COURSE INFORMATION

Text:

No text. The instructor will provide handouts from many different books.

Evaluation:

Available grades:

non-completion: Incomplete, Withdraw, etc.

completion: A A- B+ B B- C

A:	reserved for superior performance
A- or B+:	expected grade for conscientious performance
B:	adequate work
B-:	barely adequate
C:	equivalent to failing

Grading Options:

1. Performance Quality: attendance, participation, assigned exercises
2. Grading Contract: specify a set of behaviors and an associated grade.
3. Self-determined: negotiate with instructor

Discussion:

If you prefer a clearly defined agenda, if you do well with concrete task assignments,
or if you need a schedule of activities for motivation, then **Option 1** is a good idea.

If you already understand the field, if you plan to excel in a particular area,
or if you need clear performance goals for motivation, then **Option 2** is a good idea.

If you are not concerned about grades, if you intend to do what you choose anyway,
or if you are self-motivated, then **Option 3** is a good idea.

I will notify any student who is not on a trajectory for personal success.

Languages and Style:

The course will emphasize how to think about, design, and select programming languages and metaphors for particular applications. Students may use the OS and programming languages of the choice for programming exercises. Most assignments will have an implementation component.

Designing the Curriculum

Your name:

1. Write down three questions that you would most like answered by this class.

2. Write down two things that almost everybody in the class will understand by June.

3. Write down the one thing that most concerns or worries you about taking this class.

Assignment 1

A short oral presentation in class.

Tell a detailed story about a programming experience you have had.

A good story revolves around a character and a life experience. It might have some tension, some humor, a critical event, and some learning.

A programming story should be about or include code, before and after the critical event.

You should include what you learned from the experience, and perhaps how you would like things to change.

Tell why you chose the particular story that you did choose. Why is it interesting or important to tell? Is there a moral?

Course Syllabus

Week 1:
Overview of programming metaphors.
Curriculum planning.
Assignment 1: Tell a programming story

Week 2:
Overview of major languages. Pseudocode
Syntax, parsers, BNF, automata.
Assignment 1 due.

Week 3:
FORTRAN, subroutines, name space
ALGOL, hierarchy, blocks
Assignment 2: Pseudocode compiler

Week 4:
PASCAL, simplicity, data typing
Dynamic and static scoping, control structures.
Program semantics and pragmatic modeling
Assignment 3: Pseudocode emulator
4/18: Assignment 2 due

Week 5:
ADA, modularity, abstraction
packages, concurrency
Assignment 4: Semantic model
Assignment 3 due.

Week 6:
LISP, functional style, symbol processing, recursion, garbage collection
A small interpreted language.
Assignment 4 due.

Week 7:
PROLOG, declarative style
logical programming, pattern-matching
Assignment 5 (major):

Week 8:
Smalltalk and JAVA, object-oriented style
data abstraction and modularity, agents

Week 9:
Mathematica
modern and new techniques
Assignment 5 due

Week 10:
Closure.

Versions of Factorial

Focal concepts:

Each of these encodings of the *factorial function* is functionally equivalent. How they achieve the functionality differs.

Almost all are legitmate Mathematica code. Since the core process in Mma is the same for each encoding, we have a demonstration that all are *statically equivalent*. Dynamically, ie how the code runs, all are different.

The *style of encoding* should match as closely as possible the form of the natural problem. Second, the style should match the coder's natural way of thinking about the problem.

Types of *dynamic differences* include:

- ***Syntactic sugar***: the same dynamic behavior (ie the same language). Macros expand the sugared notation *at read-time* into standard notation. Eg:

```
(a + b)  ==>  +[a,b]

declare a=5;  (a + b)
```

- ***Functional syntactic sugar***: shorter and specialized versions of functions. The compiler usually standardizes these variants. Eg, all of the various loop constructs are the same.

```
for i=1 to n do Process[i]

i:=0; (do Process[i]; i:=i+1 until i=n)

dotimes[n, Process[#]]

StreamProcess[IntegerStream[1, n]]
```

- ***Functional model difference***: different processes for achieving the same functional objective. Most of these compile into different machine instructions, but a good optimizing compiler might standardize some of them. Eg: iteration vs recursion vs mapping

```
do[i from 1 to n, acc from nil, Process[i, acc]]

(if i=n, acc, Process[i-1, F[acc, i]])

(if i=n, 0, F[i, Process[i-1]])

map[Process, {1,i,n}]
```

- ***Operational difference***: different engines achieve the same objective but use different operational characteristics. Eg:

```
F[1]=1; F[n]= G[n, F[n-1]]

(if test[n] then (res:=F[i], ++i) else res)

(send F, n)
```

- ***Mathematical difference***: different mathematical computations achieve the same objective but use different models. Eg:

```
F[n] = G[n]              eg Fac[n]=Gamma[n+1]

Decode[Process[Encode[F,n]]]

When (F[Guess[n1] - F[Guess[n2]] = <small>), F[n1]
```

- ***Level of Implementation difference***: different processes occur at different levels of abstraction. Eg:

```
2 + 5 = 7

010 + 101 = 111

r1=Load[i0]; r2=Fetch[j0]; r3=Add[r1,r2]; Store[r3]

b0 = xor[i0,j0]; b[1] = xor[i1,j1]
```

VERSIONS

```
1.  proceduralFactorial[n] :=
      if ( Integer[n] and Positive[n] )
            then
                  Block[ {iterator = n,
                          result = 1  },
                    While[ iterator != 1,
                        result := result * iterator;
                        iterator := iterator - 1 ];
                    return result]
            else  Error


2.  sugaredProceduralFactorial[n] :=
      Block[  {result = 1},
          Do[  result = result * i, {i, 1, n} ];
          result]
```

```
3.  loopFactorial[n] :=
      { For[ i=1 to n, i++, result := i*result ];
        result }


4.  guardedFactorial[n, result] :=
      Precondition:     Integer[n] and Positive[n]        /also end condition
      Invariant:        factorial[n] = n * factorial[n - 1]
      Body:             guardedFactorial[ (n - 1), (n * result) ]
      PostCondition:    result = Integer[result] and Positive[result]
                                 and (result >= n)


5.  assignmentFactorial[n] :=
      { product := 1;
        counter := 1;
        return assignmentFactorialCall[n, product, counter] }


6.  assignmentFactorialCall[n, product, counter] :=
      if[ (counter > n)
            then
                  return product
            else
                  { product := (counter * product);       /error if these are
                    counter := (counter + 1);             /in reverse order
                    return assignmentFactorialCall[n, product, counter] } ]


7.  recursiveFactorial[n] :=
      if[  n == 1, 1,  n*recursiveFactorial[n - 1] ]


8.  rulebasedFactorial[1] = 1;
    rulebasedFactorial[n] := n * rulebasedFactorial[n - 1]


9.  accumulatingFactorial[n, result] :=
      if[ (n = 0)
            then
                  return result
            else
                  return accumulatingFactorial[ (n - 1), (n * result) ]


10. upwardAccumulatingFactorial[product counter max] :=
      if[ (counter > max)
            then
                  return product
            else
                  return upwardAccumulatingFactorial[ (counter * product)
                                                      (counter + 1)
                                                      max ] ]
```

```
11. mathematicalFactorial[n] =
      Apply[ Times, Range[n] ]


12. generatorFactorial[n]
      Times[ i, Generator[i, 1, n] ]


13. combinatorFactorial :=
      Y f< n< COND (=0 n) 1 (* n (f (-1 n))) >>


14. sugaredCombinatorFactorial =
      S (CP COND =0 1) (S * (B FAC -1)))


15. integralFactorial[n] =  Gamma[ n + 1 ]  :=
      integral[ 0 to Infinity, (t^n * e^(1 - n)), dt ]


16. streamOfFactorials  =
    streamAttach[ 1 streamTimes[streamOfFactorials streamOfPositiveIntegers] ]
streamOfPositiveIntegers =
    streamAttach[ 1 streamBuild[ Add1 CurrentStreamValue ] ]


17. JamesCalculusFactorial[n] =
      Decode[Standardize[Do[Stack[Encode[i], acc] {i,1,n}]]]

      Stack[jf, acc] =
            Subst[jf UnitToken acc]
```

From Abelson and Sussman, *Structure and Interpretation of Computer Programs*

18. abstractMachineFactorial = <p385>

19. registerMachineFactorial = <p511>

20. compiledFactorial = <p596-7>

Good Books

Below is the best books (IMHO) in most of areas covered in this class.

Programming Languages

B.J. MacLennan (1999)
Principles of Programming Languages, Third Edition
Oxford. ISBN 0-19-511306-3

M.L. Scott (2000)
Programming Language Pragmatics
Morgan-Kauffman. ISBN 1-55860-442-1

R.W. Sebesta (1999)
Concepts of Programming Languages, Fourth Edition
Addison-Wesley. ISBN 0-201-38596-1

Functional Programming

B.J. MacLennan (1990)
Functional Programming; Practice and Theory
Addison-Wesley. ISBN 0-201-13744-5

R. Plasmeijer and M vanEekelen (1993)
Functional Programming and Parallel Graph Rewriting
Addison-Wesley. ISBN 0-201-41663-8

Programming Theory

N.D. Jones (1997)
Computability and Complexity from a Programming Perspective
MIT Press. ISBN 0-262-10064-9

Data Structures, Algorithms and Programming

H. Abelson and G.J. Sussman (1996)
Structure and Interpretation of Computer Programs, Second Edition
McGraw-Hill. ISBN 0-07-000484-6

Comprehensive Reference on Algorithms

T.H. Cormen, C.E. Leiserson, and R.L. Rivest (1990)
Introduction to Algorithms
MIT Press. ISBN 0-07-013143-0

Very High-level Programming

S. Wolfram (1996)
The Mathematica Book, Third Edition
Wolfram Media, Cambridge U. Press. ISBN 0-521-58889-8

Theory of Computation Complexity

J.E. Savage (1998)
Models of Computation
Addison-Wesley. ISBN 0-201-89539-0

Computer Architecture

J.L. Hennessy and D.A. Patterson (1996)
Computer Architecture: A Quantitative Approach, Second Edition
Morgan-Kaufmann. ISBN 1-55860-329-8

R.Y. Kain (1996)
Advanced Computer Architecture
Prentice-Hall. ISBN 0-13-007741-0

Compilers

S.S. Muchnick (1997)
Advanced Compiler Design and Implementation
Morgan-Kaufmann. ISBN 1-55860-320-4

Understanding Computing in Simple Language

R. P. Feynman (A.J.G. Hey and R.W. Allen, Eds) (1996)
Feynman Lectures on Computation
Addison-Wesley. ISBN 0-201-48991-0

Programming Style

D.E. Knuth (1992)
Literate Programming
CSLI. ISBN 0-9370-7380-6

Assignment II: Pseudocode Syntax

Hand in to instructor at beginning of class.

Pseudocode is a computing language which is designed to convey ideas, specifications, and algorithms. It eliminates as many implementation details as possible. In theory, a pseudocode program should be executable, given that the lower-level details are provided.

Design the syntax of a pseudocode programming language.

You will need to:

1. Select a small set of primitives for abstracting control, data, and names.
2. Decide upon a lexical form for your language primitives
3. Decide upon a syntax which structures how primitives are combined.
4. Use standard techniques to define acceptable lexical and syntactic forms.

Standard syntax specification techniques include task decomposition, regular languages, finite state acceptors, formal grammars, BNF and/or diagrammatic BNF.

Language primitives can be seen as addressing control, data, or naming. Control primitives are included in imperative languages to steer the course of program evaluation. Data primitives provide typing and abstraction. A language may provide a single data type, or several basic built-in types, or user-defined types. Naming primitives determine the binding of names to values, and the location of names and values in memory. Primitives that we have discussed in class include `loop`, `logic`, `let`, and `domain theories`. Others may include `sequence`, `order comparisons`, `subroutines`, `hierarchy`, and `i/o`.

Important:

1. The task is to think clearly and carefully about the meaning of the 19 *Principles of Programming Languages* (handed out in the first week). Check your design decisions for conformance to each of these principles.

2. You must *be explicit about the tasks* which your pseudocode is intended to address. Ask why each primitive and each structure is included in the design. What part of the task does each particular structure address?

3. Attempt to avoid structures which are intended to enhance implementation efficiency. Pseudocode is not intended to address implementation efficiency, rather it should maximize readability, comprehension, and absence of ambiguity.

4. The most difficult part of language design is minimizing interactions between primitives when they are combined.

Challenge: Implement a lexical scanner and syntactic parser for your language.

Language Evolution

c. 1950 machine pseudocodes

1956 **FORTRAN I** (no data types)
- compiled
- i/o formatting
- subroutines
- IF, DO (no nesting)

1958 **LISP**
- uniform data structure (lists)
- functional programming (a formal model of computation)

1960 **ALGOL 60**
- data typing
- block structure
- pass by value, pass by name
- recursion

1960 **COBOL**
- macros
- hierarchical data structures
- long names

1964 **BASIC**
- remote terminal access
- easy to use

1965 **PL/I** (ALGOL+FORTRAN+COBOL)
- general purpose
- concurrency
- runtime error handling
- pointers
- sub-array reference

1967 **SIMULA 67**
- co-routines
- classes, data abstraction

1968 **ALGOL 68**
- orthogonality (few constructs and combinations)
- user defined data types
- dynamic arrays

1971 **PASCAL**
- teaching language (simple, expressive)

1972 **C**
- rich operator set
- OS-based (Unix)

1972 **PROLOG** (inefficient, few applications)
- declarative
- formal model using logic

1980 **Smalltalk**
- pure object-orientation
- methods
- messages, send
- software development environment (windowing)

1985 **Ada**
- committee design (too large and complex)
- encapsulation
- exception handling
- generic procedures
- concurrent tasks and synchronization

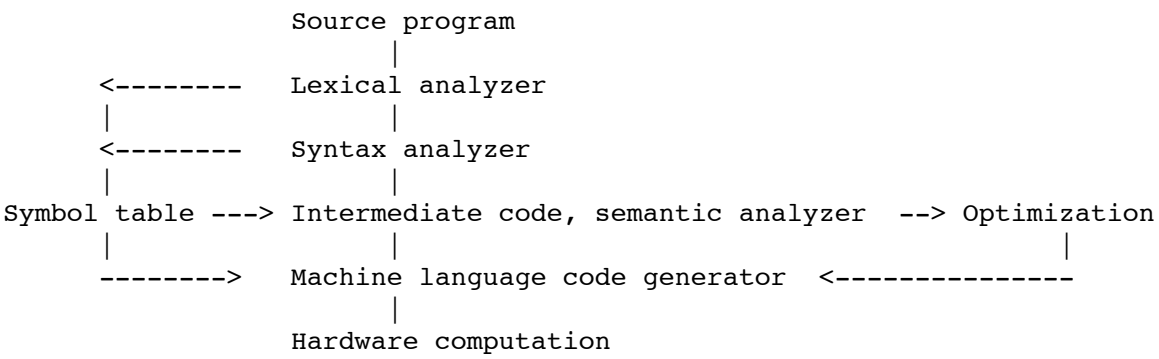
1985 **C++**
- predefined classes
- overloading
- templating (parameterized classes)

1988 **Mathematica**
- very high level mathematical language
- string rewrite engine, string uniform data structure
- all programming styles
- highly integrated development environment

1993 **Java**
- reference types (no pointers)
- Boolean control (no control arithmetic)
- pure methods, applets (no functions or subprograms)
- threads
- garbage collection
- limited coercions

Syntactic Modeling Tools

The Compilation Process



Lexical and Syntactic Analysis

Programs are *strings*.

Lexical analysis scans the program string for valid character sequences. *Syntactic analysis* parses the program string for valid word sequences.

Structural rules for both character and words strings are defined by a *context-free language*. Formal specification techniques include BNF, diagrammatic BNF, finite automata, production rules, and syntax graphs.

Formal Languages

alphabet	a finite set of symbols	`{i,j,k,+,*,0,1,2,...}`
string	a finite ordered list of symbols	
language	all possible strings using a given alphabet	
grammar	a subset of all possible strings constrained by a set of composition rules	

Classes of Languages

These categories form a hierarchy, in that regular languages are fully contained in context-free languages, etc.

