The SECD Machine

- Already discussed *intepreter* for
  functional programming language

  Now: write a *compiler*

⇒ translate functional program to form
that can be directly executed on
virtual machine

- Compiled code can be
  - ∗ executed many times
  - ∗ may be optimized

SECD Virtual Machine Architecture

- SECD machine uses four stacks:

  s: *Stack* evaluates expressions

  e: *Environment* stores current value list

  c: *Control* stores instructions of "active" function

  d: *Dump* stores suspended **s**, **e** and **c** stacks while a new "activation" executes.

- Supports machine operations:

  `ADD, AP, ATOM, CAR, CDR, CONS, DIV, DUM, EQ, JOIN, LD, LDC, LDF, LEQ, MUL, NIL, RAP, REM, RTN, SEL, STOP, SUB`

Operations

- Each operation defined by affect on four stacks:

  `s e c d` ↦ `s' e' c' d'`

- Represent each stack by *s-expression*
  with dot notation
  (leftmost position denotes top of stack)

Eg: `(a b . s)` ≡ `(a . (b . s))`
= stack whose top element is `a`, then `b`
rest of stack is denoted `s`

Arithmetic Operators

- Each arithmetic operator
  - pops top two elements from evaluation stack **s**
  - operates on these elements, then
  - pushes result onto **s** stack

```
(a b . s) e (ADD . c) d  ↦  (a+b . s) e c d
(a b . s) e (SUB . c) d  ↦  (a-b . s) e c d
(a b . s) e (MUL . c) d  ↦  (a*b . s) e c d
(a b . s) e (DIV . c) d  ↦  (a/b . s) e c d
(a b . s) e (REM . c) d  ↦  (a rem b . s) e c d
```

Eg: `(1 3 5 7.s) e (ADD SUB MUL.c) d` ↦
`(4 5 7.s) e (SUB MUL.c) d` ↦
`(-1 7.s) e (MUL.c) d` ↦
`(-7.s) e c d`

Relational Operators

- Each relational operator
  - pops top two elements from **s** evaluation stack
  - compares these values
  - pushes result onto **s** stack

    FALSE represented by 0

    TRUE by 1 (or any non-0 value)

```
(a b.s) e (EQ.c) d    ↦  (a=b.s) e c d
(a b.s) e (LEQ.c) d   ↦  (a≤b.s) e c d
```

Eg:
```
(1 3 5 7.s) e (ADD LEQ.c) d  ↦
(4 5 7.s) e (LEQ.c) d  ↦
(1 7.s) e c d
```


Structural Operators

- Each structural operator
  - pops appropriate elements from **s**
  - computes result
  - pushes result onto **s** stack

```
(a.s) e (ATOM.c) d  ↦ (b.s) e c d
```
where b = 1 if a is atom
b = 0 otherwise

```
(a b.s) e (CONS.c) d  ↦  ((a.b).s) e c d
((a.b).s) e (CAR.c) d  ↦  (a.s) e c d
((a.b).s) e (CDR.c) d  ↦  (b.s) e c d
```

Eg:
```
(1 (3).s) e (CONS CDR CAR ATOM.c) d  ↦
((1 3).s) e (CDR CAR ATOM.c) d  ↦
((3).s) e (CAR ATOM.c) d  ↦
(3.s) e (ATOM.c) d  ↦
(1.s) e c d
```


Load Operators

- **LDC** (load constant) operator
  - pops element from **c** control stack
  - pushes it onto **s** evaluation stack

```
s e (LDC x.c) d  ↦  (x.s) e c d
```

- **NIL** (load NIL) operator
  - pushes NIL onto **s** evaluation stack

```
s e (NIL.c) d  ↦ (NIL.s) e c d
```

Eg: s e (NIL LDC 3.c) d ↦

(NIL.s) e (LDC 3.c) d ↦

(3 NIL.s) e c d

- **LD** (load) operator
  - pops an element from **e** environment stack
  - pushes it onto **s** evaluation stack

. . . but what is environment. . .

Environment

- s e (LD (i.j).c) d ↦ (x.s) e c d
  where x = locate((i.j), e)

- Encode environment as a
  list of lists of values
  . . . accessed using two-index dotted-pair
  first index determines which list;
  second index determines which
  element in that list

```
index(n, s) = if eq(n, 1) then car(s)
     else index(n-1, cdr(s))

locate(p, e) = index(cdr(p), index(car(p), e))
```

Eg:
```
locate((2.1), ((1 3) ((5 6) 4))) =
index(cdr((2.1)), index(car((2.1)), ((1 3) ((5 6) 4)))) =
index(1, index(2, ((1 3) ((5 6) 4))) =
index(1, if eq(2, 1) then car((1 3) ((5 6) 4))
       else index(1, cdr((1 3) ((5 6) 4)))) =
index(1, index(1, cdr((1 3) ((5 6) 4))) =
```

```
index(1, index(1, (((5 6) 4))) =
index(1, ((5 6) 4)) =
(5 6)
```

Eg: `s ((1 3) ((5 6) 4)) (LD (2.1) CAR LD (1.1) ADD.c) d` $\mapsto$

`((5 6).s) ((1 3) ((5 6) 4)) (CAR LD (1.1) ADD.c) d` $\mapsto$

`(5.s) ((1 3) ((5 6) 4)) (LD (1.1) ADD.c) d` $\mapsto$

`(1 5.s) ((1 3) ((5 6) 4)) (ADD.c) d` $\mapsto$

`(6.s) ((1 3) ((5 6) 4)) c d`

Basic "Function" ality

- **LDF** ("load function")
  - pops function from `c` control stack
  - pushes closure onto `s` evaluation stack,

  context of closure copied from environment stack

  s e (LDF f.c) d $\mapsto$ ((f.e).s) e c d

Eg: `(5).s ((2)) (LDF (LD (1.1) LD (2.1) ADD RTN).c) d` $\mapsto$

`((LD (1.1) LD (2.1) ADD RTN).((2))) (5).s) ((2)) c d`

Control Operators

- **STOP** leaves machine in idle state.

