Tutorial 3 Extend the Expression Language Compiler for LLVM IR

ECE 466/566 Spring 2025

Due Feb. 17, 2025

1. Objectives and Prerequisites

In Tutorial 2, we generated LC-3 code directly from the rules in our parser. Now, we're going to generate LLVM IR instead. While the overall process is very similar to printing assembly, we need to handle some aspects differently. In so doing, we'll get a deeper look at:

- Using LLVM include files and libraries
- Insights for generating LLVM IR:
    - The LLVM Builder object simplifies constants for us automatically.
    - Inside LLVM, registers are represented as pointers.
    - Loads aren't easy to optimize or remove.

An example of generating LLVM IR has already been covered in the video on Generating LLVM IR. Please watch this if you haven't already.

2. Extend Tutorial 2 code with a few grammar changes

Allow Multiple Statements

To make this example more interesting, we'll extend the Scanner and Parser from Tutorial 2 so that it can match multiple expressions in a row instead of just matching a single expression. To signal the end of the expressions, we'll place a return statement at the end of the file. Now, a complete program may look like this:

```
R0 = 5;
R1 = R0 + [R1] + R2;
return R1; /* end of program */
```

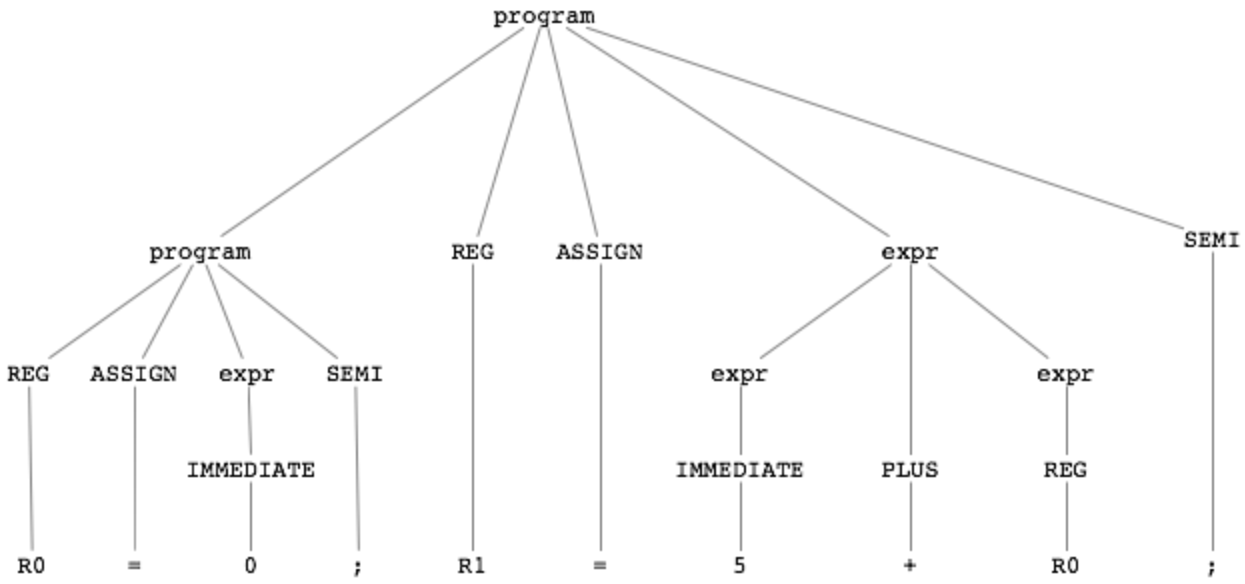
A few things to note about this program.

1. The first update of R0 should be used by the second expression.
2. R1 is both a source and a destination register for the second expression.
3. Our language allows a register to be used without defining it first, like R2 in the code above. While this may make sense when generating assembly code, we'll interpret this as an error and we should generate a warning. Also, note that this case is an example of an inherited attribute, since you can only figure it out by remembering which registers have already been updated by previous expressions.

