IR generation using LLVM

Akshay Khole - 1100226

Alan Nilsson - 1069854

COEN 259 - Compilers

Winter 2015

Preface

In computer science, a compiler is a tool that helps run programs written in a high level language on a computer hardware. While doing so, the program is processed through a number of “passes” by the compiler. Some of these phases involve intermediate code generation, optimization, byte code and machine code generation. Every language needs a compiler that will help run the programs written in those languages on the hardware. Every language needs a compiler. Most compilers have components that are common and can be reused. LLVM is a tool that is used to create compilers using reusable components. LLVM focuses on modularity and code optimization. Hence it has gained a lot of popularity in the recent years. Through this project, we aim to create something of educational value for future students trying to learn LLVM.

Acknowledgement

We would like to thank Dr. Ming-Hwa Wang for his guidance, constant supervision and for providing us with the necessary teaching required for completing the project. We would like to thank the authors of the research papers mentioned in the bibliography that helped us with our research.

Table of Contents

List of tables and figures

1. Abstract

The LLVM IR is quickly becoming more and more important and pervasive in the field of computer science and compiler technologies. Many research institutions and commercial interests are seeing the advantages of the LLVM compiler framework. Given this interest, it becomes more important that students learn this technology. However, there are few examples of using the LLVM frameworks for learning purposes, especially in the context that compiler courses are often taught. Most introductory compilers courses introduce, and use, the two open source tools Flex and Bison for lexical analysis and parser generation respectively. Examples of using these two tools with the LLVM IR generation framework would be a useful study aid in compiler design and LLVM usage.

2. Introduction

Objective

Our objective is to provide a simple and easy to understand example of using Flex, Bison, and the LLVM frameworks to produce LLVM IR for a simplistic calculator language.

The problem

The LLVM (formerly the Low Level Virtual Machine) is an extremely powerful compiler infrastructure framework designed for compile-time, link-time, and run-time optimizations of programs written in your favorite programming language. LLVM works on several different platforms, and its primary claim to fame is generating code that runs fast. LLVM is built around the Intermediate Representation of code.

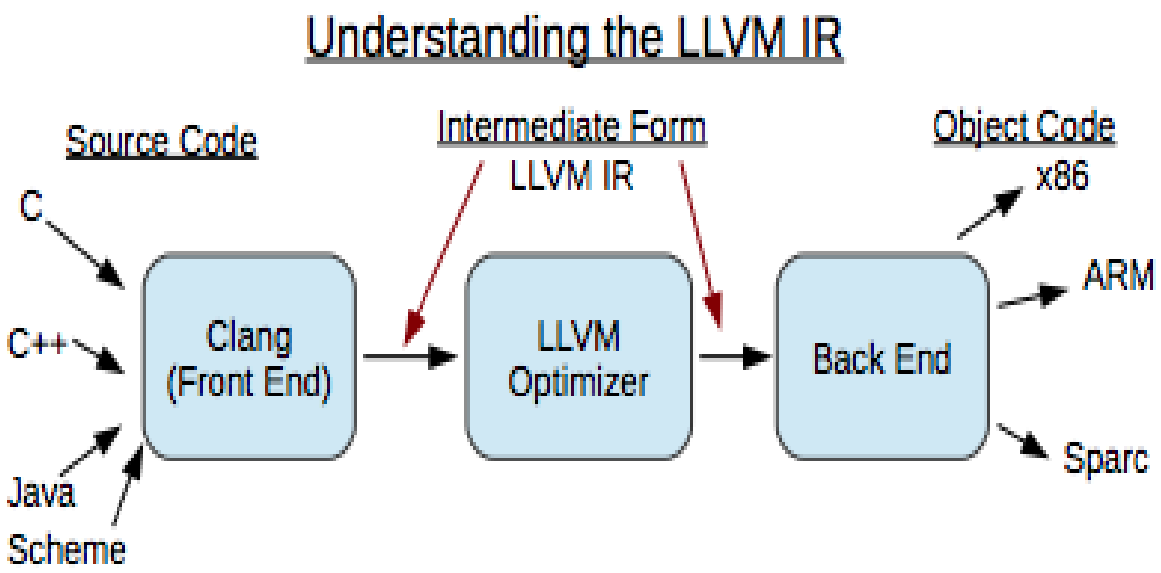


Figure 1

The LLVM infrastructure is bundled with the LLVM core libraries. These libraries include the optimizers that operate on the source code as well as the IR. The optimization happens at multiple stages of the code generation process. The optimization capabilities of LLVM is one of the strongest reasons why LLVM has gained popularity.

A research titled “A LLVM Based Compiler for COFFEE” quotes:
“*Comparing with traditional GCC compiler, LLVM is a compiler framework that is parallel programming friendly. It has built-in support for OpenCL [1] that is a programming model designed for parallel programs. LLVM front end converts the OpenCL program into LLVM IR [2], an intermediate presentation that is intended to be used as input for the LLVM compiler back end. The back end is then responsible for translating the LLVM IR into target specific language which is usable by the parallel computing capable hardware, for example GPU. LLVM framework is anticipated to become a preferable choice for compiler developers who are targeting for the parallel computing capable hardware due to its good parallel programming support.*”

This emphasizes the importance of learning LLVM. This research is also aimed at teaching Compiler engineers and people already involved in compiler development. Hence, for students taking a graduate level compiler class, our project aims to help them get started with LLVM.

Since LLVM is quickly becoming the compiler generation infrastructure of choice for many compiler engineers, it has become essential to include LLVM in advanced compiler classes along with traditional tools like Flex and Bison. Through this project, we wish to create documentation and know-how on how to integrate LLVM with Flex and Bison for building frontends.

Why this project is related to this class

We believe it is self-evident how this subject is related to a graduate level course in compiler design. However, to be explicit, this is relevant because it can be used as a study aid and an example for students attempting to learn compiler construction and more importantly, studying to use the LLVM compiler frameworks.

Why other approach is no good

The papers read to date present what the LLVM frameworks are and how they have been used in certain projects. The papers are all high level with very little practical instruction on using the LLVM frameworks. This lack of practical usage examples combined with the use of flex and bison hinders understanding of LLVM and compilers in general.

Why we think our approach is better

We believe that our approach to providing a concrete example, even if for an academic language, can benefit many who are trying to learn compiler technology or the LLVM frameworks. When LLVM is used along with Flex and Bison, tools students already acquainted with, it makes it more interesting as well as provides real-world examples to study LLVM.

