Tiger Compiler Reference Manual

Edition February 9, 2004

Akim

The Tiger Project

This document describes the Tiger project for EPITA students as of February 9, 2004. More information is available on the EPITA Tiger Compiler Project Home Page[1].

Tiger is a language introduced by Andrew Appel[2] in his book Modern Compiler Implementation[3]. This document is by no means sufficient to produce an actual Tiger compiler, nor to understand compilation. You are **strongly** encouraged to buy and read Appel's book: it is an *excellent* book.

There are several differences with the original book, the most important being that EPITA students have to implement this compiler **in C++ and using modern object oriented programming techniques**. You ought to buy the original book, nevertheless, pay extreme attention to implementing the version of the language specified below, not that of the book.

[1] EPITA Tiger Compiler Project Home Page, `http://www.lrde.epita.fr/cgi-bin/twiki/view/Projects/TigerCompiler`.

[2] Andrew Appel, `http://www.cs.princeton.edu/~appel/`.

[3] Modern Compiler Implementation, `http://www.cs.princeton.edu/~appel/modern/java/`.

1 Tiger Language Reference Manual

1.1 Lexical Specifications

Keywords ‘`array`’, ‘`if`’, ‘`then`’, ‘`else`’, ‘`while`’, ‘`for`’, ‘`to`’, ‘`do`’, ‘`let`’, ‘`in`’, ‘`end`’, ‘`of`’, ‘`break`’, ‘`nil`’, ‘`function`’, ‘`var`’, and ‘`type`’

Symbols ‘`,`’, ‘`:`’, ‘`;`’, ‘`(`’, ‘`)`’, ‘`[`’, ‘`]`’, ‘`{`’, ‘`}`’, ‘`.`’, ‘`+`’, ‘`-`’, ‘`*`’, ‘`/`’, ‘`=`’, ‘`<>`’, ‘`<`’, ‘`<=`’, ‘`>`’, ‘`>=`’, ‘`&`’, ‘`|`’, and ‘`:=`’

White characters
Space and tabulations are the only white space characters supported. Both count as a single character when tracking locations.

End-of-line
End of lines are ‘`\n\r`’, and ‘`\r\n`’, and ‘`\r`’, and ‘`\n`’, freely intermixed.

Strings The strings are ANSI-C strings: enclosed by ‘`"`’, with support for the following escapes:

‘`\a`’, ‘`\b`’, ‘`\f`’, ‘`\n`’, ‘`\r`’, ‘`\t`’, ‘`\v`’
control characters.

num The character which code is *num* in octal. *num* is composed of exactly three octal characters, and any invalid value is a scan error.

\x*num* The character which code is *num* in hexadecimal (upper case or lower case or mixed). *num* is composed of exactly 2 hexadecimal characters.

‘`\\`’ A single backslash.

‘`\"`’ A double quote.

character
If no rule above applies, this is an error.

All the other characters are plain characters and are to be included in the string. In particular, multi-line strings are allowed.

Comments
Like C comments, but can be nested:

```
Code
/* Comment
   /* Nested comment */
   Comment */
Code
```

Identifiers Identifiers start with a letter, followed by any number of alphanumeric characters plus the underscore. Case sensitive.

id ::= letter { letter | digit | '_' }
letter ::=
'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' |
'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' |
'y' | 'z' |
'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' |
'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' |
'Y' | 'Z'
digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' |

Numbers There are only integers in Tiger.

integer ::= digit { digit }
op ::= '+' | '-' | '*' | '/' | '=' | '<>' | '>' | '<' | '>=' | '<=' | '&' | '|'

Invalid characters

Any other character is invalid.

1.2 Syntactic Specifications

We use Extended BNF, with '[' and ']' for zero or once, '(' and ')' for grouping, and '{' and '}' for any number of repetition including zero.

```
program ::= exp
```

```
exp ::=
  # Literals.
    'nil'
  | integer
  | string

  # Array and record creations.
  | type-id '[' exp ']' 'of' exp
  | type-id '{'[ id '=' exp { ',' id '=' exp } ] '}'

  # Variables, field, elements of an array.
  | lvalue

  # Function call.
  | id '(' [ exp { ',' exp }] ')'

  # Operations
  | '-' exp
  | exp op exp
  | '(' exps ')'

  # Assignment
  | lvalue ':=' exp

  # Control structures
  | 'if' exp 'then' exp ['else' exp]
  | 'while' exp 'do' exp
  | 'for' id ':=' exp 'to' exp 'do' exp
  | 'break'
  | 'let' decs 'in' exps 'end'

lvalue ::= id
  | lvalue '.' id
  | lvalue '[' exp ']'
exps ::= [ exp { ';' exp } ]

decs ::= { dec }
dec ::=
  # Type declaration
   'type' id '=' ty
  # Variable declaration
  | 'var' id [ ':' type-id ] ':=' exp
  # Function declaration
  | 'function' id '(' tyfields ')' [ ':' type-id ] '=' exp
```

```
# Types
ty ::= type-id
   | ‘{’ tyfields  ‘}’
   | ‘array’ ‘of’ type-id
tyfields ::= [ id ‘:’ type-id { ‘,’ id ‘:’ type-id } ]
type-id ::= id

op ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘=’ | ‘<>’ | ‘>’ | ‘<’ | ‘>=’ | ‘<=’ | ‘&’ | ‘|’
```

