The Tiger Compiler Project

Edition February 24, 2004

Akim Demaille

Table of Contents

Nul n'est censé ignorer la loi.

Everything exposed in this document is expected to be known.

This document[1] details the various tasks the "Compilation" students must complete. It was last edited on February 24, 2004.

[1] `http://www.lrde.epita.fr/~akim/compil/assignments/assignments.html`.

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 5.1 [Modern Compiler Implementation], page 113) 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, I am wrong.

Basically this document contains three kinds of informations:

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 13, and Section 2.5 [Evaluation], page 25.

Incremental
: You should read these parts as and when needed. This includes mostly Chapter 4 [Compiler Stages], page 39.

Auxiliary
: This information is provided to help you: just go there when you feel the need, Chapter 5 [Tools], page 113, and Chapter 3 [Tarballs], page 29. 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 my download area[3].
- The developer documentation of the Tiger Compiler[4].
- Most of the material I provide (lecture notes, older exams, current tarballs etc.) is in my compilation 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 9 months, 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/~akim/download`.
[4] `http://www.lrde.epita.fr/~akim/compil/tc-doc/`.
[5] `http://www.lrde.epita.fr/~akim/compil`.

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 T2 fails now that we implemented T4? 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 January to September (and optionally further). Each four 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 Metrowerks (CodeWarrior), HP, Sun, Intel, M$ as examples.

Understanding Computers

Too many students still have a very fuzzy mental picture of what is a computer, 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 114, 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 120):

> 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/index.html`.

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

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

[10] `http://www.research.att.com/~bs/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 120.

Compiler The project aims at the implementation of a compiler, but *this is a minor issue*. The field of compilers is a wonder 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 116, 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 119, 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 today C is probably the first employer of programmers in the world, so 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 given its roots, it naturally suffers from many defects. But 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*. Note for instance that Bjarne Stroustrup's list of C++ Applications[11] mentions that the following ones are written in C++:

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 various flavors of Visual C++ - mostly 6.0 and 7.0 but we do have a few holdouts still using 5.0 :-(and some products like Windows XP use more recent

[11] `http://www.research.att.com/~bs/applications.html`.

builds of the compiler. The list would include major products like:

- Windows XP
- Windows NT (NT4 and 2000)
- Windows 9x (95, 98, Me)
- Microsoft Office (Word, Excel, Access, PowerPoint, Outlook)[...]

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

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 me.

The years correspond to the class, e.g., Tiger 2005 refers to EPITA class 2005, i.e., the project ran from January 2003 to 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 (outputting comments in the assembly showing the original code), no bells, no whistles, no etc. This also implies that sometimes interested students will feel we "stole" the pleasure to write nice pieces of code from them; understand that we actually provided code to the *other* students. However, 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 my 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 I grew tired of fixing the tarballs, and in order to have a robust, efficient (albeit some piece of pain in the neck sometimes) distributions[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 to the standard package interface (see Section 2.4 [Delivery], page 25).

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` 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 I was not enough aware.

1.4.3 Tiger 2003

During this year, I was helped by:

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

Comaintainers
: Alexandre Duret-Lutz, Thierry Géraud.

Delivery date were:

Stage	Delivery
T1	Monday, December 18th 2000 at noon
T2	Friday, February 23th 2001 at noon
T3	Friday, March 30th 2001 at noon
T4	Tuesday, June 12th 2001 at noon
T5	Monday, September 17th 2001 at noon

Some groups have reached T6.

Criticisms include:

The C++ compiler is broken
: I 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 my account into '`rm -rf ~`'. Some students and myself 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, we have decent compilers made available by students, 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 delivered, 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 delivered 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 I 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 guess. But it conflicts with the previous item...

1.4.4 Tiger 2004

During this year, I 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.

Delivery date were:

Stage	**Delivery**
T2	Tuesday, March 4th 2002 at noon
T3	Friday, March 15th 2002 at noon
T4	Friday, April 12th 2002 at noon
T5	Friday, June 14th 2002, at noon
T6	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 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. Note too, that even though using 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 7.

Ending on T6 is frustrating
: Several students were frustrated by the fact we had to stop at T6: the reference compiler did not have any back-end. Continuing onto T7 was offered to several groups, and some of them actually finished the compiler. We took there work, adjusted it, and it became the base of the reference compiler of 2005. The most significant effort was made by Daniel Gazard.

Double delivery is intractable
: Students were allowed to deliver twice their project — with a small penalty — if they failed to meet the so-called "first delivery 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 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	Delivery
T0	Friday, January 24th 2003 at noon
T1	Friday, February 14th 2003 at noon
T2	Friday, March 14th 2003 at noon
T4	Friday, April 25th 2003 at noon
T3	Rush from Saturday, May 24th at 18:00 to Sunday at noon
T56	Friday, June 20th 2003, at noon
T7	Friday, July 4th 2003 at noon
T78	Friday, July 18th 2003 at noon
T9	Monday, September 8th 2003 at noon

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, denunciate 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

[13] Please, let me know who I forgot!

don't do as we did, and make sure you 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 2.3.5 [File Conventions], page 21), 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/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 7.

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 2.3 [Coding Style], page 15) 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 [T3], page 53, and Section 4.7 [T5], page 60), 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 T2 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 [T2 Goals], page 47) 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 T5), many students are deadly lost.

We are considering splitting T5 into two: T5- which would be limited to programs without escaping variables, and T5+ 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 (see Section 2.3.2 [Use of Foreign Features], page 15). Hints at possible extensions are provided, and finally, alternative implementation are suggested for each stage, for instance see Section 4.4.6 [T2 Improvements], page 53.

1.4.6 Tiger 2006

Deliveries were:

Stage	**Delivery**
T0	Wednesday, February 4th 2004 at noon
T1	Sunday, February 8th 2004 at noon
T2	Sunday, March 7th 2004 at noon
T3	Sunday, March 21th 2004

2 Instructions

2.1 Interactions

Bare 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 `epita.cours.compile`. You need to have a very good reason to send a message to the assistants or to Akim, as it usually annoys us, what is not in your interest.

  The news group `epita.cours.compile` is dedicated to the compilation lecture, the Tiger project, and attached 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 T1	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, or 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 publicized**) and the full error message from the compiler/tool.

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

A pertinent content
: Trolls are "exvited" from here, invite them elsewhere.

2.2 Groups

Starting with T1, assignments are to be done by groups of four.

The first cause of failures to the Tiger project is human problems within the groups. I 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, I 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 life long 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 of the project. The whole P1 is a testbed 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 the friends, that will be hard. Plus it will be even harder to reveal the problems your group is having.

Recruit people you can depend on

Trust should be your first criterion.

Members should have similar programming skills
Weak programmers should run away from skill programmers

The worst "good idea" you could have is "I'm a poor programmer, I should be in a group of skilled programmers: I will learn a lot from them". By experience, I can assure you that 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 you don't understand, they are often poor teachers. The day before the delivery, the leader does your assignments, because saving the group is now what matters. 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 delivery, 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 my advice: if you have difficulties with programming, be with other people like you. Your chances are better together.

And don't forget 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 student having not participated enough to learn what was to be learned. Three first year students with one repeater is OK, but a different ratio is asking for troubles.

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 tarball 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: denunciate the problem, 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 the LRDE or LSE or 3ie, etc. If your group is seriously unbalanced (three skilled people is OK, otherwise be four), 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.

2.3 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 advices and tips. As the captain Barbossa would put it, "actually, it's more of a guideline than a rule."

2.3.1 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, 3.2 or higher (see Section 5.4 [GCC], page 123).

2.3.2 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
: As is provided by the unstable Debian package `loki`. See [Modern C++ Design], page 119, for more information about Loki.

The Boost Library
: As provided by the unstable Debian packages `libboost-*`. See [Boost.org], page 115.

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 me your proposal; acceptance is required to use them.

2.3.3 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 5.5 [Valgrind], page 123). 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::auto_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. You should strive to prevent code duplication to sneak 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 `aborts`, 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
comparable_to (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::comparable_to (const Type &rhs)
{
  return &rhs == this || dynamic_cast <Nil *> (&rhs);
}

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

bool
comparable_to (const Type &lhs, const Type &rhs)
{
  return lhs->comparable_to (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. Note that the ability of `typeid` on static hierarchies can be a pitfall; 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[1]. Note that if you provide `A` with a virtual methods 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 9 of see [Thinking in C++ Volume 2], page 121: Run-time type identification[2].

`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:

---

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

[2] `http://www.cs.virginia.edu/~th8k/ticpp/vol2/html/Chap09.htm`.

```
template <typename T>
void
swap (T &b, 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. Note that `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);
}
```

`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 116.

`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 such words are likely to be used by your compiler or standard library[3].

For instance, write:

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

See [CStupidClassName], page 116.

`Name your typedef` *`foo`*`_type` [Rule]

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

[3] Actually, it is '`_[A-Z]`' which is reserved.

```
typedef std::map< const Symbol, Entry_T > map_type;
typedef std::list< map_type > symtab_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 TypeVisitor : public ast::DefaultVisitor<ast::non_const_kind>
{
  typedef ast::DefaultVisitor<ast::non_const_kind> super_type;
  using super_type::visit;
// ...
```

2.3.4 Use of STL

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

For instance, instead of declaring

```
typedef set::set<const Temp *> temp_set_t;
```

declare

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

typedef set::set<const Temp *, temp_compare> temp_set_t;
```

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.

Make functor classes adaptable (ES40) [Rule]

When you write unary or binary predicates to use in interaction with stl, make sure to derive from std::unary_function or std::binary_function. For instance:

```
/// Object function to compare two Temp*.
struct temp_ptr_less
  : public std::binary_function <const Temp*, const Temp*, bool>
{
  bool operator() (const Temp *s1, const Temp *s2) const;
};
```

Prefer algorithm call 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 (ES44)` [Rule]

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[4] on the Internet.

2.3.5 File Conventions

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

`Declarations in '*.hh'` [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, and include it at the end of this header 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[5], GotW034[6]):

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

  ```
  #include "foo.hh"
  ```

  write

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

- if you need output stream, then include '`ostream`', not '`iostream`'. Actually, if you merely need to declare the existence of streams, you might want to include '`iosfwd`'.

`Inlined definitions in '*.hxx'` [Rule]

If there are definitions that should be loaded in different places (definitions of templates, inline functions etc.), then declare and document them in the '`*.hh`' file, and implement them in the '`*.hxx`' file. Note that this file should first include its corresponding '`*.hh`' file, the latter including itself this file. It is indeed surprising, but the header guards make this work properly.

`Definitions of functions and variables in '*.cc'` [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);
}
```

[4] `http://www.informit.com/isapi/product_id~%7BEB0D6EE6-6DDC-48B4-A730-19EE22B8B486%7D/content/index.asp`.

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

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

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).

`'lib*.hh' and 'lib*.cc' are pure` [Rule]
There should be only pure functions in the interface of a module. That means that the functions in these files should not depend upon globals, nor have side effects of global objects. Of course no global variable can be defined here either.

`'fwd.hh' exports 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. 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.

`'*-tasks.hh' and '*-tasks.cc' are impure` [Rule]
Tasks, as designed currently, are *the* place for side effects. That's where globals such as the current ast, the current assembly program, etc., are defined and modified.

2.3.6 Matters of Style

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

`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:
  typedef std::string string_type;
```

```
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 ();

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

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

private:
  typedef std::string string_type;
  string_type bar_;
  int baz_;
}
```

and add useful Doxygen comments.

`Prefer Doxygen Documentation to plain comments` [Rule]

We use Doxygen (see Section 5.12 [Doxygen], page 128) to maintain the developer documentation of the Tiger Compiler.

`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 writing ChangeLogs.

`Write Documentation in Doxygen` [Rule]
Documentation is a genuine part of programming, just as testing. The quality of this documentation can change the grade.

`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);
```

Of course, Doxygen documentation is not appropriate everywhere.

`Use foo_get, not get_foo` [Rule]
Accessors have standardized names: `foo_get` and `foo_set`.

`Use 'rebox.el' to markup 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 it is advised to use '`rebox.el`' (provided in '`config/`') to build them. 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.  |
`-----------------------------*/
```

`Use print 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 `print` as a member function. Use this kind of prototype:

```
std::ostream& Class::print (std::ostream& ostr [, ...]) const
```

where the ellipsis denote optional additional arguments. Note that print returns the stream.

2.4 Delivery

Each group must provide a tarball, made via '`make distcheck`'. All the information about the delivery per se is given on the Yaka's Delivery Page[7].

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 5.3.1 [Package Name and Version], page 122). The following commands must work properly:

```
$ bunzip2 -cd bardec_f-tc-n.tar.bz2 | tar xvf -
$ cd bardec_f-tc-n
$ export CC=gcc-3.2
$ export CXX=g++-3.2
$ ./configure
$ make
$ cd src
$ ./tc /tmp/test.tig
$ cd ..
$ make distcheck
```

For more information on the tools, see Section 5.3 [The GNU Build System], page 121, Section 5.4 [GCC], page 123.

Your tarball must be done via '`make distcheck`' (see Section 5.3.3 [Making a Tarball], page 122). Any tarball which is not built thanks to '`make distcheck`' (this is easy to see: they include files we don't want, and don't contain some files we need...) will be penalized with at least '`### tarball_not_clean`'.

2.5 Evaluation

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

[7] `http://etudiant.epita.fr:8000/~yaka/doc/rendus.html`.

2.5.1 Automated Evaluation

Each stage of the compiler will be evaluated by an automatic corrector. As soon as the tarball are delivered, the logs are available on '`http://www.lrde.epita.fr/~akim/compil`', in the directory corresponding to your class and stage. For instance, 2004 students ought to read '`http://www.lrde.epita.fr/~akim/compil/2004/4/bardec_f-tc-4.log`'.

We stress that automated evaluation enforces the requirements: you *must* stick to what is being asked. For instance, for T3 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.

If you find some unexpected errors (your project does compile with the reference compiler, some files are missing, your output is slightly incorrect etc.) **immediately** send a new tarball to yaka@epita.fr with '`[Tiger]`' as prefix of the subject. This corresponds to '`### patch`'.

Do not wait for the final marks to be computed, this is extremely irritating, and doomed to failure. You must understand that (i) you increase our workload, and (ii) anyway this is the wrong approach, the Tiger Compiler is a big project which must be continuously improved.

If, anyway, you send a tarball to fix your problems long after the initial date, you will be flagged as '`### super_late`', which impact on the mark is quite bad...

2.5.2 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 I wish to make clear: I, Akim, and the other examiners, will probably be harsh (maybe even very harsh), but this does not mean I disrespect you, or judge you badly.

You are here to defend your project and knowledge, I'm 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 me something, and I will never buy something from someone who cries when I'm 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.

2.5.3 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 [T0 Goals], page 40, for instance) on which you will be interrogated.

Note to the 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.

Note to the examiners: broken tarballs.

If you fixed the tarball or made whatever modification, you *must* run '`make distcheck`' again, and replace the tarball they delivered with the new one. Do not keep the old tarball, do not install it 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.

2.5.4 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 T3 compiler will be exercised on T1, T2, and T3. If there are still errors on T1 and T2 tests, they will pessimize the result of T3 tests. The older the errors are, the more expensive they are.

As an example, here are the formulas to compute the global success rate of T3 and T5:

```
global-rate-T3 := rate-T3 * (+ 2 * rate-T1
                             + 1 * rate-T2) / 3
global-rate-T5 := rate-T5 * (+ 4 * rate-T1
```

```
+ 3 * rate-T2
+ 2 * rate-T3
+ 1 * rate-T4) / 10
```

Because a project which fail half of the time is not a project that deserves half of 20, the global-rate is elevated to 1.7 before computing the mark:

```
mark-T3 := roundup (power (global-rate-T3, 1.7) * 20 - malus-T3, 1)
```

where '`roundup (x, 1)`' is x rounded up to one decimal ('`roundup (15, 1) = 15`', '`roundup (15.01, 1) = 15.1`').

When the project is also evaluated by a human, '`power`' is not used. Rather, the success rate modifies the mark given by the examiner:

```
mark-T2 := roundup (eval-T2 * global-rate-T2 - malus-T2, 1)
```

3 Tarballs

3.1 Given Tarballs

The naming scheme for provided tarballs is different from the scheme you must follow (see Section 2.4 [Delivery], page 25). Our naming scheme looks like '`2004-tc-2.0.tar.bz2`'[1]. If we update the tarballs, they will be named '`2004-tc-2.x.tar.bz2`'. But *your* tarball *must* be named '*`login`*`-tc-2.tar.bz2`', even if you send a second version of your project.

We also (try to) provide patches from one tarball to another. For instance '`2006-tc-1.0-2.0.diff.bz2`'[2] is the difference from '`2006-tc-1.0.tar.bz2`'[3] to '`2006-tc-2.0.tar.bz2`'[4]. You are encouraged to read this file as understanding a patch is expected from any Unix programmer. Just run '`bzless 2006-tc-1.0-2.0.diff.bz2`'.

To apply the patch:

1. go into the top level of your current tarball
2. remove any file which name might cause confusion afterward ('`find . -name '*.orig' -o -name '*.rej' | xargs rm`')
3. run '`bzcat 2006-tc-1.0-2.0.diff.bz2 | patch -p1`'
4. look for all the failures ('`find . -name '*.rej`') and fix them by hand once you understood why the patch did not apply

You might need to repeat the process to jump from a version x to x + 2 via version x + 1.

3.2 Project Layout

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

3.2.1 The Top Level

'`AUTHORS`' In the top level of the distribution, there must be a file '`AUTHORS`' 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 the first in the list. Do not include emails other than those of EPITA. I repeat: give the '`6_1@epita.fr`' address. Note that the file '`AUTHORS`' is automatically distributed, 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`' Various free information.

'`argp/`' The command line parser we use.

[1] `http://www.lrde.epita.fr/~akim/compil/download/2004-tc-2.0.tar.bz2`.
[2] `http://www.lrde.epita.fr/~akim/compil/download/2006-tc-1.0-2.0.diff.bz2`.
[3] `http://www.lrde.epita.fr/~akim/compil/download/2006-tc-1.0.tar.bz2`.
[4] `http://www.lrde.epita.fr/~akim/compil/download/2006-tc-2.0.tar.bz2`.

‘`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 ‘`src`’ Directory

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

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

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

3.2.3 The ‘`src/misc`’ Directory

Convenient C++ routines.

`contract.hh` (*src/misc/*) [File]
A useful improvement over ‘`cassert`’.

`escape.hh` (*src/misc/*) [File]
This file implements a means to output string while escaping non printable characters. An example:

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

Understanding how `escape` works is required starting from T2.

`set.hh` (*src/misc/*) [File]
A wrapper around `std::set` that introduce convenient operators (`operator+` and so forth).

`timer.hh` (*src/misc/*) [File]
`timer.cc` (*src/misc/*) [File]
A class that makes it possible to have timings of the compilation process, as when using ‘`--time-report`’ with `gcc`, or ‘`--report=time`’ with `bison`. It is used in the `Task` machinery, but can be used to provide better timings (e.g., separating the scanner from the parser).

3.2.4 The ‘`src/task`’ Directory

No namespace for the time being, but it should be `task`. Delivered for T1. A generic scheme to handle the components of our compiler, and their dependencies.

3.2.5 The ‘`src/symbol`’ Directory

Namespace ‘`symbol`’, delivered for T1.

`symbol.hh` (*src/symbol/*) [File]
The handling of the symbols. In the 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) as much faster.

The class `symbol::Symbol` is an implementation of this idea. See the lecture notes.

`table.hh` (*src/symbol/*) [File]
The handling of generic symbol tables, i.e., it is independent of functions, types and variables.

3.2.6 The 'src/ast' Directory

Namespace '`ast`', delivered for T2. Implementation of the abstract syntax tree. The file '`ast/README`' gives an overview of the involved class hierarchy.

`location.hh` (*src/ast/*) [File]
`position.hh` (*src/ast/*) [File]
These files are now simply forwarding the definitions of `yy::Position` and `yy::Location` as provided by Bison.

`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 `non_const_kind` or `const_kind`. This allows to define to 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.hh` (*src/ast/*) [File]
Implementation of the `DefaultVisitor` class, which walks the abstract syntax tree, doing nothing. It is mainly used as a basis for deriving other visitors. Actually, just as above, there is a template, so that we have two different default visitors: `DefaultVisitor<const_kind>` and `DefaultVisitor<non_const_kind>`.

`print-visitor.hh` (*src/ast/*) [File]
Implementation of the `PrintVisitor` class, which performs pretty-printing in the tiger compiler.

3.2.7 The 'src/parse' Directory

Namespace '`parse`'. Delivered during T1.

`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/type' Directory

Namespace '`type`'. Type checking.

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

`types.hh` (*src/type/*) [File]
The definition of all the types. You are free to use whatever layout you wish (several files); we have a single '`types.hh`' file.

`type-entry.hh` (*src/type/*) [File]
Definitions of `type::TypeEntry`, `type::VarEntry`, and `type::FunEntry`, used in `type::TypeEnv` to associate data to types, variables, and functions (obviously).

`type-env.hh` (*src/type/*) [File]
The types environment, comprising three symbol tables: types, functions, and variables, used by the `type::TypeVisitor`.

3.2.9 The 'src/temp' Directory

Namespace `temp`, delivered for T5.

`temp.hh` (*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.

`label.hh` (*src/temp/*) [File]
We need labels for `jump`s, for functions, strings etc.

3.2.10 The 'src/tree' Directory

Namespace `tree`, delivered for T5. 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 `symbol::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`).

3.2.11 The 'src/frame' Directory

Namespace '`frame`', delivered for T5.

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

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

3.2.12 The 'src/translate' Directory

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

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

`libtranslate.cc` (*src/translate/*) [File]
The compiled module.

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

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

`level.hh` (*src/translate/*) [File]
`level.cc` (*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::Node` to delay their translation until the actual use is known.

`level-entry.hh` (*src/translate/*) [File]
All the information that the environment must keep about variables and functions.

`level-env.hh` (*src/translate/*) [File]
The levels environment, containing '`LevelVarEntry`''s and '`LevelFunEntry`''s. We don't need to store information related to types here.

`translation.hh` (*src/translate/*) [File]
functions used by the `translate::TranslateVisitor` 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.

`translate-visitor.hh` (*src/translate/*) [File]
Implements the class '`TranslateVisitor`' 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 a '`ast::SimpleVar`':

```
virtual void operator() (const SimpleVar& e)
{
  const Access &access = env_.var_access_get (e.name_get ());
  exp_ = simpleVar (access, *level_);
}
```

3.2.13 The 'src/canon' Directory

Namespace `tree`.

3.2.14 The 'src/assem' Directory

Namespace `assem`, delivered for T7.

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

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

`fragment.hh` (*src/assem/*) [File]
`fragment.cc` (*src/assem/*) [File]
Implementation of `assem::Fragment`, `assem::ProcFrag`, and `assem::DataFrag`. They are comparable to `translate::Fragment`: aggregate some informations 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.hh` (*src/assem/*) [File]
`libassem.cc` (*src/assem/*) [File]
The interface of the module, and its implementation.

3.2.15 The 'src/target' Directory

Namespace `target`, delivered for T7. Some data on the back end. Given in full.

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

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

`mips-cpu.hh` (*src/target/*) [File]
`mips-target.hh` (*src/target/*) [File]
The description of the MIPS (actually, SPIM/Mipsy) target.

`ia32-cpu.hh` (*src/target/*) [File]
`ia32-target.hh` (*src/target/*) [File]
Description of the i386. This is not part of the project, it is left only as an incomplete source of inspiration.

`target-tasks.cc` (*src/target/*) [File]
`target-tasks.hh` (*src/target/*) [File]
The command line interface to specify the target architecture.

3.2.16 The 'src/codegen' Directory

Namespace `codegen`, delivered for T7.

`mips` (*src/codegen/*) [File]
`ia32` (*src/codegen/*) [File]
The instruction selection per se split into a generic part, and a target specific (MIPS and IA32) part. See Section 3.2.17 [src/codegen/mips], page 36, and Section 3.2.18 [src/codegen/ia32], page 36.

`assembly.hh` (*src/codegen/*) [File]
The abstract class `codegen::Assembly` which is the interface for elementary assembly instructions generation.

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

`libcodegen.hh` (*src/codegen/*) [File]
`libcodegen.cc` (*src/codegen/*) [File]
Converting `translate::Fragment`s into `assem::Fragment`s.

`codegen-tasks.hh` (*src/codegen/*) [File]
`codegen-tasks.cc` (*src/codegen/*) [File]
Command line interface.

`tiger-runtime.c` (*src/codegen/*) [File]
This is the Tiger runtime, written in C, based on Andrew Appel's '`runtime.c`'[5]. The actual '`runtime.s`' file for MIPS was written by hand, but the ia32 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 a' la C string. The length part is due to conformance to the 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/Mipsy's `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`
: I don't know how Appel wants to support '`"bar" < "foo"`' since he doesn't provide `strcmp`. We do. But note that anyway his implementation of '`"foo" != "fooo"`' is more efficient than ours, 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 `t_main`, which is the "main" that your compiler should provide.

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

3.2.17 The 'src/codegen/mips' Directory

Namespace `codegen::mips`, delivered for T7. Code generation for MIPS R2000.

`runtime.s` (*src/codegen/mips/*) [File]
`runtime.cc` (*src/codegen/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 3.2.16 [src/codegen], page 35, '`tiger-runtime`'.

`spim-assembly.hh` (*src/codegen/mips/*) [File]
`spim-assembly.cc` (*src/codegen/mips/*) [File]
Our assembly language (syntax, opcodes and layout); it abstracts the generation of MIPS 2000 instructions. `codegen::mips::SpimAssembly` derives from `codegen::Assembly`.

`codegen.hh` (*src/codegen/mips/*) [File]
`codegen.cc` (*src/codegen/mips/*) [File]
Our real and only back end: a translator from LIR to ASSEM using the MIPS 2000 instruction set defined by `codegen::mips::SpimAssembly`. It is implemented as a maximal munch. `codegen::mips::Codegen` derives from `codegen::Codegen`.

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

3.2.18 The 'src/codegen/ia32' Directory

Namespace `codegen::ia32`, delivered for T7. Code generation for IA32. 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.

`runtime.s` (*src/codegen/ia32/*) [File]
`runtime.cc` (*src/codegen/ia32/*) [File]
The Tiger runtime in IA32 assembly language: `print` etc. The C++ file '`runtime.cc`' is built from '`runtime.s`': do not edit the former. See Section 3.2.16 [src/codegen], page 35, '`tiger-runtime`'.

`gas-assembly.hh` (*src/codegen/ia32/*) [File]
`gas-assembly.cc` (*src/codegen/ia32/*) [File]
Our assembly language (syntax, opcodes and layout); it abstracts the generation of IA32 instructions using Gas' syntax. `codegen::ia32::GasAssembly` derives from `codegen::Assembly`.

`codegen.hh` (*src/codegen/ia32/*) [File]
`codegen.cc` (*src/codegen/ia32/*) [File]
The IA32 back-end: a translator from LIR to ASSEM using the IA32 instruction set defined by `codegen::ia32::GasAssembly`. It is implemented as a maximal munch. `codegen::ia32::Codegen` derives from `codegen::Codegen`.

`gas-layout.hh` (*src/codegen/ia32/*) [File]
`gas-layout.cc` (*src/codegen/ia32/*) [File]
How IA32 fragments are to be displayed. In other words, that's where the (global) syntax of the target assembly file is selected.

3.2.19 The 'src/graph' Directory

Namespace `graph`, a generic implementation of graphs. Delivered for T7.

`graph.hh` (*src/graph/*) [File]
`graph.hxx` (*src/graph/*) [File]
Oriented and undirected graphs.

`handler.hh` (*src/graph/*) [File]
`handler.hxx` (*src/graph/*) [File]
Abstractions/indirections for graph nodes and edges.

`iterator.hh` (*src/graph/*) [File]
`iterator.hxx` (*src/graph/*) [File]
Iterating over nodes and edges of graphs.

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

3.2.20 The 'src/liveness' Directory

Namespace `liveness`, delivered for T8.

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

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

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

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

3.2.21 The 'src/regalloc' Directory

Namespace `regalloc`, register allocation, delivered for T9.

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

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

`libregalloc.hh` (*src/regalloc/*) [File]
`libregalloc.cc` (*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.

`regalloc-tasks.hh` (*src/regalloc/*) [File]
`regalloc-tasks.cc` (*src/regalloc/*) [File]
Command line interface.

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`'[6]. It contains the following directories:

'`good`' These programs are correct.

'`scan`' These programs have lexical errors.

'`parse`' These programs have syntax errors.

'`type`' These programs contain type mismatches.

[6] `http://www.lrde.epita.fr/~akim/compil/download/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 delivery 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.

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.

4.2 T0, Naive Scanner and Parser

2006-T0 delivery is Wednesday, February 4th 2004 at noon.

This section has been updated for EPITA-2006.

T0 is a weak form of T1: the scanner and the parser are written, but the framework is simplified (see Section 4.3.4 [T1 Code to Write], page 45).

[1] `http://www.lrde.epita.fr/~akim/compil/tc-doc/`.

Relevant lecture notes include: ‘`compilation-lecture.pdf`’[2].

4.2.1 T0 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 `yylval` to pass token values 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.
- First use of a C++ feature: the `std::string` class.

4.2.2 T0 Samples

Running T0 basically consists in looking at exit values:

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

File 4.1: ‘`simple.tig`’

```
$ tc simple.tig
```

Example 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 simple.tig
error Starting parse
error Entering state 0
error Reading a token: --(end of buffer or a NUL)
error --accepting rule at line 99 ("print")
error Next token is 259 ("identifier" simple.tig:1.0-4: print)
error Shifting token 259 ("identifier"), Entering state 2
error Reading a token: --accepting rule at line 52 (" ")
error --accepting rule at line 58 ("(")
error Next token is 264 ("(" simple.tig:1.6)
error Reducing via rule 81 (line 403), "identifier" -> funid
error state stack now 0
error Entering state 18
error Next token is 264 ("(" simple.tig:1.6)
error Shifting token 264 ("("), Entering state 59
error Reading a token: --accepting rule at line 103 (""")
error --accepting rule at line 172 ("Hello, World!")
error --accepting rule at line 159 ("\n")
error --accepting rule at line 134 (""")
error Next token is 258 ("string" simple.tig:1.7-23: Hello, World!
error )
error Shifting token 258 ("string"), Entering state 1
error Reducing via rule 19 (line 213), "string" -> exp
```

[2] `http://www.lrde.epita.fr/~akim/compil/lecture-notes/compilation-lecture.pdf`.

```
error state stack now 59 18 0
error Entering state 102
error Reading a token: --accepting rule at line 59 (")")
error Next token is 265 (")" simple.tig:1.24)
error Reducing via rule 46 (line 284), exp -> args.1
error state stack now 59 18 0
error Entering state 104
error Next token is 265 (")" simple.tig:1.24)
error Reducing via rule 45 (line 279), args.1 -> args
error state stack now 59 18 0
error Entering state 103
error Next token is 265 (")" simple.tig:1.24)
error Shifting token 265 (")"), Entering state 123
error Reducing via rule 20 (line 216), funid "(" args ")" -> exp
error state stack now 0
error Entering state 13
error Reading a token: --(end of buffer or a NUL)
error --accepting rule at line 53 ("
error ")
error --(end of buffer or a NUL)
error --EOF (start condition 0)
error Now at end of input.
error Reducing via rule 1 (line 163), exp -> program
error state stack now 0
error Entering state 12
error Now at end of input.
```

Example 2: *SCAN=1 PARSE=1 tc simple.tig*

A lexical error must be properly diagnosed *and reported*. The following (generated) examples display the location: *this is not required for T0*; nevertheless, an error message on the standard error output is required.

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

File 4.2: 'back-zee.tig'

```
$ tc back-zee.tig
error back-zee.tig:1.0-2: unrecognized escape: \z
⇒2
```

Example 3: *tc back-zee.tig*

Similarly for syntactical errors.

```
a++
```

File 4.3: 'postinc.tig'

```
$ tc postinc.tig
error postinc.tig:1.2: syntax error, unexpected "+"
error Parsing Failed
⇒3
```

Example 4: *tc postinc.tig*

4.2.3 T0 Code to Write

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

You must write:

‘`scantiger.ll`’
: The scanner.

  `yylval` supports strings, integers and even symbols. Nevertheless, symbols (i.e., identifiers) are returned as plain strings for the time being: the class `symbol::Symbol` is introduced in T1.

  The environment variable `SCAN` enables Flex scanner debugging traces.

‘`parsetiger.yy`’
: The parser, and maybe `main` if you wish.

  There is no requirement to implement `YYPRINT` support.

  The environment variable `PARSE` enables Bison parser debugging traces, i.e., running

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

  sets `yydebug` to 1.

‘`tc.cc`’
: Optionally, you may write your driver, i.e., `main`, in this file. Putting it into ‘`parsetiger.yy`’ is OK in T0 as it is reduced to its simplest form with no option support. Of course the exit status must conform to the Tiger Compiler Reference Manual[3].

‘`Makefile`’
: This file is mandatory. Running `make` *must build an executable* `tc`. The GNU Build System is not used: there is no need for Autoconf, Automake etc.

The requirements on the tarball are the same as usual, see Chapter 3 [Tarballs], page 29.

4.2.4 T0 Improvements

Possible improvements include:

4.3 T1, Scanner and Parser

2006-T1 delivery is Sunday, February 8th 2004 at noon.

This section is updated for EPITA-2006.

Scanner and parser are properly running, but the abstract syntax tree is not built yet. Differences with T0 include:

GNU Build System
: Autoconf, Automake are used.

Options, Tasks
: The compiler supports basic options via in the Task module.

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 T1 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

[3] `http://www.lrde.epita.fr/~akim/compil/tiger.html`.

[4] `http://www.lrde.epita.fr/~akim/compil/lecture-notes/dev-tools.pdf`.

[5] `http://www.lrde.epita.fr/~akim/compil/lecture-notes/scanner.pdf`.

'`Makefile.in`'s exist) you should depend on `make` only. See Section 5.3 [The GNU Build System], page 121.

Integration into an existing framework
: Putting your own code into the provided tarball.

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.

Location Tracking
: Issues within the scanner and the parser.

Implementation of a simple C++ class
: The code for `symbol::Symbol` is incomplete.

A first standard container: `std::set`
: The implementation of the `symbol::Symbol` class relies on `std::set`.

The Flyweight design pattern
: The `Symbol` class is an implementation of the Flyweight design pattern.

4.3.2 T1 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 test01.tig
```

Example 5: *`tc test01.tig`*

If there are lexical errors, the exit status is 2, and a an error message is output on the standard error output. Note that 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 unterminated-comment.tig
error unterminated-comment.tig:2.1-3.0: unexpected end of file in a comment
⇒2
```

Example 6: *`tc 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 type-nil.tig
error type-nil.tig:1.12-14: syntax error, unexpected "nil", expecting "identifier"
error Parsing Failed
⇒3
```

Example 7: *tc 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 8: *tc C:/TIGER/SAMPLE.TIG*

The option '`--parse-trace`', which relies on Bison's `%debug` directive, and the use of `YYPRINT`, must work properly:

```
a + "a"
```

File 4.7: '`a+a.tig`'

```
$ tc --parse-trace --parse a+a.tig
error Starting parse
error Entering state 0
error Reading a token: Next token is 259 ("identifier" a+a.tig:1.0: a)
error Shifting token 259 ("identifier"), Entering state 2
error Reading a token: Next token is 271 ("+" a+a.tig:1.2)
error Reducing via rule 76 (line 382), "identifier" -> varid
error state stack now 0
error Entering state 17
error Reducing via rule 38 (line 259), varid -> lvalue
error state stack now 0
error Entering state 14
error Next token is 271 ("+" a+a.tig:1.2)
error Reducing via rule 37 (line 255), lvalue -> exp
error state stack now 0
error Entering state 13
error Next token is 271 ("+" a+a.tig:1.2)
error Shifting token 271 ("+"), Entering state 43
error Reading a token: Next token is 258 ("string" a+a.tig:1.4-6: a)
error Shifting token 258 ("string"), Entering state 1
error Reducing via rule 19 (line 213), "string" -> exp
error state stack now 43 13 0
error Entering state 81
error Reading a token: Now at end of input.
error Reducing via rule 31 (line 244), exp "+" exp -> exp
error state stack now 0
error Entering state 13
error Now at end of input.
error Reducing via rule 1 (line 163), exp -> program
error state stack now 0
error Entering state 12
error Now at end of input.
```

Example 9: *tc --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 T1 Given Code

Some code is provided: '`2006-tc-1.0.tar.bz2`'[6]. See Section 3.2.1 [The Top Level], page 29, Section 3.2.2 [src], page 30, Section 3.2.7 [src/parse], page 31, Section 3.2.3 [src/misc], page 30.

4.3.4 T1 Code to Write

Be sure to read Flex and Bison documentations and tutorials, see Section 5.6 [Flex & Bison], page 125.

'`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.