To make this work, we'll modify the grammar as follows:

```
program:   REG ASSIGN expr SEMI  { /* insert action */             }
         | program REG ASSIGN expr SEMI { /* insert action */ }
         | program RETURN REG SEMI { /* insert action; program is
done */ }
;
```

The recursive definition of `program` allows it to match multiple statements. Here's a parse tree that illustrates what's happening:



Note that the top-level program node now has a child that's a program node too. As statements are added, new program nodes will be added to the top of the tree to support inclusion of new statements.

Allow Function Arguments

We will be interpreting this program as a function. It has arguments and a return statement. To denote the arguments, we'll support special arguments using the regular expression [Aa][0-3] to represent four argument registers.

```
R0 = A0 + A1; // add two arguments
return R0;    // return sum
```

Modify the Grammar

Modify the grammar from the Tutorial 2 folder so that the program rule looks the same as below. I've put a working version of Tutorial 2 in github at `ncstate_ece566_spring2025/tutorial/3/`. Use the `git pull` command to get these updates.

```
program:   REG ASSIGN expr SEMI  { /* insert action */             }
         | program REG ASSIGN expr SEMI { /* insert action */ }
         | program RETURN REG SEMI { return 0; /* program is done */
}
;

expr:        IMMEDIATE
           | REG
           | ARG
           | expr PLUS expr
           | expr MINUS expr
           | LPAREN expr RPAREN
           | MINUS expr
           | LBRACKET expr RBRACKET
;
```

(In order to support the return statement and argument registers, you'll need to extend the scanner too. But, that should be easy, so the code for that is omitted. Try to do it on your own!)

Test your new grammar and make sure it works. Here are the commands:

Manual Commands	Cmake Commands

`bison -y -d -o expr.y.cpp expr.y`
`flex -oexpr.lex.cpp expr.lex`
`clang++ -o expr expr.y.cpp expr.lex.cpp -ly -ll`
	`cd tutorial/3`
	`mkdir build`
	`cd build`
	`cmake ..`
	`make`

Note, we'll use C++ name extensions (e.g. expr.y.cpp) for our files even when coding in C just to keep the examples simple.

3. Add Support for LLVM IR

As described in Generating LLVM IR, we need to setup a module, function, and basic block before we can generate LLVM instructions. This means we need our own main function so that we can setup the LLVM module before parsing starts.

Take a look at the main Function

If you haven't noticed, we have our own main function in expr.y. We will add code to set up the LLVM module before we call the parser:

```
int main() {
  yydebug = 0;
  yyin = stdin;
  // yyparse() triggers parsing of the input
  if (yyparse()==0) {
    // all is good
  } else {
    printf("There was a problem! Read error messages above.\n");
  }
  return 0;
}
```

Now, since we are making our own main, you may (or may not) need to declare a prototype for yylex. Just to be safe, at the top of the expr.y file, add this:

```
int yylex();
```

Build the Module

Inside main and before we call yyparse, we can perform all of the setup that was described in the video (Generating LLVM IR).

So, at the top of the expr.y and expr.lex file, we need to include the following headers:

C	`#include "llvm-c/Core.h"`
	`#include "llvm-c/BitReader.h"`
	`#include "llvm-c/BitWriter.h"`

C++	`#include "llvm/IR/LLVMContext.h"`
	`#include "llvm/IR/Value.h"`
	`#include "llvm/IR/Function.h"`
	`#include "llvm/IR/Type.h"`
	`#include "llvm/IR/IRBuilder.h"`
	
	`#include "llvm/Bitcode/BitcodeReader.h"`

```
#include "llvm/Bitcode/BitcodeWriter.h"
#include "llvm/Support/SystemUtils.h"
#include "llvm/Support/ToolOutputFile.h"
#include "llvm/Support/FileSystem.h"

using namespace llvm;
```

IMPORTANT: In the expr.lex file, make sure you add them before the include statement for expr.y.hpp.

Then, we need to add code that creates a Module, creates a function, makes an initial basic block, and creates a builder. First, declare the Builder to be a global variable by placing it's declaration at the top of the expr.y file in the initial code block marked by %{ and %}.

