COMPUT325: SECD Virtual Machine

Dr B. Price and Dr. R. Greiner

4th November 2004
Real Functional Languages

- \( \lambda \)-calculus defines the semantics of functional languages
Real Functional Languages

- λ-calculus defines the semantics of functional languages
- λ-calculus (and therefore any abstraction of λ-calculus like pure Lisp) can be implemented in λ-calculus
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
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.
The SECD Machine

- Java Virtual Machine implements simple underlying operations for imperative and object-oriented languages
- Simple Machine implemented on dozens of platforms

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
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)
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
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
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
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
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
The SECD is a stack-based computer (like postscript or fourth)
Stacks

- The SECD is a stack-based computer (like postscript or fourth)
- Stacks are represented as a list
  - \( L = (s_1 \ s_2 \ s_3 \ s_4 \ldots \ s_n) \)
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 $L = (s_1 . \ R)$
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!*
The SECD Stacks

- 4 Special registers point to 4 stacks
The SECD Stacks

- 4 Special registers point to 4 stacks
  - S=Scratch (for operands of operations and evaluated results)
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)
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)
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)
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
A sample scratch stack with operands for PLUS:
\[ S=\langle 1, 2, \text{rest} \rangle \]

Result of PLUS is left in place of the operands
\[ S=\langle 3, \text{rest} \rangle \]
The SECD Machine Operations

- The state of the SECD machine is determined by the four stack registers
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.
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.
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

\[
secd \rightarrow s' e' c' d'
\]
Simple SECD Program I

- Program = <list of primitive functions> + <immediate operands>

- A simple program to load the constants 3 and 5 onto the scratch stack (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 (secd) = (nil nil (LDC 3 LDC 5).nil nil)

- Programs are processed one operation at a time using rewrite rules
Simple SECD Program I

- Program = <list of primitive functions> + <immediate operands>

- A simple program to load the constants 3 and 5 onto the scratch stack
  
  \[(LDC \ 3 \ LDC \ 5)\]

  - Here, LDC is the primitive function "Load Constant"
  - And 3 is an immediate operand
Simple SECD Program I

- Program = <list of primitive functions> + <immediate operands>

- A simple program to load the constants 3 and 5 onto the scratch stack
  
  (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
  
  (s e c d) = (nil nil (LDC 3 LDC 5).nil nil)
Simple SECD Program I

- Program = <list of primitive functions> + <immediate operands>

- A simple program to load the constants 3 and 5 onto the scratch stack
  
  \((\text{LDC } 3 \ \text{LDC } 5)\)

  - 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
The rewrite rule for LDC is

\[ \text{se}(\text{LDC } x . \ c) \ d \rightarrow x.\text{se}c\ d \]
Simple SECD Program II

- The rewrite rule for LDC is
  \[ s \ e \ (\text{LDC} \ x \ . \ c) \ d \rightarrow x.s \ e \ c \ d \]
- The constant \( x \) is pushed onto the front of the scratch stack \( s \)
Simple SECD Program II

- The rewrite rule for LDC is
  \[ s \ e \ (LDC \ x \ . \ c) \ d \rightarrow x.s \ e \ c \ d \]

- 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 \( c \)
The rewrite rule for LDC is

\[ s \ e \ (LDC \ x \ . \ c) \ d \rightarrow x.s \ e \ c \ d \]

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:
The rewrite rule for LDC is
\[ s e (LDC \ x \ . \ c) \ d \rightarrow x.s e c d \]

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:
\[ s e (LDC \ 3 \ LDC \ 5).c \ d \]
The rewrite rule for LDC is

\[ s \ e \ (LDC \ x \ . \ c) \ d \rightarrow x.s \ e \ c \ d \]

The constant \( x \) is pushed onto the front of the scratch stack \( s \).

The \texttt{LDC \ x} operation is popped off of the code stack, leaving its tail \( c \).

Execution of our simple program yields:

\[
\begin{align*}
  &s \ e \ (LDC \ 3 \ LDC \ 5) . c \ d \\
  &3.s \ e \ (LDC \ 5) . c \ d
\end{align*}
\]
Simple SECD Program II

- The rewrite rule for LDC is
  \[ s\ e\ (LDC \ x\. \ c)\ d \rightarrow x.s\ e\ c\ d \]

- 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:
  
  \[
  \begin{align*}
  &s\ e\ (LDC\ 3\ LDC\ 5).c\ d \\
  &3.s\ e\ (LDC\ 5).c\ d \\
  &(5\ 3).s\ e\ c\ d
  \end{align*}
  \]
Additional Operations

- Typical arithmetic operations: ADD, SUB, MUL, DIV, REM, etc.
Typical arithmetic operations: ADD, SUB, MUL, DIV, REM, etc.

addition:

\[(m \ n \ s) \ e (ADD \ c) \ d\]
Typical arithmetic operations: ADD, SUB, MUL, DIV, REM, etc.

addition:

\[(m \ n \ . \ s) \ e \ (\text{ADD} \ . \ c) \ d\]

\[\rightarrow(p \ . \ s) \ e \ c \ d\ ;; \text{where } p=m+n\]
Additional Operations

- Typical arithmetic operations: ADD, SUB, MUL, DIV, REM, etc.

  addition:
  
  \[(m \ n \ . \ s) \ e (ADD \ . \ c) \ d \]
  
  \[\rightarrow (p \ . \ s) \ e c \ d \ ; ; \text{where} \ p=m+n\]

- Relational functions such as = > < are also defined

  compare:

  \[(m \ n \ . \ s) \ e (> \ . \ c) \ d\]
Additional Operations

- Typical arithmetic operations: ADD, SUB, MUL, DIV, REM, etc.

  addition:
  \[(m \ n \ . \ s) \ e \ (\text{ADD} \ . \ c) \ d \]
  \[\rightarrow (p \ . \ s) \ e \ c \ d \ ;; \text{where } p=m+n\]