1.3 Semantics

1.3.1 Declarations

arrays The size of the array does not belong to the type. Arrays are always initialized.

```
let type int_array = array of int
    var  table := int_array[100] of 0
in ... end
```

builtin types

There are two builtin types, `int` and `string`. They may be redefined.

equivalence

Two record types or two array types are equivalent iff there are issued from the same definition. As in C, unlike Pascal, structural equivalence is rejected.

Type aliases do not build new types, hence they are equivalent.

```
let
  type a = int
  type b = int
  var a := 1
  var b := 2
in
  a = b           /* OK */
end
```

Name spaces

There are three name spaces: types, variables and functions. Appel uses only two, since variables and functions share the same name space. The motivation, as suggested by Sbastien Carlier, is probably related to the fact that in the second part of his book, while describing functional extensions of Tiger, functions can be assigned to variables.

```
let type a = {a : int}
    var  a := 0
    function a (a : a) : a = a{a = a.a}
in
    a (a{a = a})
end
```

But three name spaces is easier to implement than two.

1.3.2 Expressions

Boolean operators

Appel is not clear wrt the values returned by the operators `&&` and `||`. Because they are implemented as syntactic sugar, one could easily make '`123 || 456`' return '`1`' or '`123`'. *For the time being* the "right" result is considered being '`123`'. Similarly '`123 && 456`' is '`456`'. This is unnatural, but it is what is the most consistent with the suggested implementation. In the future (a different class), this might change. But anyway, *no test will depend on this.*

Precedence and Associativity

Precedence of the *op* (high to low):

```
* /
+ -
>= <= = <> < >
&
|
```

All the associative operators are associative to the left.

2 Predefined Entities

These entities are *predefined*, i.e., they are available when you start the Tiger compiler, but a Tiger program may redefine them.

2.1 Predefined Types

There are two predefined types:

'`int`' which is the type of all the literal integers.

'`string`' which is the type of all the literal strings.

2.2 Predefined Functions

Some runtime function may fail if some assertions are not fulfilled. In that case, the program must exit with a properly labelled error message, and with exit code 120. Please, note that the error messages are standardized, and must be exactly observed. Any difference, in better or worse, is a failure to comply with the (this) Tiger Reference Manual.

`chr` (*`code` : int*) [string]
: Return the one character long string containing the character which code is *code*. If *code* does not belong to the range [0..255], raise a runtime error: '`chr: character out of range`'.

`concat` (*`first`: string, `second`: string*) [string]

`exit` (*`status`: int*) [void]
: Exit the program with exit code *status*.

`flush` () [void]

`getchar` () [string]

`not` (*`boolean`: int*) [int]

`ord` (*`string`: string*) [int]

`print` (*`string`: string*) [void]

`print_int` (*`int`: int*) [void]
: Note: this is an EPITA extension. Output *int* in its decimal canonical form (equivalent to '`%d`' for `printf`).

`size` (*`string`: string*) [int]

`substring` (*`string`: string, `first`: int, `length`: int*) [string]
: Return a string composed of the characters of *string* starting at the *first* character (0 being the origin), and composed of *length* characters (i.e., up to and including the character *first* + *length*).

  Let *size* be the size of the *string*, the following assertions must hold:

  - 0 <= *first*
  - 0 <= *length*
  - *first* + *length* <= *size*

  otherwise a runtime failure is raised: '`substring: arguments out of bounds`'.

3 Implementation

3.1 Invoking `tc`

Synopsis:

```
tc option... file
```

where *file* can be '`-`', denoting the standard input.

Global options are:

'`-h`'
'`--help`' Display the help message, and exit successfully.

'`--version`'
Display the version, and exit successfully.

'`--task-list`'
List the registered tasks.

'`--task-order`'
Report the order in which the tasks will be run.

The options related to scanning and parsing (T1) are:

'`--scan-trace`'
Enable Flex scanners traces.

'`--parse-trace`'
Enable Bison parsers traces.

'`--parse`' Parse the file given as argument.

The options related to the AST (T2) are:

'`-A`'
'`--ast-display`'
Display the AST.

'`-D`'
'`--ast-delete`'
Reclaim the memory allocated for the AST.

The options related to type checking (T4) are:

'`-T`'
'`--types-check`'
Perform type checking (which is not done by default). Note the spelling.

The options related to escapes computation (T3) are:

‘`-e`’
‘`--escapes-compute`’
: Compute the escapes.

