Assignment 2: Compiling to LLVM
Due: 11:59pm, Tuesday, February 8, 2011

Overview

The `javac` compiler (and `ToJVM`) generates JVM bytecode. A somewhat lower-level—but still portable—bytecode is LLVM, which stands for *Low-Level Virtual Machine*. Your task in this assignment is to write a simple compiler to produces LLVM assembly language. This can then be automatically translated to assembly code for a particular architecture, and then assembled and linked into runnable machine language.

Repository Setup

All provided files can be retrieved from the subversion repository, and your work will be submitted by committing it into the repository. Set up your working copy of the repository as follows:

1. You may work on this assignment alone. You may also work another student *as long as you strictly follow the CS 70 pair-programming rules*.
   
   If you are working with a partner, If you work as a pair, then figure out both your usernames and put them *in alphabetical order*. In one of your two accounts, go to the `cs132s11` directory and (careful with the spelling!) run
   
   ```bash
   svn mkdir <first-username>-<second username>
   ```
   
   (e.g., `svn mkdir asampson-kmarsh`, not `kmarsh-asampson`, even though Marsh comes alphabetically before Sampson).
   
2. In your directory (or your pair’s directory),
   
   ```bash
   svn copy ../src/a1 .
   ```
   
   to get a copy of the initial files you’ll need.

3. Use `svn commit` in your `cs132s11` directory to make these setup steps permanent.

4. The code you are given can parse and translate programs containing a single statement that prints an integer (e.g., `test-input0`). Try this by running `make test0`. Look at the LLVM code in `test-input0.ll`, and verify that you can run the `test0` binary.

   At this point you should be able to start programming, and committing your changes.

---

1Work only on the assignment when you’re physically together; be logged in only to a single computer, with one keyboard and screen for the two of you; at every point one student is primarily typing and the other is watching and making suggestions; the two roles must be regularly exchanged and shared equally during the assignment.
<program> ::= <statement>

<statement> ::= print <expr> ;
| <identifier> = <expr> ;
| if ( <test> ) <statement>
| if ( <test> ) <statement> else <statement>
| do <statement> while ( <test> ) ;
| { <statements> }

<statements> ::= ε
| <statement> <statements>

<expr> ::= <nonnegative-integer-constant>
| <identifier>
| - <expr>
| <expr> + <expr>
| <expr> - <expr>
| <expr> * <expr>
| <expr> / <expr>
| <expr> & <expr>
| <expr> | <expr>
| ~ <expr>
| ( <expr> )

<test> ::= <expr> == <expr>
| <expr> != <expr>
| <expr> < <expr>
| <expr> <= <expr>
| <expr> > <expr>
| <expr> >= <expr>

Figure 1: Grammar for the Input Language
The MiniTree Input Language

The “Minitree” language you must compile is described in Figure 1. It differs from Microtree in a few small ways:

- if and do statements can now contain an arbitrary integer comparison, rather than comparing a single expression against zero.
- if statements now have an optional else
- We extend the binary operators to include subtraction and division on signed integers, as well as bitwise and, or, and not.
- Parentheses remain optional, except in if and do statements. (The grammar in Figure 1 is obviously ambiguous. Your parser should produce the same precedence and associativity as Java.)

1 Parsing

For this assignment, you should use alex and happy to create your lexer and parser. I strongly suggest you extend the lexer and parser bit by bit, using ParseTest at each stage to verify things compile and run correctly!

- Complete the file Lexer.x to include all tokens for Minitree concrete syntax. (You’ll need to extend the Haskell token type with all the kinds of tokens you need.)
- Complete the file Parser.y to parse Minitree concrete syntax. This involves describing all your tokens, all your grammar rules, and the code for each rule.

The ParseTest program has been updated to call your new lexer and parser. You can compile it via

```
make ParseTest
```

As before, the ParseTest program reads in a file and prints both the list of tokens and the abstract syntax.

2 Generating LLVM

The second part of the assignment is to complete the code in ToLLVM.hs, to translate Minitree abstract syntax into LLVM assembly language. The only code you should need to change is to extend the transExp and transStm functions, plus any helper functions you add.

Start by reading Appendix A to get an introduction to LLVM.

You will notice that the existing code in ToLLVM.hs is written in monadic style, using a “translation monad” defined specifically for this assignment. The hope is that this will make your life
easier (and more like how you’d write code in languages other than Haskell), by skipping all the “plumbing” (passing and returning extra values, and threading them from each call to the next) you probably saw in ToJVM.hs. The TM monad provides the following handy operations (to be used with do and <-):

- freshLabel :: TM String
  Produces a brand-new assembly-language label (e.g., L1, L2, etc.)

- freshTemp :: TM String
  Produces a brand-new variable name (e.g., %t1, %t2, etc.)

- lookupVarI :: VarName -> TrM VarName
  Given a Minitree variable, returns the LLVM temporary variable that will contain its address. (If this is the first time the Minitree variable has been encountered, the monad will allocate a fresh temporary to contain its address.)

There are two other operations defined by the monad, but you shouldn’t need to call them; they’re used only by the skeleton code you were provided.

- makeAllocs :: TrM [CodeLine]
  Based on all the calls to lookupVarI, generate all the necessary alloca instructions for the start of your LLVM code. [This is called for you in the skeleton code, so you don’t need to worry about it.]

- runTR :: TrM a -> a
  Runs the given translation computation. [This is also called for you in the skeleton code.]

3 Testing

Come up with a more interesting test input for Minitree, and name it interesting-test. (Don’t forget to svn add it to the repository!)