Context sensitive languages have rules

`A --> B` such that `|A| =< |B|`

That is, the result `B` is never smaller that the input `B`

Context free languages have rules

`A --> B` such that `A` is a single non-terminal string

That is, the input A never branches.

Regular languages have rules

`A --> B` such that `B` is a terminal or a terminal and a non-terminal

That is, the output `B` either ends the rule, or triggers another terminating rule.

Context-free Languages

These three operations define a *regular language*:

JOIN	concatenate tokens and strings
OR	choose between alternatives for one location
LOOP	Kleene closure, Kleene star *

The Kleene star, `s*`, is a notation for repeating a given string zero or more times. The Kleene plus, `r+`, means repeat the given string one or more times.

RECURSION	handle nested structures

Expressed as production rules, recursion allows

`A --> B` such that `B` can contain reference to `A`

For termination, at least one rule associated with `A` must not contain recursive reference to `A`.

Regular languages are defined by placing a constraint on context-free languages, that of not permitting recursion. Recursion is necessary to construct nested and hierarchical forms; regular languages permit only construction of flat strings and lists.

Backus-Naur Form

One way to specify structural rules is using BNF, Backus-Naur Form. BNF is a collection of transformation, or pattern-matching, rules to apply to a given expression. BNF defines a regular language. Valid token strings can be specified by

Base cases:	empty string	`E`
	single character	`u`
Compound cases:	concatenation	`r*s`
	disjunction	`r|s`
	repetition	`r*`

Parentheses are used to disambiguate forms.

A diagrammatic version of regular language specifications uses

JOIN	`A --> B`	arrow from `A` to `B`
OR	`A -->` `  \-->`	branching paths
LOOP	`A -->` `\<--/`	Kleene star

Example: PASCAL numerical strings

```
digit       -->   0|1|2|3|4|5|6|7|8|9

integer     -->   digit digit*

number      -->   integer ((. integer)| E)
```

Language classes and grammars were developed by Noam Chomsky. Computing languages became connected to grammars because Backus' BNF formalism was equivalent to Chomsky's. Thus, a context-free language can be defined as a BNF form with recursion.

Examples of Grammars

Balanced Parentheses

alphabet

```
{(,)}
```

strings

```
{(,),S}
```

grammar1

```
S --> empty
S --> S S
S --> (S)
```

grammar2

```
S  --> S1*
S1 --> (S)
```

Star allows zero occurrences

Simple Algebra

alphabet

```
{,0,1,2,...,a,b,c,...,+,*,(,)}
```

strings

```
{+,*,(...), id, Term, Factor, Expression}
```

grammar

```
Expression --> Expression + Term | Term

Term --> Term * Factor | Factor

Factor --> (Expression) | id
```

Terms, Factors, and Expressions are defined formally by the rules of the grammar. Intuitively, an Expression is any valid linear algebraic form. A Term is two forms multiplied together. A Factor is an id or any Expression (separated by parentheses for grouping).

Simple Arithmetic Parsing Example

```
                    (3 * (4 + 5)) * (2 + 7)

(3 * (4 + 5)) * (2 + 7)              expression1
                                                --> term1
(3 * (4 + 5)) * (2 + 7)              term1
                                                --> term2 * factor1
(3 * (4 + 5))                        term2
                                                --> factor2
(3 * (4 + 5))                        factor2
                                                --> (expression2)
 3 * (4 + 5)                         expression2
                                                --> term3
 3 * (4 + 5)                         term3
                                                --> term4 * factor3
 3                                   term4
                                                --> factor4
 3                                   factor4
                                                --> id1
     (4 + 5)                         factor3
                                                --> (expression3)
      4 + 5                          expression3
                                                --> expression4 + term5
      4                              expression4
                                                --> term6
      4                              term6
                                                --> factor5
      4                              factor5
                                                --> id2
          5                          term5
                                                --> factor6
          5                          factor6
                                                --> id3
               (2 + 7)               factor1
                                                --> (expression5)
                2 + 7                expression5
                                                --> expression6 + term7
                2                    expression6
                                                --> term8
                2                    term8
                                                --> factor7
                2                    factor7
                                                --> id4
                    7                term7
                                                --> factor8
                    7                factor      --> id5
```

FORTRAN

FORTRAN was the first language with the design goal of *efficient performance*. Consequently, the constructs of the language are designed to accommodate a specific machine architecture. FORTRAN was also essentially a numerical processing language for scientific computation. The new features introduces by the language include:

Subprograms

- modularity
- communication through parameter binding
- procedural abstraction
- libraries

Two-part programs

- declarations
  - non-executable, compile-time directives
    - memory allotment
    - names for the memory spaces
    - initial contents of memory
- instructions
  - executable, runtime
    - computation through assignment (arithmetic and move ops)
    - control flow through IF and DO
    - input/output

Several processing stages (for efficiency)

- compile
  - relocatable object code (subprograms may move)
- link
  - thread libraries and external references
- load
  - absolute memory format
- execute
  - program in memory controls computer

Imperative programming

flow and control governed by programmable control logic

GOTO

- single low-level transfer of control
- confusing mental model
- static and dynamic models don't match

DO loop

initialization, iteration, and return all directly controlled within DO

Coercion

allow mixed operations

Limited arrays

optimize memory
use array index as memory address (rather than computing new addresses)

Implications of Subprograms

```
SUBROUTINE  <name>  <formal parameters>
```

Inefficient, naive invocation:

Substitute the subroutine for its name in the main calling body of code, and substitute the calling values for the formal parameters

Pass by Reference (FORTRAN's solution)

Substitute the *location* of the subroutine in memory
for its name in the main calling body of code

difficult to understand dynamic behavior
security risk when locations can be accessed

Pass by Value (preferred)

Substitute values for parameters in subroutine
Run subroutine in place
Return result to calling context

requires activation record to keep track of bookkeeping

Activation Records

parameter bindings	new values passed to the subroutine
resume address	place to return control when subroutine is done
dynamic link	location of caller's activation record, for returning results
temporary storage	for subroutine bookkeeping

Subprogram Invocation

To CALL

1. Place parameter binding values in callee's activation record.
2. Save caller state and resume address in caller's activation record.
3. Place pointer to caller's activation record in callee's activation record
4. Enter subprogram (callee's) first instruction

To RETURN

5. Transfer to callee's resume address.
6. When caller gains control, restore caller state.
7. When subroutine has return values (i.e. callee is a function),
   place return values in caller's activation record.

Name Structures (Environments, Symbol Tables)

Environments define *context* and *meaning*.

Sample name space:

name	*type*	*location*	*value*
IF	reserved	0247	<control>
i	integer	0248	3
res	list	0249	(a b c)
my-plus	function	0250	<body>

Each subprogram has its own name space for local variables.

Subprogram names must be global.

In FORTRAN, COMMON blocks declare shared data. This *aliasing* makes code maintenance confusing.

ALGOL-60 and ALGOL-68

ALGOL-60 was developed cooperatively between the US and Europe, with the goal of standardizing an international general-purpose programming language. Its primary design goal was *portability*. Since I/O devices were nowhere close to standization, ALGOL depended upon external libraries for device drivers. That is, ALGOL had no `read` or `print` statements.

The language definition is 15 pages long, exemplifying brevity and clarity through the use of BNF descriptions. ALGOL-60 introduced several central programming tools, in particular

Hierarchical structure evolved into *structured programming:*

Typed procedures (functions)

Stack as central runtime data structure
managed by block structuring; dynamic array sizing

Compound statements (regularity)
any valid operation can be written wherever variables can be written
block structure and nested scoping
implicit inheritance of accessible (within scope) variables

No magic values
name lengths and array size and dimension are not restricted

Strong typing
all forms are typed without ambiguity

Generalized control structures
`while`, `until`, `for`, recursion

Free Format
layout of the program is not specified by the language; reserved words are sacred

Implications of Block Structuring

Nested blocks introduced the issue of *variable scoping*.

Blocks enhance *modularity* and *maintenance* of large programs.

Blocks permit *efficient storage* management of stacks.

Symbol tables can be *factored* into smaller units for each block.
Symbol tables are accessible to the user, which compromises portability.

Blocks need to be *bracketed* to delineate start and end points. `Begin` and `end` are introduced as generic brackets. In ALGOL, `begin-end` brackets both group statements and delimit blocks. This undermines orthogonality.

Dynamic and Static Scoping

Static scoping: procedure definitions are called in the defining context at compile-time. Variable bindings can always be determined by reading the source code.

Dynamic scoping: procedure definitions are called in the calling environment at run-time. Variable bindings depend on dynamic context. *Example:*

```
begin integer m;          ;outer block
  procedure P;
    m := 1;
    begin integer m;
      m := 2;
      P                          ;first call to P
    end;
  P                              ;second call to P
end
```

In *static scoping*, the assignment `m:=1` refers to the variable `m` in the outer block. Static means that no matter when a particular procedure is called, the binding context of its variables does not change. Thus both calls to procedure `P` use `m=1`.

In *dynamic scoping*, the assignment to `m` depends upon the calling context. The first call to `P` is in the context of `m=2`, thus it uses that value. The second call to `P` is in the outer context where `m=1`. [Note that the value `m=2` is changed to `m=1` immediately upon entering `P`.]

Parameter Passing

Pass by value: the *actual value* of the variable being passed to a procedure is copied into the formal parameter of the procedure. Secure, but inefficient for arrays. For input variables only.

Pass by name: the *name* of the variable being passed to a procedure is copied into the formal parameter of the procedure. Powerful, but dangerous and expensive.

Each type of parameter passing differs in when it looks at the value of the variable being passed (the actual) into the procedure (the formal parameter):

Pass by value: When a procedure is called, the formal is bound to the value of the actual as a snapshot. Later changes in the actual will not be seen by the procedure. Early inspection time.

Pass by reference: When a procedure is called, the formal is bound to a reference to the actual. The reference cannot change (i.e. the location of the actual can not be changed), but the value it refers to can change.

Pass by name: When a procedure is called, the formal is bound to special address-returning function (a thunk). Although the thunk does not change, the address it returns and the value in the address location can vary. Very late inspection time.

Functional Programming

Imperative vs Applicative

Imperative languages rely heavily on assignment and on programmed changes in memory to accomplish a program's objectives. Control is basically routing processes from one assignment statement to another.

Applicative languages rely on function application, passing computational results from one function to another. Control is achieved through function nesting.

Functions are mathematical objects which have single entry and exit points. Functions return a value. A function itself *names* its return value. If the function is unevaluated, then the name is compound. When the function is evaluated, it returns a simpler name. Since functions are names, the text of a function, such as F[x], serves the same purpose as a variable, to name a particular part of a computation. More generally, function names can be compounded, so that two composed functions yield a compound name for the composition of the two functions.

The hallmark of a functional program is that it is composed of function invocations and conditionals only. Functional programs contain no variables, no loops, no explicit sequencing, and no assignments.

Functions do not permit side effects. Therefore whenever a function is evaluated with the same arguments, it produces the same results. This is called ***referential transparency***; a type of what-you-see-is-what-you-get. Absence of side effects makes the meaning, or semantics, of a functional program fairly simple. Another advantage is a simple syntax which is easy to parse and error-check. Thus both syntax and semantics of functional programs is cleaner that imperative programs.

Imperative languages are architecture specific in that they evolved for programming a vonNeumann processor. Applicative languages suggest a functional architecture. Many such machines have been built in university settings, however a diversity of hardware architectures has yet to reach personal computing.

Uniformity and Simplicity

Pure LISP is unique in that it has only one data structure (the ***list***), and two control structures (***function-invocation as recursion*** and ***if-then-else***). This smallness and uniformity makes the language design and implementation very elegant and efficient.

The most significant implication of uniformity is that data and program have the same structure (both are lists). Thus LISP programming emphasizes *metaprogramming*: writing programs which return other programs as output. LISP encourages *extreme abstraction*. For example, rather than writing a parser for a particular language, LISP style would be to write a parser generator which takes a language as input and returns a parsing program for that language as output.

Another level of uniformity in functional languages is that during processing, all functions have only one argument. The process of converting binary (and in general n-ary) functions to unary functions is called *currying*. Schematically:

```
F[a,b] => G[a][b]
```

The function `G[a]` is a functional, returning a function `G'`. `G'` then applies to the single argument `b`.

Another simpler way to achieve unary functions is to collect all arguments into a list and pass the list as a single argument. Schematically:

```
F[a,b] => G[(a,b)]
```

Recursion vs Iteration

Almost all mathematical structures are defined and proved using induction. Recursion implements induction. Thus recursive style has these benefits:

- aligns with the mathematical approach
- easier for most data structures in most cases
- elegant and difficult to learn

Here is an example function which is straight-forward when written recursively and quite difficult when written iteratively. It tests the equality of two trees.

```
equal[x,y] =def=
      (atom[x] and atom[y] and x=y) or
        (not[atom[x]] and not[atom[y]] and
          equal[first[x],first[y]] and equal[rest[x],rest[y]])
```

The iterative version requires parallel iteration of two variables, one for tree `x` and one for tree `y`. The difficulty for the iterative version occurs when `x` and `y` are not atoms. How does a loop manage to deconstruct the left and right branches of the tree without using recursion?

This example illustrates that although iteration and recursion are theoretically equivalent, from a programming point of view, recursion is both more powerful and more elegant than iteration.

Imperative and applicative styles can be contrasted by comparing iteration with recursion.

Iterative: `for I in 1..X do <accumulate results in A>`

The above imperative loop contains three variables, each with a significantly different purpose.

`I`: the *iteration* variable, declared and scoped by the loop itself

`A`: the *accumulation* variable, again scoped within the loop, used to return values

`X`: the *input* variable, used as a parameter of the function containing the loop

Functional programs minimize the use of variables, since functional programs do not rely on memory manipulation for storing computational results.

Applicative: `if X=0 then nil else (A + <recur on incremented X>)`

In applicative, or functional, programming, the loop construct is replaced by recursion. Each recursive call modifies the input parameter until that parameter reaches a ground value.

Iteration variables are eliminated in favor of structured recursive descent

Accumulation variables are eliminated in favor of accumulating the results of each recursive function call as they return values to the containing function.

Input variables directly incorporate iterations and accumulations. The input variable is incremented to achieve iteration. The output of a recursive function incorporates the accumulated results.

Pure Functions and Lambda

Pure functions, or **operators**, are functions which do not have any arguments. Rather operators apply to their context, to whatever forms are juxtaposed to the operator. Operators have a space for arguments, but the arguments are not constrained to any particular type, to a reduced expression, or even to data.

Lambda calculus is the mathematical system upon which functional programming is based. It has two defined operations, *apply* and *abstract*. A lambda expression defines an operator. For example, the *square* operator:

```
square =def= lambda[#,#*#]
```

The hashmark # stands in place of a variable, but it is not a variable per se. Rather it represents a *space* or a *hole*, a place where any form can be placed. The second form, *#*#*, is not an argument, it is rather the body of the lambda expression. A hole stands in place of the **context** of the expression.

The *Rule of Application* is simply that any form following the lambda expression is substituted in place of the hashmark. Thus, lambda expressions can take functions or data structures as "arguments". In the above example:

```
lambda[#,#*#] 4 ==> 4*4 ==> 16

lambda[#,#*#] F[x+1] ==> F[x+1]*F[x+1]
```

In the first example, the `4` following the square-lambda replaces the hashmark, leaving a body `4*4` to be evaluated. In the second example, the hashmark is replaced by a function call. In general:

```
Apply[E1,E2] =def= <substitute E2 for # in E1>
```

The *Rule of Abstraction* is the inverse of apply; it provides a mechanism for removing the dependence on variables by functional expressions. In general:

```
Abstract[F[x]] =def= lambda[#,F]
```

Mapping and Functionals

Consider squaring each member of a list of integers. Here are three programs representing three different styles:

Iterative:
```
for each element in list do
  <square element; store into accumulator>
```

Recursive:
```
if list=nil then nil else
  <add square[first[list]] to recur[rest[list]]
```

Mappable:
```
map <square> onto list
```

A data structure is *mappable* when a function applied to the entire list gives the same result as applying the function directly to each element in the list.

