Compilers and Interpreters

Why Interpretation

- ❖A higher degree of machine independence: high portability.
- ❖Dynamic execution: modification or addition to user programs as execution proceeds.
- ❖Dynamic data type: type of object may change at runtime
- ❖Easier to write – no synthesis part.
- ❖Better diagnostics: more source text information available

Why Study Compilers?

- Influences on programming language design
- Influences on computer design
- Compiling techniques are useful for software development
  - Parsing techniques are often used
  - Learn practical data structures and algorithms
  - Basis for many tools such as text formatters, structure editors, silicon compilers, design verification tools,…
- So you may write more efficient code
  - Writing a compiler requires an understanding of almost all important CS subfields

The Structure of a Compiler

Analysis

- **Lexical analysis** (Linear Analysis) : stream of characters are grouped into *tokens*
- **Syntax analysis** (Hierarchical Analysis): tokens are grouped hierarchically with collective meaning
- **Semantic Analysis**: ensure the components of a program fit together.

Synthesis

Lexical Analysis Example

Pay := Base + Rate* 60

- **Lexical analysis:**

  characters are grouped into seven tokens:

  Pay, Base, Rate are identifiers

  := is assignment symbol

  + and * are operators

  60 is a number

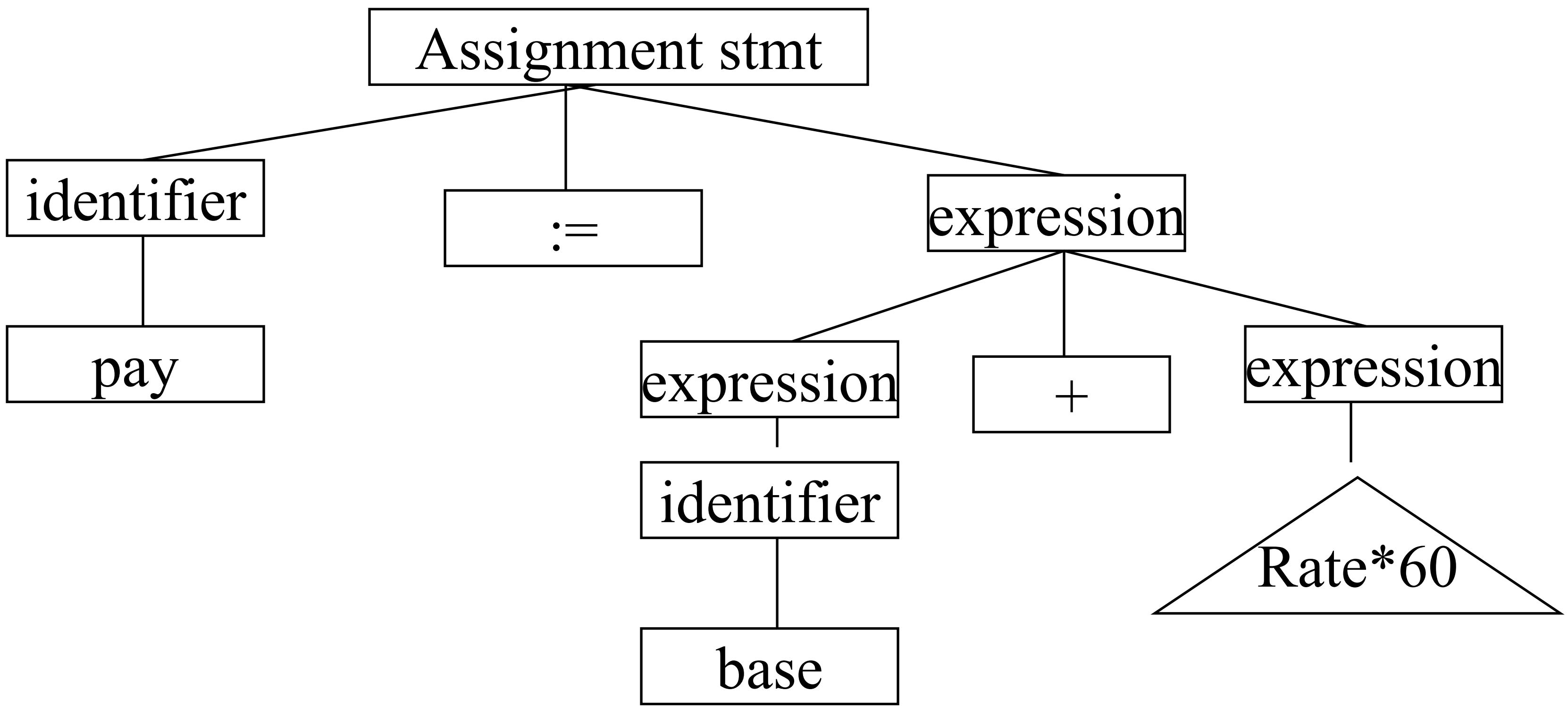
- Error example:

  pay := base + rate^^60

Syntax Analysis Example

Pay := Base + Rate* 60

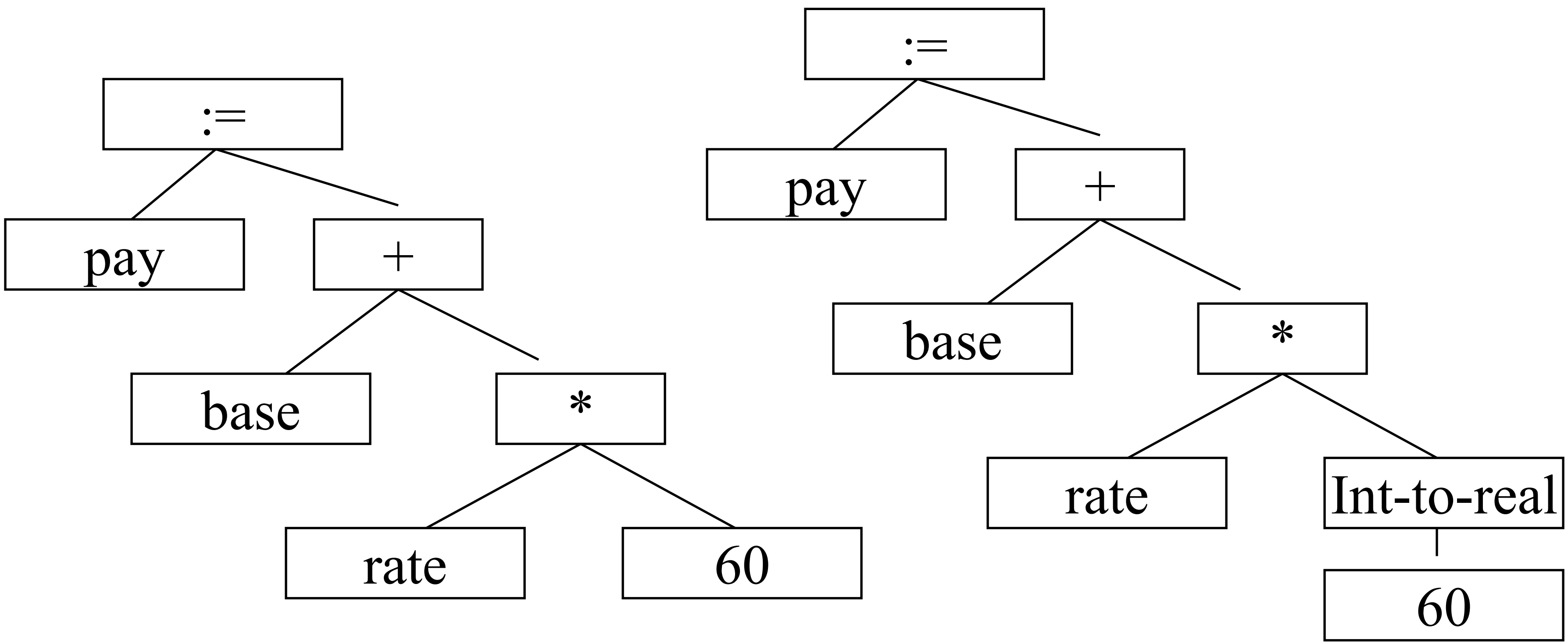
- The seven tokens are grouped into a parse tree

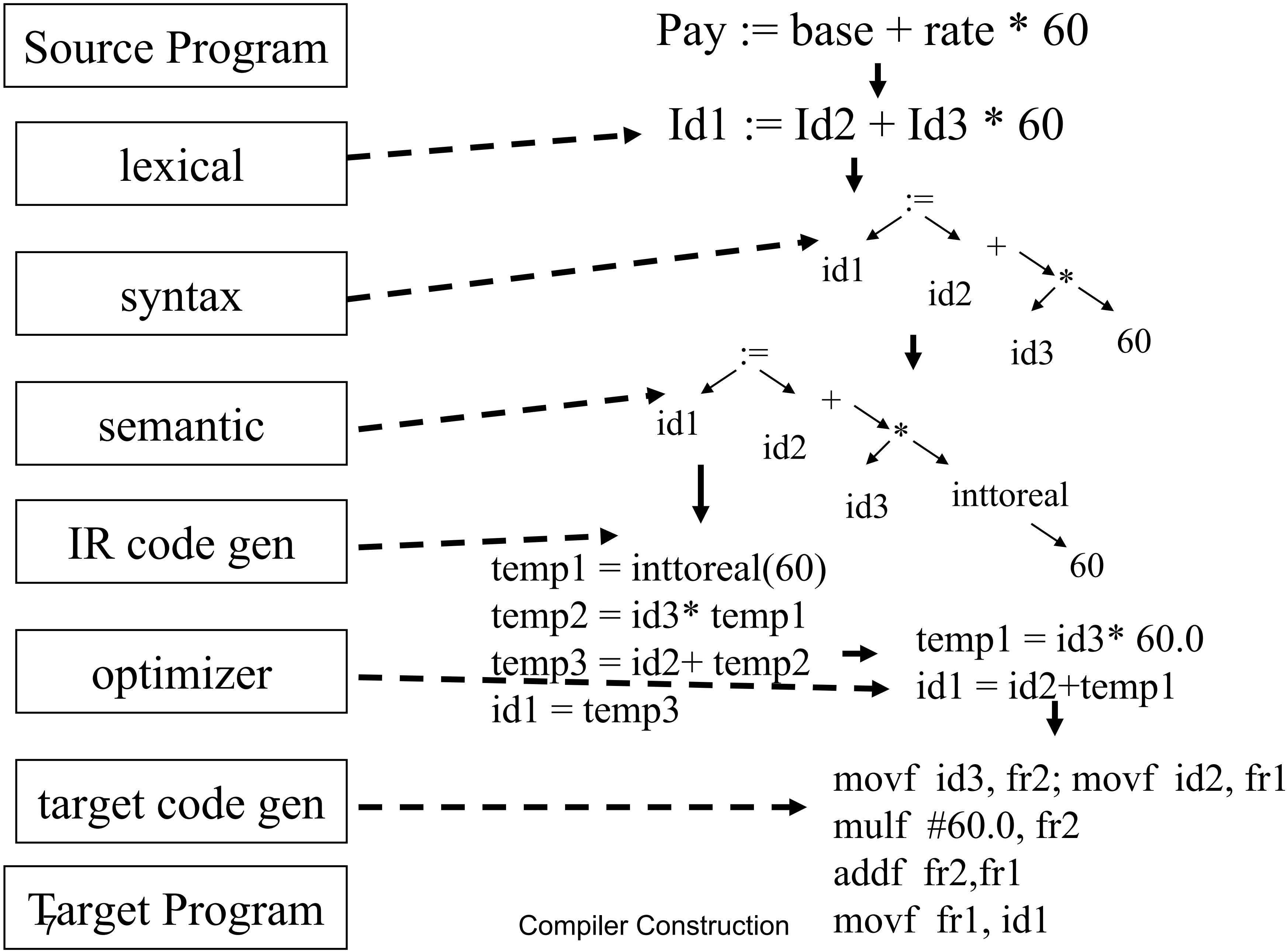


Semantic Analysis Example

Pay := Base + Rate* 60

- Checks for semantic errors and gathers type information for code generation.

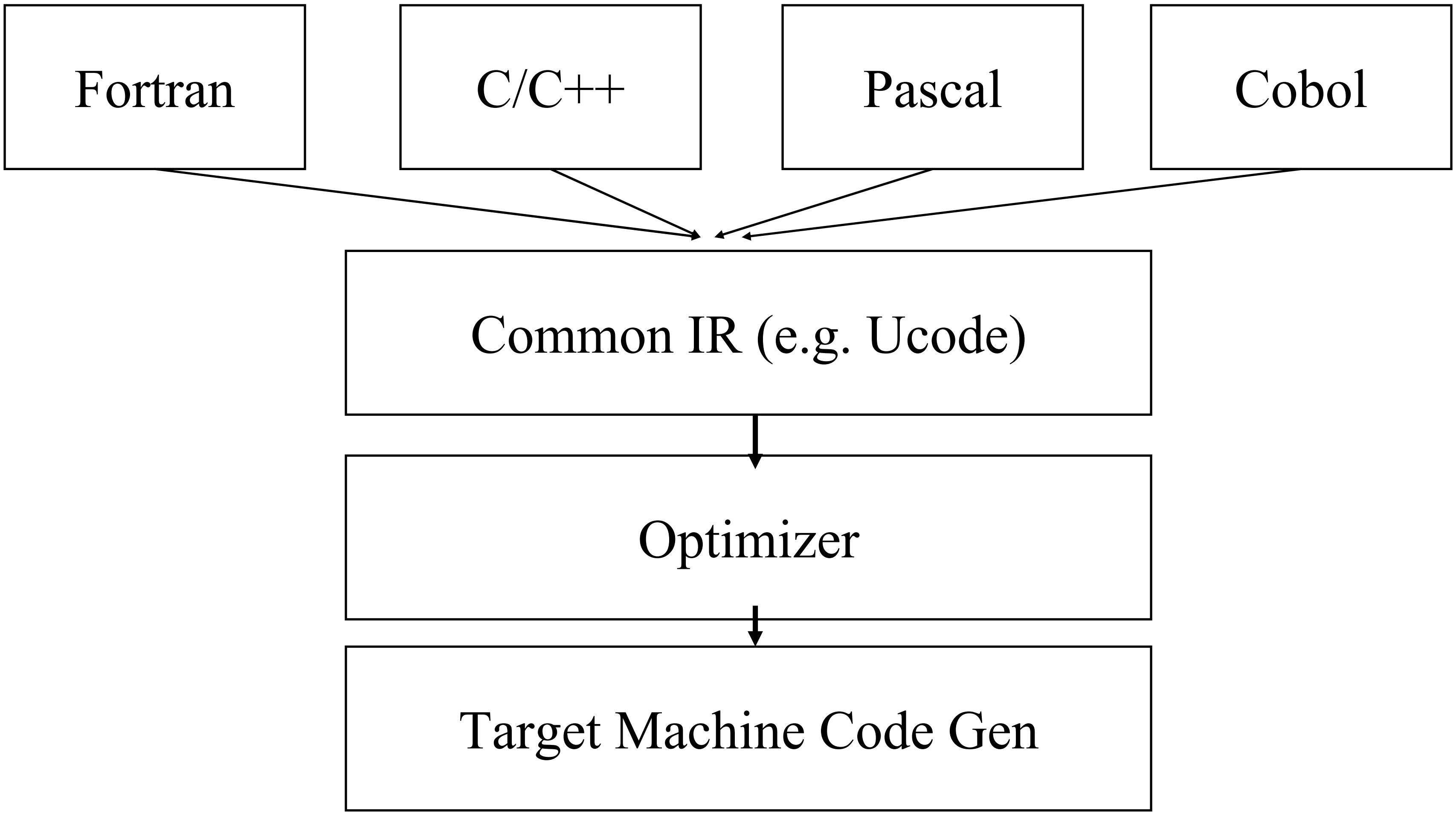



Source Program
lexical
syntax
semantic
IR code gen
optimizer
target code gen
Target Program
Pay := base + rate * 60
Id1 := Id2 + Id3 * 60
:=
id1
id2
+
*
id3
60
:=
id1
id2
+
*
id3
inttoreal
60
temp1 = inttoreal(60)
temp2 = id3* temp1
temp3 = id2+ temp2
id1 = temp3
temp1 = id3* 60.0
id1 = id2+temp1
movf id3, fr2; movf id2, fr1
mulf #60.0, fr2
addf fr2,fr1
movf fr1, id1

Grouping of Compiler Phases

- Front end
  - ❖ Consist of those phases that depend on the source language but largely independent of the target machine.
- Back end
  - ❖ Consist of those phases that are usually target machine dependent such as optimization and code generation.