```
...
\"          yylval->str = new std::string (); BEGIN SC_STRING;

<SC_STRING>{ /* Handling of the strings.  Initial " is eaten. */
     \" {
       BEGIN INITIAL;
       return STRING;
     }
...
     \\x[0-9a-fA-F]{2}  {
       yylval->str->append (1, strtol (yytext + 2, 0, 16));
     }
...
}
```

- Symbols (i.e., identifiers) must be returned as `symbol::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 IF;
...
%%
...
```

  See the lecture notes, and have a look at the scanner and parser chapters of this draft[7].

[6] `http://www.lrde.epita.fr/~akim/compil/download/2006-tc-1.0.tar.bz2`.

[7] `http://www.lrde.epita.fr/~akim/compil/gnuprog2/`.

‘src/parse/parsetiger.yy’
- The grammar must be complete but without actions.
- Complete yy::Parser::print_ to implement ‘--parse-trace’ support (see Section 4.3.2 [T1 Samples], page 43). yy::Parser::print_ is the C++ equivalent of the yyprint feature for C parsers, see the Bison documentation. Pay special attention to the display of strings and identifiers.

‘src/symbol/symbol.hxx’
The class symbol::Symbol keeps a single copy of identifiers, see Section 3.2.5 [src/symbol], page 30. Its implementation in ‘src/symbol/symbol.hxx’ is incomplete.

The most delicate part is the constructor symbol::Symbol::Symbol (const std::string &s): just bare in mind that (i) you must make sure that the string ‘s’ is inserted in the set, and (ii) save in this new symbol::Symbol object a reference to this inserted string. Carefully read the documentation of std::set::insert.

4.3.5 T1 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 it actually means

```
exp: "string" { $$ = $1; };
```

which is not type coherent. So write this instead:

```
exp: "string" {};
```

Where is ast::Exp?
Its real definition will be provided with T2, so meanwhile you have to provide a fake. We recommend for a forward declaration of ‘ast::Exp’ in ‘libparse.hh’.

‘misc/test-ref’ fails to compile properly
My bad, sorry about that: it will be activated in a later stage. Meanwhile, comment its content.

4.3.6 T1 Improvements

Possible improvements include:

4.4 T2, Building the Abstract Syntax Tree

This section was last updated for EPITA-2006 on 2004-02-18.

2006-T2 delivery is Sunday, March 7th 2003 at noon.

At the end of this stage, the compiler can build abstract syntax trees of Tiger programs and pretty-print them. The parser is equipped with error recovery. The memory is properly deallocated on demand.

The code must follow our coding style and be documented, see Section 2.3 [Coding Style], page 15, and Section 5.12 [Doxygen], page 128.

Relevant lecture notes include ‘ast.pdf’[8].

[8] http://www.lrde.epita.fr/~akim/compil/lecture-notes/ast.pdf.

4.4.1 T2 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 2.3 [Coding Style], page 15.

Memory Leak Trackers
: Using tools such as Valgrind (see Section 5.5 [Valgrind], page 123) to track memory leaks.

Error recovery with Bison
: Using the `error` token, and building usable ASTs.

Using STL containers
: The AST uses `std::list`, `symbol::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.

The Composite design pattern
: The AST hierarchy is an implementation of the Composite pattern.

The Visitor design pattern
: The `PrintVisitor` is an implementation of the Visitor pattern.

Writing good developer documentation (using Doxygen)
: The AST must be properly documented.

4.4.2 T2 Samples

Here are a few samples of the expected features.

4.4.2.1 T2 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 -A simple-fact.tig
/* == Abstract Syntax Tree. == */
let
   function fact (n : int) : int =
      if (n = 0)
         then 1
         else (n * fact ((n - 1)))
in
   fact (10)
end
```
Example 10: *tc -A simple-fact.tig*

Passing '`-D`', '`--ast-delete`', reclaims the memory associated to the AST. Valgrind will be used to check that no memory leaks, see Section 5.5 [Valgrind], page 123.

No heroic effort is asked for silly options combinations.

```
$ tc -D simple-fact.tig
```
Example 11: *tc -D simple-fact.tig*

```
$ tc -DA simple-fact.tig
error tasks.cc:22: Precondition 'the_program' failed.
⇒134
```
Example 12: *tc -DA 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 -AD string-escapes.tig
/* == Abstract Syntax Tree. == */
print ("\"EPITA\"\n")
```
Example 13: *tc -AD 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 -AD 1s-and-2s.tig
/* == Abstract Syntax Tree. == */
if (1 = 1)
   then (2 = 2)
   else 0
```
Example 14: *tc -AD 1s-and-2s.tig*

```
$ tc -AD 1s-and-2s.tig >output.tig
```
Example 15: *tc -AD 1s-and-2s.tig >output.tig*

```
$ tc -AD output.tig
/* == Abstract Syntax Tree. == */
if (1 = 1)
   then (2 = 2)
```

```
   else 0
```
Example 16: *tc -AD output.tig*

For loops must be properly displayed, i.e., although we use a `ast::VarDec` for the index of the loop, you must not display '`var`':

```
/* Valid let and for. */
let
  var a := 0
in
  for i := 0 to 100 do (a := a+1; ())
end
```
File 4.11: '`for-loop.tig`'

```
$ tc -AD for-loop.tig
/* == Abstract Syntax Tree. == */
let
   var a := 0
in
   for i := 0 to 100 do
      (
         a := (a + 1);
         ()
      )
end
```
Example 17: *tc -AD for-loop.tig*

Parentheses must not stack for free; in fact, you must even remove them.

```
((((((((((0))))))))))
```
File 4.12: '`parens.tig`'

```
$ tc -AD parens.tig
/* == Abstract Syntax Tree. == */
0
```
Example 18: *tc -AD parens.tig*

As a result, *anything output by '`tc -AD`' is equal to what '`tc -AD | tc -AD -`' displays!*

4.4.2.2 T2 Chunks

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 -T foo-bar.tig
```
Example 19: *tc -T 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 -T foo-stop-bar.tig
error foo-stop-bar.tig:1.28-33: unknown function: bar
⇒4
```

Example 20: *tc -T foo-stop-bar.tig*

The same applies to types.

We shall name *chunk* a continuous series of type (or function) declaration.

Within a chunk, duplicate names are invalid, while they are valid for separated chunks:

```
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 -T fbfsb.tig
error fbfsb.tig:3.4-28: function redefinition: foo
error fbfsb.tig:1.4-28: first definition
⇒4
```

Example 21: *tc -T fbfsb.tig*

It behaves exactly as if chunks were part of embedded `let in end`. This is why our Tiger compilers will treat chunks as syntactic sugar: an internal `let` (i.e., a `LetExp`) may only have a single chunk of declarations. If the input has several chunks, they must be split into several `let`:

```
$ tc -A fbfsb.tig
/* == Abstract Syntax Tree. == */
let
   function foo () : int =
      0
   function bar () : int =
      1
   function foo () : int =
      2
in
   let
      var stop := 0
   in
      let
         function bar () : int =
```

```
            3
        in
           0
        end
    end
end
```

Example 22: *tc -A fbfsb.tig*

Given the type checking rules for variables, whose definitions cannot be recursive, chunks of variable declarations are reduced to a single variable.

```
let var foo := 1
    var foo := foo + 1
    var foo := foo + 1
in
  foo
end
```

File 4.16: '`fff.tig`'

```
$ tc -A fff.tig
/* == Abstract Syntax Tree. == */
let
   var foo := 1
in
   let
      var foo := (foo + 1)
   in
      let
         var foo := (foo + 1)
      in
         foo
      end
   end
end
```

Example 23: *tc -A fff.tig*

4.4.2.3 T2 Error Recovery

Another part of T2 is the improvement of your parser: it must be robust to some forms of errors. Observe that on the following input:

```
(
  1;
  (2, 3);
  (4, 5);
  6
)
```

File 4.17: '`multiple-parse-errors.tig`'

several parse errors are reported, not merely the first one:

```
$ tc multiple-parse-errors.tig
error multiple-parse-errors.tig:3.4: syntax error, unexpected ",", ex-
pecting ";"
error multiple-parse-errors.tig:4.4: syntax error, unexpected ",", ex-
pecting ";"
```

```
⇒3
```

Example 24: *tc multiple-parse-errors.tig*

Of course, the exit status still reveals the parse error. Be sure that your error recovery does not break the rest of the compiler...

```
$ tc -AD multiple-parse-errors.tig
error multiple-parse-errors.tig:3.4: syntax error, unexpected ",", ex-
pecting ";"
error multiple-parse-errors.tig:4.4: syntax error, unexpected ",", ex-
pecting ";"
/* == Abstract Syntax Tree. == */
(
   1;
   ();
   ();
   6
)
⇒3
```

Example 25: *tc -AD multiple-parse-errors.tig*

4.4.3 T2 Given Code

Some code is provided: '`2006-tc-2.0.tar.bz2`'[9]. The transition from the previous versions can be done thanks to the following diffs: '`2006-tc-1.0-2.0.diff`'[10].

For a description of the new modules, see Section 3.2.3 [src/misc], page 30, Section 3.2.5 [src/symbol], page 30, and Section 3.2.6 [src/ast], page 31.

4.4.4 T2 Code to Write

What is to be done:

'`src/parse/parsetiger.yy`'

Implement error recovery.

There should be at least three uses of the token `error`. Read the Bison documentation about it.

Chuncks — 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::VarDecs`, and `ast::TypeDecs`; they are implemented thanks to `ast::AnyDecs`).

'`src/ast`' The abstract syntax tree module must be completed. There should remain no '`FIXME:`' anywhere in the code we gave. Several files are missing in full. See '`src/ast/README`' for additional information on the missing classes.

'`src/ast/default-visitor.hh`'

Complete the `DefaultVisitor` class, the neutral traversals of ASTs. The `DefaultVisitor` must be a sound basis for your further work on the Tiger compiler.

'`src/ast/print-visitor.hh`'

The `PrintVisitor` class must be written entirely.

[9] `http://www.lrde.epita.fr/~akim/compil/download/2006-tc-2.0.tar.bz2`.

[10] `http://www.lrde.epita.fr/~akim/compil/download/2006-tc-1.0-2.0.diff`.

4.4.5 T2 FAQ

‘`src/ast/README`’
: This file should not have been shipped. Do not take its content too seriously.

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.

`bison`
: Be sure to read its dedicated section: Section 5.6 [Flex & Bison], page 125.

Memory leaks in the parser during error recovery
: The generated parser from current versions of Bison does not offer a means to reclaim the memory of the symbols that are thrown away. Hence, memory leaks won't be tested on invalid input.

Memory leaks in the standard containers
: See Section 5.5 [Valgrind], page 123, for a pointer to the explanation and solution.

4.4.6 T2 Improvements

Possible improvements include:

Use CVS Bison
: CVS Bison, i.e., the development version of Bison, provides means to release symbols during error recovery in the C++ parsers. But much work remains to be done. You may either try it, or even improve Bison itself. Contact Akim.

A more Elaborate Visitor Hierarchy
: See [spot], page 120, for an example on how the use of a clean visitor hierarchy and auxiliary functions enhances the readability, maintainability, and expressiveness of the code.

Using Generic Visitors
: Andrei Alexandrescu has done a very interesting work on generic implementation of Visitors, see [Modern C++ Design], page 119. 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 118.

4.5 T3, Computing the Escaping Variables

This section was updated for Tiger 2004. The project will be taken on Friday, March 15th, at noon.

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`’.

Be sure to read the chapter “Escapes” in the lecture notes.

4.5.1 T3 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 Class Template
Writing a Visitor from scratch
Using methods from parents classes
Inner functions and their impact on memory management at runtime

4.5.2 T3 Samples

This example demonstrates the computation and display of escaping variables/formals. Notice that by default, all variable 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 escaping := "I rule the world!\n"
   var not_escaping := "Peace on Earth for humans of good will.\n"
   function print_slogan (not_escaping: string) =
      (print (not_escaping); print (escaping))
in
   print_slogan (not_escaping)
end
```

File 4.18: ‘variable-escapes.tig’

```
$ tc -EeE variable-escapes.tig
/* == Escapes. == */
let
   var /* escaping */ escaping := "I rule the world!\n"
in
   let
      var /* escaping */ not_escaping := "Peace on Earth for humans of good will.\n"
   in
      let
         function print_slogan (/* escaping */ not_escaping : string) =
            (
               print (not_escaping);
               print (escaping)
            )
      in
         print_slogan (not_escaping)
      end
   end
end
/* == Escapes. == */
let
   var /* escaping */ escaping := "I rule the world!\n"
in
   let
      var not_escaping := "Peace on Earth for humans of good will.\n"
   in
      let
         function print_slogan (not_escaping : string) =
            (
               print (not_escaping);
               print (escaping)
            )
```

```
        in
           print_slogan (not_escaping)
        end
     end
end
```

Example 26: *tc -EeE variable-escapes.tig*

Run your compiler on '`merge.tig`' and to study its output. There is a number of silly mistakes that people usually do on T3: 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.*

4.5.3 T3 Code To Write

`ast::PrintVisitor`

Be sure to display the '`/* escaping */`' flag where needed, and *only* where needed. If you don't pay attention, you might display meaningless flags due to implementation details.

`escapes::EscapesVisitor`

Write the class `escapes::EscapesVisitor` in '`src/escapes/escapes-visitor.hh`'.

You are suggested to implement three additional classes:

`Definition`

An abstract class which is used to instantiate the template class `symbol::Table` into `Table <Definition>`.

`escape_set` (*void*) [virtual void]
Sets the escape to true.

`int` *depth_* [Variable]
Depth at which this object has been created.

`depth_get` () *const* [int]
Returns the depth associated to this `Definition` object.

`VariableDefinition`

Inherits from `Definition`. It has one additional attribute, a `VarDec &`. The method `escape_set` is implemented, and when invoked, set the `escapes` flags of the corresponding `VarDec`.

`FormalDefinition`

Inherits from `Definition`. To be designed by yourself. Do not forget that the `ast` class used to register formals is used elsewhere, and it would be a pity that your implementation makes no difference... Be sure to write a test that verifies that your implementation is not abused. I have one such test...

Equip `ast` All the sites where variables and formals (i.e., the arguments of the functions being *defined*, not being used) are introduced must be equipped with the `escape_get` and `escape_set` methods. Most probably the code was already given, and is using '`const_cast`'s; try to use `mutable` instead.

Modify the code so that each definition of an escaping variable/formal is *preceded* by the comment '`/* escaping */`' if the flag `display_escapes_p` is true. See the item "Driver" for an example.

4.5.4 T3 FAQ

Dwarf errors

It appears that at EPITA, the linker is unable to read the output of G++ 3.2 when given '`-ggdb`'. So don't pass it.

4.5.5 T3 Improvements

Possible improvements include:

4.6 T4, Type Checking

This section was last updated for EPITA-2005 on 2003-04-08.

2005-T4 delivery is Friday, April 25th 2003 at noon.

At the end of this stage, the compiler type checks Tiger programs. Clear error messages are required.

Relevant lecture notes include '`type-checking.pdf`'[11].

4.6.1 T4 Goals

Things to learn during this stage that you should remember:

– What type-checking is

4.6.2 T4 Samples

Type checking is optional, invoked by '`--types-check`' or '`-T`':

```
1 + "2"
```

File 4.19: '`int-plus-string.tig`'

```
$ tc int-plus-string.tig
```

Example 28: *tc int-plus-string.tig*

```
$ tc int-plus-string.tig --types-check
error int-plus-string.tig:1.0-6: type mismatch
error   right operand type: string
error   expected type: int
⇒4
```

Example 29: *tc int-plus-string.tig --types-check*

When there are several type errors, it is admitted that some remain hidden by others.

```
unknown_function (unknown_variable)
```

File 4.20: '`unknowns.tig`'

```
$ tc unknowns.tig --types-check
error unknowns.tig:1.0-34: unknown function: unknown_function
⇒4
```

Example 31: *tc unknowns.tig --types-check*

Be sure to check the type of all the constructs.

```
if 1 then 2
```

File 4.21: '`bad-if.tig`'

```
$ tc bad-if.tig --types-check
error bad-if.tig:1.0-10: type mismatch
```

[11] `http://www.lrde.epita.fr/~akim/compil/lecture-notes/type-checking.pdf`.

```
error   then clause type: int
error   else clause type: void
⇒4
```

Example 33: *tc bad-if.tig --types-check*

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 4.22: 'mutuals.tig'

```
$ tc mutuals.tig --types-check
```

Example 35: *tc mutuals.tig --types-check*

In case you are interested, the result is:

```
$ tc -H mutuals.tig >mutuals.hir
```

Example 36: *tc -H mutuals.tig >mutuals.hir*

```
$ havm mutuals.hir
22
```

Example 37: *havm mutuals.hir*

4.6.3 T4 Given Code

Some code is provided: '2005-tc-4.3.tar.bz2'[12]. The transition from the previous versions can be done thanks to the following diffs: '2005-tc-2.1-4.0.diff'[13], '2005-tc-4.0-4.1.diff'[14], '2005-tc-4.1-4.2.diff'[15], '2005-tc-4.2-4.3.diff'[16]. See Section 3.2.3 [src/misc], page 30.

