Scheme

Shares many features with ML:
• expression-oriented
• list-oriented, garbage-collected heap-based
• functional
  • functions are first-class values
  • largely side-effect free
• strongly typed
• highly regular and expressive

Unlike ML:
• dynamically typed, not statically typed
• lacks
  • pattern matching
  • exceptions (but has continuations)
  • modules (but some Scheme extensions have good modules)
• syntax blends data and program

Lisp designed by McCarthy in late 50’s
Scheme dialect introduced by Steele and Sussman in mid 70’s

Syntax

Program ::= { Definition | Expr }

Definition ::= (define id Expr) | (define (id fn id formal1 ... id formalN) Expr)

Expr ::= id | Constant | SpecialForm | (Expr fn Expr arg1 ... Expr argN)

Constant ::= int | float | string | symbol | (lambda (id formal1 ... id formalN) Expr) | ...

SpecialForm ::= (if Expr test Expr then Expr else Expr) | ...

Uniform prefix “calls”

Examples:
(+ 3 4) → 7
(+ (* 3 8) (/ 8 2)) → 28
(define seven (+ 3 4))
seven → 7
(+ seven 8) → 15
(define (square n) (* n n))
(square seven) → 49
(define (fact n)
  (if (<= n 0) 1
      (* n (fact (- n 1))))
(fact 20) → 2432902008176640000

Prefix operators & function calls is regular, and unambiguous, but not “traditional”
• don’t have to define precedence and associativity!
• can have 0, 1, 2, or many arguments to a “binary” operator

Special forms

Regular call expressions evaluation all arguments then invoke procedure
• user-defined procedures work this way

Special forms are special “functions” where arguments aren’t all treated as expressions to be evaluated first
• can define new special forms using special macros

Example:
(define x 0)
(define y 5)
(if (= x 0) 0 (/ y x)) → 0
(define (my-if test then else)
  (if test then else))
(my-if (= x 0) 0 (/ y x)) → error!
Other special forms

`cond` like if-elseif...-else chain:

```
(cond ((> x 0) 1)
     ((= x 0) 0)
     (else -1))
```

Short-circuiting and and or (like ML’s andalso and orelse)

```
(or (= x 0) (> (/ y x) 5) ...)
```

let: “simultaneous” local variable bindings:

```
(define x 1) (define y 2) (define z 3)
(let ((x 5)
       (y (+ 3 4))
       (z (+ x y z)))
  (+ x y z)) → 5+7+(1+2+3)=18
```

let*: “sequential” local variable bindings (like ML’s let):

```
(let* ((x 5)
       (y (+ 3 4))
       (z (+ x y z)))
  (+ x y z)) → 5+7+(5+7+3)=27
```

Lists

Translation between ML and Scheme

<table>
<thead>
<tr>
<th>ML</th>
<th>Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>()</td>
</tr>
<tr>
<td>x :: xs</td>
<td>(cons x xs)</td>
</tr>
<tr>
<td>[x, y, z]</td>
<td>(list x y z)</td>
</tr>
<tr>
<td>hd(lst)</td>
<td>(car lst)</td>
</tr>
<tr>
<td>tl(lst)</td>
<td>(cdr lst)</td>
</tr>
<tr>
<td>null(lst)</td>
<td>(null? lst)</td>
</tr>
</tbody>
</table>

Examples:

```
(define lst (list 5 6 7 8)) → (5 6 7 8)
(define lst2 (cons lst)) → (4 5 6 7 8)
(+ (car lst) (car lst2)) → 9
(define lst3 (cdr lst)) → (6 7 8)
• lst, lst2, and lst3 have shared subpieces
```

Dynamic typing

There are no static types, neither explicit nor inferred
Any variable, and any data structure, can hold any type of value
Values have (run-time) types, variables are typeless

Typechecking is performed only when absolutely necessary
E.g.
- car & cdr check that argument is a cons cell, and
- + checks that arguments are numbers, but
- cons and list check nothing!

Lists can be heterogenous:

```
(list 3 4.5 () "hi" (list 3 5))
→ (3 4.5 () "hi" (3 5))
• lists in Scheme fulfill roles of both tuples and lists in ML
```

E.g. an association list of key-value pairs:

```
(define Zips (list (list "Seattle" 98195)
                   (list "Boston" 02115)
                   (list "Reston" 22091)))
→ ("Seattle" 98195)
   ("Boston" 02115)
   ("Reston" 22091))
```

Type testing

Programs can test the type of values at run-time

Some type-testing predicates:

```scheme
null?
pair?
symbol?
boolean?
number? integer? ...
string? ...
```
Typechecking terms

**Static vs. dynamic typing**: when are type checks performed?
- static: before execution
- dynamic: during execution

Pure static typing is too restrictive, so statically typed languages often mix static and dynamic checking

**Strong vs. weak typing**: how comprehensive are type checks?
- strong: guarantee no run-time misuses
- weak: don’t

The two dimensions are independent

Type errors are a somewhat arbitrary subclass of program errors
Typechecking doesn’t address non-type errors

Quoting

List literals via `quote` or `'` special form:

- `(list 3 (list 4 5) 6)` → `(3 (4 5) 6)`
- `(quote (3 (4 5) 6))` → `(3 (4 5) 6)`
- `'(3 (4 5) 6)` → `(3 (4 5) 6)`

Quoted identifiers are `symbol` constants:

- `'positive` → `positive`
- `(car 'if (> a b) 3 4))` → `if`

Programs and data share same regular syntax
Makes it very easy to write programs that build, take apart, and transform programs