Common Back-end Compiling System



Compiling Passes

- Several phases can be implemented as a single pass consist of reading an input file and writing an output file.
- A typical multi-pass compiler looks like:
  - First pass: preprocessing, macro expansion
  - Second pass: syntax-directed translation, IR code generation
  - Third pass: optimization
  - Last pass: target machine code generation

A Short History of Compiler Construction

- 1945—1960 Code Generation

  How to generate code for a given machine

  The goal was to match the efficiency of assembly coding.

- 1960—1975 Parsing

  Many new languages came out. Automatic parsing became more important.

- 1975—presentCode Optimization

  RISC machines. Multiprocessors (SMP, CMP)

Cousins of Compilers

- Preprocessors
- Assemblers
  - Compiler may produce assembly code instead of generating relocatable machine code directly.
- Loaders and Linkers
  - Loader copies code and data into memory, allocates storage, setting protection bits, mapping virtual addresses, .. Etc
  - Linker handles relocation and resolves symbol references.
- Debugger

Compiler Constructions Tools

- First Fortran compiler took 18 person-years. Now with compiler construction tools, you may build one in a semester.
- Translator writing tools:
  - Scanner generator
  - Parser generator
  - Syntax directed translation engines
  - Automatic code generator
  - Data flow analyzer generator

Cross-Compilation and Bootstrapping

- Intel introduced the new 64-bit architecture IA-64, and a few generation of processors: Itanium, McKinley, Madison, Montecito.

Q: How to create the first C compiler on the Itanium?

a) Write a C compiler in Itanium machine code

b) Develop a Cross-compiler (and use it to compile itself into C/Itanium).

c) Leave it to MicroSoft

Cross-Compilation and Bootstrapping

- **Quiz: How to create the first C compiler on the new Itanium if no cross-compilers to use?**
- **Answer:** Bootstrapping.

  A subset of C is selected (e.g. C--) and a simple compiler is written in assembly code, called this compiler C0.

  Rewrite this subset compiler using the subset (C--), compile it with C0, get a new compiler called C1.

  Write a more complete set of C in C--, compiled with C1, get a new compiler C2

  Repeat the process until a complete C compiler is done

Compiler Construction (750421)

A Compulsory Module for Students in

Computer Science Department

Faculty of IT / Philadelphia University

Second Semester 2006/2007

Compiler Construction (750421)

Lecturer: Dr. Nadia Y. Yousif

Email: nyaaqob@philadelphia.edu.jo
nadiayy@hotmail.com

Room: IT 332

Course Outline

- Aims
- Objectives
- Assessment and Passing the Subject
- Lectures and Practice Classes
- Lecturer and Consultation
- Recommended Reading
- Course Overview

Aims of This Module

- to show how to apply the theory of language translation introduced in the prerequisite courses to build compilers and interpreters.
- to cover the building of translators both from scratch and using compiler generators.
- to identify and explore the main issues of the design of translators.
- To know the topics: compiler design, lexical analysis, parsing, symbol tables, declaration and storage management, code generation, and optimization techniques.
- To practice with a compiler for a small language

Course Objectives

1- Understand the structure of compilers.

2- Understand the basic techniques used in compiler construction such as lexical analysis, top-down, bottom-up parsing, context-sensitive analysis, and intermediate code generation.

3- Understand the basic data structures used in compiler construction such as abstract syntax trees, symbol tables, three-address code, and stack machines.

4- Design and implement a compiler using a software engineering approach.

5- Use generators (e.g. Lex and Yacc)

Assessment and Passing

- There are **three assessment** components:
  - Two midterm exams worth 15% of the marks each
  - Assignments worth 20% of the marks
  - Final exam (written (40%) + Project (10%))

- You need to achieve an overall mark of 50% to pass the course.

Lectures and Practice Classes

- Lectures will be held at:

  10:10 am on Sundays, Tuesdays, Thursdays, Room 7415

  Practical work will be held in a lab as self learning.

- After three weeks, students are expected to work on practice problems, or on their assignments
- The lecturer will be available to comment on, and help with, solutions during the practice class.

Lecturer and Consultation

- **Lecturer:**
  Dr. Nadia Y. Yousif
  Faulty of IT, Room 332, Phone Ext: 2544
  email: nyaaqob@philadelphia.edu.jo
- **Consultation**
  - The primary time for consultation is during the practice classes
  - Other consultation at the office hours (in room 332) on:
    (Sun, Tues, Thu) 13:00 – 14:00
    (Mod, Wed) 13:45 – 15:15

Recommended Reading

- **The text book is:**

  Alfred V. Aho, Ravi Sethi and Jeffry D. Ulman, Compilers Principles, Techniques and Tools, Addison Wesley, 1986,

  ISBN: 0- 201- 10088- 6

- **Supporting References**:

  1- W. Appel, Modern Compiler Implementation in Java, Prentice Hall, 2002

  2- D. Watt, Brown, Programming Language Processors in Java: Compilers and Interpreters, Prentice hall, 2000

Course Overview

- **Introduction to Compiling**: The role of language translation in the programming process; Comparison of interpreters and compilers, language translation phases, machine-dependent and machine-independent aspects of translation, language translation as a software engineering activity
- **Lexical Analysis**: Application of regular expressions in lexical scanners,
- **Lexical Analysis:** hand coded scanner vs. automatically generated scanners
- **Lexical Analysis:** formal definition of tokens, implementation of finite state automata.
- **Syntax Analysis**: Revision of formal definition of grammars, BNF and EBNF; bottom-up vs. top-down parsing,
- **Syntax Analysis**: tabular vs. recursive-descent parsers, error handling,
- **Parsers Implementation:** automatic generation of tabular parsers, symbol table management, the use of tools in support of the translation process

Course Overview (Cont.)

- **Semantic Analysis:** Data type as set of values with set of operations, data types, type- checking models, semantic models of user-defined types, parametric polymorphism, subtype polymorphism, type-checking algorithms
- **Intermediate Representation**, **code generation**: Intermediate and object code, intermediate representations, implementation of code generators
- **Code generation**: code generation by tree walking; context sensitive translation, register use.
- **Code optimization**: Machine-independent optimization; data-flow analysis; loop optimizations; machine-dependent optimization
- **Error Detection and RecoveryError Repair,**
- **Compiler Implementation**
- **Compiler design options and examples:** C Compilers, C++, Java Compilers

Chapter 1
Introduction to Compiling

- **Topic to cover**:
  - Overview of Compilers
    - The phases of Compilers
    - The Tasks of the Compilation Process
    - Analysis of the Source Program
    - Intermediate Code Generation
    - Loaders and linkers

Chapter 1
Overview of Compilers (Cont.)

- A **translator** inputs and then converts a **source program** into an **object or target** program.
- **Source program** is written in a source language
- **Object program** belongs to an object language
- A translators could be: **Assembler, Compiler, Interpreter**

Assembler:

source program (in assembly language) → Assembler → object program (in machine language)

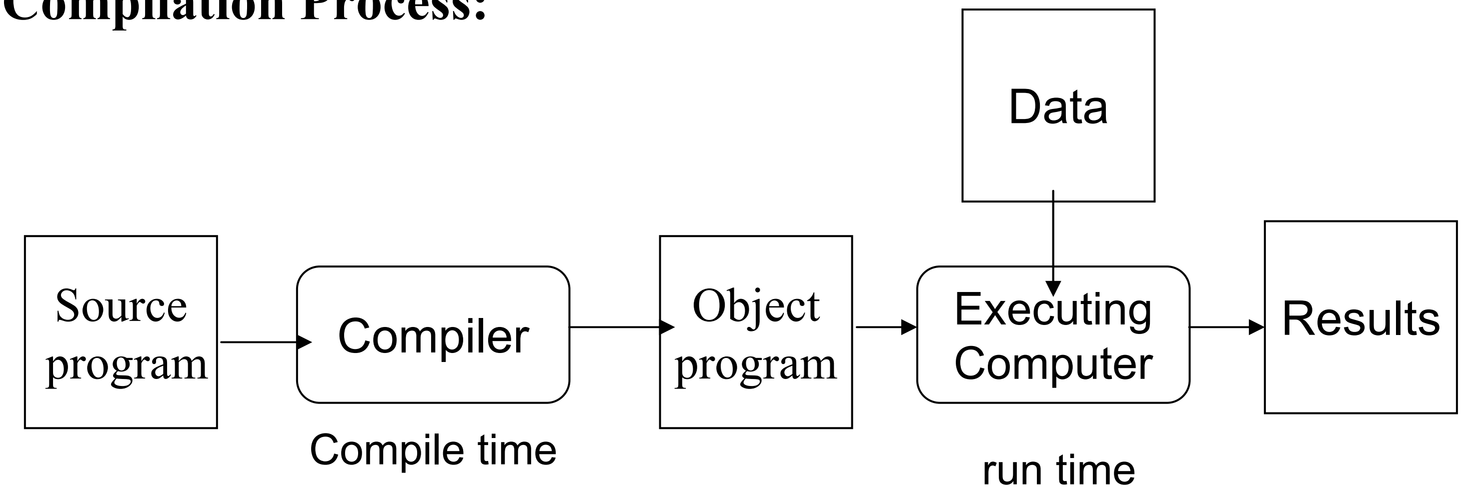
Chapter 1
Overview of Compilers (Cont.)

- **Compiler**: translates a source program written in a High-Level Language (HLL) such as Pascal, C++ into computer's machine language (Low-Level Language (LLL)).
  * The time of conversion from source program into object program is called **compile time**
  * The object program is executed at **run time**

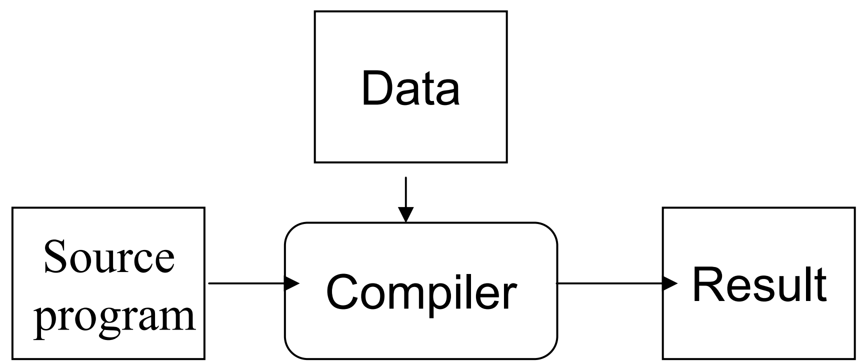
- **Interpreter**: processes an internal form of the source program and data at the same time (at run time); no object program is generated.

Chapter 1
Overview of Compilers (Cont.)

Compilation Process:



Interpretive Process:



Chapter 1
Overview of Compilers (Cont.)

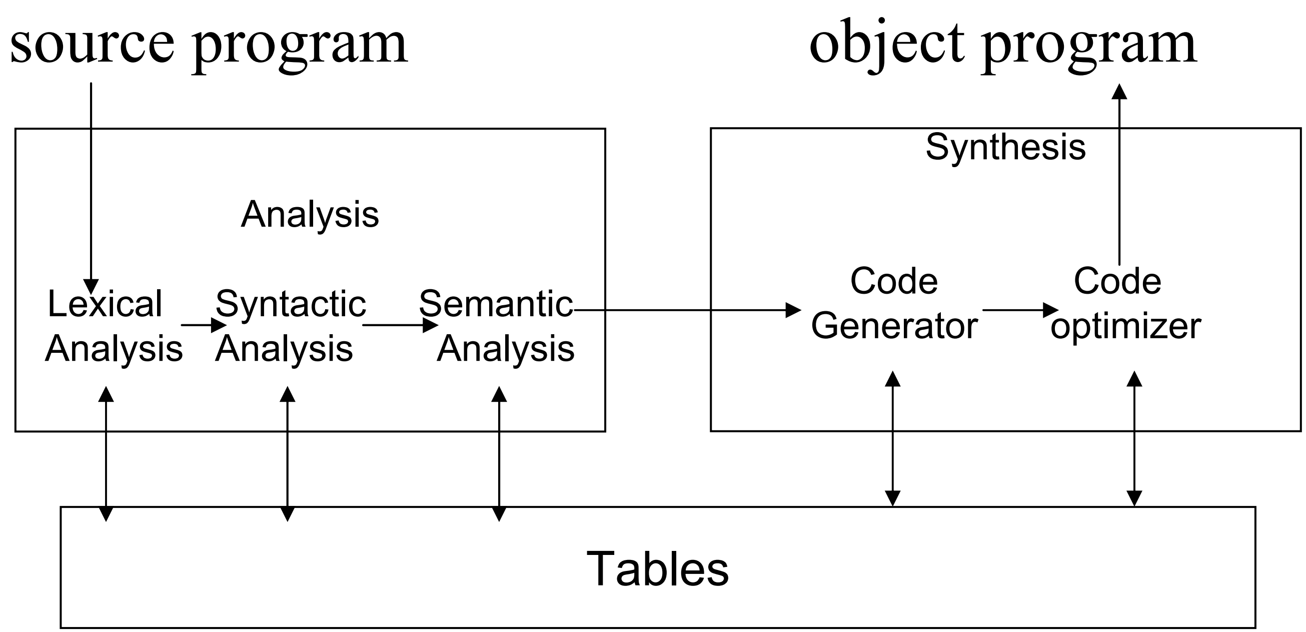
- Compiler writing spans
  - programming languages
  - machine architecture
  - language theory
  - algorithms
  - software engineering

