Tiger Compiler Reference Manual

Edition April 17, 2016

Akim Demaille and Roland Levillain

This document presents the EPITA version of the Tiger language and compiler. This revision, , was last updated April 17, 2016.

The Tiger Project

This document describes the Tiger project for EPITA students as of April 17, 2016. It is available under various forms:

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

More information is available on the EPITA Tiger Compiler Project Home Page[6].

Tiger is derived from a language introduced by Andrew Appel[7] in his book Modern Compiler Implementation[8]. 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] `https://www.lrde.epita.fr/~tiger//tiger.html`.
[2] `https://www.lrde.epita.fr/~tiger//tiger.split`.
[3] `https://www.lrde.epita.fr/~tiger//tiger.pdf`.
[4] `https://www.lrde.epita.fr/~tiger//tiger.txt`.
[5] `https://www.lrde.epita.fr/~tiger//tiger.info`.
[6] `http://tiger.lrde.epita.fr/`.
[7] `http://www.cs.princeton.edu/~appel/`.
[8] `http://www.cs.princeton.edu/~appel/modern/`.

Table of Contents

1 Tiger Language Reference Manual

This document defines the Tiger language, derived from a language introduced by Andrew Appel in his "Modern Compiler Implementation" books (see Section "Modern Compiler Implementation" in *The Tiger Compiler Project*). We insist so that our students buy this book, so we refrained from publishing a complete description of the language. Unfortunately, recent editions of this series of book no longer address Tiger (see Section "In Java - Second Edition" in *The Tiger Compiler Project*), and therefore they no longer include a definition of the Tiger compiler. As a result, students were more inclined to xerox the books, rather than buying newer editions. To fight this trend, we decided to publish a complete definition of the language. Of course, the definition below is not a verbatim copy from the original language definition: these words are ours.

1.1 Lexical Specifications

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

Object-related keywords
The keywords '`class`', '`extends`', '`method`' and '`new`' are reserved for object-related constructions. They are valid keywords when the object extension of the language is enabled, and reserved words if this extension is disabled (i.e., they cannot be used as identifiers in object-less syntax).

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. Valid character codes belong to an extended (8-bit) ASCII set, i.e. values between 0 and 255 in decimal (0 and 377 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. Likewise, expected values belong to an extended (8-bit) ASCII set.

'`\\`'
: 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. Identifiers are case sensitive. Moreover, the special '`_main`' string is also accepted as a valid identifier.

```
id ::= letter { letter | digit | '_' } | '_main'
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 '`{`' and '`}`' for any number of repetition including zero.

```
program ::=
    exp
  | decs
```

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

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

  # Object creation.
  | 'new' type-id

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

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

  # Method call.
  | lvalue '.' 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
  # Class definition (alternative form).
  | ‘class’ id [ ‘extends’ type-id ] ‘{ ’ classfields ‘} ’
  # Variable declaration.
  | vardec
  # Function declaration.
  | ‘function’ id ‘(’ tyfields ‘)’ [ ‘:’ type-id ] ‘=’ exp
  # Primitive declaration.
  | ‘primitive’ id ‘(’ tyfields ‘)’ [ ‘:’ type-id ]
  # Importing a set of declarations.
  | ‘import’ string

vardec ::= ‘var’ id [ ‘:’ type-id ] ‘:=’ exp

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

# Types.
ty ::=
   # Type alias.
     type-id
   # Record type definition.
   | ‘{ ’ tyfields  ‘} ’
   # Array type definition.
   | ‘array’ ‘of’ type-id
   # Class definition (canonical form).
   | ‘class’ [ ‘extends’ type-id ] ‘{ ’ classfields ‘} ’
tyfields ::= [ id ‘:’ type-id { ‘,’ id ‘:’ type-id } ]
type-id ::= id

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

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

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

Comparison operators (`<`, `<=`, `=`, `<>`, `>`, `>=`) are not associative. All the remaining operators are left-associative.

1.3 Semantics

1.3.1 Declarations

import An `import` clause denote the same expression where it was (recursively) replaced by the set of declarations its corresponding import-file contains. An import-file has the following syntax (see Section 1.2 [Syntactic Specifications], page 3, for a definition of the symbols):

```
import-file ::= decs
```

Because the syntax is different, it is convenient to use another extension. We use `*.tih` for files to import, for instance:

```
/* fortytwo-fn.tih. */
function fortytwo() : int = 42

/* fortytwo-var.tih. */
import "fortytwo-fn.tih"
var fortytwo := fortytwo()

/* fortytwo-main.tig. */
let
  import "fortytwo-var.tih"
in
  print_int(fortytwo); print("\n")
end
```

is rigorously equivalent to:

```
let
  function fortytwo() : int = 42
  var fortytwo := fortytwo()
in
  print_int(fortytwo); print("\n")
end
```

There can never be a duplicate-name conflict between declarations from different files. For instance:

```
/* 1.tih */
function one() : int = 1

let
  import "1.tih"
  import "1.tih"
in
  one() = one()
end
```

is *valid* although

```
let
  function one() : int = 1
  function one() : int = 1
in
  one() = one()
end
```

is not: the function `one` is defined twice in a row of function declarations.

Importing a nonexistent file is an error. A imported file may not include itself, directly or indirectly. Both these errors must be diagnosed, with status set to 1 (see Section 4.2 [Errors], page 35).

When processing an import directive, the compiler starts looking for files in the current directory, then in all the directories of the include path, in order.

name spaces

There are three name spaces: types, variables and functions. The original language definition features two: variables and functions share the same name space. The motivation, as noted by Sbastien Carlier, is that in FunTiger, in the second part of the book, 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
```

Three name spaces support is easier to implement.

