University Simple C Compiler (USCC)

CS 502: Compiling and Programming Systems
Purdue University

Contents

Thanks to Dr. Changhee Jung for talking through some ideas, being a second set of eyes, as well as creating the Makefiles.

Introduction

USCC is the reference compiler for the University Simple C language, which is a subset of Standard C. Further information on the language itself is outlined in the USC Language Reference document.

The target output language for USCC, by default, is the LLVM bitcode IR. At the moment, the code only compiles and runs on Mac OS X 10.10.x or higher. It has been tested against the LLVM 3.5.0 release.

Usage

By default, USCC will emit LLVM bitcode to a ".bc" file. So for example:

```
$ uscc quicksort.usc
```

Would output a file called quicksort.bc, which contains LLVM bitcode.

The generic syntax for USCC is:

```
uscc [OPTIONS] <input>
```

Note that the compiler only accepts a single input file.

And the following command-line options currently supported:

`-a` `--print-ast`	Output the AST to stdout once the syntax and semantic analysis is complete. By default, this will prevent the compiler from generating LLVM IR. However, if LLVM IR or generation is still desired, -b can also be specified.
`-b` `--bitcode`	This flag is only used if you want to emit the LLVM bitcode in addition to some other output (such as the AST)
`-h` `--help`	Prints usage instructions.
`-o OUTFILE` `--output OUTFILE`	Specify the target output file. The default output file is the input with its last extension replaced with either ".bc" or ".s". Note that if -b and -s are specified simultaneously, -o is ignored.
`-p` `--print-bc`	This outputs a human-readable version of the bitcode to stdout.
`-O`	Enables optimization passes. Can be combined with the other flags (most useful combined with -p)

Errors

Syntax and semantic errors in source are reported by USCC in the following format:

```
filename:line:col: error: message
diagnostic message
```

Time was invested in providing good diagnostics for error messages. When a parse or semantic error is detected, a caret is placed at the exact point of error.

For example, the following source file:

```
int main()
{
	{
		return ((1));
		{
			return "Hello world!";
			return (x);
			return;
		}
	}
}
```

Yields the following error messages:

```
test003.usc:6:11: error: Expected type int in return statement
                  return "Hello world!";
                         ^
test003.usc:7:12: error: Use of undeclared identifier 'x'
                  return (x);
                          ^
test003.usc:8:10: error: Invalid empty return in non-void function
                  return;
                        ^
test003.usc:11:1: error: USC requires non-void functions to end with a return
}
^
4 Error(s)
```

Other errors such as a failure to specify an input file will yield messages with this format:

```
uscc: error: message
```

Building USCC

On Mac OS X, the easiest way to build USCC is via the uscc.xcodeproj. It will build provided the following directory hierarchy:

(any directory)
- `lib` (sym link to llvm/Debug+Asserts/lib)
- `bin` (sym link to llvm/Debug+Asserts/bin)
- `llvm`[1] (sym link to the root of llvm source directory)
- `uscc`

Alternatively, USCC can be built via standard makefiles contributed by **Dr. Changhee Jung**. These makefiles will work not only on Mac OS X, but on other UNIX flavors such as Linux and Solaris. The easiest way to build with the makefiles is as follows:

- `cd uscc` (move to the `uscc` package directory above)
- `make` (Build everything)

The makefiles also support the following build rules:
- `make clean` (delete all objects, libraries, and the executable)
- `make depend` (generate dependencies for uscc source files)

Since USCC leverages C++11 features, it must be built with a compiler that supports `-std=c++11` or `-std=gnu++11`. This and other compiler/linker flags can be modified in `uscc/Makefile.variables`.

USCC can also be built in Windows using Cygwin. However, the process is a bit more complex and full instructions are provided in a separate "Building USCC on Windows" document.

External Code

Nearly all of the USCC source code represents original work. However, there are a couple of exceptions, and these are noted below.

Lexical Analysis

The scanner is not written from scratch, and instead uses Flex. The Flex input file is scan/usc.l. During the build, a FlexLexer.cpp file is generated as the output of Flex in C++ mode.

Command Line Options Parsing

To parse command line options, the external "ez Option Parser" library was used. The code for the command line parser is in uscc/ezOptionParser.hpp.

