Real Functional Languages

- $\lambda$-calculus defines the semantics of functional languages
- $\lambda$-calculus (and therefore any abstraction of $\lambda$-calculus like pure Lisp) can be implemented in $\lambda$-calculus
- But how can we practically implement $\lambda$-calculus or another functional language on real hardware
- The basic unit of representation in a digital computer is not the $\lambda$-function
The SECD Machine

- Java Virtual Machine implements simple underlying operations for imperative and object-oriented languages
- Simple Machine implemented on dozens of platforms
- SECD machine is a virtual machine for functional languages
  - used in many implementations (LispMe for Palm Pilot)
- SECD implements
  - primitives for
    - values like integers - represented by bits as usual
    - composite structures like cons cells - represented by 2-element pointer vectors and
  - four special internal registers to represent computation state
  - operations to carry out computations
  - heap of memory cells

Stacks

- The SECD is a stack-based computer (like postscript or fourth)
- Stacks are represented as a list
  - $L = (s_1 \ s_2 \ s_3 \ s_4 \ldots \ s_n)$
- A dot in a list introduces its tail
  - Let $R = (s_2 \ s_3 \ s_4 \ldots \ s_n)$
  - Then $L = (s_1 \ . \ R)$
- We can easily refer to the first $m$ elements of a stack as
  - $(s_1 \ s_2 \ldots \ s_m \ . \ <\text{rest}>)$
  - *notice how the dot is used!*

Dr B. Price and Dr. R. Greiner

COMPUT325: SECD Virtual Machine
The SECD Stacks

- 4 Special registers point to 4 stacks
  - S=Scratch (for operands of operations and evaluated results)
  - E=Environment (stack of variable bindings in force)
  - C=Code (stack of primitive operations to execute in the active function)
  - D=Dump (stack of suspended computations)
    - each suspended computation has
      - a stack,
      - environment and
      - code body (S,E,C)

- Items stored in stacks may be atoms, or lists

Simplified Example

- A sample scratch stack with operands for PLUS:
  \[ S = (1 \ 2 \ . \ \text{rest} ) \]

- Result of PLUS is left in place of the operands
  \[ S = (3 \ . \ \text{rest}) \]
The SECD Machine Operations

- The state of the SECD machine is determined by the four stack registers
- SECD Operations transform the stacks from one state to another
- Legal transformations are defined by rewrite rules
- When the left side of the rule matches the state of the machine, the machine switches to the state given by the right side of the rule
  \[ \text{secd} \rightarrow \text{se}' \text{c}' \text{d}' \]

Simple SECD Program I

- Program = \(<\text{list of primitive functions}> + \langle\text{immediate operands}\rangle\>
- A simple program to load the constants 3 and 5 onto the scratch stack
  \((\text{LDC 3 LDC 5})\)
  - Here, LDC is the primitive function "Load Constant"
  - And 3 is an immediate operand
- The machine starts with the program loaded on the code stack
  \((\text{secd}) = (\text{nil nil (LDC 3 LDC 5).nil nil})\)
- Programs are processed one operation at a time using rewrite rules
Simple SECD Program II

- The rewrite rule for LDC is
  \[ s \cdot e (LDC\ x\ .\ c)\ d \rightarrow x \cdot s \cdot e \cdot c \cdot d \]
- The constant \( x \) is pushed onto the front of the scratch stack \( s \)
- The LDC \( x \) operation is popped off of the code stack, leaving its tail \( c \)
- Execution of our simple program yields:
  \[ s \cdot e (LDC\ 3\ LDC\ 5)\ .\ c\ d \]
  \[ 3 \cdot s \cdot e (LDC\ 5)\ .\ c\ d \]
  \[ (5\ 3) \cdot s \cdot e \cdot c\ d \]

Additional Operations

- Typical arithmetic operations: ADD, SUB, MUL, DIV, REM, etc.
  