Statement of the problem

The increasing importance of the LLVM compiler frameworks requires that this technology start to be taught or introduced in compiler courses. A practical example of using LLVM in conjunction with techniques usually taught will aid in learning LLVM.

Area or scope of investigation

Our scope will include using the advanced calculator language and grammar as introduced in Flex & Bison [1] and producing LLVM IP using the LLVM code generation frameworks.

3. Theoretical bases and literature review

Definition of the problem

As stated above, compiler courses are usually taught in the context of using Flex and Bison for lexical analysis and parser generating respectively. There is a lack of practical examples of using those tools in conjunction with the LLVM frameworks to produce LLVM IR code.

Theoretical background of the problem

There has been much research in the field of pedagogy that strongly indicates providing examples of problem greatly enhances student learning. Time and again educational research indicates that linking new material to familiar concepts enhances understanding and retention in learners [6][7]. In this case using Flex and Bison is the familiar and by linking the LLVM IR generation to these tools it is hoped that the content is better understood and retained.

To better understand LLVM, you have to learn LLVM IR and its idiosyncrasies. This process akin to learning yet another programming language.

```
#include <stdio.h>
int main( )
{
  printf("Hello World!\n");
}
```

Consider the Hello world program written in C++ above. This code can be compiled using llvm-gcc in the following way:

```
Tintin.local# llvm-gcc helloworld.cpp -S -emit-llvm
```

This generates a **.ll** file that contains the IR. The IR generated for the hello world program is as follows:

```
@.str = private constant [13 x i8] c"Hello World!\00", align 1 ;

define i32 @main() ssp {
entry:
  %retval = alloca i32
  %0 = alloca i32
  %"alloca point" = bitcast i32 0 to i32
  %1 = call i32 @puts(i8* getelementptr inbounds ([13 x i8]* @.str, i64 0, i64 0))
  store i32 0, i32* %0, align 4
  %2 = load i32* %0, align 4
  store i32 %2, i32* %retval, align 4
  br label %return
return:
  %retval1 = load i32* %retval
  ret i32 %retval1
}

declare i32 @puts(i8*)
```

Understanding the LLVM IR:

1. Comments in LLVM assembly begin with a semicolon (;) and continue to the end of the line.
2. Global identifiers begin with the at (@) character. All function names and global variables must begin with @, as well
3. Local identifiers in the LLVM begin with a percent symbol (%). The typical regular expression for identifiers is [%@][a-zA-Z$._][a-zA-Z$._0-9]*.
4. You declare a vector or array type as [no. of elements X size of each element]
5. You declare a global string constant for the helloworld string as follows: @hello = constant [13 x i8] c"Hello World!\00"
6. To define a function, begin with the define keyword followed by the return type, and then the function name. Eg. define i32 @main() { ; some LLVM assembly code that returns i32 }
7. To declare a function, begin the declaration with the declare keyword followed by the return type, the function name, and an optional list of arguments to the function. The declaration must be in the global scope. Eg. declare i32 puts(i8*)
8. To call the function, call <function return type> <function name> <optional function arguments>
9. Each function ends with a return statement. There are two forms of return statement: ret <type> <value> or ret void

Related research to solve the problem

LLVM research is mainly focused on creating backends for different hardwares. One research focuses on how the LLVM core library API can be used to generate backend microMIPS by leveraging already written modules for the MIPS architecture. A snippet that does this is shown below:

```
// Get MIPS32 binary enconding
uint32_t Binary = getBinaryCodeForInstr(TmpInst,
                                        Fixups);
unsigned Opcode = TmpInst.getOpcode();
if (IsMicroMips) {
```

```
  // Get microMIPS instruction opcode
  int NewOpcode = Mips::Std2MicroMips (Opcode,
                                       Mips::Arch_micromips);
  if (NewOpcode != 0xFFFF) {
    Opcode = NewOpcode;
    TmpInst.setOpcode (NewOpcode);
   // Get microMIPS binary enconding
    Binary = getBinaryCodeForInstr(TmpInst,
                                   Fixups);
  }
}
```

Another research explains essential concepts that must be understood by compiler developers in order to carry on the implementation of LLVM based compiler and also share experiences that they have gained during the implementation of their own LLVM compiler.

The official LLVM docs have great resources for getting involved with the LLVM community and developing compilers using the LLVM framework.

Advantage / Disadvantage of those research

A clear advantage is that LLVM can be taught from the classroom level itself. Students who are studying and researching in the field of compilers need to understand the LLVM technology. This will not only help them with their career, but also help development and community building.

Tools like Flex and Bison that are popular in the academic circles and taught in classrooms have to to complemented with the teaching of LLVM framework to ensure students are at-par with recent progress in the field of compilers.

Our solution to solve this problem

We use the advanced calculator language to demonstrate how LLVM can be used in conjunction with Flex and Bison. This will keep the scope simple as well as demonstrate real-world usage of highly sophisticated tools like LLVM.

Where our solution is different from others

Our solution is different from those seen to date because it integrates 2 commonly used open source tools directly with the LLVM frameworks. As discussed above, we believe that providing a practical example of using Flex, Bison, and LLVM frameworks together facilitates learning compiler technology and the LLVM frameworks better than simply providing descriptive text and only high level implementation details.

Why our solution is better

Knowledge is the foundation of any great work. To procure knowledge, one must go through endless hours of study and heavy research. Our solution helps ease this process a bit. This also brings a tool heavily used in building modern compilers to the classroom teaching.

4. Hypothesis

Positive / negative hypothesis

This project will help the authors and future readers not only understand LLVM but also have a real hands-on experience on work on an advanced compiler technology that is so much in demand.

Multiple hypothesis

Providing a practical example of classic compiler curriculum in the form of the Flex and Bison tools in conjunction with the LLVM frameworks to produce LLVM IR code would be a valuable learning tool to students learning compiler construction and to the authors of this paper.

5. Methodology

Generating and collecting data

Source for the lexical analysis and the bison grammar will be taken from the book Flex & Bison [1]. Various test case 'source' files for calculator language as described by the bison grammar will be generated to show the operation of the final product. By using the Flex and Bison parts presented in Flex & Bison[1] we do not need to provide descriptions of how that code operates. We leave that to others. We will implement a custom abstract syntax tree (AST) based on requirements of the LLVM code generation framework. The algorithm to produce the AST will be based on semantic actions buried in the bison grammar.

Solving the problem

We will implement a custom algorithm to walk the AST and build the intermediate representation using the LLVM code generation framework.

