University Simple C (USC) Language Reference

CS 504: Compiler Design
Stony Brook University

Contents

Motivation

There are many different languages that have been created for academic settings, and even languages that have been created for an undergraduate compilers curriculum. The most popular compiler-focused language is arguably Classroom Object Oriented Language (COOL), which was developed by Dr. Alex Aiken at Stanford. However, there were three main reasons I chose to develop something different:

1. I wanted a language that was a valid subset of an actual programming language, and in particular, the C programming language. This makes it interesting because students will have the opportunity to work with a language they are familiar with. It also means that output can be compared against production-quality compilers.
2. I wanted a language that was amenable enough to recursive descent parsing such that it wouldn't be unreasonable to ask a pair of undergraduate students to write a portion of the parser within three weeks. I chose recursive descent because the most popular C/C++ compilers (including both GCC and Clang) utilize recursive descent for almost all their parsing. This allows for very specific error messages, which are usually not present in academic compilers. Of course, a parser for the language described herein could also (rather trivially) be created using an LALR parser-generator such as Bison.
3. I wanted a language which did not have too many high-level constructs, as that would make it more manageable to construct the LLVM IR. If the language were object-oriented, it would require more effort to generate the appropriate runtime data structures within LLVM IR.

While University Simple C (abbreviated as USC) is a fairly narrow subset of Standard C, it ultimately is possible to create non-trivial programs within these restrictions. For example, an implementation of an in-place quicksort that utilizes recursion is very much possible (as shown at the end of this document).

Types and Variables

One of the most significant simplifications in USC is how it handles types and variables. The only variable types supported are `char` and `int`. It is also possible to create statically-allocated arrays of characters and integers. Type specifiers such as `static` or `volatile` are not supported. Pointers are not explicitly supported, but they are implicitly supported in that arrays are passed to functions by address. No user-defined types (`typedef`, `struct`, or `union`) are supported.

With only `char` and `int` base types, type conversions are fairly straightforward. All expressions are aggressively converted to 32-bit integer operations, and if they ultimately must be stored in a `char`, the value is truncated. This is fairly consistent with what occurs in C, though it may not technically be fully standard-compliant. There are no conversions allowed between arrays and base types.

Variable declarations follow the C89 restriction in that they must appear at the beginning of a lexical scope. Scoping follows standard Algol-style rules. Each function has its own lexical scope, and any compound statements within that function are given child scopes. This means standard variable ghosting/redeclaration rules apply. One restriction is that there are no global variables, though it admittedly would not add too much complexity to the language if globals were added.

To simplify the syntax of the declarations, USC does not allow chaining of multiple variable declarations separated by commas. Each variable declaration must appear in a separate statement. For array declarations, they must either have a fixed size specified, or have an assignment at declaration. The later can only be done with strings, since initializer lists are not supported.

Functions

All functions must be declared and implemented at the same time in source – there is no support for forward declaration. This also means that it is not possible to write code where there is a circular dependency between two functions.

Functions must return either `void`, `int`, or `char`. As for function parameters, `int`, `char`, and arrays are supported. As in Standard C, arrays are passed by address while the base types are passed by value. User-defined functions do not support variable arguments.

There must be an entry function named "main" that returns an integer. No parameters are allowed for the main function.

All functions that have a non-void return type must conclude their body with a return statement. This is because parsing and semantic analysis is completed in a single pass, so performing reachability analysis on return statements would add unnecessary overhead.

Operators and Expressions

There is a reasonable complement of mathematical and logical operators available in USC, so most of the commonly-used binary operators are available. Array subscript operators are also supported. Not very many unary operators are supported, however, other than logical not, pre-increment, pre-decrement, and "address of" in the specific case of taking the address of a subscript element of an array.

There are some restrictions on the operators – for example, pre-increment/decrement must be performed directly on an identifier, so code such as `--(x)` is not allowed. Most other operators can be performed on expressions.

For logical operators, Standard C rules are followed where any non-zero value is considered "true." However, one restriction is that any such operators on a string/array are not allowed. Logical `||` and `&&` are short-circuited as they are in Standard C.

One important note is that assignment in USC is *not* treated as an expression. This greatly simplifies expression evaluation, since expressions such as `(x = 5) * 5` are not possible.

Comments

Single line `//` comments are supported. Multiline comments are not supported.

Statements

There are only a few types of statements supported in USC, but they still allow for fairly complex logic. Compound statements (multiple statements surrounded by braces) are allowed, as are assignment statements, return statements, expression statements, and null statements.

In terms of control flow, `if` (with or without `else`) is supported, but the only type of loop that's supported is `while`. This isn't really a problem since all `for` and `do`/`while` loops can easily be changed to `while` loops. Jumps in flow such as `break`, `continue`, or `goto` are also not allowed. This makes all control flow in USC reducible. Finally, there is no support for `switch` statements, but these can be emulated with a series of chained `else if`s.

Memory

As there are no explicit pointers types, there is no dynamic memory allocation support in USC. This means that all variables are local to the function they are declared in. Of course, arrays are implicitly passed by address. This means that it is possible to create arrays with a single element to pass it by address.

