Hey! I'm on Twitter! If you like this article or blog, **make sure to follow me** to get updates and more information.

Follow @lsegal No thanks

Writing Your Own Toy Compiler Using Flex, Bison and LLVM

By Loren Segal on September 09, 2009 at 918:118:506 AM

Update (March 19 2010): this article was updated for **LLVM 2.6** thanks to a great patch by John Harrison. He rocks!

I've always been interested in compilers and languages, but interest only gets you so far. A lot of the concepts of compiler design can easily go way over most programmers' heads, even the intelligent ones. Needless to say, I've tried, without much success, to write a small toy language/compiler before. I'd usually get caught up at the semantic parsing stage. And again, needless to say, this post is mostly inspired by my latest attempt, though this one has been much more successful (so far).

Fortunately over the last few years I've been involved in some projects that helped give me perspective and experience on what's really involved in building a compiler. The other thing I've been lucky to have in my corner this time is the help of LLVM, a tool which I'm hardly qualified to talk too much about, but it's been quite handy in implementing most of the business end (read: complex aspects) of my toy compiler.

So Why Are You Reading This?

Maybe you want to see what I've been doing with my time. It's more likely, however, that you're probably interested in compilers and languages as I am, and have probably been hitting similar roadblocks. You've probably wanted to try this but never found the resources, or did but couldn't quite follow. The goal of this article is to provide such a resource and explain in a relatively step by step manner how to create the most basic-but-functional compiler from start to "finish".

I won't be covering much theory, so if you haven't brushed up on your BNF grammars, AST data structures and the basic compiler pipeline, I suggest you do so. That said, I plan on keeping this as simple as possible. The goal, of course, is to make this an easy-to-understand introductory resource for people interested but not experienced with compilers.

What You Will End Up With

If you follow this article, you should end up with a language that can define functions, call functions, define variables, assign data to variables and perform basic math operations. It will support two basic types, doubles and integers. Some of the functionality is unimplemented, so you can have the satisfaction of actually implementing some of this stuff yourself and get the hang of writing a compiler with a little help.

Let's Get Some Questions Out of the Way

1. What languages do I need to know?

The tools we'll be using are C/C++ based. LLVM is specifically C++ and our toy

About

I'm Loren Segal, a programmer, Rubyist, author of YARD, musician, photographer, writer, former Amazon Web Services SDK Developer, and founder of Polyvex Interactive game studio. Me

Other Articles

Gucumber is Cucumber for Go

My Week With GitHub's Atom Editor

Samus Helps You Publish Your Open Source Software

The Audio Diaries Ch. 2 – Using RubyAudio

The Audio Diaries Ch. 1 — May I Have Your Attenuation Please?

More...

language will follow suit since there are some niceties of OOP and the STL (C++'s stdlib) that make for fewer lines of code. In addition to C, both Lex and Bison have their own syntax which may seem daunting at first, but I'll try to explain as much as possible. The grammar we're dealing with is actually very tiny (~100 LOC), so it should be feasible.

2. Is this really complicated?

Yes and no. There is plenty of stuff going on here and it might seem scary at first, but honestly, this is about as simple as it gets. We will also use a lot of tools to abstract the layers of complexity and make it manageable.

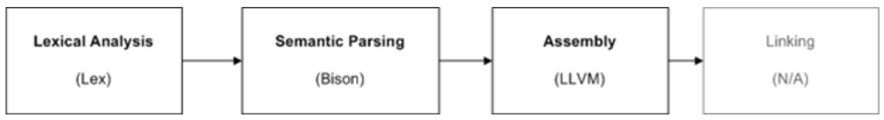
3. How long will it take?

What we will be building took me about 3 days of toying with, but I have a few failed attempts under my belt and those do have impact on my comprehension. Of course, this will be a "follow me" kind of deal, so it should be much shorter for you. To understand completely everything might take a little longer, but you should be able to run through all of this stuff (and hopefully understand a good amount of it) in an afternoon.

So, if you're ready, let's get started.

The Basic Compiler Recipe

Although you should already pretty much know this, a compiler is really a grouping of three to four components (there are some more sub-components) where data is fed from one to the next in a pipeline fashion. We will be using a different tool to help us build out each of these components. Here is a diagram of each step and the tool we will use:



You'll notice the linker step is greyed out. Our toy language won't be supporting compile time linking (plus most languages don't do compile time linking anymore anyway). To do our lexing we will be using the open source tool Lex, mostly available these days as Flex. Lexing usually goes hand in hand with semantic parsing, which we'll be performing with the help of Yacc, better known as Bison. Finally, once semantic parsing is done we can walk over our AST and generate our "bytecode", or machine code. For this, we'll be using LLVM, which technically generates intermediate bytecode, but we will be using LLVM's JIT (Just In Time) compilation to execute this bytecode on our machine.