1.3.1.1 Type Declarations

arrays The size of the array does not belong to the type. Index of arrays starts from 0 and ends at size - 1.

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

Arrays are initialized with the *same* instance of value. This leads to aliasing for entities with pointer semantics (strings, arrays and records).

```
let
  type rec = { val : int }
  type rec_arr = array of rec
  var  table := rec_arr[2] of rec { val = 42 }
in
  table[0].val := 51
```

```
  /* Now table[1].val = 51. */
end
```

Use a loop to instantiate several initialization values.

```
let
  type rec = { val : int }
  type rec_arr = array of rec
  var  table := rec_arr[2] of nil
in
  for i := 0 to 1 do
    table[i] := rec { val = 42 };
  table[0].val := 51
  /* table[1].val = 42. */
end
```

records Records are defined by a list of fields between braces. Fields are described as "fieldname : type-id" and are separated by a coma. Field names are unique for a given record type.

```
let
  type indexed_string = {index : int, value : string}
in
  ...
end
```

classes (See also Section 1.3.1.4 [Method Declarations], page 18.)

Classes define a set of attributes and methods. Empty classes are valid. Attribute declaration is like variable declaration; method declaration is similar to function declaration, but uses the keyword `method` instead of `function`.

There are two ways to declare a class. The first version (known as *canonical*) uses `type`, and is similar to record and array declaration :

```
let
  type Foo = class extends Object
  {
    var bar := 42
    method baz() = print("Foo.\n")
  }
in
  /* ... */
end
```

The second version (known as *alternative* or Appel's) doesn't make use of `type`, but introduces classes declarations directly. This is the syntax described by Andrew Appel in his books:

```
let
  class Foo extends Object
  {
    var bar := 42
    method baz() = print("Foo.\n")
```

```
  }
in
  /* ... */
end
```

For simplicity reasons, constructs using the alternative syntax are considered as *syntactic sugar* for the canonical syntax, and are *desugared* by the parser into this first form, using the following transformation:

```
'class' Name [ 'extends' Super ] '{ ' Classfields '} '
=> 'type' Name '=' 'class' [ 'extends' Super ] '{ ' Classfields '} '
```

where *Name*, *Super* and *Classfields* are respectively the class name, the super class name and the contents of the class (attributes and methods) of the class.

In the rest of the section, Appel's form will be often used, to offer a uniform reading with his books, but remember that the *main syntax is the other one*, and *Appel's syntax is to be desugared into the canonical one.*

Declarations of class members follow the same rules as variable and function declarations: *consecutive* method declarations constitute a block (or chunk) of methods, while a block of attributes contains only *a single one* attribute declaration (several attribute declarations thus form several blocks). An extra rule holds for class members: there shall be no two attributes with the same name in the same class definition, nor two methods with the name.

```
let
  class duplicate_attrs
  {
    var a := 1
    method m() = ()
    /* Error, duplicate attribute in the same class.  */
    var a := 2
  }
  class duplicate_meths
  {
    method m() = ()
    var a := 1
    /* Error, duplicate method in the same class.  */
    method m() = ()
  }
in
end
```

Note that this last rule applies only to the strict scope of the class, not to the scopes of inner classes.

```
let
  type C = class
  {
    var a := 1
    method m() =
```

```
      let
        type D = class
        {
          /* These members have same names as C's, but this is allowed
             since they are not in the same scope.  */
  var a := 1
  method m() = ()
        }
      in
      end
  }
in
end
```

Objects of a given class are created using the keyword `new`. There are no constructors in Tiger (nor destructors), so the attributes are always initialized by the value given at their declaration.

```
let
  class Foo
  {
    var bar := 42
    method baz() = print("Foo.\n")
  }
  class Empty
  {
  }
  var foo1 : Foo := new Foo
  /* As for any variable, the type annotation is optional.  */
  var foo2 := new Foo
in
  /* ... */
end
```

The access to a member (either an attribute or a method) of an object from outside the class uses the *dotted* notation (as in C++, Java, C#, etc.). There are no visibility qualifier/restriction (i.e., all attributes of an object accessible in the current scope are accessible in read and write modes), and all its methods can be called.

```
let
  class Foo
  {
    var bar := 42
    method baz() = print("Foo.\n")
  }
  var foo := new Foo
in
  print_int(foo.bar);
```

```
  foo.baz()
end
```

To access to a member (either an attribute or a method) from within the class where it is defined, use the `self` identifier (equivalent to C++'s Or Java's *this*), which refers to the current instance of the object.

```
let
  class Point2d
  {
    var row : int := 0
    var col : int := 0

    method print_row() = print_int(self.row)
    method print_col() = print_int(self.col)
    method print() =
    (
      print("(");
      self.print_row();
      print(", ");
      self.print_col();
      print(")")
    )
  }
in
  /* ... */
end
```

The use of `self` is mandatory to access a member of the class (or of its super class(es)) from within the class. A variable or a method not preceded by '`self.`' won't be looked up in the scope of the class.

```
let
  var a := 42
  function m() = print("m()\n")

  class C
  {
    var a := 51
    method m() = print("C.m()\n")

    method print_a()      = (print_int(a); print("\n"))
    method print_self_a() = (print_int(self.a); print("\n"))

    method call_m()      = m()
    method call_self_m() = self.m()
  }

  var c := new C
in
```

```
  c.print_a();       /* Print '42'.  */
  c.print_self_a();  /* Print '51'.  */

  c.call_m();        /* Print 'm()'.  */
  c.call_self_m()    /* Print 'C.m()'.  */
end
```

`self` cannot be used outside a method definition. In this respect, `self` cannot appear in a function or a class defined within a method (except within a method defined therein, of course).

```
let
  type C = class
  {
    var a := 51
    var b := self              /* Invalid.  */
     method m () : int =
      let
        function f () : int =
          self.a               /* Invalid.  */
      in
        f() + self.a           /* Valid.  */
      end
  }
  var a := new C
in
  a := self                    /* Invalid. */
end
```

`self` is a read-only variable and cannot be assigned.

