# CS/CE 4337 Programming Language Paradigms

## Summer 2025

**Instructor: Dr. Yi Zhao**

# Introduction

# Instructor

**Yi Zhao, Ph.D.**

https://cs.utdallas.edu/people/faculty/zhao-yi/

Assistant Professor of Instruction

Faculty Career Fellow

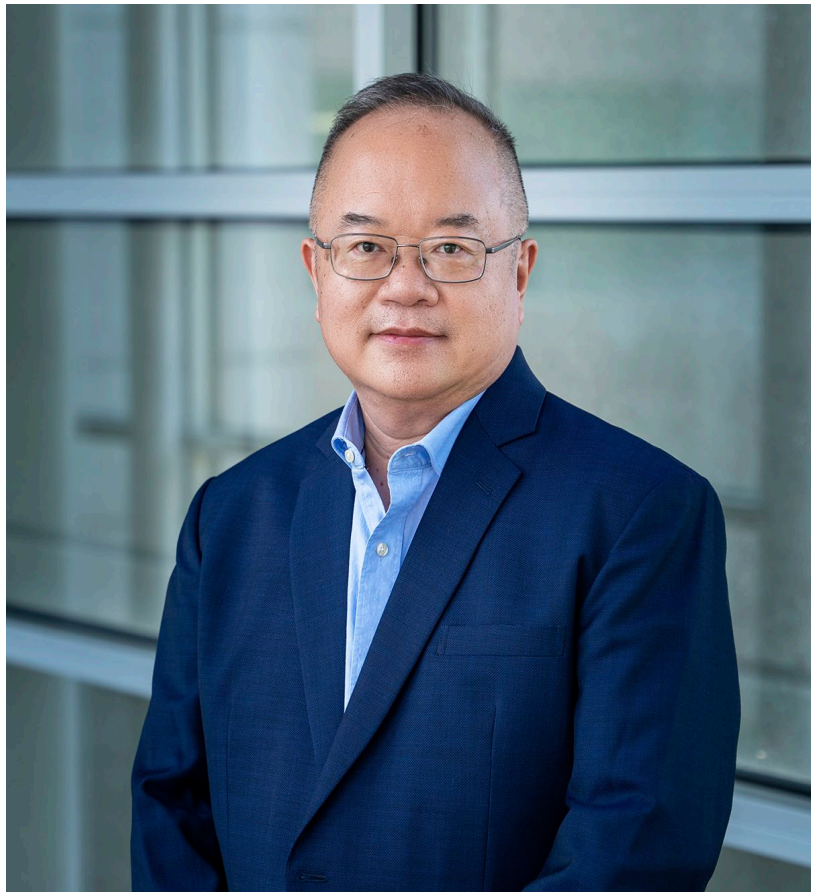
Department of Computer Science

The University of Texas at Dallas

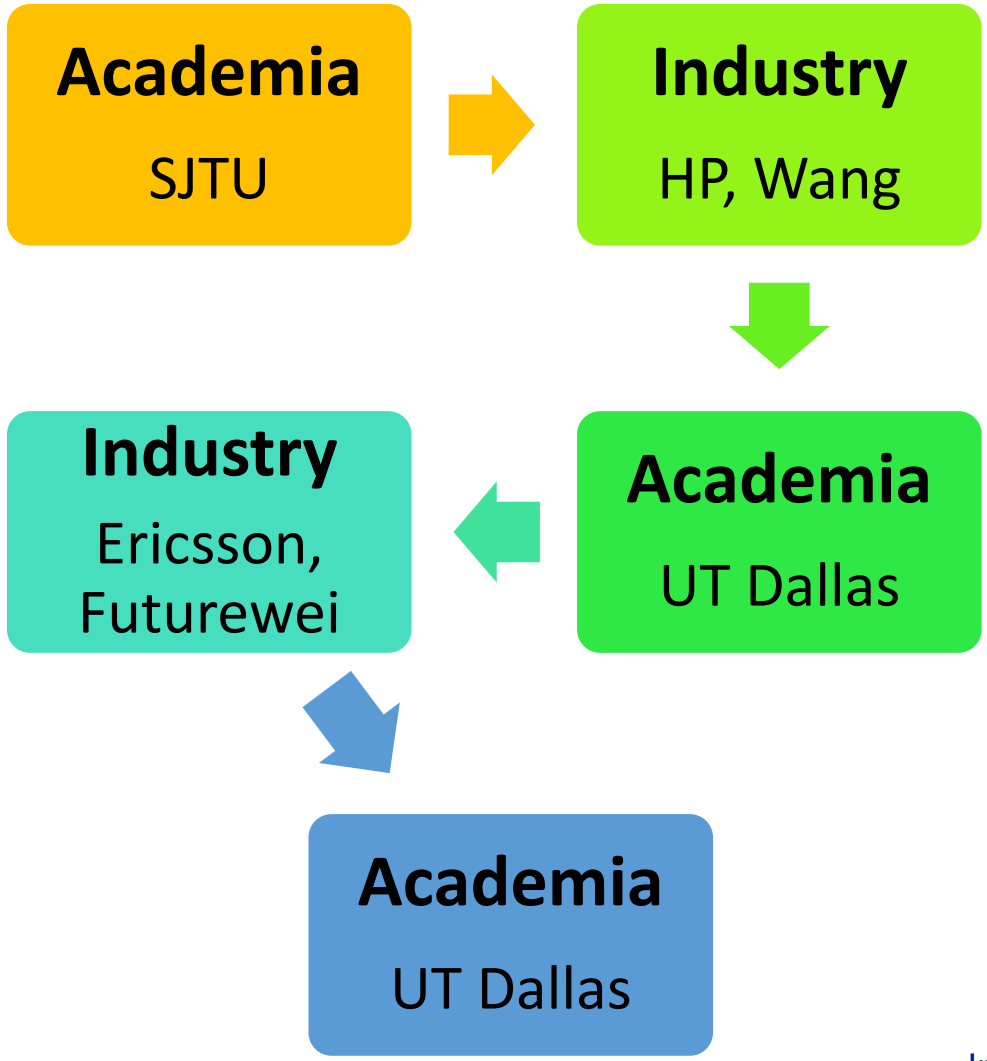
Richardson, TX 75083, USA

Office: ECSN 2.916, phone 972 883 2693

yi.zhao@utdallas.edu



# Yi Zhao, Ph.D.




- Lecturer, Researcher, Professor, EMBA mentor
- Engineer, Technical lead/coordinator, Specialist engineer, Consultant
- Team leader, Project manager, Program manager, Senior executive, Founders

- Corporate development and program management
- Technology, Business and Strategy
- Education and career advising

- Global industry leadership
- Community leadership

# Instructor

## EDUCATION

**Ph.D. Computer Science** (AI and NLP), University of Texas at Dallas, Richardson, Texas, USA, 1996
**MS Computer Science**, University of Texas at Dallas, Richardson, Texas, USA, 1992
**BE / ME Computer Engineering**, Shanghai Jiao Tong University, Shanghai, China, 1983, 1986

## CAREER SUMMARY

**Assistant Professor of Instruction**, Department of Computer Science, UT Dallas, Richardson, Texas, USA, 2021-
**Faculty Career Fellow**, Erik Jonsson School of Engineering and Computer Science, UTD, Richardson, USA, 2022-
**Founder,** InfoWhiz, Plano, TX, USA, Consulting on 5G, Open RAN, AI, Social Mining, IoT, and Standards, 2019 –
**Co-Founder**, InfoAuthN AI, Plano, TX, USA, Solution on disinformation mitigation, 2021 -
**VP of Standards and Industry Development**, Futurewei Technologies Inc., Plano, TX, USA, 2009 – 2019
**VP of Operation**, Futurewei Technologies Inc., Plano, TX, USA, 2000 – 2009
**Project Manager/Specialist Engineer/Technical Coordinator**, Ericsson Inc., Richardson, TX, USA, 1996 – 2000
**Teaching Assistant**, the University of Texas at Dallas, Richardson, TX, USA, Research on AI, 1991 – 1992
**Research Assistant**, the University of Texas at Dallas, Richardson, TX, USA, Research on AI, 1992 – 1996
**Technical Lead/Program Manager**, Hewlett-Packard, Cupertino, California, USA/Shanghai, China, 1988 -1990
**Lecturer**, Computer Science, Shanghai Jiao Tong University, Shanghai, China, 1986 – 1991

# Instructor

## SELECTED ACCOMPLISHMENTS

***Technology, Business, and Strategy***

- Strategically identified, acquired, and consolidated a major startup into a leading research lab that enabled the product breakthrough in the global tier one markets.
- Board Members of major industry organizations (ATIS, INCITS, the US National Committee for IEC, ISOC) advising on regional and global technology trends and strategic standards needs and new initiatives on technologies and ecosystem.
- Board Director in Open Patent Alliance leading the WiMAX patent pool negotiation and industry negotiation on LTE patent pool, and maximizing the company benefits while keeping the industry balance.
- Won the company's first two wireless contracts in the U.S. market.
- Promoted and advanced open source with the focus on the integration of standards and open source and provided leadership on open source in IEEE, JTC 1, and other organizations.
- Technology and business consulting in 5G, Open RAN, AI, IoT, autonomous driving, and standards.

***Program Management and Corporate Development***

- As the company's first U.S. employee, successfully established the Futurewei and grew the organization from scratch to a top research center (1000+ employees).
- Established and led the company's largest overseas competence center for standards and industry development comprising 100+ senior experts with key leadership positions in various industry organizations such as 3GPP, IETF, IEEE, JTC1, etc.
- Established and managed the company IETF program from zero to 100+ participants and made the company a major contributor and a leader in IETF/IRTF communities.
- Initiated and led company's program on ISO/IEC JTC 1 standards through multiple national bodies and made company a world leader on JTC 1 standards of AI, MPEG, and Cloud.
- Established and managed the North America IPR program to quickly build up a strong IPR portfolio resulting in a solid foundation for IPR litigations and licensing.
- Initiated and managed the SONET product design program creating the first revenue in the US market.
- Launched User Centered Design program which led to the institution of a global UCD Design Center.
- Established and led the Export Control Management program to comply with the U.S. regulations which promoted the initiation of the corporate global compliance program.
- Project managers for SS7 signaling software development and mainframe OS functional testing.

# Instructor

## Teaching Experience

- Instructor of following courses in UTD
  - **ECS 1100**: Introduction to Engineering and Computer Science
  - **CS/SE 2340**: Computer Architecture
  - **CS/CE/SE 3345**: Data Structure and Algorithm Analysis
  - **CS/CE 4337**: Programming Language Paradigms
  - **CS 5330**: Computer Science II
  - **SE 6388**: Project Management (for Executive Master in Software Engineering)
- Instructor of computer science undergraduate/graduate courses in programming, automata, and programming methodologies in SJTU and won the Best Teaching of the Year.
- Provided corporate and industry training on technology, strategy, and industry standards.
- Student Career Advisor, Computer Science, UT Dallas

# Instructor

## GLOBAL INDUSTRY LEADERSHIP

**Board Director**, Alliance for Telecommunications Industry Solutions (ATIS)

**Executive Board Director**, International Committee for Information Technology Standards (INCITS) – The US National Body for ISO/IEC/JTC 1

**Council Member**, the U.S. National Committee of the International Electrotechnical Commission (USNC/IEC)

**Board Director**, Open Patent Alliance (OPA) – WiMax Patent Pool

**Board Director**, IPsphere Forum

**Council Member**, the ISOC (Internet Society) Corporate Advisory Council

**Chair**, The US delegation to JETI (JTC 1 Emerging Technology Initiatives) on AI, Quantum Computing, Autonomous Vehicle, and other emerging technologies.

**Chair**, INCITS SG on Open-Source initiative

**JTC 1 Liaison** to IEC/TC65 on smart manufacturing

## AWARDS

The Distinguished Alumni Award from The University of Texas at Dallas. It is the highest honor bestowed on alumni and is given to individuals who are distinguished in their chosen profession or life's work and who demonstrate pride in the University.

## RESEARCH INTERESTS

Disinformation mitigation, 5G, AI, IoT, Standards, IPR, Innovation/Research management.

# Course Syllabus

***Course Syllabus***

**Course Information**

**CS/CE 4337.0U1 Programming Language Paradigms**

| | |
|---|---|
| Term: | Summer 2025 |
| Days & Time and Location: | Monday, Wednesday 10:00am – 12:00pm @ JSOM 12.202 |

**Professor Contact Information**

Yi Zhao, Ph.D.
https://cs.utdallas.edu/people/faculty/zhao-yi/
Phone: 972 883 2693
Email: yi.zhao@utdallas.edu (the best way to reach me)
Office hours: by appointment
Office: ECSN 2.916

**TA/Grader Information**

TBD

# Course Syllabus

**Class Participation**

Regular class participation is expected regardless of course modality. Students who fail to participate in class regularly are inviting scholastic difficulty. A portion of the grade for this course is directly tied to your participation in this class. It also includes engaging in group or other activities during class that solicit your feedback on homework assignments, readings, or materials covered in the lectures (and/or labs). Successful participation is defined as consistently adhering to university requirements, as presented in this syllabus. Failure to comply with these University requirements is a violation of the Student Code of Conduct.

**Class Recordings**

The instructor may record meetings of this course. Any recordings will be available to all students registered for this class as they are intended to supplement the classroom experience. Students are expected to follow appropriate University policies and maintain the security of passwords used to access recorded lectures. Unless the Office of Student Accessibility has approved the student to record the instruction, students are expressly prohibited from recording any part of this course. Recordings may not be published, reproduced, or shared with those not in the class, or uploaded to other online environments except to implement an approved Office of Student Accessibility accommodation. If the instructor or a UTD school/department/office plans any other uses for the recordings, consent of the students identifiable in the recordings is required prior to such use unless an exception is allowed by law. Failure to comply with these University requirements is a violation of the Student Code of Conduct.

**Class Materials**

The instructor may provide class materials that will be made available to all students registered for this class as they are intended to supplement the classroom experience. These materials may be downloaded during the course; however, these materials are for registered students' use only. Classroom materials may not be reproduced or shared with those not in class or uploaded to other online environments except to implement an approved Office of Student Accessibility accommodation. Failure to comply with these University requirements is a violation of the Student Code of Conduct.

# Course Syllabus

**Course Pre-requisites, Co-requisites, and/or Other Restrictions**

(CE 2336 or CS 2336 or CS 2337) with a grade of C or better or CS 3333, and (CE 2305 or CS 2305) with a grade of C or better and (CS 2340 or SE 2340 or CE 4304 or EE 4304).

**Course Description**

CS 4337 - Programming Language Paradigms (3 semester credit hours) Principles of design and implementation of contemporary programming languages. Formal description including specification of syntax and semantics of programming languages. Language definition structures including binding, scoping, data types, control structures, parameter passing, abstraction mechanism, and run-time considerations. Design issues of different programming languages. Language-based security. Design, implement, and debug programs in various programming language paradigms.

# Course Syllabus

**Student Learning Objectives/Outcomes**

1. Ability to identify the characteristics of programming paradigms and phases of translation.
2. Ability to understand the concepts of formal syntax and semantics of programming languages.
3. Ability to understand the concept of language-based security.
4. Ability to understand the concept of web scripting security.
5. Ability to understand the concepts of first-class values, lists, and recursion.
6. Ability to understand the concepts of the functional programming paradigm and design functional programs.
7. Ability to understand the concepts of the logic programming paradigm and design logic programs.

# Course Syllabus

**Required Textbooks and Materials**

"**Concepts of Programming Languages, 12th Edition**", Robert W. Sebesta, Pearson, 2019. ISBN-13 978-0-13-516087-9 / ISBN-10: 0-13-516087-1.

---

**Assignments & Academic Calendar**

**Exams**:
There will be midterm and final exams. Exams will be closed-book and closed-notes.

**Assignments**:
There will be regular homework assignments.

**Quizzes**:
Quizzes will be scheduled on a regular basis. It is also a way of assessing student attendance.

Plagiarism of homework answers found on the Internet is not permitted. Upon discovery, some or all the points will be deducted.

# Course Syllabus

**Midterm Exam: 9am – 1pm, July 7th, 2025**
**Final Exam: 9am – 1pm, August 13th, 2025**

**Testing Center Student Guideline:**
https://ets.utdallas.edu/testing-center/students/

**Register for your exams today!**

**Tentative Class Schedule:**

| *Session* | *Date* | *Topic* | *Material Covered* |
|---|---|---|---|
| 1 | Jun 2 | Syllabus, Introduction | Ch 1 |
| 2 | Jun 4 | Introduction | Ch 1 |
| 3 | Jun 9 | Evolution of Major Programming Languages | Ch 2 |
| 4 | Jun 11 | Syntax and Semantics of Programming Languages | Ch 3, 4 |
| 5 | Jun 16 | Syntax and Semantics of Programming Languages | Ch 3, 4 |
| 6 | Jun 18 | Syntax and Semantics of Programming Languages | Ch 3, 4 |
| 7 | Jun 23 | Syntax and Semantics of Programming Languages | Ch 3, 4 |
| 8 | Jun 25 | Introduction to Language-Based Security | TBA |
| 9 | Jun 30 | Language-based Security: Strings and buffer overflows | TBA |
| 10 | Jul 2 | Return Oriented Programming | TBA |
| **11** | **Jul 7** | **Midterm Exam (9am – 1pm)** | |
| 12 | Jul 9 | CFI-Control Flow Integrity | TBA |
| 13 | Jul 14 | Language-based Security: Java Security | TBA |
| 14 | Jul 16 | Web Scripting Security | TBA |
| 15 | Jul 21 | Web Scripting Security | TBA |
| 16 | Jul 23 | Functional Programming | Ch 15 |
| 17 | Jul 28 | Functional Programming | Ch 15 |
| 18 | Jul 30 | Functional Programming | Ch 15 |
| 19 | Aug 4 | Logic Programming | TBA |
| 20 | Aug 6 | Logic Programming | TBA |
| 21 | Aug 11 | Logic Programming; Software Testing: Basic Fuzzing | TBA |
| **22** | **Aug 13** | **Final Exam (9am – 1pm)** | |

# Course Syllabus

**Grading Policy**
The grade each student will earn from this class will be based on a weighted score calculated by using the following table:

| | |
|---|---|
| Midterm Exam | 33% |
| Final Exam | 33% |
| Homework | 24% |
| Quizzes | 10% |
| | 100% |

**Course & Instructor Policies**

- Attendance policy: missing four in-class quizzes leads to a one letter grade drop, missing five in-class quizzes leads to an F grade.
- There will be *no makeup exams* under normal circumstances.
- *No late assignments (homework/project) submission* will be accepted!
- Use my UTD email account for any correspondence. List the course and section number in the subject of email.

# Course Syllabus

Grades will be assigned according to the scale below:

| Weighted Score | Grade |
|---|---|
| 93.0 - 100 | A |
| 90.0 - 92.9 | A- |
| 87.0 - 89.9 | B+ |
| 83.0 - 86.9 | B |
| 80.0 - 82.9 | B- |
| 77.0 - 79.9 | C+ |
| 73.0 - 76.9 | C |
| 70.0 - 72.9 | C- |
| 67.0 - 69.9 | D+ |
| 60.0 - 66.9 | D |
| Below 60.0 | F |

# Relax and enjoy the semester!

# CS/CE 4337 Programming Language Paradigms

## Summer 2025

**Instructor: Dr. Yi Zhao**

# Chapter 1: Preliminaries

# What is a Language?

- Programming Languages *are* Languages
- Both natural human languages and programming languages can be formally described using the same attributes
- The only significant difference?
  - Programming Languages have unambiguous grammar
  - Natural Human Languages have ambiguous grammar

**Ambiguous Grammar:**

- **"I saw the man with the telescope."**
- **"Visiting relatives can be boring."**

# Attributes of Languages

- **Alphabet**
- **Lexicon**: the vocabulary of a language
- **Syntax**: refers to the rules that govern the ways in which words combine to form phrases, clauses, and sentences.
- **Grammar**: whole system and structure of a language
- **Semantics**: how words are to be understood.
- **Orthography**: deals with the conventions of writing
- **Morphology**: focuses on the internal structure of words and how they are formed.
- **Phonology**: examines the sound patterns and phonetic elements of a language.
- **Typology**: involves the classification of languages based on shared structural features.

# Language

- Language
  - Set of strings, sentences, statements

- Grammar (formal description of a language)
  - Syntax (structure)
  - Semantics (meaning)

# Categories of Languages

- Natural Human Languages
  - English, French, Chinese, Greek, Russian, Latin, etc.
- Constructed Human Languages
  - **Sociological/Political**: Esperanto, Toki Pona, Interlingua, Gestuno (sign)
  - **Fiction**: Klingon, Quenya, Sindarin, Na'vi
  - **Research**: LOGLAN, Lojban, Lojban++
- Programming Languages

**Esperanto:** is an artificial international language based on words common to the chief European languages.

**Klingon** : is the constructed language spoken by a fictional alien race, ...

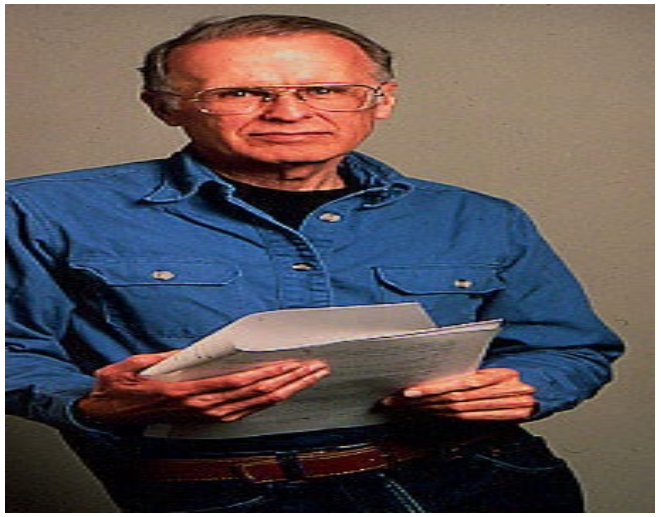
**Loglan** (an abbreviation for "logical language") was created to investigate whether people speaking a "logical language" would in some way think more logically.

# Linguistics

- Link between syntax, grammar, and semantics
  - Noam Chomsky
  - John Backus



**Noam Chomsky** is an American theoretical linguist, cognitive scientist and philosopher (born December 7, 1928)



**John Warner Backus** was an American computer scientist. He led the team that invented the first widely used high-level programming language (FORTRAN) and was the inventor of the **Backus-Naur form (BNF)**
(December 3, 1924 – March 17, 2007)

# One or Many?

- Wouldn't it be great if there were just ONE programming language?
- One programming language to rule them all?



- Why are there so many different programming languages?
- How many?

# Shakespeare

- A programming language designed for educational and creative purposes, where the source code is written in a way that resembles Shakespearean plays.
- An esoteric programming language, which prioritizes artistic expression and unconventional syntax over practicality.

```
The Merchant of Venice.

Romeo, a young man with remarkable patience.
Juliet, a likewise young woman of remarkable grace.

Act I: The setup.
Scene I: Learning to say "Hello World".

[Exeunt]

Romeo:
Am I better than you?

Juliet:
If so, let us proceed to scene II.

[Exeunt]

Scene II: Saying "Hello World".
[Enter Romeo and Juliet]

Romeo:
Listen to thy heart!

[Exeunt]
```

# Whitespace

- Developed by Edwin Brady and Chris Morris at the University of Durham –England (Released 1 April 2003)

- Unlike most programming languages, which ignore or assign little meaning to most whitespace characters, the Whitespace interpreter ignores any non-whitespace characters.
  - **Only spaces (ASCII code 32), tabs (ASCII code 9), and line feeds (ASCII code 10) have meaning.**

- An interesting consequence of this property is that a Whitespace program can easily be contained within the whitespace characters of a program written in another language,

  - Except possibly in languages that depend on spaces for syntax validity such as Python

# 1.1 Reasons for Studying Concepts of Programming Languages

a) Increased ability to express ideas

b) Improved background for choosing appropriate languages

c) Increased ability to learn new languages

d) Better understanding of the significance of implementation

e) Better use of languages that are already known

f) Overall advancement of computing

# a) Increased Ability To Express Ideas

- It is generally accepted that the depth of people's thinking is affected by the expressiveness of the language in which they communicate their thoughts.

- For example, a C programmer who had learned the structure and uses of associative arrays in Perl might design structures that simulate associative arrays in that language.

**Associative arrays**, also called maps or dictionaries, are abstract data types that can hold data in (key, value) pairs.

**Example:**

```
salaries = ("Gary" => 75000, "Perry" => 57000, "Mary" => 55750,
"Cedric" => 47850);
```

# b) Improved Background For Choosing Appropriate Languages

- Many professional programmers have had little formal education in computer science; rather, they have developed their programming skills independently or through in-house training programs.

- Many programmers, when given a choice of languages for a new project, use the language with which they are most familiar, even if it is poorly suited for the project at hand.

- Some of the features of one language often can be simulated in another language.

  - However, it is preferable to use a feature whose design has been integrated into a language than to use a simulation of that feature, which is often less elegant, more cumbersome, and less safe.

# c) Increased Ability To Learn New Languages

- Once a thorough understanding of the fundamental concepts of languages is acquired, it becomes far easier to see how these concepts are incorporated into the design of the language being learned.
  - For example, programmers who understand the concepts of object-oriented programming will have a much easier time learning Java

- The same phenomenon occurs in natural languages. The better you know the grammar of your native language, the easier it is to learn a second language.
  - Furthermore, learning a second language has the benefit of teaching you more about your first language.

## d) Better Understanding Of The Significance Of Implementation

- In learning the concepts of programming languages, it is both interesting and necessary to touch on the implementation issues that affect those concepts.

  For example, knowing how function calls and recursions are implemented will help a programmer write and use functions more intelligently.

  - This knowledge leads to the ability to use a language more intelligently, as it was designed to be used.

- Certain kinds of program bugs can be found and fixed only by a programmer who knows some related implementation details.

## e) Better Use Of Languages That Are Already Known

- Many current programming languages are large and complex.
- Accordingly, it is uncommon for a programmer to be familiar with and use all of the features of a language being used.
- By studying the concepts of programming languages, programmers can learn about previously unknown and unused parts of the languages they already use and begin to use those features.

