Cminus: A Language for Practice Implementation

1 Purpose

This document describes the Cminus programming language. It is intended to provide enough detail to allow implementation of a parser (context-free analyzer). This document contains information that relevant to the some portions of each assignment in this course.

2 Introduction

Cminus is a programming language designed for practice implementation. Cminus is an extremely simplified version of C in which one may perform simple integer calculations. Cminus is intended to be simple enough to implement in a single semester by any student willing to put in some effort. Each feature included in the language was added specifically to illustrate some problem that arises in the design and implementation of a very simple compiler.

Cminus supports three basic data types: *integer*, `char` and *float*. Each type may be aggregated into a one dimensional array. The language is intended to be *strongly typed*; that is, the type of each expression should be determinable at compile time. Since there is no *boolean* data type, integers are used as logical values.

Control structures in Cminus are limited. It has an *if* statement, a *while* statement and a *compound* statement. In addition, there is support for procedure calls.

3 Lexical Properties of Cminus

In this section, any sequence of characters denoted inside of a pair of `"`'s indicates the actual string literal found inside of the `"`'s.

1. In Cminus, blanks are significant.
2. In Cminus, all keywords are reserved; that is, the programmer cannot use a Cminus keyword as the name of a variable. The valid keywords are:

   ⟨CHAR⟩ → `"char"`
   ⟨ELSE⟩ → `"else"`
   ⟨EXIT⟩ → `"exit"`
   ⟨FLOAT⟩ → `"float"`
   ⟨IF⟩ → `"if"`
   ⟨INT⟩ → `"int"`
   ⟨READ⟩ → `"read"`
   ⟨RETURN⟩ → `"return"`
   ⟨VOID⟩ → `"void"`
   ⟨WHILE⟩ → `"while"`
   ⟨WRITE⟩ → `"write"`

   (Note that Cminus is *case sensitive,* that is, the variable `X` differs from `x`. Thus, `if` is a keyword, but `IF` can be a variable name.)

3. The following special characters have meanings in a Cminus program.

⟨AND⟩ → `"&&"`
⟨ASSIGN⟩ → `"="`
⟨CM⟩ → `","`
⟨DIVIDE⟩ → `"/"`
⟨DQ⟩ → `"\""`
⟨EQ⟩ → `"=="`
⟨GE⟩ → `">="`
⟨GT⟩ → `">"`
⟨LBR⟩ → `"{"`
⟨LBK⟩ → `"["`
⟨LE⟩ → `"<="`
⟨LP⟩ → `"("`
⟨LT⟩ → `"<"`
⟨MINUS⟩ → `"-"`
⟨NE⟩ → `"!="`
⟨NOT⟩ → `"!"`
⟨OR⟩ → `"||"`
⟨PLUS⟩ → `"+"`
⟨RBR⟩ → `"}"`
⟨RBK⟩ → `"]"`
⟨RP⟩ → `")"`
⟨SC⟩ → `";"`
⟨SQ⟩ → `"'"`
⟨TIMES⟩ → `"*"`

4. Comments are delimited by the characters `/*` and `*/`. A `/*` begins a comment; it is valid in no other context. A `*/` ends a comment; it cannot appear inside a comment. Comments may appear before or after any other token.

5. Identifiers are written with upper and lowercase letters and are defined as follows:

⟨LETTER⟩ → `"a"` | `"b"` | `"c"` | ··· | `"z"` | `"A"` | `"B"` | ··· | `"Z"`
⟨DIGIT⟩ → `"0"` | `"1"` | `"2"` | ··· | `"9"`
⟨ID⟩ → ⟨LETTER⟩ (⟨LETTER⟩ | ⟨DIGIT⟩)*

The implementor may restrict the length of identifiers so long as identifiers of at least 31 characters are legal.

6. Constants are defined as follows:

⟨POSITIVE⟩ → `"1"` | `"2"` | `"3"` | ... | `"9"`
⟨INTCON⟩ → ⟨POSITIVE⟩ ⟨DIGIT⟩* | "`0`"
⟨FLOATCON⟩ → ⟨INTCON⟩ ⟨DOT⟩ (⟨DIGIT⟩)*
⟨CHARCON⟩ → ⟨SQ⟩ ¬(⟨SQ⟩) ⟨SQ⟩

Special string constants are acceptable in `write` statements:

⟨STRING⟩ → ⟨DQ⟩ (¬(⟨DQ⟩))* ⟨DQ⟩

4 Cminus Syntax

This section gives a syntactic description of Cminus. The sections following the grammar provide implementation notes on the various parts of the grammar.

The grammar, as stated, defines the language. It may require some massaging before implementation with any particular parser generator system.