The Tiger language supports single inheritance thanks to the keyword `extends`, so that a class can inherit from another class declared previously, or declared in the same block of class declarations. A class with no manifest inheritance (no `extends` statement following the class name) automatically inherits from the built-in class `Object` (this feature is an extension of Appel's object-oriented proposal).

Inclusion polymorphism is supported as well: when a class Y inherits from a class X (directly or through several inheritance links), any object of Y can *be seen as* an object of type X. Hence, objects have two types: the static type, known at compile time, and the dynamic (or exact) type, known at run time, which is a subtype of (or identical to) the static type. Therefore, an object of static type Y can be assigned to a variable of type X.

```
let
  /* Manifest inheritance from Object: an A is an Object.  */
  class A extends Object {}
  /* Implicit inheritance from Object: a  B is an Object.  */
  class B {}

  /* C is an A.  */
```

```
  class C extends A {}

  var a  : A := new A
  var b  : B := new B
  var c1 : C := new C
  /* When the type is not given explicitly, it is inferred from the
     initialization; here, C2 has static and dynamic type C.  */
  var c2 := new C

  /* This variable has static type A, but dynamic type C.  */
  var c3 : A := new C
in
  /* Allowed (upcast).  */
  a := c1

  /* Forbidden (downcast).  */
  /* c2 := a */
end
```

As stated before, a class can inherit from a class[1] declared previously (and visible in the scope), or from a class declared in the same block of *type* declarations (recall that a class declaration is in fact a type declaration). Recursive inheritance is not allowed.

```
let
  /* Allowed: A declared before B.  */
  class A {}
  class B extends A {}

  /* Allowed: C declared before D.  */
  class C {}
  var foo := -42
  class D extends C {}

  /* Allowed: forward inheritance, with E and F in the same
     block.  */
  class F extends E {}
  class E {}

  /* Forbidden: forward inheritance, with G and H in different
     blocks.   */
  class H extends G {}
  var bar := 2501
  class G {}

  /* Forbidden: recursive inheritance.  */
  class I extends J {}
```

[1] A super class can only be a *class* type, and not another kind of type.

```
  class J extends I {}

  /* Forbidden: recursive inheritance and forward inheritance
     with K and L in different blocks.  */
  class K extends L {}
  var baz := 2097
  class L extends K {}

  /* Forbidden: M inherits from a non-class type.  */
  class M extends int {}
in
  /* ... */
end
```

All members from the super classes (transitive closure of the "is a" relationship) are accessible using the dotted notation, and the identifier `self` when they are used from within the class.

Attribute redefinition is not allowed: a class cannot define an attribute with the same name as an inherited attribute, even if it has the same type. Regarding method overriding, see Section 1.3.1.4 [Method Declarations], page 18.

Let us consider a block of type definitions. For each class of this block, any of its members (either attributes or methods) can reference any type introduced in scope of the block, *including the class type enclosing the considered members.*

```
let
  /* A block of types.  */
  class A
  {
    /* Valid forward reference to B, defined in the same block
       as the class enclosing this member.  */
    var b := new B
  }
  type t = int
  class B
  {
    /* Invalid forward reference to C, defined in another block
       (binding error).  */
    var c := new C
  }

  /* A block of variables.  */
  var v : t := 42

  /* Another block of types.  */
  class C
  {
  }
in
```

```
end
```

However, a class member cannot reference another member defined in a class defined later in the program, in the current class or in a future class (except if the member referred to is in the same block as the referring member, hence in the same class, since a block of members cannot obviously span across two or more classes). And recall that class members can only reference previously defined class members, or members of the same block of members (e.g., a chunk of methods).

```
let
  /* A block of types.  */
  class X
  {
    var i := 1

    /* Valid forward reference to self.o(), defined in the same
       block of methods.  */
    method m() : int = self.o()
    /* Invalid forward reference to self.p(), defined in another
       (future) block of methods (type error).  */
    method n() = self.p()
    /* Valid (backward) reference to self.i, defined earlier.  */
    method o() : int = self.i

    var j := 2

    method p() = ()

    var y := new Y

    /* Invalid forward reference to y.r(), defined in another
       (future) class (type error).  */
    method q() = self.y.r()
  }

  class Y
  {
    method r() = ()
  }
in
end
```

To put it in a nutshell: *within a chunk of types*, forward references to classes are allowed, while forward references to members are limited to the block of members where the referring entity is defined.

recursive types

Types can be recursive,

```
let
```

```
  type stringlist = {head : string, tail : stringlist}
in
  ...
end
```

or mutually recursive (if they are declared in the same chunk) in Tiger.

```
let
  type indexed_string = {index : int, value : string}
  type indexed_string_list = {head : indexed_string, tail :
  indexed_string_list}
in
  ...
end
```

but there shall be no cycle. This

```
let
  type a = b
  type b = a
in
  ...
end
```

is invalid.

type equivalence

Two types are equivalent iff there are issued from the same type construction (array or record construction, or primitive type). 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

let
  type a = {foo : int}
  type b = {foo : int}
  var va := a{foo = 1}
  var vb := b{foo = 2}
in
  va = vb
end


is invalid, and must be rejected with exit status set to 5.
```

1.3.1.2 Variable Declarations

variables

There are two forms of variable declarations in Tiger: the short one and the long one.

In the short form, only the name of the variable and the initial value of the variable are specified, the variable type is "inferred".

