# APPENDIX

# Tiger Language Reference Manual

The Tiger language is a small language with nested functions, record values with implicit pointers, arrays, integer and string variables, and a few simple structured control constructs.

## A.1 LEXICAL ISSUES

**Identifiers:** An *identifier* is a sequence of letters, digits, and underscores, starting with a letter. Uppercase letters are distinguished from lowercase. In this appendix the symbol *id* stands for an identifier.

**Comments:** A comment may appear between any two tokens. Comments start with /* and end with */ and may be nested.

## A.2 DECLARATIONS

A declaration-sequence is a sequence of type, value, and function declarations; no punctuation separates or terminates individual declarations.

$$decs \rightarrow \{dec\}$$

$$\begin{aligned} dec &\rightarrow tydec \\ &\rightarrow vardec \\ &\rightarrow fundec \end{aligned}$$

In the syntactic notation used here, $\epsilon$ stands for the empty string and $\{x\}$ stands for a possibly empty sequence of $x$'s.

## DATA TYPES

The syntax of types and type declarations in Tiger is

*tydec* → **type** *type-id* = *ty*

*ty* → *type-id*
→ { *tyfields* } *(these braces stand for themselves)*
→ **array of** *type-id*

*tyfields* → $\epsilon$
→ *id* : *type-id* {, *id* : *type-id*}

**Built-in types:** Two named types `int` and `string` are predefined. Additional named types may be defined or redefined (including the predefined ones) by type declarations.

**Records:** Record types are defined by a listing of their fields enclosed in braces, with each field described by *fieldname* : *type-id*, where *type-id* is an identifier defined by a type declaration.

**Arrays:** An array of any named type may be made by **array of** *type-id*. The length of the array is not specified as part of its type; each array of that type can have a different length, and the length will be decided upon array creation, at run time.

**Record distinction:** Each declaration of a record or array type creates a new type, incompatible with all other record or array types (even if all the fields are similar).

**Mutually recursive types:** A collection of types may be recursive or mutually recursive. Mutually recursive types are declared by a consecutive sequence of type declarations without intervening value or function declarations. Each recursion cycle must pass through a record or array type.

Thus, the type of lists of integers:

```
type intlist = {hd: int, tl: intlist}

type tree = {key: int, children: treelist}
type treelist = {hd: tree, tl: treelist}
```

But the following declaration sequence is illegal:

```
type b = c
type c = b
```

**Field name reusability:** Different record types may use the same field names (such as the `hd` field of `intlist` and `treelist` in the example above).

## VARIABLES

*vardec* → **var** *id* := *exp*
→ **var** *id* : *type-id* := *exp*

In the short form of variable declaration, the name of the variable is given, followed by an expression representing the initial value of the variable. In this case, the type of the variable is determined from the type of the expression.

In the long form, the type of the variable is also given. The expression must have the same type.

If the initializing expression is **nil**, then the long form must be used.

Each variable declaration creates a new variable, which lasts as long as the scope of the declaration.

## FUNCTIONS

*fundec* → **function** *id* ( *tyfields* ) = *exp*
→ **function** *id* ( *tyfields* ) : *type-id* = *exp*

The first of these is a procedure declaration; the second is a function declaration. Procedures do not return result values; functions do, and the type is specified after the colon. The *exp* is the body of the procedure or function, and the *tyfields* specify the names and type of the parameters. All parameters are passed by value.

Functions may be recursive. Mutually recursive functions and procedures are declared by a sequence of consecutive function declarations (with no intervening type or variable declarations):

```
function treeLeaves(t : tree) : int =
        if t=nil then 1
        else treelistLeaves(t.children)
function treelistLeaves(L : treelist) : int =
        if L=nil then 0
        else treeLeaves(L.hd) + treelistLeaves(L.tl)
```

## SCOPE RULES

**Local variables:** In the expression **let** ··· *vardec* ··· **in** *exp* **end**, the scope of the declared variable starts just after its *vardec* and lasts until the **end**.

**Parameters:** In **function** *id* ( ··· $id_1$ : $id_2$ ··· ) = *exp* the scope of the parameter $id_1$ lasts throughout the function body *exp*.

**Nested scopes:** The scope of a variable or parameter includes the bodies of any function definitions in that scope. That is, access to variables in outer scopes is permitted, as in Pascal and Algol.

**Types:** In the expression **let** ··· *tydecs* ··· **in** *exps* **end** the scope of a type identifier starts at the beginning of the consecutive sequence of type declarations defining it and lasts until the **end**. This includes the headers and bodies of any functions within the scope.