The **map** function takes two arguments, a function and a list. The function argument applies to a single item, or element on the list. The map function takes care of the mechanics of iteration. When a function takes another function as an argument, that function is called a *functional*, a function which acts on other functions. Many languages do not allow functionals.

Filtering is a type of mapping in which the applied function is a Boolean test for membership in the returned list. The filter function is a functional:

```
filter[test-fn, list] =def= map[test-fn, list]

test-fn[i] =def= if <i meets criteria> then i else <nothing>
```

Functionals can also return functions as output. Here is an example which selects the appropriate operator for a data type:

```
pick-op[i] =def=
  case type[i]
    integer:      +               ;add
    list:         cons            ;push onto list
    string        ·               ;prefix
```

Another example is the functional `apply`: to apply a binary function to a list of arguments the arguments are taken two at a time, combined using the function, and then the result is returned to the list. When all arguments are processed, a single value remains. *Example*:

```
apply[+,(1,3,5)] ==> apply[+,(4,5)] ==> 9
```

Mapping and similar functionals treat entire *data structures as single objects*. This is an example of the extreme abstraction of functional languages. Functionals provide the right level of abstraction for any compound data structure.

In all processes which address a program rather than a data structure (within a program) require function-level analysis: the "objects" being transformed are functions rather than data. Examples of program manipulation include proof of correctness, optimization and compilation, program derivation, and metaprogramming.

Combinators

Combinators can be seen as macros for lambda calculus. More accurately, they are a set of functions which provide the same functionality as lambda calculus without the lambda construct. Combinators encapsulate all control structures. Another perspective is that combinators are a pattern-matching language for functions.

The syntax of a lambda calculus expression is a sequence of forms. For example, factorial expressed as a pure function (@ is used as another blank notation, like #):

```
fac =def= Y lambda[#, lambda[@, if @=0 then 1 else @*#[@-1]]]
```

The `Y` combinator is the way lambda calculus handles recursion. In effect, it rewrites the entire definition whenever it is called recursively.

There are two elementary combinators: `K` and `S`. `K` is True, and `S` is Sequence. These combinators replace lambda expressions using the following set of recursively defined rules:

```
lambda[#,E] ==> K E                    when E = constant, variable, or combinator

lambda[#,var] ==> S K K                when var=#

lambda[#,E1 E2] ==> S lambda[@,E1] lambda[%,E2]
```

A set of pattern matching rules reduces combinator expressions. For example,

```
K E1 E2 ==> E1
```

This can be read as a conditional:

```
if K then E1 else E2                   where K = True.
```

A Small Interpreted Language

What would you need to build a small computing language based on mathematical principles? The language should be simple, Turing equivalent (i.e.: it can compute anything that any other language can compute) and relatively easy to use. Assume the computing hardware is constrained to vonNeumann processes, with memory, an ALU, and appropriate registers. We will also assume that we know about formal mathematical languages and the necessary mathematical pieces: representation, recognizer, constructor, accessor, invariants/facts, functions, and induction/recursion.

Base *Representation* of Atoms

First, the *alphabet* of a language is simply a collection of unique identifiers, called *atoms*. The essential memory management trick is to divide each memory cell into two parts, an address part (call it **First**) and a contents part (call it **Rest**). Addresses are also called *pointers*. We begin with an array of empty cells, each having some empty representation in both the **First** and the **Rest** parts. This is the *free list* of memory cells.

The ground: We need an atom which means nothing, the null atom. Call it **nil**.

The symbol table: This table consists of a collection of non-empty memory cells, one cell for each atom in the language. The **First** part of an atom cell contains nil. The actual literal representation of the atom is in the **Rest** of the cell. The symbol table is a dynamic array.

Constructor of Compound Expressions

We need to construct compound expressions. Consider an expression which uses two atoms, say FOO BAR. The symbol table contains each atom, so all we need is a way to connect them. This can be done simply by building another memory cell which contains the two addresses of FOO and BAR. We put all atom addresses in the **First** part of a cell (see cell 005 below) and connecting addresses in the **Rest** part. The instruction to build connecting cells is called **Cons**. The end of an expression has **nil** in the **Rest**.

If we build the expression (TRUE BAR TRUE FOO) in cell 007, memory would look like this:

Address	First	Rest	
000	nil	nil	symbol table
001	nil	FOO	
002	nil	BAR	
003	nil	BAZ	
004	nil	TRUE	end of symbol table
005	001	006	the expression (FOO BAZ)
006	003	000	end of expression
007	004	008	the expression (TRUE BAR TRUE FOO)
008	002	009	
009	004	010	
010	001	000	end of expression
011	empty	empty	begin free list
...			

To construct an expression, we **Cons** smaller pieces together. For instance:

```
Cons JOHN (TRUE BAR TRUE FOO) ==> (JOHN TRUE BAR TRUE FOO)
```

The operational memory changes are:

```
011         nil         JOHN        the atom JOHN
012         011         007         connect JOHN to (TRUE BAR TRUE FOO)
013         empty       empty
```

Consider **Cons**ing two compound expressions together:

```
Cons (FOO BAZ) (TRUE BAR TRUE FOO) ==> (FOO BAZ TRUE BAR TRUE FOO)
```

This operation is slightly more complex. For the entire expression to begin in cell 012, we need memory to end up as

```
011         003         007               (BAZ TRUE BAR TRUE FOO)
012         001         011               (FOO BAZ TRUE BAR TRUE FOO)
013         empty       empty
```

Several design decisions are involved with this result. Technically, we have used *structure sharing* for (TRUE BAR TRUE FOO) since both the original four atom expression and the final six atom expression use some of the same memory cells. However, the front of the expression, (FOO BAZ) is not engaged in structure sharing, and this may seem a little unsymmetrical. As it is, (TRUE BAR TRUE FOO) is confounded with **Rest** (**Rest** (FOO BAZ TRUE BAR TRUE FOO)).

An alternative which would allow us to continue to refer to the original would be to duplicate the four atom expression entirely in constructing the six atom expression.

Note also that the construction is slightly different, rather than adding a symbol cell, as in the case of JOHN, we have added a *cons cell*. To acknowledge these differences, we might consider **Cons** of two compound expressions to be a different operation. Call it **Append**. Now the first object in a **Cons** operation is restricted to be an atom. **Append** is used when the first object is compound. To keep the language simple, we would want to be able to build new operations out of the existing ones. For this, we use a recursive definition:

```
Append <obj1> <obj2> =def=
  If Isa-atom <obj1>
    then ERROR
    else if Is-empty <obj1>
      then <obj2>
      else Cons (First <obj1>)(Append (Rest <obj1>) <obj2>)
```

This recursive definition first does a *type-check* on <obj1>. It then tests the *base case*, that <obj1> is **nil**. **Append**ing nothing onto <obj2> results in <obj2>. Otherwise we proceed one piece at a time. The recursion bottoms-out when **Rest** <obj1> is **nil**. For this to be the case, <obj1> must have only one atom, as in (BAZ), which is **Cons**ed onto <obj2>. At that time, BAZ is the **First** of <obj1>. Just prior to this case, <obj2> is actually (BAZ TRUE BAR TRUE FOO), since we have **Cons**ed BAZ to (TRUE BAR TRUE FOO). <Obj1> is (FOO BAZ), and we are about to **Cons First** <obj1>, i.e. FOO, onto (BAZ TRUE BAR TRUE FOO).

This description has backed up from the end to the beginning. Tracing the events in memory:

```
Append nil (TRUE BAR TRUE FOO) ==> (TRUE BAR TRUE FOO)

011          000          007          Append nil
012          empty        empty        begin free list
```

By definition, cell 011 is the same as 007, so operationally this step is not necessary to take. We leave 011 free, treating **Append**ing **nil** as a *no-op*.

```
Cons BAZ (TRUE BAR TRUE FOO) ==> (BAZ TRUE BAR TRUE FOO)

011          003          007

Cons FOO (BAZ TRUE BAR TRUE FOO) ==> (FOO BAZ TRUE BAR TRUE FOO)

012          001          011
013          empty        empty
```

What we have done here is to specify exactly the sequence of operations on memory that result in the action of **Append**ing. And we have used the single construction tool of **Cons**.

This example illustrates the close connection between a software program, the attendant changes in memory, and the hardware architecture which unites both.

Recognizer of Atoms

The *recognizer* of each atom is a function which looks in the symbol table for the memory cell which contains that atom. For instance, the predicate **Isa-atom** is true if its argument can be found in the **Rest** portion of the symbol table. At this point, we have three separate memory areas (or uses): *free cells*, *atom cells*, and *cons cells*.

Isa-atom: Atom cells are recognized by having **nil** in the **First** part.
Is-empty: Empty expressions can be uniquely recognized because they have **nil** in the **Rest** part.
Equal: Tests if two atoms are the same atom.
Isa-expression: **Cons** cells are recognized as those cells having two addresses. An expression ends with **nil** in the **Rest** part.

The above are close to operational definitions. Here are some slightly more elaborated operational definitions. We will assume that each part of a memory cell (address, first, rest) has eight bits.

Is-empty <obj>:

Assign **nil** a special binary code, 00000000, and put it in address 00000000.
An object is empty, that is, it is equal to **nil**, if the **Rest** part is equal to the code of **nil**.
To distinguish **nil** from an empty cell on the free list, we could put a special code in free list cells, perhaps 11111111. A better approach is to use only seven bits of the address for address information, and use the eighth bit for marking if a cell is free. This is the basis for many *garbage collection* algorithms.

```
Is-empty <obj> =def= Equal (Rest <obj>) 00000000
```

Isa-atom <obj>:

Test the encoding of <obj> against all the encodings in the **Rest** part of memory which also have **nil** in the **First** part.

```
Isa-atom <obj> =def= for some memory cell
  (Equal (First <obj>) nil) and (Equal (Rest <obj>) <obj>)
```

Here is another *design choice*: is **nil** an atom or not? If it is not an atom, we will have to have special tests for atoms vs **nil**. For simplicity, let's say it is an atom:

```
(Isa-atom nil) is True
```

This design choice is our first *fact*, or invariant.

More generally:

```
Isa-atom (Is-empty <obj>) =def=
      True iff (Is-empty <obj>) is True
```

Recognizer of Expressions

We can use the instructions **First** and **Rest** to access and decompose all expressions. (**First** <obj>) looks at the first part of memory for the specific object, (**Rest** <obj>) looks at the rest part.

To recognize compound expressions, we test to see if each part of that expression is in the memory table, and the linking structure of the expression matches the rules for constructing that expression. Operationally:

```
Isa-expression <obj> =def=
  (Isa-expression (First <obj>)) and
  (Isa-expression (Rest <obj>))
```

Since we know that decomposing an expression will end in either atoms or nil, we will have to add those rules:

```
Isa-expression (Isa-atom <obj>) =def=
  True iff (Isa-atom <obj>) is True

(Isa-expression nil) is True.
```

This is another application of recursive decomposition. The rules specify the base cases, while the definition specifies the general recursive case. The two together specify a program.

The definition above is another example of *pseudo-code*, that is, machine specific instructions written in a mathematical style that is independent of the specifics of any programming language, yet specific enough to be implemented in any language. Of course, a *high level programming language* accepts something very close to pseudo-code specification as valid input. Another strong advantage of pseudo-code is that it can be proven to be correct using the Induction Principle.

The primary reasons that current programming languages appear to be very different than pseudo-code are

1. Many programming tasks lack a formal model (i.e. they are hacks).

2. Many programming languages lack mathematical structure (i.e. they are machine architecture specific.)

Accessors of Atoms and Expressions

First and **Rest** are the accessors. They let us take apart an expression. In this implementation, **First** and **Rest** have simple mappings onto the idealized physical structure of memory.

All objects except **nil** are constructed by **Cons**. Since **Cons** uses two objects as arguments, this means that all **First** and **Rest** parts are also objects. Eventually all objects end in **nil**, so **nil** is also an object, although a very special kind.

Cons is related to **First** and **Rest** by the following invariant, or rule:

```
<obj> = Cons (First <obj>) (Rest <obj>)
```

This says that all valid objects have been constructed by **Cons** to have a **First** part and a **Rest** part in memory. Alternatively, all objects in memory can be accessed through their **First** and **Rest** parts. The essential mathematical condition is that all valid objects are decomposable into unique subcomponents which bottom-out at the base cases. This is simply to say that all compound expressions are defined recursively.

Although recursive composition and decomposition are necessary to define data structures and algorithms, the more important aspect of recursive definition is to provide access to proof through the Induction Principle. Procedural languages do not provide this capability; they are thus immature. Declarative, functional, and mathematical programming languages all provide the capability of abstract proof (minimally in pseudo-code).

Note that recognizing, constructing, and accessing an expression involve almost the same steps. The difference is in the initial goal and the final result.

	GOAL	PROCESS	RESULT
Constructor:	build a pattern	rearrange memory	the pattern is in memory
Recognizer:	test a pattern	access memory	true if the pattern is accessible
Accessor:	get a pattern	access memory	return the pattern if found

SUMMARY of the ABSTRACT DATA STRUCTURE FUNCTIONS

First <obj> returns the expression indicated by the **First** of the <obj>

Rest <obj> returns the expression indicated by the **Rest** of the <obj>

Is-empty <obj> returns True if the cell containing <obj> has **nil** in **Rest**.

Isa-atom <obj> returns True if the <obj> is in the **Rest** part of a cell and **nil** is in the **First** part.

Isa-expression <obj> returns True if the <obj> has either **nil** or any address in the **First** part.

Equal <obj1> <obj2> In the case of atoms, returns True if both objects are in the **Rest** of the same symbol cell. In the case of compound expressions, returns True if following the addresses in the **First** leads to the same set of **Rest** symbol cells.

Cons <obj1> <obj2> builds an expression by adding <obj1> to the front of <obj2>

Invariants

The *equality invariant* (also called the Uniqueness Axiom) assures that each object is unambiguous. That is, objects are the same object when they are equal; equal objects are constructed and deconstructed in exactly the same way. This is a physical kind of equality, *structural equality*, in that the structure of memory is the same for two objects. It is not necessary that the same memory cells are used for both objects (structure-sharing), just that the contents of memory for both objects are the same. Recursively,

```
Equal <obj1> <obj2> =def=
  (Equal (First <obj1>) (First <obj2>)) and
  (Equal (Rest <obj1>) (Rest <obj2>))
```

We need to support this definition with base cases. For instance,

```
(Equal nil nil) is True
```

This is also an example of the *Induction Principle* at work in our implementation. To implement an equality test for expressions, the computation will test for identical structure over all memory cells of both objects. The Induction Principle is the only guarantee that this recursive process will end. The only end point is (**Equal nil nil**), all other cases are failures.

Note that equality for atoms is also covered in the above definition. What happens, though, when we have two atoms which have the same encodings, but each is in a different memory cell? This is an inconvenience for an implementation, since testing each object would require looking through the entire symbol table. A better approach is to insist that each atom is unique and occurs only once in the symbol table. This is why Equality and Uniqueness are the same ideas.

The uniqueness of atoms is implemented by having each new atom *register* itself in the symbol table. In the background, when an unrecognized, new atom is entered, the implementation verifies that it is new, and then puts it in the symbol table. To do this is to *intern* the atom. If the atom already exists, then the address of the cell which contains that object is associated with the new input.

A different kind of equality refers to equality under transformation. The actual expressions may be different, but transformation rules allow us to say that the meaning of the different expressions is the same. This is *semantic equality*, also called *algebraic equality* and *mathematical equality*. Only defined transformations are allowed; all transformations (with the exception of **Cons**) are required to keep meaning consistent. It takes a special *symbolic architecture* to implement mathematical equality, mainly because transformations refer to sets or classes of objects rather than to specific objects. In the above, we have designed a *literal architecture*, as yet it has no capacity for dealing with sets of objects.

Now on to the functional part of the language. We will elect to use *lambda calculus* as the mathematical model.

Functions and *Recursions*

A function is an expression with the function name first and then the arguments. (The order of operators and arguments is somewhat arbitrary, just so long as it is consistent and unambiguous.) For example:

```
+ 3 4
```

The Arithmetic Logic Unit (ALU) can process logical and arithmetic operators when applied to atoms. Internally, both arithmetic and logic are encoded by binary sequences, so it is the responsibility of the operator, or of a *type test*, to make sure that expressions meant to represent numbers are channeled to the arithmetic units and expressions intended to represent logic are channeled to the logic units.