```
let
  var foo := 1  /* foo is typed as an integer */
in
  ...
end
```

In the long form, the type of the variable is specified. Since one cannot infer a record type for `nil`, the long form is mandated when declaring a variable initialized to `nil`.

```
let
  type foo = {foo : int}
  var bar : foo := nil       /* Correct.   */
  var baz       := nil       /* Incorrect. */
in
  ...
end
```

1.3.1.3 Function Declarations

functions To declare a function, provide its return value type:

```
let
  function not (i : int) : int =
    if i = 0 then
      1
    else
      0
in
  ...
end
```

A procedure has no value return type.

```
let
  function print_conditional(s : string, i : int) =
    if i then
      print(s)
    else
      print("error")
in
  print_conditional("foo", 1)
end
```

Functions can be recursive, but mutually recursive functions must be in the same sequence of function declarations (no other declaration should be placed between them).

See the semantics of function calls for the argument passing policy (see Section 1.3.2 [Expressions], page 20).

primitive A primitive is a built-in function, i.e., a function which body is provided by the runtime system. See Section 3.2 [Predefined Functions], page 28, for the list of standard primitives. Aside from the lack of body, and henceforth the absence of translation, primitive declarations behave as function declarations. They share the same name space, and obey the same duplicate-name rule. For instance:

```
let
  primitive one() : int
  function  one() : int = 1
in
  ...
end


is invalid, and must be rejected with exit status set to 4.
```

1.3.1.4 Method Declarations

Overriding methods

When a method in a class overrides a method of a super class, the overridden method (in the super class) is no longer accessible. Dynamic dispatch is performed, using the exact type of the object (known at run time) to select the method according to this exact type.

However, the interface of the accessible attributes and callable methods remains restricted to the static interface (i.e., the one of the static type of the object).

```
let
  class Shape
  {
    /* Position.  */
    var row := 0
    var col := 0

    method print_row() = (print("row = "); print_int(self.row))
    method print_col() = (print("col = "); print_int(self.col))

    method print() =
      (
        print("Shape = { ");
        self.print_row();
        print(", ");
        self.print_col();
        print(" }")
```

```
    )
  }

  class Circle extends Shape
  {
    var radius := 1

    method print_radius() = (print("radius = "); print_int(self.radius))

    /* Overridden method.  */
    method print() =
      (
        print("Circle = { ");
        self.print_row();
        print(", ");
        self.print_col();
        print(", ");
        self.print_radius();
        print(" }")
      )
  }

  /* C has static type Shape, and dynamic (exact) type Circle.  */
  var c : Shape := new Circle
in
  /* Dynamic dispatch to Circle's print method.  */
  c.print();

  /* Allowed.  */
  c.print_row()

  /* Forbidden: 'print_radius' is not a member of Shape (nor of its
     super class(es)).  */
  /* c.print_radius() */
end
```

Method invariance

Methods are invariant in Tiger: each redefinition of a method in a subclass shall have the exact same signature as the original (overridden) method. This invariance applies to

1. the number of arguments,
2. the types of the arguments,
3. the type of the return value[2].

```
let
  class Food {}
```

[2] Which is not the case in C++, where methods have *covariant* return values.

```
  class Grass extends Food {}

  class Animal
  {
    method eat(f : Food) = ()
  }

  class Cow extends Animal
  {
    /* Invalid: methods shall be invariant.  */
    method eat(g : Grass) = ()
  }
in
end
```

1.3.2 Expressions

L-values The '`l-values`' (whose value can be read or changed) are: elements of arrays, fields of records, instances of classes, arguments and variables.

Valueless expressions
Some expressions have no value: procedure calls, assignments, `if`s with no `else` clause, loops and `break`. Empty sequences ('`()`') and `let`s with an empty body are also valueless.

Nil The reserved word `nil` refers to a value from a `record` or a `class` type. Do not use `nil` where its type cannot be determined.

```
let
  type any_record = {any : int}
  var nil_var : any_record := nil
  function nil_test(parameter : any_record) : int = ...
  var invalid := nil             /* no type, invalid */
in
  if nil <> nil_var then
    ...
  if nil_test(nil_var) then
    ...
  if nil = nil then ...         /* no type, invalid */
end
```

Integers An integer literal is a series of decimal digits (therefore it is non-negative). Since the compiler targets 32-bit architectures, since it needs to handle signed integers, a literal integer value must fit in a signed 32-bit integer. Any other integer value is a scanner error.

Booleans There is no Boolean type in Tiger: they are encoded as integers, with the same semantics as in C, i.e., 0 is the only value standing for "false", anything else stands for "true".

Strings A string constant is a possibly empty series of printable characters, spaces or escapes sequences (see Section 1.1 [Lexical Specifications], page 2) enclosed between double quotes.

