The Pascal-0 Language

Pedro Vasconcelos, DCC/FCUP

September 2022

1 Overview

Pascal is a strongly-typed, high-level procedural language designed by Niklaus Wirth in the 1970s and that was popular for teaching and application development from the 1980s until roughly the 2000s. This document defines a language inspired by Pascal, called *Pascal-0* suitable as a source language for the implementation of a small compiler in the context of an undergraduate compilers course.

Pascal-0 is a small imperative language with integers, booleans, arrays and strings, basic control flow structures and functions. The most notable differences from the full Pascal language are: Pascal-0 disallows nested function and procedures and the only structured types supported are one-dimensional arrays. These restrictions greatly simplify code generation for this language compared to full Pascal.

2 Lexical Aspects

Whitespace characters (spaces, newlines or tabulations) may appear between any tokens and are ignored. Comments are delimited by `(*` and `*)` and may also occur between any tokens. Multi-line comments are allowed, but nested comments are not.

An *identifier* is a sequence of letters (`a` to `z` or `A` to `Z`), digits (`0` to `9`) or underscores (`_`) begining with a letter or underscore. Identifiers are case-insensitive, i.e. `xyz123`, `Xyz123` and `XYZ123` are all equivalent.

The following keywords are reserved and cannot be used as identifiers: `program function procedure const var begin end if then else while do for to true false div mod integer boolean string array of break`. Keywords are case-insensitive, i.e. `program`, `Program` and `PROGRAM` are all equivalent.

A *numeral* is a sequence of one or more decimal digits (`0` to `9`) representing a decimal integer value without sign; negative numbers can be obtained by applying the unary operator `-`.

A *string literal* is a sequence of zero or more printable characters between single quotes (`'`).

The following charaters are punctuation signs: `, . : ; ( ) [ ]`.

The infix operators are: `+ - * = <> < <= > >= div mod and or not :=`.

3 Grammar

In the following subsections we present the full grammar for Pascal-0. We use capital names for non-terminals, `teletype` font for keywords and operators, and *italic* font for other terminals; we also use | to separate alternatives and ε for the empty production.

We will also informally discuss the semantics of some constructs to clarify diferences between Pascal-0, standard Pascal and the C language.

3.1 Declarations

$$
\begin{array}{rcl}
\mathit{ConstDecls} & : & \texttt{const}\ \mathit{ConstDefSeq} \\
 & | & \varepsilon \\
\mathit{VarDecls} & : & \texttt{var}\ \mathit{VarDefSeq} \\
 & | & \varepsilon \\
\mathit{ConstDef} & : & \mathit{identifier}\ \texttt{=}\ \mathit{numeral}\ \texttt{;} \\
\mathit{ConstDefSeq} & : & \mathit{ConstDef}\ \mathit{ConstDefSeq} \\
 & | & \mathit{ConstDef} \\
\mathit{VarDef} & : & \mathit{identifier}\ \texttt{:}\ \mathit{Type}\ \texttt{;} \\
\mathit{VarDefSeq} & : & \mathit{VarDef}\ \mathit{VarDefSeq} \\
 & | & \mathit{VarDef} \\
\mathit{Type} & : & \mathit{BasicType} \\
 & | & \mathit{ArrayType} \\
\mathit{BasicType} & : & \texttt{integer} \\
 & | & \texttt{boolean} \\
 & | & \texttt{string} \\
\mathit{ArrayType} & : & \texttt{array [}\ \mathit{Constant}\ \texttt{..}\ \mathit{Constant}\ \texttt{] of}\ \mathit{BasicType} \\
\mathit{Constant} & : & \mathit{numeral} \\
 & | & \mathit{identifier}
\end{array}
$$

There are separate declarations for constants and variables and both are optional. Declarations for constants simply define symbolic names associated with numeric literals; declarations for variables define their types.

The types of arrays include a range of valid indices; for example `array [1..10] of integer` is the type of arrays of 10 integer values with indices starting at 1. Note that identifiers declared as constants can be used in array ranges; this is possible because (unlike variables) their value is known at compile time.

3.2 Expressions

Expr	:	*numeral*
	|	*string-literal*
	|	`true`
	|	`false`
	|	*VarAccess*
	|	*Expr Binop Expr*
	|	*Unop Expr*
	|	`(` *Expr* `)`
	|	*identifier* `(` *ExprList* `)`
VarAccess	:	*identifier*
	|	*identifier* `[` *Expr* `]`
Unop	:	`-` | `not`
BinOp	:	`+` | `-` | `*` | `div` | `mod` | `=` | `<>` | `<` | `>` | `<=` | `>=` | `and` | `or`
ExprList	:	*ExprList1*
	|	ε
ExprList1	:	*Expr* `,` *ExprList1*
	|	*Expr*

Pascal uses a single equals sign `=` for the equality relational operator; the not-equal operator is `<>`.

