CSimple: Language Manual

This manual describes the syntax and (some) semantics of CSimple, a simple 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

- `boolean`
- `char`
- `void`
- `int`
- `string`
- `intp`
- `charp`
- `if`
- `else`
- `while`
- `do`
- `for`
- `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
  ```

- int: An integer literal can be a decimal, hex, octal or binary number.

  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)
  0X02A   : Hex
  07234   : Octal (any number that starts with zero and has digits 0-7)
  0111b   : Binary (any number ending with 'b' and having digits 0-1)
  ```

- 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. String literals are implicitly terminated by a null character (like in C).

  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 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. We have two pointers in our language, a pointer to a character (of type `charp`), and a pointer to an integer (of type `intp`).

  Technically, a pointer variable stores the memory address of the value (or the address of the variable) that it points to. An integer pointer can only point to integer 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, 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`. It can be used where an `intp` or `charp` is valid.

Examples:

```
charp x;     /* x is a pointer to a character variable */
int x, z;
intp y;

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;`
`,`	Used in variables lists	`int x, y, z;`
`|`	For integer expression: Absolute value of i	`|i|`
`|`	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 of two forms:

First form: /* comments */

Second form (ends with newline characters): // comment

Correct (legal):

```
/* this is my block
        comment */

// my line comment
```

Incorrect (illegal):

```
/* this is my block comment //
```

Programs

A program is composed of many functions, just listed one after another. Every legal program should have one and only one function: 'void 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. The return value of a function may be assigned to a variable of the appropriate type. Each non void function should return value of appropriate type. The void function are not allowed to return values, but may use "return;" statement.

Correct (legal):

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

void main() {
    int a;
    a = foo();
}
```

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

```
void main() {
    int a;
    a = foo();
}
```

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

Functions

Functions are declared as:

```
type function_id "(" parameter_list ")" "{" body_of_function "}"
```

function_id is the name of the function. Read below for more details. *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 `boolean`, `char`, `int`, `charp`, or `intp`. *type* is the type of the return value and must be either `boolean`, `char`, `int`, `charp`, or `intp`. *body_of_function* contains function declarations, variable declarations, and statements.

You may declare one or more functions inside the body of a function, thus, nested functions are possible with this language. The last statement in a non void function must be a return statement. Void function may have empty body.

Correct (legal):

```
int foo(int i, int j, int k) {
    intp i;
    boolean fee(int l, int i, char n) {
        return true;
    }
    return 0;
}
```

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

Correct (legal):

```
int foo() { return 0; }
void foo_2() {  }
char f234() { return '0';}
```

Incorrect (illegal):

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

Function Body

The *body_of_function* can contain nested function declarations, variable declarations, and statements. Declaration must come before statements.

Correct (legal):

```
int foo(int i, char j, string k)
{
    int k;            // variable declarations
    int square(int t) // function declarations
    {
        int temp;
        temp = t* t;
        return temp;
    }

    int total;              /* variable declarations */

    k=5;                    /* statements */
    total = i+square(k);
    return total-10;
}
```

Variable Declarations

Variables are declared in the following syntax:

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

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

Correct (legal):

```
int i;
boolean m, n;
char c;
string s1[10], s2[20];
```

Strings (character arrays)

Arrays are declared with the following syntax:

```
"string" ID1"[" INTEGER_LITERAL "]" "," ID2"[" INTEGER_LITERAL "]" ","
ID3"[" 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):

```
string a[1000], b[1000];
char c;
int i;
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 1000 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):

```
string a[10], b[10];
char c;
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, different **loop** statements, or a **code block**.

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.

Examples:

```
   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.

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):

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

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

Incorrect (illegal):

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

The syntax for if, if/else, do-while, 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;

"do"  "{"
       body_of_nested_statement
"}" "while" "("  expression  ")" ";"

"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 syntax for the return statement is:

```
return expression;  //non void functions
return;             //void functions
```

The type of the expression of the return statement must match the return type declared for the function.

Correct (legal):

```
int foo() { return 0; }
int foo_2() { int a; a = 2; return a; }
int foo_3(){ return foo(); }    //correct if "foo" is of "int" type
int foo_4() { if (true) { return 4; } return 10; }
```

Incorrect (illegal):

```
int foo_5() { return true;  }
int foo_6() { if (true) return 5; }
```

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, characters, boolean, 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
0x012
true
-5
^x
^(p+5)
!false
```

Incorrect (illegal):

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

Function Call

Correct (legal):

```
a = foo(i, j); /* 'a' has been declared already */
goo("hello");
```

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; }
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 `charp`.

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 `int`. Also, you cannot multiply a pointer with a value or a variable. When you add the result of an expression to a `charp` (or subtract an expression from a `charp`), the resulting type is still a `charp`.

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 `intp` 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):

```
charp z;    /* x is a pointer to a char variable */
int x;
intp y;
x = 5;
y = &x;
x = 6;

charp x;
string y[10];
char z;
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):

```
booleanp x;         /* no such pointer type */
x = &(1+3)

char x;
intp y;
y = &x;             /* address of x is of type charp */

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