**Functions:** In the expression **let** ··· *fundecs* ··· **in** *exps* **end** the scope of a function identifier starts at the beginning of the consecutive sequence of function declarations defining it and lasts until the **end**. This includes the headers and bodies of any functions within the scope.

**Name spaces:** There are two different name spaces: one for types, and one for functions and variables. A type *a* can be "in scope" at the same time as a variable *a* or a function *a*, but variables and functions of the same name cannot both be in scope simultaneously (one will hide the other).

**Local redeclarations:** A variable or function declaration may be hidden by the redeclaration of the same name (as a variable or function) in a smaller scope; for example, this function prints "6 7 6 8 6" when applied to 5:

```
function f(v: int) =
let var v := 6
 in print(v);
    let var v := 7 in print (v) end;
    print(v);
    let var v := 8 in print (v) end;
    print(v)
end
```

Functions hide variables of the same name, and vice versa. Similarly, a type declaration may be hidden by the redeclaration of the same name (as a type) in a smaller scope. However, no two functions in a sequence of mutually recursive functions may have the same name; and no two types in a sequence of mutually recursive types may have the same name.

## A.3 VARIABLES AND EXPRESSIONS

### L-VALUES

An *l-value* is a location whose value may be read or assigned. Variables, procedure parameters, fields of records, and elements of arrays are all *l*-values.

*lvalue* → *id*
→ *lvalue* . *id*
→ *lvalue* [ *exp* ]

**Variable:** The form *id* refers to a variable or parameter accessible by scope rules.

**Record field:** The dot notation allows the selection of the correspondingly named field of a record value.

**Array subscript:** The bracket notation allows the selection of the correspondingly numbered slot of an array. Arrays are indexed by consecutive integers starting at zero (up to the size of the array minus one).

## EXPRESSIONS

***l*-value:** An *l*-value, when used as an expression, evaluates to the contents of the corresponding location.

**Valueless expressions:** Certain expressions produce no value: procedure calls, assignment, if-then, while, break, and sometimes if-then-else. Therefore the expression `(a:=b)+c` is syntactically correct but fails to type-check.

**Nil:** The expression **nil** (a reserved word) denotes a value *nil* belonging to every record type. If a record variable *v* contains the value *nil*, it is a checked run-time error to select a field from *v*. **Nil** must be used in a context where its type can be determined, that is:

| | |
|---|---|
| `var a : my_record := nil` | OK |
| `a := nil` | OK |
| `if a <> nil then ...` | OK |
| `if nil <> a then ...` | OK |
| `if a = nil then ...` | OK |
| `function f(p: my_record) = ... f(nil)` | OK |
| `var a := nil` | Illegal |
| `if nil = nil then ...` | Illegal |

**Sequencing:** A sequence of two or more expressions, surrounded by parentheses and separated by semicolons (*exp*; *exp*; ... *exp*) evaluates all the expressions in order. The result of a sequence is the result (if any) yielded by the last of the expressions.

**No value:** An open parenthesis followed by a close parenthesis (two separate tokens) is an expression that yields no value. Similarly, a **let** expression with nothing between the **in** and **end** yields no value.

**Integer literal:** A sequence of decimal digits is an *integer constant* that denotes the corresponding integer value.

**String literal:** A string constant is a sequence, between quotes (`"`), of zero or more printable characters, spaces, or escape sequences. Each escape sequence is introduced by the escape character `\`, and stands for a character sequence. The allowed escape sequences are as follows (all other uses of `\` being illegal):

| | |
|---|---|
| `\n` | A character interpreted by the system as end-of-line. |
| `\t` | Tab. |
| `\^c` | The control character c, for any appropriate c. |
| `\ddd` | The single character with ASCII code ddd (3 decimal digits). |
| `\"` | The double-quote character (`"`). |
| `\\` | The backslash character (`\`). |
| `\f___f\` | This sequence is ignored, where f___f stands for a sequence of one or more formatting characters (a subset of the non-printable characters including at least space, tab, newline, formfeed). This allows one to write long strings on more than one line, by writing `\` at the end of one line and at the start of the next. |

**Negation:** An integer-valued expression may be prefixed by a minus sign.

**Function call:** A function application *id*() or *id*(*exp*{, *exp*}) indicates the application of function *id* to a list of actual parameter values obtained by evaluating the expressions left to right. The actual parameters are bound to the corresponding formal parameters of the function definition and the function body is bound using conventional static scoping rules to obtain a result. If *id* actually stands for a procedure (a function returning no result), then the function body must produce no value, and the function application also produces no value.

**Arithmetic:** Expressions of the form *exp op exp*, where *op* is `+,-,*,/`, require integer arguments and produce an integer result.

