Scheme

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 0.3 true - quotient ...

- Combinations: (quotient 10 2). (not true). ...

Numbers are self-evaluating; symbols are bound to values. Call expressions have an operator and 0 or more operands.

A combination that is not a call expression is a special form:

- If expression: (if <predicate> <consequent> <alternative>)
- Binding names: (define <name> <expression>)
- New procedures: (define (<name> <formal parameters>) <body>)

Lambda expressions evaluate to anonymous procedures.

Two equivalent expressions:

- (define (plus4 x) (+ x 4))
- (define ((lambda (x y z) (+ x y (square z))) 1 2 3)

An operator can be a combination too:

- ((lambda (x y z) (+ x y (square z))) 1 2 3)

Scheme Special Forms

- Scheme Built-In Procedures
- Scheme Special Forms
- Lambda

Scheme Lists

In the late 1950s, computer scientists used confusing names.

- cons: Two-argument procedure that creates a list
- car: Procedure that returns the first element of a pair
- cdr: Procedure that returns the second element of a pair
- nil: The empty list

They also used a non-obvious notation for linked lists.

- A (linked) Scheme list is a pair in which the second element is nil or a pair.
- Scheme lists are written as space-separated combinations.

- A dotted list has an arbitrary value for the second element of the last pair. Dotted lists may not be well-formed lists.

Examples:

- (define (mu (x) (+ x y)))
- (define b 4)
- (define a 1)
- (define (abs x) (if (< x 0) (- x) x))
- (define pi 3.14)
- (define total 0)
- (define (new x n) (+ x n))
- (define (factorial n) (if (= n 0) 1 (* n (factorial (- n 1)))))

A function that can apply any function expression to any list of arguments:

- (define (call-func func-expression func-args)
  (apply (eval func-expression) func-args))

Scheme Tail Calls

A procedure call that has not yet returned is active. Some procedure calls are tail calls. A Scheme interpreter should support an unbounded number of active tail calls.

A tail call is a call expression in a tail context, which are:

- The last body expression in a lambda expression
- Expressions 2 & 3 (consequent & alternative) in a tail context
- All sub-expressions in a tail context cond
- The last sub-expression in a tail context and, or, begin, or let

Examples:

- (define (factorial n k)
  (if (= n 0) k
  (+ (* n (- (+ k 1) n)) (factorial (- n 1))))))

- (define (length s)
  (if (null? s) 0
  (+ (length (cdr s)) 1)))

- (define (length-tail s n)
  (if (null? s) n
  (if (null? (cdr s))
  (+ 1 (length-tail (append (list 1) 3) 4))
  (length (append (list 1) 3) 4))

- (define (factorial 12 3 4)
  => 4
  (map (lambda (x) (+ x 3)) '3 4)
  => (7 8 9)

- (filter odd? '2 3 4))
  => 3
  (reduce + '2 3 4 5)
  => 15
  (list (length (list 1 2 3 4) 5)
  (list (cons 1 nil) size 'size))
  => (1 5 size)
  (define (length-tail s n)
  (if (null? s) n
  (if (null? (cdr s))
  (+ 1 (length (cdr s))}))

Not a tail call
A basic interpreter has two parts: a parser and an evaluator.

### Expression Trees

- A Scheme list is written as elements in parentheses: `()`, `a`, `...`, `element> ... element> A Scheme list`

- Each `<element>` can be a combination or atom (primitive).

- Base case: one expression.

- Each call to scheme_read consumes the input tokens for exactly one expression, which may be nested.

- Syntactic analysis identifies the hierarchical structure of an expression.

- Call to scheme_read consumes the input tokens for exactly one expression.

- Base case: symbols and numbers

- Recursive call: scheme_read sub-expressions and combine them

#### Calculator

The Calculator language has primitive expressions and call expressions

**Calculator Expression**

- `( + 3 (* 4 5) (+ 6 7 8))`

**Expression Tree**

```
                   +
         +        +
        *        7
       /  
      +  
     /   
    +   8
   /   /  
  6   5   1
```

**Exceptions in Python**

- Exceptions are raised with a raise statement.

```
raise <expr>
```

- `<expr>` must evaluate to a subclass of BaseException or an instance of one.

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
```

**Regular Expressions**

- Matches any character `.\.`

```
\w\w\w\w
```

- Matches letters, numbers or _`

```
\w\w\w\w
```

- Matches a digit `_d\d`

```
\w\w\w\w
```

- Matches a whitespace `_d\d`

```
\w\w\w\w
```

- Encloses a list of options or ranges `_d\w\d` and `_d\w\d`

```
\w\w\w\w
```

- A word followed by `.\.`

```
\w\w\w\w
```

- A letter or number (or _`

```
\w\w\w\w
```

- Any letter (upper or lower case `_a-zA-Z`)

```
\w\w\w\w
```

- Exactly three letters

```
\w\w\w\w
```

- One or more letters/numbers

```
\w\w\w\w
```

### Backus-Naur Form

Rules consist of TERMINALS (all caps) and non-terminals (lower case).

#### TERMINALS only contain /reg ex/, "text", and other TERMINALS non-terminals can expand into either non-terminals or TERMINALS.

- item1 | item2
- (item item...) | Group items
- item? - Zero or one instances of item ("maybe")
- item* - Zero or more instances of item
- item+ - One or more instances of item

```
item1 | item2 | (item item...) | item? | item* | item+
```

#### Calculator BNF:

```
?calc_expr: NUMBER | calc_op
calc_op: \("(\ OPERATOR calc_expr\")\"
OPERATOR: \"\+\" | \"\-\" | \"\*\" | \"\/\"
```

The parse tree:

```
  calc_op
  \+ \- \* \/
  \+ \- \* \/
  \+ \- \* \/
  \+ \- \* \/
```

Remove nodes whose rules start with ? and have only 1 child, replacing them with that child.

Remove unnamed literals (like "\+" entirely.)