We will use C++ to match the LLVM frameworks. Tools that will be used for this project include the LLVM framework and tool suite, the Clang compiler, the Flex lexical analyzer, and the Bison parser generator.

Generating output

Output will be generated by the LLVM code generation framework in the form of LLVM IR code. The LLVM tool lli will then be used to run the IR code to produce the final output. This final output will be the numerical computation as specified in the calculator 'source' language.

Testing against hypothesis

We are building a compiler that uses flex and bison along with LLVM modules to create a calculator language. This language should be able to accept numeric input and produce correct output.

6. Implementation

Our implementation is fairly simple. To implement this project, we used Flex to generate a lexer and Bison to generate a parser. We used the LLVM API's for generating IR generator.

Code

The source code is provided in the appendices. The code layout is as follows:

1. calc.l
   The lex file that tokenizes input given from the console. This lexer basically splits up the the input string into useful chunks that are matched using regular expressions.

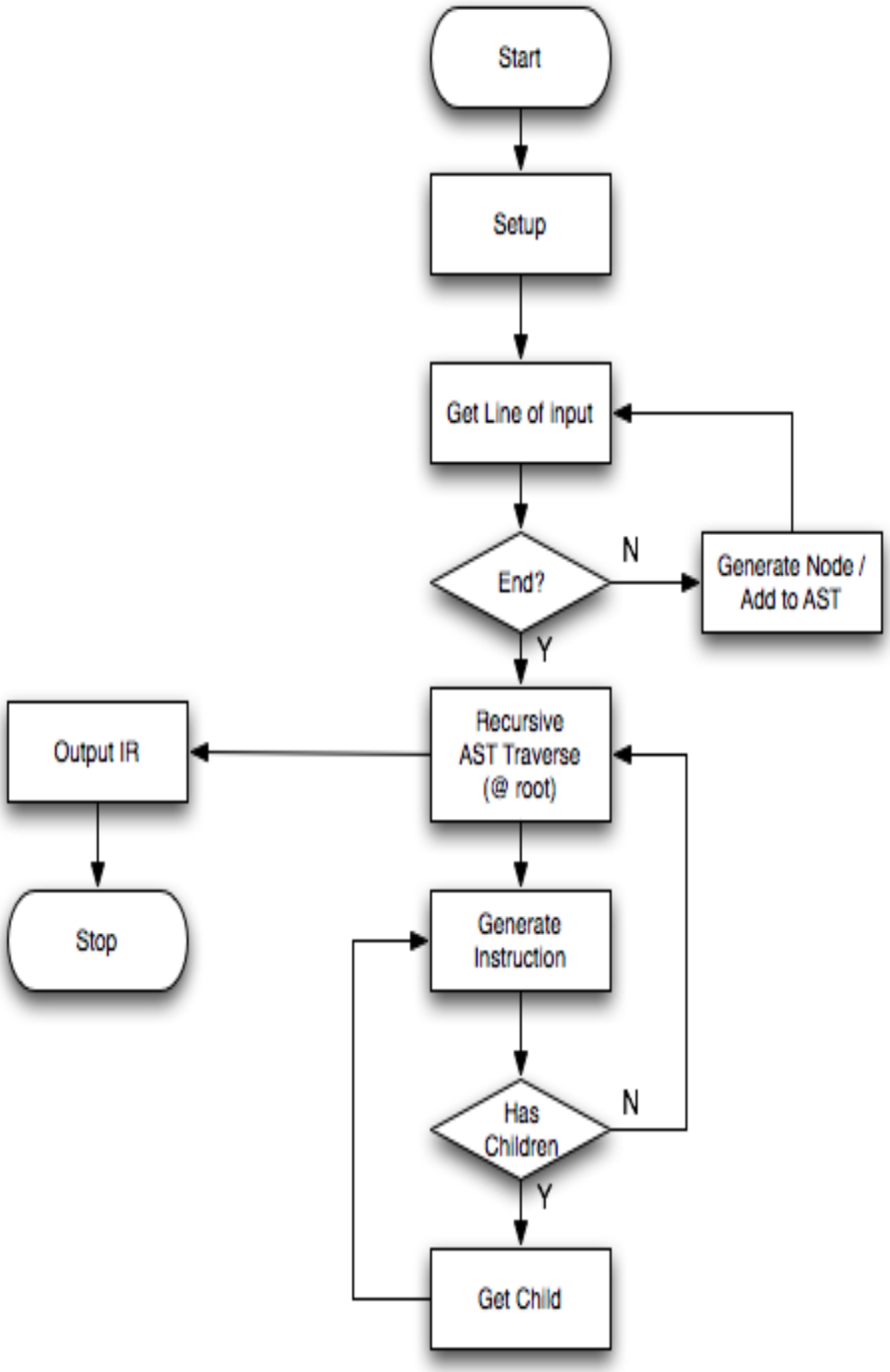
2. calc.y
   This is the bison parser. We define the BNF grammar for our language in this file. The parser receives token from the lexer. It then tries to match the a single token or multiple token with the grammar rules that we specified.

   Once a rule is matched, it executes the action block defined right next to it. In the action blocks, we build an Abstract Syntax Tree (AST).

3. main.cpp
   This is the entry point of the project. This file sets up the environment required for our language. It initializes the llvm API. After the parser makes a run through the input and generates the AST, the AST is traversed by our program to build the IR code using the llvm framework's api calls.

4. fb3-2.cpp
   This file contains some helper methods that help build and traverse the AST.

Design document and flowchart



7. Data analysis and discussion

Output generation

The output can be generated by building the project in Xocde and running it. Upon successful build, a binary is generated. In our case, the binary is “calc”. Example output is as follows:

```
[Sperky:calc/build/Debug] alan%  ./calc
> a=1;
> b=-5;
> c=a*-b;
> d=(c+3)*2;
> e=d>15;
>

------------------------------------------------------------
Dump AST
------------------------------------------------------------
= a const 1.000000
= b - const 5.000000
= c * <a> - <b>
= d * + <c> const 3.000000 const 2.000000
> = e <d> const 15.000000


------------------------------------------------------------
LLVM IR
------------------------------------------------------------
; ModuleID = 'calc'

@0 = private unnamed_addr constant [12 x i8] c"Result: %f\0A\00"

define void @main() {
entrypoint:
  %a = alloca double
  store double 1.000000e+00, double* %a
  %b = alloca double
  store double -5.000000e+00, double* %b
  %c = alloca double
  %0 = load double* %a
  %1 = load double* %b
  %negtmp = fsub double 0.000000e+00, %1
  %multmp = fmul double %0, %negtmp
  store double %multmp, double* %c
  %d = alloca double
  %2 = load double* %c
```

```
  %addtmp = fadd double %2, 3.000000e+00
  %multmp1 = fmul double %addtmp, 2.000000e+00
  store double %multmp1, double* %d
  %e = alloca double
  %3 = load double* %d
  store double %3, double* %e
  %cmptmp = fcmp ugt double %3, 1.500000e+01
  %4 = uitofp i1 %cmptmp to double
  %5 = call i32 (i8*, double, ...)* @printf(i8* getelementptr inbounds ([12 x i8]* @0, i32 0, i32 0), double %4)
  ret void
}

declare i32 @printf(i8*, double, ...)
[Sperky:calc/build/Debug] alan%  lli calc.ll
Result: 1.000000
```