C
```
static LLVMBuilderRef  Builder;
```

C++
```
static LLVMContext TheContext;
static IRBuilder<> Builder(TheContext);
```

Inside main and before yyparse, we need to perform these actions:

C
```
// Make a Module
LLVMModuleRef Module = LLVMModuleCreateWithName("Tutorial3");

LLVMTypeRef i32 = LLVMInt32Type();
LLVMTypeRef *args[4] = {i32,i32,i32,i32};

// Make a void function type with 4 arguments
LLVMTypeRef IntFnTy =
LLVMFunctionType(LLVMInt32Type(),args,4,0);

// Make a void function named main (the start of the
program!)
LLVMValueRef Fn =
LLVMAddFunction(Module,"myfunction",IntFnTy);

// Add a basic block to main to hold new instructions
LLVMBasicBlockRef BB = LLVMAppendBasicBlock(Fn,"entry");

// Create a Builder object that will construct IR for us
```

	`Builder = LLVMCreateBuilder();`
	`// Ask builder to place new instructions at end of the`
	`// basic block`
	`LLVMPositionBuilderAtEnd(Builder,BB);`
	
	`// Now we're ready to make IR, call yyparse()`

C++	`// Make Module`
	`Module *M = new Module("Tutorial3", TheContext);`
	
	`Type *i32 = Builder.getInt32Ty();`
	`std::vector<Type*> args = {i32,i32,i32,i32};`
	
	`// Create void function type with no arguments`
	`FunctionType *FunType =`
	`  FunctionType::get(Builder.getInt32Ty(),args,false);`
	
	`// Create a main function`
	`Function *Function = Function::Create(FunType,`
	`  GlobalValue::ExternalLinkage, "myfunction",M);`
	
	`//Add a basic block to main to hold instructions`
	`BasicBlock *BB = BasicBlock::Create(TheContext, "entry",`
	`  Function);`
	`// Ask builder to place new instructions at end of the`
	`// basic block`
	`Builder.SetInsertPoint(BB);`
	
	`// Now we're ready to make IR, call yyparse()`

Only if parsing is successful should we write out the code to file. So, put the following code inside the if-statement if yyparse()==0.

C	`if (yyparse()==0) {`
	`  // Build the return instruction for the function`
	`  LLVMBuildRet(Builder,LLVMConstInt(LLVMInt32Type(),0,0));`
	`  LLVMWriteBitcodeToFile(Module,"main.bc");`
	
	`  // Dump LLVM IR to the screen for debugging`
	`  LLVMDumpModule(Module);`
	`}`

C++	if (yyparse()==0) {
	    std::error_code EC;
	    raw_fd_ostream OS("main.bc",EC,sys::fs::OF_None);
	    WriteBitcodeToFile(*M,OS);
	
	    // Dump LLVM IR to the screen for debugging
	    M->print(errs(),nullptr,false,true);
	}

Compile the Code and Link with LLVM Libraries

We don't need to change yyparse() yet to test the code. We can go ahead and run this to generate a main function that does nothing. But, first, we need to modify our build process a bit.

The bison and flex commands remain the same, but the clang/gcc command changes because we need to tell the compiler how to find the LLVM headers and how to link with the LLVM libraries. Fortunately, the llvm-config command makes this easy.

To get a list of the flags needed for a C compiler:

```
llvm-config-19 --cflags
```

For the C++ flags:

```
llvm-config-19 --cxxflags
```

To get a list of the flags needed to link with the libraries:

```
llvm-config-19 --ldflags
```

To get the list of libraries:

```
llvm-config-19 --libs
llvm-config-19 --system-libs
```

The system-libs refers to libraries on the host required to properly link, as opposed to libraries from LLVM.