4.1 BNF

The following grammar describes the context-free syntax of Cminus:

⟨program⟩	→	(⟨declaration⟩)$^+$
⟨declaration⟩	→	⟨type⟩ ⟨ID⟩ ⟨LP⟩ (⟨paramDeclList⟩)? ⟨RP⟩ ⟨LBR⟩ (⟨varDecl⟩)*(⟨statement⟩)* ⟨RBR⟩
	|	⟨varDecl⟩
⟨paramDeclList⟩	→	⟨paramDecl⟩ (⟨CM⟩ ⟨paramDecl⟩)*
⟨paramDecl⟩	→	⟨type⟩ ⟨identifier⟩
⟨varDecl⟩	→	⟨type⟩ ⟨identifier⟩ (⟨CM⟩ ⟨identifier⟩)* ⟨SC⟩
⟨identifier⟩	→	⟨ID⟩ (⟨LBK⟩ ⟨INTCON⟩ ⟨RBK⟩)?
⟨type⟩	→	⟨INT⟩
	|	⟨CHAR⟩
	|	⟨FLOAT⟩
⟨statement⟩	→	⟨assignment⟩
	|	⟨callStatement⟩
	|	⟨ifStatement⟩
	|	⟨whileStatement⟩
	|	⟨ioStatement⟩
	|	⟨returnStatement⟩
	|	⟨exitStatement⟩
	|	⟨cpdStatement⟩
⟨assignment⟩	→	⟨variable⟩ ⟨ASSIGN⟩ ⟨expr⟩ ⟨SC⟩
⟨ifStatement⟩	→	⟨IF⟩ ⟨LP⟩ ⟨Expr⟩ ⟨RP⟩ ⟨cpdStatement⟩ (⟨ELSE⟩ ⟨cpdStatement⟩)?
⟨callStatement⟩	→	⟨ID⟩ ⟨LP⟩ ⟨argList⟩ ⟨RP⟩ ⟨SC⟩
⟨whileStatement⟩	→	⟨WHILE⟩ ⟨LP⟩ ⟨expr⟩ ⟨RP⟩ ⟨Statement⟩
⟨ioStatement⟩	→	⟨READ⟩ ⟨LP⟩ ⟨variable⟩ ⟨RP⟩ ⟨SC⟩
	|	⟨WRITE⟩ ⟨LP⟩ (⟨expr⟩ | ⟨STRING⟩) ⟨RP⟩ ⟨SC⟩
⟨returnStatement⟩	→	⟨RETURN⟩ ⟨expr⟩ ⟨SC⟩
⟨exitStatement⟩	→	⟨EXIT⟩ ⟨SC⟩
⟨cpdStatement⟩	→	⟨LBR⟩ (⟨statement⟩)* ⟨RBR⟩

⟨expr⟩ → ⟨simpleExpr⟩
| ⟨expr⟩ ⟨OR⟩ ⟨simpleExpr⟩
| ⟨expr⟩ ⟨AND⟩ ⟨simpleExpr⟩

⟨simpleExpr⟩ → ⟨addExpr⟩
| ⟨simpleExpr⟩ ⟨EQ⟩ ⟨addExpr⟩
| ⟨simpleExpr⟩ ⟨NE⟩ ⟨addExpr⟩
| ⟨simpleExpr⟩ ⟨LE⟩ ⟨addExpr⟩
| ⟨simpleExpr⟩ ⟨LT⟩ ⟨addExpr⟩
| ⟨simpleExpr⟩ ⟨GE⟩ ⟨addExpr⟩
| ⟨simpleExpr⟩ ⟨GT⟩ ⟨addExpr⟩

⟨addExpr⟩ → ⟨mulExpr⟩
| ⟨addExpr⟩ ⟨PLUS⟩ ⟨mulExpr⟩
| ⟨addExpr⟩ ⟨MINUS⟩ ⟨mulExpr⟩

⟨mulExpr⟩ → ⟨factor⟩
| ⟨mulExpr⟩ ⟨TIMES⟩ ⟨factor⟩
| ⟨mulExpr⟩ ⟨DIVIDE⟩ ⟨factor⟩

⟨factor⟩ → ⟨variable⟩
| ⟨constant⟩
| ⟨ID⟩ ⟨LP⟩ (⟨argList⟩)? ⟨RP⟩
| ⟨NOT⟩ ⟨expr⟩
| ⟨LP⟩ ⟨expr⟩ ⟨RP⟩

⟨variable⟩ → ⟨ID⟩
| ⟨ID⟩ ⟨LBK⟩ ⟨expr⟩ ⟨RBK⟩

⟨constant⟩ → ⟨INTCON⟩
| ⟨CHARCON⟩
| ⟨FLOATCON⟩

⟨argList⟩ → ⟨expr⟩ (⟨CM⟩ ⟨expr⟩)*

4.2 Section Notes

4.2.1 Declarations

Cminus has four standard types: `int`, `char`, `void` and `float`. Integers, characters and floats occupy a single machine "word". Void types are only for procedures that do not return a value. The standard types may be composed into a structured array type. An identifier may represent one of six types of objects:

1. an integer variable
2. an integer array
3. a character variable
4. a character array
5. a floating point variable
6. a floating point array

Identifiers are declared to be variables or arrays in a variable declaration. Only singly dimensioned arrays are permitted in Cminus. Indexing begins at 0 as in C and Java.