Output analysis

From the output above, we can see how the AST is nodes are defined and how the IR is generated. For example, for the expression:

```
>  a=1;
```

We get the AST node:

```
= a const 1.000000
```

Here we are storing the symbol name, the type of symbol and its value in double precision. Similarly we define representation for other inputs.

Also, in the IR, we see how the symbol ‘a’ is allocated a double type and assigned value:

```
  %a = alloca double
  store double 1.000000e+00, double* %a
```

From the output we understanding what exactly is happening for each line for input.

Compare output against hypothesis

A practical IR generator teaches us how the process of building the IR works. It also teaches us how to use the LLVM framework.

We have generated output for each of the steps the input is going through before running the executable. Demonstration of each of these steps helps us connect compiler theory with practical implementation leading to a much better understanding of the concept.

Abnormal case explanation

An input like:
1 = a

Is what we can call abnormal and will result in syntax error. To maintain simplicity, we support only assignment of identifiers and integer / float values.

8. Conclusions and recommendations

Summary and conclusions

In summary, we were able to create a simple working compiler front end that generates IR for the simple calculator language that does basic math operations.

To conclude, a majority of our time was spent in studying official documentation, research blogs, mailing lists and whitepapers. By writing this report, we hope to make it easier for future students to read this document and understand the process of building an efficient compiler using the LLVM framework.

Recommendations for future studies

We recommend students get a good grasp of the theory from the Dragon book. After the basic concepts are clear, understanding and using LLVM to build your compiler would be simple enough.

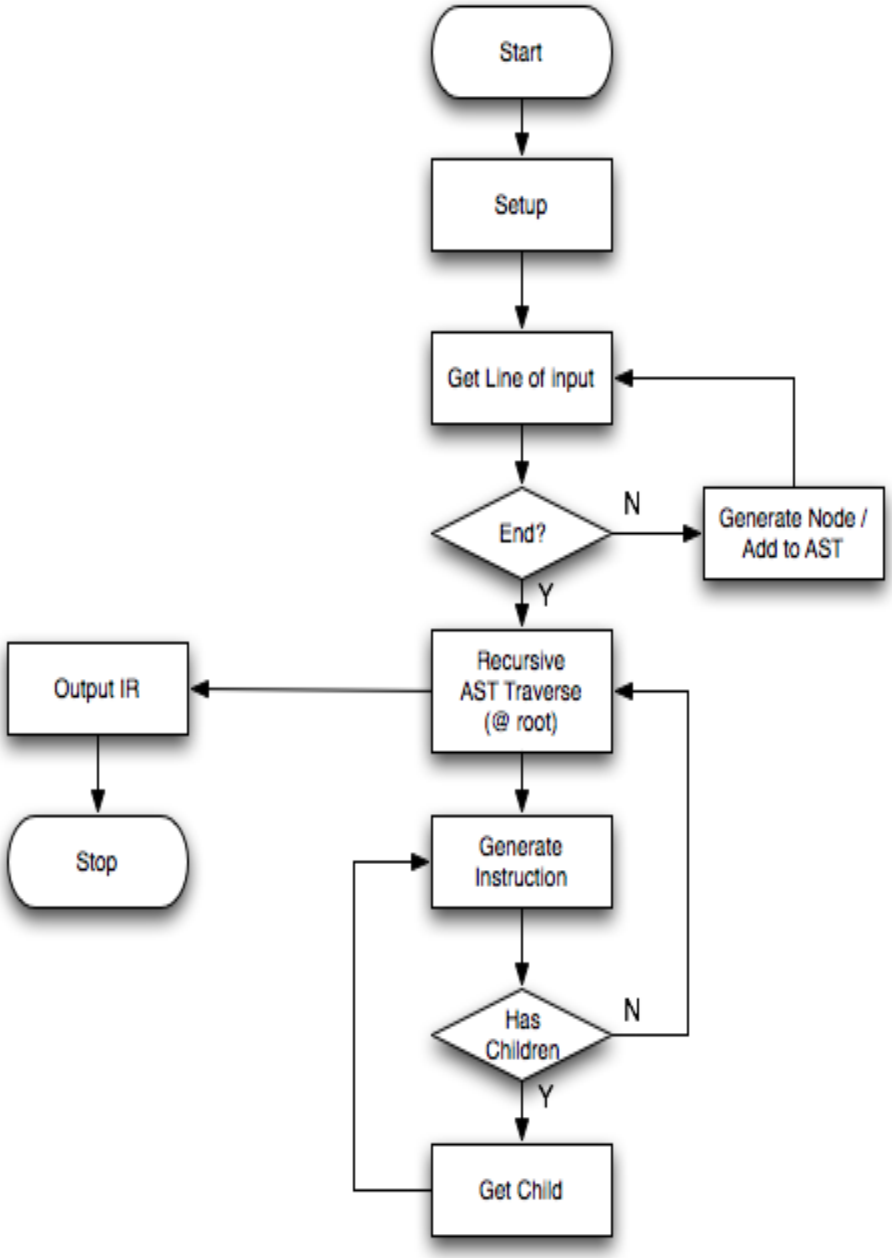
For future studies, we should delve into specifics of generating backend. This requires us to study the methods for backend generation using LLVM as well as computer architecture.

