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`
- `elif`
- `else`
- `while`
- `for`
- `var`
- `par# (where # is the integer number)`
- `return`
- `null`
- `do`
- `returns`
- `begin`
- `end`
- `def`
- `call`
- `and`
- `not`
- `or`

Operator Lexemes

We support the following operators:

- `/`
- `=`
- `==`
- `>`

- `>=`
- `<`
- `<=`
- `-`
- `!=`
- `+`
- `*`
- `&`

Literal Lexemes

- bool: `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
                                characters
  "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 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. 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
    type char*:c;  #-> c is a pointer to a character variable <-#
    type int:x,z;
    type int*:y;
begin
    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 <-#
end
```

Other Lexemes

Lexem	Use	Example
`;`	Each statement ends with a semicolon	`i = 0;`
`,`	Used in variables	
`|`	For strings: Declared length of string s	`|s|`
begin	Start block of code	
end	End block of code	
`(`	Begin parameter list	
`)`	End parameter list	
`[`	Begin string (character array) index	
`]`	End string (character array) index	
var	Start variable declarations (until “begin”)	

Description of Program Structure

Comments

Comments in this language are block comments . The form is:

#-> comments <-#

Correct (legal):

```
#-> this is my
        comment <-#
```

Incorrect (illegal):

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

Programs

A program is composed of many functions/procedures, just listed one after another. Procedure is a function that does not return any value. 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 too. Any function must be defined before the point of call.

Correct (legal):

```
def foo(): returns int
begin
    return 0;
end

def _main_():
var
   type int:a;
begin
    a = foo();
end
```

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

```
def _main_():
var
    type int:a;
begin
    a = foo();
end

def foo(): returns int
begin
      return 0;
end
```

Functions

Functions are declared as:

```
"def" id "(" parameter_list ")": returns type
"var"
       Declarations
"begin"
       body
"end"

where type is not void.
```

Procedures are declared as:

```
"def" id "(" parameter_list ")":
"var"
       declarations
"begin"
       body
"end"
```

parameter_list are the parameters you have declared for the function. This list can be empty. The types of the function 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*`. In the case of procedure, no return type should be specified. *declarations* contain variable declarations. *body* contains function declarations and statements.

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

Correct (legal):

```
def foo(par1 int:i; par2 int:j; par3 int:k): returns int
begin
    def fee(par1 int:l; par2 char:m; par3 real:x): returns bool
    begin
        return TRUE;
    end
    return 0;
end


def goo(par1 int:i; par2 int:j; par3 int:k):
begin
    def fee(par1 real:m; par2 real:n; par3 real:x): returns bool
    begin
        return TRUE;
    end
    call fee(2,3.5,0.4);
end
```

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): `foo , foo_2, f234`

Incorrect (illegal): `9foo, _rip`

A *parameter_list:* you can pass multiple types of variables, and as many variables as you want. Each parameter should start with par# such that # represents the index of the parameters in the list. You must separate parameters with a semicolon.

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):

```
def foo(par1 int:i; par2 int:j; par3 int:k; par4 bool:l; par5 bool:m;
par6 bool:n): returns int begin return 0; end
def fee(par1 int:a): begin  end
```

Incorrect (illegal):

```
def fee(par1 a): begin  end  #-> no type defined <-#
def foo(par1 int:i, par2 int:j): begin end #-> parameters must be
separated by semicolon <-#
```

Body

The *body* can contain nested function/procedures declarations and statements after all the declarations.

Correct (legal):

```
def foo(par1 int:i; par2 int:j; par3 int:k): returns int
var                                          #-> variable declarations <-#
    type int:temp;
begin
    def square(par1 int:t): returns int    #-> func declarations <-#
    var
        type int:temp;
    begin
        temp = t*t;
        return temp;
    end
    total = 1;                              #-> statements <-#
    return total;
end
```

Variable Declarations

Variables are declared in the following syntax:

```
"type"  TYPE:ID1,ID2,…,IDN;
```

Variables may be assigned in the declaration

Correct (legal):

```
var
        type int:i:0;
        type bool:m:TRUE,ns:FALSE;
        type char:c:'a';
```

Strings (character arrays)

Arrays are declared with the following syntax:

