Exploring Languages with Interpreters and Functional Programming Chapter 46

H. Conrad Cunningham

4 November 2018

Contents

46 Calculator: Compilation **2**
46.1 Chapter Introduction . . . 2
46.2 Stack Virtual Machine . . . 2
46.2.1 Instruction set syntax . . . 2
46.2.2 Instruction set semantics . . . 2
46.2.3 Machine execution . . . 3
46.2.4 Compilation . . . 4
46.2.5 Source code . . . 4
46.3 Exercise Set A . . . 4
46.4 Conditional Expressions . . . 5
46.4.1 Extending the Expression Language . . . 5
46.4.2 Extending the stack virtual machine (UNFINISHED) . . . 6
46.4.3 Extending the compiler (UNFINISHED) . . . 7
46.5 Exercise Set B (UNFINISHED) . . . 7
46.6 Acknowledgements . . . 7
46.7 References . . . 8
46.8 Terms and Concepts . . . 8

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of November 2018 is a recent version of Firefox from Mozilla.

46 Calculator: Compilation

46.1 Chapter Introduction

This is a partially developed chapter.

TODO: - Complete and revise the conditional expression sections as needed (e.g., the compilation subsection does not discuss the handling of labels/addresses sufficiently) - Consider adding separate compilation units and linking of units together

46.2 Stack Virtual Machine

Consider a stack virtual machine as a means for executing the Expression Language. The operation of this machine is similar to the operation of a calculator that uses Reverse Polish Notation (or postfix notation) such as the calculators from Hewlett-Packard.

46.2.1 Instruction set syntax

Consider a stack-based virtual machine with a symbolic instruction set defined by the following abstract syntax:

```
data SInstr = SVal Int
            | SVar String
            | SPop
            | SSwap
            | SDup
            | SAdd
            | SMul
              deriving (Show, Eq)
```

46.2.2 Instruction set semantics

Suppose the state of the virtual machine consists an *evaluation stack* of values and a program counter indicating the next instruction to be executed. Further suppose the above instructions have the following semantics. The machine executes much like a calculator that uses "reverse Polish notation".

- `SVal i` pushes value `i` onto the top of the evaluation stack.
- `SVar v` pushes the value of "variable" `v` from the current environment onto the top of the evaluation stack. (Here we are simulating a memory with the environment.)

- SPop removes the top element from the stack. (That is, if the stack from the top is 10:xs, then the resulting stack is xs.)
- SSwap exchanges the top two elements on the stack. (That is, if the stack from the top is 10:20:xs, then the resulting stack is 20:10:xs.)
- SDup pushes another copy of the top element onto the stack. (That is, if the stack from the top is 10:xs, then the resulting stack is 10:10:xs.)
- SAdd pops the top two elements from the stack, adds the second to the first, and pushes the result back on top of the stack. (That is, if the stack from the top is 10:20:xs then the resulting stack is 30:xs.)
- SMul pops the top two elements from the stack, multiplies the second times the first, and pushes the result back on top of the stack. (That is, if the stack from the top is 10:20:xs then the resulting stack is 200:xs.)

We extend this instruction set later to provide other operations.

46.2.3 Machine execution

We can define a simple skeletal execution mechanism for the Stack Virtual Machine as follows. Function execSInstr takes the state, environment, and instruction and returns the modified state and environment. (This version does not modify the environment, but a version in the future may do so.)

```
data SState = SState [Int] Int
              deriving (Show, Eq)

execSInstr :: SState -> Env -> SInstr -> (SState, Env)
execSInstr (SState es pc) env (SVal i) =
    (SState (i:es) (pc+1), env)
execSInstr (SState es pc) env (SVar v) =
    case lookup v env of
        Just i  -> (SState (i:es) (pc+1), env)
        Nothing -> error ("Variable " ++ show v ++ " undefined")
execSInstr (SState es pc) env SPop  =
    (SState es pc, env) -- REPLACE
execSInstr (SState es pc) env SSwap =
    (SState es pc, env) -- REPLACE
execSInstr (SState es pc) env SDup  =
    (SState es pc, env) -- REPLACE
execSInstr (SState es pc) env SAdd  =
    case es of
        (r:l:xs) -> (SState ((l+r):xs) (pc+1), env)
        _        -> error ("Cannot Add. Stack too short: " ++ show es)
execSInstr (SState es pc) env SMul = (SState es pc, env) -- REPLACE
```

46.2.4 Compilation

We can translate the Expression Language abstract syntax trees to sequences of stack virtual machine instructions. We call this process *code generation* and call the whole process of converting from source code to the instruction set *compilation*.