- Relational functions such as \( = \), \( > \), \( < \) are also defined

  compare:
  \[(m \ n \ . \ s) \ e \ (> \ . \ c) \ d \]
  \[\rightarrow (b \ . \ s) \ e \ c \ d \ ;; \text{where } b=T \text{ if } m>n \text{ else } b=F\]
More Complex Example

\[ \rightarrow s e (LDC \ 3 \ LDC \ 5 \ ADD \ LDC \ 10 \ >) \ d \]
More Complex Example

\[ \rightarrow s \\ \rightarrow 3.s \]

\[ e \left( \text{LDC 3 LDC 5 ADD LDC 10 >} \right) d \]

\[ e \left( \text{LDC 5 ADD LDC 10 >} \right) d \]
More Complex Example

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

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

\[ \rightarrow (5 \ 3).s \quad e \quad (ADD \ LDC \ 10 \ >) \ d \]
More Complex Example

\[ \rightarrow s \quad e \ (LDC \ 3 \ LDC \ 5 \ ADD \ LDC \ 10 \ >) \ d \]
\[ \rightarrow 3.s \quad e \ (LDC \ 5 \ ADD \ LDC \ 10 \ >) \ d \]
\[ \rightarrow (5 \ 3).s \quad e \ (ADD \ LDC \ 10 \ >) \ d \]
\[ \rightarrow 8.s \quad e \ (LDC \ 10 \ >) \ d \]
More Complex Example

\[
\begin{align*}
\rightarrow s & \quad e \ (LDC \ 3 \ LDC \ 5 \ ADD \ LDC \ 10 \ >) \ d \\
\rightarrow 3.s & \quad e \ (LDC \ 5 \ ADD \ LDC \ 10 \ >) \ d \\
\rightarrow (5 \ 3).s & \quad e \ (ADD \ LDC \ 10 \ >) \ d \\
\rightarrow 8.s & \quad e \ (LDC \ 10 \ >) \ d \\
\rightarrow 10 \ 8 .s & \quad e \ (>) \ d
\end{align*}
\]
More Complex Example

\[
\begin{align*}
\rightarrow s & \quad e \ (\text{LDC 3 LDC 5 ADD LDC 10 >}) \ d \\
\rightarrow 3.s & \quad e \ (\text{LDC 5 ADD LDC 10 >}) \ d \\
\rightarrow (5 \ 3).s & \quad e \ (\text{ADD LDC 10 >}) \ d \\
\rightarrow 8.s & \quad e \ (\text{LDC 10 >}) \ d \\
\rightarrow 10 \ 8.s & \quad e \ (>) \ d \\
\rightarrow T.s & \quad e \ \text{nil} \ d
\end{align*}
\]
Branching: SEL and JOIN

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

Let \( T \) be 'true' and \( F \) be 'false'

- IF \( T \) is on the stack, do subprogram \( \langle C_1 \rangle \) store rest of code \( \text{cr}(T . s) \text{sel}(\langle C_1 \rangle \langle C_2 \rangle . \text{cr}) \rightarrow s \langle C_1 \rangle (\text{cr}. d) \)

- IF \( F \) is on the stack, do subprogram \( \langle C_2 \rangle \) store rest of code \( \text{cr}(F . s) \text{sel}(\langle C_1 \rangle \langle C_2 \rangle . \text{cr}) \rightarrow s \langle C_2 \rangle (\text{cr}. d) \)
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
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'
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
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

\[(T . s) e (SEL \langle C1 \rangle \langle C2 \rangle . cr) d ;; s e c d\]
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 \texttt{cr}

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

\[
(T . s) e (\text{SEL} \langle C1 \rangle \langle C2 \rangle . cr) d \ ; ; s e c d
tagged
\]

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

\[
(T \cdot s) \se (SEL \langle C1 \rangle \langle C2 \rangle . \cr) \to (T \cdot s) \se (C1) \se (cr \cdot d)
\]

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

\[
(F \cdot s) \se (SEL \langle C1 \rangle \langle C2 \rangle . \cr) \to (F \cdot s) \se (C2) \se (cr \cdot d)
\]
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 \( cr \)

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

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

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

- Join restores the suspended main program from the dump stack

\[ \text{se (JOIN . c) (cr . d)} \rightarrow \text{se cr d} \]
"Un-branching" : JOIN

- Join restores the suspended main program from the dump stack
  \[ s \leftarrow (\text{JOIN . c}) \ (\text{cr . d}) \rightarrow s \ e \ cr \ d \]
- SEL and JOIN work together to implement "IF" behaviour
"Un-branching" : JOIN

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

- Join restores the suspended main program from the dump stack
  
  \[ \text{SE} \ (\text{JOIN} . \ c) \ (\text{cr} . \ d) \rightarrow \text{SE} \ \text{cr} \ \text{d} \]

- SEL and JOIN work together to implement "IF" behaviour

- An example of an abstract IF and the equivalent SECD code:

  IF 5 > 3
  
  THEN m-n
  
  ELSE 0

  LDC 8
"Un-branching" : JOIN

- Join restores the suspended main program from the dump stack

  \[ s e (\text{JOIN} \ . \ c) \ (cr \ . \ d) \rightarrow s e \ cr \ d \]

- 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)
  ```
"Un-branching": JOIN

- Join restores the suspended main program from the dump stack
  \[ \text{SELECT} \ (\text{JOIN} \ . \ c) \ (\text{cr} \ . \ d) \rightarrow \text{SELECT} \ cr \ d \]

- SEL and JOIN work together to implement "IF" behaviour

- An example of an abstract IF and the equivalent SECD code:
  ```
  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

\[
\text{se} \ (\text{LDC 3 LDC 5 > SEL (LDC } m \text{ LDC } n \text{ SUB JOIN)} \\
\text{ (LDC 0 JOIN) LDC 8) cd }
\]
Complex Branching Example

```
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
```
Complex Branching Example

\[
\begin{align*}
  s & \quad e \left( \text{LDC 3 LDC 5} > \text{SEL} \left( \text{LDC m LDC n SUB JOIN} \right) \right. \\
  & \quad \left. \left( \text{LDC 0 JOIN} \right) \text{LDC 8} \right) \text{.c d} \\
  3.s & \quad e \left( \text{LDC 5} > \text{SEL} \left( \text{LDC m LDC n SUB JOIN} \right) \right. \\
  & \quad \left. \left( \text{LDC 0 JOIN} \right) \text{LDC 8} \right) \text{.c d} \\
  5.3.s & \quad e \left( > \text{SEL} \left( \text{LDC m LDC n SUB JOIN} \right) \right. \\
  & \quad \left. \left( \text{LDC 0 JOIN} \right) \text{LDC 8} \right) \text{.c d}
\end{align*}
\]
Complex Branching Example

\[
s \quad e \ (\text{LDC} \ 3 \ \text{LDC} \ 5 \ > \ \text{SEL} \ (\text{LDC} \ m \ \text{LDC} \ n \ \text{SUB} \ \text{JOIN}) \\
\quad \quad \quad (\text{LDC} \ 0 \ \text{JOIN}) \ \text{LDC} \ 8) \cdot c \ d
\]

