Tiger Language Specification

Martin Hirzel                      Kristoffer H. Rose

Version of November 17, 2013

Abstract

Tiger is a simple statically-typed programming language defined in Andrew W. Appel's book Modern Compiler Implementation in Java (Cambridge University Press, 1998), also known as the “Tiger book”. This document specifies the language. The semester-long project is to implement a compiler for Tiger, one milestone at a time. This document also refers to the “Dragon book”: Alfred H. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Compilers: Principles, Techniques, & Tools, Second Edition (Addison Wesley, 2007). See the class webpage for more information: http://cs.nyu.edu/courses/fall13/CSCI-GA.2130-001

1 Example: Hello, World!

The following example Tiger program prints a greeting to standard output.

/* Hello-world */
print("Hello, World!\n")

A slightly more elaborate version of the program accomplishes the same task with a function:

/* Hello-world with function */
let function hello() = print("Hello, World!\n")
in hello() end

2 Syntax

Figure 1 shows the grammar. The notation follows the conventions in the Dragon book: the arrow ‘→’ separates the head and body of a production; non-terminals are in italics; tokens are in bold; and the vertical bar ‘|’ separates choices. In addition, a variety of superscript notations indicate repetition of the preceding item: X+ repeats X one or more times, X* repeats X zero or more times separated by commas, and X*+ repeats X zero or more times separated by semicolons.

```
program  →  exp

dec     →  tyDec | varDec | funDec

tyDec   →  type tyId = ty
ty      →  tyId | arrTy | recTy
arrTy   →  array of tyId
recTy   →  { fieldDec* }
fieldDec →  id : tyId

funDec  →  function id ( fieldDec* ) = exp
         | function id ( fieldDec* ) : tyId = exp

varDec  →  var id := exp
         | var id : tyId := exp

lValue  →  id | subscript | fieldExp
subscript →  lValue [ exp ]
fieldExp →  lValue . id

exp     →  lValue | nil | intLit | stringLit
         | seqExp | negation | callExp | infixExp
         | arrCreate | recCreate | assignment
         | ifThenElse | ifThen | whileExp | forExp
         | break | letExp

seqExp  →  ( exp* )
negation →  - exp
callExp  →  id ( exp* )
infixExp →  exp infixOp exp
arrCreate →  tyId [ exp ] of exp
recCreate →  tyId { fieldCreate* }
fieldCreate →  id = exp
assignment →  lValue := exp
ifThenElse →  if exp then exp else exp
ifThen   →  if exp then exp
whileExp →  while exp do exp
forExp   →  for id := exp to exp do exp
letExp   →  let dec* in exp* end
```

Figure 1: Tiger grammar.
At the lexical level, Tiger has the following tokens:

- Punctuation and operators: ( ), [ ], { }, :, :=, +, -, * /, <>, <=, >=, <, >, ==, & , |
- Keywords: array, break, do, else, end, for, function, if, in, let, nil, of, then, to, type, var, while.
- Identifiers (id and tyId): An identifier starts with a letter, followed by zero or more letters, underscores, or digits. Keywords cannot be used as identifiers. Identifiers are case-sensitive.
- Integer literals (intLit): An integer literal is a sequence of one or more digits from 0 to 9.
- String literals (stringLit): A string literal starts and ends with double-quotes. A string can contain printable characters or escapes. Escapes start with a backslash \. The allowed escape sequences are:

```
\n Newline.
\t Tab.
\c The control character c.
\ddd ASCII code ddd (decimal).
\" Double-quote.
\ Backslash.
\s ... s\ Ignore s ... s (spaces or newlines).
```

The last escape sequence makes it possible to break long strings over multiple lines, by writing \ at the end of one line and the beginning of the next.
- Whitespace: Any whitespace outside of strings is ignored. Whitespace consists of spaces, tab, new-line, or comments. A comment starts with /s and ends with */. Comments can be nested.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Sequence</td>
</tr>
<tr>
<td>~</td>
<td>Negation</td>
</tr>
<tr>
<td>* /</td>
<td>Infix multiplicative</td>
</tr>
<tr>
<td>+, -</td>
<td>Infix additive</td>
</tr>
<tr>
<td>=, &lt;&gt;, &gt;, &lt;, &gt;=, &lt;=</td>
<td>Infix comparison</td>
</tr>
<tr>
<td>&amp;</td>
<td>Infix and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>:=</td>
<td>Assignment</td>
</tr>
</tbody>
</table>

Figure 2: Tiger operator precedence.

Figure 2 shows the operator precedence, in order from highest at the top to lowest at the bottom. Parentheses binds strongest and := binds weakest. All infix operators are left-associative, except for the comparison operators, which do not associate.

3 Scope Rules

Tiger has four kinds of identifiers: types, functions, variables, and fields.

