Mini-C Language Specification

01-26-2017

Mini-C language specification

Mini-C is a very simple C-like programming language designed for education purposes (for software that can be used by students to learn about program compilation).

It has if statements, loops (while), variables, arithmetic (+, -, *, /, %), comparison (==, !=, <, >, <=, >=), logical (!, &&, ||) operators, string concatenation.

It does not have “main” function for entry point like in C or Java, all code on “top-level” (similarly to Python, JavaScript) is executed as if it was inside of C/Java main.

Currently it does not support user-defined functions.

Data types

- int (signed 32-bit)
- double (double-precision 64-bit IEEE 754 floating point)
- bool
- string

int, bool and double work the same as in Java.

Built-in statements/expressions for input/output

- Print a string (to stdout)

```
void print(string s)
void println(string s)
```

- Read a number (from stdin)

```
int readInt()
double readDouble()
```

- Read a string with all characters until a line separator (from stdin)

```
string readLine()
```

- Convert a number or boolean to a string

```
string toString(int input)
string toString(double input)
string toString(bool input)
```

Scopes

Nested scopes are supported for variables and work the same as in most of other languages like C, Java. Blocks (curly braces, { … }) can be nested and variables defined in a scope are available only until the end of the scope. Redeclaration of variables existing in parent (or current) scopes is not allowed.

Example:

```
int a = 42;

if (a == 42) {
    int b = a + 1;
    print(toString(b)); // 43
}

{
    int b = a + 2;
    print(toString(b)); // 44
}

{
    int a = 41; // error
    print(toString(b + 1)); // error
}

int a = 40; // error
```

Code example

```
println("Hello world!");

print("Enter name: ");
string name = readLine();

print("Enter age: ");
int age = readInt();

if (age < 10) {
    println("Sorry, you are not old enough to learn about compilers");
    exit();
}

println("Hello " + name);

int n = 10;
int sum = 0;
int i = 1;
while (i <= n) {
    sum = sum + i;
    i = i + 1;
}
println("Sum of the first " + toString(n) +
        " natural numbers: " + toString(sum));

double pi = 3.141592;
int r = 5;
double area = pi * (r * r);
println("Area of a circle with radius " + toString(r) + ": " +
        toString(area));

int desiredCount = 20;
println("First " + toString(desiredCount) + " prime numbers:");
int num = 2;
int count = 0;
while (count < desiredCount) {
    bool isPrime = true;
    int j = 1;
    while (j < num / 2) {
        if (j != 1 && num % j == 0) {
            isPrime = false;
            break;
        } else
            j = j + 1;
    }
    if (isPrime) {
        print(toString(num));
        if (count < desiredCount - 1)
            print(toString(", "));
        count = count + 1;
    }
    num = num + 1;
}
```

Grammar

Grammar description using EBNF.

- `'x'` – terminal symbol.
- `x?` – zero or one occurrences of x.
- `x*` – zero or more occurrences of x.
- `x+` – one or more occurrences of x.
- `x | y` – alternative (x or y).
- `()` – group, for example `(x | y) z (x y)?`

```
program = statement*

statement = block
          | SEMI
          | assignment
          | declaration
          | if
          | while
          | 'break' SEMI
          | 'continue' SEMI
          | 'exit' '(' ')' SEMI
          | 'print' parExpression SEMI
          | 'println' parExpression SEMI

block = '{' statement* '}'

expression = literal
           | ID
           | ('!' | '-') expression
           | expression ('*' | '/' | '%') expression
           | expression ('+' | '-') expression
           | expression ('<' | '>' | '<=' | '>=') expression
           | expression ('==' | '!=') expression
           | expression ('&&') expression
           | expression ('||') expression
           | parExpression
           | 'readInt' '(' ')'
           | 'readDouble' '(' ')'
           | 'readLine' '(' ')'
           | 'toString' parExpression

parExpression = '(' expression ')'

assignment = ID assignmentOp expression SEMI

declaration = type ID (assignmentOp expression)? SEMI

if = 'if' parExpression statement ('else' statement)?

while = 'while' parExpression statement

assignmentOp = '='
```

```
type = 'int'
     | 'double'
     | 'bool'
     | 'string'

literal = IntegerLiteral
        | FloatingPointLiteral
        | StringLiteral
        | BooleanLiteral

IntegerLiteral = DIGIT+
FloatingPointLiteral = DIGIT+ '.' DIGIT+
StringLiteral = '"' (CHAR | '\"')* '"'
BooleanLiteral = 'true' | 'false'

SEMI = ';'

ID = (LETTER | '_') (LETTER | DIGIT | '_')*

DIGIT = '0' | ... | '9'
LETTER = 'a' | ... | 'z' | 'A' | ... | 'Z'

CHAR = <unicode character, as in Java>
```

Whitespace characters (`' '`, `'\t'`, `'\r'`, `'\n'`) are skipped outside of tokens.