To summarize, the steps are as follows:

1. **Lexical Analysis with *Flex***: Split input data into a set of tokens (identifiers, keywords, numbers, brackets, braces, etc.)
2. **Semantic Parsing with *Bison***: Generate an AST while parsing the tokens. Bison will do most of the legwork here, we just need to define our AST.
3. **Assembly with *LLVM***: This is where we walk over our AST and generate byte/machine code for each node. As crazy as it sounds, this is probably the *easiest* step.

Before moving too far along, you should probably consider installing Flex, Bison and LLVM, if you haven't already. We're going to need them pretty soon.

Defining Our Grammar

Our grammar is naturally the most central part of our language. From our grammar we inherently allow or disallow functionality. For our toy compiler, we will be using a standard C-like syntax because it's familiar and simple to parse. The canonical example of our toy language will be the following code:

```
int do_math(int a) {
  int x = a * 5 + 3
}

do_math(10)
```

Looks simple enough. We can see it’s typed the same way C is, but there are no semicolons separating statements. You’ll also notice there is no “return” statement in our grammar; this is something you can implement on your own.

There’s also no mechanism to print any results or anything. We will verify the correctness of our programs by inspecting the *very pretty* instruction listing that LLVM can print of the bytecode we’ve compiled, but more on that later.

Step 1. Lexical Analysis with Flex

This is the simplest step. Given our grammar, we need to break down our input into a list of known tokens. As mentioned before, our grammar has very basic tokens: identifiers, numbers (integers and floats), the mathematical operators, parentheses and braces. Our lex file “tokens.l”, which has a somewhat specialized grammar, is simply defined as:

Listing of tokens.l:

```
%{
#include <string>
#include "node.h"
#include "parser.hpp"
#define SAVE_TOKEN yylval.string = new std::string(yytext, yyleng)
#define TOKEN(t) (yylval.token = t)
extern "C" int yywrap() { }
%}

%%

[ \t\n]                 ;
[a-zA-Z_][a-zA-Z0-9_]*  SAVE_TOKEN; return TIDENTIFIER;
[0-9]+.[0-9]*           SAVE_TOKEN; return TDOUBLE;
[0-9]+                  SAVE_TOKEN; return TINTEGER;
"="                     return TOKEN(TEQUAL);
"=="                    return TOKEN(TCEQ);
"!="                    return TOKEN(TCNE);
"<"                     return TOKEN(TCLT);
"<="                    return TOKEN(TCLE);
">"                     return TOKEN(TCGT);
">="                    return TOKEN(TCGE);
"("                     return TOKEN(TLPAREN);
")"                     return TOKEN(TRPAREN);
"{"                     return TOKEN(TLBRACE);
"}"                     return TOKEN(TRBRACE);
"."                     return TOKEN(TDOT);
","                     return TOKEN(TCOMMA);
"+"                     return TOKEN(TPLUS);
"-"                     return TOKEN(TMINUS);
"*"                     return TOKEN(TMUL);
"/"                     return TOKEN(TDIV);
.                       printf("Unknown token!n"); yyterminate();

%%
```

The first section declares some specialized C code. We use a “SAVE_TOKEN” macro to keep the text of identifiers and numbers somewhere safe (instead of just the token itself), since Bison won’t have access to our ‘yytext’ variable. The first token tells us to skip all whitespace. You’ll also notice that we have some equality comparison tokens and such. Those are unimplemented for now, feel free to support them in your toy compiler!

So all we’re doing here is defining the tokens and their symbolic names. These symbols (TIDENTIFIER, etc.) will become “terminal symbols” in our grammar. We’re just returning them, but we’ve never defined them. **Where are they defined?** Why, in the bison grammar, of course. The parser.hpp file we’ve included will be generated by bison, and all the tokens inside it will be generated and available for use.

We run Flex on this tokens.l file to generate our “tokens.cpp” file, which will be compiled alongside our parser and provide the yylex() function that recognizes all of these tokens. We will run this command later though, because we need to generate that header file from bison first!

