

The Language

The language you are to compile is in some ways similar to Pascal, but uses fomr syntax from C and Ada. It supports nested functions, one-dimensional arrays and numeric data types.

1 The Program Structure

A program consists of three sections, as depicted in figure 1. The first section, *declarations* holds declarations of all global variables. The next section, *functions* holds all the functions defined in the program. The final section, *body* is a code block representing the main program body.

Figure 1 The structure of a program.

```
// First section, declarations

declare
    var1 : type1;
    var2 : type2;

// Second section, functions

function func1 ( params ) : type
begin
    ...
end;

function func2 ( params ) : type
begin
    ...
end;

// Final section, main program code block

begin
    ...
end;
```

2 Function Definitions

The structure of a function definitions is shown in figure 2. It starts out with the keyword `function`, followed by the function's name, its parameters and return type. Next come a block of local variable declarations and then local function declarations. The function is concluded with a code block for the function body.

Figure 2 Structure of a function definition.

```
function name ( param1 , param2 , ... ) : type
declare
    var1 : type1;
    var2 : type2;

    function local1 ( params ) : type
    begin
    end;

begin
    Function body;
end;
```

Functions that are declared within another function have access to the local variables and parameters of the surrounding function. The language has static scope.

Figure 3 shows some valid function definitions.

3 Declaration Blocks

Declarations can appear either at the beginning of a program, or at the beginning of a function. The purpose of a declaration block is to define the names and types of variables used in subsequent code blocks.

A declaration block starts with the keyword `declare`, followed by one or more declarations. Each declaration is an identifier followed by a colon and a type. The declaration block is terminated by the start of anything that does not look like a declaration. All declaration blocks are optional and may be omitted completely.

There are examples of declaration blocks in figures 1, 2 and 3.

4 Code Blocks

A code block starts with the keyword `begin`, which is followed by zero or more statements. The code block is terminated by `end`. Each statement must be terminated by a semicolon.

There are five types of statements: if statements, assignments, return statements, function calls and while statements. Each type of statement is described

Figure 3 Examples of valid function definitions.

```
function fac ( x : integer ) : integer
begin
    if x == 1 then begin return 1; end
    else begin return x * fac(x - 1); end if;
end;

function max3 ( x : array 5 of real ) : real
declare
    tmp : real;
    i : integer;
begin
    i := 1;
    tmp := x[0];
    while i < 5 do
        begin
            if tmp < x[i] then begin tmp = x[i]; end;
            i := i + 1;
        end while;
    return tmp;
end;
```

below.

4.1 The if statement

An if statement starts with the keyword `if`, followed by a condition, the keyword `then`, a code block, an optional `elseif` list and an optional `else` part. The structure is shown in figure 4.

The `elseif` list is a list of one or more pieces of code consisting of the keyword `elseif`, followed by a condition, the keyword `then` and a code block.

The `else` part is a piece of code consisting of the keyword `else` followed by a code block.

The last `end` in an if statement must be followed by the keyword `if`.

The if statement tests each of its conditions, specified after the `if` or `elseif` in order. If a condition is true, then the code block corresponding to that condition is executed, and when it is finished, execution resumes at the statement following the if statement. If none of the conditions are true and there is an `else` part, the code block in the `else` part is executed. If there is no `else` part, execution is simply resumed at the next statement.

Figure 5 shows some valid if statements.

4.2 The Assignment Statement

An assignment statement is simply an identifier or a reference to an array element followed by `:=` and an expression. Assignments to entire arrays are not

Figure 4 Structure of an if statement.

```
if condition then
    then part;
elseif condition then
    elseif part;
elseif condition then
    elseif part;
else
    else part;
end if;
```

Figure 5 Examples of valid if statements.

if x == 1 then	if x > y then
begin	begin
y := 3;	out := 1;
end if;	end
	elseif x < y
if x == 1 then	begin
begin	out := -1;
result := a;	end
end	else
elseif x == 2 then	begin
begin	out := 0;
result := b;	end if;
end if;	

allowed, nor are assignments to functions.

The result of the expression is converted as necessary. If it is an integer, it may be converted to a real; if it is a real, it may be truncated.

Figure 6 shows some valid assignments.

