CS 8803-O08: Tiger Language Specification

Santosh Pande
santosh.pande@cc.gatech.edu

Georgia Institute of Technology — Updated: August 28, 2022

Overview

The Tiger language is a small, imperative language with integer and float variables, arrays, and subroutines. The language specification defined here is modified from Stephen A. Edwards' *Tiger Language Reference Manual* and Andrew Appel's book *Modern Compiler Implementation in Java* (Cambridge University Press, 1998).

1 Lexical Rules

Token Type	Token Value
Keywords	
ARRAY	array
BEGIN	begin
BREAK	break
DO	do
ELSE	else
END	end
ENDDO	enddo
ENDIF	endif
FLOAT	float
FOR	for
FUNCTION	function
IF	if
INT	int
LET	let
OF	of
PROGRAM	program
RETURN	return
STATIC	static
THEN	then
TO	to
TYPE	type
VAR	var
WHILE	while

Punctuation	
COMMA	,
DOT	.
COLON	:
SEMICOLON	;
OPENPAREN	(
CLOSEPAREN	)
OPENBRACK	[
CLOSEBRACK	]
OPENCURLY	{
CLOSECURLY	}
Binary Operators	
PLUS	+
MINUS	-
MULT	*
DIV	/
POW	**
EQUAL	==
NEQUAL	!=
LESS	<
GREAT	>
LESSEQ	<=
GREATEQ	>=
AND	&
OR	|
Assignment Operators	
ASSIGN	:=
TASSIGN	=

The special lexical rules for token types matching user-defined values are as follows:

- `ID`: Sequence of one or more letters, digits, and underscores. Must start with a letter. Case sensitive.
- `COMMENT`: Begins with `/*` and ends with `*/`. Nesting is not allowed.
- `INTLIT`: Consists of a sequence of one or more digits. Should not have leading zeros. Should be unsigned.
- `FLOATLIT`: Consists of a sequence of one or more digits followed by a decimal point. May have a single leading zero only if it directly precedes the decimal point. May have zero or more digits after the decimal point. Should be unsigned.

2 Grammar

⟨tiger-program⟩ ⊨ PROGRAM ID LET ⟨declaration-segment⟩ BEGIN ⟨funct-list⟩ END

⟨declaration-segment⟩ ⊨ ⟨type-declaration-list⟩ ⟨var-declaration-list⟩

⟨type-declaration-list⟩ ⊨ ⟨type-declaration⟩ ⟨type-declaration-list⟩ | ϵ

⟨var-declaration-list⟩ ⊨ ⟨var-declaration⟩ ⟨var-declaration-list⟩ | ϵ

⟨funct-list⟩ ⊨ ⟨funct⟩ ⟨funct-list⟩ | ϵ

⟨type-declaration⟩ ⊨ TYPE ID TASSIGN ⟨type⟩ SEMICOLON

⟨type⟩ ⊨ ⟨base-type⟩ | ARRAY OPENBRACK INTLIT CLOSEBRACK OF ⟨base-type⟩ | ID

⟨base-type⟩ ⊨ INT | FLOAT

⟨var-declaration⟩ ⊨ ⟨storage-class⟩ ⟨id-list⟩ COLON ⟨type⟩ ⟨optional-init⟩ SEMICOLON

⟨storage-class⟩ ⊨ VAR | STATIC

⟨id-list⟩ ⊨ ID | ID COMMA ⟨id-list⟩

⟨optional-init⟩ ⊨ ASSIGN ⟨const⟩ | ϵ

⟨funct⟩ ⊨ FUNCTION ID OPENPAREN ⟨param-list⟩ CLOSEPAREN ⟨ret-type⟩ BEGIN ⟨stat-seq⟩ END