  addition:
  \[
  (m\ n\ .\ s)\ \cdot\ e\ (ADD\ .\ c)\ \cdot\ d
  \rightarrow (p\ .\ s)\ \cdot\ e\ \cdot\ c\ \cdot\ d
  ;;\ \text{where}\ p=m+n
  \]
- Relational functions such as \( = \), \( > \), \( < \) are also defined
  
  compare:
  \[
  (m\ n\ .\ s)\ \cdot\ e\ (>\ .\ c)\ \cdot\ d
  \rightarrow (b\ .\ s)\ \cdot\ e\ \cdot\ c\ \cdot\ d
  ;;\ \text{where}\ b=T\ \text{if}\ m>n\ \text{else}\ b=F
  \]
More Complex Example

\[ \rightarrow s \quad e \ (LDC \ 3 \ LDC \ 5 \ ADD \ LDC \ 10 \ >) \ d \]
\[ \rightarrow 3. s \quad e \ (LDC \ 5 \ ADD \ LDC \ 10 \ >) \ d \]
\[ \rightarrow (5 \ 3). s \quad e \ (ADD \ LDC \ 10 \ >) \ d \]
\[ \rightarrow 8. s \quad e \ (LDC \ 10 \ >) \ d \]
\[ \rightarrow 10 \ 8 . s \quad e \ (>\ ) \ d \]
\[ \rightarrow T . s \quad e \ nil \ d \]

Branching : SEL and JOIN

- IF statement functionality is implemented with the select and join operations.

- Select (SEL)
  - chooses between two subprograms and
  - suspends remainder of main program by putting it on the dump stack

- Let T be 'true' and F be 'false'
  - IF T is on the stack, do subprogram \( \langle C1 \rangle \) store rest of code cr

\[ (T . s) \quad e \ (SEL \ \langle C1 \rangle \ \langle C2 \rangle . \ cr) \ d \ ; ; \ s e c d \]
\[ \rightarrow s \quad e \ \langle C1 \rangle \ (cr . \ d) \]

- IF F is on the stack, do subprogram \( \langle C2 \rangle \) store rest of code cr

\[ (F . s) \quad e \ (SEL \ \langle C1 \rangle \ \langle C2 \rangle . \ cr) \ d \]
\[ \rightarrow s \quad e \ \langle C2 \rangle \ (cr . \ d) \]
"Un-branching": JOIN

- Join restores the suspended main program from the dump stack
  \[ s \leftarrow (JOIN . c) \rightarrow s \] (cr . d) \rightarrow s \ e \ cr \ d
- SEL and JOIN work together to implement "IF" behaviour
- An example of an abstract IF and the equivalent SECD code:

```plaintext
IF 5 > 3
  THEN m-n
  ELSE 0
LDC 8
≡ (LDC 3 LDC 5 > SEL

;; SEL applied to result of 3 > 5
  (LDC m LDC n SUB JOIN)
  (LDC 0 JOIN)
  LDC 8)
```

- Unlike assembler, programs may have nested structures

**Complex Branching Example**

```plaintext
s \ e (LDC 3 LDC 5 > SEL (LDC m LDC n SUB JOIN)
     (LDC 0 JOIN) LDC 8).c d
3.s \ e (LDC 5 > SEL (LDC m LDC n SUB JOIN)
       (LDC 0 JOIN) LDC 8 ).c d
5 3.s \ e ( > SEL (LDC m LDC n SUB JOIN)
         (LDC 0 JOIN) LDC 8 ).c d
T.s \ e (SEL (LDC m LDC n SUB JOIN)
       (LDC 0 JOIN) LDC 8 ).c d
s \ e (LDC m LDC n SUB JOIN) ((LDC 8).c d)
m.s \ e (LDC n SUB JOIN) ((LDC 8).c d)
(n m).s \ e (SUB JOIN) ((LDC 8).c d)
p.s \ e (JOIN) ((LDC 8).c d) ;; where p = n-m
p.s \ e (LDC 8).c d ;; where p = n-m
(8 p).s e c d ;; where p = n-m
```

