# The Tiger-0 Language

Pedro Vasconcelos, DCC/FCUP

November 2021

## 1 Overview

*Tiger* is a small language defined in the book *Modern Compiler Implementation in ML* by Andrew Appel (Cambridge University Press, 1998). This document defines a subset of Tiger, called *Tiger-0*, suitable as a target for implementing a small compiler in the context of an introductory compilers course.

Tiger-0 is a small imperative language with integers, strings, arrays, basic control flow structures and functions. The most notable differences to the Tiger language defined in Appel's book are: Tiger-0 disallows nested function definitions and record types, and supports only integer arrays.

The syntax of Tiger-0 has some similarities to Pascal and Standard ML. Readers more familiar with languages from the C family should pay special attention to different conventions regarding terminators and operators. Studying the example programs at the end of this document can also help clarify the differences.

## 2 Lexical Aspects

Whitespace characters (spaces, newlines or tabulations) may appear between any *tokens* and are ignored. Comments are delimited by `/*` and `*/` and may also occur between any tokens. Multi-line comments are allowed, but nested comments are not.

An *identifier* is a sequence of letters (`a` to `z` or `A` to `Z`), digits (`0` to `9`) or underscores (`_`) begining with a letter or underscore.

An *integer constant* is a sequence of one or more decimal digits (`0` to `9`). Constants do not have a sign; negative numbers can be obtained by applying the unary operator `-`.

A *string constant* is a sequence of zero or more printable characters between double quotes (`"`). Some escape sequences are treated to mean special characters:

`\n` Newline

`\t` Tabulation

`\"` Double quotes

`\\` Backslash (\)

The following sequences are *reserved keywords*: `break` `do` `else` `end` `for` `function` `if` `in` `let` `of` `then` `var` `while`.

The following charaters are punctuation signs: `,` `:` `;` `(` `)` `[` `]`.

The language operators are: `+` `-` `*` `/` `%` `=` `<>` `<` `<=` `>` `>=` `&` `|` `:=`.

# 3 Expressions

*expr* :

- *integer-constant*
- *string-constant*
- *lvalue*
- *expr binary-operator expr*
- `-`*expr*
- *lvalue* `:=` *expr*
- *id*`(`*expr-list*$_{opt}$`)`
- `(`*expr-seq*$_{opt}$`)`
- `if` *expr* `then` *expr*
- `if` *expr* `then` *expr* `else` *expr*
- `while` *expr* `do` *expr*
- `for` *id* `:=` *expr* `to` *expr* `do` *expr*
- `break`
- `let` *var-decl-list* `in` *expr-seq* `end`

*lvalue* :

- *id*

*expr-seq* :

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

*expr-list* :

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

Tiger-0 is an expression-based language, meaning that expressions are the fundamental syntactical construct used to build programs. Expressions may compute a value (e.g. `2*a+1`) or may be used just for side-effects, such as assigning to a variable (e.g. `a:=1`).

Note that, unlike languages from the C family, Tiger uses `:=` for assignment and `=` for comparing equality.

## 3.1 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

Note also that, unlike C, semicolons `;` are used as separators between expressions rather than after each expression.

## 3.2 Assignments and Lvalues

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.

The valid left-hand targets of assignments are called *lvalues*; in the base Tiger-0 language these can only be simple variables (e.g. `a:=2*b`). Section 6 discusses an extension to allows arrays.

## 3.3 Function Calls

A function application is an expression `id(e1,e2,...)` 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.

## 3.4 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.

The logical operators `&` and `|` are lazy logical operators on integers. They do not evaluate their right argument if evaluating the left determines the result. Logical negation is expressed by the library function `not`. 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.

## 3.5 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.

## 3.6 Variable Declarations

*var-decl-list* :

*var-decl*
*var-decl-list var-decl*

*var-decl* :

`var` *id* `:=` *expr*

The expression `let` *var-decl-list* `in` *expr-seq* `end` evaluates the declarations, binding variables to the scope of the expression sequence, which is a sequence of one or more semicolon-separated expressions. The result is that of the last expression.

A variable declaration defines a new variable and its initial value. The variable's type comes from the expression. In `let` ... *var-decl*... `in` *expr-seq* `end`, the scope of the variable declaration begins just after the declaration and closes at the `end`. A variable lasts throughout its scope. Variables and functions share the same name space.