⟨param-list⟩ ⊨ ⟨param⟩ ⟨param-list-tail⟩ | ϵ

⟨param-list-tail⟩ ⊨ COMMA ⟨param⟩ ⟨param-list-tail⟩ | ϵ

⟨ret-type⟩ ⊨ COLON ⟨type⟩ | ϵ

⟨param⟩ ⊨ ID COLON ⟨type⟩

⟨stat-seq⟩ ⊨ ⟨stat⟩ | ⟨stat⟩ ⟨stat-seq⟩

⟨stat⟩ ⊨ ⟨value⟩ ASSIGN ⟨expr⟩ SEMICOLON |
IF ⟨expr⟩ THEN ⟨stat-seq⟩ ENDIF SEMICOLON |
IF ⟨expr⟩ THEN ⟨stat-seq⟩ ELSE ⟨stat-seq⟩ ENDIF SEMICOLON |
WHILE ⟨expr⟩ DO ⟨stat-seq⟩ ENDDO SEMICOLON |
FOR ID ASSIGN ⟨expr⟩ TO ⟨expr⟩ DO ⟨stat-seq⟩ ENDDO SEMICOLON |
⟨optprefix⟩ ID OPENPAREN ⟨expr-list⟩ CLOSEPAREN SEMICOLON |
BREAK SEMICOLON |
RETURN ⟨optreturn⟩ SEMICOLON |
LET ⟨declaration-segment⟩ BEGIN ⟨stat-seq⟩ END

⟨optreturn⟩ ⊨ ⟨expr⟩ | ϵ

⟨optprefix⟩ ⊨ ⟨value⟩ ASSIGN | ϵ

⟨expr⟩ ⊨ ⟨const⟩ | ⟨value⟩ | ⟨expr⟩ ⟨binary-operator⟩ ⟨expr⟩ | OPENPAREN ⟨expr⟩ CLOSEPAREN

⟨const⟩ ⊨ INTLIT | FLOATLIT

⟨binary-operator⟩ ⊨ PLUS | MINUS | MULT | DIV | POW |
EQUAL | NEQUAL | LESS | GREAT | LESSEQ | GREATEQ | AND | OR

⟨expr-list⟩ ⊨ ⟨expr⟩ ⟨expr-list-tail⟩ | ϵ

⟨expr-list-tail⟩ ⊨ COMMA ⟨expr⟩ ⟨expr-list-tail⟩ | ϵ

⟨value⟩ ⊨ ID ⟨value-tail⟩

⟨value-tail⟩ ⊨ OPENBRACK ⟨expr⟩ CLOSEBRACK | ϵ

3 Semantic Rules

3.1 Operator Precedence

The operator precedence for the Tiger language is found below, listed from highest to lowest. Operators in the same line belong to the same group and have the same precedence.

1. ( )
2. **
3. * /
4. + -
5. == != > < >= <=
6. &
7. |

3.2 Operator Associativity

- The +, -, *, /, &, and | operators are left-associative.
- The exponentiation ** operator is right-associative. `a ** b ** c` evaluates as b raised to c first (let the result be t) and then a is raised to t.
- Though allowed by the grammar, relational operators in Tiger do not associate, e.g., `a==b==c` is a semantic error, not a syntactic error.

3.3 Operator Rules

- Operators must operate on scalar values and not arrays. For example, if either a or b were an array, then `a + b` would be a semantic error.
- Arithmetic operators { +, -, *, / } take integer or float operands, or combination of integer and float, and return an integer or float result.
- The exponentiation operator { ** } can have its left operand be an integer or a float. The right operand must be an integer.
- Relational operators { ==, !=, >, <, >=, <= } take operands which may be either both integers or both floats. They produce the integer value 1 if the comparison holds and 0 otherwise.
- Logical operators { &, | }, representing AND and OR respectively, take operands which must both be integers. They produce the integer value 1 for true or 0 for false.
- Zero (0) is considered false; non-zero is considered true.

3.4 Types

Two base types, `int` and `float`, are predefined. Additional named types can be defined from base or named types. For example:

```
type myInt = int;
type alsoAnInt = myInt;
```

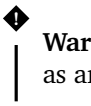
3.4.1 Array Type

Tiger supports static-sized arrays. Arrays can be created **only** by first creating an array type. The following would create an array type of five integers:

```
type intArray = array[5] of int;
```

Dereferencing an array is done by creating an index expression which must evaluate to an integer, e.g., intArray[0] accesses the first element in intArray. The array expression evaluates to an l-value or an r-value depending on where it appears. As an l-value, it evaluates to a storage at the specified index. As an r-value, it returns a value stored in that array location.

The only aggregate operation allowed on arrays is assignment when the arrays are structurally type equivalent. Any other aggregate operation on arrays is illegal; for all other operations, arrays must be operated on an element-by-element basis.



Warning: It is a semantic error to return an array from a subroutine or to send an array to a subroutine as an argument.

