Tiger Language Specification

Overview

Tiger is a block structured language similar to Pascal that allows writing programs for sensor processing. It has integer and fixed point types and nested scoping. It allows one and two dimensional arrays to be declared. It supports if-then-else, and while and for loops. Each Tiger program consists of a group of functions along with a main.

Note: Comments for the grammar statements are indicated inside /* ... */

Grammar

<tiger-program> → <type-declaration-list> <funct-declaration-list> <main-function>
<funct-declaration-list> → ϵ
<funct-declaration-list> → <funct-declaration> <funct-declaration-list>
<funct-declaration> → <ret-type> function **<id>** (<param-list>) begin <block-list> end ;
<main-function> → void main ( ) begin <block-list> end;
/* main is mandatory in every program, no parameters, no return value */
<ret-type> → void
<ret-type> → <type-id>
<param-list> → ϵ
<param-list> → <param> <param-list-tail>
<param-list-tail> → ϵ
<param-list-tail> → , <param> <param-list-tail>
<param> → **<id>** : <type-id>
<block-list> → <block> <block-tail>
<block-tail> → <block> <block-tail>
<block-tail> → ϵ
<block> → begin <declaration-segment> <stat-seq> end;
/* The body of the function is a sequence of blocks, each starts a new scope with declarations local to that scope followed by a sequence of statements */
<declaration-segment> → <type-declaration-list> <var-declaration-list>
<type-declaration-list> → ϵ
<type-declaration-list> → <type-declaration> <type-declaration-list>
<var-declaration-list> → ϵ
<var-declaration-list> → <var-declaration> <var-declaration-list>
<type-declaration> → type **<id>** = <type> ;
<type> → <base-type>
<type> → array [**<INTLIT>**] of <base-type>
<type> → array [**<INTLIT>**][**<INTLIT>**] of <base-type>
<type-id> → <base-type>
<type-id> → **<id>**
<base-type> → int | fixedpt
<var-declaration> → var <id-list> : <type-id> <optional-init> ;
<id-list> → **<id>**
<id-list> → **<id>**, <id-list>
<optional-init> → ϵ
<optional-init> → := <const>
<stat-seq> → <stat>

<stat-seq> $\rightarrow$ <stat> <stat-seq>
<stat> $\rightarrow$ <value> := <expr> ;
<stat> $\rightarrow$ if <expr> then <stat-seq> endif ;
<stat> $\rightarrow$ if <expr> then <stat-seq> else <stat-seq> endif;
<stat> $\rightarrow$ while <expr> do <stat-seq> enddo;
<stat> $\rightarrow$ for **<id>** := <index-expr> to <index-expr> do <stat-seq> enddo;
<stat> $\rightarrow$ <opt-prefix> **<id>**(<expr-list>) ;
<opt-prefix> $\rightarrow$ <value> :=
<opt-prefix> $\rightarrow$ ϵ
<stat> $\rightarrow$ break;
<stat> $\rightarrow$ return <expr> ;
<stat> $\rightarrow$ <block>
<expr> $\rightarrow$ <const>
<expr> $\rightarrow$ <value>
<expr> $\rightarrow$ <expr> <binary-operator> <expr>
<expr> $\rightarrow$ (<expr>)
<const> $\rightarrow$ **<INTLIT>**
<const> $\rightarrow$ **<FIXEDPTLIT>**
<binary-operator> $\rightarrow$ '+' | '-' | '*' | '/' | '=' | '<>' | '<' | '<=' | '>' | '>=' | '&' | '|'
/* arithmetic, comparative and logical and (&) and or (|) operators */
<expr-list> $\rightarrow$ ϵ
<expr-list> $\rightarrow$ <expr> <expr-list-tail>
<expr-list-tail> $\rightarrow$, <expr> <expr-list-tail>
<expr-list-tail> $\rightarrow$ ϵ
<value> $\rightarrow$ **<id>** <value-tail>
<value-tail> $\rightarrow$ [<index-expr>]
<value-tail> $\rightarrow$ [<index-expr>] [<index-expr>]
<value-tail> $\rightarrow$ ϵ
<index-expr> $\rightarrow$ **<INTLIT>**
<index-expr> $\rightarrow$ **<id>**
<index-expr>$\rightarrow$ <index-expr> <index-oper> <index-expr>
<index-oper> $\rightarrow$ '+' | '-' | '*'
/* + . and * are the only ones allowed in index expressions */

Lexical Rules

The scanner is expected to return the following character classes as tokens along with other information wherever necessary (such as values of constants, identifier names, etc). Each of the punctuation symbols or binary operators is detected as a token by scanner and sent to the parser for matching.

