architecture you work on for example, and set the `CONFIG_SITE` variable according to the host/system.

5.5 gcc, The gnu Compiler Collection

We use GCC 5.0, which includes both `gcc-5.0` and `g++-5.0`: the C and C++ compilers. Do not use older versions as they have poor compliance with the C++ standard. You are welcome to use more recent versions of GCC if you can use one, but the tests will be done with 5.0. Using a more recent version is often a good means to get better error messages if you can't understand what GCC 5.0 is trying to say.

There are good patches floating around to improve GCC. The GCC Bounds Checking Page[57] is an interesting example in this respect. It is however no longer maintained and we advise you to have a look at mudflap[58] instead, which is officially part of GCC.

[57] `http://williambader.com/bounds/example.html`.

[58] `http://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging`.

5.6 Clang, A C language family front end for llvm

Clang is a front end for the LLVM compiler infrastructure supporting the C, C++, Objective C and Objective C++ languages. LLVM provides a modern framework written in C++ for creating compiler-related projects.

We advise you to check your code with the `clang` (C) and `clang++` (C++) front ends (version 3.8 or more) in addition to `gcc` and `g++`. Clang may indeed report other errors and warnings. Moreover, Clang's messages are often easier to read than GCC's.

You can find more information on Clang, LLVM and other related projects on the LLVM Home Page[59].

5.7 gdb, The gnu Project Debugger

Every serious project development makes use of a debugger. Such a tool allows the programmer to examine her program, running it step by step, display/change values etc.

GDB is a debugger for programs written in C, C++, Objective-C, Pascal (and other languages). It will help you to track and fix bugs in your project. Don't forget to pass the option `-g` (or `-ggdb`, depending on your linker's abilities to handle GDB extensions) to your compiler to include useful information into the debugged program.

Pay attention when debugging a libtoolized program, as it may be a shell script wrapper around the real binary. Thus don't use

```
$ gdb tc
```

or expect errors from GDB when running the program. Use `libtool`'s `--mode=execute` option to run gdb instead:

```
$ libtool --mode=execute gdb tc
```

or the following shortcut:

```
$ libtool exe gdb tc
```

Detailed explanations can be found in the Libtool manual.

5.8 Valgrind, The Ultimate Memory Debugger

Valgrind is an open-source memory debugger for GNU/Linux on x86/x86-64 (and other environments) written by Julian Seward, already known for having committed Bzip2. It is the best news for programmers for years. Valgrind is so powerful, so beautifully designed that you definitely should wander on the Valgrind Home Page[60].

In the case of the Tiger Compiler Project correct memory management is a primary goal. To this end, Valgrind is a precious tool, as is dmalloc[61], but because STL implementations are often keeping some memory for efficiency, you might see "leaks" from your C++ library. See its documentation on how to reclaim this memory. For instance, reading the GCC's C++ Library FAQ[62], especially the item "memory leaks" in containers[63] is enlightening.

I (Akim) personally use the following shell script to track memory leaks:

```
#! /bin/sh

exec 3>&1
```

[59] `http://www.llvm.org/`.
[60] `http://valgrind.org`.
[61] `http://dmalloc.com`.
[62] `http://gcc.gnu.org/onlinedocs/libstdc++/faq.html`.
[63] `http://gcc.gnu.org/onlinedocs/libstdc++/faq.html#faq.memory_leaks`.

```
export GLIBCPP_FORCE_NEW=1
export GLIBCXX_FORCE_NEW=1
exec valgrind --num-callers=20 \
              --leak-check=yes \
              --leak-resolution=high \
              --show-reachable=yes \
              "$@" 2>&1 1>&3 3>&- |
  sed 's/^==[0-9]*==/==/' >&2 1>&2 3>&-
```

File 5.1: `v`

For instance on File 4.52,

```
$ v tc -XA 0.tig
error /opt/tiger/assignments/v: 6: exec: valgrind: not found
```

Example 5.1: *v tc -XA 0.tig*

Starting with GCC 3.4, `GLIBCPP_FORCE_NEW` is spelled `GLIBCXX_FORCE_NEW`.

As in the case of GDB, you should be careful when running a libtoolized program in Valgrind. Use the following command to make sure that this is your `tc` binary (and not the shell) that is checked by Valgrind:

```
$ libtool exe valgrind tc
```

You can ask Valgrind to run a debugger when it catches an error, using the `--db-attach` option. This is useful to inspect a process interactively.

```
$ valgrind --db-attach=yes ./tc
```

The default debugger used by Valgrind is GDB. Use the `--db-command` option to change this.