9. Bibliography

1. Flex & Bison, John Levin, © 2009 John Levine, ISBN 978-0-596-15597-1
2. Chris Lattner and Vikram Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, 2004, pg 75.
3. Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke, LLVA: A Low-level Virtual Instruction Set Architecture, Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, 2003, pg 205
4. Jozef Kolek, Zoran Jovanović, Nenad Šljivić, and Dragan Narančić, Adding microMIPS backend to the LLVM compiler infrastructure, Telecommunications Forum (TELFOR), 2013 21st, 2013, pg 1015-1018.
5. Guoqing Zhang, Tapani Ahonen, A LLVM Based Compiler for COFFEE, Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on, 2013, pg 1-5.
6. Case,R. "Gearing the demands of instruction to the development capacities of the learner", Review of educational research, 1975, 45, pgs 59-87.
7. Marek, E.A., Cowan, C.C., and Cavallo, A.M.L., "Students' Misconceptions about Diffusion: How Can They Be Eliminated", The American Biology Teacher, Vol. 56, No. 2 (Feb., 1994), pp. 74-77.
8. "Create a working compiler with the LLVM framework" http://www.ibm.com/developerworks/library/os-createcompilerllvm1/

10. Appendices

Program flowchart



Program source code with documentation

calc.l

```
%option noyywrap yylineno
%{
/*

        Adopted from "flex & bison" by John Levine
        Copyright (c) 2009 John Levine
        ISBN: 978-0-596-15597-1

*/
# include "fb3-2.h"
# include "calc_y.hpp"
%}
EXP ([Ee][-+]?[0-9]+)

%%


"+" |
"-" |
"*" |
"/" |
"=" |
"|" |
"," |
";" |
"(" |
")" { return yytext[0]; }

">"             { yylval.fn = 1; return CMP; }
"<"             { yylval.fn = 2; return CMP; }
"<>"    { yylval.fn = 3; return CMP; }
"=="    { yylval.fn = 4; return CMP; }
">="    { yylval.fn = 5; return CMP; }
"<="    { yylval.fn = 6; return CMP; }

"if"    { return IF; }
"then"  { return THEN; }
```

```
"else"    { return ELSE; }
"while" { return WHILE; }
"do"      { return DO; }
"let"     { return LET;}

"sqrt"    { yylval.fn = B_sqrt; return FUNC; }
"exp"     { yylval.fn = B_exp; return FUNC; }
"log"     { yylval.fn = B_log; return FUNC; }
"print"   { yylval.fn = B_print; return FUNC; }

[a-zA-Z][a-zA-Z0-9]*   { yylval.s = lookup(yytext); return NAME; }

[0-9]+"."[0-9]*{EXP}? |
"."?[0-9]+{EXP}?                    { yylval.d = atof(yytext); return NUMBER; }
"//".*
" "
[\t]        { return _EOF; }

\\\n        { printf("c> "); }
\n                  { return EOL; }
.                   { yyerror("Mystery character %c\n", *yytext); }

%%
```

==========

calc.y

```
/*

        Adopted from "flex & bison" by John Levine
        Copyright (c) 2009 John Levine
        ISBN: 978-0-596-15597-1

*/

%{
# include <stdio.h>
# include <stdlib.h>
# include "fb3-2.h"

extern int yylex();
extern char* yytext;
```

```
extern struct ast *root;
%}

%union {
        struct ast *a;
        double d;
        struct symbol *s;
        struct symlist *sl;
        int fn;
}

/* declare tokens */
%token <d> NUMBER
%token <s> NAME
%token <fn> FUNC
%token EOL
%token _EOF

%token IF THEN ELSE WHILE DO LET

%nonassoc <fn> CMP
%right '='
%left '+' '-'
%left '*' '/' %nonassoc '|' UMINUS

%type <a> exp stmt list explist
%type <sl> symlist

%start calclist

%%

stmt:
                IF exp THEN list                        { $$ = newflow('I', $2, $4, NULL); }
        |       IF exp THEN list ELSE list      { $$ = newflow('I', $2, $4, $6); }
        |       WHILE exp DO list                       { $$ = newflow('W', $2, $4, NULL); }
        |       exp
        ;

list:
                /* nothing */                   { $$ = NULL; }
        |       stmt ';' list                   {       if ($3 == NULL) $$ = $1;
```

```
                                                        else $$ = newast('L', $1, $3);
}
        ;

exp:
                exp CMP exp                             { $$ = newcmp($2, $1, $3); }
        |       exp '+' exp                             { $$ = newast('+', $1,$3); }
        |       exp '-' exp                             { $$ = newast('-', $1,$3);}
        |       exp '*' exp                             { $$ = newast('*', $1,$3); }
        |       exp '/' exp                             { $$ = newast('/', $1,$3); }
        |       '|' exp                         { $$ = newast('|', $2, NULL); }
        |       '(' exp ')'                             { $$ = $2; }
        |       '-' exp %prec UMINUS { $$ = newast('M', $2, NULL); }
        |       NUMBER                                          { $$ = newnum($1); }
        |       NAME                                    { $$ = newref($1); }
        |       NAME '=' exp                    { $$ = newasgn($1, $3); }
        |       FUNC '(' explist ')'        { $$ = newfunc($1, $3); }
        |       NAME '(' explist ')'        { $$ = newcall($1, $3); }
        ;


explist:
                exp
        |       exp ',' explist             { $$ = newast('L', $1, $3); }
        ;

symlist:
                NAME                                    { $$ = newsymlist($1, NULL); }
        |       NAME ',' symlist                { $$ = newsymlist($1, $3); }
        ;

calclist: /* nothing */
//      |       calclist stmt EOL                { printf("= %4.4g\n> ", eval($2));
        |       calclist list EOL                { {     if (root == NULL) root = $2;
                                                        else if (root->nodetype != 'L'){
                                                                root = newast('L', root,
$2);
                                                        }
                                                        else {
                                                                struct ast *tmp = root;
                                                                struct ast *prev_tmp =
NULL;
```

```
                                                        while (tmp->nodetype
== 'L'){
                                                                prev_tmp =
tmp;
                                                                tmp = tmp->r;
                                                        }
                                                        prev_tmp->r =
newast('L', tmp, $2);
                                                }
                                                printf("> ");
                                        }

//      |       calclist LET NAME '(' symlist ')' '=' list EOL
//                                      { dodef($3, $5, $8);
//                                              printf("Defined %s\n> ",
$3->name);
//                                      }
        |       calclist error EOL              { yyerrok; printf("> "); } ;
        |       calclist _EOF                   { return 0; }

%%

fb3-2.h
/*

        Adopted from "flex & bison" by John Levine
        Copyright (c) 2009 John Levine
        ISBN: 978-0-596-15597-1

*/
#include "llvm/IR/Value.h"
#include "llvm/IR/Instructions.h"

/* interface to the lexer */
extern int yylineno; /* from lexer */
void yyerror(const char* str,...);
int yyparse (void);

/* symbol table */
struct symbol { /* a variable name */
        char *name;
        double value;
        struct ast *func; /* stmt for the function */
```

```
        struct symlist *syms; /* list of dummy args */
        llvm::Value *llvm_value;
        llvm::AllocaInst *alloc;
};


struct symbol *lookup(char*);

/* list of symbols, for an argument list */
struct symlist {
        struct symbol *sym;
        struct symlist *next;
};

struct symlist *newsymlist(struct symbol *sym, struct symlist *next);
void symlistfree(struct symlist *sl);

typedef enum bifs {
        B_sqrt = 1,
        B_exp,
        B_log,
        B_print
}bifs_t;

/* nodes in the abstract syntax tree */
/* all have common initial nodetype */
struct ast {
        int nodetype;
        struct ast *l;
        struct ast *r;
};

struct fncall {
        int nodetype;
        struct ast *l;
        enum bifs functype;
};

struct ufncall {
        int nodetype;
        struct ast *l;
        struct symbol *s;
};
```

```
struct flow {
        int nodetype;
        struct ast *cond;
        struct ast *tl;
        struct ast *el;
};

struct numval {
        int nodetype;
        double number;
};

struct symref {
        int nodetype;
        struct symbol *s;
};

struct symasgn {
        int nodetype;
        struct symbol *s;
        struct ast *v;
};

/* build an AST */
struct ast *newast(int nodetype, struct ast *l, struct ast *r);
struct ast *newcmp(int cmptype, struct ast *l, struct ast *r);
struct ast *newfunc(int functype, struct ast *l);
struct ast *newcall(struct symbol *s, struct ast *l);
struct ast *newref(struct symbol *s);
struct ast *newasgn(struct symbol *s, struct ast *v);
struct ast *newnum(double d);
struct ast *newflow(int nodetype, struct ast *cond, struct ast *tl, struct ast *tr);

/* define a function */
void dodef(struct symbol *name, struct symlist *syms, struct ast *stmts);

/* evaluate an AST */
double eval(struct ast *);

/* delete and free an AST */
void treefree(struct ast *);
```

