Evaluation and universal machines
• What is the role of evaluation in defining a language?
• How can we use evaluation to design a language?

The Eval/Apply Cycle
• Eval and Apply execute a cycle that unwinds our abstractions
  • Reduces to simple applications of built in procedure to primitive data structures
• Key:
  • Evaluator determines meaning of programs (and hence our language)
  • Evaluator is just another program!!

Examining the role of Eval
• From perspective of a language designer
• From perspective of a theoretician

Eval from perspective of language designer
• Applicative order
• Dynamic vs. lexical scoping
• Lazy evaluation
  • Full normal order
  • By specifying arguments
  • Just for pairs
• Decoupling analysis from evaluation

static analysis: work done before execution
• straight interpreter
  environment
expression → interpreter → value
• advanced interpreter or compiler
  environment
expr → static analysis → execution → value

Reasons to do static analysis
• Improve execution performance
  • avoid repeating work if expression contains loops
  • simplify execution engine
• Catch common mistakes early
  • garbled expression
  • operand of incorrect type
  • wrong number of operands to procedure
• Prove properties of program
  • will be fast enough, won't run out of memory, etc.
  • significant current research topic
**Eval is expensive**

```
(eval '(define (fact n)
      (if (= n 1) 1 (* n (fact (- n 1)))))) GE
===> undef
```

... (eval '(fact 4) GE) ...
... (eval '(= n 1) E1) ...
which executes the case statement in `eval` four times

... (eval '(fact 3) E1) ...
... (eval '(= n 1) E2) ...
which executes the case statement in `eval` four times

- The `analyze` evaluator avoids this cost

---

**Summary of part 1**

- static analysis
  - work done before execution
  - performance
  - catch mistakes
  - prove program properties
- analyze evaluator
  - static analysis: eliminate execution cost of `eval`

---

**Strategy of the analyze evaluator**

```
analyze expr
```

- `analyze`: expression → (Env → anytype)

```
(define (a-eval exp env)
  ((analyze exp) env))
```

---

**Example of analyze: variable name lookup**

```
foo
```

```
(name pi => 3.14)
```

```
execute
```

```
3.14
```

---

**Implementing variable name lookup**

```
(define (analyze exp)
  (cond
    ((number? exp) (analyze-number exp))
    ((variable? exp) (analyze-variable exp))
    ...)
)
```

```
(define (analyze-variable exp)
  (lambda (env)
    (lookup-variable exp env)))
```

---

**Implementing number analysis**

```
(define (analyze-number exp)
  (lambda (env) exp))
```

```
(black: analysis phase) (blue: execution phase)
```
**Summary of part 2**

- Output of `analyze` is an execution procedure
  - Given an environment
  - Produces value of expression
- Within `analyze`
  - Execution phase code appears inside
    - `lambda (env) ...`
  - All other code runs during analysis phase

**Subexpressions** (hardest concept today)

(\text{analyze} '(\text{if} (= n 1) 1 (* n (...))))

- Analysis phase:
  - \text{analyze} '(= n 1) \Rightarrow pproc
  - \text{analyze} 1 \Rightarrow cproc
  - \text{analyze} '(* n (...)) \Rightarrow aproc

- Execution phase
  - \text{pproc} env \Rightarrow \#t or \#f (depending on n)
    - \#t: \text{cproc} env
    - \#f: \text{aproc} env

**Implementation of \text{analyze-if}**

\text{define (analyze-if exp)}

\{let \{pproc (analyze (if-predicate exp))\}
\{cproc (analyze (if-consequent exp))\}
\{aproc (analyze (if-alternative exp))\}\}
\(\text{lambda (env)}\)
\{if (true? (pproc env))\}
\{cproc env\}
\{aproc env\})))

- Black: analysis phase
- Blue: execution phase

**Visualization of \text{analyze-if}**

\text{analyze}
\(\text{if} (= n 1)\)
\(1\)
\(* n (...))\)

\(\text{pproc} \ldots\)
\(\text{cproc} \ldots\)
\(\text{aproc} \ldots\)

**Your turn**

- Assume the following procedures for definitions like
  \text{define} x (+ y 1)

\{definition-variable exp\}
\(x\)
\{definition-value exp\}
\(+ y 1\)
\{define-variable! name value env\}
\text{add binding to env}

- Implement \text{analyze-definition}
  - The only execution-phase work is \text{define-variable!}
  - The definition-value might be an arbitrary expression

**Implementation of \text{analyze-definition}**

\text{define (analyze-definition exp)}

\{let \{var (definition-variable exp)\}\}
\{vproc (analyze (definition-value exp))\}
\{lambda (env)\}
\{define-variable! var (vproc env) env)\})

- Black: analysis phase
- Blue: execution phase
Summary of part 3

- Within analyze
  - recursively call analyze on subexpressions
  - create an execution procedure which stores the EPs for subexpressions as local state

Implementing lambda

- Body stored in double bubble is an execution procedure
- old make-procedure
  list<symbol>, expression, Env → Procedure
- new make-procedure
  list<symbol>, [Env->anytype], Env → Procedure

\[
\text{(define (analyze-lambda exp)} \\
\text{  (let ((vars (lambda-parameters exp))} \\
\text{    (bproc (analyze (lambda-body exp)))))} \\
\text{  (lambda (env)} \\
\text{    (make-procedure vars bproc env))))}
\]

