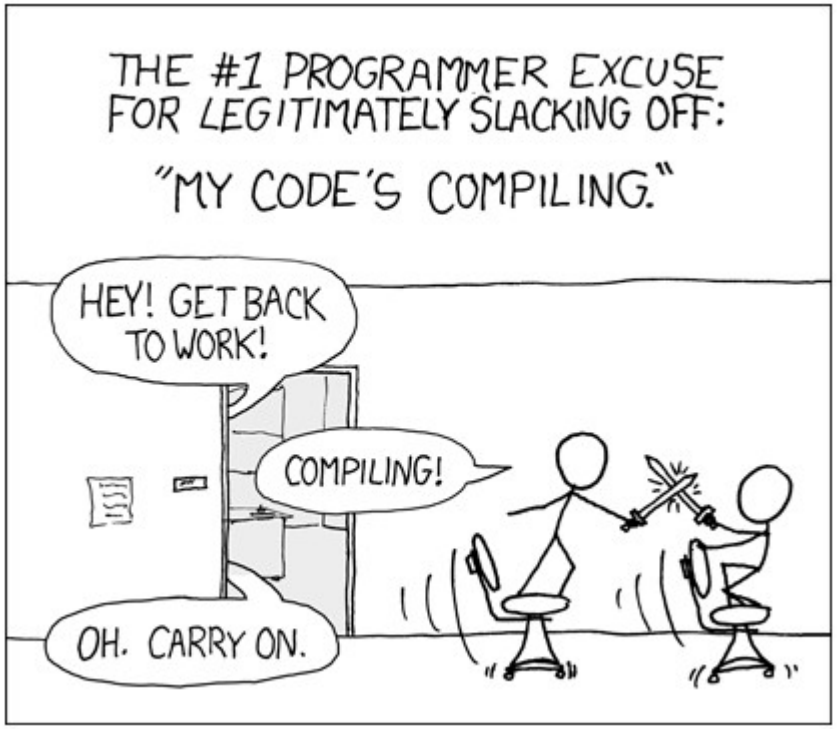
Project Compiler



CS031 – TA Help Session

November 28, 2011

Motivation

Generally, it's easier to program in higher-level languages than in assembly.

Our goal is to automate the conversion from a higher-level language to assembly with a compiler.

In this project, we will code some techniques used to create such compilers.

We will create a compiler for the made-up language Blaise.

Blaise

Blaise is a small procedural language. While not as complex as Java or C, Blaise is sufficiently powerful to program many things.

It's an LL(1) language, so at any given point, only 1 token is needed to determine which rule to follow.

Except for global variable declarations, all code goes inside procedures, and specific rules govern the usage of these procedures.

Blaise Sample Code

```
var int fibs[20];

procedure int main() {
  int index;
  index := 0;
  while (index < 20) {
    if (index < 2)
      fibs[index] := 1;
    else
      fibs[index] := fibs[index - 2] + fibs[index - 1];
    endif
    output(fibs[index]);
    index := index + 1;
  }
}
```

Language features:

- Global variables (keyword var)
- Procedures, including main (keyword procedure)

Support Code

We've provided lots of code. The only support class that you may modify is MIPSCodeGenerator.

Support code includes:

- Many nodes (for building your parse tree)
- Classes to help with scoping and procedures (including stack frames).
- A class to help with code generation

Compiler Stages

Scanning – Turn code into tokens (done for you!).

Parsing – Turn the tokens into the abstract syntax tree. Make sure the grammar is followed.

Semantic analysis – Make sure the AST "makes sense" (variable scope, type analysis, procedure usage). Gather info for code gen (variables and frames).

Code generation – Turn the AST into Assembly.

Parsing

Make sure you understand the tokens generated in the scanning phase.

Given the grammar, look at tokens one at a time (LL(1) grammar). Based on which rules you follow, create nodes appropriately. If no rule fits, throw a SyntaxException.

Use helper methods and consider recursion!

These nodes and their children form an abstract syntax tree, rooted in a NodeProgram, which will be traversed in the remaining steps.

