INTRODUCTION

2 The Variety Of Programming Languages
3 SOME LANDMARK GENERAL PUPOSE LANGUAGES.

CLASSIFICATION OF PROGRAMMING LANGUAGE,

4 By Application, Environment, Math Model
5 PROGRAMMING- MACHINE, FUNCTION, AND LOGIC,
Program As Imperatives (Machine Commands), s a Function, As Logic Assertions Based Queries
6 CLASSIFICATION By Math Model figure, Facilities Important For All General Purpose Languages
7 THREE EXAMPLE LANGUAGES, Imperative, Functional, Logical figure

TRANSLATION AND LANGUAGE

8 HIGHER LEVEL to MACHINE LANGUAGE-Interpretation , Compilation-figure
9 COMPILATION - Four Phases-figure
10 COMPILATION Four Phases EXAMPLE figure
11 BINDINGS

RECURSIVE FUNCTION DEVELOPMENT OF BASIC OPERATIONS

12 Recursion-Development Of Sum, Product , Powers, Based on Successor/Pedecessor
13 COMPARING SYNTAXs FOR RECURSION (Math and Scheme)

Program Examples Scheme, Prolog, C++, P-Threads

14 EXAMPLE: FUNCTIONAL-SCHEME
15 EXAMPLE: DECLARATIVE-PROLOG: DEFINITIONS AND QUERY CALLS and RESPONSE
16 EXAMPLE: PROCEDURAL-IMPERATIVE-C++
17 EXAMPLE: CONCURRENT-PTHREADS

18 Comparison Of Language Types [“Choice Of Language”]

Comparison Of Scheme and Prolog On the Same Problem-Banking

19 PROLOG and SCHEME BANKING
20 C++ BANKING

Prolog Inference Engine

21 Prolog Inference Engine Example Banking
22 Prolog Inference Engine Example Weather Activities

The Relation of Scheme and Prolog, Banking

23 SNOBOL EXAMPLE

CONTENTS

© 2006 M C Paull

I The Variety Of Programming Languages

Amongst their many activities computer scientists originate, implement, and use Programming Languages. Facilities with which to **construct *any* function** are found in all general purpose programming language. This power can be achieved with **suprisingly simple (Turjng Machine**) means, but a viable language must allow these things to be done gracefully and safely. The search for constructs, and concepts and ways of embedding them in programming languages, has provided the impetus for the evolution of programming languages. It is these constructs and the variety of ways they are embedded in different languages which is our subject.

I.1 **History of Computer Languages ("Toward Higher Level Languages")**

The evolution of language facilities has been punctuated by the appearance of new languages.
The themes of this evolution included:

1. Growing abstraction, i.e., higher level constructs. Built -n functions which do more and more. (IO, Matrix, String operations)
2. Growing facilities within the language for defining abstractions, including.facility to define functions, data-structures, and abstract-data structure and their generalization = classes definitions
3. Growing facilities for structuring program into seperate modules which can be handled independently.

Initially computers were used for mathematical calculations, the goal of programming language development was to make the description of algorithms for solving mathematical problems as natural as possible. Language are designed for ease in the description of algorithms, specification of I/O, and/or ease in understanding and/or designing algorithms that will run efficiently in the use of time and/or memory, One may need vocabulary for speaking about pictures, text, mathematics as well ascontrol of IO devises, exs. disks, and printers, robots, telephones, space vehicles.

For the most part the languages considered here are **general purpose**, i.e. they are capable of describing any function. However they use suprisingly different vocabularies and grammars to specify those functions.
One emphasizes a **mathematical (functional) approach,** another a **logical approach,** another an **imperative-**do this ten do this..iterarive approach.The functions operate on a variety of data types ex. text, numbers, and control a variety of IO devices.
We will cover a number of general purpose languages based on different approaches to achieving that generality. There has been extensive work, including language description, compilation and interpretation, data-structure development, modulization, etc. on these languages, and many languages of this type.

Some Landmark General Pupose Languages.

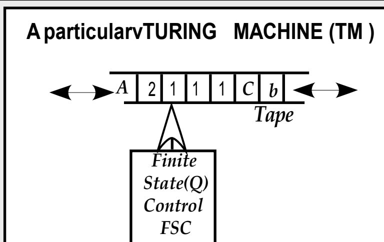
Over time general purpose languages have evolved to make allow programs development to become easier, safer, and more efficient. Usually a new languages is implemented to incorporate one or more new facilities for achieving ease, safety and efficiently.
Any language listed below which is still in use is no longer in its pristine form. Shortly after such a new language appears its conceptual inventions begin to be incorporated in older languages. So most of the l

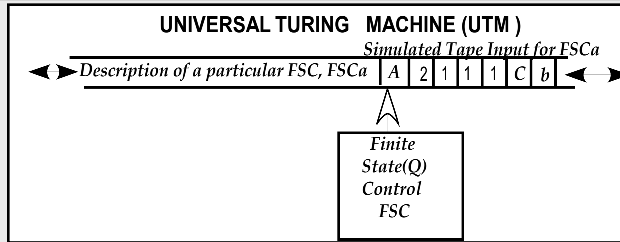
General Purpose Programming Language-Can describe any describable function. Over time almost all have incorporated the following capabilities:

Growing Abstraction, i.e., higher level constructs.

Growing facilities within the Language for defining Abstractions,

Growing facilities for structuring program into seperate modules





1 **Turing Machine, Church Logic, Kleene Recursive Function Theory**
Each capable of computing any computable function (Turing, Church, Markov Thesis) 1936.
2. **Binary or Octal Machine Code** Late 1940's.
3. **Symbolic Assembly Language** Early 1950's

4. **FOR**mula**T**R**AN**slation Design By Implementers**: types, subprograms (modularity and abstraction), formatted input/output).** 1956

5. **CO**mmon **B**usiness **O**riented **L**anguage **Designed By Users (Business)** 1959

6. **LIS**t **P**rocessing **Funtional** Language, uses linked lists, basic structure: list (operators) 1960

7. **A P**rogramming **L**anguage: basic structure **array** - operators &
SNOBOL a string oriented Language (pattern matching) General Purpose but emphasizing a limited class of goals. 1962

8. **ALGO**rithmic **O**riented **L**anguage 60 **(Designed by International users)** CFG **description**, (**block structure,** ***if then else*****, recursio**n. 1963

9. **B**eginners **A**ll purpose **S**ymbolic **I**nstruction **L**anguage Imperative Simple for μ-computer 1965

10. **Pr**ogramming **L**anguage/ **1** interupt or exception handling, concurrency.

11. **SIMULA**tion Language and Simula67 Cl**ass**

12. **ALGOL68** Strong orthogonality 1968

13. **PASCAL** ALGOL type languages Teachability 1969.

14. **PROG**aming with **LOG**ic Declarative Logic 1972 *****declarative

15. **SMALLTALK** Menu driven, mouse, **object oriented** uses generalized class concept 1972

16. **C** High level imperative with access directly to machine (memory etc.) 1972
C++ classes in C 1985

17. **MODULA-2** Pascal + modules, classes, Coroutines, typed procs 1982

18. **ADA** Includes all the good things developed so far 1983

19. **JAVA**

SOME LANDMARK GENERAL PUPOSE LANGUAGES.

© 2006 M C Paull

CLASSIFICATION OF PROGRAMMING LANGUAGES ("Programming Paradigms")

One way of organizing a study of variety is classification. Computer language do fall into a few classes with fundamental differences between them. Each of these classes can be further partitioned on the bases of less basic, but nevertheless significant differences. There is at present no great tree of classification that naturally covers all aspects of languages, but rather a number of criss-crossing dimensions of classification. Some of these are:

1. By the **Application areas f**or which they are favored.
2. By the E**nvironment** in which they are useful
3. By the mathematical model on which they are based.

Classification By Application

By and large each computer language is capable of doing anything any other language (provided the machine has the basic hardware) can do (Turing machine) But each provides easy ways to express some limited set of concepts. These concept are determined by the application intended for the language. So we have:

:

1. **Commercial Languages** (Large Files, Personnel, Accounting, Inventory) CoBOL (Good Data Structures For This Class Of Problems)
2. **Scientific (Mathematical)** ForTran (Good Mathematical Language, Effcient Compilation Available)

3. For **Program Systems Development**
   C (Language Gives Access To How Things Are To Be Done-Pointers). C-Threads With System command additions-ex. fork, etc.
4. For S**tring manipulation SNOBOL**

Special Purpose

5. Script or Command (Communicating With System) J**CL, UNIX. (Special Purpose Language Having Apprpriate commands for that use)**
   UNIX. c
6. **Editors** (Word-Picture Processing)
   SCRIBE
   TEX (All Special Purpose Languages With Appropriate Commands)

I.2.1 **Classification By Environment** Another dimension along which languages range is the environment in which they will be used.