So, the compile command the long way becomes (skip ahead to use make/cmake):

C++	1. flex -o expr.lex.cpp expr.lex 2. bison -d -o expr.y.cpp expr.y 3. clang++ -c -o expr.y.o expr.y.cpp `llvm-config-19 --cxxflags` 4. clang++ -c -o expr.lex.o expr.lex.cpp `llvm-config-19 --cxxflags` 5. clang++ -o expr expr.y.o expr.lex.o `llvm-config-19 --ldflags --libs` -ly -ll `llvm-config-19

	`--system-libs``

After you compile, run your program and give it some input. At the end, make sure it created a main.bc file in the same directory. Use llvm-dis to look inside it:

```
llvm-dis-19 main.bc
less main.ll
```

You'll get something like this:

```
; ModuleID = 'main.bc'

define void @main(i32 %0, i32 %1, i32 %2, i32 %3) {
entry:
  ret i32 0
}
```

4. Generate LLVM IR in each rule

We'll now modify the rules of the grammar to emit LLVM IR rather than LC-3 instructions.

Building Instructions and Constants in the LLVM IR

To build instructions, we'll make extensive use of the Builder interface in LLVM IR. The Builder is an object that makes it easier to build instructions by hiding some of the details of creating objects that represent instructions. Some of the relevant functions we need for this tutorial are:

- `LLVMBuildAdd` or `IRBuilder::CreateAdd`
- `LLVMBuildSub` or `IRBuilder::CreateSub`
- `LLVMBuildNeg` or `IRBuilder::CreateNeg`
- `LLVMBuildIntToPtr` or `IRBuilder::CreateIntToPtr`
- `LLVMBuildLoad` or `IRBuilder::CreateLoad`

The Builder also supports building constant values. For this example, we'll assume that all constants are 32-bit integers. Such a constant can be made as follows for the constant value 100:

C	`// make the constant 100 in LLVM IR` `LLVMValueRef constant_100 = LLVMConstInt(LLVMInt32Type(), 100, 0);`
C++	`Value* int100 = Builder.getInt32(100);`

Arguments

For the argument registers, we can access them using an interface through the function object in LLVM.

C	`// Remember Fn from the creation of the function;`
	`// May need to make Fn a global variable.`
	`LLVMValueRef arg0 = LLVMGetParam(Fn, 0);`

C++	`Function *F = Builder.GetInsertBlock()->getParent();` `Argument* arg0 = F->getArg(0); // get 0th argument`

Registers in LLVM IR

In Tutorial 2, we associated rules of the grammar with registers. But, now, we'll associate rules of the grammar with instructions in LLVM IR.

All instructions in LLVM are represented using a single type. In C, it's called `LLVMValueRef` in C++ it's `Value`. You may remember some code like this from the video:

C	`LLVMValueRef result = LLVMBuildAdd(Builder, op1, op2, "add");`
C++	`Value *result = Builder.CreateAdd(op1, op2, "add");`

In this case, the literal meaning of `result` is ***a pointer*** to the add instruction. But, in terms of IR, we think of `result` as the register that holds the value computed by the `Add` instruction. Even though we are now associating rules with instructions, it's the same thing as associating rules with registers in the IR. However, a difference is that we don't get to say which register it is, instead LLVM gets to decide that for us. And, whenever we use `result`, it's the same as referring to the result of the computation in whatever register LLVM selected.

Map Registers in the Language to Instructions in LLVM

As we parse statements in the Expression Language, we will map registers in the expression language (R0-R7) to the instructions in the LLVM IR that we generate.

We need an array to hold the mappings. The array should be large enough to have an element for each register in the source language, R0-R7:

C	`LLVMValueRef regs[8] = {NULL};`
C++	`Value *regs[8] = {NULL};`

Just to make sure we don't have an NULL values in our regs array once we start parsing, add a loop to main to set all to an object that represents 0 in the main function:

C	`for(int i=0; i<8; i++) {`
	`  regs[i] = LLVMConstInt(LLVMInt32Type(), 0, 0);`
	`}`

C++	`for(int i=0; i<8; i++) {`
	`  regs[i] = Builder.getInt32(0);`
	`}`

If we generate a constant value to put in R0, then we update the `regs` array with that value:

C	`// R0 = 5` `regs[0]  = LLVMConstInt(LLVMInt32Type(),5,0);`
C++	`// R0 = 5` `regs[0] = Builder.getInt32(5);`

Now, we if need to add two registers together, in Expression Language, like this:

```
R1 =  R0 + R2;
```

We can use code in our compiler that's similar to this:

C	`regs[1] = LLVMBuildAdd(Builder,regs[0],regs[2],"");`
C++	`regs[1] = Builder.CreateAdd(regs[0],regs[2]);`

In the grammar, we'll modify the logic to work as follows:

C	`%union {`
	`  int reg;`
	`  int imm;`
	`  LLVMValueRef val;`
	`}`

C++	`%union {` `  int reg;`

	int imm; Value* val; }

```
%type <reg> REG ARG
%type <imm> IMMEDIATE
%type <val> expr
```

Note, we are now specifying that all expr nodes are LLVMValueRef (or Value*). This means that all rules produce a result that's an LLVMValueRef (or Value*). Let's consider the rule for IMMEDIATE. We can build a constant, which as it happens, returns an LLVMValueRef (or Value*):

C	`expr: IMMEDIATE { $$ = LLVMConstInt(LLVMInt32Type(),$1,0); }`
C++	`expr: IMMEDIATE { $$ = Builder.getInt32($1); }`

This actions says that the an IMMEDIATE produces a constant value specified in $1.

For the REG, we just want to use the value that was previously assigned to the register:

```
expr: REG {  $$ = regs[$1]; }
```

So that we can handle programs made up of only constant expressions (like R0=1), let's go ahead and implement the rule that matches a program:

```
program:   REG ASSIGN expr SEMI    { regs[$1] = $3; }
```

This rule may seem too simple. But, since there is no such thing as a register, all we really need to do is update the mapping in the regs array. The mapping specifies the current value for each register, as described earlier in the document.

Similarly, the second program rule becomes:

```
program:  program REG ASSIGN expr SEMI    { regs[$2] = $4; }
```

However, you may want to add some error checking into this rule to make sure that you're accessing a legal index, i.e. $2 >= 0 and $2 < 8.

Finally, the last part of the program rule builds a return instruction:

```
program:  program RETURN REG SEMI
{
   Builder.CreateRet(regs[$3]); // build a return instruction
   return 0; // exit parser
}
```

Compile and test your code. Make sure you enter programs that only use the features of the language you've implemented, otherwise you may get segmentation faults within the LLVM libraries.

Try the following:

```
R0 = 10;
R1 = R0 + 5;
R2 = R1 + 3;
return R2;
```

Do you see any code in the generated module? How does the output relate to the input? What if we change it:

```
R0 = A0;
R1 = R0 + 5;
R2 = R1 + 3;
return R2;
```

Or:

```
R0 = 10;
R1 = R0 + 5;
R2 = R1 + A0;
return R2;
```

What differences do you notice?

Constant Folding

Some optimizations are easy for LLVM to do on the fly. In particular, operations involving constant values can be simplified during compile time. For example, what if we build an add operation like this:

```
%1 = add i32  0, 10
```

There is no need to actually execute this instruction. It's obvious that it will compute the number 10 so we should not waste processor cycles for work we can do at compile time. The Builder

object in LLVM automatically handles this kind of simplification when we ask it to make instructions that are operating on constants. This form of optimization is often referred to as Constant Folding.

Constant Folding can be applied on most logical and arithmetic operations involving constants because the compiler can simulate the meaning of the operation at compile time and pre-compute results. All the compiler needs is basic calculator logic.

Constant Folding helps optimize the construction of IR, and doing it during IR construction is valuable. If you recall in Tutorial 2, we generated a lot of unnecessary code for handling constants. Getting rid of that code required extra work. It's nice that the builder object does it for us transparently. However, it can also be surprising, and even misleading, because instead of generating instructions the Builder makes a constant. So, even though you've asked for an instruction, you don't get one and there is nothing in the IR that indicates what happened.

Now that you understand what's happening, hopefully it won't be confusing!