‘`-E`’
‘`--escapes-display`’
: Display the escapes. Note that this option does not imply ‘`--escapes-compute`’, so that it is possible to check that the defaults (everybody escapes) are properly implemented.

The options related to the high level intermediate representation are:

‘`--hir-compute`’
: Translate to HIR. Implies ‘`--types-check`’.

‘`-H`’
‘`--hir-display`’
: Display the high level intermediate representation. Implies ‘`--hir-compute`’.

The options related to the low level intermediate representation are:

‘`--canon-trace`’
: Trace the canonicalization of HIR to LIR.

‘`--canon-compute`’
: Canonicalize the LIR fragments.

‘`-C`’
‘`--canon-display`’
: Display the canonicalized intermediate representation *before* basic blocks and traces computation. Implies ‘`--lir-compute`’. It is convenient to determine whether a failure is due to canonicalization, or traces.

‘`--traces-trace`’
: Trace the basic blocks and traces canonicalization of HIR to LIR.

‘`--traces-compute`’
: Compute the basic blocks from canonicalized hir fragments. Implies ‘`--canon-compute`’.

‘`--lir-compute`’
: Translate to LIR. Implies ‘`--traces-compute`’. Actually, it is nothing but a nice looking alias for the latter.

‘`-L`’
‘`--lir-display`’
: Display the low level intermediate representation. Implies ‘`--lir-compute`’.

The options related to the instruction selection are:

‘`--inst-compute`’
: Convert from LIR to pseudo assembly with temporaries. Implies ‘`--lir-compute`’.

‘`-I`’
‘`--inst-display`’
: Display the pseudo assembly, (without the runtime prologue). Implies ‘`--inst-compute`’.

‘`-R`’
‘`--runtime-display`’
: Display the assembly runtime prologue for the current target.

The options related to the liveness information are:

‘`-F`’
‘`--flowgraphs-dump`’
: Save each function flow graph in a Graphviz file. Implies ‘`--inst-compute`’.

‘`-V`’
‘`--liveness-dump`’
: Save each function flow graph enriched with liveness information in a Graphviz file. Implies ‘`--inst-compute`’.

‘`-N`’
‘`--interference-dump`’
: Save each function interference graph in a Graphviz file. Implies ‘`--inst-compute`’.

3.2 Errors

Compile errors must be reported on the standard error flow with precise error location. Examples include:

```
$ echo "1 + + 2" | ./tc -
error standard input:1.4: syntax error, unexpected "+"
error Parsing Failed
```

and

```
$ echo "1 + () + 2" | ./tc -T -
error standard input:1.0-5: type mismatch
error   right operand type: void
error   expected type: int
```

Note that the symbol error is not part of the actual output. It is only used in this document to highlight that the message is produced on the standard error flow. Do not include it as part of the compiler's messages.

The compiler exit value should reflect faithfully the compilation status. The possible values are:

0
: Everything is all right.

1
: Some error which does not fall into the other categories occurred. For instance, `malloc` or `fopen` failed, a file is missing etc.

Note that an optional option (such as '`--hir-use-ix`') must cause `tc` to exit 1 if it does not support it. If you don't, be sure that your compiler will be exercised on these optional features, and it will most probably have 0.

2 Error detected during the scanning, e.g., invalid character.

3 Parse error.

4 Semantical error such as type incompatibility.

64 (`EX_USAGE`)
The command was used incorrectly, e.g., with the wrong number of arguments, a bad flag, a bad syntax in a parameter, or whatever. This is the value used by `argp`.

When several errors have occurred, the least value should be issued, not the earliest. For instance:

```
(let error in end; %)
```

should exit 2, not 3, although the parse error was first detected.

In addition to compiler errors, the compiled programs may have to raise a runtime error, for instance when runtime functions received improper arguments. In that case use the exit code 120, and issue a clear diagnostic. Note that because of the basic MIPS model we target which does not provide the standard error output, the message is to be output onto the standard output.

3.3 Extensions

A strictly compliant compiler must behave exactly as specified in this document and in Andrew Appel's book, and as demonstrated by the samples exhibited in this document, and in the "Reports"[1] document.

Nevertheless, you are entirely free to extend your compiler as you wish, as long as this extension is enabled by a non standard option. Extensions include:

ANSI Colors
Do not do that by default, in particular without checking if the output `isatty`, as the correction program will not appreciate.

Language Extensions
If for instance you intend to support loop-expression, the construct must be rejected (as a syntax error) if the corresponding option was not specified.

In **any case**, if you don't implement an extension that was suggested (such as '`--hir-use-ix`', then **you must not accept the option**. If the compiler accepts an option, then the effect of this option will be checked. For instance, if your compiler accepts '`--hir-use-ix`' but does not implement it, then be sure to get 0 on these tests.

[1] "Reports", `http://www.lrde.epita.fr/people/akim/compil/reports.html`.