```
let
  var s := "\t\124\111\107\105\122\n"
in
  print(s)
end
```

Record instantiation

A record instantiation must define the value of all the fields and in the same order as in the definition of the record type.

Class instantiation

An object is created with `new`. There are no constructors in Tiger, so `new` takes only one operand, the name of the type to instantiate.

Function call

Function arguments are evaluated from the left to the right. Arrays and records arguments are passed by reference, strings and integer are passed by value.

The following example:

```
let
  type my_record = {value : int}
  function reference(parameter : my_record) =
    parameter.value := 42
  function value(parameter : string) =
    parameter := "Tiger is the best language\n"
  var rec1 := my_record{value = 1}
  var str := "C++ rulez"
in
  reference(rec1);
  print_int(rec1.value);
  print("\n");
  value(str);
  print(str);
  print("\n")
end
```

results in:

```
42

C++ rulez
```

Boolean operators

Tiger Boolean operators normalize their result to 0/1. For instance, because `&` and `|` can be implemented as syntactic sugar, one could easily make '`123 | 456`' return '`1`' or '`123`': make them return '`1`'. Andrew Appel does not enforce this for '`&`' and '`|`'; we do, so that the following program has a well defined behavior:

```
print_int("0" < "9" | 42)
```

Arithmetic

Arithmetic expressions only apply on integers and return integers. Available operators in Tiger are : +,-,* and /.

Comparison

Comparison operators ('=', '<>', and '<=', '<', '>=', '>') return a Boolean value.

Integer and string comparison

All the comparison operators apply to pairs of strings and pairs of integers, with obvious semantics.

String comparison

Comparison of strings is based on the lexicographic order.

Array and record comparison

Pairs of arrays and pairs of records *of the same type* can be compared for equality ('=') and inequality ('<>'). Identity equality applies, i.e., an array or a record is only equal to itself (shallow equality), regardless of the contents equality (deep equality). The value `nil` can be compared against a value which type is that of a record or a class, e.g. '`nil = nil`' is invalid.

Arrays, records and objects cannot be ordered: '<', '>', '<=', '>=' are valid only for pairs of strings or integers.

Void comparison

In conformance with A. Appel's specifications, any two void entities are equal.

Assignment

Assignments yield no value. The following code is syntactically correct, but type incorrect:

```
let
  var foo := 1
  var bar := 1
in
  foo := (bar := 2) + 1
end
```

Note that the following code is valid:

```
let
  var void1 := ()
  var void2 := ()
  var void3 := ()
in
  void1 := void2 := void3 := ()
end
```

Array and record assignment

Array and record assignments are shallow, not deep, copies. Therefore aliasing effects arise: if an array or a record variable a is assigned another variable b of the same type, then changes on b will affect a and vice versa.

```
let
  type bar = {foo : int}
  var rec1 := bar{foo = 1}
  var rec2 := bar{foo = 2}
in
  print_int(rec1.foo);
  print(" is the value of rec1\n");
  print_int(rec2.foo);
  print(" is the value of rec2\n");
  rec1 := rec2;
  rec2.foo = 42;
  print_int(rec1.foo);
  print(" is the new value of rec1\n")
end
```

Polymorphic (object) assignment

Upcasts are valid for objects because of inclusion polymorphism.

```
let
  class A {}
  class B extends A {}
  var a := new A
  var b := new B
in
  a := b
end
```

Upcasts can be performed when defining a new object variable, by forcing the type of the declared variable to a super class of the actual object.

```
let
  class C {}
  class D extends C {}
  var c : C := new D
in
end
```

Tiger doesn't provide a downcast feature performing run time type identification (RTTI), like C++'s `dynamic_cast`.

```
let
  class E {}
  class F extends E {}
  var e : E := new F
  var f := new F
in
  /* Invalid: downcast.  */
```

```
  f := e
end
```

Polymorphic (object) branching

Upcast are performed when branching between two class instantiations.

Since every class inherits from `Object`, you will always find a common root.

```
let
  class A {}
  class B extends A {}
in
  if 1 then
    new A
  else
    new B
end
```

Sequences A sequence is a possibly empty series of expressions separated by semicolons and enclosed by parenthesis. By convention, there are no sequences of a single expression (see the following item). The sequence is evaluated from the left to the right. The value of the whole sequence is that of its last expression.

```
let
  var a := 1
in
  a := (
         print("first exp to display\n");
         print("second exp to display\n");
         a := a + 1;
         a
       ) + 42;
  print("the last value of a is : ");
  print_int(a);
  print("\n")
end
```

Parentheses

Parentheses enclosing a single expression enforce syntactic grouping.

Lifetime Records and arrays have infinite lifetime: their values lasts forever even if the scope of their creation is left.

```
let
  type bar = {foo : int}
  var rec1 := bar{foo = 1}
in
  rec1 := let
            var rec2 := bar{foo = 42}
          in
            rec2
          end;
```

```
  print_int(rec1.foo);
  print("\n")
end
```

if-then-else

In an if-expression:

```
if exp1 then
  exp2
else
  exp3
```

exp1 is typed as an integer, *exp2* and *exp3* must have the same type which will be the type of the entire structure. The resulting type cannot be that of `nil`.

if-then In an if-expression:

```
if exp1 then
  exp2
```

exp1 is typed as an integer, and *exp2* must have no value. The whole expression has no value either.

while In a while-expression:

```
while exp1 do
  exp2
```

exp1 is typed as an integer, *exp2* must have no value. The whole expression has no value either.

for The following `for` loop

```
for id := exp1 to exp2 do
  exp3
```

introduces a fresh variable, *id*, which ranges from the value of *exp1* to that of *exp2*, inclusive, by steps of 1. The scope of *id* is restricted to *exp3*. In particular, *id* cannot appear in *exp1* nor *exp2*. The variable *id* cannot be assigned to. The type of both *exp1* and *exp2* is integer, they can range from the minimal to the maximal integer values. The body *exp3* and the whole loop have no value.

break A break terminates the nearest enclosing loop (`while` or `for`). A break must be enclosed by a loop. A break cannot appear inside a definition (e.g., between `let` and `in`), except if it is enclosed by a loop, of course.

let In the let-expression:

```
let
  decs
in
  exps
end
```

decs is a sequence of declaration and *exps* is a sequence of expressions separated by a semi-colon. The whole expression has the value of *exps*.

2 Language Extensions

Numerous extensions of the Tiger language are defined above. These extensions are *not* accessible to the user: if he uses one of them in a Tiger program, the compiler must reject it. They are used internally by the compiler itself, for example to desugar using concrete syntax. A special flag of the parser must be turned on to enable them.

2.1 Additional Lexical Specifications

Additional keywords and identifiers.

'`_cast`' Used to cast an expression or a l-value to a given type.

'`_decs`', '`_exp`', '`_lvalue`', '`_namety`'
: These keywords are used to plug an existing AST into an AST being built by the parser. There is a keyword per type of pluggable AST (list of declarations, expression, l-value, type name).

Reserved identifiers
: They start with an underscore, and use the same letters as standard identifiers. These symbols are used internally by the compiler to name or rename entities. Note that '`_main`' is still a valid identifier, not a reserved one.

  reserved-id ::= '_' { letter | digit | '_' }

2.2 Additional Syntactic Specifications

Grammar extensions
: In addition to the rules of the standard Tiger grammar (see Section 1.2 [Syntactic Specifications], page 3), extensions adds the following productions.

```
# A list of decs metavariable
decs ::= '_decs' '(' integer ')' decs

exp ::=
  # Cast of an expression to a given type
   '_cast' '(' exp ',' ty ')'
  # An expression metavariable
 | '_exp' '(' integer ')'

lvalue ::=
  # Cast of a l-value to a given type
   '_cast' '(' lvalue ',' ty ')'
  # A l-value metavariable
 | '_lvalue' '(' integer ')'

# A type name metavariable
type-id ::= '_namety' '(' integer ')'
```

Metavariables
: The '`_decs`', '`_exp`', '`_lvalue`', '`_namety`' keywords are used as metavariables, i.e., they are names attached to an (already built) AST. They *don't create new*

AST *nodes*, but are used to *retrieve existing nodes*, stored previously. For instance, upon an `_exp(51)` statement, the parser fetches the tree attached to the metavariable 51 (an expression) from the parsing context (see the implementation for details).

2.3 Additional Semantics

Casts

A `_cast` statement changes the type of an expression or an l-lvalue to a given type. Beware that the type-checker is forced to accept the new type as is, and must trust the programmer about the new semantics of the expression/l-value. Bad casts can raise errors in the next stages of the back-end, or even lead to invalid output code.

Casts work both on expressions and l-values. For instance, these are valid casts:

```
_cast("a", int)

_cast(a_string, int) := 42
```

(Although these examples could produce code with a strange behavior at execution time.)

Casts are currently only used in concrete syntax transformations inside the bounds checking extension and, as any language extension, are forbidden in standard Tiger programs.

3 Predefined Entities

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

3.1 Predefined Types

There are three predefined types:

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

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

'`Object`' which is the super class type on top of every class hierarchy (i.e., the top-most super class in the transitive closure of the generalization relationship).

3.2 Predefined Functions

Some runtime function may fail if some assertions are not fulfilled. In that case, the program must exit with a properly labeled error message, and with exit code 120. The error messages must follow the standard. 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]
: Concatenate *first* and *second*.

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

`flush` () [void]
: Flush the output buffer.

`getchar` () [string]
: Read a character on input. Return an empty string on an end of file.

`not` (*boolean: int*) [int]
: Return 1 if *boolean* = 0, else return 0.

`ord` (*string: string*) [int]
: Return the ascii code of the first character in *string* and -1 if the given string is empty.

`print` (*string: string*) [void]
: Print *string* on the standard output.

`print_err` (*string: string*) [void]
: Note: this is an EPITA extension. Same as `print`, but the output is written to the standard error.

`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]

Return the size in characters of the *string*.

`strcmp` (`a`: *string*, `b`: *string*) [int]

Note: this is an EPITA extension. Compare the strings *a* and *b*: return -1 if *a* < *b*, 0 if equal, and 1 otherwise.

`streq` (`a`: *string*, `b`: *string*) [int]

Note: this is an EPITA extension. Return 1 if the strings *a* and *b* are equal, 0 otherwise. Often faster than `strcmp` to test string equality.

`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* - 1).

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`'.

4 Implementation

4.1 Invoking `tc`

Synopsis:

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

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

Global options are:

`-?`
`--help` Display the help message, and exit successfully.

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

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

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

The options related to the file library (TC-1) are:

`-p`
`--library-prepend`
Prepend a directory to include path.

`-P`
`--library-append`
Append a directory to include path.

`--library-display`
Report the include search path.

The options related to scanning and parsing (TC-1) are:

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

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

`--parse` Parse the file given as argument (objects forbidden).

`--prelude=`*`prelude`*
Load the definitions of the file *prelude* before the actual argument. The result is equivalent to parsing:

```
let
  import "prelude"
in
  /* The argument file. */
```

```
end
```

To disable any prelude file, use *no-prelude*. The default value is `builtin`, denoting the builtin prelude.

`-X`
`--no-prelude`
: Don't include prelude.

The options related to the AST (TC-2) are:

`-o`
`--object` Enable object constructs of the language (class and method declarations, object creation, method calls, etc.).

`--object-parse`
: Same as `--object --parse`, i.e. parse the file given as argument, allowing objects.

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

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

The options related to escapes computation (TC-3) are:

`--bound` Make sure bindings (regular or taking overloading or objects constructs into account) are computed.

`-b`
`--bindings-compute`
: Bind the name uses to their definitions (objects forbidden).

`-B`
`--bindings-display`
: Enable the bindings display in the next `--ast-display` invocation. This option does not imply `--bindings-compute`.

`--object-bindings-compute`
: Bind the name uses to their definitions, allowing objects. consistency.

The options related to the renaming to unique identifiers (TC-R) are:

`--rename` Rename identifiers (objects forbidden).

The options related to escapes computation (TC-E) are:

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

`-E`
`--escapes-display`
: Enable the escape display. This option does not imply `--escapes-compute`, so that it is possible to check that the defaults (everybody escapes) are properly implemented. Pass `-A` afterward to see its result.

The options related to type checking (TC-4) are:

`-T`
`--typed`
: Make sure types (regular or taking overloading or objects constructs into account) are computed.

`--types-compute`
: Compute and check (regular) types (objects forbidden).

`--object-types-compute`
: Compute and check (regular) types, allowing objects.

The options related to desugaring (TC-D) are:

`--desugar-for`
: Enable the translation of `for` loops into `while` loops.

`--desugar-string-cmp`
: Enable the desugaring of string comparisons.

`--desugared`
: Make sure syntactic sugar (regular or taking overloading into account) has been removed from the AST.

`--desugar`
: Remove syntactic sugar from the AST. Desired translations must be enabled beforehand (e.g. with `--desugar-for` or `--desugar-string-cmp`).

`--overfun-desugar`
: Like `--desugar` but with support for overloaded functions (see TC-A).

The options related to the inlining optimization (TC-I) are:

`--inline`
: Inline bodies of (non overloaded) functions at call sites.

`--overfun-inline`
: Inline bodies of functions (overloaded or not) at call sites.

`--prune`
: Remove unused (non overloaded) functions.

`--overfun-prune`
: Remove unused functions (overloaded or not).

The options related to the bounds checking instrumentation (TC-B) are:

`--bounds-checks-add`
: Add dynamic bounds checks.

`--overfun-bounds-checks-add`
: Add dynamic bounds checks, with support for overloading.

The options related to overloading support (TC-A) are:

`--overfun-bindings-compute`
: Binding variables, types, and breaks as usual, by bind function calls to the set of function definitions baring the same name.

`-O`
`--overfun-types-compute`
: Type-check and resolve (bind) overloaded function calls. Implies `--overfun-bindings-compute`.

The options related to the desugaring of object constructs (TC-O) are:

`--object-desugar`
: Translate object constructs from the program into their non object counterparts, i.e., transform a Tiger program into a Panther one.

The options related to the high level intermediate representation (TC-5) are:

`--hir-compute`
: Translate to HIR (objects forbidden). Implies `--typed`.

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

The options related to the LLVM IR translation (TC-L) are:

`--llvm-compute`
: Translate to LLVM IR.

`--llvm-runtime-display`
: Enable runtime displaying along with the LLVM IR.

`--llvm-display`
: Display the LLVM IR.

The options related to the low level intermediate representation (TC-6) 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 (TC-7) 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 (TC-8) 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`.

The options related to the target are:

`--callee-save=`*num*
`--caller-save=`*num*
: Set the maximum number of callee/caller save registers to *num*, a positive number. Note that (currently) this does not reset the current target, hence to actually change the behavior, one needs '`--callee-save=0 --target-mips`'.

`--target-mips`
: Set the target to Mips.

`--target-ia32`
: This optional flag sets the target to IA-32.

`--target-default`
: If no target is selected, select Mips. This option is triggered by all the options that need a target.

`--target-display`
: Report information about the current target.

The options related to the register allocation are:

`--asm-coalesce-disable`
: Disable coalescence.

`--asm-trace`
: Trace register allocation.

`-s`
`--asm-compute`
: Allocate the registers.

`-S`
`--asm-display`
: Display the final assembler, runtime included.

4.2 Errors

Errors must be reported on the standard error output. The exit status and the standard error output must be consistent: the exit status is 0 if and only if there is no output at all on the standard error output. There are actually some exceptions: when tracing (scanning, parsing, etc.) are enabled.

Compile errors must be reported on the standard error flow with precise error location. The format of the error output must exactly be

```
location: error message
```

where the *location* includes the file name, initial position, and final position. There is no fixed set of error messages.

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
```

Warning: 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 same applied to ⇒.

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.

An unsupported option must cause `tc` to exit 64 (`EX_USAGE`) even if related to a stage option otherwise these optional features will be tested, and it will most probably have 0. For instance, a TC-5 delivery that does not support bounds checking must not accept `--bounds-checking`.

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

3 Parse error.

4 Identifier binding errors such as duplicate name definition, or undefined name use.

5 Type checking errors (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. 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.

4.3 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 see Section "Assignments" in `assignments`.

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.