1. **Batch**
2. **Interactive** (Mixed Design and Testing)
3. **Real Time** (Solution Required In Short Time (Control Railroad))

Classification By Mathematical Model

Classification of languages according to application or environment is relatively shallow. The application will mainly effect the syntax, while the environment has little effect on the language except for adding some specialized commands. Part of the reason that some of these languages remain favored is that they have incorporated new language developments, and the fact that a large numbers of programs written in these languages are still in use. In fact <u>almost all languages used today have the same desireable features</u>

© 2006 M C Paull

interesting classificationfollows from the common purpose of all programming languages and the different approaches to its attainment. (see figure 1) That common purpose is to describe how a collection of information or data is to be processed in a form which can be transformed mechanically for implementation by a computer.

1 **PROGRAMMING- MACHINE, FUNCTION, AND LOGIC**

A program may be thought of as a **function applied to arguments formed from its input data, and whose result is the output of the program**. Viewed this way, programming languages can be partitioned according to which of the universal systems developed by mathematicians for defining all defineable functions. These systems include those based on the a) **Turing or Von -Neumann machine**, b) those based on **RecursiveFunction Theory,** c) and those on **Symbolic Logic.**

a) Program As Imperatives (Machine Commands)

Turing described a language for programming a machine which had a finite number of internal states (memory) as well as a infinite or potentially infinite tape for input, output, and intermediate memory. This Turing machine reads one of a finite set of symbols from a tape and depending on the value of that symbol, and the state, moves the tape forward or back and writes to the tape. This dependence is given by a finite set of commands which is the program of the Turing machine. Turing showed that even with these simple means all defineable functions could be constructed. Furthermore there is a Universal Turing Machine with similar construction containig a “Program” on its tape which can simulate the action of any particular Turing Machine.The rudiments of a procedural imperative language are here.

The basic conditional *if,* storage, and *input/outpu*t commands are already present. The development of more practical computers with **memory parcelled into nameable storage words a**nd viewed as variables whose domains could be numbers, strings, or characters etc., and the inclusion in hardware of basiic **arithmetic operations** led to elaborations on the procedural language while still maintaining the basic conception. Values can be read to or from these variables. Functions are performed on these variables and the results are are assigned to other variables, all conditioned by the outcome of these calculations.

The hardware of the modern computer supports these operations in a simple direct form with “machine language“, more sophisticated versions were later developed as higher level languages. The closer the language is to the machine language the more intimately can be the control of the computer.
This is the model for the **procedural-imperative** languages.

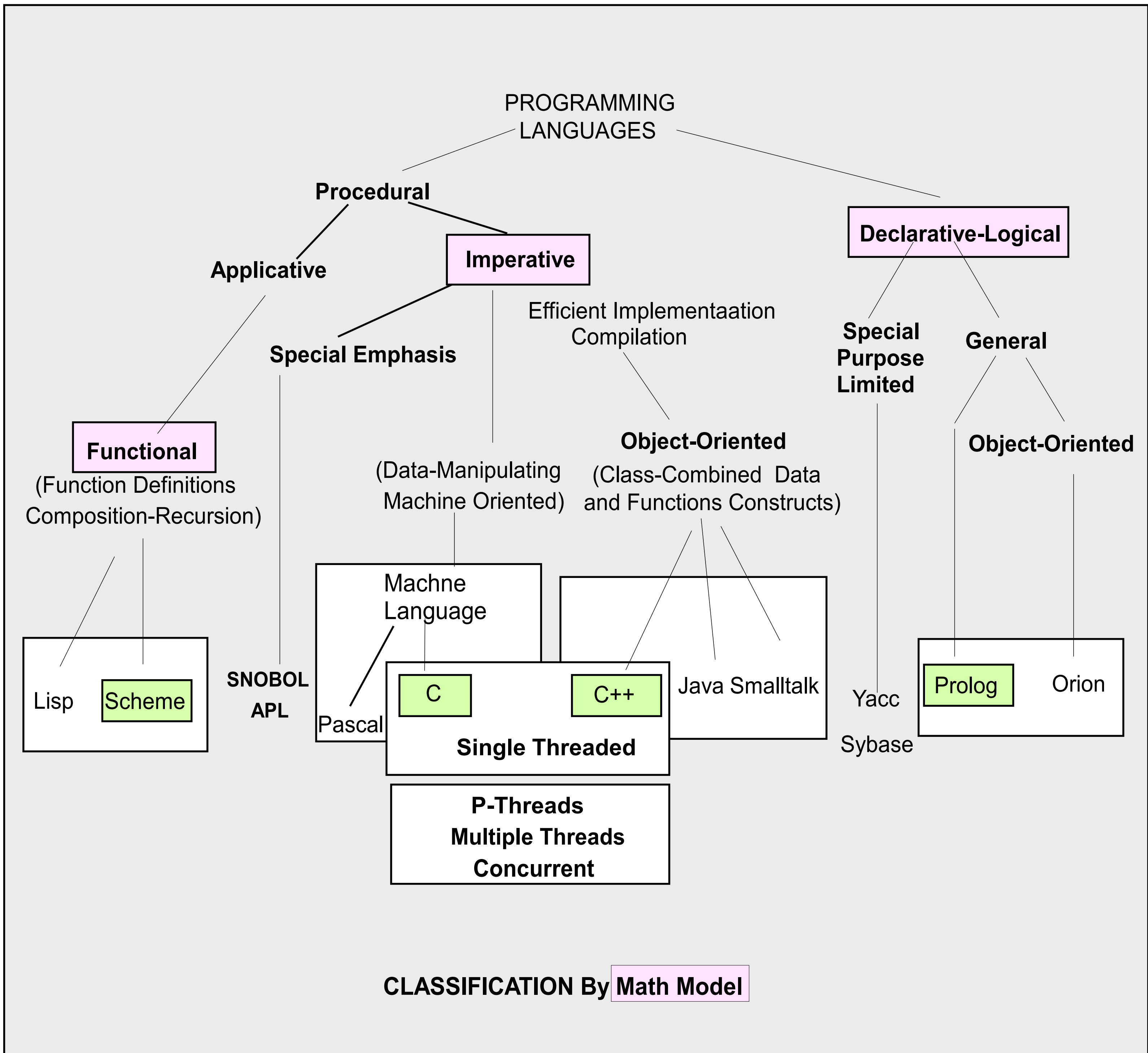
b) Program as a Function

In recursive function theory it is shown that all functions possible in any computer language can be defined starting with a small set of very simple functions provided there is a facility to define new functions using composition, and recursion of a simple kind. This is the model for the **procedural-functional (or applicative)** languages. An entire program is viewed as a function, F. whose arguments are the prorgams inputs and whose result is the program output. Then this functions is defined as the result of the compostion of a series of simpler functions, The simpler functions again are viewed as the composition of still simpler functions etc. This defining of functions in terms of simpler functions continues until the builtin functions of the languageare invoked

c) Program As Logic Assertions Based Queries

Finally using formal logic all defineable functions can be constructed. This leads to a declarative language, in particular a logic language of which **Prolog** is an approximation. More generally declarative languages are designed for stating the problem to be solved, rather than specifying how to solve the problem, as in the procedural languages. Declarative languages are often special purpose. Yacc is such a language. The translation that a compiler is to implement is described, and the compiler for Yacc produces the compiler.

