https://lug.mines.edu/attend
Where did LISP come from?
The Lambda Calculus

- Alonzo Church was researching the foundations of mathematics in the '40s and created the Lambda Calculus
- A way to do math with only function calls

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Think of as...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction</td>
<td>( \lambda x.x )</td>
<td>function definition</td>
</tr>
<tr>
<td>Application</td>
<td>((\lambda x.x \ y))</td>
<td>function call</td>
</tr>
<tr>
<td>(\alpha)-conversion</td>
<td>((\lambda x.x \ \lambda y.x) \rightarrow (\lambda z.z \ \lambda y.x))</td>
<td>renaming variables</td>
</tr>
<tr>
<td>(\beta)-reduction</td>
<td>((\lambda z.z \ \lambda y.x) \rightarrow \lambda y.x)</td>
<td>plugging in expressions</td>
</tr>
<tr>
<td>Currying</td>
<td>(\lambda x.(\lambda y.x))</td>
<td>(f(x, y) = x)</td>
</tr>
</tbody>
</table>
The Lambda Calculus

- Alonzo Church was researching the foundations of mathematics in the '40s and created the Lambda Calculus
- A way to do math with only function calls

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Think of as...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction</td>
<td>( \lambda x.x )</td>
<td>function definition</td>
</tr>
<tr>
<td>Application</td>
<td>((\lambda x.x \ y))</td>
<td>function call</td>
</tr>
<tr>
<td>(\alpha)-conversion</td>
<td>((\lambda x.x \ \lambda y.x) \rightarrow (\lambda z.z \ \lambda y.x))</td>
<td>renaming variables</td>
</tr>
<tr>
<td>(\beta)-reduction</td>
<td>((\lambda z.z \ \lambda y.x) \rightarrow \lambda y.x)</td>
<td>plugging in expressions</td>
</tr>
<tr>
<td>Currying</td>
<td>(\lambda x.(\lambda y.x))</td>
<td>(f(x, y) = x)</td>
</tr>
</tbody>
</table>
The Lambda Calculus

- Alonzo Church was researching the foundations of mathematics in the '40s and created the Lambda Calculus
- A way to do math with only function calls

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Think of as...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction</td>
<td>( \lambda x . x )</td>
<td>function definition</td>
</tr>
<tr>
<td>Application</td>
<td>((\lambda x . x) y)</td>
<td>function call</td>
</tr>
<tr>
<td>(\alpha)-conversion</td>
<td>((\lambda x . x) \lambda y . x) (\rightarrow) ((\lambda z . z) \lambda y . x)</td>
<td>renaming variables</td>
</tr>
<tr>
<td>(\beta)-reduction</td>
<td>((\lambda z . z) \lambda y . x) (\rightarrow) (\lambda y . x)</td>
<td>plugging in expressions</td>
</tr>
<tr>
<td>Currying</td>
<td>(\lambda x . (\lambda y . x))</td>
<td>(f(x, y) = x)</td>
</tr>
</tbody>
</table>
The Lambda Calculus (con’t)

- Church Numerals:
  - Let us say some function represents 1.
    \[ \lambda f. \lambda x. fx \]
  - Then let us create a function, \( \text{SUCC}(x) \), that returns the successor of what we call it on.
    \[ \lambda n. \lambda f. \lambda x. f(nfx) \]
  - \( 2 = \text{SUCC}(1) = \lambda f. \lambda x. f(fx) \)
  - \( 3 = \text{SUCC}(\text{SUCC}(1)) = \lambda f. \lambda x. f(f(fx)) \)
  - ...and so on...

- Coincidentally, the Lambda Calculus is Turing-complete.

- But I like function names and normal-people numbers.
• Church Numerals:
  • Let us say some function represents 1.
    \( \lambda f.\lambda x.fx \)
  • Then let us create a function, \( \text{SUCC}(x) \), that returns the successor of what we call it on.
    \( \lambda n.\lambda f.\lambda x.f(nfx) \)
  • \( 2 = \text{SUCC}(1) = \lambda f.\lambda x.f(fx) \)
  • \( 3 = \text{SUCC}(\text{SUCC}(1)) = \lambda f.\lambda x.f(f(fx)) \)
  • ...and so on...