Another technique to make Valgrind and GDB interact is to use Valgrind's gdbserver and the `vgdb` command (see Valgrind's documentation for detailed explanations).

5.9 Flex & Bison

We use Bison 3.0.4.19-fbaf[64], that is able to produce a C++ parser combined with modern features such as GLR, variants and complete symbols. If you don't use this Bison, you will be in trouble.

The original papers on Lex and Yacc are:

Johnson, Stephen C. [1975].
: Yacc: Yet Another Compiler Compiler[65]. Computing Science Technical Report No. 32, Bell Laboratories, Murray hill, New Jersey.

Lesk, M. E. and E. Schmidt [1975].
: Lex: A Lexical Analyzer Generator[66]. Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey.

These introductory guides can help beginners:

Thomas Niemann.
: A Compact Guide to Lex & Yacc[67].

  An introduction to Lex and Yacc.

[64] `https://www.lrde.epita.fr/~tiger/download/bison-3.0.4.19-fbaf.tar.xz`.

[65] `http://epaperpress.com/lexandyacc/download/yacc.pdf`.

[66] `http://epaperpress.com/lexandyacc/download/lex.pdf`.

[67] `http://www.epaperpress.com/lexandyacc/index.html`.

Collective Work

Programming with GNU Software[68].

Contains information about Autoconf, Automake, Gperf, Flex, Bison, and GCC.

The Bison documentation[69], and the Flex documentation[70] are available for browsing.

5.10 havm

HAVM is a `Tree` (HIR or LIR) programs interpreter. It was written by Robert Anisko so that EPITA students could exercise their compiler projects before the final jump to assembly code. It is implemented in Haskell, a pure non strict functional language very well suited for this kind of symbolic processing. HAVM was coined on both Haskell, and VM standing for Virtual Machine.

Resources:

- Required version is HAVM 0.27
- HAVM Home Page[71]
- HAVM Documentation[72]
- Feedback can be sent to LRDE's Projects Address[73].
- There are some *known bugs* that cause HAVM to execute incorrectly HIR programs. This happens when some `jump` break the recursive structure of the program, i.e., when a `jump` goes outside its enclosing structure (`seq`, or `eseq` etc.).

  Examples of Tiger sources onto which HAVM is likely to behave incorrectly include:

```
while 1 do
  print_int((break; 1))
```

  File 5.2: `ineffective-break.tig`

  or

```
if 0 | 0 then 0 else 1
```

  File 5.3: `ineffective-if.tig`

  See HAVM's documentation[74] for details, node "Known Problems"[75].

5.11 MonoBURG

MonoBURG is a code generator generator, a tool that produces a function from a tree-pattern description of an instruction set. If you think of Bison being a program generating an AST generator from concrete syntax, you can see MonoBURG as a program generating an Assem generator from LIR trees.

MonoBURG is named after BURG, a program that generates a fast tree parser using BURS (Bottom-Up Rewrite System). MonoBURG is part of the Mono Project[76] and has been extended by Michaël Cadilhac for the needs of the Tiger Project.

[68] `https://www.lrde.epita.fr/~tiger/doc/gnuprog2/`.
[69] `http://www.gnu.org/software/bison/manual/`.
[70] `https://westes.github.io/flex/manual/`.
[71] `http://www.lrde.epita.fr/wiki/Havm`.
[72] `https://www.lrde.epita.fr/~tiger/doc/havm.html`.
[73] `mailto:projects@lrde.epita.fr`.
[74] `https://www.lrde.epita.fr/~tiger/doc/havm.html`.
[75] `https://www.lrde.epita.fr/~tiger/doc/havm.html#Known-Problems`.
[76] `http://www.mono-project.com/`.

Resources:

- Required version is MonoBURG 1.0.6a
- MonoBURG Home Page[77]
- Feedback can be sent to LRDE's Projects Address[78].

Some papers on code generator generators are available in the bibliography. See [BURG - Fast Optimal Instruction Selection and Tree Parsing], page 238, and [Engineering a simple efficient code generator generator], page 241.

5.12 Nolimips

Nolimips (formerly Mipsy) is a MIPS simulator designed to execute simple register based MIPS assembly code. It is a minimalist MIPS virtual machine that, contrary to other simulators (see Section 5.13 [SPIM], page 253), supports unlimited registers. The lack of a simulator featuring this prompted the development of Nolimips.

Its features are:

- sufficient support of MIPS instruction set
- infinitely many registers

It was written by Benoît Perrot as an LRDE member, so that EPITA students could exercise their compiler projects after instruction selection but before register allocation. It is implemented in C++ and Python.