Model of A Compiler

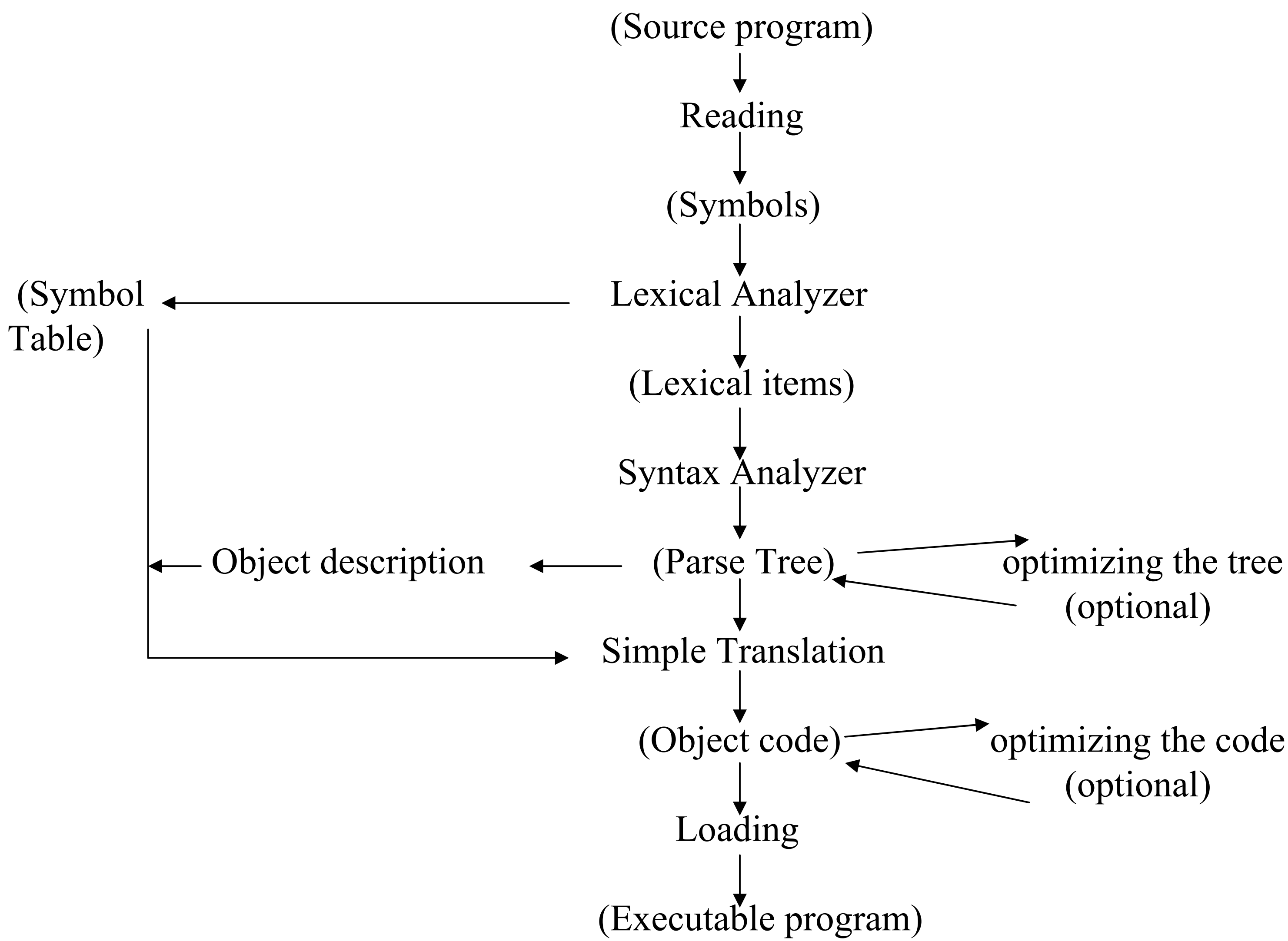
- A compiler must perform two tasks:

  - analysis of source program

  - synthesis of its corresponding program



Tasks of Compilation Process and Its Output



Tasks of Compilation Process and Its Output

- Each tasks is assigned to a phase, e.g. Lexical Analyzer phase, Syntax Analyzer phase, and so on.
- Each task has input and output.
- Any thing between brackets in the last figure is output of a phase.
- The compiler first analyzes the program, the result is representations suitable to be translated later on:

  - Parse tree

  - Symbol table

Parse Tree and Symbol Table

- Parse tree defines the program structure; how to combine parts of the program to produce larger part and so on.

- Symbol table provides
  - the associations between all occurrences of each name given in the program.
  - It provides a link between each name and it declaration.

Example on Compilation Process

```
Main ()
{ int a; double b;
  a = 1;
  b = 1.5;
  a = b + 2;
  cout << a;
}
```

Example on Compilation Process (Cont.)

- First, the ***Reading*** phase reads the source program and produces **symbols**.
- ***Lexical Analyzer*** (or scanner) takes the symbols and separates them into **tokens,** e.g.
  - constants
  - variable names
  - keywords (if, while, switch, etc.)
  - operators (+, -, *, /, <, >, etc)
- Each token is given a unique internal representation number, e.g.;
  - variable name is given 1,
  - constant is 2
  - addition operation is 3
  - etc.

Example on Compilation Process (Cont.)

Example: a = b + 2;

would be tokenized by the Lexical analyzer into a sequence of tokens:

a	1
=	10
b	1
+	3
2	2
;	27

- The other output from Lexical analyzer is the **symbol table** to contain constants, labels, and variable names.
- A table entry for a variable may contain:
  - its name
  - its type (int, double, etc.)
  - object program address
  - its value
  - line in which it is declared

Example on Compilation Process (Cont.)

- Variables in Symbol table:

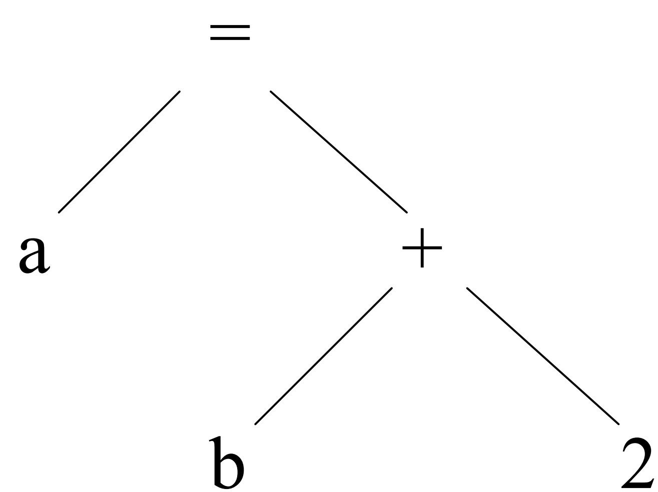
Name	Descriptor
main	
a	
b	
cout	

- Tokens may take the form of pairs of items:
  - First item gives the address or location of the token in the symbol table
  - Second item is the representation number of the token
- Advantages of such approach: all tokens are represented by fixed-length information: an address (or location) and an integer

<u>Example on Compilation Process (Cont.)</u>

- Syntax Analyzer takes the tokens as input and produces a **<u>parse tree</u>**:
- E.g., for the statement a = b + 2;

  the tree is



Example on Compilation Process (Cont.)

- *Object description (Semantics)* phase:
  - places descriptions in symbol table.

Name	Descriptor
main	function
a	variable, int, #1
b	variable, double, #2
cout	Function, address from loader

- ***Simple Translation*** phase takes the parse tree and produces the object code. E.g. The code of the statement a = b + 2; is

```
load  r1, #2
add   r1, 2
fix   r1
store r1, #1
```

Ch 2. A Simple One-Pass Compiler

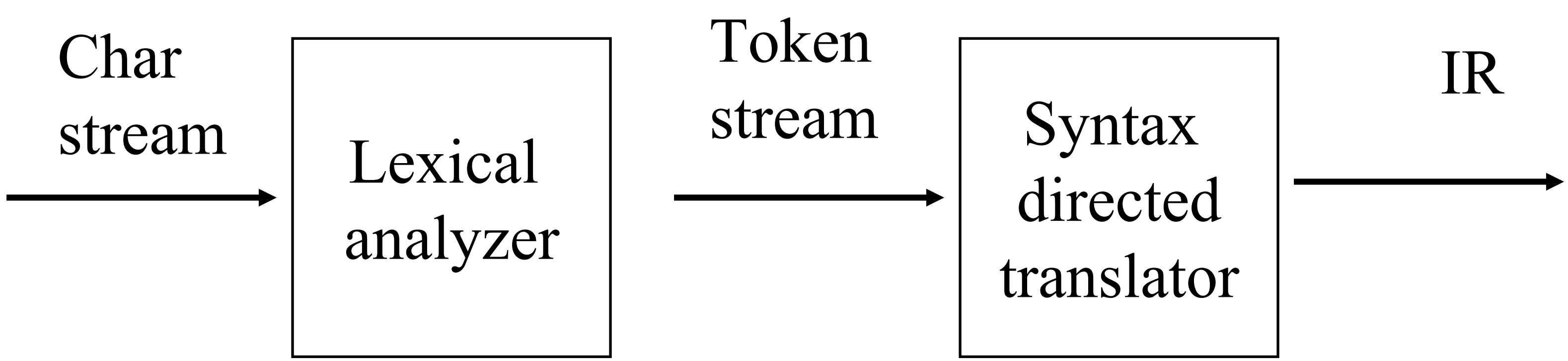
- Build a simple Infix expression to Postfix form translator
- Focus on the front-end: lexical analysis, syntax analysis, and IR code generation
- Cover basic techniques that will be discussed in details in Ch 3 through Ch 6

Overview

- CFG (Context Free Grammar) is used to define the source language
  - To define what the program looks like, i.e. the syntax
  - To guide program translation
- Infix and Postfix form
  - Infix form: for example, A+B+C*D
  - Postfix form: AB+CD*+
  - Postfix notation can be converted directly into code for a stack machine, for example

    push A, push B, +, push C, push D, *, +, store

Structure of the Simple Compiler



(parser and code gen)

Syntax Definition

The syntax of an if statement

If (a > b) a++; else b++;

can be defined as follows:

stmt → if (expr) stmt else stmt

This rewriting rule is called a ***production*.**

A grammar is simply a set of rewriting rules in the following form:

A → B C D … Z

where **A** is the left-hand side (LHS) of the production.

B C D … Z is the right-hand side (RHS).

Production Example

stmt → ***if*** (expr) stmt ***else*** stmt

- LHS is the name of the syntactic construct; the RHS shows a possible form of the syntactic construct.
- LHS are always ***nonterminals***, RHS can have ***terminals*** and ***nonterminals***.
- "if", "(" ")" "else" lexical elements are called *tokens*, or *terminal* symbols
- Stmt is ***nonterminal*.**
- Expr is also ***nonterminal*.**

CFG

- A set of tokens
- A set of non-terminals
- A set of productions
- A start symbol (one of the non-terminals)

Example: Expressions consisting of digits and plus and minus signs

1. List → List + Digit
2. List → List – Digit
3. List → Digit
4. Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Strings

- The empty string (string containing no tokens) is denoted by e
- A string is derived by repeatedly applying productions to a non-terminal symbol
- The ***language*** defined by a grammar is the token strings that can be derived from the start symbol

Example

How to derive 9-2+4?

String Derivation

1. List → list + digit
2. List → list – digit
3. List → digit
4. Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

How to derive
9 – 2 + 4 ?

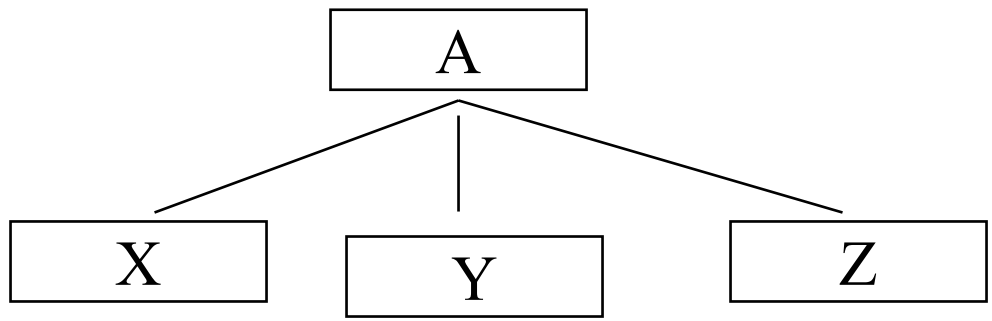
Step 1: <list>	start symbol
Step 2: <list> + <digit>	rule 1
Step 3: <list> - <digit> + <digit>	rule 2
Step 4: <digit> - <digit> + <digit>	rule 3
Step 5: 9 – 2 + 4	rule 4 applied 3 times

What would happen if we select rule 2 at step 2?
Parsing is trying to come up with a derivation of a valid string.

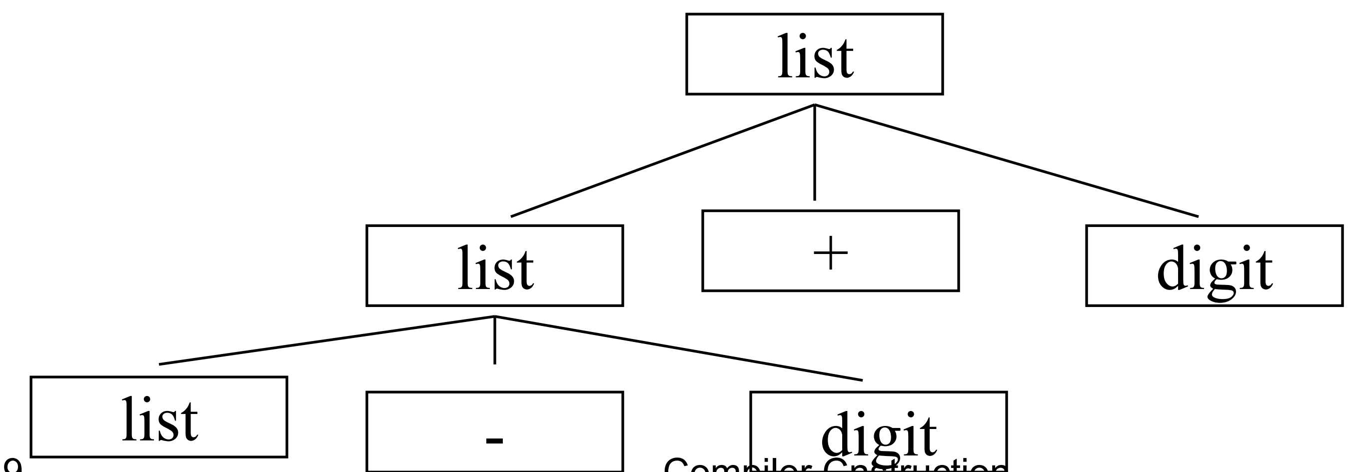
Parse Tree

- A Parse Tree shows how the start symbol derives a string.

  Example: A→ XYZ



Example: list → 9 – 2 +4



Parse Tree

- A Parse Tree has the following properties
  - The root is labeled by the start symbol
  - Each leaf is labeled by a token or by e
  - Each internal node is labeled by a non-terminal
  - If A is the non-terminal labeling, and X,Y,Z are labels of the children of A from left to right, then A→ XYZ is a production
- ***Parsing*** is the process of finding a parse tree for a given string of tokens.
- A grammar can have more than one parse tree for a given string. Such a grammar is ***ambiguous***.