• Coincidentally, the Lambda Calculus is Turing-complete.

• But I like function names and normal-people numbers.
The Lambda Calculus (con’t)

• Church Numerals:
  • Let us say some function represents 1.
    \( \lambda f.\lambda x.fx \)
  • Then let us create a function, \( \text{SUCC}(x) \), that returns the successor of what we call it on.
    \( \lambda n.\lambda f.\lambda x.f(nfx) \)
    • \( 2 = \text{SUCC}(1) = \lambda f.\lambda x.f(fx) \)
    • \( 3 = \text{SUCC}(\text{SUCC}(1)) = \lambda f.\lambda x.f(f(fx)) \)
    • ...and so on...

• Coincidentally, the Lambda Calculus is Turing-complete.

• But I like function names and normal-people numbers.
The Lambda Calculus (con’t)

- Church Numerals:
  - Let us say some function represents 1.
    \[ \lambda f. \lambda x. fx \]
  - Then let us create a function, SUCC(x), that returns the successor of what we call it on.
    \[ \lambda n. \lambda f. \lambda x. f(nfx) \]
  - \( 2 = \text{SUCC}(1) = \lambda f. \lambda x. f(fx) \)
  - \( 3 = \text{SUCC}(\text{SUCC}(1)) = \lambda f. \lambda x. f(f(fx)) \)
  - ...and so on...

- Coincidentally, the Lambda Calculus is Turing-complete.

- But I like function names and normal-people numbers.
The Lambda Calculus (con’t)

- **Church Numerals:**
  - Let us say some function represents 1. \( \lambda f. \lambda x. fx \)
  - Then let us create a function, \( \text{SUCC}(x) \), that returns the successor of what we call it on. \( \lambda n. \lambda f. \lambda x. f(nfx) \)
  - \( 2 = \text{SUCC}(1) = \lambda f. \lambda x. f(fx) \)
  - \( 3 = \text{SUCC}(	ext{SUCC}(1)) = \lambda f. \lambda x. f(f(fx)) \)
  - ...and so on...

- Coincidentally, the Lambda Calculus is Turing-complete.

- But I like function names and normal-people numbers.
The Lambda Calculus (con’t)

- Church Numerals:
  - Let us say some function represents 1. \( \lambda f.\lambda x.fx \)
  - Then let us create a function, \( \text{SUCC}(x) \), that returns the successor of what we call it on.
    \( \lambda n.\lambda f.\lambda x.f(nfx) \)
  - \( 2 = \text{SUCC}(1) = \lambda f.\lambda x.(fx) \)
  - \( 3 = \text{SUCC}(\text{SUCC}(1)) = \lambda f.\lambda x.(f(fx)) \)
  - ...and so on...

- Coincidentally, the Lambda Calculus is Turing-complete.

- But I like function names and normal-people numbers.
The Lambda Calculus (con’t)

- Church Numerals:
  - Let us say some function represents 1.
    \[ \lambda f.\lambda x.fx \]
  - Then let us create a function, SUCC(x), that returns the successor of what we call it on.
    \[ \lambda n.\lambda f.\lambda x.f(nfx) \]
  - \( 2 = SUCC(1) = \lambda f.\lambda x.f(fx) \)
  - \( 3 = SUCC(SUCC(1)) = \lambda f.\lambda x.f(f(fx)) \)
  - ...and so on...

- Coincidently, the Lambda Calculus is Turing-complete.

- But I like function names and normal-people numbers.
The Lambda Calculus (con’t)

• Church Numerals:
  • Let us say some function represents 1. 
    \( \lambda f.\lambda x.fx \)
  • Then let us create a function, SUCC\( (x) \), that returns the successor of what we call it on. 
    \( \lambda n.\lambda f.\lambda x.f(nfx) \)
  • \( 2 = \text{SUCC}(1) = \lambda f.\lambda x.f(fx) \)
  • \( 3 = \text{SUCC}(\text{SUCC}(1)) = \lambda f.\lambda x.f(f(fx)) \)
  • ...and so on...