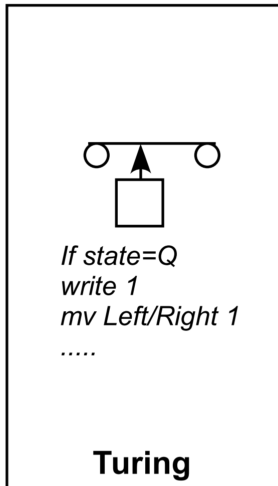
© 2006 M C Paull

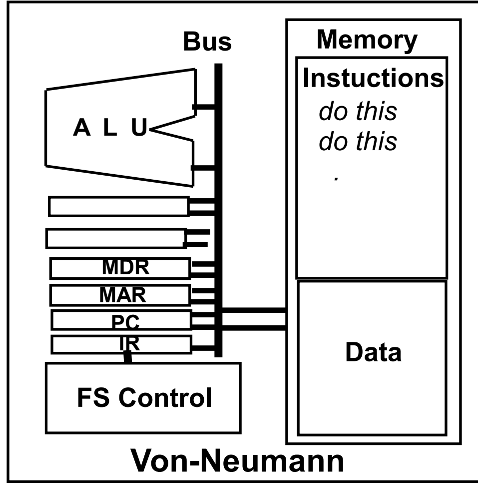


Facilities Important For All General Purpose Languages

Some facilities though not initially found universally, are so valuable that they have become or are becoming universal in general purpose languages. This includes **extensibility** (the facilities for constructing **new data-structures, new functions, macros**, and generally new constructs, which can then be used as though they were the basic ones initially supplied) .

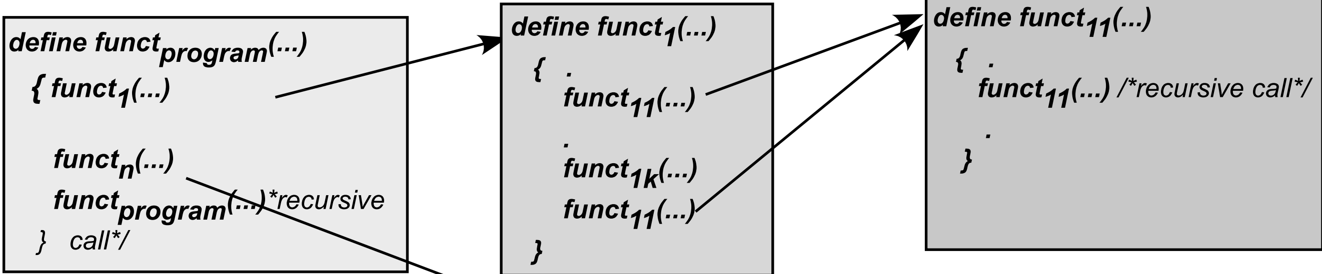
Also the more abstract ability to define new data types is widespread in modern languages. This means one can define a new type of data, ex. an array with array elements,and the functions to be executed on them and then declare many instances of this type. These allows one to tailor the language to ones current application.





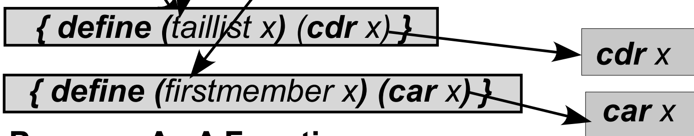

```
while(nextch != EOF)
  { if frand() >= .5 && ((ch=b.leave()) !='\0'))
      putchar(ch);
    else
      if(b.enter(nextch))
        nextch = getchar();
  }
```

Program As A Procedure Of Imperatives



The entire Program is a function defined in terms of simpler functions, with conditionals, some built in and some but, some not yet defined. No *while* , *for* , *repeat*, etc. recursion instead.

{ define (lastmember x)**(cond [(null**?**(** taillist x))**((** firstmember x) **]**
[else (lastmember**(** taillist x**))]) }**



cdr x

car x

Program As A Function

Asserted To Be True Facts (Data Base)

nicer(al,,sue) /al is nicer than sue/
nicer(mary, al) /mary is nicer than sue/
*nicer(X,Z) **:-** nicer((X,Y)**,** nicer(Y,Z) / if nicer(X,Y) & nicer(Y,Z) **then** nicer(X,Z), (“,” = &)/*

Given the Asserted True facts is the following True ? (Queries)

***?-**nicer(al,Z) /al is nicer than Z, what value can Z take?/*
Z=sue

greatereal(X,Y) :- real(X), real(Y), >(X,Y) /real( ) is builtin/
greaterreal(X,Y) :- greatereal(X,Z), greatereal(Z,Y),

?-greatereal(10.0, 5.0)

Program As A Proof

Three Types Of Language, Imperative, Functional, Logical

© 2006 M C Paull

Language constructs for defining abstract data types consisting of encapsulated groups of data structures and functions which operate only on those structures. This facility is found in more and more languages. These allow the object oriented approach to programming, i.e., viewing the process as a collection of interacting abstract data structures-, which,in addition to providing great flexibility, guards against errors that arrise in design of large programs because of faulty communication of programmers. It should be noted that some of these concepts are finding their way in, in fact were originated in, some special purpose languages, i.e, simulation and data base languages.

TRANSLATION AND LANGUAGE“(IMPLEMENTATION“)

Translation, Compilers , Interpreters, Related Language Properties

There are other ways in which languages in the same class differ. Some of these are listed below.

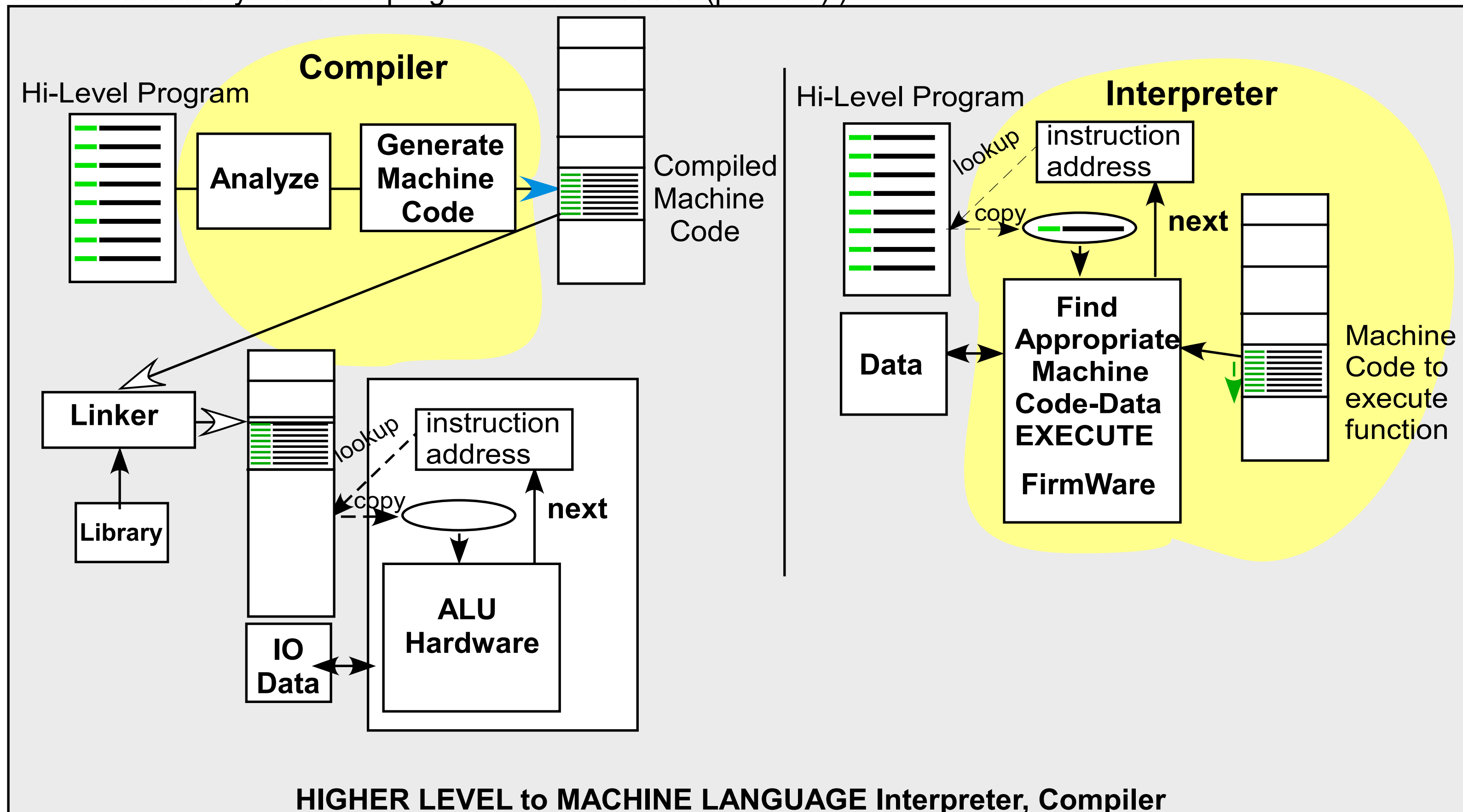
1. All higher-level programming languages need to be processed by the computer in order to produce their results.

