Tiger Language Reference Manual

Table of Contents

Overview

Credit: Modified from Stephen A. Edwards' "Tiger Language Reference Manual" and from Appel's "Modern Compiler Implementation in C ".

Grammar

<tiger-program> → `main let` <declaration-segment> `in begin` <stat-seq> `end`

<declaration-segment> → <type-declaration-list> <var-declaration-list> <funct-declaration-list>

<type-declaration-list> → NULL

<type-declaration-list> → <type-declaration> <type-declaration-list>

<var-declaration-list> → NULL

<var-declaration-list> → <var-declaration> <var-declaration-list>

<funct-declaration-list> → NULL

<funct-declaration-list> → <funct-declaration> <funct-declaration-list>

<type-declaration> → `type` id = <type>;

<type> → <type-id>

<type> → `array` [`INTLIT`] `of` <type-id >

<type> → id

<type-id> → `int` | `float`

<var-declaration> → var <id-list>:<type><optional-init>;

<id-list> → id

<id-list> → id, <id-list>

<optional-init> → NULL

<optional-init> → := <const>

<funct-declaration> → `function` id (<param-list>) <ret-type> begin <stat-seq> end;

<param-list> → NULL

<param-list> → <param> <param-list-tail>

<param-list-tail> → NULL

<param-list-tail> → , <param> <param-list-tail>

<ret-type> → NULL

<ret-type> → : <type>

<param>→ id : <type>

<stat-seq> → <stat>

<stat-seq> → <stat> <stat-seq>

<stat> → <lvalue> <l-tail> := <expr> ;

<l-tail> → := <lvalue> <l-tail>

<l-tail> → NULL

<stat> → `if` <expr> `then` <stat-seq> `endif` ;

<stat> → `if` <expr > `then` <stat-seq> `else` <stat-seq> `endif;`

<stat> → `while` <expr> do <stat-seq> enddo;

<stat> → `for` id := <expr> `to` <expr> do <stat-seq> `enddo;`

<stat> → <opt-prefix> id(<expr-list>) ;

<opt-prefix> → <lvalue> :=

<opt-prefix> → NULL

<stat> → `break;`

<stat> → `return`<expr> ;

<stat>→ let <declaration-segment> `in` <stat-seq> end

<expr> → <const>

→ <lvalue>

→ <expr> <binary-operator> <expr>

→ (<expr>)

<const> → INTLIT

<const> → FLOATLIT

<binary-operator> → `**` | `+` | `-` | `*` | `/` | `==` | `!=` | `<` | `>` | `<=` | `>=` | `&` | `|`

<expr-list> → NULL

<expr-list>→ <expr> <expr-list-tail>

<expr-list-tail>→, <expr> <expr-list-tail>

<expr-list-tail> → NULL

<lvalue>→ id <lvalue-tail>

<lvalue-tail> → [<expr>]

<lvalue-tail> → NULL

Lexical rules

The lexical rules are:

- **identifier**: sequence of one or more letters, digits, and underscores. Must start with a letter can be followed by zero or more of letter, digit or underscore. Case sensitive.
- **comment**: begins with /* and ends with */. Nesting is not allowed.
- **integer constant**: sequence of one or more digits. Should not have leading zeroes. Should be unsigned.
- **float constant**: Must have at least one or more digits before the decimal points followed by a decimal point. There could be zero or more digits after the decimal point. Floats are also unsigned.

Reserved (key) words

```
main        array       return      break       do          else        end
for         function    if          in          let         of          then
to          type        var         while       endif       begin       enddo
int         float
```

Punctuation symbols

```
,:;()[]{}.
```

Binary operators

```
+ - * / ** == != < > <= >= & | =
```

Assignment operators

```
:= =
```

Operator precedence

Highest to lowest. Operators in the same line belong to the same group, and they have the same precedence.

```
( )
**
* /
+ -
== != > < >= <=
& |
```

Operators

Rules:

- Binary operators take integer or float operands and return an integer or a float result.
- Comparison operators compare their operands, which may be either both integer or both float and produce the integer value 1 if the comparison holds (indicating a true) and integer 0 otherwise (indicating a false).
- The binary operators == and != can compare any two operands of the same type and return either integer 0 or 1. Integers are the same if they have the same value.

- The binary operators ** (exponent), +, -, *, and / require two operands and return a result. A mixed expression between an integer and a float is permitted. The result of a mixed expression is float, except for the ** operator.
- The exponentiation operator ** can have its left operand as integer or a float. The right operand must be an integer; it is a semantic error otherwise.
- The +, -, *, and / operators are left-associative. On the other hand, ** is right-associative, that is, a ** b ** c evaluates as b raised to c first (let the result be t) and then a is raised to t. The assignment operator is also right associative e.g. a := b := c; first evaluates c and assigns the value of c to that of b; this new value of b is then assigned to a. An integer value is auto-promoted to a float during the assignment but assignment of a float to integer is a type mismatch error.
- No aggregate operations are allowed, that is, all the operators must operate on scalar values (and not on arrays as aggregates).
- Zero (0) is considered false; one (1) is considered true.
- Parentheses group expressions in the usual way.
- The comparison operators do not associate, for example, a=b=c is a semantic error.
- The logical operators & and | are logical AND and OR operators. They take a logical and/or of the conditional results and produce the combined result.
- An aggregate operation on arrays is illegal. Arrays must be operated on an element-by-element basis.