4.6.4 T4 Code to Write

What is to be done.

`symbol::Table< class Entry_T >`

Write the class template `symbol::Table` in 'src/symbol/table.hh' which is a table of symbols dedicated to storing some data which type is `Entry_T *`. In short, it maps a `symbol::Symbol` to an `Entry_T *` (that should ring a bell...). You are encouraged to implement something simple, based on stacks (see `std::stack` or `std::list`) and maps (see `std::map`).

`symbol::Table` is a class template as it is used by virtually all the AST visitors (e.g., `escapes::EscapesVisitor`, `type::TypeVisitor`, `translate::TranslateVisitor` etc.)

`symbol::Table` must provide this interface:

[12] http://www.lrde.epita.fr/~akim/compil/download/2005-tc-4.3.tar.bz2.
[13] http://www.lrde.epita.fr/~akim/compil/download/2005-tc-2.1-4.0.diff.
[14] http://www.lrde.epita.fr/~akim/compil/download/2005-tc-4.0-4.1.diff.
[15] http://www.lrde.epita.fr/~akim/compil/download/2005-tc-4.1-4.2.diff.
[16] http://www.lrde.epita.fr/~akim/compil/download/2005-tc-4.2-4.3.diff.

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

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

`put` (*Symbol key, Entry_T* **&** *value*) [void]
Associate *value* to *key* in the current scope.

`get` (*Symbol key*) *const* [Entry_T *]
If *key* was associated to some `Entry_T` in the open scopes, return the most recent insertion. Otherwise return the empty pointer.

`print` (*std::ostream* **&** *ostr*) *const* [void]
Send the content of this table on *ostr in a readable manner*, the top of the stack being displayed *last*.

'`src/type/types.hh`'
The Singletons `type::String`, `type::Int`, and `type::Void` are to be implemented. Using templates would be particularly appreciated to factor the code between the four singleton classes.

`type::Named` is almost entirely given.

`type::Array` is even simpler than the four Singletons.

`type::Record` is somewhat incomplete.

Pay extra attention to the implementation of `type::operator== (const Type& a, const Type& b)`, `type::Type::assignable_to` and `type::Type::comparable_to`.

'`src/type/type-entry.hh`'
This file is really a empty nutshell, so we give it complete so that you concentrate on things that matter. Nonetheless you will be asked questions on this file, so study it.

'`src/type/type-env.hh`'
The constructor of `type::TypeEnv` must be completed: it must fill the environment with the definition of builtin types and functions. See the Tiger Reference Manual.

The handling of types is left as an example, you still have to implement the variables and functions support.

`type::TypeVisitor`
Of course this is the most tricky part. I hope there are enough comments in there so that you understand what is to be done. Please, post your questions and help me improve it.

Tasks Each module *Foo* exports its tasks via the file '*foo*`/`*foo*`-tasks.hh`'. You must clean up your code to use the latest sources for `Tasks`, and make sure that your '`configure.ac`' no longer includes '*foo*`/lib`*foo*`.hh`' in '`src/modules.hh`'.

4.6.5 T4 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 4.23: ‘is_devil.tig’

```
$ tc is_devil.tig --types-check
error is_devil.tig:1.8-33: type mismatch
error   then clause type: int
error   else clause type: string
error is_devil.tig:1.0-33: type mismatch
error   left operand type: string
error   right operand type: int
⇒4
```

Example 39: *tc is_devil.tig --types-check*

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`.

4.6.6 T4 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. Note that this type is not usable though.

kinds (`kind_get`, etc.)

Some of the ast components have features such as `kind_`, `kind_get` and so forth. These are to be used only in T5, you don't have to complete them now.

The `TypeVisitor` is not a `ConstVisitor`

One of the tasks of the type checking is to pass additional information to the translation. For instance, since `<` is overloaded (for integers and strings), the translation needs to know the types of the arguments. In a traditional compiler, type checking and translation would be performed simultaneously, but our Tiger Compiler, in order to simplify its architecture, has two different passes for each. Hence, the `TypeVisitor` will have to leave notes on the AST for the `TranslateVisitor`, therefore it cannot be a const visitor once T5 implemented. It can perfectly be const during T4.

4.6.7 T4 Improvements

Possible improvements include:

A `Runtime` class

The Tiger language specify a “standard library” comprising functions such as `print`, `getchar` and so forth. This stage requires to know the signature of these builtins to type check their uses, the stage T5 need the signature to implement correctly the call protocol, and some other parts of the compiler might need them.

It is a bad designed that the knowledge about these builtins is scattered in various places, but to avoid departing too much from Appel's modelisation[17], we kept it this way. You might want to make one anyway.

More flexibility on the runtime

Maybe the runtime should actually be declared as a form of "prelude", as Haskell people call it: a file that is always read, initializing the environment. It would contains things such as:

```
function substring (string: string, first: int, count: int) : string
  = __builtin "substring"
/* ... */
```

This would keep all things together, and would make it easier to implement extensions in the language. For instance, one could add:

```
function abort () = __builtin "abort"
```

Walking that track goes beyond the simplicity and minimality that the Tiger Projects aims at for (generalist) first year students.

4.7 T5, Translating to the High Level Intermediate Representation

This section was last updated for EPITA-2005 on 2003-06-10.

2005-T56 delivery is Friday, June 20th, at noon.

At the end of this stage the compiler translates the AST into the high level intermediate representation, HIR for short. And, of course, all the errors of previous stages have been fixed.

Relevant lecture notes include '`intermediate.pdf`'[18].

4.7.1 T5 Goals

Things to learn during this stage that you should remember:

"Functional" programming in C++

See for instance the use of `std::binary_function` to sort `Temp*`.

Traits Traits are a useful technique that allows to write (compile time) functions ranging over types. See Section A.1 [Glossary], page 129.

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).

4.7.2 T5 Samples

T5 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.

---

[17] This statement is unfair to Andrew Appel: since in his modelisation type checking and translation are performed in a single step, the information about the builtins remains in a single place.

[18] `http://www.lrde.epita.fr/~akim/compil/lecture-notes/intermediate.pdf`.

4.7.2.1 T5 Primitive Samples

This example is probably the simplest Tiger program.

```
0
```

File 4.24: '0.tig'

```
$ tc --hir-display 0.tig
/* == High Level Intermediate representation. == */
# Routine: Main
label Main
# Prologue
# Body
sxp
    const 0
# Epilogue
label end
```

Example 41: *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 4.25: 'arith.tig'

```
$ tc -H arith.tig
/* == High Level Intermediate representation. == */
# Routine: Main
label Main
# Prologue
# Body
sxp
    binop (+)
        const 1
        binop (*)
            const 2
            const 3
# Epilogue
label end
```

Example 43: *tc -H arith.tig*

You should use `havm` to exercise your output.

```
$ tc -H arith.tig >arith.hir
```

Example 44: *tc -H arith.tig >arith.hir*

```
$ havm arith.hir
```

Example 45: *havm arith.hir*

Unfortunately, without actually printing something, you won't see the final result, which means you need to implement calls. Fortunately, you can ask `havm` for a verbose execution:

```
$ havm --trace arith.hir
error plaining
error unparsing
error checking
error checkingLow
```

```
error evaling
error  call ( name Main ) []
error 8.8-8.15:  const 1
error 10.12-10.19:  const 2
error 11.12-11.19:  const 3
error 9.8-11.19:  binop (*) 2 3
error 7.4-11.19:  binop (+) 1 6
error 6.0-11.19:  sxp 7
error  end call ( name Main ) [] = 0
```

Example 46: *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 4.26: '`if-101.tig`'

```
$ tc -H if-101.tig
/* == High Level Intermediate representation. == */
# Routine: Main
label Main
# Prologue
# Body
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
# Epilogue
label end
```

Example 48: *tc -H if-101.tig*

And even more difficult control structure uses:

```
while 101
  do (if 102 then break)
```

File 4.27: '`while-101.tig`'

```
$ tc -H while-101.tig
/* == High Level Intermediate representation. == */
# Routine: Main
label Main
```

```
# Prologue
# Body
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
# Epilogue
label end
```

Example 50: *tc -H while-101.tig*

4.7.2.2 T5 Optimizing Cascading If

Our compiler optimizes the number of jumps needed to compute nested `if`, using '`translate::Ix`' where a plain use of '`translate::Cx`', '`Nx`', and '`Ex`' is possible, but less efficient.

Consider the following sample:

```
if 11 | 22 then print ("OK\n")
```

File 4.28: '`boolean.tig`'

a naive implementation will probably produce too many successive `cjump` instructions:

```
$ tc --hir-naive -H boolean.tig
/* == High Level Intermediate representation. == */

label l3
        "OK\n"
# Routine: Main
label Main
# Prologue
```

```
# Body
seq
    cjump ne
        eseq
        seq
            cjump ne
                const 11
                const 0
                name l0
                name l1
            label l0
            move
                temp t0
                const 1
            jump
                name l2
            label l1
            move
                temp t0
                const 22
            jump
                name l2
            label l2
        seq end
            temp t0
        const 0
        name l4
        name l5
    label l4
    sxp
        call
            name print
            name l3
        call end
    jump
        name l6
    label l5
    sxp
        const 0
    jump
        name l6
    label l6
seq end
# Epilogue
label end
```

Example 52: *tc --hir-naive -H boolean.tig*

```
$ tc --hir-naive -H boolean.tig >boolean-1.hir
```

Example 53: *tc --hir-naive -H boolean.tig >boolean-1.hir*

```
$ havm --profile boolean-1.hir
error /* Profiling.  */
error fetches from temporary : 1
```

```
error fetches from memory    : 0
error binary operations      : 0
error function calls         : 1
error stores to temporary    : 1
error stores to memory       : 0
error jumps                  : 2
error conditional jumps      : 2
error /* Execution time.  */
error number of cycles : 16
OK
```

Example 54: *havm --profile boolean-1.hir*

If you carefully analyze the cause of this pessimization, it is related to the computation of an intermediary expression (the value of '`11 | 22`') which is later decoded as a condition. A proper implementation will produce:

```
$ tc -H boolean.tig
/* == High Level Intermediate representation. == */

label l0
        "OK\n"
# Routine: Main
label Main
# Prologue
# Body
seq
    seq
        cjump ne
            const 11
            const 0
            name l4
            name l5
        label l4
        cjump ne
            const 1
            const 0
            name l1
            name l2
        label l5
        cjump ne
            const 22
            const 0
            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
# Epilogue
label end
```

Example 55: *tc -H boolean.tig*

```
$ tc -H boolean.tig >boolean-2.hir
```

Example 56: *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 57: *havm --profile boolean-2.hir*

4.7.2.3 T5 Builtin Calls Samples

But the game becomes more interesting when you implement function calls (which is easier than compiling functions). `print_int` is probably the first builtin to implement:

```
(print_int (101); print ("\n"))
```

File 4.29: '`print-101.tig`'

```
$ tc -H print-101.tig >print-101.hir
```

Example 59: *tc -H print-101.tig >print-101.hir*

```
$ havm print-101.hir
101
```

Example 60: *havm print-101.hir*

Complex values, arrays and records, also need calls to the runtime system:

```
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 4.30: '`print-list.tig`'

```
$ tc -H print-list.tig
/* == High Level Intermediate representation. == */

label l0
        "\n"
# Routine: Main
```

```
label Main
# Prologue
move
    temp t2
    temp fp
move
    temp fp
    temp sp
move
    temp sp
    binop (-)
        temp sp
        const 4
# Body
seq
    move
        mem
            temp $fp
        eseq
        seq
            move
                temp t1
                call
                    name malloc
                    const 8
                call end
            move
                mem
                    binop (+)
                        temp t1
                        const 0
                const 1
            move
                mem
                    binop (+)
                        temp t1
                        const 4
                eseq
                seq
                    move
                        temp t0
                        call
                            name malloc
                            const 8
                        call end
                    move
                        mem
                            binop (+)
                                temp t0
                                const 0
                        const 2
                    move
```

```
                        mem
                            binop (+)
                                temp t0
                                const 4
                        const 0
                seq end
                    temp t0
        seq end
            temp t1
    seq
        sxp
            call
                name print_int
                mem
                    binop (+)
                        mem
                            binop (+)
                                mem
                                    temp $fp
                                const 4
                        const 0
            call end
        sxp
            call
                name print
                name l0
            call end
    seq end
seq end
# Epilogue
move
    temp sp
    temp fp
move
    temp fp
    temp t2
label end
```

Example 62: *tc -H print-list.tig*

```
$ tc -H print-list.tig >print-list.hir
```

Example 63: *tc -H print-list.tig >print-list.hir*

```
$ havm print-list.hir
2
```

Example 64: *havm print-list.hir*

4.7.2.4 T5 Samples with Variables

Here is an example which demonstrates the usefulness of information about escapes: when escaping variables are not computed, they are all 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 4.31: ‘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 (-)
        temp sp
        const 12
# Body
seq
    move
        mem
            temp $fp
        const 1
    seq
        move
            mem
                binop (+)
                    temp $fp
                    const -4
            const 2
        seq
            move
                mem
                    binop (+)
                        temp $fp
                        const -8
                const 3
            seq
```

```
                    move
                        mem
                            temp $fp
                        const 2
                    move
                        mem
                            binop (+)
                                temp $fp
                                const -8
                        binop (+)
                            binop (+)
                                mem
                                    temp $fp
                                mem
                                    binop (+)
                                        temp $fp
                                        const -4
                            mem
                                binop (+)
                                    temp $fp
                                    const -8
                    sxp
                        call
                            name print_int
                            mem
                                binop (+)
                                    temp $fp
                                    const -8
                        call end
                    sxp
                        call
                            name print
                            name l0
                        call end
                seq end
            seq end
        seq end
    seq end
    # Epilogue
    move
        temp sp
        temp fp
    move
        temp fp
        temp t0
    label end
```

Example 66: *tc -H vars.tig*

But 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
    move
        temp t0
        const 1
    seq
        move
            temp t1
            const 2
        seq
            move
                temp t2
                const 3
            seq
                move
                    temp t0
                    const 2
                move
                    temp t2
                    binop (+)
                        binop (+)
                            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
    seq end
seq end
# Epilogue
label end
```

Example 67: *tc -eH vars.tig*

```
$ tc -eH vars.tig >vars.hir
```

Example 68: *tc -eH vars.tig >vars.hir*

```
$ havm vars.hir
7
```

Example 69: *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 4.32: '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 (-)
        temp sp
        const 8
move
    mem
        temp $fp
    temp i0
move
    mem
        binop (+)
            temp $fp
            const -4
    temp i1
# Body
move
    temp $v0
    eseq
    seq
        cjump eq
            mem
                binop (+)
                    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 (*)
                    mem
                        binop (+)
                            temp $fp
                            const -4
                    call
                        name l0
                        mem
                            temp $fp
                        binop (-)
                            mem
                                binop (+)
                                    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
        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
# Epilogue
label end
```

Example 71: *tc -H fact15.tig*

```
$ tc -H fact15.tig >fact15.hir
```

Example 72: *tc -H fact15.tig >fact15.hir*

```
$ havm fact15.hir
2004310016
```

Example 73: *havm fact15.hir*

And finally, you should support escaping variables. See ⟨undefined⟩ [variable-escapes.tig], page ⟨undefined⟩.

```
$ tc -eH variable-escapes.tig
/* == High Level Intermediate representation. == */

label l0
        "I rule the world!\n"

label l1
        "Peace on Earth for humans of good will.\n"
# Routine: print_slogan
label l2
# Prologue
move
    temp t2
    temp fp
move
    temp fp
    temp sp
move
    temp sp
    binop (-)
        temp sp
        const 4
move
    mem
        temp $fp
    temp i0
move
    temp t1
    temp i1
# Body
seq
    sxp
        call
            name print
```

```
            temp t1
        call end
    sxp
        call
            name print
            mem
                mem
                    temp $fp
        call end
seq end
# 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 (-)
        temp sp
        const 4
# Body
seq
    move
        mem
            temp $fp
        name l0
    seq
        move
            temp t0
            name l1
        sxp
            call
                name l2
                temp $fp
                temp t0
            call end
    seq end
seq end
# Epilogue
```

```
move
    temp sp
    temp fp
move
    temp fp
    temp t3
label end
```

Example 74: *tc -eH variable-escapes.tig*

4.7.3 T5 Given Code

Some code is provided, see Section 4.8.3 [T6 Given Code], page 92. See Section 3.2.9 [src/temp], page 32, Section 3.2.10 [src/tree], page 32, Section 3.2.11 [src/frame], page 32, Section 3.2.12 [src/translate], page 33.

4.7.4 T5 Code to Write

You are encouraged to try first very simple examples: '`nil`', '`1 + 2`', '`"foo" < "bar"`' etc. Then consider supporting variables, and finally handle the case of the functions.

Driver
: The driver must performs the translation when given '`--hir-compute`', but displays the result iff the option '`-H`' was given. Obviously, an input that has not been type-checked cannot be translated, so '`--hir-compute`' implies '`--types-check`'.

`TypeVisitor`
: The `TranslateVisitor` often needs additional type information to proceed, especially expression versus instruction. Hence, you'll have to update the `TypeVisitor` to leave notes on the AST using `kind_set` and so forth.

'`src/translate/fragment.hh`'
: There remains to implement `translate::ProcFrag::print` which outputs the routine themselves *plus* the glue code (allocating the frame etc.).

'`src/translate/level-env.hh`'
: Code is missing. In particular, bear in mind that the initial environment is *not* empty...

'`src/translate/translation.hh`'
: There are many holes to fill.

'`src/translate/translate-visitor.hh`'
: There are holes to fill.

4.7.5 T5 Options

This section documents possible extensions you could implement in T5.