For **Declarative language** program, P, it is usually necessary to run a very general algorithm, an Interpreter, (sometimes called inference engine), with P as its input “data to get these results.

A **Procedural language** source program, being modeled on machines, or function definition, can be translated to a program in the language of the machine that it is to run on. Each expression in the Source or input (generally Hi-Level) language generates a number of instructions in the Object (generally Machine) language. For these languages either an **Interpreter** executing the Object code as it generated or a **Compiler** to produce a translation of the entire programs to be run later may be available.

So **Interpreters** hold the Hi-Level language program, scan it and produces and execute the Machine language instructions that implement it line by line. **Compilers** on the other hand produce a complete translation of the entire Hi-Level input program into machine language which may then be run.

Whereas one can quickly run a program if the Source language is **interpreted,** large amounts of storage for both the Interpreter code and the source code must be provided. Interpreters are relatively slow since they cannot take advantage of optimization opportunities provided by looking at the entire program before final translation. Compilers on the other hand, though requiring more processing before producing runnable code, can be designed to produce **efficient Object code.** Constructs to make coding safe, and generate efficient code, etc. ex., types, block structure, pointers, are available when efficiency is important. (They do not necessarily make the programmers life easier (pointers).)



HIGHER LEVEL to MACHINE LANGUAGE Interpreter, Compiler

In fact *some Interpretation is required to run most "Compiled" Code*. That is because, for example, to run *IO functions,* the reads, writes, etc. appearing in the program serve in effect as calling sequences to builtin Operating System procedures which "interpret the reads and writes.

2. Languages need to be described with precision and clarity. There are **Grammars** for achieving precise definitions of **Syntax**, and as well, though in somewhat less satisfactory way, their **Semantics**.. Languages can be easy or hard to read, self-documenting, or diffcult to follow without extensive comments. There are some principles to be considered for accomplishing that end.

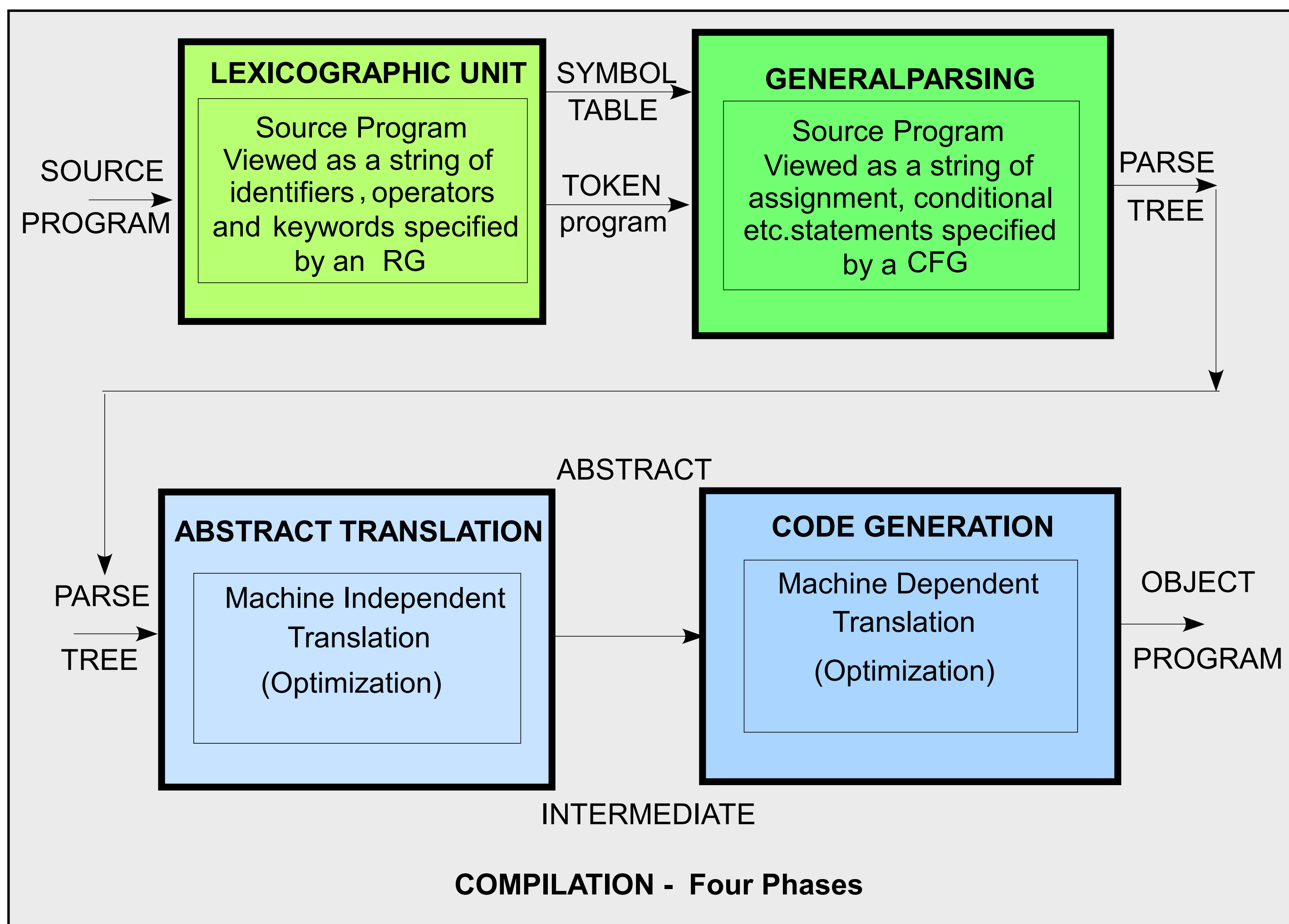
The Compiler Structure

Our **first major topic** will be on formal ways to describe programing languages. These are used directly in guiding the compilation, translation to machine language. So it is appropriate to look a little more closely at theCompiler.

Typically compiler design is broken into two analysis parts: **Lexicographic** (Lex) Unit and General **Parsing**, and two synthesis parts; **Abstract Translation** and **Code Generation** **T**he function of each of the these parts is shown below.

The Lexicographic unit receives the program as written in the language to be compiled. Its purpose is to detect identifiers, operators, reserved words, and punctuation. It then replaces these with compact uniform codes (tokens) which includes pointer to a table for details to be used at a later stage of compilation. The output of Lex is a table, the Symbol table, together with a copy of the input, with some parts compacted(pointing to the Symbol Table, thus considerably shorter than the input

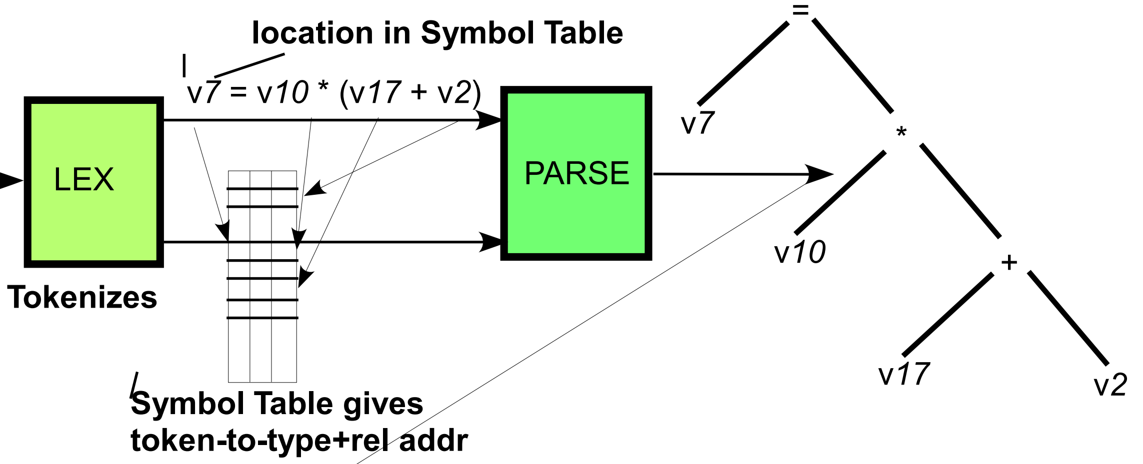
The General Parser analyzes the output of Lex, its algebraic expressions, conditionals, etc., to determine the basic operations that the machine must execute, and the order in which they are to be executed. Its analysis results generally in a "parse tree" which represents each elementary operation