---

[1] LLVM should be built in the same directory where the source exists, i.e., the configure script should be execute in the root of llvm source directory. For example,

```
cd llvm-3.5.0.src; CC=clang CXX=clang++ ./configure --disable-optimized;
```

Code Overview

Here is an overview of which files correspond to the different parts of the compiler. Generally, the comments within the files are verbose, so the source files can be consulted for further information.

One thing to note is that the code is most definitely written in a C++11 dialect. So there are plenty uses of STL, shared pointers, auto, range-based for loops, and so on.

This section is broken down by subdirectory within the main uscc directory.

uscc

main.cpp – Main entry point and driver for the compiler as a whole.

ezOptionParser.hpp – External command line parsing library.

scan

Tokens.h/cpp/def – Defines all the valid tokens for the language, with some additional information used mostly to help with error messages.

usc.l – Input file for Flex. The rules in here are pretty straightforward.

FlexLexer.h/cpp – Output generated by Flex in C++ mode.

parse

Parse.h – Declares the `Parser` class that drives the scanner, the recursive descent parsing, and the semantic analysis. There are quite a few helper functions declared here, as well, including those that make it easy to match specific tokens (or sequences of tokens) and for generating error messages.

This file also declares all of the mutually-recursive functions used by the recursive descent parser. Generally, each function corresponds to a specific grammar rule, with some exceptions. The end result of constructing a `Parser` class is an AST intermediate representation.

Parse.cpp – Implements all of the `Parser` helper functions, as well as a few of the top-level recursive descent functions.

ParseExpr/ParseStmt.cpp – These files implement the vast majority of the recursive descent functions.

ParseExcept.h/cpp – Define and implement the exceptions that are utilized by the `Parser` class. Exceptions seemed like a great way to catch and recover from syntax errors during the recursive descent.

Types.h – Defines the type `enum` used internally by the `Parser` and the AST nodes.

ASTNodes.h – Defines all of the AST Nodes that are used by the AST. Since most of the code for the AST construction is fairly simple, a lot of the functions are implemented inline.

ASTNodes/ASTExpr/ASTStmt.cpp – Implements some functions for the nodes where they were complex enough to not warrant placing inline.

ASTPrint.cpp – Implements the `printNode` member function for every node in the AST, which is used by the -a command line option.

ASTEmit.cpp – This is where most of the code for the LLVM IR generation is contained. There is one function for every AST node, and it constructs all of the basic blocks and instructions within the hybrid CFG uses by LLVM.

Symbols.h/cpp – These files define and implement the `Identifier`, `SymbolTable`, and `StringTable` classes. In the .cpp, there is some code related to generating the LLVM IR. When generating the code for a function, all of the variables declared within the scope of the function (regardless of nesting) have their storage allocated via the `emitIR` function defined inside `ScopeTable`. There is similarly some code for generating global string constants in `StringTable`.

Emitter.h/.cpp – These files facilitate the other main actions of USCC. It kicks off the LLVM IR construction from the root AST node. The functions for printing, writing to a file, and verifying the LLVM IR is also contained in here.

opt

SSABuilder.h/cpp – These files implement Static Single Assignment, and is implemented using the algorithm outlined in "Simple and Efficient Construction of SSA Form" (Braun et. al.)

Passes.h/Passes.cpp and ConstantOps.cpp/ConstantBranch.cpp/DeadBlocks.cpp/LICM.cpp – These define the optimization passes. At the moment, there is constant propagation, constant branch folding, dead block removal, and loop invariant code motion.

tests

This directory contains the multitude of test cases. To execute the test suites, simply run: `python [testsuite]`

The three test suites are:

testParse.py – These test basic AST generation without semantic analysis. However, since there is no way to disable semantic analysis, the vast majority of these test cases will fail in the reference compiler. These test cases are more useful when the parser is actively being developed during an assignment for students.

testSemant.py – These extensively test the semantic analysis. Many of these tests look at expected errors, though some also look at expected AST generation from code that passes semantic analysis.

testEmit.py – These test the emitted LLVM bitcode by leveraging the lli tool. Each USC file ran by this test suite has its output compared against the expected.

testOpt.py – Like testEmit.py, except it runs with optimizations enabled.