- Type identifiers are declared by tyDec. The scope of a type identifier starts at the beginning of the consecutive sequence of tyDecs that define it, and lasts until the end of the enclosing letExp. This rule makes it possible for types to be recursive (when a type directly refers to itself) or even mutually recursive (when a type indirectly refers to itself via other types).
- Function identifiers are declared by funDec. The scope of a function identifier starts at the beginning of the consecutive sequence of funDecs that define it, and lasts until the end of the enclosing letExp. This rule makes it possible for functions to be recursive or even mutually recursive.
- There are three language constructs that declare variable identifiers: a varDec; a fieldDec used as a formal parameter of a function; and a forExp. In the varDec case, the scope starts after the varDec and lasts until the end of the enclosing letExp. In the fieldDec case, the scope is the function body. And in the forExp case, the scope is the loop body.
- Field identifiers are declared by a fieldDec in a recTy, which becomes their scope.

There are two namespaces for identifiers: one for types and one for variables and functions. An identifier can be used simultaneously in both namespaces.

Tiger uses lexical scoping. In other words, scopes nest, with identifiers in inner scopes hiding identifiers in outer scopes. It is a compiler error to define the same identifier in the same scope and namespace more than once.

4 Types and their Relations

Tiger has the following types:

- int is a signed integer.
- string is an immutable character string.
- Arrays are references to mutable collections of elements.
- Records are references to mutable structures with fields. Each field has a unique name within the record and a type.
• Certain expressions produce no value. We refer to
their type as \textit{void}. Expressions of type \textit{void} must
not appear where a value is expected.
The type identifiers \texttt{int} and \texttt{string} are pre-defined at
the top-level scope of the program.
Each recursive cycle of types must pass through at
least one array or record. For example, the sequence
\texttt{type a := type b = a of tyDecs} is illegal.
Each declaration of an array or record type intro-
duces a new type. For example, types \texttt{a} and \texttt{b}
declared by \texttt{type a := \{f : int\} type b := \{f : int\}} are incompat-
ible, even though they have the same structure.
On the other hand, after the declaration \texttt{type c := d},
types \texttt{c} and \texttt{d} are aliases referring to the same type.
The \texttt{nil} value does not have a type by itself; instead,
\texttt{nil} belongs to all record types.
Assignment, parameter passing, and comparison
operates on the value for \texttt{string} and \texttt{int}, but op-
erates on the reference for arrays and records.

5 Type Rules

The type rules for declarations are:
• \texttt{funDec}: If the declaration does not specify a return
type, the return type is \textit{void}. Either way, the return
type must match the type of the body.
• \texttt{varDec}: If the declaration explicitly specifies a

type, it must match the type of the initializer. The
type of the variable is the explicitly specified type,
or, if missing, the initializer type.
The type rules for l-values are:
• \texttt{id}: The identifier must refer to a variable. The
result type is the type of the variable.
• \texttt{subscript}: The base expression must have an array
type, and the index must be of type \texttt{int}. The result
type is the element type of the array.
• \texttt{fieldExp}: The base expression must have a record
type, and the identifier must name a field of the
record. The result type is the type of the field.
The type rules for expressions are:
• \texttt{nil:} Can only be used in a context where the spe-
cific record type can be determined (initializer of
typed \texttt{varDec}, assignment, comparison using \texttt{<>} or
\texttt{=} where the other operand has a known type, or
actual parameter to a function call).
• \texttt{intLit}: Has type \texttt{int}.
• \texttt{stringLit}: Has type \texttt{string}.
• \texttt{seqExp}: If the sequence is empty, the type is \textit{void},
otherwise, the type is that of the last expression.
• \texttt{negation}: Both the operand and the result are \texttt{int}.
• \texttt{callExp}: The identifier must refer to a function.
The number and types of actual and formal pa-
rameters must be the same. The type of the call is
the return type of the function.
• \texttt{infixExp}: The rules depend on the operator:
  • \texttt{+, -, *, /}: The operands must be of type \texttt{int}
    and the result type is \texttt{int}.
  • \texttt{=, <>:} The operand types must match and the
    result type is \texttt{int}.
  • \texttt{>, <, >=, <=}: The operand types must match and
    must be \texttt{int} or \texttt{string}. The result type is \texttt{int}.
  • \texttt{&}: The operands must be \texttt{int} and the result
    type is \texttt{int}.
• \texttt{arrCreate}: The \texttt{tyId} must refer to an array type.
The expression in square brackets must be \texttt{int}, and
the expression after \texttt{of} must match the element
type of the array. The result type is the array type.
• \texttt{recCreate}: The \texttt{tyId} must refer to a record type,
and the order, names, and types of fields must
match. The result type is the record type.
• \texttt{assignment}: The type of the \texttt{lValue} and the \texttt{exp}
  must match. The result type is \textit{void}.
• \texttt{ifThenElse}: The condition type must be \texttt{int}, and
the then-clause and else-clause must have the same
which becomes the result type.
• \texttt{ifThen}: The condition type must be \texttt{int}, and the
then-clause must be of type \texttt{void}. The result type
is also \textit{void}.
• \texttt{whileExp}: The condition type must be \texttt{int}, and the
body type must be \texttt{void}. The result type is \textit{void}.
• \texttt{forExp}: The start and end index must be of type
\texttt{int}. The variable is of type \texttt{int} and must not be
assigned to in the body. The body must be of type
\texttt{void}. The result type is \texttt{void}.
• \texttt{break}: Can only be used in a \texttt{whileExp} or \texttt{forExp}.
The result type is \textit{void}.
• \texttt{letExp}: If the body is empty, the type is \textit{void}, oth-
otherwise, the type is that of the last body expression.