Example:

```
int x,y;
```

```
int a[15];
float vector[100];
```

4.2.2 Function Declarations

The semantics of function definition are simple. A function returns the value of the expression specified in the first `return` statement that it executes.

Example:

```
int max (int a, int b) {
   if (a < b) {
      return b;
   }
   else {
      return a;
   }
}
```

4.2.3 Procedure Declarations

A procedure is declared as a function with a `void` return type. No return statement may appear in a procedure.

Example:

```
void swap (int a[2]) {
   int temp;

   temp = a[0];
   a[0] = a[1];
   a[1] = temp;
}
```

4.2.4 Assignment Statement

The assignment statement requires that the *left hand side* (the ⟨variable⟩ non-terminal) and *right hand side* (the ⟨expr⟩ non-terminal) evaluate to have the same type. If they have different types, either coercion is required or a context-sensitive error has occurred. The coercion rules for assignment are simple. If both sides are numeric (of type `int` or `float`), the right hand side is converted to the type of the left hand side. If either side is is of type `char`, both sides must be `char` (or else the procedure contains a context-sensitive error).

4.2.5 If Statement

The grammar for the `if-else` construct is written to eliminate the dangling else ambiguity. This is done by forcing a ⟨cpdStatement⟩ in each of the then- and else-clauses. To evaluate an if statement, the expression is evaluated. For an integer value, Cminus defines `0` as *false*; any other value is equivalent to *true*.

Examples:

```
if (c == d) { d = a; }
if (b == 0) { b = 2*a; } else { b = b/2; }
```

4.2.6 While Statement

The while statement provides a simple mechanism for iteration. Cminus's while statement behaves like the while statement in many other languages; it executes the statement in the loop's body until the controlling expression becomes false. The controlling expression will be treated as a boolean value encoded into an `int` expression.

4.2.7 Input-Output Statements

Cminus provides two primitives for input and output. The `read` and `write` statements are intended to provide direct access to primitives implemented in the target abstract machine.

Examples:

```
read (x);
write (x+y);
write ('error');
```

4.2.8 Return Statements

Cminus allows functions to return values. The type of the return value needs to be the same type as the type of the function, or it must be converted to that type.

Example:

```
int f() {
    return 7;
}
```

4.2.9 Exit Statement

An exit statement in Cminus exits the program completely.

4.2.10 Expressions

Cminus expressions compute simple values of type `int`, `char` or `float` For both integer and floating point numbers, arithmetic and comparison are defined.

Coercion: If an expression contains operands of only one type, evaluation is straight forward. When an operand contains mixed types, the situation is more complex. If an *Addop* or *Mulop* has an `int` operand and a `float` operand, the `int` operand should be converted to a `float` before the operation is performed. A context-sensitive error occurs if a value of type `char` appears in an arithmetic expression.

Relational operators always produce an integer. Comparisons between integers and floats produce integer results. To perform the comparison, the integer is converted to a float. For the numbers, comparison is based

on both sign and magnitude. Comparisons between characters and numbers make no sense; they are illegal. For characters, comparison is based on location in the standard ASCII collating sequence. Any comparison between unlike types constitutes a context-sensitive error.

Operator Precedence: Operator precedences in Cminus are specified in the table below. Multiplication and division have the highest priority, `&&` and `||` have the lowest.

Operator	Precedence
`*, /`	5
`+, -`	4
`<, <=, =, >=, >, !=`	3
`!`	2
`&&, ||`	1

Booleans: Because Cminus has no booleans, relational expressions are defined to yield integer results. Thus, a relational expression of the form `a == b` is considered to be an arithmetic expression whose value is `0` if the relation does not hold and not `0` otherwise. Hence, both the `if-else` and `while` statements test integer values; the expression is considered *false* if it evaluates to `0` and to *true* if it evaluates to anything else. Consider the following example which tests for either of two conditions being true:

```
{
    read (a); read (b); read (c); read (d);
    if ((a == b) + (c < d)) { write ('error'); }
}
```

Note that relational expressions should be enclosed in parentheses because they have very low precedence. In the above example, the special operator `||` could have been used. In Cminus the operator `||` takes two integer operands. `||` produces the result `0` if both operands evaluate to `0`; otherwise, it produces `1`. The operator `&&` evaluates to `1` if both operands are nonzero; otherwise it evaluates to `0`. The unary logical operator `!` evaluates to `1` if its argument is zero and to `0` otherwise.

Function/Procedure Invocation: Cminus uses parentheses to indicate invocation and square brackets to indicate subscripting of an array. This simplifies the grammar — many languages use parentheses for both purposes.Parameters are passed call-by-value. Note the the value of an array is the address of the array, just as in C.

5 An Example Program

The following program represents a simple example program written in Cminus. This program successively reads pairs of integers from the input file and prints out their product.

```
int main() {
   int x, y;
      read(x); read(y);
      while ((x != 0) || (y != 0)) {
         write (x*y);
         read (x); read (y);
```

```
        }
        exit;
}
```