Sample Output

Here is the output that is generated by USCC for quicksort.usc (the source for quicksort.usc is already listed at the end of the USC Language Reference).

AST

Command Line:

```
$ uscc -a quicksort.usc
```

Program Output:

```
---Function: int partition
------ArgDecl: char[] array
------ArgDecl: int left
------ArgDecl: int right
------ArgDecl: int pivotIdx
------CompoundStmt:
---------Decl: char pivotVal
------------ArrayExpr:
---------------ArraySub: array
-----------------IdentExpr: pivotIdx
---------Decl: int storeIdx
------------IdentExpr: left
---------Decl: int i
------------IdentExpr: left
---------Decl: char temp
---------AssignStmt: temp
------------ArrayExpr:
---------------ArraySub: array
-----------------IdentExpr: pivotIdx
---------AssignArrayStmt:
------------ArraySub: array
---------------IdentExpr: pivotIdx
------------ArrayExpr:
---------------ArraySub: array
-----------------IdentExpr: right
---------AssignArrayStmt:
------------ArraySub: array
---------------IdentExpr: right
------------IdentExpr: temp
---------WhileStmt
------------BinaryCmp <:
---------------IdentExpr: i
---------------IdentExpr: right
------------CompoundStmt:
---------------IfStmt:
-----------------BinaryCmp <:
--------------------ToIntExpr:
------------------------ArrayExpr:
--------------------------ArraySub: array
----------------------------IdentExpr: i
--------------------ToIntExpr:
------------------------IdentExpr: pivotVal
-----------------CompoundStmt:
```

```
--------------------AssignStmt: temp
-----------------------ArrayExpr:
--------------------------ArraySub: array
------------------------------IdentExpr: i
--------------------AssignArrayStmt:
-----------------------ArraySub: array
--------------------------IdentExpr: i
-----------------------ArrayExpr:
--------------------------ArraySub: array
------------------------------IdentExpr: storeIdx
--------------------AssignArrayStmt:
-----------------------ArraySub: array
--------------------------IdentExpr: storeIdx
-----------------------IdentExpr: temp
--------------------ExprStmt
-----------------------IncExpr: storeIdx
---------------ExprStmt
-----------------IncExpr: i
---------AssignStmt: temp
------------ArrayExpr:
---------------ArraySub: array
-----------------IdentExpr: storeIdx
---------AssignArrayStmt:
------------ArraySub: array
---------------IdentExpr: storeIdx
------------ArrayExpr:
---------------ArraySub: array
-----------------IdentExpr: right
---------AssignArrayStmt:
------------ArraySub: array
---------------IdentExpr: right
------------IdentExpr: temp
---------ReturnStmt:
------------IdentExpr: storeIdx
---Function: void quicksort
------ArgDecl: char[] array
------ArgDecl: int left
------ArgDecl: int right
------CompoundStmt:
---------Decl: int pivotIdx
---------IfStmt:
------------BinaryCmp <:
---------------IdentExpr: left
---------------IdentExpr: right
------------CompoundStmt:
---------------AssignStmt: pivotIdx
-----------------BinaryMath +:
--------------------IdentExpr: left
--------------------BinaryMath /:
-----------------------BinaryMath -:
--------------------------IdentExpr: right
--------------------------IdentExpr: left
-----------------------ConstantExpr: 2
---------------AssignStmt: pivotIdx
-----------------FuncExpr: partition
```

```
--------------------IdentExpr: array
--------------------IdentExpr: left
--------------------IdentExpr: right
--------------------IdentExpr: pivotIdx
--------------ExprStmt
------------------FuncExpr: quicksort
--------------------IdentExpr: array
--------------------IdentExpr: left
--------------------BinaryMath -:
------------------------IdentExpr: pivotIdx
------------------------ConstantExpr: 1
--------------ExprStmt
------------------FuncExpr: quicksort
--------------------IdentExpr: array
--------------------BinaryMath +:
------------------------IdentExpr: pivotIdx
------------------------ConstantExpr: 1
--------------------IdentExpr: right
---------ReturnStmt: (empty)
---Function: int main
------CompoundStmt:
---------Decl: char[36] letters
------------StringExpr: thequickbrownfoxjumpsoverthelazydog
---------ExprStmt
------------FuncExpr: quicksort
---------------IdentExpr: letters
---------------ConstantExpr: 0
---------------ConstantExpr: 34
---------ExprStmt
------------FuncExpr: printf
---------------StringExpr: %s

---------------IdentExpr: letters
---------ReturnStmt:
------------ConstantExpr: 0
```