Figure 6 Examples of assignment statements.

```
x := 3;
x := 2 - (x * 2);
x := fac(y);
```

4.3 The Function Call

A function call consists of an identifier, followed by a left parenthesis, a comma-separated list of expressions and a right parenthesis. When used as a statement, the result of the function call is simply discarded.

Figure 7 contains some valid function call statements.

Figure 7 Examples of function call statements.

```
ackerman(2,3);
fac(4);
initialize();
```

4.4 The Return Statement

A return statement consists of the keyword `return` followed by an expression.

The return statement causes the currently executing function to return the value of the expression. Calling return from the body of the main program, or with the wrong data type, is an error.

4.5 The While Statement

A while statement consists of the keyword `while`, followed by a condition, the keyword `do`, a code block and finally the keyword `while`. Figure 8 shows the structure of a while statement.

While loops are the only kind of loop in the language. The loop body is executed until the condition is false at the beginning of the loop body. The condition is also tested before entering the loop for the first time.

Figure 9 shows some valid while statements.

5 Types

There are only two predefined types, `integer` and `real`. Integers are signed 32-bit integer quantities. Reals are double-precision floating point numbers.

Figure 8 Structure of the while statement.

```
while condition do
begin
    Loop body;
end while;
```

Figure 9 Examples of valid while statements.

```
x := 1;
while x < 10 do
begin
    y := y + calculate(x)
    x := x + 1;
end while;

while x == 1 and x < 5 do
begin
    x := random();
end while;
```

It is possible to construct arrays from integers and reals. The syntax for an array is *array size of type*, where *size* is the number of elements in the array (an integer constant) and *type* is the element type (either *integer* or *real*.)

6 Identifiers

An identifier is an arbitrarily long string of characters. The first character must be a letter. Any subsequent characters must be letters, digits or underscores. Here are some valid identifiers: `tmp`, `g_04`, `integer_constant`.

7 Numeric Constants

There are two kinds of numeric constants. An integer constant consists of a string of digits. A real constant consists of a string of digits, a decimal point, a string of digits and an optional exponent. There must be at least one digit on one side of the decimal point. The optional exponent consists of the letter 'E' an optional sign and an integer. A real constant can also be a string of digits followed by an exponent.

Here are some valid integers: 123, 4711, 17

Here are some valid reals: .12, 1.2, 3., 1.2E-3, .3E44, 12E5

8 Array References

An array reference consists of an identifier, a left square bracket, an expression and a right square bracket. Array elements are numbered from 0 and up, so `a[0]` is the first element in the array `a` and `a[1]` is the second element.

The expression used as the index must return an integer.

9 Expressions

Expressions are used on the right-hand side of assignments, as the index in array references. Expressions may contain the following binary operators:

`x ^ y` The result is `x` raised to the power of `y`.

`- x` The result is the unary negation of `x`.

`x * y` The result is the multiplication of `x` and `y`.

`x / y` The result is `x` divided by `y`.

`x + y` The result is `x` plus `y`.

`x - y` The result is `x` minus `y`.

If all operands are of the same type, then the result will also be of that type. This means that division of two integers will produce a truncated value. If one of the operands is an integer and the other is a real, then the integer must be converted to a real before executing the operations. Arrays are not permitted in expressions, although references to array elements are.

The precedence levels (highest first) and associativity of the operators are:

<code>^</code>	Right-associative
<code>-</code>	(unary minus)
<code>*</code> , <code>/</code>	Left-associative
<code>+</code> , <code>-</code>	Left-associative

Parentheses may be used in the conventional manner to force the order of evaluation.

Expressions may also contain numeric constants, variables, array references and function calls.

10 Conditions

Conditions consists of binary relations and logical operators. The following operators exist (both `x` and `y` are expressions):

`x >= y` True if `x` is greater than or equal to `y`.

`x <= y` True if `x` is less than or equal to `y`.

`x > y` True if `x` is greater than `y`.

`x < y` True if `x` is less than `y`.

$x == y$ True if x is equal to y.

$x <> y$ True if x is not equal to y.

The logical operators are

a and b True if a and b are both true.

a or b True if one or both of a and b are true.

not c True if c is not true.

Parenteses may be used within conditions in the same way as in expressions. The binary relations have higher precedence than the logical operators, so **a == b and c < d** is the same as **(a == b) and (c < d)**. Among the logical operators, **not** has higher precedence than **and**, which has higher precedence than **or**. Finally, the constants **true** and **false** are allowed as conditions.