

# Tiger Language Reference Manual

Prof. Stephen A. Edwards  
Columbia University

This document describes the Tiger language defined in Andrew Appel's book *Modern Compiler Implementation in Java* (Cambridge University Press, 1998).

The Tiger language is a small, imperative language with integer and string variables, arrays, records, and nested functions. Its syntax resembles some functional languages.

## 1 Lexical Aspects

An *identifier* is a sequence of letters, digits, and underscores that starts with a letter. Case is significant.

Whitespace (spaces, tabs, newlines, returns, and formfeeds) or comments may appear between tokens and is ignored. A *comment* begins with */\** and ends with *\*/*. Comments may nest.

An integer constant is a sequence of one or more decimal digits (i.e., 0123456789). There are no negative integer constants; negative numbers may be obtained by negating an integer constant using the unary *-* operator.

A string constant is a sequence of zero or more printable characters, spaces, or escape sequences surrounded by double quotes ". Each escape sequence starts with a backslash \ and stands for some sequence of characters. The escape sequences are

<code>\n</code>	Newline
<code>\t</code>	Tab
<code>\"</code>	Double quote
<code>\\</code>	Backslash
<code>\^c</code>	Control- <i>c</i> , where <i>c</i> is one of @A...Z[\]^_.
<code>\ddd</code>	The character with ASCII code <i>ddd</i> (three decimal digits)
<code>\...\</code>	Any sequence of whitespace characters (spaces, tabs, newlines, returns, and formfeeds) surrounded by \s is ignored. This allows string constants to span multiple lines by ending and starting each with a backslash.

The reserved words are `array break do else end for function if in let nil of then to type var while`.

The punctuation symbols are `, : ; ( ) [ ] { } . + - * / = < > <= >= & | :=`

## 2 Expressions

A Tiger program is a single *expr*.

*expr*:

*string-constant*  
*integer-constant*  
`nil`  
*lvalue*  
`- expr`  
*expr binary-operator expr*

*lvalue* := *expr*  
*id* (*expr-list*<sub>opt</sub>)  
(*expr-seq*<sub>opt</sub>)  
*type-id* { *field-list*<sub>opt</sub> }  
*type-id* [*expr*] of *expr*  
`if` *expr* `then` *expr*  
`if` *expr* `then` *expr* `else` *expr*  
`while` *expr* `do` *expr*  
`for` *id* := *expr* `to` *expr* `do` *expr*  
`break`  
`let` *declaration-list* `in` *expr-seq*<sub>opt</sub> `end`

*expr-seq*:

*expr*  
*expr-seq* ; *expr*

*expr-list*:

*expr*  
*expr-list* , *expr*

*field-list*:

*id* = *expr*  
*field-list* , *id* = *expr*

### 2.1 Lvalues

*lvalue*:

*id*  
*lvalue* . *id*  
*lvalue* [*expr*]

An *l-value* represents a storage location that can be assigned a value: variables, parameters, fields of records, and elements of arrays. The elements of an array of size *n* are indexed by 0;1;;; *n* ∈ 1.

### 2.2 Return values

Procedure calls, assignments, if-then, while, break, and sometimes if-then-else produce no value and may not appear where a value is expected (e.g., (*a* := *b*) + *c* is illegal). A `let` expression with nothing between the `in` and `end` returns no value.

A sequence of zero or more expressions in parenthesis (e.g., (*a* := 3; *b* := *a*)) separated by semicolons are evaluated in order and returns the value produced by the final expression, if any. An empty pair of parenthesis () is legal and returns no value.

### 2.3 Record and Array Literals

The expression *type-id* { *field-list*<sub>opt</sub> } (zero or more fields are allowed) creates a new record instance of type *type-id*. Field

names, expression types, and the order thereof must exactly match those of the given record type.

The expression *type-id* [ *expr* ] of *expr* creates a new array of type *type-id* whose size is given by the expression in brackets. Initially, the array is filled with elements whose values are given by the expression after the *of*. These two expressions are evaluated in the order they appear.

## 2.4 Function Calls

A function application is an expression *id* ( *expr-list<sub>opt</sub>* ) with zero or more comma-separated expression parameters. When a function is called, the values of these actual parameters are evaluated from left to right and bound to the function's formal parameters using conventional static scoping rules.

## 2.5 Operators

The binary operators are + - \* / = <> < > <= >= & |

Parentheses group expressions in the usual way.

A leading minus sign negates an integer expression.

The binary operators +, -, \*, and / require integer operands and return an integer result.

The binary operators >, <, >=, and <= compare their operands, which may be either both integer or both string and produce the integer 1 if the comparison holds and 0 otherwise. String comparison is done using normal ASCII lexicographic order.

The binary operators = and <> can compare any two operands of the same (non-valueless) type and return either integer 0 or 1. Integers are the same if they have the same value. Strings are the same if they contain the same characters. Two objects of record type are the same if they refer to the same record. Two arrays are the same if they refer to the same array. That is, records and arrays are compared using "reference" or "pointer" equality, not componentwise.

The logical operators & and | are lazy logical operators on integers. They do not evaluate their right argument if evaluating the left determines the result. Zero is considered false; everything else is considered true.

Unary minus has the highest precedence followed by \* and /, then + and -, then =, <>, >, <, >=, and <=, then &, then |, then finally : =.

The +, -, \*, and / operators are left associative. The comparison operators do not associate, e.g., a=b=c is erroneous, but a=(b=c) is legal.

## 2.6 Assignment

The assignment expression *lvalue* := *expr* evaluates the expression then binds its value to the contents of the *lvalue*. Assignment expressions do not produce values, so something like a := b := 1 is illegal.