fb3-2.cpp

```
/*

        Adopted from "flex & bison" by John Levine
        Copyright (c) 2009 John Levine
        ISBN: 978-0-596-15597-1

*/
/*
* helper functions for fb3-2 */
# include <stdio.h>
# include <stdlib.h>
# include <stdarg.h>
# include <string.h>
# include <math.h>
# include "fb3-2.h"

/* symbol table */
/* simple symtab of fixed size */
#define NHASH 9997
struct symbol symtab[NHASH];


/* hash a symbol */
static unsigned symhash(char *sym)
{
        unsigned int hash = 0; unsigned c;
        while((c = *sym++)) hash = hash*9 ^ c;
        return hash;
}

struct symbol * lookup(char* sym)
{
        struct symbol *sp = &symtab[symhash(sym)%NHASH];
        int scount = NHASH;
        while(--scount >= 0) {
                if(sp->name && !strcmp(sp->name, sym)) return sp;
                if(!sp->name) {
                        sp->name = strdup(sym);
                        sp->value = 0;
                        sp->func = NULL;
                        sp->syms = NULL;
```

```
                sp->llvm_value = NULL;
                sp->alloc = NULL;
                return sp;
            }

            if(++sp >= symtab+NHASH) sp = symtab;
        }
        yyerror("symbol table overflow\n");
        abort();
}

struct ast *newast(int nodetype, struct ast *l, struct ast *r)
{
        struct ast *a = (struct ast *)malloc(sizeof(struct ast));
        if(!a){
                yyerror("out of space");
                exit(0);
        }
        a->nodetype = nodetype;
        a->l = l;
        a->r = r;
        return a;
}

struct ast * newnum(double d)
{
        struct numval *a = (struct numval *)malloc(sizeof(struct numval));
        if(!a) {
                yyerror("out of space");
                exit(0);
        }
        a->nodetype = 'K'; a->number = d;
        return (struct ast *)a;
}

struct ast *newcmp(int cmptype, struct ast *l, struct ast *r)
{
        struct ast *a = (struct ast *)malloc(sizeof(struct ast));
        if(!a) {
                yyerror("out of space");
                exit(0);
        }
        a->nodetype = '0' + cmptype; a->l = l;
```

```
	a->r = r;
	return a;
}

struct ast *newfunc(int functype, struct ast *l)
{
	struct fncall *a = (struct fncall *)malloc(sizeof(struct fncall));
	if(!a) {
		yyerror("out of space");
		exit(0);
	}
	a->nodetype = 'F';
	a->l = l;
	a->functype = (enum bifs)functype;
	return (struct ast *)a;
}

struct ast *newcall(struct symbol *s, struct ast *l)
{
	struct ufncall *a = (struct ufncall *)malloc(sizeof(struct ufncall));
	if(!a) {
		yyerror("out of space");
		exit(0);
	}
	a->nodetype = 'C';
	a->l = l;
	a->s = s;
	return (struct ast *)a;
}

struct ast * newref(struct symbol *s)
{
	struct symref *a = (struct symref *)malloc(sizeof(struct symref));
	if(!a) {
		yyerror("out of space");
		exit(0);
	}
	a->nodetype = 'N';
	a->s = s;
	return (struct ast *)a;
}

struct ast *newasgn(struct symbol *s, struct ast *v)
```

```
{
	struct symasgn *a = (struct symasgn *)malloc(sizeof(struct symasgn));
	if(!a) {
		yyerror("out of space");
		exit(0);
	}
	a->nodetype = '=';
	a->s = s;
	a->v = v;
	return (struct ast *)a;
}

struct ast *newflow(int nodetype, struct ast *cond, struct ast *tl, struct ast *el)
{
	struct flow *a = (struct flow *)malloc(sizeof(struct flow));
	if(!a) {
		yyerror("out of space");
		exit(0);
	}
	a->nodetype = nodetype; a->cond = cond;
	a->tl = tl;
	a->el = el;
	return (struct ast *)a;
}

/* free a tree of ASTs */
void treefree(struct ast *a)
{
	switch(a->nodetype) {
		/* two subtrees */ case '+':
		case '-':
		case '*':
		case '/':
		case '1': case '2': case '3': case '4': case '5': case '6': case 'L':
			treefree(a->r);

		/* one subtree */
		case '|':
		case 'M': case 'C': case 'F':
			treefree(a->l);

		/* no subtree */
		case 'K': case 'N':
```

```
                break;

        case '=':
                free( ((struct symasgn *)a)->v); break;

                /* up to three subtrees */
        case 'I': case 'W':
                free( ((struct flow *)a)->cond);
                if( ((struct flow *)a)->tl) treefree( ((struct flow *)a)->tl);
                if( ((struct flow *)a)->el) treefree( ((struct flow *)a)->el);
                break;

        default:
                printf("internal error: free bad node %c\n", a->nodetype);
        }
    free(a); /* always free the node itself */
}

struct symlist *newsymlist(struct symbol *sym, struct symlist *next)
{
    struct symlist *sl = (struct symlist *)malloc(sizeof(struct symlist));
    if(!sl) {
        yyerror("out of space");
        exit(0);
    }
    sl->sym = sym;
    sl->next = next;
    return sl;
}

/* free a list of symbols */ void
symlistfree(struct symlist *sl) {
    struct symlist *nsl;
    while(sl) {
        nsl = sl->next; free(sl);
        sl = nsl;
    }
}

/* define a function */
void dodef(struct symbol *name, struct symlist *syms, struct ast *func) {
    if(name->syms) symlistfree(name->syms);
    if(name->func) treefree(name->func);
```

```
	name->syms = syms;
	name->func = func;
}
```

