Command Overview

- Assignment
- Control Structures
- Natural Semantics for commands
- Iterators
- Exceptions
Commands or Statements

- Change *state* of machine
- State of computer corresponds to contents of memory and any external devices (I/O)
- State sometimes called *store*
Store vs Environment

- Note distinction between “state” and “environment”
  - Environment is mapping between identifiers and values (including locations).
  - State includes mapping between locations and values.
- Values in store or memory are *storable* versus *denotable* (or *bindable*)
- Symbol table depends on declarations and scope — static
- Environment tells where to find values — dynamic
- State depends on previous computation — dynamic
If have compiler, use symbol table when generating code to determine meaning of all identifiers. At run-time, symbol table no longer needed (hard coded into compiled code), but state and environment change dynamically.

In interpreter, may have to keep track of symbol table, environment, and state at run-time. (Could avoid using state if there is no “aliasing” in the language.)
Variables

- Value Model — variable is container
  - $l$-value — expression that refers to location
    
    ```
    a = b + c;
    (f(a) + 3)\rightarrow b[c] = 2;
    ```
  - $r$-value — expression that refers to value

- Reference Model — variable is reference
  - Every variable occurrence is an $l$-value
  - Requires dereference — may be implicit
Assignment

- Store value in location named by variable
  \[ \text{variable} := \text{expressions} \]
- Order of evaluation can be important, especially if there are side-effects. Usually left-side evaluated first, then right-side.
  \[ \text{A}[f(j)] := j * f(j) + j-- \]
  difficult to predict value if \( f \) has side effect of changing \( j \)
- Two kinds of assignments:
  1. assignment by copying, and
  2. assignment by sharing (useful with dynamic typing or in OOLs)
Control Structures

- Statements for combining other expressions and statements
  - Sequencing: $S; T$
  - Selection: if-then-else
  - Repetition: while-do
Early Control Structures

- FORTRAN started with very primitive control structures:
  - GO TO n
  - GO TO (17, 43, 12, 99), I (also other variants)
  - IF(arith exp) 17, 43, 12 means go to statement number 17 if arith exp is negative, 43 if zero, and 12 if positive
  - DO label ivble = 1, 20, 2

- Very close to machine instructions

- Why bother with repetition, if can do it all with goto’s?
  “The static structure of a program should correspond in a simple way with the dynamic structure of the corresponding computation.” Dijkstra letter to editor.
Progress in Control Structures

- ALGOL 60 more elaborate:
  - GO TO 99
  - IF - THEN - ELSE - (hierarchical)
  - Baroque loop constructions
    for i := 3, 7, 11 step 1 until 16,
    i/2 while i >= 1, 2 step i until 32 do ..
    all expressions re-evaluated each time through loop: 3, 7, 11, 12, 13, 14, 15, 16, 8, 4, 2, 1, 2, 4, 8, 16, 32.
  - switch — cross between case and computed go to.
Progress in Control Structures

• Pascal expanded but simplified:
  – go to
  – if-then-else
  – for, while, repeat-until
  – labelled case — Hoare’s most important invention
    * clear and efficient, construct jump table, optimize depending on size,
    * self-documenting
    * Modula 2 improved by adding otherwise clause
Progress in Control Structures

- Ada like Pascal but more uniform loop with exit

  iteration specification loop
  
  loop body
  
  end loop.

where iteration specification can be:

- while condition

- for v in discrete range (e.g. for i in 1..10 loop .. end loop)

  (Note: loop variable is implicitly declared with restricted scope)

- Also can have vanilla loop which can be left with exit statement.
Natural Semantics for Commands

- Must keep track of store
- Treated as a function from locations to storable values
- Commands are essentially expressions with side-effects
- Need to indicate side-effect in rules
- To show evaluation of command expression $e$ in environment $\rho$ with store $s$, use the notation $(e, \rho, s) \Downarrow (v, s')$ where $v$ is a storable value, and $s'$ is the store after evaluation of $e$
- Modified store is part of result
Natural Semantics for Commands (cont)

- Example: if-then-else

\[
(b, \rho, s) \downarrow (\text{true, } s') \quad (e_1, \rho, s') \downarrow (v, s'')
\]

\[
(\text{if } b \text{ then } e_1 \text{ else } e_2, \rho, s) \downarrow (v, s''')
\]

- Rule says that

1. if \(b\) is evaluated with store \(s\), and the evaluation of \(b\) has a side-effect, it yields a new store \(s'\) in which \(e_1\) is evaluated, and

2. if \(e_1\) has a side-effect then evaluation of the command gives a new store \(s''\)
Rules for Expressions

1. \((\text{id}, \rho, s) \downarrow (s(loc), s)\) where \(loc = \rho(\text{id})\)

2. \((\text{id}++$, $\rho, s) \downarrow (v, s[loc/v + 1])\) where \(loc = \rho(\text{id}), v = s(loc)\)

   Note: \(s' = s[loc/v + 1]\) is a store identical to \(s\) except \(s'(loc) = v + 1\).