Array and record assignment is by reference, not value. Assigning an array or record to a variable creates an alias, meaning later updates of the variable or the value will be reflected in both places. Passing an array or record as an actual argument to a function behaves similarly.

A record or array value persists from the time it is created to the termination of the program, even after control has left the scope of its definition.

## 2.7 nil

The expression *nil* represents a value that can be assigned to any record type. Accessing a field from a *nil*-valued record is a runtime error. *Nil* must be used in a context where its actual record type can be determined, thus the following are legal.

```
var a : rec := nil      a := nil
if a <> nil then ...    if a = nil then ...
function f(p: rec) = f(nil)
```

But these are illegal.

```
var a := nil            if nil = nil then ...
```

## 2.8 Flow control

The if-then-else expression, written *if* *expr* *then* *expr* *else* *expr* evaluates the first expression, which must return an integer. If the result is non-zero, the second expression is evaluated and becomes the result, otherwise the third expression is evaluated and becomes the result. Thus, the second and third expressions must be of the same type or both not return a value.

The if-then expression, *if* *expr* *then* *expr* evaluates its first expression, which must be an integer. If the result is non-zero, it evaluates the second expression, which must not return a value. The if-then expression does not return a value.

The while-do expression, *while* *expr* *do* *expr* evaluates its first expression, which must return an integer. If it is non-zero, the second expression is evaluated, which must not return a value, and the while-do expression is evaluated again.

The for expression, *for* *id* := *expr* *to* *expr* *do* *expr*, evaluates the first and second expressions, which are loop bounds. Then, for each integer value between the values of these two expressions (inclusive), the third expression is evaluated with the integer variable named by *id* bound to the loop index. The scope of this variable is limited to the third expression, and may not be assigned to. This expression may not produce a result and is not executed if the loop's upper bound is less than the lower bound.

The break expression terminates the innermost enclosing while or for expression that is enclosed in the same function/procedure. The break is illegal outside this.

## 2.9 Let

The expression *let* *declaration-list* *in* *expr-seq<sub>opt</sub>* *end* evaluates the declarations, binding types, variables, and functions to the scope of the expression sequence, which is a sequence of zero or more semicolon-separated expressions. The result is that of the last expression, or nothing if there are none.

## 3 Declarations

*declaration-list*:

*declaration*  
*declaration-list declaration*

*declaration*:

*type-declaration*  
*variable-declaration*  
*function-declaration*

### 3.1 Types

*type-declaration:*

`type type-id = type`

*type:*

`type-id`

`{ type-fieldsopt }`

`array of type-id`

*type-fields:*

`type-field`

`type-fields , type-field`

*type-field:*

`id : type-id`

Tiger has two predefined types: `int` and `String`. New types may be defined and existing types redefined as follows.

The three forms of *type* refer to a type (creates an alias in a declaration), a record with named, typed fields (like a C struct, different records may reuse field names), and an array.

Type expressions (e.g., `fx: intg, array of ty`) create distinct types, so two array types with the same base or two records with identical fields are different. Type `a=b` is an alias.

A sequence of type declarations (i.e., with no intervening variable or function declarations) may be mutually recursive. No two defined types in such a sequence may have the same name. Each recursion cycle must pass through a record or array type.

In `let ... type-declaration ... in expr-seqopt` end, the scope of the type declaration begins at the start of the sequence of type declarations to which it belongs (which may be a singleton) and ends at the end.

Type names have their own name space.

### 3.2 Variables

*variable-declaration:*

`var id := expr`

`var id : type-id := expr`

This declares a new variable and its initial value. If the type is not specified, the variable's type comes from the expression.

In `let ... variable-declaration ... in expr-seqopt` end, the scope of the variable declaration begins just after the declaration and ends at the end. A variable lasts throughout its scope.

Variables and functions share the same name space.

### 3.3 Functions

*function-declaration:*

`function id ( type-fieldsopt ) = expr`

`function id ( type-fieldsopt ) : type-id = expr`

The first form is a procedure declaration; the second is a function. Functions return a value of the specified type; procedures are only called for their side-effects. Both forms allow the specification of a list of zero or more typed arguments, which are passed by value. The scope of these arguments is the *expr*.

The *expr* is the body of the function or procedure.

A sequence of function declarations (i.e., with no intervening variable or type declarations) may be mutually recursive. No two functions in such a sequence may have the same name.

In `let ... function-declaration ... in expr-seqopt` end, the scope of the function declaration begins at the start of the sequence of function declarations to which it belongs (which may be a singleton) and ends at the end.

## 4 Standard Library

`function print(s : string)`

Print the string on the standard output.

`function printi(i : int)`

Print the integer on the standard output.

`function flush()`

Flush the standard output buffer.

`function getchar() : string`

Read and return a character from standard input; return an empty string at end-of-file.

`function ord(s : string) : int`

Return the ASCII value of the first character of *s*, or `€1` if *s* is empty.

`function chr(i : int) : string`

Return a single-character string for ASCII value *i*. Terminate program if *i* is out of range.

`function size(s : string) : int`

Return the number of characters in *s*.

`function substring(s: string, f: int, n: int): string`

Return the substring of *s* starting at the character *f* (first character is numbered zero) and going for *n* characters.

`function concat (s1: string, s2: string): string`

Return a new string consisting of *s1* followed by *s2*.

`function not(i : int) : int`

Return 1 if *i* is zero, 0 otherwise.

`function exit(i : int)`

Terminate execution of the program with code *i*.

## 5 Example

`let /* The eight queens solver from Appel */`

`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 " 0" 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`