The syntax rules above rely on operator precedence rules for resolving ambiguity; the order of precedence is as follows.

operators	precedence	category
`not`, unary `-`	highest (first)	unary operators
`*`, `div`, `mod`, `and`	second	multiplicative operators
`+`, `-`, `or`	third	additive operators
`<>`, `=`, `<`, `>`, `<=`, `>=`	lowest (last)	relational operators

Note that, unlike languages based on C-syntax, relational operators have lower precedence than logical ones, so parenthesis are needed in expressions such as `(x>0) and (x<10)`.

Arithmetic and logical binary operators (`+`, `-`, `*`, `div`, `mod`, `and`, `or`) associate to the left, i.e., `a+b-c` should be parsed as `(a+b)-c`. Relational operators (`=`, `<>`, `<`, `>`, `<=`, `>=`) are non-associative, i.e., it is an error to write `1<x<10`.

Evaluation of logical operators `and` and `or` is short-circuiting, i.e., the right-hand expression is not evaluated when the left-hand one determines the result.

Note that relational operations are allowed only between integers and logical operations are valid only between booleans. It is a type error to use a boolean in a context where an integer is expected or vice-versa.

3.3 Statements

$$
\begin{array}{rcl}
\mathit{Stm} & : & \mathit{AssignStm} \\
 & | & \mathit{IfStm} \\
 & | & \mathit{WhileStm} \\
 & | & \mathit{ForStm} \\
 & | & \mathit{BreakStm} \\
 & | & \mathit{ProcStm} \\
 & | & \mathit{CompoundStm} \\
\mathit{AssignStm} & : & \mathit{VarAccess}\ \texttt{:=}\ \mathit{Expr} \\
\mathit{IfStm} & : & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Stm} \\
 & | & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Stm}\ \texttt{else}\ \mathit{Stm} \\
\mathit{WhileStm} & : & \texttt{while}\ \mathit{Expr}\ \texttt{do}\ \mathit{Stm} \\
ForStm & : & \texttt{for}\ \mathit{identifier}\ \texttt{:=}\ \mathit{Expr}\ \texttt{to}\ \mathit{Expr}\ \texttt{do}\ \mathit{Stm} \\
BreakStm & : & \texttt{break} \\
ProcStm & : & \mathit{identifier}\ \texttt{(}\ \mathit{ExprList}\ \texttt{)} \\
CompoundStm & : & \texttt{begin}\ \mathit{StmList}\ \texttt{end} \\
\mathit{StmList} & : & \mathit{Stm}\ \texttt{;}\ \mathit{StmList} \\
 & | & \mathit{Stm}
\end{array}
$$

Statements include assignments, conditionals, loops and procedure calls. Compound statements are non-empty sequences of statements; note that the semicolon (`;`) is a separator rather than a terminator (as C-like languages), hence the statement before `end` does not have a trailing semicolon.

Note that the alternatives for *IfStm* introduce the "dangling-else" ambiguity, e.g. a statement such as

```
if a then if b then a:=1 else a:=0
```

can be parsed in two diferent ways because the `else a:=0` statement can be matched with either `if`. This should be resolved in the usual way of associating the `else` with the lexically closer `if`, i.e. it should be parsed as

```
if a then begin if b then a:=1 else a:=0 end
```

The `for` statement

$$\texttt{for}\ i\texttt{:=}\ \mathit{expr}_1\ \texttt{to}\ \mathit{expr}_2\ \texttt{do}\ \mathit{stm}$$

can be seen as equivalent to

$$
\begin{array}{l}
i\ \texttt{:=}\ \mathit{expr}_1\texttt{;} \\
\texttt{while}\ i\ \texttt{<=}\ \mathit{expr}_2\ \texttt{do} \\
\texttt{begin} \\
\quad \mathit{stm}\ \texttt{;} \\
\quad i\ \texttt{:=}\ i+1 \\
\texttt{end}
\end{array}
$$

Note that this semantics for `for` is similar to that of the C language: expr_2 is computed at every iteration hence changes to any variables used in expr_2 inside the loop will affect the number of iterations. In standard Pascal expr_2 is computed once before the loop and this value used to compare against the loop variable at each iteration, so the number of iterations is fixed at the begining of the loop. For example, the loop

```
n := 10;
for i:= 1 to n do
   begin
    writeln(i);
    n := 20;
   end
```

will run for 20 iterations in Pascal-0 but only 10 iterations in standard Pascal.

The `break` statement is an extension to Standard Pascal that allows early termination of the enclosing `while` or `for` loop. It is an error to use `break` outside of a loop.

3.4 Procedures and Functions

$$
\begin{array}{rl}
\mathit{Proc}: & \mathit{ProcHeader}\ \mathit{ProcBody}\ \texttt{;} \\
\mathit{ProcHeader}: & \texttt{procedure}\ \mathit{identifier}\ \texttt{(}\ \mathit{ParamList}\ \texttt{)}\ \texttt{;} \\
| & \texttt{function}\ \mathit{identifier}\ \texttt{(}\ \mathit{ParamList}\ \texttt{)}\ \texttt{:}\ \mathit{BasicType}\ \texttt{;} \\
\mathit{ProcBody}: & \mathit{VarDecls}\ \mathit{CompoundStm} \\
\mathit{ParamList}: & \mathit{ParamList1} \\
| & \varepsilon \\
\mathit{ParamList1}: & \mathit{Param}\ \texttt{;}\ \mathit{ParamList1} \\
| & \mathit{Param} \\
\mathit{Param}: & \mathit{identifier}\ \texttt{:}\ \mathit{Type}
\end{array}
$$

Pascal-0 distinguishes between procedures (which do not return a value and are used only for side-effects) and functions (which return a value and may or may not produce side-effects). Note that there is no return statement. Instead the result value is defined by assigning to the name of the function; see the examples in Section 5.