COMPILATION - Four Phases

© 2006 M C Paull

int stat: 5050
.
.
plc = stat * (pos + xx)



vj.r = *relative address of vj
add v7.r, 0
add v7.r, v2.r
mult v7.r, v10.r
2-address

introduced intermediate variables and addresses
v1000 = v17 + v2
v1001 = v10 * v1000
v7 = v1000

ABSTRACT TRANS

OR

CODE GEN

OR

cla v17.r
add v2.r
mult v10.r
sto v7.r
1-address

OPT

v7 = v17 + v2
v7 = v10 * v7

2-address
op a b operation
op has arguments a nd b
result is placed in a

1-address
operations have
accumulator register
as unstated argument

COMPILATION Four Phases EXAMPLE

The Abstract Translation takes the parse which comes from the General Parser and generates abstract code in a generic assembly language, which can be in turn translated into particular code or assembly language. For arithmetic operations a three address assembly language can serve this abstract purpose.

It is the Code Generator turns that abstract code into code for a particular machine. The Code Generator is the only part of the compilation that is machine, but not source program, dependent. All the other parts are source program, but not machine, dependent.

Optimizations can be done in either or both the Abstact Translation and Code Generation parts of the compiler.

Optimization done in the first of these, like removing assignments uneffected by cycling through a loop, i.e., whose right side is unchanged in the loop, from that loop., or replacinga number of common sub-expression instances by a variable to which that common sub expression is assigned, is machine independent. Those done in the second are machine dependent, like using registers for temporary memory when possible, or using exotic machine dependent assembly language constructs to replace a sequence of Abstract Language instructions.

© 2006 M C Paull

BINDINGS

A Program passes from its first symbolic written form to its binary form in the machine. Various decisions about transforming the symbolic form to the ultimate binary form are made at various times in this transformation. Decisions about where and when these "bindings"are to take place are have been made in the Language Design as to where these transformations are to be made as to how and when various " bindings" are to take place. At design timealso the way variables, operations, etc. in the mind of the Programmer are to be bound to symbols was decided.

BINDINGS (Entity ---> Aspect Of Its Ultimate Representation) BINDING TIME (When that Association is made- Execution,or Translation, or Implementation, or Design Time)

EXECUTION TIME:
    Variables(Name) ----> Value (Numerical, Character).
 On Entry To Subroutine or Block:
    Formal Parameters ----> Actual Parameters, Storage Location
 Throughout Execution.
    Variable ---> Values (Assignment Statements)

TRANSLATION (COMPILER) TIME:
Programmer Determined
  Variable ----> Name, Type
Translator Chosen
  Data Object Simple ----> Location, Layout
  Data Object Complex ----> Location, Layout
Loader Chosen
  Program, Subprogram Locations ----> Actual Addresses

LANGUAGE IMPLEMENTATION (RUN) TIME
   Numbers, Integer, Real, ---- Representation

LANGUAGE DEFINITION TIME
   Statements ----> Format, Punctuation

X = X + 10

Variable (X) -----> Allowable Values and/or Structure (Real,Integer,Array, List)Times those are Set.
Constant(10) -----> Allowable Representations must be set
    All Numbers given in decimal notation must be Bound to Binary numbers
Operator(+) ------> Arguments Acceptable, Different meanings for different arguments. Determined
    at Design, Compile or Run Time

Early and Late Bindings. (Fortran-Translation Tmie (Efficiency), Lisp (Execution Time (Flexability-less to specify)).

© 2006 M C Paull

The Role of Recursion

Two of these 3 Languages are heavily dependent on the definition of functions. Such definitions are, in turn,heavily dependent on the use of Recursion. Furthermore function definition and recursion in particular provide a compact way susceptible of formal proof, of defining functions in all of the three languages to be studied. Before looking at examples of the languages a brief introduction to recursive function definition is given.

Recursion-Development Of Sum, Product , Powers, Based on Successor/Predecessor

Fundamental For All Language Types. Definitions:

0 is an (positive) integer. If x is an integer the S(x), the succesor of x is an integer. If x is an integer other than 0 its predecessor S-1(x) is an integer. S((S-1(x)) = S-1(S(x)) = x. We name some integers S(0) = 1, S(1) = 2, S(2) = 3, etc.. Now we define Addition

SUM

x + y = sum(x,y)

sum(x,y) = y if x == 0
sum(x,y) = sum(S^{-1}(x), S(y)) if x != 0

*2 + 2 = sum(2,2) = *sum(1,3) = sum(0,4) = 4 TRUE by Definition*

while equivalent.

```
sum(x, y)
  { while(x > 0)
      {x= x-1; y=y+1}
    return(y);            }
```

PRODUCT

The Product of x and y is the= (y + ...+ y + y + (y + 0)

*x * y = prod(x ,y, 0)*

prod(x,y, s) = 0 if x == 0
prod(x,y, s) = sum(y,s) if x == 1
prod(x,y, s) = prod(S^{-1}(x), y, sum(y, s)) if x > 1

3 x 2 = prod(3,2,0) = prod(2, 2, sum(2,0)) = prod(2, 2, sum(1,1)) =
prod(2, 2, sum(0, 2) = prod(2, 2, 2) = prod(1, 2, sum(2,2)) =
prod(1, 2, sum(1,3)) = prod(1, 2, sum(0,4)) = prod(1, 2, 4) = sum(2,4) = 6

while equivalent.

```
if (x==0)  prod(x, y, s)== 0;
  else { while(x >1)
           {x = x-1; s=sum(y,s);}
            return(sum(y,s));       }
```

POWERS

x^p = exp(p,x,x)

exp(p,x,m,) = 1 if p == 0
exp(p,x,m) = m if p= = 1
exp(p,x,m) = exp(S^{-1}(p), x, prod(x, m, 0)) if p > 1

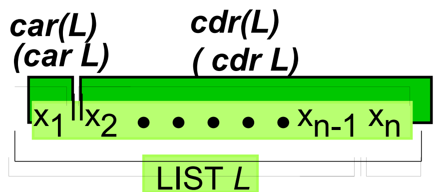
2^3 = exp(3,2,0) = exp(2, 2, prod(2,2,0)) = exp(2, 2, prod(1, 2, sum(2,0)) =
exp(2, 2, prod(1, 2, 2)) = exp(2, 2, sum(2, 2)) = exp(2, 2, 4) = exp(1, 2, prod(2,4|,0)) =
exp(1, 2, prod(1,2, sum(2,4))) = exp(1, 2, prod(1,2, 6)) = exp(1, 2, 8) = 8

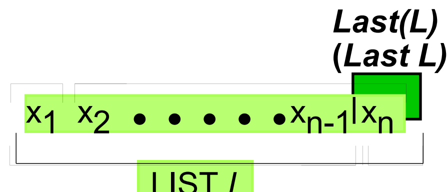
while equivalent.

```
exp(p,x,m)
 { while(p >1)
     {p=p-1; x; prod(x, m, 0;) }
   if (p==1) return m
   else return prod(x, m, 0); }
```

RECURSIVE FUNCTION DEVELOPMENT OF BASIC OPERATIONS

© 2006 M C Paull

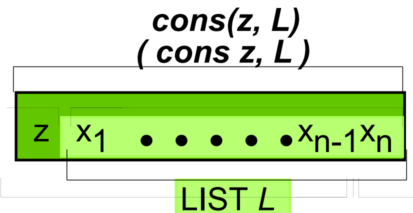


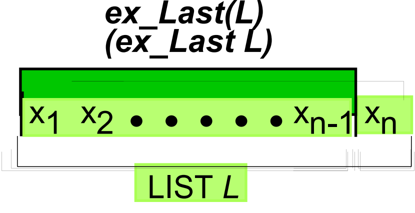


last(L) = ***car****(L)* *last(L) = last(cdr(L))*	*if |L| = 1* *if |L| > 1*	**Math Syntax**
{ define (*last L)****(cond [(null****?(cdr L))* ***(car*** *L)* ***]*** * ***[else*** *(last(* ***cdr*** *L))* ***]) }***		**Scheme Syntax**

last (<a b 21 d33>) = last(<a b 21 d33>)= last(cdr(<a b 21 d33>)) = last(< b 21 d33>) = last(< 21 d33>) = last(< d33>) = car(< d33>) = d33

