Language description

This manual describes the syntax and (some) semantics of a simple procedural language. Please note that some of the examples in this description might be legal syntactically, but are not semantically. You should be mindful of those cases.

Lexical Description

Keywords Lexemes

- `bool`
- `char`
- `int`
- `real`
- `string`
- `int*`
- `char*`
- `real*`
- `if`
- `else`
- `while`
- `for`
- `var`
- `func`
- `proc`
- `return`
- `null`

Operator Lexemes

We support the following operators, which follow the operator precedence table from the language C:

- `&&`
- `/`
- `=`
- `==`
- `>`
- `>=`
- `<`
- `<=`
- `-`
- `!`
- `!=`

- `||`
- `+`
- `*`
- `&`
- `^`

Literal Lexemes

- boolean: `"true"` or `"false"`
- char: A character literal is a single, printable character, enclosed in single quotes.

  Examples:

  ```
  'a' : lowercase a
  'A' : uppercase a
  "a" : not a character; there are double quotes, and hence, this
  is a string
  ```

- integers (int): An integer literal can be a decimal or hex.

  Examples:

  ```
  100     : Decimal (cannot start with Zero if it is NOT zero)
  0x01F   : Hex (any number beginning with 0x or 0X and digits 0-
  9,A,B,C,D,E,F)
  ```

- reals (similar to double in C language)

  Examples: 3.14, -34.9988, 45.3E-23, -4E+2101, +.2E4, 4.e-67

- string: A string is an array of characters. It is written as a sequence of characters enclosed by double quotes. The closing double quotes must be on the same line as the opening quotes. That is, they cannot be separated by a newline. To make things easier, a string cannot contain a double quote character, since this would terminate the string.

  Examples:

  ```
  "this is a string"       : simple string that contains 16
  character
  "this is \"invalid\""    : invalid string, double quotes cannot
  be escaped
  "this is no newline\n"   : string that contains 20 characters,
  including a backslash and a lowercase n
  ""                       : empty strings are okay
  ```

- identifier: An identifier literal can be a variable, procedure or function name. Identifiers must start with an alpha character (upper or lowercase letter), followed by zero or more digits, "_", and/or other alpha characters.

- pointers: A pointer is a type that *points to* a value of its base type.

  Technically, a pointer variable stores the memory address of the value (or the address of the variable) that it points to. An integer/double pointer can only point to integer/double variables. A char pointer can point to a char variable, or an element of a string (as a string is nothing else than an array of characters).

  There are two operators that are only valid for pointers. One is the dereference operator (using the '^' character). The dereference operator allows us to directly access the variable that the pointer references (that is, the variable that it points to). The second operator that can be used in connection with pointers is the `address of` operator (using the '&' character). This operator can only be applied to integer variables, real variables, character variables, and string (character array) elements. It takes the address of this variable, and its result can be assigned to a pointer of the appropriate type.

  Finally, there is a special keyword (token) that represents a pointer that points nowhere (an empty pointer, or an invalid pointer). This keyword is `null`.

  Examples:

```
var x : char*;    /% x is a pointer to a character variable %/
var x, z : int;
var y    : int*;

x = 5;
y = &x;     /% We take the address of x and assign it to y.
               As a result, y points to x, which is 5. %/
x = 6;      /% y still points to x, which is 6 now %/
z = ^y;     /% Dereference y, and assign to z the value that y
points to (which is 6). %/
y = null;   /% y is now the NULL pointer %/

z = ^^y;    /% illegal; you can only use a dereference operation
once %/
```

```
&x = y;     /% illegal; cannot use the address operator on the
left hand side of an assignment %/
```

Other Lexemes

Lexem	Use	Example
`;`	Each statement ends with a semicolon	`i = 0;`
`:`	Variable declarations use colons	`var i : int;`
`,`	Used in parameter lists	`func foo(i, j : int) {}`
`|`	For strings: Declared length of string s	`|s|`
`{`	Start block of code	
`}`	End block of code	
`(`	Begin parameter list	
`)`	End parameter list	
`[`	Begin string (character array) index	
`]`	End string (character array) index	

Description of Program Structure

Comments

Comments in this language are block comments (C-style). The form is /% comments %/.