• Coincidentally, the Lambda Calculus is Turing-complete.

• But I like function names and normal-people numbers.
• LISP is a family of programming languages based on the Lambda Calculus.

• Languages descended from FORTRAN use infix notation: 
  \[ 5 + 2 * 3 \]

• LISP-dialects use S-Expressions: 
  \[(+ 5 (* 2 3))\]

John McCarthy, creator of LISP
LISP is a family of programming languages based on the Lambda Calculus.

Languages descended from FORTRAN use infix notation:

\[ 5 + 2 \times 3 \]

LISP-dialects use S-Expressions:

\[(+ 5 (* 2 3))\]

John McCarthy, creator of LISP
LISP

- LISP is a family of programming languages based on the Lambda Calculus.
- Languages descended from FORTRAN use infix notation: 5 + 2 * 3
- LISP-dialects use S-Expressions: (+ 5 (* 2 3))

John McCarthy, creator of LISP
Other Properties of LISP

- No difference between code and data.
- **Quoting**: “Don’t evaluate this S-Expression yet.”

```lisp
> (cons (- 2 1) 5)
'(1 . 5)
```

```lisp
> (cons '(- 2 1) 5)
'((- 2 1) . 5)
```

- Many LISP-dialects are interpreted currently, but LISP-machines used Just-In-Time Compiling.
- Symbolic Computation
Other Properties of LISP

- No difference between code and data.
- **Quoting**: “Don’t evaluate this S-Expression yet.”
  
  ```lisp
  > (cons (- 2 1) 5)
  '(1 . 5)
  ```

  ```lisp
  > (cons '(- 2 1) 5)
  '((- 2 1) . 5)
  ```

- Many LISP-dialects are interpreted currently, but LISP-machines used Just-In-Time Compiling

- Symbolic Computation
Other Properties of LISP

- No difference between code and data.
- **Quoting**: “Don’t evaluate this S-Expression yet.”

  \[
  \text{> (cons } (- 2 1) 5) \\
  \text{'(1 . 5)}
  \]

  \[
  \text{> (cons '(- 2 1) 5) } \\
  \text{'((- 2 1) . 5)}
  \]

- Many LISP-dialects are interpreted currently, but LISP-machines used
  Just-In-Time Compiling

- Symbolic Computation
Other Properties of LISP

- No difference between code and data.
- **Quoting**: “Don’t evaluate this S-Expression yet.”
  