**Comparison:** Expressions of the form *exp op exp*, where *op* is `=,<>,>,<,>=,<=`, compare their operands for equality or inequality and produce the integer 1 for true, 0 for false. All these operators can be applied to integer operands. The equals and not-equals operators can also be applied to two record or array operands of the same type, and compare for "reference" or "pointer" equality (they test whether two records are the same instance, not whether they have the same contents).

**String comparison:** The comparison operators may also be applied to strings. Two strings are equal if their contents are equal; there is no way to distinguish strings whose component characters are the same. Inequality is according to lexicographic order.

**Boolean operators:** Expressions of the form *exp op exp*, where *op* is `&` or `|`, are short-circuit boolean conjunctions and disjunctions: they do not evaluate the right-hand operand if the result is detemined by the left-hand one. Any nonzero integer value is considered true, and an integer value of zero is false.

**Precedence of operators:** Unary minus (negation) has the highest precedence. Then operators `*,/` have the next highest (tightest binding) precedence, fol-

lowed by +,-, then by =,<>,>,<,>=,<=, then by &, then by |.

**Associativity of operators:** The operators *,/, +, - are all left-associative. The comparison operators *do not associate*, so a=b=c is not a legal expression, although a=(b=c) is legal.

**Record creation:** The expression *type-id* {*id*=*exp*{, *id*=*exp*}} or (for an empty record type) *type-id* { } creates a new record instance of type *type-id*. The field names and types of the record expression must match those of the named type, in the order given. The braces { } stand for themselves.

**Array creation:** The expression *type-id* [$exp_1$] **of** $exp_2$ evaluates $exp_1$ and $exp_2$ (in that order) to find $n$, the number of elements, and $v$ the initial value. The type *type-id* must be declared as an array type. The result of the expression is a new array of type *type-id*, indexed from 0 to $n - 1$, in which each slot is initialized to the value $v$.

**Array and record assignment:** When an array or record variable $a$ is assigned a value $b$, then $a$ references the same array or record as $b$. Future updates of elements of $a$ will affect $b$, and vice versa, until $a$ is reassigned. Parameter passing of arrays and records is similarly *by reference*, not by copying.

**Extent:** Records and arrays have infinite extent: each record or array value lasts forever, even after control exits from the scope in which it was created.

**Assignment:** The assignment statement *lvalue* := *exp* evaluates the *lvalue*, then evaluates the *exp*, then sets the contents of the *lvalue* to the result of the expression. Syntactically, := binds weaker than the boolean operators & and |. The assignment expression produces no value, so that (a:=b)+c is illegal.

**If-then-else:** The if-expression **if** $exp_1$ **then** $exp_2$ **else** $exp_3$ evaluates the integer expression $exp_1$. If the result is nonzero it yields the result of evaluating $exp_2$; otherwise it yields the result of $exp_3$. The expressions $exp_2$ and $exp_3$ must have the same type, which is also the type of the entire if-expression (or both expressions must produce no value).

**If-then:** The if-expression **if** $exp_1$ **then** $exp_2$ evaluates the integer expression $exp_1$. If the result is nonzero, then $exp_2$ (which must produce no value) is evaluated. The entire if-expression produces no value.

**While:** The expression **while** $exp_1$ **do** $exp_2$ evaluates the integer expression $exp_1$. If the result is nonzero, then $exp_2$ (which must produce no value) is executed, and then the entire while-expression is reevaluated.

**For:** The expression **for** *id* := $exp_1$ **to** $exp_2$ **do** $exp_3$ iterates $exp_3$ over each integer value of *id* between $exp_1$ and $exp_2$. The variable *id* is a new variable implicitly declared by the **for** statement, whose scope covers only $exp_3$, and may not be assigned to. The body $exp_3$ must produce no value. The upper and lower bounds are evaluated only once, prior to entering the body of the loop. If the upper bound is less than the lower, the body is not executed.

**Break:** The **break** expression terminates evaluation of the nearest enclosing while-expression or for-expression. A **break** in procedure *p* cannot terminate a loop in procedure *q*, even if *p* is nested within *q*. A **break** that is not within a **while** or **for** is illegal.

**Let:** The expression **let** *decs* **in** *expseq* **end** evaluates the declarations *decs*, binding types, variables, and procedures whose scope then extends over the *expseq*. The *expseq* is a sequence of zero or more expressions, separated by semicolons. The result (if any) of the last *exp* in the sequence is then the result of the entire let-expression.

**Parentheses:** Parentheses around any expression enforce syntactic grouping, as in most programming languages.

### PROGRAMS

Tiger programs do not have arguments: a program is just an expression *exp*.