Correct (legal):

```
/% this is my comment %/
```

Incorrect (illegal):

```
/* wrong language */
```

Programs

A program is composed of many functions and procedures, just listed one after another. Every legal program should have one and only one procedure: 'Main()'. This is case sensitive, so main() is incorrect. Of course, a program can have user defined functions and procedures too. Any procedure must be defined before the point of call. The difference between function and procedure is that function should return value of some type and procedure does not return any value.

Correct (legal):

```
func foo() return int
{
    return 0;
}

proc Main()
{
    var a : int;
    a = foo();
}
```

Incorrect (illegal): foo is used before it is declared

```
proc Main()
{
    var a : int;
    a = foo();
}

func foo() return int
{
    return 0;
}
```

Procedures and function

Functions are declared as:

```
"func" id "(" parameter_list ")" "return" type "{" body "}"
```

Procedures are declared as:

```
"proc" id "(" parameter_list ")" "{" body "}"
```

Note the placement of the "()" and "{}" symbols. These must go exactly there. *id* is the name of the procedure or function and must follow the keyword "proc" or “func” respectively. Read below for more details. *parameter_list* are the parameters you have declared for the procedure of function. This list can be empty. The types of the function and procedure arguments must be either `bool`, `char`, `int`, `real`, `char*`, `real*`, or `int*`. *type* is the type of the return value of the function and must be either `bool`, `char`, `int`, `real`, `char*`, `real*`, or `int*`. *body* contains procedure declarations, variable declarations, and statements.

You may declare one or more functions and procedures inside the body of a procedure or procedure, thus, nested procedures and functions are possible with this language. The last statement in a function must be a return statement, and it can also appear anywhere within a code block.

Correct (legal):

```
func foo(i, j, k : int) return int
{
    func fee(l, m, n : int) return bool
    {
        return true;
    }
    return 0;
}


proc goo(i, j, k : int)
{
    func fee(l, m, n : int) return bool
    {
        return true;
    }
    fee(2,3,4);
}
```

The *id* can be any string starting with an alpha character (upper or lowercase letter) and can contains digits, "_", or other alpha characters.

Correct (legal):

```
func foo() return int { return 0; }
func foo_2() return int { return 0; }
func f234() return int { return 0;}
```

Incorrect (illegal):

```
func 9foo() return int { return 0; }
func _rip() return int { return 0; }
```

A *parameter_list* is somewhat complicated. You can pass multiple types of variables, and as many variables as you want. However, you must list the same variable types together and separate them with a comma. You must separate different types with a semicolon. The correct syntax is as follows:

```
"("  IDA_0  ","  ...  ","  ID0_N  ":"  TYPE1  ";" IDB_0  ","  IDB_N
":"  TYPE2;  ...  TYPEN  ")"
```

Notice that the last type does not have a semicolon after it. If you only pass in one type of variable, you would not need to have a semicolon and putting one in should produce an error.

Correct (legal):

```
func foo(i, j, k : int; l, m, n : bool) return int { return 0; }
proc fee(a, b : int)  { }
func fei(a, b, c : int; d, e, f : bool; g, h : int) return int { return
0; }
```

Incorrect (illegal):

```
proc foo(i, j, k) { } /% no type defined %/
proc foo(i j k : int) { } /% IDs must be separated by comma %/
```

Body

The *body* can contain nested function/procedure declarations, variable declarations, and statements after all declarations. This makes our language very much like C, because you must declare everything first.

Correct (legal):

```
func foo(i, j, k : int) return int
{
    func square(t : int) /% func/proc declarations %/ return int
    {
        var temp : int;
        temp = t*t;
        return 0;
    }

    var total : int;          /% variable declarations %/

    total = 1;                    /% statements %/
    return total;
}
```

Variable Declarations

Variables are declared in the following syntax:

```
"var"  ID1  ","  ID2  ","  ID3  ","  ...  ","  IDN  ":"  TYPE  ";"
```

Variables must first be declared before they can be assigned. This is the only way to declare them.

