Semantics of minipascal

Wuu Yang
National Chiao-Tung University, Taiwan, Republic of China
July 28, 2022

1 Rules for the minipascal Language

This minipascal language is a simplified version of the Pascsal language. If you have doubts with the minipascal language (syntax or semantics), please discuss your doubts with me. You may implement only a subset of the above features (for instance, overloading may be skipped). You will get points for the features that you implement.

- The minipascal language does not include cpp commands, such as #include, #if, etc. You may ignore them in your projects.
- In minipascal, upper-case letters and lower-case letters in names are considered equivalent. For instance, aBcD = abcd = ABCD. This is different from the rules in the C language.
- Comments are marked with two slashes and extend to the end of the line. For example,
  `// this is a comment.`
  Comments may also be enclosed in the pair /* and */, such as
  `/* hello, this is also a comment. */`
  This comments may span multiple lines. You may invent your own syntax for comments if necessary.
- The difference between `num` and `integer` tokens (please see 02-MINI-PASCAL-GRAMMAR.txt): `integer` is a keyword. It is a token. It stands for the name of a type. In contrast, `num` is the token for all numbers, such as 3.14, -999, +123, 2.71, etc. You need to define scientific

notation such as 2.503E-7 as a `num` token. You need to define a reasonable expression for num, such as 123, -34.764, 2.53E+24, +0.001E21, 99.456E-12, etc. Note that signed numbers, such as -34.764 and +9876, are difficult to handle. Please see below.

- STRING is the keyword that represents a basic type. You need to define a reasonable regular expression for string constants, such as `"abcdef"` and `"alpha\"beta"` and `"good\\bad\\ugly"`. In minipascal, string constants are used only in a printString function. There are no operators on strings. However, you may even build a library for functions related to strings (for extra points). A string constant may not span across multiple lines.

- A programmer cannot define new type names.

- We use name equivalence for type compatibility. You need to design a reasonable type system for minipascal. You can use a simple type system for minipascal.

- We disallow addition/subtraction/multiplication/division between an integer and a real number. We also disallow comparison between an integer and a real number. Furthermore, we will not test comparison between two strings. We do not allow assigning a real value to an integer variable. We do not allow assigning an integer value to a real variable either.

- A function's value is the value of the variable whose name is the same as the function. For example,

```
function  addition(a, b: integer) : integer;
  begin
     addition := a + b;  // this is the return value
  end;
```

If a function did not set up a return value, a compiler may generate an error message.

- We may assign an arry to another. For example,

  ```
  var a, b: array [ 23 .. 57 ] of integer;
  a := b;
  ```

- All parameters are passed by value.

- We can assign a whole array to a variable. For example,

  ```
      VAR a, b : array [ 1 .. 10 ] of array [ 1 .. 10 ] of Integer;
      a[5] := b[3];
  ```

- Array indices could be negative. For example,

  ```
      VAR a : array [ -5 .. 5 ] of integer ;
      a[ -5 ] := -5;
      a[ -1 ] := -10;
  ```

- To call a procdure, we simply refer to the name of the procedure plus appropriate parameters. This corresponds to the following two rules in the minipascal grammar:

  ```
  procedure_statement ::= id
  | id ( expression_list )
  ```

- We allow overloading. A function and a variable may have the same name if the type of the function's return value and the type of the variable are different. There could be multiple functions with the same name if they will never cause trouble in any minipascal programs. You will earn extra points if you implement overloading.

- Minipascal is not an object-oriented language. However, you may extend minipascal to include object-oriented features, such as objects and inheritance (extra points).

- A number could be prefixed with an optional positive/negative sign. A number could be an integer or a floating-point number. You need to define the regular expression for a number in an appropriate way. Note that the string `123-456` should be considered as three tokens `123`, `-`, and `456`. Note that the string `123--456` should be considered as three tokens `123`, `-`, and `-456`. Be careful this issue is not easy to implement correctly.

  Because signed numbers are difficult to handle, in a minpascal program, the programmer needs to write signed numbers inside a pair of parentheses, such as "`array [ (-5) .. 5 ] of integer`", "`x = 5 - (-23);`" and "`y = (+456) + (-789);`". A signed number must be enclosed in a pair of parentheses. There is no space inside the pair of parentheses. This rule would make your scanner easier. You need to write a regular expression for signed numbers. However, you could earn extra credits if your compiler can handle the usual signed numbers.

- In minipascal, there are strings, such as `"hello"`. A string constant is enclosed in a pair of double quotes. A string constant may not contain end-of-lines and double quotes unless they are properly escaped, as in C. You need to define a regular exrepssion for string constants. There are no char type, no char variables, no char constants.

- You may add overload resolution to the compiler (for extra points).
- An identifier begins with an English letter a-z and A-Z and may include English letters, digits (0-9), and the underscores _.
- Illegal characters should be reported character by character.
- You need to implement the following three built-in functions:

```
printInt( a+15 );
  printReal( b+c/3.14 );
  printString( "The size of the graph is" );
```

  We will provide you examples for the implementation of the built-in functions in RISC-V.

2 Syntax of minipascal

Additional notes:

1. In minipascal, there must be an ELSE clause.

```
if expression then statement else statement.
```

2. In minipascal, ; is used to separate two statements. If there is ony one statement, sometimes we may omit the semicoln ;.

3. There are conflict(s) if you use the following grammar to generate LR(1) parsers. Please fix the conflict(s) by modifying the grammar.

```
prog    ::= PROGRAM id ( identifier_list ) ;
       declarations
       subprogram_declarations
       compound_statement
       .
```

```
identifier_list ::= id
       | identifier_list , id

declarations ::= declarations VAR identifier_list : type ;
       | lambda

type ::= standard_type
       | ARRAY [ num .. num ] OF type

standard_type ::= INTEGER
       | REAL
       | STRING

subprogram_declarations ::=
       subprogram_declarations subprogram_declaration ;
       | lambda

subprogram_declaration ::=
       subprogram_head
       declarations
       subprogram_declarations
       compound_statement

subprogram_head ::= FUNCTION id arguments : standard_type ;
       | PROCEDURE id arguments ;

arguments ::= ( parameter_list )
       | lambda
```

```
parameter_list ::= optional_var identifier_list : type
       | optional_var identifier_list : type ; parameter_list

optional_var   ::= VAR
       | lambda

compound_statement ::= begin
       optional_statements
       end

optional_statements ::= statement_list
       | lambda

statement_list ::= statement
       | statement_list ; statement

statement ::= variable := expression
       | procedure_statement
       | compound_statement
       | IF expression THEN statement ELSE statement
       | WHILE expression DO statement
       | lambda

variable ::= id tail

tail     ::= [ expression ] tail
       | lambda

procedure_statement ::= id
       | id ( expression_list )
```

```
expression_list ::= expression
       | expression_list , expression

expression ::= boolexpression
       | boolexpression AND boolexpression
       | boolexpression OR  boolexpression

boolexpression ::= simple_expression
       | simple_expression relop simple_expression

simple_expression ::= term
       | simple_expression addop term

term ::= factor
       | term mulop factor

factor ::= id tail
       | id ( expression_list )
       | num
       | stringconst
       | ( expression )
       | not factor

addop ::= + | -

mulop ::= * | /

relop ::= <
       | >
```

```
        | =
        | <=
        | >=
        | !=
```