Example

A subset of C grammer:

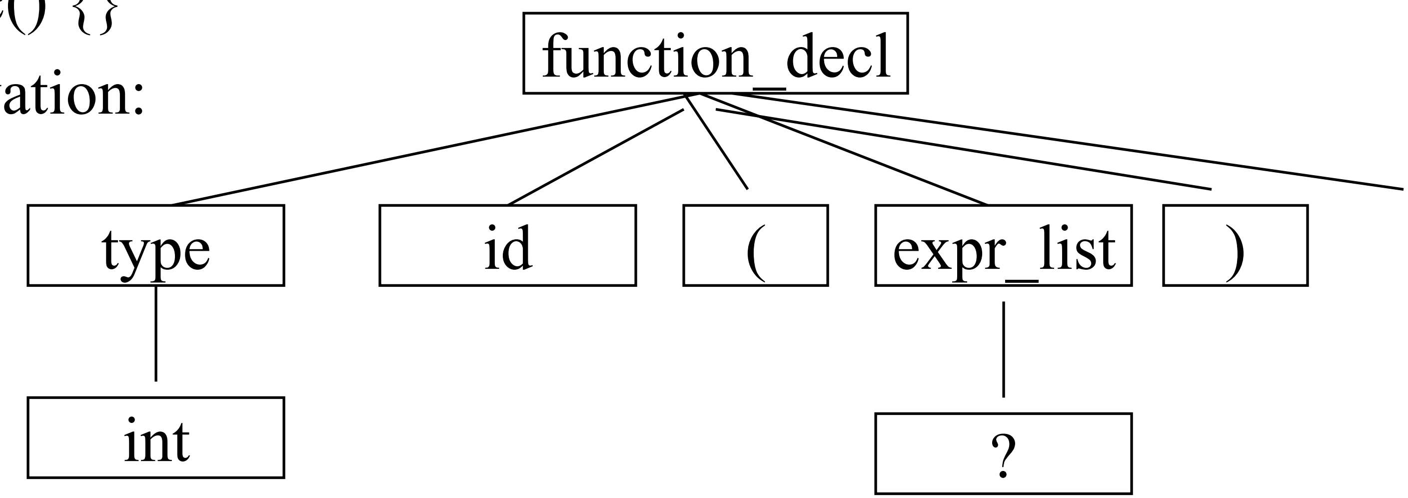
<function_decl> → <type> id (<expr_list>) { <block> }

<block> → <decl_list> <statement_list> | e

A C program – a token string

int func() {}

A derivation:



Parsing: to come up with the parse tree and validate the token string is a correct C program.

Ambiguous Grammar

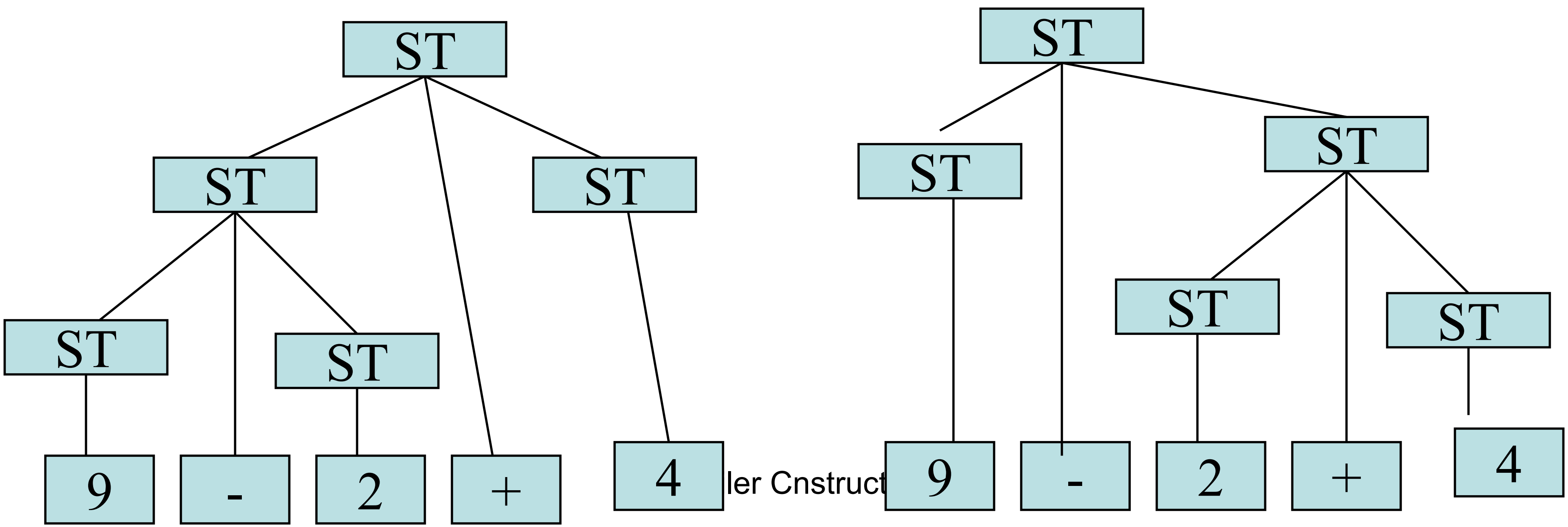
- Example CFG

  String → String + String

  String → String – String

  String → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The string 9-2+4 can have two parse trees



Association and Precedence of Operators

- Left Association and Right Association
  - An operator associates to the left if an operand has operators on both side, and the operand is taken by the operator to its left.
  - Left associative operators:
    - +, -, *, /
    - e.g. A + B + C, A*B*C
  - Right associative operators:
    - Exponential and the assign operator
    - e.g. A = B =C

Exercise

1. List → list + digit
2. List → list – digit
3. List → digit

+ and – are left associative operators. If they are right associative operators, how would the CFG be different.

Precedence of Operators

* (multiply) and / (divide) have higher precedence than + and -. How is precedence of operators handled?

- We can create two non-terminals for the two levels of precedence.
- **Example CFG**

  Expr → expr + term | expr – term | term

  Term → term * factor | term / factor | factor

  Factor → digit | (expr)

Example

- **Example CFG**
  1. Expr → expr + term | expr – term | term
  2. Term → term * factor | term / factor | factor
  3. Factor → digit | (expr)

How is 9 – 5 – 2 * 4 derived?

expr → expr – term
→ expr – term – term
→ term – term – term * factor
→ factor – factor – factor * factor
→ 9 – 5 – 2 * 4

Syntax Directed Translation

- **Syntax-Directed Definition**
  - CFG + semantic rules
  - It specifies the translation of a construct in terms of **attributes associated with its grammar symbol and semantic rules associated with each production**. Semantic rules can be actual code known as semantic routines (or action routines).
  - Attributes can be a type, a string, a memory location, … etc.
  - A parse tree showing the attribute values at each node is called an *annotated parse tree*.

Example 1

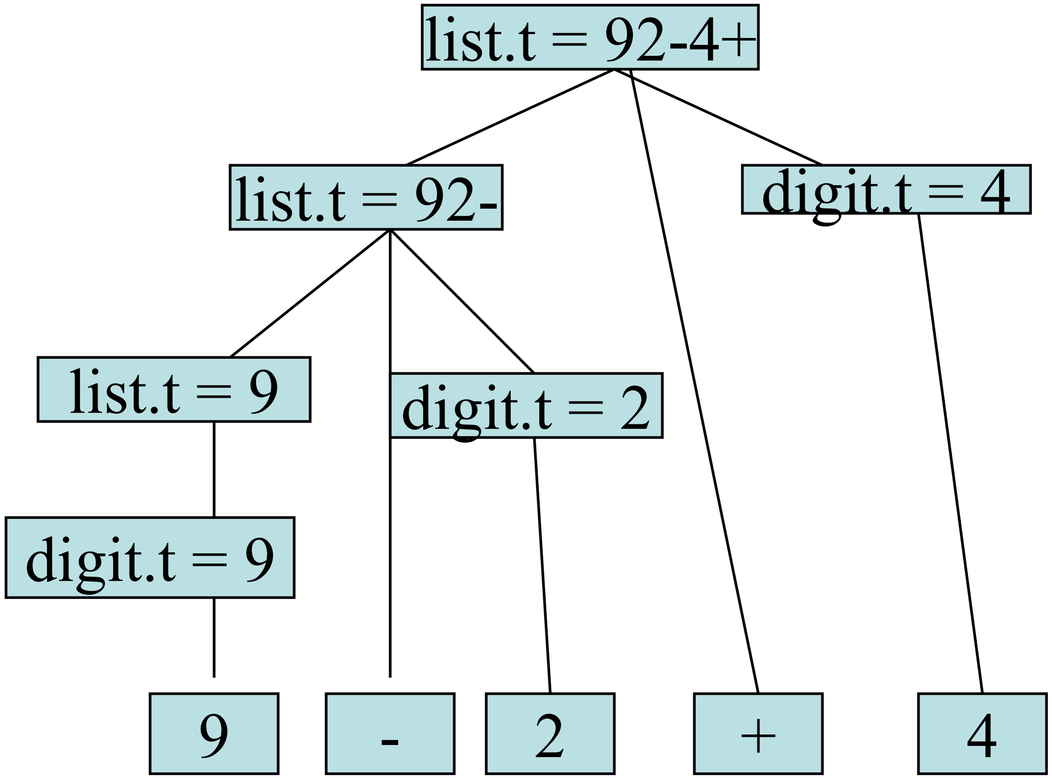
Syntax-Directed Definition

- Type Checking
  - E → E1 op E2 { if E1.type != E2.type convert();}

    type is an attribute

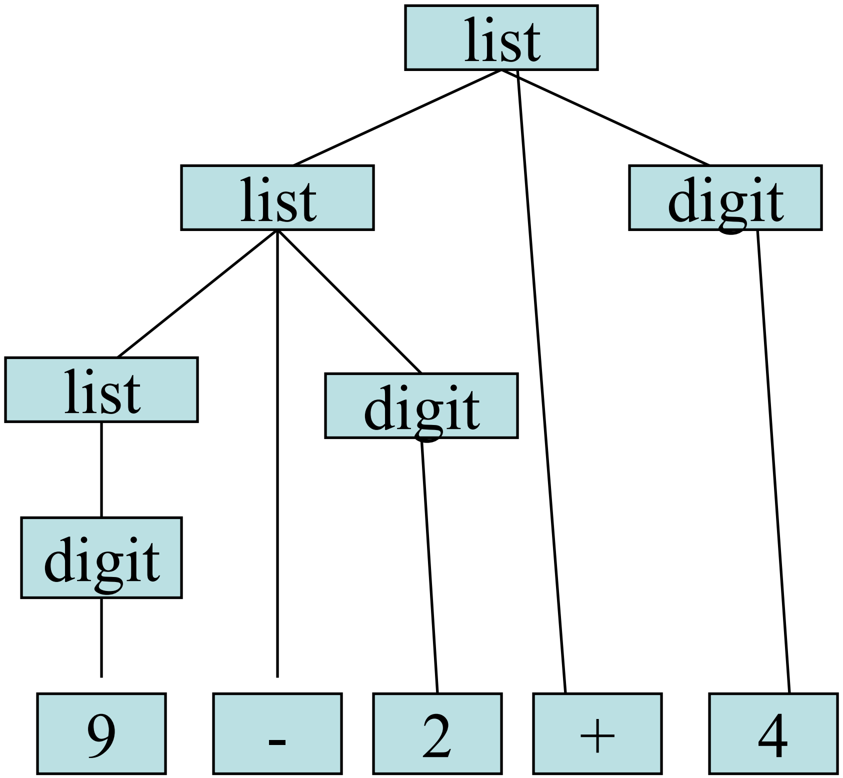
    semantic rule specified by C code
- Translation
  - E → E1 op E2 { emit(E.loc,":=", E1.loc, op, E2.loc);}

Annotated Parse Tree

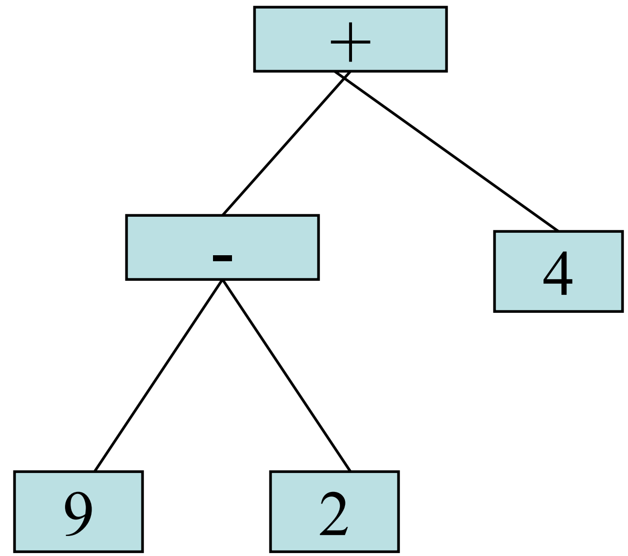


Annotated Parse Tree

Parse Tree and Syntax Tree

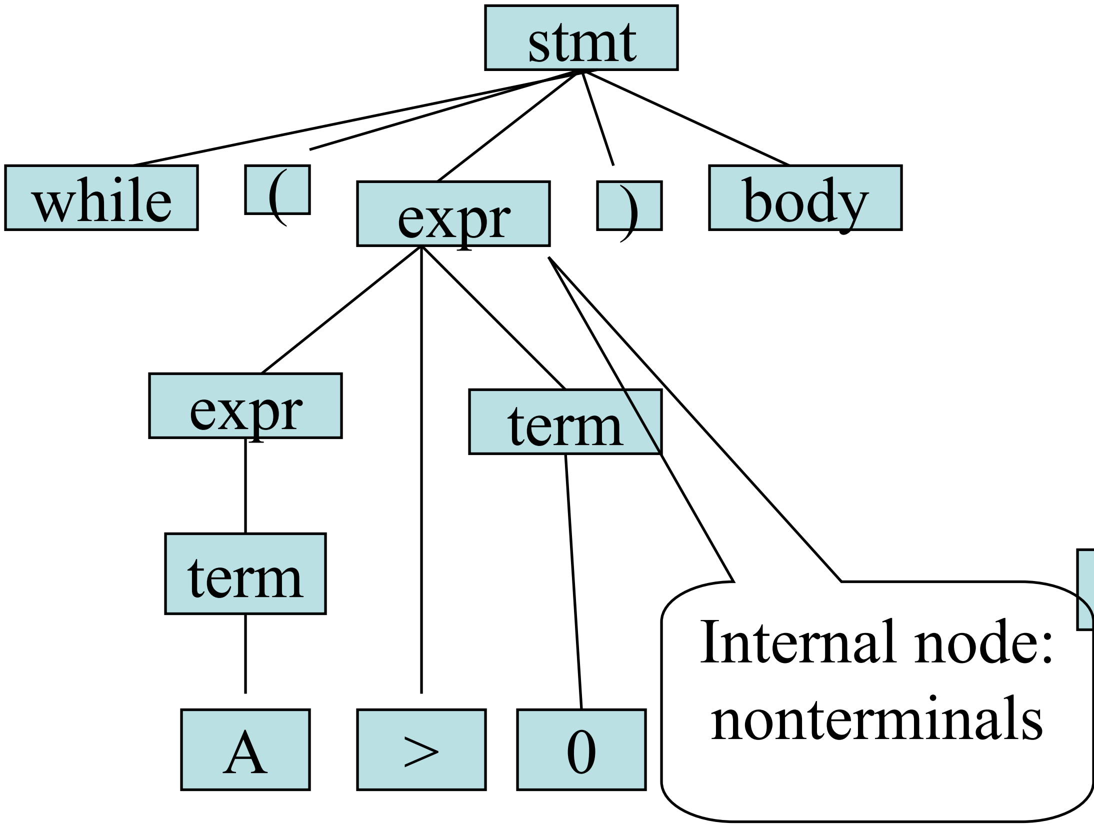


Parse Tree

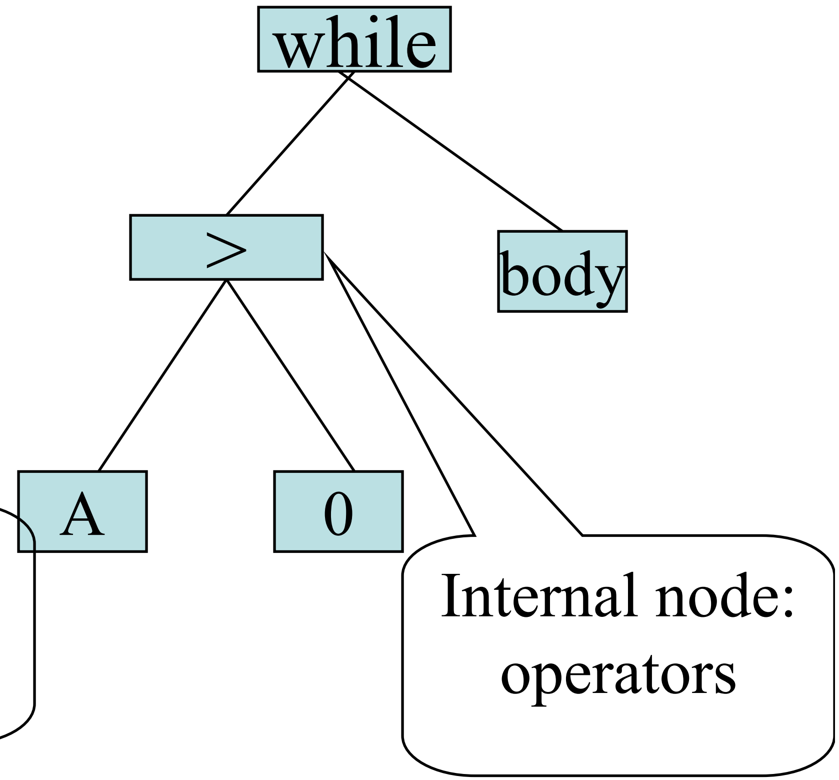


Syntax Tree

Parse Tree and Syntax Tree



Parse Tree



Syntax Tree

Example 2

Syntax-directed definition for infix to postfix translation

Production	Semantic Rules
Expr → expr1 + term	Expr.t := expr1.t || term.t || "+"
Expr → expr1 - term	Expr.t := expr1.t || term.t || "-"
Expr → term	Expr.t := term.t
Term → 0	Term.t := "0"
Term → 1	Term.t := "1"

Translation Schemes

- A procedural specification for defining a translation
- Translation scheme is like syntax-directed definition except that the order of evaluation of semantic rules is explicit.