  ```lisp
  > (cons (- 2 1) 5)
  '(1 . 5)
  
  > (cons '(- 2 1) 5)
  '((- 2 1) . 5)
  
  - Many LISP-dialects are interpreted currently, but LISP-machines used Just-In-Time Compiling
  
  - **Symbolic Computation**
## LISP-dialects

<table>
<thead>
<tr>
<th>Year</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1955</td>
<td>LISP 1, 1.5, LISP 2 (abandoned)</td>
</tr>
<tr>
<td>1960</td>
<td>Maclisp, Interlisp</td>
</tr>
<tr>
<td>1965</td>
<td>Lisp Machine Lisp</td>
</tr>
<tr>
<td>1970</td>
<td>Scheme, R5RS, R6RS, R7RS small</td>
</tr>
<tr>
<td>1975</td>
<td>NIL</td>
</tr>
<tr>
<td>1980</td>
<td>Common Lisp</td>
</tr>
<tr>
<td>1985</td>
<td>Le Lisp</td>
</tr>
<tr>
<td>1990</td>
<td>T</td>
</tr>
<tr>
<td>1995</td>
<td>Emacs Lisp, AutoLISP, OpenLisp, PicoLisp</td>
</tr>
<tr>
<td>2000</td>
<td>EuLisp, ISLISP, newLISP, Racket</td>
</tr>
<tr>
<td>2005</td>
<td>GNU Guile, Visual LISP, Qi, Qill, Shen, Clojure, Arc, LFE, Hy</td>
</tr>
</tbody>
</table>
LISP-machines
Motivation

Implicit typing, function calls, and pointers are slow...
If you’re using an architecture meant for procedural languages.
Implicit typing, function calls, and pointers are slow...
If you’re using an architecture meant for procedural languages.
The problem with implicit typing...

- C:
  ```c
  int x = 2;
  int f(int y){
      return x * y + 1;
  }
  ```

- Racket, a LISP-dialect:
  ```scheme
  (let ((x 2))
      (define (f y) (+ (* x y) 1)))
  ```
The problem with implicit typing...

- C:
  ```c
  int x = 2;
  int f(int y){
      return x * y + 1;
  }
  ```

- Racket, a LISP-dialect:
  ```scheme
  (let ((x 2))
      (define (f y) (+ (* x y) 1)))
  ```
Tagged Data

- 32-bit CPU → registers hold 32 bits
  - integer value

- What if we make the upper 2 bits represent the type?
  - type tag integer value

- Problem: our CPU operates on 32 bits at a time.
  - \( x += y \) with no tag bits:
    - \( x += y \) with 2 tag bits:
      - If tag bits of \( x \) == tag bits of \( y \):
        - \( \text{or } s3, s0, s4 \)
        - \( \text{srl } s4, 2, s4 \)
        - \( \text{sll } s4, 2, s4 \)
        - \( \text{add } s0, s4, s0 \)

- We could use another register, but...
Tagged Data

- 32-bit CPU → registers hold 32 bits
  
<table>
<thead>
<tr>
<th>integer value</th>
</tr>
</thead>
</table>

- What if we make the upper 2 bits represent the type?
  
<table>
<thead>
<tr>
<th>type tag</th>
<th>integer value</th>
</tr>
</thead>
</table>

- Problem: our CPU operates on 32 bits at a time.
  
  - \( x \pm y \) with no tag bits:
    - \( \text{x += y} \) with 2 tag bits:
      
      # If tag bits of x == tag bits of y:
      
      \[ \text{or } s3, 0, s4 \]
      
      \[ \text{srl } s4, 2, s4 \]
      
      \[ \text{sll } s4, 2, s4 \]
      
      \[ \text{add } s0, s4, s0 \]
Tagged Data

- 32-bit CPU → registers hold 32 bits

- What if we make the upper 2 bits represent the type?

- Problem: our CPU operates on 32 bits at a time.

  - $x += y$ with no tag bits:
    ```assembly
    add $s0, $s3, $s0
    ```

  - $x += y$ with 2 tag bits:
    ```assembly
    # If tag bits of $x$ == tag bits of $y$:
    or $s3, $0, $s4
    srl $s4, 2, $s4
    sll $s4, 2, $s4
    add $s0, $s4, $s0
    ```

  - We could use another register, but...
Tagged Data

- 32-bit CPU → registers hold 32 bits
  - integer value

- What if we make the upper 2 bits represent the type?
  - type tag | integer value

- Problem: our CPU operates on 32 bits at a time.
  - \( x += y \) with no tag bits:
    
    ```
    add $s0 $s3 $s0
    ```
  - \( x += y \) with 2 tag bits:
    
    ```
    # If tag bits of \( x \) == tag bits of \( y \):
    or $s3, $0, $s4
    srl $s4, 2, $s4
    sll $s4, 2, $s4
    add $s0, $s4, $s0
    ```
  - We could use another register, but...
Tagged Data

- 32-bit CPU $\rightarrow$ registers hold 32 bits
  - integer value

- What if we make the upper 2 bits represent the type?
  - type tag integer value

- Problem: our CPU operates on 32 bits at a time.
  - $x += y$ with no tag bits:
    - add $s0 \ s3 \ s0$
  - $x += y$ with 2 tag bits:
    - # If tag bits of $x ==$ tag bits of $y$
      - or $s0, s0, s4$
      - srl $s4, 2, s4$
      - sll $s4, 2, s4$
      - add $s0, s4, s0$
    - We could use another register, but...