Grammar inference examples

(This section is included as a way of showing that the grammar is correct, also it may be useful for other students.)

```
string s = readLine();
print(s);

program => statement, statement => declaration, statement =>
=> type, ID, assignmentOp, expression, ';', statement =>
=> 'string', 's', '=', 'readLine', '(', ')', ';', statement =>
=> 'string', 's', '=', 'readLine', '(', ')', ';', 'print', parExpression, ';' =>
=> 'string', 's', '=', 'readLine', '(', ')', ';', 'print', '(', expression, ')', ';'
=> 'string', 's', '=', 'readLine', '(', ')', ';', 'print', '(', ID, ')', ';' =>
=> 'string', 's', '=', 'readLine', '(', ')', ';', 'print', '(', 's', ')', ';'

cnt = cnt + 1;

program => statement =>  assignment => ID, assignmentOp, expression, ';' =>
=> 'cnt', '=', expression, ';' => 'cnt', '=', expression, '+', expression, ';' =>
=> 'cnt', '=', ID, '+', literal, ';' => 'cnt', '=', 'cnt', '+', IntegerLiteral, ';'
=> 'cnt', '=', 'cnt', '+', '1', ';'

while (true) {
    if (1 == 1)
        break;
}

program => statement => while => 'while', parExpression, statement =>
=> 'while', '(', expression, ')', statement =>
=> 'while', '(', literal, ')', statement =>
=> 'while', '(', BooleanLiteral, ')', statement =>
=> 'while', '(', 'true', ')', statement =>
=> 'while', '(', 'true', ')', block =>
=> 'while', '(', 'true', ')', '{', statement, '}' =>
=> 'while', '(', 'true', ')', '{', if, '}' =>
=> 'while', '(', 'true', ')', '{', 'if', parExpression, statement, '}' =>
=> 'while', '(', 'true', ')', '{', 'if', '(', expression, ')', statement, '}' =>
=> 'while', '(', 'true', ')', '{', 'if', '(', expression, '==', expression, ')',
statement, '}' =>
=> 'while', '(', 'true', ')', '{', 'if', '(', literal, '==', literal, ')', statement,
'}' =>
=> 'while', '(', 'true', ')', '{', 'if', '(', IntegerLiteral, '==', IntegerLiteral,
')', statement, '}' =>
=> 'while', '(', 'true', ')', '{', 'if', '(', '1', '==', '1', ')', statement, '}' =>
=> 'while', '(', 'true', ')', '{', 'if', '(', '1', '==', '1', ')', 'break', ';', '}'

bool b = -a > 0;

program => statement => declaration => type, ID, assignmentOp, expression, ';' =>
=> 'bool', 'b', '=', expression, ';' =>
=> 'bool', 'b', '=', expression, '>', expression, ';' =>
=> 'bool', 'b', '=', '-', expression, '>', expression, ';' =>
=> 'bool', 'b', '=', '-', ID, '>', expression, ';' =>
=> 'bool', 'b', '=', '-', 'a', '>', expression, ';' =>
=> 'bool', 'b', '=', '-', 'a', '>', literal, ';' =>
=> 'bool', 'b', '=', '-', 'a', '>', IntegerLiteral, ';' =>
=> 'bool', 'b', '=', '-', 'a', '>', '0', ';'
```

```
if (!b || a > 10.0)
    exit;
else {
    int c;
}

program => statement => if => 'if', parExpression, statement, 'else', statement =>
=> 'if', '(', expression, ')', statement, 'else', statement =>
=> 'if', '(', expression, '||', expression, ')', statement, 'else', statement =>
=> 'if', '(', '!', expression, '||', expression, ')', statement, 'else', statement =>
=> 'if', '(', '!', ID, '||', expression, ')', statement, 'else', statement =>
=> 'if', '(', '!', 'b', '||', expression, '>', expression, ')', statement, 'else',
statement =>
=> 'if', '(', '!', 'b', '||', ID, '>', expression, ')', statement, 'else', statement
=> 'if', '(', '!', 'b', '||', 'a', '>', expression, ')', statement, 'else', statement
=> 'if', '(', '!', 'b', '||', 'a', '>', literal, ')', statement, 'else', statement =>
=> 'if', '(', '!', 'b', '||', 'a', '>', FloatingPointLiteral, ')', statement, 'else',
statement =>
=> 'if', '(', '!', 'b', '||', 'a', '>', '10.0', ')', statement, 'else', statement =>
=> 'if', '(', '!', 'b', '||', 'a', '>', '10.0', ')', 'exit', '(', ')', ';', 'else',
statement
=> 'if', '(', '!', 'b', '||', 'a', '>', '10.0', ')', 'exit', '(', ')', ';', 'else',
block =>
=> 'if', '(', '!', 'b', '||', 'a', '>', '10.0', ')', 'exit', '(', ')', ';', 'else',
'{', statement, '}' =>
=> 'if', '(', '!', 'b', '||', 'a', '>', '10.0', ')', 'exit', '(', ')', ';', 'else',
'{', declaration, '}' =>
=> 'if', '(', '!', 'b', '||', 'a', '>', '10.0', ')', 'exit', '(', ')', ';', 'else',
'{', type, assignmentOp, ID, ';', '}' =>
=> 'if', '(', '!', 'b', '||', 'a', '>', '10.0', ')', 'exit', '(', ')', ';', 'else',
'{', 'int', '=', 'c', ';', '}'
```