Aspects of Language Design

- **Syntax**: How the language looks to the user.
- **Semantics**: What the language means; how it behaves when executed.
- **Pragmatics**: Practical aspects of working with the language in a development environment.
What’s Most Important?

• Semantics, by far

• There can be many different syntaxes for the same semantics.
Syntaxes: “Hello World!” in Various Languages
They all mean basically the same thing.

// Hello World in Java

```java
class HelloWorld {
    static public void main(String args[]) {
        System.out.println("Hello World!");
    }
}
```

; Hello World in Scheme

```
(display "Hello, world!")
```

(newline)

// Hello World in Python

```
#include <iostream>

main() {
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

% Hello World! in LaTeX

```
\documentclass{article}
\begin{document}
Hello World!
\end{document}
```

# Hello World in Python

```
print "Hello World"
```

% Hello World in Prolog

```
:- write('Hello World!'), nl.
```
Scheme Syntax

• Scheme finesses the issue of syntax by taking a rather minimalist approach.

• The syntax of Scheme reflects little more than the “abstract syntax” of a language.
Abstract Syntax?

• Abstract syntax means recognizing that various language constructs have specific parts of specific types, but paying little attention to how the connection of those parts is specified.
Abstract Syntax Example: “if” construct

• Typical “if” has 3 parts:
  • test part
  • true branch
  • false branch
“if” in C or Java

if(...)
    /* conditional part */
    {
        . . . /* true branch */
    }
else
    {
        . . . /* false branch */
    }
“if” in Python

if ... :    # conditional part
    ...
    # true branch
else:
    ...
    # false branch
“if” in Scheme

(if ... ; conditional part
    ... ; true branch
    ... ; false branch
)
Scheme’s “if” is Abstract

• Scheme’s “if” looks like everything else in Scheme:
  ( keyword ... parts ... )

• This is both good and bad.
The Bad

- Since everything looks alike, there are few visual clues, other than keywords and indentation, to differentiate one construct from another.
The Good

• The syntax is very uniform, so that part of compilation is made trivial.

• Adding new constructs to the language is easy; we don’t have to stop and think up symbols.

• Programs can be read as data easily, allowing new interpreters or transformation systems to be constructed.
Same semantics, Different syntax

• We’ll develop the language using abstract syntax.

• If later desired, we can add on a front-end to provide different syntax.
Making our Own Scheme-Like Language

• Entire S expressions can be read with one statement, so no scanning step is necessary.

• An S expression is either:
  • an atom (symbol, numeral, string, etc.)
  • a list (begins and ends with parens)
S Expression Examples

• (This is one S expression)
• (So are the following)
  • abcd
  • efg345
  • 678
  • (a (deeply (nested (expression)))))
• (S stands for “symbolic”)}
Let’s say we want a language to do logic design.

The syntax will resemble a subset of Scheme.

The focus is on logic functions and equations.
Language Elements

• Domain: 0 for false, 1 for true

• Functions: and, or, not, xor, nand, nor, implies, =

• Definable variables

• if, let, let, let*

• User-definable functions via lambda expressions
Language Semantics

• The semantics of the language will be defined by giving it’s Eval function.

• We’ll use a capital “E”, because “eval” is already built into Scheme (it is an interpreter for Scheme itself).
Example Arguments to Eval

- 0
- (and 1 0)
- (or (and 1 (not 0)) (nor 1 0))
- \(x\)
- (or (and \(x\) (not \(y\))) (nor \(z\) \(w\)))
- (if \(x\) \(y\) \(z\))
- (let (((\(x\) (and \(y\) \(z\))) (w (nor \(y\) \(z\)))) (or \(x\) \(w\)))
- and others
Defining Eval

(define (Eval exp env) ...)

returns the result of evaluating the exp argument, which will ultimately come in as an S expression from the command line.

The purpose of env will be discussed later.
Basis for Eval

• The basis consists of $S$ expressions that are not lists, such as 0, 1, and identifiers.

• The recursion part consists of lists that represent composites, such as function application, etc.
Dichotomy

(define (Eval exp env)
  (if (list? exp)
      (Eval-composite exp env) ; e.g. '(and 0 1)
      (Eval-basic exp env))) ; e.g. 0, 1, 'x
Eval-basic

(define (Eval-basic exp env)
  (cond
   ((constant? exp) exp)
   ((variable? exp) (get-value exp env))
   (else (Eval-error "unrecognized" exp))))

All functions are subject to later modification as we design.