- 32-bit CPU $\rightarrow$ registers hold 32 bits
  - integer value

- What if we make the upper 2 bits represent the type?
  - type tag
  - integer value

- Problem: our CPU operates on 32 bits at a time.
  - $x += y$ with no tag bits:
    
    ```
    add $s0, $s3, $s0
    ```

  - $x += y$ with 2 tag bits:
    
    ```
    or $s3, $0, $s4
    srl $s4, 2, $s4
    sll $s4, 2, $s4
    add $s0, $s4, $s0
    ```

  - We could use another register, but...
Tagged Data

- 32-bit CPU → registers hold 32 bits

  integer value

- What if we make the upper 2 bits represent the type?

  type tag  integer value

- Problem: our CPU operates on 32 bits at a time.
  - $x += y$ with no tag bits:
    
    ```
    add $s0 $s3 $s0
    ```
  
  - $x += y$ with 2 tag bits:
    
    ```
    # If tag bits of $x ==$ tag bits of $y$:
    or $s3$, $0$, $s4$
    srl $s4$, 2, $s4$
    sll $s4$, 2, $s4$
    add $s0$, $s4$, $s0$
    ```

  - We could use another register, but...
Tagged Data

- **32-bit CPU → registers hold 32 bits**
  
  **integer value**

- **What if we make the upper 2 bits represent the type?**
  
  **type tag**  **integer value**

- **Problem: our CPU operates on 32 bits at a time.**
  
  - $x += y$ with no tag bits:
    
    ```
    add $s0 $s3 $s0
    ```
  
  - $x += y$ with 2 tag bits:
    
    ```
    # If tag bits of $x ==$ tag bits of $y$:
    or $s3, $0, $s4
    srl $s4, 2, $s4
    sll $s4, 2, $s4
    add $s0, $s4, $s0
    ```

  - We could use another register, but...
Tagged Data

- 32-bit CPU → registers hold 32 bits

- What if we make the upper 2 bits represent the type?

- Problem: our CPU operates on 32 bits at a time.
  - $x += y$ with no tag bits:
    ```assembly
    add $s0 $s3 $s0
    ```
  - $x += y$ with 2 tag bits:
    ```assembly
    # If tag bits of $x == tag bits of $y:
    or $s3, $0, $s4
    srl $s4, 2, $s4
    sll $s4, 2, $s4
    add $s0, $s4, $s0
    ```
  - We could use another register, but...
Tagged Data

- 32-bit CPU → registers hold 32 bits
  
  integer value

- What if we make the upper 2 bits represent the type?
  type tag integer value

- Problem: our CPU operates on 32 bits at a time.
  
  - $x += y$ with no tag bits:
    add $s0$ $s3$ $s0$
  
  - $x += y$ with 2 tag bits:
    
    # If tag bits of $x ==$ tag bits of $y$:
    or $s3$, $0$, $s4$
    srl $s4$, 2, $s4$
    sll $s4$, 2, $s4$
    add $s0$, $s4$, $s0$
    
- We could use another register, but...
Let’s build our own CPU for LISP that doesn’t have these problems.

- Registers hold cons cells instead of integers
- Longer word size (40 bits instead of 32)
- Parallelize type checking
Let’s build our own CPU for LISP that doesn’t have these problems.

- Registers hold cons cells instead of integers
  - Longer word size
    (40 bits instead of 32)
  - Parallelize type checking
Let’s build our own CPU for LISP that doesn’t have these problems.

- Registers hold cons cells instead of integers
- Longer word size (40 bits instead of 32)
- Parallelize type checking
Let’s build our own CPU for LISP that doesn’t have these problems.

- Registers hold cons cells instead of integers
- Longer word size (40 bits instead of 32)
- Parallelize type checking
Function Calls

- In procedural hardware, program is traversed by incrementing / decrementing the Program Counter and Stack Pointer.
- Push data onto the stack when you make a function call, and pop it off when done.
- Need to save contents of registers when you made the call so you can get back to where you were when you return.
- LISP is a language made of function calls and pointers.
Function Calls

- In procedural hardware, program is traversed by incrementing / decrementing the Program Counter and Stack Pointer.
- Push data onto the stack when you make a function call, and pop it off when done.
- Need to save contents of registers when you made the call so you can get back to where you were when you return.
- LISP is a language made of function calls and pointers.
Function Calls

- In procedural hardware, program is traversed by incrementing / decrementing the Program Counter and Stack Pointer.
- Push data onto the stack when you make a function call, and pop it off when done.
- Need to save contents of registers when you made the call so you can get back to where you were when you return.