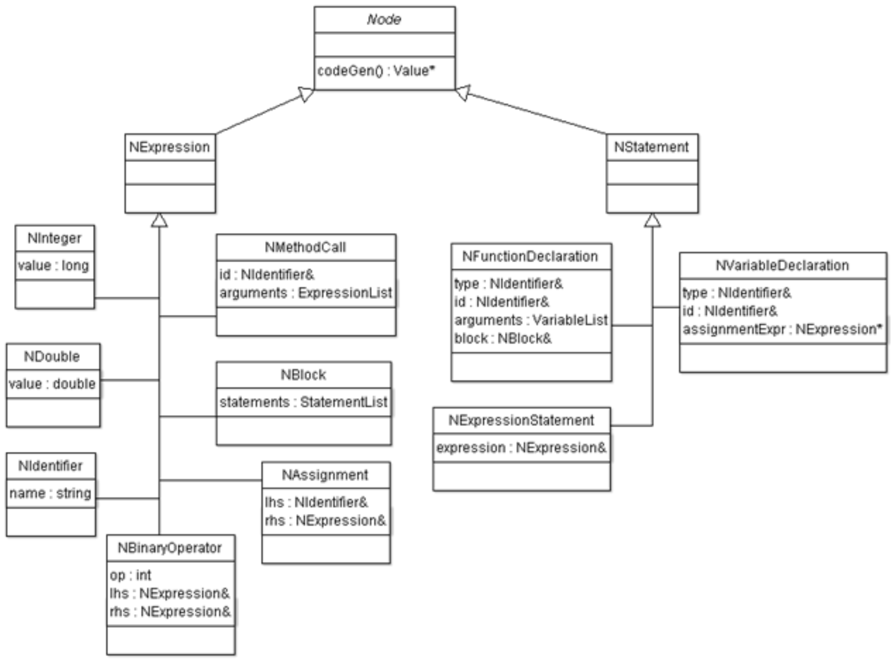
Step 2. Semantic Parsing with Bison

This is the challenging part of our mission. Generating an accurate, unambiguous grammar is not simple and takes a little bit of practice. Fortunately, our grammar is both simple and, for your benefit, mostly completed. Before we get into implementing our grammar, though, we need to talk design for a bit.

Designing the AST

The end product of semantic parsing is an AST. As we will see, Bison is quite elegantly optimized to generate AST's; it's really only a matter of plugging our nodes into the grammar.

The same way a string such as "int x" represents our language in text form, our AST is what represents our language in memory (before it is assembled). As such, we need to build up the data structures we will be using before we have the chance of plugging them into our grammar file. This process is pretty straightforward because we are basically creating a structure for every semantic that can be expressed by our language. Method calls, method declarations, variable declarations, references, these are all candidates for AST nodes. A complete diagram of the nodes in our language is as follows:



The C++ that represents the above specifications is:

Listing of node.h:

```
#include <iostream>
#include <vector>
#include <llvm/Value.h>

class CodeGenContext;
class NStatement;
class NExpression;
class NVariableDeclaration;

typedef std::vector<NStatement*> StatementList;
typedef std::vector<NExpression*> ExpressionList;
typedef std::vector<NVariableDeclaration*> VariableList;

class Node {
public:
    virtual ~Node() {}
    virtual llvm::Value* codeGen(CodeGenContext& context) { }
};

class NExpression : public Node {
};

class NStatement : public Node {
};

class NInteger : public NExpression {
public:
    long long value;
    NInteger(long long value) : value(value) { }
    virtual llvm::Value* codeGen(CodeGenContext& context);
};

class NDouble : public NExpression {
public:
    double value;
    NDouble(double value) : value(value) { }
    virtual llvm::Value* codeGen(CodeGenContext& context);
};

class NIdentifier : public NExpression {
public:
    std::string name;
    NIdentifier(const std::string& name) : name(name) { }
    virtual llvm::Value* codeGen(CodeGenContext& context);
};

class NMethodCall : public NExpression {
public:
    const NIdentifier& id;
    ExpressionList arguments;
    NMethodCall(const NIdentifier& id, ExpressionList& arguments) :
        id(id), arguments(arguments) { }
    NMethodCall(const NIdentifier& id) : id(id) { }
    virtual llvm::Value* codeGen(CodeGenContext& context);
};

class NBinaryOperator : public NExpression {
public:
    int op;
    NExpression& lhs;
    NExpression& rhs;
    NBinaryOperator(NExpression& lhs, int op, NExpression& rhs) :
        lhs(lhs), rhs(rhs), op(op) { }
    virtual llvm::Value* codeGen(CodeGenContext& context);
};

class NAssignment : public NExpression {
public:
    NIdentifier& lhs;
    NExpression& rhs;
    NAssignment(NIdentifier& lhs, NExpression& rhs) :
        lhs(lhs), rhs(rhs) { }
    virtual llvm::Value* codeGen(CodeGenContext& context);
};

class NBlock : public NExpression {
public:
    StatementList statements;
    NBlock() { }
    virtual llvm::Value* codeGen(CodeGenContext& context);
};

class NExpressionStatement : public NStatement {
public:
    NExpression& expression;
    NExpressionStatement(NExpression& expression) :
        expression(expression) { }
    virtual llvm::Value* codeGen(CodeGenContext& context);
};

class NVariableDeclaration : public NStatement {
public:
    const NIdentifier& type;
    NIdentifier& id;
    NExpression *assignmentExpr;
    NVariableDeclaration(const NIdentifier& type, NIdentifier& id) :
        type(type), id(id) { }
    NVariableDeclaration(const NIdentifier& type, NIdentifier& id, NExpression *assignment
        type(type), id(id), assignmentExpr(assignmentExpr) { }
    virtual llvm::Value* codeGen(CodeGenContext& context);
};

class NFunctionDeclaration : public NStatement {
public:
    const NIdentifier& type;
    const NIdentifier& id;
    VariableList arguments;
    NBlock& block;
    NFunctionDeclaration(const NIdentifier& type, const NIdentifier& id,
            const VariableList& arguments, NBlock& block) :
        type(type), id(id), arguments(arguments), block(block) { }
    virtual llvm::Value* codeGen(CodeGenContext& context);
};
```