Avoid “Magic Numbers”

(define logic-true 1) ; May wish to change later
(define logic-false 0)

(define (true? x) (equal? logic-true x))
(define (false? x) (equal? logic-false x))
(define (constant? x) (or (true? x) (false? x)))

(define (variable? var)
  (symbol? var))
Eval-error

(define (Eval-error msg exp) (error msg exp))

error is built-in and throws an exception, meaning that a jump out of Eval takes place.

error does not return a value.
First Test Cases

(define base () ; Discussed later
(test (Eval 0 base ) logic-false)
(test (Eval 1 base ) logic-true)
Eval-composite

• When Eval-composite is called, we already know the argument is a list.

• But it could be empty.

  (define (Eval-composite exp env)
    (if (null? exp)
        (Eval-error "empty list is meaningless" exp)
        (Eval-operator (first exp) (rest exp) env))))
(define (Eval-operator operator actuals env)
  (case operator
    ('not (Eval-not actuals env))
    ('and (Eval-and actuals env))
    ('or (Eval-or actuals env))
    (else (Eval-error "unrecognized operator in "
      (cons operator actuals))))))

;; actuals means actual arguments
Eval-not

(define (Eval-not actuals env)
  (if (length1? actuals)
      (if (true? (Eval (first actuals) env))
          logic-false
          logic-true)
      (Eval-error "wrong arguments to not operator" actuals)))

(define (length1? x)
  (and (list? x) (not (null? x)) (null? (rest x)))))
First test cases for not

(test (Eval '(not 0) base) logic-true)
(test (Eval '(not 1) base) logic-false)
Eval-and

(define (Eval-and actuals env)
  (if (null? actuals)
      logic-true
      (if (true? (Eval (first actuals) env))
          (Eval-and (rest actuals) env)
          logic-false)))
Test cases

(test (Eval '(and 0 0) base) logic-false)
(test (Eval '(and 0 1) base) logic-false)
(test (Eval '(and 1 0) base) logic-false)
(test (Eval '(and 1 1) base) logic-true)
Other operators are similar

- or, nand, nor, xor, implies, etc.
Testing Nested Expressions

• Must do something similar for other operators (and, or, ...)

(test (Eval '(not (not 0))) 0)
(test (Eval '(not (not 1))) 1)
(test (Eval '(not (not (not 0)))) 1)
(test (Eval '(not (not (not 1)))) 0)
Handling Variables

• The env ("environment") argument holds the bindings of variables to values.

• A convenient way to represent it is by an association list.

• Then assoc can be used to look up the variable and get the binding.
Getting the Value of a Variable

(define (get-value var env)
  (let ((found (assoc var env)))
    (if found
      (second found)
      (Eval-error "unbound variable" var))))
let special form

- *let* is one means by which variables get bindings.

- A *let* form consists of a list of “equations” and a result expression.

- An “equation” is a list of pairs: a lhs variable and a rhs expression.

- The rhs’s are evaluated in the outer environment, then the result expression is evaluated in a new environment created by adding bindings to the outer environment.
Eval-let

(define (Eval-let actuals env)
  (if (length2? actuals)
      (let* (
          (equations (first actuals))
          (result-exp (second actuals))
        )
        (if (well-formed? equations)
            (let* (
                (lhs-vars (map first equations))
                (rhs-vals (map (lambda (eqn) (Eval (second eqn) env)) equations))
                (new-env (add-bindings lhs-vars rhs-vals env))
            )
            (Eval result-exp new-env))
            (Eval-error "error in equations of let" (cons 'let actuals)))
        (Eval-error "let construct must be a list of length two" (cons 'let actuals))))
add-bindings

(define (add-bindings vars vals env)
  (if (null? vars)
      env
      (add-bindings (rest vars) (rest vals) (cons (list (first vars) (first vals)) env))))
Example Test for Nested *let*

(test (Eval
  '(let ((x 0) (y 1))
    (let ((u (and x y)) (v (or x y)))
      (and (not u) v)))
    base)
  logic-true)
Making a Command-Line Interface

• A command-line interface (CLI) is one that sequentially accepts input from a user or file and reacts to it.

• Examples are: Scheme command line, Python command line, Unix command line, Windows DOS command line.
Read-Eval-Print Loop (REPL)

- A REPL is a special case of a CLI, that repeats the following steps:
  - Read an expression
  - Evaluation the expression
  - Print the result
- This “loop” continues until some kind of end is signalled, such as:
  - A special expression, or
  - end-of-file.
The procedure `read` reads an entire Scheme expression at once, returning the value read:

```
(read)
```

This value is then handed to an evaluator, to produce a value, and then another procedure, `print`, can print the result:

```
(print ....)
```
Looping

• Looping is handled by an un-ending tail-recursive call.

• For termination, read will return a special value end-of-file when end-of-file occurs.

• A conditional test can be used to determine whether or not to continue looping.
Prompting

- Most CLI’s have some form of prompt to indicate to the user that the program is awaiting input.

- This can be accomplished by another print that prints the prompt.
Sequencing

• We want procedures to be called in a specific order:
  • prompt
  • read
  • check for end-of-file
  • if end-of-file, return
  • evaluate
  • print result
  • call recursively
begin form

• begin is a special form in Scheme that forces evaluation in first-to-last order.

• The value of the form is that of the last argument in the form.

\[ > \text{(begin 'a 'b 'c)} \]
\[ c \]
Sample REPL

define (read-eval-print)
  (begin
    (prompt)
    (let ((expression (read)))
      (if (eof-object? expression)
          expression
          (begin
            (print (Eval expression ())))
            (read-eval-print)
          )))
  )
Supporting Definitions

(define (prompt)
  (newline)
  (display ">"))
Example from REPL User's View

> (and 1 0)
0
> (and 1 1)
1
> (or 1 0)
1
> (or (and 1 0) (and 1 1))
1
> (not (or (and 1 0) (and 1 1)))
0
EOF in Dr. Scheme

• Click on the `eof` symbol to force end-of-file

Dr. Scheme type-in window
EOF in Unix/Linux

- The standard convention for end-of-file in Unix/Linux is to enter control-d
Handling User Errors

• If user input results in error being called, the exception will throw out of the REPL and looping stops.

• To avoid this, we can catch the exception with an error handler.

• The error-handler is “wrapped around” the call to the evaluator.

• It returns a designated value when an exception is caught.
Example REPL with Error Handler

(define (error-handler x)
  (display (list "*** error:" x)))

(define (read-eval-print)
  (begin
    (prompt)
    (let (
      (expression (read))
    )
    (if (eof-object? expression)
      expression
      (begin
        (with-handlers (((lambda(x) #t) error-handler))
          (print (Eval expression ())))
        (read-eval-print)
      ))
  ))
User’s View Upon Error

> (and 0 1)
0
> (foo 0 1)
(*** error: #(struct:exn:fail
    unrecognized operator in (foo 0 1)
    #<continuation-mark-set>))
> (or 0 1)
1

A “struct” is being returned by the handler. We can present nicer error messages by extracting the message part.
Advice

• Debug most of your program *without* the error handler in the REPL, because it will catch your errors too, but not stop looping.

• This can be annoying, because you have to enter *eof*, then edit.

• Add the error handler in after (you think) everything else is working.