Dr B. Price and Dr. R. Greiner  COMPUT325: SECD Virtual Machine
List Operations

NIL adds nil to scratch stack
\[ s \ e \ (\text{NIL} \ . \ c) \ d \rightarrow \]
\[ (\text{NIL} \ . \ s) \ e \ c \ d \]

CONS replaces top two elements of s with their consed pair
\[ (x \ y \ . \ s) \ e \ (\text{CONS} \ . \ c) \ d \rightarrow \]
\[ (x . y).s \ e \ c \ d \]

CAR replaces top element with its CAR
\[ (x . y) . s \ e \ (\text{CAR} \ . \ c) \ d \rightarrow \]
\[ x . s \ e \ c \ d \]

CDR replaces top element with its CDR
\[ (x . y) . s \ e \ (\text{CDR} \ . \ c) \ d \rightarrow \]
\[ y . s \ e \ c \ d \]

List Operators Example

\[ s \ e \ (\text{LDC} \ 1 \ \text{LDC} \ 2 \ \text{CONS} \ \text{CDR}).c \ d \]
\[ \rightarrow 1 . s \ e \ (\text{LDC} \ 2 \ \text{CONS} \ \text{CDR}).c \ d \]
\[ \rightarrow 2 \ 1 . s \ e \ (\text{CONS} \ \text{CDR}).c \ d \]
\[ \rightarrow (2 \ 1).s \ e \ (\text{CDR}).c \ d \]
\[ \rightarrow 1 . s \ e \ c \ d \]
SECD User Defined Functions

- SECD, being stack based, put args on stack, then applies function

- The procedure for definition of user-functions is also backwards in spirit:
  - \(\lambda\)-calculus application: \((\langle\text{function-def}\rangle\langle\text{arguments}\rangle)\)
  - SECD application: \(\langle\text{arguments}\rangle\langle\text{function-def}\rangle\text{AP}\)

SECD Function Definition Idiom

- Roughly
  - Construct list of function arguments on the scratch stack
    - Cons together loaded constants "LDC" or results of prior computations
  - Create closure = (function, environment) and put on scratch stack "LDF"
  - Eval closure using apply "AP"
    - saves current scratch, code and environment stacks
    - create fresh scratch stack for new function
    - make new environment from closure environment plus stack arguments
    - create code stack from closure function
    - when needed, copy arguments from env and put on scratch
    - do computation and leave results on scratch stack
  - Restore stacks
Creating Arguments on Scratch

- This is just ordinary list construction

- To pass the arguments 1, 2 to a user function, create list \((1\ 2)\)

\[
\text{NIL} \quad ;; \quad \text{nil} . \ s \\
\text{LDC 2} \quad ;; \quad 2\ \text{nil} . \ s \\
\text{CONS} \quad ;; \quad (2) . \ s \\
\text{LDC 1} \quad ;; \quad 1\ (2) . \ s \\
\text{CONS}) \quad ;; \quad (1\ 2) . \ s
\]

- The argument list as a whole is typically called \(v\)

- We could represent scratch with arg list on top as: \(v . s\)

Load Function: LDF

- Create cons cell \(\langle f . e \rangle\) to hold closure on scratch stack
  - Copy function \(f\) from code to car of closure
  - Copy current environment \(e\) to cdr of closure

- The rewrite rule is:

\[
v.s \quad e \quad \text{(LDF}\ f .\ c) \quad d \quad \rightarrow \quad ;;\ s\ e\ c\ d \\
((f . e) \ v.s) \quad e \quad c \quad d
\]

- Could create as many closures with environment \(e\) as desired
Apply Function: AP

- Saves current machine state onto dump stack
- Installs code from closure $f$ into code stack
- Creates new environment consisting of arguments from control stack $v$ + environment saved in closure $e'$

$$%
((f.e') \ v.s) \ e \ (AP \ c) \ d \to \NIL \ v.e' \ f \ s \ e \ c \ . \ d
%
$$