Examples

rest → + term { *print("+")* } rest

while_stmt → while *#Startwhile* <b_exper> *#Whiletest* do begin <stmn> end *#Finish*

Startwhile, whiletest and Finish are procedures

Translation Schemes

Production	Semantic Actions
Expr → expr1 + term	{ print (“+”) }
Expr → expr1 - term	{print(“-”) }
Expr → term	
Term → 0	{print(“0”) }
Term → 1	{print(“1”) }

Semantic action routine is called when the production rule is applied

Parsing

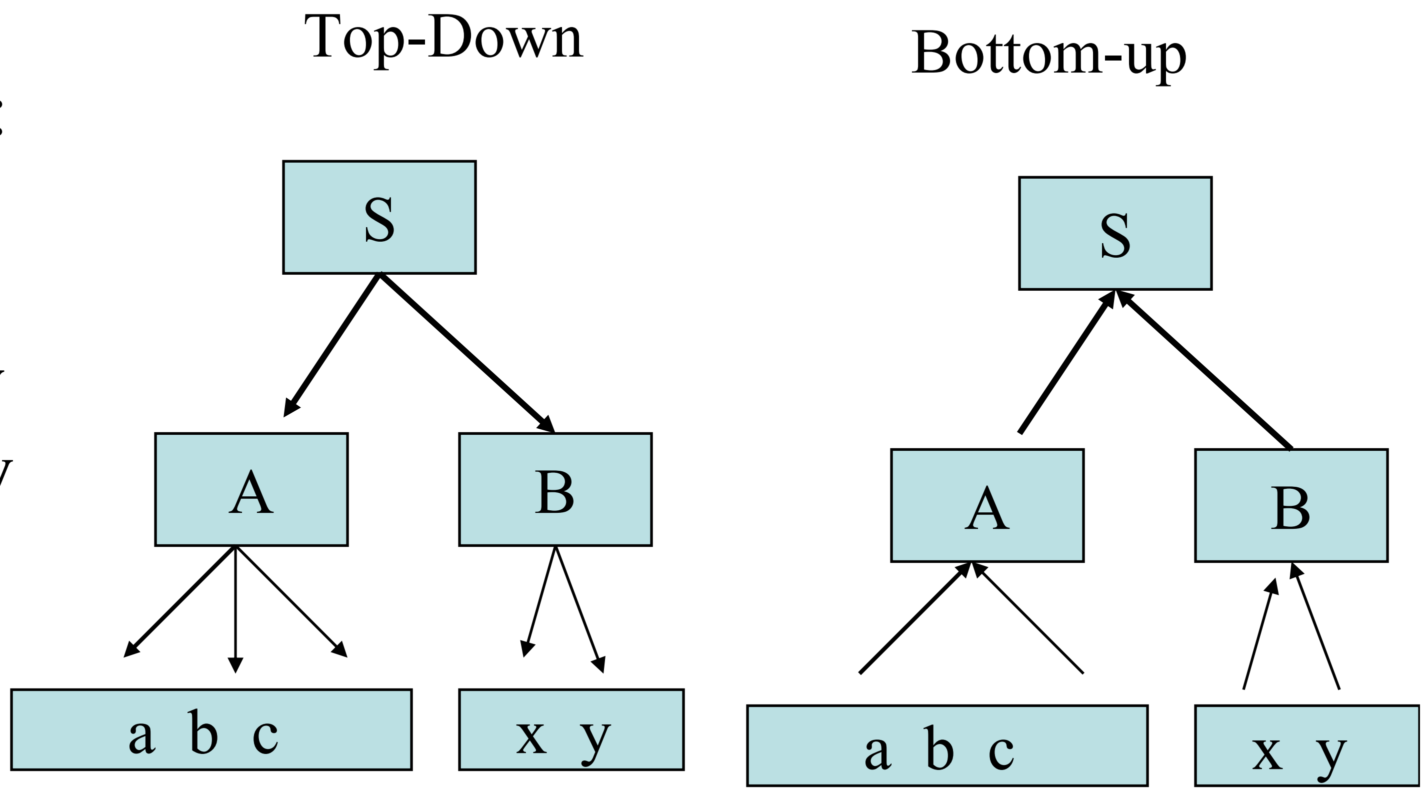
- Parsing is to determine if a string of token can be generated by a grammar
- Two common parsing methods: Top-down and Bottom-up
- Top-down starts at the root and proceeds towards leaves
- Bottom-up starts at the leaves and proceeds towards the root

Example

String: abcxy

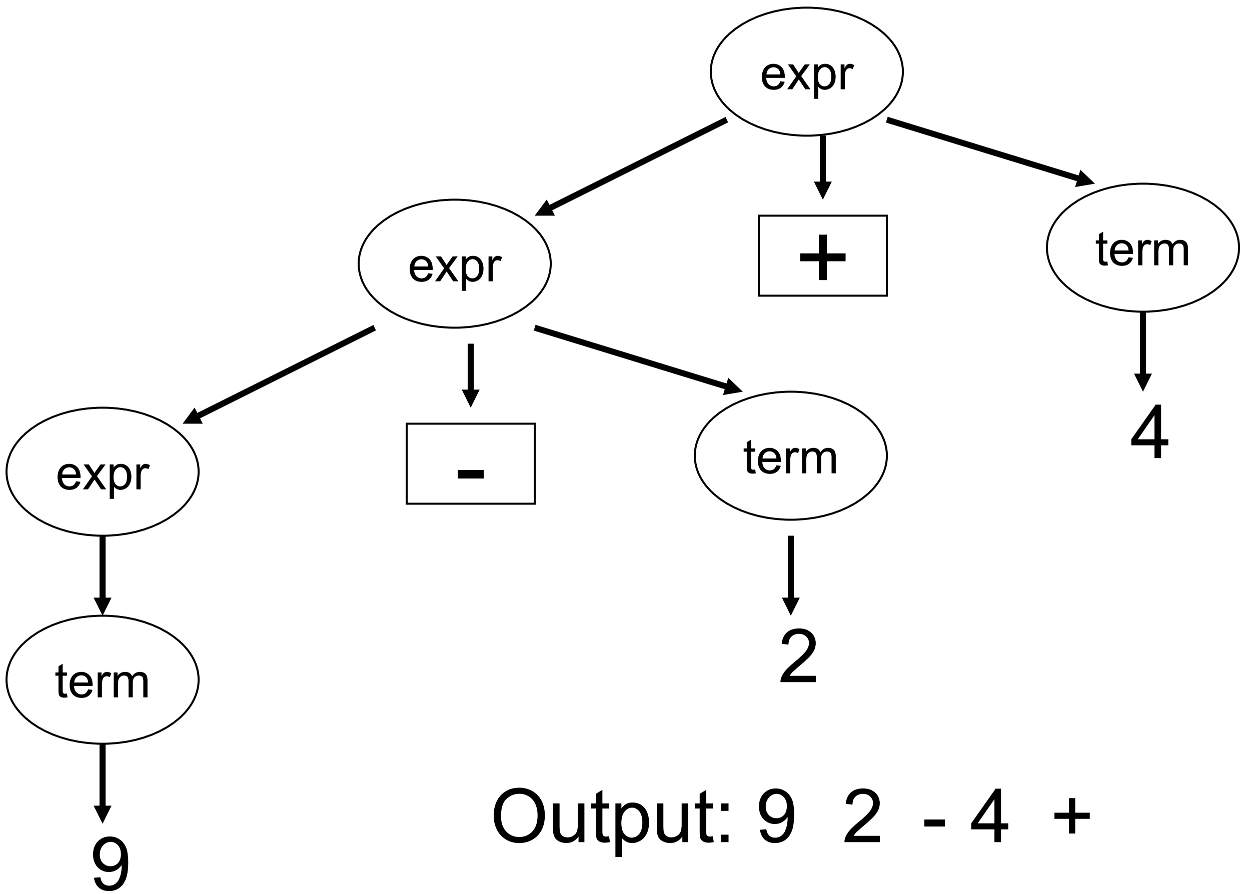
Productions:

S→ AB
A → abc | w
B → def | xy



Example (9-2+4)

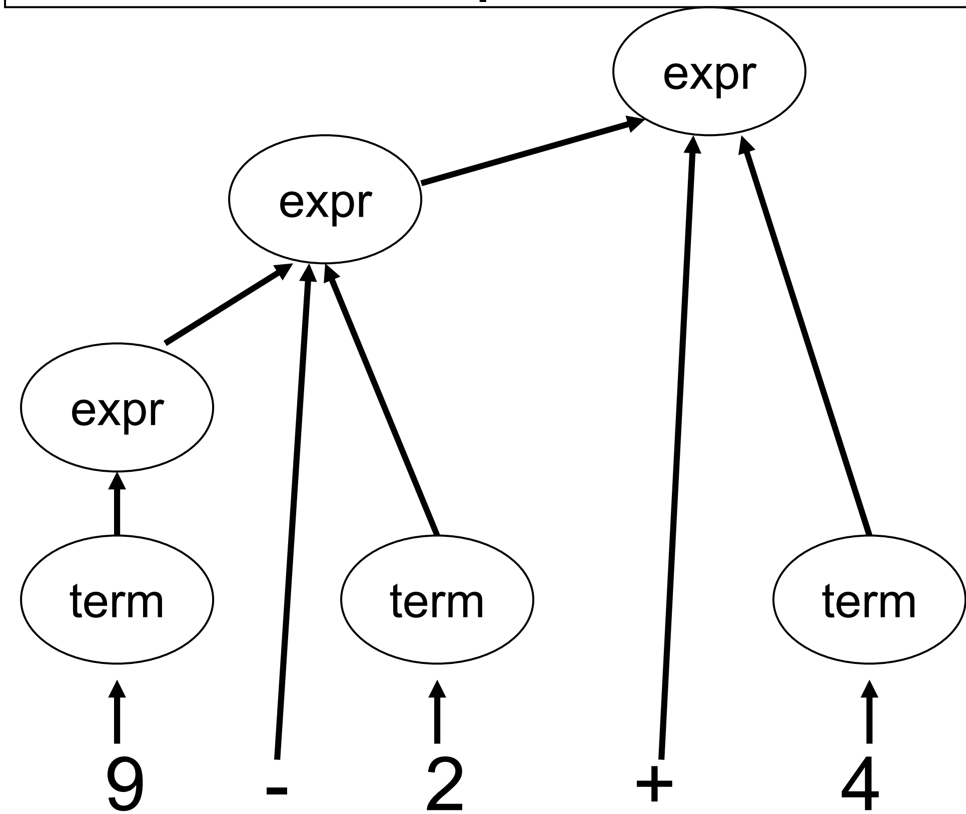
Construct the parse tree top-down



Output: 9 2 - 4 +

Example (9-2+4)

Construct the parse tree bottom-up



Output: 9 2 - 4 +

Two common parsing methods

Top-down	Bottom-up
Easy to understand	Can handle a larger class of grammars
Efficient parsers can be built by hand	Efficient parsers can be built by tools
Some restrictions on grammars. May need to change productions	Less restrictions placed on grammars
Also known as predictive parsing	More commonly used in production compilers

Top-down Parsing

type → simple | ^id |

array [simple] of type

simple → scalar_type |

num dotdot num |

id

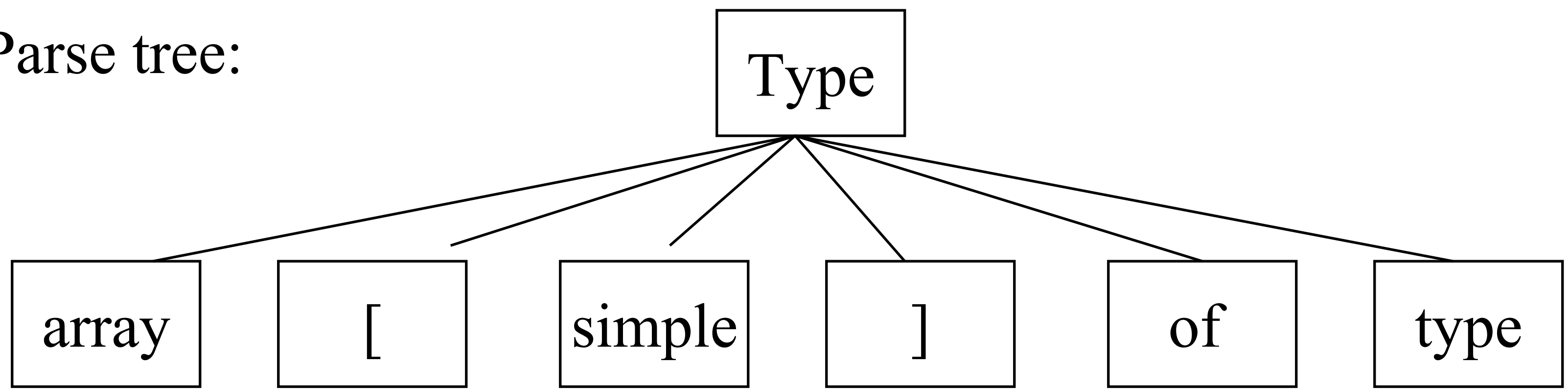
scalar_type → (id_list)