Arrays

Tiger supports static arrays. An array of any named type can be made by `array [intlit] of type-id` of length `intlit`. Arrays can be created only by first creating a type. For example:

```
type intArray = array [5] of int;
var integers : intArray := 16;
```

Dereferencing an array can be done by creating an index expression which must be only integer type. For example:

```
A[2*i + j]
```

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 in which we store a value. As an r-value it returns a value stored in that array location.

Types

Two named types, `int` and `float`, are predefined. Additional named types can be defined or redefined (including the predefined ones) by type declarations.

There are two production rules for type in the Tiger grammar. These two rules govern:

- a type (creates an alias in a declaration)
- an array from a base type

Type equivalence enforced in the compiler is name type equivalence. That is, if variables a and b are defined to be of same named type, then they have equivalent type. If two variables are structurally equivalent (such as two arrays of the same length and type) but have different names, they do not have equivalent types.

In `let ... type-declaration ... in expr-seq? 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.

Assignment

The assignment expression `lvalue := expr` evaluates the expression and then binds the value to the contents of the `lvalue`. The assignment operator is right-associative. Aggregate assignment is illegal on arrays and must be done on an element by element basis.

Control Flow

The `if-then-else` expression, written `if expr then <stat-seq> else <stat-seq> endif` 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 third part under else clause is evaluated.

The `if-then` expression, `if expr then <stat-seq> endif`, evaluates its first expression, which must be an integer. If the result is non-zero, it evaluates the statements under then clause.

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

The `for` expression, for `id := expr to expr do <stat-seq> enddo`, 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 part <stat-seq> is evaluated with the integer variable named by id bound to the loop index. This part is not executed if the loop's upper bound is less than the lower bound.

Let

The expression `let declaration-list in <stat-seq> 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 statements in <stat-seq>.

Let scoping rules

Let statements can be nested as shown in the grammar. The scopes follow the classic block structure. Here's how the block structure works for Tiger. Each `let` statement opens a new scope which ends at the corresponding end of the `let` statement. The binding rules for `let` follow block structure are:

- A declaration of an entity (types, variables and functions) with a given name in a given scope hides its declaration from outer scopes (if any). For example, consider a declaration of a `variable : TechStudent` in inner scope, it will hide all the outer scope declarations of `TechStudent` if any.
- When a scope is closed by the corresponding end, all entities declared in that scope are automatically destroyed. That is, the lifetime of entities is limited to the scope in which they are declared and when the scope is closed, the entity declared in that scope ceases to exist.
- Lookup rules: The binding of a name is decided in the following manner: first the name is looked up in the current scope, if the look-up finds the declared name in the current scope, it is bound to the corresponding entity. If not found, the lookup proceeds to the outer scopes one by one until the name is found. If a declaration of the name is not found anywhere, it means the entity is undeclared and constitutes a semantic error. Once a name is found in a given scope, the lookup stops and does not proceed to outer scopes. As an example, consider scope 1 being the outermost scope and inner scopes being 2 and 3 respectively (innermost scope here is scope 3). Consider the lookup in scope 3. The lookup will work as follows: first the entity will be looked up in innermost scope 3, if found the name is bound to that declaration, if not the look-up proceeds to scope 2. If the entity is found in scope 2, the name will be bound to that declaration, if not the lookup continues to outermost scope 3 and the procedure is repeated. If all three lookups fail, it is a case of semantic error of undeclared entity.

Variables

A variable declaration declares a new variable and its initial value (optional). The lifetime of a variable is limited to its scope. Variables, types and functions share the same name space. Re-declaration of the same name in the same scope is illegal.

Functions

Tiger supports two types of functions:

- Procedure: A procedure declaration has no return type. Procedures are only called for their side- effects.
- Function: a function declaration 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.

Standard library

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

function printi(i : int)	Print the integer on the standard output.
function printf(f : float)	Print the float on the standard output.

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.

Sample Tiger programs

Sort

```
/* selection sort */
main let

    type catList = array[8] of int;
    var cats : catList;
    var index : int;
    var lowestIndex : int;
    var subIndex : int;
    var temp : int;
    var size : int;
    var x : int;
    var smallest : int;
    var first : int;

in begin

    cats[0] := 7;
    cats[1] := 2;
    cats[2] := 10;
    cats[3] := 20;
    cats[4] := 5;
    cats[5] := 6;
    cats[6] := 44;
    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
```

Calculate slope

```
main let

    var x1, x2, y1, y2 : int;
    var slope : int;
    var numerator : int;
    var denominator : int;
    var intercept : int;

in begin

    x1 := y1 := 1;
    x2 := y2 := 2;
    numerator := (y2 - y1);
    denominator := (x2 - x1);
    slope := numerator / denominator;
    intercept := y2 - slope * x2;

    x2 := 7;
    y2 := slope * x2 + intercept;
    slope := (y2 - y1) / (x2 - x1);

    printi(slope == 1);
    printi(intercept == 0);

end
```