Both procedures and functions can take any number of arguments and may declare local variables in the body. The body ends in a compound statement, i.e. a sequence of statements delimited by `begin` and `end`.

Parameter passing for basic types is by value; the Pascal `var` modifier for passing parameters by reference is not supported. However, arrays are passed by reference, i.e. passing an array to a procedure or function can be implemented by passing a pointer to its start address. This means that the procedure or function may modify the contents of the array; such side-effects will be observable in the caller.

Note that procedures and functions can take parameters of any type (including arrays) but functions may only return values of basic types.

3.5 Programs

Program	:	*ProgramHeader ProgramBody* .
ProgramHeader	:	`program` *identifier* ;
ProgramBody	:	*ConstDecls ProcDecls VarDecls CompoundStm*
ProcDecls	:	*Proc ProcDecls*
	|	ε

Program is the start symbol for the grammar.

A Pascal-0 program has a header followed by a body. The program body consists of constant declarations, followed by procedure declarations, then variable declarations and finally a compound statement. The scope rules are as follows:

- constants can be used in procedures, variable definitions and the compound statement;
- variables can only be used in the compound statement;
- procedures can be directly or indirectly recursive and can be used in the compound statement.

In particular, note that variables declared in the program cannot be accessed in the procedures.

4 I/O library

Pascal-0 supports only the following basic I/O functions and procedures:

`readint()` read an integer from the standard input.

`writeint(n)` write integer `n` to the standard output.

`writestr(s)` write a string `s` to the standard output.

Note that there is no procedure for reading a string. In fact, the only operation on strings (apart from assigning to variables and procedure parameters) is `writestr`. There are no operations for modifying strings; hence the only strings used by a program are the string literals in program text and they can only be used for output operations.

5 Example Programs

5.1 Sum of squares

```
(* Compute the sum of squares from 1 to 10. *)
program SumSquares;
var s : integer;
    n : integer;
begin
  s := 0;
  n := 1;
  while n <= 10 do
    begin
      s := s + n*n;
```

```
      n := n + 1
    end;
  writeint(n)
end.
```

5.2 Recursive Factorial

```
(* Compute factorial of 10 recursively. *)
program RecursiveFactorial;
function fact(n: integer): integer;
begin
  if n>0 then
     fact := n*fact(n-1)
  else
     fact := 1
end;
begin
  writeint(fact(10))
end.
```

5.3 Naive Prime Number Test

```
(* Naive prime number test *)
program PrimeNumberTest;
function is_prime(n: integer): boolean;
var d : integer;
begin
  d := 2;
  while d<n do
     begin
       if n mod d=0 then break;
       d := d + 1
     end;
  is_prime := (n>1) and (d=n)
end;
var i : integer;
begin
   i := readint();
   writeint(i);
   if is_prime(i) then
      writestr(' is prime')
   else
      writestr(' is NOT prime')
end.
```

5.4 Tabulate Fibonnacci Numbers

```
(* Build and print a table of
   the first 20 Fibonnacci numbers *)
program Fibonnacci;
```

```
const n = 20;
var
   fib : array[0..n] of integer;
   i   : integer;
begin
   fib[0] := 0;
   fib[1] := 1;
   for i := 2 to n do
      fib[i] := fib[i-1]+fib[i-2];
   for i := 0 to n do
      writeint(fib[i])
end.
```

5.5 Recursive QuickSort

```
(* Recursive QuickSort in Pascal-0 *)
program QuickSort;

const N = 10; (* Size of array to sort *)

function partition(vec : array[1..N] of integer;
                   l   : integer;
                   u   : integer): integer;
var i : integer; m : integer; temp : integer;
begin
   m := l;
   for i := l+1 to u do
      if(vec[i] < vec[l]) then
      begin
         m := m + 1;
         temp := vec[i];
         vec[i] := vec[m];
         vec[m] := temp
      end;
   temp := vec[l];
   vec[l] := vec[m];
   vec[m]:= temp;
   partition := m
end;


procedure qsort_rec(vec : array[1..N] of integer;
                    l   : integer;
                    u   : integer);
var m : integer;
begin
   if l<u then
      begin
         m := partition(vec, l, u);
         qsort_rec(vec, l, m-1);
```

```
         qsort_rec(vec, m+1, u)
      end
end;

var
   arr : array[1..N] of integer;
   i   : integer;

begin
   for i := 1 to N do
      arr[i] := readint();

   qsort_rec(arr, 1, N);

   for i := 1 to N do
      writeint(arr[i])
end.
```