Resources:

- Required version is Nolimips 0.10
- Nolimips Home Page[79]
- Nolimips Documentation[80]
- Feedback can be sent to LRDE's Projects Address[81].

5.13 spim

The SPIM documentation reads:

> SPIM S20 is a simulator that runs programs for the MIPS R2000/R3000 RISC computers. SPIM can read and immediately execute files containing assembly language. SPIM is a self-contained system for running these programs and contains a debugger and interface to a few operating system services.
>
> The architecture of the MIPS computers is simple and regular, which makes it easy to learn and understand. The processor contains 32 general-purpose 32-bit registers and a well-designed instruction set that make it a propitious target for generating code in a compiler.
>
> However, few years ago, the obvious question was: why use a simulator when many people have workstations that contain a hardware, and hence significantly faster, implementation of this computer? One reason was that these workstations are not generally available. Another reason was that these machine will not persist for many years because of the rapid progress leading to new and faster computers. Unfortunately, the trend is to make computers

[77] `https://www.lrde.epita.fr/wiki/MonoBURG`.
[78] `mailto:projects@lrde.epita.fr`.
[79] `http://www.lrde.epita.fr/wiki/Nolimips`.
[80] `https://www.lrde.epita.fr/~tiger/doc/nolimips.html`.
[81] `mailto:projects@lrde.epita.fr`.

> faster by executing several instructions concurrently, which makes their architecture more difficult to understand and program. The MIPS architecture may be the epitome of a simple, clean RISC machine. Nowadays, the MIPS architecture is no more a common architecture.
>
> In addition, simulators can provide a better environment for low-level programming than an actual machine because they can detect more errors and provide more features than an actual computer. For example, SPIM has a X-window interface that is better than most debuggers for the actual machines.
>
> Finally, simulators are an useful tool for studying computers and the programs that run on them. Because they are implemented in software, not silicon, they can be easily modified to add new instructions, build new systems such as multiprocessors, or simply to collect data.

SPIM is written and maintained by James R. Larus on SourceForge.[82].

5.14 swig

Our compiler provides two different user interfaces: one is a command line interface fully written in C++, using the "Task" system, and the other is a binding of the primary functions into the Python script language (see Section 5.15 [Python], page 254. This binding is automatically extracted from our modules using SWIG.

The SWIG home page[83] reads:

> SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. SWIG is primarily used with common scripting languages such as Perl, Python, Tcl/Tk, and Ruby, however the list of supported languages also includes non-scripting languages such as Java, OCAML and C#. Also several interpreted and compiled Scheme implementations (Guile, MzScheme, Chicken) are supported. SWIG is most commonly used to create high-level interpreted or compiled programming environments, user interfaces, and as a tool for testing and prototyping C/C++ software. SWIG can also export its parse tree in the form of XML and Lisp s-expressions. SWIG may be freely used, distributed, and modified for commercial and non-commercial use.

5.15 Python

We promote, but do not require, Python as a scripting language over Perl because in our opinion it is a cleaner language. A nice alternative to Python is Ruby[84].

The Python Home Page[85] reads:

> Python is an interpreted, interactive, object-oriented programming language. It is often compared to Tcl, Perl, Scheme or Java.
>
> Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries, as well as to various windowing systems (X11, Motif, Tk, Mac, MFC). New built-in modules are easily written in C or C++. Python is also usable as an extension language for applications that need a programmable interface.

[82] `http://spimsimulator.sourceforge.net/`.

[83] `http://www.swig.org/`.

[84] `http://www.ruby-lang.org/en/`.

[85] `http://www.python.org`.

> The Python implementation is portable: it runs on many brands of UNIX, on Windows, OS/2, Mac, Amiga, and many other platforms. If your favorite system isn't listed here, it may still be supported, if there's a C compiler for it. Ask around on news:comp.lang.python – or just try compiling Python yourself.
>
> The Python implementation is copyrighted but freely usable and distributable, even for commercial use.

5.16 Doxygen

We use Doxygen[86] as the standard tool for producing the developer's documentation of the project. Its features *must* be used to produce good documentation, with an explanation of the role of the arguments etc. The quality of the documentation will be part of the notation. Details on how to use proper comments are given in the Doxygen Manual[87].

The documentation produced by Doxygen must not be included, but the target `html` must produce the HTML documentation in the `doc/html` directory.

[86] `http://www.doxygen.org/index.html`.

[87] `http://www.stack.nl/~dimitri/doxygen/manual.html`.