6 Dynamic Semantics

The runtime behaviors of variable declarations are:
• \texttt{varDec}: Evaluate the expression, and initialize the
  variable to that value.
The runtime behaviors of l-values are:

- **id**: The result is the current value of the variable.
- **subscript**: Evaluate the base expression to obtain a reference to an array. Evaluate the index expression to obtain an index. Indexing is zero-based. The result is the element at that index.
- **fieldExp**: Evaluate the base expression to obtain a reference to a record. The result is the value of the field in the record.

The runtime behaviors of expressions are:

- **nil**: The result is a null-reference to a record.
- **intLit**: The result is the integer value.
- **stringLit**: The result is the string value.
- **seqExp**: Evaluate each `exp` in order. If the sequence is empty, there is no result, otherwise, the result of the last `exp` is the result of the `seqExp`.
- **negation**: Signed integer negation.
- **callExp**: Evaluate each parameter `exp` in order. Copy the actual parameters to the formals. Run the body of the callee. The result is the return value from the callee.
- **infixExp**: The behaviors depend on the operator:
  - `+`, `-`, `*`, `/`: Add, subtract, multiply, or divide the two integer operands.
  - `=`, `<>`: Equality and inequality are by-value for `int` and `string`, and by-reference for records and arrays. The result is 1 for true or 0 for false.
  - `>`, `<`, `>=`, `<=`: Magnitude comparison of `int` values, or lexicographic comparison of `string` values. The result is 1 for true or 0 for false.
  - `&`, `|`: Logical boolean conjunction and disjunction using short-circuit semantics. In other words, if the value is already known after evaluating the left operand, do not evaluate the right operand. Any non-zero integer is considered true, and 0 is false.
- **arrCreate**: Evaluate the size expression. Allocate a new array of the appropriate size. Evaluate the initializer expression. Copy its value into all elements. The result is the reference to the new array.
- **recCreate**: Allocate a new record, and initialize its fields using the field expressions. The result is the reference to the new record.
- **assignment**: Evaluate the `lValue` to a location and the `exp` to a value. Copy the value to the location.
- **ifThenElse**: Evaluate the condition. If it is non-zero, evaluate the then-clause and use its result, else evaluate the else-clause and use its result.
- **ifThen**: Evaluate the condition. If it is non-zero, evaluate the then-clause.
- **whileExp**: Evaluate the condition. If it is non-zero, evaluate the loop body and start over.
- **forExp**: Evaluate the lower and upper bound (only once before entering the loop). If the upper bound is less than the lower bound, the body is not executed. Otherwise, the body is executed once for every value between the lower and upper bound inclusive, with the iteration variable set accordingly.
- **break**: Terminate evaluation of the immediately enclosing `whileExp` or `forExp`.
- **letExp**: Evaluate each `dec` that is a `varDec` in order, then evaluate each `exp` in order. The result of the last `exp` is the result of the `letExp`.

The runtime behavior in the following situations is unspecified:

- Using a `subscript` with an out-of-bounds index.
- Using a `fieldExp` on `nil`.
- Overflowing the stack or running out of heap space.

### 7 Intrinsic Functions

The following functions are pre-defined at the toplevel scope of the program. They are part of the runtime system and form the standard library for Tiger.

- `print(s : string)` — Prints s to standard output.
- `flush()` — Flushes standard output.
- `getchar()` : `string` — One character from standard input, or empty string for end-of-file.
- `ord(s : string)` : `int` — ASCII value of first character of s, or -1 for empty string.
- `chr(i : int) : string` — Single-character string for ASCII value i, or halt program if out-of-range.
- `size(s : string)` : `int` — Length of s.
- `substring(s : string, first : int, n : int)` : `string` — Substring from s[first] to s[first+n-1] inclusive (zero-based indexing), or empty string if n<0 or first or first+n are out of range.
- `concat(s1 : string, s2 : string)` : `string` — Concatenation of s1 and s2.
- `not(i : int)` : `int` — if i=0 then 1 else 0.
- `exit(i : int)` — Halt the program with code i.

Tiger Language Specification, ©Martin Hirzel and Kristoffer H. Rose 2013