Usually, variables must be declared and initialized before use. The only exceptions are loop variables in the `for` construct e.g. the following expression is valid with no explicit declaration of variable `i`:

```
for i:=1 to 10 do print(i)
```

Note that it is not possible to use a loop variable outside of the loop because the scope of the loop variable is limited to the body of the loop.

# 4 Programs

*program* :
`let` *decl-list* `in` *expr-seq*

*decl-list* :
*decl*
*decl-list decl*

*decl* :
*var-decl*
*fun-decl*

*fun-decl* :
`function` *id*`(`*type-fields*$_{opt}$`)` `=` *expr*
`function` *id*`(`*type-fields*$_{opt}$`)`:*type-id* `=` *expr*

*type-fields* :
*type-field*
*type-fields* `,` *type-field*

*type-field* :
*id* `:` *type-id*

A *program* is list of global variable or function definitions followed by an expression sequence.

Note that Tiger-0 does not allow nested function definitions, that is, functions can only be declared at the outermost `let` expression of a program.

## 4.1 Function Declarations

There are two forms of function declarations. 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 can have the same name.

## 4.2 Types

*type-id* :
`int`
`string`
`intArray`

Tiger-0 has only two basic types, `int` and `string`, and one structured type `intArray` for arrays of integers (see Section 6).

# 5 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 scani() : int`** Read an integer from the standard input.

# 6 Arrays

*expr* :

⋮

`intArray [` *expr* `] of` *expr*

*lvalue* :

*id*

*id* `[` *expr* `]`

This section describes an extension to Tiger-0 for simple integer arrays.

The expression `intArray [` *expr* `] of` *expr* creates a new array of integers 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. For example, `intArray [10] of 0` creates a 10 element array with each element initialized to 0.

Array indexing is expressed by *id* `[` *expr* `]` where *id* is an array identifier and *expr* is an (integer) index. Valid indices are from 0 to $N-1$ for an $N$-element array.

Array assignment is by reference, not value. Assigning an array 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.

# 7 Example Programs

## 7.1 Sum of squares

```
1  /* Compute the sum of squares from 1 to 10 */
2  let
3    var s := 0
4    var n := 1
5  in
6    while n <= 10 do
7       (s := s + n*n;
```

```
8         n := n + 1);
9     printi(n)
```

### 7.2 Recursive Factorial

```
1  let
2    function fact(n: int): int =
3       if n>0 then n*fact(n-1)
4       else 1
5  in
6    printi(fact(10))
```

### 7.3 Naive prime number test

```
1  /* Test prime numbers */
2  let
3    function is_prime(n:int): int =
4      let
5          var d := 2
6      in
7          while d<n & n%d<>0 do
8              d := d+1;
9          n>1 & d=n
10     end
11 in
12   let
13     var i := scani()
14   in if is_prime(i) then
15     print("prime")
16   else
17     print("not prime")
18 end
```

### 7.4 Fibonnaci numbers

```
1  /* Tabulate some Fibonnaci numbers */
2  let
3     var N := 20
4     var fib := intArray [ N ] of 0
5  in
6     fib[1] := 1;
7     for i := 2 to N-1
8     do fib[i] := fib[i-1] + fib[i-2];
9     for i := 0 to N-1
10    do printi(fib[i])
```

### 7.5 8-Queens problem

```
1  /* Solver for the 8 queens problem by Andrew Appel */
2  let
```

```
3     var N := 8
4     var row := intArray [ N ] of 0
5     var col := intArray [ N ] of 0
6     var diag1 := intArray [ N+N-1 ] of 0
7     var diag2 := intArray [ N+N-1 ] of 0
8
9     function printboard() =
10       (for i := 0 to N-1
11        do (for j := 0 to N-1
12            do print(if col[i]=j then " O" else " .");
13            print("\n"));
14        print("\n"))
15
16    function try(c:int) =
17       if c=N then printboard()
18       else for r := 0 to N-1
19            do if row[r]=0 &
20                  diag1[r+c]=0 & diag2[r+7-c]=0
21               then (row[r] := 1; diag1[r+c] := 1;
22                     diag2[r+7-c] := 1; col[c] := r;
23                     try(c+1);
24                     row[r] := 0; diag1[r+c] := 0;
25                     diag2[r+7-c] := 0)
26  in try(0)
```