\[
3.s \quad e \ (\text{LDC} \ 5 \ > \ \text{SEL} \ (\text{LDC} \ m \ \text{LDC} \ n \ \text{SUB} \ \text{JOIN}) \\
\quad \quad \quad (\text{LDC} \ 0 \ \text{JOIN}) \ \text{LDC} \ 8) \cdot c \ d
\]

\[
5 \quad 3.s \quad e \ (\ > \ \text{SEL} \ (\text{LDC} \ m \ \text{LDC} \ n \ \text{SUB} \ \text{JOIN}) \\
\quad \quad \quad (\text{LDC} \ 0 \ \text{JOIN}) \ \text{LDC} \ 8) \cdot c \ d
\]

\[
T.s \quad e \ (\text{SEL} \ (\text{LDC} \ m \ \text{LDC} \ n \ \text{SUB} \ \text{JOIN}) \\
\quad \quad \quad (\text{LDC} \ 0 \ \text{JOIN}) \ \text{LDC} \ 8) \cdot c \ d
\]
Complex Branching Example

\[
\begin{align*}
    s & \quad e \ (LDC \ 3 \ LDC \ 5 > SEL \ (LDC \ m \ LDC \ n \ SUB \ JOIN) \ (LDC \ 0 \ JOIN) \ LDC \ 8).c \ d \\
    3.s & \quad e \ (LDC \ 5 > SEL \ (LDC \ m \ LDC \ n \ SUB \ JOIN) \ (LDC \ 0 \ JOIN) \ LDC \ 8).c \ d \\
    5 \ 3.s & \quad e \ (SEL \ (LDC \ m \ LDC \ n \ SUB \ JOIN) \ (LDC \ 0 \ JOIN) \ LDC \ 8).c \ d \\
    T.s & \quad e \ (SEL \ (LDC \ m \ LDC \ n \ SUB \ JOIN) \ (LDC \ 0 \ JOIN) \ LDC \ 8).c \ d \\
    s & \quad e \ (LDC \ m \ LDC \ n \ SUB \ JOIN) \ (LDC \ 8).c \ d
\end{align*}
\]
Complex Branching Example

\[
\begin{align*}
  s & \quad e \ (LDC \ 3 \ LDC \ 5 \ > \ SEL \ (LDC \ m \ LDC \ n \ SUB \ JOIN) \ (LDC \ 0 \ JOIN) \ LDC \ 8) . c \ d \\
 3.s & \quad e \ (LDC \ 5 \ > \ SEL \ (LDC \ m \ LDC \ n \ SUB \ JOIN) \ (LDC \ 0 \ JOIN) \ LDC \ 8) . c \ d \\
 5 \ 3.s & \quad e \ ( > \ SEL \ (LDC \ m \ LDC \ n \ SUB \ JOIN) \ (LDC \ 0 \ JOIN) \ LDC \ 8) . c \ d \\
 5 \ 3.s & \quad e \ ( > \ SEL \ (LDC \ m \ LDC \ n \ SUB \ JOIN) \ (LDC \ 0 \ JOIN) \ LDC \ 8) . c \ d \\
 5 \ 3.s & \quad e \ ( > \ SEL \ (LDC \ m \ LDC \ n \ SUB \ JOIN) \ (LDC \ 0 \ JOIN) \ LDC \ 8) . c \ d \\
 T.s & \quad e \ (SEL \ (LDC \ m \ LDC \ n \ SUB \ JOIN) \ (LDC \ 0 \ JOIN) \ LDC \ 8) . c \ d \\
 s & \quad e \ (LDC \ m \ LDC \ n \ SUB \ JOIN) \ (LDC \ 8) . c \ d \\
m.s & \quad e \ (LDC \ n \ SUB \ JOIN) \ (LDC \ 8) . c \ d 
\end{align*}
\]
Complex Branching Example

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)
Complex Branching Example

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
Complex Branching Example

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
Complex Branching Example

\[
\begin{align*}
  s & \quad e \ (LDC \ 3 \ LDC \ 5 > \ SEL \ (LDC \ m \ LDC \ n \ SUB \ JOIN) \ (LDC \ 0 \ JOIN) \ LDC \ 8) . c \ d \\
  3.s & \quad e \ (LDC \ 5 > \ SEL \ (LDC \ m \ LDC \ n \ SUB \ JOIN) \ (LDC \ 0 \ JOIN) \ LDC \ 8) . c \ d \\
  5.3.s & \quad e \ ( > \ SEL \ (LDC \ m \ LDC \ n \ SUB \ JOIN) \ (LDC \ 0 \ JOIN) \ LDC \ 8) . c \ d \\
  T.s & \quad e \ (SEL \ (LDC \ m \ LDC \ n \ SUB \ JOIN) \ (LDC \ 0 \ JOIN) \ LDC \ 8) . c \ d \\
  s & \quad e \ (LDC \ m \ LDC \ n \ SUB \ JOIN) \ (LDC \ 8) . c \ d \\
  m.s & \quad e \ (LDC \ n \ SUB \ JOIN) \ (LDC \ 8) . c \ d \\
  (n \ m).s & \quad e \ (SUB \ JOIN) \ (LDC \ 8) . c \ d \\
  p.s & \quad e \ (JOIN) \ (LDC \ 8) . c \ d \ ;; \ where \ p = n - m \\
  p.s & \quad e \ (LDC \ 8) . c \ d \ ;; \ where \ p = n - m \\
  (8 \ p).s & \quad e \ c \ d \ ;; \ where \ p = n - m
\end{align*}
\]
List Operations

NIL adds nil to scratch stack
List Operations

NIL adds nil to scratch stack

\[
\text{s e (NIL . c) d} \rightarrow \text{(NIL . s) e c d}
\]
List Operations

NIL adds nil to scratch stack
\[
\begin{align*}
\text{s} & \quad \text{e} \quad (\text{NIL} \ . \ c) \quad \text{d} \quad \rightarrow \\
(NIL \ . \ s) & \quad \text{e} \quad c \\
\end{align*}
\]

