ANTLR 3 / Examples

LLVM

Created by TerenceP
Last updated: Dec 29, 2009 • Version comment • 16 min read

I wanted to build something using LLVM in my new book, Language Implementation Patterns, but I ran out of room. I left a link to this page in the book instead. **Note**: *This is not text cut from the book; it's just some text I slapped together around a nice clean example of how to use ANTLR+StringTemplate with LLVM to build a compiler.*

LLVM is one of the best pieces of open-source software available; check it out!

Introduction

Compilers have two main components called the front-end and the back-end. The front-end parses the input language, builds a symbol table, performs semantic analysis, and emits a highly processed version of the input called an intermediate representation (IR). The back-end creates even more data structures such as control-flow graphs, optimizes the IR, maps IR computation values to machine registers, and emits assembly language as text. An assembler does the final pass to convert the assembly code to raw binary machine code.

The front-end is usually less challenging than the back-end, though implementing the intricate rules of semantic analysis can be demanding and tedious. (C++ might be the only language whose front-end rivals the back-end in complexity.) The overall task of building a compiler then is much more manageable if we can avoid building a back-end by reusing some existing software.

Either we have to reuse an existing compiler or we have to use a compiler construction library. The simplest way is to translate a new language to an existing language for which a compiler already exists. This is the approach taken by the original C++ compiler took. `cfront` translated C++ to C, which worked well because C++'s core semantics are the same as C's semantics. Unfortunately, the semantics are not always close enough for a straightforward translation. Consider translating a few lines of Java code to C. This simple method and variable definition:

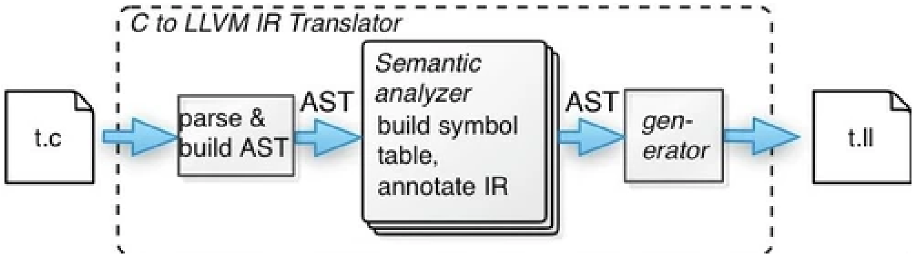
```
void f() {
    ArrayList a = new ArrayList();
    ...
}
```

is roughly the same as this C function:

```
void f() {
    struct ArrayList *a = calloc(1, sizeof(struct ArrayList));
    ...
}
```

Looks great until you realize that C does not have automatic garbage collection. Because `f()` could pass a to another function, we cannot fake it by simply putting `free(a);` at the end of `f()`. C is just plain uncooperative when it comes to garbage collection. Compiling Java to native code means we need to translate it to something lower level than C with more control over the machine.

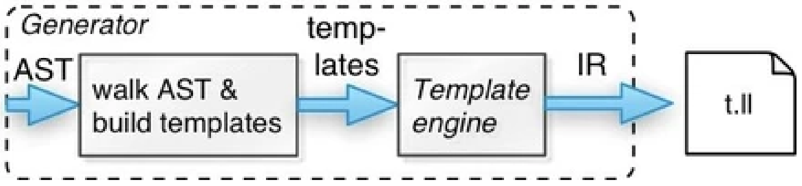
Instead of translating to a high-level language, we can translate source code to the IR of a compiler toolkit. Until recently, the toolkits for building compiler back-ends have either been insufficiently general, unstable, or difficult to use. The *Low Level Virtual Machine* (LLVM), `http://www.llvm.org`, changes all of that. LLVM started at the University of Illinois Urbana-Champaign but has grown significantly and is now partially supported by Apple Computer. Here's what the overall dataflow will look like in our compiler:



We're going to read C subset source code and build an AST from it (See Chapter 4 in Language Implementation Patterns). This example doesn't have it, but normally we'd build a symbol table (See Chapter 7) and then do semantic analysis (See Chapter 8) by walking over the tree (See Chapter 5).

