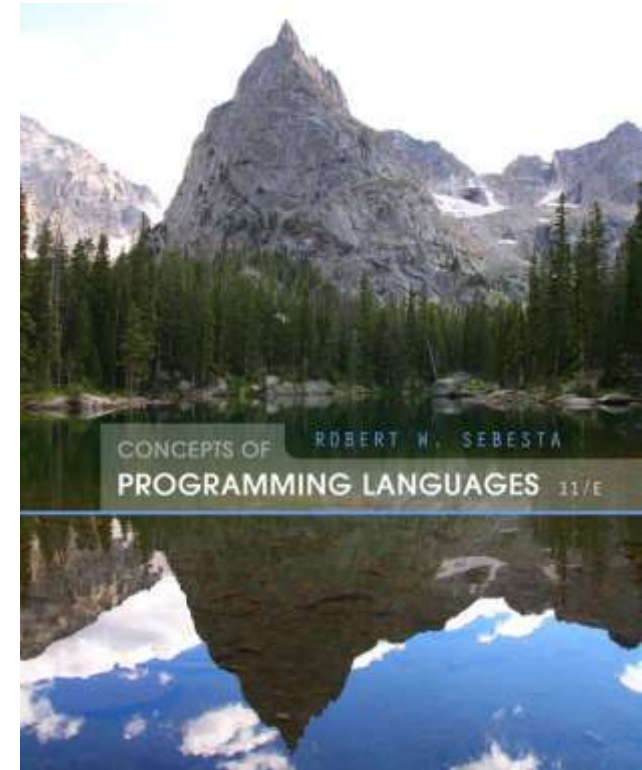


BBM 301 –
Programming Languages
Fall 2019, Lecture 1

General Information

- **Webpage:** <http://web.cs.hacettepe.edu.tr/~bbm301>
- **Teaching Assistants:**
- **Textbook:**
 - Robert W. Sebesta,
“Concepts of Programming Languages”,
Pearson, 11th Edition (also 10th is OK)
- **Time and Place:**
 - Tuesdays 13:00 – 15:45
 - Section 1 @ D8, Section 2 @ D9
- **Communication:**
 - We will use Piazza
<https://piazza.com/hacettepe.edu.tr/fall2019/bbm301>



Topics

- Introduction to Programming Languages
- Names, Bindings, Scope
- Syntax and Semantics
- Functional Languages
- Data Types
- Expressions, Assignments
- Control Statements
- Subprograms
- Implementing Subprograms
- Logic Languages

Grading Policy (Tentative)

- 10% Quizzes (4-5 quizzes)
 - 20% Project (2 phases)
 - 30% Midterm exam
 - 40% Final exam
-
- Attendance is mandatory.

Lecture 1:

Introduction to Programming Languages

What is a (Programming) Language?

A language is a vocabulary and set of grammatical rules for communication between people.

A programming language is a vocabulary and set of grammatical rules for instructing a computer to perform specific tasks. (Definition from webopedia)

Why Study Programming Languages?

- One or two languages is not enough for a computer scientist.
- You should know
 - the general concepts beneath the requirements
 - choices in designing programming languages.

**Q: How many Programming Languages do
you know?**

How many programming languages are out there?

700 +

Source: Wikipedia (excluding dialects of BASIC)

https://en.wikipedia.org/wiki/List_of_programming_languages

New Languages will Keep Coming

Be prepared to program in new languages

Languages undergo constant change

- FORTRAN 1953
- ALGOL 60 1960
- C 1973
- C++ 1985
- Java 1995

Evolution steps: 12 years per widely adopted language

- are we overdue for the next big one?

... or is the language already here?

- *Hint:* are we going through a major shift in what computation programs need to express?
- your answer here:

JavaScript

Python

· programs are distributed
· more interactive
· "installed" as part of web page

Language as a thought shaper

We will cover less traditional languages, too.

The reason:

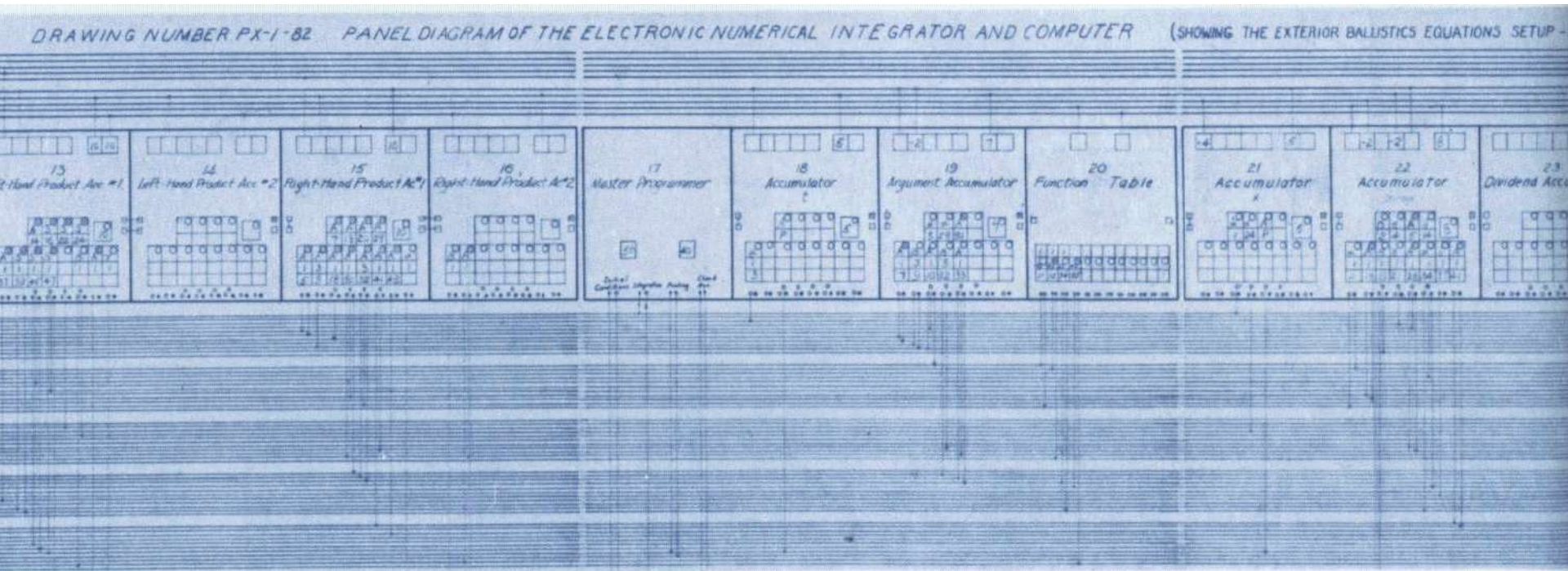
A language that doesn't affect the way you think about programming, is not worth knowing.

an Alan Perlis epigram <<http://www.cs.yale.edu/quotes.html>>

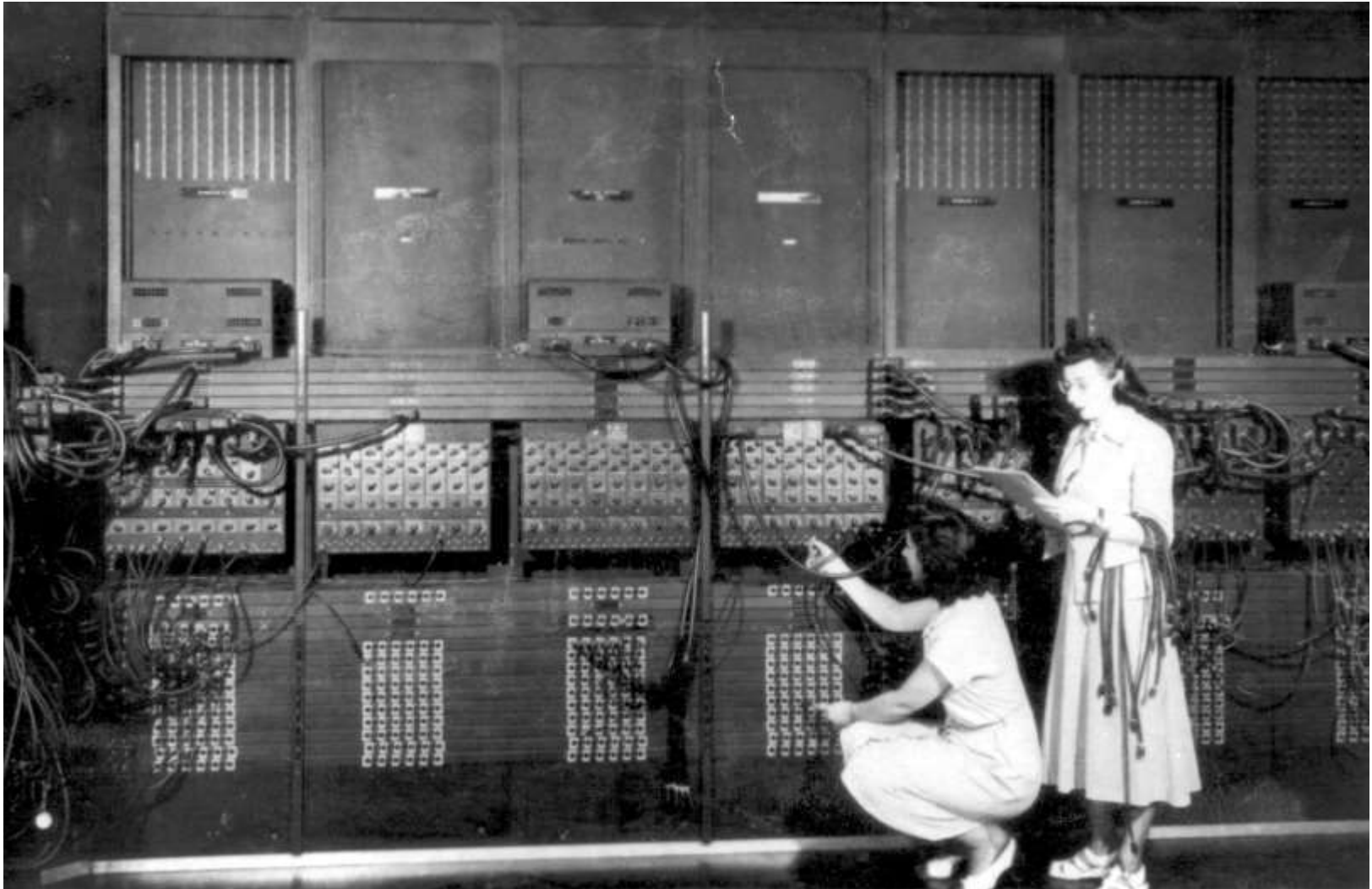
One of thought-shaper languages is Prolog.
You will both program in it and implement it.

ENIAC (1946, University of Philadelphia)

ENIAC program for external ballistic equations:



Programming the ENIAC



ENIAC (1946, Univ. of Philadelphia)

programming done by

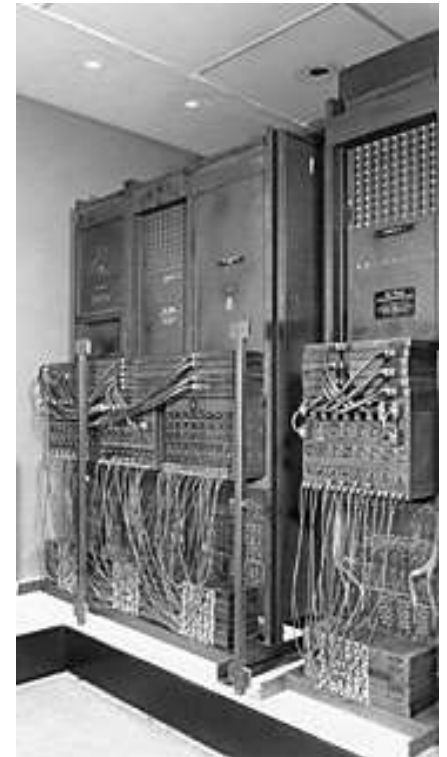
- rewiring the interconnections
- to set up desired formulas, etc

Problem (what's the tedious part?)

- programming = rewiring
- slow, error-prone

solution:

- store the program in memory!
- birth of *von Neuman* paradigm



Reasons for Studying Concepts of Programming Languages

Increased Ability to Express Ideas

- Natural languages:
 - The depth at which people think is influenced by the expressive power of the language.
 - The more appropriate constructs you have, the easier it is to communicate.
 - It is difficult for people to conceptualize structures that they cannot describe verbally.

Increased Ability to Express Ideas

- Programming Languages:
 - This is similar for PL's. The language in which you develop software puts limits on the kinds of data structures, control structures and abstractions that can be used.
 - Awareness of a wider variety of programming language features can reduce such limitations in software development.
 - Programmers increase the range of software development by learning new language constructs.
 - For example, if you learn associate arrays in Perl, you can simulate them in C.

Improved background for choosing appropriate languages

- Not every programming language can be suitable for all the software requirements.
- Many programmers learn one or two languages specific to the projects.
- Some of those languages may no longer be used.
- When they start a new project they continue to use those languages which are old and not suited to the current projects.

Improved background for choosing appropriate languages

- However another language may be more appropriate to the nature of the project.
 - Lots of text processing -> Perl may be a good option.
 - Lots of matrix processing -> MATLAB can be used.
- If you are familiar with the other languages, you can choose better languages.
- Studying the principles of PLs provides a way to judge languages:
 - “The advantages of Perl for this problem are.....”, “The advantages of Java are....”

Increased ability to learn new languages

- Programming languages are still evolving
 - Many languages are very new, new languages can be added in time.
- If you know the programming language concepts you will learn other languages much easier.
 - For example, if you know concept of object oriented programming, it is easier to learn C++ after learning Java
- Just like natural languages,
 - The better you know the grammar of a language, the easier you find to learn a second language.
 - learning new languages actually causes you to learn things about the languages you already know

Languages in common use (2018)

Oct 2018	Oct 2017	Change	Programming Language	Ratings	Change
1	1		Java	17.801%	+5.37%
2	2		C	15.376%	+7.00%
3	3		C++	7.593%	+2.59%
4	5	▲	Python	7.156%	+3.35%
5	8	▲	Visual Basic .NET	5.884%	+3.15%
6	4	▼	C#	3.485%	-0.37%
7	7		PHP	2.794%	+0.00%
8	6	▼	JavaScript	2.280%	-0.73%
9	-	▲▲	SQL	2.038%	+2.04%
10	16	▲▲	Swift	1.500%	-0.17%
11	13	▲	MATLAB	1.317%	-0.56%
12	20	▲▲	Go	1.253%	-0.10%
13	9	▼▼	Assembly language	1.245%	-1.13%
14	15	▲	R	1.214%	-0.47%
15	17	▲	Objective-C	1.202%	-0.31%
16	12	▼▼	Perl	1.168%	-0.80%
17	11	▼▼	Delphi/Object Pascal	1.154%	-1.03%

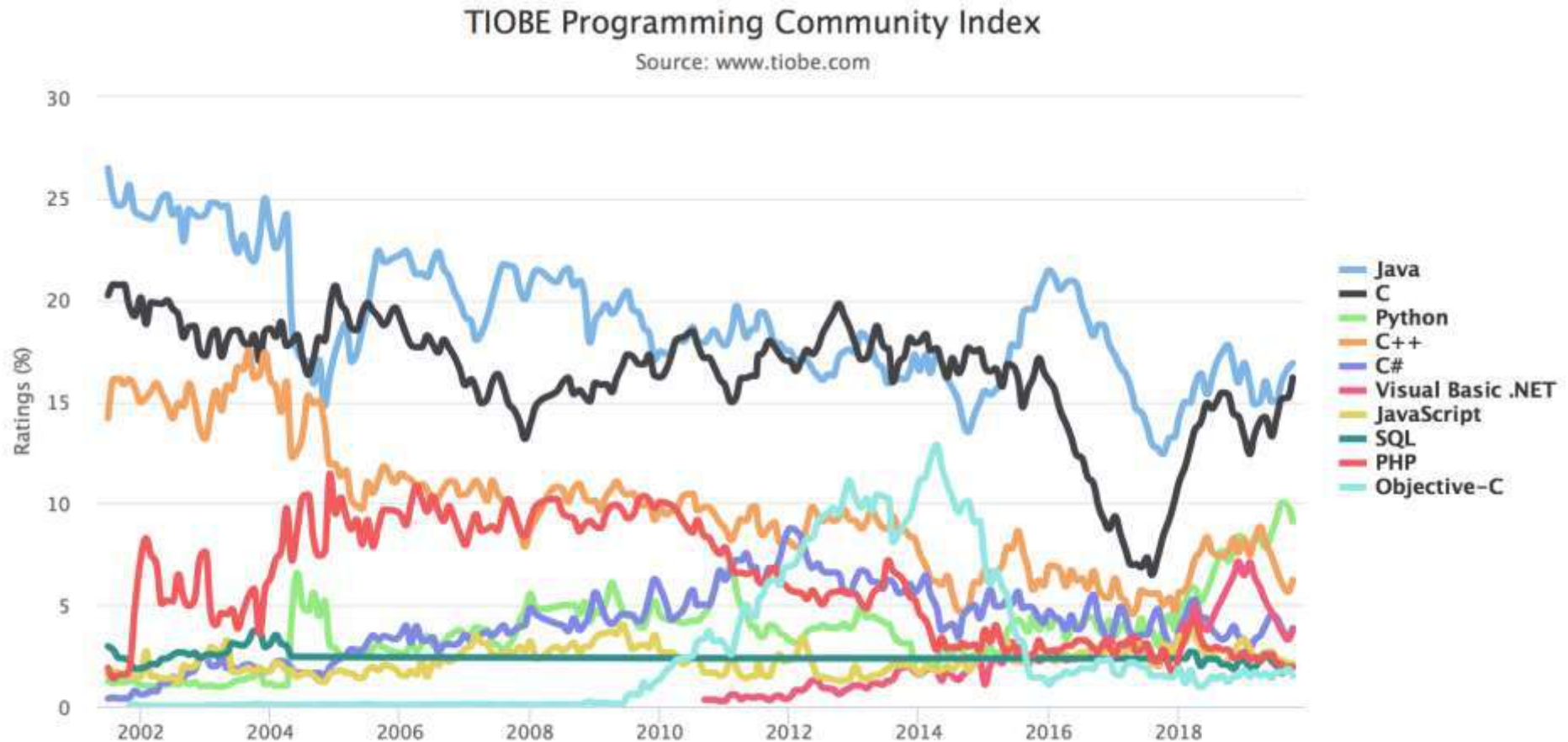
TIOBE programming community index

Languages in common use (2019)

Oct 2019	Oct 2018	Change	Programming Language	Ratings	Change
1	1		Java	16.884%	-0.92%
2	2		C	16.180%	+0.80%
3	4	⬆	Python	9.089%	+1.93%
4	3	⬇	C++	6.229%	-1.36%
5	6	⬆	C#	3.860%	+0.37%
6	5	⬇	Visual Basic .NET	3.745%	-2.14%
7	8	⬆	JavaScript	2.076%	-0.20%
8	9	⬆	SQL	1.935%	-0.10%
9	7	⬇	PHP	1.909%	-0.89%
10	15	⬆	Objective-C	1.501%	+0.30%
11	28	⬆	Groovy	1.394%	+0.96%
12	10	⬇	Swift	1.362%	-0.14%
13	18	⬆	Ruby	1.318%	+0.21%
14	13	⬇	Assembly language	1.307%	+0.06%
15	14	⬇	R	1.261%	+0.05%
16	20	⬆	Visual Basic	1.234%	+0.58%
17	12	⬇	Go	1.100%	-0.15%

TIOBE programming community index

Languages in common use (today)



Change of indexes of languages

To see the bigger picture, please find below the positions of the top 10 programming languages of many years back. Please note that these are *average* positions for a period of 12 months.

Programming Language	2019	2014	2009	2004	1999	1994	1989
Java	1	2	1	1	3	-	-
C	2	1	2	2	1	1	1
Python	3	7	6	6	22	20	-
C++	4	4	3	3	2	2	2
Visual Basic .NET	5	9	-	-	-	-	-
C#	6	5	5	7	17	-	-
JavaScript	7	8	8	9	13	-	-
PHP	8	6	4	5	-	-	-
SQL	9	-	-	89	-	-	-
Objective-C	10	3	26	36	-	-	-
Lisp	32	17	16	13	14	5	3
Pascal	219	15	14	88	7	3	21

An Interactive List

Language Types (click to hide)



Web



Mobile
































Enterprise



Embedded

<https://spectrum.ieee.org/sc/interactive-the-top-programming-languages-2017>

Language Rank	Types	Spectrum Ranking
1. Python	 	100.0
2. C	  	100.0
3. Java	  	99.4
4. C++	  	96.9
5. C#	  	88.6
6. R		88.1
7. JavaScript	 	85.3
8. PHP		81.1
9. Go	 	75.7
10. Swift	 	74.3
11. Arduino		72.4
12. Ruby	 	72.0
13. Assembly		71.7
14. Matlab		68.6
15. Scala	 	68.0

Better understanding of significance of implementation

- The best programmers are the ones having at least understanding of **how things work under the hood**
 - Understand the implementation issues
- You can simply write a code and let the compiler do everything, but knowing implementation details helps you to **use a language more intelligently** and **write the code that is more efficient**
- Also, it allows you to visualize how a computer executes language constructs
 - Cost optimization; e.g. recursion is slow

Better use of languages that are already known

- Many programming languages are large and complex
 - It is uncommon to know all the features
- By studying the concepts of programming languages, programmers can learn about previously unknown parts of the languages easily.

Overall advancement of computing

- New ways of thinking about computing, new technology, hence need for new appropriate language concepts
- Not to repeat history
 - Although ALGOL 60 was a better language (with better block structure, recursion, etc) than FORTRAN, it did not become popular. If those who choose languages are better informed, better languages would be more popular.

Develop your own language

Are you kidding? No. Guess who developed:

- PHP
- Ruby
- JavaScript
- Perl

Done by smart hackers like you

- in a garage
- not in academic ivory tower

Our goal: learn good academic lessons

- so that your future languages avoid known mistakes

Ability to Design New Languages

- You may need to design a special purpose language to enter the commands for a software that you develop.
 - A language for an robotics interface
- Studying PL concepts will give you the ability to design a new language suitable and efficient for your software requirements.

Language Evaluation Criteria

The main criteria needed to evaluate various constructs and capabilities of programming languages

- Readability
- Writability
- Reliability
- Cost

Language Evaluation Criteria

The main criteria needed to evaluate various constructs and capabilities of programming languages

- **Readability**
- Writability
- Reliability
- Cost

Readability

Ease with which programs can be read and understood

- in the early times, efficiency and machine readability was important
- 1970s-Software life cycle: coding (small) + maintenance (large)

Readability is important for maintenance

- Characteristics that contribute to readability:
 - Overall simplicity
 - Orthogonality
 - Control statements
 - Data types and structures
 - Syntax Considerations

Overall Simplicity

- A manageable set of features and constructs
 - Large number of basic components - difficult to learn
- Minimal feature multiplicity
 - **Feature multiplicity:** having more than one way to accomplish an operation

- e.g. In Java

```
count = count + 1
```

```
count += 1
```

```
count ++
```

```
++count
```

Overall Simplicity

- Minimal operator overloading
 - **Operator overloading:** a single operator symbol has more than one meaning
 - This can lead to reduced readability if users are allowed to create their own and do not do it sensibly
 - Example:
 - using + for integer and floating point addition is acceptable and contributes to simplicity
 - but if a user defines + to mean the sum of all the elements of two single dimensional arrays is not, different from vector addition
- Simplest does not mean the best
 - Assembly languages: Lack the complex control statements, so program structure is less obvious

Orthogonality

- A relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language
- Every possible combination of primitives is legal and meaningful.
- Example:
 - Four primitive data types : integer, float, double and character
 - Two type operators : array and pointer
 - If the two type operators can be applied to themselves and the four primitive data types, a large number of data structures can be defined
 - However, if pointers were not allowed to point to arrays, many of those useful possibilities would be eliminated

Orthogonality

- **Example** : Adding two 32-bit integers residing in memory or registers, and replacing one of them with the sum

- IBM (Mainframe) Assembly language has two instructions:

`A Register1, MemoryCell1`

`AR Register1, Register2`

More restricted
Less writable

Not orthogonal

meaning

$\text{Register1} \leftarrow \text{contents}(\text{Register1}) + \text{contents}(\text{MemoryCell1})$

$\text{Register1} \leftarrow \text{contents}(\text{Register1}) + \text{contents}(\text{Register2})$

- VAX Assembly language has one instruction:

`ADDL operand1, operand2`

orthogonal

meaning

$\text{operand2} \leftarrow \text{contents}(\text{operand1}) + \text{contents}(\text{operand2})$

Here, either operand can be a register or a memory cell.

Orthogonality

- Orthogonality is closely related to simplicity
- The more orthogonal the design of a language, the fewer exceptions the language rules require
- Too much orthogonality can cause problems as well:
- ALGOL68 is the most orthogonal language.
 - Every construct has a type
 - Most constructs produce values
 - This may result in extremely complex constructs,

Ex: A conditional can appear as the left side of an assignment statement, as long as it produces a location:

```
(if (A<B) then C else D) := 3
```

- This extreme form of orthogonality leads to unnecessary complexity
- *Functional languages offer a good combination of simplicity and orthogonality.*

Data Types and Structures

- Facilities for defining data types and data structures are helpful for readability
 - If there is no boolean type available then a flag may be defined as integer:

`found = 1 (instead of found = true)`

May mean something is found as boolean or what is found is 1

- An array of record type is more readable than a set of independent arrays

Syntax considerations

- **Identifier Forms:**
 - restricting identifier length is bad for readability.
 - FORTRAN77 identifiers can have at most 6 characters.
 - ANSI BASIC : an identifier is *either a single character or a single character followed by a single digit*.
 - Availability of word concatenating characters (e.g., `_`) is good for readability.
- **Special Words:** Readability is increased by special words(e.g., **begin, end, for**).
 - In PASCAL and C, **end** or `}` is used to end a compound statement. It is difficult tell what an end or `}` terminates.
 - However, ADA uses **end if** and **end loop** to terminate a selection and a loop, respectively.

Syntax Considerations

- **Forms and Meaning:** Forms should relate to their meanings. Semantics should directly follow from syntax.

For example,

`sin(x)` => should be the sine of x,
not the sign of x or cosign of x.

- **grep** is hard to understand for the people who are not familiar with using regular expressions

`grep : g/reg_exp/p`
=> `/reg_exp/` : search for that `reg_exp`
`g` : scope is whole file , make it global
`p` : print

Language Evaluation Criteria

The main criteria needed to evaluate various constructs and capabilities of programming languages

- Readability
- **Writability**
- Reliability
- Cost

Writability

- Ease of creating programs
- Most of the characteristics that contribute to readability also contribute to writability
- Characteristics that contribute to writability
 - Simplicity and Orthogonality
 - Support for abstraction
 - Expressivity
- Writability of a language can also be application dependent

Writability

- Simplicity and orthogonality
 - Few constructs, a small number of primitives, a small set of rules for combining them
- Support for abstraction
 - The ability to define and use complex structures or operations in ways that allow details to be ignored
- Expressivity
 - A set of relatively convenient ways of specifying operations
 - Strength and number of operators and predefined functions

Simplicity and orthogonality

- Simplicity and orthogonality are also good for writability.
- When there are large number of constructs, programmers may not be familiar with all of them, and this may lead to either misuse or disuse of those items.
- A smaller number of primitive constructs (simplicity) and consistent set of rules for combining them (orthogonality) is good for writability
- However, too much orthogonality may lead to undetected errors, since almost all combinations are legal.

Support for abstraction

- **Abstraction:** ability to define and use complicated structures and operations in ways that allows ignoring the details.
- PLs can support two types of abstraction:
 - process
 - data
- Abstraction is the key concept in contemporary programming languages
- The degree of abstraction allowed by a programming language and the naturalness of its expressions are very important to its writability.

Process abstraction

- The simplest example of abstraction is subprograms (e.g., methods).
- You define a subprogram, then use it by ignoring how it actually works.
- Eliminates replication of the code
- Ignores the implementation details
 - e.g. sort algorithm

Data abstraction

- As an example of data abstraction, a tree can be represented more naturally using pointers in nodes.
- In FORTRAN77, where pointer types are not available, a tree can be represented using 3 parallel arrays, two of which contain the indexes of the offspring, and the last one containing the data.

Expressivity

- Having more convenient and shorter ways of specifying computations.
- For example, in C,

```
count++;
```

is more convenient and expressive than

```
count = count + 1;
```

`for` is more easier to write loops than `while`

Language Evaluation Criteria

The main criteria needed to evaluate various constructs and capabilities of programming languages

- Readability
- Writability
- **Reliability**
- Cost

Reliability

Reliable: it performs to its specifications under all conditions

- Type checking
 - Testing for type errors
- Exception handling
 - Intercept run-time errors and take corrective measures
- Aliasing
 - Presence of two or more distinct referencing methods for the same memory location
- Readability and writability
 - The easier a program to write, the more likely it is correct.
 - Programs that are difficult to read are difficult both to write and modify.

Type Checking

- Testing for type errors in a given program either by the compiler or during program execution
- The compatibility between two variables or a variable and a constant that are somehow related (e.g., assignment, argument of an operation, formal and actual parameters of a method).
- **Run-time** (Execution-time) checking is expensive.
- **Compile-time** checking is more desirable.
- The earlier errors in programs are detected, the less expensive it is to make the required repairs

Type Checking

- Original C language requires no type checking neither in compilation nor execution time. That can cause many problems.
 - Current version required all parameters to be type-checked
- For example, the following program in original C compiles and runs!

```
foo (float a) {  
    printf ("a: %g and square(a): %g\n", a, a*a);  
}  
main () {  
    char z = 'b';  
    foo(z);  
}
```

- Output is : a: 98 and square(a): 9604

Language Evaluation Criteria

The main criteria needed to evaluate various constructs and capabilities of programming languages

- Readability
- Writability
- Reliability
- **Cost**

Cost

- Types of costs:
 1. **Cost of training the programmers:** Function of simplicity and orthogonality, experience of the programmers.
 2. **Cost of writing programs:** Function of the writability

Note: These two costs can be reduced in a good programming environment

3. **Cost of compiling programs:** cost of compiler, and time to compile
 - First generation Ada compilers were very costly

Cost

4. **Cost of executing programs:** If a language requires many run-time type checking, the programs written in that language will execute slowly.
- Trade-off between compilation cost and execution cost.
 - Optimization: decreases the size or increases the execution speed.
 - Without optimization, compilation cost can be reduced.
 - Extra compilation effort can result in faster execution.
 - More suitable in a production environment, where compiled programs are executed many times

Cost

5. **Cost of the implementation system.** If expensive or runs only on expensive hardware it will not be widely used.
6. **Cost of reliability** – important for critical systems such as a power plant or X-ray machine
7. **Cost of maintaining programs.** For corrections, modifications and additions.
 - Function of readability.
 - Usually, and unfortunately, maintenance is done by people other than the original authors of the program.
 - For large programs, the maintenance costs is about 2 to 4 times the development costs.

Other Criteria for Evaluation

- **Portability:** The ease with which programs can be moved from one implementation to another
- **Generality:** The applicability to a wide range of applications
- **Well-definedness:** The completeness and precision of the language's official definition

Language Design Trade-Offs

- Reliability vs. cost of execution
 - Example: Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs
- Readability vs. writability
 - Example: APL provides many powerful operators for arrays (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability
- Writability (flexibility) vs. reliability
 - Example: C++ pointers are powerful and very flexible but are unreliable

Programming Domains

- Scientific Applications (first digital computers –1940s)
 - Large floating-point arithmetic, execution efficiency, arrays and matrices, counting loops
 - **Examples:** FORTRAN, ALGOL 60, C
- Business Applications (1950s)
 - Producing elaborate reports, decimal numbers and character data
 - **Examples:** COBOL(1960s), Spreadsheets, Wordprocessors, Databases(SQL)
- Artificial Intelligence
 - Symbolic programming (names rather than numbers, linked lists rather than arrays)
 - **Examples:** LISP(1959), PROLOG(early 1970s)

Programming Domains (cont'd.)