# f) Overall advancement of computing

- If programmers (in general) had greater knowledge of programming language concepts, the software industry would do a better job of adopting languages based upon their merits rather than upon political and other forces.
- E.g., Algol 60 never made large inroads in the U.S., despite being superior to FORTRAN.
- E.g. Eiffel is not particularly popular, despite being a great language!

# 1.2 Programming Domains

- **Scientific applications**
  - Large numbers of floating-point computations; use of arrays
  - **Fortran (***Formula Translating)*
- **Business applications**
  - Produce reports, using decimal numbers and characters
  - **COBOL (Common Business Oriented Language)**
- **Artificial intelligence**
  - Symbols rather than numbers manipulated; use of linked lists
  - **LISP (List Processing)**
  - **PROLOG (logic Programming)**

# Programming Domains (cont.)

- **Systems programming**
  - Requires efficiency because of continuous use
  - **C**

- **Web Software**
  - Diverse collection of languages:
    - markup (e.g., **HTML**),
    - scripting (e.g., **PHP**),
    - general-purpose (e.g., **Java**)

This list is from the textbook, but it is neither exhaustive nor definitive.

# Other Programming Domains

- **Embedded Systems**
  - Software embedded on a closed system for a single purpose
    - For example, Automobiles, FitBit, refrigerators
  - **Java**, **Python**, **C**, **MIPS, ARM, X86**

- **3-D Graphics and Animation**
  - Render very hi-res animation for cinema and television
  - **Java**, **C**

- **Gaming**
  - Render 3-D graphics in real time in response to user input
  - **C++**, **C#**

# 1.3 Language Evaluation Criteria

a) **Readability:** the ease with which programs can be read and understood

b) **Writability:** the ease with which a language can be used to create programs

c) **Reliability:** conformance to specifications (i.e., performs to its specifications)

d) **Cost:** the ultimate total cost

# a) Evaluation Criteria: Readability

- The most important criterion for judging a programming language is the readability and understandability of the program.
- Language constructs initially were designed more from the computer's perspective than from the user's perspective.
- Since the ease of maintenance largely depends on the readability of a program, readability has become an important criterion for measuring the quality of programs and programming languages.

  →

  **The result is a shift from a machine-oriented focus to a human-oriented focus.**

# Readability: Overall Simplicity

- Too many features make the language difficult to learn. Programmers tend to learn a subset of the language and ignore its other features.

- Diversity in a feature makes the feature complex → having multiple ways of accomplishing a particular operation

  For example, in Java:

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

  Although the last two statements have slightly different meanings in some contexts, all four have the same meaning when used as stand-alone expressions.

# Readability: Overall Simplicity (cont.)

- Most assembly language statements are models of simplicity.
- However, it is this simplicity that makes assembly language programs less readable. **Because they lack more complex control statements.**
- What is needed:
  - ✓ **A manageable set of features and constructs**
  - ✓ **Minimal feature multiplicity**
  - ✓ **Minimal operator overloading**

Example of Operator Overloading:

int = int + int

float = float + float

struct = struct + struct

array = array + array

int = array + array

# Evaluation Criteria: Readability (cont.)

- **Orthogonality** in a programming language: **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**.
  - Orthogonality makes the language easy to learn and read.

  For example,

  In C, you can use operators (+, -, *, /) with various data types (int, float, double) interchangeably

  → don't need to have integer +, FP +, …

  → An orthogonal design!

# Orthogonality Example:

Adding two 32-bit integer values that reside in either memory or registers and replacing one of two values with the sum.

- The IBM mainframes have two instructions:

```
A     Reg1, memory_cell      // Reg1 ← (Reg1) + (memory_cell)
AR    Reg1, Reg2             // Reg1 ← (Reg1) + (Reg2)
```

- The VAX addition instruction for 32-bit integer value is:

```
ADDL operand_1, operand_2    //operand_2 ← (operand_1) + (operand_2)
```

In this case, either operand can be a register or a memory cell.

The VAX instruction design is orthogonal in that a single instruction can use either registers or memory cells as the operands.

The IBM design is not orthogonal.

# Evaluation Criteria: Readability (cont.)

**Data types**

- The presence of adequate facilities for defining data types and data structures in a language is another significant aid to readability.

**Example**: suppose a numeric type is used for an indicator flag because there is no Boolean type in the language.

In such a language, we might have an assignment such as

timeout = 1

Whose meaning is unclear,

whereas in a language that includes Boolean types we would have

timeout = true

# Evaluation Criteria: Readability (cont.)

## Syntax Considerations

- The syntax of the elements of a language has a significant effect on readability.
- The following are examples of syntactic design choices that affect readability:

- **Identifier forms: Limiting identifiers to very short lengths reduces readability.**
  - Example: In Fortran 77, identifiers can only contain up to six characters.
  - Example: In ANSI BASIC (1978), an identifier may consist only of a single letter, or a single letter followed by a single digit.

- **Special Words:** The appearance of a program and the readability of a program are greatly affected by language special words (while, class, for,...).
  - **Example**: C or some other languages use braces or **end** for pairing control structures. It is difficult to determine which group is being ended.
  - **Example**: Ada uses **end if** to terminate a selection construct, and **end loop** to terminate a loop construct.
  - **Example: Fortran 95 and Ada allow programmers to use special words (e.g., DO, END) as legal variable names.**

# b) Evaluation Criteria: Writability

- It measures how easy it is for a language to create programs for a selected problem area.

- Most of the language characteristics that affect readability also affect writability.

- **Simplicity and orthogonality**
  - A smaller number of primitive constructs and a consistent set of rules for combining them (that is, orthogonality) is much better than simply having a large number of primitives.

  - **A programmer can design a solution to a complex problem after learning only a simple set of primitive constructs.**

# Evaluation Criteria: Writability (cont.)

- **Support for abstraction**
  - Abstraction means the ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored.
  - Programming languages can support two distinct categories of abstraction, **process** and **data**.
  - A simple example of process abstraction is the use of subprogram to implement a sort algorithm that is required several times in a program. Without the subprogram, the sort code would have to be replicated in all places where it was needed.
  - As an example of data abstraction, consider a binary tree that stores integer data in its nodes. In Fortran 77, three parallel integer arrays, where two integers are used as subscripts to specify descendant nodes.
  - In C++ and Java, theses trees can be implemented by using an abstraction of a tree node in the form of a simple class with two pointers (or references) and an integer.

# c) Evaluation Criteria: Reliability

**A program is said to be reliable if it performs to its specifications under all conditions.**

- **Type checking**: simply testing for type errors in a given program, either by the compiler or during program execution.
  - ✓ The sooner an error is detected, the less expensive it is to make the repairs needed.
  - ✓ **Java requires type checking of nearly all variables and expressions at compile time.**

- **Exception handling**: The ability to intercept runtime errors, take corrective action, and then continue goes a long way toward reliability

- **Aliasing**: Two or more different reference methods or names for the same storage unit. In C language, variables and pointers may refer to the same memory location.

  **- It is now widely accepted that aliasing is a dangerous feature in a language.**

- Readability and writability: both influence reliability.

# Language Evaluation Criteria

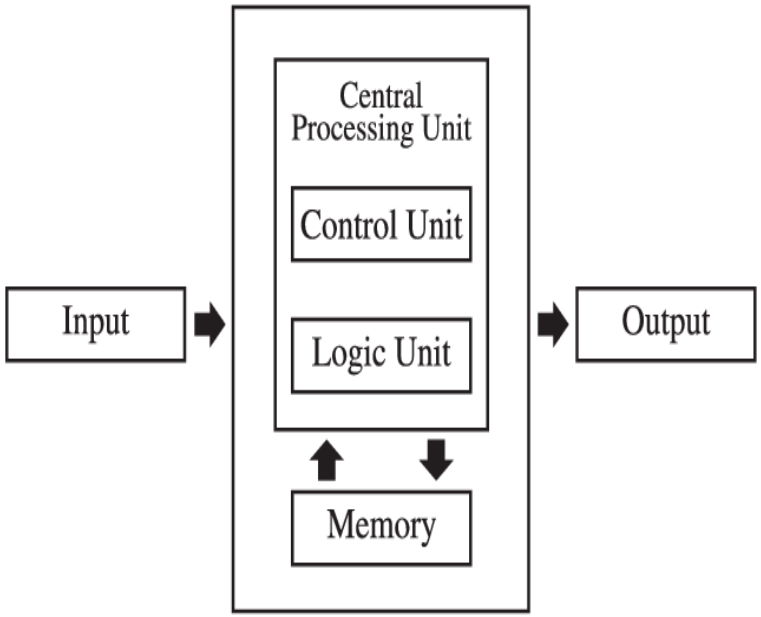
| | CRITERIA | | |
|---|---|---|---|
| Characteristic | READABILITY | WRITABILITY | RELIABILITY |
| Simplicity | • | • | • |
| Orthogonality | • | • | • |
| Data types | • | • | • |
| Syntax design | • | • | • |
| Support for abstraction | | • | • |
| Expressivity | | • | • |
| Type checking | | | • |
| Exception handling | | | • |
| Restricted aliasing | | | • |

# Evaluation Criteria: Cost

- Training programmers to use language
- Writing programs
- Compiling programs
- Executing programs
- Language implementation system "Free compilers is the key, success of Java"
- Reliability, does the software fail?
- Maintaining programs: Maintenance costs can be as high as two to four times as much as development costs.
- Portability "standardization of the language"
- Generality (the applicability to a wide range of applications)
- Well-definedness (The completeness and precision of the language's official definition)

# 1.4 Influences on Language Design

- **Computer Architecture**
  - Languages are developed around the dominant computer architecture, known as the **von Neumann architecture**



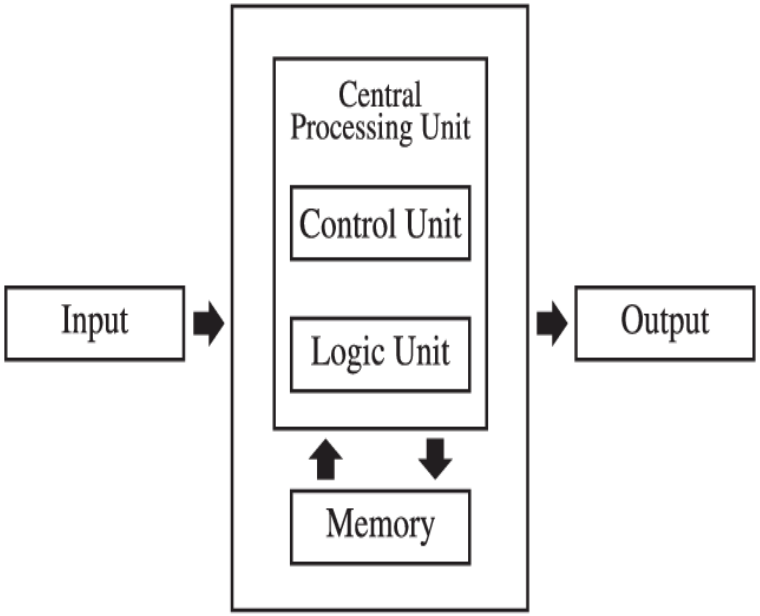

- **Programming Design Methodologies**
  - New software development methodologies (e.g., object-oriented software development) led to new programming paradigms and by extension, new programming languages

# von Neumann Architecture

- Is based on a stored-program concept introduced by John Von Neumann.
- In this stored-program concept, programs and data are stored in a separate storage unit called memories and are treated the same.
- This novel idea meant that a computer built with this architecture would be much easier to **reprogram.**

**John von Neumann** (1903-1957) is best known as a mathematician and one of the leading figures in early 20th century mathematics. During World War II, he played an important role in atomic research. He worked on computational problems for the U.S. military during and after World War II, most notably on the ENIAC computer at Penn's Moore School of Electrical Engineering.

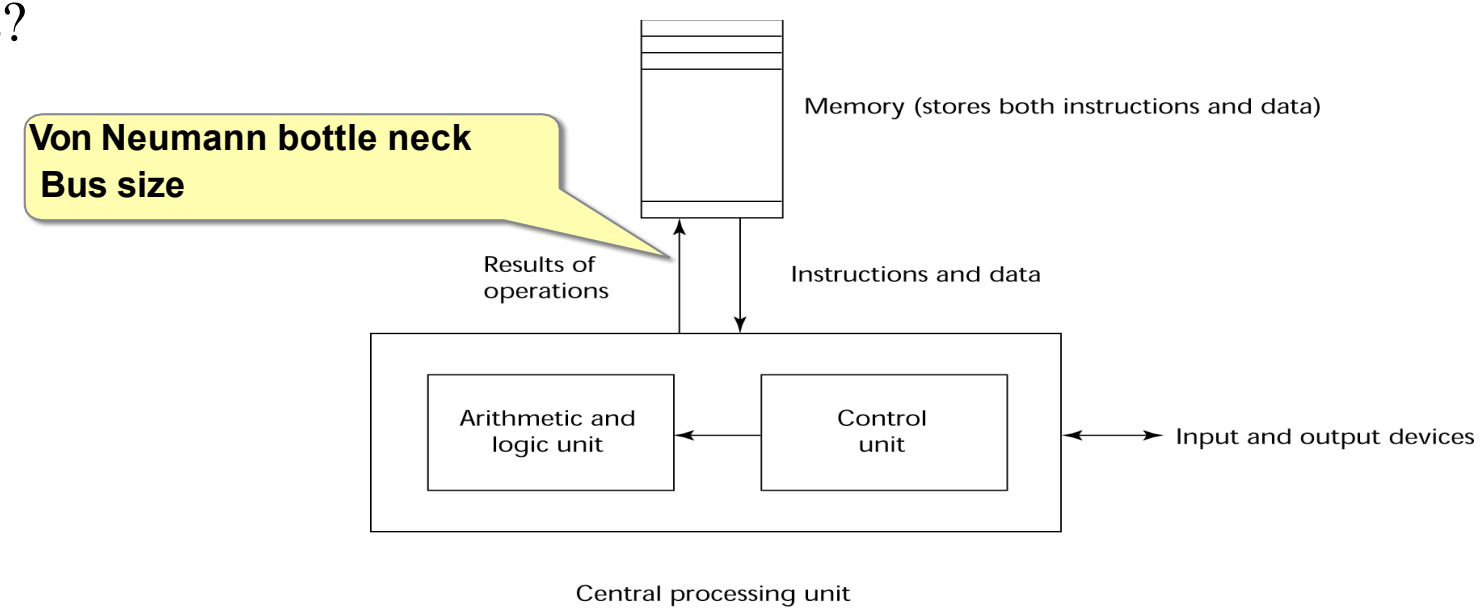





The von Neumann Architecture

# Von Neumann Architectural Bottleneck

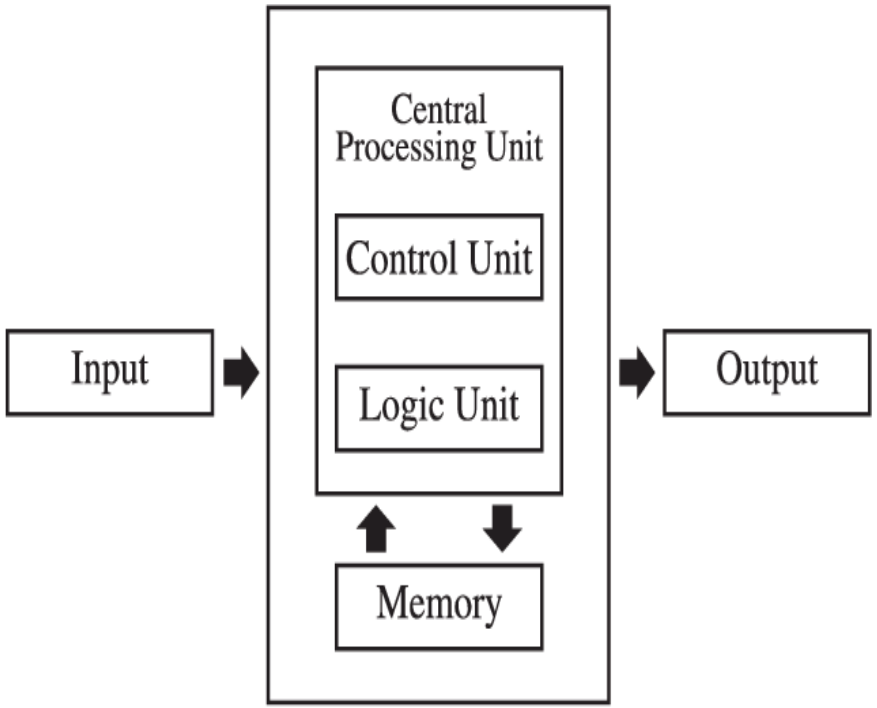
- Connection speed between a computer's memory and its processor determines the speed of a computer
- Program instructions often can be executed much faster than the speed of the connection; the connection speed thus results in a bottleneck
- Known as the von Neumann bottleneck; it is the primary limiting factor in the speed of computers
  - Workarounds?



# Computer Architecture Influence

- Well-known computer architecture: **Von Neumann**
- **Imperative languages**, most dominant, because of von Neumann computers
  - Data and programs stored in memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - Basis for imperative languages
    - Variables model memory cells
    - Assignment statements model piping
    - Iteration is efficient

- **Imperative programming**: Programming with an explicit sequence of commands that update state.
- **Procedural programming:** a subset of imperative programming which utilizes subroutines.




The von Neumann Architecture

# Programming Methodologies Influences

- **1950s and early 1960s:** Simple applications
  - worry about machine efficiency
  - GOTO
- **Late 1960s:** People efficiency became important; readability, better control structures
  - structured programming
  - top-down design and stepwise refinement
- **Late 1970s:** A shift from procedure-oriented to data-oriented
  - data abstraction
- **Middle 1980s:** Object-oriented programming
  - Data abstraction + inheritance + polymorphism

# 1.5 Language Categories

- **Imperative Programming Languages**
  - Central features are variables, assignment statements, and iteration
  - Include languages that support object-oriented programming (OOP)
  - Includes scripting languages
  - Includes the visual languages
  - Language examples: **C**, **C++**, **C#**, **Objective-C**, **Java**, **Perl**, **Python**, **Ruby**, **JavaScript**, **Visual BASIC .NET**

# Language Categories (cont.)

- **Functional Programming Language**
  - The main way to do computation is to apply a function to the given parameters
  - Functions are "first-class citizens"
  - Lambda calculus
  - Language examples: **LISP**, **Haskell**, **Racket (PLT Scheme)**, **Clojure**, **Scala**, **ML**, **F#**

Example of LISP functions:

```
(defun factorial (n)
  (if (<= n 1)
     1
     (* n (factorial (- n 1)))))
```

Example of Lambda calculus:

(λx. x * 2)

Apply the lambda function to a specific argument, 3:

(λx. x * 2) 3

The evaluation is 6

# Language Categories (cont.)

- **Logic Programming Languages**
  - Rule-based language
  - Rules are specified in no particular order
  - The language implementation system must choose an order in which the rules are used
  - Language examples: **Prolog**, **Answer Set Programming (ASP)**, **Datalog**

Both functional language and logic language are subcategories of **declarative languages**

Language examples: SQL, HTML, XSLT, Prolog, …

# Prolog Example

**Example : Below student-professor relation table shows the facts, rules, goals and their english meanings.**

| **Facts** | **English meanings** |
|---|---|
| studies(charlie, csc135). | // charlie studies csc135 |
| studies(olivia, csc135). | // olivia studies csc135 |
| studies(jack, csc131). | // jack studies csc131 |
| studies(arthur, csc134). | // arthur studies csc134 |
| | |
| teaches(kirke, csc135). | // kirke teaches csc135 |
| teaches(collins, csc131). | // collins teaches csc131 |
| teaches(collins, csc171). | // collins teaches csc171 |
| teaches(juniper, csc134). | // juniper teaches csc134 |
| | |
| **Rules** | |
| professor(X, Y) :- teaches(X, C), studies(Y, C). | // X is a professor of Y if X teaches C and Y studies C. |
| | |
| **Queries / Goals** | |
| ?- studies(charlie, What). | // charlie studies what? OR What does charlie study? |
| ?- professor(kirke, Students). | // Who are the students of professor kirke. |

# Language Categories (cont.)

- **Markup/programming hybrid**
  - Markup languages extended to support some other programming
  - Examples: **JSTL**, **XML**, **XSLT**, **JSON**, **SQL**
  - Many are more accurately **labeled as Protocols**, not complete programming languages

# 1.6 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 (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

# 1.7 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

- 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

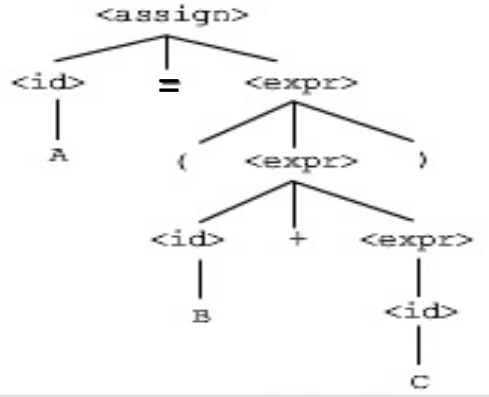
For example, this assignment:

```
a = b + c * d;
```

Has the intermediate code:

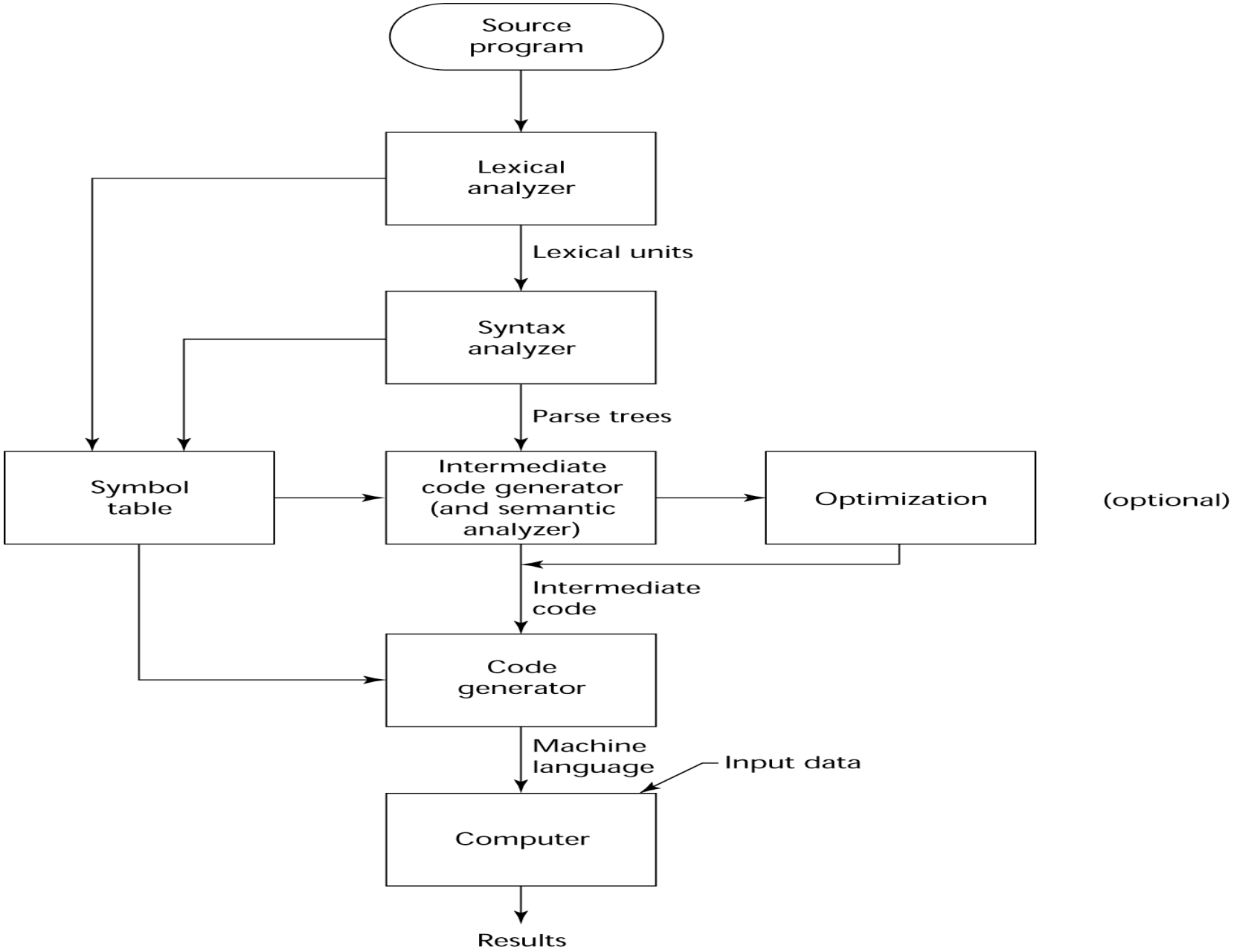
```
r1 = c * d;
r2 = b + r1;
a = r2
```

**Example**: Parse tree




**Parse Tree for the Statement A = ( B + C )**

# The Compilation Process

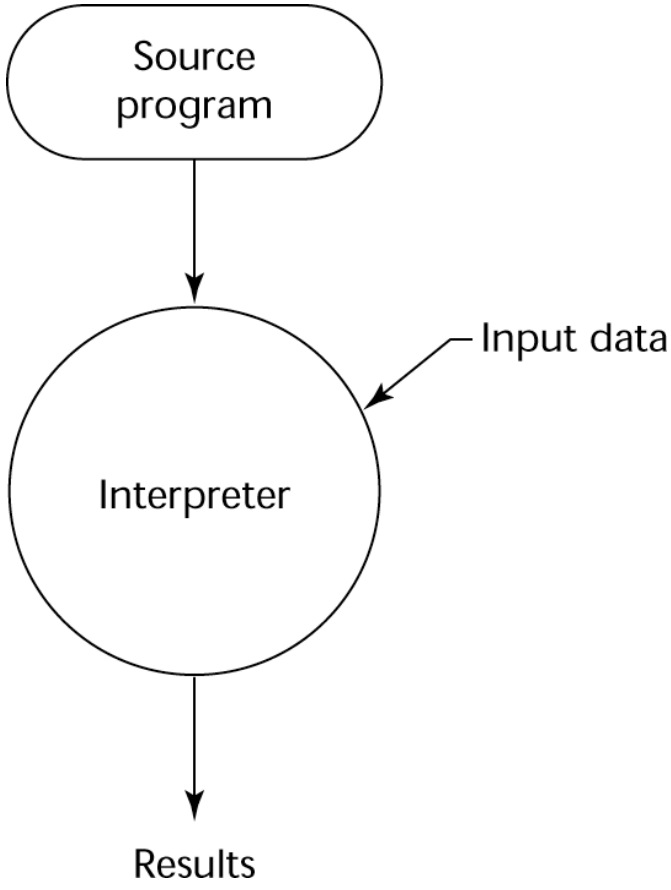


# Additional Compilation Terminologies

- Linking: the process of collecting system program units and linking them to a user program
  - `.o`
  - `.obj`
- Load module (executable image): both user code and system code are brought into memory
  - `.exe`