Once we have the .ll file (the IR), LLVM can't take over to generate machine code. LLVM's IR is somewhere between assembly code and a C-like high-level language. LLVM's IR is really a virtual instruction set for a RISC-like virtual machine. LLVM can both interpret the IR and compile it down to highly-optimized assembly code.

The book describes the semantic analysis stuff pretty well, so let's focus on walking ASTs to generate LLVM's intermediate representation:



Our generator is a tree grammar (a formalization of the handbuilt visitor) that yields a large tree of templates. Each template represents a single translated chunk of the input. As we walk the tree, we'll build up a hierarchy by embedding the templates returned for one rule in the template result of another.

A quick look at LLVM's IR

The IR doc will be your friend.

We're going to use LLVM's text-based representation of its IR rather than building up objects in memory using Java API. When there's a nice domain-specific language (DSL) available, I always prefer it over using a library. The details of using library are usually harder to learn than the DSL and you have to learn a new one for each language. The text-based IR is language neutral sense that any language can generate that text.

Let's start with a trivial example. Here's how we can represent statement `print 99` in the IR:

```
; define string constant "%d\n"
@s = internal constant [4 x i8] c"%d\0A\00"
; declare C function: extern int printf(char *, ...)
declare i32 @printf(i8 *, ...)
```

```
; define main entry point like main() in a C program
define i32 @main() {
    ; get the address of 1st char of s: &s[0]
    ; reuseable for all printf calls
    %ps = getelementptr [4 x i8]* @s, i64 0, i64 0
    ; get 99 into a register; t0 = 99+0 (LLVM has no load int instruction)
    %t0 = add i32 99,0
    ; call printf with "%d\n" and t0 as arguments
    call i32 (i8 *, ...)* @printf(i8* %ps, i32 %t0)
    ret i32 0 ; return 0
}
```

To boil that IR down to machine code and execute, first get the output into a file, `t.ll`. Then, run the LLVM tool chain:

```
$ llvm-as -f t.ll   # assemble IR to bitcode file t.bc
$ llc -f t.bc       # compile instructions to assembly code file t.s (x86 on my box)
$ llvm-gcc -o t t.s # assemble x86 down to machine code and link (or use plain gcc)
$ ./t               # execute the program t in current directory
99                  # the output
$
```

LLVM also has an interpreter that you can invoke on the bitcodes (in LLVM parlance) without having to use `llc` and `gcc` execute the program:

```
$ lli t.bc
99
$
```

To learn more about the IR, you should use the llvm-base gcc C compiler. You can type in some C code and then ask the compiler to generate IR instead of assembly code:

```
llvm-gcc -emit-llvm -S -c mycode.c
```

The generated `mycode.s` file contains LLVM IR (not assembly code as you'd expect with just -S option) that shows one way to represent the statements in the C program.

Here are a few things to keep in mind when reading the documentation
for LLVM's IR and generating IR:

- llvm calls .ll files modules.
- global variables start with @, but registers and local variables start with %.
- declarations for external funcs go first in a module then globals then local functions.
- global variables define pointer values not values.
- start all basic blocks with a label (functions automatically get one).

C subset to compile

For this project, we'll build a compiler for a simple C subset. The language supports arrays but very few operators and only integer types. The features are:

- int type
- functions with void or int return type and multiple arguments
- if statements
- while statements
- return statements
- printf("s", n) method; allows simplified version with format string and single integer argument. This is the only place a string is allowed (i.e., you can't use it in a general expression nor as an argument to a user-defined function)
- operators: +, -, ==, !=
- arrays
- globals and locals
- no pointers. LLVM IR uses them though.

Mapping from source language to IR

As with any translation project, we must start by identifying precisely what input translates to what output. I usually start out with a few small, real examples and then abstract from there what the individual statements and expressions translated to.