```
"type" "string:" ID1 "[" INTEGER_LITERAL "]" "," ID2
"[" INTEGER_LITERAL "]" "," ...  "," IDN "[" 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
        type string:a[30],b[100]:"moshe",s[11];
        type char:c;
        type int:i;
begin
        c = 'e';
        a[19] = 'f';
        a[4+2] = 'g';
        b = a;
        b[3] = c;
        a = "test";
        i = |b|;
end
```

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

        type string:a[30],b[100]:"moshe";
        type char:c;
begin
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 "=" 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 <-#
   str[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 literals.

A **code block** starts with a "begin" and ends with an "end". It may contain function declarations and statements (in this specific order). Both 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 (in this case, “var” section with variable declarations may be added before the “begin” of the nested code block and these variables can be used inside this nested code block, but not outside of it).

Correct (legal):

```
def foo(): returns int
var
    type int:x;
begin
    var
       type int:y;
    begin
        x = 1;
        y = 2;
        begin
            x = 2;
        end
        y = 3;
    end
    return 0;
end

def goo(): returns int
begin
    begin
        begin end   #->empty code blocks are okay,
                                although not very useful <-#
    end
    return 0;
end
```

The syntax for a function/procedure call statement is:

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

"call" function_id "(" expression0 "," expression1 "," ... expressionN
")" ";"
```

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

```
"if" expression ":" "var" declarations "begin" nested_statements "end"

"if" expression ":" statement;

"if" expression ":" "var" declarations "begin" nested_statements "end"
"else" ":" "var" declarations "begin" nested_statements "end"

"if" expression ":" statement; "else" statement;

. . . (all different combinations of single statements and blocks)

"if" expression ":" "var" declarations "begin" nested_statements "end"
"elif" expression ":" "var" decl "begin" nested_statements "end"
"elif" expression ":" "var" decl "begin" nested_statements "end"
. . .
"else" ":" "var" decl  "begin" nested_statements "end"
```

```
"if" expression ":" statement;
"elif" expression ":" statement;
"elif" expression ":" statement;
. . .
"else" ":" statement;

. . . (all different combinations of single statements and blocks)

"while" expression ":" "var" decl "begin" nested_statements "end"

"while" expression ":" statement;

"do" ":"
"var" declarations
"begin"
      nested_statements
"end"
"while" ":" expression ";"

"for" "(" init ";" expression ";" update ")" ":"
"var" declarations "begin" nested_statements "end"

"for" "(" init ";" expression ";" update ")" ":" statement;
```

Here, *nested_statemens* is similar to a code block. The body of a nested statements can be empty.

Return Statement

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

```
return expression;

return literal;
```

Correct (legal):

```
def foo(): returns int begin return 0; end

def foo_2(): returns int
var
    type int:a;
begin
    a = 2;
    return a;
end
```

```
def foo_3(): returns int
begin
    if TRUE:
    begin
         return foo();
    end
    return 0;
end

def foo_4():
var
    type int:a;
begin
    a = 2;
end
```

Incorrect (illegal):

```
def foo_5(): returns int begin return TRUE; end
def foo_6(): returns int begin if TRUE: begin return 0; end end
def foo_7(): begin return 0; end
```

Expressions

An expression's syntax is as follows:

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

Operators have the same precedence as in C/C++.

Expressions

Correct (legal):

```
3 || 2
(3 + 2) / 3 - 5 * 2
TRUE and FALSE or FALSE
5
3.234
TRUE
-5
*x
*(p+5)
not FALSE
a == b
```

Function call

Correct (legal):

```
a = call foo(i, j);
```

Procedure call

```
call foo(i, j);
```

if/else/loop statements

Correct (legal):

```
if 3 > 2:    #-> (3>2) also correct <-#
var
    type int:a:7;
begin
    #-> ...statements... <-#
    i = a+6;
end

#-> more examples … <-#

if a+b > 4 and (num*b <=sum or not m==n):
begin j = 3; end
else:
begin
     k = 4;
     while TRUE:
     begin l = 2; k = l + j; end
end

if (TRUE): i = 5;

if not FALSE:
begin
      j = 3;
end
else: x = x-1;

while FALSE: x = x + 1;

for (i=0; i<10 and a!=sum; i=i+2):
begin a = a + i; end

do:
var
    type int:a:5;
begin
    a = a + i;
    i= i + 1;
end
while i<=10;
```

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.

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 other types. 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
   type int:x;
   type int*:y;
begin
   x = 5;
   y = &x;
   x = 6;
end

var
   type char*:x;
   type string y[10];
   type char:z;
begin
   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' <-#
end
```

Incorrect (illegal):

```
type bool*:x;         #-> no such pointer type <-#


x = &(1+3);


var
    type char:x;
    type int*:y;
begin
    y = &x;          #-> address of x is of type char* <-#
end


var
    type char*x;
    type char:y;
begin
    x = &(&y);
end
#-> can only take the address of variable or array element, and (&y) is
an expression <-#
```