4.7.5.1 T5 Bounds Checking

Implementing bounds checking is quite simple: have the program die when the program accesses an invalid subscript in an array. For instance, the following code "succeeds" with a non-bounds-checking compiler.

```
let type int_array = array of int
    var  size  := 2
    var  arr1  := int_array [size] of 0
    var  arr2  := int_array [size] of 0
    var  two   := 2
    var  m_one := -1
in
  arr1[two]   := 3;
  arr2[m_one] := -1;

  print_int (arr1[1]);
  print ("\n");
  print_int (arr2[0]);
  print ("\n")
end
```

File 4.33: 'bounds-violation.tig'

```
$ tc -H bounds-violation.tig >bounds-violation.hir
```

Example 76: *tc -H bounds-violation.tig >bounds-violation.hir*

```
$ havm bounds-violation.hir
-1
3
```

Example 77: *havm bounds-violation.hir*

When run with '`--bounds-checking`', your compiler produces code that diagnoses such cases, and exits with status 120. Something like:

```
error bounds-violation.tig:8.2-17: index out of arr1 bounds (0 .. 1): 2
⇒120
```

4.7.5.2 T5 Optimizing Static Links

Warning: this optimization is *difficult* to do it 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 "Modern Implementation of Compilers"), 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:

```
$ tc -E fact.tig
/* == Escapes. == */
let
   function fact (/* escaping sl *//* escaping */ n : int) : int =
      if  (n = 0)
         then 1
         else  (n * fact ( (n - 1)))
in
   fact (10)
end

$ tc -eE fact.tig
/* == Escapes. == */
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:

```
$ tc -eE escaping-sl.tig
/* == Escapes. == */
let
   var      toto := 1
   function outer (/* escaping sl */) : int =
     let function inner (/* sl */) : int = toto
     in inner () end
in
  outer ()
end
```

   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      toto := 1
   function outer () : int =
     let function inner () : int = toto
     in inner () end
   function sister () : int = outer ()
in
  sister ()
end
```

4.7.6 T5 Improvements

Possible improvements include:

Using `boost::variant` to implement `Temp` and `Label`
: The two sibling classes `Temp` and `Label` clearly implement a `union` in the sense of C. But C++ virtually forbids objects in classes: only pod is allowed, this is why our design does not use it.

Some people have worked hard to implement `union` 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)[19], Discriminated Unions (ii)[20], Discriminated Unions (iii)[21] for an introduction to the techniques. We would use `boost::variant` (see [Boost.org], page 115) if this material was not too advanced for first year students.

I strongly encourage you to read these enlightening articles.

Implement 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 120, for an explanation on how this approach was successfully implemented. See the ATermLibrary[22] for a general implementation of maximally shared trees.

4.8 T6, Translating to the Low Level Intermediate Representation

This section was last updated for EPITA-2005 on 2003-05-15.

2005-T56 delivery is Friday, June 20th, at noon.

There will be no additional code: there is no "holes" to fill, you have to write the whole thing. Consequently, you may start T6 as soon as you want.

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`'[23].

4.8.1 T6 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[24].

[19] `http://www.cuj.com/documents/s=7984/cujcexp2004alexandr/`.
[20] `http://www.cuj.com/documents/s=7982/cujcexp2006alexandr/`.
[21] `http://www.cuj.com/documents/s=7980/cujcexp2008alexandr/`.
[22] `http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ATermLibrary`.
[23] `http://www.lrde.epita.fr/~akim/compil/lecture-notes/intermediate.pdf`.
[24] `http://www.google.com/search?q=term+rewriting+system`.

More "Functional" Programming in C++

A lot of T6 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`, `std::unary_function`, `std::not1` etc.

4.8.2 T6 Samples

There are several stages in T6.

4.8.2.1 T6 Canonicalization Samples

The first task in T6 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 4.34: '`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
    temp t2
    temp fp
move
    temp fp
    temp sp
move
    temp sp
    binop (-)
        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
        move
            temp t3
            const 0
        sxp
            call
                name l0
                temp $fp
                const 1
                eseq
                    move
```

```
                    temp t3
                    binop (+)
                        temp t3
                        const 1
                temp t3
        call end
seq end
# Epilogue
label end
```

Example 79: *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 (-)
        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 (+)
                temp t3
                const 1
        sxp
            call
                name l0
                temp t5
                const 1
                temp t3
            call end
        label l6
    seq end
```

```
# Epilogue
label end
```

Example 80: *tc -eL preincr-1.tig*

But please note 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 4.35: '`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 (-)
        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 (+)
                temp t3
                const 1
        sxp
            call
```

```
            name l0
            temp t5
            temp t6
            temp t3
        call end
    label l6
seq end
# Epilogue
label end
```

Example 82: *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 83: *tc -eH preincr-2.tig >preincr-2.hir*

```
$ havm preincr-2.hir
0, 1
```

Example 84: *havm preincr-2.hir*

```
$ tc -eL preincr-2.tig >preincr-2.lir
```

Example 85: *tc -eL preincr-2.tig >preincr-2.lir*

```
$ havm preincr-2.lir
0, 1
```

Example 86: *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 dead wrong: '`temp t`' is a subexpression: 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 4.36: 'move-mem.tig'

```
$ tc -eL move-mem.tig >move-mem.lir
```

Example 88: *tc -eL move-mem.tig >move-mem.lir*

```
$ havm move-mem.lir
100
200
```

Example 89: *havm move-mem.lir*

You also ought to get rid of nested calls:

```
print (chr (ord ("\n")))
```

File 4.37: '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 91: *tc -L nested-calls.tig*

In fact there are only two valid call forms: '`sxp (call (...))`', and '`move (temp (...), call (...))`'.

Note that, 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 4.38: '`seq-point.tig`'

should always produce:

```
$ tc -L seq-point.tig >seq-point.lir
```

Example 93: *tc -L seq-point.tig >seq-point.lir*

```
$ havm seq-point.lir
1 -> 2
2 -> 3
3 -> 4
14
```

Example 94: *havm seq-point.lir*

independently of the what IR you ran. Note that *it has nothing to do with the precedence of the operators!*

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;
```

```
}
```

4.8.2.2 T6 Scheduling Samples

Once your `eseq` and `call` canonicalized, normalize `cjump`s: 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 4.39: '`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 96: *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 4.40: '`broken-while.tig`'

```
$ tc -H broken-while.tig
/* == High Level Intermediate representation. == */
# Routine: Main
```

```
label Main
# Prologue
# Body
seq
    label l1
    seq
        cjump ne
            const 10
            const 0
            name l8
            name l9
        label l8
        cjump ne
            const 1
            const 0
            name l2
            name l0
        label l9
        cjump ne
            const 20
            const 0
            name l2
            name l0
    seq end
    label l2
    seq
        seq
            cjump ne
                const 30
                const 0
                name l6
                name l7
            label l6
            cjump ne
                const 1
                const 0
                name l3
                name l4
            label l7
            cjump ne
                const 40
                const 0
                name l3
                name l4
        seq end
        label l3
        jump
            name l0
        jump
            name l5
        label l4
        jump
```

```
            name l0
        label l5
    seq end
    jump
        name l1
    label l0
seq end
# Epilogue
label end
```

Example 98: *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 l8
        name l9
    label l9
    cjump ne
        const 20
        const 0
        name l2
        name l0
    label l0
    jump
        name l11
    label l2
    cjump ne
        const 30
        const 0
        name l6
        name l7
    label l7
    cjump ne
        const 40
        const 0
        name l3
        name l4
    label l4
    jump
        name l0
    label l3
    jump
        name l0
```

```
        label l6
        cjump ne
            const 1
            const 0
            name l3
            name l13
        label l13
        jump
            name l4
        label l8
        cjump ne
            const 1
            const 0
            name l2
            name l14
        label l14
        jump
            name l0
        label l11
    seq end
    # Epilogue
    label end
```

Example 99: *tc -L broken-while.tig*

4.8.3 T6 Given Code

Some code is provided: '`2005-tc-6.1.tar.bz2`'[25]. The transition from the previous versions can be done thanks to the following diffs: '`2005-tc-4.3-6.0.diff`'[26], '`2005-tc-6.0-6.1.diff`'[27].

It includes most of the canonicalization.

4.8.4 T6 Code to Write

Everything you need.

4.8.5 T6 Improvements

Possible improvements include:

4.9 T7, Instruction Selection

2005-T7 delivery is Friday, July 4th 2003 at noon.

This section was last updated for EPITA-2004 and EPITA-2005 on 2003-07-02.

Please note that the **2005-T7 delivery is an option**: there will be no grade, and a single upload will be accepted. The tests from T0 to T7 tests will be run on the tarball. The goal is to help you see your mistakes, and how your T7 is running to be able to proceed in peace onto T8. There will be no penalty if you don't take advantage of this possibility.

At the end of this stage, the compiler produces the very low level intermediate representation: ASSEM. This output is target dependent, and we aim at MIPS, as we use Mipsy to run it.

[25] `http://www.lrde.epita.fr/~akim/compil/download/2005-tc-6.1.tar.bz2`.

[26] `http://www.lrde.epita.fr/~akim/compil/download/2005-tc-4.3-6.0.diff`.

[27] `http://www.lrde.epita.fr/~akim/compil/download/2005-tc-6.0-6.1.diff`.

Relevant lecture notes include ‘`instr-selection.pdf`’[28].

4.9.1 T7 Goals

Things to learn during this stage that you should remember:

risc vs. cisc etc.
First introduction to assembly
Memory hierarchy/management at runtime

`auto_ptr` The Target module uses an `auto_ptr` pointer to manipulate without efforts a pointer to the current target, and to guarantee it is released (`delete`) at the end of the run.

4.9.2 T7 Samples

The goal of T7 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 4.11 [T9], page 105.

```
1 + 2 * 3
```

File 4.41: ‘`seven.tig`’

```
$ tc --inst-display seven.tig
# == Final assembler ouput. == #
# Routine: Main
t_main:
        move    t5, $s0
        move    t6, $s1
        move    t7, $s2
        move    t8, $s3
        move    t9, $s4
        move    t10, $s5
        move    t11, $s6
        move    t12, $s7
l0:
        li      t3, 2
        mul     t2, t3, 3
        li      t4, 1
        add     t1, t4, t2
l1:
        move    $s0, t5
        move    $s1, t6
        move    $s2, t7
        move    $s3, t8
        move    $s4, t9
        move    $s5, t10
        move    $s6, t11
        move    $s7, t12
```

Example 101: *`tc --inst-display seven.tig`*

Please, note that at this stage, the control flow analysis and the liveness analysis are not performed yet, therefore the compiler cannot know what registers are really to be saved. That’s why in the previous output it saves "uselessly" all the callee-save registers on `main`

[28] `http://www.lrde.epita.fr/~akim/compil/lecture-notes/instr-selection.pdf`.

entry. The next stage, which combines control flow analysis, liveness analysis, and register allocation, will make it useless. For your information, it results in:

```
$ tc -sI seven.tig
# == Final assembler ouput. == #
# Routine: Main
t_main:
        sw      $fp, ($sp)
        move    $fp, $sp
        sub     $sp, $sp, 8
        sw      $ra, -4 ($fp)
l0:
        li      $t0, 2
        mul     $t1, $t0, 3
        li      $t0, 1
        add     $t0, $t0, $t1
l1:

        lw      $ra, -4 ($fp)
        move    $sp, $fp
        lw      $fp, ($fp)
        jr      $ra
```

Example 102: *tc -sI seven.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 4.42: 'add.tig'

```
$ tc -e --mipsy-display add.tig
# == Final assembler ouput. == #
# Routine: add
l0:
        sw      $fp, -4 ($sp)
        move    $fp, $sp
        sub     $sp, $sp, 12
        sw      $ra, -8 ($fp)
        sw      $a0, ($fp)
        move    t0, $a1
        move    t1, $a2
        move    t7, $s0
        move    t8, $s1
        move    t9, $s2
        move    t10, $s3
        move    t11, $s4
        move    t12, $s5
        move    t13, $s6
        move    t14, $s7
l2:
        add     t6, t0, t1
```

```
        move    $v0, t6
l3:
        move    $s0, t7
        move    $s1, t8
        move    $s2, t9
        move    $s3, t10
        move    $s4, t11
        move    $s5, t12
        move    $s6, t13
        move    $s7, t14

        lw      $ra, -8 ($fp)
        move    $sp, $fp
        lw      $fp, -4 ($fp)
        jr      $ra

.data
l1:
        .word 1
        .asciiz "\n"
.text

# Routine: Main
t_main:
        sw      $fp, ($sp)
        move    $fp, $sp
        sub     $sp, $sp, 8
        sw      $ra, -4 ($fp)
        move    t19, $s0
        move    t20, $s1
        move    t21, $s2
        move    t22, $s3
        move    t23, $s4
        move    t24, $s5
        move    t25, $s6
        move    t26, $s7
l4:
        move    $a0, $fp
        li      t15, 2
        move    $a1, t15
        li      t16, 3
        move    $a2, t16
        jal     l0
        move    t4, $v0
        move    $a0, $fp
        li      t17, 1
        move    $a1, t17
        move    $a2, t4
        jal     l0
        move    t5, $v0
        move    $a0, t5
        jal     print_int
```

```
        la      t18, l1
        move    $a0, t18
        jal     print
l5:
        move    $s0, t19
        move    $s1, t20
        move    $s2, t21
        move    $s3, t22
        move    $s4, t23
        move    $s5, t24
        move    $s6, t25
        move    $s7, t26

        lw      $ra, -4 ($fp)
        move    $sp, $fp
        lw      $fp, ($fp)
        jr      $ra
```

Example 104: *tc -e --mipsy-display add.tig*

Once your function calls work properly, you can start using mipsy to check the behavior of your compiler.

```
$ tc -eH add.tig >add.hir
```

Example 105: *tc -eH add.tig >add.hir*

```
$ havm add.hir
6
```

Example 106: *havm add.hir*

Unfortunately, you need to adjust the output of 'tc', using `t123`, to mipsy conventions: '`$x123`'.

```
$ tc -eR --mipsy-display add.tig >add.instr
```

Example 107: *tc -eR --mipsy-display add.tig >add.instr*

```
$ sed -e's/\([^$a-z]\)t\([0-9][0-9]*\)/\1$x\2/g' add.instr >add.mipsy
```

Example 108: *sed -e's/\([^$a-z]\)t\([0-9][0-9]*\)/\1$x\2/g' add.instr >add.mipsy*

```
$ mipsy --unlimited-regs --execute add.mipsy
6
```

Example 109: *mipsy --unlimited-regs --execute add.mipsy*

You must also complete the runtime. No difference must be observable between a run with havm and another with mipsy:

```
substring ("", 1, 1)
```

File 4.43: '`substring-0-1-1.tig`'

```
$ tc -eH substring-0-1-1.tig >substring-0-1-1.hir
```

Example 111: *tc -eH substring-0-1-1.tig >substring-0-1-1.hir*

```
$ havm substring-0-1-1.hir
substring: arguments out of bounds
⇒120
```

Example 112: *havm substring-0-1-1.hir*

```
$ tc -e --mipsy-display substring-0-1-1.tig
# == Final assembler ouput. == #
```

```
.data
l0:
        .word 0
        .asciiz ""
.text

# Routine: Main
t_main:
        sw      $fp, ($sp)
        move    $fp, $sp
        sub     $sp, $sp, 8
        sw      $ra, -4 ($fp)
        move    t4, $s0
        move    t5, $s1
        move    t6, $s2
        move    t7, $s3
        move    t8, $s4
        move    t9, $s5
        move    t10, $s6
        move    t11, $s7
l1:
        la      t1, l0
        move    $a0, t1
        li      t2, 1
        move    $a1, t2
        li      t3, 1
        move    $a2, t3
        jal     substring
l2:
        move    $s0, t4
        move    $s1, t5
        move    $s2, t6
        move    $s3, t7
        move    $s4, t8
        move    $s5, t9
        move    $s6, t10
        move    $s7, t11

        lw      $ra, -4 ($fp)
        move    $sp, $fp
        lw      $fp, ($fp)
        jr      $ra
```

Example 113: *tc -e --mipsy-display substring-0-1-1.tig*

$ tc -eR --mipsy-display substring-0-1-1.tig >substring-0-1-1.instr

Example 114: *tc -eR --mipsy-display substring-0-1-1.tig >substring-0-1-1.instr*

$ sed -e's/\([^$a-z]\)t\([0-9][0-9]*\)/\1$x\2/g' substring-0-1-1.instr >substring-0-1-1.mipsy

Example 115: *sed -e's/\([^$a-z]\)t\([0-9][0-9]*\)/\1$x\2/g' substring-0-1-1.instr >substring-0-1-1.mipsy*

$ mipsy --unlimited-regs --execute substring-0-1-1.mipsy

```
substring: arguments out of bounds
⇒120
```

Example 116: *mipsy --unlimited-regs --execute substring-0-1-1.mipsy*

4.9.3 T7 Given Code

Below is listed where to find the tarball depending on your class. For more information about the T7 code delivered see Section 3.2.15 [src/target], page 34, Section 3.2.14 [src/assem], page 34, Section 3.2.16 [src/codegen], page 35.

2004-T7 A lot of code is provided. Actually, that's a real problem: since last year, the Tiger compiler has evolved a lot, and the integration of the new features will probably be painful. The most striking difference with last year being the Task handling.

The additional code is provided as:

- '`2004-tc-7.6.tar.bz2`'[29], the whole tarball.
- '`2004-tc-5.3-7.0.diff`'[30], '`2004-tc-5.3-7.3.diff`'[31], '`2004-tc-5.3-7.4.diff`'[32], '`2004-tc-5.3-7.5.diff`'[33], the differences with the latest tarball that was delivered.
- '`2004-tc-7.0-7.1.diff`'[34], '`2004-tc-7.1-7.2.diff`'[35], '`2004-tc-7.2-7.3.diff`'[36], '`2004-tc-7.3-7.4.diff`'[37], '`2004-tc-7.4-7.5.diff`'[38], '`2004-tc-7.5-7.6.diff`'[39], the differences with previous versions of the '`2004-tc-7`' tarball.

There are two ways to continue the projects:

minor upgrade
: If you do not want to upgrade your 2004 compiler into the 2005 form, just copy the relevant files from the tarball. See below. Adjust your driver so that '`--inst-compute`' and '`--inst-display`' be recognized. Of course, '`--inst-compute`' implies '`--lir-compute`'.

major upgrade
: You want to upgrade to the 2005 system. Expect massive surgery... Contrary to the previous case, I would recommend starting from the tarball we delivered, and copy your files into there. For a start, copy all the files that are *not* in the new tarball: it's probably not wrong.