Looking at the simple print example from above, we find that all of the strings in print statements need to appear first in this form (for ith string s):

```
@s<i> = internal constant [<s.lengthInBytes> x i8] c"<s>"
```

Next, consider what the actual `printf("%d\n", expression)` function call translates to:

```
; get the address of 1st char of s: &s[0]
; reuseable for all printf calls
%ps = getelementptr [4 x i8]* @s, i64 0, i64 0
call i32 (i8 *, ...)* @printf(i8* %ps, i32 %t) ; t is reg with expression to print
```

The getelementptr instruction can be pretty confusing; if I were you, I would start out by simply memorizing that we need that instruction to reference a string. (You can in more deeply into it by reading The Often Misunderstood GEP Instruction.)

Here's a list of some of the mappings we'll need:

Source	**LLVM IR**
int constant `n`	`%t = add i32 n,0`
`int x`	`@x = global i32 0 ; global var`
`int a[n]`	`@a = global [n x i32] zeroinitializer ; global array`
`a`	`%t = load i32* %a`
`a[i]`	`i32* getelementptr ([10 x i32]* @a, i32 0, i32 i)`

Source	LLVM IR

`void f()`	`define void @f() {`
	`    ret void`
	`}`

`int i`	`%i = alloca i32`
`int a[10]`	`%a = alloca [10 x i32] ; allocate local array`

`void f(int a, int b\[10]) {`
`    x =`
`}`
	The arguments are constant values/references, so we can't store values into the arguments directly in the IR. We have to create local space and store the argument into at first. Don't worry though LLVM will optimize away all of the alloca and store instructions.
	`define void @f(i32 %a_arg, i32* %b_arg) nounwind {`
	`; alloca, stores can be optimized away`
	`; make space for arg a and ptr to int (array arg b)`
	`    %a = alloca i32`
	`    %b = alloca i32*`
	`; store constant arguments into memory locations`
	`    store i32 %a_arg, i32* %a`
	`    store i32* %b_arg, i32** %b`
	`;       ... can mess with args now ...`
	`        %t0 = add i32 33, 0 ; t0 = 33`
	`        store i32 %t0, i32* %a       ; a = t0`
	`        %t1 = load i32* %a           ; t1 = a;`
	`    ret void`
	`}`

`int f() {return 1;}`	`define i32 @f() nounwind {`
	`    %rv = add i32 1, 0 ; gets optimized`
	`    ret i32 %rv`
	`}`

Here are a few complete examples. First, factorial:

```
int f(int n) {
    if ( n<=1) return 1;
    return n*f(n-1);
}
```

```
declare i32 @printf(i8 *, ...)
define i32 @fact(i32 %n_arg) {
```

```
; init arg(s): n
    %n = alloca i32
    store i32 %n_arg, i32* %n
; if ( n<=1) return 1;
    %r1 = load i32* %n
    %r2 = add i32 1, 0
    %r3 = icmp sle i32 %r1, %r2
    br i1 %r3, label %true5, label %false5
    true5:
; return 1;
    %r4 = add i32 1, 0
    ret i32 %r4
    false5:
; return n*fact(n-1);
    %r6 = load i32* %n
; fact(n-1)
    %r7 = load i32* %n
    %r8 = add i32 1, 0
    %r9 = sub i32 %r7, %r8
    %r10 = call i32(i32)* @fact(i32 %r9)
    %r11 = mul i32 %r6, %r10
    ret i32 %r11
    ret i32 0 ; auto-generated and eliminated by LLVM
}
```

And here's an example that messes with a local integer and a global array.

```
int a[10]; // must be a constant size
void f() { int x; x=2; a[x] = 99; x = a[2]; printf("%d\n", x); }
```

```
declare i32 @printf(i8 *, ...)
@s1 = internal constant [4 x i8] c"%d\0A\00"
; int a[10];
@a = global [10 x i32] zeroinitializer
define void @f() {
; init arg(s):
    ; int x;
    %x = alloca i32
; x=2;
    %r2 = add i32 2, 0
    store i32 %r2, i32* %x
; a[x] = 99;
    %r4 = add i32 99, 0
    %r3 = load i32* %x
; array_ptr.reg=5
    %r5 = bitcast [10 x i32]* @a to i32*
    %r6 = getelementptr i32* %r5, i32 %r3
    store i32 %r4, i32* %r6
; x = a[2];
    %r7 = add i32 2, 0
; array_ptr.reg=9
    %r9 = bitcast [10 x i32]* @a to i32*
    %r10 = getelementptr i32* %r9, i32 %r7
    %r8 = load i32* %r10
    store i32 %r8, i32* %x
; printf(%d\n, x);
    %r11 = getelementptr [4 x i8]* @s1, i32 0, i32 0
    %r12 = load i32* %x
    call i32 (i8*, ...)* @printf(i8* %r11, i32 %r12)
    ret void
}
```