Input string

array [1..100] of integer

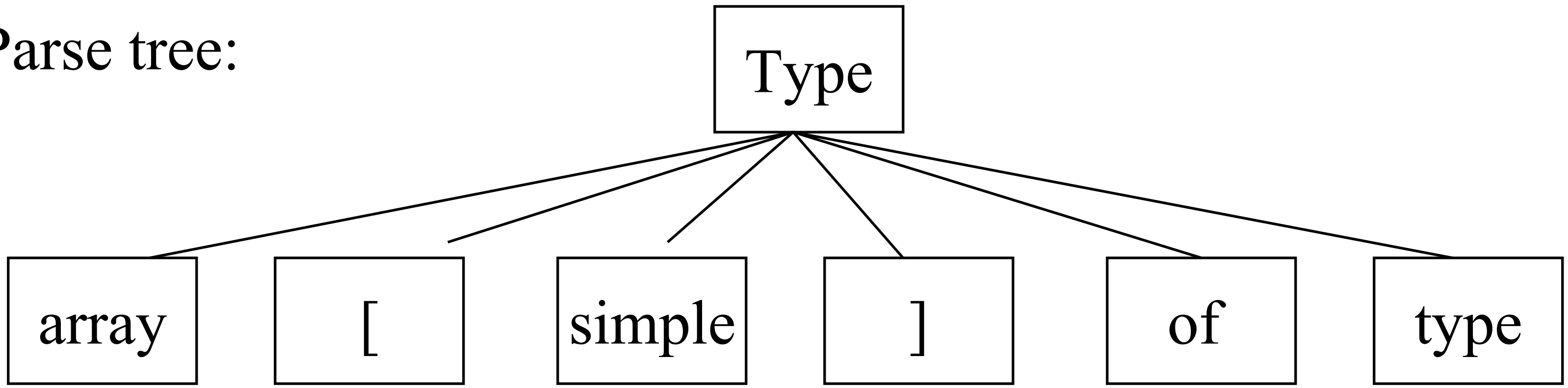
Input: array [1..100] of integer

Parse tree:



Input: array [1..100] of integer

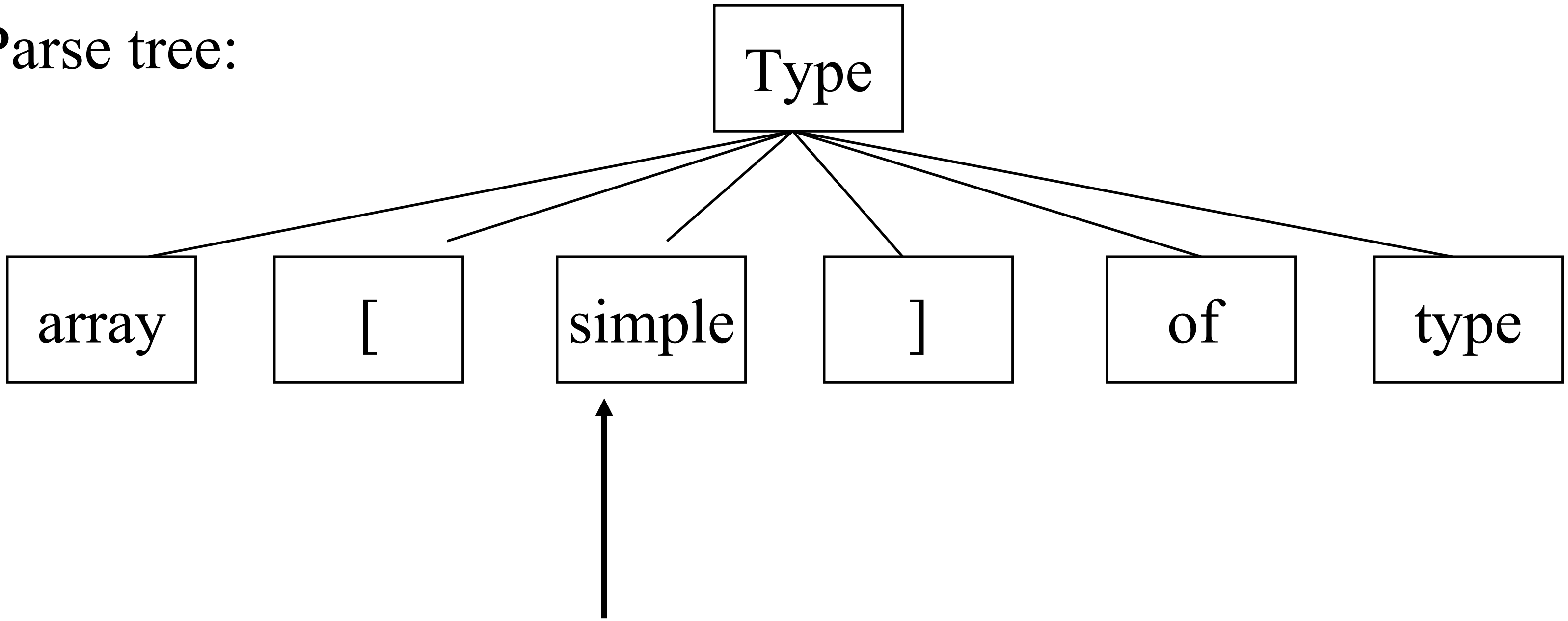
Parse tree:



Input: array [1..100] of integer

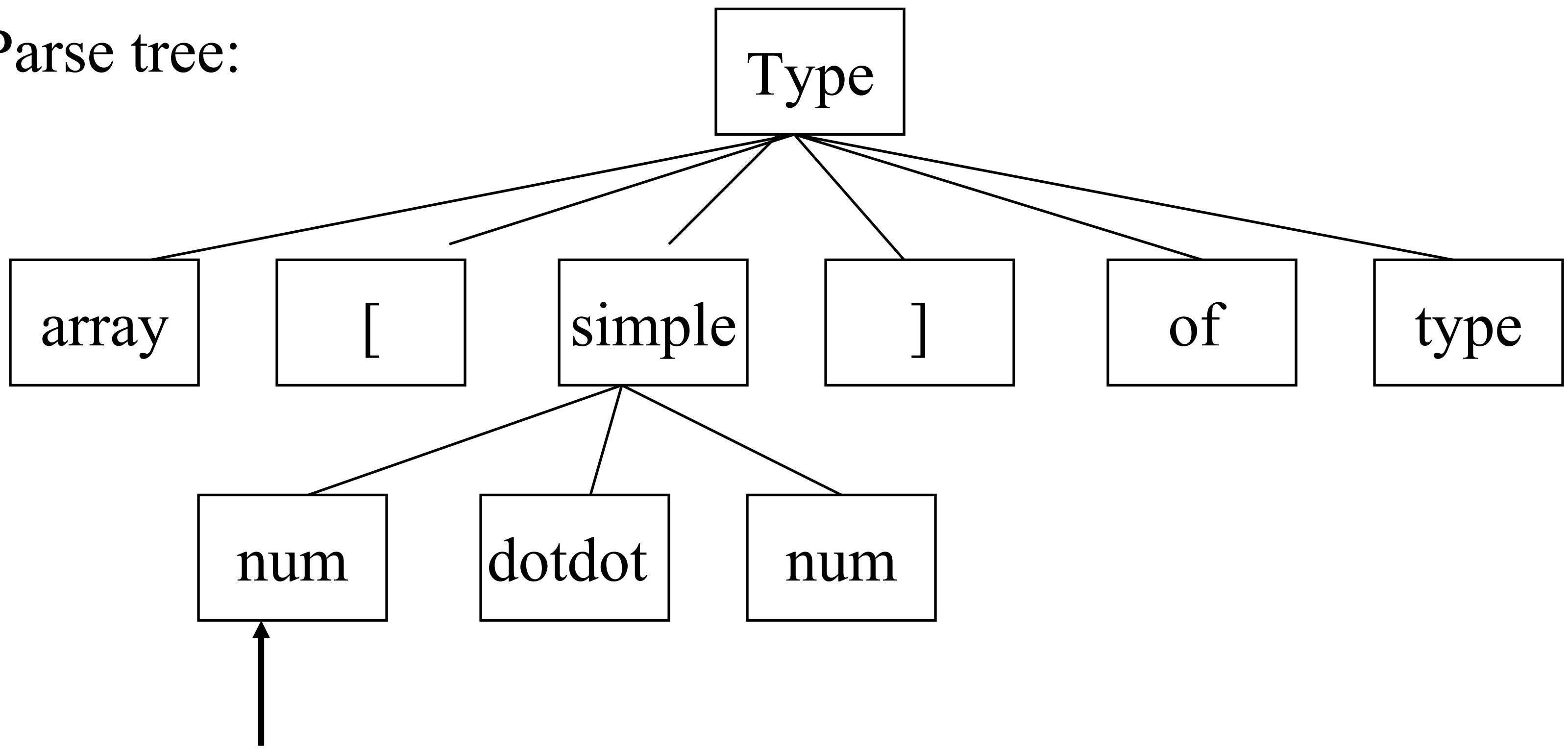
1..100 is
num dotdot num
(three tokens)

Parse tree:



Input: array [1..100] of integer

Parse tree:



- The non-terminal *simple* will be expanded with production simple→ num dotdot num
- In predictive parsing, there will be no backtracking
- If Pascal type is defined as follows:

Simple →   **scalar_type |**

       num dotdot num |

       type_id |

       integer_const

What will happen?

Simple → scalar_type |

num dotdot num |

type_id |

integer_const

What will happen?

array [100] of integer is now legal

but in array [1..100] of integer, 1 may be returned as “num” or “integer_const”, which gives two productions to expand !!

What to do in this case?

Predictive Parsing

- Predictive parsing is a recursive decent parsing, in which we execute a set of recursive procedures to process the input. Each procedure is associated with a non-terminal of a grammar.
- Example

Procedure type

begin

if lookahead in {"(", id, num} then simple();

else if lookahead = "^" then match("^"); match(id)…

else if lookahead = "array" then match("array"),match ("[", simple(); match("]"); match("of"), type();

end

Predictive Parsing

- In predictive parsing, the lookahead symbol can uniquely select the procedure for each non-terminal.
- The FIRST(a) is defined to be the set of tokens that appear as the first symbols that can be generated from a.

  example:

  FIRST(*simple*) = {"(", id, num}
- e-production is used as default when no other productions can be used.

Constructing a Predictive Parser

- Create a procedure for each non-terminal
- It decides which production to use by look at the lookahead symbol
- The procedure mimicking the right hand side: non-terminal will be a call, and a token match with the lookahead will cause the next token to be read.
- Action routines can be copied into the parser.

Left Recursion Removal

Example of left recursion

expre → expr + term

A → Aa | b { b, ba, baa, baaa, …..}

To eliminate left recursion, we can rewrite the productions.

A → b R

R → a R | e { b, ba, baa, baaa, …..}

Exercise

CFG

1. Expr → Expr + term
2. Expr → Expr – term
3. Expr → term

What is α ?

α is + term and – term

What is β ?

After rewriting:

β is term

expr → term Rest

Rest → + term Rest

Rest → - term Rest

Rest → e

New syntax definition after left recursion eliminated:

Expr → Term Rest

Rest → + Term Rest

Rest → - Term Rest

Rest → e

Term → 0

….

Term → 9

Adding translation scheme to them:

Expr → Term Rest

Rest → + Term {print('+')} Rest

Rest → - Term {print('-')} Rest

Rest → e

Term → 0 {print('0')}

….

Term → 9 {print('9')}

A translator for simple expressions

```
Expr() {
        Term();
        Rest(); }
Rest() {
        if (lookahead == '+') {
        match('+'); Term(); putchar('+'); Rest(); }
        else if (lookahead == '-') {
         match('-'); Term(); putchar('-'); Rest(); }
        }
Term() {
        if (isdigit(lookahead)) {
           putchar(lookahead); match(lookahead); }
        else error(); }
```

Summary

- A syntax-directed translator for simple expressions
- Syntax definition – using CFG
- Syntax-directed schemes – CFG plus semantic routines
- Predictive parsing – recursive desent parsing with unique FIRST()
- Left recursion elimination

Recursive Descent Parsing Exercise

stmt → **if** *expr* **then** *stmt tail*
tail → **else** *stmt* | ε

How to write the procedure for stmt?

Recursive Descent Parsing Exercise

stmt → **if** *expr* **then** *stmt tail*
tail → **else** *stmt* | ε

```
stmt()
{
    match('if');
    expr();
    match('then');
    stmt();
    tail();
}
```

Recursive Descent Parsing Exercise

stmt → **if** *expr* **then** *stmt tail*
  | **while** (*expr*) *stmt*

```
stmt()
{
   if (lookahead == 'if')
     {match('if'); expr(); match('then'); stmt(); tail(); }
   else (lookahead == 'while')
     {match('while'); match('('); expr(); match(')');
        stmt(); }
   else error();
}
```

More Exercise

A → B { C } | D E; | F;

```
A()
{
   if (lookahead == ? )
      {B; match ('{'); C; match ('}');}
   else (lookahead ==? )
      {D(); E();}
   else (lookahead == ? )
         F();
   else error();
}
```

1) FIRST(B)
2) FIRST(B{)
3) FIRST(B { C })
4) FIRST(B,D,F)

Which of above are true?

(2) And (3)

Left Recursion Removal

Example of left recursion

expr → expr + term

A → Aa | b { b, ba, baa, baaa, …..}

To eliminate left recursion, we can rewrite the productions.

A → b R

R → a R | e { b, ba, baa, baaa, …..}

Exercise

CFG

1. expr → Expr + term
2. expr → Expr – term
3. expr → term

α is + term and – term

After rewriting:

expr → term Rest

Rest → + term Rest

Rest → - term Rest

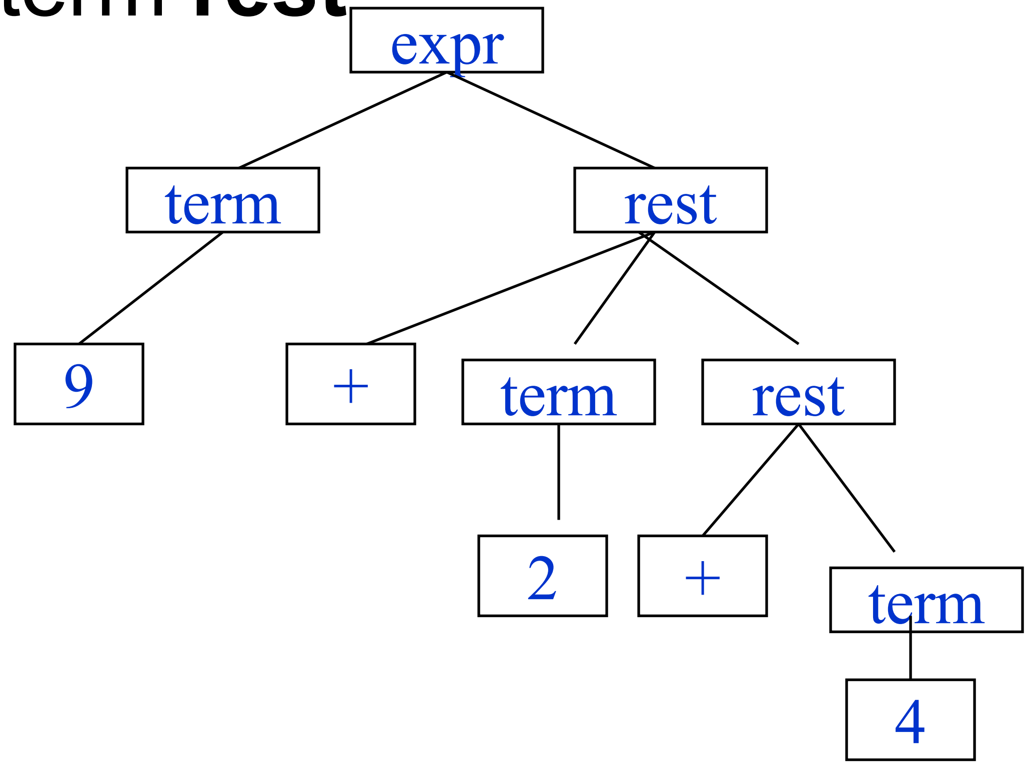
Rest → e

β is term

Parse Tree

expr → term rest

rest → + term **rest**

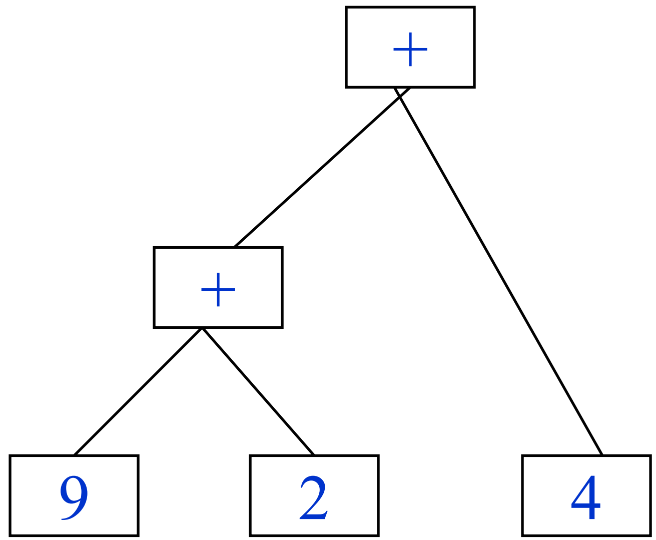


Identical Syntax Tree

expr → term rest

rest → + term **rest**

expr → expr + term



Exercise

Dim $\rightarrow$ Dim [expr]

| [expr]

example: A[100], A[100][5], A[B[i][j]]

To eliminate left recursion of this production, what is a?

what is b?

what is the transformation?