Again, fairly straightforward. We’re not using any getters or setters here, just public data members; there’s really no need for data hiding. Ignore the codeGen method for now. It will come in handy later, when we want to export our AST as LLVM bytecode.

Back to Bison

Moo. I mean, whatever Bison say. Here's a short tangent:

It takes a little while to wrap your head around the awesome of bison.
3:05 PM Sep 16th from twhirl
lsegal
Loren Segal

@lsegal what's to wrap your head around? It's tasty, it's lean... oh, you weren't talking about meat?
3:13 PM Sep 16th from web in reply to lsegal
avibryant
Avi Bryant

Anyway, here comes the most complex part and by the same token, the hardest to explain. It's not technically complicated, but I'd have to spend a while discussing the details of Bison's syntax, so let's take a lot of it for granted right now. Realize this, though: we'll be declaring the makeup of each of our valid statements and expressions (the ones representing the nodes we've defined above) using terminal and non-terminal symbols, basically like a BNF grammar. The syntax is also similar:

```
if_stmt : IF '(' condition ')' block { /* do stuff when this rule is encountered */ }
        | IF '(' condition ')'       { ... }
        ;
```

The above would define an if statement (*if* we supported it). The real difference is that with each grammar comes a set of actions (inside the braces) which are executed after it is recognized. This is done recursively over the symbols in leaf-to-root order, where each non terminal is eventually merged into one big tree. The "$$" symbol you will be seeing represents the current root node of each leaf. Furthermore, "$1" represents the leaf for the 1st symbol in the rule. In the above example, the "$$" we assigned from our "condition" rule would be available as "$3" in our "if_stmt" rule. It might start becoming clear how our AST ties into this. We will basically be assigning a node to "$$" at each of our rules which will eventually be merged into our final AST. If not, don't worry. The Bison part of our toy language is again, the most complex portion of our language. It might take a while to sink in. Besides, you haven't even seen the code yet, so, here it is:

Listing of parser.y:

```
%{
    #include "node.h"
    NBlock *programBlock; /* the top level root node of our final AST */

    extern int yylex();
    void yyerror(const char *s) { printf("ERROR: %sn", s); }
%}

/* Represents the many different ways we can access our data */
%union {
    Node *node;
    NBlock *block;
    NExpression *expr;
    NStatement *stmt;
    NIdentifier *ident;
    NVariableDeclaration *var_decl;
    std::vector<NVariableDeclaration*> *varvec;
    std::vector<NExpression*> *exprvec;
    std::string *string;
    int token;
}

/* Define our terminal symbols (tokens). This should
   match our tokens.l lex file. We also define the node type
   they represent.
 */
%token <string> TIDENTIFIER TINTEGER TDOUBLE
%token <token> TCEQ TCNE TCLT TCLE TCGT TCGE TEQUAL
%token <token> TLPAREN TRPAREN TLBRACE TRBRACE TCOMMA TDOT
%token <token> TPLUS TMINUS TMUL TDIV

/* Define the type of node our nonterminal symbols represent.
   The types refer to the %union declaration above. Ex: when
   we call an ident (defined by union type ident) we are really
   calling an (NIdentifier*). It makes the compiler happy.
 */
%type <ident> ident
%type <expr> numeric expr
%type <varvec> func_decl_args
%type <exprvec> call_args
%type <block> program stmts block
%type <stmt> stmt var_decl func_decl
%type <token> comparison

/* Operator precedence for mathematical operators */
%left TPLUS TMINUS
%left TMUL TDIV
```

```
%start program

%%

program : stmts { programBlock = $1; }
        ;

stmts : stmt { $$ = new NBlock(); $$->statements.push_back($<stmt>1); }
      | stmts stmt { $1->statements.push_back($<stmt>2); }
      ;

stmt : var_decl | func_decl
     | expr { $$ = new NExpressionStatement(*$1); }
     ;

block : TLBRACE stmts TRBRACE { $$ = $2; }
      | TLBRACE TRBRACE { $$ = new NBlock(); }
      ;

var_decl : ident ident { $$ = new NVariableDeclaration(*$1, *$2); }
         | ident ident TEQUAL expr { $$ = new NVariableDeclaration(*$1, *$2, $4); }
         ;

func_decl : ident ident TLPAREN func_decl_args TRPAREN block
            { $$ = new NFunctionDeclaration(*$1, *$2, *$4, *$6); delete $4; }
          ;

func_decl_args : /*blank*/  { $$ = new VariableList(); }
          | var_decl { $$ = new VariableList(); $$->push_back($<var_decl>1); }
          | func_decl_args TCOMMA var_decl { $1->push_back($<var_decl>3); }
          ;

ident : TIDENTIFIER { $$ = new NIdentifier(*$1); delete $1; }
      ;

numeric : TINTEGER { $$ = new NInteger(atol($1->c_str())); delete $1; }
        | TDOUBLE { $$ = new NDouble(atof($1->c_str())); delete $1; }
        ;

expr : ident TEQUAL expr { $$ = new NAssignment(*$<ident>1, *$3); }
     | ident TLPAREN call_args TRPAREN { $$ = new NMethodCall(*$1, *$3); delete $3; }
     | ident { $<ident>$ = $1; }
     | numeric
     | expr comparison expr { $$ = new NBinaryOperator(*$1, $2, *$3); }
     | TLPAREN expr TRPAREN { $$ = $2; }
     ;

call_args : /*blank*/  { $$ = new ExpressionList(); }
          | expr { $$ = new ExpressionList(); $$->push_back($1); }
          | call_args TCOMMA expr  { $1->push_back($3); }
          ;

comparison : TCEQ | TCNE | TCLT | TCLE | TCGT | TCGE
           | TPLUS | TMINUS | TMUL | TDIV
           ;

%%
```

Generating Our Code

So we have our "tokens.l" file for Flex and our "parser.y" file for Bison. To generate our C++ source files from these definition files we need to pass them through the tools. Note that Bison will also be creating a "parser.hpp" header file for Flex; it does this because of the `–d` switch, which separates our token declarations from the source so we can include and use those tokens elsewhere. The following commands should create our parser.cpp, parser.hpp and tokens.cpp source files.

```
$ bison -d -o parser.cpp parser.y
$ lex -o tokens.cpp tokens.l
```

If everything went well we should now have 2 out of 3 parts of our compiler. If you want to test this, create a short main function in a main.cpp file:

Listing of main.cpp:

```
#include <iostream>
#include "node.h"
extern NBlock* programBlock;
extern int yyparse();

int main(int argc, char **argv)
{
    yyparse();
    std::cout << programBlock << std::endl;
    return 0;
}
```

You can then compile your source files:

```
$ g++ -o parser parser.cpp tokens.cpp main.cpp
```

You will need to have installed LLVM by now for the `llvm::Value` reference in "node.h". If you don't want to go that far just yet, you can comment out the codeGen() methods in node.h to test just your lexer/parser combo.

If all goes well you should now have a "parser" binary that takes source in stdin and prints out a hopefully nonzero address representing the root node of our AST in memory.

Step 3. Assembling the AST with LLVM

The next step in a compiler is to naturally take this AST and turn it into machine code. This means converting each semantic node into the equivalent machine instruction. LLVM makes this very easy for us, because it abstracts the actual instructions to something that is similar to an AST. This means all we're really doing is translating from one AST to another.