- Notice that the environment takes the form of a list of lists of values
- The first list, $v$, being the arguments and the second list, $e'$, being the lexical environment $f$ was defined in
- The code for $f$ has been put on the code stack
- The original scratch, env and code stacks are saved on dump

Retrieving Arguments and Variables

- The "LD" function retrieves values from environment
- Values are not retrieved by name, but by index in the list
- Like compiling an identifier to a "relative" memory address.
- Suppose the value 'x' is stored in slot j of nested environment $i$

$$%
s\ e\ (LD\ (i.j).c)\ d\to\ (x.s)\ e\ c\ d\ ;;\ where\ x = locate((i.j),\ e)
%
$$

- Locate returns the the $j^{th}$ value from the $i^{th}$ list of environment $e$
- Retrieve $j^{th}$ immediate argument with LD (1,j)
- Retrieve $j^{th}$ variable in immediate environment with LD (2,j)
- Arguments reside in a local environment for the called function
Returning from a Function Call

- The RTN function restores the machine state on call completion
  - Copies_saved_stacks from dump back to original registers
  - Pushes returned value x from function onto top of restored stack

\[ x . s' \ e' \ \text{RTN.c'} \ s \ e \ c \ . \ d \to \]
\[ x . s \ e \ c \ d \]

- Returning function’s stacks are discarded, except:
  - Value returned by function is copied to head of restored scratch

- Other stacks restored to pre-call state

Compiling a Function Application

- The square function applied to 3

\[
\text{(square 3)}
\]
\[
\text{(NIL LDC 3 CONS)} \quad ;; \text{build arguments}
\]
\[
\text{LDF ( LD (1.1)} \quad ;; \text{code for square in a sublist}
\text{LD (1.1)} \quad ;; \text{code loads 2 copies of arg from env}
\text{MUL} \quad ;; \text{then multiplies}
\text{RTN )}
\]
\[
\text{AP)} \quad ;; \text{apply square function}
\]
Evaluating a Function Call

Let $F = (\text{LD (1.1) LD (1.1) MUL RTN})$ so that the application of square is

\[
\begin{align*}
\text{s} & \quad e \ (\text{NIL LDC 3 CONS LDF F AP}).c \quad d \quad ;; \ s \ e \ c \ d \\
\text{nil.s} & \quad e \ (\text{LDC 3 CONS LDF F AP}).c \quad d \\
3 \text{ nil.s} & \quad e \ (\text{CONS LDF F AP}).c \quad d \\
(3).s & \quad e \ (\text{LDF F AP}).c \quad d \\
(F.e) \ (3).s & \quad e \ (\text{AP}).c \quad d \\
\text{nil} & \quad (3).e \ F \quad ((s \ e \ c) \ . \ d)
\end{align*}
\]

Evaluating Function Body and Returning

\[
\begin{align*}
\text{nil} \ (3).e \ F \quad ((s \ e \ c) \ . \ d)
\end{align*}
\]

RECALL: $F = (\text{LD (1.1) LD (1.1) MUL RTN})$

\[
\begin{align*}
\text{nil} \ (3).e \ (\text{LD (1.1) LD (1.1) MUL RTN}) \ (s \ e \ c \ . \ d)
\end{align*}
\]

\[
\begin{align*}
3 \quad (3).e \ (\text{LD (1.1) MUL RTN}) \ (s \ e \ c \ . \ d)
\end{align*}
\]

\[
\begin{align*}
3 \ 3 \quad (3).e \ (\text{MUL RTN}) \ (s \ e \ c \ . \ d)
\end{align*}
\]

\[
\begin{align*}
9 \quad (3).e \ (\text{RTN}) \ (s \ e \ c \ . \ d)
\end{align*}
\]

\[
\begin{align*}
9.s \ e \ c \ d
\end{align*}
\]
Named Functions

- In the previous example we apply an immediate function.
- Generally we want to apply named functions. Let square(x) = x*x IN square(3).
- This is equivalent to $(\lambda f \mid f(3)) (\lambda x \mid x*x)$.