Parsing - Example

```
<exp> ::= <sum> EOF

<sum> ::= <prod> <sumSfx>

<sumSfx> ::= + <sum>
<sumSfx> ::= - <sum>
<sumSfx> ::=

<prod> ::= <fact> <factSfx>

<factSfx> ::= * <prod>
<factSfx> ::=

<fact> ::= integer
<fact> ::= identifier
```

Does x * 4 - 3 parse? How?

What about 6 + 3 * x?

Semantic Analysis

Even if code can be parsed into an AST, it may not be correct.

Goals: Catch semantic errors by traversing the tree, gather information about variables and frames.

Errors may arise in scoping, type checking, or procedure usage.

This step is more complicated if you choose to implement extra credit.

Semantic Errors

Scoping errors occur when a variable is used "out of scope," meaning they haven't yet been declared in the scope in which they're used. Scoping errors also happen when a variable is declared twice in the same scope (no variable shadowing in Blaise!)

A block, implicit or explicit, creates a scope in Blaise. Consider using the provided Scope class to track variables' scopes.

Variables must be used as the same type as which they were declared. ints must be used as ints and not bools or arrays. 3 > true makes no sense; if (1) doesn't either (in Blaise, anyways).

Semantic Errors

Procedures can cause numerous semantic errors. There must be a no-args procedure named main. Arguments must be used correctly (right number and right type). One can make procedure calls only to procedures declared earlier in the code.

For simplicity, no procedure overloading is allowed. Additionally, return 0; is implicitly called at the end of every procedure, so you don't need to check for valid return statements. Be sure no variable has the same name as a procedure!

How do we check all of this?

Visitor Pattern

Visitor Pattern – a common design pattern for compilers. We create a visitor for each task (semantic analysis, code gen, printing the AST, etc.) with a method for each node type. The node passes itself to the visitor.

Ex: public void handleExprPlus(NodeExprPlus n)

Advantage: easy to add new operations (just create 1 new class)

Disadvantage: harder to add new nodes (need to edit many classes)

Semantic Analysis (cont.)

Information about variables must be collected.

Arguments and local variables for a procedure are placed on the stack in a frame. Each of these variables has an offset from the frame pointer that allows us to access the variable. You have to push and pop frames during procedure calls, but we provide code to help you set them up.

Note that our frames are laid out differently than those from lecture for simplicity. In particular, arguments and local variables are put together. (See HW09 for details!)

Code Generation

We want to turn our beautiful, error-free AST into MIPS assembly code.

How? Visitor Pattern again.

We want to write code to a file for each node in the AST.

We have provided a class that will generate MIPS instructions for you (MIPSCodeGenerator).

You must write the visitor that calls MIPSCodeGenerator's methods (e.g. you may call genPop, which writes the MIPS code needed to pop something off of the stack.

Code Generation (cont.)

You are writing code for a stack-based machine.

The result of each operation is pushed onto the stack, to be popped by other operations

The less than relational operator assumes that its two arguments are on the stack and pushes its result onto the stack.

What should we do if we calculate a sum for a NodeExprPlus and the result is in $s1?

Code Generation (cont.)

Code Generation (cont.)

Our frame support code assumes that all $s registers, plus $ra and $fp are pushed to the stack each time a procedure is called, so your code generator should do so.

Hints:

Remember to return 0 at the end of every procedure. Also, be careful to restore your stack pointer after procedure calls!

Global and local variables will be stored in different places, so be sure to distinguish between them.

Testing

How do you know that your compiler works?

We have provided several functional Blaise programs and several buggy Blaise programs.

You should write your own, more-exhaustive test cases. Make sure you test valid **and** invalid Blaise code.

Does your compiler generate arrays correctly? Does it correctly identify semantic errors? Does short-circuiting work correctly? Recursion?

Test your compiler thoroughly!

Extra Credit

Numerous extra credit options exist.

Only attempt these after your compiler is fully functional.

If you attempt extra credit, hand in your regular compiler in addition to the one with extra credit.

In general, you will be adding functionality to the Blaise language and altering your compiler appropriately.

If you have any questions, ask at the end of the help session or e-mail the TAs.

Final Comments

Start soon – each part of the assignment may take a while.

Make sure you understand scoping and frames, as well as the visitor pattern before starting Semantic Analysis.

Any questions about the assignment?

If you have any further questions, see a TA on hours or e-mail the list. Good luck!