You can imagine that this process will involve us walking over our AST from our root node, and for each node we emit bytecode. This is where the `codeGen` method that we defined for our nodes comes in handy. For example, when we visit an `NBlock` node (semantically representing a collection of statements), we call `codeGen` on each statement in the list. It looks like this:

```
Value* NBlock::codeGen(CodeGenContext& context)
{
    StatementList::const_iterator it;
    Value *last = NULL;
    for (it = statements.begin(); it != statements.end(); it++) {
        std::cout << "Generating code for " << typeid(**it).name() << std::endl;
        last = (**it).codeGen(context);
    }
    std::cout << "Creating block" << std::endl;
    return last;
}
```

We implement this method for all of our nodes, then call it as we go down the tree, implicitly walking us through our entire AST. We keep a new `CodeGenContext` class around to tell us where to emit our bytecode.

A Big Caveat About LLVM

One of the downsides of LLVM is that it's *really* hard to find useful documentation. Their online tutorial and other such docs are wildly out of date, and there's barely any information for the C++ API unless you really dig for it. If you installed LLVM on your own, check the 'docs' as it has "more" up to date documentation.

I've found the best way to learn LLVM is by example. There are some quick examples of programmatically generating bytecode in the 'examples' directory of the LLVM archive as well, and there's also the LLVM live demo site which can emit C++ API code for a C program as input. This is a great way to find out what instructions something like "int x = 5;" will emit. I used the demo tool to implement most of the nodes.

Without further ado, I'll be listing both the codegen.h and codegen.cpp files below.

Listing of codegen.h:

```
#include <stack>
#include <llvm/Module.h>
#include <llvm/Function.h>
#include <llvm/Type.h>
#include <llvm/DerivedTypes.h>
#include <llvm/LLVMContext.h>
#include <llvm/PassManager.h>
#include <llvm/Instructions.h>
#include <llvm/CallingConv.h>
#include <llvm/Bitcode/ReaderWriter.h>
#include <llvm/Analysis/Verifier.h>
#include <llvm/Assembly/PrintModulePass.h>
#include <llvm/Support/IRBuilder.h>
#include <llvm/ModuleProvider.h>
#include <llvm/Target/TargetSelect.h>
#include <llvm/ExecutionEngine/GenericValue.h>
```

```
#include <llvm/ExecutionEngine/JIT.h>
#include <llvm/Support/raw_ostream.h>

using namespace llvm;

class NBlock;

class CodeGenBlock {
public:
    BasicBlock *block;
    std::map<std::string, Value*> locals;
};

class CodeGenContext {
    std::stack<CodeGenBlock *> blocks;
    Function *mainFunction;

public:
    Module *module;
    CodeGenContext() { module = new Module("main", getGlobalContext()); }

    void generateCode(NBlock& root);
    GenericValue runCode();
    std::map<std::string, Value*>& locals() { return blocks.top()->locals; }
    BasicBlock *currentBlock() { return blocks.top()->block; }
    void pushBlock(BasicBlock *block) { blocks.push(new CodeGenBlock()); blocks.top()->blo
    void popBlock() { CodeGenBlock *top = blocks.top(); blocks.pop(); delete top; }
};
```