Implementing apply: execution phase

\[
\text{(define (execute-application proc args)} \\
\text{  (cond } \\
\text{    ((primitive-procedure? proc) ...)} \\
\text{    ((compound-procedure? proc)} \\
\text{      (procedure-body proc)} \\
\text{      (extend-environment (parameters proc)} \\
\text{        (map (lambda proc) (operands proc)))} \\
\text{      (environment proc)))))}
\]

Implementing apply: analysis phase

\[
\text{(define (analyze-application exp)} \\
\text{  (let ((fproc (analyze (operator exp)))} \\
\text{    (aprocs (map analyze (operands exp)))))} \\
\text{  (lambda (env) } \\
\text{    (execute-application} \\
\text{      (fproc env))} \\
\text{    (map (lambda (aproc) (aproc env)) aprocs)))})
\]

Summary of part 4

- In the analyze evaluator,
  - double bubble stores execution procedure, not expression

What is Eval really?

- Suppose you were a circuit designer
  - Given a circuit diagram, you could transform it into an electric signal encoding the layout of the diagram
  - Now suppose you wanted to build a circuit that could take any such signal as input (any other circuit) and could then reconfigure itself to simulate that input circuit
  - What would this general circuit look like???
- Suppose instead you describe a circuit as a program
  - Can you build a program that takes any program as input and reconfigures itself to simulate that input program?
  - Sure – that’s just EVAL!! – it’s a UNIVERSAL MACHINE
It wasn’t always this obvious

- “If it should turn out that the basic logics of a machine designed for the numerical solution of differential equations coincide with the logics of a machine intended to make bills for a department store, I would regard this as the most amazing coincidence that I have ever encountered”

Howard Aiken, writing in 1956 (designer of the Mark I “Electronic Brain”, developed jointly by IBM and Harvard starting in 1939)

Why a Universal Machine?

- If EVAL can simulate any machine, and if EVAL is itself a description of a machine, then EVAL can simulate itself
- This was our example of `meval`
- In fact, EVAL can simulate an evaluator for any other language
- Just need to specify syntax, rules of evaluation
- An evaluator for any language can simulate any other language
- Hence there is a general notion of computability – idea that a process can be computed independent of what language we are using, and that anything computable in one language is computable in any other language

Turing’s insight

- Alan Mathison Turing
- 1912-1954

- Was fascinated by Gödel’s incompleteness results in decidability (1933)
- In any axiomatic mathematical system there are propositions that cannot be proved or disproved within the axioms of the system
- In particular the consistency of the axioms cannot be proved.
- Led Turing to investigate Hilbert’s Entscheidungsproblem
- Given a mathematical proposition could one find an algorithm which would decide if the proposition was true or false?
- For many propositions it was easy to find such an algorithm.
- The real difficulty arose in proving that for certain propositions no such algorithm existed.
- In general – Is there some fixed definite process which, in principle, can answer any mathematical question?
- E.g., Suppose want to prove some theorem in geometry
  - Consider all proofs from axioms in 1 step
  - … in 2 steps ….

- Turing proposed a theoretical model of a simple kind of machine (now called a Turing machine) and argued that any “effective process” can be carried out by such a machine
- Each machine can be characterized by its program
- Programs can be coded and used as input to a machine
- Showed how to code a universal machine
- Wrote the first EVAL!

The halting problem

- If there is a problem that the universal machine can’t solve, then no machine can solve, and hence no effective process
- Make list of all possible programs (all machines with 1 input)
- Encode all their possible inputs as integers
- List their outputs for all possible inputs (as integer, error or loops forever)
- Define \( f(n) = \text{output of machine } n \text{ on input } n \), plus 1 if output is a number
- Define \( f(n) = 0 \) if machine \( n \) on input \( n \) is error or loops
- But I can’t be computed by any program in the list!!
- Yet we just described process for computing it?!
- Bug is that can’t tell if a machine will always halt and produce an answer
The Halting theorem

- Halting problem: Take as inputs the description of a machine M and a number n, and determine whether or not M will halt and produce an answer when given n as an input.
- Halting theorem (Turing): There is no way to write a program (for any computer, in any language) that solves the halting problem.

Turing’s history

- Published this work as a student
  - Got exactly two requests for reprints
    - One from Alonzo Church (professor of logic at Princeton)
      - Had his own formalism for notion of an effective procedure, called the lambda calculus
  - Completed Ph.D. with Church, proving Church-Turing Thesis:
    - Any procedure that could reasonably be considered to be an effective procedure can be carried out by a universal machine (and therefore by any universal machine)
- Worked as code breaker during WWII
  - Key person in Ultra project, breaking German’s Enigma coding machine
  - Designed and built the Bombe, machine for breaking messages from German Airforce
  - Designed statistical methods for breaking messages from German Navy
  - Spent considerable time determining counter measures for providing alternative sources of information so Germans wouldn’t know Enigma broken
  - Designed general-purpose digital computer based on this work
  - Turing test: argued that intelligence can be described by an effective procedure – foundation for AI
  - World class marathoner – fifth in Olympic qualifying (2:46:03 – 10 minutes off Olympic pace)
  - Working on computational biology – how nature “computes” biological forms.
  - His death