Correct (legal):

```
var i : int;
var m, n : bool;
var c : char;
var s : string[20];
```

Incorrect (illegal):

```
var i = 5 : int;
```

Strings (character arrays)

Arrays are declared with the following syntax:

```
"var"  ID1  ","  ID2  ","  ID3  ","  ...  ","  IDN  ":"  "string"  "["
INTEGER_LITERAL  "]"  ";"
```

Strings can be assigned as a normal variable. You can also assign string literals to string variables. Individual string elements can be assigned character values, or they can be used as part of an expression. Their indexing element is also an expression. By using the bar |s|, one can compute the length of the string as it was declared.

Correct (legal):

```
var a, b : string[100];
var c : char;
var i: int;
c = 'e';
a[19] = 'f';
a[4+2] = 'g';
b = a;
b[3] = c;
a = "test";   /% basically equivalent to a[0] = 't'; a[1] = 'e'; a[2] =
's'; a[3] = 't'; a[4] = '\0'; %/
i = |s|;      /% this assigns 100 to variable i, since the length
operator returns the size of the character array %/
```

Essentially, a string element is exactly like a character type and the string variable itself is simply a new type. The following are not legal uses of strings:

Incorrect (illegal):

```
var a, b : string[100];
var c : char;
c = 'e';     /% everything up to this is OK %/
c = a;  /% type mismatch, can't assign string type to character type %/
    (a + 4)[0] = 'e';
```

```
    /% cannot add anything to array elements - they are not pointers %/
```

Statements

Statements can be many things: an **assignment** statement, a **function call** statement, an **if** statement, an **if-else** statement, a **while** statement, a **code block**, etc.

The syntax for an assignment statement is:

```
lhs "=" expression ";"
lhs "=" STRING_LITERAL ";"
```

Here, lhs -- which stands for left-hand side (of the assignment) -- specifies all legal targets of assignments. Specifically, our grammar accepts three different lhs items:

```
   x = expr;                /% lhs is variable identifier %/
   string[expr] = expr;     /% lhs is string element %/
   ^ptr = expr;             /% lhs is dereferenced pointer %/
```

We cannot assign values to arbitrary expressions. After all, what sense would make a statement such as

```
(5+2) = x;
```

Thus, we have to limit the possible elements that can appear on the left-hand side of the assignment as discussed above.

The right-hand side of assignments is less restrictive. It includes expressions, as well as string literals (we have to mention string literals explicitly, since strictly speaking, they are not expressions).

A **code block** starts with a "{" and ends with a "}". It may contain variable declarations and statements (again, in this specific order). Both variable declarations and statements are optional. Thus, a code block can be empty. Of course, since a code block is a statement, code blocks can be nested within code blocks.

Correct (legal):

```
func foo() return int
{
    var x : int;
    {
        var y : int;
        x = 1;
        y = 2;
        {
            x = 2;
        }
        y = 3;
     }
    return 0;
}

func foo() return int
{
    {
        {}   /% empty code blocks are okay, although not very useful %/
    }
    return 0;
}
```

Incorrect (illegal):

```
func foo() return int
{
    var x : int;
    {
        x = 1;
        var y : int;
        /% must declare all variables before any statement %/
    }
    return 0;
}
```

The syntax for a function call statement is:

```
lhs "=" function_id "(" expression0 "," expression1 "," ... expressionN
")" ";"
```

The syntax for if, if/else, for and while statements is shown below.

```
"if" "(" expression ")" "{" body_of_nested_statement "}"

"if" "(" expression ")" statement;

"if" "(" expression ")" "{" body_of_nested_statement "}"
"else" "{" body_of_nested_statement "}"

"if" "(" expression ")" statement; "else" statement;
…
```

```
"while" "(" expression ")" "{" body_of_nested_statement "}"

"while" "(" expression ")" statement;

"for" "(" inits ";" expression ";" updates ")"
"{" body_of_nested_statement "}"

"for" "(" inits ";" expression ";" updates ")" statement;
```

Here, *body_of_nested_statement* is similar to a code block, in that it may contain variable declarations and statements (in this specific order). The body of a nested statement can be empty.