- LISP is a language made of function calls and pointers.
Function Calls

- In procedural hardware, program is traversed by incrementing / decrementing the Program Counter and Stack Pointer.
- Push data onto the stack when you make a function call, and pop it off when done.
- Need to save contents of registers when you made the call so you can get back to where you were when you return.
- LISP is a language made of function calls and pointers.
The SECD Abstract Machine

- **C-Register (Control)**
  - Instead of Program Counter, point to cell whose CAR points to the next instruction (instruction cell).
  - Instructions are one integer with no register fields, etc.
    \[ \text{add } \$_s0, \$_s4, \$_s0 \]
  - Instead, CDR of instruction cell points to such information

- **E-Register (Environment)**
  - Points to list of current function arguments
  - Added to when function environment is modified

- **D-Register (Dump)**
  - Points to list whose last 3 cells save S, E, and C registers from caller function

- **S-Register (Stack)**
  - Points to list used as stack for built-in functions (ex: + − × /)
  - Top of stack is cell whose CAR points to value to be processed
The SECD Abstract Machine

- **C-Register (Control)**
  - Instead of Program Counter, point to cell whose CAR points to the next instruction (instruction cell).
  - Instructions are one integer with no register fields, etc.
    
    ```
    add $s0, $s4, $s0
    ```
  - Instead, CDR of instruction cell points to such information

- **E-Register (Environment)**
  - Points to list of current function arguments
  - Added to when function environment is modified

- **D-Register (Dump)**
  - Points to list whose last 3 cells save S, E, and C registers from caller function

- **S-Register (Stack)**
  - Points to list used as stack for built-in functions (ex. +, −, /, ×)
  - Top of stack is cell whose CAR points to value to be processed
The SECD Abstract Machine

- **C-Register** (Control)
  - Instead of Program Counter, point to cell whose CAR points to the next instruction (instruction cell).
  - Instructions are one integer with no register fields, etc.
    \[ \text{add } \$s0, \$s4, \$s0 \]
  - Instead, CDR of instruction cell points to such information

- **E-Register** (Environment)
  - Points to list of current function arguments
  - Added to when function environment is modified

- **D-Register** (Dump)
  - Points to list whose last 3 cells save S, E, and C registers from caller function

- **S-Register** (Stack)
  - Points to list used as stack for built-in functions (ex. +, -, /, x)
  - Top of stack is cell whose CAR points to value to be processed
The SECD Abstract Machine

- **C-Register** (Control)
  - Instead of Program Counter, point to cell whose CAR points to the next instruction (instruction cell).
  - Instructions are one integer with no register fields, etc.
    \[
    \text{add } \$s0, \$s4, \$s0
    \]
  - Instead, CDR of instruction cell points to such information

- **E-Register** (Environment)
  - Points to list of current function arguments
  - Added to when function environment is modified

- **D-Register** (Dump)
  - Points to list whose last 3 cells save S, E, and C registers from caller function

- **S-Register** (Stack)
  - Points to list used as stack for built-in functions (ex.: +, -, /, *)
  - Top of stack is cell whose CAR points to value to be processed
The SECD Abstract Machine

- **C-Register** (Control)
  - Instead of Program Counter, point to cell whose CAR points to the next instruction (instruction cell).
  - Instructions are one integer with no register fields, etc.
    - `add $s0, $s4, $s0`
  - Instead, CDR of instruction cell points to such information

- **E-Register** (Environment)
  - Points to list of current function arguments
  - Added to when function environment is modified

- **D-Register** (Dump)
  - Points to list whose last 3 cells save S, E, and C registers from caller function

- **S-Register** (Stack)
  - Points to list used as stack for built-in functions (ex. +, -, *, /, 
  - Top of stack is cell whose CAR points to value to be processed
The SECD Abstract Machine

- **C-Register (Control)**