3.4.2 Type Equivalence

Type equivalence enforced for Tiger is structural type equivalence. With structural equivalence, two types are equal if, and only if, they have the same base type and dimension. If two variables are structurally equivalent (such as two arrays of the same length and base type) but have different names, they are still equivalent types.

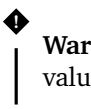
```
type arrayA = array[10] of float;
type arrayB = array[10] of float;
type arrayC = array[10] of int;
type arrayD = array[15] of float;
```

The above type declarations will lead to only `arrayA` and `arrayB` being structurally type equivalent. Structural type inferencing is useful for determining the base type of the scalar value which is used in type checking semantics.

3.4.3 Type Conversion

An integer value will be auto-promoted to a float in the following circumstances (assume variable `a` is of type float):

- direct assignment: a := 6;
- assignment on return: a := getInteger();
- arithmetic operation: 1.2 / 6
- function return type is listed as float, but the function returns an integer value.
- function signature calls for a float argument, but the function is called with an integer value.



Warning: All narrowing conversions should be considered semantic errors, e.g., assignment of a float value to a variable declared as an integer.

3.5 Variables

Variables represent named memory locations that store values. The lifetime and visibility of a variable is limited to its scope.

3.5.1 Storage Class

When a varible is declared, it is assigned a storage class. Two storage classes are available for variables, `var` and `static`.

Variables declared as `static` are stored in the static data section and will be automatically initialized to zero if an initializer is not provided. Variables declared as `var` are stored on the call stack and are not automatically initialized. Parameters of subroutines are implicity stack based (`var`).

3.5.2 Declaration and Initialization

A variable declaration creates a new variable and optionally initializes its value. Example scalar variable declarations:

```
static a : int;
static b : int := 5;
var c, d  : float;
var e, f : float := 1.0;
```

To declare an array variable, you must first create an array type. Example array variable declarations:

```
type intArray = array[10] of int;
static arrayA : intArray;
var arrayB : intArray := 25;
var arrayC : intArray;
```

The above declaration creates three integer arrays, arrayA, arrayB, and arrayC. Each array has the capacity to hold ten integers. arrayA will be automatically initialized to zero, as it is a static variable. arrayB will have each of the 10 memory locations initialized to the value 25. arrayC is not assigned an initial value, so its value will remain undefined until manually assigned.

3.5.3 Assignment

The assignment statement evaluates the expression on the right and then binds the result to the contents of the `value`. For example:

```
var a, b : int;
a := 10;
b := a;
```

Assignment is allowed between arrays that are structurally type equivalent (same size and type). During an aggregate assignment of arrayB to arrayA, respective elements of arrayB are assigned to that of arrayA. For example:

```
arrayA := arrayB;
```

Warning: Use of a variable before assignment is undefined behavior.

3.6 Control Flow

The `if-then` expression evaluates the first expression which must return an integer. If the result is non-zero, the statements under the then clause are evaluated.

The `if-then-else` expression evaluates the first expression which must return an integer. If the result is non-zero, the statements under the then clause are evaluated. Otherwise, the statements under the else clause are evaluated.

The `while-do` expression evaluates the first expression which must return an integer. If it is non-zero, the body of the loop is evaluated and the while-do expression is evaluated again.

The `for` expression evaluates the first and second expressions which are integer loop bounds. Then, for each integer value between the values of the two expressions (inclusive), the statements are evaluated with the integer variable named by the identifier bound to the loop index. This part is not executed if the loop's upper bound is less than its lower bound.

3.7 Subroutines

Tiger supports two types of subroutines:

- Procedure: has no return value
- Function: has a return value of a specified type

Both forms allow the specification of a list of zero or more typed arguments. Arguments are passed by value. A procedure may, but is not required to, include an empty return statement. In contrast, a function must return a value. For semantic checking, it is permissible to check only that a function contains a return <value> statement, not that all paths are covered.

Warning: Returning a value from a procedure is a semantic error. Not returning a value from a function is a semantic error.

An executable Tiger program must contain a special function named **main**. On compilation, main will represent the starting point of the program.

Tiger supports recursive subroutine calls.

3.8 Scope

Tiger programs use lexical scoping. There are three types of scope creation: global, subroutine, and let statement.

3.8.1 Global Scope

There is a single global scope created by the initial `let` keyword at the top of a Tiger program which extends to the end of the program. This scope can be accessed by all of the subroutines within the program and contains the variables declared at the top of the tiger program, as well as the names of the subroutines themselves.