Return Statement

The last statement in a function must be a return statement. The syntax for the return statement is:

```
return expression
```

Correct (legal):

```
func foo() return int { return 0; }
func foo_2() return int { var a: int; a = 2; return a; }
func foo_3() return int { if (true) { return foo(); } return 0; }
```

Incorrect (illegal):

```
func foo_3() return int {return true; }
func foo_3() return int {if (true) {return foo(); } }
```

Expressions

An expression's syntax is as follows:

```
expression operator expression
   OR
operator expression
```

This implies that expressions are recursive by nature. Look at how statements are defined though! Statements and expressions are not equivalent! Expressions can be just IDS or certain LITERALS (integers, reals, characters, bool, or the null pointer). These examples assume you have declared each variable and function already. Expressions and an assignment statement are NOT equivalent! Operators have the same precedence as in C/C++.

Expressions

Correct (legal):

```
3 || 2
(3 + 2) / 3 - 5 * 2
true && false || false
5
3.234
true
-5
^x
^(p+5)
!false
a == b
```

Incorrect (illegal):

```
a = b
i = j = k = 2
&(x + y)
```

Procedure Call

Correct (legal):

```
a = foo(i, j); /% 'a' has been declared already %/
```

if/else/while statements

Correct (legal):

```
if(3 > 2)
{
    /%...statements...%/
    i = 5; /% i has been declared above%/
}
if(true) { j = 3; } else { k = 4; }
while(true) { l = 2; k = l + j; }
if(true) i = 5;
if(true) { j = 3; } else x = x -1;
while(false) x = x + 1;
```

Pointers

Note that pointers require some special attention: you cannot take the address of just any expression. This is the case because an expression might not actually have a memory address where it is stored. For instance, `&(5+3)` is undefined, because the result `8` does not have to be stored in memory but could be stored in a register instead.

Therefore, we are allowing the use of the address of operator (`&`) only on variable identifiers and string (character array) elements. When you take the address of a variable, you can use the result in an expression. However, you cannot take the address of an arbitrary expression.

When taking the address of a `string`, indexing is required (`&string` is illegal, but `&string[0]` is legal). Note that the type of `&string[0]` is `char*`.

Our language also supports some pointer arithmetic for `char` pointers: you can add and subtract from a pointer. If you add or subtract to a `char` pointer, then you should advance to the next or previous character respectively. We do not support pointer arithmetic for pointers to `integer`. Also, you cannot multiply a pointer with a value or a variable. When you add the result of an expression to a `char*` (or subtract an expression from a `char*`), the resulting type is still a `char*`.

If you perform pointer arithmetic and you point outside of your allocated string, then the behavior is undefined.

`null` assigns a value of 0 to a pointer. Note that this is *very different* from assigning the value 0 to an integer that an `int*` might reference! Instead, it means that the pointer does not point to any legal variable / value. When you dereference the `null` pointer, the result is undefined, and your program likely crashes (with a null pointer exception).

You can compare two pointers. In this case, you don't compare the values that the pointers reference. Instead, you compare the memory addresses that they point to. When two pointers reference the same variable (the same memory location), then a comparison operation yields true, false otherwise. You can also compare a pointer with `null` to check if it is valid.

Correct (legal):

```
var x : int;
var y : int*;
x = 5;
y = &x;
x = 6;
var x : char*;
var y : string[10];
var z : char;
y = "foobar";
x = &y[5];          /% x points to 'r' %/
z = ^(x - 5);       /% z is 'f' %/
y = "barfoo";       /% z is still 'f', but x now points to 'o' %/
```

Incorrect (illegal):

```
var x : bool*;      /% no such pointer type %/

x = &(1+3);

var x : char;
var y : int*;
y = &x;             /% address of x is of type char* %/

var x : char*;
var y : char;
x = &(&y);
/% can only take the address of variable or array element, and (&y) is
an expression %/
```