Identifier (<id>): A sequence of one or more letters, digits, and underscores. Must start with an upper or lower case letter. Identifiers are case sensitive, i.e. "abc" and "Abc" represent different identifiers.

Comment: Begins with "/*" and ends with "*/". Nesting is not allowed. No token is returned for these; comments are simply detected and then ignored by the scanner, which proceeds to the next token.

Integer literal (<INTLIT>): A sequence of one or more digits. Should not have leading zeroes. Should be unsigned.

Fixed Point Literal (<FIXEDPTLIT>): An integer (without leading zeroes) followed by a decimal point followed at least one and up to three digits (can have trailing zeroes). For example, legal fixed point literals could be: 0.0, 12.0, 12.123, 0.12, etc., examples of illegal literals are: .0 (digit before decimal missing), 12. (should have at least one digit after decimal), and 12.1234 (more than 3 digits after the decimal).

Reserved (key) words

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

Punctuation Symbols/Binary operators

, : ; ( ) [ ] + - * / = <> < <= > >= & | :=

Suggested token names for the above are respectively (in the same order):

COMMA COLON SEMI LPAREN RPAREN LBRACK RBRACK PLUS MINUS MULT DIV EQ NEQ LESSER LESSEREQ GREATER GREATEREQ AND OR ASSIGN

Operator Precedence (Highest to Lowest)

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

Grammatical and Semantic Specification

Functions

Each Tiger program must have a "main" which does not return a value (like a void in C) and can not have any input parameters. main() always occurs at the end of every Tiger program and could be preceded by zero or more functions. Functions can have parameters and return types. All functions are declared in the global scope and are thus callable everywhere.

A function can have a void return type (which means it does not return any value) or can return a value of a given type. In the latter case, a function must contain an explicit return statement with a given value in its body—in the former, such a statement must be absent. A function can be parameter-less or can have a list of input parameters with types. Arrays can be passed as aggregates via parameter passing; arrays could also be passed on an element-by-element basis. All the parameters are passed by value-copy semantics; the incoming actual arguments are copied to the formal ones and the return values are copies back upon call return.

Scoping

Tiger is a block structured language with a global scope in which all functions are defined making them accessible everywhere. Functions can not be defined in any other scope. In addition to functions, Tiger also allows the definition of derived types in global scopes that makes them accessible everywhere. Such types are typically used for implementing parameter passing in Tiger since they are accessible at the call site as well as in the function headers and bodies for defining variables.

In addition, Tiger follows a strict block-structured scoping discipline. Each function body defines a new scope (marked with "begin" and "end") and all the function parameters are accessible within the complete function body except when superseded by local block re-definitions. Each block can (optionally) have its own set of local definitions of types and variables, which persist during the block's scope and supersede any outer scope definition of a variable. Blocks can be nested (identified by "begin" and "end") and define nested scopes. The Tiger compiler follows the classic block structure rules for static binding analysis. A given name is looked up in the innermost scope and, if found, that binding takes effect for that name during that scope, otherwise, the compiler attempts to look it up in the outer scopes in the order of nesting (i.e., names in scope 3 will first be looked up in current scope, else in the surrounding scope 2, then in surrounding scope 1, etc) . The resolution stops when a name is found in an outer scope and otherwise can continue all the way until the parameters' scope and finally into the global scope. If no binding can be established, the compiler reports a semantic error due to an undeclared value. The binding is applicable to both types as well as names. In the presence of multiple entities, as per the above block structured look-up, the innermost binding is therefore established and the entity declaration in the outer scope is superseded by the inner scope. Types and variables share their name spaces and, therefore, it is illegal to have the same name for a type and a variable. The lifetime of a variable is limited to its scope.