main.cpp

```
//
//  main.cpp
//  calc
//
//  Created by Alan Nilsson on 2/28/15.
//  Copyright (c) 2015 Alan Nilsson, Akshay Khole. All rights reserved.
//

#include "llvm/ADT/ArrayRef.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/Constants.h"
#include "llvm/IR/Instructions.h"
#include "llvm/Support/raw_ostream.h"
#include <vector>
#include <string>

#include <iostream>
#include <stdarg.h>
# include "fb3-2.h"

using namespace llvm;


void yyerror(const char* str,...)
{
	va_list ap;
	va_start(ap, str);
	fprintf(stderr, "%d: error: ", yylineno);
	vfprintf(stderr, str, ap);
	fprintf(stderr, "\n");
}

struct ast *root = NULL;
```

```
void dump_ast(struct ast *a);
Value* build_ir(struct ast *root);
llvm::Module *module;
llvm::LLVMContext & context = llvm::getGlobalContext();
llvm::IRBuilder<> builder(getGlobalContext());

#define BUILD
int main(int argc, const char * argv[]) {
#ifdef BUILD
	module = new llvm::Module("calc", context);
	Value *result = NULL;

	/* Setup main function and entry point */
	llvm::FunctionType *funcType = llvm::FunctionType::get(builder.getVoidTy(), false);
	llvm::Function *mainFunc =
		llvm::Function::Create(funcType, llvm::Function::ExternalLinkage, "main", module);
	llvm::BasicBlock *entry = llvm::BasicBlock::Create(context, "entrypoint", mainFunc);
	builder.SetInsertPoint(entry);

	/* Get user expression input */
	printf("> ");
	yyparse();

	/* build up IR command from parse tree */
	result = build_ir(root);
	printf("\n");
//#else
	/* Get user expression input */
//	printf("> ");
//	yyparse();
	printf("------------------------------------------------------------\n");
	printf("Dump AST\n");
	printf("------------------------------------------------------------\n");
	dump_ast(root);
	printf("\n\n\n");
#endif

#ifdef BUILD
	/* setup function call to printf */
	llvm::Value *formatStr = builder.CreateGlobalStringPtr("Result: %f\n");

	std::vector<llvm::Type *> putsArgs;
		putsArgs.push_back(builder.getInt8Ty()->getPointerTo());
```

```
		putsArgs.push_back(builder.getDoubleTy());

	llvm::ArrayRef<llvm::Type*>  argsRef(putsArgs);
	llvm::FunctionType *putsType =
		llvm::FunctionType::get(builder.getInt32Ty(), argsRef, true);
	llvm::Constant *putsFunc = module->getOrInsertFunction("printf", putsType);

	/* call printf */
	if (result) builder.CreateCall2(putsFunc, formatStr, result);

	/* return from main */
	builder.CreateRetVoid();

	printf("------------------------------------------------------------\n");
	printf("LLVM IR\n");
	printf("------------------------------------------------------------\n");
	module->dump();
	FILE *f = fopen("calc.ll", "w");
	if (f){
		raw_ostream *os =  new raw_fd_ostream(f->_file, true);
		module->print(*os, NULL);
	}
#endif


}

Value* build_ir(struct ast *root)
{
	if(!root) return NULL;
	Value *rv = NULL;

	switch(root->nodetype) {
		/* constant */
		case 'K':{
				double d_val = ((struct numval *)root)->number;
				rv = ConstantFP::get(context, APFloat(d_val));
			}
			break;

		/* name reference */
		case 'N':{
				AllocaInst *a = ((struct symref *)root)->s->alloc;
```

```
                        if (!a){
                                printf("Undefined variable: %s\n", ((struct symref
*)root)->s->name);
                                exit(1);
                        }
                        rv = builder.CreateLoad(a);
                }
                break;

        /* assignment */
        case '=':{
                        /* get values from ast nodes */
                        const char * name = ((struct symasgn *)root)->s->name;

                        /* alloc var and store in value */
                        AllocaInst *a = builder.CreateAlloca(builder.getDoubleTy(), 0, name);
                        Value *val = build_ir(((struct symasgn *)root)->v);
                        builder.CreateStore(val, a);
                        rv = val;

                        /* alloc inst back to symbol table for later use */
                        ((struct symasgn *)root)->s->alloc = a;
                }
                break;

        case '+':{
                        Value *op1 = build_ir(root->l);
                        Value *op2 = build_ir(root->r);
                        rv = builder.CreateFAdd(op1, op2, "addtmp");
                }
                break;

        case '-':{
                        Value *op1 = build_ir(root->l);
                        Value *op2 = build_ir(root->r);
                        rv = builder.CreateFSub(op1, op2, "subtmp");
                }
                break;

        case '*':{
                        Value *op1 = build_ir(root->l);
                        Value *op2 = build_ir(root->r);
                        rv = builder.CreateFMul(op1, op2, "multmp");
```

```
                }
                break;

        case '/':{
                        Value *op1 = build_ir(root->l);
                        Value *op2 = build_ir(root->r);
                        rv = builder.CreateFDiv(op1, op2, "divtmp");
                }
                break;

        case 'M':{
                        Value *zero = ConstantFP::get(context, APFloat(0.0));
                        rv = builder.CreateFSub(zero, build_ir(root->l), "negtmp");
                }
                break;

        case '1':{
                        Value *op1 = build_ir(root->l);
                        Value *op2 = build_ir(root->r);
                        Value *t = builder.CreateFCmpUGT( op1, op2, "cmptmp");
                        rv = builder.CreateUIToFP(t,
Type::getDoubleTy(getGlobalContext()));
                }
                break;

        case '2':{
                        Value *op1 = build_ir(root->l);
                        Value *op2 = build_ir(root->r);
                        Value *t = builder.CreateFCmpULT( op1, op2, "cmptmp");
                        rv = builder.CreateUIToFP(t,
Type::getDoubleTy(getGlobalContext()));
                }
                break;
                break;

        case '3':{
                        Value *op1 = build_ir(root->l);
                        Value *op2 = build_ir(root->r);
                        Value *t = builder.CreateFCmpUNE( op1, op2, "cmptmp");
                        rv = builder.CreateUIToFP(t,
Type::getDoubleTy(getGlobalContext()));
                }
                break;
```

```
                        break;

                case '4':{
                                Value *op1 = build_ir(root->l);
                                Value *op2 = build_ir(root->r);
                                Value *t = builder.CreateFCmpUEQ( op1, op2, "cmptmp");
                                rv = builder.CreateUIToFP(t,
Type::getDoubleTy(getGlobalContext()));
                        }
                        break;
                        break;

                case '5':{
                                Value *op1 = build_ir(root->l);
                                Value *op2 = build_ir(root->r);
                                Value *t = builder.CreateFCmpUGE( op1, op2, "cmptmp");
                                rv = builder.CreateUIToFP(t,
Type::getDoubleTy(getGlobalContext()));
                        }
                        break;
                        break;

                case '6':{
                                Value *op1 = build_ir(root->l);
                                Value *op2 = build_ir(root->r);
                                Value *t = builder.CreateFCmpULE( op1, op2, "cmptmp");
                                rv = builder.CreateUIToFP(t,
Type::getDoubleTy(getGlobalContext()));
                        }
                        break;

                /* this is a connecting node to bridge to sub-trees together */
                case 'L':
                        build_ir(root->l);
                        rv = build_ir(root->r); break;
        }
        return rv;
}

void dump_ast(struct ast *a)
{
        if(!a) return;
```

```
switch(a->nodetype) { /* constant */
        case 'K':
                printf("const %f ", ((struct numval *)a)->number);
                break;

        /* name reference */
        case 'N':
                printf("<%s> ", ((struct symref *)a)->s->name);
                break;

        /* assignment */
        case '=':
                printf("%c %s ", a->nodetype, ((struct symref *)a)->s->name);
                dump_ast(((struct symasgn *)a)->v);
                break;

        /* expressions */
        case '+':
                printf("%c ", a->nodetype);
                dump_ast(a->l); dump_ast(a->r); break;
        case '-':
                printf("%c ", a->nodetype);
                dump_ast(a->l); dump_ast(a->r); break;
        case '*':
                printf("%c ", a->nodetype);
                dump_ast(a->l); dump_ast(a->r); break;
        case '/':
                printf("%c ", a->nodetype);
                dump_ast(a->l); dump_ast(a->r); break;
        case '|':
                printf("%c ", a->nodetype);
                dump_ast(a->l); dump_ast(a->r); break;
        case 'M':
                printf("- ");
                dump_ast(a->l); break;
        case '1':
                printf("> ");
                dump_ast(a->l); dump_ast(a->r); break;
        case '2':
                printf("< ");
                dump_ast(a->l); dump_ast(a->r); break;
        case '3':
                printf("!= ");
```

```
                dump_ast(a->l); dump_ast(a->r); break;
        case '4':
                printf("== ");
                dump_ast(a->l); dump_ast(a->r); break;
        case '5':
                printf(">= ");
                dump_ast(a->l); dump_ast(a->r); break;
        case '6':
                printf("<= ");
                dump_ast(a->l); dump_ast(a->r); break;

        case 'L':
                dump_ast(a->l); printf("\n");dump_ast(a->r); break;
    }
}
```