```
# Be in the new tarball before running this.
for i in $(find .)
do
  if test ! -f ../my-old-working-directory/$i; then
```

[29] `http://www.lrde.epita.fr/~akim/compil/download/2004-tc-7.6.tar.bz2`.
[30] `http://www.lrde.epita.fr/~akim/compil/download/2004-tc-5.3-7.0.diff`.
[31] `http://www.lrde.epita.fr/~akim/compil/download/2004-tc-5.3-7.3.diff`.
[32] `http://www.lrde.epita.fr/~akim/compil/download/2004-tc-5.3-7.4.diff`.
[33] `http://www.lrde.epita.fr/~akim/compil/download/2004-tc-5.3-7.5.diff`.
[34] `http://www.lrde.epita.fr/~akim/compil/download/2004-tc-7.0-7.1.diff`.
[35] `http://www.lrde.epita.fr/~akim/compil/download/2004-tc-7.1-7.2.diff`.
[36] `http://www.lrde.epita.fr/~akim/compil/download/2004-tc-7.2-7.3.diff`.
[37] `http://www.lrde.epita.fr/~akim/compil/download/2004-tc-7.3-7.4.diff`.
[38] `http://www.lrde.epita.fr/~akim/compil/download/2004-tc-7.4-7.5.diff`.
[39] `http://www.lrde.epita.fr/~akim/compil/download/2004-tc-7.5-7.6.diff`.

```
      cp $i ../my-old-working-directory/$i
    fi
  done
```

And then, build it step by step.

2005-t7 The additional code is provided as:

- '`2005-tc-7.3.tar.bz2`'[40], the whole tarball.
- '`2005-tc-6.1-7.0.diff`'[41], '`2005-tc-6.1-7.1.diff`'[42], the differences with the latest tarball that was delivered.
- '`2005-tc-7.0-7.1.diff`'[43], '`2005-tc-7.1-7.2.diff`'[44], '`2005-tc-7.2-7.3.diff`'[45], the differences with previous versions of the '`2004-tc-7`' tarball.

4.9.4 T7 Code to Write

There is not much code to write:

- `Codegen::munchMove` ('`src/codegen/mips/codegen.cc`')
- `SpimAssembly::move_build` ('`src/codegen/mips/spim-assembly.cc`'): build a move instruction using MIPS 2000 standard instruction set.
- `SpimAssembly::binop_build` ('`src/codegen/mips/spim-assembly.cc`'): build arithmetic binary operations (addition, multiplication, etc.) using MIPS 2000 standard instruction set.
- `SpimAssembly::load_build`, `SpimAssembly::store_build` ('`src/codegen/mips/spim-assembly.cc`'): build a load (respectively a store) instruction using MIPS 2000 standard instruction set. Here, the indirect addressing mode is used.
- `SpimAssembly::cjump_build` ('`src/codegen/mips/spim-assembly.cc`'): translate conditional branch instructions (branch if equal, if lower than, etc.) into MIPS 2000 assembly.
- You have to complete the implementation of the runtime in '`src/codegen/mips/runtime.s`':

  `strcmp`

  `print_int`
  : This is the easiest, as it's just a call to the appropriate "syscall".

  `substring`
  `concat`
  : These ones are quite delicate.

  Information on MIPS 2000 assembly instructions may be found in SPIM manual.

Completing the following routines will be needed during register allocation only (see Section 4.11 [T9], page 105):

- `Codegen::allocate_frame` ('`src/codegen/mips/codegen.cc`')
- `Codegen::rewrite_program` ('`src/codegen/mips/codegen.cc`')

[40] `http://www.lrde.epita.fr/~akim/compil/download/2005-tc-7.3.tar.bz2`.
[41] `http://www.lrde.epita.fr/~akim/compil/download/2005-tc-6.1-7.0.diff`.
[42] `http://www.lrde.epita.fr/~akim/compil/download/2005-tc-6.1-7.1.diff`.
[43] `http://www.lrde.epita.fr/~akim/compil/download/2005-tc-7.0-7.1.diff`.
[44] `http://www.lrde.epita.fr/~akim/compil/download/2005-tc-7.1-7.2.diff`.
[45] `http://www.lrde.epita.fr/~akim/compil/download/2005-tc-7.2-7.3.diff`.

4.9.5 T7 Improvements

Possible improvements include:

4.10 T8, Liveness Analysis

2005-T8 delivery is Friday, July 18th 2003 at noon.

This section was last updated for EPITA-2004 and EPITA-2005 on 2003-07-02.

Relevant lecture notes include '`liveness.pdf`'[46].

4.10.1 T8 Goals

Things to learn during this stage that you should remember:

– Graph handling

4.10.2 T8 Samples

Branching is of course a most interesting feature to exercise:

```
1 | 2 | 3
```

File 4.44: '`ors.tig`'

```
$ tc -I ors.tig
# == Final assembler ouput. == #
# Routine: Main
t_main:
        move    t4, $s0
        move    t5, $s1
        move    t6, $s2
        move    t7, $s3
        move    t8, $s4
        move    t9, $s5
        move    t10, $s6
        move    t11, $s7
l5:
        li      t1, 1
        bne     t1, 0, l3
l4:
        li      t2, 2
        bne     t2, 0, l0
l1:
l2:
        j       l6
l0:
        j       l2
l3:
        li      t3, 1
        bne     t3, 0, l0
l7:
        j       l1
l6:
        move    $s0, t4
        move    $s1, t5
```

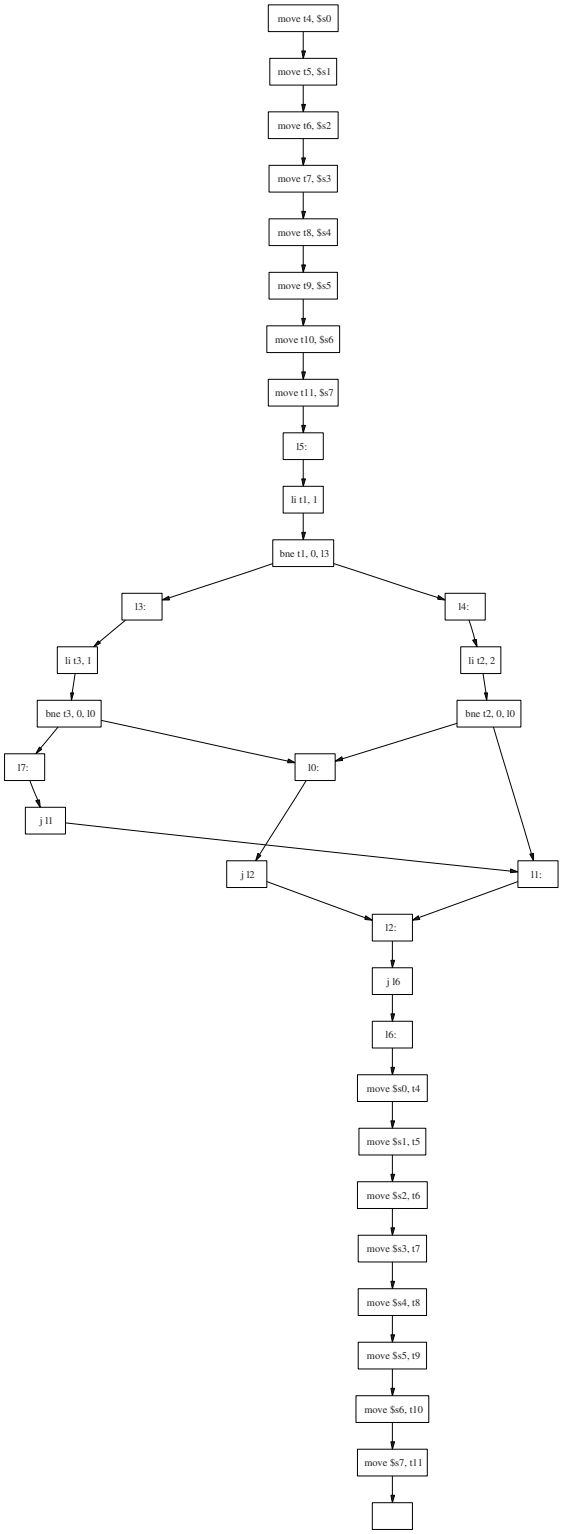
[46] `http://www.lrde.epita.fr/~akim/compil/lecture-notes/liveness.pdf`.

```
        move    $s2, t6
        move    $s3, t7
        move    $s4, t8
        move    $s5, t9
        move    $s6, t10
        move    $s7, t11
```

Example 118: *tc -I ors.tig*

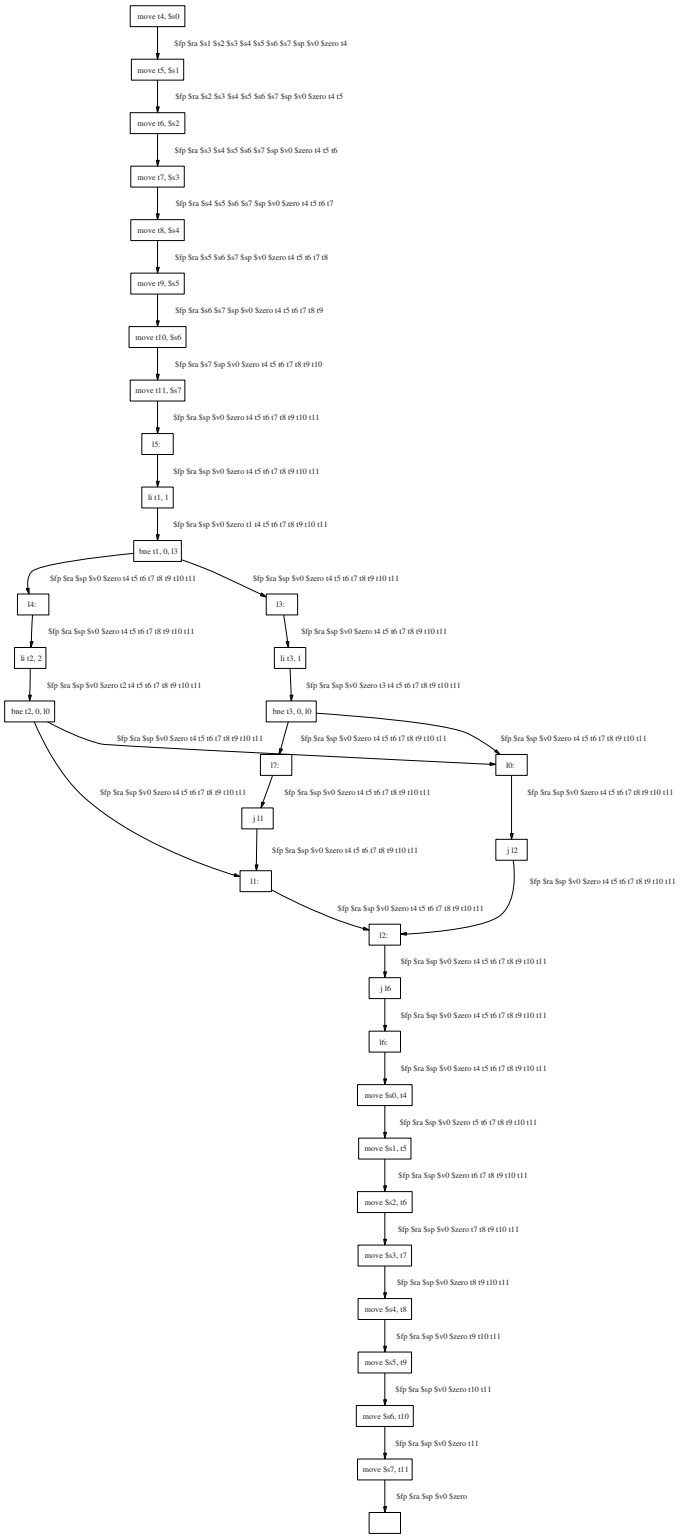
```
$ tc -F ors.tig
```

Example 119: *tc -F ors.tig*



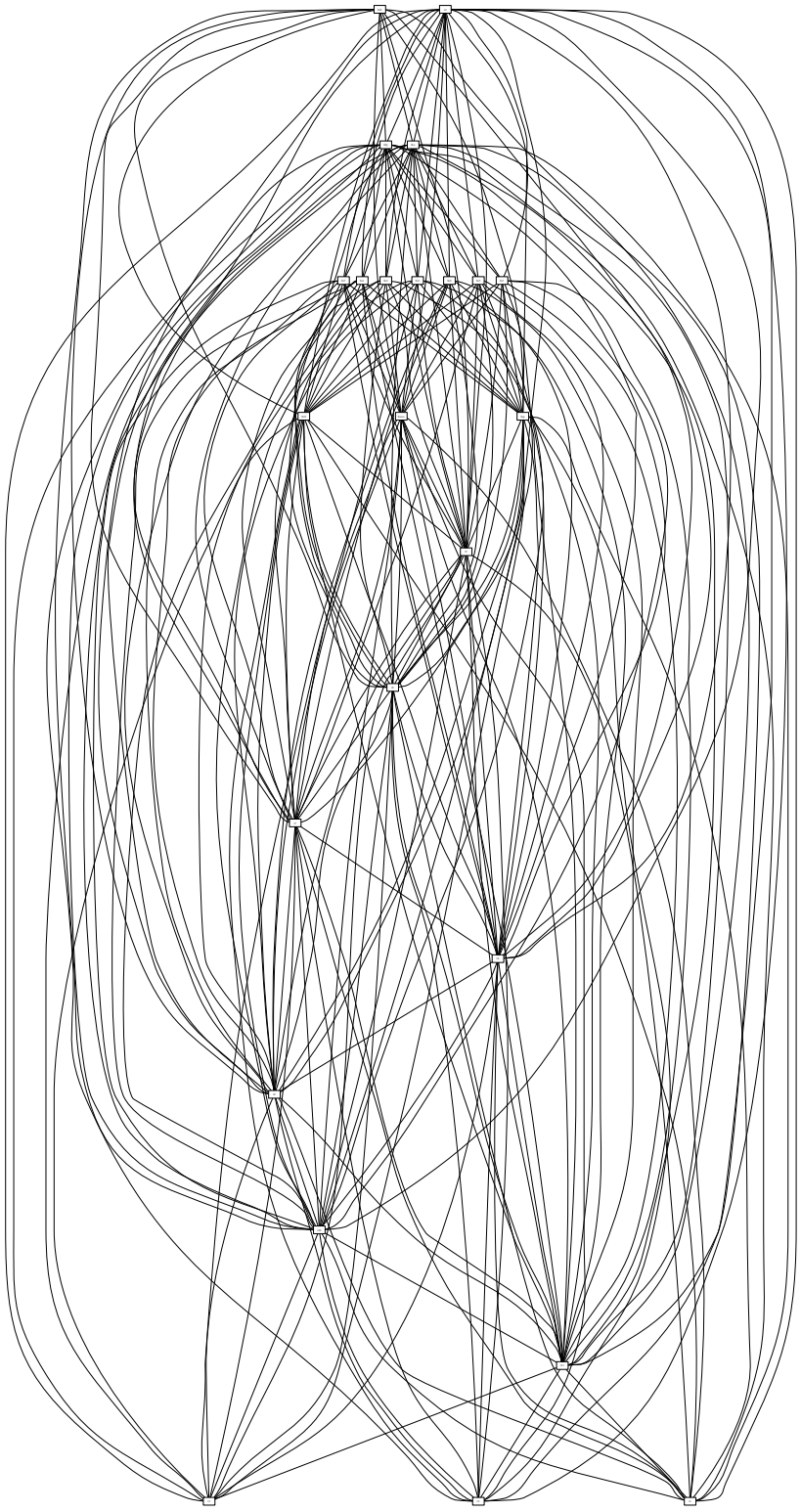
File 120: `'Main-Main-flow.dot'`

```
$ tc -V ors.tig
```
Example 121: `tc -V ors.tig`



File 122: `'Main-Main-liveness.dot'`

```
$ tc -N ors.tig
```
Example 123: `tc -N ors.tig`



File 124: `'Main-Main-interference.dot'`

4.10.3 T8 Given Code

You are provided with the following code:

– '`2005-tc-8.0.tar.bz2`'[47], the whole tarball.
– '`2005-tc-7.2-8.0.diff`'[48], '`2005-tc-7.3-8.0.diff`'[49], the differences with the latest tarball that was delivered.

To read the description of the new modules, see Section 3.2.19 [src/graph], page 37, Section 3.2.20 [src/liveness], page 37.

4.10.4 T8 Code to Write

'`src/graph/graph.hh`'
'`src/graph/graph.hxx`'
Implement the topological sort.

'`src/liveness/flowgraph.hh`'
Write the constructor, which is where the `FlowGraph` is actually constructed from the assembly fragments.

'`src/liveness/liveness.cc`'
Write the constructor, which is where the `Liveness` (a decorated `FlowGraph`) is built from assembly instructions.

'`src/liveness/interference-graph.cc`'
In `InterferenceGraph::compute_liveness`, build the graph.

4.10.5 T8 Improvements

Possible improvements include:

4.11 T9, Register Allocation

2005-T9 delivery is on Monday, September 8th 2003 at noon.

This section was last updated for EPITA-2004 and EPITA-2005 on 2003-08-19.

At the end of this stage, the compiler produces code that is runnable using Mipsy.

Relevant lecture notes include '`regalloc.pdf`'[50].

4.11.1 T8 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

4.11.2 T9 Samples

This section will not demonstrate the output of the option '`-S`', '`--asm-display`', since it includes the Tiger runtime, which is quite long. We simply use '`-I`', '`--instr-display`' which has the same effect *once the registers allocated*, i.e., once '`-s`', '`--asm-compute`' executed. 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. See ⟨undefined⟩ [seven.tig], page ⟨undefined⟩, for instance:

[47] `http://www.lrde.epita.fr/~akim/compil/download/2005-tc-8.0.tar.bz2`.
[48] `http://www.lrde.epita.fr/~akim/compil/download/2005-tc-7.2-8.0.diff`.
[49] `http://www.lrde.epita.fr/~akim/compil/download/2005-tc-7.3-8.0.diff`.
[50] `http://www.lrde.epita.fr/~akim/compil/lecture-notes/regalloc.pdf`.