## A.4 STANDARD LIBRARY

Several functions are predefined:

`function print(s : string)`
*Print* `s` *on standard output.*

`function flush()`
*Flush the standard output buffer.*

`function getchar() : string`
*Read a character from standard input; return empty string on end of file.*

`function ord(s: string) : int`
*Give ASCII value of first character of* `s`*; yields* `-1` *if* `s` *is empty string.*

`function chr(i: int) : string`
*Single-character string from ASCII value* `i`*; halt program if* `i` *out of range.*

`function size(s: string) : int`
*Number of characters in s.*

`function substring(s:string, first:int, n:int) : string`
*Substring of string* `s`*, starting with character* `first`*,* `n` *characters long. Characters are numbered starting at 0.*

`function concat (s1: string, s2: string) : string`
*Concatenation of* `s1` *and* `s2`.

`function not(i : integer) : integer`
*Return* `(i=0)`.

`function exit(i: int)`
*Terminate execution with code* `i`.

## A.5 SAMPLE Tiger PROGRAMS

On this page and the next are two complete Tiger programs; Program 6.3 (page 133) is a fragment (one function) of a Tiger program.

### QUEENS.TIG

```
/* A program to solve the 8-queens problem */
let
  var N := 8

  type intArray = array of int

  var row := intArray [ N ] of 0
  var col := intArray [ N ] of 0
  var diag1 := intArray [N+N-1] of 0
  var diag2 := intArray [N+N-1] of 0

  function printboard() =
     (for i := 0 to N-1
       do (for j := 0 to N-1
            do print(if col[i]=j then " O" else " .");
           print("\n"));
       print("\n"))

  function try(c:int) =
   if c=N
   then printboard()
   else for r := 0 to N-1
         do if row[r]=0 & diag1[r+c]=0 & diag2[r+7-c]=0
            then (row[r]:=1; diag1[r+c]:=1; diag2[r+7-c]:=1;
                  col[c]:=r;
                  try(c+1);
                  row[r]:=0; diag1[r+c]:=0; diag2[r+7-c]:=0)

 in try(0)
end
```

This program prints out all the ways to put eight queens on a chessboard so that no two are in the same row, column, or diagonal. It illustrates arrays and recursion. Suppose we have successfully placed queens on columns 0 to $c-1$. Then `row[r]` will be 1 if the $r$th row is occupied, `diag1[d]` will be 1 if the $d$th lower-left-to-upper-right diagonal is occupied, and `diag2[d]` will be 1 if the $d$th upper-left-to-lower-right diagonal is occupied. Now, `try`($c$) attempts to place the queens in rows $c$ to $N-1$.

## MERGE.TIG

This program reads two lists of integers from the standard input; the numbers in each list should be sorted in increasing order, separated by blanks or newlines; each list should be terminated by a semicolon.

The output is the merge of the two lists: a single list of integers in increasing order.

The `any` record is used to simulate call by reference in Tiger. Although `readint` cannot update its argument (to signify whether any more numbers remain on the input), it can update *a field* of its argument.

The assignment `any:=any{any=0}` illustrates that a name can mean a variable, a type, and a field, depending on context.

```
let   type any = {any : int}
      var buffer := getchar()

 function readint(any: any) : int =
  let var i := 0
      function isdigit(s : string) : int =
          ord(buffer)>=ord("0") & ord(buffer)<=ord("9")
   in while buffer=" " | buffer="\n" do buffer := getchar()
      any.any := isdigit(buffer);
      while isdigit(buffer)
       do (i := i*10+ord(buffer)-ord("0");
           buffer := getchar());
      i
  end

 type list = {first: int, rest: list}

 function readlist() : list =
    let var any := any{any=0}
        var i := readint(any)
     in if any.any
         then list{first=i,rest=readlist()}
         else (buffer := getchar(); nil)
    end

 function merge(a: list, b: list) : list =
   if a=nil then b
   else if b=nil then a
   else if a.first < b.first
      then list{first=a.first,rest=merge(a.rest,b)}
      else list{first=b.first,rest=merge(a,b.rest)}

 function printint(i: int) =
  let function f(i:int) = if i>0
           then (f(i/10); print(chr(i-i/10*10+ord("0"))))
   in if i<0 then (print("-"); f(-i))
      else if i>0 then f(i)
      else print("0")
  end

 function printlist(l: list) =
   if l=nil then print("\n")
   else (printint(l.first); print(" "); printlist(l.rest))

  /* BODY OF MAIN PROGRAM */
 in printlist(merge(readlist(), readlist()))
end
```