One way to implement the difference between logic and arithmetic is to assign another single bit in the memory cell that records the type of object in that cell. Note that silicon gates process only logic. Thus arithmetic objects must be encoded into a logical form for processing. In computation, *logic is fundamental*, arithmetic is derivative.

All logic functions can be defined in terms of a single function, so we need only one primitive logic function. Let's use **IfThenElse** (**Nand** and **Nor** are alternatives).

```
IfThenElse <obj1> <obj2> <obj3>
```

IfThenElse will evaluate <obj1> and then either evaluate <obj2> (if <obj1> is True) or evaluate <obj3> (if <obj1> is False). Here we have another function which uses different types of objects (the first example was **Cons**). In particular, <obj1> must be a logical type, returning either True or False.

Function composition permits complex sequences of operations. A function expression can be put in any place that an atom can be put, since all functions will reduce to single atoms. To separate composed functions, we can use parentheses to contain each function expression. We will choose to evaluate all inner arguments first, then use these results to evaluate outer

functions. Lambda calculus permits another order of evaluation, outermost first. This choice is a design decision, and is based on mathematical characteristics of each form of evaluation.

For example, an innermost evaluation:

```
(* (+ 3 4) (+ 1 2)) or ((3 + 4) * (1 + 2))
```

means that expressions with atoms as arguments are evaluated, or *reduced*, first.

The memory for this object would look like this:

Address	First	Rest	
000	nil	nil	symbol table
001	nil	1	
002	nil	2	
003	nil	3	
004	nil	4	
005	nil	+	
006	nil	*	end of symbol table
007	006	008	expression ((3 + 4) * (1 + 2))
008	005	009	
009	003	010	
010	004	011	
011	005	012	
012	001	013	
013	002	000	end of expression
014	empty	empty	begin free list
...			

There are several things to note about the above memory configuration.

Operators and numbers are not distinguished in memory, they are distinguished by what happens when they are handed to the ALU.

Each operator has two arguments, but we have no way to have two references in one memory cell. The solution is to order the expression so that operators are followed by their arguments. When an operator is fetched for evaluation, the machine code recognizes that that operator requires two more fetches. Should a fetch return another operator, then the first operator waits until the second operator converts its two arguments into one result.

Fetches occur by following the addresses in sequence. This is efficient since the address register (the register which keeps track of what to fetch next) need only be decremented by one to find the next memory cell.

It is possible to turn all functions into one argument functions (the technique is called *currying*). This is effectively what has happened by storing the expression in the *operator first* form (also known as reverse Polish notation).

Finally, consider how close the syntax of many programming languages is to what actually happens at the *register transfer level* of the computer. We are still at the very early stages of development of computing languages, since the syntax reflects low level data shuffling rather than high level task requirements. Progress means moving our profession toward human capabilities, and moving away from low level machine details.

We need a way to define arbitrary functions and a way to bind the variables of functions to values for the ALU to process. For example

Square <obj> =def= (* <obj> <obj>)

so that **Square** 4 => (* 4 4) => 16

First consider variables, names which stand for any valid object. We have been using the names <obj1>, <obj2>, etc. as variables names. The angle brackets notate that the name in question is not the name of a single thing, but rather it is the name of a class, or *set*, of things, all of which are of a particular *type*.

Variables (or *parameters*, when the names are arguments of a function) are atoms also, so they are simply added to the symbol table. To assign a value to a variable symbol, we can put a reference to the location of the value we wish to associate with the variable in the **First** part of the memory cell for the variable. Thus variables are distinguished from objects representing a specific value because their **First** part is not **nil**. It is an error to access a *variable* which has **nil** as the **First** part. Objects which do have **nil** in the **First** part are called *ground objects*.

The function which assigns ground objects to variable objects is called **Let**.

We can use this same mechanism to store the definitions of functions. The memory cell which contains the name of the function in the **Rest** part can contain the address of the definition of the function in its **First** part. Consider the memory configuration for the above definition of Square:

Address	First	Rest	
000	nil	nil	symbol table
010	nil	OBJECT	
011	nil	*	end of symbol table
012	013	SQUARE	function definition
013	011	014	
014	010	015	
015	010	000	end of function definition

When the call **Square** 4 is added to memory we get:

016	nil	4	symbol table
017	012	018	function call (Square 4)
018	016	000	

To bind OBJECT to the value 4, we use the call **Let** object 4:

019	nil	LET	symbol table
020	019	021	function call (Let object 4)
021	010	022	
022	016	000	

Finally we need to get the processor to actually evaluate the function call. Let's call this **Eval**. We can actually make **Eval** the default. Whenever a new expression is added it can be automatically evaluated. This just shifts the issue to needing an instruction to stop evaluation. Let's call this evaluation stopper **Quote**.

What the above memory configuration contains is **Quote** (**Square** 4), which simply puts the *data structure* **Square** 4 into memory. If we write the *function* **Square** 4, then evaluation will happen automatically. This process consists of changing the value of object from **nil** to 4, and following the sequence until a single atom is returned. That is, the function **Let** says to the processor: go to the symbol which immediately follows **Let** and put the address of the second symbol which follows **Let** (i.e. 4) in its **First** part. This results in

```
010          016          OBJECT
```

Now the definition of **Square** will find the value of OBJECT and use it rather than using the symbolic variable "OBJECT". And, of course, symbolic variables are the only items in the symbol table which can contain something other than **nil**.

There is a slight problem here because the symbol "OBJECT" might be used in more than one function. This can be handled in one of two ways:

1) make sure all of the symbols are unique, or

2) divide the symbol table into subtables which associate and isolate each function with its own variables.

Finally, we simply use *recursion* directly as repeated actions of the same sort, since nothing in the above structuring stops this from working.

The *Function* Eval

In the above description, evaluation is an implicit action of the ALU. By claiming evaluation is automatic, we are committed to wiring the ALU in a specific way. However the above mechanism for handling memory can be made flexible by *defining **Eval** in the programming language itself*. This process is called *meta-circular evaluation,* cause it uses a language itself to define how that language should behave. All we have to do is to define the evaluation function by telling the system what to do when an expression is typed in. The function **Eval** takes two arguments, the expression to be evaluated and the *binding environment*, that is, an address of the memory array which contains all of the primitive functions and atoms (and any other symbols which we may have added) in the language. The binding environment contains the definitions of all user defined functions, and the values of each of the variables (function arguments).

Since the binding environment does not change in this example, (i.e. we have not designed the language to establish separate environments for each function call), we will treat the token **Eval** to mean "Eval-in-environment"

The definition of **Eval** which follows uses only primitive functions introduced above. Some of the syntax has been changed to make it more readable.

This **Eval** function recognizes seven operators:

First	Rest	Cons	
IfThenElse	Equal	Quote	Let

In addition, **Eval** uses built-in tests to determine the types of objects, as operationalized above.

Is-empty	Isa-atom	Isa-expression

```
Eval exp  =def=

If Isa-atom exp
 Then                                                    ;process atom
   If ((Is-empty exp) or (Equal exp (Quote True)))
    Then                                                 ;return the SYMBOL
      exp
    Else                                                 ; or its VALUE
      Get-value-in-env exp
 Else                                                    ;process expression
   If Isa-atom (First exp)
    Then                                                 ;process Atom in First*
      Let token (First exp)                              ;naming the atom
        If Equal token (Quote Quote)
         Then                                            ;return what follows
           Second exp
         Else                                              ;other operators
           If Equal token (Quote IfThenElse)
            Then                                         ;process logic operator
              EvalLogic (Rest exp)
            Else                                           ;other operators
              If Equal token (Quote First)
               Then                                      ;First of Eval of Rest
                 First (Eval (Second exp))
               Else                                        ;other operators
                 If Equal token (Quote Rest)
                  Then                                   ;Rest of Eval of Rest
                    Rest (Eval (Second exp))
                  Else                                     ;other operators
                    If Equal token (Quote Isa-atom)
                     Then                                ;Isa-atom Eval of Rest
                       Isa-atom (Eval (Second exp))
                     Else                                  ;other operators
                       If Equal token (Quote Cons)
                        Then                             ;Cons Eval of Rest**
                          Cons (Eval (Second exp))
                               (Eval (Third exp))
                        Else                               ;other operators
                          If Equal token (Quote Equal)
                           Then                          ;Equal Eval of args
                             Equal (Eval (Second exp))
                                   (Eval (Third exp))
                           Else                          ;replace the token with
                             Eval (Cons                  ;its value
                                    (Get-value-in-env token) (Rest exp))
    Else                                                   ;compound First
      If Isa-expression (First exp)
       Then                                              ;process expression
         EvalExp exp
       Else ERROR
```

```
EvalLogic exp  =def=

If Equal (Eval (First exp)) (Quote True)          ;if First is TRUE
 Then                                             ;Eval second argument
   Eval (Second exp)
 Else                                             ;Eval third argument
   Eval (Third exp))


EvalExp exp   =def=

If Is-empty exp                                   ;if at the end
 Then                                             ;return ground
   nil
 Else                                             ;Eval the parts
  Cons (Eval (First exp)) (Eval (Rest exp))       ; and put them together
```

Notes:

* process Atom in First: Here we have defined a syntax for parsing. Every expression begins with an atom or is an atom. If an expression begins with an atom, that atom is taken by the processor to be an operator, and thus a processing instruction. The operator **Quote** is the no-op.

** Cons Eval of Rest: This is again a syntax constraint. Once we have removed the beginning operator of an expression, what follows is either an atom, or another expression which itself begins with an atom operator.

The syntax of the language is thus:

```
Expression ::= Atom | (Atom Expression*)
```

The Kleene star means that an operator atom can have any number of following arguments. Note that this BNF specification is one of a *regular language*.

Finally, note that a meta-circular language can evaluate its own definition of **Eval**, since the above definition is self-consistent.

The Punch Line

The above programming language actually exists, it is one of the very few oldest programming languages still in active use. Its name is *LISP*.

In 1955, John McCarthy followed a similar line of reasoning in developing LISP. Currently LISP stands uniquely among programming languages in that it is rigorous, efficient, largely machine independent, and permits simulation of all other programming language models (such as procedural, functional, object-oriented, logical, and mathematical). As well, when the function **Eval** is processing input, LISP is *interpreted*, responding dynamically to new inputs and definitions, and requiring no compilation or linking. It thus provides a powerful interactive programming environment which supports real-time debugging and symbolic proof of correctness.

Pure LISP

LISP is a unique language in the following ways:

- symbolic rather than numeric computation.
- functional/applicative style.
- indefinitely extensible.
- interpreted/interactive rather than compiled. LISP can be compiled after debugging.
- uniform data representation. Programs are data, which means LISP programs can modify themselves at run-time.
- LISP is written in LISP. This *bootstrapping* means that the LISP evaluation mechanism and compiler are easily available to the programmer for modification and customization.

Pure LISP excludes most of the programming ideas which lead to poor code. Most programming language innovations (such as garbage collection, streams, closures and continuations, symbol packages, first-class errors, object orientation, provability) were pioneered in LISP. Pure LISP does not allow:

- destructive data operations
- gotos
- explicit pointers and dereferencing
- side effects (only the direct results of the function being processed)
- unbound and global variables
- do loops (use recursion instead, this rule is not firm)
- block structure (functions provide grouping)

Primitives

LISP has a very small kernel of primitive functions. These are:

nil	empty list, false, nothing to return
atom	predicate to determine valid labels
eq	predicate to test equality of atoms
car, cdr	selectors/accessors of list data structure
cons	constructor for list data structures
cond	basic logic function
eval, quote	controlling the difference between program and data

Special functions have a non-standard format. Some important special functions include:

setq	setting or assigning labels to the results of functions
list	constructing a list

defun	defining named functions
lambda	constructing an unnamed function
let	defining the scope of variables

LISP debugging tools include:

<whitespace>	ignored by the evaluator
trace	follow the evaluation sequence
pprint	print data in pretty form
read-eval-print	the basic evaluation process

Disadvantages of LISP (and their solutions)

- *hard to read syntax with lots of parentheses*
  redefine the syntax to look the way you want it to

- *one data type*
  build the data types you want and wrap them in an abstraction barrier

- *inefficiency*
  no longer true, LISP runs at 95% the speed of C. It is possible to write inefficient LISP programs, but the rules to avoid this are straight forward and can be learned with practice. It is easier to write inefficient programs in other languages.

- *many dialects*
  the community has standardized on Common LISP. Dialects built from the same foundation are a good idea.

- *no first class functions*
  dialects for higher order programming are available (i.e. Scheme)

Storage Reclamation

Most programming languages use *explicit erasure* to reclaim storage cells. This is a bad idea since it makes a low-level maintenance chore the responsibility of the programmer. As well, it violates security. Suppose the value in a cell is erased, but the cell is still referenced by some data structures. These *dangling pointers* are unprotected and undocumented, and the source of difficult to trace errors.

Once automated way to keep track of memory usage is *reference counting*. Whenever a cell is used, or referred to, by part of a program, the reference count of that cell is increased by one. When a cell has no existing references, that cell is not accessible to the current program, and is thus on the list of free cells.

Another approach is *garbage collection*. Here inaccessible cells are simply abandoned. When the list of free cells is exhausted, the processor interrupts normal computation and enters a

garbage collection phase. A mark-and-sweep garbage collector passes twice over all memory cells. On the first pass, inaccessible cells are marked as such. On the second pass, the marked cells are returned to free storage. A serious problem for garbage collection is *nonuniform response time*, in that processing halts while garbage collection is occurring. If there are many cells to be reclaimed, this interrupt may be several seconds.

Some Observations about the LISP Language

- All valid expressions are valid programs. This provides arbitrary granularity. Programming consists of building up hierarchical languages built on a solid foundation.

- You are always in control of what is data and what is process. Programming is building data, then testing processes on it, then making those processes into data, and so on.

- All defined functions are provable, that is they are data structures you can talk about, and the way to talk about them is to assert their correctness.

- The programmer is always part of the computation. The **read-eval-print** loop can be seen as an interactive dialog. **Read** means listen to what the person says. **Eval** means do what the person asks you. **Print** means tell the person the results of the request.

- All objects are the same. There are base objects (atomic data) and compound objects built from atomic objects. Atomic objects (atoms) are the pieces of a program, the bricks. Function composition is the cement holding the atoms together. Nothing else is happening. Atoms define your conceptualization, the pieces of the world. Functions just define bigger pieces. Object-orientation is function composition turned inside out.

- Variables are just convenient and arbitrary names for compound objects. So a variable is meaningful only when it is in the same context as the object it names. This is called *scoping*.

- Function names are also variables. You can rename functions at any time, and you should always use names that are meaningful to you. *Write languages not programs*. Think like a human, not like a computer and write code that matches human thought.

- There are always two levels when programming, the syntactic and the semantic: what you see and what you mean. Representation and value. Try to align the two by defining the look of a program to remind you of its meaning. In general, programs that look good are good.

- Formulate knowledge in terms of patterns, and look for those patterns. Patterns can be abstract, with many things of the same class fitting a particular spot.

- Formulate operations as functions. Operations can be abstract, with many functions fitting the same operation. Use operations that address all objects at the same time. For example, rather than explicitly checking each object for a property (by writing a DO loop), just ask if the property is true for everything (using the function EVERY).

Assignment III: Pseudocode Emulation

Nothing to hand in.

Implement an emulator for your pseudocode formal syntax.

An *emulator* of a program is a different program, usually in a different language, that does the same thing as the target program. Emulators are often built for hardware: a software module performs the same functionality as the target hardware, but in software. Software emulation is usually much slower. Another example is programs which simulate, say, a Windows environment on a Mac OS; these programs emulate Windows on a Mac.

In Assignment II, you designed a language fragment and formalized it with BNF or another structuring tool. In this assignment, you will implement the syntax of your language. (You may elect to use a different fragment, or a completely different formal specification.)

The assignment is simple if there is a one-to-one correspondence between your specification and some existing language. For example, if you specify a WHILE construct, then the specification language can translate directly to WHILE is some existing language like C. What gets tricky is verifying that the correspondence holds for all cases and for all implementation strategies.