Ok, it's time to look at the sample solution.

A Strategy for using ANTLR + StringTemplate + LLVM

Translating a high-level programming language down to LLVM's SSA IR means recognizing the various subphrases of an input sentence and emitting IR instructions. When combined in the correct order, the instructions together implement the intended meaning of the input sentence. This solution shows you how to combine ANTLR and StringTemplate to generate IR that LLVM can compile down to machine code.

The basic strategy is to build an AST from the input using a parser and then walk it twice: once to build a simple table and again to emit IR instructions. Here is the critical path in the main program that shows the complete process

```
// CREATE LEXER/PARSER THAT CREATES AST FROM INPUT
CLexer lexer = new CLexer(new ANTLRInputStream(input));
TokenRewriteStream tokens = new TokenRewriteStream(lexer);
CParser parser = new CParser(tokens);
parser.setTreeAdaptor(cTreeAdaptor);
CParser.translation_unit_return ret = parser.translation_unit();
CommonTree t = (CommonTree)ret.getTree();
System.out.println("; "+t.toStringTree());

// MAKE SYM TAB
SymbolTable symtab = new SymbolTable();

// LOAD TEMPLATES (via classpath)
ClassLoader cl = CC.class.getClassLoader();
InputStream in = cl.getResourceAsStream(templatesFilename);
Reader rd = new InputStreamReader(in);
StringTemplateGroup templates = new StringTemplateGroup(rd);
rd.close();

CommonTreeNodeStream nodes = new CommonTreeNodeStream(cTreeAdaptor, t);
nodes.setTokenStream(tokens);

// DEFINE/RESOLVE SYMBOLS
DefRef def = new DefRef(nodes, symtab); // use custom constructor
```

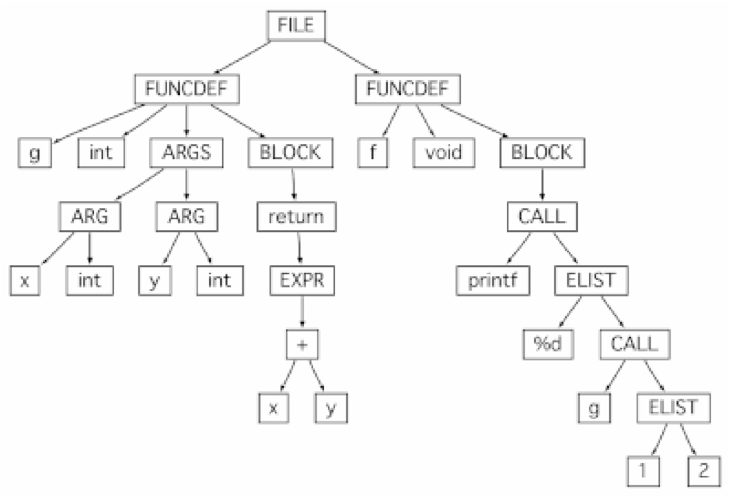
```
def.downup(t); // trigger symtab actions upon certain subtrees
//System.out.println("globals: "+symtab.globals);

// GENERATE CODE
nodes.reset();
Gen walker = new Gen(nodes, symtab);
walker.setTemplateLib(templates);
Gen.translation_unit_return ret2 = walker.translation_unit();

// EMIT IR
// uncomment next line to learn which template emits what output
//templates.emitDebugStartStopStrings(true);
String output = ret2.getTemplate().toString();
System.out.println(output);
```