This explains why we weren't seeing any IR before. We have only written programs in the Expression Language that compute constants, and LLVM just computes the constants for us.

In fact, we will not see any code unless we use an argument or a load. The reason is simple: it's impossible to know what's in memory or an argument without considering a larger program context, so LLVM can't just optimize a load or argument away, at least not without doing a lot more work.

Load Expression

So, let's take a look at the load operation. First, try to implement it on your own. Then, proceed as follows.

We can build a load instruction using LLVMBuildLoad2. It's tempting to implement it as follows:

C	`expr: LBRACKET expr RBRACKET`
	`{`
	`  $$ = LLVMBuildLoad2(Builder,LLVMInt32Type(),$2,"");`
	`}`

C++	`expr: LBRACKET expr RBRACKET`
	`{`
	`  $$ = Builder.CreateLoad(Builder.getInt32Ty(),$2);`
	`}`

Try it out and see what happens.

What you will notice is that LLVM complains that the IR is invalid (but possibly not until you try to disassemble it). Load instructions must take a pointer as their operand. So, we must convert the expression into a pointer using the `inttoptr` instruction in LLVM.

C	`expr: LBRACKET expr RBRACKET`
	`{`
	`  // Make a an i32*`
	`  LLVMTypeRef int32ptr = LLVMPointerType(LLVMInt32Type(),0);`
	`  // Convert $2 into an i64*`
	`  LLVMValueRef int2ptr =`
	`LLVMBuildIntToPtr(Builder,$2,int32ptr,"");`
	`  // Build the load instruction`
	`  $$ = LLVMBuildLoad2(Builder,LLVMInt32Type(),int2ptr,"");`
	`}`

C++	`expr: LBRACKET expr RBRACKET`
	`{`
	`  Value * tmp = Builder.CreateIntToPtr($2,`
	`  PointerType::get(Builder.getInt32Ty(),0));`
	`  $$ = Builder.CreateLoad(Builder.getInt32Ty(),tmp);`
	`}`

Compile and test your code. Test with a program like this:

```
R0 = [100]; return R0;
```

Look at the main.bc file that was created by disassembling it. It should have a load instruction.

Now, implement any remaining rules; they will be similar to this one:

C	`expr: expr PLUS expr { $$ = LLVMBuildAdd(Builder,$1,$3,"a");}`
C++	`expr: expr PLUS expr { $$ = Builder.CreateAdd($1,$3,"add");}`

After you implement the remaining rules, compile and test your code. Try out a variety of programs and look at the IR they generate.

5. Questions and Other Things to Try On Your Own

Questions

1. What changes to the grammar allow it to support multi-statement programs?
   The changes to the grammar that allowed for the support of multi-statement programs was implementing a sequence of mathematical operations and expressions through parsing of multiple statements. In this tutorial, that would include "program" and "expr" where the program steps through the defined grammar.

2. Why do we use LLVMValueRef/Value* as the type for the expr non-terminal?
   We use "LLVMValueRef/Value*" as the type for the expr non-terminal because it verifies that the LLVM IR values are produced properly and can eventually be used in later expressions. Additionally, it is used to determine a distinct LLVM value after parsing occurs.
3. What is constant folding, and how does it help in this program?
   Constant folding is when expressions are determined in compile time rather than in the time it is run throughout the code (overall runtime). Constant folding is effective in this program because it eases the runtime of the code while it is compiling, and it produces more efficient LLVM IR.
4. Why do we only see code in the output that depends upon arguments or loads?
   We only see code in the output that depends upon arguments or loads because they are the values within the code that are different throughout the execution of the program. These values vary in range so the tutorial program produces code that can deal with the variation.

Things to try on your own (not for submission)

1. Can you extend the grammar and code generator to support stores?
2. Can you support other operators?
3. Rather than supporting registers (R0-R7), could you support arbitrary variable names?

Grading

Submit answers to the questions and your final code to Tutorial 3 on Moodle/GradeScope.

[50 points] Answers to questions 1-4 checked for completeness.
[50 points] Code checked for completeness.