Exercise

Dim → Dim [expr]
| [expr]

a is [expr]

β is [expr]

Transformation:

Dim → [expr] R

R → [expr] R | e

Exercise

Stmtlist → Stmtlist; Stmt

| Stmt

To eliminate left recursion of this production,

what is a?

what is b?

what is the transformation?

Exercise

Stmtlist $\rightarrow$ Stmtlist; Stmt
| Stmt

a is ;Stmt
β is Stmt

Transformation:

Stmtlist $\rightarrow$ Stmt R
R $\rightarrow$; Stmt R | e

- Extending the simple compiler into a more practical one

  So far, the compiler handles only digits and +/- operators.

  e.g. 9+2-5

  Need to be more practical

  e.g. 129 + count * (months / 12);

Extending the Compiler

- Extending the simple compiler into a more practical one
  - White space
  - Constants (numbers)
  - Identifiers and keywords
  - IR code generation for an abstract stack machine
  - Examples on translating statements

Lexical Analysis

- Removal of white space and comments

```
while (1) {
    t = getchar();
    if (t== ‘ ‘ || t == ‘\t’ || t == ‘\n’)
    /* strip off blanks, tabs, new lines */
}
```

- Numbers

  token + attribute value

```
while ( isdigit(t)) {
      value = value*10 + t – ‘0’;
      t = getchar(); }
```

- Identifier

```
if (isalpha(t)) {
int b = 0;
while ( isalnum(t)) {
        lexbuff[b++] = t;
        t = getchar();
}
```

A symbol table is needed to distinguish identifiers.

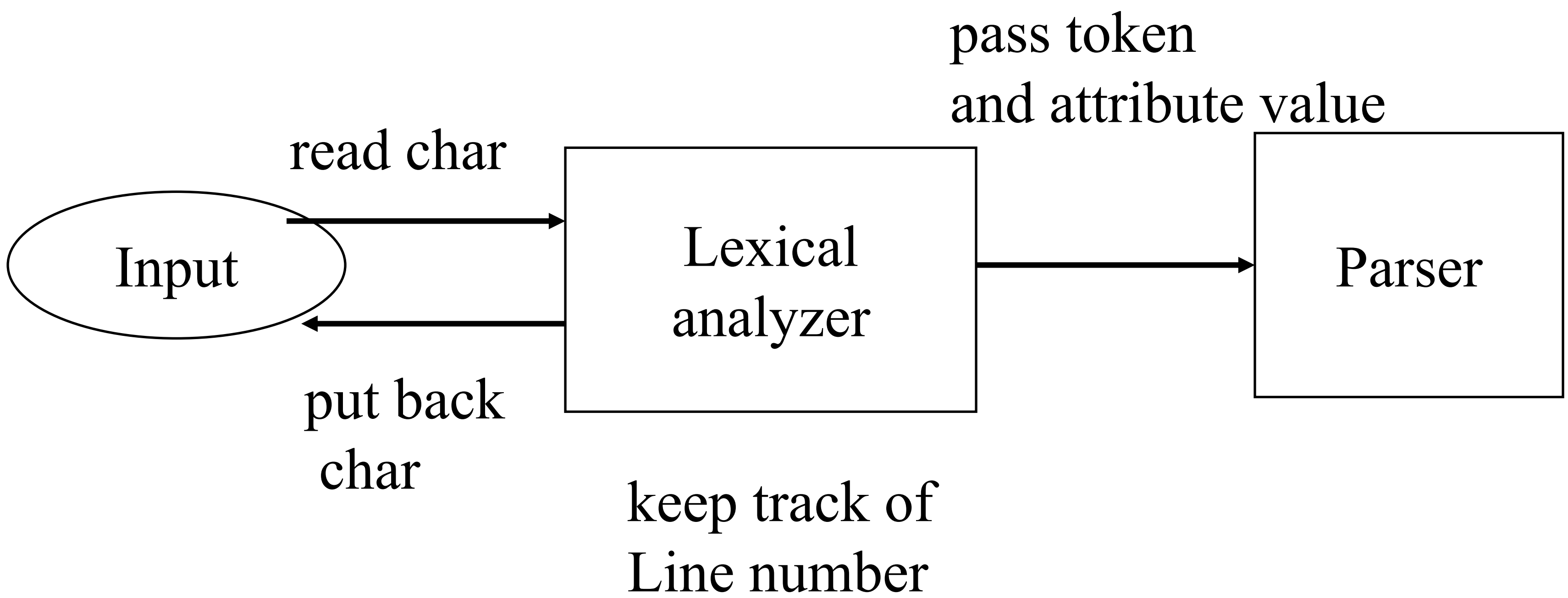
- Keyword

fixed char strings to identify certain constructs, e.g. **begin**

- Reserved word

keywords that may not be used as identifiers

- Interface to the lexical analyzer



- How to distinguish the “<“ token from the “<=“ token when the scanner read the “<“ character?

  the scanner must read ahead

- The scanner is often implemented as a procedure called by the parser, returning a token at a time.

- Input buffer

  A block of characters is read into the buffer at a time – for I/O efficiency.

  A pointer keeps track of how many characters have been analyzed.

Symbol Table

- Symbol table is a database that contains information about identifiers (procedure names, variable names, labels, … etc). It can be used to communicate among multiple compiling phases.
  - Symbol table interface

    Insert(s, t): return the index of a new entry for string s, token t.

    lookup(s): return the index of entry for string s, or 0 if not found
  - Handling reserved words

    We may initialize the symbol table by inserting all reserved words.

Symbol Table (cont.)

- Symbol table implementation

  Symbol table is probably the most complex data structure in the compiler. A good design needs to meet the following requirement:
  - Fast access
  - Easy to maintain
  - Flexible
  - Supporting nested scope

Symbol Table (cont.)

- Fast access

  Search methods: linear search, binary search, indexed search, hash table, …
- Easy to maintain

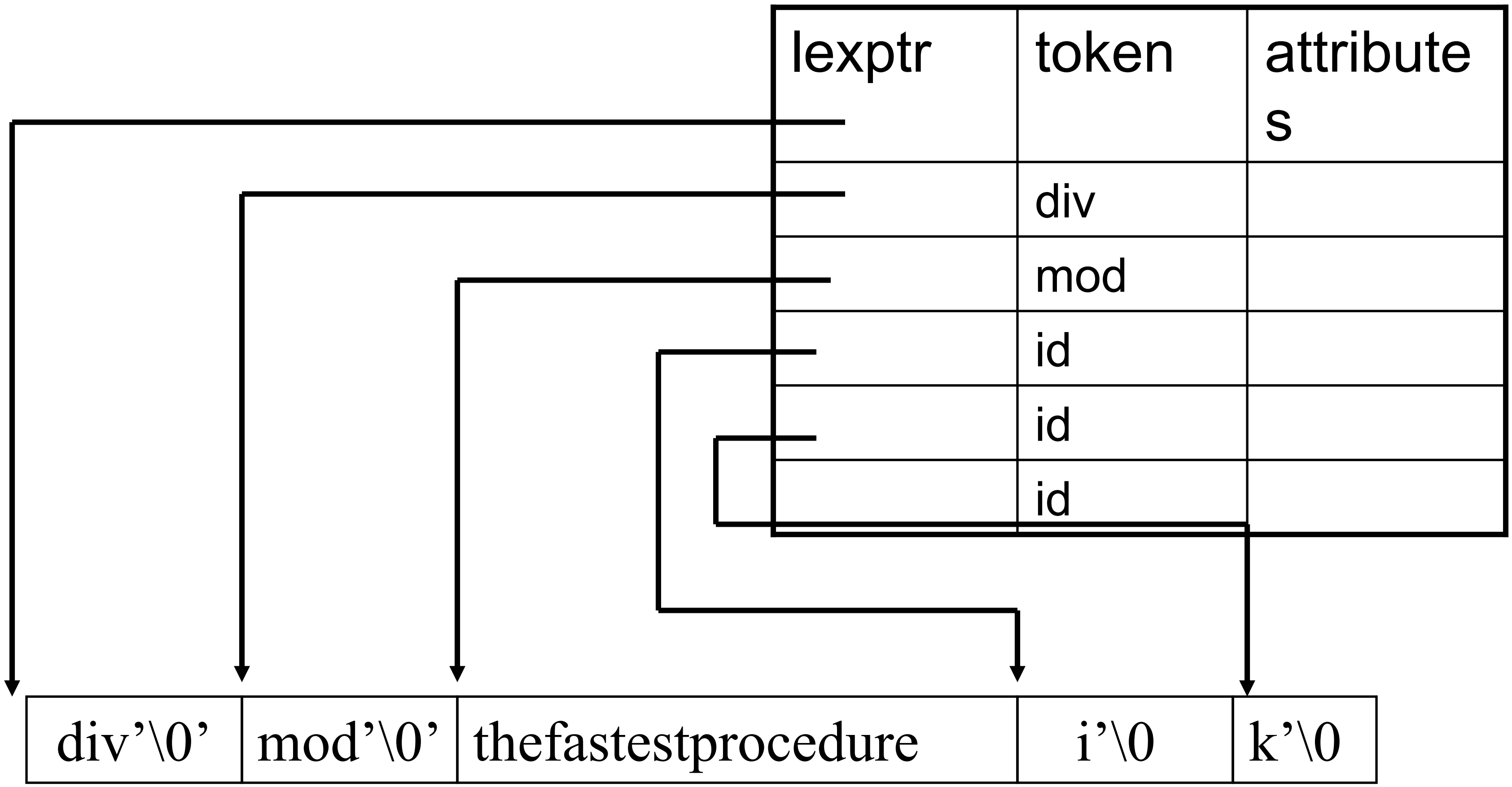
  Insertion, deletion, …
- Flexible

  Dynamic allocation, …
- Supporting nested scope

  All names in the current scope be visible, and can be taken out when the scope is closed

Symbol Table (cont.)

- A sample implementation



Abstract Stack Machine

- Most compilers use abstract stack machine for IR

Machine model

instruction memory

1 push 5
2 rvalue 2
3 +
4 rvalue 3
5 *
6 pop
7 …

Stack

7
16

Top

data memory

1	0
2	11
3	7
4	..
5	..
6	..
7	..

Instructions

Push c
Rvalue L
Lvalue L
+,-,*,/

L-value and R-value

- $A := A + 1$

  The left side A means the location where the result is stored

  The right side A means the data value stored in memory

  The term *L-value* refers to locations and *R-value* refers to values.

Translation of Expressions

Expression

A+B

rvalue A

rvalue B

+

Expression

A:= (10+B)*C +5

lvalue A

push 10

rvalue B

+

rvalue C

*

push 5

+

:=

Syntax-directed definition

- Example of definition

  stmt → id := expr

  { stmt.t := ‘lvalue’ || id.lexeme || expr.t || ‘:=‘ }

- Example of translation scheme

  stmt → id {emit(‘lvalue’, id.lexeme);}

  := expr { emit(‘:=‘);}

Assignment #1

- Need to extend the simple compiler to perform "constant folding"

Input:

100 + 25 – A;

Current output:

100

25

+

A

-

New Output:

125

A

-

Need to modify translation scheme:

For example:

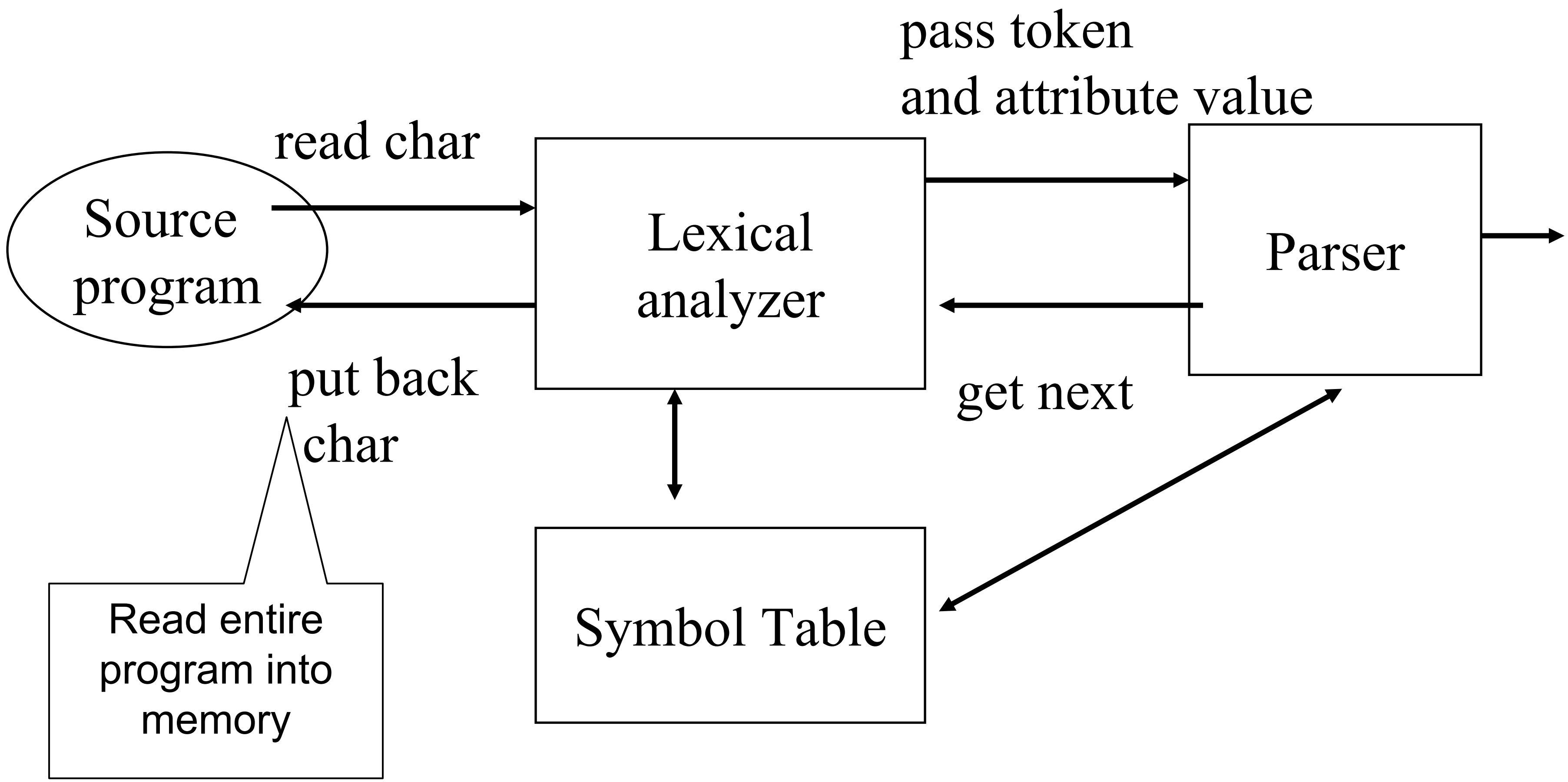
factor → (expr)
| id {print (id.lexeme);}
| num {print (num.value);}

The new compiler should not print the attributes so quickly. They shall be delayed until no opportunities for folding are observed.

