# 198:314, Sects1-3
# Principles of Programming Languages
# Fall 2001

**Prof. Barbara G. Ryder**
**Core 311, 732-445-3699**
**ryder@cs.rutgers.edu**
**http://www.cs.rutgers.edu/~ryder**



# Introduction

- **Administrivia**
- **Why study PLs?**
- **Paradigms: imperative, functional, object-oriented, logic**
- **History of PLs**

# 198:314 Fall 2001

- Class webpage
  - http://remus.rutgers.edu/cs314
  - Look at for course information, rules, grading, important dates, etc.
  - Read policy on academic dishonesty
  - Co-ordinated class, all programming projects, book homeworks and tests are same across all lectures
- MW5 Lecture webpage
  - http://remus.rutgers.edu/cs314/f2001/ryder/
  - Lecture notes available online in pdf
  - Only print out 2-up or 4-up to save paper
  - Recitation attendance and participation *counts* in your final grade
- Four programming projects posted and submitted for grading electronically



# Syllabus

- Introduction
- Formal Languages - RE's, FSA's
- Logic Programming (Prolog)
- Names and Binding
- Imperative Programming (C )
- Block Structure
- Object-oriented Programming (C++)
- Types
- Functional Programming (Scheme)
- Formal Languages - Grammars
- Concurrency

# Course Goals

- To gain understanding of basic structures of programming languages
  - Types, control structures, naming conventions
- To study different language paradigms
  - To ensure an appropriate language is selected for a task
  - Object-oriented, functional, imperative, logic
- To make learning new programming languages easier by knowing shared features



# What is a programming language?

"a language intended for use by a person to express a process by which a computer can solve a problem" -Hope and Jipping

"a set of conventions for communicating an algorithm" - E. Horowitz

" the art of programming is the art of organizing complexity" - Dijkstra, 1972

# Why learn more than one PL?

- Each language paradigm encourages thinking about a problem in a particular manner
  - Finding a natural match between problem and PL
- Somewhat different functionality supplied by different paradigms
- Computer professionals must be multi-lingual
  - PLs change over time as computer architecture changes
  - Specific applications sometimes result in specialized PLs
  - Need to understand each PLs functionality and limitations



# Imperative Paradigm

- Underlying notion of an abstract machine
  - Von Neumann architecture
    - Store (memory)
    - Accumulator (ALU)
    - Load/store into memory
  - Key operation: assignment

# Imperative PLs

Sum up twice each number from 1 to N.

**Fortran**

```
      SUM = 0
      DO 11 K=1,N
      SUM = SUM + 2*K
11    CONTINUE
```

**C**

```
sum = 0;
for (k = 1; k <= n; ++k)
    sum += 2*k;
```

**Pascal**

```
sum := 0;
for k := 1 to n do
    sum := sum + 2*k;
```



# Functional Paradigm

- **Process of problem solution expressed as a sequence of operations on the data**
  - (Pure) value binding through parameter passing
  - No store accessible through names
  - No iteration
  - Key operation: function application (with recursion)

## Functional Paradigm

**Scheme**

```
(define (sum n)
   (if (= n 0)
      0
      (+ (* n 2) (sum (- n 1)))
   )
)

(sum 4) evaluates to 20
```



## Logic Paradigm

- **Program is a formal description of characteristics required of a problem solution**
  - **Programs tell *what should be* not *how to make it so***
  - **Solutions through a reasoning process called theorem proving**
  - **Key operation: unification**

# Logic Paradigm

```
sum(0,0).
sum(N,S) :- NN is N - 1,
            sum(NN, SS),
            S is N * 2 + SS.
```

**Prolog**

```
?- sum(1,2).
yes
?- sum (2,4).
no
?-sum(20,S).
S = 420
?-sum (X,Y).
X = 0 = Y
```



# Object-oriented Paradigm

- **Organizes a program to be operations on abstract representations of the data**
  - **Objects with data abstraction and information hiding**
    - **Object implementation is hidden from user**
  - **Actions performed on objects (messages)**
  - **Can combine with imperative or functional paradigm easily**
  - **Key operation: message passing**

# Object-oriented Paradigm

**Java**

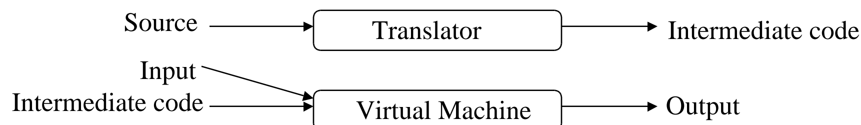
```
class intSet : public Set
{ public: intSet() { }
//inherits Set add_element(), Set del_element()
//from Set class, defined as a set of Objects
  public int sum( ){
      int s = 0;
      SetEnumeration e = new SetEnumeration(this);
      while (e.hasMoreElements()) do
      { s =s + ((Integer)e. nextElement()).intValue(); }
      return s;
  }
}
```