We consider compilation of the Expression Language to the stack virtual machine in Exercise Set A.

46.2.5 Source code

- Stack Virtual Machine?

46.3 Exercise Set A

In this exercise set, we consider the Stack Virtual Machine and translation of the Expression Language's abstract syntax trees to equivalent sequences of instructions.

1. Complete the development of the function `execSInstr`, adding the code for the `SPop`, `SSwap`, `SDup`, and `SMul` instructions.

2. Extend the Stack Virtual Machine instruction set (i.e., `SInstr`) to support the extensions to the `Expr` data type defined in Exercise Set A (i.e., `Sub`, `Div`, `Neg`, `Min`, and `Max`). The operators take top value as their *right* operands and the value under that as the *left* operand.

3. Develop a Haskell function

   ```
   execSeq :: SState -> Env -> [SInstr] -> (SState, Env)
   ```

   that executes a sequence of Stack Virtual Machine instructions given the initial state and environment. (Although the machine in this case study so far does not modify the environment, allow for the future possibility of modification. A later exerces may extend the Expression Language to add assignment statements, imperative loops, and variable and function declarations.)

   Also develop a function `exec` that executes a sequence of instructions from an initially empty stack with the given environment and returns the result on top of the stack after execution. (You may use `execSeq`.)

   ```
   exec :: Env -> [SInstr] -> Int
   ```

4. Develop a Haskell function

   ```
   compile :: Expr -> [SInstr]
   ```

that translates the extended expression tree from Exercise Set A to a sequence of Stack Virtual Machine instructions as extended in this exercise set.

5. Develop a Haskell function `compGo` that takes an expression tree, simplifies, compiles, and executes it using the given environment. You may use the functions `exec` and `compile` from the previous exercises.

```
compGo :: Env -> Expr -> Int
```

46.4 Conditional Expressions

Let's examine how to extend the ELI Calculator language to include comparisons and conditional expressions.

46.4.1 Extending the Expression Language

TODO: This was introduced as a operator in a previous chapter.

Suppose that we redefine `Expr` to include binary operators `Eq` (equality) and `Lt` (less-than comparison), logical unary operator `Not`, and the ternary conditional expression `If` (if-then-else).

```
data Expr = ...
          | Eq  Expr Expr
          | Lt  Expr Expr
          | Not Expr
          | If  Expr Expr Expr
          ...
            deriving Show
```

This extended language does not have Boolean values. We represent "false" by integer 0 and "true" by a nonzero integer, primarily by 1.

We express the semantics of the various Expression Language expressions as follows:

- `Eq l r` evaluates to the value 1 if `l` and `r` have the same value and to 0 otherwise.
- `Lt l r` evaluates to the value 1 if the value of `l` is smaller then the value of `r` and to 0 otherwise.
- `Not i` evaluates to 1 if `i` is zero and evaluates to 0 if `i` is nonzero.
- `If c l r` first evaluates `c`; if `c` is nonzero, the `if` evaluates to the value of `l`; otherwise the `if` evaluates to the value of `r`.

46.4.2 Extending the stack virtual machine (UNFINISHED)

TODO: This discussion in the remainder of the Conditional Expression section is not complete! In particular, the discussion of labels/addresses must be clarified and expanded—probably changed.

Suppose we redefine `SInstr`, the Stack Virtual Machine to include the new instructions:

```
data SInstr = ...
            | SEq
            | SLt
            | SLnot
            | SLabel String
            | SGo    String
            | SIfZ   String
            | SIfNZ  String
             deriving (Show, Eq)
```

These Stack Virtual Machine instructions execute as follows:

- `SEq` pops the top two values from the stack; if the values are equal, it pushes a `1` onto the stack; otherwise, it pushes a `0`. (For example, if the stack from the top is `3:4:xs`, the resulting stack is `0:xs`.)
- `SLt` pops the top two values from the stack; if the second value is smaller than the top value, it pushes a `1` onto the stack; otherwise, it pushes a `0`. (For example, if the stack from the top is `3:4:xs`, the resulting stack is `0:xs`.)
- `SLnot` pops the top value from the stack; if the top is `0`, it pushes `1` back onto the stack; if it is nonzero, it pushed `0` back onto the stack. (For example, if the stack from the top is `0:xs`, the resulting stack is `1:xs`. If the stack is `7:xs`, then the result is `0:xs`.)
- `SLabel n` does not change the stack. It is a pseudo-instruction to enable a jump to this point in the program using label `n`.
- `SGo n` makes the next instruction to be executed the one labelled `n`; it does not change the stack.
- `SIfZ n` pops the value from the top of the stack; if this value is zero, then the next instruction executed will be the one labelled `n`; otherwise the next instruction is the one following the `SIfZ` instruction.
- `SIfNZ n` pops the value from the top of the stack; if this value is nonzero, then it makes the next instruction executed the one labelled `n`; otherwise the next instruction is the one following the `SIfNZ` instruction.