Repeated:

$(\lambda f \mid f(3)) (\lambda x \mid x*x)$

Thus, we must apply the function body as an argument to a $\lambda$ in order to name it.

NIL
LDF (LD (1.1) LD (1.1) MUL RTN) ; square: $<(\lambda y y*y),e>$
CONS ; scratch: $<(\text{square}.e>).s$
LDF (
  NIL LDC 3 CONS ; arg on stack (3)
  LD (1 . 1) ; retrieve closure $\langle\text{square}.e\rangle$
  AP ; app closure to arg
  RTN)
;; scratch: $(\lambda f \mid f 3) (\langle\text{square}.e\rangle).s$
AP ;; apply f to $(\langle\text{square}.e\rangle)$
Trace of Named Functions I

Let $X = (\text{LD (1.1) LD (1.1) MUL RTN})$
Let $F = (\text{NIL LDC 3 CONS LD (1.1) AP RTN})$

```
s e (\text{NIL LDF X CONS LDF F AP}).c d
NIL.s e (\text{LDF X CONS LDF F AP}).c d
(X.e) NIL.s e (\text{CONS LDF F AP}).c d
((X.e)).s e (\text{LDF F AP}).c d ;; closure as arg
```

;; Now have closure and arguments on top of scratch stack

```
(F.e) ((X.e)).s e (\text{AP}).c d
nil ((X.e) e) F (s e c.d)
```

;; Note, closure for $X$ is 1st value in first frame

Trace of Named Functions II

Recall $F = (\text{NIL LDC 3 CONS LD (1.1) AP RTN})$

;; We omit dump parameter here:

```
nil ((X.e) e) F
nil ((X.e) e) (\text{NIL LDC 3 CONS LD (1.1) AP RTN})
nil.nil ((X.e) e) (\text{LDC 3 CONS LD (1.1) AP RTN})
3 nil.nil ((X.e) e) (\text{CONS LD (1.1) AP RTN})
(3).nil ((X.e) e) (\text{LD (1.1) AP RTN})
```

;; We load closure from $X$ from environment (ie the SQUARE function)

```
(x.e) (3).nil ((X.e) e) (\text{AP RTN})
```

:::Evaluation of square now proceeds as in the anonymous case

Dr B. Price and Dr. R. Greiner COMPUT325: SECD Virtual Machine
Recursive Functions

- As in meta-interpretation of \( \lambda \)-calculus, we build a self-referencing closure

\[
\text{LETREC } f(x) = \langle \text{BODY} \rangle \text{ IN } f(y)
\equiv (\lambda f \mid f(y)) \langle \text{BODY} \rangle
\]

- When we pass function body in, we use the closure mechanism

\[
C \equiv < \langle \text{BODY} \rangle, f \leftarrow C > \
E \equiv < f(v), f \leftarrow C >
\]

Recursive Functions

- Build a self-referencing environment in two steps
  - DUM creates an unused slot in the environment
    (\texttt{NIL} is used to fill the slot, but this is irrelevant)

\[
\begin{align*}
\text{se} \ (\text{DUM} \ \text{LDF} \ F \ . \ c) \ d \\
\rightarrow & \text{se} \ \texttt{NIL}.e \ (\text{LDF} \ F \ . \ c) \ d \\
\rightarrow & (F.\texttt{NIL}.e) \ \texttt{NIL}.e \ c \ d
\end{align*}
\]
  - \texttt{RAP} (recursive apply) calls rplaca to assign slot to be \( v \)

\[
\begin{align*}
((f.\texttt{NIL}.e') \ v.s) & \quad (\texttt{NIL}.e) \\
\texttt{NIL} & \quad \text{rplaca}((\texttt{NIL}.e'), v) \ f \quad \texttt{(s e c.d)} \\
\texttt{NIL} & \quad (v.e') \quad f \quad \texttt{(s e c.d)}
\end{align*}
\]
Recursive Length