# Pure Interpretation

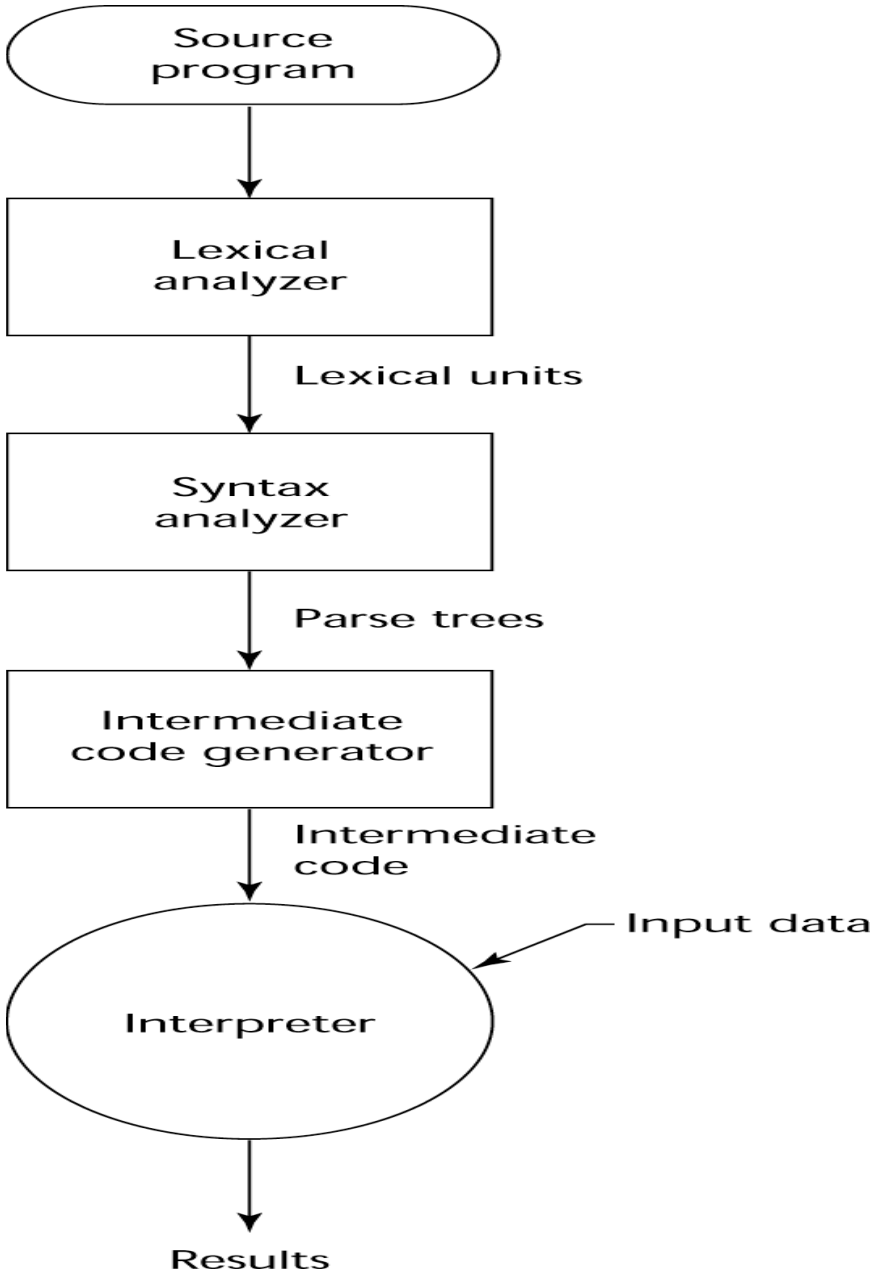
- No translation
- Easier implementation of programs (run-time errors can easily and immediately be displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
- Rarely seen in traditional high-level languages these days
- Significant return with some web scripting languages (e.g., JavaScript, PHP)



# 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
- Examples
  - Perl programs are partially compiled to detect errors before interpretation
  - Implementation of Java is hybrid; the intermediate form, **bytecode**, provides portability to any machine that has a bytecode interpreter and a run-time system — together, these are called Java Virtual Machine (JVM).

# Hybrid Implementation Process



# Just-in-Time Implementation Systems

- Initially translate programs to an intermediate language
- Then compile the subprograms (in intermediate code) into machine code when they are called
- JIT systems are widely used for Java programs
- .NET languages are implemented with a JIT system
- **In essence, JIT systems are delayed compilers**

# Preprocessors

- Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included

- A preprocessor processes a program immediately before the program is compiled to expand embedded preprocessor macros

- A well-known example: C preprocessor
  - expands #include, #define, and similar macros
  - E.g., **#include** "myLib.h“

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

# Summary

- The study of programming languages is valuable for several 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
- Major influences on language design have been machine architecture and software development methodologies
- The major methods of implementing programming languages: compilation, pure interpretation, and hybrid implementation

# Chapter 2: Evolution of the Major Programming Languages

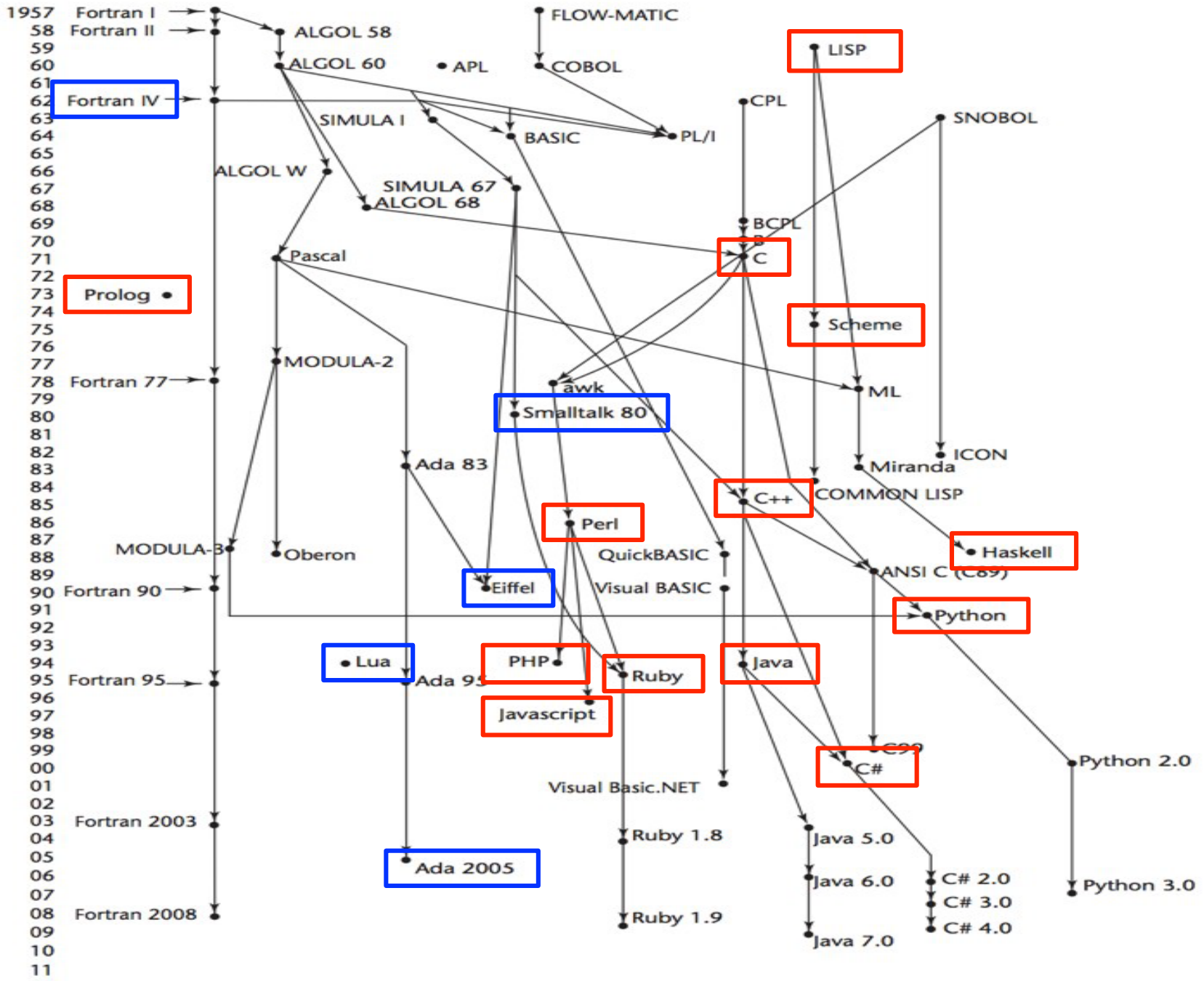
# Chapter 2 Topics

- 2.2 – Minimal Hardware Programming: Pseudocodes
- 2.3 – The IBM 704 and Fortran
- 2.4 – Functional Programming: LISP
- 2.5 – The First Step Toward Sophistication: ALGOL 60
- 2.6 – Computerizing Business Records: COBOL
- 2.7 – The Beginnings of Timesharing: BASIC
- 2.8 – Everything for Everybody: PL/I
- 2.9 – Two Early Dynamic Languages: APL and SNOBOL
- 2.10 – The Beginnings of Data Abstraction: SIMULA 67

# Chapter 2 Topics

- 2.11 – Orthogonal Design: ALGOL 68
- 2.12 – Some Early Descendants of the ALGOLs
- 2.13 – Programming Based on Logic: Prolog
- 2.14 – History's Largest Design Effort: Ada
- 2.15 – Object-Oriented Programming: Smalltalk
- 2.16 – Combining Imperative and Object-Oriented Features: C++
- 2.17 – An Imperative-Based Object-Oriented Language: Java
- 2.18 – Scripting Languages
- 2.19 – The Flagship .NET Language: C#
- 2.20 – Markup/Programming Hybrid Languages

# Genealogy of Common Languages



# Minimal Hardware Programming

The machines of early days were difficult to program because of the lack of supporting software. So programming was done in machine code, which is both tedious and error-prone

- What was wrong with using machine code?
  - Poor readability
  - Poor modifiability
  - Expression coding was tedious
  - Machine deficiencies--no indexing or floating point

**Assembly Code**

```
IMM   R0,   0x80
LOAD  R0,   R0
IMM   R1,   0x84
LOAD  R1,   R1
IMM   R2,   0x0
IMM   R3,   0x4
IMM   R4,   0x0
IMM   R5,   0x1
STORE R0,   R2
ADD   R0,   R0, R3
ADD   R4,   R4, R5
BNE   0x20, R4, R1
```

**Machine Code**

```
0x 60 00 00 80
0x A4 00 00 00
0x 60 01 00 84
0x A4 01 01 00
0x 60 02 00 00
0x 60 03 00 04
0x 60 04 00 00
0x 60 05 00 01
0x 08 00 00 02
0x 20 00 00 03
0x 20 04 04 05
0x 11 20 04 01
```

# Pseudocodes: Short Code

- Short Code developed by John Mauchly in 1949 for BINAC (**BINary Automatic Computer**) computers
  - Expressions were coded by byte-pair values
  - A word has 72 bits, grouped as 12 six-bits bytes
  - Many equations can be coded in a word
  - Example:

```
a = (b + c) / b * c is coded to:
X3 = ( X1 + Y1 ) / X1 * Y1  /substitute variables w/ X1,X3,Y1
X3 03 09 X1 07 Y1 02 04 X1 Y1 /substitute operators and
                              parentheses, =(03), ((09),
                              +(07), )(02), /(04); Note: no
                              multiplication code
```

→ `0000X30309X107Y10204X1Y1 /a word (12 bytes/72 bits)`

# Pseudocodes: Speedcoding

- Speedcoding developed by Backus in 1954 for IBM 701
  - Pseudo ops for arithmetic and math functions, including floating-point ops
  - Conditional and unconditional branching
  - Auto-increment registers for array access
  - Slow!
  - Only 700 words left for user program

# IBM 704 and Fortran

- **Fortran 0**: 1954 - not implemented
- **Fortran I**: 1957
  - Designed for the new IBM 704, which had index registers and floating-point hardware
  - This led to the idea of compiled programming languages, since there was nowhere to hide the cost of interpretation (especially for floating-point operations)
  - Environment of development
    - Computers were small and unreliable
    - Applications were scientific
    - No programming methodology or tools
    - Machine efficiency was the most important concern

# Fortran I Overview

- First implemented version of Fortran
  - Names could have up to six characters
  - Post-test counting loop (DO)
  - Formatted I/O: Iw, Fw.d, A, etc.
  - User-defined subprograms
  - Three-way selection statement (arithmetic **IF**)

    If (expression) S1, S2, S3

    S1 if expression < 0

    S2 if expression == 0

    S3 if expression > 0
  - No data typing statements

**Example of a Fortran I Program:**

```
      PROGRAM FATTORIALE

      WRITE(6,1000)
1000  FORMAT(' DAMMI UN NUMERO ... PER FAVORE PICCOLO ...')
      READ(5,*) N

      A=1.0
      DO 10 I=1,N
      A=A*I
   10 CONTINUE

      WRITE(6,3000) N,A
3000  FORMAT(' IL FATTORIALE DI: ',I5,' RISULTA: ',F15.0)

      STOP
      END
```

The first letter of the name determines the data type implicitly:

1st letter of the name is I, J, K, L, M, or N → integer; otherwise, floating-point

# Fortran I Overview (continued)

- First implemented version of FORTRAN
  - No separate compilation
  - Compiler released in April 1957, after 18 worker-years of effort
  - Programs larger than 400 lines rarely compiled correctly, mainly due to poor reliability of 704
  - Code was very fast
  - Quickly became widely used

# Fortran II

- Distributed in 1958
  - Independent compilation
  - Fixed the bugs

# Fortran IV

- Evolved during 1960-62
  - Explicit type declarations
  - Logical selection statement
  - Subprogram names could be parameters
  - ANSI standard in 1966

# Fortran 77

- Became the new standard in 1978 (ANSI, 1978a)
  - Character string handling
  - Logical loop control statement
  - **IF-THEN-ELSE**statement

# Fortran 90

- Most significant changes from Fortran 77
  - Modules
  - Dynamic arrays
  - Pointers
  - Recursion
  - CASE statement
  - Parameter type checking

# Latest versions of Fortran

- Fortran 95 (INCITS/ISO/IEC, 1997) – relatively minor additions, plus some deletions
- Fortran 2003 – support for OOP, procedure pointers, interoperability with C
- Fortran 2008 (ISO/IEC 1539-1, 2010) – blocks for local scopes, co-arrays, Do Concurrent
- Fortran 2018 (ISO/IEC 1539-1:2018) – further interoperability with C, additional parallel features, conformance the new version of the IEEE floating-point standard, etc.
- Fortran 2023 (ISO/IEC 1539-1:2023) was published in November 2023

# Functional Programming: LISP

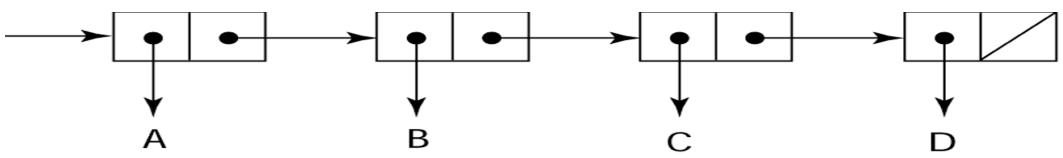
- **LIS**t **P**rocessing language
  - Designed at MIT by John McCarthy
- AI research needed a language to
  - Process data in lists (rather than arrays)

    **Example:** `(A (B C) D (E (F G)))`
  - Symbolic computation (rather than numeric)
    - Only two data types: atoms and lists
    - Syntax is based on lambda calculus

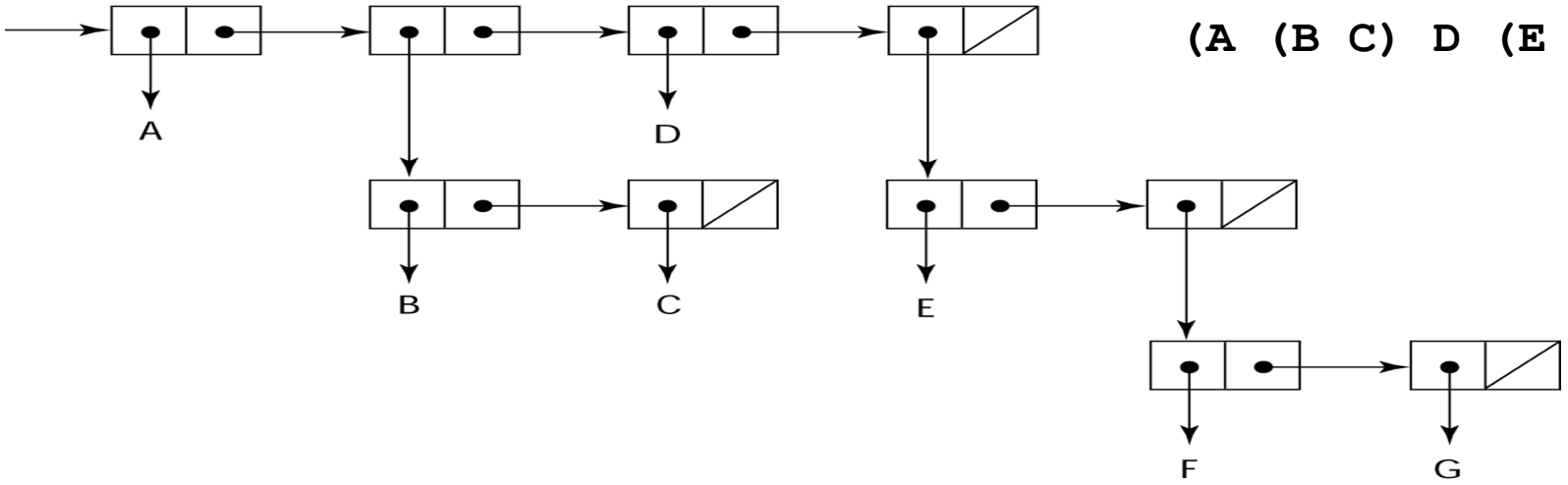
**Example: Factorial Function in LSIP**

```
(defun factorial (n)
                 (cond ((= n 0) 1)
                       (t (* n (factorial (- n 1))))
                  )
)
```

# Representation of Two LISP Lists




**(A B C D)**




**(A (B C) D (E (F G)))**

# LISP Evaluation

- Pioneered functional programming
  - No need for variables or assignment
  - Control via recursion and conditional expressions
- Still the dominant language for AI
- COMMON LISP and Scheme are contemporary dialects of LISP
- ML, Haskell, and F# are also functional programming languages, but use very different syntax

**Garbage collection** was invented by John McCarthy around 1959 to simplify manual memory management in Lisp

# Scheme

It is a dialect of Lisp with a strong focus on functional programming concepts, but it also supports procedural programming and imperative-style programming when needed.

- Developed at MIT in mid 1970s
- Small
- Extensive use of static scoping
- Functions as first-class entities
- Simple syntax (and small size) make it ideal for educational applications

**Example: Functions as first-class entities**

```
(define (doit3 proc)
            (proc 3)
        )

(doit3 sqr)    ; evaluates to 9
(doit3 add1)   ; evaluates to 4
(doit3 sqrt)   ; evaluates to 1.732050807568
```

a function as an argument

# COMMON LISP

- An effort to combine features of several dialects of LISP into a single language
- Large, complex, used in industry for some large applications

# Scala

- **Sca***lable* **La***nguage*
- Full functional support
- Very strong type system (all types inherits from class Any)
- Runs in JVM
- Introduced in 2004

```
def factorial(x: BigInt): BigInt =
  if (x == 0) 1 else x * factorial(x - 1)
```

# The First Step Toward Sophistication: ALGOL 60

- Environment of development
  - FORTRAN had (barely) arrived for IBM 70x
  - Many other languages were being developed, all for specific machines
  - No portable language; all were machine-dependent
  - No universal language for communicating algorithms
- ALGOL 60 was the result of efforts to design a universal language

# ALGOL 58

- Concept of *type* was formalized
- Names could be any length
- Arrays could have any number of subscripts
- Subscripts were placed in brackets
- Compound statements (begin … end)
- Semicolon as a statement separator
- Assignment operator was :=
- If had an else-if clause
- No I/O → make it machine independent

# ALGOL 60 Overview

- Modified ALGOL 58 at 6-day meeting in Paris
- New features
  - Block structure (local scope)
  - Two parameter passing methods
  - Subprogram recursion
  - Stack-dynamic arrays
  - Still no I/O and no string handling

```
begin
comment classic recursive procedure;
        integer nn, nf;

        integer procedure factorial(n); value n; integer n;
        begin
                if n = 1 then factorial := 1
                else factorial := n * factorial(n-1)
        end;

        nn := 5;
        nf := factorial(nn);
        outinteger ( 1 , nf)

end
```

# ALGOL 60 Evaluation

- Successes
  - It was the standard way to publish algorithms for over 20 years
  - All subsequent imperative languages are based on it
  - First machine-independent language
  - First language whose syntax was formally defined (BNF)
- Failure
  - Never widely used, especially in U.S.
  - Reasons
    - ✓ Lack of I/O and the standardized character set made programs non-portable
    - ✓ Too flexible → hard to implement
    - ✓ Entrenchment of Fortran
    - ✓ Formal syntax description
    - ✓ Lack of support from IBM

# Computerizing Business Records: COBOL

- Environment of development
  - UNIVAC was beginning to use FLOW-MATIC
  - USAF was beginning to use AIMACO
  - IBM was developing COMTRAN

**FLOW-MATIC (1957)**, originally known as B-0 (Business Language version 0) was the first English-like data processing language.

# COBOL Historical Background

- Based on FLOW-MATIC
- FLOW-MATIC features
  - Names up to 12 characters, with embedded hyphens
  - English names for arithmetic operators (no arithmetic expressions)
  - Data and code were completely separate
  - The first word in every statement was a verb

# COBOL Design Process

- First Design Meeting (Pentagon) - May 1959
- Design goals
  - Must look like simple English
  - Must be easy to use, even if that means it will be less powerful
  - Must broaden the base of computer users
  - Must not be biased by current compiler problems
- Design committee members were all from computer manufacturers and DoD branches
- Design Problems: arithmetic expressions? subscripts? Fights among manufacturers

# COBOL Evaluation

- Contributions
  - First macro facility in a high-level language
  - Hierarchical data structures (records)
  - Nested selection statements
  - Long names (up to 30 characters), with hyphens
  - Separate data division

# COBOL: DoD Influence

- First language mandated by DoD
  - would have failed without DoD
- Still the most widely used business applications language

# COBOL Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FACTORIAL.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 N PIC 9(4).
77 A PIC S9(4) VALUE 0.
77 F PIC 9(4) VALUE 1.
PROCEDURE DIVISION.
PARA.
    DISPLAY "ENTER A NUMBER.".
    ACCEPT N.
    PERFORM PARA1 UNTIL A = N.
    DISPLAY "THE FACTORIAL IS".
    DISPLAY F.
    STOP RUN.
PARA1.
    ADD 1 TO A.
    COMPUTE F = F * A.
```

# The Beginning of Timesharing: BASIC

- Designed by Kemeny & Kurtz at Dartmouth
- Design Goals:
  - Easy to learn and use for non-science students
  - Must be “pleasant and friendly”
  - Fast turnaround for homework
  - Free and private access
  - User time is more important than computer time
- Current popular dialect: Visual BASIC
- First widely used language with time-sharing

An example of a **BASIC Program:**

```
10 REM Temperature conversion program
20 PRINT ”Temperature in Fahrenheit”;
30 INPUT F
40 LET C=(F-32)*5/9
50 PRINT F;” F equals ”;
60 PRINT C;” C”
70 END
```

The output is:

```
Temperature in Fahrenheit? 98
98 F equals 36.66667 C
```

**BASIC: B**eginner’s **A**ll-purpose **S**ymbolic **I**nstruction **C**ode

# Everything for Everybody: PL/I

- Designed by IBM and SHARE
- Computing situation in 1964 (IBM's point of view)
  - Scientific computing
    - IBM 1620 and 7090 computers
    - FORTRAN
    - SHARE user group
  - Business computing
    - IBM 1401, 7080 computers
    - COBOL
    - GUIDE user group

# PL/I: Background

- By 1963
  - Scientific users began to need more elaborate I/O like COBOL had; business users began to need floating point and arrays for MIS
  - It looked like many shops would begin to need two kinds of computers, languages, and support staff → too costly
- The obvious solution
  - Build a new computer to do both kinds of applications
  - Design a new language to do both kinds of applications

# PL/I: Design Process

- Designed in five months by the 3 X 3 Committee
  - Three members from IBM, three members from SHARE
- Initial concept
  - An extension of Fortran IV
- Initially called NPL (New Programming Language)
- Name changed to PL/I in 1965

# PL/I: Evaluation

- **PL/I** contributions
  - First unit-level concurrency
  - First exception handling
  - Switch-selectable recursion
  - First pointer data type
  - First array cross sections
- Concerns
  - Many new features were poorly designed
  - Too large and too complex

**array cross sections**: selecting a subset of elements from an array based on certain criteria or conditions.