LLVM Bitcode

Command Line:

```
$ uscc -p quicksort.usc
```

Program Output:

```
; ModuleID = 'main'

@.str = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@.str1 = private unnamed_addr constant [36 x i8]
c"thequickbrownfoxjumpsoverthelazydog\00", align 1

declare i32 @printf(i8*, ...)

define i32 @partition(i8* %array, i32 %left, i32 %right, i32 %pivotIdx) {
entry:
  %temp = alloca i8, align 1
  %i = alloca i32, align 4
  %storeIdx = alloca i32, align 4
  %pivotVal = alloca i8, align 1
  %pivotIdx.addr = alloca i32, align 4
  store i32 %pivotIdx, i32* %pivotIdx.addr
  %array.addr = alloca i8*, align 8
  store i8* %array, i8** %array.addr
  %right.addr = alloca i32, align 4
  store i32 %right, i32* %right.addr
  %left.addr = alloca i32, align 4
  store i32 %left, i32* %left.addr
  %0 = load i32* %pivotIdx.addr
  %1 = load i8** %array.addr
  %2 = getelementptr inbounds i8* %1, i32 %0
  %3 = load i8* %2
  store i8 %3, i8* %pivotVal
  %4 = load i32* %left.addr
  store i32 %4, i32* %storeIdx
  %5 = load i32* %left.addr
  store i32 %5, i32* %i
  %6 = load i32* %pivotIdx.addr
  %7 = load i8** %array.addr
  %8 = getelementptr inbounds i8* %7, i32 %6
  %9 = load i8* %8
  store i8 %9, i8* %temp
  %10 = load i32* %right.addr
  %11 = load i8** %array.addr
  %12 = getelementptr inbounds i8* %11, i32 %10
  %13 = load i8* %12
  %14 = load i32* %pivotIdx.addr
  %15 = load i8** %array.addr
  %16 = getelementptr inbounds i8* %15, i32 %14
  store i8 %13, i8* %16
  %17 = load i8* %temp
  %18 = load i32* %right.addr
  %19 = load i8** %array.addr
  %20 = getelementptr inbounds i8* %19, i32 %18
```

```
  store i8 %17, i8* %20
  br label %while.cond

while.cond:                                       ; preds = %if.end, %entry
  %21 = load i32* %i
  %22 = load i32* %right.addr
  %cmp = icmp slt i32 %21, %22
  %23 = zext i1 %cmp to i32
  %tobool = icmp ne i32 %23, 0
  br i1 %tobool, label %while.body, label %while.end

while.body:                                       ; preds = %while.cond
  %24 = load i32* %i
  %25 = load i8** %array.addr
  %26 = getelementptr inbounds i8* %25, i32 %24
  %27 = load i8* %26
  %conv = sext i8 %27 to i32
  %28 = load i8* %pivotVal
  %conv1 = sext i8 %28 to i32
  %cmp2 = icmp slt i32 %conv, %conv1
  %29 = zext i1 %cmp2 to i32
  %tobool3 = icmp ne i32 %29, 0
  br i1 %tobool3, label %if.then, label %if.end

while.end:                                        ; preds = %while.cond
  %30 = load i32* %storeIdx
  %31 = load i8** %array.addr
  %32 = getelementptr inbounds i8* %31, i32 %30
  %33 = load i8* %32
  store i8 %33, i8* %temp
  %34 = load i32* %right.addr
  %35 = load i8** %array.addr
  %36 = getelementptr inbounds i8* %35, i32 %34
  %37 = load i8* %36
  %38 = load i32* %storeIdx
  %39 = load i8** %array.addr
  %40 = getelementptr inbounds i8* %39, i32 %38
  store i8 %37, i8* %40
  %41 = load i8* %temp
  %42 = load i32* %right.addr
  %43 = load i8** %array.addr
  %44 = getelementptr inbounds i8* %43, i32 %42
  store i8 %41, i8* %44
  %45 = load i32* %storeIdx
  ret i32 %45

if.then:                                          ; preds = %while.body