(letrec (f (λx m | (if (null x) m (f (cdr x) (+ m 1) )) )
(f '(1 2 3) 0 )
(DUM ;; (nil . e)
NIL LDF( ;; (λx m | ... 
LD (1.1) NULL SEL ;; if null x
(LD (1.2) JOIN) ;; then return m
;; else
(NIL LDC 1 ld (1.2) ADD CONS ;; form (q) where q=m+1
LD (1.1) CDR CONS ;; form (z q) where z=(cdr x)
LD (2.1) AP JOIN) ;; Apply f to (z q)
RTN)
CONS ;; Arg list contains closure: ( (F.e) ) . s
LDF ;; (λf | ..
(NIL LDC 0 CONS LDC (1 2 3) CONS LD (1.1) ;; (F (1 2 3) 0)
AP RTN)
;; f v .s ≡ (λf.(nil . e)) ( (λx m.(nil . e)) ) .s
RAP) ;; ≡(rplca (nil . e) v), where v= ( (λx m.(nil . e)) )

Recursive Length Notes

▶ The key to the previous is example is the last two lines:
;;;; f v .s ≡ (λf.(nil . e)) ( (λx m.(nil . e)) )
RAP) ;; ≡(rplca (nil . e) v), where v= ( (λx m.(nil . e)) )

▶ Notice that when nil is replaced by v= ( (λx m.(nil . e)) ),
the closure (λx m.(nil . e)) has its first environment
frame pointing back to itself

▶ When this closure is executed, the arguments the closure are
called on will become the new first frame

▶ The self referencing point will become the first argument of
the second frame (i.e., LD (2.1))
Recursive Functions: Fact

Suppose we were interested in this code:

\[
\begin{align*}
\text{let } & x = 3 \text{ and one } = 1 \text{ in} \\
\text{letrec } & f(n, m) = \\
& \quad \text{if } (\text{eq } n \ 0) \text{ then one } \\
& \quad \quad \text{else } f(n - 1, n \times m) \\
\text{in } & f(x, \text{one})
\end{align*}
\]

This translation is not quite right: \( f \) cannot refer to itself:

\[
\begin{align*}
(\lambda x, \text{one } | \\
\quad (\lambda f | f(x, \text{one})) \\
\quad \quad (\lambda n \  m \ |
\quad \quad \quad \text{if } (\text{eq } n \ 0) \text{ then one } \text{ else } f(n - 1, n \times m))
\end{align*}
\]

Recursive Fact in SECD Code

\[
\begin{align*}
\text{(nil ldc 1 cons ldc 3 cons} \\
\text{ldf} \\
\text{(dum} \\
\text{nil} \\
\text{ldf} \\
\quad \text{(ldc 0 ld (1,1) eq sel} \\
\quad \quad \text{(ldc 1 join)} \\
\quad \text{(nil} \\
\quad \quad \text{ld(1,2) ld(1,1) MPY CONS} \\
\quad \quad \text{LD (3,2) LD(1,1) SUB CONS} \\
\quad \quad \text{LD(2,1) AP JOIN}) \\
\quad \quad \text{RTN} \\
\quad \quad \text{CONS} \\
\quad \text{LDF (NIL LD(2.2) CONS} \\
\quad \quad \text{LD (2.1) CONS} \\
\quad \quad \text{LD (1.1) AP RTN} \\
\quad \quad \text{RAP RTN}) \\
\quad \quad \text{AP})
\end{align*}
\]
STOP

- Signals that computation should be halted
- Ending programs with STOP forces programmer to be explicit
- Allows virtual machine to signal an error if it runs out of instructions for some other reason

Practicalities

- We quickly run out of memory without garbage collection
- Variety of collection strategies with different properties:
  - Reference Count
  - Mark and Sweep
  - Generation Scavenging (Baker’s algorithm)