Building the ASTs

The C.g parser/lexer builds ASTs (not parse trees, which contain the rules that the parser used to match the input) that look like this:



ANTLR's DOTTreeGenerator class generated that DOT-based tree from following input:

```
int g(int x, int y) { return x+y; }
void f() { printf("%d\n", g(1,2)); }
```

In text form, the tree looks like:

```
(FILE
  (FUNCDEF g int (ARGS (ARG x int) (ARG y int))
    (BLOCK (return (EXPR (+ x y))))
  )
  (FUNCDEF f void
    (BLOCK (CALL printf (ELIST %d\n (CALL g (ELIST 1 2)))))
  )
)
```

Building the symbol table

Our next goal is track defined in the program and then to annotate the AST with those symbol pointers. The DefRef.g tree filter specifies a number of tree pattern / action pairs. For each subtree that matches the pattern, ANTLR will execute the associated action (it's like AWK for trees). There are two special rules that specify which rules to apply on the way down in which to apply on the way back up:

```
topdown
    :   enterBlock
    |   enterFunction
    |   varDeclaration
    ;

bottomup
    :   exitBlock
    |   exitFunction
    |   idref
    |   call
    ;
```

For example, as we enter code block subtrees on the way down, we want to push a new scope on the scope stack. After we process the code block subtree, we want to pop that scope off on the way back up:

```
enterBlock
    :   BLOCK {currentScope = new LocalScope(currentScope);}// push scope
    ;
exitBlock
    :   BLOCK
        {
        // System.out.println("locals: "+currentScope);
        currentScope = currentScope.getEnclosingScope();    // pop scope
        }
    ;
```

The top of the scope stack is the current scope, so when we see a variable declaration subtree, we can simply define a VariableSymbol in that scope:

```
varDeclaration // global, parameter, or local variable
    :   ^((VARDEF|EXTERNVAR|ARG) ID type_tree)
        {
        // System.out.println("line "+$ID.getLine()+": def "+$ID.text+
        //                    " type "+$type_tree.type);
        VariableSymbol vs = new VariableSymbol($ID.text,$type_tree.type);
        currentScope.define(vs);
        $ID.symbol = vs; // annotate the tree with the symbol pointer
        }
    ;
```

We also to have to make sure that symbol references get resolved as well (and we annotate the AST):

```
call:   ^(CALL ID .)
```

```
        {
        Symbol s = currentScope.resolve($ID.text);
        $ID.symbol = s;
        // System.out.println("line "+$ID.getLine()+": call "+s);
        }
    ;
```

The ANTLR tree pattern matchers are really handy because it lets us focus our we care about, rather than having to build an entire visitor or tree grammar.

I'll let you look at the source code for the symbol table itself because it's fairly straightforward. To really learn how these things work, see Chapter 7 in Language Implementation Patterns.

IR generation

The final step is to generate IR by walking the AST. Here, we need a full visitor to walk the tree because we have something to do for each note. We'll use a tree grammar, Gen.g, instead of building a visitor by hand.

As the final tree walking phase recognizes the input substructures represented by the AST, it creates and combines StringTemplate (ST) instances. The template returned by the start rule of the tree grammar is the overall file template. The tree grammar only cares about the name and attributes we inject into the templates. It doesn't depend on what's inside the templates. To change the output language, then, all we have to do is swap out one group of templates for another. You'll see both llvm.stg and C.stg template group files. **The same generator can emit LLVM IR or C code depending on which templates we use.** Here's how the main program launches the code generation phase:

```
// GENERATE CODE
nodes.reset(); // rewind tree node stream to walk again
Gen walker = new Gen(nodes, symtab);
walker.setTemplateLib(templates); // set templates to use!
walker.translation_unit();        // walk to create template hierarchy
```

Notice as you read the source code in Gen.g, that there are no output literals in the tree grammar. Everything dealing with the output language is encapsulated in the templates. Conversely, there's no model logic in the templates. This model-view separation is the linchpin behind building retargetable code generators.

