Writing a Mini-Lisp Compiler

Carlo Hamalainen

carlo-hamalainen.net

28 May 2013
About 13 years ago I did an BSc/BInfTech combined degree.

A few of the IT subjects were fun:

- Computational complexity. NP-complete, etc.
- Operating systems. We wrote our own memory pager in C.
- Functional programming. Gofer on MS-DOS.
- Compilers. We wrote a compiler for a mini-Java language, in C.
The Architecture of Symbolic Computers (Peter M. Kogge, 1991)
The Architecture of Symbolic Computers (Peter M. Kogge, 1991)

http://blog.fogus.me/2012/07/25/some-lisp-books-and-then-some/

“The best Lisp book I’ve ever read and could ever possibly read.”

http://www.loper-os.org/?p=13

“The Architecture of Symbolic Computers (Peter M. Kogge) is quite possibly the most useful resource I have come across in my quest thus far. If you are interested in Lisp Machine revival, non-von Neumann computation, or the dark arts of the low-level implementation of functional programming systems, you will not be disappointed.”
Here’s the plan:

1. Define a virtual machine: memory, registers, opcodes.
2. Write an emulator.
3. Define a mini-Lisp language (let, lambdas, etc).
4. Write a compiler for mini-Lisp to the virtual machine.
16K per box!

http://www.computer-history.info/Page4.dir/pages/IBM.704.dir/
IBM 704/709 - circa 1950

Operator console.

http://www.computer-history.info/Page4.dir/pages/IBM.704.dir/
IBM 701 console
The IBM 704 had 36-bit machine words:

- address part: 15 bits
- decrement part: 15 bits
- prefix part: 3 bits
- tag part: 3 bits

Corresponding instructions:

- CAR: Contents of Address part of Register
- CDR: Contents of Decrement part of Register

Think of CAR as head and CDR as tail. Store linked lists with ease.
Store this list:

\[ [1, 2, 3] \]

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>(6, 7)</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>(8, 9)</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>(10, 11)</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>(nil, nil)</td>
</tr>
</tbody>
</table>
[LDC, [3, 4], LDF, [LD, [1, 2], LD, [1, 1], ADD, RTN], AP, WRITEI, STOP,]
A linked list can represent a stack:

- push: insert at front, return new head.
- pop: remove from front, return cdr (tail) of original head.

We can run simple computations if we have two things:

- A stack for temporary results (like an accumulator register); and
- a pointer to the next instruction to execute.
S: temporary results stack.
C: program counter for currently executing code.
LDC: Load Constant
ADD: integer ADDition
STOP: STOP the machine

Add 3 and 4:

1. S: []  C: [LDC, 3, LDC, 4, ADD, STOP]
2. S: [3]  C: [LDC, 4, ADD, STOP]
3. S: [4, 3]  C: [ADD, STOP]
4. S: [7]  C: [STOP]
LDC = 100
ADD = 101
STOP = 102

S = []
C = [LDC, 3, LDC, 4, ADD, STOP]

def opcode_LDC(S, C):
    assert C[0] == LDC
    C.pop(0)  # pop LDC
    S.insert(0, C.pop(0))  # push value onto S

def opcode_ADD(S, C):
    assert C[0] == ADD
    C.pop(0)  # pop ADD
    val1 = S.pop(0)  # first value
    val2 = S.pop(0)  # second value
    S.insert(0, val1 + val2)  # push result
Here's the machine emulator:

S = []
C = [LDC, 3, LDC, 4, ADD, STOP]

while True:
    if C[0] == STOP: break
    elif C[0] == LDC:
        opcode_LCD(S, C)
    elif C[0] == ADD:
        opcode_ADD(S, C)
    else:
        assert False
Summary of the machine so far:

- **Memory**: a cell contains an integer or (car, cdr) tuple.
- **Registers**: S (stack), C (code).
- **Opcodes**: \texttt{ADD}, \texttt{LDC}, \texttt{STOP}.

This doesn’t do much — we need functions of some sort.
Lambda expressions:

- **Python:** `(lambda x, y: x + y)(3, 4)
- **Haskell:** `(\x y -> x + y) 3 4
- **Haskell’** *(\x y -> (+) x y) 3 4
- **Lisp:** `((LAMBDA (x y) (+ x y)) (3 4))

Problems:

- Can’t store “x”, so remember \( x = \) first parameter, etc.
- Save environment before call: new register D (dump).
- Store arguments: new register E (environment).
- New opcodes:
  - LD: Load identifier. e.g. \([LD, 1]\) = load value of \( x \).
  - LDF: Load Function. Push a closure onto S.
  - AP: Apply function. Evaluate a closure.
  - RTN: Return from function. Restore state.
Carlo Hamalainen (carlo-hamalainen.net)  Writing a Mini-Lisp Compiler  28 May 2013  18 / 37
S: `[LD, 2, LD, 1, ADD, RTN], [3, 4]`  # a closure
E: []
C: `[AP, STOP]`
D: []

S: []
E: `[3, 4]`
C: `[LD, 2, LD, 1, ADD, RTN]`
D: `[S=[], E=[], C=[STOP]]`

S: `[4]`
E: `[3, 4]`
C: `[LD, 1, ADD, RTN]`
D: `[S=[], E=[], C=[STOP]]`
S: [3, 4]
E: [[3, 4]]
C: [ADD, RTN]
D: [[S=[], E=[], C=[STOP]]]

S: [7]
E: [[3, 4]]
C: [RTN]
D: [[S=[], E=[], C=[STOP]]]

S: [7]
E: []
C: [STOP]
D: []
SECD machine summary so far:

- **Memory**: a cell contains an integer or (car, cdr) tuple.
- **Registers**: S (stack), E (environment), C (code), D (dump).
- **Opcodes**: ADD, LDC, STOP LD, LDF, RTN, AP.
- **Can evaluate lambda expressions.**
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>integer addition</td>
</tr>
<tr>
<td>MUL</td>
<td>integer multiplication</td>
</tr>
<tr>
<td>SUB</td>
<td>integer subtraction</td>
</tr>
<tr>
<td>DIV</td>
<td>integer division</td>
</tr>
<tr>
<td>NIL</td>
<td>push nil pointer onto the stack</td>
</tr>
<tr>
<td>CONS</td>
<td>cons the top of the stack onto the next list</td>
</tr>
<tr>
<td>LDC</td>
<td>push a constant argument</td>
</tr>
<tr>
<td>LDF</td>
<td>load function</td>
</tr>
<tr>
<td>AP</td>
<td>function application</td>
</tr>
<tr>
<td>LD</td>
<td>load a variable</td>
</tr>
<tr>
<td>CAR</td>
<td>value of car cell</td>
</tr>
<tr>
<td>CDR</td>
<td>value of cdr cell</td>
</tr>
<tr>
<td>DUM</td>
<td>setup recursive closure list</td>
</tr>
<tr>
<td>RAP</td>
<td>recursive apply</td>
</tr>
</tbody>
</table>
JOIN  #  C = pop dump  
RTN   #  return from function  
SEL   #  logical selection (if/then/else)  
NULL  #  test if list is empty  

WRITEI  #  write an integer to the terminal  
WRITEC  #  write a character to the terminal  

READC  #  read a single character from the terminal  
READI  #  read an integer from the terminal  

ZEROP  #  test if top of stack = 0  [nonstandard opcode]  
GT0P   #  test if top of stack > 0  [nonstandard opcode]  
LT0P   #  test if top of stack < 0  [nonstandard opcode]  

STOP   #  halt the machine
Final steps...

- Define a simple Lisp language.
- Write a function `compile(x)` to translate Lisp forms into SECD machine code.
# 3 + 4
[ADD, 3, 4]

# lambda x, y: x + y
[LAMBDA, ['x', 'y'], [ADD, 'x', 'y']]

# (lambda x, y: x + y)(3, 4)
[[LAMBDA, ['x', 'y'], [ADD, 'x', 'y']], 3, 4]

# x = 5; y = 7; x - y
[LET, ['x', 'y'], [5, 7], [SUB, 'x', 'y']]
Square-bracket Lisp

[ADD, 3, 4]

>>> compile([ADD, 3, 4])
[LDC, 4, LDC, 3, ADD, STOP]

```python
def compile(x):
    if x[0] == ADD:
        x.pop(0)
        arg1 = x.pop(0)
        arg2 = x.pop(0)
        return [LDC, compile(arg1), \n                LDC, compile(arg2), \n                ADD] \n        + compile(x)
    elif ...
```
# lambda x, y: x + y
[LAMBDA, ['x', 'y'], [ADD, 'x', 'y']]

>>> compile([LAMBDA, ['x', 'y'], [ADD, 'x', 'y']])
[LDF, [LD, [1, 2], LD, [1, 1], ADD, RTN]]
# (lambda x, y: x + y)(3, 4)
[[LAMBDA, ['x', 'y'], [ADD, 'x', 'y]], 3, 4]

```python
>>> c = [[LAMBDA, ['x', 'y'], [ADD, 'x', 'y]], 3, 4]
>>> compile(c)
[NIL, LDC, 4, CONS, LDC, 3, CONS, LDF, [LD, [1, 2], LD, [1, 1], ADD, RTN], AP, STOP]
```
# x = 5; y = 7; x - y
[LET, ['x', 'y'], [5, 7], [SUB, 'x', 'y']]

>>> c = [LET, ['x', 'y'], [5, 7], [SUB, 'x', 'y']]
>>> compile(c)
[NIL, LDC, 7, CONS, LDC, 5, CONS, LDF, [LD, [1, 2], LD, [1, 1], SUB, RTN], AP, STOP]
Length of a list?

\[
> \text{let } f \ x \ m = \text{if null } x \\
> \quad \text{then } m \\
> \quad \text{else } f \ (\text{tail } x) \ (m + 1) \\
> f \ [1, 2, 3] \ 0 \\
3
\]
Square-bracket Lisp:

- Enough to write basic programs: LET, LAMBDA, IF.
- Recursion using LETREC.
- No destructive assignment: pure Lisp.
- Closures can be evaluated in the future, always same value.
- Code as data.
- Lambda expressions ↔ S-expressions.
- Relies on machine to do garbage collection (say what?).
The CURRY chip (Ramsdell, 1986)

- 0.4Mhz
- CPU chip: about 9000 transistors, performed graph reduction (combinators S, K, I).
- Memory chip: garbage collection.
- Timing chip: overall timing and control console.
MultiLISP: parallel Lisp

- Pure lambda calculus ⇒ opportunities for parallelism.
- Concert machine at MIT: 24-way Motorola 68000-based shared-memory multiprocessor (early 80s?).
- MultiLISP → SECD-like abstract machine called MCODE.
- MCODE interpreter (3000 lines of C).
- Unfortunately slow due to MCODE, but parallel benefits could be measured.
- Used *futures* for evaluation; see also Swift:
  http://www.ci.uchicago.edu/swift/main/
Symbolics Lisp Machine
Symbolics Lisp Machine

- Instruction set: stack machine, similar to SECD.
- Hardware support:
  - virtual memory
  - garbage collection
  - data type testing
Symbolics Lisp Machine keyboard

modifier keys: HYPER SUPER META CONTROL
My SECD emulator and mini-Lisp compiler:
https://github.com/carlohamalainen/pysecd