Warning: Variables defined in the global section must be declared **static** or else it is a semantic error.

3.8.2 Subroutine Scope

Every subroutine definition establishes a new scope for its parameters that extends for the entire subroutine.

3.8.3 Let Statement Block Scope

Local scopes are created by `let` statement blocks. Let blocks are delineated by the keywords: `let, begin, end`. Variables and types declared between `let` and `begin` may then be used in the statements between `begin` and `end`. Let statement blocks can be nested as shown in the grammar. The scopes follow the classic block structure. Each let statement creates a new scope which ends at the corresponding `end` keyword.

Warning: Variables defined in let statement blocks must be declared **var** or else it is a semantic error.

3.8.4 Scope Binding Rules

- Variables, named types, and subroutines all share the same namespace.
- A declaration of an entity (types, variables) with a given name in a given scope hides its declaration from outer scopes (if any).
- When a scope is closed by the corresponding end, all entities declared in that scope are automatically destroyed. The lifetime of the entities is limited to the scope in which they are declared.
- There is no forward referencing. All names must be defined before use.
- The binding of a name is decided in the following manner:
  1. The name is looked up in the current scope.
  2. If not found, the lookup proceeds to the outer scopes, one by one.
  3. Once a name is found in a given scope, the lookup stops.
  4. If not found in any scope, the name is undefined.

Warning: Use of an undefined name is a semantic error.

Warning: Redeclaration of the same name in the same scope is a semantic error.

3.9 Standard Library

Tiger programs can call functions from the standard library without defining them.

Definition	Usage
`function printi(i : int)`	print the integer to the standard output, followed by a newline
`function printf(f : float)`	print the float to the standard output, followed by a newline
`function not(i : int) : int`	return 1 if i is zero; otherwise, 0
`function exit(i : int)`	terminate execution of the program with code i

4 Appendix: Example Programs

demo_print.tiger

```
/* Test Program: print out integers */

program demo_print
let
begin

    function printout(x : int, y : int, z : int)
    begin
        printi(x);
        printi(y);
        printi(z);
    end

    function main() : int
    begin
        let
            var a : int := 10;
            var b : int := 20;
        begin
            printout(a * 2, b - a, a + b);
        end
        return 0;
    end

end
```

demo_slope.tiger

```
/* Test Program: calculate slope  */

program demo_slope
let
    type point = array[2] of float;
    static p1, p2 : point;
    static rise : float;
    static run : float;
    static slope : float;
    static yintercept : float;
    static xintercept : float;
begin

    function main() : int
    begin
        p1[0] := 2.0;
        p1[1] := 1.0;
        p2[0] := 3.0;
        p2[1] := 3.0;

        rise := p2[1] - p1[1];
        run  := p2[0] - p1[0];

        if ((run == 0.0) | (rise == 0.0)) then
            printi(0);
            exit(1);
        endif;

        slope := rise / run;
        yintercept := p2[1] - slope * p2[0];
        xintercept := (0 - yintercept) / slope;

        printf(slope);
        printf(xintercept);
        printf(yintercept);
        return 0;
    end

end
```

demo_selection_sort.tiger

```
/* Test Program: selection sort */

program demo_selection_sort
let
begin

   function main() : int
   begin
      let
        type catList = array[8] of int;
        var cats : catList;
        var index : int;
        var lowestIndex : int;
        var subIndex : int;
        var size : int;
        var smallest : int;
        var first : int;
      begin
        cats[0] := 7;
        cats[1] := 2;
        cats[2] := 14;
        cats[3] := 24;
        cats[4] := 5;
        cats[5] := 6;
        cats[6] := 49;
        cats[7] := 33;
        size := 8;

        for index := 0 to size - 1 do
          lowestIndex := index;
          for subIndex := (index + 1) to (size - 1) do
            if (cats[subIndex] < cats[lowestIndex]) then
              lowestIndex := subIndex;
            endif;
          enddo;

          /* swap minimum to front of sub array */
          first := cats[index];
          smallest := cats[lowestIndex];
          cats[index] := smallest;
          cats[lowestIndex] := first;
        enddo;

        for index := 0 to size - 1 do
          printi(cats[index]);
        enddo;
      end
      return 0;
   end

end
```