```
$ tc -sI seven.tig
# == Final assembler ouput. == #
# Routine: Main
t_main:
        sw      $fp, ($sp)
        move    $fp, $sp
        sub     $sp, $sp, 8
        sw      $ra, -4 ($fp)
l0:
        li      $t0, 2
        mul     $t1, $t0, 3
        li      $t0, 1
        add     $t0, $t0, $t1
l1:

        lw      $ra, -4 ($fp)
        move    $sp, $fp
        lw      $fp, ($fp)
        jr      $ra
```

Example 125: *tc -sI seven.tig*

```
$ tc -S seven.tig >seven.s
```

Example 126: *tc -S seven.tig >seven.s*

```
$ mipsy --execute seven.s
```

Example 127: *mipsy --execute seven.s*

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. */
t1 -> $t0
t2 -> $t1
t3 -> $t0
t4 -> $t0
t5 -> $s0
t6 -> $s1
t7 -> $s2
t8 -> $s3
t9 -> $s4
t10 -> $s5
t11 -> $s6
t12 -> $s7
```

Example 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 4.45: '`print-seven.tig`'

```
$ tc -sI print-seven.tig
# == Final assembler ouput. == #
.data
l0:
```

```
        .word 1
        .asciiz "\n"
.text

# Routine: Main
t_main:
        sw      $fp, ($sp)
        move    $fp, $sp
        sub     $sp, $sp, 8
        sw      $ra, -4 ($fp)
l1:
        li      $t0, 2
        mul     $t1, $t0, 3
        li      $t0, 1
        add     $a0, $t0, $t1
        jal     print_int
        la      $a0, l0
        jal     print
l2:

        lw      $ra, -4 ($fp)
        move    $sp, $fp
        lw      $fp, ($fp)
        jr      $ra
```

Example 130: *tc -sI print-seven.tig*

```
$ tc -S print-seven.tig >print-seven.s
```

Example 131: *tc -S print-seven.tig >print-seven.s*

```
$ mipsy --execute print-seven.s
7
```

Example 132: *mipsy --execute 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 4.46: ‘print-many.tig’

```
$ tc -eIs --tempmap-display -I --time-report print-many.tig
```

```
error Execution times (seconds)
error  1: parse                : 0.01   (   25%)   0      (    0%)   0.01   (   20%)
error  8: liveness analysis    : 0.01   (   25%)   0      (    0%)   0.01   (   20%)
error  8: liveness edges       : 0      (    0%)   0.01   (  100%)   0.01   (   20%)
error  9: coalesce             : 0.01   (   25%)   0      (    0%)   0.01   (   20%)
error  9: register allocation  : 0.01   (   25%)   0      (    0%)   0.01   (   20%)
error Cumulated times (seconds)
error  1: parse                : 0.01   (   25%)   0      (    0%)   0.01   (   20%)
error  7: inst-display         : 0.03   (   75%)   0.01   (  100%)   0.04   (   80%)
error  8: liveness analysis    : 0.01   (   25%)   0      (    0%)   0.01   (   20%)
error  8: liveness edges       : 0      (    0%)   0.01   (  100%)   0.01   (   20%)
error  9: asm-compute          : 0.03   (   75%)   0.01   (  100%)   0.04   (   80%)
error  9: coalesce             : 0.01   (   25%)   0      (    0%)   0.01   (   20%)
error  9: register allocation  : 0.03   (   75%)   0.01   (  100%)   0.04   (   80%)
error  rest                    : 0.04   (  100%)   0.01   (  100%)   0.05   (  100%)
error  TOTAL (seconds)         : 0.04   user,      0.01   system,    0.05   wall
# == Final assembler ouput. == #
.data
l0:
        .word 1
        .asciiz "\n"
.text

# Routine: Main
t_main:
        move    t33, $s0
        move    t34, $s1
        move    t35, $s2
        move    t36, $s3
        move    t37, $s4
        move    t38, $s5
        move    t39, $s6
        move    t40, $s7
l1:
        li      t0, 0
        li      t1, 55
        li      t2, 11
        li      t3, 66
        li      t4, 22
        li      t5, 77
        li      t6, 33
        li      t7, 88
        li      t8, 44
        li      t9, 99
        li      t31, 0
        add     t30, t31, t0
        add     t29, t30, t0
        add     t28, t29, t1
        add     t27, t28, t1
        add     t26, t27, t2
        add     t25, t26, t2
        add     t24, t25, t3
```

```
        add     t23, t24, t3
        add     t22, t23, t4
        add     t21, t22, t4
        add     t20, t21, t5
        add     t19, t20, t5
        add     t18, t19, t6
        add     t17, t18, t6
        add     t16, t17, t7
        add     t15, t16, t7
        add     t14, t15, t8
        add     t13, t14, t8
        add     t12, t13, t9
        add     t11, t12, t9
        move    $a0, t11
        jal     print_int
        la      t32, l0
        move    $a0, t32
        jal     print
l2:
        move    $s0, t33
        move    $s1, t34
        move    $s2, t35
        move    $s3, t36
        move    $s4, t37
        move    $s5, t38
        move    $s6, t39
        move    $s7, t40

/* Temporary map. */
t0 -> $a0
t1 -> $t9
t2 -> $t8
t3 -> $t7
t4 -> $t6
t5 -> $t5
t6 -> $t4
t7 -> $t3
t8 -> $t2
t9 -> $t1
t11 -> $a0
t12 -> $t0
t13 -> $t0
t14 -> $t0
t15 -> $t0
t16 -> $t0
t17 -> $t0
t18 -> $t0
t19 -> $t0
t20 -> $t0
t21 -> $t0
t22 -> $t0
t23 -> $t0
```

```
t24 -> $t0
t25 -> $t0
t26 -> $t0
t27 -> $t0
t28 -> $t0
t29 -> $t0
t30 -> $t0
t31 -> $t0
t32 -> $a0
t33 -> $s0
t34 -> $s1
t35 -> $s2
t36 -> $s3
t37 -> $s4
t38 -> $s5
t39 -> $s6
t40 -> $s7

# == Final assembler ouput. == #
.data
l0:
        .word 1
        .asciiz "\n"
.text

# Routine: Main
t_main:
        sw      $fp, ($sp)
        move    $fp, $sp
        sub     $sp, $sp, 8
        sw      $ra, -4 ($fp)
l1:
        li      $a0, 0
        li      $t9, 55
        li      $t8, 11
        li      $t7, 66
        li      $t6, 22
        li      $t5, 77
        li      $t4, 33
        li      $t3, 88
        li      $t2, 44
        li      $t1, 99
        li      $t0, 0
        add     $t0, $t0, $a0
        add     $t0, $t0, $a0
        add     $t0, $t0, $t9
        add     $t0, $t0, $t9
        add     $t0, $t0, $t8
        add     $t0, $t0, $t8
        add     $t0, $t0, $t7
        add     $t0, $t0, $t7
        add     $t0, $t0, $t6
```

```
        add     $t0, $t0, $t6
        add     $t0, $t0, $t5
        add     $t0, $t0, $t5
        add     $t0, $t0, $t4
        add     $t0, $t0, $t4
        add     $t0, $t0, $t3
        add     $t0, $t0, $t3
        add     $t0, $t0, $t2
        add     $t0, $t0, $t2
        add     $t0, $t0, $t1
        add     $a0, $t0, $t1
        jal     print_int
        la      $a0, l0
        jal     print
l2:

        lw      $ra, -4 ($fp)
        move    $sp, $fp
        lw      $fp, ($fp)
        jr      $ra
```

Example 134: *tc -eIs --tempmap-display -I --time-report print-many.tig*

4.11.3 T9 Given Code

The code is provided under the following forms:

- '2005-tc-9.0.tar.bz2'[51], '2005-tc-9.1.tar.bz2'[52], '2005-tc-9.2.tar.bz2'[53], '2005-tc-9.3.tar.bz2'[54], the whole tarball.
- '2005-tc-8.0-9.0.diff'[55], '2005-tc-8.0-9.1.diff'[56], '2005-tc-8.0-9.2.diff'[57], '2005-tc-8.0-9.3.diff'[58], the differences with the latest tarball that was delivered.
- '2005-tc-9.0-9.1.diff'[59], '2005-tc-9.1-9.2.diff'[60], '2005-tc-9.1-9.3.diff'[61], '2005-tc-9.2-9.3.diff'[62], the differences with previous versions of the 'tc-9' tarball. The most significant differences are that we no longer use the `color_register` attribute for `Cpu`, that the runtime properly sets the exit status, and that its error messages are standardized.

To read the description of the new module, see Section 3.2.21 [src/regalloc], page 37.

[51] http://www.lrde.epita.fr/~akim/compil/download/2005-tc-9.0.tar.bz2.
[52] http://www.lrde.epita.fr/~akim/compil/download/2005-tc-9.1.tar.bz2.
[53] http://www.lrde.epita.fr/~akim/compil/download/2005-tc-9.2.tar.bz2.
[54] http://www.lrde.epita.fr/~akim/compil/download/2005-tc-9.3.tar.bz2.
[55] http://www.lrde.epita.fr/~akim/compil/download/2005-tc-8.0-9.0.diff.
[56] http://www.lrde.epita.fr/~akim/compil/download/2005-tc-8.0-9.1.diff.
[57] http://www.lrde.epita.fr/~akim/compil/download/2005-tc-8.0-9.2.diff.
[58] http://www.lrde.epita.fr/~akim/compil/download/2005-tc-8.0-9.3.diff.
[59] http://www.lrde.epita.fr/~akim/compil/download/2005-tc-9.0-9.1.diff.
[60] http://www.lrde.epita.fr/~akim/compil/download/2005-tc-9.1-9.2.diff.
[61] http://www.lrde.epita.fr/~akim/compil/download/2005-tc-9.1-9.3.diff.
[62] http://www.lrde.epita.fr/~akim/compil/download/2005-tc-9.2-9.3.diff.

4.11.4 T9 Code to Write

‘`src/liveness/interference-graph.hh`’
‘`src/liveness/interference-graph.cc`’
: Unfortunately, the way the moves were encoded in the tarball we delivered for T8 is not right for T9. We could have used glue code to provide backward compatibility, but it was a poor solution yielding low quality code. Therefore the interface of the `InterferenceGraph` was upgraded, which will require some modifications in your existing code.

  Rest assured that little work will actually be needed: the main modification is related to the fact that moves are now encoded as a list of pairs, while before we had a map mapping a node to the set of nodes in its move-related to.

‘`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. Note that there are some error in this book, reported on his web page (see Section 5.1 [Modern Compiler Implementation], page 113).

  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/libregalloc.cc`’
: Run the register allocation on each code fragment. Remove the useless moves.

‘`src/codegen/mips/codegen.cc`’
: If your compiler supports spills, implement `Codegen::rewrite_program`.

4.11.5 T9 FAQ

`rv` vs. `$v0`
: Our graph coloring implementation cannot support aliases for hard registers: it thinks that if it has a different name, it is different. Since this is a reasonable claim, rather than torturing the algorithm until it accepts `rv` and `$v0` designate a single guy, we decided to change the implementation of `rv` and `fp` in the frame module to use those of the current target: `$v0` and `$fp` for MIPS. This has a strong influence on havm, of course. It was modified to support these changes, so make sure to use 0.18 or higher.

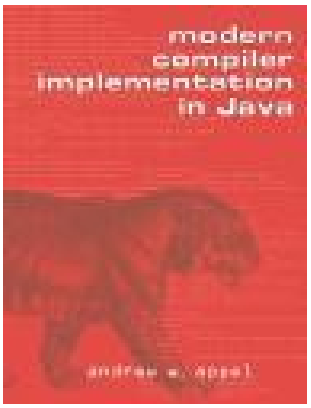
4.11.6 T9 Improvements

Possible improvements include:

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 Modern Compiler Implementation



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.

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 — 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. I recommend this edition, unless you have severe problems with functional programming.

[1] `http://www.cs.princeton.edu/~appel/modern/`.
[2] `http://www.cs.princeton.edu/~appel/`.
[3] `http://www.lrde.epita.fr/~akim/compil/tiger.html`.

This book addresses many more issues than the sole Tiger Project as we implement it. In other words, it is an extremely interesting book which 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]), some which are not.

5.2 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 5.1 [Modern Compiler Implementation], page 113. A suggested ordered small selection of books is:

1. Section 5.1 [Modern Compiler Implementation], page 113
2. [C++ Primer], page 115
3. [Design Patterns: Elements of Reusable Object-Oriented Software], page 117
4. [Effective C++], page 117
5. [Effective STL], page 118

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 Francis Gabon[7]. To buy these books, we recommend Le Monde en Tique[8], a bookshop which has demonstrated several times its dedication to its job, and its kindness to EPITA students/members.

`Bjarne Stroustrup` [Web Site]

Bjarne Stroustrup[9] is the author of C++, which he describes as (The C++ Programming Language[10]):

> 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[11] 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

[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] `mailto:Francis.Gabon@epita.fr`.
[8] `http://www.lmet.fr`.
[9] `http://www.research.att.com/~bs/homepage.html`.
[10] `http://www.research.att.com/~bs/C++.html`.
[11] `http://www.linuxjournal.com/article.php?sid=7099`.

> 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[12] is worth the browsing.

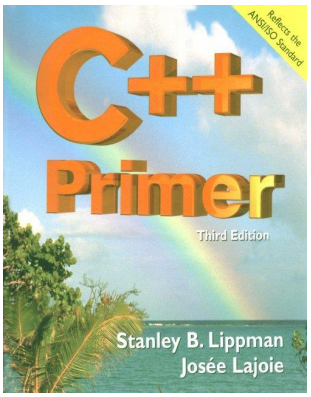
`Boost.org` [Web Site]

The Boost.org web site[13] reads:

> The Boost web site provides free peer-reviewed portable C++ source libraries. The emphasis is on libraries which 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.

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[14], the Boost Smart Pointer Library[15], the Boost Variant Library[16] etc.

`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 one should note that 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[17].

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).

[12] `http://www.research.att.com/~bs/applications.html`.

[13] `http://www.boost.org`.

[14] `http://www.boost.org/libs/spirit/index.html`.

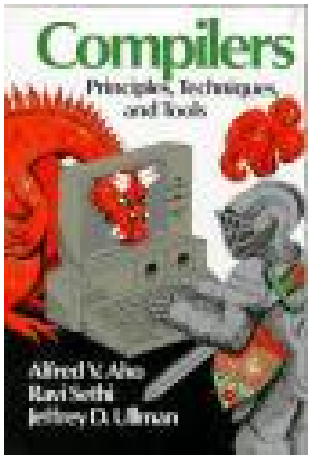
[15] `http://www.boost.org/libs/smart_ptr/index.htm`.

[16] `http://www.boost.org/regression-logs/cs-win32_metacomm/doc/html/variant.html`.

[17] `http://www.awl.com/cseng/titles/0-201-82470-1`.

`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 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[18] etc.

it is fairly technical
: The book can be hard to read for the beginner, contrary to Section 5.1 [Modern Compiler Implementation], page 113.

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[19], 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]

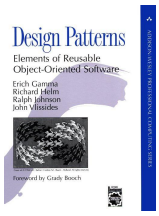
This short paper, CStupidClassName[20], explains why naming classes `CLikeThis` is stupid, but why lexical conventions are nevertheless very useful. It turns out we follow the same scheme that is emphasized there.

[18] To be fair, the Dragon Book leaves a single page (not sheet) to graph coloring.

[19] `http://www.cs.berkeley.edu/~aiken/cool/`.

[20] `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]

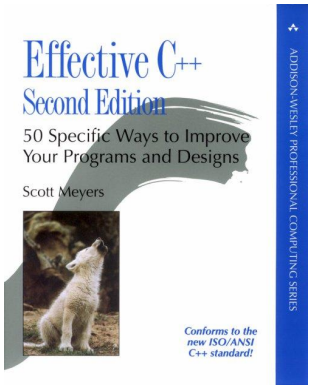


Published by Addison-Wesley; ISBN: 0-201-63361-2.

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[21]. A pre-version of this book is available on the Internet as a paper: Design Patterns: Abstraction and Reuse of Object-Oriented Design[22].

You may find additional information about Design Patterns on the Portland Pattern Repository[23].

`Effective C++` – *Scott Meyers* [Book]



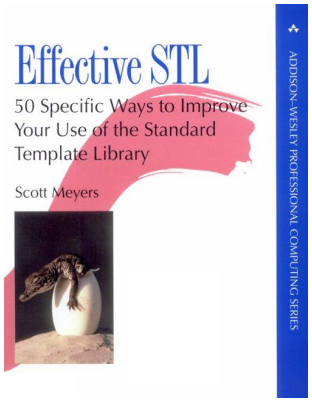
288 pages; Publisher: Addison-Wesley Pub Co; 2nd edition (September 1997); ISBN: 0-201-92488-9

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 STL Addison-Wesley Page[24].

In this document, EC*n* refers to item *n* in Effective C++.

[21] `http://www.awl.com/cseng/titles/0-201-63361-2`.
[22] `http://citeseer.nj.nec.com/gamma93design.html`.
[23] `http://c2.com/cgi/wiki?PortlandPatternRepository`.
[24] `http://www.awl.com/cseng/titles/0-201-74962-9`.

`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[25].

In this document, ES*n* refers to item *n* in Effective STL.

`Generic Visitors in C++` – *Nicolas Tisserand* [Technical Report]

This report is available on line from Visitors Page[26]: Generic Visitors in C++[27]. 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 inter-dependent 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 119, 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[28] (the famous GotW) is freely available. In this document, GotW*n*[29] refers to the item number *n*.

[25] `http://www.awl.com/cseng/titles/0-201-74962-9`.

[26] `http://www.lrde.epita.fr/cgi-bin/twiki/view/Projects/Visitors`.

[27] `http://www.lrde.epita.fr/cgi-bin/twiki/view/Publications/20030528-Seminar-Tisserand-Report`.

[28] `http://www.gotw.ca/gotw/`.