Listing of codegen.cpp

```
#include "node.h"
#include "codegen.h"
#include "parser.hpp"

using namespace std;

/* Compile the AST into a module */
void CodeGenContext::generateCode(NBlock& root)
{
    std::cout << "Generating code...n";

    /* Create the top level interpreter function to call as entry */
    vector<const Type*> argTypes;
    FunctionType *ftype = FunctionType::get(Type::getVoidTy(getGlobalContext()), argTypes,
    mainFunction = Function::Create(ftype, GlobalValue::InternalLinkage, "main", module);
    BasicBlock *bblock = BasicBlock::Create(getGlobalContext(), "entry", mainFunction, 0);

    /* Push a new variable/block context */
    pushBlock(bblock);
    root.codeGen(*this); /* emit bytecode for the toplevel block */
    ReturnInst::Create(getGlobalContext(), bblock);
    popBlock();

    /* Print the bytecode in a human-readable format
       to see if our program compiled properly
     */
    std::cout << "Code is generated.n";
    PassManager pm;
    pm.add(createPrintModulePass(&outs()));
    pm.run(*module);
}

/* Executes the AST by running the main function */
GenericValue CodeGenContext::runCode() {
    std::cout << "Running code...n";
    ExistingModuleProvider *mp = new ExistingModuleProvider(module);
    ExecutionEngine *ee = ExecutionEngine::create(mp, false);
    vector<GenericValue> noargs;
    GenericValue v = ee->runFunction(mainFunction, noargs);
    std::cout << "Code was run.n";
    return v;
}

/* Returns an LLVM type based on the identifier */
static const Type *typeOf(const NIdentifier& type)
{
    if (type.name.compare("int") == 0) {
        return Type::getInt64Ty(getGlobalContext());
    }
    else if (type.name.compare("double") == 0) {
        return Type::getDoubleTy(getGlobalContext());
    }
    return Type::getVoidTy(getGlobalContext());
}

/* -- Code Generation -- */

Value* NInteger::codeGen(CodeGenContext& context)
{
    std::cout << "Creating integer: " << value << std::endl;
    return ConstantInt::get(Type::getInt64Ty(getGlobalContext()), value, true);
}

Value* NDouble::codeGen(CodeGenContext& context)
{
    std::cout << "Creating double: " << value << std::endl;
    return ConstantFP::get(Type::getDoubleTy(getGlobalContext()), value);
}

Value* NIdentifier::codeGen(CodeGenContext& context)
```

```
{
    std::cout << "Creating identifier reference: " << name << std::endl;
    if (context.locals().find(name) == context.locals().end()) {
        std::cerr << "undeclared variable " << name << std::endl;
        return NULL;
    }
    return new LoadInst(context.locals()[name], "", false, context.currentBlock());
}

Value* NMethodCall::codeGen(CodeGenContext& context)
{
    Function *function = context.module->getFunction(id.name.c_str());
    if (function == NULL) {
        std::cerr << "no such function " << id.name << std::endl;
    }
    std::vector<Value*> args;
    ExpressionList::const_iterator it;
    for (it = arguments.begin(); it != arguments.end(); it++) {
        args.push_back((**it).codeGen(context));
    }
    CallInst *call = CallInst::Create(function, args.begin(), args.end(), "", context.curr
    std::cout << "Creating method call: " << id.name << std::endl;
    return call;
}

Value* NBinaryOperator::codeGen(CodeGenContext& context)
{
    std::cout << "Creating binary operation " << op << std::endl;
    Instruction::BinaryOps instr;
    switch (op) {
        case TPLUS:     instr = Instruction::Add; goto math;
        case TMINUS:    instr = Instruction::Sub; goto math;
        case TMUL:      instr = Instruction::Mul; goto math;
        case TDIV:      instr = Instruction::SDiv; goto math;

        /* TODO comparison */
    }

    return NULL;
math:
    return BinaryOperator::Create(instr, lhs.codeGen(context),
        rhs.codeGen(context), "", context.currentBlock());
}

Value* NAssignment::codeGen(CodeGenContext& context)
{
    std::cout << "Creating assignment for " << lhs.name << std::endl;
    if (context.locals().find(lhs.name) == context.locals().end()) {
        std::cerr << "undeclared variable " << lhs.name << std::endl;
        return NULL;
    }
    return new StoreInst(rhs.codeGen(context), context.locals()[lhs.name], false, context.
}

Value* NBlock::codeGen(CodeGenContext& context)
{
    StatementList::const_iterator it;
    Value *last = NULL;
    for (it = statements.begin(); it != statements.end(); it++) {
        std::cout << "Generating code for " << typeid(**it).name() << std::endl;
        last = (**it).codeGen(context);
    }
    std::cout << "Creating block" << std::endl;
    return last;
}

Value* NExpressionStatement::codeGen(CodeGenContext& context)
{
    std::cout << "Generating code for " << typeid(expression).name() << std::endl;
    return expression.codeGen(context);
}

Value* NVariableDeclaration::codeGen(CodeGenContext& context)
{
    std::cout << "Creating variable declaration " << type.name << " " << id.name << std::e
    AllocaInst *alloc = new AllocaInst(typeOf(type), id.name.c_str(), context.currentBlock
    context.locals()[id.name] = alloc;
    if (assignmentExpr != NULL) {
        NAssignment assn(id, *assignmentExpr);
        assn.codeGen(context);
    }
    return alloc;
}

Value* NFunctionDeclaration::codeGen(CodeGenContext& context)
{
    vector<const Type*> argTypes;
    VariableList::const_iterator it;
    for (it = arguments.begin(); it != arguments.end(); it++) {
        argTypes.push_back(typeOf((**it).type));
    }
    FunctionType *ftype = FunctionType::get(typeOf(type), argTypes, false);
    Function *function = Function::Create(ftype, GlobalValue::InternalLinkage, id.name.c_s
    BasicBlock *bblock = BasicBlock::Create(getGlobalContext(), "entry", function, 0);

    context.pushBlock(bblock);

    for (it = arguments.begin(); it != arguments.end(); it++) {
        (**it).codeGen(context);
    }

    block.codeGen(context);
    ReturnInst::Create(getGlobalContext(), bblock);
    context.popBlock();
    std::cout << "Creating function: " << id.name << std::endl;
    return function;
}
```