- Systems Programming

- System software: Operating system and all of the programming support tools of a computer system
- Efficient and fast execution, low-level features for peripheral device drivers
- **Examples:** PLS/2(IBM Mainframe), BLISS (Digital), C (Unix)

- Scripting Languages

- List of commands (Script) to be executed is put in a file.
- **Examples:** sh, csh, tcsh, awk, gawk, tcl, perl, javascript

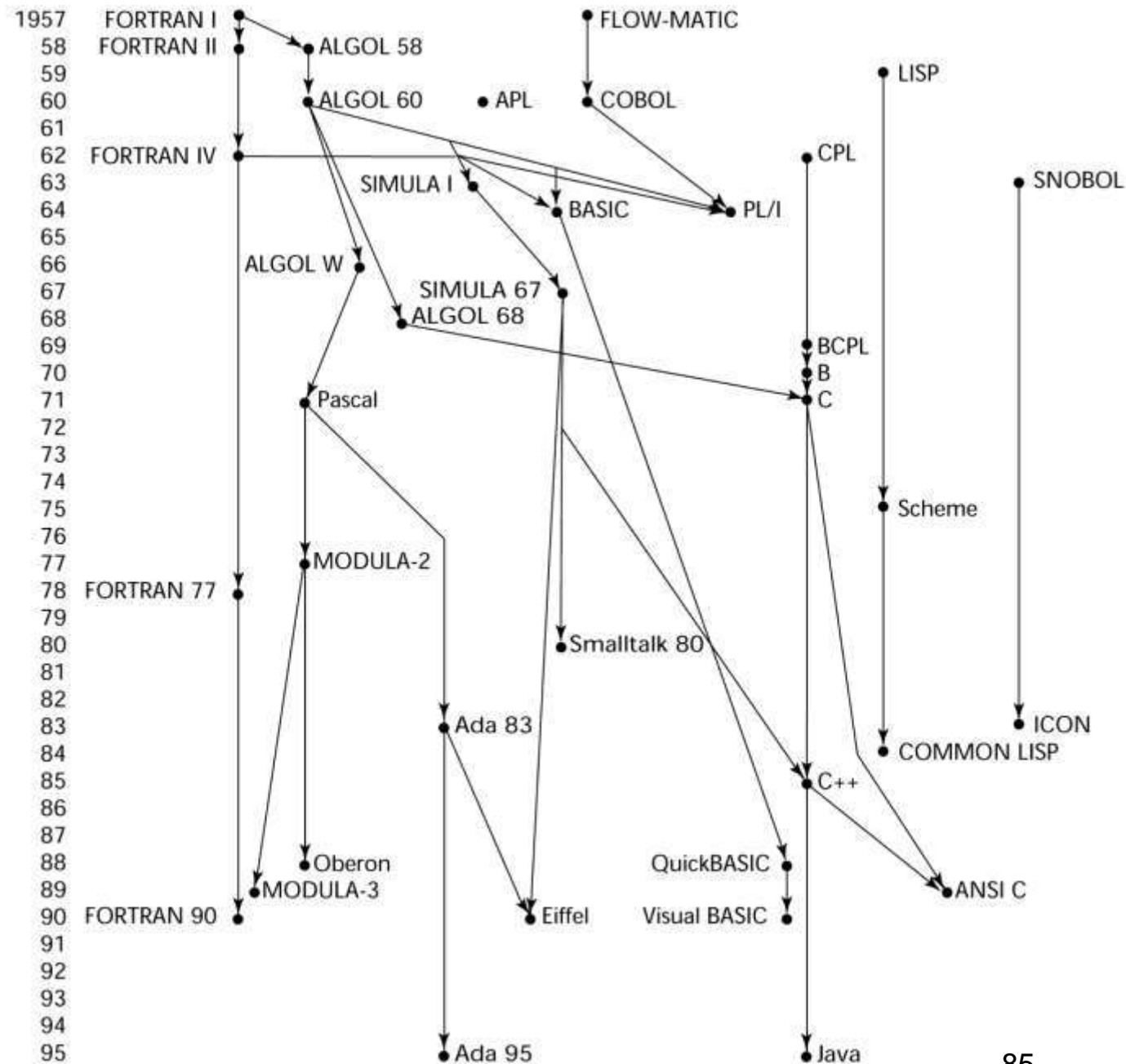
- Special-Purpose languages

- **Examples:** RPG (Business Reports), SPICE (Simulation of Electronic Circuitry), SPSS (Statistics), Latex (Document preparation). HTML, XML (web prog.)

Language Categories

- **Imperative**
 - Central features are variables, assignment statements, and iteration
 - Include languages that support object-oriented programming
 - Include *scripting languages*
 - Include the *visual languages*
 - Examples: C, Java, Perl, Visual BASIC .NET, C++
- **Functional**
 - Main means of making computations is by applying functions to given parameters
 - Examples: LISP, Scheme, ML, F#
- **Logic**
 - Rule-based (rules are specified in no particular order)
 - Example: Prolog
- **Markup/programming hybrid**
 - Markup languages extended to support some programming
 - Examples: JSTL, XSLT

Genealogy of PLs



Living & Dead Languages

- Hundreds of programming languages popped up in the 1960s, most quickly disappeared
- Some dead:
 - JOVIAL, SNOBOL, Simula-67, RPG, ALGOL, PL/1, and many, many more
- Some still kicking:
 - LISP (1957)
 - BASIC (1964)
 - Pascal (1970)
 - Prolog (1972)
 - And of course, C (1973)

“Hello World” in different languages

http://www.all-science-fair-projects.com/science_fair_projects_encyclopedia/Hello_world_program

Java

```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello, world!");  
    }  
}
```

Assembly Language

```
bdos      equ 0005H      ; BDOS entry point
start:    mvi c,9         ; BDOS function: output string
          lxi      d,msg$  ; address of msg
          call     bdos    ; return to CCP
msg$:     db 'Hello, world!$'
end       start
```

FORTRAN

```
PROGRAM
```

```
HELLO
```

```
WRITE (*,10)
```

```
10 FORMAT('Hello, world!')
```

```
STOP
```

```
END
```

COBOL

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. HELLO-WORLD.  
ENVIRONMENT DIVISION.  
DATA DIVISION.  
PROCEDURE DIVISION.  
DISPLAY "Hello, world!".  
STOP RUN.
```

Ada

```
with Ada.Text_IO;  
procedure Hello is  
begin  
    Ada.Text_IO.Put_Line ("Hello, world!");  
end Hello;
```

C

```
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
    return 0;
}
```


C++

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

C#

```
using System;  
class HelloWorldApp  
{  
    public static void Main()  
    {  
        Console.WriteLine("Hello, world!");  
    }  
}
```

Scala

```
object HelloWorld extends App {  
    println("Hello, World!")  
}
```

LISP

```
(format t "Hello world!~%" )
```

PERL

```
print "Hello, world!\n";
```

Prolog

```
write('Hello world'),nl.
```

Python

```
print("Hello World!")
```

Swift

```
import Swift  
print("Hello World!")
```


HTML

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>Hello, world!</title>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
</head>
<body>
<p>Hello, world!</p>
</body>
</html>
```

Summary

- The study of programming languages is valuable for a number of reasons:
 - Increase our capacity to use different constructs
 - Enable us to choose languages more intelligently
 - Makes learning new languages easier
- Most important criteria for evaluating programming languages include:
 - Readability, writability, reliability, cost

An optional exercise

List three new languages, or major features added to established major languages, that have appeared in the last seven years. For each language, answer with one sentence these questions. (Use your own critical thinking; do not copy text from the Web.)

- *Why did the languages appear? Or, why have these features been added?* Often, a new language is motivated by *technical* problems faced by programmers. Sometimes the motivation for a new feature is *cultural*, related to, say, the educational background of programmers in a given language.
- *Who are the intended users of this language/feature?* Are these guru programmers, beginners, end-users (non-programmers)?
- *Show a code fragment that you find particularly cool.* The fragment should exploit the new features to produce highly readable and concise code.

Links that may help you start your exploration of the programming language landscape:

- <http://lambda-the-ultimate.org/>
- <http://bit.ly/ddH47v>
- <http://www.google.com>

Names, Bindings, Type Checking and Scopes

BBM 301 – Programming
Languages

Today

- Introduction
- Names
- Variables
- The Concept of Binding
- Type Inference
- Scope
- Scope and Lifetime
- Referencing Environments
- Named Constants

Introduction

- Imperative programming languages are abstractions of the underlying von Neumann computer architecture
 - **Memory** – stores both instructions and data
 - **Processor** – provides operations for modifying the contents of the memory
- Variables are characterized by attributes
 - To design a data type, must consider scope, lifetime, type checking, initialization, and type compatibility

Abstraction

- Abstractions for memory are **variables**
- Sometimes abstraction is very close to characteristics of cells.
 - e.g. Integer – represented directly in one or more bytes of a memory
- In other cases, abstraction is far from the organization of memory.
 - e.g. Three dimensional array.
 - requires software mapping function to support the abstraction

Names

- Variables, subprograms, labels, user defined types, formal parameters all have names.
- Design issues for names:
 - What is the maximum length of a name?
 - Are names case sensitive or not?
 - Are special words reserved words or keywords?

Names (continued)

- **Length**

- If too short, they cannot be connotative
- Language examples:
 - Earliest languages : single character
 - FORTRAN 95: maximum of 31 characters
 - C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31 characters
 - C#, Ada, and Java: no limit, and all are significant
 - C++: no limit, but implementers often impose one

Name Forms

- Names in most PL have the same form:
 - A letter followed by a string consisting of letters, digits, and underscore characters
 - In some, they use special characters before a variable's name
- Today “camel” notation is more popular for C-based languages (e.g. `myStack`)
- In early versions of Fortran – embedded spaces were ignored. e.g. following two names are equivalent

`Sum Of Salaries`

`SumOfSalaries`

Names (continued)

- **Special characters**

- PHP: all variable names must begin with dollar signs
- Perl: all variable names begin with special characters (\$, @, %), which specify the variable's type
- Ruby: variable names that begin with @ are instance variables; those that begin with @@ are class variables

Names (continued)

- **Case sensitivity**

- In many languages (e.g. C-based languages) uppercase and lowercase letters in names are distinct
 - e.g. rose, ROSE, Rose
- Disadvantage: readability (names that look alike are different)
 - Names in the C-based languages are case sensitive
 - Names in others are not
 - Worse in C++, Java, and C# because predefined names are mixed case (e.g. `IndexOutOfBoundsException`)
- Also bad for writability since programmer has to remember the correct cases

Names (continued)

- **Special words**

- An aid to readability; used to delimit or separate statement clauses

- A **keyword** is a word that is special only in certain contexts, e.g., in Fortran

`Real VarName` (*Real is a data type followed with a name, therefore Real is a keyword*)

`Real = 3.4` (*Real is a variable*)

`INTEGER REAL`

`REAL INTEGER`

This is allowed but not readable.

Names (continued)

- **Special words**

- A ***reserved word*** is a special word that cannot be used as a user-defined name
 - Can't define `for` or `while` as function or variable names.
 - Good design choice
 - Potential problem with reserved words: If there are too many, many collisions occur (e.g., COBOL has 300 reserved words!)

Special Words

- **Predefined names:** have predefined meanings, but can be redefined by the user
- Between special words and user-defined names.
- For example, built-in data type names in Pascal, such as `INTEGER`, normal input/output subprogram names, such as `readln`, `writeln`, are predefined.
- In Ada, `Integer` and `Float` are predefined, and they can be redefined by any Ada program.

Variables

- A **variable** is an abstraction of a memory cell
- It is not just a name for a memory location
- A variable is characterized by a collection of attributes
 - Name
 - Address
 - Value
 - Type
 - Scope
 - Lifetime

Variable Attributes – Name

- Most variables are named (often referred as identifiers).
- Although nameless variables do exist (e.g. pointed variables).

Variable Attributes – Address

- **Address** - the memory address with which it is associated
- It is possible that the same name refer to different locations
- in different parts of a program:
 - A program can have two subprograms `sub1` and `sub2` each of defines a local variable that use the same name, e.g. `sum`
- in different times:
 - For a variable declared in a recursive procedure, in different steps of recursion it refers to different locations.
- Address of a variable is sometimes called **l-value**, because address is required when a variable appears on the left side of an assignment.

Variable Attributes – Aliases

- Multiple identifiers reference the same address – more than one variable are used to access the same memory location
- Such identifier names are called **aliases**.
- Aliases are created via pointers, reference variables, C and C++ unions
- Aliases are harmful to readability (program readers must remember all of them)

Variable Attributes – Type

- **Type** – determines
 - the range of values the variable can take, and
 - the set of operators that are defined for values of this type.
 - in the case of floating point, type also determines the precision
- For example `int` type in Java specifies a range of
-2147483648 to 2147483647

Variable Attributes – Value

- The contents of the location with which the variable is associated
- e.g. $l_value \leftarrow r_value$ (assignment operation)
 - The l-value of a variable is its address
 - The r-value of a variable is its value

Variable Attributes – Abstract memory cell

- **Abstract memory cell** – the physical cell or collection of cells associated with a variable
 - Physical cells are 8 bits
 - This is too small for most program variables

The concept of Binding

- A **binding** is association between
 - entity \leftrightarrow attribute (such as between a variable and its type or value), or
 - operation \leftrightarrow symbol
- **Binding time** is the time at which a binding takes place.
 - important in the semantics of PLs

Possible Binding Times

- **Language design time** – bind operator symbols to operations
 - * is bound to the multiplication operation,
 - pi=3.14159 in most PL's.
- **Language implementation time**
 - bind floating point type to a representation
 - `int` in C is bound to a range of possible values
- **Compile time** -- bind a variable to a type in C or Java

Possible Binding Times (continued)

- **Link time**

- A call to the library subprogram is bound to the subprogram code.

- **Load time**

- A variable is bound to a specific memory location.
- e.g. bind a C or C++ `static` variable to a memory cell

- **Runtime**

- A variable is bound to a value through an assignment statement.
- A local variable of a Pascal procedure is bound to a memory location.

Binding Times

- **Example:**

– `count = count + 5`

- The type of `count` is bound at compile time
- The set of possible values of `count` is bound at compiler design time
- The meaning of the operator symbol `+` is bound at compile time, when the types of its operands have been determined
- The internal representation of the literal `5` is bound at compiler design time
- The value of `count` is bound at execution times with this statement

Static and Dynamic Binding

- A binding is **static** if it first occurs before run time and remains unchanged throughout program execution.
- A binding is **dynamic** if it first occurs during execution or can change during execution of the program

Type Bindings

- Before a variable can be referenced in a program, it must be bound to a data type.
- Two important aspects
 - How is a type specified?
 - When does the binding takes place?
- If static, the type may be specified by either an explicit or an implicit declaration

Static Type Binding – Explicit/Implicit Declarations

- **explicit declaration** (by statement)
 - A statement in a program that lists variable names and specifies that they are a particular type
- **implicit declaration** (by first appearance)
 - Means of associating variables with types through default conventions, rather than declaration statements. First appearance of a variable name in a program constitutes its implicit declaration
- **Both creates static binding to types**

Static Type Binding

- Most current PLs require explicit declarations of all variables
 - Some exceptions: Perl, Javascript, ML
- Early languages (Fortran, BASIC) have implicit declarations
 - e.g. In Fortran, if not explicitly declared, an identifier starting with I, J, K, L, M, N are implicitly declared to integer, otherwise to real type
- Implicit declarations are not good for reliability and writability because misspelled identifier names cannot be detected by the compiler
 - e.g. In Fortran variables that are accidentally left undeclared are given default types, and leads to errors that are difficult to diagnose

Static Type Binding

- Some problems of implicit declarations can be avoided by requiring names for specific types to begin with a particular special characters
- Example: In Perl
 - `$apple : scalar`
 - `@apple : array`
 - `%apple : hash`

Dynamic Type Binding

- Type of a variable is not specified by a declaration statement, nor it can be determined by the spelling of its name (JavaScript, Python, Ruby, PHP, and C# (limited))
- Type is bound when it is assigned a value by an assignment statement.
- **Advantage:** Allows programming flexibility.
example languages: Javascript and PHP
- e.g. In JavaScript
 - `list = [10.2 5.1 0.0]`
 - `list` is a single dimensioned array of length 3.
 - `list = 73`
 - `list` is a simple integer.

Dynamic Type Binding – Disadvantages

1. Less reliable: compiler cannot check and enforce types.

- Example: Suppose \mathbb{I} and \mathbb{X} are integer variables, and \mathbb{Y} is a floating-point.

- The correct statement is

$$\mathbb{I} := \mathbb{X}$$

- But by a typing error

$$\mathbb{I} := \mathbb{Y}$$

- Is typed. In a dynamic type binding language, this error cannot be detected by the compiler.

\mathbb{I} is changed to float during execution.

- The value of \mathbb{I} becomes erroneous.

Dynamic Type Binding – Disadvantages

2. Cost:

- Type checking must be done at run-time.
- Every variable must have a descriptor to maintain current type.
- The correct code for evaluating an expression must be determined during execution.
- Languages that use dynamic type bindings are usually implemented as interpreters (LISP is such a language).

Type Inference

- ML is a PL that supports both functional and imperative programming
- In ML, the types of most expressions can be determined without requiring the programmer to specify the types of the variables

- General syntax of ML

```
fun function_name(formal parameters) =  
  expression;
```

- The type of an expression and a variable *can be determined by the type of a constant* in the expression
- Examples

```
fun circum (r) = 3.14 *r*r; (circum is real)
```

```
fun times10 (x) = 10*x; (times10 is integer)
```

[Note: **fun** is for function declaration.]

Type Inference

```
fun square (x) = x*x;
```

- Determines the type by the definition of * operator
- Default is `int`. if called with `square(2.75)` it would cause an error
- ML does not coerce real to int

- It could be rewritten as:

```
fun square (x: real) = x*x;
```

```
fun square (x):real = x*x;
```

```
fun square (x) = (x:real)*x;
```

```
fun square (x) = x*(x:real);
```

- In ML, there is no overloading, so only one of the above can coexist

- Purely functional languages Miranda and Haskell uses Type Inference.

Storage Bindings and Lifetime

- **Allocation:** process of taking the memory cell to which a variable is bound from a pool of available memory
- **Deallocation:** process of placing the memory cell that has been unbound from a variable back into the pool of available memory
- **Lifetime of a variable:** Time during the variable is bound to a specific memory location
- According to their lifetimes, variables can be separated into four categories:
 - static,
 - stack-dynamic,
 - explicit heap-dynamic,
 - implicit dynamic.

Static Variables

- Static variables are bound to memory cells before execution begins, and remains bound to the same memory cells until execution terminates.
- **Applications:** globally accessible variables, to make some variables of subprograms to retain values between separate execution of the subprogram
- Such variables are **history sensitive**.
- **Advantage:** Efficiency. Direct addressing (no run-time overhead for allocation and deallocation).
- **Disadvantage:** Reduced flexibility (no recursion).
- If a PL has only static variables, it cannot support recursion.
- Examples:
 - All variables in FORTRAN I, II, and IV
 - Static variables in C, C++ and Java

Stack-Dynamic Variables

- **Storage binding:** when declaration statement is elaborated (in run-time).
- **Type binding:** statical.
- The local variables get their type binding statically at compile time, but their storage binding takes place when that procedure is called. Storage is deallocated when the procedure returns.
- Local variables in C functions.

Stack-Dynamic Variables

- **Advantages:**
 - Dynamic storage allocation is needed for recursion. Each subprogram can have its own copy of the variables
 - Same memory cells can be used for different variables (efficiency)
- **Disadvantages:** Runtime overhead for allocation and deallocation
- In C and C++, local variables are, by default, stack-dynamic, but can be made static through static qualifier.

```
foo ()  
{  
    static int x;  
    ...  
}
```

All attributes other than storage is statically bound to this type of variables

Explicit Heap-Dynamic Variables

- Nameless variables
- storage allocated/deallocated by explicit run-time instructions
- can be referenced only through pointer variables
- e.g. dynamic objects in C++ (via `new` and `delete`), all objects in Java
- types can be determined at run-time
- storage is allocated when created explicitly

Explicit Heap-Dynamic Variables

- Example:

- In C++

```
int *intnode;           // Create a pointer
intnode = new int;      // Create the heap-dynamic variable
...
delete intnode;         // Deallocate the heap-dynamic variable
```

- Advantages:

- Required for dynamic structures (e.g., linked lists, trees)

- Disadvantages:

- Difficult to use correctly, costly to refer, allocate, deallocate.

Implicit Heap-Dynamic Variables

- Storage and type bindings are done when they are assigned values.
- **Advantages:**
 - Highest degree of flexibility (generic code)
- **Disadvantages:**
 - Runtime overhead for allocation/deallocation and maintaining all the attributes which can include array subscript types and ranges.
 - Loss of error detection by compiler
- Examples: All variables in APL; all strings and arrays in Perl, JavaScript, and PHP.

Variable Attributes – Scope

- **Scope** of a variable is the **range of statements** in which the **variable is visible**.
- The *local variables* of a program unit are those that are declared in that unit
- The *nonlocal variables* of a program unit are those that are visible in the unit but not declared there
- *Global variables* are a special category of nonlocal variables
- The scope rules of a language determine how references to names are associated with variables

Static Scope

- Scope of variables can be determined statically
 - by looking at the program
 - prior to execution
- First defined in ALGOL 60.
- Based on program text
- To connect a name reference to a variable, you (or the compiler) must find the declaration

Static Scope

- **Search process:**
 - search declarations,
 - first locally,
 - then in increasingly larger enclosing scopes,
 - until one is found for the given name

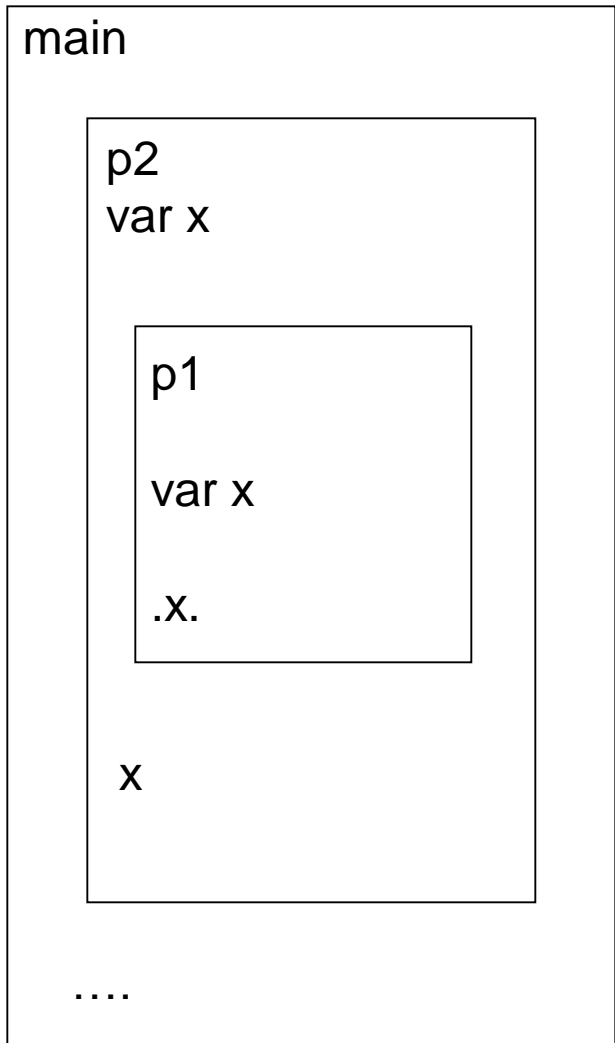
Static Scope

- In all static-scoped languages (except C), procedures are nested inside the main program.
- Some languages also allow nested subprograms, which create nested static scopes
 - Ada, JavaScript, Common LISP, Scheme, Fortran 2003+, F#, and Python - do
 - C based languages – do not
- In this case all procedures and the main unit create their scopes.

Static Scope

- Enclosing static scopes (to a specific scope) are called its static ancestors
- the nearest static ancestor is called a **static parent**

Static Scope



- `main` is the static parent of `p2` and `p1`.
- `p2` is the static parent of `p1`

Static Scope

```
Procedure Big is
  x : integer
  procedure sub1 is
    begin - of
      sub1
    .... x ....
  end - of sub1
  procedure sub2 is
    x: integer;
    begin - of
      sub2
    ....
  end - of sub2
begin - of big
...
end - of big
```

- The reference to variable `x` in `sub1` is to the `x` declared in procedure `Big`
- `x` in `Big` is hidden from `sub2` because there is another `x` in `sub2`

Static Scope

- In some languages that use static scoping, regardless of whether nested subprograms are allowed, some variable declarations can be hidden from some other code segments
- e.g. In C++

```
void sub1() {  
    int count;  
    ...  
    while (...) {  
        int count;  
        ...  
    }  
    ...  
}
```

- The reference to `count` in while loop is local
- `count` of `sub1()` is hidden from the code inside the while loop

Static Scope

- Variables can be hidden from a unit by having a "closer" variable with the same name
- C++ and Ada allow access to these "hidden" variables
 - In Ada: `unit.name`
 - In C++: `class_name::name`

Blocks

- Some languages allow new static scopes to be defined without a name.
- It allows a section of code its own local variables whose scope is minimized.
- Such a section of code is called a block
- The variables are typically stack dynamic so they have their storage allocated when the section is entered and deallocated when the section is exited
- Blocks are first introduced in Algol 60

Blocks

- In Ada

...

```
declare TEMP: integer;
```

```
begin
```

```
TEMP := FIRST;
```

```
FIRST := SECOND;
```

Block

```
SECOND := TEMP;
```

```
end;
```

...

Blocks

C and C++ allow blocks.

```
int first, second;  
...  
first = 3; second = 5;  
{ int temp;  
    temp = first;  
    first = second;  
    second = temp;  
}  
...
```

temp is undefined here.

Blocks

- C++ allows variable definitions to appear anywhere in functions. The scope is from the definition statement to the end of the function
- In C, all data declarations (except the ones for blocks) must appear at the beginning of the function
- `for` statements in C++, Java and C# allow variable definitions in their initialization expression. The scope is restricted to the `for` construct

Dynamic Scope

- APL, SNOBOL4, early dialects of LISP use dynamic scoping.
- COMMON LISP and Perl also allows dynamic scope but also uses static scoping
- In **dynamic scoping**
 - scope is based on the calling sequence of subprograms
 - not on the spatial relationships
 - scope is determined at run-time.

Dynamic Scope

```
Procedure Big is
  x : integer
  procedure sub1 is
    begin - of sub1
    .... x ....
    end - of sub1
  procedure sub2 is
    x: integer;
    begin - of sub2
    ....
    end - of sub2
begin - of big
...
end - of big
```

- When the search of a local declaration fails, the declarations of the dynamic parent is searched
- **Dynamic parent is the calling procedure**

- **Big calls sub2**
- **sub2 calls sub1**
- **Dynamic parent of sub1 is sub2, sub2 is Big**

Dynamic Scope

```
procedure big
  var x: integer;
  procedure sub1;
  begin
    ... x ...
  end; {sub1}
  procedure sub2;
    var x: integer;
    begin
      sub1 ;
    end;
  begin
    sub2;
    sub1;
  end;
end;
```

dynamic scoping
(called from big)

static scoping

dynamic scoping
(called from sub1)

To determine the correct meaning of a variable, first look at the local declarations.

For **static** or **dynamic** scoping, the local variables are the same.

In **dynamic scoping**, look at the dynamic parent (calling unit).

In **static scoping**, look at the static parent (unit that declares, encloses).

Referencing Environments

- The **referencing environment** of a statement is the collection of all names that are visible in the statement
- In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes
- A subprogram is **active** if its execution has begun but has not yet terminated
- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms

Summary

- Case sensitivity and the relationship of names to special words represent design issues of names
- Variables are characterized by the sextuples: name, address, value, type, lifetime, scope
- Binding is the association of attributes with program entities
- Scalar variables are categorized as: static, stack dynamic, explicit heap dynamic, implicit heap dynamic
- Scope of a variable is the range of statements in which the variable is visible and can be static, or dynamic.

BBM 301 –
Programming Languages
Fall 2019, Lecture 3

Today

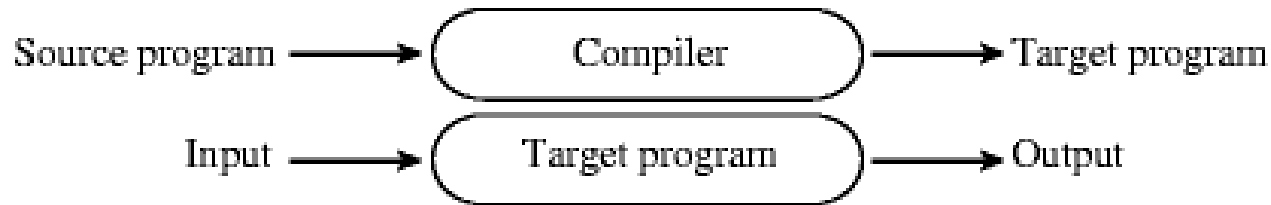
- Implementation Methods
- Describing Syntax and Semantics (Chapter 3)

Implementation Methods

- **Compilation**
 - Programs are translated into machine language; includes JIT systems
 - Use: Large commercial applications
- **Pure Interpretation**
 - Programs are interpreted by another program known as an interpreter
 - Use: Small programs or when efficiency is not an issue
- **Hybrid Implementation Systems**
 - A compromise between compilers and pure interpreters
 - Use: Small and medium systems when efficiency is not the first concern

Compilation and Interpretation

- *Compiler* translates into target language (machine language), then goes away
- The target program is the locus of control at execution time



- *Interpreter* stays around at execution time
- Implements a virtual machine whose machine language is the high-level language

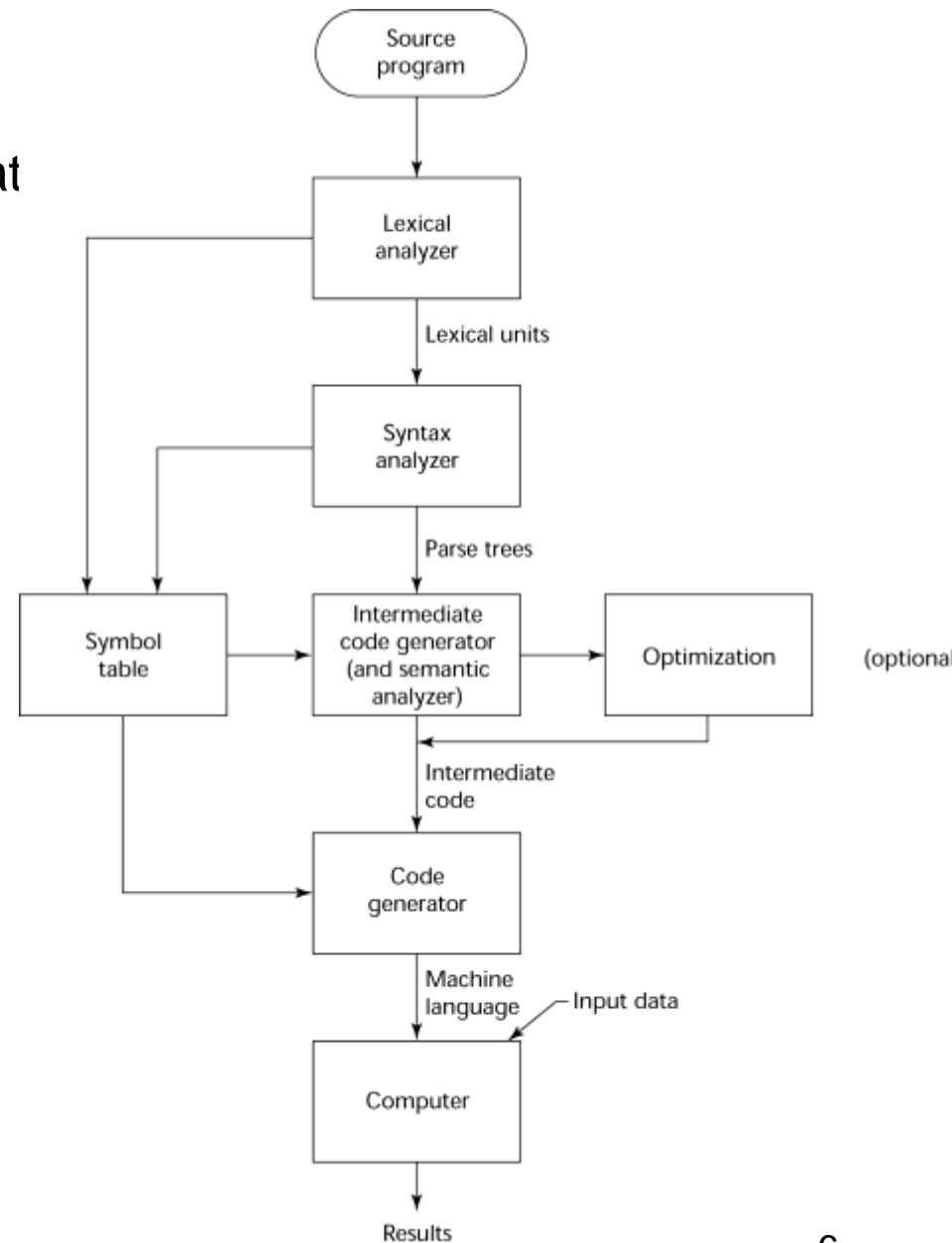


Compilation

- Translate high-level program (source language) into machine code (machine language)
- **Slow translation, fast execution**
- Compilation process has several phases:
 - lexical analysis: converts characters in the source program into lexical units
 - syntax analysis: transforms lexical units into *parse trees* which represent the syntactic structure of program
 - Semantics analysis: generate intermediate code
 - code generation: machine code is generated
- Example: C, C++, COBOL, Ada