```
/* Read in a line, which contains a string,
/* and then print every subsequent line that contains that string. */

find_strings: procedure options (main);
   declare pattern character (100) varying;
   declare line    character (100) varying;
   declare (line_no, end_file) fixed binary;

   end_file = 0;
   on endfile (sysin) end_file = 1;

   get edit (pattern) (L);
   line_no = 1;
   do while (end_file = 0);
      if index(line, pattern) > 0 then
         put skip list (line_no, line);
      line_no = line_no + 1;
      get edit (line) (L);
   end;
```

# Two Early Dynamic Languages: APL and SNOBOL

- Characterized by dynamic typing and dynamic storage allocation
- Variables are untyped
  - A variable acquires a type when it is assigned a value
- Storage is allocated to a variable when it is assigned a value

# APL: A Programming Language

- Designed as a hardware description language at IBM by Ken Iverson around 1960
  - Highly expressive (many operators, for both scalars and arrays of various dimensions)
  - Programs are very difficult to read
- Still in use; minimal changes

# SNOBOL (String-Oriented Symbolic Language)

- Designed as a string manipulation language at Bell Labs by Farber, Griswold, and Polensky in 1964
- Powerful operators for string pattern matching
- Slower than alternative languages (and thus no longer used for writing editors)
- Still used for certain text processing tasks

# The Beginning of Data Abstraction: SIMULA 67

- Designed primarily for system simulation in Norway by Nygaard and Dahl
- Based on ALGOL 60 and SIMULA I
- Primary Contributions
  - Coroutines - a kind of subprogram
  - Classes, objects, and inheritance

# Orthogonal Design: ALGOL 68

- From the continued development of ALGOL 60 but not a superset of that language
- Source of several new ideas (even though the language itself never achieved widespread use)
- Design is based on the concept of orthogonality
  - A few basic concepts, plus a few combining mechanisms

# ALGOL 68 Evaluation

- Contributions
  - User-defined data structures
  - Reference types
  - Dynamic arrays (called flex arrays)
- Comments
  - Less usage than ALGOL 60
  - Had strong influence on subsequent languages, especially Pascal, C, and Ada

# Pascal (1971)

- Developed by Wirth (a former member of the ALGOL 68 committee)
- Designed for teaching structured programming
- Small, simple, nothing really new
- Largest impact was on teaching programming
  - From mid-1970s until the late 1990s, it was the most widely used language for teaching programming

# C (1972)

- Designed for systems programming
  - at Bell Labs by Dennis Richie
- Evolved primarily from **BCLP** and **B**, but also **ALGOL 68**
- Powerful set of operators, but poor type checking
- Initially spread through UNIX
- Though designed as a systems language, it has been used in many application areas

# Programming Based on Logic: Prolog

- Developed, by Comerauer and Roussel (University of Aix-Marseille), with help from Kowalski ( University of Edinburgh) in 1972.
- Based on formal logic
- Non-procedural
- Can be summarized as being an intelligent database system that uses an inferencing process to infer the truth of given queries
- Comparatively inefficient
- Few application areas

# World First Programmer

- **Augusta Ada King**, **Countess of Lovelace** (née **Byron**; 10 December 1815 – 27 November 1852)
- An English mathematician and writer, chiefly known for her work on **Charles Babbage**'s proposed mechanical general-purpose computer, the **Analytical Engine**.
- The last of Lovelace's notes, **Note G**, was designed to calculate Bernoulli numbers using Analytical Engine → the first algorithm specifically for a computer.



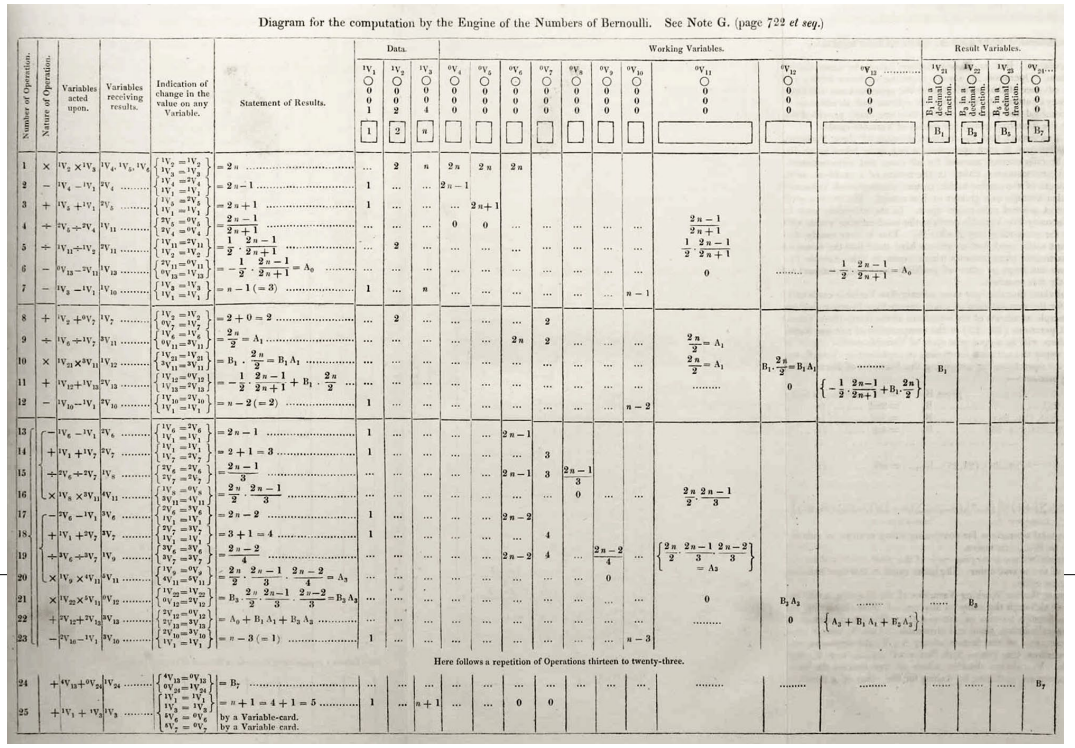


Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 *et seq.*)

# History's Largest Design Effort: Ada

- Designed for embedded and real-time systems, primarily for the U.S. Department of Defense.
- Huge design effort, involving hundreds of people, much money, and about eight years
- Sequence of requirements (1975-1978)
- Named **Ada** after Augusta Ada Byron
  - Widely recognized as the first programmer.

# Ada Evaluation

- Contributions
  - Packages - support for data abstraction
  - Exception handling
  - Generic program units
  - Concurrency - through the tasking model
- Comments
  - Competitive design
  - Included all that was then known about software engineering and language design

# Ada 95

- Ada 95 (began in 1988)
  - Support for OOP through type derivation
  - Better control mechanisms for shared data
  - New concurrency features
  - More flexible libraries
- Ada 2005
  - Interfaces and synchronizing interfaces
- Popularity suffered because the DoD no longer requires its use but also because of popularity of C++

# Object-Oriented Programming: Smalltalk

- Developed at Xerox PARC, initially by Alan Kay, later by Adele Goldberg
- ***First full implementation of an object-oriented language (data abstraction, inheritance, and dynamic binding)***
- Pioneered the graphical user interface design
- Promoted OOP

# C++

- Combining Imperative and Object-Oriented Programming
- Developed at Bell Labs by Stroustrup in 1980
- Evolved from **C** and **SIMULA 67**
- Facilities for object-oriented programming, taken partially from SIMULA 67
- A large and complex language, in part because it supports both procedural and OO programming
- Rapidly grew in popularity, along with OOP
- ANSI standard approved in November 1997

# Related OOP Languages

- **Objective-C** (designed by Brad Cox – early 1980s)
  - C plus support for OOP based on **Smalltalk**
  - Uses Smalltalk's method calling syntax
  - Used by Apple for systems programs
- **Delphi** (Borland)
  - Pascal plus features to support OOP
  - More elegant and safer than C++
- **Go** (designed at Google - 2009)
  - Loosely based on **C**, but also quite different

# An Imperative-Based Object-Oriented Language: Java

- Developed at Sun in the early 1990s
  - C and C++ were not satisfactory for embedded electronic devices
- Influenced by C++, but
  - Significantly simplified: intentionally omitted certain features found in C++, such as explicit pointer arithmetic, explicit memory management through pointers, and certain type coercions.
  - Has references, but not pointers
  - Memory Management: automatic memory management through garbage collection
  - Includes support for applets and a form of concurrency
  - Platform independence

# Java Evaluation

- Eliminated many unsafe features of C++
- Supports concurrency
- Libraries for applets, GUIs, database access
- Portable: Java Virtual Machine concept, JIT compilers
- Widely used for Web programming
- Use increased faster than any previous language
- Most recent version, Java SE 24, released in March 2025

# Scripting Languages for the Web

- **Perl**
  - Designed by Larry Wall—first released in 1987
  - Variables are statically typed but implicitly declared
  - Three distinctive namespaces, denoted by the first character of a variable's name
  - Powerful, but somewhat dangerous
  - Gained widespread use for CGI programming on the Web
  - Also used for a replacement for UNIX system administration language

# Scripting Languages for the Web (cont.)

- **JavaScript**
  - Began at Netscape, but later became a joint venture of Netscape and Sun Microsystems
  - A client-side **HTML**-embedded scripting language, often used to create dynamic **HTML** documents
  - Purely interpreted
  - Related to Java only through similar syntax

# Scripting Languages for the Web (cont.)

- **PHP**
  - "PHP: Hypertext Preprocessor" (a recursive acronym) is a widely-used open source-based general-purpose scripting language that is especially suited for web development and can be embedded into HTML.
  - A server-side HTML-embedded scripting language, often used for form processing and database access through the Web
  - Purely interpreted

# Scripting Languages for the Web (cont.)

- **Python**
  - An OO interpreted scripting language
  - Type checked but dynamically typed
  - Used for CGI (Common Gateway Interface) programming and form processing
  - Supports lists, tuples, and hashes

# Scripting Languages for the Web (cont.)

- **Ruby**
  - Designed in Japan by Yukihiro Matsumoto (a.k.a, “Matz”)
  - Began as a replacement for Perl and Python
  - A pure object-oriented scripting language
    - All data are objects
  - Most operators are implemented as methods, which can be redefined by user code
  - Purely interpreted

# The Flagship .NET Language: C#

- Part of the .NET development platform (2000)
- Based on C++ , Java, and Delphi
- Includes pointers, delegates, properties, enumeration types, a limited kind of dynamic typing, and anonymous types
- Is evolving rapidly

# Markup/Programming Hybrid Languages

- XSLT
  - eXtensible Markup Language (XML): a metamarkup language
  - eXtensible Stylesheet Language Transformation (XSTL) transforms XML documents for display
  - Programming constructs (e.g., looping)
- JSP
  - Java Server Pages: a collection of technologies to support dynamic Web documents
  - JSTL, a JSP library, includes programming constructs in the form of HTML elements

# Summary

- Development, development environment, and evaluation of a number of important programming languages
- Perspective into current issues in language design

# Chapters 3 and 4

Syntax and Semantics

# Topics

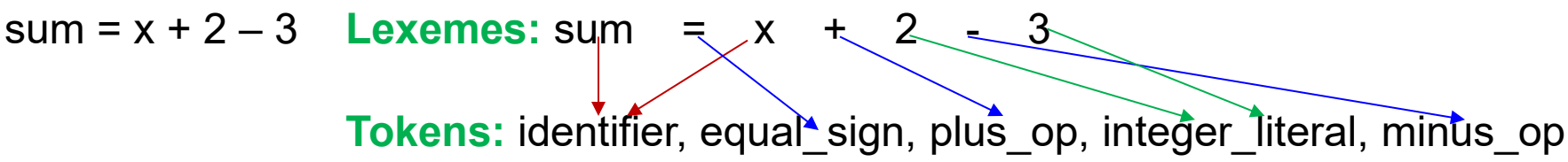
- Introduction
- The General Problem of Describing Syntax
- Formal Methods of Describing Syntax
- Attribute Grammars
- Describing the Meanings of Programs: Dynamic Semantics

# Introduction

- **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
    - Language designers
    - Implementers
    - Programmers (the users of the language)

# Describing Syntax: Terminology

- A **sentence** is a string of characters over some alphabet
- A **language** is a set of sentences
- A **lexeme** is the lowest level syntactic unit of a language
  - Programming e.g., *, total, sum, myVar, if, 24.678
  - Natural Language words and punctuation e.g., `cat, ?, swim`
- A **token** is a category of lexemes (e.g., "identifier")
  - Programming e.g., `identifier, int_literal, ;(semicolon)`
  - Natural Language part-of-speech, role e.g., `noun, verb, quotation mark`




**A token is defined by a pattern.**
**A lexeme is identified as an instance of a particular token if the scanned characters match the pattern of that token.**

# Common Tokens

- **identifiers**: names the programmer chooses
- **keywords**: names already in the programming language
- **separators** (also known as punctuators): punctuation characters
- **operators**: symbols that operate on arguments and produce results



- **literals**: numeric, logical, textual

# Example: Lexemes and Tokens

```
index = 2 * count + 17;
```

| Lexemes | Tokens |
|---|---|
| index | identifier |
| = | equal_sign |
| 2 | int_literal |
| * | mult_op |
| count | identifier |
| + | plus_op |
| 17 | int_literal |
| ; | semicolon |

# 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
  - Used in code generation, text generation, test case generation, etc.

# Context-Free Grammars and BNF

- Context-Free Grammars
  - Developed by Noam Chomsky in the mid-1950s
  - Language generators, meant to describe the syntax of natural languages
  - Define a class of languages called context-free languages

- Backus-Naur Form (1959)
  - Invented by John Backus to describe the syntax of Algol 58
  - BNF is equivalent to context-free grammars

# BNF Fundamentals

- In BNF, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (also called *nonterminal symbols,* or just *nonterminals*)

- *Terminals* are lexemes or tokens

- Collection of Production Rules: define how non-terminals can be expanded or replaced by a sequence of symbols, which can include both terminals and other non-terminals.
  - 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

# BNF Fundamentals (cont.)

- Nonterminals are often enclosed in angle brackets
  - Examples of BNF rules:

```
<ident_list> → <identifier>| <identifier>, <ident_list>

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

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

- Grammar: a finite non-empty set of rules

- A *start symbol* is a special element of the nonterminals of a grammar
  - Examples: <program>, <start>

# BNF Rules

- An abstraction (nonterminal symbol) can have more than one RHS

**<stmt> → <single_stmt>**
**| begin <stmt_list> end**

- An abstraction can be described using recursion

**<ident_list> → <ident>**
**| <ident>, <ident_list>**

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

# An Example Grammar

<program> → **begin** <stmt_list> **end**

<stmt_list> → <stmt> | <stmt>; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C | D

<expression> → <var> + <var> | <var> - <var> | <var>

Examples of programs that are syntactically correct with respect to the above BNF:

```
begin
   A = B + C
end
```

```
begin
   A = B;
   C = A + D
end
```

```
begin
   A = B;
   C = A + D;
   B = C
end
```

# Derivation Example

A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (**all terminal symbols**)

Example: **begin**
A = B – C
**end**

<program> → **begin** <stmt_list> **end**
<stmt_list> → <stmt> | <stmt>; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C | D
<expression> → <var> + <var> | <var> - <var> | <var>

Derivation: <program> ← start symbol
=> **begin** <stmt_list> **end**
=> **begin** <stmt> **end**
=> **begin** <var> = <expression> **end**
=> **begin** A = <expression> **end**
=> **begin** A = <var> - <var> **end**
=> **begin** A = B - <var> **end**
=> **begin** A = B - C **end**

# Derivations

- **Sentential form**
  - A string of symbols in a derivation
- **A sentence** is a sentential form that has only terminal symbols
- **Leftmost derivation**
  - the replaced non-terminal is always the leftmost non-terminal
- **Rightmost derivation**
  - the replaced non-terminal is always the rightmost non-terminal
- A derivation may be neither leftmost nor rightmost

# Another Example of Derivation

```
begin
        A = B + C;
        B = C
end
```

<program> → **begin** <stmt_list> **end**
<stmt_list> → <stmt> | <stmt>; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C | D
<expression> → <var> + <var> | <var> - <var> | <var>

Rightmost Derivation: <program> (start symbol)

=> **begin** <stmt_list> **end**
=> **begin** <stmt>; <stmt_list> **end**
=> **begin** <stmt>; <stmt> **end**
=> **begin** <stmt>; <var> = <expression> **end**
=> **begin** <stmt>; <var> = <var> **end**
=> **begin** <stmt>; <var> = C **end**
=> **begin** <stmt>; B = C **end**
=> **begin** <var> = <expression>; B = C **end**
=> **begin** <var> = <var> + <var>; B = C **end**
=> **begin** <var> = <var> + C; B = C **end**
=> **begin** <var> = B + C; B = C **end**
=> **begin** A = B + C; B = C **end**

# Parse Tree

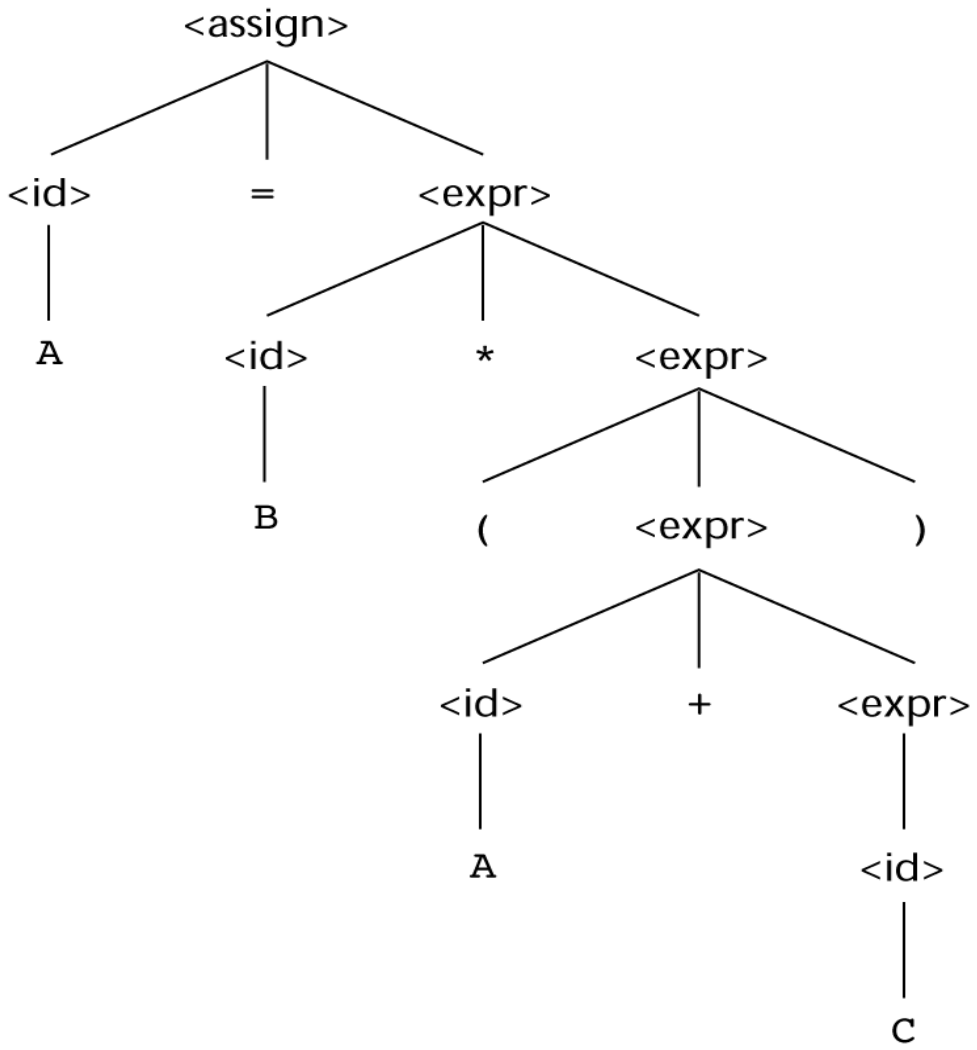
- A hierarchical structure that shows the derivation process

<assign> → <id> = <expr>
<id> → A | B | C | D
<expr> → <id> + <expr>
| <id> * <expr>
| ( <expr> )
| <id>

Sentence: **A = B * (A + C)**

**Leftmost derivation:**

<assign>
⇒ <id> = <expr>
⇒ A = <expr>
⇒ A = <id> * <expr>
⇒ A = B * <expr>
⇒ A = B * ( <expr> )
⇒ A = B * ( <id> + <expr> )
⇒ A = B * ( A + <expr> )
⇒ A = B * ( A + <id> )
⇒ A = B * ( A + C )



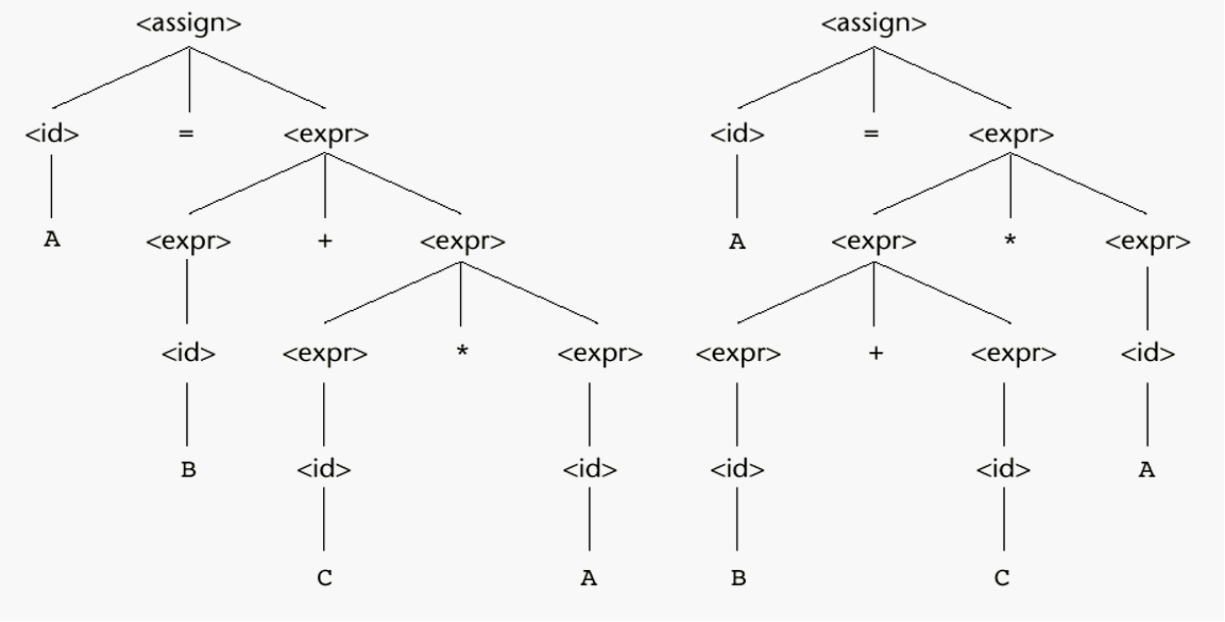
# Ambiguity in Grammars

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

<assign> → <id> = <expr>
<id> → A | B | C | D
<expr> → <expr> + <expr>
| <expr> * <expr>
| ( <expr> )
| <id>

A = B + C * A

Both leftmost derivation



# Example (cont.)

- How to evaluate B + C * A?
  - B + (C * A)?
  - (B + C) * A?
- Grammar

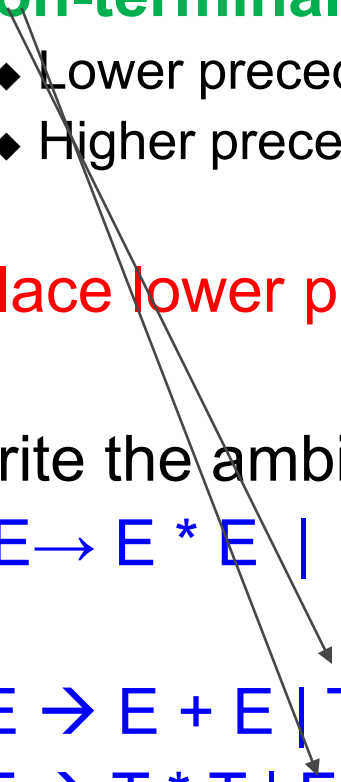
  <expr> -> <expr> * <expr> | <expr> + <expr> | (<expr>) | <id>

  is ambiguous

Issues: operator precedence and associativity

# Remove Ambiguity in Grammars

- Step I: **Specify precedence**
  - Build precedence into grammar by using a different **non-terminal** for each precedence level. Insight:
    - Lower precedence — tend to be higher in the tree (close to root)
    - Higher precedence — tend to be lower in the tree (far from root)
  - Place lower precedence non-terminals higher up in the tree



- Rewrite the ambiguous grammar:

  E→ E * E | E + E | ( E ) | id

  as:

  E → E + E | T  //lowest precedence **+**

  T → T * T | F  //middle precedence ***

  F → (E) | id  //highest precedence **( )**

E: Expression
T: Term
F: Factor

# Remove Ambiguity in Grammars (cont.)

- Step II: **Specify associativity**
  - Allow recursion only on either left or right non-terminal
    - **Left associative — recursion on left non-terminal**
    - **Right associative — recursion on right non-terminal**
- After step I, still ambiguous due to associativity
  - E → E + E ... **allows both left/right associativity (problem)**
- Rewrite:

E → E + E | T  //lowest precedence +
T → T * T | F  //middle precedence *
F → (E) | id  //highest precedence ()
to
E → E + T | T  // + is left-associative
T → T * F | F  // * is left-associative
F → (E) | id

# Example 1

- Grammar E → E * E | E + E | ( E ) | id is ambiguous
  - Rewrite to (non-ambiguous)

Step I:

E → E + E | T

T → T * T | F

F → (E) | id

Step II:

E → E + T | T

T → T * F | F

F → (E) | id

- The id + id * id has only one parse tree:

E ⇒ E + T

⇒ T + T

⇒ F + T

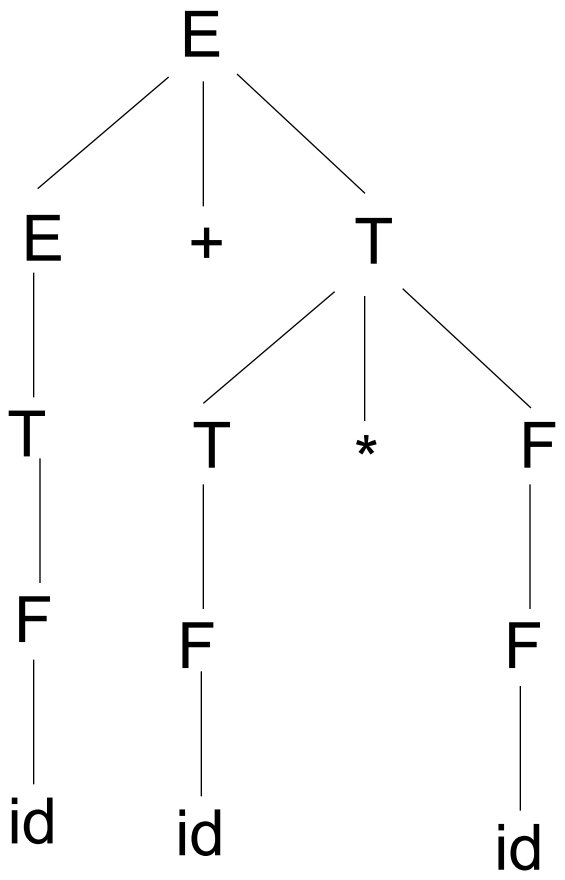
⇒ id + T

⇒ id + T * F

⇒ id + F * F

⇒ id + id * F

⇒ id + id * id



# Example 2

## Step I: Specify precedence

**Before:**

<assign> → <id> = <expr>

<id> → A | B | C | D

<expr> → <expr> + <expr>
| <expr> * <expr>
| ( <expr> )
| <id>

**After:**

<assign> → <id> = <expr>

<id> → A | B | C | D

<expr> → <expr> + <expr>
| <term>

<term> → <term> * <term>
| <factor>

<factor> → ( <expr> )
| <id>

# Example 2 (cont.)

## Step II: Specify associativity

**From Step I:**

<assign> → <id> = <expr>

<id> → A | B | C | D

<expr> → <expr> + <expr>
| <term>

<term> → <term> * <term>
| <factor>

<factor> → ( <expr> )
| <id>

**After Step 2:**

<assign> → <id> = <expr>

<id> → A | B | C | D

<expr> → <expr> + <term>
| <term>

<term> → <term> * <factor>
| <factor>

<factor> → ( <expr> )
| <id>

**Left-recursive rule**

# A parse tree after removing ambiguity

A = B + C + A

<assign> → <id> = <expr>

<id> → A | B | C | D

<expr> → <expr> + <term>

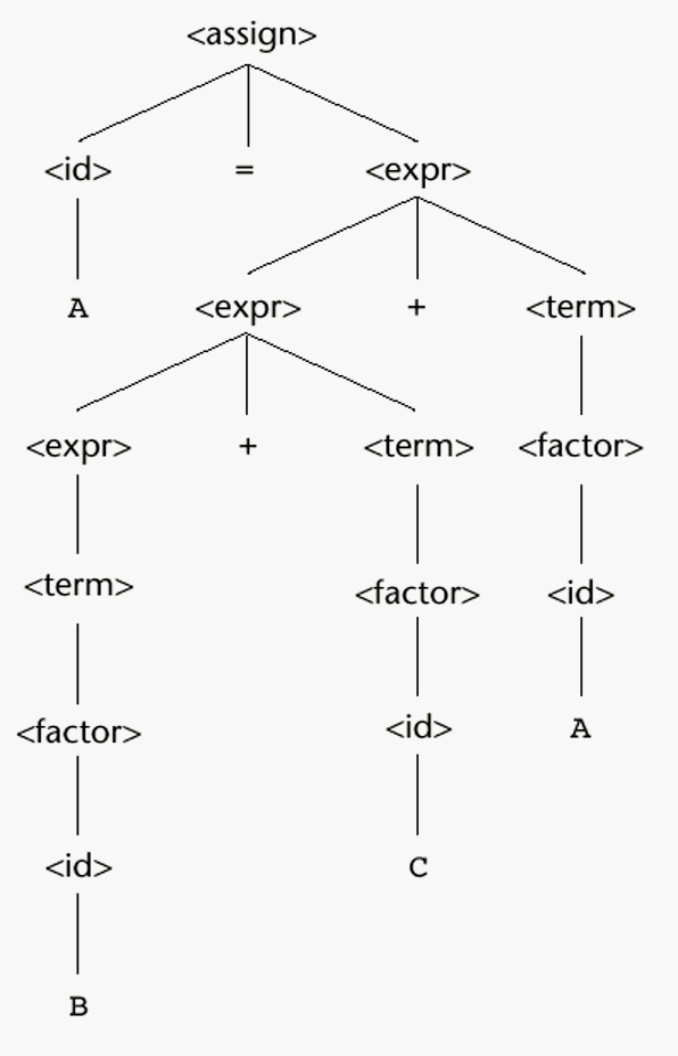
| <term>

<term> → <term> * <factor>

| <factor>

<factor> → ( <expr> )

| <id>

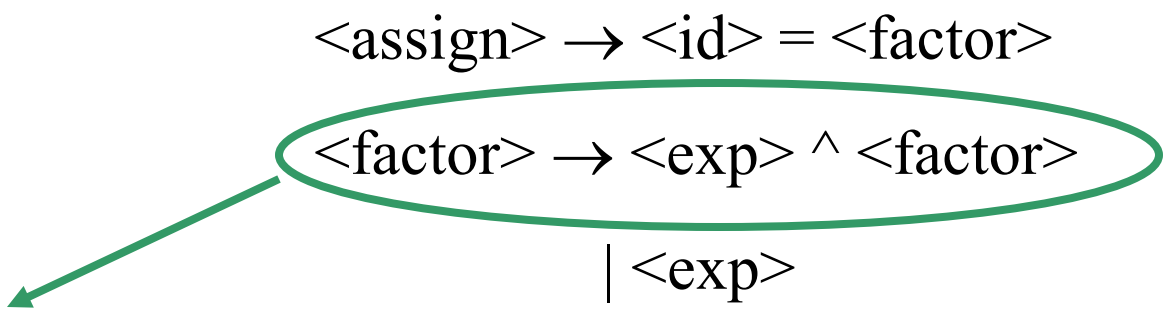



**Leftmost derivation and Left-associative**

# Example 3: Associativity of Operators

In most language, the exponentiation operator is **right associative**.

2^3^2 = 2^9 = 512, instead of 8^2 = 64




**Right-recursive rule**

<exp> → (<expr>) | <id>

<id> → A | B | C | D

**Exercise:** Draw the parse tree for A = B^C^D
(use rightmost derivation)

# Ambiguous Grammar for if statement

- Is the following grammar ambiguous?

`if-then-else` grammar:

```
<stmt> ->  if (<logic_expr>) <stmt>
         | if (<logic_expr>) <stmt> else <stmt>
         | <assign-stmt>
```

**<assign-stmt> → <id> = <expr>**

**Given the following if statement:**

**if ( x > 5) if (y = 5) z=12 else z = 20**

**We have two different derivations for this statement**



```
if ( x > 5)
   if (y = 5)
      z=12
   else  z = 20
```

```
if ( x > 5)
   if (y = 5)
      z=12
else  z = 20
```

**Dangling-else**: an optional **else** clause in a nested conditional structure becomes ambiguous.

# The Dangling Else: A Fix

- else matches the closest unmatched then clause
- We can describe this in the grammar

Ambiguous **if statement** →

```
<stmt> -> if (<logic_expr>) <stmt>
        | if (<logic_expr>) <stmt> else <stmt>
        | <assign-stmt>
```

**Unambiguous if statement:**

<stmt> → <UIF>
| <MIF>
| <assign-stmt>

<UIF> → if (<logical_expr>) <stmt>
| if (<logical_expr>) **<MIF>** else <UIF> /* nested if, last one is UIF */

<MIF> → if (<logical_expr>) <MIF> else <MIF> /* all if are matched */
| <assign-stmt>

<logical_expr> → <id> **>** <id>
| <id> **<** <id>
| <id> = <id>

# Extended BNF

- BNF rules may grow unwieldy for complex languages
- Extended BNF
  - Provide extensions to “abbreviate” the rules into much simpler forms
  - Does not enhance descriptive power of BNF
  - Increase readability and writability

# Extended BNF (cont.)

- Optional parts of an RHS are placed in brackets ([ ])

  replace: <select_stmt> → **if** ( <expr> ) <stmt>
  | **if** ( <expr> ) <stmt> **else** <stmt>

  with: <select_stmt> → **if** ( <expr> ) <stmt> **[ else** <stmt> **]**

- When a single element must be chosen from a group, the options are placed in parentheses and separated by the OR operator, |.

  replace: <term> → <term> **+** <const>
  | <term> **-** <const>

  with: <term> → <term> **(+ | -)** <const>

- Put repetitions (0 or more) in braces ({ })

  replace: <id_list> → <id> | <id>, <id_list>

  with: <id_list> → <id> **{** , <id> **}**

# Extended BNF (Example)

BNF:

<expr> → <expr> + <term>
| <expr> - <term>
| <term>

<term> → <term> * <factor>
| <term> / <factor>
| <factor>

<factor> → <exp> ^ <factor>
| <exp>

<exp> → ( <expr> )
| <id>

EBNF:

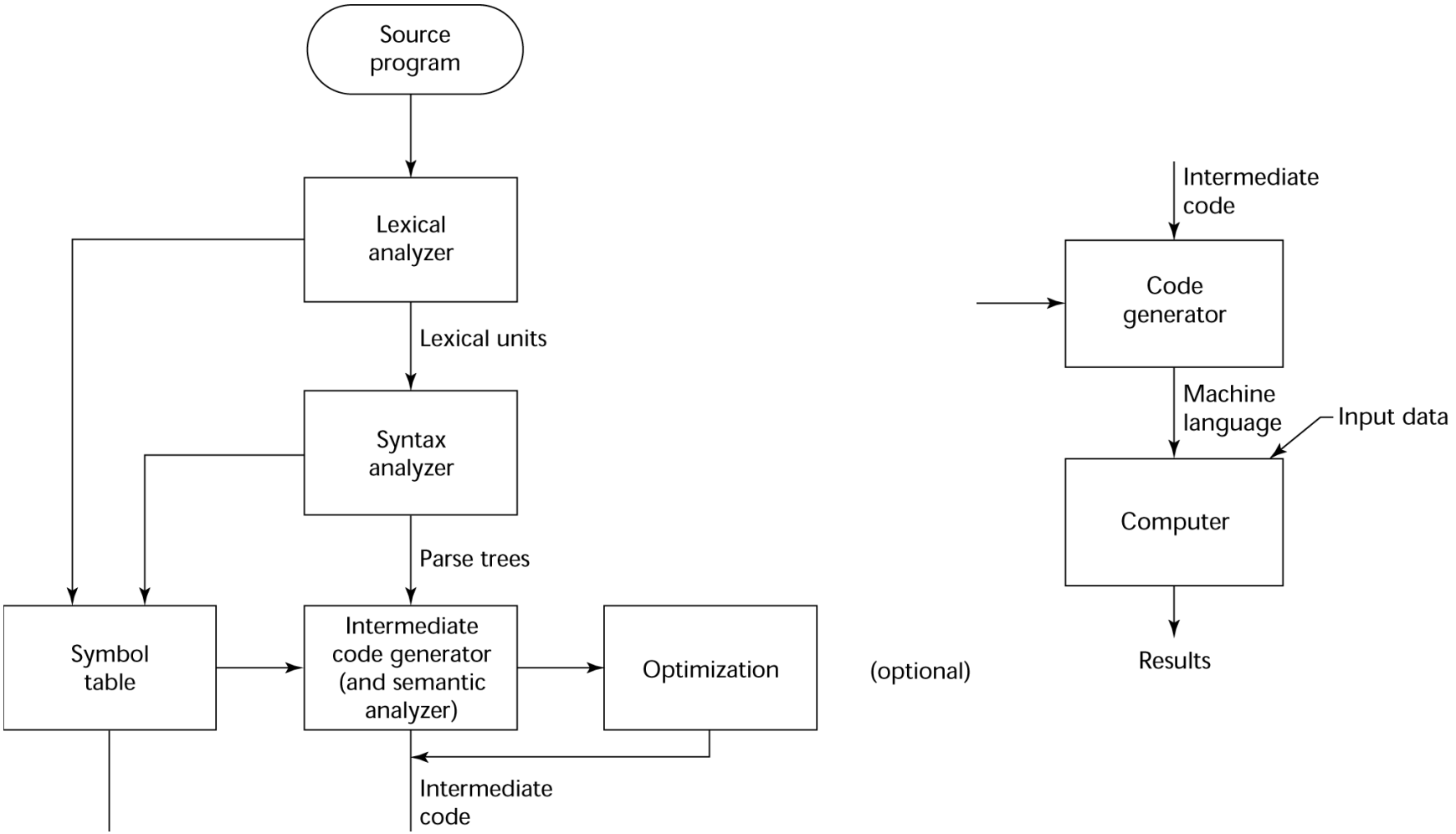
<expr> → <term> {(+|-) <term>}

<term>→<factor>{(*|/)<factor>}

<factor> →<exp>{^ <exp>}

<exp> → ( <expr> )
| <id>

# Compilation



# Syntax Analysis

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a ***lexical analyzer*** (mathematically, a finite automaton based on a regular grammar)
  - A high-level part called a ***syntax analyzer***, or parser (mathematically, a push-down automaton based on a context-free grammar, or BNF)

# Reasons to Separate Lexical and Syntax Analysis

- *Simplicity* - less complex approaches can be used for lexical analysis; separating them simplifies the parser
- *Efficiency* - separation allows optimization of the lexical analyzer
- *Portability* - parts of the lexical analyzer may not be portable, but the parser always is portable

# Lexical Analyzer

- A pattern matcher for character strings
- The “front-end” for the parser
- Identifies substrings of the source program that belong together => lexemes
  - Lexemes match a character pattern, which is associated with a lexical category called a token

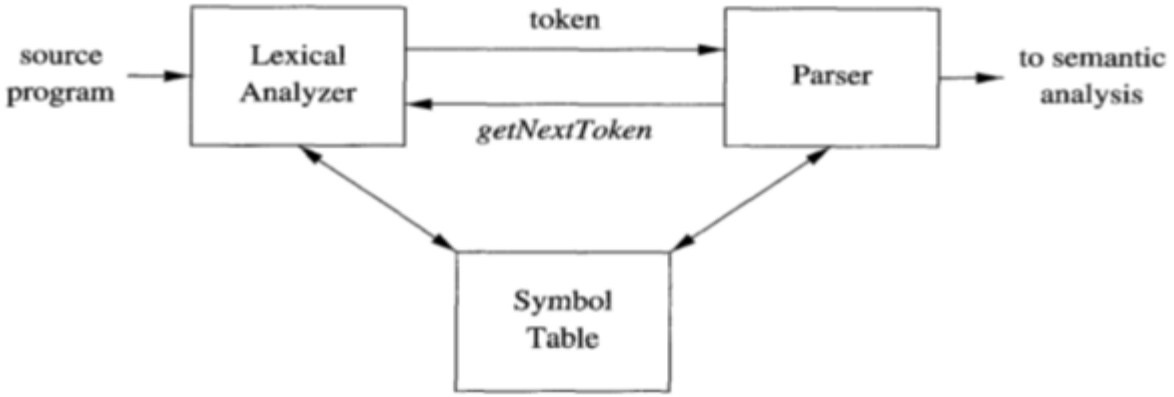
Example: sum = B – 5;

| **Lexeme** | **Token** |
|---|---|
| sum | ID (identifier) |
| = | ASSIGN_OP |
| B | ID |
| - | SUBTRACT_OP |
| 5 | INT_LIT (integer literal) |
| ; | SEMICOLON |

# Lexical Analyzer (cont.)

**Functions:**

- Extract lexemes from a given input string and produce the corresponding tokens, while skipping comments and blanks
- Insert lexemes for user-defined names into symbol table, which is used by later phases of the compiler
- Detect syntactic errors in tokens and report such errors to user
- Provide the interface between the source program and the parser



# Lexical Analyzer (cont.)

- The lexical analyzer is usually a function that is called by the parser when it needs the next token
- Three approaches to building a lexical analyzer:
  1. Write a formal description of the tokens and use a software tool that constructs a table-driven lexical analyzer from such a description
  2. Design a state diagram that describes the tokens and write a program that implements the state diagram
  3. Design a state diagram that describes the tokens and hand-construct a table-driven implementation of the state

State Transition Table recognizing identifier, numbers and few operators

| State | Digit (0-9) | Letter (A-Z, a-z) | + | - | * | / | Whitespace | Other |
|---|---|---|---|---|---|---|---|---|
| **S0** | S1 | S2 | OP | OP | OP | OP | S0 | ERR |
| **S1** | S1 | ERR | F | F | F | F | F | ERR |
| **S2** | S2 | S2 | F | F | F | F | F | ERR |

# State Diagram Design

- A naïve state diagram would have a transition from every state on every character in the source language - such a diagram would be very large!

# State Diagram Design (cont.)

- In many cases, transitions can be combined to simplify the state diagram
  - When recognizing an identifier, all uppercase and lowercase letters are equivalent
    - ✓ Use a character class that includes all letters
  - When recognizing an integer literal, all digits are equivalent
    - ✓ Use a character class that includes all digits
- Reserved words and identifiers can be recognized together (rather than having a part of the diagram for each reserved word)
  - Use a table lookup to determine whether a possible identifier is in fact a reserved word

# Lexical Analyzer

- Useful subprograms (functions) in the lexical analyzer:
  - **`lookup`**
    - determines whether the string in lexeme is a reserved word (returns a code)
  - **`getChar`**
    - reads the next character of input string, puts it in a global variable called **`nextChar`**, determines its character class (letter, digit, etc.) and puts the class in **`charClass`**
  - **`addChar`**
    - Appends nextChar to the current lexeme

```
/* Character classes */
#define LETTER 0
#define DIGIT 1
#define UNKNOWN 99

/* Token codes */
#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LEFT_PAREN 25
#define RIGHT_PAREN 26
```

```
/*****************************************************/
/* addChar - a function to add nextChar to lexeme */
void addChar() {
  if (lexLen <= 98) {
    lexeme[lexLen++] = nextChar;
    lexeme[lexLen] = 0;
  }
  else
    printf("Error - lexeme is too long \n");
}

/*****************************************************/
/* getChar - a function to get the next character of
             input and determine its character class */
void getChar() {
  if ((nextChar = getc(in_fp)) != EOF) {
    if (isalpha(nextChar))
      charClass = LETTER;
    else if (isdigit(nextChar))
          charClass = DIGIT;
        else charClass = UNKNOWN;
   }
   else
     charClass = EOF;
}
```

# Lexical Analyzer

**How to build a lexical analyzer**?

- Create a state transition diagram first (Finite State Machine)
  - A state diagram is a directed graph
  - Nodes are labeled with state names **(none-terminals)**
    - One of the nodes is designated as the start node
  - Arcs are labeled with input characters that cause the state transitions

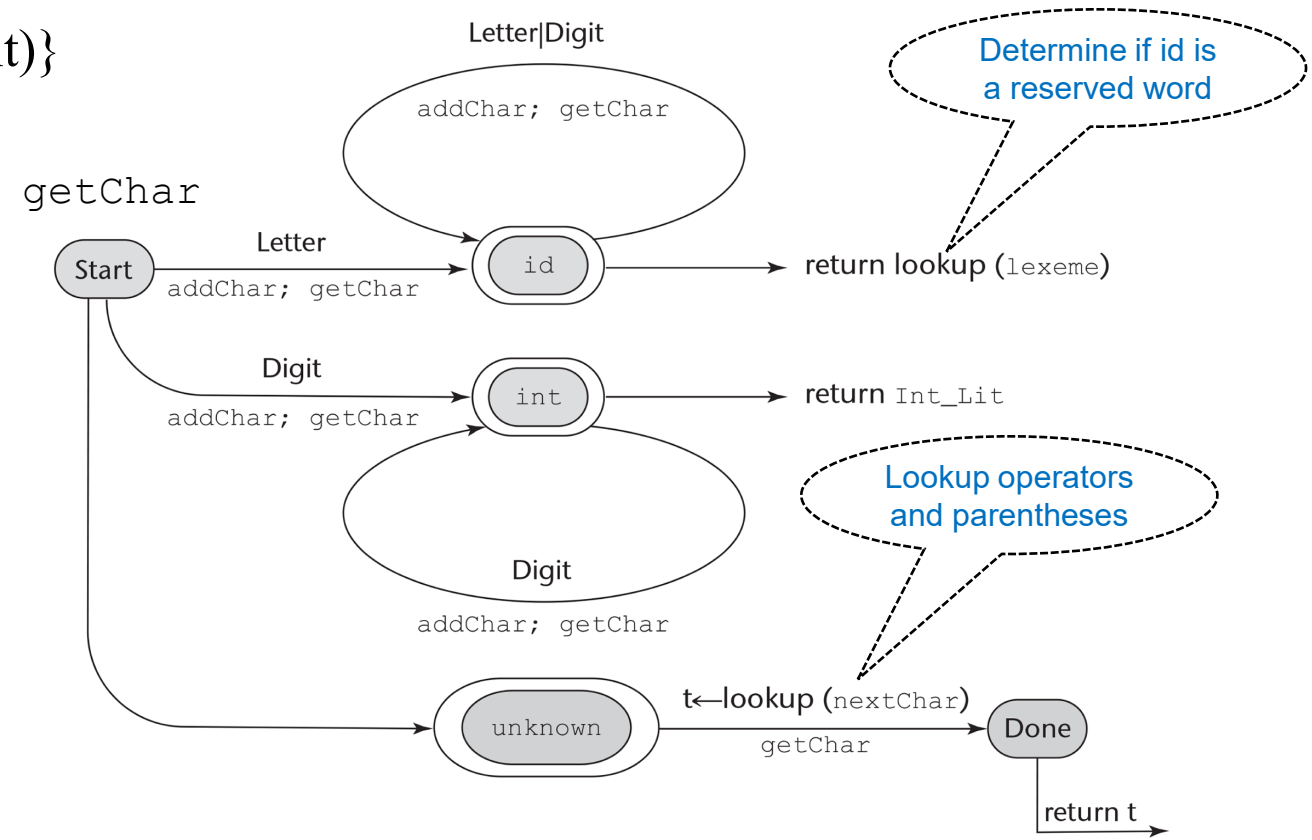
# State Transition Diagram

## A state diagram to recognize identifies, integers, parentheses and arithmetic operators

Letter → A | B | C |…| Z | a | b | … | z
Digit → 0 | 1 | 2 | … | 9
id → Letter{(Letter|Digit)}
int → Digit{Digit}

```
/* Character classes */
#define LETTER 0
#define DIGIT 1
#define UNKNOWN 99


/* Token codes */
#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LEFT_PAREN 25
#define RIGHT_PAREN 26
```



# A C Implementation of a Lexical Analyzer

Textbook (12th Edition), pages 166-171

```
/* front.c - a lexical analyzer system for simple
             arithmetic expressions */

#include <stdio.h>
#include <ctype.h>

/* Global declarations */
/* Variables */
int  charClass;
char  lexeme [100];
char  nextChar;
int  lexLen;
int  token;
int  nextToken;
FILE *in_fp, *fopen();
```

```
/* Function declarations */
void  addChar();
void  getChar();
void  getNonBlank();
int  lex();

/* Character classes */
#define LETTER 0
#define DIGIT 1
#define UNKNOWN 99

/* Token codes */
#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LEFT_PAREN 25
#define RIGHT_PAREN 26
```

```
/*****************************************************/
/* main driver */
main() {

/* Open the input data file and process its contents */
   if  ((in_fp = fopen("front.in", "r")) == NULL)
     printf("ERROR - cannot open front.in \n");
   else  {
     getChar();
     do  {
       lex();
     }  while  (nextToken != EOF);
   }
}
```

```
/*****************************************************/
/* lookup - a function to lookup operators and parentheses
            and return the token */
int  lookup(char  ch) {
  switch  (ch) {
    case  '(':
      addChar();
      nextToken = LEFT_PAREN;
      break;

    case  ')':
      addChar();
      nextToken = RIGHT_PAREN;
      break;

    case  '+':
      addChar();
      nextToken = ADD_OP;
      break;

    case  '-':
      addChar();
      nextToken = SUB_OP;
      break;

    case  '*':
      addChar();
      nextToken = MULT_OP;
      break;

    case  '/':
      addChar();
      nextToken = DIV_OP;
      break;

     default:
       addChar();
       nextToken = EOF;
       break;
   }
   return  nextToken;
}
```

```
/*****************************************************/
/* addChar - a function to add nextChar to lexeme */
void addChar() {
  if (lexLen <= 98) {
    lexeme[lexLen++] = nextChar;
    lexeme[lexLen] = 0;
  }
  else
    printf("Error - lexeme is too long \n");
}

/*****************************************************/
/* getChar - a function to get the next character of
             input and determine its character class */
void getChar() {
  if ((nextChar = getc(in_fp)) != EOF) {
    if (isalpha(nextChar))
      charClass = LETTER;
    else if (isdigit(nextChar))
           charClass = DIGIT;
         else  charClass = UNKNOWN;
   }
   else
     charClass = EOF;
}

/*****************************************************/
/* getNonBlank - a function to call getChar until it
                 returns a non-whitespace character */
void getNonBlank() {
  while  (isspace(nextChar))
    getChar();
}
```

```
/*****************************************************/
/* lex - a simple lexical analyzer for arithmetic
         expressions */
int lex() {
  lexLen = 0;
  getNonBlank();
  switch  (charClass) {
  /* Parse identifiers */
      case  LETTER:
        addChar();
        getChar();
        while  (charClass == LETTER || charClass == DIGIT) {
          addChar();
          getChar();
        }
      nextToken = IDENT;
      break;

  /* Parse integer literals */
      case  DIGIT:
        addChar();
        getChar();
        while  (charClass == DIGIT) {
          addChar();
          getChar();
        }
        nextToken = INT_LIT;
        break;

  /* Parentheses and operators */
      case  UNKNOWN:
        lookup(nextChar);
        getChar();
        break;

  /* EOF */
      case  EOF:
        nextToken = EOF;
        lexeme[0] = 'E';
        lexeme[1] = 'O';
        lexeme[2] = 'F';
        lexeme[3] = 0;
        break;
    } /* End of switch */
    printf("Next token is: %d, Next lexeme is %s\n",
           nextToken, lexeme);
    return  nextToken;
  }  /* End of function lex */
```

# A C implementation of a lexical analyzer (cont.)

The output of the lexical analyzer of front.c when used on (sum + 47) / total,

Next token is: 25 Next lexeme is (
Next token is: 11 Next lexeme is sum
Next token is: 21 Next lexeme is +
Next token is: 10 Next lexeme is 47
Next token is: 26 Next lexeme is )
Next token is: 24 Next lexeme is /
Next token is: 11 Next lexeme is total
Next token is: -1 Next lexeme is EOF

```
/* Character classes */
#define LETTER 0
#define DIGIT 1
#define UNKNOWN 99


/* Token codes */
#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LEFT_PAREN 25
#define RIGHT_PAREN 26
```

# The Parser

- Goals of the parser, given an input program:
  - Find all syntax errors; for each, produce an appropriate diagnostic message and recover quickly
  - Produce the parse tree, or at least a trace of the parse tree, for the program

# Parsers (Syntax Analyzers)

- Two categories of parsers:
  - **Top down**
    - produces the parse tree, beginning at the root
    - Order is that of a leftmost derivation
    - Traces or builds the parse tree in preorder
  - **Bottom up**
    - produces the parse tree, beginning at the leaves
    - Order is that of the reverse of a rightmost derivation
- Useful parsers look only one token ahead in the input

# Notation Conventions

A small set of notational conventions for grammar symbols and strings to simplify the representation of grammar rules.