  - Instead of Program Counter, point to cell whose CAR points to the next instruction (instruction cell).
  - Instructions are one integer with no register fields, etc.
    
    ```
    add $s0, $s4, $s0
    ```
  - Instead, CDR of instruction cell points to such information

- **E-Register (Environment)**
  - Points to list of current function arguments
  - Added to when function environment is modified

- **D-Register (Dump)**
  - Points to list whose last 3 cells save S, E, and C registers from caller function

- **S-Register (Stack)**
  - Points to list used as stack for built-in functions (ex. +, -, /, *)
  - Top of stack is cell whose CAR points to value to be processed
The SECD Abstract Machine

- **C-Register (Control)**
  - Instead of Program Counter, point to cell whose CAR points to the next instruction (instruction cell).
  - Instructions are one integer with no register fields, etc.
    \[
    \text{add } \$s0, \$s4, \$s0
    \]
  - Instead, CDR of instruction cell points to such information

- **E-Register (Environment)**
  - Points to list of current function arguments
  - Added to when function environment is modified

- **D-Register (Dump)**
  - Points to list whose last 3 cells save S, E, and C registers from caller function

- **S-Register (Stack)**
  - Points to list used as stack for built-in functions (ex: \(+/\times/\div\))
  - Top of stack is cell whose CAR points to value to be processed
The SECD Abstract Machine

- **C-Register** (Control)
  - Instead of Program Counter, point to cell whose CAR points to the next instruction (instruction cell).
  - Instructions are one integer with no register fields, etc.
    \[
    \text{add } \$s0, \$s4, \$s0
    \]
  - Instead, CDR of instruction cell points to such information

- **E-Register** (Environment)
  - Points to list of current function arguments
  - Added to when function environment is modified

- **D-Register** (Dump)
  - Points to list whose last 3 cells save S, E, and C registers from caller function

- **S-Register** (Stack)
  - Points to list used as stack for built-in functions (ex. +, -, /, *)
  - Top of stack is cell whose CAR points to value to be processed
The SECD Abstract Machine

- **C-Register** (Control)
  - Instead of Program Counter, point to cell whose CAR points to the next instruction (instruction cell).
  - Instructions are one integer with no register fields, etc.
    
    ```
    add $s0, $s4, $s0
    ```
    
  - Instead, CDR of instruction cell points to such information

- **E-Register** (Environment)
  - Points to list of current function arguments
  - Added to when function environment is modified

- **D-Register** (Dump)
  - Points to list whose last 3 cells save S, E, and C registers from caller function

- **S-Register** (Stack)
  - Points to list used as stack for built-in functions (ex. + − / ×)
  - Top of stack is cell whose CAR points to value to be processed
The SECD Abstract Machine

- **C-Register** (Control)
  - Instead of Program Counter, point to cell whose CAR points to the next instruction (instruction cell).
  - Instructions are one integer with no register fields, etc.
    \[ \text{add } \$s0, \$s4, \$s0 \]
  - Instead, CDR of instruction cell points to such information

- **E-Register** (Environment)
  - Points to list of current function arguments
  - Added to when function environment is modified

- **D-Register** (Dump)
  - Points to list whose last 3 cells save S, E, and C registers from caller function

- **S-Register** (Stack)
  - Points to list used as stack for built-in functions (ex. + − /×)
  - Top of stack is cell whose CAR points to value to be processed
The SECD Abstract Machine

- **C-Register** (Control)
  - Instead of Program Counter, point to cell whose CAR points to the next instruction (instruction cell).
  - Instructions are one integer with no register fields, etc.
    \[ \text{add } \$s0, \$s4, \$s0 \]
  - Instead, CDR of instruction cell points to such information

- **E-Register** (Environment)
  - Points to list of current function arguments
  - Added to when function environment is modified

- **D-Register** (Dump)
  - Points to list whose last 3 cells save S, E, and C registers from caller function

- **S-Register** (Stack)