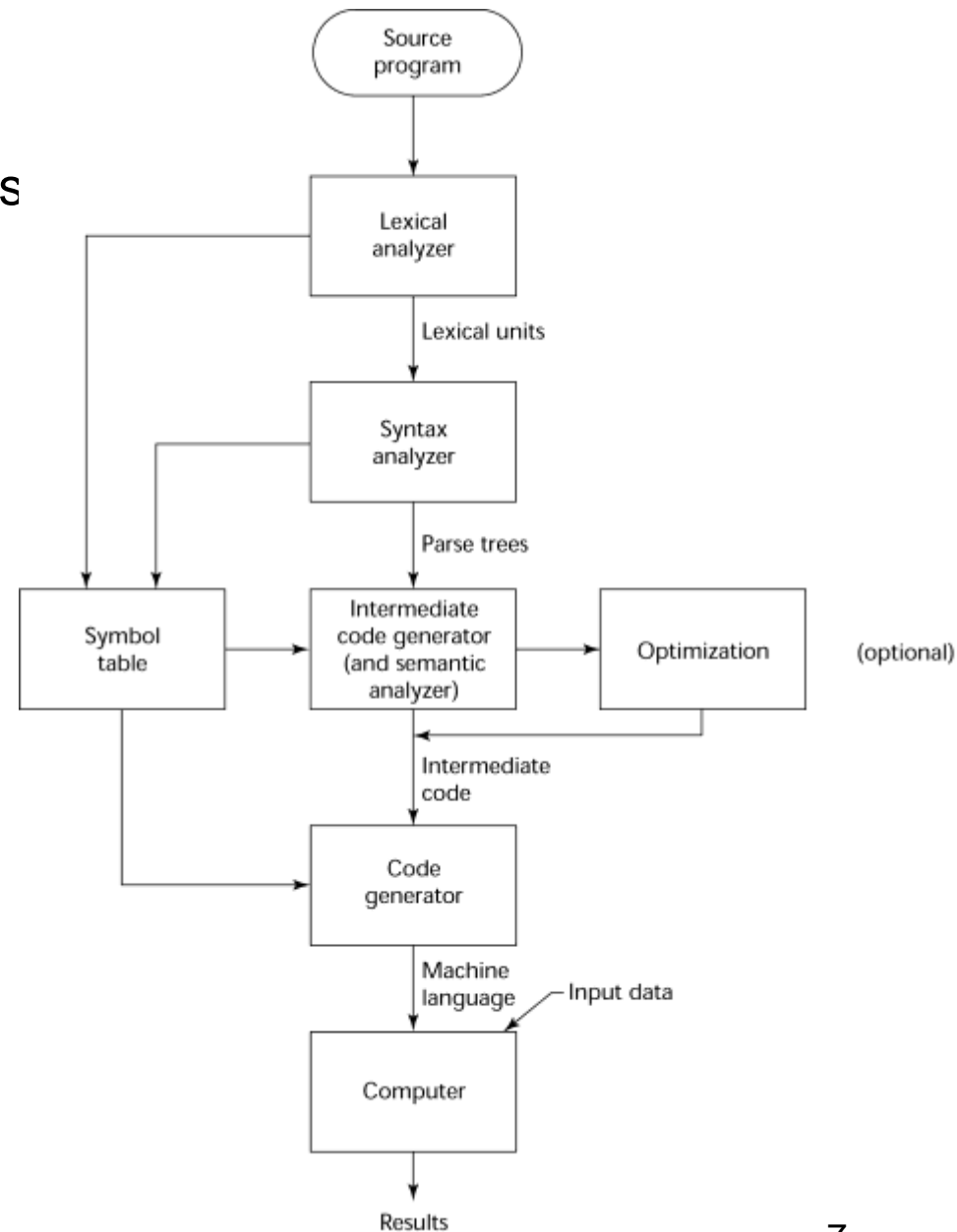
Compilation

- **Source language:** The language that the compiler translates
- **Lexical analyzer:** gathers the characters of the source program into lexical units (e.g. identifiers, special words, operators, punctuation symbols) Ignores the comments
- **Syntax analyzer:** takes the lexical units, and use them to construct parse trees, which represent the syntactic structure of the program
- **Intermediate code generator:** Produces a program at an intermediate level. Similar to assembly languages



Compilation

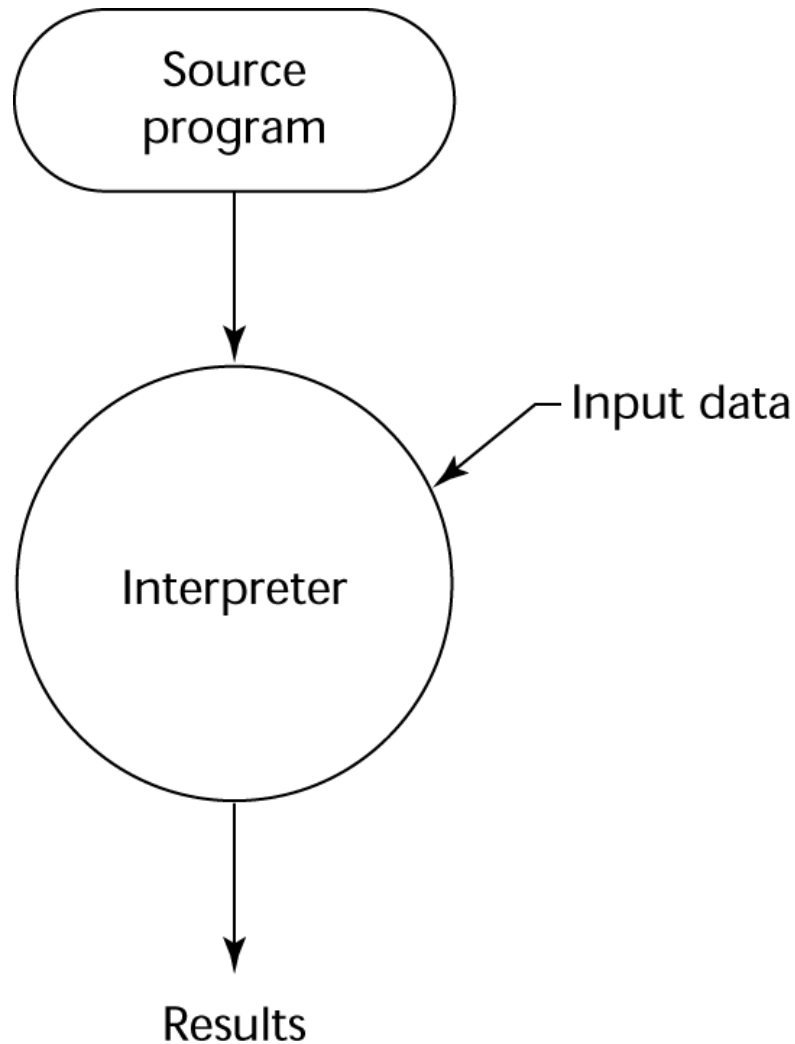
- **Semantic analyzer:** checks for errors that are difficult to check during syntax analysis, such as type errors.
- **Optimization (optional):** Used if execution speed is more important than compilation speed.
- **Code generator:** Translates the intermediate code to machine language program.
- **Symbol table:** serves as a database of type and attribute information of each user defined name in a program. Placed by lexical and syntax analyzers, and used by semantic analyzer and code generator.



Pure Interpretation

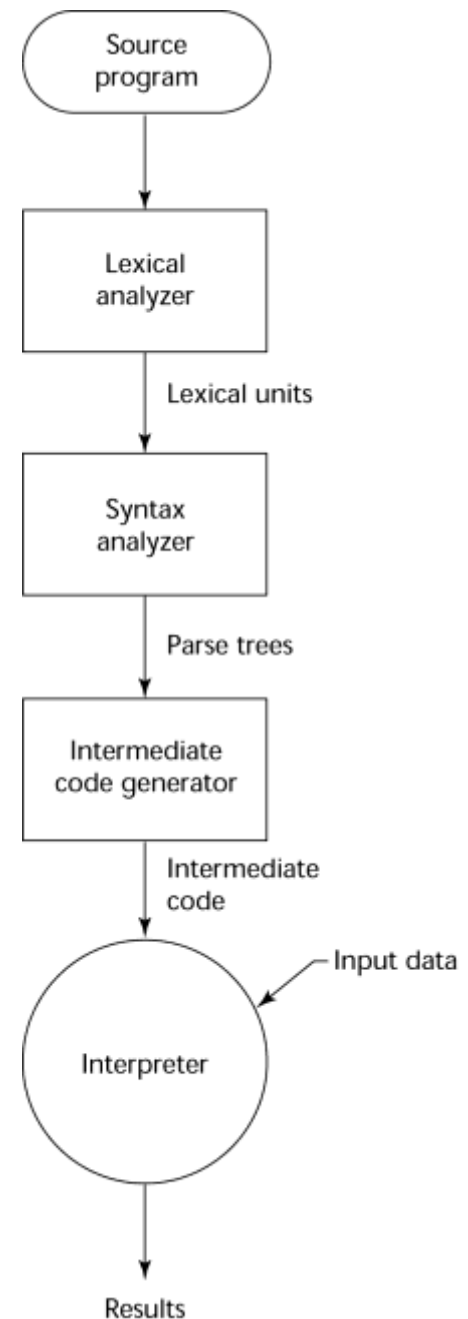
- No translation.
- Each statement is decoded and executed individually.
- The interpreter program acts as a software simulation of a machine
- **Advantages:**
 - Easier implementation of programs
 - Debugging is easy (run-time errors can easily and immediately be displayed)
- **Disadvantages:**
 - Slower execution (10 to 100 times slower than compiled programs, and statement decoding is the major bottleneck)
 - Often requires more space
- Now rare for traditional high-level languages but available in some Web scripting languages (e.g., JavaScript, PHP)

Pure Interpretation Process



Hybrid Implementation Systems

- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation
 - Example: Perl programs are partially compiled to detect errors before interpretation



Today

- Implementation Methods
- Describing Syntax and Semantics (Chapter 3)

Creating computer programs

- Each programming language provides **a set of primitive operations**
- Each programming language provides **mechanisms for combining primitives** to form more complex, but legal, expressions
- Each programming language provides **mechanisms for deducing meanings** or values associated with computations or expressions

Aspects of languages

- Primitive constructs
 - ***Programming language***: numbers, strings, simple operators
 - ***English*** : words
- **Syntax**— which strings of characters and symbols are well-formed
 - ***Programming language***: $3.2 + 3.2$ is a valid C expression
 - ***English***: “cat dog boy” is not syntactically valid, as not in form of acceptable sentence

Aspects of languages

- Static semantics – which syntactically valid strings have a meaning
 - English – “**I are big**” has form <noun> <intransitive verb> <noun>, so syntactically valid, but is not valid English because “I” is singular, “are” is plural
 - Programming language – for example, <literal> <operator> <literal> is a valid syntactic form, but **2.3/”abc”** is a static semantic error!

Aspects of languages

- **Semantics** – what is the meaning associated with a syntactically correct string of symbols with no static semantic errors
 - English – can be ambiguous
 - “I cannot praise this student too highly”
 - “Yaşlı adamın yüzüne dalgın dalgın baktı.”
 - Programming languages – always has exactly one meaning
 - But meaning (or value) may not be what programmer intended

Today

- **Syntax:** the form or structure of the expressions, statements, and program units
- **Semantics:** the meaning of the expressions, statements, and program units
- Syntax and semantics provide a language's definition
 - Users of a language definition
 - Other language designers
 - Implementers
 - Programmers (the users of the language)

Example: Syntax and Semantics

- `while` statement in Java
- **syntax:** `while (<boolean_expr>)`
`<statement>`
- **semantics:** when *boolean_expr* is true,
statement will be executed
- *The meaning of a statement should be clear from its syntax (Why?)*

Describing Syntax: Terminology

- ***Alphabet:*** Σ , All strings: Σ^*
- A ***sentence*** is a string of characters over some alphabet
- A ***language*** is a set of sentences, $L \subseteq \Sigma^*$

Describing Syntax: Terminology

- A *language* is a set of sentences
 - Natural languages: English, Turkish, ...
 - Programming languages: C, Fortran, Java,...
 - Formal languages: a^*b^* , 0^n1^n
- String of the language:
 - Sentences
 - Program statements
 - Words (aaaaabb, 000111)

Lexemes

- A **lexeme** is the lowest level syntactic unit of a language (e.g., `*`, `sum`, `begin`)
- Lower level constructs are given not by the syntax but by lexical specifications.
- Examples: identifiers, constants, operators, special words.

`total, sum_of_products, 1254, ++, (:`

- So, a language is considered as **a set of strings of lexemes** rather than strings of chars.

Token

- A ***token*** of a language is a category of lexemes
- For example, ***identifier*** is a token which may have lexemes, or instances, `sum` and `total`

Example in Java Language

<code>x = (y+3.1) * z_5 ;</code>	
<code>x = (y+3.1) * z_5 ;</code>	
<u>Lexemes</u>	<u>Tokens</u>
x	identifier
=	equal_sign
(left_paren
)	right_paren
for	for
y	identifier
+	plus_op
3.1	float_literal
*	mult_op
z_5	identifier
;	semi_colon

Describing Syntax

- *Higher level constructs are given by syntax rules.*
- **Syntax rules** specify which strings from Σ^* are in the language
- Examples: organization of the program, loop structures, assignment, expressions, subprogram definitions, and calls.

Elements of Syntax

- An alphabet of symbols
- Symbols are **terminal** and **non-terminal**
 - **Terminals** cannot be broken down
 - **Non-terminals** can be broken down further
- Grammar rules that express how symbols are combined to make legal sentences
- Rules are of the general form
`non-terminal symbol ::= list of zero or more terminals or non-terminals`
- One uses rules to recognize (parse) or generate legal sentences

Formal Definition of Languages

- **Recognizers**

- A recognition device reads input strings over the alphabet of the language and decides whether the input strings belong to the language
- Example: syntax analysis part of a compiler

- **Generators**

- A device that generates sentences of a language
- One can determine if the syntax of a particular sentence is syntactically correct by comparing it to the structure of the generator

Formal Methods of Describing Syntax

- **Grammars: formal language-generation mechanisms.**
- In the mid-1950s, Chomsky, described four classes of generative devices (or grammars) that define four classes of languages.
 - Two of these grammar classes, named context-free and regular, turned out to be useful for describing the syntax of programming languages.
 - **Regular grammars:** The forms of the tokens of programming languages
 - **Context-free grammars:** The syntax of whole programming

Regular Language vs CFL

- Tokens can be generated using three formal rules
 - Concatenation
 - Alternation (|)
 - Kleene closure (repetition an arbitrary number of times)(*)
- Any sets of strings that can be defined by these three rules is called **a regular set** or a **regular language**
- Any set of strings that can be defined if we add recursion is called **context-free language (CFL)**.

Context-Free Grammars

- **Context-Free Grammars**
 - Developed by Noam Chomsky in the mid-1950s
 - Language generators, meant to describe the syntax of natural languages
 - Define the class of context-free languages
 - Programming languages are contained in the class of CFL' s.

Backus-Naur Form (BNF)

- A notation to describe the syntax of programming languages.
- Named after
 - John Backus – Algol 58
 - Peter Naur – Algol 60
- A ***metalanguage*** is a language used to describe another language.
- **BNF is a metalanguage used to describe PLs.**

BNF Fundamentals

- BNF uses abstractions for syntactic structures.

$\langle \text{LHS} \rangle \rightarrow \langle \text{RHS} \rangle$

- LHS: abstraction being defined
- RHS: definition
- “ \rightarrow ” means “can have the form”
- Sometimes $::=$ is used for \rightarrow

BNF Fundamentals

- Example, Java *assignment* statement can be represented by the abstraction **<assign>**
- **<assign> → <var> = <expression>**
- This is a *rule* or *production*
- Here, **<var>** and **<expression>** must also be defined.
- example instances of this abstraction can be
`total = sub1 + sub2`
`myVar = 4`

BNF Fundamentals

- These abstractions are called **Variables** or **Nonterminals** of a Grammar.
- Lexemes and tokens are the **Terminals** of a grammar.
- Nonterminals are often enclosed in angle brackets
- Examples of BNF rules:
`<ident_list> → identifier | identifier, <ident_list>`
`<if_stmt> → if <logic_expr> then <stmt>`

BNF Fundamentals

- A formal definition of **rule**:

A **rule** has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals

$$\langle \text{LHS} \rangle \rightarrow \langle \text{RHS} \rangle$$

- **Grammar**: a finite non-empty set of rules

An initial example

- Consider the sentence “**Mary greets John**”
- A simple grammar
 - <sentence> ::= <subject><predicate>**
 - <subject> ::= Mary**
 - <predicate> ::= <verb><object>**
 - <verb> ::= greets**
 - <object> ::= John**

Alternations

- Multiple definitions can be separated by | (OR).
<object> ::= John | Alfred
- This adds “Mary greets Alfred” to legal sentences
<subject> ::= Mary | John | Alfred
<object> ::= Mary | John | Alfred
- Alternation to the previous grammar
<sentence> ::= <subject><predicate>
<subject> ::= <noun>
<predicate> ::= <verb><object>
<verb> ::= greets
<object> ::= <noun>
<noun> ::= Mary | John | Alfred

Infinite Number of Sentences

**<object> ::= John |
 John again |
 John again and again |
 **

Instead use recursive definition

**<object> ::= John |
 John <repeat factor>
<repeat factor> ::= again |
 again and <repeat factor>**

A rule is recursive if its LHS appears in its RHS

Simple example for PLs

- How you can describe simple arithmetic?

<expr> ::= <expr> <operator> <expr> | <var>

< operator > ::= + | - | * | /

<var> ::= a | b | c | ...

<var> ::= <signed number>

<signed number> ::= + <number> | - <number>

<number> ::= <number> <digit> | <digit>

....

....

Identifiers

<identifier> → <letter> |
<identifier><letter> |
<identifier><digit>

PASCAL/Ada If Statement

<if_stmt> → if <logic_expr> then <stmt>

**<if_stmt> → if <logic_expr> then <stmt>
 else <stmt>**

Or

<if_stmt> → if <logic_expr> then <stmt>

**| if <logic_expr> then <stmt> else
 <stmt>**

Grammars and Derivations

- A grammar is a generative device for defining languages
- The sentences of the language are **generated** through a sequence of applications of the rules, starting from the special nonterminal called ***start symbol***.
- Such a generation is called a **derivation**.
- Start symbol represents a complete program. So it is usually named as **<program>**.

An Example Grammar

`<program>` \rightarrow `begin <stmt_list> end`
`<stmt_list>` \rightarrow `<stmt> |`
 `<stmt> ; <stmt_list>`
`<stmt>` \rightarrow `<var> := <expression>`
`<var>` \rightarrow `A | B | C`
`<expression>` \rightarrow `<var> |`
 `<var> <arith_op> <var>`
`<arith_op>` \rightarrow `+ | - | * | /`

Derivation

- In order to check if a given string represents a valid program in the language, we try to derive it in the grammar.
- Derivation starts from the start symbol `<program>`.
- At each step we replace a nonterminal with its definition (RHS of the rule).

Derivations

- Every string of symbols in a derivation is a ***sentential form***
- A ***sentence*** is a sentential form that has only terminal symbols
- A ***leftmost derivation*** is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

An Example Derivation

An Example Derivation

Derive string: **begin A := B; C := A * B end**

<program> ⇒ begin <stmt_list> end
⇒ begin <stmt> ; <stmt_list> end
⇒ begin <var> := <expression>; <stmt_list> end
⇒ begin A := <expression>; <stmt_list> end
⇒ begin A := B; <stmt_list> end
⇒ begin A := B; <stmt> end
⇒ begin A := B; <var> := <expression> end
⇒ begin A := B; C := <expression> end
⇒ begin A := B; C := <var><arith_op><var> end
⇒ begin A := B; C := A <arith_op> <var> end
⇒ begin A := B; C := A * <var> end
⇒ begin A := B; C := A * B end

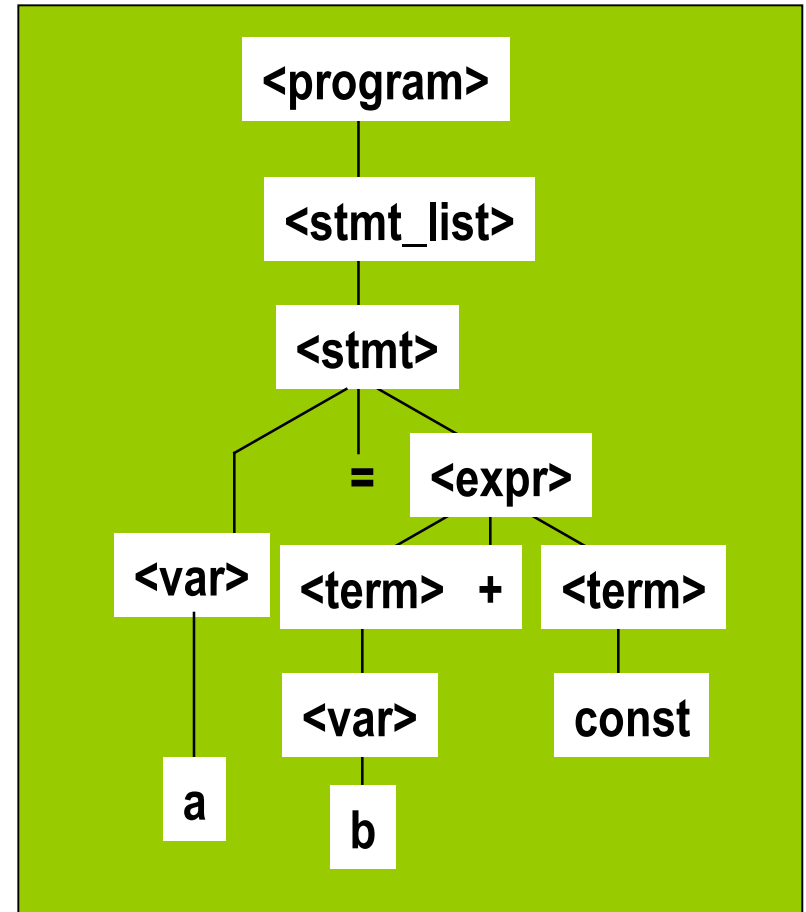
sentential
form

If always the leftmost nonterminal is replaced, then it is called **leftmost derivation**.

Parse Tree

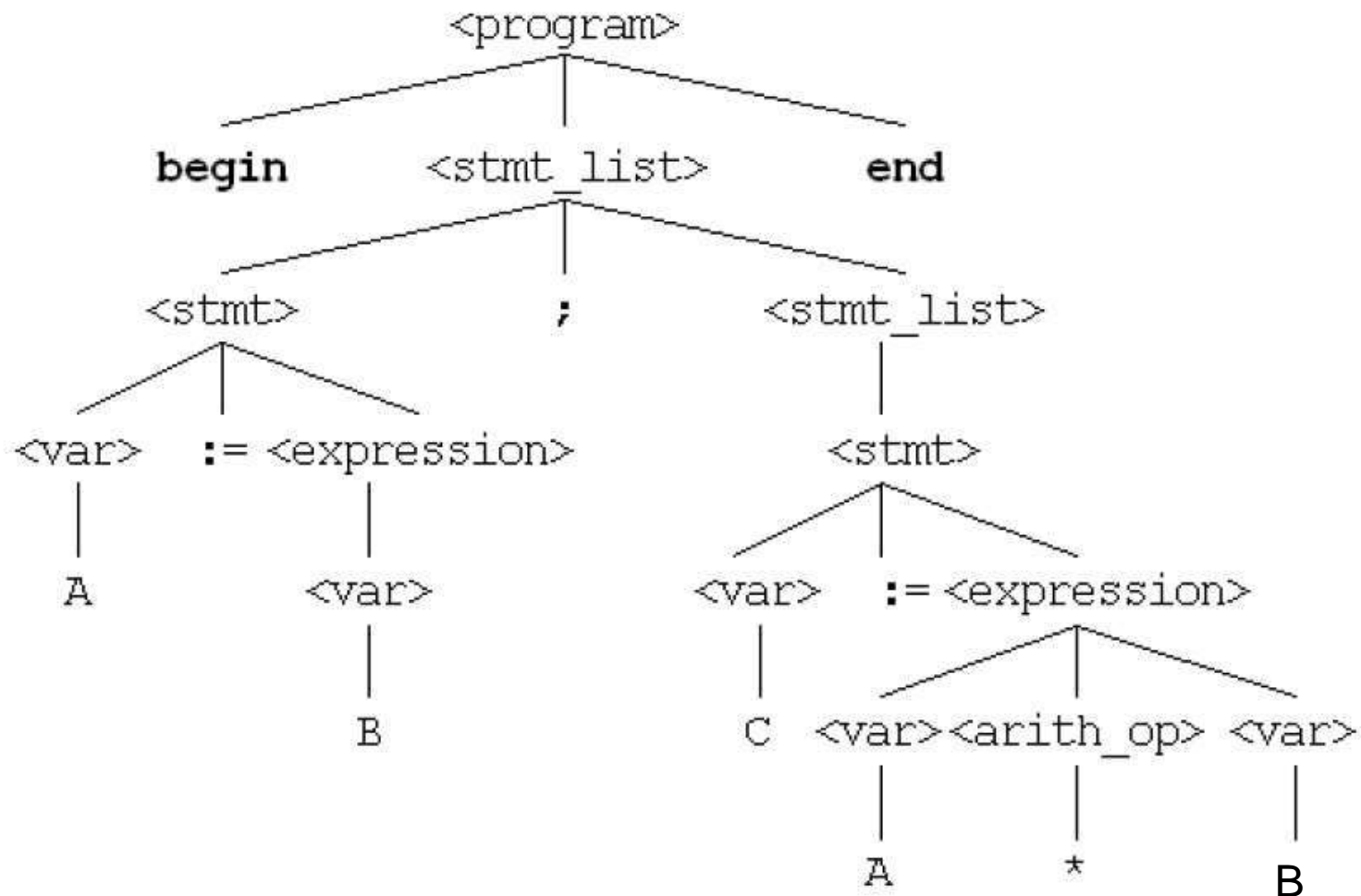
- A hierarchical representation of a derivation

<program> → **<stmt_list>**
<stmt_list> → **<stmt>**
 | **<stmt>** ; **<stmt_list>**
<stmt> → **<var>** = **<expr>**
<var> → **a** | **b** | **c** | **d**
<expr> → **<term>** + **<term>**
 | **<term>** - **<term>**
<term> → **<var>** | **const**



Parse Tree of the Example

Parse Tree of the Example



Ambiguity (Belirsizlik) in Grammars

- A grammar is ***ambiguous*** if and only if it generates a sentential form that has two or more distinct parse trees

Example

- Given the following grammar

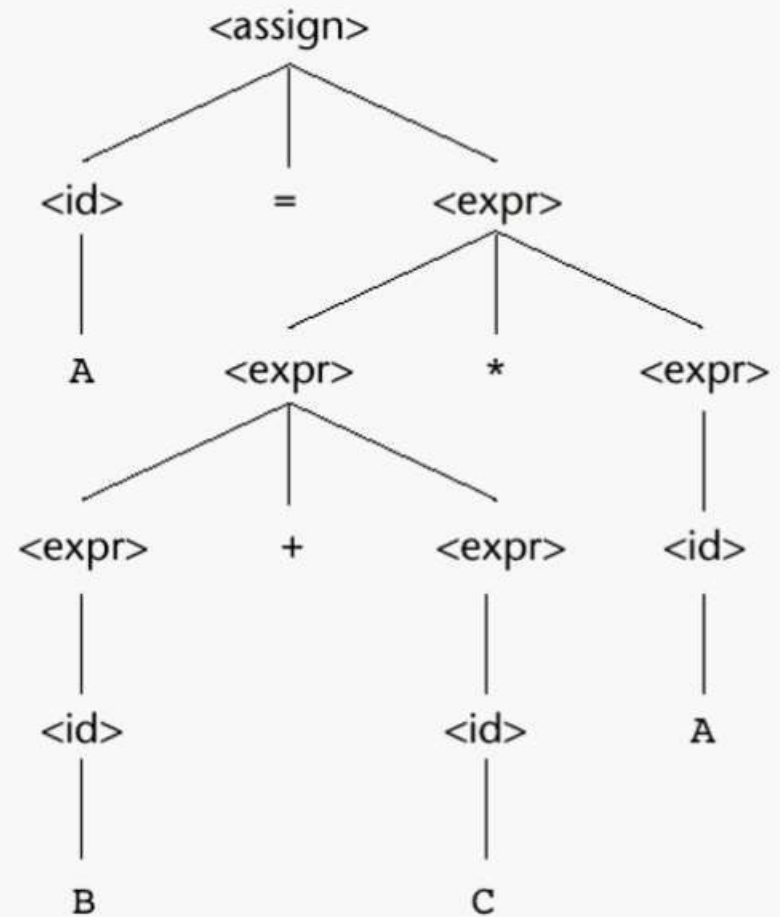
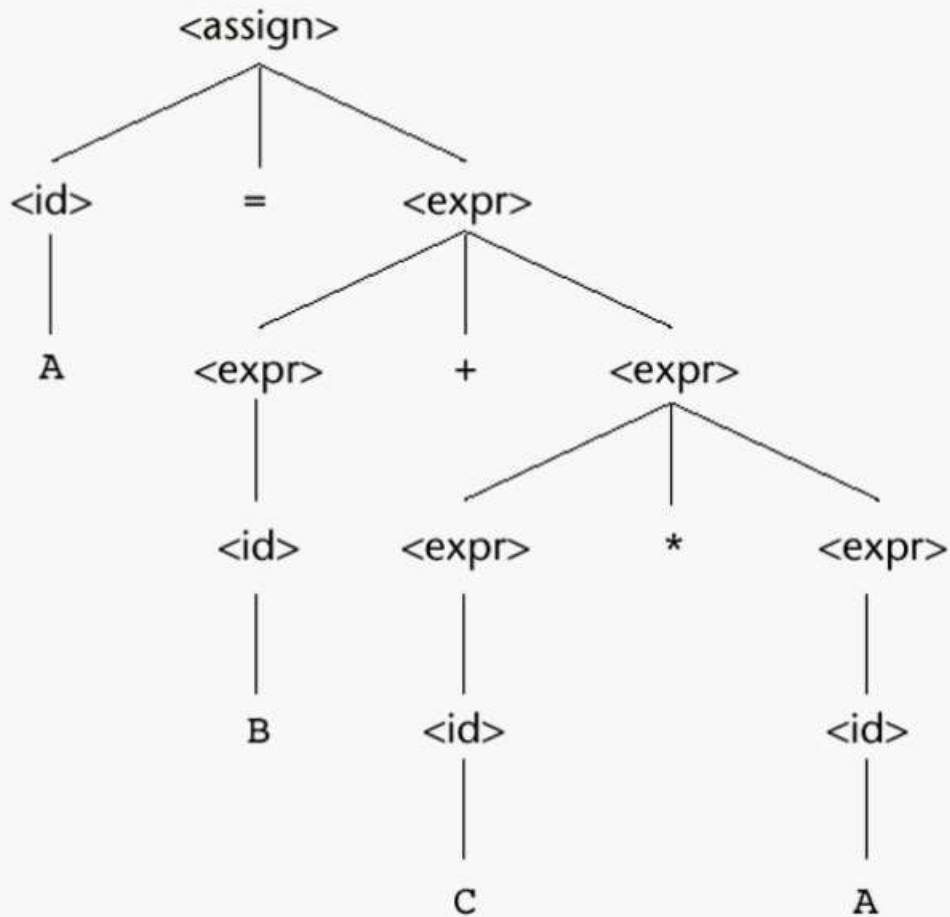
$\langle \text{assign} \rangle ::= \langle \text{id} \rangle = \langle \text{expr} \rangle$

$\langle \text{id} \rangle ::= A \mid B \mid C$

**$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{expr} \rangle$
 $\mid \langle \text{expr} \rangle * \langle \text{expr} \rangle$
 $\mid (\langle \text{expr} \rangle)$
 $\mid \langle \text{id} \rangle$**

Parse Tree(s) for $A = B + C * A$

Parse Trees for $A = B + C * A$



Ambiguity

- The grammar of a PL must not be ambiguous
- There are solutions for correcting the ambiguity
 - Operator precedence
 - Associativity rules