***last(L)* = (*last L*) = The last member of List L**





ex_last(L) = all except last member of L=If |L| = n, the first |L-1| members of L

ex_last(L) = <> *ex_last(L) = car(L)* *ex_last(L)) = cons(car(L) , ex_last((cdr(L))*	*if |L| = 1 or L = 0* *if |L| = 2* *if |L| > 1*	**Math Syntax**
{[1] ***define (***[2]*ex_last L)*[1]*(*[2]***cond [***[3]***(***[4]***null?*** *L)*[3] *'())]*[2] * ***[***[3]***(***[4]***null?(***[5]*cdr L)*[4]*)*[3] *'()]*[2] ***[***[3] ***else*** ***(***[4]***cons (***[5]***car L)***[5]*(*[4]*ex_last (*[6]***cdr*** *L)*[5]*)*[4]*)*[3] ***]***[2] ***)***[1] ***}***[0]		**Scheme Syntax**

ex_last_member(<a b 21 d33>)	*= cons(car (<a b 21 d33>), ex_last(cdr(< a b 21 d33>))=*
cons(a ,ex_last(cdr(< b 21 d33>))	*= cons(a ,cons(b, ex_las(cdr(< 21 d33>))) =*
cons(a ,cons(b, cons(21,ex_last(cdr(< d33>))))	*= cons(a ,cons(b, cons(21, cons(d33,ex_last(< >)))) =*
cons(a ,cons(b, cons(21, cons(d33, (< >)))))	*= <a, b, 21m d33>*

***ex_last(L)* = (*ex_last L*) = The List consisting of the first through next-to-last member of L**

* I have different kinds of brackets, i.e., [..], {..}, (..) for some clarity-but they should all be (..)

COMPARING SYNTAXs FOR RECURSION (Math and Scheme)

III **Examples Of Languages To Be Studied**

To aid in better understanding the significant features of languages we will introduce you to a number of languages which span the range of significant concepts found in todays computer languages.

A sampling of programs written in these languages is giiven in the following pages..

Functional Language

Lisp is the grand-daddy of such languages and is closely related to the lambda-calculus, a system in recursive function theory, for describing any defineable function. There is one basic data-structure used throughout, namely the **list. Scheme** is a version of Lisp which is more consistent in some respects than Lisp, making it suitable for academic study. Basically one defines functions on the data structure which then form parts of other higher level definitions--**recursion is critical**-there are no whiles or other iterative structures.

if LIST x = x_1 x_2 • • • • • x_{n-1} x_n then

(*reverse* x) **=** x_n x_{n-1} • • • • • x_2 x_1

{ define *(reverse L)* *(* ***cond*** *[(****null****? L) ()]*
*[****else*** *(****cons*** *(lastmember L) (reverse(headlist L)))])* ***}***

An Alternative Definition of reverse:

{ define ***(****reverse L)* *(* ***cond*** *[(****null****? L) ()]*
[***else*** *(****append*** *(reverse (taillist L)) (* ***lis****t (firstmember L))])* ***}***

In recursion we assume the function to be defined for list L has already been defined for a sublist of L and we give an explicit result for a size 1 list.

Definitions Of *reverse* **Function-Recursion**

EXAMPLE: FUNCTIONAL-SCHEME

Strict Declarative

Here relations asserted to be true are listed in a “Data Base”. These relations involve variables (start with uppercase letter) and constants numbers and words starting with lower case letters. The form of the simplest of these assertions is *relation(list of variables and constants).* The relation is a word starting with lower case letter) . So *bigger(6,,5)*. provides a simple example of a Data Base. This may be viewed as defining a function *bigger(X,Y),* All that is known about it is that if we call *bigger(6,5)* the return is *yes*. If we call *bigger(X,5)* that X must be 6. if we call bigger(X,Y) that X must be 6 and Y must be 5..We can think of *bigger(X,Y)* as defining a function call that returns a value to X and Y-but the only values it can return are those we have told it are legitimate in our Data Base. If now we add, bigger(7,6) to the Data Base and also *bigger(X,Y) :- bigger(X,Z) , bigger(Z,W)* (the comma is read “&”). Then if the constants cX, cY, and cZ appear in the Data Base in the contexts bigger(cX,cZ) and bigger(cZ,cW) , then the query bigger(cX,CY) will be return true.

Data Base Consists of Relations

FORM: <relation-name><(Cap start, and non-cap start) arguments>

variable **constant**

Like <relation-name>(X, Y, 3, ...). You choose relation-name(There are some **built-ins**)

EXAMPLE:

nicer(al,,sue) *meaning al is nicer than sue*

nicer(mary, al) *meaning mary i*

Data Base Also Contains Implications In Things Related

EXAMPLE:

nicer(X,Z) :- nicer((X,Y), nicer(Y,Z) *meaning if nicer(X,Y), nicer(Y,Z)* ***then*** *nicer(X,Z), (“,” is read as and)*

Querys Call and Response

FORM: <relation-name><(Cap start, and non-cap start), arguments> **Question Mark**

Like? -<relation-name>(X, Y, 3, ...), Can also be and (,) of relations

EXAMPLES:

?- nicer(X,sue)

X = al

X = mary

?-nicer(al,Z)?

Z=sue

?-nicer(mary,sue)

True

?-nicer(sue,mary)

False

?-nicer(X,Y)?

X=al Y=sue

X=mary, Y=al

X=mary, Y=suel

COMPONENTS OF PROLOG PROGRAM

EXAMPLE: DECLARATIVE-PROLOG: DEFINITIONS AND QUERY CALLS and RESPONSE

III.1. Procedural-Imperative + **Object-Oriented Language [C++]** Here the basic jobs to be done are formulated with typical Procedural-imperative commands like ***whiles if..then..elses***, procedures defs., etc.

In addition, with an Object-Oriented language one can make data-structure the central concept in a language. Here an abstract data-structure consists of a collection of data types plus operations on that collection. i.e. a "graph" together with operations like degree-of-vertex(x), neighbor-of- vertex(x). In a program. More than one such graph may be needed, each with operations confined to its vertices and edges. So means are provided to create a graph template with which to declare any variable(s) to be a graph. This idea is incorporated in the object-oriented languages by definiting a \class", which is a template of data and operations{a variable can be declared to be an instance of the \class". A construct which is declared a member of a class is called an object. Also important in such a language are relations between classes-two may contain the same function or one may contain a sub-data construct or a subfunction of another. Language is needed to express these commonalities to help detect optimizations.

Now a program can be thought of as consisting of a set of Objects, each of whose data can be individually manipulated by its associated functions (or messages-in Smalltalk). A class can be built from other classes like sub-routines. There can be communication between classes through data accesible to both.

```
const int MAXBUF = 4              "MAXBUF=4"
class buffer                      "define class buffer"
 { public
   buffer()  {size=MAXBUF+1; front=rear=0;}   "initialization"Data Structure buffer
   int     enter(char);                       "name of public function on Data Structure"
   char    leave();                           "name of public function"
  private
   char buf [MAXBUF+1];                       "private character array"
   int size, front, rear;                     "private integer variable"
   int succ(int i) {return (i+1) % size;}     "private function definition-add 1 mod size of buffer"
}
   int    buffer: : enter(char x)             "defining public function"buffer:enter
          { if ( succ(rear) == front) return 0 ;   "if rear+1/%size = front return 0 (~enter)else
                                                      enter return 1
             buf[rear]=x; rear = succ(rear);    "rear always is at empty entry"
             return 1;
             }
   char   buffer: :leave()                     " defining public function leave"
          { if (rear == front) return '\0';    "leave at front -- if front = rear return end of string'0/'"
   int      x = buf[front]; front = succ(front);
            return 1;
            }
```