5 The Reference Implementation

The so-called "reference compiler" is the compiler the LRDE develops to (i) prototype what students will have to implement, and to (ii) control the output from student compilers. It might be useful to some to see the name we gave to our options. The following is informative only, the exact contract for a conforming implementation of a Tiger compiler is defined above, Chapter 4 [Implementation], page 30.

```
$ tc --help
Tiger Compiler, Copyright (C) 2004-2017  LRDE.:

0. Tasks:
  --task-list                          list registered tasks
  --task-graph                         show task graph
  --task-selection                     list tasks to be run
  --time-report                        report execution times

1. Parsing:
  --scan-trace                         trace the scanning
  --parse-trace                        trace the parse
  --prelude STRING                     name of the prelude.  Defaults to
                                       "builtin" denoting the builtin prelude
  -X [ --no-prelude ]                  don't include prelude
  --parse                              parse a file
  --library-display                    display library search path
  -P [ --library-append ] DIR          append directory DIR to the search path
  -p [ --library-prepend ] DIR         prepend directory DIR to the search path

2. Abstract Syntax Tree:
  -A [ --ast-display ]                 display the AST
  --ast-dump                           dump the AST
  --tikz-style                         enable TikZ-style output in AST dumping

2.5 Cloning:
  --clone                              clone the Ast

3. Bind:
  --bound                              default the computation of bindings to
                                       Tiger (without objects nor overloading)
  -b [ --bindings-compute ]            bind the identifiers
  -B [ --bindings-display ]            enable bindings display in the AST
  --rename                             rename identifiers to unique names

3. Callgraph:
  --escapes-sl-compute                 compute the escaping static links and the
                                       functions requiring a static link
  --escapes-sl-display                 enable static links' escapes in the AST
  --callgraph-compute                  build the call graph
```

```
  --callgraph-dump                    dump the call graph
  --parentgraph-compute               build the parent graph
  --parentgraph-dump                  dump the parent graph

3. Escapes:
  -e [ --escapes-compute ]            compute the escaping variables and the
                                      functions requiring a static link
  -E [ --escapes-display ]            enable escape display in the AST
  --escapes-check                     check that escape tags are correct
  --escapes-necessary-check           check that tagged variables are escaping
  --escapes-sufficient-check          check that escaping variables are tagged
  --escapes-tags-display              enable escape tags display in the AST

4. Type checking:
  -T [ --typed ]                      default the type-checking to Tiger
                                      (without objects nor overloading)
  --types-compute                     check for type violations

4.5 Type checking with overloading:
  --overfun-bindings-compute          bind the identifiers, allowing function
                                      overloading
  -O [ --overfun-types-compute ]      check for type violations, allowing
                                      function overloading

5. Translation to High Level Intermediate Representation:
  --hir-compute                       translate to HIR
  -H [ --hir-display ]                display the HIR
  --hir-naive                         don't use "Ix" during the translation

5.5. Translation to LLVM Intermediate Representation:
  --llvm-compute                      translate to LLVM IR
  --llvm-runtime-display              enable runtime displayingalong with the
                                      LLVM IR
  --llvm-display                      display the LLVM IR

6. Translation to Low Level Intermediate Representation:
  --canon-compute                     canonicalize
  --canon-trace                       trace the canonicalization of the LIR
  -C [ --canon-display ]              display the canonicalized IR
  --traces-compute                    make traces
  --traces-trace                      trace the traces computation
  --lir-compute                       translate to LIR (alias for
                                      --trace-compute)
  -L [ --lir-display ]                display the low level intermediate
                                      representation

7. Target selection:
```

```
  -i [ --inst-compute ]             select the instructions
  -R [ --runtime-display ]          display the runtime
  --inst-debug                      enable instructions verbose display
  --rule-trace                      enable rule reducing display
  --garbage-collection              enable garbage collection
  -I [ --inst-display ]             display the instructions
  -Y [ --nolimips-display ]         display Nolimips compatible instructions
                                    (i.e., allocate the frames and then
                                    display the instructions
  --targeted                        default the target to MIPS
  --target-mips                     select MIPS as target
  --target-ia32                     select IA-32 as target
  --target-arm                      select ARM as target
  --target-display                  display the current target
  --callee-save NUM                 max number of callee save registers
  --caller-save NUM                 max number of caller save registers
  --argument NUM                    max number of argument registers

8. Liveness:
  -F [ --flowgraph-dump ]           dump the flowgraphs
  -V [ --liveness-dump ]            dump the liveness graphs
  -N [ --interference-dump ]        dump the interference graphs

9. Register Allocation:
  --asm-coalesce-disable            disable coalescence
  --asm-trace                       trace register allocation
  -s [ --asm-compute ]              allocate the registers
  -S [ --asm-display ]              display the final assembler

Desugaring and bounds-checking:
  --desugar-for                     desugar 'for' loops
  --desugar-string-cmp              desugar string comparisons
  --desugared                       Default the removal of syntactic sugar
                                    from the AST to Tiger (without
                                    overloading)
  --desugar                         desugar the AST
  --overfun-desugar                 desugar the AST, allowing function
                                    overloading
  --raw-desugar                     desugar the AST without recomputing
                                    bindings nor types
  --bounds-checks-add               add dynamic bounds checks
  --overfun-bounds-checks-add       add dynamic bounds checks with support
                                    for overloading
  --raw-bounds-checks-add           add bounds-checking to the AST without
                                    recomputing bindings nor types

Inlining:
```

```
  --inline                              inline functions
  --overfun-inline                      inline functions with support for
                                        overloading
  --prune                               prune unused functions
  --overfun-prune                       prune unused functions with support for
                                        overloading

Object:
  -o [ --object ]                       enable object extensions
  --object-parse                        parse a file, allowing objects
  --object-bindings-compute             bind the identifiers, allowing objects
  --object-types-compute                check for type violations, allowing
                                        objects
  --object-rename                       rename identifiers to unique names,
                                        allowing objects
  --object-desugar                      remove object constructs from the program
  --raw-object-desugar                  remove object constructs from the program
                                        without recomputing bindings nor types
  --overfun-object-bindings-compute     bind the identifiers, allowing function
                                        overloading with object
  --overfun-object-types-compute        check for type violations, allowing
                                        function overloading with object
  --overfun-object-rename               rename identifiers to unique names,
                                        allowing function overloading with
                                        objects
  --overfun-object-desugar              remove object constructs from the
                                        programallowing function overload-
ing with
                                        objects

Temporaries:
  --tempmap-display                     display the temporary table

  -? [ --help ]                         Give this help list
  --usage                               Give a short usage message
  --version                             Print program version
```

Example 5.1: *tc --help*