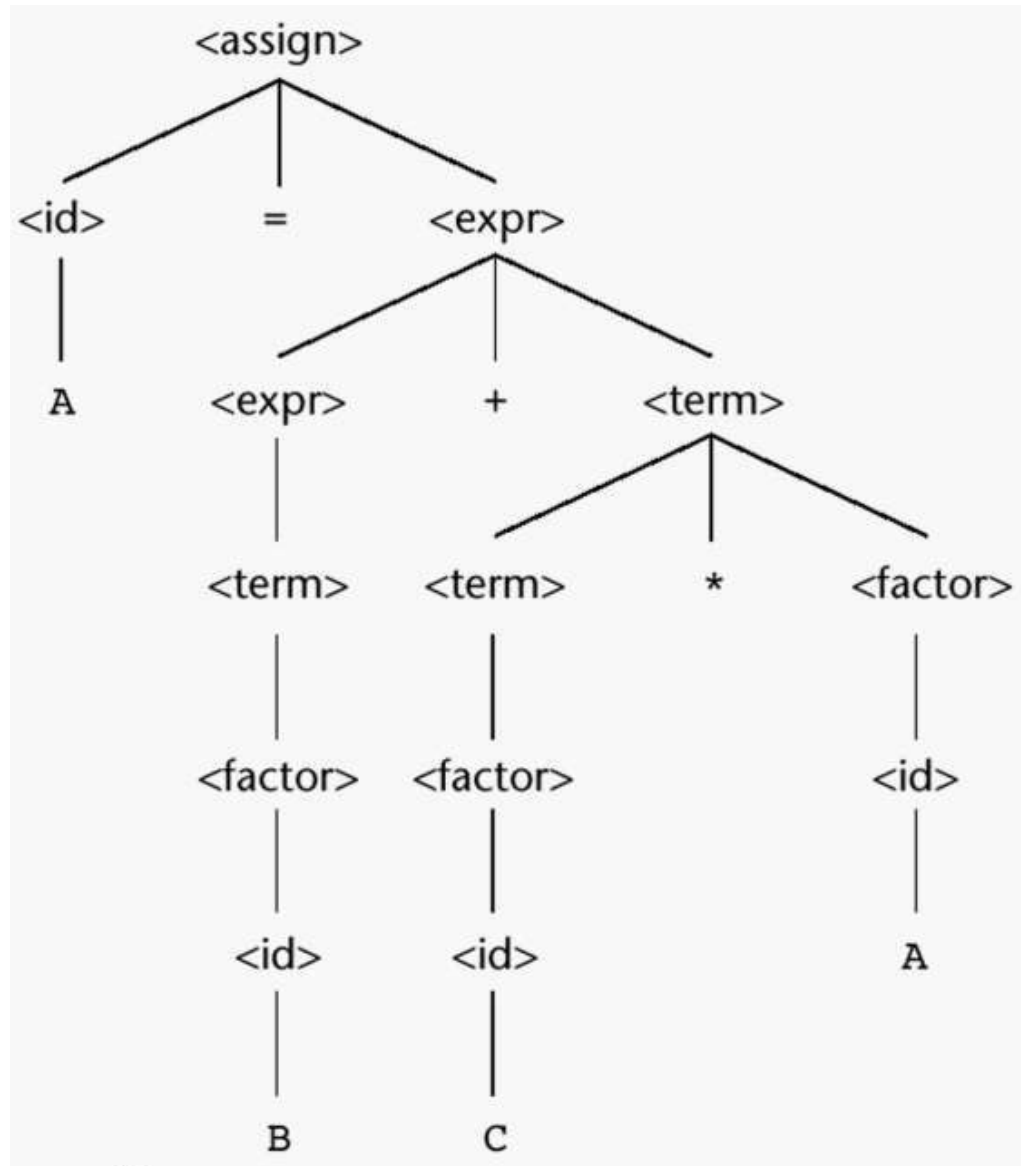
Operator Precedence

- In mathematics * operation has a higher precedence than +
- This can be implemented with extra nonterminals

```
<assign> ::= <id> = <expr>  
<id> ::= A | B | C  
<expr> ::= <expr> + <expr>  
           | <expr> * <expr>  
           | (<expr>)  
           | <id>
```

```
<assign> ::= <id> = <expr>  
<id> ::= A | B | C  
<expr> ::= <expr> + <term>  
           | <term>  
<term> ::= <term> * <factor>  
           | <factor>  
<factor> ::= (<expr>)  
           | <id>
```

Unique Parse Tree for $A = B + C * A$



Associativity of Operators

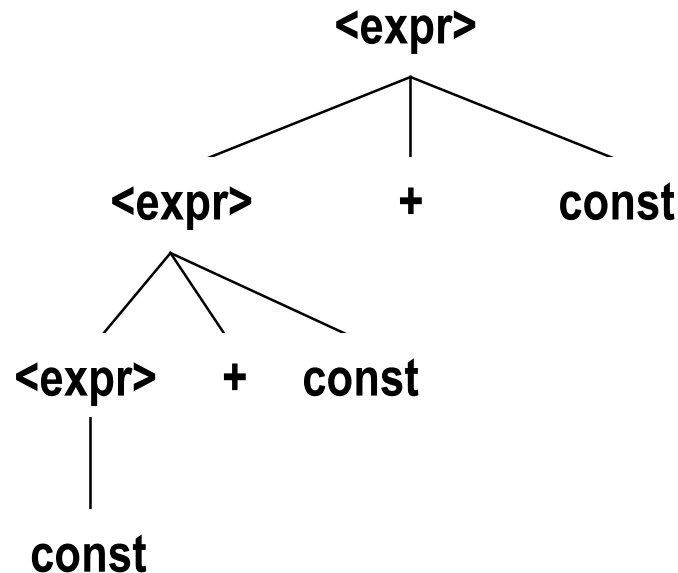
- What about equal precedence operators?
- In math addition and multiplication are associative
 $A+B+C = (A+B)+C = A+(B+C)$
- However computer arithmetic may not be associative
- Ex: for floating point addition where floating points values store 7 digits of accuracy, adding eleven numbers together where one of the numbers is 10^7 and the others are 1 result would be $1.000001 * 10^7$ only if the ten 1s are added first
- Subtraction and division are not associative
 $A/B/C/D = ? \quad ((A/B)/C)/D \neq A/(B/(C/D))$

Associativity of Operators

- Operator associativity can also be indicated by a grammar

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \text{const}$ (ambiguous)

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \text{const} \mid \text{const}$ (unambiguous)



Associativity (birleşirlik)

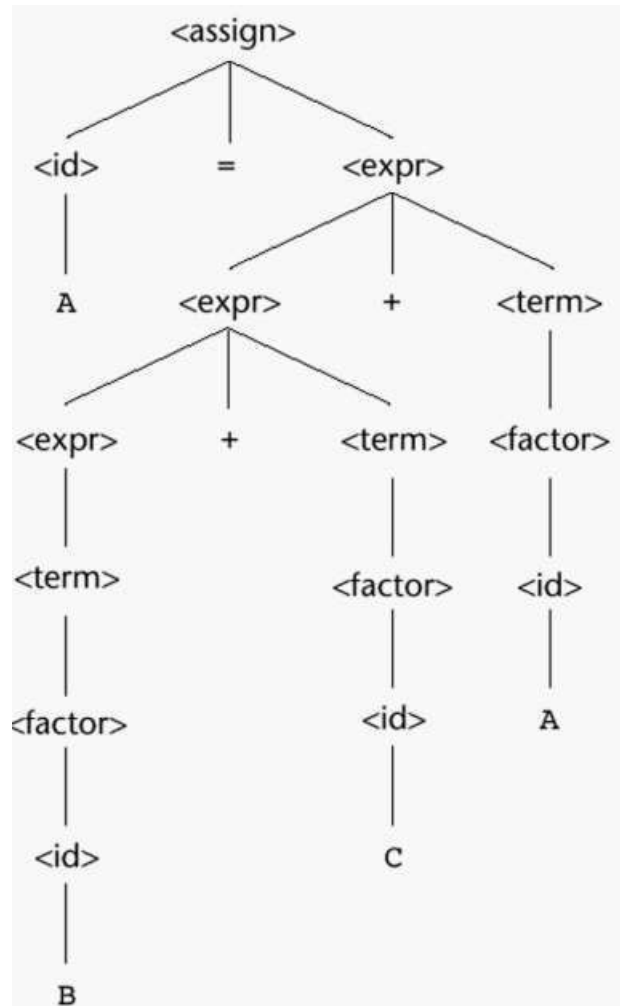
- In a BNF rule, if the LHS appears at the beginning of the RHS, the rule is said to be **left recursive**
- **Left recursion specifies left associativity**

$$\begin{aligned} \langle \mathbf{expr} \rangle &::= \langle \mathbf{expr} \rangle + \langle \mathbf{term} \rangle \\ &| \langle \mathbf{term} \rangle \end{aligned}$$

- Similar for the right recursion
- In most of the languages exponential is defined as a right associative operation

$$\begin{aligned} \langle \mathbf{factor} \rangle &::= \langle \mathbf{expr} \rangle ** \langle \mathbf{factor} \rangle \\ &| \langle \mathbf{expr} \rangle \\ \langle \mathbf{expr} \rangle &::= (\langle \mathbf{expr} \rangle) \\ &| \langle \mathbf{id} \rangle \end{aligned}$$

A parse tree for $A = B + C + A$ illustrating the associativity of addition



Left associativity

Left addition is lower than the right addition

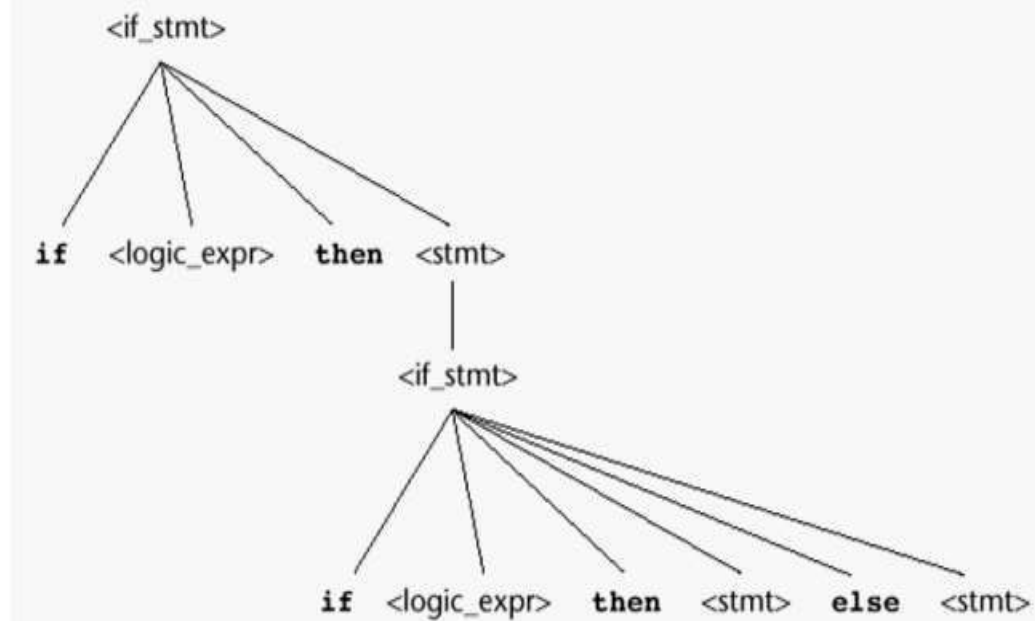
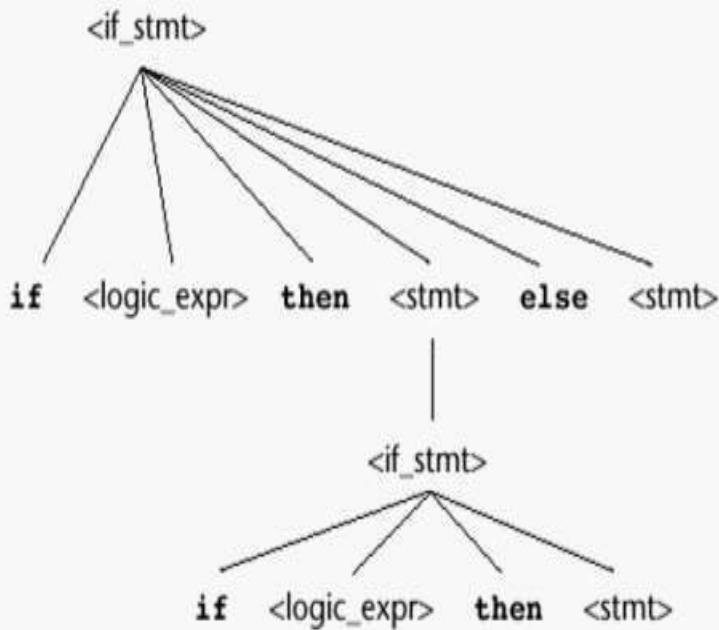
Is this ambiguous?

<stmt> ::= <if_stmt>

<if_stmt> ::= if <logic_expr> then <stmt>

| if <logic_expr> then <stmt> else <stmt>

Derive for : If C1 then if C2 then A else B



An Unambiguous grammar for “if then else”

- Dangling else problem: there are more if then else
- To design an unambiguous if-then-else statement we have to decide which `if` a dangling `else` belongs to
- Most PL adopt the following rule:
 - “an else is matched with the closest previous unmatched if statement”
 - (unmatched if = else-less if)

Has a unique parse tree



```
<stmt> ::= <matched> | <unmatched>  
<matched> ::= if <logic_expr> then <matched> else <matched>  
           | any non-if-statement  
<unmatched> ::= if <logic_expr> then <stmt>  
               | if <logic_expr> then <matched> else <unmatched>
```

Draw the parse tree

```
<stmt> ::= <matched> | <unmatched>  
<matched> ::= if <logic_expr> then <matched> else <matched>  
            | any non-if-statement  
<unmatched> ::= if <logic_expr> then <stmt>  
                | if <logic_expr> then <matched> else <unmatched>
```

If C1 then if C2 then A else B

Extended BNF

- *EBNF: Same power but more convenient*

- Optional parts are placed in brackets []

[X] : X is optional (0 or 1 occurrence)

<writeln> ::= WRITELN [(<item_list>**)]**

<selection> ::= if (<expr>**) **<stmt>** [else**<stmt>**]**

- Repetitions (0 or more) are placed inside braces { }

{X}: 0 or more occurrences

<identlist> = **<identifier> {,**<identifier>**}**

- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

(X1 | X2 | X3) : choose X1 or X2 or X3

<term> → **<term> (+ | -) const**

BNF vs Extended BNF

- BNF

```
< expr> ::= <expr> + <term>
          | <expr> - <term>
          | <term>
<term> ::= <term> * <factor>
          | <term> / <factor>
          | <factor>
<factor> ::= <expr> **
<factor>
          | <expr>
<expr> ::= (<expr>)
          | <id>
```

BNF vs Extended BNF

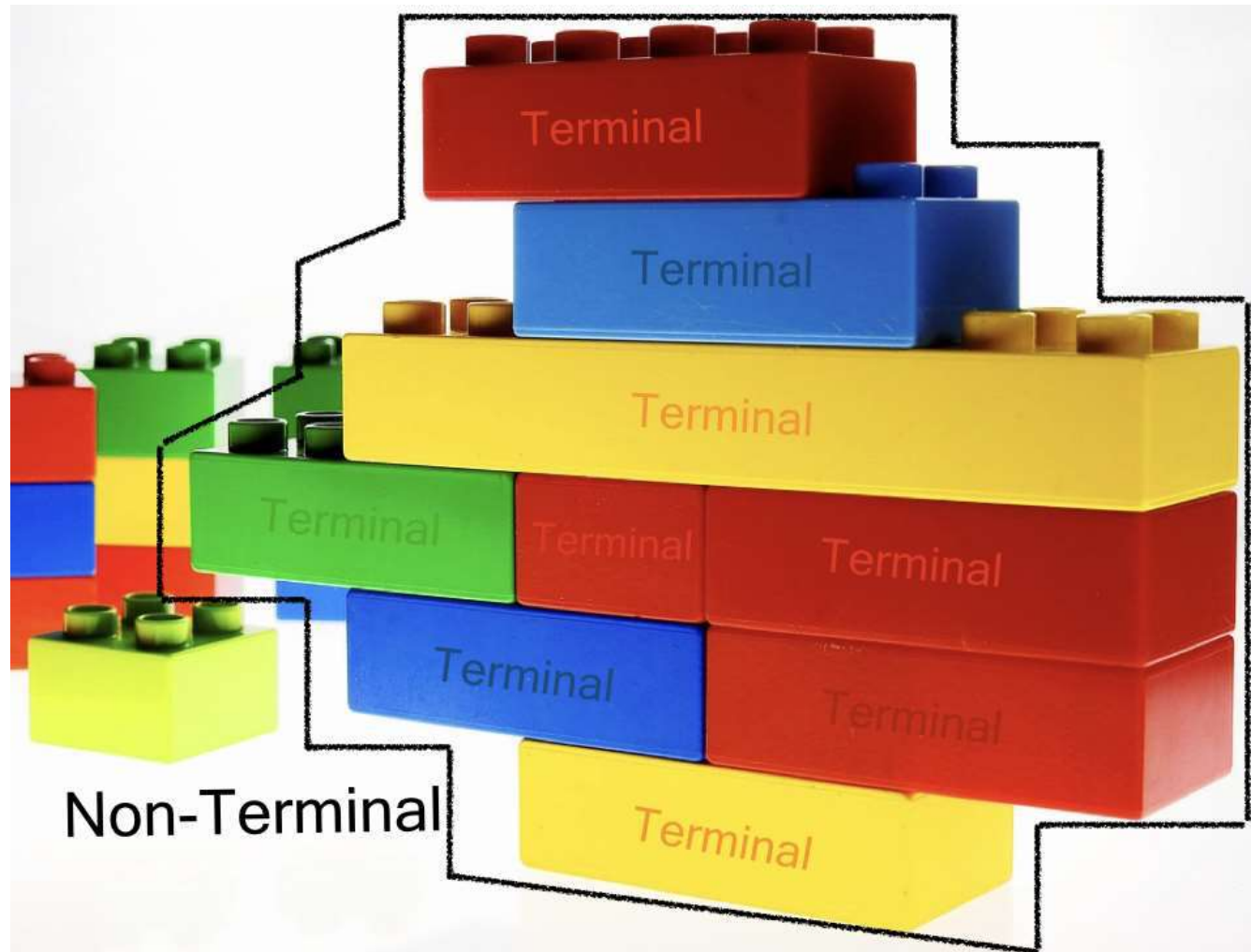
- EBNF

```
<expr> ::= <term> {(+ | -) <term>}  
<term> ::= <factor> {(* | /) <factor>}  
<factor> ::= <expr> {**<expr>}  
<expr> ::= (<expr>)  
           | id
```

Recent Variations in EBNF

- Alternative RHSs are put on separate lines
- Use of a colon instead of =>
- Use of `opt` for optional parts
- Use of `oneof` for choices

Additional Notes on Terminals and NonTerminals



Additional Notes on Terminals and NonTerminals

- Terminals are the smallest block we consider in our grammars.
- A terminal could be either:
 - a quoted literal
 - a regular expression
 - a name referring to a terminal definition.

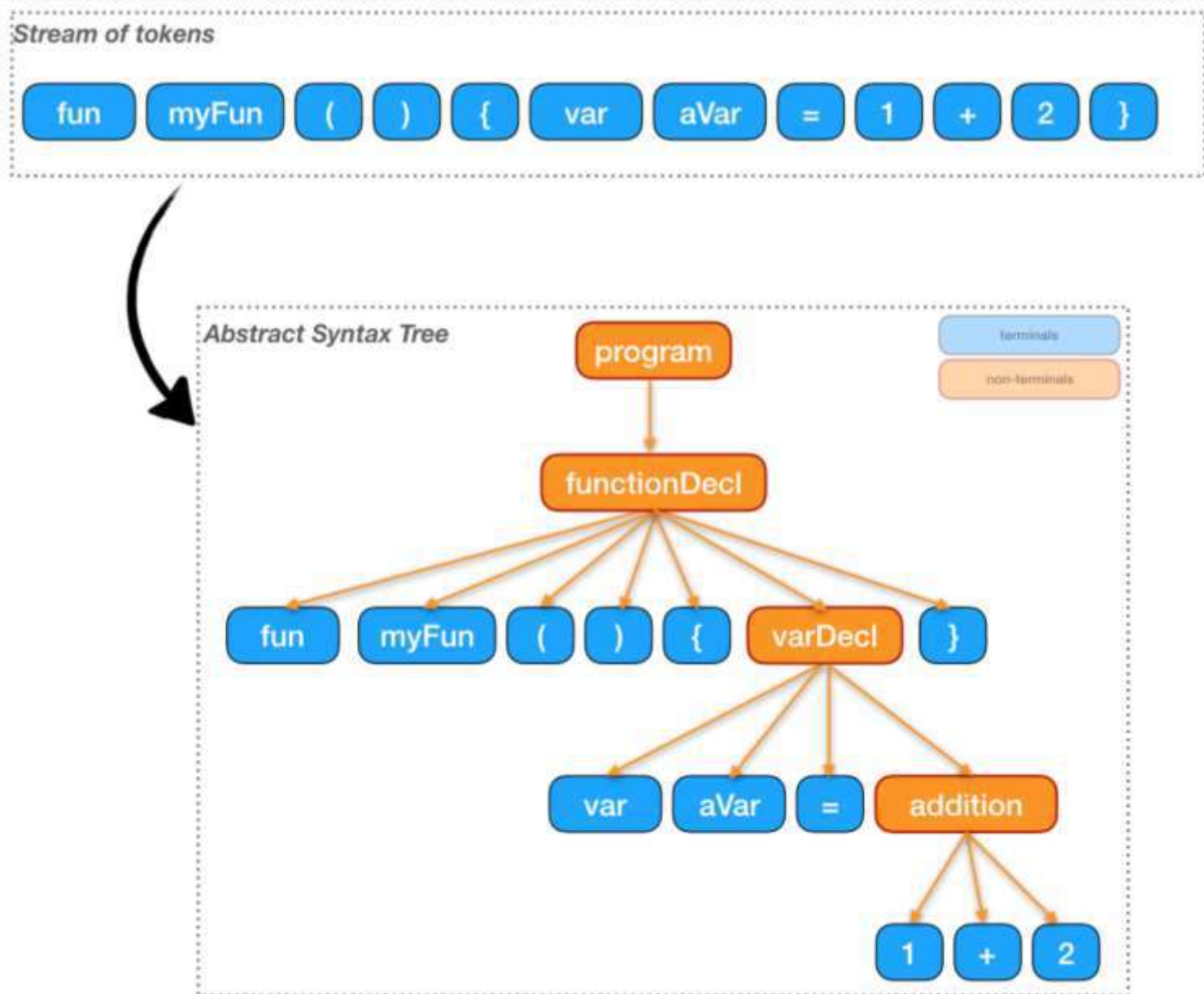
Terminals

- Let's see some typical terminals:
 - *identifiers*: these are the names used for variables, classes, functions, methods and so on.
 - *keywords*: almost every language uses keywords. They are exact strings that are used to indicate the start of a definition (think about `class` in Java or `def` in Python), a modifier (`public`, `private`, `static`, `final`, etc.) or control flow structures (`while`, `for`, `until`, etc.)

Terminals

- *literals*: these permit to define values in our languages. We can have string literals, numeric literal, char literals, boolean literals (but we could consider them keywords as well), array literals, map literals, and more, depending on the language
- *separators and delimiters*: like colons, semicolons, commas, parenthesis, brackets, braces
- *whitespaces*: spaces, tabs, newlines.
- *comments*

Terminals and Non-terminals

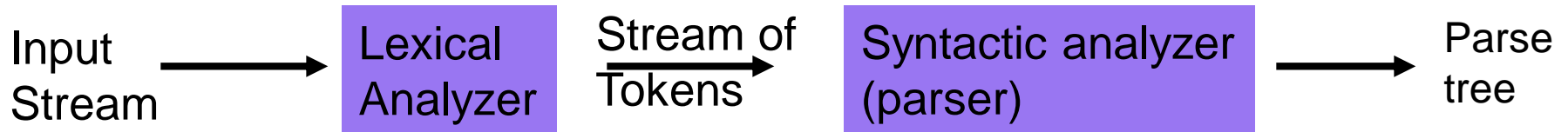


Non-terminals

- Examples of non-terminals are:
 - *program/document*: represent the entire file
 - *module/classes*: group several declarations together
 - *functions/methods*: group statements together
 - *statements*: these are the single instructions. Some of them can contain other statements. Example : loops
 - *expressions*: are typically used within statements and can be composed in various ways

Lex – A tool for lexical analysis

Lexical and syntactic analysis



- **Lexical analyzer:** scans the input stream and converts sequences of characters into tokens.
(char list) → (token list)
- **Lex** is a tool for writing lexical analyzers.
- **Syntactic Analysis:** reads tokens and assembles them into language constructs using the grammar rules of the language.
- **Yacc** is a tool for constructing parsers.

Lexical Analysis

- What do we want to do? Example:

```
if (i == j)
    z = 0;
else
    z = 1;
```

- The input is just a sequence of characters:

```
if (i == j) \n\tz = 0; \nelse \n\tz = 1;
```

- **Goal:** Partition input strings into substrings
 - And classify them according to their role

Tokens

- Output of lexical analysis is a list of tokens
- A token is a syntactic category
 - In English:
 - noun, verb, adjective, ...
 - In a programming language:
 - Identifier, Integer, Keyword, Whitespace, ...
- Parser relies on token distinctions:
 - e.g., identifiers are treated differently than keywords

Example

- Recall:

```
if (i == j) \n\tz = 0; \nelse \n\tz = 1;
```

- Token-lexeme pairs returned by the lexer:

- <Keyword, “if”>
- <Whitespace, “ ”>
- <OpenPar, “(”>
- <Identifier, “i”>
- <Whitespace, “ ”>
- <Relation, “==”>
- <Whitespace, “ ”>
- ...

Implementation of A Lexical Analyzer

- The lexer usually discards **uninteresting** tokens that don't contribute to parsing.
- Examples: Whitespaces, Comments
 - Exception: which language cares about whitespaces?
- The goal is to partition the string. That is implemented by reading left-to-right, recognizing one token at a time.
- Lexical structure described can be specified using ***regular expressions***.

Regular Expressions

In computing, a **regular expression**, also referred to as "regex" or "regexp", provides a concise and flexible means for **matching strings of text**, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a **regular expression processor**.

http://en.wikipedia.org/wiki/Regular_expression

Regular Expressions

- Regular expressions are used in many programming languages and software tools to specify patterns and match strings.
- Regular expressions are well suited for matching lexemes in programming languages.
- Regular expressions use a finite alphabet of symbols and defined by the operators
 - (i) union
 - (ii) concatenation
 - (iii) Kleene closure.

Lex – A Lexical Analyzer Generator

M. E. Lesk and E. Schmidt

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. This manual, however, will only discuss generating analyzers in C on the UNIX system, which is the only supported form of Lex under UNIX Version 7. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

Introduction

- **Lex:**

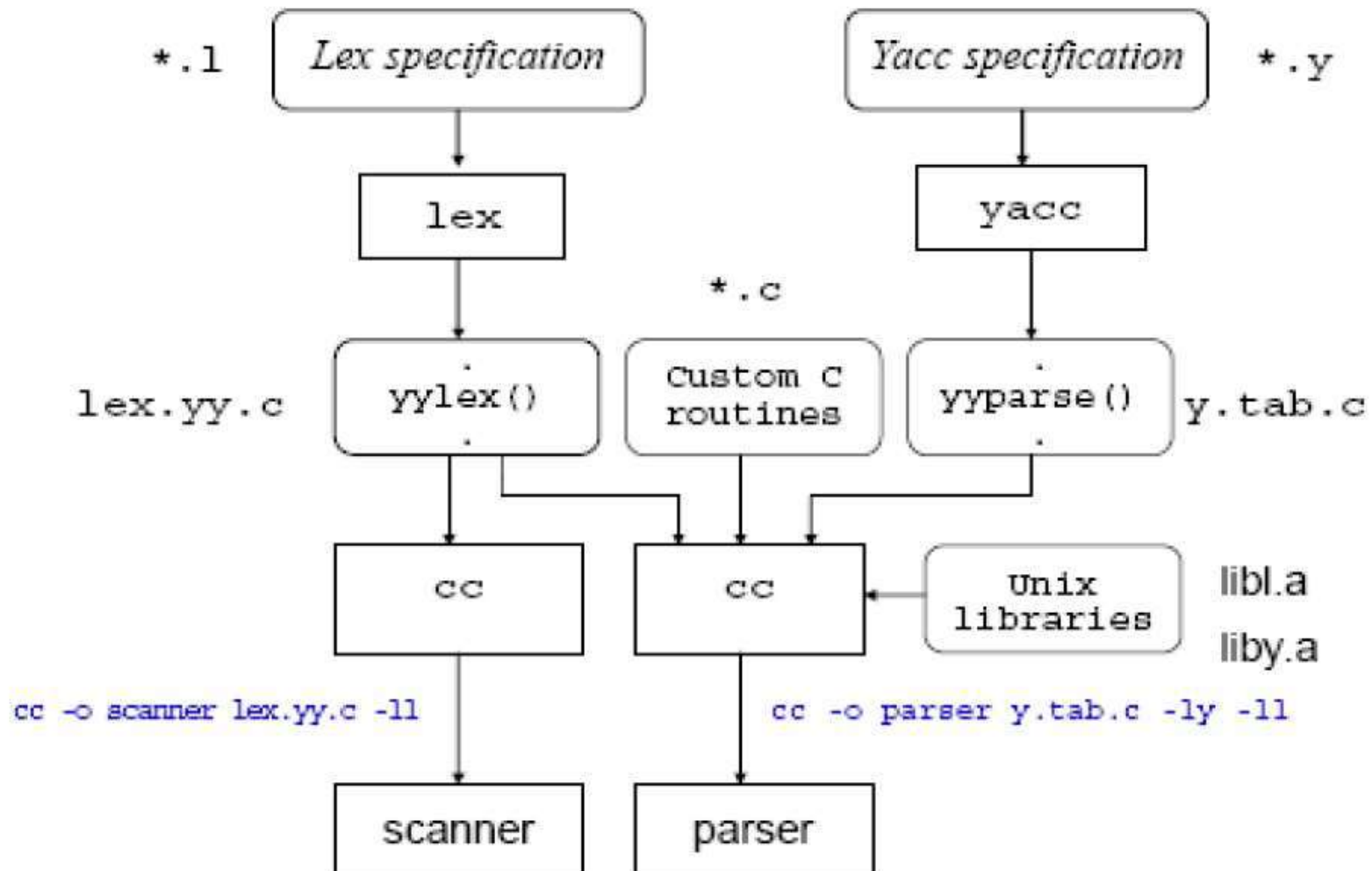
- reads in a collection of regular expressions, and uses it to write a C or C++ program that will perform lexical analysis. This program is almost always faster than one you can write by hand.

- **Yacc:**

- reads in the output from lex and parses it using its own set of regular expression rules. This is almost always slower than a handwritten parser, but much faster to implement.

Yacc stands for “Yet Another Compiler Compiler”.

Using `lex` and `yacc` tools



Running `lex`

On Unix system

```
$ lex mylex.l
```

it will create `lex.yy.c`

then type

```
$ gcc -o mylex lex.yy.c -lfl
```

The open-source version of `lex` is called
“`flex`”

Using `lex`

- Contents of a `lex` program:

Declarations

`%%`

Translation rules

`%%`

Auxiliary functions

- The declarations section can contain declarations of variables, manifest constants, and regular definitions. The declarations section can be empty.
- The translation rules are each of the form
`pattern {action}`
 - Each pattern is a regular expression which may use regular definitions defined in the declarations section.
 - Each action is a fragment of C-code.
- The auxiliary functions section starting with the second `%%` is optional. Everything in this section is copied directly to the file `lex.yy.c` and can be used in the actions of the translation rules.

Simple lex program (ex0.1)

ex0.1 : (edit with emacs, vi, etc.)

```
%%
```

```
. | \n  ECHO;
```

```
%%
```

(\$ is the unix prompt)

```
$lex ex0.1
```

```
$gcc -o ex0 lex.yy.c -lfl
```

```
$ls
```

```
ex0 ex0.1 lex.yy.c
```

Simple lex program (ex0.1)

```
$vi test0
```

```
$cat test0
```

```
ali
```

```
Veli
```

```
$ cat test0 | ex0      (or $ex0 < test0)
```

```
ali
```

```
Veli
```

Simply echos the input file contents

Example lex program (ex1.1)

Ex1.1 :

%%

zippy printf("I RECOGNIZED ZIPPY");

\$cat test1

zippy

ali zip

veli and zippy here

zipzippy

ZIP

\$cat test1 | ex1

I RECOGNIZED ZIPPY

ali zip

veli and I RECOGNIZED ZIPPY here

zipI RECOGNIZED ZIPPY

ZIP

Example lex program (ex2.1)

```
%%
```

```
zip printf("ZIP");  
zippy printf("ZIPPY");
```

```
$cat test1 | ex2
```

```
ZIPPY
```

```
ali ZIP
```

```
veli and ZIPPY here
```

```
ZIPZIPPY
```

lex matches the input string the longest regular expression possible!

Example lex program (ex3.1)

```
%%
```

```
monday|tuesday|wednesday|thursday|friday|  
saturday|sunday printf("<%s is a day.>",  
    yytext);
```

```
$cat test3
```

```
today is wednesday september 27
```

```
$ex3 < test3
```

```
today is <wednesday is a day> september 27
```

- Lex declares an external variable called `yytext` which contains the matched string

Designing patterns

Designing the proper patterns in `lex` can be very tricky, but you are provided with a broad range of options for your regular expressions.

- `.` A dot will match any single character except a newline.
- `*`, `+` Star and plus used to match zero/one or more of the preceding expressions.
- `?` Matches zero or one copy of the preceding expression.

Designing patterns

- | A logical 'or' statement - matches either the pattern before it, or the pattern after.
- ^ Matches the very beginning of a line.
- \$ Matches the end of a line.
- / Matches the preceding regular expression, but only if followed by the subsequent expression.