  %46 = load i32* %i
  %47 = load i8** %array.addr
  %48 = getelementptr inbounds i8* %47, i32 %46
  %49 = load i8* %48
  store i8 %49, i8* %temp
  %50 = load i32* %storeIdx
  %51 = load i8** %array.addr
  %52 = getelementptr inbounds i8* %51, i32 %50
```

```
  %53 = load i8* %52
  %54 = load i32* %i
  %55 = load i8** %array.addr
  %56 = getelementptr inbounds i8* %55, i32 %54
  store i8 %53, i8* %56
  %57 = load i8* %temp
  %58 = load i32* %storeIdx
  %59 = load i8** %array.addr
  %60 = getelementptr inbounds i8* %59, i32 %58
  store i8 %57, i8* %60
  %61 = load i32* %storeIdx
  %inc = add i32 %61, 1
  store i32 %inc, i32* %storeIdx
  br label %if.end

if.end:                                           ; preds = %if.then, %while.body
  %62 = load i32* %i
  %inc4 = add i32 %62, 1
  store i32 %inc4, i32* %i
  br label %while.cond
}

define void @quicksort(i8* %array, i32 %left, i32 %right) {
entry:
  %pivotIdx = alloca i32, align 4
  %right.addr = alloca i32, align 4
  store i32 %right, i32* %right.addr
  %left.addr = alloca i32, align 4
  store i32 %left, i32* %left.addr
  %array.addr = alloca i8*, align 8
  store i8* %array, i8** %array.addr
  %0 = load i32* %left.addr
  %1 = load i32* %right.addr
  %cmp = icmp slt i32 %0, %1
  %2 = zext i1 %cmp to i32
  %tobool = icmp ne i32 %2, 0
  br i1 %tobool, label %if.then, label %if.end

if.then:                                          ; preds = %entry
  %3 = load i32* %left.addr
  %4 = load i32* %right.addr
  %5 = load i32* %left.addr
  %sub = sub i32 %4, %5
  %div = sdiv i32 %sub, 2
  %add = add i32 %3, %div
  store i32 %add, i32* %pivotIdx
  %6 = load i8** %array.addr
  %7 = getelementptr inbounds i8* %6, i32 0
  %8 = load i32* %left.addr
  %9 = load i32* %right.addr
  %10 = load i32* %pivotIdx
  %call = call i32 @partition(i8* %7, i32 %8, i32 %9, i32 %10)
  store i32 %call, i32* %pivotIdx
  %11 = load i8** %array.addr
  %12 = getelementptr inbounds i8* %11, i32 0
```

```
  %13 = load i32* %left.addr
  %14 = load i32* %pivotIdx
  %sub1 = sub i32 %14, 1
  call void @quicksort(i8* %12, i32 %13, i32 %sub1)
  %15 = load i8** %array.addr
  %16 = getelementptr inbounds i8* %15, i32 0
  %17 = load i32* %pivotIdx
  %add2 = add i32 %17, 1
  %18 = load i32* %right.addr
  call void @quicksort(i8* %16, i32 %add2, i32 %18)
  br label %if.end

if.end:                                           ; preds = %if.then, %entry
  ret void
}

define i32 @main() {
entry:
  %letters = alloca [36 x i8], align 1
  %0 = bitcast [36 x i8]* %letters to i8*
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %0, i8* getelementptr inbounds ([36 x i8]*
@.str1, i32 0, i32 0), i64 36, i32 1, i1 false)
  %1 = getelementptr inbounds [36 x i8]* %letters, i32 0, i32 0
  call void @quicksort(i8* %1, i32 0, i32 34)
  %2 = getelementptr inbounds [36 x i8]* %letters, i32 0, i32 0
  %3 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.str, i32
0, i32 0), i8* %2)
  ret i32 0
}

; Function Attrs: nounwind
declare void @llvm.memcpy.p0i8.p0i8.i64(i8* nocapture, i8* nocapture readonly, i64,
i32, i1) #0

attributes #0 = { nounwind }
```