RAT18S Syntax Directed Translation

Justin Chin

May 7, 2018

Problem Statement

Compiler design is often broken into different phases, lexical analysis, syntax analysis, intermediate code generation, optimization and object code generation. However, it is possible to sidestep intermediate code generation entirely, through syntax directed translation. This allows semantic actions to be taken in the same pass as parsing, that is, we can attach semantic meaning to procedures in the recursive descent.

For example, if we want to generate machine code for an assignment statement, we can do that directly inside the recursive descent, without having to work from a completed parse tree. For a simple, context-free, imperative language like Rat18S, this simplifies a lot of the work required to compile source code into useable machine code.

The target machine for this project is a hypothetical stack-based virtual machine. We ignore the possibility of registers (which come standard in typical von Neumann computer architecture), and focus only on the stack, to simplify the code generation. This machine has 18 instructions, similar to basic intel x86 machine operations (such as PUSHI (Push Immediate), PUSHM (Push Memory), JUMPZ (Jump if zero), etc).

How to use the program

Windows

1. Open command prompt and navigate to the directory containing `main.exe`
2. Execute `main.exe \all > results.txt`
3. Enter path for a file to analyze

Design of the program

Not much of the program has changed overall from the recursive descent parser it was built from. There are three primary differences, the introduction of a symbol table and instruction table into the ParserState, and the addition of semantic actions to generate the virtual machine code. The symbol and instruction tables are native Haskell lists, which are grow whenever needed. To improve performance, it would be possible to implement the symbol table as a hash table or map of some kind, which would significantly improve execution time for large symbol tables, as Haskell lists suffer

from the same drawbacks as linked lists. For the purposes of this toy compiler, the list structure will suffice.

In addition to the symbol and instruction tables, semantic actions were added to some of the procedures in the recursive descent, these being the IF statement, WHILE statement, ASSIGNMENT statement, SCAN and PUT Statements. Implementing these were relatively straightforward with some assembly language knowledge, as the virtual machine this compiler targets is a simple stack based architecture, without any registers.

The main challenge in designing the program was implementing the `backPatch` procedure, which needed to be called whenever there was a condition that could evaluate to either true or false. Because of how the recursive descent works, we would need to push a `JUMPZ` instruction before knowing exactly where in the instruction table we would need to jump to, because the parser had not seen that far. This proved to be particularly challenging given Haskell's lazy evaluation and functional purity. The lazy evaluation made it difficult to ascertain when, and in what order certain operations would be evaluated. Haskell will not evaluate something until its value is required. This in addition with the inability to easily modify the instruction table in-place (data structure in Haskell are pure), meaning that to change the state of the instruction table, I needed to replace the entire instruction table.

I was able to achieve this by implementing a `replaceAtIndex` procedure with type `Int -> a -> [a] -> [a]`. Essentially, given an index, some element of type a, and a collection (i.e list) of elements of type a, return a new collection of type a. The idea is similar to Python's slice notation.

Limitations

Declarations on their own line

Declarations must be on their own line. The way the recursive descent was designed does not easily allow me to associate IDs with a type if they are declared `int x, y, z;`, because the resulting parse looks like the following:

```
(DeclarationList
 (Declaration1 QualifierInt
               (IDs
                (Token ...)
                (IDsPrime
                 (IDs
                  (Token ...)
                  (IDsPrime
                   (IDs
                    (Token ...)))))))))
```

Notice how the type information is not associated with each individual token, but with the declaration list itself. In retrospect it should be possible to carry some sort of state which could be read for the type of symbol that is being declared, however this would need to be wrapped up into the ParserState State Monad, which is already growing to an unwiedly size.

No function definitions

Because simplified Rat18S does not allow function definitions, this syntax directed translation was not written with them in mind, and has not been tested on Rat18S source files that have function definitions. I imagine it would fail, since separate scopes were not implemented for this project.

Cannot end file with While or If statement

I was unable to implement a method for selectively adding a label instruction at the end of the file if in a while or if statement. I don't think it would be that hard to do, but time is of the essence as finals approach.

Shortcomings

Boolean arithmetic not explicitly forbidden

Because of the way I wrote the parser and the parser table, it is cumbersome to explicitly forbid boolean arithmetic. It shouldn't be too hard to implement, just time consuming, since I'll need to thread along some additional state to make the checks.