Designing patterns

- **[]** Brackets are used to denote a character class, which matches any single character within the brackets. If the first character is a '^', this negates the brackets causing them to match any character except those listed. The '-' can be used in a set of brackets to denote a range.
- **" "** Match everything within the quotes literally - don't use any special meanings for characters.
- **()** Group everything in the parentheses as a single unit for the rest of the expression.

Regular expressions in `lex`

- `a` matches `a`
- `abc` matches `abc`
- `[abc]` matches `a`, `b` or `c`
- `[a-f]` matches `a`, `b`, `c`, `d`, `e`, or `f`
- `[0-9]` matches any digit
- `x+` matches one or more of `x`
- `x*` matches zero or more of `x`
- `[0-9]+` matches any integer
- `(...)` grouping an expression into a single unit
- `|` alternation (or)
- `(a|b|c)*` is equivalent to `[a-c]*`

Regular expressions in `lex`

- `x?` `x` is optional (0 or 1 occurrence)
- `if(def)?` matches `if` or `ifdef` (equivalent to `if|ifdef`)
- `[A-Za-z]` matches any alphabetical character
- `.` matches any character except newline character
- `\.` matches the `.` character
- `\n` matches the newline character
- `\t` matches the tab character
- `\\` matches the `\` character
- `[\t]` matches either a space or tab character
- `[^a-d]` matches any character other than `a`, `b`, `c` and `d`

Examples

- Real numbers, e.g., 0, 27, 2.10, .17
 - $[0-9]^* (\backslash .) ? [0-9]^+$
- To include an optional preceding sign:
 - $[+-] ? [0-9]^* (\backslash .) ? [0-9]^+$
- Integer or floating point number
 - $[0-9]^+ (\backslash . [0-9]^+) ?$
- Integer, floating point, or scientific notation.
 - $[+-] ? [0-9]^+ (\backslash . [0-9]^+) ? ([eE] [+-] ? [0-9]^+) ?$

A slightly more complex program (ex4.1)

```
%%
```

```
[\t ]+ ;
```

```
monday|tuesday|wednesday|thursday|friday|  
saturday|sunday printf("%s is a day.",yytext);  
[a-zA-Z]+ printf("<%s is not a day.>",yytext);
```

ex5.1

%%

[\t]+ ;

Monday|Tuesday|Wednesday|Thursday|Friday

printf("%s is a week day.",yytext) ;

Saturday|Sunday

printf("%s is a weekend.",yytext) ;

[a-zA-Z]+

printf("%s is not day.",yytext) ;

Structure of a `lex` program

Declarations

`%%`

Translation rules

`%%`

Auxiliary functions

- The declarations section can contain declarations of variables, manifest constants, and regular definitions. The declarations section can be empty.
- The translation rules are each of the form
`pattern {action}`
 - Each pattern is a regular expression which may use regular definitions defined in the declarations section.
 - Each action is a fragment of C-code.
- The auxiliary functions section starting with the second `%%` is optional. Everything in this section is copied directly to the file `lex.yy.c` and can be used in the actions of the translation rules.

Declarations

```
%%  
[+-]?[0-9]* (\.)? [0-9]+ printf("FLOAT");
```

The same lex specification can be written as:

```
digit [0-9]
```

```
%%  
[+-]? {digit}* (\.)? {digit}+ printf("FLOAT");
```

input: ab7.3c--5.4.3+d++5

output: abFLOATc-FLOATFLOAT+d+FLOAT

Declarations

`digit [0-9]`

`sign [+ -]`

`%%`

`float val;`

```
{sign}?{digit}* (\.)?{digit}+ {sscanf(yytext, "%f", &val);  
                                printf(">%f<", val);  
                                }
```

Input => Output

`ali-7.8veli => ali>-7.800000<veli`

`ali--07.8veli => ali->-7.800000<veli`

`+3.7.5 => >3.700000<>0.500000<`

Declarations

```
/* echo-uppercase-words.1 */  
%%  
[A-Z]+[ \t\n\.\,] printf("%s",yytext) ;  
. ; /* no action specified */
```

The scanner for the specification above echo all strings of capital letters, followed by a space tab (\t) or newline (\n) dot (\.) or comma (\,) to stdout, and all other characters will be ignored.

Input

```
Ali VELI A7, X. 12  
HAMI BEY a
```

Output

```
VELI X.  
HAMI BEY
```

Declarations

Declarations can be used in Declarations

```
/* def-in-def.1 */  
alphabetic [A-Za-z]  
digit [0-9]  
alphanumeric ({alphabetic}|{digit})  
%%  
{alphabetic}{alphanumeric}*  
    printf("Variable");  
\, printf("Comma");  
\{ printf("Left brace");  
\:|= printf("Assignment");
```

Lex file structure

Definitions

%%

Regular expressions and associated actions
(rules)

%%

User routines

Important Note: Do not leave extra spaces and/or empty lines at the end of the lex specification file.

Auxiliary functions

- The user sub-routines section is for any additional C or C++ code that you want to include. The only required line is:

```
main() { yylex(); }
```

- This is the main function for the resulting program.
- **Lex** builds the `yylex()` function that is called, and will do all of the work for you.
- Other functions here can be called from the rules section

Rule order

- If more than one regular expression match the same string the one that is defined earlier is used.

```
/* rule-order.1 */
```

```
%%
```

```
for printf("FOR") ;
```

```
[a-z]+ printf("IDENTIFIER") ;
```

for input

```
for count := 1 to 10
```

the output would be

```
FOR IDENTIFIER := 1 IDENTIFIER 10
```

Rule order

- However, if we swap the two lines in the specification file:

```
%%
```

```
[a-z]+ printf("IDENTIFIER") ;
```

```
for printf("FOR") ;
```

for the same input

the output would be

```
IDENTIFIER IDENTIFIER := 1 IDENTIFIER  
10
```

Example Number Identifications (ex6.1)

```
%%  
[\\t ]+ /* Ignore Whitespace */;  
  
[+-]?[0-9]+(\\. [0-9]+)?([eE][+-]?[0-9]+)?  
printf(" %s:number", yytext);  
  
[a-zA-Z]+  
printf(" %s:NOT number", yytext);  
%%  
main() { yylex(); }
```


Counting Words (ex7.1)

```
%{
int char_count = 0;
int word_count = 0;
int line_count = 0;
}%
word      [^ \t\n]+
eol       \n
%%
{word} {word_count++; char_count+=yyleng;}
{eol} {char_count++; line_count++;}
. char_count++;
%%
main() {
yylex();
printf("line_count = %d , word_count = %d,
char_count = %d\n", line_count, word_
count, char_count);
}
```

lex also provides a count **yyleng** of the number of characters matched

Counting words (cont'd.)

```
$ cat test8
```

```
how many words
```

```
and how many lines
```

```
are there
```

```
in this file
```

```
$ex7 < test8
```

```
line_count = 5, word_count =  
    12, char_count = 58
```

ex8.1

```
%%  
    int k;  
-?[0-9]+    {  
    k = atoi(yytext);  
    printf("%d", k%7 == 0 ? k+3 :k+1);  
}  
-?[0-9\.] +    ECHO;  
[A-Za-z][A-Za-z0-9]+    printf("<%s>",yytext);  
%%
```

ex9.1

```
%{  
int  lengs[100];  
%}  
%%  
[a-z]+ {lengs[yyvaleng]++ ;  
if(yyvaleng==1) printf("<%s> ", yytext); }  
. |  
\n ;  
%%  
yywrap()  
{  
    int i;  
printf("Lenght    No. words\n");  
for(i=0; i<100; i++) if(lengs[i] >0)  
printf("%5d%10d\n", i, lengs[i]);  
return(1) ;  
}
```

yywrap is called whenever lex reaches an end-of-file

```
WS      [ \t]+
```

```
%%
```

```
    int total=0;
```

```
I      total += 1;
```

```
IV     total += 4;
```

```
V      total += 5;
```

```
IX     total += 9;
```

```
X      total += 10;
```

```
XL     total += 40;
```

```
L      total += 50;
```

```
XC     total += 90;
```

```
C      total += 100;
```

```
CD     total += 400;
```

```
D      total += 500;
```

```
CM     total += 900;
```

```
M      total += 1000;
```

```
{WS}   |
```

```
\n     return total;
```

```
%%
```

```
int main (void) {
```

```
    int first, second;
```

```
    first = yylex ();
```

```
    second = yylex ();
```

```
    printf ("%d + %d = %d\n", first, second,  
            first+second);
```

```
    return 0;
```

```
}
```

romans.1

Yacc

Lexical vs. Syntactic Analysis

Phase	Input	Output
Lexer	Sequence of characters	Sequence of tokens
Parser	Sequence of tokens	Parse tree

- **Lex** is a tool for writing lexical analyzers.
- **Yacc** is a tool for constructing parsers.

The Functionality of the Parser

- Input: sequence of tokens from lexer
 - e.g., the **lex** files you will write in the 1st phase of your course project.
- Output: **parse tree** of the program
 - Also called an **abstract syntax tree**
- Output: **error** if the input is not valid
 - e.g., “parse error on line 3”

Example

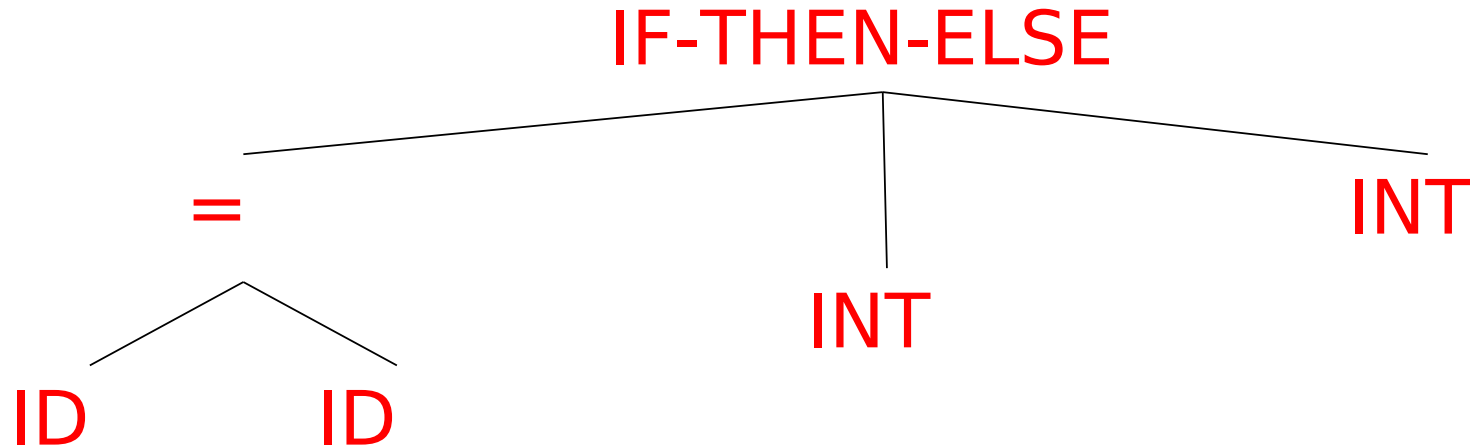
- Cool program text

`if x = y then 1 else 2 fi`

- Parser input (tokens)

IF ID = ID THEN INT ELSE INT FI

- Parser output (tree)



The Role of the Parser

- Not all sequences of tokens are programs
 - `then x * / + 3 while x ; y z then`
- The parser **must distinguish between valid and invalid sequences of tokens**
 - We need context free grammars.
- **Yacc** stands for **y**et **a**nother **c**ompiler to **c**ompiler.
 - Reads a specification file that codifies the grammar of a language and generates a parsing routine

YACC – Yet Another Compiler-Compiler

Stephen C. Johnson

Bell Laboratories,
Murray Hill, New Jersey 07974

ABSTRACT

Computer program input generally has some structure; in fact, every computer program which does input can be thought of as defining an “input language” which it accepts. The input languages may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, standard input facilities are restricted, difficult to use and change, and do not completely check their inputs for validity.

Yacc provides a general tool for controlling the input to a computer program. The Yacc user describes the structures of his input, together with code which is to be invoked when each such structure is recognized. Yacc turns such a specification into a subroutine which may be invoked to handle the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user’s application handled by this subroutine.

The input subroutine produced by Yacc calls a user supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or, if he wishes, in terms of higher level constructs such as names and numbers. The user supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy specification.

Yacc is written in C[7], and runs under UNIX. The subroutine which is output may be in C or in Ratfor[4], at the user’s choice; Ratfor permits translation of the output subroutine into portable Fortran[5]. The class of specifications accepted is a very general one, called LALR(1) grammars with disambiguating rules. The theory behind Yacc has been described elsewhere[1,2,3].

Yacc was originally designed to help produce the “front end” of compilers; in addition to this use, it has been successfully used in many application programs, including a phototypesetter language, a document retrieval system, a Fortran debugging system, and the Ratfor compiler.

Yacc

Yacc specification describes a Context Free Grammar (CFG), that can be used to generate a parser.

Elements of a CFG:

1. **Terminals**: tokens and literal characters,
2. **Variables (nonterminals)**: syntactical elements,
3. **Production rules**, and
4. **Start rule**.

Yacc

- Format of a production rule:
`symbol: definition`
`{action}`
`;`

Example:

$A \rightarrow Bc$ is written in yacc as **`a: b 'c' ;`**

Yacc Format

- Format of a `yacc` specification file:

`declarations`

`%%`

`grammar rules and associated actions`

`%%`

`C programs`

Declarations

To define tokens and their characteristics

%token: declare names of tokens

%left: define left-associative operators

%right: define right-associative operators

%nonassoc: define operators that may not
associate with themselves

%type: declare the type of variables

Declarations

%union: declare multiple data types for semantic values

%start: declare the start symbol (default is the first variable in rules)

%prec: assign precedence to a rule

% {

C declarations directly copied to the resulting C program

% } (e.g., variables, types, macros...)

A simple yacc specification to accept $L = \{ a^n b^n \mid n \geq 1 \}.$

```
/*anbn0.y */  
%token A B  
%%  
start: anbn '\n' {return 0;}  
anbn: A B  
| A anbn B  
;  
%%  
#include "lex.yy.c"
```

lex – yacc pair

```
/* anbn0.l */  
%%  
a return (A) ;  
b return (B) ;  
. return (yytext[0]) ;  
\n return ('\n') ;
```

```
/*anbn0.y */  
%token A B  
%%  
start: anbn '\n' {return 0;}  
anbn: A B  
| A anbn B  
;  
%%  
#include "lex.yy.c"
```

Running yacc on Linux

In Linux there is no `liby.a` library for `yacc` functions
You have to add the following lines to end of your `yacc` specification file

```
int yyerror(char *s)
{
    printf("%s\n", s);
}
int main(void)
{
    yyparse();
}
```

Then type

```
gcc -o exe_file y.tab.c -lfl
```

Printing messages

If the input stream does not match **start**, the default message of "**syntax error**" is printed and program terminates.

However, customized error messages can be generated.

```
/*anbn1.y */
%token A B
%%
start: anbn '\n' {printf(" is in anbn\n");
                  return 0;}

anbn: A B
    | A anbn B
    ;
%%
#include "lex.yy.c"
yyerror(s)
char *s;
{ printf("%s, it is not in anbn\n", s);
}
```

Example Output

`$anbn`

`aabb`

`is in anbn`

`$anbn`

`acadbefbg`

`Syntax error, it is not in anbn`

A grammar to accept $L = \{a^n b^n \mid n \geq 0\}$.

```
/*anbn_0.y */
%token A B
%%
start: anbn '\n' {printf(" is in anbn_0\n");
                  return 0;}

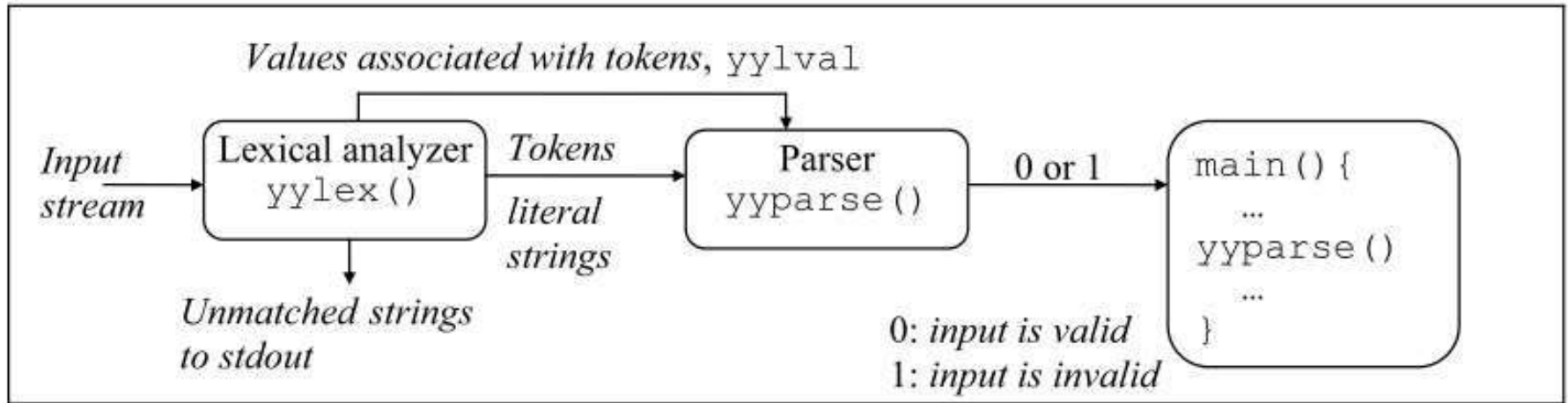
anbn: empty
    | A anbn B
    ;

empty: ;
%%
#include "lex.yy.c"
yyerror(s)
char *s;
{ printf("%s, it is not in anbn_0\n", s);
```

Recursive Rules

Although right-recursive rules can be used in **yacc**, **left-recursive rules are preferred**, and, in general, generate more efficient parsers.

yyval



yylex() function returns an integer, the token number, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable **yyval**.

yyval

The type of `yyval` is `int` by default. To change the type of `yyval` use macro `YYSTYPE` in the declarations section of a `yacc` specifications file.

```
%{  
#define YYSTYPE double  
%}
```

If there are more than one data types for token values, `yyval` is declared as a `union`.

yyval

Example with three possible types for **yyval**:

```
%union{  
double real; /* real value */  
int integer; /* integer value */  
char str[30]; /* string value */  
}
```

Example:

yytext="0012", type of yyval:int, value of yyval: 12

yytext="+1.70", type of yyval:float, value of yyval:1.7

Token types

- The type of associated values of tokens can be specified by **%token** as

%token <real> REAL

%token <integer> INTEGER

%token <str> IDENTIFIER STRING

- Type of variables can be defined by **%type** as

%type <real> real-expr

%type <integer> integer-expr

To return values for tokens from a lexical analyzer:

```
/* lexical-analyzer.l */
alphabetic [A-Za-z]
digit [0-9]
alphanumeric ({alphabetic}|{digit})

[+-]?{digit}* (\.)?{digit}+ {sscanf(yytext, %lf",
                                &yylval.real);
                                return REAL;
                                }
{alphabetic}{alphanumeric}* {strcpy(yylval.str, yytext);
                              return IDENTIFIER;
                              }
```

Positional assignment of values for items

\$\$: left-hand side

\$1: first item in the right-hand side

\$n: n^{th} item in the right-hand side

Example: Printing integers

```
/*print-int.l*/
%%
[0-9]+ {sscanf(yytext, "%d", &yyval);return(INTEGER);}
\n return(NEWLINE);
. return(yytext[0]);
```

```
/* print-int.y */
%token INTEGER NEWLINE
%%
lines: /* empty */
| lines NEWLINE
| lines line NEWLINE {printf("=%d\n", $2);}
| error NEWLINE {yyerror("Reenter:"); yyerrok;}
;
line: INTEGER {$$ = $1;}
;
%%
#include "lex.yy.c"
```

Yacc provides a special symbol for handling errors. The symbol is called `error` and it should appear within a grammar-rule.

Example continued

\$print-int

7

=7

007

=7

zippy

syntax error

Reenter:

—

Operator Precedence

- All of the tokens on the same line are assumed to have the same precedence level and associativity;
- The lines are listed in order of increasing precedence or binding strength.

```
%left '+' '-'  
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative.

Example: simple calculator - lex

```
/* calculator.l */
integer      [0-9]+
dreal        ([0-9]*\.[0-9]+)
ereal        ([0-9]*\.[0-9]+[Ee][+-]?[0-9]+)
real         {dreal}|{ereal}
nl           \n
%%
[ \t]        ;
{integer}    { sscanf(yytext, "%d", &yylval.integer);
               return INTEGER;
             }
{real}       { sscanf(yytext, "%lf", &yylval.real);
               return REAL;
             }
\+           { return PLUS; }
\-           { return MINUS; }
\*           { return TIMES; }
\/           { return DIVIDE; }
\(           { return LP; }
\)           { return RP; }
{nl}         { extern int lineno; lineno++;
               return NL;
             }
.            { return yytext[0]; }
%%
int yywrap() { return 1; }
```

Example: simple calculator - yacc

```
/* calculator.y */
%{
#include <stdio.h>
%}

%union{ double    real; /* real value */
        int      integer; /* integer value */
        }

%token <real> REAL
%token <integer> INTEGER
%token PLUS MINUS TIMES DIVIDE LP RP NL
%type <real> rexpr
%type <integer> iexpr
%left PLUS MINUS
%left TIMES DIVIDE
%left UMINUS
```

Example: simple calculator - yacc

```
%%
lines: /* nothing */
      | lines line
      ;

line:  NL
      | iexpr NL
        { printf("%d) %d\n", lineno, $1); }
      | rexpr NL
        { printf("%d) %15.8lf\n", lineno, $1); }
      ;

iexpr: INTEGER
      | iexpr PLUS iexpr
        { $$ = $1 + $3; }
      | iexpr MINUS iexpr
        { $$ = $1 - $3; }
      | iexpr TIMES iexpr
        { $$ = $1 * $3; }
      | iexpr DIVIDE iexpr
        { if($3) $$ = $1 / $3;
          else { yyerror("divide by zero"); }
        }
      | MINUS iexpr %prec UMINUS
        { $$ = - $2; }
      | LP iexpr RP
        { $$ = $2; }
      ;
```

Example: simple calculator - yacc

```
rexpr: REAL
    | rexr PLUS rexr
      { $$ = $1 + $3;}
    | rexr MINUS rexr
      { $$ = $1 - $3;}
    | rexr TIMES rexr
      { $$ = $1 * $3;}
    | rexr DIVIDE rexr
      { if($3) $$ = $1 / $3;      else { yyerror( "divide by zero" );      }      }
    | MINUS rexr %prec UMINUS
      { $$ = - $2;}
    | LP rexr RP
      { $$ = $2;}
    | iexpr PLUS rexr
      { $$ = (double)$1 + $3;}
    | iexpr MINUS rexr
      { $$ = (double)$1 - $3;}
    | iexpr TIMES rexr
      { $$ = (double)$1 * $3;}
    | iexpr DIVIDE rexr
      { if($3) $$ = (double)$1 / $3;  else { yyerror( "divide by zero" );  }  }
  }
  | rexr PLUS iexpr
    { $$ = $1 + (double)$3;}
  | rexr MINUS iexpr
    { $$ = $1 - (double)$3;}
  | rexr TIMES iexpr
    { $$ = $1 * (double)$3;}
  | rexr DIVIDE iexpr
    { if($3) $$ = $1 / (double)$3;  else { yyerror( "divide by zero" );  }  }
  ;
```

```

/* lex specification */
%%
a return A;
b return B;
\n return NL;
. ;
%%
int yywrap() { return 1; }

```

Actions between Rule Elements

input: ab
output: 1452673

input: aa
output: 14526 syntax error

input: ba
output: 14 syntax error

```

/* yacc specification */
%{
#include <stdio.h>
}%
%token A B NL
%%
s: {printf("1");}
  a
  {printf("2");}
  b
  {printf("3");}
  NL
  {return 0;}
;
a: {printf("4");}
  A
  {printf("5");}
;
b: {printf("6");}
  B
  {printf("7");}
;
%%
#include "lex.yy.c"
int yyerror(char *s) {
  printf ("%s\n", s);
}

int main(void) { yyparse(); }

```

References

- <http://memphis.compilertools.net/interpreter.html>
- <http://www.opengroup.org/onlinepubs/007908799/xcu/yacc.html>
- <http://dinosaur.compilertools.net/yacc/index.html>

Functional Programming Languages

BBM 301 – Programming Languages

Introduction

- The design of the imperative languages is based directly on the *von Neumann architecture*
 - Efficiency is the primary concern, rather than the suitability of the language for software development
- The design of the functional languages is based on *mathematical functions*
 - A solid theoretical basis that is also closer to the user, but relatively unconcerned with the architecture of the machines on which programs will run

Mathematical Functions

- A mathematical function is a *mapping* of members of one set, called the *domain set*, to another set, called the *range set*
- In math functions, the evaluation order is controlled by recursion
- They don't have side effects: same value given the same arguments

Lambda Expressions

- Lambda expressions describe nameless functions
- A *lambda expression* specifies the parameter(s) and the mapping of a function in the following form

$$\lambda (x) \quad x * x * x$$

for the function $\text{cube } (x) = x * x * x$

- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression

e.g., $(\lambda (x) \quad x * x * x) (2)$

which evaluates to 8

Functional Forms

- A higher-order function, or *functional form*, is one that either takes functions as parameters or yields a function as its result, or both

Function Composition

- A functional form that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the application of the second

Form: $h \equiv f \circ g$

which means $h(x) \equiv f(g(x))$

For $f(x) \equiv x + 2$ and $g(x) \equiv 3 * x$,

$h \equiv f \circ g$ yields $(3 * x) + 2$

Apply-to-all

- A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters

Form: α

For $h(x) \equiv x * x$

$\alpha(h, (2, 3, 4))$ yields $(4, 9, 16)$

Fundamentals of Functional Programming Languages

- The objective of the design of a FPL is to mimic mathematical functions to the greatest extent possible
- The basic process of computation is fundamentally different in a FPL than in an imperative language
 - In an imperative language, operations are done and the results are stored in variables for later use
 - Management of variables is a constant concern and source of complexity for imperative programming
- In an FPL, variables are not necessary, as is the case in mathematics

Fundamentals of Functional Programming Languages (cont'd.)

- *Referential Transparency* - In an FPL, the evaluation of a function always produces the same result given the same parameters
- *Tail Recursion* – Writing recursive functions that can be automatically converted to iteration

LISP Data Types and Structures

- LISP was developed by John McCarthy at MIT in 1959
- *Data object types*: originally only **atoms** and **lists**
- *List form*: parenthesized collections of sublists and/or atoms
e.g., (A B (C D) E)
- Originally, LISP was a typeless language
- LISP lists are stored internally as single-linked lists

LISP Interpretation

- Lambda notation is used to specify functions and function definitions. Function applications and data have the same form.
e.g., If the list (A B C) is interpreted as data it is
a simple list of three atoms, A, B, and C
If it is interpreted as a function application, it means that the function named A is applied to the two parameters, B and C

Origins of Scheme

- A mid-1970s dialect of LISP, designed to be a cleaner, more modern, and simpler version than the contemporary dialects of LISP
- Uses only static scoping
- Functions are first-class entities
 - They can be the values of expressions and elements of lists
 - They can be assigned to variables, passed as parameters, and returned from functions

The Scheme Interpreter

- In interactive mode, the Scheme interpreter is an infinite read-evaluate-print loop (REPL)
 - This form of interpreter is also used by Python and Ruby
- Expressions are interpreted by the function `EVAL`
- Literals evaluate to themselves

Primitive Function Evaluation

- Parameters are evaluated, in no particular order
- The values of the parameters are substituted into the function body
- The function body is evaluated
- The value of the last expression in the body is the value of the function

Primitive Functions

- **Primitive Arithmetic Functions:** `+`, `-`, `*`, `/`, `ABS`, `SQRT`, `REMAINDER`, `MIN`, `MAX`
e.g., `(+ 5 2)` yields 7
- **QUOTE** - takes one parameter; returns the parameter without evaluation
 - `QUOTE` is required because the Scheme interpreter, named `EVAL`, always evaluates parameters to function applications before applying the function. `QUOTE` is used to avoid parameter evaluation when it is not appropriate
 - `QUOTE` can be abbreviated with the apostrophe prefix operator
`' (A B)` is equivalent to `(QUOTE (A B))`

Function Definition: LAMBDA

- Lambda Expressions

- Form is based on λ notation

e.g., (LAMBDA (x) (* x x))

x is called a bound variable

- Lambda expressions can be applied to parameters

e.g., ((LAMBDA (x) (* x x)) 7)

- LAMBDA expressions can have any number of parameters

(LAMBDA (a b x) (+ (* a x x) (* b x)))

Special Form Function: DEFINE

- A Function for constructing functions: `DEFINE` - Two forms:

1. To bind a symbol to an expression

e.g., `(DEFINE pi 3.141593)`

Example use: `(DEFINE two_pi (* 2 pi))`

2. To bind names to lambda expressions

e.g., `(DEFINE (square x) (* x x))`

Example use: `(square 5)`

- The evaluation process for `DEFINE` is different! The first parameter is never evaluated. The second parameter is evaluated and bound to the first parameter.

Output Functions

- (DISPLAY expression)
- (NEWLINE)

Numeric Predicate Functions

- $\#T$ (or $\#t$) is true and $\#F$ (or $\#f$) is false (sometimes $()$ is used for false)
- $=$, $<>$, $>$, $<$, $>=$, $<=$
- $EVEN?$, $ODD?$, $ZERO?$, $NEGATIVE?$
- The NOT function inverts the logic of a Boolean expression

Control Flow: IF

- Selection- the special form, IF

`(IF predicate then_exp else_exp)`

e.g.,

```
(IF (<> count 0)
    (/ sum count)
    0)
```

Control Flow: COND

- Multiple Selection - the special form, COND

General form:

(COND

 (*predicate_1* *expr* {*expr*})

 (*predicate_1* *expr* {*expr*})

 . . .

 (*predicate_1* *expr* {*expr*})

 (ELSE *expr* {*expr*}))

- Returns the value of the last expression in the first pair whose predicate evaluates to true

Example of COND

```
(DEFINE (compare x y)
  (COND
    ((> x y) "x is greater than y")
    (< x y) "y is greater than x")
    (ELSE "x and y are equal")
  )
)
```

List Functions: CONS and LIST

- `CONS` takes two parameters, the first of which can be either an atom or a list and the second of which is a list; returns a new list that includes the first parameter as its first element and the second parameter as the remainder of its result

e.g., `(CONS 'A '(B C))` returns `(A B C)`

- `LIST` takes any number of parameters; returns a list with the parameters as elements

e.g. `(LIST 'apple 'orange 'grape)` returns `(apple orange grape)`

List Functions: CAR and CDR

- CAR takes a list parameter; returns the first element of that list

e.g., (CAR ' (A B C)) yields A

(CAR ' ((A B) C D)) yields (A B)

- CDR takes a list parameter; returns the list after removing its first element

e.g., (CDR ' (A B C)) yields (B C)

(CDR ' ((A B) C D)) yields (C D)

List Functions: CAR and CDR

- `(DEFINE (second a_list) (CAR (CDR a_list)))`

Once this function is evaluated, it can be used, as in

`(second ' (A B C))` = returns B

- Some of the most commonly used functional compositions in Scheme are built in as single functions.

`(CAAR x)` = `(CAR (CAR x))`

`(CADR x)` = `(CAR (CDR x))`

`(CADDAR x)` = `(CAR (CDR (CDR (CAR x))))`.

`(CADDAR ' ((A B (C) D) E))` = `(C)`

Predicate Function: EQ?

- EQ? takes two symbolic parameters; it returns #T if both parameters are atoms and the two are the same; otherwise #F

e.g., (EQ? 'A 'A) yields #T

(EQ? 'A 'B) yields #F

- Note that if EQ? is called with list parameters, the result is not reliable
- Also EQ? does not work for numeric atoms

Predicate Function: EQV?

- EQV? is like EQ?, except that it works for both symbolic and numeric atoms; it is a value comparison, not a pointer comparison

(EQV? 3 3) yields #T

(EQV? 'A 3) yields #F

(EQV? 3.4 (+ 3 0.4)) yields #T

(EQV? 3.0 3) yields #F (floats and integers are different)

Predicate Functions: LIST? and NULL?

- LIST? takes one parameter; it returns #T if the parameter is a list; otherwise #F

(LIST? ' ()) yields #T

- NULL? takes one parameter; it returns #T if the parameter is the empty list; otherwise #F
 - Note that NULL? returns #T if the parameter is ()
 - e.g. (NULL? ' (())) yields #F

Example Scheme Function: `member`

- `member` takes an atom and a simple list; returns `#T` if the atom is in the list; `#F` otherwise

```
(DEFINE (member atm lis)
(COND
  ((NULL? lis) #F)
  ((EQ? atm (CAR lis)) #T)
  ((ELSE (member atm (CDR lis)))
  ) )
```

Example Scheme Function: `equalsimp`

- `equalsimp` takes two simple lists as parameters; returns `#T` if the two simple lists are equal; `#F` otherwise