Appendix A Appendices

A.1 Glossary

Contributions to this section (as for the rest of this documentation) will be greatly appreciated.

activation block
: Portion of dynamically allocated memory holding all the information a (recursive) function needs at runtime. It typically contains arguments, automatic local variables etc. Implemented by the class `frame::Frame` (see Section 4.14 [TC-5], page 132).

build
: The machine/architecture on which the program is built. For instance, EPITA students typically *build* their compiler on GNU/Linux. Contrast with "target" and "host".

curriculum
: From WordNet: n : a course of academic studies; "he was admitted to a new program at the university" (syn: "course of study", "program", "syllabus").

Guru of the Week

GotW
: See Section 5.3 [Bibliography], page 236.

HAVM
: HAVM is a `Tree` (HIR or LIR) programs interpreter. See Section 5.10 [HAVM], page 252.

host
: The machine/architecture on which the program is run. For instance, EPITA students typically run their Tiger Compiler on GNU/Linux. Contrast with "build and "target".

IA-32
: The official new name for the i386 architecture.

scholarship
: It is related to "scholar", not "school"! It does not mean "scolarité".

  From WordNet:

  - n 1: financial aid provided to a student on the basis of academic merit.
  - 2: profound knowledge (syn: "eruditeness", "erudition", "learnedness", "learning").

  See "schooling" and "curriculum".

schooling
: From WordNet:

  - n 1: the act of teaching at school.
  - 2: the process of being formally educated at a school; "what will you do when you finish school?" (syn: "school").
  - 3: the training of an animal (especially the training of a horse for dressage).

snippet
: A piece of something, e.g., "code snippet".

stack frame
: Synonym for "activation block".

static hierarchy
: A hierarchy of classes without virtual methods. In that case there is no (inclusion) polymorphism. For instance:

```
struct A     { };
struct B: A  { };
```

SPIM — SPIM S20 is a simulator that runs programs for the MIPS R2R3000 RISC computers. See Section 5.13 [SPIM], page 253.

target — The machine (or language) aimed at by a compiling tool. For instance, our target is principally MIPS. Compare with "build" and "host".

traits — Traits are a useful technique that allows to write (compile time) functions ranging over types. See [Traits], page 246, for the original presentation of traits. See [Modern C++ Design], page 243, for an extensive use of traits.

`vtable` — For a given class, its table of pointers to virtual methods.

A.2 GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given

in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgments" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any

sections entitled "Acknowledgments", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that

specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

A.2.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being *list*"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

A.3 Colophon

This is version of `assignments.texi`, last edited on April 16, 2018, and compiled 4 June 2018, using:

```
$ tc --version
tc (LRDE Tiger Compiler 1.63)
$Id: 6e8714f086f9562cd0e03931a93b8a714f9c90a3 $

Akim Demaille            Alain Vongsouvanh        Alexandre Duret-Lutz
Alexis Brouard           Arnaud Fabre             Ashkan Kiaie-Sandjie
Axel Manuel              Benoît Perrot            Benoît Sigoure
Benoît Tailhades         Cédric Bail              Christophe Duong
Clément Vasseur          Cyprien Orfila           Daniel Gazard
Fabien Ouy               Etienne Renault          Francis Maes
Francis Visoiu Mistrih   Gilles Walbrou           Guillaume Duhamel
Guillaume Marques        Jérémie Simon            Julien Roussel
Julien Grall             Laurent Gourvénec        Léo Ercolanelli
Loïc Banet               Michaël Cadilhac         Matthieu Simon
Moray Baruh              Nicolas Burrus           Nicolas Pouillard
Nicolas Teck             Pablo Oliveira           Pierre-Louis Dagues
Pierre-Yves Strub        Pierre De Abreu          Quôc Peyrot
Raphaël Poss             Razik Yousfi             Roland Levillain
Robert Anisko            Sarasvati MoutoucomarapouléSébastien Broussaud
Sébastien Piat           Stéphane Molina          Théophile Ranquet
Thierry Géraud           Valentin David           Yann Grandmaître
Yann Popo                Yann Régis-Gianas
```

Example A.1: `tc --version`

```
$ havm --version
HAVM 0.27
Written by Robert Anisko.

Copyright (C) 2002-2003  Robert Anisko
```

```
Copyright (C) 2003-2007, 2009, 2011-2014  EPITA Research and Development Lab-
oratory (LRDE).
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Example A.2: *havm --version*

```
$ nolimips --version
nolimips (Nolimips) 0.10
Written by Benoit Perrot.

Copyright (C) 2003, 2004, 2005, 2006, 2008, 2009, 2010, 2012 Benoit Perrot.
nolimips comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute and modify it
under certain conditions; see source for details.
```

Example A.3: *nolimips --version*

A.4 List of Files

A.5 List of Examples

A.6 Index

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y