Main Program: Gets characters from input into buffer and print characters from buffer to output, each with probability 1/2(.5) . But makes sure *rear* always points to an empty cell in buffer and *leave* never removes an empty cell .(frand is a random number generator)

```
#include <stdio.h>                    "bring in IO functions, constants"
#include <math.h>                     "bring in random generator (frand)
int     main()                        "defining function main"
        { buffer b;                   "declaring b to be a buffer"
 int    ch, nextch = getchar();       "As long as no EOF
         while(nextch != EOF)          -with prob .5: if next char != eof&rear != front putchar  from buffer
           { if frand() >= .5 && ((ch = b.leave()) != '\0'))    else
                           putchar(ch);                        with prob .5 if rear +1 != front  enter char into
             else if(b.enter(nextch)) nextch = getchar(); }    buffer & getchar
          while((ch =b.leave()) != '\0') { putchar(ch);  }     If EOF then empty buffer till EOF detected
        return 0; }
```

EXAMPLE: PROCEDURAL-IMPERATIVE-C++

© 2006 M C Paull

III.2. Procedural-Imperative + Multi-Tthreaded [C Threads]

Here again the basic jobs to be done within a thread (process) are formulated with typical Procedural-Imperative commands: *ifs, while*s etc., but being multi-threaded means there is more than 1 thread that is running, effectively, at the same time. This introduces new problems. If two processes want to both use the same variables to communicate. A means to guarantee that they don't interfere with each other is required. This is the **mutex-lock If it is l**ocked any othercall will block.

```
void producer_funct(void);
void consumer_funct(void);

char    buffer;                                   Global (Shared By ll Threads)
int     buffer_contents = 0;
pthread_mutex_t mutex;                            mutex is type mutex (values 1 and 0) To make
                                                      sure only one process rums at a time
struct timespec   delay;                          delay is a struct of type timespec

main()                                            main is a thread (runs the producer)
{  pthread_t consumer;                            consumer is a thread
   delay. tv_spec_sec = 2;                        size of delays in secs + nsecs.
   delay tv_nsec = 0;

    pthread_mutex_init(&mutex, pthread_mutexattr_default);    Initialize mutex=1
                                                              no mutex attri in many versions

    pthread_create( &consumer,                    create a thread with procedure consumer
                    pthread_attr_default,         thread attribute
                    (void*)&consumer_funct,       thread consumer executes cunsumer_funct
                    NULL                          no arguments passed to producer_fundct
                  );
    producer_funct();                             execute producer-funct in thread main
}

 void producer_funct();
  { while(1)
      {  pthread_mutex_lock( &mutex);             if mutex =1 block, if = 0 continue
          if (buffer_contents != N)               test buffer: not empty? if so fill
            {buffer = new_item();  buffer_contents=buffer_contents+1;}
         pthread_mutex_unlock( &mutex);           mutex = 1
         pthread_delay_np( &delay);               without delay-"spin-lock"
       }
   }

void consumer_funct()
 {
  { while(1)
      {  pthread_mutex_lock(&mutex);              if mutex =1 block, if = 0 continue
           if (buffer_contents != 0)              test buffer: not empty
             {consume(buffer_item();  buffer_contents=buffer_contents-1}  t
         pthread_mutex_unlock( &mutex);              mutex = 1
         pthread_delay_np( &delay);
        }      Define Seperate Procedures-will become Sepersate Threads
  }

  pthread_mutex_destroy(); destroys mutexes
```

EXAMPLE: CONCURRENT-PTHREADS

Comparison Of Language Types [“Choice Of Language”]
With the differences in these three languages illustrated above we can begin to see why one might choose amongst them. The given problem, is one, but speed of programming, whether the program must run efficiently, the amount of time allowed for programming, the range of results one needs, the expected longevity of the program, are often the determinants

Procedural vs Declarative
Declarative languages are generally of a higher level (more abstract) than Procedural ones. One only need state the problem precisely to get its solution. As a direct consequence of the greater abstractness Declarative languages generally run much slower than their Procedural kin because they involve an algorithm of great generality, thus slow, in order transform a problem, one of an unlimited number of problems, into an algorithm that solves that problem. In order to avoid this large toll the general algorithm is limited unless notified by the problem specifier to remove the limitation at a specific point in the specification. Such notifications facilities require the user to understand the general algorithm embedded in the compiler, and to be able to nudge the language toward the procedural.

Strict Functional
Given a few basic data types and builtin functions and given the facility for defining functions in terms of built-ins or interms of previously defined ones, one can define any function (pg 11). The builtins can be based on simple arithmetic, or simple arithmetic and basic operaions on lists.. (as in Scheme). Also decisions on these data types must be possible. Generally in these languages each function call returns a value. So a function call can bean argument. There are no repeating iterative commands like *whiles* of *for*s, the repetetive facility is provided by recursion-functions calling themselves. These are languages usually Interpreted rather than compiled making optimization difficult. They are not intended for production applications

Applicative-Functional
Applicative languages are useful when we are dealing with a small number of instance of a small number of data types, while at the same time there are function that are to be applied to a number of these data types. In the applicative languages subroutines use is common, and can be implememted effciently. These languages are useful for algorithm centered programs and are normally compiled so optimization is feasible. A shortest path problem, a program for solving sets of equations are examples.

Object-oriented languages
On the other hand are particularly useful when there are to be many instances of each data type and the operation on the difierent data types are distinct. In the extreme form of object oriented languages when a subroutine is applicable to more than one data type it must be repeated in each. In Smalltalk, and C++ there is a way in which one data type can “inherit“ a function from another data type. This mars the simplicity of the simple scheme in order to make greater use of a single function definition.

I

The program is to open bank accounts, make some transactions (deposits-withdrawals, and to keep a running balance.

PROLOG

```
open(1,51).
open(2,46).
balance(A,I,C) :- open(A,X),cum(A,I,Z),C is X + Z.
cum(A0,0).
cum(A,I,X) :- W is I - 1,cum(A,W,Y),trans(A,I,V), X is Y+V.
trans(1,1,-3).
trans(1,2,11).
trans(2,1,-17).
trans(2,2,22).
```

FUNCTION or Relation DEFINITIONS

```
? - balance(1,2,C)
C = 59
```

FUNCTION or Relation CALLS

INTERPRETATIONS:

open(1,51).

account *1 is open*ed with 51dollars

----

balance(A,I,C) :- open(A,X),cum(A,I,Z),C is X + Z.

balance **in account *A* after *I* transactions is *C*, IF account *A is open*ed with *X* dollars**,&

account A has accumulated Z dollars after I transactions & **where *C is X + Y.***

----

cum(A0,0). ***account A has accumulated 0 dollars after 0 transactions***

----

cum(A,I,X) :- W is I - 1,cum(A,W,Y),trans(A,I,V), X is Y+V

account A has accumulated X dollars after I transactions IF

after W (W = I-1) transactions, account A has accumulated Y dollars &

account A transferred V (+ or -) dollars on its I'th transaction, & *where X is Y + V*

SCHEME

```
( define (open A) ( cond ((eq? A 1) 51)
                         (else 2) ) )
( define (balance A I) (+ (open A)   (cum A I)) )
( define (cum A I) ( cond ((eq? I 0) 0)
                          (else
                          (+  (cum A (- I 1)) (  (trans A I))) ) )
( define (trans A B) ( cond ((eq? A 1) (trans1 B))
                            (else        (trans2 B)) ) )
( define (trans1 B) ( cond ((eq? B 1) -3)
                           (else      11) ) )
( define (trans2 B) ( cond ((eq? B 1) -17)
                           (else        22) ) )
```

FUNCTION DEFINITIONS

```
(balance 1 2)
  59
```

FUNCTION CALLS

Recursion is critical for both **Prolog and Scheme.**

(define (cum A I) (cond ((eq? I 0) 0 (else (+ (cum A (- I 1)) (trans A I))

Interpretation	Comparable Prolog.
(cum A,I) = 0 if I = 0:	*:- cum(A,0,[0])* |
(cum(A,I)=(cum(A,I-1)+trans(A,I)	*:- W is I-1,cum(A,W,Y),trans(A,I,V),Xis Y+*

© 2006 M C Paull

C++

```
    include<stream.h>
class checkbook
  {
   float balance = 0;
   int acctno;
   int top;
   public:
   checkbook(int no, float init_deposit)
     {
      acctno = no;
      balance = init_deposit;
      cout << "\nWELCOME ACCOUNT " << acctno;
      cout << " with initial deposit of "<< init_deposit << "\n\n" ;
      }
    void deposit(float amount)
     {
      balance = balance + amount;
      cout << " customer " << acctno << " has deposited " << amount << "\n";
     }
    void withdraw(float amount);
    void report_balance(void);
   };
  void checkbook::withdraw(float amount)
           {
            balance = balance - amount;
            cout << " customer " << acctno << " has withdrawn " << amount << "\n";
            }
   void checkbook::report_balance(void)
            {
             cout << " customer " << acctno <<" has a balance of " << balance << "\n";
             }
```