You can view the assignment as one of *metaprogramming*. You will be writing a program in say language A. This program takes another program in language B (the BNF spec for example) as input and translates it into a third program as output which is in language A but does the functionality of the input in language B.

A good emulator will include a lexical scanner and a syntactic parser to assure structural properties of the output.

PASCAL

The Algol language introduced many new concepts into language design, and as a consequence, spawned a number of new languages (e.g. PL/I) all of which were very complex and unmanageable. Pascal was a teaching language designed to reduce this burgeoning language complexity.

Another idea at the time was to develop *extensible languages*, based on a small kernel of functionality. Extensions added application specific functionality. However, extensible languages turned out to be very inefficient, since variable extensions made parsing and compiling difficult. As well, extensions were reduced to kernel functions, adding another level of language complexity which could not be optimized. Since kernel errors were not expressed in the application specific language, diagnosis was not transparent.

Pascal combines simplicity with generality, a result of learning how the innovations of Algol can be efficiently and elegantly combined.

New concepts introduced in Pascal include:

- *enumeration types*
  using names rather than numbers to represent finite sets
  compiled like an array in contiguous memory, efficient
  e.g.: type DayOfWeek = {Mon, Tues, Wed, Thur, Fri, Sat, Sun}

- *subrange types*
  for contiguous subgroups
  e.g.: type Weekday = {Mon .. Fri} of DayOfWeek

- *set types*
  for arbitrary collections
  efficient, encoded as binary array indicating set membership
  bit level operations for union and intersection
  subsets easy to define, e.g.: S := [1,3,5,6]
  e.g.: type set of 1..9

- *strong typing of arrays*
  static array types since typing is determined at compile-time
  cannot write dynamic array manipulation procedures

- *name structures* include bindings for
  constants, types, variables, functions, labels

- *case statement*

Pascal's control structures embody the principles of structured programming. Control structures have one entry and one exit point. All statements can be compound. Pascal eliminated the idea of block structure, a precursor to structured programming.

C

The C language mixes characteristics of several language generations, it is an amalgam of structured high-level features, low level implementation features, and even machine-level features. It lacks support for nested procedures and modular programming, and is machine architecture specific. It's creator Dennis Ritchie says: "C is quirky, flawed, and enormously successful."

Summary of Block Structuring

Activation Records

An activation record represents the state of a procedure or function call. It holds all the information relevant to one execution unit, or activation. Thus a procedure consists of

1. the program code — fixed, static, not part of the activation record
2. the activation record — dynamic, keeps track of context and computational results

The activation record itself consists of

1. ip: the *instruction pointer* to the next statement to be executed after the procedure call returns. Also called the *resumption address*.

2. ep: the *environment pointer* identifies the bindings and scope of variables

   2a: local context: names declared by the procedure;
   local parameters and variables.
   Also the *static link* to the nonlocal scope

   2b. nonlocal context: names declared by *surrounding* procedures
   Also the *dynamic link* to the activation record of the caller.

A static link is required to locate the environment of the definition. A dynamic link is required to locate the environment of the caller.

Environments

An *environment* is simply a binding list of names and their values. Which names are in an environment is determined by the scoping rules of the language. *Scoping rules* define how to locate the values of names which are not immediately local to the procedure being executed. The *context of a procedure* is the set of names declared by that procedure, together with the names declared in the surrounding procedures, with "surrounding" being defined by scoping rules. Every name and variable is local to some procedure, the default being the top level, or main procedure. The activation record of that procedure contains the name and its binding.

FORTRAN activation records are compiled statically; names are assigned a permanent memory address. Languages which permit recursion require dynamic creation of activation records, and dynamic searching of the context for variable bindings, since more than one copy of a procedure may be active at the same time. Since a primary cost in computation is finding variables and values, it is impractical to dynamically search for the context of every variable. Instead a *two-coordinate method* is used:

1. the *ep* accesses the activation record of the current environment (calling procedure).
2. an *offset* locates the variable within the activation record

Scoping

Static scoping: a procedure executes in the environment of its definition (syntactic structure)

Dynamic scoping: a procedure executes in the environment of its caller (semantic structure)

Procedure Activation

To activate a procedure:

1. Save the state of the caller
   Put the current ip in the caller's activation record.
   The local ep is already in the caller's activation record.
   The nonlocal ep is already in the static link of the caller's activation record.

2. Create an activation record for the called procedure
   Put the actual bindings of parameters in the parameter part.
   These must be evaluated first, returning either
   1) a value (call by value), or
   2) an address (call by reference), or
   3) a thunk (call by name) [thunk=address returning function]
   Add the static link to the environment of definition.
   The ip is not relevant until the called procedure calls a procedure itself.
   The dynamic link points to the activation record of the caller.

3. Enter the called procedure in the context of the new activation record.

To exit a procedure, basically reverse the above process.

1. Delete the called procedure's activation record.
2. Restore the state of the caller.

Closures

In languages which can pass procedures as parameters, the procedure is passed as closure. A ***closure*** is a ip-ep pair:

1. The ip contains the entry address of the actual procedure.
2. The ep contains a pointer to the environment of definition.

In order to *pass functions* as parameters, the entire activation record approach must be changed, leading to a functional programming regime.

Blocks

A *block* is a container for a collection of operations. Technically, blocks are implemented the same as are procedures. Blocks are degenerate procedures; the ep and its dynamic link are not needed in the block activation record.

Displays

An alternative method to searching up a scoping chain for nonlocal variables is to have all accessible contexts stored in an array which is searched directly when a nonlocal is referenced. This produces constant look-up times.

Efficiencies

Both static and dynamic environments can be nested, often many levels deep. To locate a nonlocal variable, the static environment must be searched outwardly. This is very inefficient. Here is a listing of the costs of various static operations. *SD* stands for the static distance between the context using a variable or procedure and the context defining the variable or procedure.

Operation	*Memory references*	*Display references*
local variable	1	2
variable access	SD + 1	2
procedure call	SD + 3	6
procedure return	2	5
pass procedure	SD + 2	
formal procedure call	5	
goto	SD	

We can see than when operations are deeply nested, display is better than searching scoping chains.

Assignment IV: Pseudocode Semantics

A three-minute presentation to the class.

Describe the semantics of a small pseudocode fragment.

There is no generally agreed upon model of the semantics, or meaning, of computation. This assignment may require research and creativity.

1. Select a small pseudocode fragment.

2. Define the way it behaves at the register-transfer level. Ie: what interactions with memory occur; what parts are moved and to where; what processes change the configuration of memory. What are the exact changes? This is the *operational semantics*.

3. Describe the assurances that the fragment does what it is supposed to do. Develop a set of preconditions and postconditions. Attempt to use the postconditions to prove the preconditions, and thus to prove the correctness of the program fragment. This is the *axiomatic semantics*.

4. Think about other possible ways that you can clearly and unambiguously define or describe the intended functionality of your code fragment.

Semantics

Semantics refers to the behavior of a program, while ***syntax*** refers to its structure. At this time there is no generally agreed upon representation for semantics. Theories of program semantics are also hotly debated. (In my opinion this is because the semantics of imperative languages is tied to machine architecture, and thus does not even address the relevant issue of program meaning.)

Static Semantics

Static semantics refers to programming language characteristics which cannot be expressed in BNF form, but can be verified by a compiler. This does not refer to semantics at all, thus it is a misnomer. It does make obvious that even syntax is inadequately defined, and the entire enterprise of assuring the meaning or behavior of a program is suspect, at least for imperative languages.

Examples:

Type compatibility rules:

Consider adding an integer to a real:

```
3 + 4.1 = ?
```

The types of these objects do not match. In a strongly typed language, such an addition would be a typing error. More conveniently, languages have ***coercion rules*** which permit some types to be dynamically converted to other types. In the above example, `Int -> Real`.

Variable declaration:

How can you assure that all variables are declared before they are used? This structural requirement cannot be stated in BNF, so it requires a meta-language stronger than BNF to talk about how the program behaves.

Closing brackets:

In some block structured languages, the beginning and end of each block is labeled:

```
begin do <body> end do
```

There is no way to specify that the name of the `end` statement (`do` here) matches the name of the `begin` statement.

Attribute Grammars

Attribute grammars extend the expressability of context-free grammars like BNF, to include the checks for static semantics. An attribute grammar is a context-free parse tree with each token augmented with a set of attributes (such as type and initialization information).

Intrinsic attributes are those properties of leaf nodes (i.e. names) which are not contained in the parse tree itself. The type of a variable is an example. It is usually included in the symbol table, but may not show up as part of the parse tree. These are also called *synthesized attributes*, since they pass information up the parse tree starting at the variable names.

In contrast, *inherited attributes* are those that are passed down the parse tree, properties that depend on the operator structure of the parse tree.

Example: Simple Assignment Statements for adding two numbers

Here is the attribute grammar for typing for an assignment statement, `A := B + C`

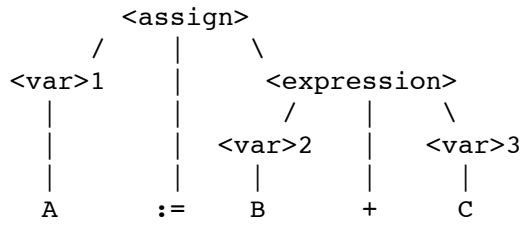
Syntax BNF:

```
<assign> ::=  <var> := <expression>

<expression> ::=  <var> + <var>  |  <var>

<var> ::=  A  |  B  |  C
```

Parse tree:



Static semantics:

We will assume that `{A,B,C}` are either integers or reals.

actual-type: a synthesized attribute which stores the actual type of a `<var>` or an `<expression>`. In the case of an `<expression>`, the type is computed given the types of the component `<var>`s.

expected-type: an inherited attribute which stores the expected type of `<expression>`. It is determined by the type of the `<var>` on the left-hand-side of the expression statement.

Attribute grammar:

Syntax: `<assign> ::= <var> := <expression>`
Semantics: `<var>.actual-type implies <expression>.expected-type`

Syntax: `<expression> ::= <var>1 + <var>2`
Semantics:
```
if (<var>1.actual-type = int) and (<var>2.actual-type = int)
  then int else real
<expression>.actual-type = <expression>.expected-type
```

Syntax: `<expression> ::= <var>2`
Semantics:
```
<var>.actual-type implies <expression>.actual-type
<expression>.actual-type = <expression>.expected-type
```

Syntax: `<var> ::= A | B | C`
Semantics: `lookup-type[<var>.string] implies <var>.actual-type`

One of the difficulties of attribute grammars is that specifying the semantic rules for an actual programming language is very difficult due to language size and complexity.

Types of Dynamic Semantics

The semantics of non-imperative languages is usually clear and straight-forward, because these languages were designed with their meaning in mind.

Functional languages: substitution semantics defined by *lambda calculus*

Logical languages: implication semantics defined by *predicate calculus*

Object-oriented languages: mathematical semantics defined by *domain theories*

There have been several approaches to specifying semantics for ***imperative languages***, none are entirely satisfactory.

Dynamic Semantics for Imperative Languages

A clear definition of semantics is necessary for

1. knowing how the language actually works
2. compiler design
3. proof of correctness

For program understanding, the semantic specification must also be small and intelligible.

Operational Semantics

Operational semantics defines the meaning of a program by executing its statements, either in hardware, or simulated in software. Meaning is the changes which occur in a machine's state, or memory.

Naturally the specifics of a machine architecture, its hardware implementation, and its operating system interact strongly with operational semantics. This makes operational semantics difficult to understand and very machine specific (i.e. non-portable).

The concept of a ***virtual machine*** was introduced to buffer operational semantics from machine specific details. The changes of state of the virtual machine (i.e. the software simulation) define program meaning; that is, a program is defined in terms of another program. For this idea to work, we must first translate a program into appropriate low-level statements in the assembly language of the virtual machine (i.e. statements about virtual registers, memory, and data movement). Then we must cast the changes in machine state in terms of changes that align with the programmer's intentions.

Since operational semantics, even on a virtual machine, is defined in terms of algorithms rather than mathematics, knowledge of machine changes does not help with program understanding or verification. As yet, there is no principled theory of algorithms.

Denotational Semantics

Denotational semantics is a rigorous attempt to define program behavior in terms of ***recursive function theory***. Each language object is defined both as a mathematical object and as a function which maps instances of the language object (as they occur in a program) onto the appropriate mathematical object. The problem with this approach is that it is not at all clear what the appropriate mathematical objects are for programming constructs.

Example: ***Binary Numbers***

Domain: `N` the set of all non-negative integers

We will associate each decimal number with some non-negative integer. The functional mapping `M` follows :

```
M['0'] = 0

M['1'] = 1

M[<binary-number> '0'] = 2 * M[<binary-number>]

M[<binary-number> '1'] = 2 * M[<binary-number>] + 1
```

Note that this is implemented by a recursive function:

```
M[bin] =def=
  if bin='0' then 0
    elseif bin='1' then 1
      elseif last[bin]='0' then 2* M[but-last[bin]]
        elseif last[bin]='1' then 2* M[but-last[bin]] +1
          else ERROR
```

The functions `last` and `butlast` are accessors for the binary number, decomposing by separating the last digit from all digits butlast. If we reversed the binary number before decomposition, then the accessors would be `first` and `rest`. That is:

```
last[bin] =def= first[reverse[bin]]

butlast[bin] =def= reverse[rest[reverse[bin]]]
```

Since recursive functions are a model for computation, there is a close relationship between operational semantics and denotational semantics: both require a virtual machine to identify the state changes which define the meaning of a computation. Denotational semantics is an improvement, since it relies on mathematical functions for definition rather than on algorithms.

Example: ***Assignment Statement***

Let the environment `E` (i.e. the state of the computation) be represented by pairs `(name1, value1)` of variable names and their current values. The mathematical function M defines the meaning of the assignment `x := <expression>`

```
M[x := expression, E] =
  if M[expression, E] = ERROR then ERROR        ;expression is not valid
    else E' = {...(namei', valuei')...}         ;new environment
      where for j=1..n
        if namej=x then                         ;compare names
          M[expression,E]
          else valuej' = get-value-from-environment[namej, E]
```

Axiomatic Semantics

This method evolved out of proving program correctness. The idea is that each program statement is surrounded by *constraints* (preconditions and postconditions) which specify the behavior of program variables. The language of these constraints is Predicate calculus. The constraints are called ***assertions***. Assertions are specified in a program by enclosing them in curly brackets.

Example:

```
{x isa integer, x>0}
  sum := 2*x + 1
{sum > 2}
```

The precondition specifies constraints on the input `x`. The postcondition specifies constraints on the output `sum`.

In designing preconditions, the ***weakest precondition*** is the least restrictive assertion which still guarantees the validity of the postcondition. If you can compute the weakest precondition given the postcondition, then a correctness proof for the statement can be developed. Continuing the example:

Given `((sum > 2) and (sum = 2x+1)),`

```
2 < 2x+1
1 < 2x
x >= 1/2
```

We derive that `x` is greater than `1/2`. The assertion that `x` is an integer cannot be proved, but it is know from the static semantics analysis. Thus

```
x>0   is true
```

It is also true that `x>5` (or any other integer) when `sum>10`. Since there are cases where this precondition is not true (i.e. when `sum<11`), `x>5` is not the weakest precondition.

The usual abstract syntax for assertions is:

```
{P} S {Q}
```

where `P` is the precondition, `Q` is the postcondition, and `S` is the statement.

Axioms cannot cover sequences of statements, since that would require a separate axiom for each different sequence of program statement types. Rather sequences are analyzed using rules of inference of the form

Preconditions imply `S1`; `S1` implies `S2`; ...; `Sn` implies postconditions

Thus to use axiomatics, the entire mechanism of proofs in predicate calculus is needed. This is both hard to understand and difficult to use. Like denotational semantics, although axiomatic semantics is a tractable idea, it becomes very complex for normal programming languages, and is thus of very limited utility.