CONS replaces top two elements of s with their consed pair

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

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

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

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

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

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

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

CDR replaces top element with its CDR
List Operations

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

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

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

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

$$\text{se} \ (\text{LDC} \ 1 \ \text{LDC} \ 2 \ \text{CONS} \ \text{CDR})$$
List Operators Example

\[ s \quad e \ (LDC \ 1 \ LDC \ 2 \ CONS \ CDR) . c \ d \]
\[ \rightarrow 1 . s \quad e \ (LDC \ 2 \ CONS \ CDR) . c \ d \]
List Operators Example

\[
\begin{align*}
  s & \rightarrow e \ (LDC\ 1\ \ LDC\ 2\ \ CONS\ \ CDR).c\ d \\
  \rightarrow 1.s & \rightarrow e \ (LDC\ 2\ \ CONS\ \ CDR).c\ d \\
  \rightarrow 2\ 1.s & \rightarrow e \ (CONS\ \ CDR).c\ d
\end{align*}
\]
List Operators Example

```
s e (LDC 1 LDC 2 CONS CDR).c d
→1.s e (LDC 2 CONS CDR).c d
→2 1.s e (CONS CDR).c d
→(2 1).s e (CDR).c d
```
List Operators Example

\[
\begin{align*}
  s & \quad e \ (LDC \ 1 \ LDC \ 2 \ CONS \ CDR).c \ d \\
  \rightarrow 1.s & \quad e \ (LDC \ 2 \ CONS \ CDR).c \ d \\
  \rightarrow 2 \ 1.s & \quad e \ (CONS \ CDR).c \ d \\
  \rightarrow (2 \ 1).s & \quad e \ (CDR).c \ d \\
  \rightarrow 1.s & \quad e \ c \ d
\end{align*}
\]
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: \(\langle\text{arguments}\rangle\langle\text{function-def}\rangle\) AP
SECD Function Definition Idiom

- Roughly

- Roughly
SECD Function Definition Idiom

- Roughly
  - Construct list of function arguments on the scratch stack
  - Cons together loaded constants "LDC" or results of prior computations
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"
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"
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
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
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
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 \(= (\text{function}, \text{environment})\) 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
Roughly

- Construct list of function arguments on the scratch stack
  - Cons together loaded constants "LDC" or results of prior computations
- Create closure \((\text{function}, \text{environment})\) 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
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
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)

LDC 2 ;; 2 nil . s
CONS ;; (2) . s
LDC 1 ;; 1 (2) . s
CONS) ;; (1 2) . s
Creating Arguments on Scratch

- This is just ordinary list construction

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

(NIL ;; nil . s
Creating Arguments on Scratch

- This is just ordinary list construction

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

```
(NIL ;; nil . s
LDC 2 ;; 2 nil . s
```

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

We could represent scratch with argument list on top as:

\( v.s \)
Creating Arguments on Scratch

- This is just ordinary list construction

- To pass the arguments 1, 2 to a user function, create list (1 2)
  
  \[
  \text{(NIL ;; nil . s)}
  \]
  \[
  \text{LDC 2 ;; 2 nil . s}
  \]
  \[
  \text{CONS ;; (2) . s}
  \]
Creating Arguments on Scratch

- This is just ordinary list construction

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

```plaintext
(NIL ;; nil . s
LDC 2 ;; 2 nil . s
CONS ;; (2) . s
LDC 1 ;; 1 (2) . s
```
Creating Arguments on Scratch

- This is just ordinary list construction

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

  (NIL ;; nil . s
  LDC 2 ;; 2 nil . s
  CONS ;; (2) . s
  LDC 1 ;; 1 (2) . s
  CONS) ;; (1 2) . s
Creating Arguments on Scratch

- This is just ordinary list construction

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

  (NIL ;; nil . s
  LDC 2 ;; 2 nil . s
  CONS ;; (2) . s
  LDC 1 ;; 1 (2) . s
  CONS) ;; (1 2) . s

- The argument list as a whole is typically called \texttt{v}
Creating Arguments on Scratch

- This is just ordinary list construction

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

  (NIL ;; nil . s
  LDC 2 ;; 2 nil . s
  CONS ;; (2) . s
  LDC 1 ;; 1 (2) . s
  CONS) ;; (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, (f . e) to hold closure on scratch stack

The rewrite rule is:

\[ v.s_e (LDF f . c) d \rightarrow ;; sec d ((f . e) v.s e c d) \]

Could create as many closures with environment e as desired.
Load Function: LDF

- Create cons cell \( (f . e) \) to hold closure on scratch stack
  - Copy function \( f \) from code to car of closure
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:
$v.s\ e\ (LDF\ f .\ c)\ d\ \rightarrow\ ;;\ s\ e\ (c\ (f .\ e))\ v.s\ e\ c\ d$  
Could create as many closures with environment $e$ as desired
Load Function: LDF

- Create cons cell, \((f \ . \ e)\) 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 (LDF \ f \ . \ c) \quad d \quad \rightarrow \quad ;; \quad s \quad e \quad c \quad d
  
  ((f.e) \ v.s) \quad e \quad c \quad d
  \]
Load Function: LDF

- Create cons cell, \((f \ . \ e)\) 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:
  
  \[
  v.s \quad e \quad (\text{LDF } f \ . \ c) \quad d \quad \rightarrow \quad ;; \quad s \quad e \quad c \quad d
  \]

  \[
  ((f.e) \quad v.s) \quad e \quad c \quad d
  \]

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

- Saves current machine state onto dump stack
Apply Function: AP

- Saves current machine state onto dump stack
- Installs code from closure $f$ into code stack

Notice that the environment takes the form of a list of lists. The first list, $v$, being the arguments and the second list, $e'$, being the lexical environment 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.
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 $e'$

\[
((f.e')\ v.s)\ e\ (AP\ .\ c)\ d\ \rightarrow \\
NIL\ v.e'\ f\ sec\ .\ d
\]
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 $e'$

$$( (f \cdot e') \cdot v \cdot s) \cdot e \quad (AP \cdot c) \cdot d \rightarrow NIL \quad v \cdot e' \quad f \quad s \cdot e \cdot c \cdot d$$

- Notice that the environment takes the form of a list of lists of values
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 $e'$

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

- 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
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 \( e' \)

\[
((f \cdot e') \cdot v \cdot s) \cdot e \rightarrow (f \cdot e') \cdot v \cdot s \cdot e \cdot c \cdot d
\]

- 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
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 \( e' \)

\[
((f.e') \, v.s) \, e \quad (AP \, . \, c) \, d \rightarrow \text{NIL} \quad v.e' \quad f \quad s \quad e \quad c \quad d
\]

- 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 as

\[(x.s)\]  
where \(x = locate((i.j), e)\)

Locate returns the jth value from the ith list of environment e.

Retrieve jth immediate argument with LD (1,j).

Retrieve jth variable in immediate environment with LD (2,j).