```
(DEFINE (equalsimp lis1 lis2)
  (COND
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) #F)
    ((EQ? (CAR lis1) (CAR lis2))
     (equalsimp (CDR lis1) (CDR lis2)))
    (ELSE #F)
  ))
```

Example Scheme Function: equal

- `equal` takes two general lists as parameters; returns `#T` if the two lists are equal; `#F` otherwise

```
(DEFINE (equal lis1 lis2)
  (COND
    ((NOT (LIST? lis1)) (EQ? lis1 lis2))
    ((NOT (LIST? lis2)) #F)
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) #F)
    ((equal (CAR lis1) (CAR lis2))
     (equal (CDR lis1) (CDR lis2)))
    (ELSE #F)
  ))
```

Example Scheme Function: **append**

- `append` takes two lists as parameters; returns the first parameter list with the elements of the second parameter list appended at the end

```
(DEFINE (append lis1 lis2)
  (COND
    ((NULL? lis1) lis2)
    (ELSE (CONS (CAR lis1)
                  (append (CDR lis1) lis2))))
))
```

`(append '(A B) '(C D R))` returns `(A B C D R)`

`(append '((A B) C) '(D (E F)))` returns `((A B) C D (E F))`

Example Scheme Function: LET

- General form:

```
(LET (
    (name_1 expression_1)
    (name_2 expression_2)
    ...
    (name_n expression_n) )
  body
)
```

- Evaluate all expressions, then bind the values to the names; evaluate the body

LET Example

```
(DEFINE (quadratic_roots a b c)
  (LET (
    (root_part_over_2a
      (/ (SQRT (- (* b b) (* 4 a c))) (* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a))))
  (DISPLAY (+ minus_b_over_2a root_part_over_2a))
  (NEWLINE)
  (DISPLAY (- minus_b_over_2a root_part_over_2a))
  ))
```


Tail Recursion in Scheme

- Definition: A function is *tail recursive* if its recursive call is the last operation in the function
- A tail recursive function can be automatically converted by a compiler to use iteration, making it faster
- Scheme language definition requires that its language systems convert all tail recursive functions to use iteration

Tail Recursion in Scheme (cont'd.)

- Example of rewriting a function to make it tail recursive, using helper a function

Original:

```
(DEFINE (factorial n)
  (IF (= n 0)
      1
      (* n (factorial (- n 1)))
  ))
```

Tail recursive:

```
(DEFINE (facthelper n factpartial)
  (IF (= n 0)
      factpartial
      facthelper((- n 1) (* n factpartial)))
  ))

(DEFINE (factorial n)
  (facthelper n 1))
```

Functional Form - Composition

- If h is the composition of f and g , $h(x) = f(g(x))$

```
(DEFINE (g x) (* 3 x))
```

```
(DEFINE (f x) (+ 2 x))
```

```
(DEFINE h x) (+ 2 (* 3 x)) ) (The composition)
```

- In Scheme, the functional composition function `compose` can be written:

```
(DEFINE (compose f g) (LAMBDA (x) (f (g x))))
```

```
((compose CAR CDR) '((a b) c d)) yields c
```

```
(DEFINE (third a_list)
```

```
  ((compose CAR (compose CDR CDR)) a_list))
```

is equivalent to `CADDR`

Functional Form – Apply-to-All

- Apply to All - one form in Scheme is `map`
 - Applies the given function to all elements of the given list;

```
(DEFINE (map fun lis)
  (COND
    ((NULL? lis) ())
    (ELSE (CONS (fun (CAR lis))
                  (map fun (CDR lis))))
  ))
```

```
(map (LAMBDA (num) (* num num num)) '(3 4 2 6))
yields (27 64 8 216)
```

Functions That Build Code

- It is possible in Scheme to define a function that builds Scheme code and requests its interpretation
- This is possible because the interpreter is a user-available function, `EVAL`

Adding a List of Numbers

```
((DEFINE (adder lis)
  (COND
    ((NULL? lis) 0)
    (ELSE (EVAL (CONS '+ lis))))
))
```

- The parameter is a list of numbers to be added; `adder` inserts a `+` operator and evaluates the resulting list
 - Use `CONS` to insert the atom `+` into the list of numbers.
 - Be sure that `+` is quoted to prevent evaluation
 - Submit the new list to `EVAL` for evaluation

Applications of Functional Languages

- LISP is used for artificial intelligence
 - Knowledge representation
 - Machine learning
 - Natural language processing
 - Modeling of speech and vision
- Scheme is used to teach introductory programming at some universities
- Support for functional programming is increasingly creeping into imperative languages

Comparing Functional and Imperative Languages

- Imperative Languages:
 - Efficient execution
 - Complex semantics
 - Complex syntax
 - Concurrency is programmer designed
- Functional Languages:
 - Simple semantics
 - Simple syntax
 - Inefficient execution
 - Programs can automatically be made concurrent

Subprograms

BBM 301 – Programming Languages

Fundamentals of Subprograms

- Each subprogram has a single entry point
- The calling program is suspended during execution of the called subprogram
 - Therefore, only one subprogram is in execution at a given time
- Control always returns to the caller when the called subprogram's execution terminates

Basic Definitions

- A *subprogram definition* describes the interface to and the actions of the subprogram abstraction
 - In Python, function definitions are executable; in all other languages, they are non-executable

```
if ...  
    def fun1 (...);  
else  
    def fun2 (...);  
    ...
```

Basic Definitions

- A *subprogram header* is the first part of the definition, including the name, the kind of subprogram, and the formal parameters

FORTRAN example:

```
SUBROUTINE name (parameters)
```

C example:

```
void adder(parameters)
```

- A *subprogram call* is an explicit request that the subprogram be executed

Basic Definitions (cont'd.)

- The *parameter profile (aka signature)* of a subprogram is the number, order, and types of its parameters
- The *protocol* is a subprogram's parameter profile and, if it is a function, its return type

Basic Definitions (cont'd.)

- Function declarations in C and C++ are often called *prototypes*
- A *subprogram declaration* provides the protocol, but not the body, of the subprogram
- A *formal parameter* is a dummy variable listed in the subprogram header and used in the subprogram
- An *actual parameter* represents a value or address used in the subprogram call statement

Actual/Formal Parameter Correspondence

- Positional
 - The binding of actual parameters to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth
 - Safe and effective
- Keyword
 - The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter
 - *Advantage*: Parameters can appear in any order, thereby avoiding parameter correspondence errors
 - *Disadvantage*: User must know the formal parameter's names

Parameters

Example in Ada,

```
SUMER (LENGTH => 10,  
      LIST => ARR,  
      SUM => ARR_SUM) ;
```

Formal parameters: LENGTH, LIST, SUM.

Actual parameters: 10, ARR, ARR_SUM.

The programmer doesn't have to know the order of the formal parameters.

But, must know the names of the formal parameters.

Formal Parameter Default Values

- In certain languages (e.g., C++, Python, Ruby, Ada, PHP), formal parameters can have default values (if no actual parameter is passed)
 - In C++, default parameters must appear last because parameters are positionally associated

Ada example:

```
function Comp_Pay (Income: Float;  
                  Exampctions: Integer := 1;  
                  Tax_Rate: Float) return Float;
```

Therefore, the call doesn't have to provide values for all parameters.

A sample call may be

```
Pay := Comp_Pay (2000.0, Tax_Rate => 0.23);
```

Procedures and Functions

- There are two categories of subprograms
 - *Procedures* are collection of statements that define parameterized computations
 - *Functions* structurally resemble procedures but are semantically modeled on mathematical functions
 - They are expected to produce no side effects
 - In practice, program functions have side effects

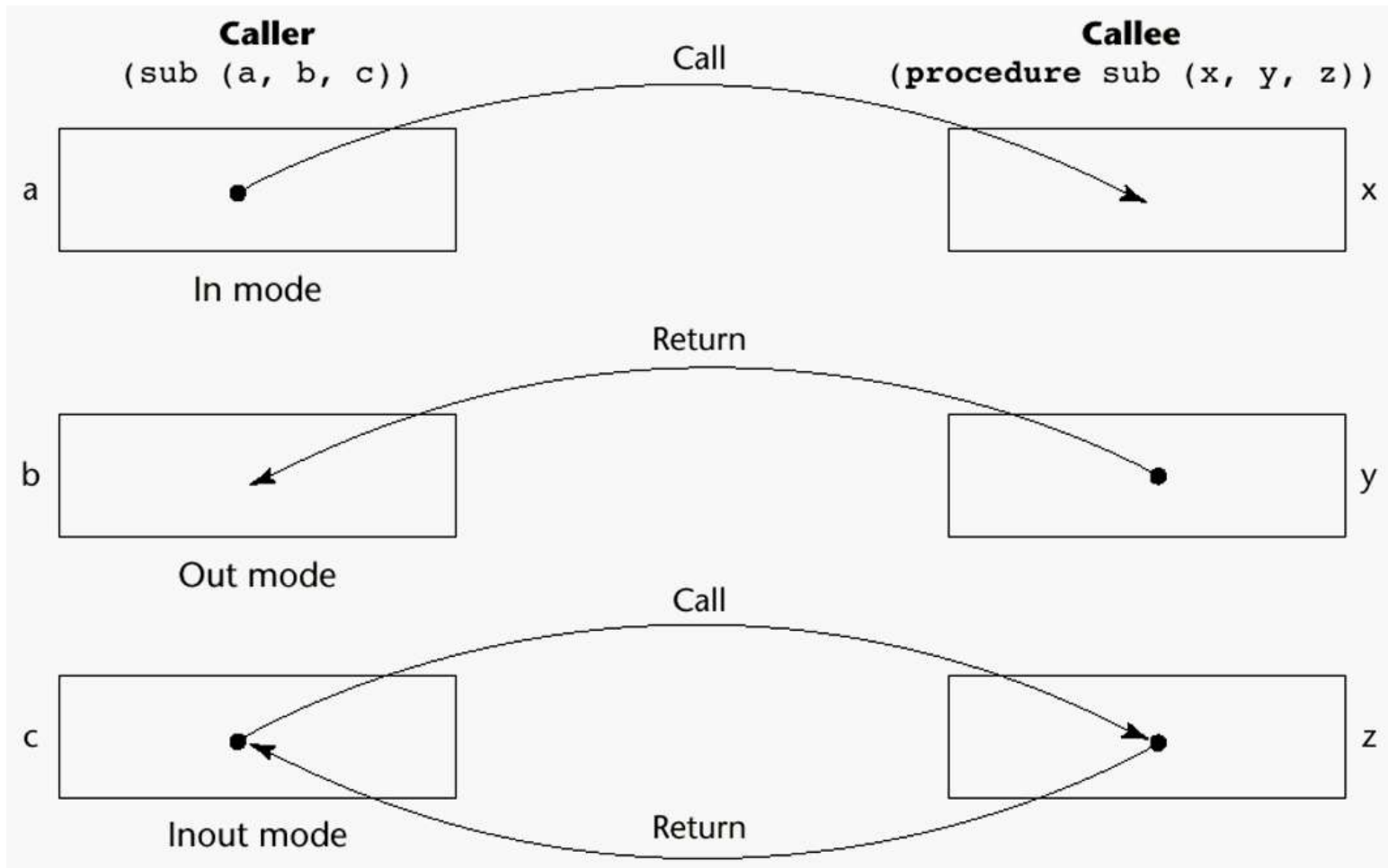
Design Issues for Subprograms

- Are local variables static or dynamic?
- Can subprogram definitions appear in other subprogram definitions?
- What parameter passing methods are provided?
- Are parameter types checked?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
- Can subprograms be overloaded?
- Can subprogram be generic?

Semantic Models of Parameter Passing

- In mode
- Out mode
- Inout mode

Models of Parameter Passing



Parameter Passing Methods

- Ways in which parameters are transmitted to and/or from called subprograms
 - Pass-by-value
 - Pass-by-result
 - Pass-by-value-result
 - Pass-by-reference
 - Pass-by-name

Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialize the corresponding formal parameter
 - Normally implemented by copying
 - Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)
 - *Advantages: Actual variable is protected.*
 - *Disadvantages: additional storage is required (stored twice) and the actual move can be costly (for large parameters)*

Pass-by-Result (Out Mode)

- No value is transmitted to the subprogram
- The corresponding formal parameter acts as a local variable
- its value is transmitted to caller's actual parameter when control is returned to the caller
 - Require extra storage location and copy operation
- Potential problems:
 - `sub (p1, p1) ;` whichever formal parameter is copied back will represent the current value of p1
 - `sub (list[sub], sub) ;` Compute address of list[sub] at the beginning of the subprogram or end?

Pass-by-Result (Out Mode)

Problem: Actual parameter collision definition:

```
subprogram sub(x, y) { x <- 3 ; y <- 5 ; }  
call:  
sub(p, p)
```

what is the value of p here ? (3 or 5?)

- The values of x and y will be copied back to p. Which ever is assigned last will determine the value of p.
- The order is important
- The order is implementation dependent ⇒ Portability problems.

Pass-by-Result (Out Mode)

Problem: Time to evaluate the address of the actual parameter

- at the time of the call
- at the time of the return
- The decision is up to the implementation.

Definition:

```
subprogram sub(x)
```

```
i <- 5                is changed as a global variable here
```

```
x <- ..
```

```
call:
```

```
i <- 3
```

```
sub(A[i])
```

Is A[3] or A[5] is changed?

The decision is up to the implementation.

Pass-by-Value-Result (inout Mode)

- A combination of pass-by-value and pass-by-result
- Sometimes called pass-by-copy
- Formal parameters have local storage
- Disadvantages:
 - Those of pass-by-result
 - Those of pass-by-value
- The value of the actual parameter is used to initialize the corresponding formal parameter
 - the formal parameter acts as a local parameter
 - At termination, the value of the formal parameter is copied back.

Pass-by-Reference (Inout Mode)

- Pass an access path
 - Also called pass-by-sharing
 - Advantage:
 - Passing process is efficient
 - no copying
 - no duplicated storage
 - Disadvantages
 - Slower accesses (compared to pass-by-value) to formal parameters
 - Potentials for unwanted side effects (collisions)
 - Unwanted aliases (access broadened)
- ```
fun(total, total); fun(list[i], list[j]); fun(list[i], i);
```

# Pass-by-Reference

Dangerous: Actual parameter may be modified unintentionally.

Aliasing:

definition:

```
subprogram sub(x, y)
```

call:

```
sub(p, p)
```

Here, x and y in the subprogram are aliases.

- Another way of aliasing:

definition:

```
var x global variable
```

```
subprogram sub(y)
```

call:

```
sub(x)
```

Here, x and y in the subprogram are aliases.

# Pass by : Example

## Example of call by value versus value-result versus reference

The following examples are in an Algol-like language.

```
begin
integer n;
procedure p(k: integer);
 begin
 n := n+1;
 k := k+4;
 print(n);
 end;
n := 0;
p(n);
print(n);
end;
```

### OUTPUT:

call by value: 1 1

call by value-result: 1 4

call by reference: 5 5

# Pass by : Example

# Pass-by-Name (Inout Mode)

- By textual substitution: Actual parameter is textually substituted for the corresponding formal parameter in all occurrences in the subprogram.
- **Late binding:** actual binding to a value or an address is delayed until the formal parameter is assigned or referenced.
- Allows flexibility in late binding



# Pass-by-name

- If the actual parameter is a scalar variable, then it is equivalent to pass-by-reference.
- If the actual parameter is a constant expression, then it is equivalent to pass-by-value.
- **Advantage:** flexibility
- **Disadvantage:** slow execution, difficult to implement, confusing.

# Pass-by-name

```
procedure BIG;
integer GLOBAL;
integer array LIST[1:2];
procedure SUB (P: integer);
begin
 P := 3;
 GLOBAL := GLOBAL + 1;
 P := 5;
end;
begin
 LIST[1] := 2;
 LIST[2] := 2;
 GLOBAL := 1;
 SUB(LIST[GLOBAL]);
end.
```

```
LIST[GLOBAL] := 3
GLOBAL := GLOBAL + 1
LIST[GLOBAL] := 5
```

## ***Execution:***

```
LIST[1] := 3
GLOBAL := 1 + 1
LIST[2] := 5
```

# Pass-by-Name Elegance: Jensen's Device

- Passing expressions into a routine so they can be repeatedly evaluated has some valuable applications.
- Consider calculations of the form: "sum  $x_i \times i$  for all  $i$  from 1 to  $n$ ." How could a routine Sum be written so that we could express this as

`sum(i, 1, n, x[i]*i) ?`

- Using pass-by-reference or pass-by-value, we cannot do this because we would be passing in only a single value of  $x[i]*i$ , not an expression which can be repeatedly evaluated as " $i$ " changes.
- Using pass-by-name, the expression  $x[i]*i$  is passed in without evaluation.

# Pass-by-Name Elegance: Jensen's Device

```
real procedure Sum(j, lo, hi, Ej);
value lo, hi;
integer j, lo, hi;
real Ej;
begin
 real S; S := 0;
 for j := lo step 1 until hi do
 S := S + Ej;
 Sum := S
end;
```

- Each time through the loop, evaluation of  $E_j$  is actually the evaluation of the expression  $x[i]*i = x[j]*j$ .

# Pass-By-Name Security Problem (Severe)

- A sample program:
  - procedure swap (a, b);
  - integer a, b, temp;
  - begin temp := a; a := b; b:= temp end;
- Effect of the call swap(x, y):
  - temp := x; x := y; y := temp
- Effect of the call swap(i, x[i]):
  - temp := i; i := x[i]; x[i] := temp
  - It doesn't work! For example:

|              |       |          |          |
|--------------|-------|----------|----------|
| Before call: | i = 2 | x[2] = 5 |          |
| After call:  | i = 5 | x[2] = 5 | x[5] = 2 |

- It is very difficult to write a correct swap procedure in Algol.

# Pass by : Example

## Example of call by value versus call by name

```
begin
integer n;
procedure p(k: integer);
 begin
 print(k);
 n := n+10;
 print(k);
 n := n+5;
 print(k);
 end;
n := 0;
p(n+1);
end;
```

### OUTPUT

call by value: 1 1 1

call by name: 1 11 16

# Pass by : Example

## Example of call by reference versus call by name

This example illustrates assigning into a parameter that is passed by reference or by name

```
begin
array a[1..10] of integer;
integer n;
procedure p(b: integer);
 begin
 print(b);
 n := n+1;
 print(b);
 b := b+5;
 end;
a[1] := 10;
a[2] := 20;
a[3] := 30;
a[4] := 40;
n := 1;
p(a[n+2]);
new_line;
print(a);
end;
```

### OUTPUT

call by reference: 30 30 10 20 35 40

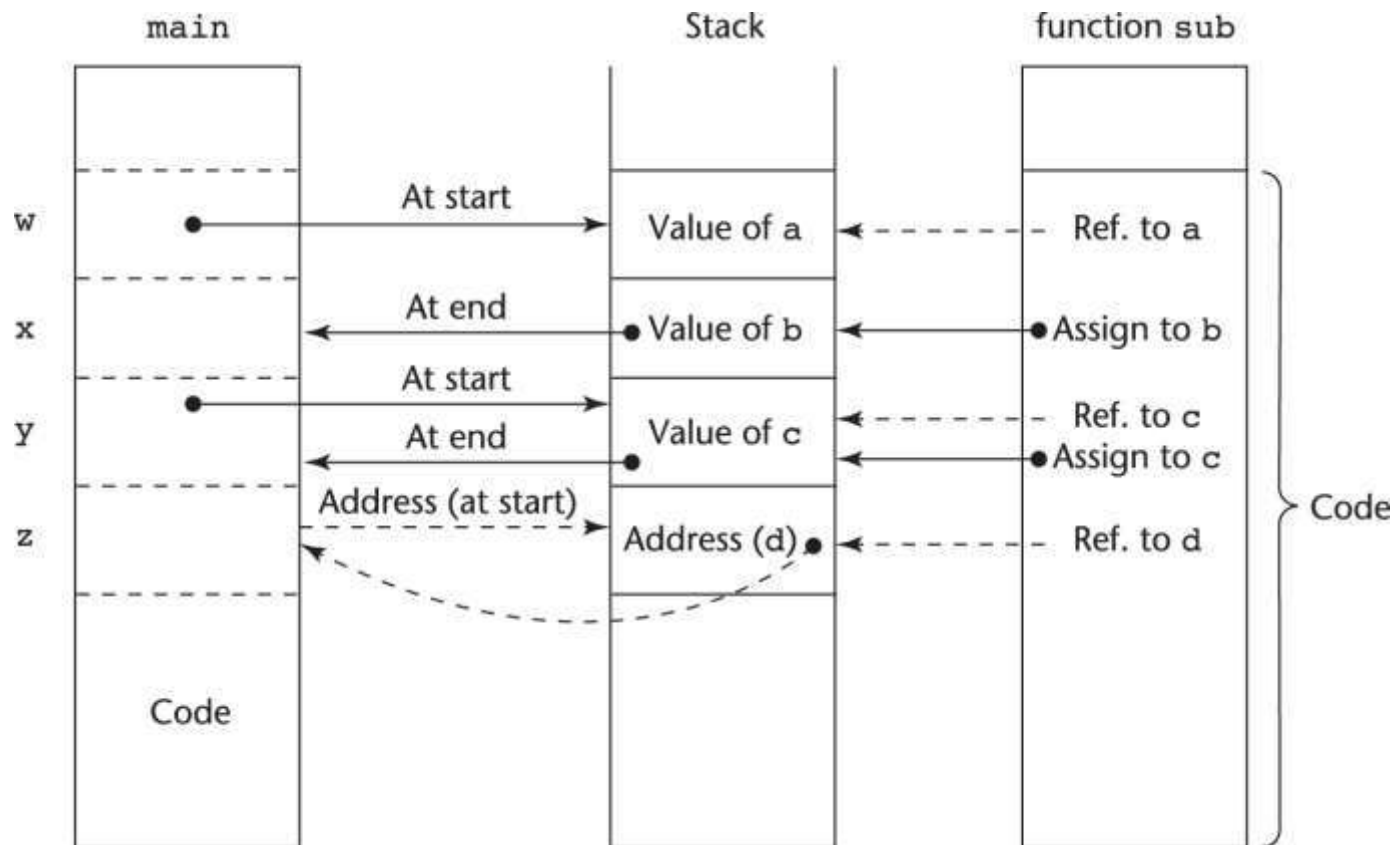
call by name: 30 40 10 20 30 45

# Implementing Parameter-Passing Methods

- In most language parameter communication takes place thru the run-time stack
- Pass-by-reference is the simplest to implement; only an address is placed in the stack
- A subtle but fatal error can occur with pass-by-reference and pass-by-value-result:
  - A formal parameter corresponding to a constant can mistakenly be changed



# Implementing Parameter-Passing Methods



Function header: `void sub(int a, int b, int c, int d)`

Function call in main: `sub(w, x, y, z)`

(pass **w** by value, **x** by result, **y** by value-result, **z** by reference)

# Parameter Passing Methods of Major Languages

- C
  - Pass-by-value
  - Pass-by-reference is achieved by using pointers as parameters
- C++
  - A special pointer type called reference type for pass-by-reference
- Java
  - All parameters are passed are passed by value
  - Object parameters are passed by reference

- Ada

```
procedure Adder (A: in out Integer;
 B: in Integer;
 C: out Float);
```

- Three semantics modes of parameter transmission: `in`, `out`, `in out`; `in` is the default mode
- Formal parameters declared `out` can be assigned but not referenced; those declared `in` can be referenced but not assigned; `in out` parameters can be referenced and assigned

# Design Considerations for Parameter Passing

- Two important considerations
  - Efficiency
  - One-way or two-way data transfer
- But the above considerations are in conflict
  - Good programming suggest limited access to variables, which means one-way whenever possible
  - But pass-by-reference is more efficient to pass structures of significant size

# Parameters that are Subprogram Names

- It is sometimes convenient to pass subprogram names as parameters
- Issues:
  1. Are parameter types checked?
  2. What is the correct referencing environment for a subprogram that was sent as a parameter?

# Parameters that are Subprogram Names: Referencing Environment

- Q: *What is the referencing environment for executing the passed subprogram? (For non local variables)*
- *Shallow binding*: The environment of the call statement that enacts the passed subprogram
  - Most natural for dynamic-scoped languages
- *Deep binding*: The environment of the definition of the passed subprogram
  - Most natural for static-scoped languages
- *Ad hoc binding*: The environment of the call statement that passed the subprogram

# Parameters that are Subprograms

Example:

```
function sub1() {
 var x; 2: declared in
 function sub2() {
 window.status = x;
 } // sub2
 function sub3() {
 var x;
 x = 3;
 sub4(sub2); 3: passed in
 } // sub3
 function sub4(subx) {
 var x;
 x = 1;
 subx(); 1: called by
 } // sub4
 x = 2;
 sub3();
} // sub1
```

Passed subprogram S2

Output:

is called by S4

is declared in S1

is passed in S3

*1- Shallow binding*

*2- Deep binding*

*3- Ad hoc binding*

# Overloaded Subprograms

- An *overloaded subprogram* is one that has the same name as another subprogram in the same referencing environment
  - Every version of an overloaded subprogram has a unique protocol (different number of arguments, etc)
  - The correct meaning (the correct code) to be invoked is determined by the *actual parameter* list.
  - In case of *functions*, the *return type* may be used to distinguish.
- Ada, Java, C++, and C# allow users to write multiple versions of subprograms with the same name

# Generic Subprograms

- A *generic or polymorphic subprogram* takes parameters of different types on different activations
- *The same formal parameter can get values of different types.*
- *Ada and C++ provide Generic (Polymorphic) Subprograms*



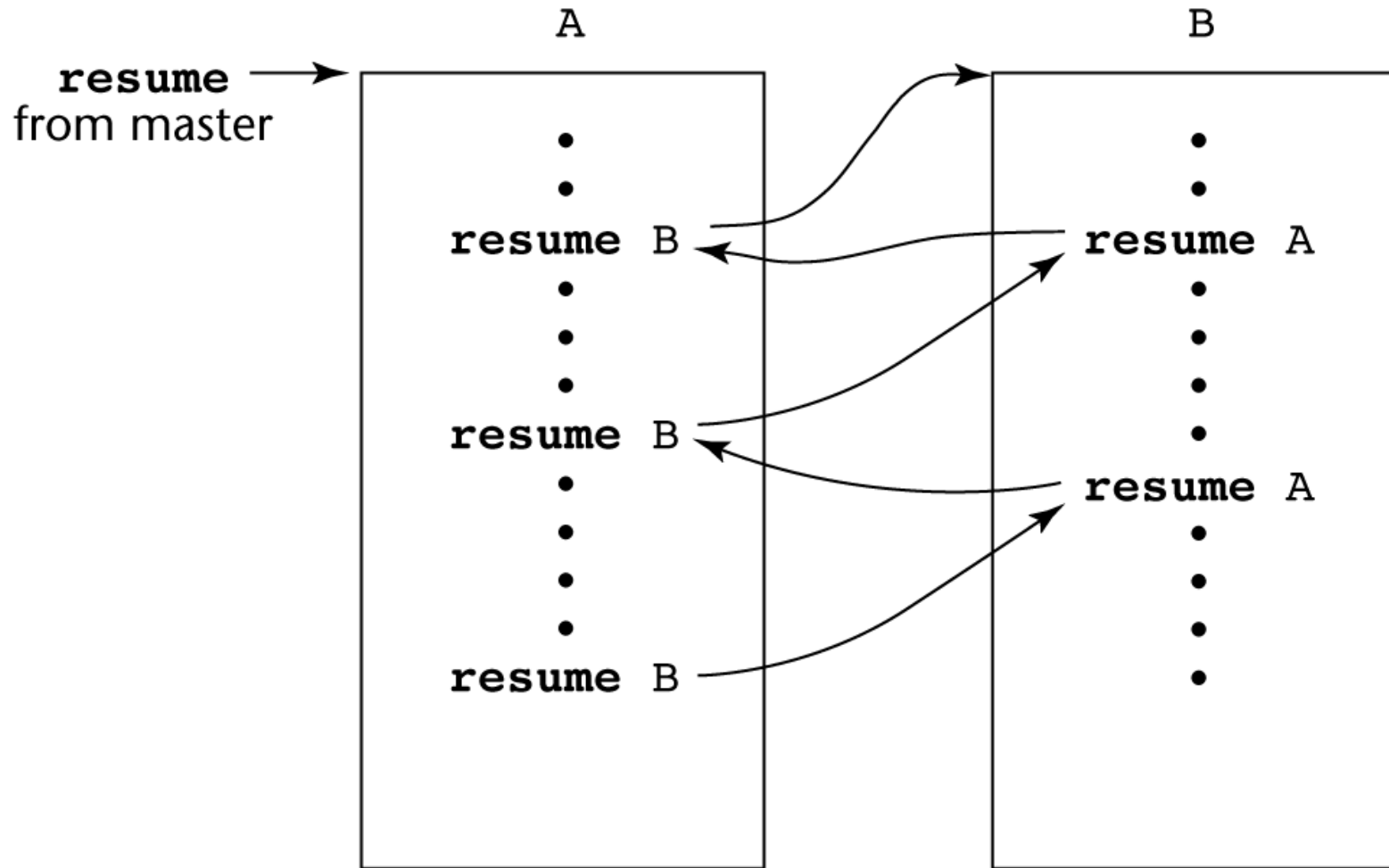
# Design Issues for Functions

- Are side effects allowed?
  - Parameters should always be in-mode to reduce side effect (like Ada)
- What types of return values are allowed?
  - Most imperative languages restrict the return types
  - C allows any type except arrays and functions
  - C++ is like C but also allows user-defined types
  - Ada subprograms can return any type (but Ada subprograms are not types, so they cannot be returned)
  - Java and C# methods can return any type (but because methods are not types, they cannot be returned)
  - Python and Ruby treat methods as first-class objects, so they can be returned, as well as any other class

# Coroutines

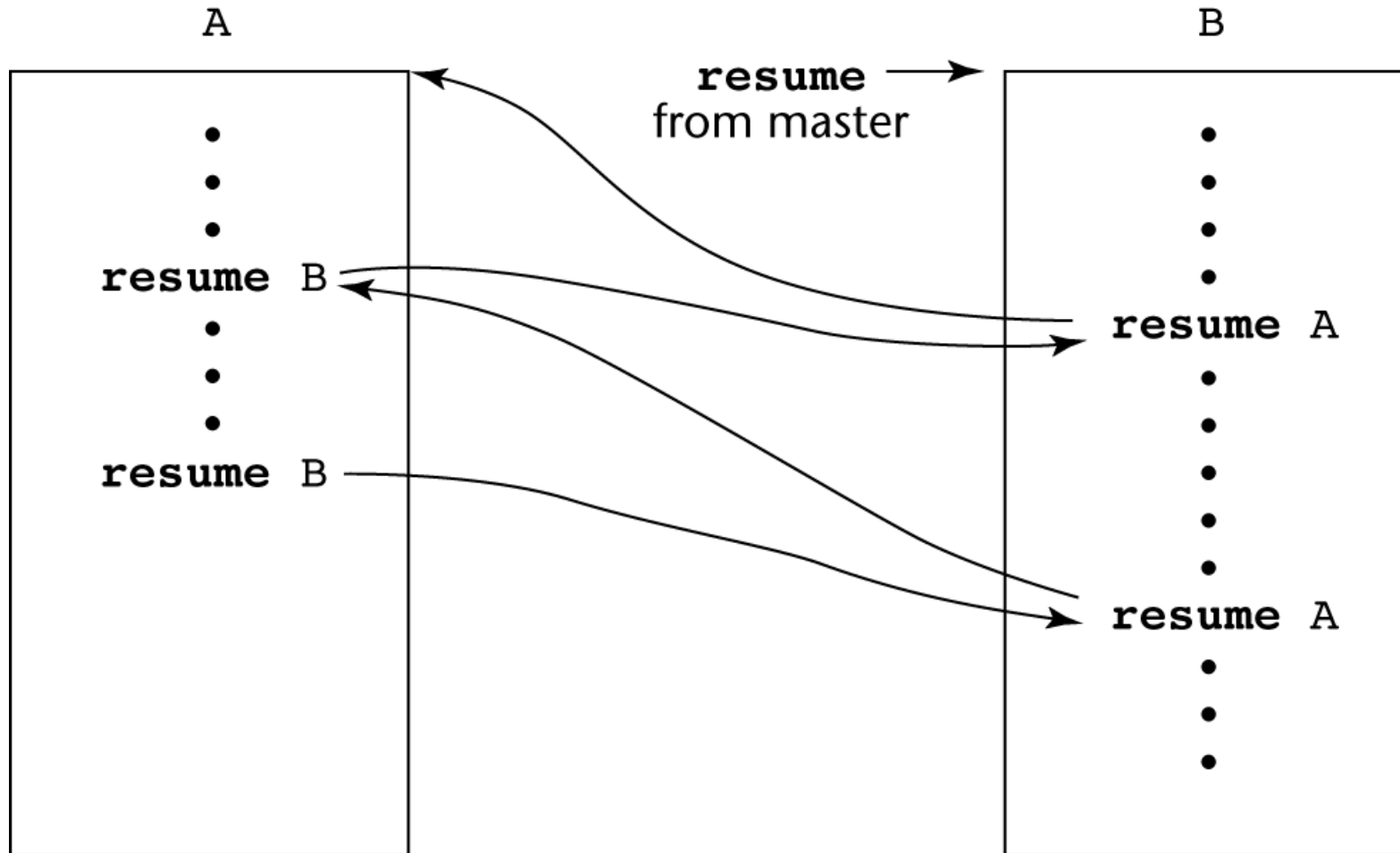
- A *coroutine* is a special kind of a subprogram that has multiple entries and controls them itself
- Also called *symmetric control*: caller and called coroutines are on a more equal basis
- *Coroutines call is a resume*. The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine
- Coroutines provide *quasi-concurrent execution* of program units (the coroutines); their execution is interleaved, but not overlapped
- Coroutines are history sensitive, thus they have static variables.
- Coroutines are created in an application by a special unit called master unit, which is not a coroutine.
- A coroutine may have an initialization code that is executed only when it is created.
- Only one coroutine executes at a given time.

# Coroutines Illustrated: Possible Execution Controls



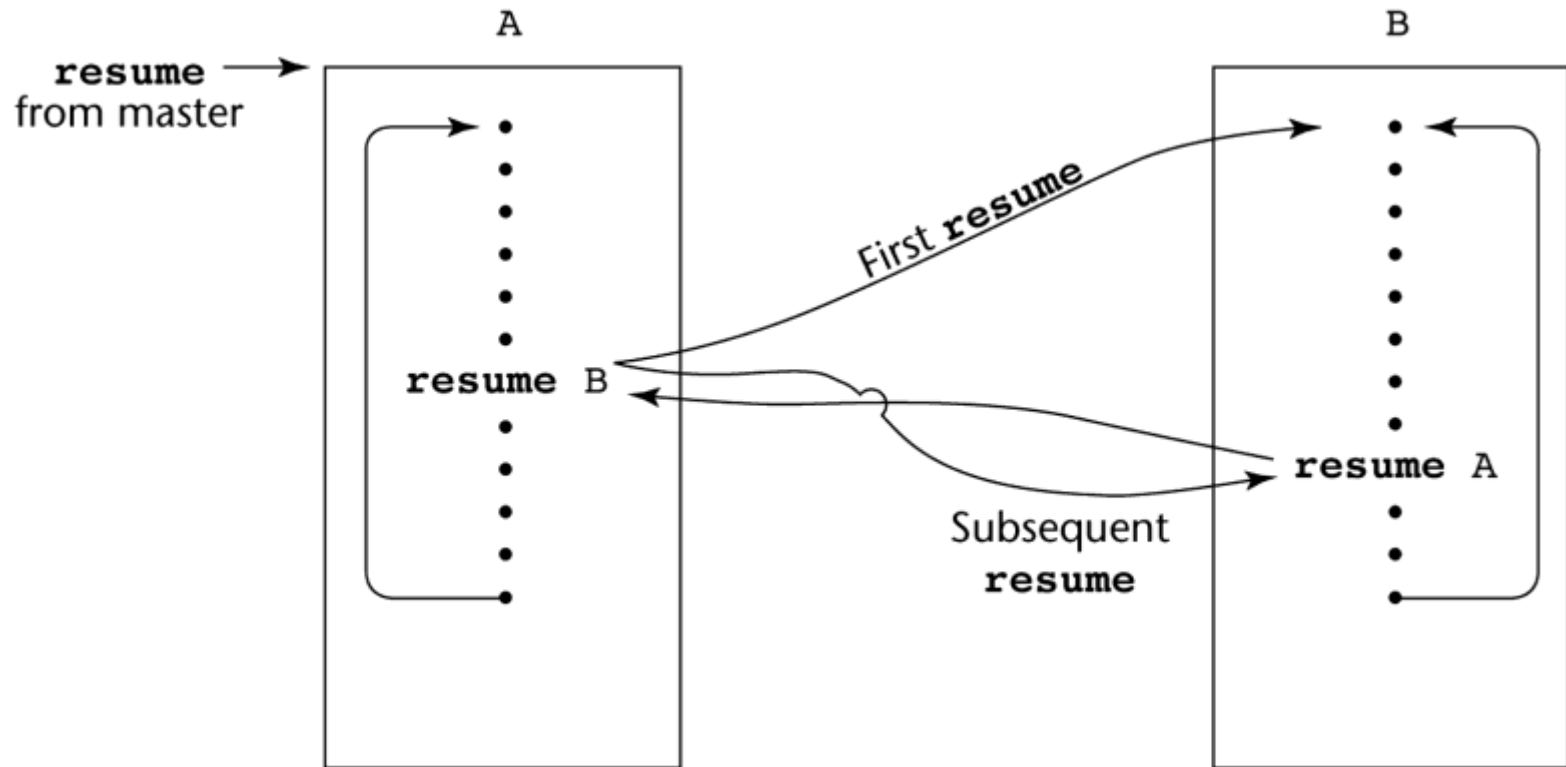
(a)

# Coroutines Illustrated: Possible Execution Controls



(b)

# Coroutines Illustrated: Possible Execution Controls with Loops



# Implementing Subprograms

BBM 301 – Programming Languages

# The General Semantics of Calls and Returns

- The subprogram call and return operations of a language are together called its *subprogram linkage*
- General semantics of subprogram calls
  - Parameter passing methods
  - Stack-dynamic allocation of local variables
  - Save the execution status of calling program
  - Transfer of control and arrange for the return
  - If subprogram nesting is supported, access to nonlocal variables must be arranged

# The General Semantics of Calls and Returns

- General semantics of subprogram returns:
  - Out mode and inout mode parameters must have their values returned
  - Deallocation of stack-dynamic locals
  - Restore the execution status
  - Return control to the caller



# Implementing “Simple” Subprograms

- Simple Programs:
  - Subprograms cannot be nested
  - All local variables are static
  - Ex: Early versions of Fortran
- Call Semantics:
  - Save the execution status of the caller (caller/callee)
  - Pass the parameters (caller)
  - Pass the return address to the subprogram (caller)
  - Transfer control to the subprogram (caller)

# Implementing “Simple” Subprograms

- Return Semantics:
  - If pass-by-value-result or out mode parameters are used, move the current values of those parameters to their corresponding actual parameters (callee)
  - If it is a function, move the functional value to a place the caller can get it (callee)
  - Restore the execution status of the caller (caller/callee)
  - Transfer control back to the caller (callee)

# Storage required for call/return actions

- Required storage:
  - Status information about the caller
  - Parameters
  - return address
  - return value for functions
- These along with local variables and the subprogram code, form the complete collection of information a subprogram needs to execute and then return to the caller

# Implementing “Simple” Subprograms: Parts

- Two separate parts: the actual code (constant) and the non-code part (local variables and data that can change)
- The format, or layout, of the non-code part of an executing subprogram is called an *activation record*
- The form of an activation record is static.
- An *activation record instance (ARI)* is a concrete example of an activation record (the collection of data for a particular subprogram activation)

# An Activation Record for “Simple” Subprograms

Because languages with simple subprograms do not support recursion, there can be only one active version of a given subprogram at a time

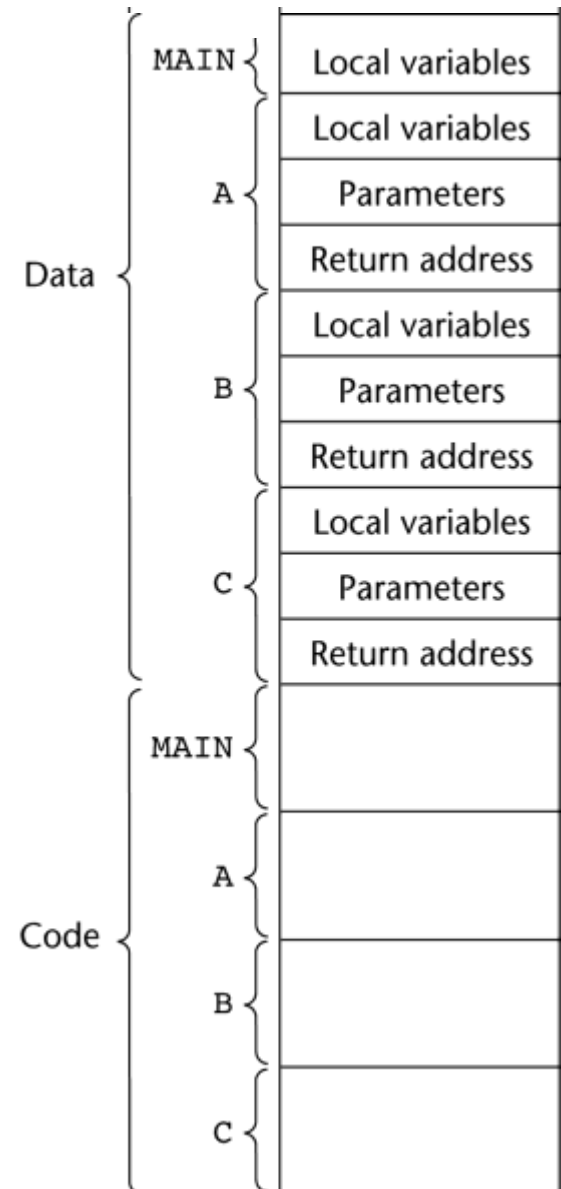
|                 |
|-----------------|
| Local variables |
| Parameters      |
| Return address  |

Therefore, there can be only a single instance of the activation record for a subprogram

Since activation record instance of a simple subprogram has fixed size it can be statically allocated. It could be also attached to the code part

# Code and Activation Records of a Program with “Simple” Subprograms

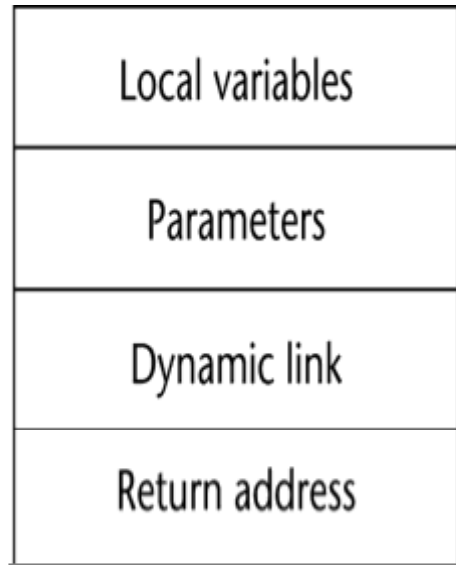
- Note that code could be attached to ARIs
- Also, the four program units could be compiled at different times
- Linker put the compiled parts together when it is called for the main program



# Implementing Subprograms with Stack-Dynamic Local Variables

- Advantage of Stack-Dynamic Local Variables
  - Support for recursion
- More complex activation record
  - The compiler must generate code to cause implicit allocation and deallocation of local variables
  - Recursion must be supported (adds the possibility of multiple simultaneous activations of a subprogram)
  - Each activation needs its own activation record and the number of activations is limited only by the memory size of the machine.

# Typical Activation Record for a Language with Stack-Dynamic Local Variables



↑  
Stack top

**Return address:** pointer to the code segment of the caller and an offset address in that code segment of the instruction following the call

**Dynamic link:** pointer to the top of the activation record instance of the caller

In static scoped languages this link is used to provide traceback information when a run-time error occurs.

In dynamic-scoped languages, the dynamic link is used to access non-local variables



# Implementing Subprograms with Stack-Dynamic Local Variables

- In most languages,
  - The format of the activation record is known at compile time
  - In many cases, the size is also known, because all the local data are of fixed size.
  - In some languages, like Ada, the size of a local array can depend on the value of an actual parameter, so in those cases the format is static, but the size can be dynamic.

# Implementing Subprograms with Stack-Dynamic Local Variables: Activation Record

- An activation record instance is dynamically created when a subprogram is called
- Because the call and return semantics specify that the subprogram last called is the first to complete, it is reasonable to use a stack, so
  - Activation record instances reside on the run-time stack
- Every subprogram activation (recursive or non-recursive) creates a new instance of an activation record on the stack.
- The *Environment Pointer (EP)* must be maintained by the run-time system. It always points at the base of the activation record instance of the currently executing program unit.

# An Example: C Function

```
void sub(float total, int part)
{
 int list[5];
 float sum;
 ...
}
```

|                |          |
|----------------|----------|
| Local          | sum      |
| Local          | list [4] |
| Local          | list [3] |
| Local          | list [2] |
| Local          | list [1] |
| Local          | list [0] |
| Parameter      | part     |
| Parameter      | total    |
| Dynamic link   |          |
| Return address |          |

# Revised Semantic Call/Return Actions

- Create an activation record instance
- Save the execution status of the current program unit
- Compute and pass the parameters
- Pass the return address of the callee
- Transfer the control to the callee

# Revised Semantic Call/Return Actions

## Before (Prologue)

- Save the old EP to the stack as the dynamic link and create the new value
- Allocate local variables

## After (Epilogue)

- If pass-by-value-result or out mode parameters are used, move the current values of those parameters to their corresponding actual parameters
- If it is a function, move the functional value to a place the caller can get it
- Restore the stack pointer by setting it to the value of the current EP minus one and set the EP to the old dynamic link
- Restore the execution status of the caller
- Transfer control back to the caller

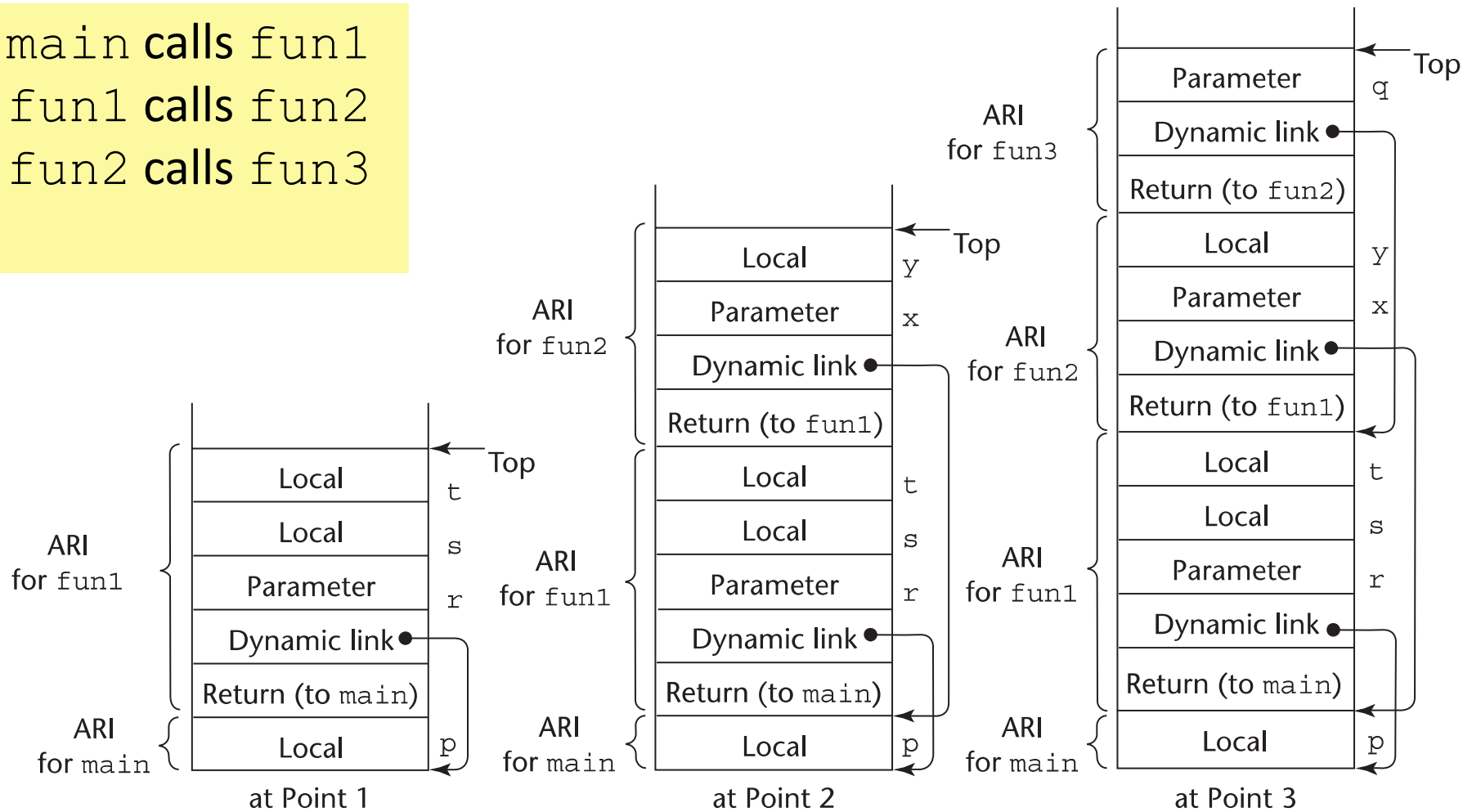
# An Example Without Recursion

```
void fun1(float r) {
 int s, t;
 ...
 fun2(s);
 ...
}
void fun2(int x) {
 int y;
 ...
 fun3(y);
 ...
}
void fun3(int q) {
 ...
}
void main() {
 float p;
 ...
 fun1(p);
 ...
}
```

main calls fun1  
fun1 calls fun2  
fun2 calls fun3

# An Example Without Recursion

main calls fun1  
fun1 calls fun2  
fun2 calls fun3



ARI = activation record instance

# Dynamic Chain and Local Offset

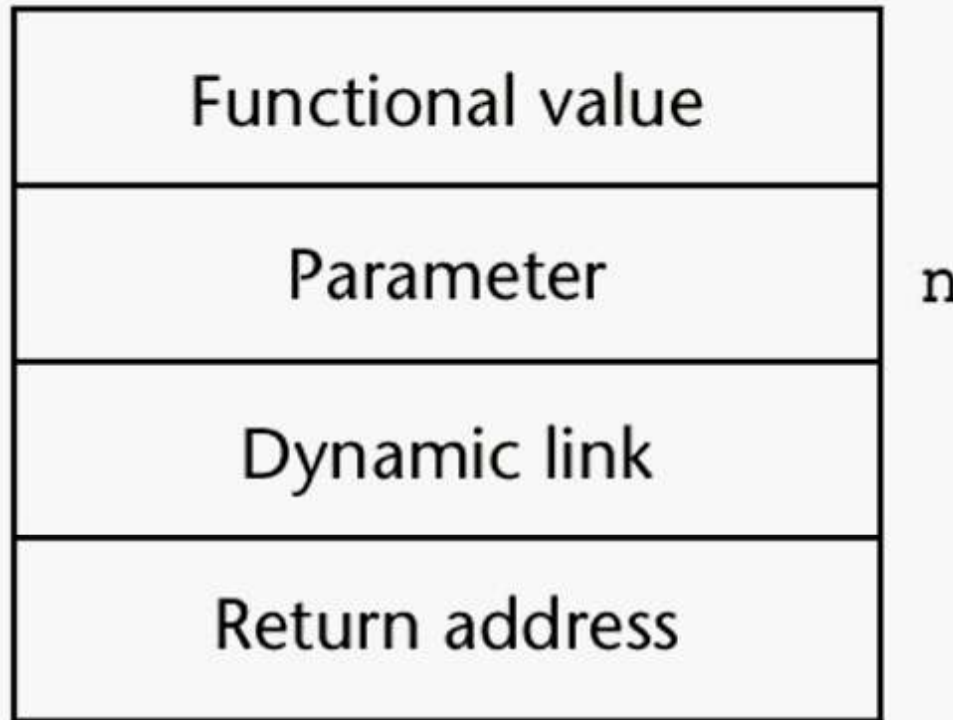
- *Dynamic chain (call chain)*: The collection of dynamic links in the stack at a given time
- Local variables can be accessed by their offset from the beginning of the activation record, whose address is in the EP. This offset is called the *local\_offset*
- The local\_offset of a local variable can be determined by the compiler at compile time
  - How?



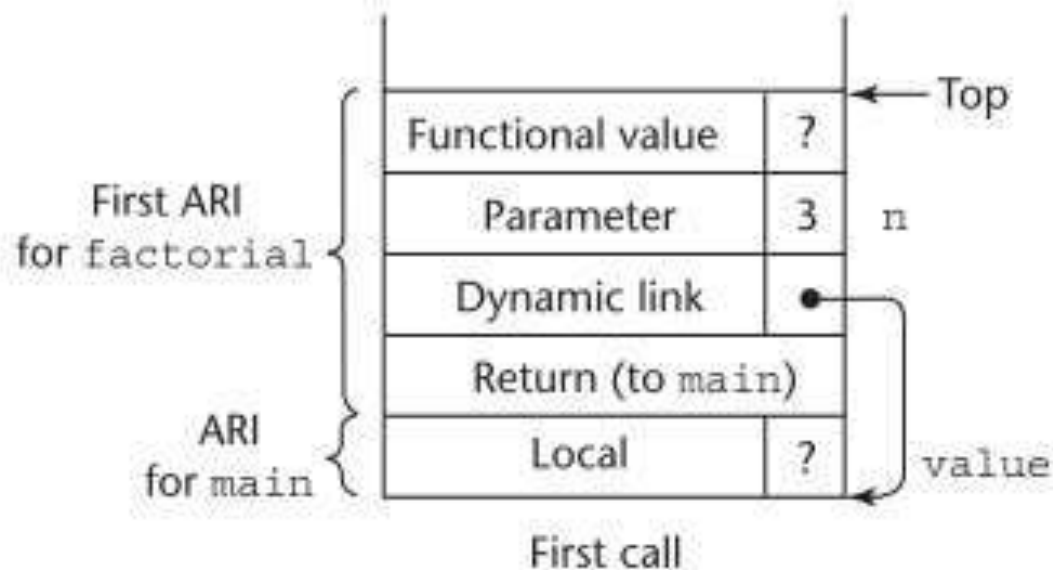
# Recursion

```
int factorial (int n) {
 <-----1
 if (n <= 1)
 return 1;
 else return (n * factorial(n - 1));
 <-----2
}
void main() {
 int value;
 value = factorial(3);
 <-----3
}
```

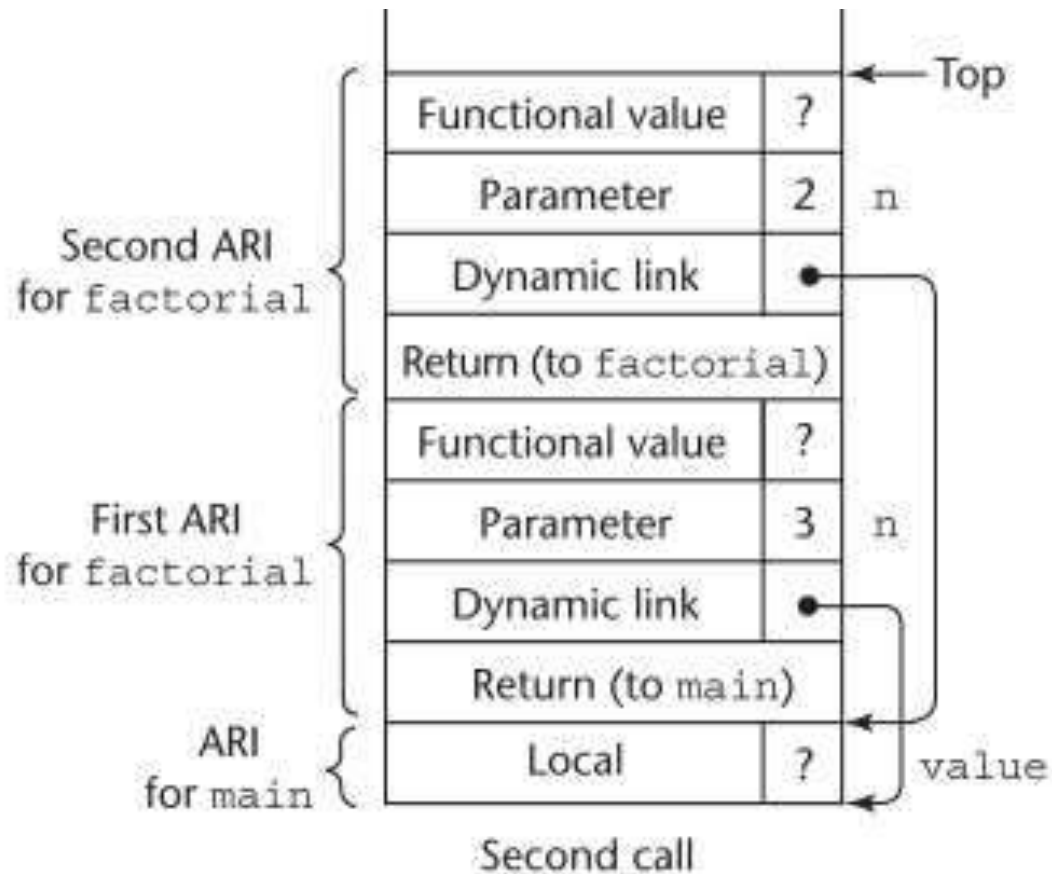
# Activation Record for factorial



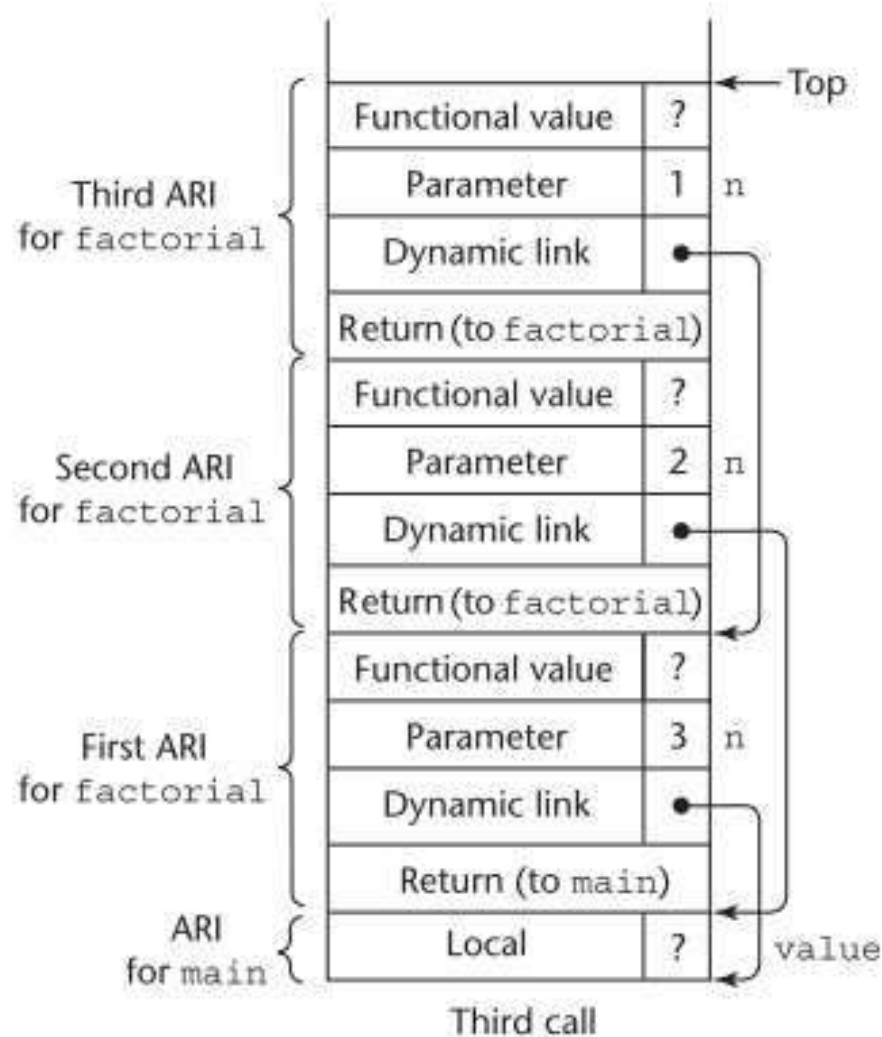
# Stack contents during execution of main and factorial



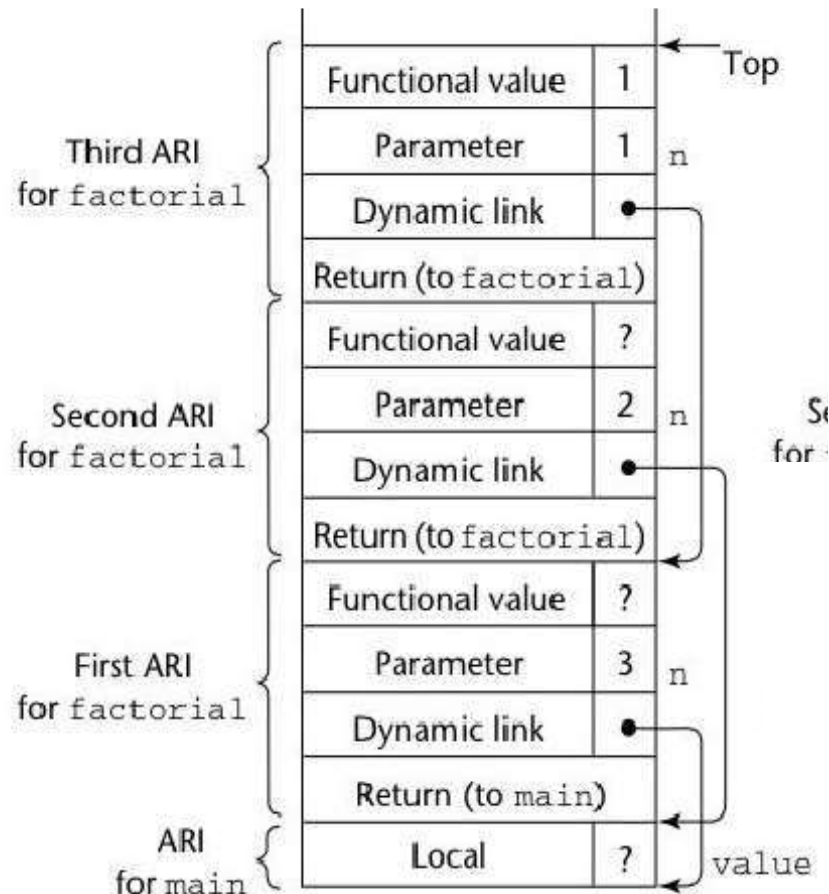
# Stack contents during execution of main and factorial



# Stack contents during execution of main and factorial

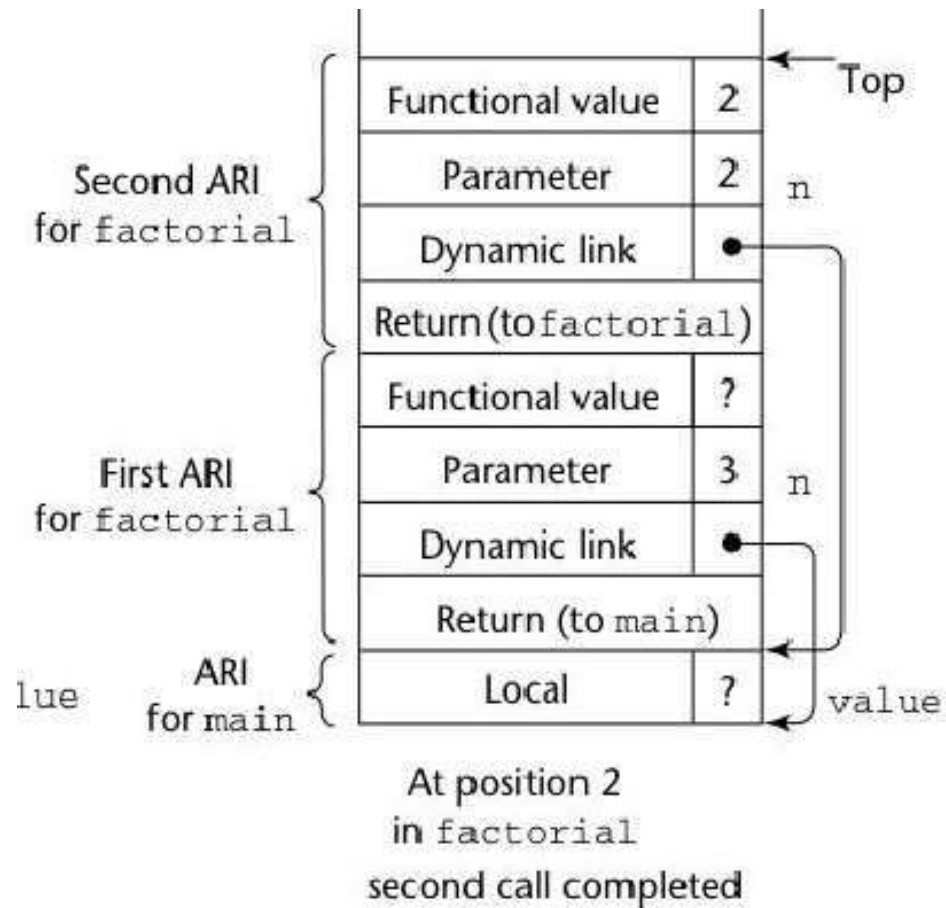


# Stack contents during execution of main and factorial

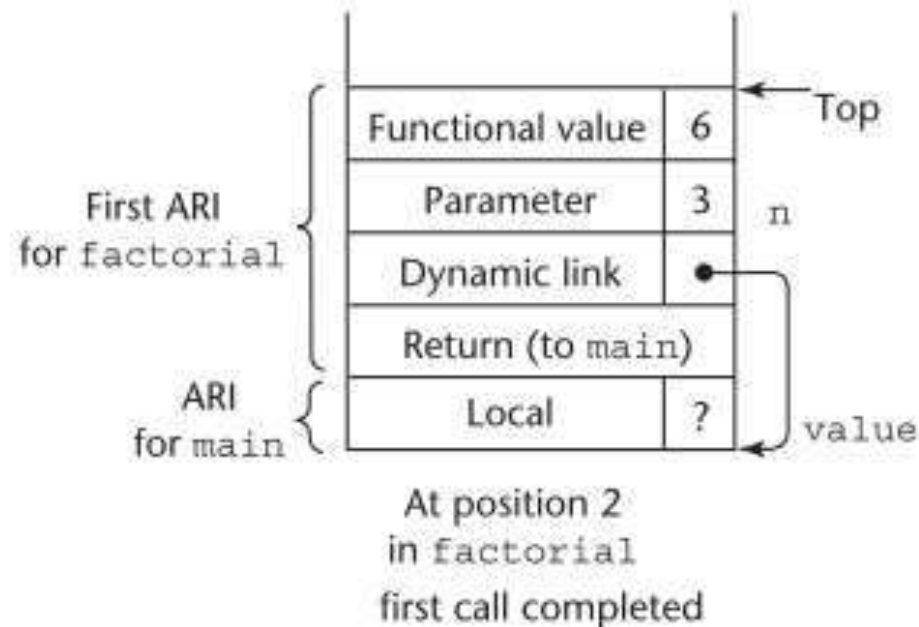


At position 2  
in factorial  
third call completed

# Stack contents during execution of main and factorial

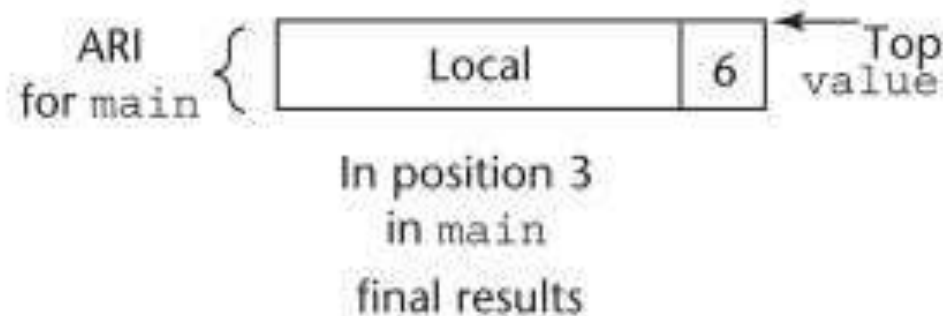


# Stack contents during execution of main and factorial





# Stack contents during execution of main and factorial



# Nested Subprograms

- Some non-C-based static-scoped languages (e.g., Fortran 95, Ada, Python, JavaScript, Ruby, and Lua) use stack-dynamic local variables and allow subprograms to be nested
- All variables that can be non-locally accessed reside in some activation record instance in the stack
- The process of locating a non-local reference:
  1. Find the correct activation record instance
  2. Determine the correct offset within that activation record instance

# Locating a Non-local Reference

- Finding the offset is easy
- Finding the correct activation record instance
  - Static semantic rules guarantee that all non-local variables that can be referenced have been allocated in some activation record instance that is on the stack when the reference is made

# Implementing Static Scoping

- A *static chain* is a chain of static links that connects certain activation record instances
- The *static link* in an activation record instance for subprogram A points to the bottom of one of the activation record instances of A's static parent
- The static chain from an activation record instance connects it to all of its static ancestors
- *Static\_depth* is an integer associated with a static scope whose value is the depth of nesting => indicates how deeply it is nested in the outermost scope

# Static Scoping (continued)

- The *chain\_offset* or *nesting\_depth* of a nonlocal reference is the difference between the *static\_depth* of the reference and *static\_depth* of the procedure containing its declaration
- A reference to a variable can be represented by the pair:  
(*chain\_offset*, *local\_offset*),  
where *local\_offset* is the offset in the activation record of the variable being referenced

# Example Ada Program\*