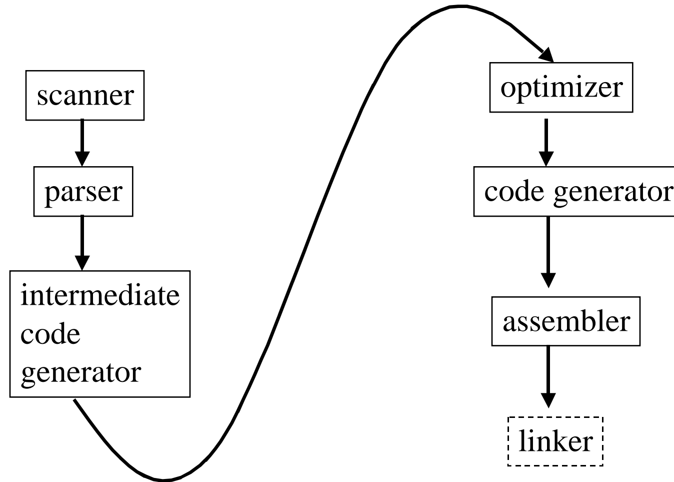
# Translation

- ***Compilation*: translation of a program written in a high-level PL into a form that is executable on the machine**
- ***Interpretation*: a program is translated and executed one statement at a time**
- **Most PL systems are a mixture of these two**
  - **Interpreted: Java, Scheme, Prolog**
  - **Compiled: Fortran, C, C++**

*Scott, p 10*



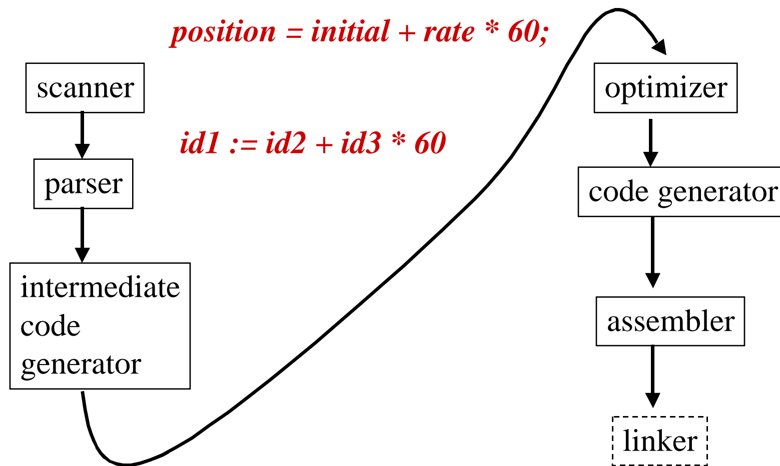
# Compilation




Introduction 1, CS314 Fall01, BGR

17

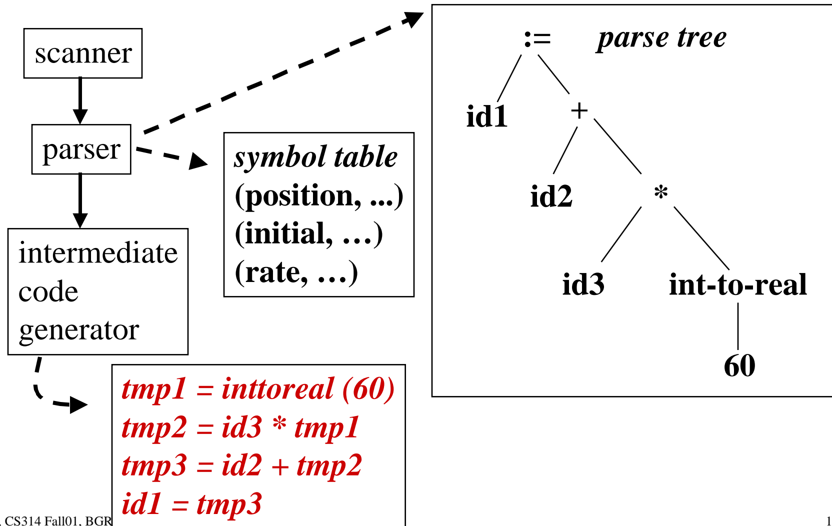
# Compilation




Introduction 1, CS314 Fall01, BGR

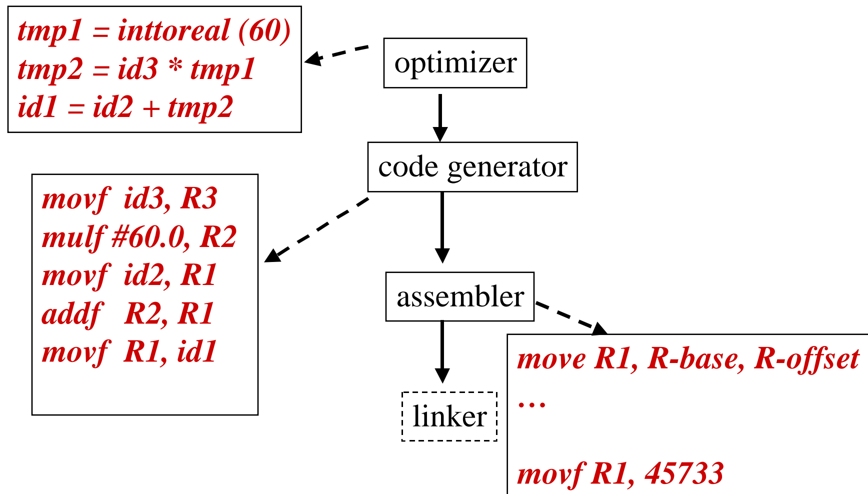
18

# Compilation






# Compilation



## Desiderata for PL Design

- **Readability**
  - comments, names, (…) syntax
- **Simple to learn**
- **Orthogonal**
  - small number of concepts combine regularly and systematically (without exceptions)
- **Portability**
  - language standardization
- **Abstraction**
  - data and control abstractions