  `s e (STOP.c) d` $\mapsto$ `s e (STOP.c) d`

- **SEL** ("select subcontrol")
  - pop top element `v` from `s` (evaluation)

  If `v`$\equiv$ `F` then
  deletes next operation from `c` (control)
  If `v`$\equiv$ `T` then
  deletes operation *after* next from `c`

  In each cases:
  deletes rest of control stack,
  after placing it on dump.

  `(x.s) e (SEL cT cF.c) d` $\mapsto$ `s e cX (c.d)`

  where
  `cX = cT if x` $\neq$ `0 (T)`
  `cX = cF if x` $=$ `0 (F)`

Other Control Operations

- **JOIN** ("rejoin main control")
  restores control from dump stack.

  s e (JOIN.c') (c.d) $\mapsto$ s e c d

Eg: `(3.s) ((7 9)) (ATOM SEL (LD (1.1) JOIN)`
`(LD (1.2) JOIN STOP) d` $\mapsto$

`(1.s) ((7 9)) (SEL (LD (1.1) JOIN)`
`(LD (1.2) JOIN STOP) d` $\mapsto$

`s ((7 9)) (LD (1.1) JOIN) (STOP).d` $\mapsto$

`(7.s) ((7 9)) (JOIN) (STOP).d` $\mapsto$

`(7.s) ((7 9)) (STOP) d`

- **RTN** (return) operation
  - restores top "activation record" from dump stack
  - transfers top stack element from "dead" evaluation stack to restored evaluation stack

```
(x.z) e' (RTN.q) (s e c.d)  ↦ (x.s) e c d
```

Eg: `(7 9) ((5) (2)) (RTN LDC 1) ((3) ((1)) (STOP).d)  ↦`

`(7 3) (1) (STOP) d`

Function Application

- **AP** (apply function)
  - saves current machine state onto dump stack
  - installs code from closure on top of evaluation stack into the control stack,
  - inserts argument list from below closure into environment of closure
  - installs result into environment stack

```
((f.e') v.s) e (AP.c) d
    ↦ NIL (v.e') f (s e c.d)
```

Eg: `((LD (1.1) LD (2.1) ADD RTN).((2)) (5).s) ((7)) (AP.c) d  ↦`

`NIL ((5) (2)) (LD (1.1) LD (2.1) ADD RTN) (s ((7)) c.d)`

Recursive Function Application

- **DUM** (dummy)
  creates "dummy" environment
  with NIL as first sublist

. . . used to implement recursive `letrec` blocks

```
s e (DUM.c) d  ↦ s (NIL.e) c d
```

- **RAP** (recursive apply) operation
  - identical to `AP` operation EXCEPT
  - uses `rplaca` to replace the dummy cell set by a previous DUM instruction, instead of inserting the argument list directly into the closure's environment.

```
((f.NIL.e') v.s) (NIL.e) (RAP.c) d  ↦

NIL rplaca((NIL.e'), v) f (s e c.d)
```


The Compiler

- Compiler is straightforward,
  just compiles well-formed expressions:

- **Constants/Variables**

  `NIL` ⇒ `(NIL)`
  `number` ⇒ `(LDC number)`
  `id` ⇒ `(LD (i.j))` where `(i.j)` is index for `id` in `E`

- **Built-in functions**

  `(f e1 ... en)` ⇒ `(en' e1' f')`
  [reverse polish]

Eg: `(+ 6 2)` ⇒ `(LDC 2 LDC 6 ADD)`
`(* (+ 6 2) 3)` ⇒ `(LDC 3 LDC 2 LDC 6 ADD MUL)`

Special functions/forms

- (if e1 e2 e3) ⇒
  (e1' SEL (e2' JOIN) (e3' JOIN))

Eg: (if (atom x) x 0) ⇒

```
(LD (i.j) ATOM SEL (LD (i.j) JOIN) (LDC 0 JOIN))
```

- (lambda (x1 ... xn) e) ⇒ (LDF (e' RTN))

Eg: (lambda (x y) (+ x y)) ⇒

```
   (LDF (LD (1.2) LD (1.1) ADD RTN))
```

(e e1...en) ⇒

```
   (NIL en' CONS ... e1' CONS e' AP)
```

Eg: ((lambda (x y) (+ x y)) 1 2) ⇒

```
(NIL LDC 1 CONS LDC 2 CONS LDF (LD (1.2) LD (1.1)
ADD RTN) AP)
```

(let (x1.e1)...(xn.en) e) ⇒

```
   (NIL en' CONS ... e1' CONS LDF (e' RTN) AP)
```

Eg: (let (x.1) (y.2) (+ x y)) ⇒

```
   (NIL LDC 2 CONS LDC1 CONS LDF (LD (1.2) LD (1.1)
ADD RTN) AP)
```