3. \[\begin{array}{c}
   (\text{e}_1, \rho, s) \downarrow (v_1, s') \\
   (\text{e}_2, \rho, s') \downarrow (v_2, s'')
   \end{array}\]

   \[\frac{}{(\text{e}_1 + \text{e}_2, \rho, s) \downarrow (v_1 + v_2, s'')}\]
Rules for Commands

- The result of evaluating a command is a state only.

1. 

\[
\frac{(e, \rho, s) \downarrow (v, s')}{(x := e, \rho, s) \downarrow s'[loc/v] \text{ where } \rho(x) = \text{loc}}
\]

2.

\[
\frac{(C_1, \rho, s) \downarrow s' \quad (C_2, \rho, s') \downarrow s''}{(C_1 ; C_2, \rho, s) \downarrow s''}
\]

3.

\[
\frac{(b, \rho, s) \downarrow (\text{true}, s') \quad (C_1, \rho, s') \downarrow s''}{(\text{if } b \text{ then } C_1 \text{ else } C_2, \rho, s) \downarrow s''}
\]

plus a similar rule for the case when \(b\) is false
Rules for Commands

1. 

\[
\frac{(b, \rho, s) \downarrow (false, s')}{(while \ b \ do \ C, \rho, s) \downarrow s'}
\]

2. 

\[
\frac{(b, \rho, s) \downarrow (true, s') \quad (C, \rho, s') \downarrow s'' \quad (while \ b \ do \ C, \rho, s'') \downarrow s'''}{(while \ b \ do \ C, \rho, s) \downarrow s''''}
\]

Notice how similar definition of semantics for while E do C is to

if E then begin
  C;
  while E do C
end
Iterators

- Clu allows definition of user-defined iterators (abstract over control structures):

  ```clu
  for c : char in string_chars(s) do ... 
  ```

  where have defined:

  ```clu
  string_chars = iter (s : string) yields (char);
  index : Int := 1;
  limit : Int := string$size (s);
  while index <= limit do
    yield (string$fetch(s, index));
    index := index + 1;
  end;
  end string_chars;
  ```
Iterators (cont)

- Behave like restricted type of co-routine
  - Each time at top of loop continue executing iterator code from where last left off
  - When hit `yield` statement then return the associated value
  - When hit end of iterator, quit loop
- Can be implemented on stack similarly to procedure call
- Now available in Sather, C++, and Java
Exceptions

- Need mechanism to handle exceptional conditions
- Example: Trying to pop element off of an empty stack
- Clearly corresponds to mistake of some sort, but stack module doesn’t know how to respond
- Without exception handling:
  - print error message and halt
  - function/procedure returns boolean success flag — programmer has to check
  - Add procedure parameter which handles exceptions
Exceptions

- Exception mechanism in programming languages allows raising an exception which is sent back to caller for handling.
- A robust program is able to recover from exceptional conditions, rather than just halting (or crashing).
- Typical exceptions:
  - Arithmetic or I/O faults (e.g., divide by 0, read int and get char, array or subrange bounds, etc.)
  - failure of precondition,
  - unpredictable conditions (read past end of file, end of printer page, etc.),
  - tracing program flow during debugging.
- Raised exception must be handled or program will fail
Ada Exception Handling

- Raise exception with `raise exception_name`
- Attach exception handlers to subprogram body, package body, or block

```ada
begin
  C
exception
    when excp_name1 => C'
    when excp_name2 => C''
    when others => C'
end
```
Locating Exception Handler

- When an exception is raised, must be handled or caught
- Typical approach to locating handler
  - Look for handler in current block (or subprogram)
  - If not there, force return from unit and raise same exception to routine which called current one
  - Continue up the dynamic links until find handler or get to outer level and fail.
- Semantics of raising and handling exceptions is dynamic rather than static
- Handler can attempt to handle exception, but give up and raise another exception
Resuming After Exceptions

- Once exception is handled what happens next?
- Ada: return from the procedure (or unit) containing the handler — called *termination* model.
- PL/I: re-execute statement where failure occurred (makes sense for read errors, for example) unless handler forces otherwise (with goto) — called *resumption* model
- Eiffel (an OOL): uses variant of resumption model.
- ML: exceptions can pass parameter to exception handlers (like values in datatype). Otherwise very similar to Ada.
ML Exceptions

- Example program to check for balanced parenthesis in a string

```ml
 datatype 'a stack = EmptyStack | Push of 'a * ('a stack);
 exception empty;
 fun pop EmptyStack = raise empty
 | pop(Push(n, rest)) = rest;
 fun top EmptyStack = raise empty
 | top (Push(n, rest)) = n;
 fun IsEmpty EmptyStack = true
 | IsEmpty (Push(n, rest)) = false;

 exception nomatch;

 fun buildstack nil initstack = initstack
 | buildstack (#"::rest) initstack = buildstack rest (Push(#"", initstack))
 | buildstack (#")::rest) (Push(#"", bottom)) = buildstack rest bottom
 | buildstack (#")::rest) initstack = raise nomatch
 | buildstack (fst::rest) initstack = buildstack rest initstack;

 fun balanced string = (IsEmpty(buildstack (explode string) EmptyStack))
  handle nomatch => false;
```
ML Exceptions (cont)

- Notice that need to put parentheses around the expression to which the handler is associated – awkward
- Might argue that this is not unexpected situation. Just a way fancy way of introducing goto’s.