Operator Calculus

Functional programming and mathematical programming are similar in that both support a formal semantics. Here are some examples of lambda calculus abstraction and evaluation.

```
F1[x] = +[x,1]                      x+1

F2[x,y] = *[x,-[2,y]]               x(2-y)
```

Abstract F1:

```
F1[x] = +[x,1]                x-->#

F1 = [#,+1 #]
```

Apply F1:

```
F1[3] = [#,+1 #] 3 = (subst 3 # (+1 #)) = +1 3 = 4
```

Abstract F2:

```
F2[x,y] = *[x,-[2,y]]         x-->%  y-->@

F2 = [%, *% (-2 y)] = [@, [%, *% (-2 @)]]
```

Apply F2:

```
F2[3,8] = [@, [%, *% (-2 @)]] 3 8 = [@, [%, *% (-2 @)] 3] 8
```

Note: The binding `x=3` must descend inward so that the abstraction `x=%` is paired correctly with the binding `3`. The direct descent used above is *non-standard* (I have changed the notation slightly so that the rule is easier to follow).

There are two possible evaluation orders, *normal* and *applicative*. These align with the mathematical and accumulative forms of recursion.

Normal evaluation expands functions within functions, without accumulating intermediate results. When all functions are evaluated, the entire expression is then simplified. Should some bindings not be available, normal evaluation returns the remaining function. In normal order, a function that is not fully defined could be passed for expansion (lenient semantics).

Applicative evaluation is standard in programming languages. Innermost functions are evaluated, reduced to ground, and then that reduced result is handed to the next outer function. In applicative order, all variables to be evaluated are guaranteed to be bound (strict semantics).

Evaluation strategies can be mixed, any step can be of either type. As with all algebraic languages, the order of application of substitutions does not matter.

Applicative Order, innermost leftmost first (data-driven, eager, call-by-value)

```
F2[3,8] = [@, [%, *% (-2 @)] 3] 8

        = [@, (subst 3 % (*% (-2 @))] 8

        = [@, (*3 (-2 @))] 8

        = (subst 8 @ (*3 (-2 @))

        = (*3 (-2 8)) = *3 -6 = -18
```

Normal Order, outermost leftmost first (demand-driven, lazy, call-by-need)

```
F2[3,8] = [@, [%, *% (-2 @)] 3] 8

        = (subst 8 @ [%, *% (-2 @)]) 3

        = [%, *% (-2 8)] 3

        = (subst 3 % (*% (-2 8)))

        = (*3 (-2 8)) = -18
```

Composing `F1` and `F2`:

```
F1[F2[x,y]]

F1 = [#,+1 #]
F2 = [@, [%, *% (-2 @)]]

F3 = F1[F2] = [#,+1 #] [@, [%, *% (-2 @)]]
```

Evaluating `F3`:

```
F3[3,8] = [#,+1 #] [@, [%, *% (-2 @)]] 3 8

        = [#,+1 #] [@, [%, *% (-2 @)] 3] 8
```

Note that there are three forms in sequence: `F1 F2 8.` We could apply `F1` to `F2` first (normal), or we could apply `F2` to `8` (mixed), or we could apply the inner abstraction in `F2` to `3` (applicative).

Applicative Order:

```
F3[3,8] = [#,+1 #] [@, [%, *% (-2 @)] 3] 8

        = [#,+1 #] [@, (subst 3 % (*% (-2 @)))] 8

        = [#,+1 #] [@, (*3 (-2 @))] 8

        = (subst [@, (*3 (-2 @))] # (+1 #)) 8

        = +1 [@, (*3 (-2 @))] 8

        = +1 (subst 8 @ (*3 (-2 @))

        = +1 (*3 (-2 8)) = -17
```

Normal Order:

```
F3[3,8] = [#,+1 #] [@, [%, *% (-2 @)] 3] 8

        = [#,+1 #] (subst 8 @ [%, *% (-2 @)]) 3

        = [#,+1 #] [%, *% (-2 8)] 3

        = [#,+1 #] (subst 3 % (*% -6))

        = [#,+1 #] (*3 -6)

        = [#,+1 #] -18

        = (subst -18 # (+1 #))

        = +1 -18 = -17
```

In the following example of the recursive factorial function, the `Y` combinator in lambda calculus specifies the recursive use of a function. Its behavior is to substitute the function definition whenever an operator hole is filled with the name of the recursive function.

```
FACTORIAL:     (define FAC (n) (if n=0 then 1 else (* n (FAC (-1 n)))))

Abstract  n --> %   FAC --> Y #

FAC = Y [#, [%, (COND (=0 %) 1 (* % (# (-1 %))))]]

FAC 2 = Y [#, [%, (COND (=0 %) 1 (* % (# (-1 %))))]] 2

      = Y [#, [%, (COND (=0 %) 1 (* % (# (-1 %))))] 2]

      = Y [#, (subst 2 % (COND (=0 %) 1 (* % (# (-1 %)))))]

      = Y [#, (COND (=0 2) 1 (* 2 (# (-1 2))))]

      = Y [#, (COND False 1 (*2 (# 1)))]

      = Y [#, (*2 (# 1))]

      = (subst FAC # (*2 (# 1)))

      = *2 (FAC 1)

      = *2 (Y [#, [%, (COND (=0 %) 1 (* % (# (-1 %))))]] 1)

      = *2 Y [#, [%, (COND (=0 %) 1 (* % (# (-1 %))))] 1]

      = *2 Y [#, (subst 1 % (COND (=0 %) 1 (* % (# (-1 %)))))]

      = *2 Y [#, (COND (=0 1) 1 (* 1 (# (-1 1))))]

      = *2 Y [#, (COND False 1 (* 1 (# 0)))]

      = *2 Y [#, (*1 (# 0))]

      = *2 (subst FAC # (*1 (# 0)))

      = *2 (*1 (FAC 0))

      = *2 (*1 (Y [#, [%, (COND (=0 %) 1 (* % (# (-1 %))))]] 0))

      = *2 *1 Y [#, [%, (COND (=0 %) 1 (* % (# (-1 %))))] 0]

      = *2 *1 Y [#, (subst 0 % (COND (=0 %) 1 (* % (# (-1 %)))))]

      = *2 *1 Y [#, (COND (=0 0) 1 (* 0 (# (-1 0))))]

      = *2 *1 Y [#, (COND True 1 (*0 (# -1)))]

      = *2 *1 Y [#, 1]

      = *2 *1 (subst FAC # 1)

      = *2 *1 1 = 2
```

Logic Programming

Prolog, and logic programming languages, take a formal approach to writing programs, using *Predicate Calculus* as the mathematical model. Unlike functional languages, logic languages provide a different type of programming interactivity: ***declarative style***. *Non-procedural programming* relies on an inbuilt computational engine within a language. Rather than describing how the computational engine should behave, a logic program describes the problem and the structure of the desired solution. The logic engine then searches possible structures to find the answer. This search process is not under user control, rather it is prescribed by the rules of logical inference. Logic program is synonymous with ***automated theorem proving***, in that the process of constructing an result is the same process as proving that result, given the input program as axioms.

Languages that do not require knowledge of control structures, internal implementations, and machine level details are called ***higher-level languages***. These languages represent the future of programming constructs. The syntax of high-level languages approaches direct mathematical description.

Example: The specification of a sort program for a set `S` of records might be:

```
For all i,j in S: if i<j then S[i] =< S[j]
```

Note that this program does not specify an algorithm; the choice of implementation strategy is under control of the logic engine.

Technique

Each statement, or clause, in a Prolog program makes an ***assertion***. An assertion can be either a statement of ***fact*** or a ***rule***. The difference between facts and rules is that facts do not contain variables, whereas rules do. Computation is initiated when a query is placed with the set of facts. ***Queries*** are the way to state the goals of a program.

A logic program returns the result of a query. When the query is answered, Prolog returns an example case for which the query is true. When a query result is false, either the query can be shown to be false through deduction, or there is simply no enough information in the program database to deduce a result to the given query.

Prolog generates answers purely through ***pattern-matching***. Patterns without variables must have exactly matching syntax. Variables within patterns can match arbitrary patterns. The matching technique is called ***unification***, which finds an assignment of values to the variables in a form which makes the goal statement syntactically identical to the head of some clause.

Prolog assumes that all available information is in its fact base. All statements that can be proved true are derived from the facts and rules of the Prolog program. This approach works well for most problems, especially for object-oriented and entity models.

It is possible to view Prolog clauses as definitions of procedures. Prolog relationships then become procedure invocations. One significant difference is that logic clauses can be processed ("evaluated") in any order, making logic programming a parallel process. Due to difficulties described below, Prolog itself cannot be executed in parallel. Another primary difference between conventional and logic programming is that the relational forms in logic do not distinguish between input and output.

No Data Types

There are no algorithms and no data structures in Prolog. Data types are defined implicitly by their properties. Properties are defined, as might be expected, by the mathematical model of the data structure. That is, Prolog data structures are necessarily and unavoidably abstract. Accessors, constructors and recognizers are all the same thing.

The implication token in Prolog, `:-`, can be read as "right-hand-side implies left-hand-side". Thus `A :- B` is "`A` is asserted if `B` is asserted".

Example, lists:

Prolog uses a shorthand notation for `cons`, `head`, and `tail`, the constructors and accessors of lists.

Mathematical	*Prolog*
`list = cons[head,tail]`	`list[[X|L]] :- list[L]`
`head = head[cons[head,tail]]`	`:- head[[X|L],X]`
`tail = tail[cons[head,tail]]`	`:- tail[[X|L],L]`

The "list" object is an abstraction, not an implementation. Therefore `[X|L]` can be interpreted as a generic decomposition operator; the underlying model and implementation could be of sets, list, arrays, or any other one dimensional data structure.

Example, append:

```
:- append[[],L,L]

append[ [X|L], M, [X|N] ] :- append[L,M,N]
```

This can be read: If appending `L` to `M` gives `N`, then appending `(cons X L)` to `M` gives `(cons X N)`.

The components of a list, or of any data structure are accessed implicitly, as part of the accessor-like decomposition of a form.

Example, set membership:

```
:- member[X,[X|T]]
member[X,[H|T]] :- member[X,T]
```

Example, sort:

```
:- sort[[]]
:- sort[[X]]
sort[[A,B|T]] :- A =< B, sort[[B|T]]
```

The form `[A,B|T]` separates the first two members of the list from the rest.

No Algorithms

Logic programming is designed to separate the control of a computation from the code. In fact, the ordering of clauses in a Prolog program determines the control flow. It is mandatory to put the base case of an inductive definition before the inductive case, since Prolog looks for matches by sequentially examining each clause in order.

Example, adding integers:

```
:- sum[0,X,X]

sum[succ[X],Y,succ[Z]] :- sum[X,Y,Z]
```

The successor function increments an integer by `1`. The first clause says:

```
0+X=X
```

This is the base of the recursive definition. The second clause is the inductive case. It says:

```
if X+Y=Z, then (X+1)+Y=(Z+1)
```

There are two evaluation methods for Prolog engines, top-down (also called backward-chaining), and bottom-up (called forward-chaining). In ***backward chaining***, we start from the goal and attempt to match clauses until the matching arrives at a fact. Backward chaining is similar to recursion. In ***forward-chaining***, we start from the facts and propagate matches until the goal is reached. Forward-chaining is similar to iteration.

Horn Clauses

The Satisfiability Problem for propositional calculus is NP, so logic programs can conceivably encounter great inefficiencies. To combat this, Prolog and other logic systems often restrict the form of logical assertions to those that are computationally tractable.

All logical propositions can be expressed in ***clausal form***. A clause has the structure:

```
A or B or C... :- W and X and Y...
```

One common subset of clauses is Horn clauses. A ***Horn clause*** is a logical assertion with at most one conclusion. Horn clause forms are:

```
A :- W and X and Y...          (one conclusion)
```
or
```
:- W and X and Y...            (no conclusions, a fact)
```

Prolog notation for Horn clauses uses a comma in place of `and` in the antecedent of the clause, so Prolog code would look like:

```
A :- W, X, Y,...
```

As well, Prolog drops to leading `:-` sign when the clause is a fact.

Resolution

We tend to think of logic as *natural deduction*, that form of logic which is closest to our natural language. Almost all machine implementations of logic, however, use ***resolution***, a computational form of logic. Resolution is a standardized representation of logic consisting of a conjunction of ***clauses***. Each clause consists of a set of ***literals***. A literal is either an atom or a negated atom.

Clauses can be of three different types:

Facts:	`:- mother[Mary, John]`	(Mary is the mother of John)
Goals:	`:- father[John, X]`	(John is the father of someone, X)
Rules:	`parent[X,Y] :- father[X,Y]`	(the father of Y is a parent of Y)

A clause specifies a ***relation*** between its components. In contrast to functions, relations do not have a direction of application; they can be used to find inputs given goals as easily as finding goals given inputs.

Functional form:	`F[X,Y] => Z`
Relational form:	`R[X,Y,Z]`

A relational form can optionally be read as a data structure and as a pattern to be matched.

The resolution technique converts clausal form into sets of literals:

```
A or B or C... :- W and X and Y...
```
becomes
```
{-W,-X,-Y,A,B,C}
```

A resolution step consists of combining two clauses, one with a positive literal and one with the negative literal for the same variable. The members of each clause are combined, while the target literal is omitted, generating a new clause which is asserted into the collection of existing clauses. An example:

```
{-A,B,-C,D} + {A,B,-E}  ==>  {B,-C,D,-E}
```

Horn clauses are those sets with at most one negated literal.

```
A :- B, C               (B and C imply A)
B :- E                  (E implies B)
```

becomes

```
{-A,B,C}
{-B,E}
```

A resolution step would match B with -B, delete both, and form the new clause with the rest of the literals in both the source clauses:

```
{-A,C,E}                (C and E imply A)
```

Control Structures

A Prolog program still has control structures, they are just not under control of the user. As in lambda calculus, control of pattern-matching can occur in two basic ways: from goals to facts, and from facts to goals Also like lambda calculus, the two evaluation strategies can be mixed in any order. *Example,* the Fibonacci relation:

```
fib[0,1]
fib[1,1]
fib[N,E] :- N=M+1, M=K+1, fib[M,G], fib[K,H], E=G+H, N>1
```

The inductive clause says that the Nth Fibonacci number is E, if it is the case that

1. M is N-1th Fibonacci number
2. K is the M-1th number, ie the N-2th number.
3. The Mth Fibonacci is G.
4. The Kth Fibonacci number is H.
5. E is the sum of G (the Mth number) and H (the Kth number).
6. N is greater than 1.

The matching algorithm is straight-forward. Suppose the following fact was asserted (the Fibonacci of 2 is R):

```
fib[2,R]
```

Using a backward chaining strategy, this pattern would be matched against all the left-hand-side goals of the logic program. The first match would be

```
fib[2,R]
fib[N,E] :- N=M+1, M=K+1, fib[M,G], fib[K,H], E=G+H, N>1
```

N gets the pattern 2, E gets the pattern R, generating a new clause with substitutions:

```
fib[2,R] :- 2=M+1, M=K+1, fib[M,G], fib[K,H], R=G+H, 2>1
```

This clause reduces by arithmetic to:

```
fib[2,R] :- M=1, M=K+1, fib[M,G], fib[K,H], R=G+H

fib[2,R] :-      1=K+1, fib[1,G], fib[K,H], R=G+H

fib[2,R] :-      K=0,   fib[1,G], fib[K,H], R=G+H

fib[2,R] :-             fib[1,G], fib[0,H], R=G+H
```

For the clause to be satisfied, each conjunct on the right-hand-side must be satisfied. Thus, the above clause would call `fib[1,G]` and `fib[0,H]`, both of which are ground cases:

```
fib[1,G]
fib[1,1]
```

`G` gets bound to `1`.

```
fib[0,H]
fib[0,1]
```

`H` gets bound to `1`. This leaves `R` equal to `2.`:

```
fib[2,R] :-             fib[1,1], fib[0,1], R=1+1
fib[2,2]
```

Note that the ordering both of clauses and of conjuncts within clauses strongly interacts with the efficiency of the matching algorithm. This means that Prolog fails to achieve control structure independence.