- **Terminal symbols**: lowercase letters at the beginning of the alphabet (a, b, . . .)
- **Nonterminal symbols**: uppercase letters at the beginning of the alphabet (A, B, . . .)
- **Terminals or nonterminals**: uppercase letters at the end of the alphabet (W, X, Y, Z)
- **Strings of terminals**: lowercase letters at the end of the alphabet (w, x, y, z)
- **Mixed strings** (terminals and/or nonterminals)—lowercase Greek letters (α, β, γ, δ)

# Top-Down Parsers

- Top-down Parsers
  - Given a sentential form, $xA\alpha$ , the parser must choose the correct A-rule to get the next sentential form in the leftmost derivation, using only the first token produced by the A-rule
- The most common top-down parsing algorithms
  - Recursive descent - a coded implementation
  - Table driven implementation
- Both Recursive descent and Table driven are called LL algorithm
  - 1st "L": Left-to-right scan of input
  - 2nd "L": a leftmost derivation is generated

# Bottom-Up Parsers

- Bottom-up Parsers
  - Given a right sentential form α, the parser identifies which substring of α corresponds to the right-hand side of a grammar rule that can be reduced to derive the previous sentential form in a rightmost derivation.
  - The most common bottom-up parsing algorithms belong to the LR family, where:
    - ✓ "L" stands for a left-to-right scan of the input
    - ✓ "R" indicates that a rightmost derivation is being constructed.

Example:

S → aAc

A → aA | b

S =>

aAc =>

aaAc =>

aabc

# The Complexity of Parsing

- Parsers that work for any unambiguous grammar are complex and inefficient ($O(n^3)$, where n is the length of the input )
- Compilers use parsers that only work for a subset of all unambiguous grammars, but do it in linear time (O(n), where n is the length of the input )

# Recursive-Descent Parsing

- There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal

- EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of nonterminals

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

A grammar for simple expressions

# Recursive-Descent Parsing (cont.)

- Assume we have a lexical analyzer called `lex`, which stores the next token code in `nextToken`
- The coding process when there is only one RHS:
  - For each terminal symbol in the RHS, compare it with the next input token. If they match, proceed; otherwise, an error is encountered.
  - For each nonterminal symbol in the RHS, invoke its corresponding parsing subprogram

# Recursive-Descent Parsing (cont.)

Subprogram for **<expr>**

- **lex()**: lexical analyzer function. It gets the next lexeme and stores its token code in the global variable **nextToken**
- **Convention**: every parsing routine leaves the next token in nextToken
- Parser uses leftmost derivation
- This particular routine does not detect errors

```
/* Function expr
   Parses strings in the language
   generated by the rule:
   <expr> → <term> {(+ | -) <term>}
 */

void expr() {

/* Parse the first term */

  term();
/* As long as the next token is + or -, call
   lex to get the next token and parse the
   next term */

  while (nextToken == ADD_OP ||
         nextToken == SUB_OP){
    lex();
    term();
  }
}
```

# Recursive-Descent Parsing (cont.)

Subprogram for **<term>**

```
/* term
Parses strings in the language generated by
   the rule:
<term> -> <factor> {(* | /) <factor>)
*/
void term() {

/* Parse the first factor */
  factor();

/* As long as the next token is * or /, get
   next token and parse the next factor */
  while (nextToken == MULT_OP || nextToken
   == DIV_OP) {
    lex();
    factor();
  }
} /* End of function term */
```

# Recursive-Descent Parsing (cont.)

- A nonterminal with multiple RHS requires an initial step to determine which RHS to parse
  - The correct RHS is selected based on the next input token (the lookahead)
  - The next token is compared with the first token that each RHS can generate until a match is found
  - If no match is found, a syntax error occurs

# Recursive-Descent Parsing (cont.)

Subprogram for **<factor>**

```
/* Function factor
   Parses strings in the language generated by the
   rule: <factor> -> id | Int | (<expr>)  */

 void factor() {
 /* Determine which RHS */
   if (nextToken) == ID_CODE || nextToken == INT_CODE)
 /* For the RHS id or int, just call lex */
      lex();
/* If the RHS is (<expr>) - call lex to pass over the
   left parenthesis, call expr, and check for the right
   parenthesis */
   else if (nextToken == LP_CODE) {
            lex();
            expr();
            if (nextToken == RP_CODE)
               lex();
            else
               error();
         }  /* End of else if (nextToken == ...  */
         else error(); /* Neither RHS matches */
 }
```

# Recursive-Descent Parsing (cont.)

Trace of the lexical and syntax analyzers on **(sum + 47) / total**

```
Next token is: 25 Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>  choose (<expr>)
Next token is: 11 Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>  choose id
Next token is: 21 Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 10 Next lexeme is 47
Enter <term>
Enter <factor>  choose int
Next token is: 26 Next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 24 Next lexeme is /
Exit <factor>
```

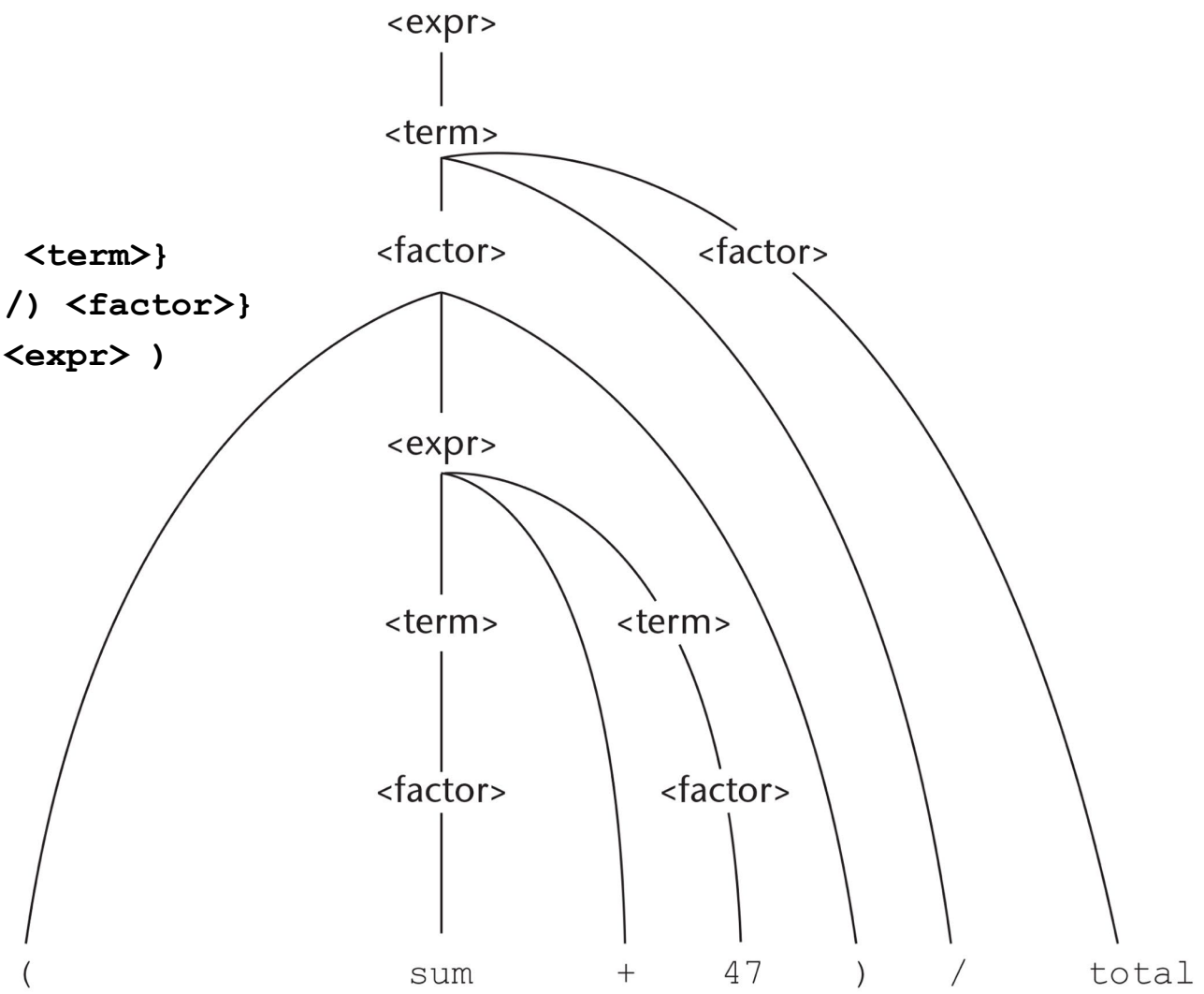
```
Next token is: 11 Next lexeme is total
Enter <factor>  choose id
Next token is: -1 Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
```

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

# Recursive-Descent Parsing (cont.)

**The parse tree for (sum + 47) / total**

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



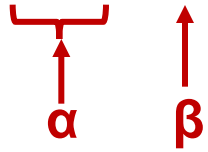
# The LL Grammar Class Problems

- **1st Problem**: The Left Recursion Problem
- If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser
  - $A \rightarrow A + B$ (direct left recursion)
  - $A \rightarrow B\ c\ D$ (indirect left recursion)
    $B \rightarrow A\ b$
- Solution: Remove direct left recursion as follows:
- For each nonterminal, A,
  1. Group the A-rules as $A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$, where none of the $\beta_i$ 's begins with A
  2. Replace the original A-rules with
     $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$
     $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

# The LL Grammar Class Problems (cont.)

An example of eliminating left recursion:

Left recursion rules: $E \rightarrow E + T \mid T$




→→

Without left recursion:

$E \rightarrow T\ E'$

$E' \rightarrow + T\ E' \mid \varepsilon$

$A \rightarrow A\,\alpha \mid \beta$ →→ $A \rightarrow \beta\ A'$

$A' \rightarrow \alpha\ A' \mid \varepsilon$

# The LL Grammar Class Problems (cont.)

- **2nd Problem**: Inability to determine the correct RHS on the basis of one token of lookahead

  Example1: A -> aB | bAb | Bb

  B -> cB | d

  able to determine the correct RHS by lookahead of one token

  Example2: A -> aB | BAb

  B -> aA | b

  → **RHS has Common prefixes**: a

  unable to determine the correct RHS by lookahead of only one token

- **left factoring** process: eliminate common prefixes

  <variable> → identifier | identifier [<expression>] →

  <variable> → identifier <new>

  <new> → ε | [<expression>]

# Bottom-up Parsing

- The process of bottom-up parsing generates the reverse of a rightmost derivation.
- The task of the bottom-up parser is to find the specific RHS, **handle**, in the right sentential form that must be reduced to get the previous sentential form.
- Example: a grammar for arithmetic expression

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow E \mid id$

Rightmost derivation for id + id * id,

$E \rightarrow E + T$

$\rightarrow E + T * F$

$\rightarrow E + T * id$

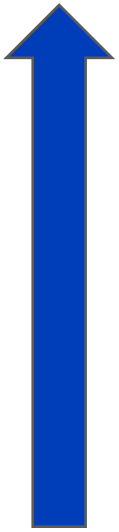
$\rightarrow E + F * id$

$\rightarrow E + id * id$

$\rightarrow T + id * id$

$\rightarrow F + id * id$

$\rightarrow id + id * id$



**Bottom-up Parsing**

**Handle**: ◯

# Bottom-up Parsing (cont.)

- The right sentential form **E + T * id** includes three RHSs, **E + T**, **T**, and **id**, but only id is the handle.
- If the RHS E + T is selected as the handle, the resulting sentential form would be E * id which is not a legal right sentential form in the given grammar.
- If the RHS T is selected as the handle, the resulting sentential form would be E + E * id which is not a legal right sentential form in the given grammar

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow E \mid id$

# Bottom-up Parsing - handle

- Definition: β is the **handle** of the right sentential form $\gamma = \alpha\beta w$ if and only if $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$

  $\Rightarrow^*_{rm}$ specifies zero or more rightmost derivation steps,

  $\Rightarrow_{rm}$ specifies a rightmost derivation step

- Definition: β is a **phrase** of the right sentential form γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1 \beta \alpha_2$

  $\Rightarrow^+$ indicates one or more derivation steps

- Definition: β is a **simple phrase** of the right sentential form γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$

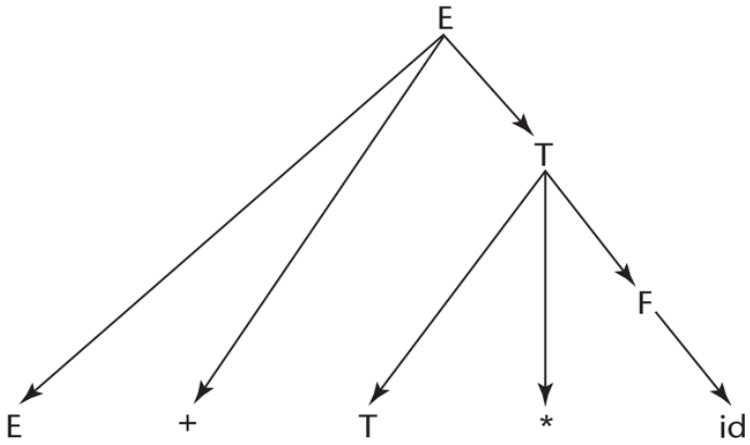
# Bottom-up Parsing – handle (cont.)

- Consider the parse tree

  $E \rightarrow E + T \mid T$

  $T \rightarrow T * F \mid F$

  $F \rightarrow E \mid id$

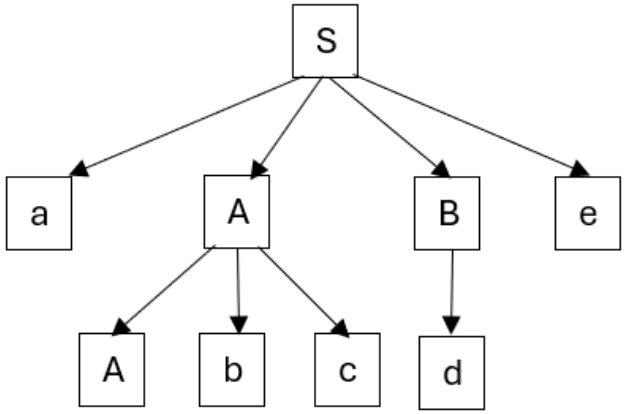



- The leaves of the parse tree comprise the sentential form **E + T * id**
- Each internal node (nonterminal) is the root of a subtree, whose leaves are a phrase. The phrases of the sentential form E + T * id are **E + T * id**, **T * id**, and **id**
- Phrases are not necessarily RHSs in the underlying grammar
- A simple phrase is always an RHS in the grammar → **id** is a simple phrase of the sentential form E + T * id
- The **handle** of any rightmost sentential form is its **leftmost simple phase**

# Bottom-up Parsing – handle (cont.)

Example:

S → aABe
A → Abc | b
B → d




S => aABe
=> aAde
=> aAbcde

Find phrase, simple phrase, and handle for aAbcde

- **Phrase**:
  aAbcde, Abc, d
- **Simple Phrase**:
  Abc, d
- **Handle**:
  Abc

# Bottom-up Parsing – Shift-Reduce Algorithms

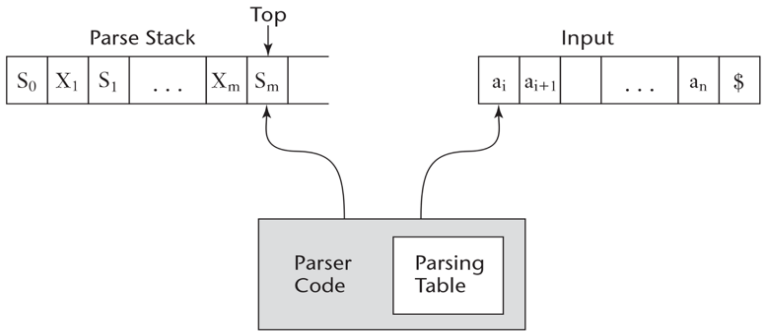
- Every bottom-up parser is a Pushdown Automation (PDA)
- Bottom-up parsers are often called shift-reduce algorithms, because **shift** and **reduce** are the two most common actions they specify
  - ✓ **Shift**: moving the next input token to the top of the parse stack
  - ✓ **Reduce**: replacing the handle on the top of the parse stack with its corresponding LHS

# Bottom-up Parsing – LR Parser

- Many different bottom-up parsing algorithms have been designed. Most of them are variations of a process called LR
- Three key advantages of LR parsers:
  - ✓ They can be built for all programming languages
  - ✓ They can detect syntax errors as soon as it is possible in a left-to-right scan.
  - ✓ The LR class of grammars is a proper superset of the class parsable by LL parsers (e.g., many left recursive grammars are LR, but none are LL).

# Bottom-up Parsing – LR Parser (cont.)

- The following figure shows the structure of an LR parser,




- The contents of the parse stack for an LR parser have the following form,

$$S_0X_1S_1X_2 \ldots X_mS_m \text{ (top)}$$

- Where the S's are state symbols and the X's are grammar symbols.
- An LR parser configuration is a pair of strings (stack, input), with the detailed form,

$$(S_0X_1S_1X_2 \ldots X_mS_m, a_ia_{i+1} \ldots a_n\$)$$

# Bottom-up Parsing – LR Parsing Table

- An **LR parsing table** has two parts, **ACTION** and **GOTO**
- The **ACTION** part has **state symbols** as its row labels and the **terminal symbols** of the grammar as its column labels.
- Given a current parser state (the state symbol on top of the parse stack), and the next symbol (token) of input, the ACTION part specifies what the parser should do (**Shift**, **Reduce**, **Accept**, or **Error**).
- The **GOTO** part has state symbols as its row labels and the nonterminal symbols as its column label.
- The GOTO part indicates which state symbols should be pushed onto the stack after a reduction is completed.

# Bottom-up Parsing – LR Parsing Table (cont.)

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow E$
6. $F \rightarrow id$

**Shift**: push the next input symbol and its state from the ACTION table onto the stack.

**Reduce**: replace the handle on the stack with its LHS, then use the GOTO table to determine the new state

**Accept**: parsing is complete with no errors detected.

**Error**: the parser triggers an error-handling routine.

| | Action | | | | | | Goto | | |
|---|---|---|---|---|---|---|---|---|---|
| State | id | + | * | ( | ) | $ | E | T | F |
| 0 | S5 | | | S4 | | | 1 | 2 | 3 |
| 1 | | S6 | | | | accept | | | |
| 2 | | R2 | S7 | | R2 | R2 | | | |
| 3 | | R4 | R4 | | R4 | R4 | | | |
| 4 | S5 | | | S4 | | | 8 | 2 | 3 |
| 5 | | R6 | R6 | | R6 | R6 | | | |
| 6 | S5 | | | S4 | | | | 9 | 3 |
| 7 | S5 | | | S4 | | | | | 10 |
| 8 | | S6 | | | S11 | | | | |
| 9 | | R1 | S7 | | R1 | R1 | | | |
| 10 | | R3 | R3 | | R3 | R3 | | | |
| 11 | | R5 | R5 | | R5 | R5 | | | |

# Bottom-up Parsing – LR Parsing Example

- The initial configuration of an LR parser: $(S_0, a_1 \ldots a_n\$)$
- Trace of parsing the string "id + id * id" using the LR algorithm.

| *Stack* | *Input* | *Action* |
|---|---|---|
| 0 | id+id*id$ | Shift 5 |
| 0id5 | +id*id$ | Reduce 6 (using GOTO[0, F]) |
| 0F3 | +id*id$ | Reduce 4 (using GOTO[0, T]) |
| 0T2 | +id*id$ | Reduce 2 (using GOTO[0, E]) |
| 0E1 | +id*id$ | Shift 6 |
| 0E1+6 | id*id$ | Shift 5 |
| 0E1+6id5 | *id$ | Reduce 6 (using GOTO[6, F]) |
| 0E1+6F3 | *id$ | Reduce 4 (using GOTO[6, T]) |
| 0E1+6T9 | *id$ | Shift 7 |
| 0E1+6T9*7 | id$ | Shift 5 |
| 0E1+6T9*7id5 | $ | Reduce 6 (using GOTO[7, F]) |
| 0E1+6T9*7F10 | $ | Reduce 3 (using GOTO[6, T]) |
| 0E1+6T9 | $ | Reduce 1 (using GOTO[0, E]) |
| 0E1 | $ | Accept |

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow E$
6. $F \rightarrow id$

| | Action | | | | | | Goto | | |
|---|---|---|---|---|---|---|---|---|---|
| State | id | + | * | ( | ) | $ | E | T | F |
| 0 | S5 | | | S4 | | | 1 | 2 | 3 |
| 1 | | S6 | | | | accept | | | |
| 2 | | R2 | S7 | | R2 | R2 | | | |
| 3 | | R4 | R4 | | R4 | R4 | | | |
| 4 | S5 | | | S4 | | | 8 | 2 | 3 |
| 5 | | R6 | R6 | | R6 | R6 | | | |
| 6 | S5 | | | S4 | | | | 9 | 3 |
| 7 | S5 | | | S4 | | | | | 10 |
| 8 | | S6 | | | S11 | | | | |
| 9 | | R1 | S7 | | R1 | R1 | | | |
| 10 | | R3 | R3 | | R3 | R3 | | | |
| 11 | | R5 | R5 | | R5 | R5 | | | |

# Bottom-up Parsing – LR Parsing Table

- A parser table can be generated from a given grammar with a tool, e.g., yacc or bison

# Summary

- Syntax analysis is a common part of language implementation
- A lexical analyzer is a pattern matcher that isolates small-scale parts (lexemes) of a program
- Syntax Analyzers are either top-down or bottom-up
  - Detects syntax errors
  - Produces a parse tree
- A recursive-descent parser is an LL parser
  - EBNF is ideal as the basis for recursive-descent parser
- Parsing problem for bottom-up parsers: find the substring of current sentential form
- The LR family of shift-reduce parsers is the most common bottom-up parsing approach

# Semantics

# Static Semantics

- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages
- Categories of constructs that are trouble:
  - Context-free, but cumbersome (e.g., types of operands in expressions)
  - Non-context-free (e.g., variables must be declared before they are used)
- Static semantics
  - The categories of language rules that further constrain syntactically correct programs
  - Nothing to do with the meaning of programs during execution.
  - It is about the legal forms of programs (syntax rather than semantics)
  - Many static semantics rules of a PL are about its type constrains.
  - Static semantics are verified at compiler time, unlike dynamic semantics (meaning) describing the effect of executing the program.
- BNF cannot describe static semantics
- **Attribute Grammar** is designed to describe both the syntax and the static semantics of PLs
  - Basic idea: add semantic information (attributes) to CFG

# Attribute Grammar

- Attribute Grammars (AGs) have additions to CFGs to carry some semantic info on parse tree nodes.
- Primary value of AGs:
  - Static semantics specification
  - Compiler design (static semantics checking)
- Static semantic rules added into CFG:
  - **Attributes**: associated with grammar symbols (terminal/nonterminal)
  - **Attribute computation functions** (**semantic functions**): associated with grammar rules used to specify how attribute values are computed
  - **Predicate function**: associated with grammar rules, state the static semantic rules of the language.

# Attribute Grammars: Example 1

- Name on the **end** of an Ada procedure must match the procedure's name:

```
Procedure   Proc_name (P: string) is
            begin
                ……
            end Proc_name
```

**This rule can not be stated in BNF**

- Attribute Grammar:

**Syntax rule**: <Proc_def> → Procedure <proc_name>[1]
begin <proc_body> end <proc_name>[2]

**Predicate rule**: <proc_name>[1].string = <proc_name>[2].string

**Note**: the string attribute of <proc_name>, <proc_name>.string, is the actual string of characters found immediately following **Procedure** or **end**

# Attribute Grammar: Definition

- A context-free grammar with the following additions:
  - For each grammar symbol *X* there is a set *A*(*X*) of attribute values
  - Each rule has a (possibly empty) set of *functions* that define certain attributes of the nonterminals in the rule.
  - Each rule has a (possibly empty) set of *predicates* to check for attribute consistency.

# Attribute Grammar: Definition (cont.)

- Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule
- A set of attributes for each symbol $X$: **A(*X*)**, which consists of two disjoint sets,
  - **S(*X*)**: **Synthesized attributes**, used to pass semantic information up a parse tree (bottom up)
  - **I(*X*)**: **Inherited attributes**, used to pass semantic information down and across a tree (top down)
- Functions of the form $\boldsymbol{S(X_0)} = f(A(X_1), \ldots, A(X_n))$ define *synthesized attributes*
- Functions of the form $\boldsymbol{I(X_j)} = f(A(X_0), \ldots, A(X_{j-1}))$, *for* $1 \le j \le n$, define *inherited attributes*
- Initially, there are ***intrinsic attributes*** on the leaves
- **Predicate function**: a Boolean expression on the union of attribute sets $\{A(X_0), \ldots, A(X_n)\}$ and a set of literal attribute values

# Attribute Grammar: Example 2

- BNF syntax

  <assign> → <var> = <expr>

  <expr> → <var> + <var> | <var>

  <var> → A | B | C

- actual_type: a synthesized attribute for <var> and <expr>
  - store the actual type (int or real) of a variable or expression.
    - ✓ <var>: the actual type is intrinsic attribute value.
    - ✓ <expr>: values is determined from the actual types of its child nodes.
- expected_type: an inherited attribute for <expr>
  - store the type (int or real) that is expected for the <expr>, as determined by the type of <var> on the left side of <assign>.