The highest level rule collects a list of the templates returned by the declaration rule using the += operator:

```
translation_unit
    :   ^(FILE d+=external_declaration+)
        -> file(decls={$d}, strings={strings})
    ;
```

where `strings` is a global list of string constants needed by `printf` references in the entire input.

The `output=template` grammar option makes each rule in the grammar automatically return a template object. The `file` identifier after the `->` operator indicates the name of a template to create. We inject attributes via the assignments in `file`'s argument list. Our output file needs the list of declarations and string constants:

```
file(decls, strings) ::= <<
declare i32 @printf(i8 *, ...)
<strings:{s | @s<i> = internal constant <string_type(s)> c"<s>"}; separator="\n">
<decls; separator="\n">
```

```
>>
```

The `strings:{s ...}` notation is a map or lambda that applies the template to each string s in `strings`.

Turning to the expression rules, now, let's make a convention: each template contains code to compute a subexpression left in a particular IR register. We'll inject the register number as attribute `reg`. Each expression rule in the tree grammar returns the template generated for the associated subtree. Other templates can access the register holding the result by accessing the `reg` attribute. For example, to generate code for an assignment statement we need to access the register containing the result of the right hand side (to store it into the identifier on the left hand side). Here's the template:

```
assign(id, rhs, descr) ::= <<
; <descr>
<rhs>
store i32 %r<rhs.reg>, i32* %<id>
>>
```

The `<rhs.reg>` template expression accesses the `reg` attribute of the template associated with the right hand side. The tree grammar rule matching assignments injects the template returned from rule `expression` into the `assign` template instance:

```
statement:
    ...
    |    ^('=' ID expression) -> assign(id={$ID.text}, rhs={$expression.st})
    ...
```

Take a look at the C.stg template group. Using `-templates C.stg` commandline option will emits C code instead of LLVM IR. To see how it works, take a look at the sign to see how it differs from the IR version:

```
assign(id, rhs, descr) ::= "<id> = <rhs>;"
```

The "interface" (template name and argument list) is the same. Only the template content differs.

Building and testing

To generate the compiler front-end, run ANTLR on the grammars and compile:

```
$ java org.antlr.Tool *.g
$ javac *.java
```

Then we can jump into the `tests` directory and run the test script. Here's the output you should see:

```
$ cd tests
$ ./go
########### call
3
########### if
4
########### fact
3628800
```

```
########### global_array
99
########### printf
hello, world
99
```

The test script runs each example through our main class `CC` to get the IR (.ll file). Then it uses LLVM's tools to create the .s assembly language file. From there, it compiles and links to the associated C main program with gcc:

```
for f in call if fact global_array printf
do
    echo "###########" $f
    java -cp "..:$CLASSPATH" CC -templates llvm.stg $f.c > $f.ll
    llvm-as -f $f.ll
    llc -f $f.bc
    llvm-gcc -o $f $f"_main.c" $f.s
    ./$f
done
```

Source code

See the attached llvm.tar.gz or llvm.zip.

Installing and configuring LLVM

For this project, we need to get llvm-based GCC front ends available and also the LLVM toolkit. First we download and then untar the packages. We also run the configure command for LLVM and then compile it (which takes quite a while so I unleash seven of my eight processors on the build):

I couldn't get 2.6 to build on 10.6.2 OS X (I'm sure it's my fault). It built ok on 10.5.8 with this process though and I just copied it over.

```
/usr/local # tar xfz llvm-gcc-4.2-2.6-i386-darwin9.tar.gz
/usr/local # export PATH="/usr/local/llvm-gcc-4.2-2.6-i386-darwin9/bin:$PATH"
/usr/local # tar xfz llvm-2.6.tar.gz
/usr/local # cd llvm-2.6
/usr/local/llvm-2.6 # ./configure --prefix=/usr/local/llvm-2.6
checking build system type... i686-apple-darwin10.2.0
checking host system type... i686-apple-darwin10.2.0
...
/usr/local/llvm-2.6 # make -j7 # build using 7 processors
...
/usr/local/llvm-2.6 # export PATH="/usr/local/llvm-2.6/Release/bin:$PATH"
```

No labels