CLASS any checkbook

FUNCTION DEFINITIONS

Code	Output
main()	
{ checkbook sam(1,51.5);	WELCOME ACCOUNT 1 wiith intial deposit of 51.5
checkbook mary(2,200);	WELCOME ACCOUNT 2 wiith intial deposit of 200

FUNCTION CALLS

Code	Output
sam.withdraw(5);	customer 1 has withdrawn 5
sam.report_balance()	customer 1 has a balance of 46.5
mary.report_balance();	customer 2 has a balance of 200
mary.deposit(200);	customer 2 has deposited 200
sam.withdraw(5.23);	customer 1 has withdrawn 5.23
sam.deposit(100);	customer 1 has deposited 100
mary.withdraw(5);	customer 2 has withdrawn 5
sam.report_balance();	customer 1 has a balance of 141.270
mary.report_balance();	customer 2 has a balance of 395
mary.withdraw(5);	customer 2 has withdrawn 5
mary.report_balance();	customer 2 has a balance of 395
};	

Trace

C++ Banking

© 2006 M C Paull

Implementation Of Declarative Language The Inference Engine

These are Languages with simple ways of declaring certain facts and implications to be true. This forms the Data Base. Also one can express queries, statements expected to be inferrable from the information in the Data Base. The response may be unproveable, true, or give variable values which make the queries true. To respond to such queries a powerful theorem prover, Inference Engine (I.E.), that can prove the query based on the facts in the Data Base is normally incorporated in an Interpreter. The following example is intended to give an indication of the process followed by the Inference Engine. In fact though it is possible to built an infallible I.E. it would be much to large and inefficient to bepractical. The one in Prolog our example language is only an approximation requiring some help, not theoretically necessary, from the programmer.

DATA BASE

*open(1, **51**).* Open account 1 with 51

A syntactically improper way (functional paraphrase) to say it is :*openedwith(1) = 51..*

*open(2, **46**).* Open account 2 with 46

balance(A, I, C) **:-** *open(A, X)***,** *cum(A, I, Z)***,** *C* **is** *X* **+** *Z.* **............1**

Horn Clause[If
- Account *A* was opened (*open*) with *X* **&**
- If the accumulated(*cum*) value of account *A* after *I* transactions was *Z* **&**
- if *C* =(*is*) *X* + *Z* **then (implies)** (*:-*)
- The balance (*balance*) in account *A*, after *I* transactions is *C*.

functional paraphrase *balance(A,I)* **=** *openedwith(A)* **+** *cumulated(A, I)*]

*cum(A,**0**,**0**).*

cum(A,I,X) **:-** *W* is *I* **- 1,** *cum(A,W,Y)***,** *trans(A, I, V)***,** *X* **is** *Y* **+** *V.*

Horn Clause[if
- *W* = *I* **-** *1* **&**
- If the accumulated (*cum*) value of account *A* after *W* (= *I*-1) transactions was *Y* **&**
- if the *Ith* transaction (*trans*) of *A* was V **&**
- *X* =(*is*) *Y* + *V* **then (implies)**
- The accumulated(*cum*) value of *A* after I transactions is *X*

functional paraphrase: *cumulated(A,I) = cumulated(A, I-1) + transaction(A, I)*],

trans(1, 1, -3). Transaction 1 for account 1 is -3

trans(1, 2, 11). Transaction 1 for account 1 is -3

trans(1, 3, -17). Transaction 1 for account 1 is -17

Simple Query

? *open(1,X)*

open(1,51)

X = 51

QUERY

?- *balance(1,0,C)*

INFERENCE ENGINE

so try A=1, I=0, and substituting in 1*:*

CAF,unify **?**

balance(1, 0, C) *The query try to match with atomic formula, then with Clause Head*

balance(1, 0, C) **:-** *open(1, X)***,** *cum(1,0, Z)***,** *C* **is** *X* **+** *Z.* working on 1st clause on the rightside:

Cls: *open(1, 51).* so try *X = 51* and further substituting in 1

balance(1, 0, C) **:-** *open(1, 51)***,** *cum(1, 0, Z)***,** *C* **is** *51* **+** *Z.* working on 2nd clause on the right side

*cum(A,**0**,**0**).* still A=1 now try also *Z = 0*

balance(1, 0, C) **:-** *open(1, 51) , cum(1, 0, 0), C* **is** *51* **+** *0.* further substitution for *C* gives

balance(1, 0, 51) **:-** *open(1, 51), cum(1, 0, 0), C* **is** *51* so *balance(1,0,C)* is true with

C = 51

?- *balance(1,2,C)*

C = 59

Prolog Inference Engine Example Banking

© 2006 M C Paull

```
weather(monday,fair)
    functional paraphrase: weather(monday) = fair
weather(tuesday,overcast).
weather(wednesday,fair).
weather(thursday,fair).
weather(friday,rain).
weather(saturday,rain).
weather(sunday,fair).
previous(monday,tuesday).
    functional paraphrase:   previous(monday) = tuesday..
                                                          DATA
                                                          BASE
previous(wednesday,thursday).
          .
previous(sunday,monday).
color(sky,blue,Day) :- weather(Day,fair).
happy(birders,Day) :- weather(Day,fair), active(birds,Day).   .............1
                 If on the day = Day the weather is fair &  if on day = Day  birds are active then ( :- )
                 birders are happy on day = Day.
   functional paraphrase: daycolor(sky,blue) = dayweather(fair): if (dayweather(fair)== dayactive(birds))
  happy(birders,Day) :- observed(rarebird,Day).
                 If on day = Day a rarebird is observed & then
                 birders are happy on day = Day.
active(birds,Today) :- previous(Day,Today)), weather(Day,fair). ............2
                 day = Day is the day previous to Today  &
                 the weather is fair on day = Day then
                 birds are active Today.
   functional paraphrase:  dayactive(birds) = previousday( dayweather(fair))
observed(rarebird,tuesday).
observed(rarebird,thursday).
```

```
? - happy(birders,When).
```

The set of logic statement above , excluding the query, is called the data-base (DB). All implications and predicates in DB are assummed to be true.:

Example Run: Start with query: *happy(birders,When).*

Match Cls head

happy(birders,When) is true if *weather(When,fair), & active(birds,When) are true*
by 1 in DB with Day <-- When

Match Atomic Forms:

```
1  weather(wednesday,fair). so  When = wednesday will be tried but will fail  because
        active(birds,wednesday):- previous(Day,wednesday)), weather(Day,fair)
                                 = previous(tuesday,wednesday)), weathe(tuesdayy,fair)
                                   but weather(tuesday, overcast) Fail
2  weather(thursday,fair)      so  When = thursday  is possible.  So unify and show that
                                   active(bird,thursday) is true if possible
                                   because from the DB Match Cls Head
        active(birds,thursday) :- previous(Day,thursday), weathe(Day,fair). by 2 in DB
    We know                       previous(wednesday,thursday) is true because it is in the DB unify
                            and  weather(wednesday,fair) is also in DB.
  Therefore we conclude that
  When (in the query) = thursday For similar reasons we also get  When = monday
                                                                  When = tuesday
                                                                  When = thursday
```

Prolog Inference Engine Example Weather Activities

© 2006 M C Paull

```
*THE PATTERN*                                   {COMMENT}
            &ANCHOR = 1                         {START SCAN AT FIRST CHARACTER}
            V   =  ANY('XYZ')                   {V IS X OR Y OR Z}
            AS =  ANY('+_')
            MD = ANY('+_')
            FAC = V | '(' *EXP ')'              {*EXP POSTPONE  EVALUATION FOR AN EXP}
            TERM = FAC | *TERM MD FAC
            EXP = AS TERM | *TERM |
+                 *EXP AS TERM
*THE PROGRAM*
 LOOP      STRING TRIM(INPUT)                               :F(END)
           STRING    EXP RPOS(0)                            :F(NOGOOD)
           OUTPUT = STRING 'IS AN EXPRESSION'               :(LOOP)
NOGOOD  OUTPUT = STRING 'IS NOT AN ESPRESSION' :(LOOP)
```

SNOBOL PROGRAM

SNOBOL EXAMPLE