Input/output listing

```
[Sperky:calc/build/Debug] alan%  ./calc
> a=1;
> b=-5;
> c=a*-b;
> d=(c+3)*2;
> e=d>15;
>

------------------------------------------------------------
Dump AST
------------------------------------------------------------
= a const 1.000000
= b - const 5.000000
= c * <a> - <b>
= d * + <c> const 3.000000 const 2.000000
> = e <d> const 15.000000


------------------------------------------------------------
LLVM IR
------------------------------------------------------------
; ModuleID = 'calc'

@0 = private unnamed_addr constant [12 x i8] c"Result: %f\0A\00"

define void @main() {
```

```
entrypoint:
  %a = alloca double
  store double 1.000000e+00, double* %a
  %b = alloca double
  store double -5.000000e+00, double* %b
  %c = alloca double
  %0 = load double* %a
  %1 = load double* %b
  %negtmp = fsub double 0.000000e+00, %1
  %multmp = fmul double %0, %negtmp
  store double %multmp, double* %c
  %d = alloca double
  %2 = load double* %c
  %addtmp = fadd double %2, 3.000000e+00
  %multmp1 = fmul double %addtmp, 2.000000e+00
  store double %multmp1, double* %d
  %e = alloca double
  %3 = load double* %d
  store double %3, double* %e
  %cmptmp = fcmp ugt double %3, 1.500000e+01
  %4 = uitofp i1 %cmptmp to double
  %5 = call i32 (i8*, double, ...)* @printf(i8* getelementptr inbounds ([12 x i8]* @0, i32 0, i32 0), double %4)
  ret void
}

declare i32 @printf(i8*, double, ...)
[Sperky:calc/build/Debug] alan%  lli calc.ll
Result: 1.000000
```

Other related material

Installation instructions:

LLVM has easy integration with Xcode and we can a leverage a powerful IDE for developing compilers using LLVM.

How to import project:

1. Open a new xcode project and give path to source code
2. Click on project
3. Go to build settings for the calc target (the only one now)
4. Scroll to the very bottom
5. Under “User-Defined” heading, change LLVM_CONFIG_PATH to the path to your llvm-config install

LLVM_CFLAGS & LLVM_LDFLAGS will work themselves when you build. If it falls down the first time around, delete the file llvm.xcconfig and it will get regenerated.