Arguments reside in a local environment for the called function.
Retrieving Arguments and Variables

- The "LD" function retrieves values from environment
- Values are not retrieved by name, but by index in the list

\[
\text{Locate returns the the } j\text{th value from the } i\text{th list of environment } e
\]

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
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.
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.

\[
\text{locate}((i.j), e) \rightarrow (x.s) \text{ e c d} \quad \Rightarrow \quad (x.s) \text{ e c d} \quad ;; \text{where } x = \text{locate}((i.j), e)
\]
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
  \[ (LD (i.j).c) d \rightarrow (x.s) e c d \]
  \[ ;; \text{where } x = \text{locate}((i.j), e) \]
- Locate returns the the j\(^{th}\) value from the i\(^{th}\) list of environment e
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
  
  \[ (x.s) \rightarrow (\text{LD}(i.j).c) \]
  
  where \( x = \text{locate}((i.j), e) \)

- Locate returns the \( j^{th} \) value from the \( i^{th} \) list of environment \( e \)

- Retrieve \( j^{th} \) immediate argument with \( \text{LD}(1,j) \)
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
  \[ \text{Locate returns the } j^{th} \text{ value from the } i^{th} \text{ list of environment } e \]
  \[ (x.s) \]
- Retrieve \( j^{th} \) immediate argument with LD (1,j)
- Retrieve \( j^{th} \) variable in immediate environment with LD (2,j)
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

  \[ (x.s) \rightarrow \text{locate}((i.j), e) \]

  \[ (x.s) \rightarrow \text{locate}((i.j), e) \]

  \[ (x.s) \rightarrow \text{locate}((i.j), e) \]

- 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
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

\[
x.s'\ e'\ RTN.c'\ s\ e\ c\ .\ d\ \rightarrow \\
x.s\ e\ c\ \ d
\]
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

$$x.s \ e' \ \text{RTN.c'} \ s \ e \ c \ \cdot \ d \ \rightarrow$$
$$x.s \ e \ c \ d$$

- Returning function’s stacks are discarded, except:
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

$$x\text{'se'} \text{RTN.c's e c . d} \rightarrow$$
$$x\text{s e c d}$$

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

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

- 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
  
  \((\text{square } 3)\)
Compiling a Function Application

- The square function applied to 3
  
  \((\text{square } 3)\)
The square function applied to 3

(square 3)

(NIL LDC 3 CONS ;; build arguments
The square function applied to 3

(square 3)
(NIL LDC 3 CONS) ;; build arguments
LDF ( LD (1.1) ) ;; code for square in a sublist
LD (1.1) ;; code loads 2 copies of arg from env
MUL ;; then multiplies
RTN )
Compiling a Function Application

- The square function applied to 3

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

Let \( F = (LD (1.1) LD (1.1) MUL RTN) \) so that the application of square is
Evaluating a Function Call

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

\[
\text{s e (NIL LDC 3 CONS LDF F AP) . c d ; ; s e c d}
\]
Evaluating a Function Call

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

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

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

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

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

\[
\begin{align*}
\text{s} & \quad e \; \text{NIL} \; \text{LDC} \; 3 \; \text{CONS} \; \text{LDF} \; F \; \text{AP}. \quad \text{c d} \\
\text{nil.s} & \quad e \; \text{LDC} \; 3 \; \text{CONS} \; \text{LDF} \; F \; \text{AP}. \quad \text{c d} \\
3 \; \text{nil.s} & \quad e \; \text{CONS} \; \text{LDF} \; F \; \text{AP}. \quad \text{c d} \\
(3).s & \quad e \; \text{LDF} \; F \; \text{AP}. \quad \text{c d}
\end{align*}
\]
Let $F = (\text{LD } 1.1) \text{ LD } 1.1 \text{ MUL RTN}$ so that the application of square is

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

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

$$
\begin{align*}
\text{s} & \quad e \ (\text{NIL LDC 3 CONS LDF F AP}) \quad \text{c} \quad \text{d} \quad \text{;; s e c d} \\
nil.s & \quad e \ (\text{LDC 3 CONS LDF F AP}) \quad \text{c} \quad \text{d} \\
3 \ \text{nil}.s & \quad e \ (\text{CONS LDF F AP}) \quad \text{c} \quad \text{d} \\
(3).s & \quad e \ (\text{LDF F AP}) \quad \text{c} \quad \text{d} \\
(F.e) \ (3).s & \quad e \ (\text{AP}) \quad \text{c} \quad \text{d} \\
nil & \quad (3).e \quad \text{F} \quad ((\text{s e c}) \ . \ \text{d})
\end{align*}
$$
Evaluating Function Body and Returning

\[
\text{nil} \ (3).e \ F \ ((s \ e \ c) \ . \ d)
\]
Evaluating Function Body and Returning

\[\text{nil } (3).e \ F \ ((s\ e\ c) \ . \ d)\]

RECALL: \(F = (\text{LD (1.1) LD (1.1) MUL RTN})\)
Evaluating Function Body and Returning

\[ \text{nil} \ (3).e \ F \ ((\text{s e c}) \ . \ d) \]

RECALL: \( F = (LD \ (1.1) \ LD \ (1.1) \ MUL \ RTN) \)

\[ \text{nil} \ (3).e \ (LD \ (1.1) \ LD \ (1.1) \ MUL \ RTN) \ (s \ e \ c \ . \ d) \]
Evaluating Function Body and Returning

\[ \text{nil} \ (3) \cdot e \ F \ ((\text{sec}) \cdot d) \]

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

\[ \text{nil} \ (3) \cdot e \ (\text{LD} (1.1) \ \text{LD} (1.1) \ \text{MUL} \ \text{RTN}) \ (\text{sec} \cdot d) \]

\[ 3 \ (3) \cdot e \ (\text{LD} (1.1) \ \text{MUL} \ \text{RTN}) \ (\text{sec} \cdot d) \]
Evaluating Function Body and Returning

\[
\text{nil } (3).e \ F ((s\ e\ c) \ . \ d) \\
\text{RECALL: } F = (LD (1.1) \ LD (1.1) \ MUL \ RTN) \\
\text{nil } (3).e \ (LD (1.1) \ LD (1.1) \ MUL \ RTN) \ (s\ e\ c \ . \ d) \\
3 \ (3).e \ (LD (1.1) \ MUL \ RTN) \ (s\ e\ c \ . \ d) \\
3\ 3 \ (3).e \ (MUL \ RTN) \ (s\ e\ c \ . \ d)
\]
Evaluating Function Body and Returning

\[
\text{nil} \ (3).e \ F \ ((\text{s e c}) \ . \ d)
\]

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

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

\[
3 \ (3).e \ (\text{LD (1.1) MUL RTN}) \ (\text{s e c . d})
\]

\[
3 \ 3 \ (3).e \ (\text{MUL RTN}) \ (\text{s e c . d})
\]

\[
9 \ (3).e \ (\text{RTN}) \ (\text{s e c . d})
\]
Evaluating Function Body and Returning

\[ \text{nil} \ (3).e \ F \ ((\text{sec}) \ . \ d) \]