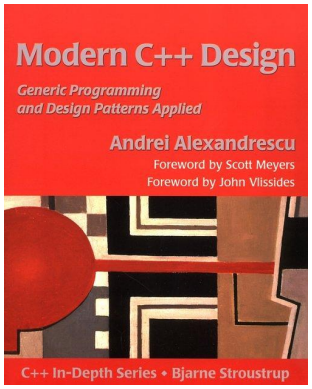
[29] `http://www.gotw.ca/gotw/n.htm`.

`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[30] justifies this lecture. It should be noted that the paper is addressing compilation **lectures**, not compilation **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 117). The code is available in the form of the Loki Library[31]. The Modern C++ Design Web Site[32] includes pointers to excerpts such as the Smart Pointers[33] chapter.

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 5.1 [Modern Compiler Implementation], page 113. In my 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 this flaw.

---

[30] `http://www.cs.arizona.edu/people/debray/papers/teaching_compilers.ps`.
[31] `http://sourceforge.net/projects/loki-lib/`.
[32] `http://www.moderncppdesign.com/book/main.html`.
[33] `http://www.aw.com/samplechapter/0201704315.pdf`.

`Parsing Techniques -- A Practical Guide` – *Dick Grune and Ceriel J. Jacob* [Book]

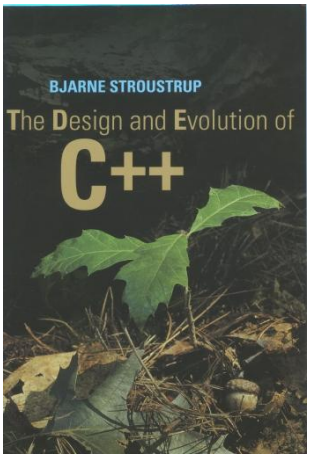
Published by the authors; ISBN: 0-13-651431-6

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[34].

`spot : une bibliothèque de vérification de propriétés de logique temporelle à temps linéaire` – *Alexandre Duret-Lutz & Rachid Rebiha* [Report]

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, you are encouraged to read the sections about the visitor hierarchy and its implementation. Another useful source of inspiration is the use of Python and Swig to write the command line interface.

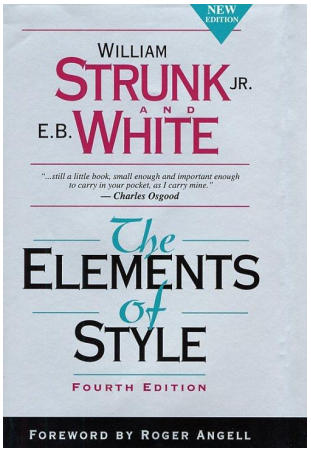
`The Design and Evolution of C++` – *Bjarne Stroustrup* [Book]



Published by Addison-Wesley, ISBN 0-201-54330-3.

No comment, since I still have not read it. It is quite famous though.

`The Elements of Style` – *William Strunk Jr., E.B. White* [Book]



[34] `http://www.cs.vu.nl/~dick/PTAPG.html`.

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*[35] on line.

`Thinking in C++ Volume 1` – *Bruce Eckel* [Book]
Published by Prentice Hall; ISBN: 0-13-979809-9

Available on the Internet: Thinking in C++ Volume 1[36]

`Thinking in C++ Volume 2` – *Bruce Eckel and Chuck Allison* [Book]
Available on the Internet: Thinking in C++ Volume 2[37].

`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[38]. 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 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[39], which includes the complete documentation on line.

5.3 The GNU Build System

Automake is used to facilitate the writing of power '`Makefile`'. Autoconf is required by Automake: we don't not address portability issues for this project.

You may read the Autoconf documentation[40], and the Automake documentation[41]. Using `info` is pleasant: '`info autoconf`' on any properly set up system. The Goat Book[42] covers the whole GNU Build System: Autoconf, Automake and Libtool.

[35] `http://coba.shsu.edu/help/strunk/`.
[36] `http://www.cs.virginia.edu/~th8k/ticpp/vol1/html/Frames.html`.
[37] `http://www.cs.virginia.edu/~th8k/ticpp/vol2/html/Index.htm`.
[38] `http://www.cantrip.org/traits.html`.
[39] `http://www.sgi.com/tech/stl/index.html`.
[40] `http://www.gnu.org/manual/autoconf/index.html`.
[41] `http://www.gnu.org/manual/automake/index.html`.
[42] `http://sources.redhat.com/autobook/`.

5.3.1 Package Name and Version

To set the name and version of your package, change the `AC_INIT` invocation. For instance, T4 for the `bardec_f` group gives:

```
AC_INIT([Bardeche Group Tiger Compiler], 4, [bardec_f@epita.fr],
        [bardec_f-tc])
```

5.3.2 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 5.4 [GCC], page 123):

```
$ ./configure CC=gcc-3.2 CXX=g++-3.2
$ make
```

Alternatively you may set `CC` and `CXX` in your environment:

```
$ export CC=gcc-3.2
$ export CXX=g++-3.2
$ ./configure && make
```

This solution is preferred as in that case the value of `CC` etc. will be used by the `./configure` invocation from '`make distcheck`' (see Section 5.3.3 [Making a Tarball], page 122).

5.3.3 Making a Tarball

Once the package autotool'ed (see Section 5.3.2 [Bootstrapping the Package], page 122), once you can run a simple '`make`', then you should be able to run '`make distcheck`' to set up the package.

The mission of '`make distcheck`' is to make sure everything will work properly. In particular it:

1. creates the tarball (via '`make dist`')
2. untars the tarball
3. configures the tarball in a separate directory (to avoid cluttering the source files with the built files).

   Arguments passed to the top level '`configure`' (e.g., `./configure CC=gcc-3.2 CXX=g++-3.2`) *will not be taken into account here.*

   Running '`export CC=gcc-3.2; export CXX=g++-3.2`' is a better way to make sure that these compilers will be used. Alternatively use `DISTCHECK_CONFIGURE_FLAGS` to specify the arguments of the embedded `./configure`:

   ```
   $ make distcheck DISTCHECK_CONFIGURE_FLAGS='--without-swig CXX=g++-
   4.0'
   ```
4. 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.

5. runs '`make check`'
6. 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 the errors, you avoid them, instead of actually getting rid of them.

At this stage, if running '`make distcheck`' does not create '`bardec_f-tc-4.tar.bz2`', then 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 GCC, The GNU Compiler Collection

We use GCC 3.2, which includes both `gcc-3.2` and `g++-3.2`: 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 3.2. Using a more recent version is often a good means to get better error messages if you can't understand what 3.2 is trying to say.

There are good patches floating around to improve GCC. In particular, you might want to use the bounds checking extension available on Herman ten Brugge Home Page[43].

5.5 Valgrind, The Ultimate Memory Debugger

Valgrind is an open-source memory debugger for x86-GNU/Linux written by Julian Seward, already known for having committed Bzip2. It is the best news for programmers for years. Unfortunately, due to EPITA's choice of NetBSD using Valgrind will not be convenient for you... Nevertheless, Valgrind is so powerful, so beautifully designed that you definitely should wander on the Valgrind Home Page[44] to learn what you are missing.

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[45], 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[46], especially the item "memory leaks" in containers[47] is enlightening.

I personally use the following shell script to track memory leaks:

[43] `http://web.inter.nl.net/hcc/Haj.Ten.Brugge`.
[44] `http://developer.kde.org/~sewardj/`.
[45] `http://dmalloc.com`.
[46] `http://gcc.gnu.org/onlinedocs/libstdc++/faq/`.
[47] `http://gcc.gnu.org/onlinedocs/libstdc++/faq/#4_4_leak`.

```
#! /bin/sh

exec 3>&1
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 ⟨undefined⟩ [0.tig], page ⟨undefined⟩,

```
$ v tc -A 0.tig
error == Memcheck, a memory error detector for x86-linux.
error == Copyright (C) 2002-2003, and GNU GPL'd, by Julian Seward.
error == Using valgrind-2.1.0, a program supervision framework for x86-
linux.
error == Copyright (C) 2000-2003, and GNU GPL'd, by Julian Seward.
error == Estimated CPU clock rate is 1667 MHz
error == For more details, rerun with: -v
error ==
error ==
error == ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
error == malloc/free: in use at exit: 50 bytes in 2 blocks.
error == malloc/free: 656 allocs, 654 frees, 37979 bytes allocated.
error == For counts of detected errors, rerun with: -v
error == searching for pointers to 2 not-freed blocks.
error == checked 5674152 bytes.
error ==
error == 18 bytes in 1 blocks are possibly lost in loss record 1 of 2
error ==    at 0x4002F202: operator new(unsigned) (vg_replace_malloc.c:162)
error ==    by 0x402C6C78: std::__default_alloc_template<true, 0>::al-
locate(unsigned) (in /usr/lib/libstdc++.so.5.0.5)
error ==    by 0x402CC567: std::string::_Rep::_S_create(unsigned, std::allocator<cha
error ==    by 0x402CD2BF: (within /usr/lib/libstdc++.so.5.0.5)
error ==    by 0x402C9AB8: std::string::string(char const*, std::allocator<char> con
error ==    by 0x805FD59: parse::tasks::parse() (tasks.cc:30)
error ==    by 0x80F8BAF: FunctionTask::execute() const (function-task.cc:18)
error ==    by 0x80FA370: TaskRegister::execute() (task-register.cc:274)
error ==    by 0x804B307: main (tc.cc:26)
error ==
error ==
error == 32 bytes in 1 blocks are still reachable in loss record 2 of 2
error ==    at 0x4002F202: operator new(unsigned) (vg_replace_malloc.c:162)
error ==    by 0x806508E: yy::Parser::parse() (parsetiger.yy:212)
error ==    by 0x805FF47: parse::parse(std::string const&, bool, bool) (libparse.cc:
error ==    by 0x805FD65: parse::tasks::parse() (tasks.cc:30)
error ==    by 0x80F8BAF: FunctionTask::execute() const (function-task.cc:18)
error ==    by 0x80FA370: TaskRegister::execute() (task-register.cc:274)
```

```
error == by 0x804B307: main (tc.cc:26)
error ==
error == LEAK SUMMARY:
error ==    definitely lost: 0 bytes in 0 blocks.
error ==    possibly lost:   18 bytes in 1 blocks.
error ==    still reachable: 32 bytes in 1 blocks.
error ==         suppressed: 0 bytes in 0 blocks.
/* == Abstract Syntax Tree. == */
0
```

Example 136: *v tc -A 0.tig*

```
$ v tc -AD 0.tig
error == Memcheck, a memory error detector for x86-linux.
error == Copyright (C) 2002-2003, and GNU GPL'd, by Julian Seward.
error == Using valgrind-2.1.0, a program supervision framework for x86-
linux.
error == Copyright (C) 2000-2003, and GNU GPL'd, by Julian Seward.
error == Estimated CPU clock rate is 1669 MHz
error == For more details, rerun with: -v
error ==
error ==
error == ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
error == malloc/free: in use at exit: 0 bytes in 0 blocks.
error == malloc/free: 665 allocs, 665 frees, 38209 bytes allocated.
error == For counts of detected errors, rerun with: -v
error == No malloc'd blocks -- no leaks are possible.
/* == Abstract Syntax Tree. == */
0
```

Example 137: *v tc -AD 0.tig*

Starting with GCC 3.4, `GLIBCPP_FORCE_NEW` is spelled `GLIBCXX_FORCE_NEW`.

5.6 Flex & Bison

We use Bison 1.875c which is able to produce a C++ parser. This Bison is unpublished, as the maintainers still have issues to fix. Nevertheless, it is usable, and perfectly functional for Tiger. It is installed in '`~akim/bin`', under the name `bison`. Be aware that Bison 1.875 produces buggy C++ parsers.

If you don't use this Bison, you will be in trouble. If you are willing to work at home, use '`bison-1.875a.tar.bz2`'[48].

The original papers on Lex and Yacc are:

Johnson, Stephen C. [1975].
: Yacc: Yet Another Compiler Compiler[49]. Computing Science Technical Report No. 32, Bell Laboratories, Murray hill, New Jersey.

Lesk, M. E. and E. Schmidt [1975].
: Lex: A Lexical Analyzer Generator[50]. Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey.

These introductory guides can help beginners:

[48] `http://www.lrde.epita.fr/~akim/compil/download/bison-1.875a.tar.bz2`.

[49] `http://epaperpress.com/lexandyacc/download/yacc.pdf`.

[50] `http://epaperpress.com/lexandyacc/download/lex.pdf`.

Thomas Niemann.

A Compact Guide to Lex & Yacc[51].

An introduction to Lex and Yacc.

Collective Work

Programming with GNU Software[52].

Contains information about Autoconf, Automake, Gperf, Flex, Bison, and GCC.

The Bison documentation[53], and the Flex documentation[54] are available for browsing.

5.7 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.

Information about HAVM can be found on HAVM Home Page[55], and feedback can be sent to LRDE's Projects Address[56].

5.8 Mipsy

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 5.9 [SPIM], page 126), supports unlimited registers. The lack of a simulator featuring this prompted the development of MIPSY.

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 the instruction selection but before the register allocation. It is implemented in C++ and Python.

Information about MIPSY can be found on MIPSY Home Page[57], and feedback can be sent to lrde's Projects Address[58].

5.9 SPIM

The SPIM documentation reads:

> SPIM S20 is a simulator that runs programs for the MIPS R2R3000 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.

[51] `http://www.epaperpress.com/lexandyacc/index.html`.
[52] `http://www.lrde.epita.fr/~akim/compil/gnuprog2/`.
[53] `http://www.lrde.epita.fr/~akim/doc/bison.html`.
[54] `http://www.lrde.epita.fr/~akim/doc/flex.html`.
[55] `http://www.lrde.epita.fr/cgi-bin/twiki/view/Projects/Havm`.
[56] `projects@lrde.epita.fr`.
[57] `http://www.lrde.epita.fr/cgi-bin/twiki/view/Projects/Mipsy`.
[58] `projects@lrde.epita.fr`.

> 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, the obvious question is: why use a simulator when many people have workstations that contain a hardware, and hence significantly faster, implementation of this computer? One reason is that these workstations are not generally available. Another reason is 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 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.
>
> 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.

5.10 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 5.11 [Python], page 127. This binding is automatically extracted from our modules using SWIG.

The SWIG home page[59] 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.

5.11 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[60].

The Python Home Page[61] reads:

---

[59] `http://www.swig.org/`.

[60] `http://www.ruby-lang.org/en/`.

[61] `http://www.python.org`.

> 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.
>
> 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.

5.12 Doxygen

We use Doxygen[62] 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[63].

The documentation produced by Doxygen must not be included, but the target `html` must produce the html documentation in the '`doc/html`' directory.

[62] `http://www.doxygen.org/index.html`.

[63] `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 4.7 [T5], page 60).

build
: The machine/architecture on which the program is being built. For instance, EPITA students typically *build* their compiler on NetBSD. 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").

HAVM
: HAVM is a `Tree` (hir or lir) programs interpreter. See Section 5.7 [HAVM], page 126.

Guru of the Week
GotW
: See Section 5.2 [Bibliography], page 114.

host
: The machine/architecture on which the program is run. For instance, EPITA students typically run their Tiger Compiler on NetBSD. Contrast with "build and "target".

IA32
: 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 5.9 [SPIM], page 126.

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 121, for the original presentation of traits. See [Modern C++ Design], page 119, 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 0.241 of '`assignments.texi`', last edited on February 24, 2004, and compiled 24 February 2004, using:

```
$ tc --version
tc (LRDE Tiger Compiler 0.66a)
Revision 0.1228 Tue, 17 Feb 2004 18:58:49 +0100

This package was written by and with the assistance of

* Akim Demaille                akim@freefriends.org
  - Maintenance.

* Alexandre Duret-Lutz         duret_g@epita.fr

* Cedric Bail                  bail_c@epita.fr
  - Initial escaping static link computation framework.

* Alexis Brouard               brouar_a@epita.fr
  - Portability of tc-check to NetBSD.

* Benoît Perrot                benoit@lrde.epita.fr
  - Extensive documentation.
  - Redesign of the Task system.
  - Design and implementation of target handling.
  - Deep clean up of every single module.
  - Third redesign of the AST, and actually, automatic generation of the AST.

* Daniel Gazard                gazard_d@epita.fr
  - Initial framework from LIR to MIPS.

* Francis Maes                  francis@lrde.epita.fr
  - Generation of static C++ Tree As Types.
```

```
* Julien Roussel                spip@lrde.epita.fr
  - "let" desugaring.

* Nicolas Burrus
  - Generation of a Swig bindings of the tc libraries to Python.
  - Implementation of a tc shell.

* Pierre-Yves Strub            strub_p@epita.fr
  - Second redesign of the AST.
  - Second redesign of Symbol.

* Quôc Peyrot                  chojin@lrde.epita.fr
  - Initial Task framework.

* Raphaël Poss                 r.poss@online.fr
  - Conversion of AST to using pointers instead of references.
  - Breakup between interfaces and implementations (.hh only -> .hxx, .cc)
  - Miscellaneous former TODO items.
  - Implementation of reference counting for Tree.

* Robert Anisko                anisko_r@epita.fr

* Sébastien Broussaud          brouss_s@epita.fr
  - Escapes torture tests.

* Stéphane Molina              molina_s@epita.fr
  - Configuration files in tc-check.

* Thierry Géraud               theo@epita.fr
  - Initial idea for visitors.
  - Initial idea for tasks.
  - Initial implementation of AST.
  - Initial implementation of Tree.

* Valentin David               david_v@epita.fr
  - Some additional tests.

* Yann Popo                    popo_y@epita.fr
  - Implementation of the Timer class.

* Yann Régis-Gianas            yann@lrde.epita.fr
  - Reimplementation of graphs.

Copyright (C) 2004 LRDE.
```

Example 138: *tc --version*

```
$ havm --version
HAVM 0.21
Written by Robert Anisko.

Copyright (C) 2003 Laboratoire de Recherche et Développement de l'EPITA.
```

```
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 139: *havm --version*

```
$ mipsy --version
mipsy (Mipsy) 0.5
Written by Benoit Perrot.

Copyright (C) 2003 Benoit Perrot.
mipsy 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 140: *mipsy --version*

A.4 List of Files

A.5 Index

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Y