# Attribute Grammar: Example 2 (cont.)

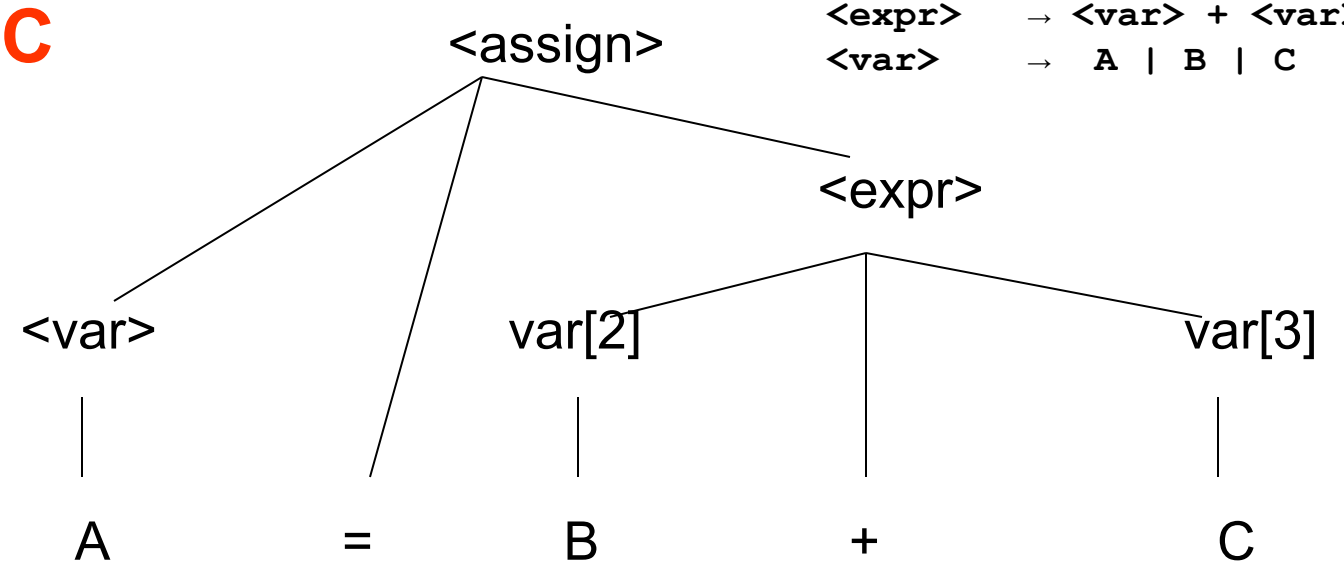
1. Syntax rule: <assign> → <var> = <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule:
   <expr>.actual_type ← if ( <var>[2].actual_type = int) and
   <var>[3].actual_type = int)
   then int
   else real
   end if
   Predicate: <expr>.actual_type = <expr>.expected_type
3. Syntax rule: <expr> → <var>
   Semantic rule:
   <expr>.actual_type ← <var>.actual_type
   Predicate: <expr>.actual_type = <expr>.expected_type
4. Syntax rule: <var> → A | B | C
   Semantic rule:
   <var>.actual_type ← lookup(<var>.string)

**Note**: Lookup function looks up a given variable name in the symbol table and returns the variable's type

# Parse Trees for Attribute Grammars

**A = B + C**

```
<assign> → <var> = <expr>
<expr>   → <var> + <var> | <var>
<var>    →  A | B | C
```




**How are attribute values computed?**

1. If all attributes were inherited, the tree could be decorated in top-down order.
2. If all attributes were synthesized, the tree could be decorated in bottom-up order.
3. If both kinds of attributes are present, some combination of top-down and bottom-up must be used.

# Parse Trees for Attribute Grammars (cont.)

**A = B + C**

1. <assign> → <var> = <expr>
   <expr>.expected_type ← <var>.actual_type

2. <expr> → <var>[2] + <var>[3]
   <expr>.actual_type ←
   if ( <var>[2].actual_type = int)
   and <var>[3].actual_type = int)
   then int
   else real
   end if
   Predicate: <expr>.actual_type = <expr>.expected_type

3. <expr> → <var>
   <expr>.actual_type ← <var>.actual_type
   Predicate: <expr>.actual_type = <expr>.expected_type

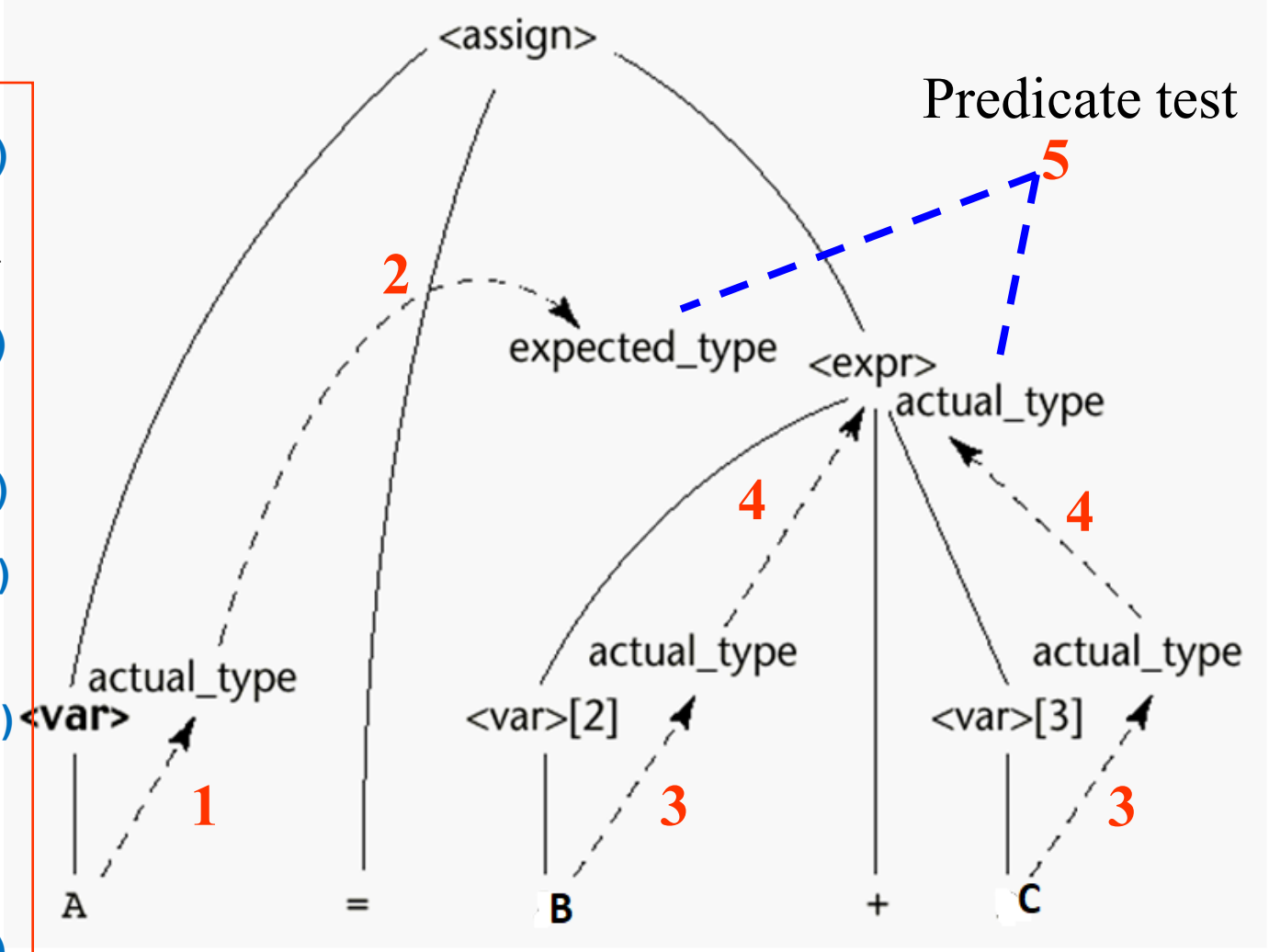
4. <var> → A | B | C
   <var>.actual_type ← lookup(<var>.string)

- <var>.actual_type ← lookup(A) (Rule 4)
- <expr>.expected_type ← <var>.actual_type (Rule 1)
- <var>[2].actual_type ← lookup(B) (Rule 4)
  <var>[3].actual_type ← lookup(C) (Rule 4)
- <expr>.actual_type ← either int or real (Rule 2)
- <expr>.expected_type = <expr>.actual_type is either TRUE or FALSE (Rule 2)

# Parse Tree for Attribute Grammars

**A = B + C**

1. <var>.actual_type ← lookup(A) **(Rule 4)**
2. <expr>.expected_type ← <var>.actual_type **(Rule 1)**
3. <var>[2].actual_type ← lookup(B) **(Rule 4)**
   <var>[3].actual_type ← lookup(C) **(Rule 4)**
4. <expr>.actual_type ← either int or real **(Rule 2)**
5. <expr>.expected_type = <expr>.actual_type is either TRUE or FALSE **(Rule 2)**



# Attribute Grammar Implementation

- Determining attribute evaluation order is a complex problem, requiring the construction of a dependency graph to show all attribute dependencies

- Difficulties in implementation
  - The large number of attributes and semantic rules required make such grammars difficult to write and read
  - Attribute values for large parse trees are costly to evaluate

- Less formal attribute grammars are used by compiler writers to check static semantic rules

## Describing the Meanings of Programs: Dynamic Semantics

<for_stmt> → **for** (<expr1>; <expr2>; <expr3>)

<assign_stmt> → <var> = <expr>;

What is the meaning of each statement?

- dynamic semantics

How do we **formally describe the dynamic semantics?**

# Dynamic Semantics

- There is no single widely acceptable notation or formalism for describing dynamic semantics
- Several needs for a methodology and notation for semantics:
  - Programmers need to know what statements mean
  - Compiler writers must know exactly what language constructs do
  - Correctness proofs would be possible
  - Compiler generators would be possible
  - Designers could detect ambiguities and inconsistencies
- Three formal methods to describe Dynamic Semantics:
  - Operational Semantics
  - Denotational Semantics
  - Axiomatic Semantics

# Operational Semantics

- The idea behind operational semantics is to describe the meaning of a statement or program by specifying the effects of running it on a machine.
- The effects on the machine are viewed as the sequence of changes in its state, where the machine's state is the collection of the values in its storage (memory, registers, etc.).
- An obvious operational semantics description, then, is given by executing a compiled version of the program on a computer.

# Operational Semantics

- Problems
  - The individual steps in the execution of machine language and the resulting changes to the state of the machine are too small and too numerous.
  - The storage of a real computer is too large and complex.




Initial State: (i1,v1), (i2,v2), …

Final State: (i1,v1'), (i2,v2'), …

- Solution: design an intermediate language and a virtual machine with purpose of clarity

# Operational Semantics (cont.)

An example:

*C Statement*

```
for (expr1; expr2; expr3) {
  ...
}
```

*Meaning*

```
        expr1;
loop:   if expr2 == 0 goto out
        ...
        expr3;
        goto loop
out:    ...
```

Human reader is the virtual computer being able to "execute" the instructions in the definition.

- Uses of operational semantics:
  - Language manuals and textbooks
  - Teaching programming languages
- Two different levels of uses of operational semantics:
  - Natural operational semantics
  - Structural operational semantics
- Evaluation
  - Good if used informally (language manuals, etc.)
  - Extremely complex if used formally (e.g., VDL)

# Denotational Semantics

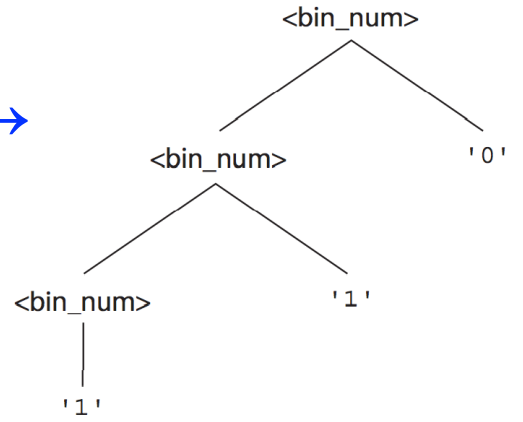
- Denotational semantics is the most rigorous and most widely known formal method for describing the meaning of programs.
- It is solidly based on **recursive function theory**.
- The process of building a denotational specification for a language:
  - Define a mathematical object for each language entity
  - Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects
- The meaning of language constructs are defined by the values of the program's variables and how they change over time.

# Denotational Semantics: Example 1

## Example: Character string representations of binary numbers

<bin_num> → '0'
| '1'
| <bin_num> '0'
| <bin_num> '1'

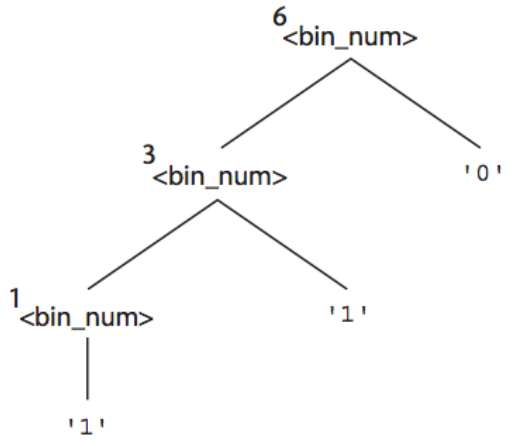
"110" →




Mapping Function, $\mathbf{M_{bin}}$:

- ✓ **Syntax domain**: character string representations of binary numbers
- ✓ **Semantic domain**: N (non-negative decimal number)

$M_{bin}$('0') = 0
$M_{bin}$('1') = 1
$M_{bin}$(<bin_num> '0') = 2 * $M_{bin}$(<bin_num>)
$M_{bin}$(<bin_num> '1') = 2 * $M_{bin}$(<bin_num>) + 1



# Denotational Semantics: Example 2

- Decimal Numbers
  - The following denotational semantics description maps decimal numbers as strings of symbols into numeric values
  - Syntax rule:

    <dec_num> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    | <dec_num> (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)
  - Denotational Semantics:

    $M_{dec}$ ('0') = 0, $M_{dec}$ ('1') = 1, …, $M_{dec}$ ('9') = 9

    $M_{dec}$ (<dec_num> '0') = 10 * $M_{dec}$ (<dec_num>)

    $M_{dec}$ (<dec_num> '1') = 10 * $M_{dec}$ (<dec_num>) + 1

    $M_{dec}$ (<dec_num> '2') = 10 * $M_{dec}$ (<dec_num>) + 2

    …

    $M_{dec}$ (<dec_num> '9') = 10 * $M_{dec}$ (<dec_num>) + 9

# The State of a Program

- The state of a program is the values of all its current variables

$$s = \{\langle i_1, v_1\rangle, \langle i_2, v_2\rangle, \ldots, \langle i_n, v_n\rangle\}$$

- Let **VARMAP** be a function that, when given a variable name and a state, returns the value of the variable in the state

$$\text{VARMAP}(i_j, s) = v_j$$

# Expressions

- Map expressions onto semantic domain: Z  U {error}
- We assume expressions are decimal numbers, variables, or binary expressions having **one** arithmetic operator (+ or *) and two operands, each of which can be a decimal number or variable
- BNF description of Expressions:

<expr> → <dec_num> | <var> | <binary_expr>
<binary_expr> → <left_expr> <operator> <right_expr>
<left_expr> → <dec_num> | <var>
<right_expr> → <dec_num> | <var>
<operator> → + | *

**Note**: the only error we consider is a variable having an undefined value.

# Expressions (cont.)

```
Me(<expr>, s) Δ= case <expr> of
      <dec_num> => Mdec(<dec_num>, s)
      <var> => if VARMAP(<var>, s) == undef
                  then error
                  else VARMAP(<var>, s)
      <binary_expr> =>
          if (Me(<binary_expr>.<left_expr>, s) == undef
              OR Me(<binary_expr>.<right_expr>, s) == undef)
          then error
          else if (<binary_expr>.<operator> == '+'
                  then Me(<binary_expr>.<left_expr>, s) +
                       Me(<binary_expr>.<right_expr>, s)
                  else Me(<binary_expr>.<left_expr>, s) *
                  Me(<binary_expr>.<right_expr>, s)
       ...
```

**Note**: Δ=: define mathematical functions

# Assignment Statements

- An assignment statement: an expression evaluation plus the setting of the target variable to the expression's value
- Mapping function: state sets to state sets ∪ {error}

$$M_a(x = E, s) \Delta= \text{if } M_e(E, s) == \text{error}$$
$$\text{then error}$$
$$\text{else } s' = \{<i_1, v_1'>, <i_2, v_2'>, \ldots, <i_n, v_n'>\},$$
$$\text{where for } j = 1, 2, \ldots, n,$$
$$v_j' = \text{VARMAP}(i_j, s) \text{ if } i_j <> x$$
$$= M_e(E, s) \text{ if } i_j == x$$

# Logical Pretest Loops

- $M_l$: map logical loop and state sets to state sets ∪ {error}
- Assume two existing mapping functions:
  - ✓ $\mathbf{M_{sl}}$: map statement list and states to states
  - ✓ $\mathbf{M_b}$: map Boolean expressions to Boolean values (or error)

```
Ml(while B do L, s) Δ=
     if Mb(B, s) == undef
        then error
        else if Mb(B, s) == false
                then s
                else if Msl(L, s) == error
                     then error
                     else Ml(while B do L, Msl(L, s))
```

# Loop Meaning

- The meaning of the loop is the value of the program variables after the statements in the loop have been executed the prescribed number of  times, assuming there have been no errors
- Essentially, the loop has been converted from iteration to recursion, where the recursive control  is mathematically defined by other recursive state mapping functions
- Recursion, when compared to iteration, is easier to describe with mathematical rigor

# Evaluation of Denotational Semantics

- Provides a rigorous way to think about programs
- Can be an aid to language design
- Has been used in compiler generation systems
- Can be used to prove the correctness of programs
- Because of its complexity, they are of little use to language users

# Axiomatic Semantics

- Based on formal logic (predicate calculus)
- Original purpose: formal program verification
- Axioms or inference rules are defined for each statement type in the language
- The logic expressions are called **assertions**
- An assertion before a statement (a **precondition**) states the relationships and constraints among variables that are true at that point in execution
- An assertion following a statement is a **postcondition**
- A **weakest precondition** is the least restrictive precondition that will guarantee the postcondition

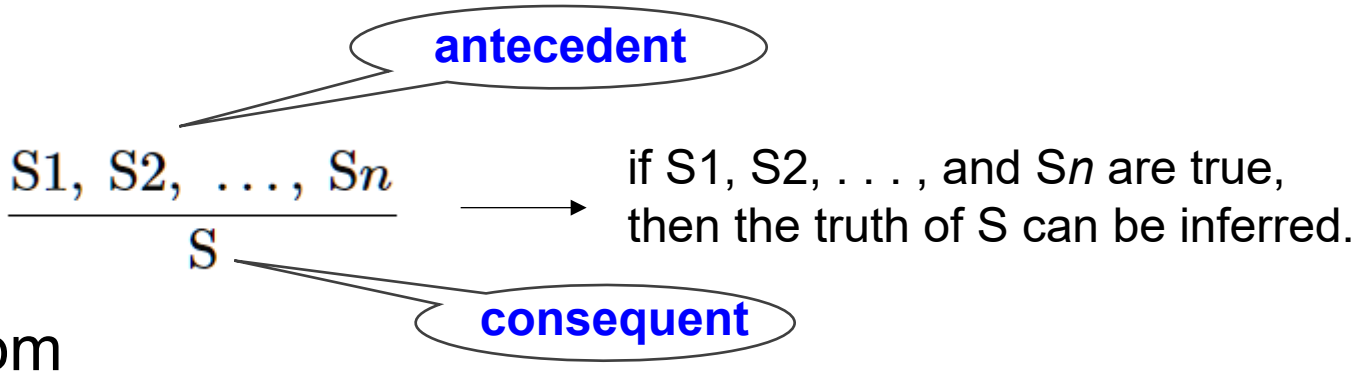
# Axiomatic Semantics Form

- **{P} statement {Q}**, P: Precondition; Q: Postcondition
- For example, a = b + 1 {a > 1}
  - ✓ One possible precondition: {b > 10}
  - ✓ Weakest precondition: {b > 0}

- Program Proof Process
  - ✓ The postcondition for the entire program is the desired result
  - ✓ Work back through the program to the first statement. If the precondition on the first statement is the same as the program specification, the program is correct.

# Inference Rules

- Inference Rule
  - ✓ A method of inferring the truth of one assertion on the basis of the values of other assertions.
  - ✓ General form of an inference rule,




- Axiom
  - ✓ A logical statement that is assumed to be true.
  - ✓ Therefore, an axiom is an inference rule without an antecedent.
- To use axiomatic semantics with a given programming language, either an axiom or an inference rule must exist for each kind of statement in the language.

# Axiomatic Semantics: Assignments

- An axiom for assignment statements
  (x = E): $\{Q_{x->E}\}$ x = E {Q}

  e.g., a = b / 2 – 1 {a < 10} → {b < 22} a = b / 2 – 1 {a < 10}

- The Rule of Consequence:

$$\frac{\{P\}\,S\,\{Q\},\ P' \Rightarrow P,\ Q \Rightarrow Q'}{\{P'\}\,S\,\{Q'\}}$$

e.g., $$\frac{\{x > 3\}\,x = x - 3\,\{x > 0\},\ (x > 5) => \{x > 3\},\ (x > 0) => (x > 0)}{\{x > 5\}\,x = x - 3\,\{x > 0\}}$$

- An inference rule for sequences of the form S1; S2

$$\frac{\{P1\}\ \ S1\ \{P2\},\ \ \{P2\}\ \ S2\ \{P3\}}{\{P1\}\ \ S1,\ S2\ \ \{P3\}}$$

e.g., {x < 2} y = 3 * x + 1; x = y + 3 {x < 10}

# Axiomatic Semantics: Selection

- An inference rules for selection: **if** B **then** S1 **else** S2

$$\frac{\{B \text{ and } P\}\ S1\ \{Q\},\ \{(\text{not } B) \text{ and } P\}\ S2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q\}}$$

- Example: **if** x > 0 **then** y = y - 1 **else** y = y + 1

  Postcondition **Q**: {y > 0}

  **then** clause: {y > 1} y = y - 1 {y > 0}

  **else** clause: {y > -1} y = y + 1 {y > 0}

  {y > 1} => {y > -1}

{y > 1} **if** x > 0 **then** y = y - 1 **else** y = y + 1 {y > 0}

# Evaluation of Axiomatic Semantics

- Creating axioms or inference rules for every statement in a language is challenging.
- While it is an effective tool for correctness proofs and provides an excellent framework for reasoning about programs, it is less beneficial for language users and compiler developers.
- Its ability to describe the meaning of a programming language is limited when it comes to language users or compiler writers.

# Summary

- BNF and context-free grammars are equivalent meta-languages
  - Well-suited for describing the syntax of programming languages
- An attribute grammar is a descriptive formalism that can describe both the syntax and the semantics of a language
- Three primary methods of semantics description
  - Operational, denotational, axiomatic



HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

# CS153: Compilers
# Lecture 6: LR Parsing

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

# Announcements

- Proj 1 out
  - Due today Thursday Sept 20, 11.59pm
- Proj 2 out
  - Due Thursday Oct 4 (14 days away)

# Today

- LR Parsing
  - Constructing a DFA and LR parsing table
  - Using Yacc

# $\mathrm{LR}(k)$

**L**eft-to-right parse

**R**ightmost derivation

Derivation expands the rightmost non-terminal

(Constructs derivation in reverse order!)

***k***-symbol lookahead

# LR($k$)

- Basic idea: LR parser has a stack and input
  - Given contents of stack and $k$ tokens look-ahead parser does one of following operations:
    - Shift: move first input token to top of stack
    - Reduce: top of stack matches rule, e.g., $X \rightarrow A\ B\ C$
      - Pop $C$, pop $B$, pop $A$, and push $X$

# Example

$E \rightarrow$ `int`

$E \rightarrow$ `(` $E$ `)`

$E \rightarrow E$ `+` $E$

**Stack**

**Input**

`(3+4)+(5+6)`

Shift `(` on to stack

# Example

$E \rightarrow$ `int`
$E \rightarrow$ `(` $E$ `)`
$E \rightarrow E$ `+` $E$

| **Stack** | **Input** |
|---|---|
| `(` | `3+4)+(5+6)` |

Shift `(` on to stack
Shift `3` on to stack

# Example

$E \rightarrow$ `int`
$E \rightarrow$ `(` $E$ `)`
$E \rightarrow E$ `+` $E$

| **Stack** | **Input** |
|---|---|
| `(3` | `+4)+(5+6)` |

Shift `(` on to stack
Shift `3` on to stack
Reduce using rule $E \rightarrow$ `int`

# Example

$E \rightarrow$ `int`
$E \rightarrow$ `(` $E$ `)`
$E \rightarrow E$ `+` $E$

| **Stack** | **Input** |
|---|---|
| `(` $E$ | `+4)+(5+6)` |

Shift `(` on to stack
Shift `3` on to stack
Reduce using rule $E \rightarrow$ `int`
Shift `+` on to stack

# Example

$E \rightarrow$ `int`
$E \rightarrow$ `(` $E$ `)`
$E \rightarrow E$ `+` $E$

| **Stack** | **Input** |
|---|---|
| `(` $E$ `+` | `4)+(5+6)` |

Shift `(` on to stack
Shift `3` on to stack
Reduce using rule $E \rightarrow$ `int`
Shift `+` on to stack
Shift `4` on to stack

# Example

$E \rightarrow$ `int`

$E \rightarrow$ `(` $E$ `)`

$E \rightarrow E$ `+` $E$

| **Stack** | **Input** |
|---|---|
| `(` $E$ `+4` | `)+(5+6)` |

Shift `(` on to stack
Shift `3` on to stack
Reduce using rule $E \rightarrow$ `int`
Shift `+` on to stack
Shift `4` on to stack
Reduce using rule $E \rightarrow$ `int`

# Example

$E \rightarrow$ `int`
$E \rightarrow$ `(` $E$ `)`
$E \rightarrow E$ `+` $E$

| **Stack** | **Input** |
|---|---|
| `(` $E$ `+` $E$ | `)+(5+6)` |

Shift `(` on to stack
Shift `3` on to stack
Reduce using rule $E \rightarrow$ `int`
Shift `+` on to stack
Shift `4` on to stack
Reduce using rule $E \rightarrow$ `int`
Reduce using rule $E \rightarrow E$ `+` $E$

# Example

$E \rightarrow$ `int`

$E \rightarrow$ `(` $E$ `)`

$E \rightarrow E$ `+` $E$

| **Stack** | **Input** |
|---|---|
| `(` $E$ | `)+(5+6)` |

Reduce using rule $E \rightarrow E$ `+` $E$

Shift `)` on to stack

# Example

$E \rightarrow \texttt{int}$

$E \rightarrow \texttt{(}E\texttt{)}$

$E \rightarrow E + E$

| Stack | Input |
|---|---|
| ( $E$ ) | + ( 5 + 6 ) |

Reduce using rule $E \rightarrow E + E$

Shift ) on to stack

Reduce using rule $E \rightarrow$ ( $E$ )

# Example

$E \rightarrow$ `int`

$E \rightarrow$ `(` $E$ `)`

$E \rightarrow E$ `+` $E$

| **Stack** | **Input** |
|---|---|
| $E$ | `+(5+6)` |

Reduce using rule $E \rightarrow E$ `+` $E$

Shift `)` on to stack

Reduce using rule $E \rightarrow$ `(` $E$ `)`

Shift + on to stack

# Example

$E \rightarrow$ `int`

$E \rightarrow$ `(` $E$ `)`

$E \rightarrow E$ `+` $E$

| **Stack** | **Input** |
|---|---|
| $E$ `+` | `(5+6)` |

Reduce using rule $E \rightarrow E + E$

Shift `)` on to stack

Reduce using rule $E \rightarrow$ `(` $E$ `)`

Shift + on to stack

... and so on ...