A LLVM Background Information

The official LLVM reference manual can be found on-line at http://llvm.org/docs/LangRef.html, but here are some of the most important points:

1. You can have as many local variables (“temporaries”) as you want; these local variables have a % prefix (e.g., %foo). Global variables/functions have a @ prefix (e.g., @main). Local variables do not have to be declared anywhere; you just assign to them to initialize them.

2. In constrast with most assembly languages, nearly all instructions specify the types of the arguments (e.g., i32 for 32-bit ints, i32* for pointers to 32-bit ints, i1 for 1-bit booleans, label for labels).
3. Here are the most important instructions that you might generate. The notation \(<cv>\) indicates a constant (e.g., 7) or a local variable (e.g., %t2), \(<v>\) indicates a local variable, and \(<l>\) indicates the name of a label with a leading \(%\) (e.g., \(%L3\)).

```assembly
store i32 <cv1>, i32* <cv2>  // store the (integer) value of <cv1> into the address <cv2>
<v> = load i32* <cv>        // load the (integer) value at address <cv> into <v>.
<v> = add i32 <cv1>, <cv2>  // addition on integers
<v> = sub i32 <cv1>, <cv2>  // subtraction on integers
<v> = mul i32 <cv1>, <cv2>  // multiplication on integers
<v> = sdiv i32 <cv1>, <cv2> // signed division on integers
<v> = and i32 <cv1>, <cv2>  // bitwise and on integers
<v> = or i32 <cv1>, <cv2>   // bitwise or on integers
<v> = xor i32 <cv1>, <cv2>  // bitwise exclusive-or on integers
br label <l>                // jump to the given label
<v> = icmp <cc> i32 <cv1>, <cv2> // Compute a 1-bit comparison result. <cc> is the comparison: eq, ne, slt, sle, sgt, sge
br i1 <cv>, label <l1>, label <l2> // Test the 1-bit <cv>; jump to <l1> if true, <l2> otherwise.
call void @printInt(i32 <cv>) // Call the printInt function and pass it this one integer argument.
ret void                    // Return statement (returning no value).
```

4. As mentioned above, if you want to refer to a label in a branch statement you need to prefix it with \(%\) (e.g., branch label \(%L17\)). But if you want to define the label, you just say

\[ \text{L17:} \]

without the percent sign.

5. Rather than referring to variables by number as in JVM, you can give them names (e.g., \(%\text{foo}\), \(%\text{t0}\), etc.). There is one tricky bit however: although these variables look a bit like registers in traditional assembly language, you’re only allowed to write to each variable once! More precisely, a variable can only be written to by one line of code in your program. (If that instruction happens to be inside a loop, and gets executed many times, that’s OK.) This aspect is more like Haskell than traditional assembly; rather than changing an existing variable, you often create a brand-new variable.

The official terminology is LLVM requires SSA ("static single assignment") form, but we haven’t talked about this yet, and it’s probably too complicated for our simple compiler anyway. So we’ll use a workaround, based only the fact that LLVM can load and store to a memory location as many times as you want. Further, we can allocate space for a variable on the stack with the bytecode

\(<v> = \text{alloca i32}\)
So, the Minitree input code

\[
\begin{align*}
\quad x &= 999 \\
\quad x &= x + 888 \\
\quad x &= x \times 777
\end{align*}
\]

should be translated as (modulo names of temporary variables)

\[
\begin{align*}
%t1 &= \text{alloca i32} \\
\quad \text{store i32 999, i32* } %t1 \\
%t17 &= \text{load i32* } %t1 \\
%t18 &= \text{add i32 } %t17, 999 \\
\quad \text{store i32 } %t18, \text{ i32* } %t1 \\
%t97 &= \text{load i32* } %t1 \\
%t98 &= \text{mul i32 } %t97, 777 \\
\quad \text{store i32 } %t98, \text{ i32* } %t1
\end{align*}
\]

Note that instead of storing \(x\)'s value in a temporary, we store its address in a temporary (%t1).

6. As a consequence of SSA form, LLVM includes no “move” instruction. Suppose you wanted to say %t2 = %t1 (or %t2 = 3). Given that in SSA form this instruction would be the unique definition of %t2, you can instead just replace all uses of %t2 by %t1 (or by 3).

7. LLVM code is a sequence of “basic blocks.” Each block can start with an label, but must end with a jump, call, or return. You may not “fall through” to a label; you have to explicitly jump to it, even if it is the very next instruction!

A simpler way to think about it is to ensure that every label must follow a jump, call, or return.

So, to skip over a sequence of code if %t8 is false,

\[
\begin{align*}
\text{br i1 } %t8, \text{ label } %L1, \text{ label } %L2 \\
\quad \text{L1:} \\
\quad \quad \text{...code to be skipped...} \\
\quad \quad \text{br label } %L2 \\
\quad \text{L2:}
\end{align*}
\]

Here the second br is mandatory, even though it only goes to the next line.
8. The skeleton code will ensure that your generated .ll file looks like:

```cpp
declare void @printInt(i32)

define void @main() {
    ...allocate room for variables on the stack...
    ...whatever code you generate...
    ret void
}
```

9. On WILKES and KNUTH, you can compile your output to a .s file (via a binary bytecode representation) by running

```
llvm-as < myfile.ll | llc > myfile.s
```

If you want to include optimization of the LLVM code, you can run

```
llvm-as < myfile.ll | opt -O2 | llc > myfile.s
```

If you want to see what the optimizer does to the bytecode (I think it’s pretty interesting!), you can run

```
llvm-as < myfile.ll | opt -O2 | llvm-dis
```

10. If you have a file runtime.c that defines a function

```cpp
void printInt(int n);
```

for printing integers, then you can get a running program by saying

```
gcc -o myfile myfile.s runtime.c
```

See the examples in the Makefile