Introduction

- Powerful programming language
    - Primitive expressions - simple data types
    - Means of combination - able to form compound elements
    - Means of abstraction - compound elements named and manipulated, named

1.1.1 - Expressions - in Scheme

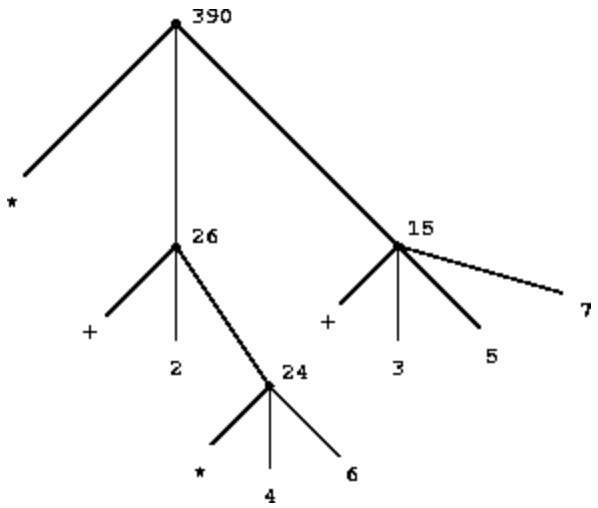
- Ex.
    - (* 122 8), known as prefix notation
        - Can take arbitrary number of elements
        - Expressions can be nested easily
    - First element = operator
    - Second & third = operands, arguments
    - Pretty-printing
        - Ex.
        - (+ (* 3
        - (+ (* 2 4)
        - (+ 3 5)))
        - (+ (- 10 7)
        - 6)
        - Operators are lined up
- Scheme always performs in read-eval-print loop

1.1.2 - Naming and the Environment

- Name defines variable
    - Value of variable = object
- In LISP
    - Abstraction using *define*
    - Ex.
      (define pokemon 69)
        - pokemon = variable
        - 69 = value referenced by the variable
- Naming/abstraction allows step by step increments, each of higher complexity
- To have abstraction/ability to remember names, have some sort of memory
    - Memory = environments (i.e. global environments)

1.1.3 - Evaluating Combinations

- LISP evaluations of a combination - uses recursion (during evaluation, the rule itself needs to be used)
- Recursive combination evaluating represented by tree
    - Ex.



- Terminal nodes - nodes on bottom - represent arguments / operands
    - Percolates values upward
- Problems structured similar to this (generally known as *tree accumulation*) can be solved by recursion
- Within recursive programs - at most basic levels, basic assumptions are necessary for evaluation of combinations

    (combinations consist of a operator being applied to two arguments in that they are evaluated, not associated)
    - Values of numerals are numbers they name (ex. 1 = 1)
    - Values of built-in operators are the machine instructions carrying out corresponding operations
    - Values of other names are defined in the environment

- Environments = very important for program execution
    - Provides a context
- **Special forms** - exceptions to general evaluation rule (operators that don't form combinations)
    - Each special form has own evaluation rule

1.1.4 - Compound Procedures

- Powerful programming language includes
    - Primitive data types - e.g. operators, numerals
    - Nesting combinations
    - Definition of abstract values (define variables)
        - More powerful - **procedure definition (define function) - compound operation given a name, can be referred to as a unit**
- General form of procedure definition

    (define (<name> <formal parameters>) <body>)
- <name> = variable/symbol associated with procedure definition, in the environment
- <formal parameters> = arguments of the procedure

- <body> = expression which the formal parameters will replace in outlined spots to which then the expression is evaluated - the “rule”

1.1.5 - Substitution Model for Procedure Application

- To evaluate compound procedure (a way to look at it)
    - Substitution model
        - Each formal parameter is replaced by the corresponding argument w/ each sub-combination
        - Reduces until final value is evaluated
        - Model != show how interpreters evaluate compound procedures
        - As evaluation become more complex, newer models needed
        - Ex.

(f 5)

where f is the procedure defined in section 1.1.4. We begin by retrieving the body of:

(sum-of-squares (+ a 1) (* a 2))

Then we replace the formal parameter a by the argument 5:

(sum-of-squares (+ 5 1) (* 5 2))

Thus the problem reduces to the evaluation of a combination with two operands and an operator sum-of-squares. Evaluating this combination involves three subproblems.

Now (+ 5 1) produces 6 and (* 5 2) produces 10, so we must apply the sum-of-squares procedure to 6 and 10. These values are substituted for the formal parameters x and y in the body of sum-of-squares, reducing the expression to

(+ (square 6) (square 10))

If we use the definition of square, this reduces to

(+ (* 6 6) (* 10 10))

which reduces by multiplication to

(+ 36 100)

136

Application Order v. Normal Order Evaluation

Normal Order	Applicative Order

• Expands the entire expression before reducing	• Evaluates each argument and operator, then applies the evaluation process • How interpreters usually handle evaluations • Used by LISP

- Both usually produces the same results, unless the expression is illegitimate

1.1.6 - Conditional Expressions and Predicates

- **SPECIAL FORM =**
- **Case Analysis**
  - In LISP
  - Used as a form of flow control - to make tests and perform procedures based on the result of those tests
  - Form

$$(cond \quad (\langle p_1 \rangle \ \langle e_1 \rangle) \quad (\langle p_2 \rangle \ \langle e_2 \rangle) \quad \vdots \quad (\langle p_n \rangle \ \langle e_n \rangle))$$