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

\[ \text{nil} \ (3).e \ (\text{LD} \ (1.1) \ \text{LD} \ (1.1) \ \text{MUL} \ \text{RTN}) \ (\text{sec} \ . \ d) \]

\[ 3 \ (3).e \ (\text{LD} \ (1.1) \ \text{MUL} \ \text{RTN}) \ (\text{sec} \ . \ d) \]

\[ 3 \ 3 \ (3).e \ (\text{MUL} \ \text{RTN}) \ (\text{sec} \ . \ d) \]

\[ 9 \ (3).e \ (\text{RTN}) \ (\text{sec} \ . \ d) \]

\[ 9.\text{sec} \ \text{cd} \]
Named Functions

- In the previous example we apply an immediate function.
Named Functions

- In the previous example we apply an immediate function
- Generally we want to apply named functions

Let \( \text{square}(x) = x \times x \) \( \text{IN} \) \( \text{square}(3) \)
Named Functions

- In the previous example we apply an immediate function
- Generally we want to apply named functions
  
  Let \( \text{square}(x) = x \times x \) IN \( \text{square}(3) \)

- This is equivalent to

  \[
  (\lambda f \mid f(3)) \ (\lambda x \mid x \times x)
  \]
Named Functions

- Repeated:

\[(\lambda f \mid f(3)) \ (\lambda x \mid x*x)\]
Named Functions

- Repeated:
  \[(\lambda f \ | \ f(3)) (\lambda x \ | \ x*x)\]

- Thus, we must apply the function body as an argument to a \(\lambda\) in order to name it
Named Functions

- 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: \(\langle\lambda y|y*y\rangle, e\)
CONS ; scratch: \(\langle\text{square.e}\rangle.s\)
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: \(\langle\lambda f \mid f\ 3\rangle\ (\langle\text{square.e}\rangle)\ .s\)
AP ;; apply f to \(\langle\text{square.e}\rangle\)
Trace of Named Functions I

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

\[
\begin{align*}
\text{s} & \quad e \quad (NIL LDF X CONS LDF F AP).c \quad d \\
\text{NIL.s} & \quad e \quad (LDF X CONS LDF F AP).c \quad d \\
(X.e) \text{ NIL.s} & \quad e \quad (CONS LDF F AP).c \quad d \\
((X.e)).s & \quad e \quad (LDF F AP).c \quad d \quad ;; \text{ closure as arg}
\end{align*}
\]

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

\[
\begin{align*}
(F.e) \quad ((X.e)).s & \quad e \quad (AP).c \quad d \\
nil & \quad ((X.e) e) \quad F \quad (s \quad e \quad c.d)
\end{align*}
\]

;; Note, closure for X is 1st value in first frame
Trace of Named Functions I

Let $X = (\text{LD } (1.1) \ \text{LD } (1.1) \ \text{MUL} \ \text{RTN})$

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

\[
\begin{align*}
\text{s} & \quad \text{e} & (\text{NIL} \ \text{LDF} \ X \ \text{CONS} \ \text{LDF} \ F \ \text{AP}).c & d \\
\text{NIL.s} & \quad \text{e} & (\text{LDF} \ X \ \text{CONS} \ \text{LDF} \ F \ \text{AP}).c & d \\
(X.e) \ \text{NIL.s} & \quad \text{e} & (\text{CONS} \ \text{LDF} \ F \ \text{AP}).c & d \\
((X.e)).s & \quad \text{e} & (\text{LDF} \ F \ \text{AP}).c & d ;; \text{ closure as arg} \\
\text{NIL} & \quad \text{e} & (\text{AP}).c & d \\
\text{NIL} & \quad ((X.e)) & F & (s \ e \ c.d) \\
\end{align*}
\]

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

;; Note, closure for $X$ is 1st value in first frame
Let $X = (\text{LD} (1.1) \text{LD} (1.1) \text{MUL} \text{RTN})$

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

\[
\begin{align*}
\text{s} & \quad \text{e} \\
\text{NIL} & \quad \text{e} \\
(X) & \quad \text{NIL} \\
((X)) & \quad \text{NIL} \\
\text{NIL} & \quad \text{e} \\
\text{NIL} & \quad \text{e} \\
\text{NIL} & \quad \text{e} \\
\text{NIL} & \quad \text{e}
\end{align*}
\]

\(\text{NIL} \text{LDF} X \text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{LDF} X \text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\]

\(\text{NIL} \text{LDF} X \text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{LDF} X \text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\]

\(\text{NIL} \text{LDF} X \text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{LDF} X \text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\]

\(\text{s sure as arg arg}

\(\text{NIL} \text{LDF} X \text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{LDF} X \text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\]

\(\text{s sure as arg arg}

\(\text{NIL} \text{LDF} X \text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{LDF} X \text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\]

\(\text{s sure as arg arg}

\(\text{NIL} \text{LDF} X \text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{LDF} X \text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\]

\(\text{s sure as arg arg}

\(\text{NIL} \text{LDF} X \text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{LDF} X \text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\]

\(\text{s sure as arg arg}

\(\text{NIL} \text{LDF} X \text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{LDF} X \text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad (\text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\quad \text{NIL} \quad (\text{CONS} \text{LDF} F \text{AP}).c \quad d
\]
Trace of Named Functions I

Let $X = (\text{LD (1.1)} \ \text{LD (1.1)} \ \text{MUL} \ \text{RTN})$

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

\[
\begin{align*}
s & \quad e \quad (\text{NIL} \ \text{LDF} \ X \ \text{CONS} \ \text{LDF} \ F \ \text{AP}) \ . c \quad d \\
\text{NIL}.s & \quad e \quad (\text{LDF} \ X \ \text{CONS} \ \text{LDF} \ F \ \text{AP}) \ . c \quad d \\
(X.e).s & \quad e \quad (\text{CONS} \ \text{LDF} \ F \ \text{AP}) \ . c \quad d \\
((X.e)).s & \quad e \quad (\text{LDF} \ F \ \text{AP}) \ . c \quad d \quad ;; \text{closure as arg}
\end{align*}
\]

\[
\begin{align*}
\text{;; Now have closure and arguments on top of scratch stack}
\end{align*}
\]

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

\[
\begin{align*}
\text{;; Note, closure for X is 1st value in first frame}
\end{align*}
\]
Trace of Named Functions I

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

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

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

$\text{;; Note, closure for X is 1st value in first frame}$
Trace of Named Functions I