# Example

$E \rightarrow$ `int`

$E \rightarrow$ `(` $E$ `)`

$E \rightarrow E$ `+` $E$

| **Stack** | **Input** |
|---|---|
| $E$ `+` `(` $E$ | `+6)` |

Reduce using rule $E \rightarrow E$ `+` $E$

Shift `)` on to stack

Reduce using rule $E \rightarrow$ `(` $E$ `)`

Shift + on to stack

... and so on ...

# Example

$E \rightarrow$ `int`

$E \rightarrow$ `(` $E$ `)`

$E \rightarrow E + E$

| **Stack** | **Input** |
|---|---|
| $E + ($ $E + E$ | ) |

Reduce using rule $E \rightarrow E + E$

Shift ) on to stack

Reduce using rule $E \rightarrow ($ $E$ )

Shift + on to stack

... and so on ...

# Example

$E \rightarrow$ `int`

$E \rightarrow$ `(` $E$ `)`

$E \rightarrow E$ `+` $E$

| **Stack** | **Input** |
|---|---|
| $E$ `+` `(` $E$ | `)` |

Reduce using rule $E \rightarrow E$ `+` $E$

Shift `)` on to stack

Reduce using rule $E \rightarrow$ `(` $E$ `)`

Shift + on to stack

... and so on ...

# Example

$E \rightarrow \texttt{int}$

$E \rightarrow \texttt{(}E\texttt{)}$

$E \rightarrow E + E$

| **Stack** | **Input** |
|---|---|
| $E + E$ | |

Reduce using rule $E \rightarrow E + E$

Shift ) on to stack

Reduce using rule $E \rightarrow$ ( $E$ )

Shift + on to stack

... and so on ...

# Example

$E \rightarrow \texttt{int}$

$E \rightarrow \texttt{(}E\texttt{)}$

$E \rightarrow E + E$

**Stack** **Input**

$E$

Reduce using rule $E \rightarrow E + E$

Shift ) on to stack

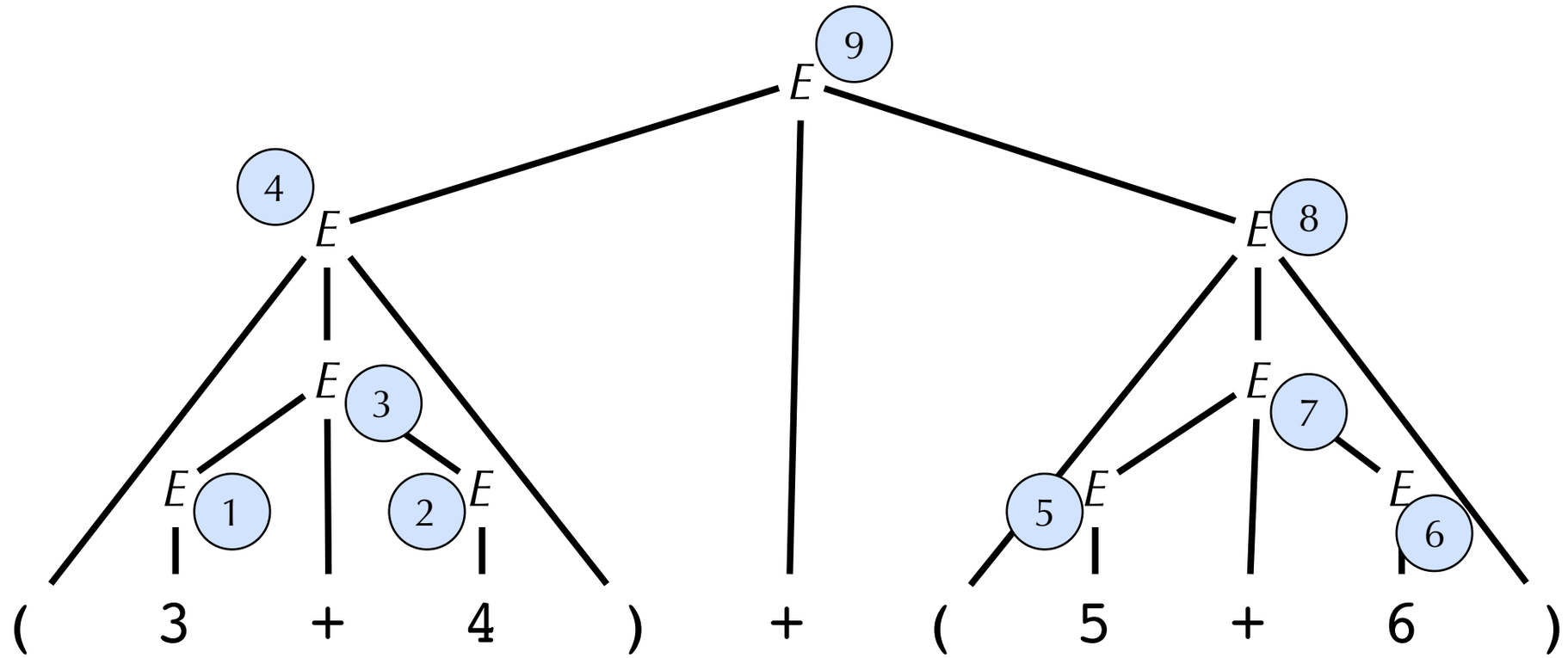
Reduce using rule $E \rightarrow$ ( $E$ )

Shift + on to stack

... and so on ...

# Rightmost derivation

- LR parsers produce a rightmost derivation




- But do reductions in reverse order

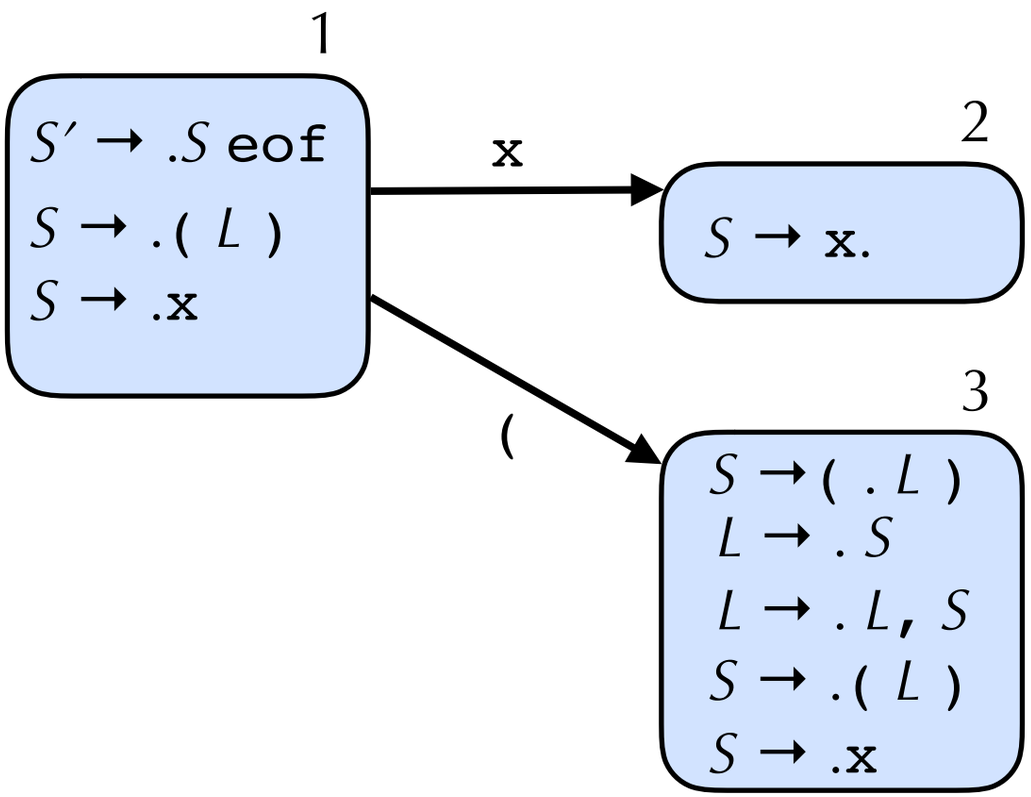
# What Action to Take?

- How does the LR(k) parser know when to shift and to reduce?
- Uses a DFA
- At each step, parser runs DFA using symbols on stack as input
  - Input is sequence of terminals and non-terminals from bottom to top
- Current state of DFA plus next *k* tokens indicate whether to shift or reduce

# Building the DFA for LR parsing

- Sketch only. For details, see Appel
- States of DFA are sets of **items**
  - An **item** is a production with an indication of current position of parser
  - E.g., Item $E \rightarrow E$ . + $E$ means that for production $E \rightarrow E + E$, we have parsed first expression $E$ have yet to parse + token
  - In general item $X \rightarrow \gamma$ . $\delta$ means $\gamma$ is at the top of the stack, and at the head of the input there is a string derivable from $\delta$

# Example: LR(0)




Add new start symbol with production to indicate end-of-file

$S' \rightarrow S$ `eof`

$S \rightarrow$ `(` $L$ `)`

$S \rightarrow$ `x`

$L \rightarrow S$

$L \rightarrow L$ `,` $S$

First item of first state: at the start of input

State 1: item is about to parse *S*: add productions for *S*

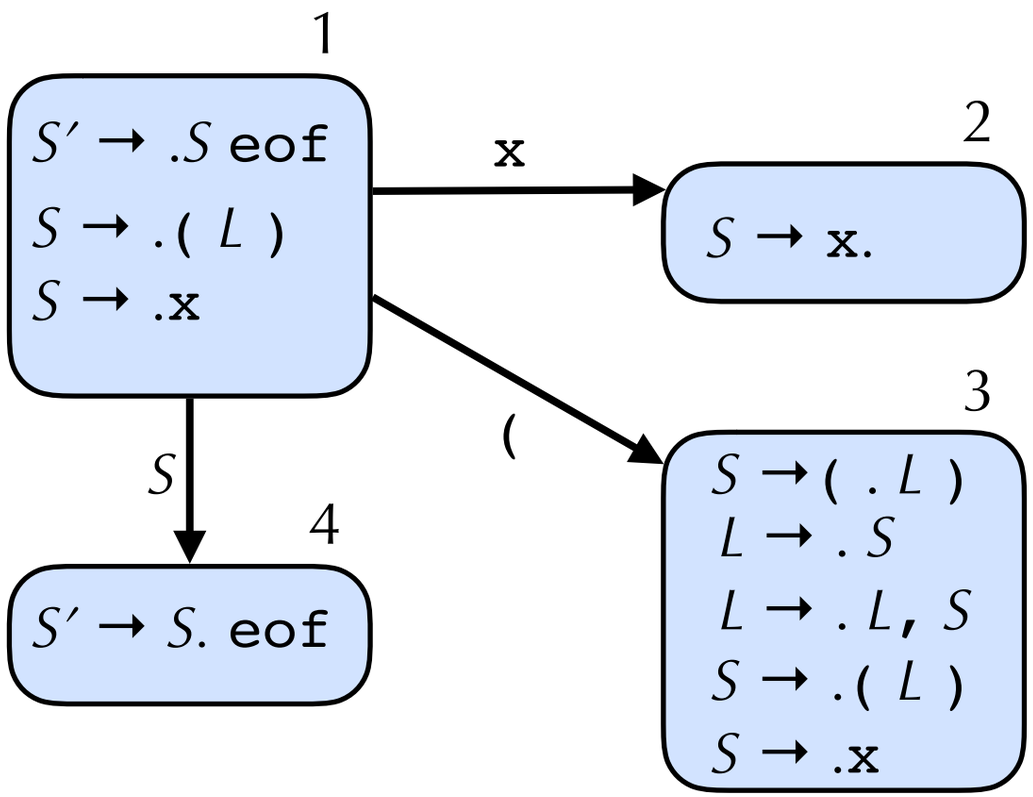
From state 1, can take `x`, moving us to state 2

From state 1, can take `(`, moving us to state 3

State 3: item is about to parse *L*: add productions for *L*

State 3: item is about to parse *S*: add productions for *S*

# Example: LR(0)




$S' \rightarrow S$ `eof`

$S \rightarrow$ `(` $L$ `)`

$S \rightarrow$ `x`

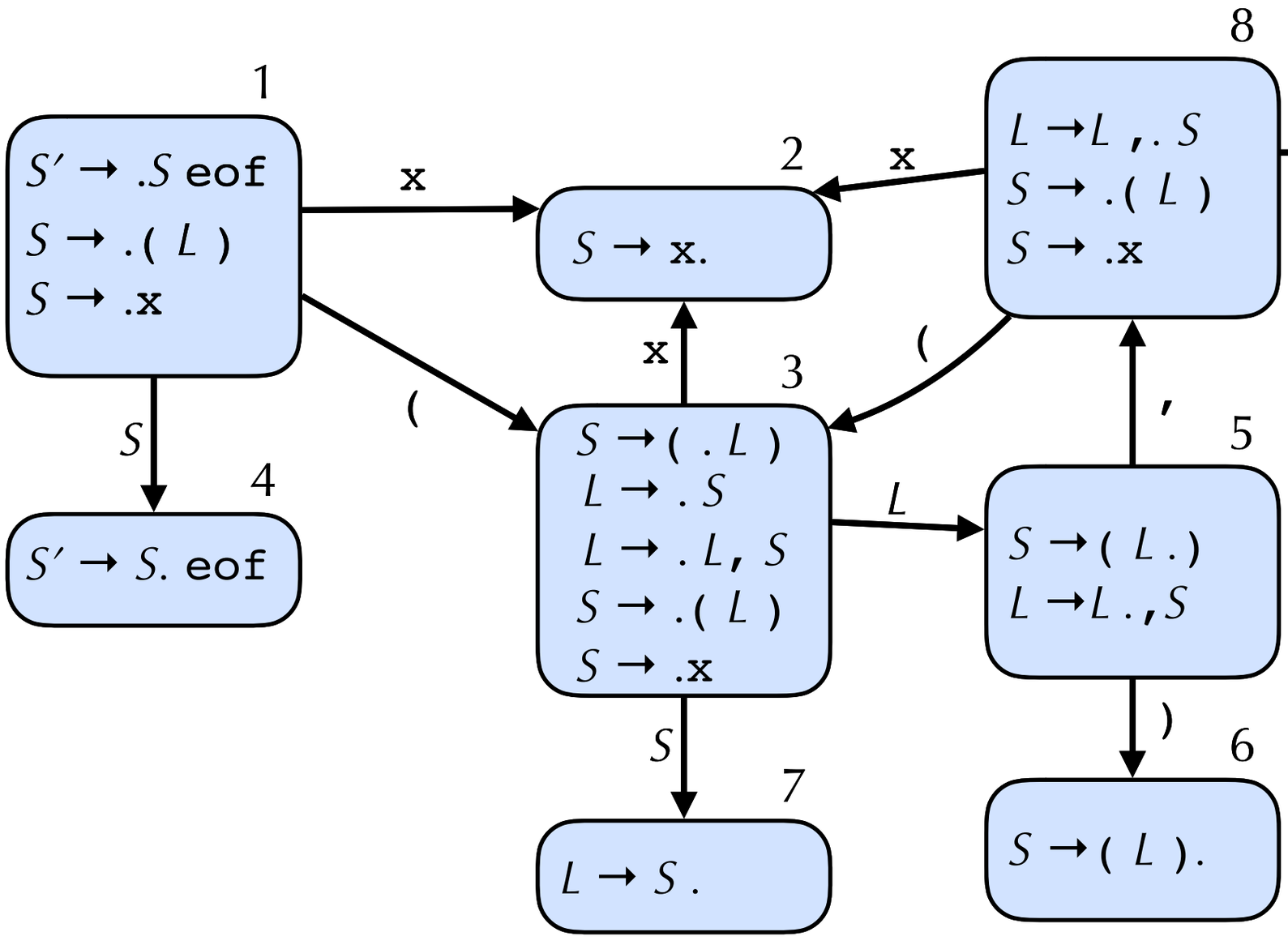
$L \rightarrow S$

$L \rightarrow L$ `,` $S$

State 1: can take *S*, moving us to state 4

State 4 is an accepting state (if at end of input)

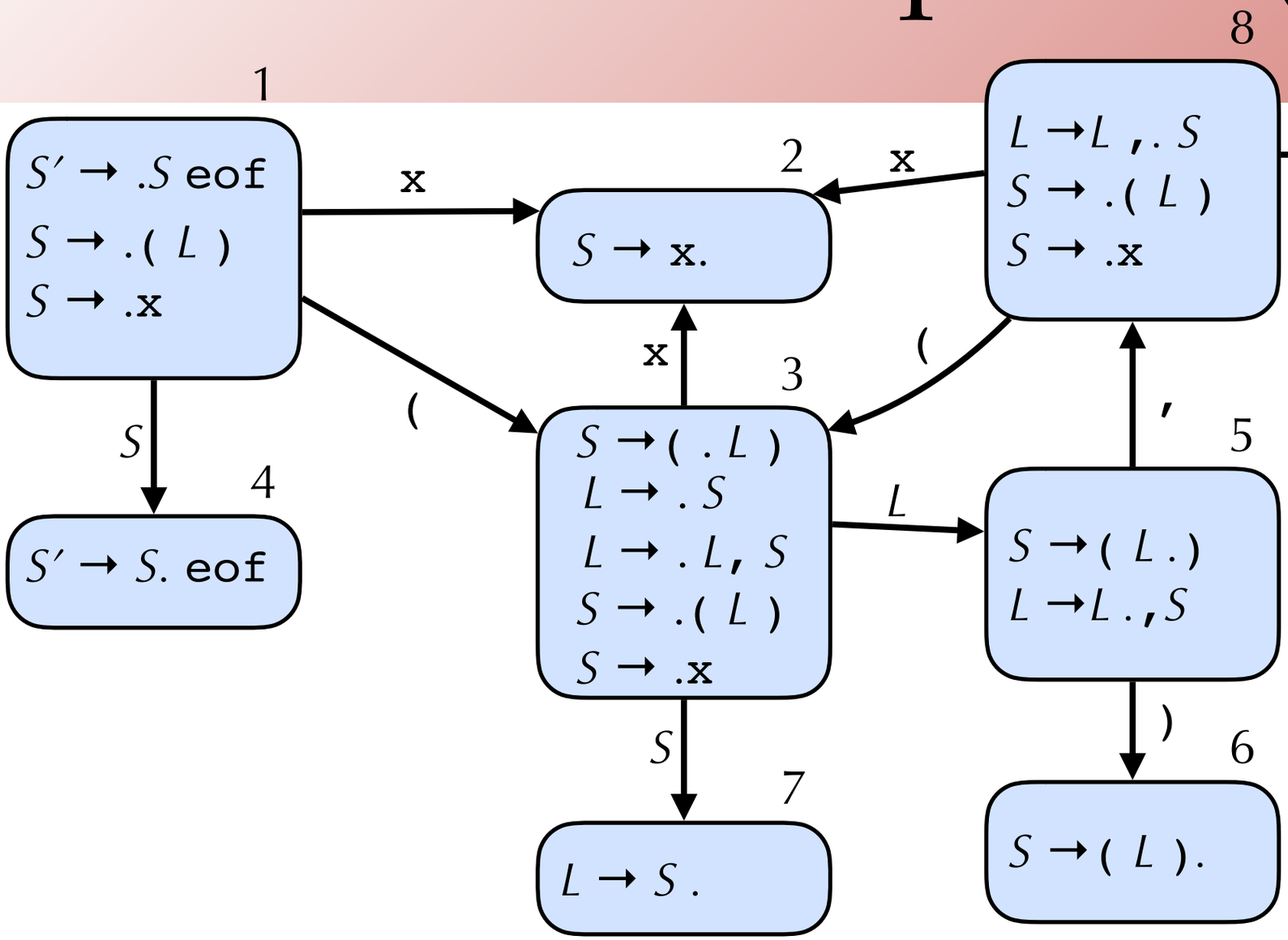
# Example: LR(0)




$S' \rightarrow S$ eof
$S \rightarrow$ ( $L$ )
$S \rightarrow$ x
$L \rightarrow S$
$L \rightarrow L$ , $S$

Continue to add states based on next symbol in item

# Example LR(0)




| State | Action |
|---|---|
| 1 | shift |
| 2 | reduce $S \rightarrow$ `x` |
| 3 | shift |
| 4 | accept |
| 5 | shift |
| 6 | reduce $S \rightarrow$ `(` $L$ `)` |
| 7 | reduce $L \rightarrow S$ |
| 8 | shift |
| 9 | reduce $L \rightarrow L$ `,` $S$ |

- Build action table
- If state contains item $X \rightarrow \gamma$`.eof` then **accept**
- If state contains item $X \rightarrow \gamma$. then **reduce** $X \rightarrow \gamma$
- If state $i$ has edge to $j$ with terminal then **shift**

# LR(1)

- In practice, LR(1) is used for LR parsing
  - not LR(0) or LR(*k*) for *k*>1

- Item is now pair ($X \rightarrow \gamma \cdot \delta$, *x*)
  - Indicates that $\gamma$ is at the top of the stack, and at the head of the input there is a string derivable from $\delta x$ (where *x* is terminal)
  - Algorithm for constructing state transition table and action table adapted. See Appel for details.
    - Closure operation when constructing states uses FIRST(), incorporating lookahead token
    - Action table columns now terminals (i.e., 1-token lookahead)
    - Note: state transition relation and action table typically combined into single table, called **parsing table**

# LR Parsing and Ambiguous Grammars

- If grammar is ambiguous, we can't construct a DFA!
- We get **conflicts**: don't know which action to take
  - Shift-reduce conflicts: don't know whether to shift or reduce
  - Reduce-reduce conflicts: don't know which production to use to reduce

# Dangling Else Problem

- Many language have productions such as

  $S \rightarrow$ `if` $E$ `then` $S$ `else` $S$

  $S \rightarrow$ `if` $E$ `then` $S$

  $S \rightarrow$ ...

- Program `if a then if b then s1 else s2` could be

  either `if a then { if b then s1 } else s2`

  or `if a then {if b then s1 else s2 }`

- In LR parsing table there will be a shift-reduce conflict
  - $S \rightarrow$ `if` $E$ `then` $S$ . with lookahead `else`: reduce
  - $S \rightarrow$ `if` $E$ `then` $S$ . `else` $S$ with any lookahead: shift
  - Which action corresponds to which interpretation of

    `if a then if b then s1 else s2` ?

# Resolving Ambiguity

- Could rewrite grammar to avoid ambiguity
  - E.g.,

$S \rightarrow O$

$O \rightarrow V$ `:=` $E$

$O \rightarrow$ `if` $E$ `then` $O$

$O \rightarrow$ `if` $E$ `then` $C$ `else` $O$

$C \rightarrow V$ `:=` $E$

$C \rightarrow$ `if` $E$ `then` $C$ `else` $C$

# Resolving Ambiguity

- Or tolerate conflicts, indicating how to resolve conflict
  - E.g., for dangling else, prefer shift to reduce.
    - i.e., for `if a then if b then s1 else s2`

      prefer `if a then {if b then s1 else s2 }`

      over `if a then { if b then s1 } else s2`
    - i.e., `else` binds to closest `if`
  - Expression grammars can express operator-precedence by resolution of conflicts
- Use sparingly! Only in well-understood cases
  - Most conflicts are indicative of ill-specified grammars

# Using Yacc

- **Y**et **A**nother **C**ompiler-**C**ompiler
- Originally developed in early 1970s
- Various versions/reimplimentations
  - Berkeley Yacc, Bison, Ocamlyacc, ...
- From a suitable grammar, constructs an LALR(1) parser
  - A kind of LR parser, not as powerful as LR(1)
  - Most practical LR(1) grammars will be LALR(1) grammars

# Ocamlyacc

- Usage: `ocamlyacc` *`options`* *`grammar`*`.mly`
- Produces output files
  - *`grammar`*`.ml`: OCaml code for a parser
  - *`grammar`*`.mli`: interface for parser

# Structure of ocamlyacc File

```
%{
  header
%}
  declarations
%%
  rules
%%
  trailer
```

- Header and trailer are arbitrary OCaml code, copied to the output file
- Declarations of tokens, start symbols, OCaml types of symbols, associativity and precedence of operators
- Rules are productions for non-terminals, with **semantic actions** (OCaml expressions that are executed with production is reduced, to produce value for symbol)

# Ocamlyacc example

- See `Lec03-parser-eg.mll`
  and output files `Lec03-parser-eg.ml`
  and `Lec03-parser-eg.mli`
- Typically, the `.mly` declares the tokens, and the lexer opens the parser module