  - Points to list used as stack for built-in functions (ex. + − / ×)
    - Top of stack is cell whose CAR points to value to be processed
The SECD Abstract Machine

- **C-Register** (Control)
  - Instead of Program Counter, point to cell whose CAR points to the next instruction (instruction cell).
  - Instructions are one integer with no register fields, etc.
    \[\text{add } \$s0, \$s4, \$s0\]
  - Instead, CDR of instruction cell points to such information

- **E-Register** (Environment)
  - Points to list of current function arguments
  - Added to when function environment is modified

- **D-Register** (Dump)
  - Points to list whose last 3 cells save S, E, and C registers from caller function

- **S-Register** (Stack)
  - Points to list used as stack for built-in functions (ex. \(+\) \(-\) \(/\) \(\times\))
  - Top of stack is cell whose CAR points to value to be processed
Taking Out the Garbage

- In LISP you don’t allocate/deallocate memory yourself
  - Incremental garbage collection
  - “Depth-first” copying improves spatial locality
  - Hardware-accelerated garbage collection
Taking Out the Garbage

- In LISP you don’t allocate/deallocate memory yourself
- Incremental garbage collection
  - “Depth-first” copying improves spatial locality
- Hardware-accelerated garbage collection
Taking Out the Garbage

- In LISP you don’t allocate/deallocate memory yourself
- Incremental garbage collection
- “Depth-first” copying improves spatial locality
- Hardware-accelerated garbage collection
Taking Out the Garbage

• In LISP you don’t allocate/deallocate memory yourself
• Incremental garbage collection
• “Depth-first” copying improves spatial locality
• Hardware-accelerated garbage collection
But what is using a LISP-machine like?

- Single-user
- Graphics
- Freedom
Complete Total Freedom

- Everything is in LISP
- Homoiconicity allows true understanding
- Can read/edit anything and everything
- Can pause code execution, debug/edit, then continue
- Richard Stallman worked with LISP-machines at MIT in the ’70s
Complete Total Freedom

- Everything is in LISP
- Homoiconicity allows true understanding
- Can read/edit anything and everything
- Can pause code execution, debug/edit, then continue
- Richard Stallman worked with LISP-machines at MIT in the '70s
Complete Total Freedom

- Everything is in LISP
- Homoiconicity allows true understanding
- Can read/edit anything and everything
- Can pause code execution, debug/edit, then continue
- Richard Stallman worked with LISP-machines at MIT in the ’70s
Complete Total Freedom

- Everything is in LISP
- Homoiconicity allows true understanding
- Can read/edit anything and everything
- Can pause code execution, debug/edit, then continue
- Richard Stallman worked with LISP-machines at MIT in the ’70s
Complete Total Freedom

• Everything is in LISP
• Homoiconicity allows true understanding
• Can read/edit anything and everything
• Can pause code execution, debug/edit, then continue
• Richard Stallman worked with LISP-machines at MIT in the ’70s
What happened?

- LISP-machines were expensive top-of-the-line equipment
- Single-person computers were unusual at the time
- DARPA shifted focus – “AI is just clever programming”
- Cheaper and more powerful ’80s computers
What happened?

- LISP-machines were expensive top-of-the-line equipment
- Single-person computers were unusual at the time
  - DARPA shifted focus – “AI is just clever programming”
- Cheaper and more powerful ’80s computers
What happened?

- LISP-machines were expensive top-of-the-line equipment
- Single-person computers were unusual at the time
- DARPA shifted focus – “AI is just clever programming”
- Cheaper and more powerful ’80s computers
What happened?

- LISP-machines were expensive top-of-the-line equipment
- Single-person computers were unusual at the time
- DARPA shifted focus – “AI is just clever programming”
- Cheaper and more powerful ’80s computers
This presentation was from the Mines Linux Users Group. A mostly-complete archive of our presentations can be found online at https://lug.mines.edu.

Individual authors may have certain copyright or licensing restrictions on their presentations. Please be certain to contact the original author to obtain permission to reuse or distribute these slides.