Types

Tiger has two base types, integers ("int") and fixed point values ("fixedpt"), which are available for declaring values throughout the program. In addition, one can declare derived types from these base types at the beginning of a Tiger program that are in global scope and therefore globally available to declare variables using them. Two types of type derivations are supported in Tiger: (a) renaming a base type and (b) creating a one or two dimensional array type from base type. The type equivalence enforced in the compiler is name type equivalence—i.e., variables *a* and *b* defined to be of same named type are considered equivalent. In other words, even if two entities are structurally equivalent (such as two arrays of same length of integers), they will not be treated as equivalent by the compiler. In fact array declarations must go via derived types as per the grammar (refer to the sample Tiger program at the end). Mixing of integers and fixed point values is allowed in Tiger for all operations. In case of arithmetic operations, the type of the result will be fixed point arising out of such mixed operations; first the integer operand will be promoted to a fixed point equivalent by the compiler and then the respective operation will be carried out. The compiler thus supports implicit type conversion. However, when both operands are of the same type, the type of the result will be of that type and no type conversion is performed by the compiler.

Operators

Operators in Tiger are classified as: arithmetic (+, -, * and /), comparative (=, <>, <, <=, >, >=) and logical (&, |). All the operators are left associative and follow precedence rules as follows (highest precedence is 1st level and lowest one is 5th level):

- 1st level : ( ) (parenthesis)
- 2nd level: *, /
- 3rd level: +, -
- 4th level: =, <>, <, <=, >, >=
- 5th level: &, |

The logical operators & and | are logical "and" and "or" operators. They take a logical "and" or "or" of the conditional results and produce true or false. Comparisons as well as arithmetic operations are possible on operands belonging to two different types but the integer operand will be (implicitly) type converted to the corresponding fixedpt value by the compiler before performing the operation. Any aggregate operation on arrays is illegal – they must be operated on on an element-by-element basis. Moreover, array index expression can only have integer operands (constants or variables) and can only be formed using the operators, +, - and *; the resulting index can therefore only be an integer. A negative array index should cause an exception and is illegal. Arrays are indexed from 0 onwards.

Control Flow

The if-then-else expression, written "if <expr> then <stat-seq> else <stat-seq> endif" evaluates the first expression, which must return a true or false value (if the type of this <expr> is not Boolean, it is a semantic error). If the result is true, the statements under the then clause are evaluated, otherwise the third part under the else clause is evaluated.

The if-then expression, "if <expr> then <stat-seq> endif" evaluates its first expression, which must be true or false (if the type of <expr> is not Boolean it is a semantic error). If the result is true, it evaluates the statements under the then clause.

The while-do expression, "while <expr> do <stat-seq>" evaluates its first expression, which must return a true or false (if the type of <expr> is not Boolean, it is a semantic error). If the result of <expr> is true, 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>" 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. The loop is executed zero times (skipped) if the loop's upper bound is less than the lower bound.

Standard Library

`function printi(i : int)`
Print the integer on the standard output.

`function printf(f : fixedpt)`
Print the fixed-point number on the standard output.

`function readi() : int`
Read an integer from the standard input. If the input is invalid (not an integer), terminate the program with an error message.

`function readf() : fixedpt`
Read a fixed point number from the standard input. If the input is invalid (not an integer), terminate the program with an error message.

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

`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 (Scalar Dot Product)

```
main()
begin
  begin
    type ArrayInt = array [100] of int; /*Declare ArrayInt as a new type */
    var X, Y : ArrayInt := 10; /*Declare vars X and Y as arrays with initialization */
    var i, sum : int := 0;
    begin
      for i := 1 to 100 do /* for loop for dot product */
        sum := sum + X[i] * Y[i];
      enddo
      printi(sum); /* library call to printi to print the dot product */
    end;
  end;
end;
```