Preprocessor/Multiple Source Files

There is no preprocessor whatsoever, and all code must be contained in a single source file. Though certainly, it would be possible to use an external preprocessor tool to preprocess the source file. However, in practice I'd suspect very few header files from a Standard C program would work within the confines of this subset.

Standard Library

In order to allow for basic output, USC allows for calls to the `printf` function from the Standard C Library. This is the only Standard Library function that is currently supported. The compiler will automatically add a declaration of `printf` to the IR if it detects use of the function, as there would otherwise be no way to forward declare the function (since USC does not support variable arguments). No validation of the parameters to `printf` is performed.

Grammar

This section contains the full context-free grammar for USC. Note that the grammar as written is most definitely not LL(1) or LR(1). There is left recursion, which would be problematic for LL(1), and there are common left prefixes, which would cause shift-reduce conflicts in strict LR(1). Of course, the grammar can be transformed to be more amenable to LL(1) or LR(1).

Terminals are shown in `monospace`. The following terminals are defined by these regular expressions (following Flex syntax):

```
id = [a-zA-Z_][a-zA-Z0-9_]*
constant = ("\'"("\\t"|"\\n"|.)"\'")|("-"?(0|([1-9][0-9]*)))
string = \"([^\\\"]|\\n|\\t)*\"
```

All other terminals are the exact text as written.

```
      Program → FunctionList
 FunctionList → FunctionList Function
              | Function
     Function → ReturnType id ( ArgumentDecl ) CompoundStmt
   ReturnType → void
              | int
              | char
 ArgumentDecl → ArgDeclList
              | ε
  ArgDeclList → ArgDeclList , ArgDecl
              | ArgDecl
      ArgDecl → VarType id ArgDeclArray
 ArgDeclArray → [ ]
              | ε
      VarType → int
              | char
 CompoundStmt → { Declarations Statements }
 Declarations → DeclList
              | ε
     DeclList → DeclList Decl
              | Decl
         Decl → VarType id DeclArray DeclAssign ;
    DeclArray → [ constant ]
              | [ ]
              | ε
   DeclAssign → = Expr
              | ε
   Statements → StmtList
              | ε
     StmtList → StmtList Stmt
              | Stmt
         Stmt → CompoundStmt
              | AssignStmt
              | IfStmt
              | WhileStmt
              | ReturnStmt
              | Expr ;
              | ;
   AssignStmt → id = Expr ;
              | id [ Expr ] = Expr ;
       IfStmt → if ( Expr ) Stmt
              | if ( Expr ) Stmt else Stmt
```

WhileStmt → `while` (*Expr*) *Stmt*
ReturnStmt → `return ;`
| `return` *Expr* `;`
Expr → *Expr* `||` *AndTerm*
| *AndTerm*
AndTerm → *AndTerm* `&&` *RelExpr*
| *RelExpr*
RelExpr → *RelExpr* `==` *NumExpr*
| *RelExpr* `!=` *NumExpr*
| *RelExpr* `<` *NumExpr*
| *RelExpr* `>` *NumExpr*
| *NumExpr*
NumExpr → *NumExpr* `+` *Term*
| *NumExpr* `-` *Term*
| *Term*
Term → *Term* * *Value*
| *Term* / *Value*
| *Term* % *Value*
| *Value*
Value → ! *Factor*
| *Factor*
Factor → (*Expr*)
| `constant`
| `string`
| `id`
| `id` [*Expr*]
| `id` (*FuncCallArgs*)
| ++ `id`
| `-- id`
| & `id` [*Expr*]
FuncCallArgs → *ArgList*
| *ε*
ArgList → *ArgList* `,` *Expr*
| *Expr*

Code Listing: Quicksort in USC

Although USC is a relatively narrow subset of C, it is still possible to write non-trivial programs, such as the following quicksort implementation:

```
// quicksort.usc
// Implements in-place quicksort algorithm
// Expected result:
// abcdeeefghhijklmnoooopqrrsttuuvwxyz

int partition(char array[], int left, int right, int pivotIdx)
{
	char pivotVal = array[pivotIdx];
	int storeIdx = left;
	int i = left;
	char temp;

	// Move pivot to end
	temp = array[pivotIdx];
	array[pivotIdx] = array[right];
	array[right] = temp;

	while (i < right)
	{
		if (array[i] < pivotVal)
		{
			// Swap array[i] and array[storeIdx]
			temp = array[i];
			array[i] = array[storeIdx];
			array[storeIdx] = temp;
			++storeIdx;
		}

		++i;
	}

	// Swap array[storeIdx] and array[right]
	temp = array[storeIdx];
	array[storeIdx] = array[right];
	array[right] = temp;

	return storeIdx;
}
```

```
void quicksort(char array[], int left, int right)
{
      int pivotIdx;

      if (left < right)
      {
            // Pick the middle point
            pivotIdx = left + (right - left) / 2;

            pivotIdx = partition(array, left, right, pivotIdx);
            quicksort(array, left, pivotIdx - 1);
            quicksort(array, pivotIdx + 1, right);
      }
}

int main()
{
      char letters[] = "thequickbrownfoxjumpsoverthelazydog";
      quicksort(letters, 0, 34);

      printf("%s\n", letters);

      return 0;
}
```