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

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

\[ ;; \text{Now have closure and arguments on top of scratch stack} \]

\[ (F.e) \quad ((X.e)).s \quad e \quad (AP).c \quad d \]
\[ nil \quad ((X.e) \ e) \quad F \quad (s \ e \ c.d) \]

\[ ;; \text{Note, closure for X is 1st value in first frame} \]
Trace of Named Functions I

Let $X = (\text{LD} (1.1) \ \text{LD} (1.1) \ \text{MUL} \ \text{RTN})$

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

$\langle (\text{NIL} \ \text{LDF} \ X \ \text{CONS} \ \text{LDF} \ F \ \text{AP}).c \ d \rangle$

$\langle (\text{LDF} \ X \ \text{CONS} \ \text{LDF} \ F \ \text{AP}).c \ d \rangle$

$\langle (\text{CONS} \ \text{LDF} \ F \ \text{AP}).c \ d \rangle$

$\langle (\text{LDF} \ F \ \text{AP}).c \ d \rangle$ ;; closure as arg

$\langle (\text{AP}).c \ d \rangle$

$\langle \text{NIL} \ ((X.e) \ e) \ F \ (s \ e \ c.d) \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$

$\langle \rangle$
Trace of Named Functions I

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

$s \quad e \quad (\text{NIL} \ \text{LDF} \ X \ \text{CONS} \ \text{LDF} \ F \ \text{AP}).c \quad d$
$\text{NIL}.s \quad e \quad (\text{LDF} \ X \ \text{CONS} \ \text{LDF} \ F \ \text{AP}).c \quad d$
$(X.e) \ \text{NIL}.s \quad e \quad (\text{CONS} \ \text{LDF} \ F \ \text{AP}).c \quad d \quad ;; \text{close-}$
$(X.e)).s \quad e \quad (\text{LDF} \ F \ \text{AP}).c \quad d \quad ;; \text{clo-}$
sure \text{as} \text{arg}$

$\text{NIL} \quad ((X.e)).s \quad e \quad (\text{AP}).c \quad d$
$\text{nil} \quad (\text{AP}).c \quad d \quad ;; \text{Note, closure for } X \text{ is } 1^{st} \text{ value in first frame}$

$(F.e) \quad ((X.e)).s \quad e \quad (\text{AP}).c \quad d$
$\text{NIL} \quad ((X.e) \ e) \ F \quad (s \ e \ c.d)$
Trace of Named Functions II

Recall \( F = (\text{NIL} \ \text{LDC} \ 3 \ \text{CONS} \ \text{LD} \ (1 \ . \ 1) \ \text{AP} \ \text{RTN}) \);

;;; We omit dump parameter here:

\[\text{nil} \ (\text{X}.\text{e}) \ \text{e}\]
\[\text{nil} \ (\text{X}.\text{e}) \ \text{e}\]
\[\text{nil} \ . \ (\text{X}.\text{e}) \ \text{e}\]
\[\text{LDC} \ 3 \ \text{CONS} \ \text{LD} \ (1 \ . \ 1) \ \text{AP} \ \text{RTN}\]
\[\text{nil} \ . \ (\text{X}.\text{e}) \ \text{e}\]
\[\text{CONS} \ \text{LD} \ (1 \ . \ 1) \ \text{AP} \ \text{RTN}\]
\[\text{nil} \ . \ (\text{X}.\text{e}) \ \text{e}\]
\[\text{LD} \ (1 \ . \ 1) \ \text{AP} \ \text{RTN}\]

;;; We load closure from \( X \) from environment (ie the SQUARE function)
\[\text{nil} \ . \ (\text{X}.\text{e}) \ \text{e}\]
\[\text{AP} \ \text{RTN}\]

;;; Evaluation of square now proceeds as in the anonymous case above
Trace of Named Functions II

Recall $F = (\text{NIL\ LDC\ 3\ CONS\ LD\ (1. 1)\ AP\ RTN})$

`; We omit dump parameter here:

\[\text{nil\ ((X.e)\ e)\ F}\]
Trace of Named Functions II

Recall \( F = ( \text{NIL} \ \text{LDC} \ 3 \ \text{CONS} \ \text{LD} \ (1 \ . \ 1) \ \text{AP} \ \text{RTN}) \)

;; We omit dump parameter here:

\[
\begin{align*}
\text{nil} & \quad ((X.e) \ e) \ F \\
\text{nil} & \quad ((X.e) \ e) \ ( \text{NIL} \ \text{LDC} \ 3 \ \text{CONS} \ \text{LD} \ (1 \ . \ 1) \ \text{AP} \ \text{RTN})
\end{align*}
\]
Trace of Named Functions II

Recall $F = (\text{NIL LDC } 3 \text{ CONS LD (1 . 1) AP RTN})$

;; We omit dump parameter here:

\[\text{nil } ((X.e) e) F\]

nil $((X.e) e) (\text{NIL LDC } 3 \text{ CONS LD (1 . 1) AP RTN})$

\[\text{nil.nil } ((X.e) e) (\text{LDC } 3 \text{ CONS LD (1 . 1) AP RTN})\]
Trace of Named Functions II

Recall $F = (\text{NIL} \text{ LDC} 3 \text{ CONS} \text{ LD} (1 \ . \ 1) \text{ AP} \text{ RTN})$

;; We omit dump parameter here:

nil $((X.e)\ e)\ F$

nil $((X.e)\ e)\ (\text{NIL} \text{ LDC} 3 \text{ CONS} \text{ LD} (1 \ . \ 1) \text{ AP} \text{ RTN})$

nil.nil $((X.e)\ e)\ (\text{LDC} 3 \text{ CONS} \text{ LD} (1 \ . \ 1) \text{ AP} \text{ RTN})$

3 nil.nil $((X.e)\ e)\ (\text{CONS} \text{ LD} (1 \ . \ 1) \text{ AP} \text{ RTN})$
Trace of Named Functions II

Recall $F = (\text{NIL} \ \text{LDC} \ 3 \ \text{CONS} \ \text{LD} \ (1 \ . \ 1) \ \text{AP} \ \text{RTN})$

;; We omit dump parameter here:

nil $\ ((X.e) \ e) \ F$

nil $\ ((X.e) \ e) \ (\text{NIL} \ \text{LDC} \ 3 \ \text{CONS} \ \text{LD} \ (1 \ . \ 1) \ \text{AP} \ \text{RTN})$

nil.nil $\ ((X.e) \ e) \ (\text{LDC} \ 3 \ \text{CONS} \ \text{LD} \ (1 \ . \ 1) \ \text{AP} \ \text{RTN})$

3 nil.nil $\ ((X.e) \ e) \ (\text{CONS} \ \text{LD} \ (1 \ . \ 1) \ \text{AP} \ \text{RTN})$