46.4.3 Extending the compiler (UNFINISHED)

We can translate the expression

```
If (Eq (Var "x") (Val 1)) (Val 10) (Val 20)
```

to a sequence of Stack Virtual Machine instructions such as:

```
[ SVar "x", SVal 1, SEq, SIfZ "else", SVal 10, SGo "end",
  SLabel "else', SVal 20, SLabel "end" ]
```

Of course, each `If` needs a unique set of labels.

46.5 Exercise Set B (UNFINISHED)

1. Extend the `eval` function to support the `Eq`, `Lt`, `Not`, and `If` operators.
2. Extend the `simplify` function to support the `Eq`, `Lt`, `Not`, and `If` operators.
3. Extend the data type `Expr` and the `eval` function to support the other comparison operators `Ne` (not equal), `Le` (less or equal), `Gt` (greater than), and `Ge` (greater or equal) and the logical operators `And` and `Or`.
4. Extend the `simplify` function to support the comparison operators `Ne`, `Le`, `Gt`, and `Ge` and the logical operators `And` and `Or` added in the previous exercise.
5. (UNFINISHED) Extend the `execSInstr`, `execSeq`, and `exec` functions from Exercise Set C to include the new Stack Virtual Machine instructions.
6. (UNFINISHED) Extend the `compile` and `compileGo` functions from Exercise Set C to include support for `Eq`, `Lt`, and `Not`.
7. (UNFINISHED) Extend the `compile` and `compileGo` functions from the previous exercise to include expressions `Ne`, `Le`, `Gt`, `Ge`, `And`, `Or`, and `If`. Each of these may need to be translated to a sequence of Stack Virtual Machine instructions.

46.6 Acknowledgements

I initially developed the ELI Calculator language (then called the Expression Language) case study for the Haskell-based offering of CSci 556, Multiparadigm Programming, in Spring 2017. I based the work in this partial chapter, in part, on ideas from:

- Sections 1.3, 2.5, and 2.7 and Chapter 8 of Peter Sestoff's *Programming Language Concepts*, Springer, 2012.
- Chapters 6 (Purely Functional State) from Paul Chiusano and Runar Bjarnason's *Functional Programming in Scala*, Manning, 2015.

I made this work a chapter of the 2017 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* It remains a separate chapter in the 2018 version of the textbook.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed, The HTML version of this document may require use of a browser that supports the display of MathML.

46.7 References

TODO: Edit this

[**Abelson 1996**]**:** Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs* (SICP), Second Edition, MIT Press, 1996.

[**Appel 1998**]**:** Andrew W. Appel. *Modern Compiler Implementation in ML*, Cambridge, 1998. (Especially section 3.2 "Predictive Parsing")

[**Chiusano 2015**]**:** Paul Chiusano and Runar Bjarnason, *Functional Programming in Scala*, Manning, 2015. (Especially chapters 6 "Purely Functional State" and 9 "Parser Combinators")

[**Fowler 2011**]**:** Martin Fowler and Rebecca Parsons. *Domain-Specific Languages*, Addison Wesley, 2011. (Especially chapter 21 "Recursive Descent Parser")

[**Kamin 1990**]**:** Samuel N. Kamin. *Programming Languages: An Interpreter-Based Approach*, Addison-Wesley, 1990.

[**Linz 2017**]**:** Peter Linz. *An Introduction to Formal Languages and Automata*, Fifth Edition, Jones and Bartlett, 2017. (Especially sections 1.2, 3.3, and 5.1)

[**Schinz-Haller 2017**]**:** Michel Schinz and Philipp Haller. A Scala Tutorial for Java Programmers, accessed February 2016.

[**Sestoft 2012**]**:** Peter Sestoft. *Programming Language Concepts*, Springer, 2012. (Especially sections 1.3, 2.5, and 2.7 and chapter 8)

[**Wikipedia 2017**]**:** Wikipedia articles "Regular Grammar", "Context-Free Grammar", "Backus-Naur Form", "Lexical Analysis", "Parsing", "LL Parser", "Recursive Descent Parser", and "Abstract Syntax".

46.8 Terms and Concepts

TODO