RE description of PL/0 lexical structure

Program ::= (Token | Whitespace)*

Token ::= Id | Integer | Keyword | Operator | Punctuation

Punctuation ::= **;** | **:** | **.** | **,** | **(** | **)**

Keyword ::= **module** | **procedure** | **begin** | **end** | **const** | **var** | **int** | **if** | **then** | **while** | **do** | **input** | **output** | **odd**

Operator ::= **:=** | * | / | **+** | **-** | **=** | **<>** | **<=** | **<** | **>=** | **>**

Integer ::= Digit$^+$

Id ::= Letter AlphaNumeric*

AlphaNumeric ::= Letter | Digit

Digit ::= **0** | ... | **9**

Letter ::= **a** | ... | **z** | **A** | ... | **Z**

Whitespace ::= <space> | <tab> | <newline>

EBNF description of PL/0 syntax

Program ::= **module Id ;** Block **Id .**

Block ::= DeclList **begin** StmtList **end**

DeclList ::= { Decl **;** }

Decl ::= ConstDecl | ProcDecl | VarDecl

ConstDecl ::= **const** ConstDeclItem { **,** ConstDeclItem }

ConstDeclItem ::= **Id :** Type **=** ConstExpr

ConstExpr ::= **Id** | **Integer**

VarDecl ::= **var** VarDeclItem { **,** VarDeclItem }

VarDeclItem ::= **Id :** Type

ProcDecl ::= **procedure Id (** [FormalDecl { **,** FormalDecl }] **) ;** Block **Id**

FormalDecl ::= **Id :** Type

Type ::= **int**

StmtList ::= { Stmt **;** }

Stmt ::= CallStmt | AssignStmt | OutStmt | IfStmt | WhileStmt

CallStmt ::= **Id (** [Exprs] **)**

AssignStmt ::= LValue **:=** Expr

LValue ::= **Id**

OutStmt ::= **output :=** Expr

IfStmt ::= **if** Test **then** StmtList **end**

WhileStmt ::= **while** Test **do** StmtList **end**

Test ::= **odd** Sum | Sum Relop Sum

Relop ::= **<=** | **<>** | **<** | **>=** | **>** | **=**

Exprs ::= Expr { **,** Expr }

Expr ::= Sum

Sum ::= Term { (**+** | **-**) Term }

Term ::= Factor { (* | /) Factor }

Factor ::= **-** Factor | LValue | **Integer** | **input** | **(** Expr **)**

Informal Definition of the Extended PL/0 Language

Comments

The extended language allows comments. A comment begins with a pound sign ("#"), and extends to the end of the line. Comments are treated as whitespace (not tokens!). The following code illustrates some comments:

```
i := x + y;  # this is a comment
j := a - - b; # this computes "a - (-b)"
# another comment; still in the comment # more comment
```

Underscores in Identifiers

The extended language allows underscores ("_") to appear anywhere that a letter can appear in an identifier. Underscores are significant. For examples, the following are all distinct, legal identifiers:

```
foo1  _foo1  f_o__o1  foo_1__  _  __  __1
```

If/Then/Else Statements

The extended language allows an optional `else` clause in `if` statements. For example, the following illustrates some legal `if` statements:

```
if i > j then
  if k < m then <some statements> end;
else
  if m = n then <some other statements>
  else <some more statements>
  end;
end;
```

Note that the explicit `end` keyword avoids the dangling-`else` ambiguity problem.

For Loops

The extended language supports a simple form of `for` loop with the following general form:

```
for <id> := <initial_expression> to <final_expression> do
  <some statements>
end;
```

The semantics of the `for` loop can be expressed as a translation into a similar `while` loop:

```
<id> := <initial_expression>;
while <id> <= <final_expression> do
  <some statements>
  <id> := <id> + 1;
end;
```

The loop index variable should be declared as an integer variable in an enclosing scope. The initial expression is evaluated once before entering the loop; the final expression is evaluated at each iteration. It is legal for the initial expression to be greater than the final expression; in this case, the body of the loop is never executed. Assignments to the loop index variable within the loop body are allowed.

Break Statements

The extended language supports the `break` statement. The break statement has the effect of terminating the immediately enclosing `while` or `for` loop. It is an error for a `break` statement to appear not nested inside a loop.