(3).nil $\ ((X.e) \ e) \ (\text{LD} \ (1 \ . \ 1) \ \text{AP} \ \text{RTN})$
Trace of Named Functions II

Recall $F = (\text{NIL LDC 3 CONS LD (1 . 1) AP RTN})$

$;; \text{We omit dump parameter here:}$

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 $(X.e) e) (\text{LD (1 . 1) AP RTN})$

$;; \text{We load closure from } X \text{ from environment (ie the } SQUARE \text{ function)}$

$(x.e) (3).nil ((X.e) e) (\text{AP RTN})$
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 $\text{SQUARE}$ function)
(x.e) (3).nil  $((X.e)\ e)\ (\text{AP RTN})$

`; Evaluation of square now proceeds as in the anonymous case above
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
Recursive Functions

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

- Build a self-referencing environment in two steps
  - DUM creates an unused slot in the environment
    \( \text{NIL} \) is used to fill the slot, but this is irrelevant
    \[
    s \ e \ (\text{DUM} \ \text{LDF} \ F . \ c) \ d \\
    \rightarrow s \ \text{NIL}.e \ (\text{LDF} \ F . \ c) \ d
    \]

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

- Build a self-referencing environment in two steps
  - DUM creates an unused slot in the environment
    (NIL is used to fill the slot, but this is irrelevant)
    \[ s \in (DUM \ LDF \ F \ . \ c) \ d \]
    \[ \rightarrow s \ NIL.e \ (LDF \ F \ . \ c) \ d \]
    \[ \rightarrow (F.NIL.e) \ NIL.e \ c \ d \]
Recursive Functions

- Build a self-referencing environment in two steps
  - DUM creates an unused slot in the environment
    (NIL is used to fill the slot, but this is irrelevant)
    
    \[
    \text{s e (DUM LDF F . c) d} \\
    \rightarrow s \text{ NIL.e (LDF F . c) d} \\
    \rightarrow (F.\text{NIL.e}) \text{ NIL.e c d}
    \]
  - RAP (recursive apply) calls rplaca to assign slot to be v
Recursive Functions

- Build a self-referencing environment in two steps
  - DUM creates an unused slot in the environment
    (NIL is used to fill the slot, but this is irrelevant)
    
    \[
    s \rightarrow (DUM \ LDF \ F \ . \ c) \ d \\
    \rightarrow s \ NIL.e \ (LDF \ F \ . \ c) \ d \\
    \rightarrow (F.NIL.e) \ NIL.e \ c \ d
    \]
  
  - RAP (recursive apply) calls rplaca to assign slot to be v
    
    \[
    ((f.NIL.e') \ v.s) \ (NIL.e) \ (RAP.c) \ d \rightarrow \\
    NIL \ rplaca((NIL.e'),v) \ f \ (s \ e \ c.d)
    \]
Recursive Functions

- Build a self-referencing environment in two steps
  - DUM creates an unused slot in the environment
    \((\text{NIL} \text{ is used to fill the slot, but this is irrelevant})\)
  
  \[
  \text{se} \ (\text{DUM LDF } F \ . \ c) \ d \\
  \rightarrow \text{se} \ \text{NIL.e} \ (\text{LDF } F \ . \ c) \ d \\
  \rightarrow (F.\text{NIL.e}) \ \text{NIL.e} \ c \ d
  \]

- RAP (recursive apply) calls \text{rplaca} to assign slot to be \(v\)

\[
((f.\text{NIL.e'}) \ v.\text{s}) \ (\text{NIL.e}) \quad (\text{RAP.c}) \ d \rightarrow \\
\text{NIL} \quad \text{rplaca}((\text{NIL.e'}),v) \ f \quad (s \ e \ c.\ d) \\
\text{NIL} \quad (v.e') \ f \quad (s \ e \ c.\ d)
\]
Recursive Length

(letrec (f (λx m | (if (null x) m (f (cdr x) (+ m 1) ))) )
  (f '(1 2 3) 0) )
Recursive Length

(letrec (f (λx m | (if (null x) m (f (cdr x) (+ m 1) )) ) )
  (f '(1 2 3) 0 )
(DUM ;; (nil . e)
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
Recursive Length

(letrec (f (λx m | (if (null x) m (f (cdr x) (+ m 1) )) ) )
  (f '(1 2 3) 0))

NIL  LDF(
  LD (1.1) NULL SEL
  (LD (1.2) JOIN)
  (NIL LDC 1 ld (1.2) ADD CONS)
  LD (1.1) CDR CONS
  LD (2.1) AP JOIN)
  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)
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 example is the last two lines:

\[
\text{RAP} \equiv (\text{rplca \ (nil \ . \ e)} \ v), \quad \text{where} \ v= ((\lambda x \ m. (\text{nil} \ . \ e)))
\]

- Notice that when \text{nil} is replaced by \text{v} = ((\lambda x \ m. (\text{nil} \ . \ e))\), the closure \((\lambda x \ m. (\text{nil} \ . \ e))\) 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:

```ml
let x = 3 and one = 1 in
letrec f(n, m) =
  if (eq n 0) then one
  else f(n - one, n × m)
in f(x, one)
```

This translation is not quite right: `f` cannot refer to itself:

```
(λx,one |
  (λf | f(x, one))
  (λn m |
    if (eq n 0) then one else f(n - one, n × m))
) (3 1)
```
Recursive Fact in SECD Code

(nil ldc 1 cons ldc 3 cons
  ldf
  (dum
   nil
   ldf
     (ldc 0 ld (1,1) eq sel
      (ldc 1 join)
      (nil
       ld(1,2) ld(1,1) MPY CONS
       LD (3,2) LD(1,1) SUB CONS
       LD(2,1) AP JOIN)
     RTN)
   CONS ;; create argument of closure (<f.e>)
   LDF (NIL LD(2.2) CONS
    LD (2.1) CONS ;; <f.e> (x,one) . s
    LD (1.1)
    AP RTN) ;; <
   RAP RTN) AP)

;; (λx,one |⟨BODY⟩) (3 1)
;; (λx,one |⟨BODY⟩) (3 1)
;; adds fact closure to scratch
;; adds a null environment
;; inner λ arg list (fact)
;; fact closure <f.e> -> scratch
;; if 0=n
;; then return 1
;; else: recurse
;; n × m
;; n - 1
;; load fact and apply

Dr B. Price and Dr. R. Greiner
COMPUT325: SECD Virtual Machine
Signals that computation should be halted
STOP

- Signals that computation should be halted
- Ending programs with STOP forces programmer to be explicit
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
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)