## History of PLs

- **Prehistory**
  - 300 B.C. Greece, Euclid invented the greatest common divisor algorithm - *oldest known algorithm*
  - ~1820-1850 England, Charles Babbage invented 2 mechanical computational devices
    - difference engine
    - analytical engine
    - Countess Ada Augusta of Lovelace, *first computer programmer*
  - Precursors to modern machines
    - 1940's United States, ENIAC developed to calculate trajectories

## History of PLs - 2

- **1950's United States, first high-level PLs invented**
  - ***Fortran*** **1954-57, John Backus (IBM on 704) designed for numerical scientific computation**
    - **fixed format for punched cards**
    - **implicit typing**
    - **only counting loops, if test versus zero**
    - **only numerical data**
    - **1957 optimizing Fortran compiler translates into code as efficient as hand-coded**



## History of PLs - 3

- ***Algol60*** **1958-60, designed by international committee for numerical scientific computation [Fortran]**
  - **block structure with lexical scope**
  - **free format, reserved words**
  - **while loops, recursion**
  - **explicit types**
  - **BNF developed for formal syntax definition**
- ***Cobol*** **1959-60, designed by committee in US, manufacturers and DoD for business data processing**
  - **records**
  - **focus on file handling**
  - **English-like syntax**

## History of PLs - 4

- *APL* 1956-60 Ken Iverson, (IBM on 360, Harvard) designed for array processing
  - functional programming style
- *LISP* 1956-62, John McCarthy (MIT on IBM704, Stanford) designed for non-numerical computation
  - uniform notation for program and data
  - new conditional control structure (COND)
  - recursion as main control structure
- *Snobol* 1962-66, Farber, Griswold, Polansky (Bell Labs) designed for string processing
  - powerful pattern matching



## Hisory of PLs - 5

- *PL/I* 1963-66, IBM designed for general purpose computing [Fortran, Algol60, Cobol]
  - user controlled exceptions
  - multi-tasking
- *Simula67* 1967, Dahl and Nygaard (Norway) designed as a simulation language [Algol60]
  - data abstraction
  - inheritance of properties
- *Algol68* 1963-68, designed for general purpose computing [Algol60]
  - orthogonal language design
  - interesting user defined types

# History of PLs - 6

- *Pascal* 1969, N. Wirth(ETH) designed for teaching programming [Algol60]
  - 1 pass compiler
  - call-by-value semantics
- *Prolog* 1972, Colmerauer and Kowalski designed for Artificial Intelligence applications
  - theorem proving with unification as basic operation
  - logic programming

- Recent
  - *C* 1974, D. Ritchie (Bell Labs) designed for systems programming
    - allows access to machine level within high-level PL
    - efficient code generated



# History of PLs - 7

- *Clu* 1974-77, B. Liskov (MIT) designed for simulation [Simula]
  - supports data abstraction and exceptions
  - precise algebraic language semantics
  - attempt to enable verification of programs
- *Smalltalk* mid-1970s, Alan Kay (Xerox Parc), considered first real object-oriented PL, [Simula]
  - encapsulation, inheritance
  - easy to prototype applications
  - hides details of underlying machine
- *Scheme* mid-1970s, Guy Steele, Gerald Sussman (MIT)
  - Static scoping and first-class functions

# History of PLs - 8

- *Concurrent Pascal* **1976 Per Brinch Hansen (U Syracuse) designed for asynchronous concurrent processing [Pascal]**
  - **monitors for safe data sharing**
- *Modula* **1977 N. Wirth (ETH), designed language for large software development [Pascal]**
  - **to control interfaces between sets of procedures or *modules***
  - **real-time programming**
- *Ada* **1979, US DoD committee designed as general purpose PL**
  - **explicit parallelism - *rendezvous***
  - **exception handling and generics (*packages*)**



# History of PLs - 9

- *C++* **1985, Bjorne Stroustrup (Bell Labs) general purpose**
  - **goal of type-safe object-oriented PL**
  - **compile-time type checking**
  - **templates**
- *Java* **~1995, J. Gosling (SUN)**
  - **aimed at portability across platform through use of JVM - abstract machine to implement the PL**
  - **aimed to *fix* some problems with previous OOPLs**
    - **multiple inheritance**
    - **static and dynamic objects**
  - **ubiquitous exceptions**
  - **thread objects**