There is certainly a great deal to take in here, however this is the part where you should start exploring on your own. I only have a couple of notes:

- We use a “stack” of blocks in our CodeGenContext class to keep the last entered block (because instructions are added to blocks)
- We also use this stack to keep a symbol table of the local variables in each block.
- Our toy program only knows about variables in its own scope. To support the idea of “global” contexts you’d need to search upwards through each block in our stack until you found a match to the symbol (rather than simply searching the top symbol table).
- Before entering a block we should push the block and when leaving it we should pop it.

The rest of the details are all related to LLVM, and again, I’m hardly an expert on that subject. But at this point, we have all of the code we need to compile our toy language and watch it run.

Building Our Toy Language

We have the code, but how to we build it? LLVM can be complicated to link, fortunately if you installed LLVM you also got an `llvm-config` binary which returns all of the compiler/linker flags you need.

We also need to update our main.cpp file from earlier to actually compile and run our code, this time:

Listing of main.cpp:

```
#include < iostream>
#include "codegen.h"
#include "node.h"

using namespace std;

extern int yyparse();
extern NBlock* programBlock;

int main(int argc, char **argv)
{
    yyparse();
    std::cout << programBlock << std::endl;

    CodeGenContext context;
    context.generateCode(*programBlock);
    context.runCode();

    return 0;
}
```

Now all we need to do is compile:

```
g++ -o parser `llvm-config --libs core jit native --cxxflags --ldflags` *.cpp
```

You can also grab this Makefile to make things easier:

Listing of Makefile:

```
all: parser

clean:
    rm parser.cpp parser.hpp parser tokens.cpp

parser.cpp: parser.y
    bison -d -o $@ $^

parser.hpp: parser.cpp

tokens.cpp: tokens.l parser.hpp
    lex -o $@ $^

parser: parser.cpp codegen.cpp main.cpp tokens.cpp
    g++ -o $@ `llvm-config --libs core jit native --cxxflags --ldflags` *.cpp
```

Running Our Toy Language

Hopefully everything compiled properly. At this point you should have your parser binary that spits out code. Play with it. Remember our canonical example? Let's see what our program does.

```
$ echo 'int do_math(int a) { int x = a * 5 + 3 } do_math(10)' | ./parser
0x100a00f10
Generating code...
Generating code for 20NFunctionDeclaration
Creating variable declaration int a
Generating code for 20NVariableDeclaration
Creating variable declaration int x
Creating assignment for x
Creating binary operation 276
Creating binary operation 274
Creating integer: 3
Creating integer: 5
Creating identifier reference: a
Creating block
Creating function: do_math
Generating code for 20NExpressionStatement
Generating code for 11NMethodCall
Creating integer: 10
Creating method call: do_math
Creating block
Code is generated.
; ModuleID = 'main'

define internal void @main() {
entry:
    %0 = call i64 @do_math(i64 10)      ;  [#uses=0]
    ret void
}

define internal i64 @do_math(i64) {
entry:
    %a = alloca i64     ;  [#uses=1]
    %x = alloca i64     ;  [#uses=1]
    %1 = add i64 5, 3   ;  [#uses=1]
    %2 = load i64* %a   ;  [#uses=1]
    %3 = mul i64 %2, %1 ;  [#uses=1]
    store i64 %3, i64* %x
    ret void
}
Running code...
Code was run.
```

Isn't that pretty cool? I'll admit it's probably less satisfying to get something running when you just copy paste from a how to guide, but getting this running should still be interesting for you. Plus, and this is the best part, this isn't the end of the guide, it's just the beginning, because this is the part where *you* start tacking on crazy features to this lexer, parser and assembler to create a language that you can call your own. You have all the building blocks, and you should have a basic idea of how to continue on.

The code, by the way, is available on Github here. I avoided saying this because the code was not the point of the article, but rather going over the code in detail was. If you're just going to grab the code it sort of defeats the purpose of writing your own compiler.

I realize this is a huge article, and there are probably mistakes, so if you run into problems, feel free to send me an email and I'll make adjustments. If you want to share some info, feel free to do that as well.

Questions? Comments? Follow me on Twitter (@lsegal) or email me.