```
procedure Main_2 is
 X : Integer;
 procedure Bigsub is
 A, B, C : Integer;
 procedure Sub1 is
 A, D : Integer;
 begin -- of Sub1
 A := B + C; <-----1
 end; -- of Sub1
 procedure Sub2(X : Integer) is
 B, E : Integer;
 procedure Sub3 is
 C, E : Integer;
 begin -- of Sub3
 Sub1;
 E := B + A; <-----2
 end; -- of Sub3
 begin -- of Sub2
 Sub3;
 A := D + E; <-----3
 end; -- of Sub2 }
 begin -- of Bigsub
 Sub2(7);
 end; -- of Bigsub
 begin
 Bigsub;
 end; of Main_2 }
```

## Example Ada Program (continued)

- Call sequence for `Main_2`

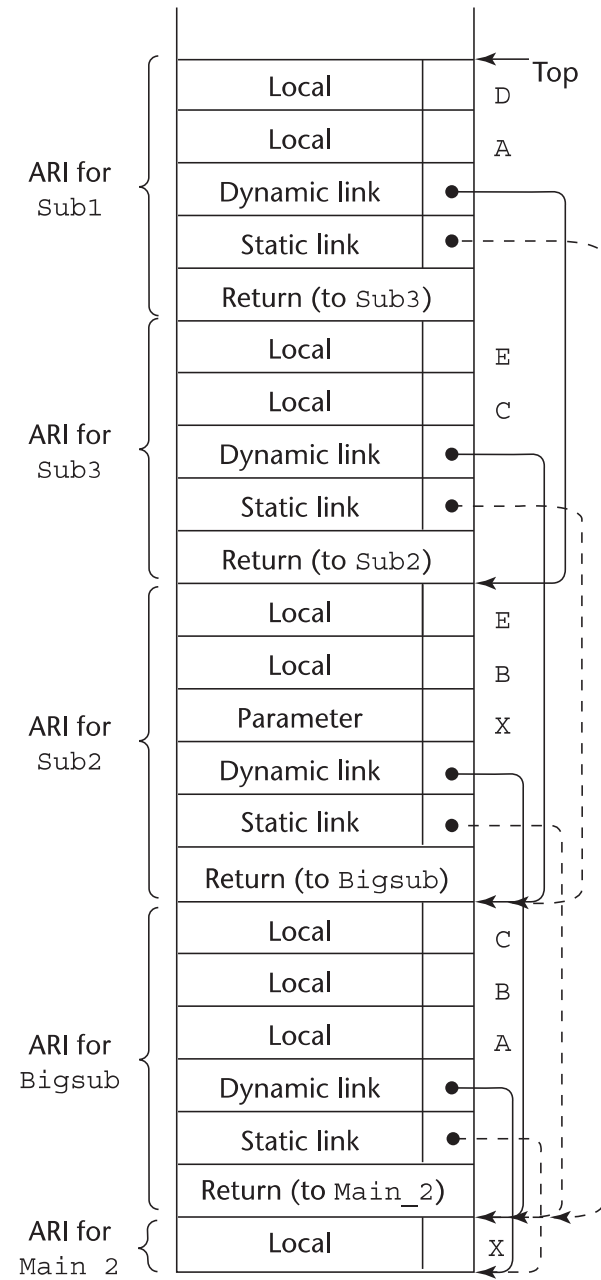
`Main_2` **calls** `Bigsub`

`Bigsub` **calls** `Sub2`

`Sub2` **calls** `Sub3`

`Sub3` **calls** `Sub1`

# Stack Contents at Position 1





# Evaluation of Static Chains

- Problems:
  1. A nonlocal reference is slow if the nesting depth is large
  2. Time-critical code is difficult:
    - a. Costs of nonlocal references are difficult to determine
    - b. Code changes can change the nesting depth, and therefore the cost

# Blocks

- Recall that blocks are user-specified local scopes for variables

- An example in C

```
{int temp;
 temp = list [upper];
 list [upper] = list [lower];
 list [lower] = temp
}
```

- The lifetime of `temp` in the above example begins when control enters the block
- An advantage of using a local variable like `temp` is that it cannot interfere with any other variable with the same name

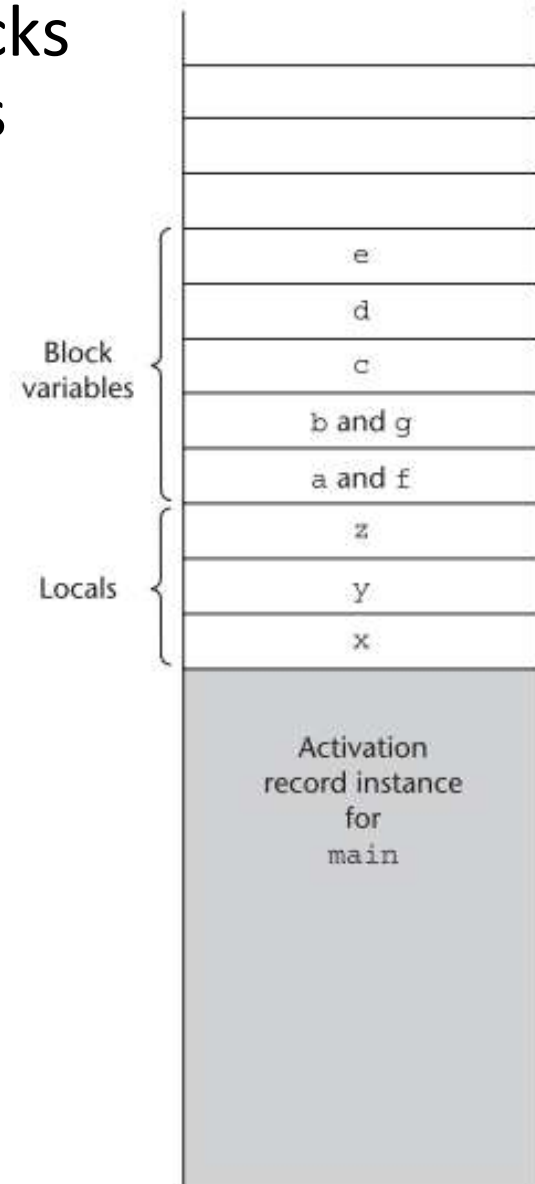
# Implementing Blocks

- Two Methods:
  1. Treat blocks as parameter-less subprograms that are always called from the same location
    - Every block has an activation record; an instance is created every time the block is executed
  2. Since the maximum storage required for a block can be statically determined, this amount of space can be allocated after the local variables in the activation record

# Implementing Blocks

- Block variable storage when blocks are not treated as parameterless procedures

```
void main() {
 int x, y, z;
 while (...) {
 int a, b, c;
 ...
 while (...) {
 int d, e;
 ...
 }
 }
 while (...) {
 int f, g;
 ...
 }
 ...
}
```

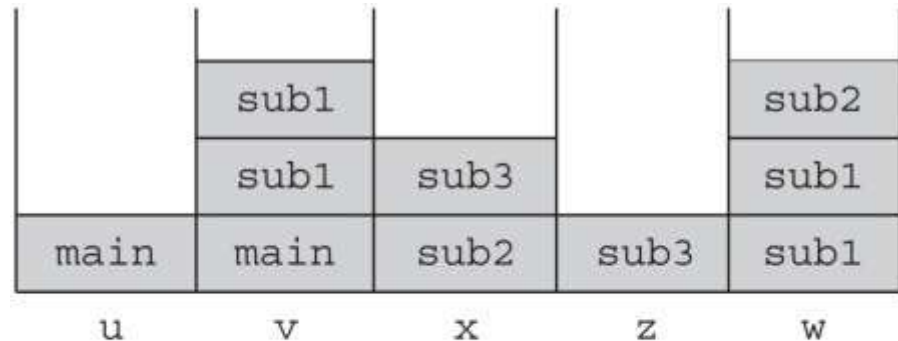


# Implementing Dynamic Scoping

- *Deep Access*: non-local references are found by searching the activation record instances on the dynamic chain
  - Length of the chain cannot be statically determined
  - Every activation record instance must have variable names
- *Shallow Access*: put locals in a central place
  - One stack for each variable name
  - Central table with an entry for each variable name

# Using Shallow Access to Implement Dynamic Scoping

```
void sub3() {
 int x, z;
 x = u + v;
 ...
}
void sub2() {
 int w, x;
 ...
}
void sub1() {
 int v, w;
 ...
}
void main() {
 int v, u;
 ...
}
```



(The names in the stack cells indicate the program units of the variable declaration.)

main calls sub1  
sub1 calls sub1  
sub1 calls sub2  
sub2 calls sub3

# Summary

- Subprogram linkage semantics requires many action by the implementation
- Simple subprograms have relatively basic actions
- Stack-dynamic languages are more complex
- Subprograms with stack-dynamic local variables and nested subprograms have two components
  - actual code
  - activation record

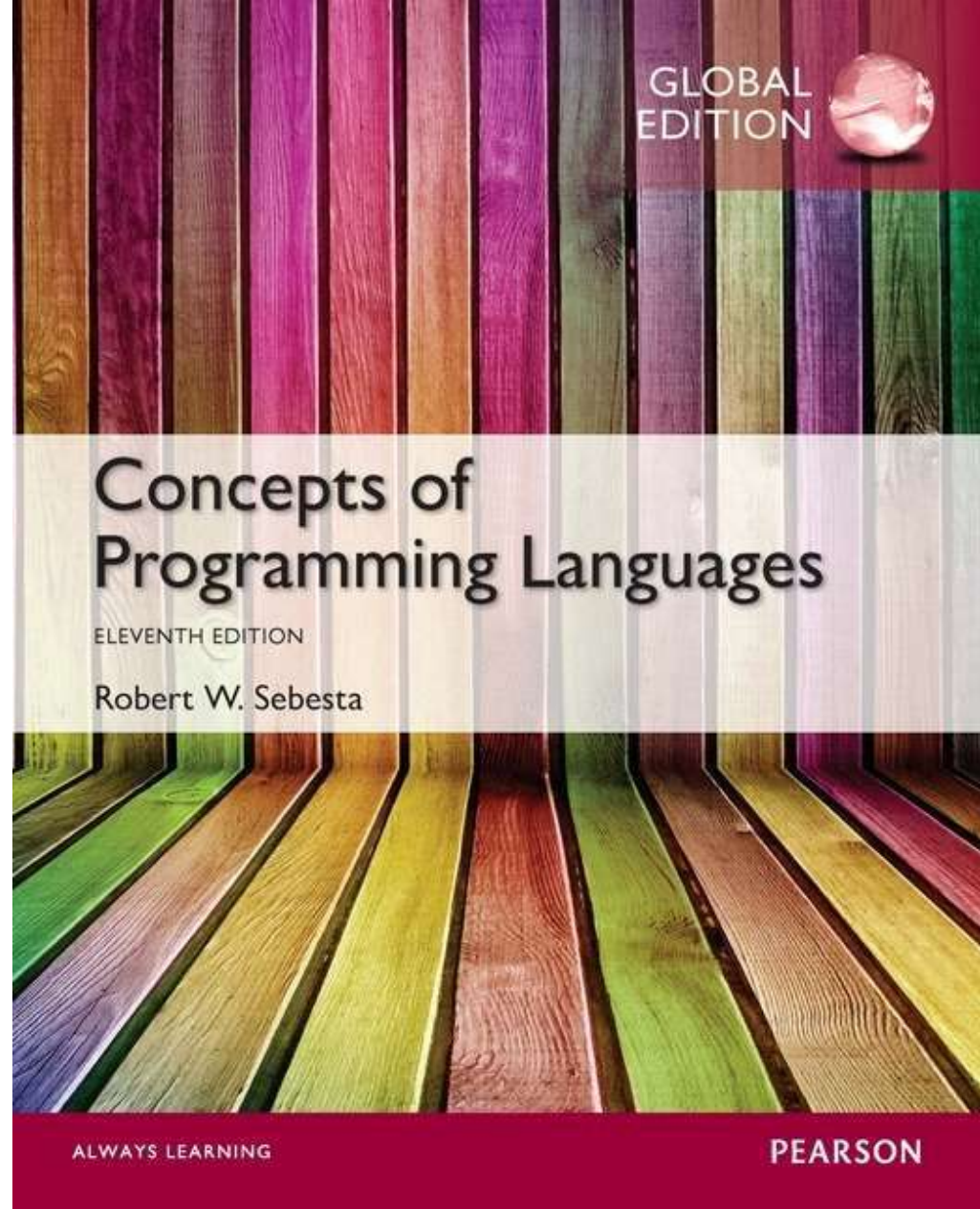
# Summary (continued)

- Activation record instances contain formal parameters and local variables among other things
- Static chains are the primary method of implementing accesses to non-local variables in static-scoped languages with nested subprograms
- Access to non-local variables in dynamic-scoped languages can be implemented by use of the dynamic chain or thru some central variable table method



# Chapter 16

## Logic Programming Languages



# Chapter 16 Topics

---

- Introduction
- A Brief Introduction to Predicate Calculus
- Predicate Calculus and Proving Theorems
- An Overview of Logic Programming
- The Origins of Prolog
- The Basic Elements of Prolog
- Deficiencies of Prolog
- Applications of Logic Programming

# Introduction

---

- Programs in logic languages are expressed in a form of symbolic logic
- Use a logical inferencing process to produce results
- *Declarative* rather than *procedural*:
  - Only specification of *results* are stated (not detailed *procedures* for producing them)

# Proposition

---

- A logical statement that may or may not be true
  - Consists of objects and relationships of objects to each other

# Symbolic Logic

---

- Logic which can be used for the basic needs of formal logic:
  - Express propositions
  - Express relationships between propositions
  - Describe how new propositions can be inferred from other propositions
- Particular form of symbolic logic used for logic programming called *predicate calculus*

# Object Representation

---

- Objects in propositions are represented by simple terms: either constants or variables
- *Constant*: a symbol that represents an object
- *Variable*: a symbol that can represent different objects at different times
  - Different from variables in imperative languages

# Compound Terms

---

- *Atomic propositions* consist of compound terms
- *Compound term*: one element of a mathematical relation, written like a mathematical function
  - Mathematical function is a mapping
  - Can be written as a table

# Parts of a Compound Term

---

- Compound term composed of two parts
  - Functor: function symbol that names the relationship
  - Ordered list of parameters (tuple)
- Examples:

```
student(jon)
```

```
like(seth, OSX)
```

```
like(nick, windows)
```

```
like(jim, linux)
```



# Forms of a Proposition

---

- Propositions can be stated in two forms:
  - *Fact*: proposition is assumed to be true
  - *Query*: truth of proposition is to be determined
- Compound proposition:
  - Have two or more atomic propositions
  - Propositions are connected by operators

# Logical Operators

---

| Name        | Symbol    | Example       | Meaning              |
|-------------|-----------|---------------|----------------------|
| negation    | $\neg$    | $\neg a$      | not a                |
| conjunction | $\cap$    | $a \cap b$    | a and b              |
| disjunction | $\cup$    | $a \cup b$    | a or b               |
| equivalence | $\equiv$  | $a \equiv b$  | a is equivalent to b |
| implication | $\supset$ | $a \supset b$ | a implies b          |
|             | $\subset$ | $a \subset b$ | b implies a          |

# Quantifiers

---

| Name        | Example       | Meaning                                       |
|-------------|---------------|-----------------------------------------------|
| universal   | $\forall X.P$ | For all X, P is true                          |
| existential | $\exists X.P$ | There exists a value of X such that P is true |

# Clausal Form

---

- Too many ways to state the same thing
- Use a standard form for propositions
- *Clausal form*:
  - $B_1 \cup B_2 \cup \dots \cup B_n \subset A_1 \cap A_2 \cap \dots \cap A_m$
  - means if all the  $A$ s are true, then at least one  $B$  is true
- *Antecedent*: right side
- *Consequent*: left side
- All predicate calculus propositions can be algorithmically converted to clausal form.

# Example Clausal Forms

---

$\text{likes}(\text{bob}, \text{trout}) \subset \text{likes}(\text{bob}, \text{fish}) \cap \text{fish}(\text{trout})$

if bob likes fish and a trout is a fish, then bob likes trout.

$\text{father}(\text{louis}, \text{al}) \cup \text{father}(\text{louis}, \text{violet}) \subset$   
 $\text{father}(\text{al}, \text{bob}) \cap \text{mother}(\text{violet}, \text{bob}) \cap \text{grandfather}(\text{louis}, \text{bob})$

if al is bob's father and violet is bob's mother and  
louis is bob's grandfather, then  
louis is either al's father or violet's father

# Predicate Calculus and Proving Theorems

---

- A use of propositions is to discover new theorems that can be inferred from known axioms and theorems
- *Resolution*: an inference principle that allows inferred propositions to be computed from given propositions

# Concept of Resolution

---

Suppose there are two propositions of the form:

$$\begin{aligned} P_1 &\subset P_2 \\ Q_1 &\subset Q_2 \end{aligned}$$

Suppose  $P_1$  is identical to  $Q_2$

$$\begin{aligned} T &\subset P_2 \\ Q_1 &\subset T \end{aligned}$$

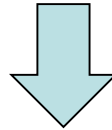
We can write

$$Q_1 \subset P_2$$

# Concept of Resolution

---

$\text{older}(\text{joanne}, \text{jake}) \subset \text{mother}(\text{joanne}, \text{jake})$   
 $\text{wiser}(\text{joanne}, \text{jake}) \subset \text{older}(\text{joanne}, \text{jake})$



$\text{wiser}(\text{joanne}, \text{jake}) \subset \text{mother}(\text{joanne}, \text{jake})$

## The mechanics:

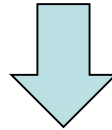
- Terms of the left sides are OR'd
- Terms of the right sides are AND'd.
- Any term that appears on both sides is removed.



# Concept of Resolution

---

$\text{father}(\text{bob}, \text{jake}) \cup \text{mother}(\text{bob}, \text{jake}) \subset \text{parent}(\text{bob}, \text{jake})$   
 $\text{grandfather}(\text{bob}, \text{fred}) \subset \text{father}(\text{bob}, \text{jake}) \cap \text{father}(\text{jake}, \text{fred})$



$\text{mother}(\text{bob}, \text{jake}) \cup \text{grandfather}(\text{bob}, \text{fred}) \subset$   
 $\text{parent}(\text{bob}, \text{jake}) \cap \text{father}(\text{jake}, \text{fred})$

## The mechanics:

- Terms of the left sides are OR'd
- Terms of the right sides are AND'd.
- Any term that appears on both sides is removed.

# Resolution

---

- *Unification*: finding values for variables in propositions that allows matching process to succeed
- *Instantiation*: assigning temporary values to variables to allow unification to succeed
- After instantiating a variable with a value, if matching fails, may need to *backtrack* and instantiate with a different value

# Proof by Contradiction

---

- *Hypotheses*: a set of pertinent propositions
- *Goal*: negation of theorem stated as a proposition
- Theorem is proved by finding an inconsistency

# Theorem Proving

---

- Basis for logic programming
- When propositions used for resolution, only restricted form can be used
- *Horn clause* – can have only two forms
  - *Headed*: single atomic proposition on left side
  - *Headless*: empty left side (used to state facts)
- Most propositions can be stated as Horn clauses

# Overview of Logic Programming

---

- Declarative semantics
  - There is a simple way to determine the meaning of each statement
  - Simpler than the semantics of imperative languages
- Programming is nonprocedural
  - Programs do not state how a result is to be computed, but rather the form of the result
  - *Programming in both imperative and functional languages is procedural*, which means that the programmer instructs the computer on exactly how the computation is to be done.

# Example: Sorting a List

---

- Describe the characteristics of a sorted list, not the process of rearranging a list

$\text{sort}(\text{old\_list}, \text{new\_list}) \subset \text{permute}(\text{old\_list}, \text{new\_list}) \cap \text{sorted}(\text{new\_list})$

$\text{sorted}(\text{list}) \subset \forall_j \text{ such that } 1 \leq j < n, \text{list}(j) \leq \text{list}(j+1)$

# The Origins of Prolog

---

- University of Aix–Marseille (Calmerauer & Roussel)
  - Natural language processing
- University of Edinburgh (Kowalski)
  - Automated theorem proving

# Terms

---

- This book uses the Edinburgh syntax of Prolog
- *Term*: a constant, variable, or structure
- *Constant*: an atom or an integer
- *Atom*: symbolic value of Prolog
- Atom consists of either:
  - a string of letters, digits, and underscores beginning with a lowercase letter
  - a string of printable ASCII characters delimited by apostrophes



# Terms: Variables and Structures

---

- *Variable*: any string of letters, digits, and underscores beginning with an uppercase letter
- *Instantiation*: binding of a variable to a value
  - Lasts only as long as it takes to satisfy one complete goal
- *Structure*: represents atomic proposition  
functor (*parameter list*)

# Fact Statements

---

- Used for the hypotheses
- Headless Horn clauses

```
female(shelley) .
```

```
male(bill) .
```

```
father(bill, jake) .
```

```
father(bill, shelley) .
```

```
mother(mary, jake) .
```

```
mother(mary, shelley) .
```

**Notice that every Prolog statement is terminated by a period.**

# Rule Statements

---

- Used for the hypotheses
- Headed Horn clause
- Right side: *antecedent* (*if* part)
  - May be single term or conjunction
- Left side: *consequent* (*then* part)
  - Must be single term
- *Conjunction*: multiple terms separated by logical AND operations (implied)

`female(shelley), child(shelley) .`

# Rule Statements

---

- The general form of the Prolog headed Horn clause statement is  
consequence :- antecedent\_expression.

“consequence can be concluded if the antecedent expression is true or can be made to be true by some instantiation of its variables.”

# Example Rules

---

```
ancestor(mary, shelley) :- mother(mary, shelley) .
```

- Can use variables (*universal objects*) to generalize meaning:

```
parent(X, Y) :- mother(X, Y) .
```

```
parent(X, Y) :- father(X, Y) .
```

```
grandparent(X, Z) :- parent(X, Y) , parent(Y, Z) .
```

Here X, Y, Z are variables (they start with uppercase letters)

# Goal Statements

---

- For theorem proving, theorem is in form of proposition that we want system to prove or disprove – *goal statement*
- Same format as headless Horn

`man(fred)`

- Conjunctive propositions and propositions with variables also legal goals

`father(X, mike)`

# Inferencing Process of Prolog

---

- Queries are called goals
- If a goal is a compound proposition, each of the facts is a subgoal
- To prove a goal is true, must find a chain of inference rules and/or facts. For goal Q:

$P_2 \text{ :- } P_1$

$P_3 \text{ :- } P_2$

...

$Q \text{ :- } P_n$

- Process of proving a subgoal called matching, satisfying, or resolution

# Approaches

---

- *Matching* is the process of proving a proposition
- Proving a subgoal is called *satisfying* the subgoal
- *Bottom-up resolution, forward chaining*
  - Begin with facts and rules of database and attempt to find sequence that leads to goal
  - Works well with a large set of possibly correct answers
- *Top-down resolution, backward chaining*
  - Begin with goal and attempt to find sequence that leads to set of facts in database
  - Works well with a small set of possibly correct answers
- Prolog implementations use backward chaining



# Subgoal Strategies

---

- When goal has more than one subgoal, can use either
  - Depth-first search: find a complete proof for the first subgoal before working on others
  - Breadth-first search: work on all subgoals in parallel
- Prolog uses depth-first search
  - Can be done with fewer computer resources

# Backtracking

---

- With a goal with multiple subgoals, if fail to show truth of one of subgoals, reconsider previous subgoal to find an alternative solution: *backtracking*
- Begin search where previous search left off
- Can take lots of time and space because may find all possible proofs to every subgoal

# Simple Arithmetic

---

- Prolog supports integer variables and integer arithmetic
- `is` operator: takes an arithmetic expression as right operand and variable as left operand

`A is B / 17 + C`

- Not the same as an assignment statement!
  - The following is illegal (Left side variable cannot be previously instantiated)

`Sum is Sum + Number.`

# Example

---

```
speed(ford,100) .
speed(chevy,105) .
speed(dodge,95) .
speed(volvo,80) .
time(ford,20) .
time(chevy,21) .
time(dodge,24) .
time(volvo,24) .
distance(X,Y) :- speed(X,Speed) ,
 time(X,Time) ,
 Y is Speed * Time.
```

**A query:** distance(chevy, Chevy\_Distance) .

# Trace

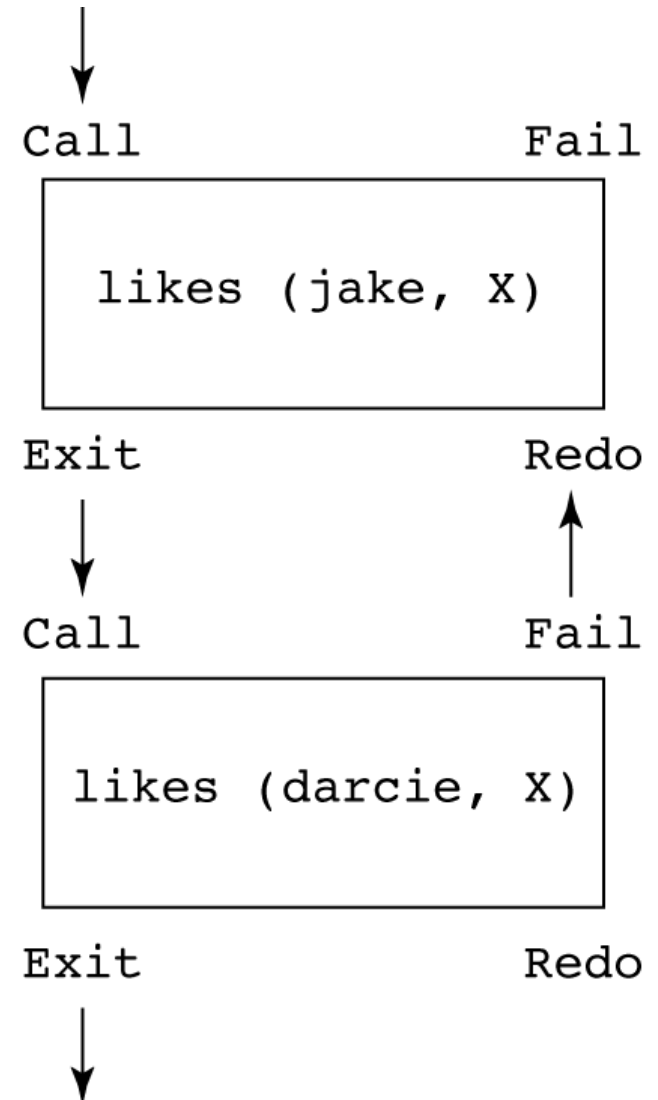
---

- Built-in structure that displays instantiations at each step
- *Tracing model* of execution – four events:
  - *Call* (beginning of attempt to satisfy goal)
  - *Exit* (when a goal has been satisfied)
  - *Redo* (when backtrack occurs)
  - *Fail* (when goal fails)

# Example

```
likes(jake, chocolate).
likes(jake, apricots).
likes(darcie, licorice).
likes(darcie, apricots).

trace.
likes(jake, X), likes(darcie, X).
(1) 1 Call: likes(jake, _0)?
(1) 1 Exit: likes(jake, chocolate)
(2) 1 Call: likes(darcie, chocolate)?
(2) 1 Fail: likes(darcie, chocolate)
(1) 1 Redo: likes(jake, _0)?
(1) 1 Exit: likes(jake, apricots)
(3) 1 Call: likes(darcie, apricots)?
(3) 1 Exit: likes(darcie, apricots)
X = apricots
```



# List Structures

---

- Other basic data structure (besides atomic propositions we have already seen): list
- *List* is a sequence of any number of elements
- Elements can be atoms, atomic propositions, or other terms (including other lists)

[apple, prune, grape, kumquat]

[]                   (*empty list*)

[X | Y]           (*head X and tail Y*)

# Append Example

---

```
append([], List, List).
append([Head | List_1], List_2, [Head | List_3]) :-
 append (List_1, List_2, List_3).
```



# More Examples

---

```
reverse([], []).
reverse([Head | Tail], List) :-
 reverse (Tail, Result),
 append (Result, [Head], List).
```

```
member(Element, [Element | _]).
member(Element, [_ | List]) :-
 member(Element, List).
```

The underscore character means an anonymous variable—it means we do not care what instantiation it might get from unification

# Deficiencies of Prolog

---

- Resolution order control
  - In a pure logic programming environment, the order of attempted matches is nondeterministic and all matches would be attempted concurrently
- The closed-world assumption
  - The only knowledge is what is in the database
- The negation problem
  - Anything not stated in the database is assumed to be false
- Intrinsic limitations
  - It is easy to state a sort process in logic, but difficult to actually do—it doesn't know how to sort

# Applications of Logic Programming

---

- Relational database management systems
- Expert systems
- Natural language processing

# Summary

---

- Symbolic logic provides basis for logic programming
- Logic programs should be nonprocedural
- Prolog statements are facts, rules, or goals
- Resolution is the primary activity of a Prolog interpreter
- Although there are a number of drawbacks with the current state of logic programming it has been used in a number of areas