Using a forward-chaining strategy, the fact patterns are matched against the antecedents on the right-hand-side of the clauses.

```
fib[0,1]
fib[1,1]
fib[N,E] :- N=M+1, M=K+1, fib[M,G], fib[K,H], E=G+H, N>1
```

Matching `fib[0,1], M` gets `0, G` gets `1,` and simplifying:

```
fib[N,E] :- N=0+1, 0=K+1, fib[0,1], fib[K,H], E=1+H, N>1
```

This fails due to a contradiction, `N` is both equal to `1` and greater than `1.` The matching then continues with the next match opportunity, `K` gets `0`, `H` gets `1`:

```
fib[N,E] :- N=M+1, M=0+1, fib[M,G], fib[0,1], E=G+1, N>1

fib[N,E] :- N=M+1, M=1,   fib[M,G],           E=G+1, N>1
```

```
fib[N,E] :- N=1+1,        fib[1,G],          E=G+1, N>1

fib[2,E] :-               fib[1,G],          E=G+1, 2>1

fib[2,E] :-               fib[1,G],          E=G+1
```

The matching starts again form the top, with nothing matching `fib[0,1].` Using `fib[1,1], G` is bound to `1`:

```
fib[1,1]
fib[2,E] :-               fib[1,G],          E=G+1

fib[2,E] :-               fib[1,1],          E=1+1

fib[2,E] :-                                  E=2

fib[2,2] :-
```

With no more antecedents, the result `fib[2,2]` becomes a fact. This fact matches the query, binding `R` to `2`, and completing the computation:

```
fib[2,2]
fib[2,R]
```

Whether or not a backward or forward chaining strategy is better depends entirely upon the structure of the fact and rule bases. In the Fibonacci example, note that forward chaining avoids recomputing the Fibonacci numbers in each recursion. When a clause fails (as in the above example), the Prolog engine must ***backtrack***, that is, it must restart pattern-matching from where it previously left off, abandoning the exploration which ended in failure (and in unavoidable wasted effort). Also note that logic programming does not distinguish between input and output, since all structures are relations rather than functions. This makes questions such as the example below as easy to answer as finding output from input:

```
sum[X,3,5]                find X
```

Here we are asking what number added to `3` yields `5`? Subtraction, the inverse of addition, is not a necessary concept. Since Prolog treats structures relationally, inverses are not used.

In fact, there are dozens of matching strategies that have been explored by the theorem proving community.

Built-in Arithmetic

In the examples above, arithmetic simplification is built-in. This is not purely logical since it requires a different kind of evaluation mechanism. To have Prolog evaluate arithmetic using pattern-matching, the rules of arithmetic must be added to the rulebase. The new Fibonacci pattern is:

```
fib[0, succ[0]]
```

```
fib[succ[0], succ[0]]

fib[N,E] :-
      sum[M,succ[0],N], sum[K,succ[0],M], fib[M,G], fib[K,H], sum[G,H,E]
```

The successor pattern computes arithmetic relations by stepping though the numbers from `1` to `N`. However using the successor relation to define numbers is very clumsy since it provides no base system to abstract magnitude.

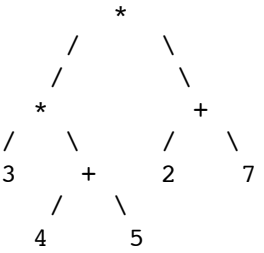
Data Structure Predicates

Compound terms can usually be expressed as trees. To represent a tree in Prolog, we label each node and specify a relationship between that node and its children.

Example: `(3 * (4 + 5)) * (2 + 7)`

String: `*[*[3,+[4,5]],+[2,7]]`

Tree:



Prolog:

```
sum[4,5,N1]
times[3,N1,N2]
sum[2,7,N3]
times[N2,N3,Ans]
```

Problems with Prolog

There are several problems with Prolog, as it was first formulated. Recently work on Prolog III has reduced these difficulties, turning Prolog into a pure constraint-based language.

Sequencing and Hard-wired Control

Logic programming still suffers from necessary sequencing of clauses and antecedents within clauses. This undermines the independence of program and control. The search strategy in Prolog is hardwired, it is depth-first. This also abandons the parallelism of a pure logic language. Since the language does not include control over different search approaches, Prolog search strategies have can be very inefficient, in some cases even non-terminating. Consider the following Prolog program:

```
ancestor[Mary,Tom]
ancestor[M,N] :- parent[M,N]
ancestor[X,Z] :- ancestor[X,Y], parent[Y,Z]
```

The intent of this code is to assert that a person is an ancestor if they are a parent, or if their parent is an ancestor. The problem is that binding ancestor results in an infinite sequence when backward chaining is used. In the first match,

`M` gets `Mary`, `N` gets `Tom`:

```
ancestor[Mary,Tom] :- parent[Mary,Tom]
```

The first rule yields the query "Is Mary Tom's parent?". This fails since there is no match to the `parent` pattern. The search backtracks, trying the second rule:

```
ancestor[Mary,Tom]
ancestor[X,Z] :- ancestor[X,Y], parent[Y,Z]
```

`X` gets `Mary`, `Z` gets `Tom`:

```
ancestor[Mary,Tom] :- ancestor[Mary,Y], parent[Y,Tom]
```

The search is looking for a missing link, `Y`, between `Mary` and `Tom`.

```
ancestor[Mary,Y]
ancestor[X',Z'] :- ancestor[X',Y'], parent[Y',Z']
```

Since the second rule has been used twice, the names of the logic variables must be changed to avoid name conflicts. The second rule is matched, in pursuit of the missing ancestor:

`X'` gets `Mary`, `Z'` gets `Y`:

```
ancestor[Mary,Y] :- ancestor[Mary,Y'], parent[Y',Y]
```

Two queries are generated, and the first one is explored. Again the first rule fails to identify the missing ancestor. The second rule is tried again:

```
ancestor[Mary,Y']
ancestor[X'',Z''] :- ancestor[X'',Y''], parent[Y'',Z'']
```

`X''` gets `Mary`, `Z''` gets `Y'`:

```
ancestor[Mary,Y'] :- ancestor[Mary,Y''], parent[Y'',Y']
```

This match continues by looking for a pattern to resolve the new antecedents:

```
ancestor[Mary,Y'']
ancestor[X''',Z'''] :- ancestor[X''',Y'''], parent[Y''',Z''']
```

The search has entered an infinite loop, looking for a binding for `Y, Y', Y''`, etc. Programming Prolog is not purely declarative, since a programmer must know the exact behavior of the search-and-match algorithm in order to avoid infinite looping.

Cuts

In order to stop needless and possibly infinite search, Prolog introduced the *cut operator*, **!**. In essence this is saying that Prolog does not have control orthogonality, the central reason for the language has been negated. For example, consider the false query "Is the factorial of 4 equal to 15?". The goal is stated as a conjunction of two assertions:

```
fac[4,A], A=15
fac[0,1]
fac[N,E] :- N1=N+1, fac[N1,R1], E=R1*N.
```

The first subgoal, `fac[4,A]`, succeeds, binding `A` to `24`. Next the second goal is attempted and it fails, since A cannot be both `24` and `15`. This causes backtracking to the last successful goal, which was the ground case of the factorial pattern as it was solved recursively. This is `fac[0,R]`. Prolog thinks the match to the first clause, `fac[0,1]`, must have been in error, so it abandons the (correct) ground case, and attempts to bind the current query to the second, inductive clause. With a forward-chaining strategy, this yields:

```
fac[0,R]
fac[N,E] :- N1=N+1, fac[N1,R1], E=R1*N.
```

`N1` gets `0, R1` gets `R:`

```
fac[N,E]  :- 0=N+1, fac[0,R], E=R*N.

fac[-1,E] :-                  E=R*-1
```

Prolog is now looking for factorial of -1, and about to enter an infinite loop. The solution to this problem is to use the cut operator:

```
fac[0,1] :- !
fac[N,E] :- N1=N+1, fac[N1,R1], E=R1*N.
```

This tells the engine to stop if a ground case is satisfied. The problem is that the programmer is now using a procedural metaphor, defining the steps of the actual computation.

Negation as Failure

Prolog uses negation is two senses. At the logical level, negation specifies the complement of its argument. At the control level, when a clause returns false (not true), negation is interpreted as the failure of a clause, which then triggers backtracking.

The essential problem is that the dual use of negation makes it impossible to make and verify negative assertions. The source of the problem is actually in using Horn clauses rather than a fully expressive logic. The limited expressability of Horn clauses eliminates the use of negated clauses (but not negated literals).

Example:

```
append[X,Y,[a,b,c]]
```

yields all possible solutions:

```
X=[], Y=[a,b,c]
X=[a], Y=[b,c]
X=[a,b], Y=[c]
X=[a,b,c], Y=[]
```

The assertion

```
not[not[append[X,Y,[a,b,c]]]]
```

should behave identically to append, since the double negation cancels out. However, the Prolog execution makes the following error. The outer not sets up the inner not as a goal. The inner not sets up append, which succeeds. Therefore the inner not fails, since `not[append[X,Y,[a,b,c]]]` is false. This failure deletes the prior successful bindings of `X` and `Y`. Finally the outer not succeeds, however the logical variables at this point are unbound.

Ada

Ada was developed in recognition of the need for modular and reliable programs. It introduced abstract data types supported by separable modules. Abstraction requires ***information hiding***, users have to access modules through an abstract interface (mathematical not implementation) which hid implementation details. The basic structure of the language closely followed Pascal.

Ada was first developed for DoD applications of *embedded computing*. To assure portability, the DoD did not allow the development of either subsets or supersets of the Ada language (this was later changed in Ada95).

Declarations

The most significant difference between Pascal and Ada is in ***declarations***, those non-executable statements in the front of a program which inform the compiler and other preprocessors about the semantics of the language. Ada declarations are of five types:

1. Object	constants and variables
2. Type	object types
3. Subprogram	functions and procedures
4. Package	(new) modules
5. Task	(new) modules which execute concurrently

Modules (packages and tasks) are disjoint environments which communicate through defined ***interfaces***. Module declarations have two parts: the interface specification and the body of the implementation. The central difference between a package and a block is that packages have names and formal parameters, while blocks do not.

Data Structures

Ada was the first to introduce ***floating-point*** and ***fixed-point*** number types. Floating-point numbers have round-off errors while fixed-point numbers have an absolute error bound.

Ada introduced new typing tools. ***Subtypes*** are subsets of a type. ***Constraints*** are restrictions on the members of a type which can be evaluated at runtime. ***Derived types*** foreshadowed object-oriented inheritance, they are types which inherit operations, functions, and attributes from a parent type.

Name Structures

The block structure of Algol still permitted global variables, in that blocks provided encapsulation of control but not of names. The problem was ***side effects***, which can be defined as hidden access to a variable. A related problem was ***indiscriminate access***, that is, no programming tools prevented access to variables, even when access was inappropriate. There

was no way in block structured languages to prevent indiscriminant access. Yet another related problem was ***vulnerability***, there was no way to preserve access to a variable, in that a new declaration might intervene between an old declaration and the use of variable, blocking the scope of the old declaration. Finally, block structure permitted ***overlapping definitions***, that is, shared access to variables. This undermines modularity.

Parnas provided two principles of information hiding:

1. One must provide the user with all the information needed to use a module, and nothing more.

2. One must provide the implementor with all the information needed to complete the module and nothing more.

That is, the user cannot write programs which access the implementation details, while the implementor has no knowledge of the context of usage of the module.

The Ada construct which supports information hiding is the ***package***. This is achieved by having two separate components of a module, the interface and the implementation. Packages control name access by mutual consent: the package implementor nominates accessible variables by making them ***public***, while the package user ***imports*** a package when its public variables need to be used.

Packages are abstracted by the notion of a ***generic package***. Generic packages provide a template which can be instantiated by multiple instances of the package. However, generic packages are difficult to compile. Here is Ada code for a type independent generic module for stacks:

```
generic
  Length : Natural := 100;
  type Element is private;
package stack is
  procedure Push (X : in Element);
  procedure Pop (X : out Element);
function Empty return Boolean;
function Full return Boolean;
Stack_Error : exception;
end Stack;
```

`Element` is a ***type parameter*** which is declared to be private. Thus the package can be instantiated with stacks of different types. Here are two examples of construction of new stacks:

```
package Stack1 is new Stack (100, Integer)
package Stack2 is new Stack (256, Character)
```

`Stack1` can accommodate 100 integers, while `Stack2` can accommodate 256 characters.

Control Structures

Ada control structures are similar to those of Pascal. Since Ada was intended for embedded applications, it was important that Ada have ***exception handling*** capabilities. Ada permits definition of exceptional circumstances, and provides mechanisms for signaling their occurrence and responding to their occurrence. Although all other names in Ada are bound statically, exceptions are bound dynamically. (Thus exceptions are exceptional.)

Ada introduced ***position-independent parameters***, that is, parameters can be in any order. This is achieved by the simple expedient of labeling parameters with names. The names identify the parameter's function. As well, parameters could be given a ***default value***. These changes in the definition of parameters make compiling more complex.

Concurrency

Ada provides a ***tasking*** facility, which allows a program to do more than one thing at a time. Tasks that are both concurrent and in communication must be synchronized. Ada ***synchronization*** is very much like mutual procedure calls. When a task has some data to communicate to another concurrent task, it calls that task, passing the data as parameter bindings. The only difference is that the first task does not halt, rather it keeps on processing. Should a concurrent task need data before it is sent by another task, that task simply waits until the data is sent. Should a task send data before it can be received, the sending task again waits until the data is received before continuing. This type of coordination is called a ***rendezvous***; the communication regime is called ***synchronized communication***. Should a rendezvous fail to take place, both tasks may wait indefinitely; this is called ***dead-lock***.

Tasks are ***tightly-coupled*** when they mutually communicate, waiting in turn for data. Tight-coupling has the disadvantage that both tasks must process at nearly the same speed. That is, the speed of processing is limited to the slowest task. To ***loosely-couple*** tasks, a buffer must be inserted into the communication stream.

Malignant Growth

Ada grew into a language which was too large, about three times larger than either Pascal or Algol. This means that the language is difficult to learn and more difficult to manage. Increase in language size can be viewed as a kind of entropy, causing the design to deteriorate over time. Another term for this is ***featuritis***, a phenomenon which is prevalent in committee designed languages. The benefits of adding a feature appear to outweigh the cost of adding a small increment to the language. Benefits seem clearer and easier to justify since they are small changes. However, their accumulated effect is a global negative. Features are by definition added piecemeal, independent of consideration of the entire language. This leads not only to excessive size, but also to feature interaction which can, at worst, increase complexity and errors exponentially.

Code Attachments

The attachment on the next page illustrates Ada code for the abstract data type Complex Number. It includes the two parts of a package: the ***public interface*** and the ***private implementation body***.

The two pages after that contain a package for the type Communication, which includes `Send` and `Receive` functions, and a buffer for loose-coupling.

The ***protected type*** construct of Ada controls concurrent access to shared data between tasks. Since it is a type definition, it is a template which must be instantiated with an object definition to create an actual instance. The effect of a protected type is to assure that only one task can execute changes to internal data structures (here, a buffer) at one time. This ensures consistent data management without the overhead of a third task. The coordination is achieved through the ***entry*** construct, which is a guarded function call. A ***guard*** allows only one active call at a time, other calls to the same entry are temporarily blocked until the controlling call is finished. The final page includes code for the Communications task, making it a concurrent procedure.

Language Generations (a recap)

Generation	*Exemplars*	*Characteristics*
0	pseudocodes	syntactic sugar for primitive assembly languages
1	FORTRAN	data and control correspond to machine architecture, linear, card-oriented
2	Algol	hierarchical name structures, block structure, strong typing, still linear and machine oriented
3	Pascal	simplicity and efficiency, user-defined data types, application-oriented
4	Ada	data abstraction, concurrency, still sequential, summary refinement of previous generations
5	LISP, Prolog, Smalltalk, JAVA	comprehensive, formal paradigms (functional, logical, object-oriented), self-documenting, recursive, provable, simulations, semantic constraints
6	?	hardware/software codesign, application specific, embedded, fine-grained strong parallelism, programming environments, language frameworks

The popular **C** language is a relatively unprincipled combination of first, second, and third generation language characteristics. The **C++** language modifies C to incorporate fourth and fifth generation characteristics, again without strongly embedded design principles.

Assignment V: A New Method

One written page recounting your experiences.

Explore a new style of programming.

1. Select a programming language and metaphor that you have not previously used. Obtain a copy of the appropriate programming language (this can be time-consuming).

2. Select a small fragment of code that you have written (alternatively you can use your pseudocode fragment from previous assignments, or you can select some code from a published source).