- $(\langle p_1 \rangle \ \langle e_1 \rangle)$ = clauses
- Test / conditional statement = $\langle p_n \rangle$ = predicate
- Cond expressions test each predicate for true
  - If True, returns the resulting $\langle e_n \rangle$ value
  - If all are false, the conditional expression is undefined
- (<else> <e>) can be placed at end of conditional statements
  - Will be the value of cond statement is all predicates are false
  - Any predicate which always evaluates to true can also be used
- *(if <predicate> <consequent> <alternative>)*
  - **If** - special form
  - If predicate is true, then consequent is the value of statement
    - Otherwise, alternative = value
    - State special - only evaluates one or the other expressions, not both (which depends on predicate's boolean value)

Common Predicates	
<	Greater than
>	Less than
=	Equal to

and *(and <e_1> ... <e_n>)*	Checks if all the <e_n> are true. If not, returns false. If all true, value of expression is value of last one
or *(or <e_1> ... <e_n>)*	If any <e_n> are true, that <e_n>is the value of the expression If all <e_n> are false, expression is false
not *(not <e>)*	Value of expression is True when <e_n> is false Otherwise, expression is True

--Interpreter evaluates in a left to right order--

1.1.7 - Example: Square Roots by Newton's Method

- Declarative descriptions
  - i.e. functions
  - Describes the properties of things
  - More in mathematics
- Imperative knowledge
  - i.e. procedures
  - Describes how to do something
  - More in computer science
- Newton's method of square roots
  - X = number to be square rooted
  - Take a guess, y
  - Divide x/y, take quotient and average, z, with x
  - Continue the process with the average with new quotient of x/z and z
  - Ex.

Sqrt(2)		
Guess	**Quotient**	**Average**
1	(2/1) = 2	((2 + 1)/2) = 1.5
1.5	(2/1.5) = 1.3333	((1.3333 + 1.5)/2) = 1.4167
1.4167	(2/1.4167) = 1.4118	((1.4167 + 1.4118)/2) =

		1.4142
1.4142	...	...

Exercise 1.6

The new-if differs from the interpreter's if in that both expressions specified within the new-if are evaluated, as opposed to only the single expression evaluated within the if special form based on the value of the predicate. Thus, when new-if is implemented within the sqrt procedure, the sqrt-iter alternative expression will repeatedly be evaluated, resulting in an infinite recursion loop.

Exercise 1.7

The test will fail for very small numbers as the tolerance range is relatively large for square roots of numbers less than 1.

For example, when inputting the value (sqrt 0.0001), the resulting value is 0.03230844833048122

While, the expected answer would be 0.01, which then creates a 200% of error.

For larger numbers, machine precision is the issue. Most machines can only represent so many places and decimals, and thus after a certain amount (as in a very large number) the value will stagnant and not be able to change. This thus applies to the difference between very large numbers, whereas the machine cannot represent such small differences. When applied to the formula of (sqrt x), (improve-guess) will not be able to provide a better guess past a certain level of precision, and with large numbers the square thus might not ever be reached a level of precision specified by (good-enough?).

Improvement to good enough

```
(define (good-enough? guess x)
  (< (abs (- (improve guess x) guess))
        (* guess 0.0000001)))
```

Exercise 1.8

```
#lang sicp

(define (cub-iter guess x)
  (if (good-enough guess x)
```

```
    guess
    (cub-iter (improve guess x) x)))

(define (good-enough guess x)
  (< (abs (- (improve guess x) guess))
    (* guess 0.0000001)))

(define (improve guess x)
  (/ (+ (/ x (sqr guess))
        (* 2 guess))
    3))

(define (sqr x)
  (* x x))

(define (cubrt x)
  (cub-iter 1.0 x))
```

1.1.8 - Procedures as Black Box Abstractions

- Complex and large programs can be broken down into parts
    - Each has an identifiable task
    - One method through *procedural abstraction* / *black box abstraction*
        - As procedures are defined, unwritten procedures can be abstracted as a name - they then become a black box
        - The black box procedures can be written at a later time, or borrowed from another source
            - Needs to be user-friendly somehow
        - Black box procedures can be anything which gets the results needed

Local Names

- Names of a procedure are local to the body of the procedure
- Meaning of procedure should be independent of name of parameter names
- Black box - has a plug and use convenience
- *Bound variable* - the names of the formal parameters
    - Procedure definition binds formal parameters to those names
    - When free variable name changed to bound name (mistakenly, etc.), known as - *capturing*
- *Free variable* = unbound variable
- *Scope* - set of expressions for which a binding (of a variable to a name) defines a name
- Of a procedure
    - It's meaning should be independent of its bound variables

    - However, most likely its meaning is dependent of the free variables used
        - (Value of resulting procedure would change if say an operator is changed to a different value)
- Formal parameters local to body of the procedure

Internal Definitions and Block Structures

- *Block Structure* - the nest of auxiliaries definitions within a main definition as to localize the definitions of the auxiliary procedures
    - Allows other procedures to have same names for auxiliaries definitions if this was for example part of a larger-scale program
- *Lexical scoping* - simplifying blocked procedures by using the bound variables (formal parameters) of the main procedures
    - Instead of passing the bound variables into each internal procedure's formal definition, the bound variables can be treated as free variables
    - Passed directly into each expression