First-Class Boolean Values

In the extended language, the results of a boolean expression can be used in any context expecting an expression, including storing boolean values into variables. A type, `bool`, is supported as the type of such variables. Two new boolean constants are supported, named `true` and `false`. Relational operators and `odd` have lower precedence than the integer operations, as in the base language. The following examples illustrate first-class boolean values:

```
var a:bool;

...

a := true;
b := i < j;
if b then
   b := odd k + m; # parses as "odd (k + m)"
else
   a := false;
end;
```

And and Or Operators

The extended language supports two new boolean operators: `and` and `or`. Both operators take boolean expressions as arguments and return a boolean value. Both are short-circuiting, meaning that the second argument is not evaluated if the result of the first argument determines the outcome of the test. The `and` operator has higher precedence than `or`, but lower precedence than all other operators. The following examples illustrate the use of `and` and `or`:

```
if a = 0 or x / a > 3 and b then <stmts> end;
# test parses as "(a = 0) or (((x / a) > 3) and b)"
# if a = 0, then don't evaluate rest of test
```

Constant Expressions

Named constants can be initialized with arbitrary integer and boolean expressions over integer literals, boolean literals, and named constants. The following examples illustrate some legal constant declarations:

```
const a:int = 3 + -4;
const b:int = 5 * a;
const c:bool = a > b or odd b;
```

Arrays

The extended language supports arrays. The type of an array is described as follows:

```
array [ <int expr> ] of <type>
```

The array type specifies the size of the array and the type of the elements. The size of an array must be a positive (non-zero) integer constant expression. The type of the elements of an array can be of any type, including another array. For example, the following are some legal array variable declarations:

```
var a:array [10] of int;
var b:array [25] of array [100] of bool;

const size:int := 5 * 100;

var c:array [size] of array [size + 1] of int;
```

Array Indexing

In the extended language, the following notation references an array element:

```
<array expr> [ <index expr> ]
```

The array expression should evaluate to some array *a*, of size *n*, and the index expression should evaluate to some integer *i*. This expression returns the i^{th} element of the array *a*, where the first element of the array is at index 0. It is an unchecked run-time error for *i* to be out of the range [0..*n*). The array expression can be another array indexing operation, in the case of accessing a multi-dimensional array. Array indexing expressions are allowed on the left-hand side of an assignment statement, to store into an array element. The following code fragment illustrates a legal use of arrays:

```
var a:array [5] of array [3] of int;
var b:array [3] of int;

a[1][i+j] := b[k-1] * 2;
```

Assignment of whole arrays or array slices is disallowed. Consequently, the following statement is illegal:

```
a[1] := b;
```

Call-By-Reference

In the extended language, a formal parameter can be annotated as being passed by reference by prefixing it with the `var` keyword. The following is an example of such a procedure:

```
procedure foo(var x:int, y:int, z:array [10] of int);
  begin
    x := x + y;
    y := y + x;
    z[5] := x * y;
  end foo;
```

The caller of a procedure containing a call-by-reference formal parameter must use an lvalue expression (the name of an assignable variable or an array index expression) as the corresponding actual parameter to the procedure; other expressions as actuals to a call-by-reference formal are illegal. The following calls are legal:

```
var a:int, b:array [10] of int;
```

```
foo(a, 17, b);
foo(b[5], a+b[7], b);
```

while the following calls are illegal:

```
const c:int = 5;
var d:array [5] of int;
foo(c, 17, b);
foo(17, a, b);
foo(a+b[7], 20, b);
foo(a, 17, d)
```

Any formal can be assigned within the body of a procedure. However, only if the formal is a call-by-reference formal does the assignment affect the caller's variables.

Arguments of scalar types can be passed either by value or by reference. Arrays must be passed by reference; the size of the array being passed must match the type of the array formal.

Functions and Function Calls with Results

In the extended language, a procedure may optionally return a result; such a value-returning procedure is called a function. To indicate that a procedure returns a result, a result type declaration is added after the list of formal parameters in the procedure declaration, as follows:

```
procedure <id> ( <formal1>, ..., <formaln> ) : <type> ;
    <block>
    <id>
```

The following are some legal function declarations:

```
procedure foo(x:int, y:bool):int;
    begin <stmts> end foo;
procedure bar():bool;
    var x:int; begin <stmts> end bar;
```

Function calls can be used anywhere that an expression of the appropriate type can be used; syntactically, function call expressions have higher precedence than other operators. Functions may also be called like procedures, as a separate statement. For example, given the above function declarations, the following is a legal statement:

```
foo(foo(3, bar()), i < foo(x+y, bar()));
```

The type of the result of a function must be a scalar type; whole arrays cannot be returned. The result is returned by value.

Return Statements

The `return` statement can be used to exit a procedure, written as follows:

```
return;
```

This construct immediately transfers control to the end of the procedure body. It is not an error to "fall off" the end of a procedure without encountering an explicit `return` statement.

To return a value from a function, an alternative form of the return statement is used, as follows:

```
return <expr> ;
```

The type of the expression returned must match the type declared for the enclosing function. It is illegal for a value-returning `return` statement to appear in a procedure not declared to return a result; it is similarly illegal for a non-value-returning `return` statement to appear in a function. It is an error to for execution to reach the end of a function's body without executing a `return` statement. The following function declaration illustrates some legal uses of `return` statements:

```
procedure foo(x:int, y:bool):int;
  begin
    if y then
      return x;
    end;
    return x * 5;
  end foo;
```