3. Transcribe and implement your code fragment in the new language.

4. What did you learn? Prepare a one to two page report on your experiences. Concentrate on the differences between the languages you are using.

Language Sources:

JAVA (pure object oriented): http://jave.sun.com

Haskell (pure functional): http://haskell.org

Scheme (functional): http://www-swiss.ai.mit/ftpdir/scheme-7.4/

Forth (tiny, threaded) http://chemlab.pc.maricopa.edu/pocket.html

LISP: http://www.franz.com/downloads/

Prolog:
http://www.cs.cmu.edu/Groups/AI/html/faqs/lang/prolog/prg/part2/faq-doc-2.html

ATLAST (tiny, embedded) and DIESEL (tiny, string-based): http://fourmilab.ch/

Screamer (constraint-based extension of LISP, untested):
http://www.cis.upenn.edu/~screamer-tools/index.html

In general: http://dir.yahoo.com/Computers_and_Internet/Programming_Languages/

JAVA

Features

simple
object-oriented (relatively pure oo, not procedural + oo extensions)
distributed
both interpreted and compiled instruction sets
robust
secure
architecture neutral
portable
high-performance
multi-threaded
dynamic

Object Orientation

class = abstraction
    class variables
    class functions
instance
    fields are instance variables
    methods are functions
hierarchy
subclasses = design by difference
inheritance
overloading
constructors
accessors
encapsulation (public, package, protected, private)

Implementation Features

virtual machine
    byte-code = machine instructions for a virtual machine (VM)
    VM maps closely to most native hardware machine
call-by-value parameter passing (compare to call-by-name, call-by-need)
    the value of an object is its reference
    copies binding into parameter field of method
automatic garbage collection
streams
type-safe references (strong typing)
exception handling
multiple threads (multitasking, lightweight)
simultaneous processes and shared objects
    locks; user provided deadlock avoidance
    automatic switching, scheduling, synchronization

Language Features

base data-types are not objects
first-class strings, read-only
international Unicode character set
first-class exceptions, checked by compiler
HTML inline interface
first-class network interface (URL, TCP, sockets)
protection and security model
class Object is root
interface concept for limited multiple inheritance

no pointers (use references instead)
no global variables (use root classes)
no goto (use catch/throw and labels)
no operator overloading (static basic operators)
no delete

Language Keyword Features

final:	constants, unforgable classes, non-overridden methods
this:	reference to self object
new:	constructs a new object or class
. :	accessor function
[ ] :	arrays
{ } :	sequential block
super:	references things from the superclass(es)
try-catch-finally:	exception handling
labeled break:	for skipping sequences and exiting loops

Packages

class libraries
functionality groups
user interface code provided
user provide application specific abstract data types

Provided Java API Packages

java.lang	the language
java.net	networking
java.io	streams and files
java.util	utilities, higher-order data-structures (enumeration, vector, stack, dictionary, hashtable)
java.awt	Abstract Window Toolkit
java.awt.image	image processing
java.awt.peer	interface with native interfaces
java.applet	basic applets

Interfaces

unique in Java
separate design inheritance from implementation inheritance
can inherit a contract without inheriting an implementation
tie together dissimilar classes for object reference
subclasses provide code for all interface methods
multiple inheritance (classes can implement multiple interfaces)
no root, does not default to Object root-class
constrained to:
- abstract class (no instances, only subclasses)
- no code, only abstract method declarations
- static and final variables
- public methods

Exceptions

catch and throw handlers
programmer declared compile-time errors
cleanly checks for errors without cluttering code
try/catch/throw environment
finally clean-up

Protection

runtime system does not permit memory access

public	full access by all classes
package	access by classes in common library
protected	access by subclasses only
private	no access by other classes

Streams

usually paired as InputStream, OutputStream
Piped, Filter, Buffered
StreamTokenizer

System Programming Classes

Runtime	(state of Java at runtime)
Process	(running java process)
System	(state of environment)
Math	(standard computations)
Native	(foreign function interface)

Abstract Window Toolkit (AWT)

- embedding within the local browser
- standard component set
  - button, checkbox, choice, label, list
  - scrollbar, textarea, textfield,
  - windows, menus, dialog boxes
- containers
  - graphical collections of components
- layout management
- event handling
  - mouse clicks and movements
  - keyboard
- graphics
  - drawing, color, fonts, clipping, image handling

Sample HTML Applet Call

```
<HTML>
<HEAD>
<TITLE>Applet Page</TITLE>
</HEAD>
<BODY>
<H4>This is an example of a Java applet:</H4>
<HR> <APPLET CODE="MyApplet.class" WIDTH=100 HEIGHT=50> </APPLET> <HR>
</BODY>
</HTML>
```

Sample Applet

```
import java.applet.Applet;
import java.awt.Graphics;
public class MyApplet extends Applet
      {public void paint(Graphics g)
            { g.drawString("Hello world.", 5, 10); } }
```

Web Resources

http://java.sun.com/	...from the Source
http://www.rpi.edu/~decemj/works/java.html/	a Java book author
http://www.gamelan.com/	registry of programs
http://sunsite.unc.edu/javafaq/javafaq.html	FAQs
http://www.well.com/user/yimmit/	links to resources
http://www.natural.com/	major developer
http://www.io.org/~mentor/J__Notes.html	more resources
http://www.acm.org/~ops/java.html	ACM resources
http://www.yahoo.com/Computers/Languages/Java/	search engine resources
http://rendezvous.com/Java/hierarchy	class diagrams

DEFINITIONS

Abstract (oo)
a class which is intended to have no instances

Accessor (prog)
special function to retrieve hierarchical data

Applet (java)
a dynamic, interactive program that runs inside a Web page

Attributes (prog)
the instance variables of an object

Bytecode (java)
machine instructions for a virtual machine

Casting (java)
changing the type of data. Coercing.

Class (oo)
template which abstracts objects with similar features

Clipping (graphics)
redrawing within a container

Constructor (prog)
special method for creating and initializing new instances

Contract (java)
semantics of the set of methods, no class implementation

Encapsulation (oo)
limited access to class methods and fields (public, package, protected, private)

Errors (java)
Runtime violation of system constraints; usually not recoverable.

Exceptions (java)
Compiler checked violations of typing, ranges, assignments; usually catchable.

Finalizer (java)
special method for closing and reclaiming old instances. Inverse of constructor.

Garbage Collection (java)
automated management of memory

Inheritance (oo)
hierarchy of included functionality; design and implementation by difference.
Single inheritance: inherit from only one superclass (tree)
Multiple inheritance: inherit from several superclasses (DAG)

Instance (oo)
concrete digital objects with bound properties. Same as Object.

Interface (java)
limited type of class which provides multiple inheritance
abstract class, no method implementation, static and final variables

Method (oo)
the functions within an object or a class

Overriding (oo)
subclass methods which redefine superclass methods

Package (oo, java)
set of classes, usually with functional similarities. Same as Class Library

Polymorphism (oo)
objects belong to all classes in their class hierarchy

Signature (prog)
the abstract form of a method (name, type of object returned, parameter list)

Statement (prog)
A program component. Expressions return a value; declarations define a scope.

Streams (prog, java)
A communication path between data source and destination

Subclass (oo)
the class(es) below a class in the class hierarchy

Superclass (oo)
the class(es) above a class in the class hierarchy

Threads (prog)
basic unit for multitasking, used for long processes

Variable (prog)
the data within an object or a class

Virtual Machine (java)
software which emulates a physical machine

Smalltalk

Object-oriented programming is a fifth-generation style which emphasizes simulation of the behavior of objects. Smalltalk was the first pure object-oriented (oo) language; Java is the most popular oo language currently.

Alan Kay lead the development of Smalltalk, following his intuition in the late 60s that personal computing (which did not exist at the time) would not succeed without a more friendly programming language. Recall that Algol and other second generation languages of the time were still closely tied to mainframe computing, and thus in the domain of computer specialists. Smalltalk was the programming language for Kay's seminal idea of the laptop computer. The developmental philosophy was:

> *Simple things should be simple, complex things should be possible.*

Smalltalk evolved from Simula (Nygaard), a simulation language, and LOGO (Papert), a very simple pedagogical language used to teach programming to children 8-12 years old. The design was also heavily influenced by research in developmental psychology (Dewey, Montesorri, Piaget, Bruner), and by Sutherland's Sketchpad, the first prototype VR system (late 60s). It was also conceived as part of an integrated graphical development environment which was developed at Xerox PARC in the early 70s, and lead eventually to the MacIntosh WIMP interface (Windows, Icons, Menus, Pointing device).

These are the characteristics which were seen to compose a user-friendly language:

object-oriented simulation
graphical interface
interactive (interpreted)
programming through dialog
integrated development environment

Objects, Messages and Methods

The programming unit in oo languages is the object. Objects have both state and behavior. Behavior is triggered by sending messages to objects. Repetitive behavior is simplified by control structures. The functions which define object behavior are called methods.

It is important to realize the oo programming on a single processor machine is simply a reorganization of code at the user level. Messages are procedure invocations. Objects are data structures. Methods are functions. Consider finding the area of a rectangle:

`rectangle[area-of]`	object-oriented
`area-of[rectangle]`	functional

Swapping function and argument turns object-oriented into functional programming.

An abstract object, one with parameters rather than bound values, is a class. Instances are created by instantiating a class description with values. In functional terms, a class is a set of operators, an instance is applicative expansion of an operator. Class inheritance is normal expansion of an operator.

The data values inside an object represent the properties and relations in which that object participates. The methods inside an object simulate the behavior of the corresponding semantic object. Objects then hold the state of the computation. In general, each object acts as an autonomous agent that is responsible for its own behavior, and is not responsible for the behavior of any other object. The memory organization of a computation is organized around objects; the information usually carried in function activation frames is stored within each object.

Classes

A class definition specifies all the properties and behaviors which are common to all instances of that class. Decomposing a problem into classes is analogous to functional decomposition in functional languages.

The biological analogy is that classes are genotypes while instances are phenotypes. Classes contain the organization of an entity, those characteristics which are present in all individuals of that type. Instances contain the structure of an entity, those characteristics which differentiate one individual from another.

Classes (abstract objects) also have methods. Class methods are used to create new instances. When a `new` message is sent to a class, the class constructs a copy of itself and binds the class parameters to the instance values conveyed by the `new` message.

A class hierarchy develops when the class abstraction principle is applied to classes themselves. (This is simply saying that operators can be composed without instantiation.) The class hierarchy is an organizational technique at the interface. Unfortunately, semantic objects are not orthogonal, class decomposition (like function decomposition) can be achieved in many equally valid ways, each way being appropriate for some behaviors, and blind to other behaviors.

For example, consider the class `mammal`. A biological taxonomy places mammals in a kingdom-phylum-species hierarchy, comparing mammals to other living creatures. A pragmatic classification, on the other hand, may classify mammals by their utility, as pets, beasts of burden, pests, sources of food, etc. A geological classification may classify mammals by their ecological zone, temperate, island, arctic, etc. Each of these classification schemes is orthogonal to the others, however a particular mammal gets classified by each differently.

The original solution was multiple inheritance, objects could inherit from several classes. This idea introduces as many problems as it solves. For instance, inherited methods from two classes may be contradictory. Inheritance from many classes builds objects which are far larger than any class they may inherit from. And from an coding perspective, multiple inheritance is extremely difficult to implement, essentially doubling the size of an oo compiler. Multiple inheritance, from a functional perspective, is attempting to insert control logic into lambda

calculus, thus undermining the semantics of the model. For these reason, the Java language does not provide the option of multiple inheritance.

As an example of a class hierarchy, a partial listing of the built-in classes in Smalltalk is presented on the next page.

- Object
  - Magnitude
    - Character
    - Date
    - Time
    - Number
      - Float
      - Fraction
      - Integer
        - LargePositiveInteger
        - LargeNegativeInteger
        - SmallInteger
  - Collection
    - SequenceableCollection
      - LinkedList
      - ArrayedCollection
        - Array
        - Bitmap
        - String
      - Interval
      - OrderedCollection
    - Bag
    - MappedCollection
    - Set
  - DisplayObject
    - DisplayMedium
      - Form
        - Cursor
        - DisplayScreen
    - InfiniteForm
    - OpaqueForm
    - Path
      - Arc
      - Curve
      - Line
      - Spline
  - Behavior

In this class hierarchy we see a structured decomposition of concepts from mathematics (e.g. collections), programming (e.g. float numbers), and graphics (e.g. DisplayObject). The decomposition is rather ad hoc (e.g. bags are a mathematical extension of sets but both are listed at the same level of abstraction), and is quite interface dependent (e.g. DisplayObejcts assume a WIMP interface).

Overloading, Information Hiding and Extensibility

Overloading refers to using the same token to trigger different behaviors in different objects. Since the organizational structure isolates bindings and methods within specific objects and classes, the issues of scoping and binding regimes are not troublesome. That is, oo techniques remove the machine dependencies associated with names in procedural languages. Again from the functional perspective, this is simply that functional organization does not require variables.

Classes serve as abstract data types, therefore they provide an abstraction barrier between model and implementation. These ideas are the same as those which motivated packages, generic packages, and tasks in Ada. Classes provide enforced modularity.

Due to class inheritance, Smalltalk is flexible and extensible. The programmer can define a new class which inherits from existing classes, thus significantly reducing both programming effort and possibility of errors.

Message Sending and Protocols

In procedural languages, programs are active and data structures are passive. This reflects a hardware architecture model in which memory is used to passively store, while computational circuitry is used to modify bits and words. In oo, objects are active, they respond to communications from other objects, modify themselves, and send messages intended to communicate with and change other objects.

The set of messages a particular object responds to is called its *protocol*. It is an error to send a message to an object which does not include that type of message in its protocol. Names in Smalltalk are *not typed*, any name can be associated with any object. Instead, protocols provide strong type checking, in that all valid messages are responded to by the receiving object. Since messages are sent at runtime, Smalltalk uses *dynamic*, as opposed to static, *type checking*. Dropped messages signal run-time errors which halt computation (without crashing the system), much like an interpreted language.

All objects are identified by a pointer, or *object reference*. Since there are no functions in an oo language, the cost of storing both variables and message invocations is identical. Activation frames are no longer a relevant concept. OO techniques allow algorithms to be factored out of the program without complication.

In a single processor system, messages are still procedure invocations. The major difference from procedural approaches is that instances of objects are constructed and initialized dynamically, at run-time, rather than statically in the object code.

There are three message formats in Smalltalk. For messages with no parameters, the name of the message serves as a keyword to trigger the message functionality. For messages with one or more parameters, keywords are again used to identify parameters, as in

```
Box grow: 100 color: green scrollbar: false
```

This approach however is awkward for numerical operators, for instance

```
x plus: 2
```

sends the plus message to the object `x`, with the parameter binding `2`. Smalltalk originally began as a pure oo language, even each number was an object. In a design concession, Smalltalk was changed to treat numerical computational more conventionally. So,

```
x + 2
```

is written instead, although this code still sends the + message to the object `x`.

Top Level

Smalltalk is *meta-circular*, its main loop is written in Smalltalk:

```
true whileTrue: [Display put: user run]
```

The `true` object is sent the `whileTrue` message which is bound to the object generated by sending the `Display` object a `put` message bound to `user`. Inside `user`, an object `userTask` responds to the `run` message by reading an expression, evaluating it, and returning the result to `Display`. `Run` is defined as:

```
run =def= Keyboard read eval print
```

Note that the Smalltalk style is similar to function invocation. The operational semantics of the language is to send the leading object the first message which follows. The object returns a different object which then responds to the next remaining message, and so on.

Above, the `Keyboard` object responds to the `read` message by prompting the user interface and returning the string typed by the user. This input string then responds to the message `eval` by calling the Smalltalk interpreter for evaluation. The resultant object responds to the message `print` by printing the result to output.

Concurrency

The autonomous nature of objects makes concurrency natural for oo languages. In Smalltalk, concurrency is achieved by having an object capable of processing a set of messages. Smalltalk uses the functional concept of *mapping*. To run several concurrent tasks, `map` the object over the tasks. This in turn is converted into time-sharing threads by the os.

```
concurrent-run =def= scheduler map: [...]
```