Ch 3. Lexical Analysis

- How to **specify** and **implement** a lexical analyzer
- Using regular expressions (RE) to define tokens
- How to construct a lexical analyzer by hand
- Pattern-directed language: Lex
- Theory behind scanner generator: converting RE into transition table

The Role of a Lexical Analyzer



Lexical Analyzer

- Functions
  - Grouping input characters into tokens
  - Stripping out comments and white spaces
  - Correlating error messages with the source program
- Issues (why separating lexical analysis from parsing)
  - Simpler design
  - Compiler efficiency
  - Compiler portability (e.g. Linux to Win)

Typical Tokens in a PL

- Symbols

  +, -, *, /, =, <, >, ->, …
- Keywords

  if, while, struct, float, int, …
- Integer and Real (floating point) literals

  123, 123.45
- Char (string) literals
- Identifiers
- Comments
- White space

- Tokens, Patterns and Lexemes
  - Pattern: A rule that describes a set of strings
  - Token: A set of strings in the same pattern
  - Lexeme: The sequence of characters of a token

Token	Sample Lexemes	Pattern
IF	IF	IF
ID	abc, n, count,…	Letters+digit
Number	3.14, 1000	Numerical constant
;	;	;

Token Attribute

- E = C1 ** 10

Token	Attribute
ID	Index to symbol table entry E
=	
ID	Index to symbol table entry C1
**	
NUM	10

Case Study

- When blanks are not significant (as in Fortran and Algol68)

DO 5 I = 1.25

DO 5 I = 1, 25

DO5I is an ID

This is a DO loop
So 7 tokens will get generated

Case Study (cont.)

- Should 1. And .10 be legal constant?

  If yes, then how to tell

  Is 1..10 a range ? or two constants 1. And .10?

  Note that 1. And .10 are both allowed in Fortran, but not allowed in Pascal and Ada – because Pascal and Ada support ranges.

Case Study (cont.)

- When key words are not reserved words (such as in PL/1)

example 1:

IF THEN THEN THEN = ELSE ELSE ELSE = THEN;

Which THEN is an identifier?

Which THEN is the key word?

example 2:

Declare (arg1, arg2, arg3, …)

Is Declare a subroutine name or is it the key word?

Case Study (cont.)

- Assume begin and end are not reserved in Pascal.

example 3:

```
begin
   begin;
   end;
   end;
   begin;
end
```

How to parse this code fragment?

For example 1 and 2, ambiguity can be solved by multiple characters look ahead.

Lexical Error and Recovery

- Error detection
- Error reporting
- Error recovery
  - Delete the current character and restart scanning at the next character
  - Delete the first character read by the scanner and resume scanning at the character following it.
  - How about runaway strings and comments?

Regular Expressions

- Why RE?
  - Suitable for specifying the structure of tokens in programming languages

- Basic concept

  A RE defines a set of strings (called regular set)
  - Vocabulary/Alphabet: a finite character set V
  - Strings are built from V via catenation
  - Three basic operations: concatenation, alternation (|) and closure (*).

Example

- The identifier in Pascal can be defined as

  letter (letter | digit) *

- More examples
  - a | b denotes the set {a,b}
  - (a|b) (a|b) denotes the set {aa, ab, ba, bb}
  - a* denotes { e, a, aa, aaa, …}
  - (a|b)* denotes all strings of a's and b's

Regular Definition

- We may give names to regular expressions and to define regular expressions using these names

letter → A | B | C …. | a | b | c …. | z

digit → 0 | 1 | 2 | … | 9

id → letter (letter | digit) *

digits → digit digit*

Regular Definition

- Question: How is regular definition different from CFG?
- Answer:

  CFG allows recursion – a non-terminal can show up in the right hand side of itself.

  Regular Definition does not allow recursion. It is like Macro definition.

Common notations and metacharacter

- * means repeat zero or more times
- + means repeat one or more times
- ? means repeat zero or one time
- [abc] means a | b | c
  - [] form a character class which matches any char listed
  - A negate class can be specified by an upper arrow e.g. [^ab] matches any char except a and b.

Common notations (cont.)

- [a-z] denotes the RE a | b | c …. | z
- A dot matches any character, except a new line
- () used for grouping, e.g. (a|b)
- ^ (upper arrow) to anchor the pattern to the start of a line
- $ to anchor the pattern to the end of a line.

Special Characters

- The following characters have special meaning when they are inside a char class.
  - { start of macro name
  - } end of macro name
  -] end of a char class
  - \- range of characters
  - ^ negative char class
  - \ take away special meaning of the next char
- Other metacharacters, such as *,+,? are not special in a char class. For instance, [*?] matches * or ?.

Operator Precedence

<table>
<tr><th>Operator</th><th>Description</th><th>Level</th></tr>
<tr><td>()</td><td>Grouping</td><td>1</td></tr>
<tr><td>[]</td><td>Char class</td><td>2</td></tr>
<tr><td>*+?</td><td>Repeat</td><td>3</td></tr>
<tr><td>Catenation</td><td>Catenation</td><td>4</td></tr>
<tr><td>|</td><td>Or</td><td>5</td></tr>
<tr><td>^ $</td><td>Anchor to the beginning or the end of a line</td><td>6</td></tr>
</table>

Notation Examples

- [a-z]
- [z-a9-0] this is ok
- [a\-z]
- [a^b]
- [^a-z]
- [0-z] non-portable warning
- ({letter})*
- ab*

Exercise

- How to match any char?

  (. | \n)

- Specify a char class of three chars: (,), and -.

  [()-] or [-()],
  but not [(-)]

More Token Definition Examples

- Token While

  while = “while”

- White space

  WhiteSpace = [\t]+

  newline is often treated differently to track line number

- Comments that begin with – and end with \n

  comment = --.*\n

- Fixed decimal constants

  RealConstant= digits\.digits (how about digit*\.digit* ?)

Exercise

- Define an identifier, composed of letters, digits and underscores. It must begins with a letter, ends with a letter or digit, and no consecutive underscores.

  letter → [A-Za-z]

  digit → [0-9]

  letter (_?(letter|digit))*

- Define floating point constants. It can be any of the following forms

  1.25e12 .25 .12e+10 12.5e-5 12e6

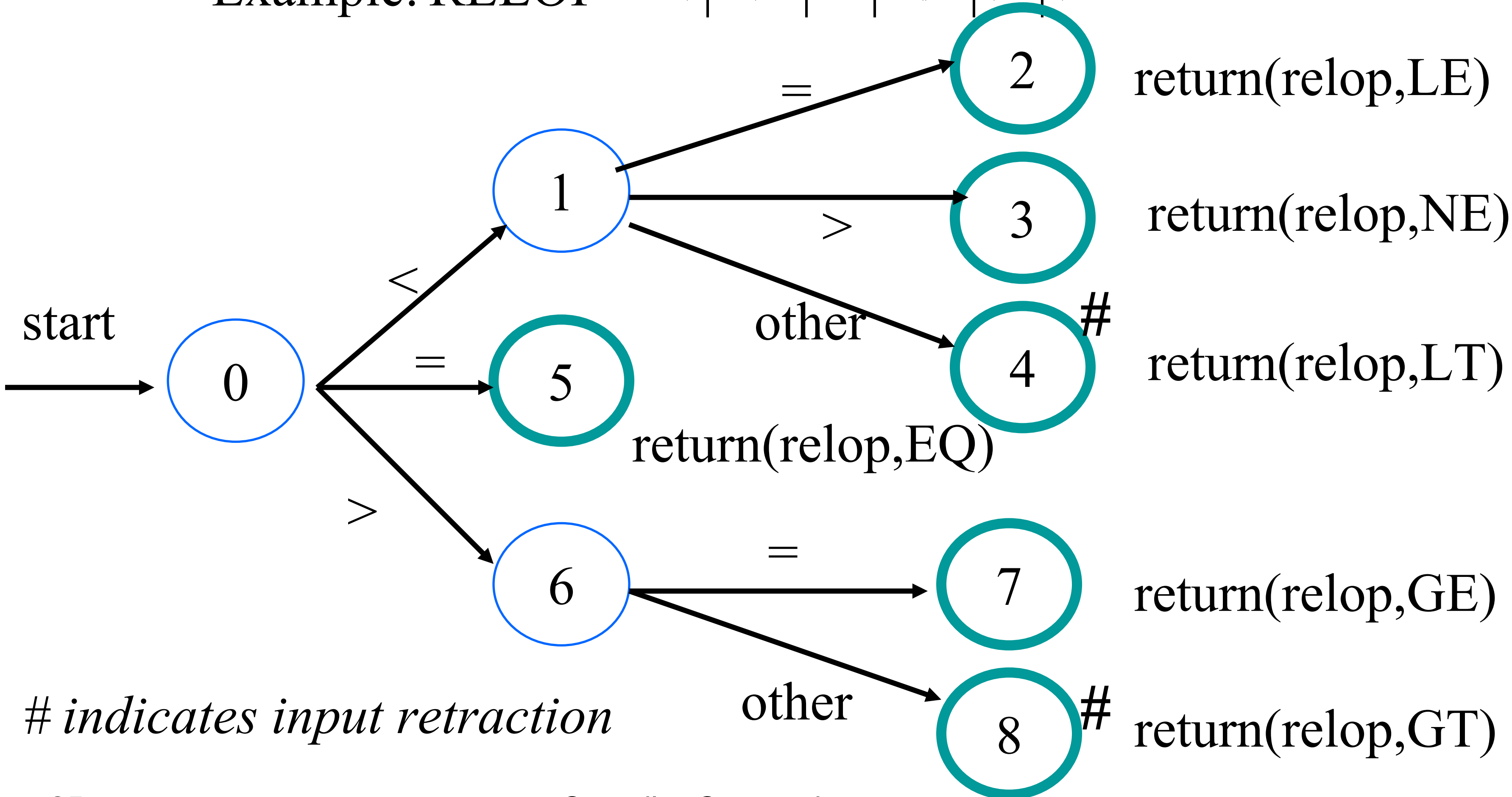
Non-regular set

- RE can denote a fixed number of unspecified number of repetitions of a given construct. Its best use is for describing identifiers, constants, … etc.
- RE can not be used to describe balanced or nested structures, such as nested loops, nested if-then-else.

Recognition of Tokens

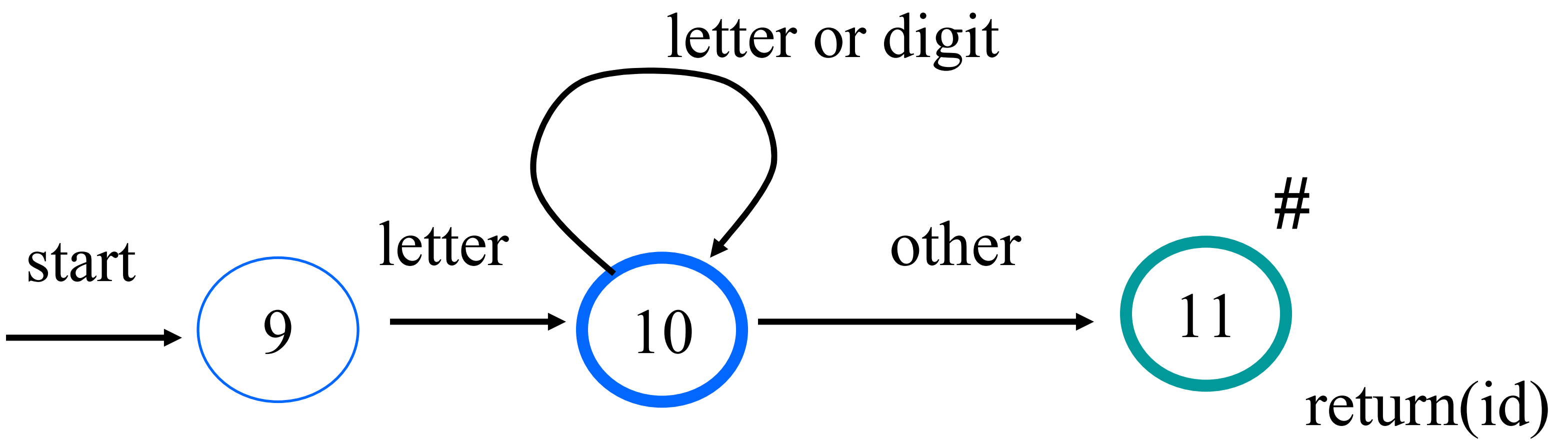
- Transition Diagram

  Example: RELOP = < | <= | = | <> | > | >=



- Transition Diagram

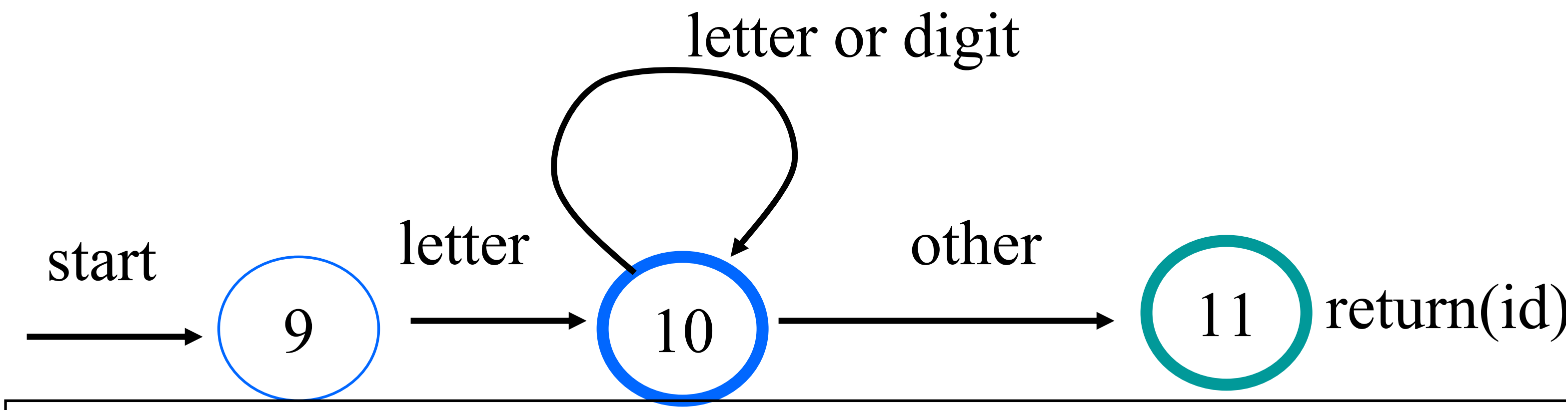
  Example: ID = letter(letter | digit) *



 # indicates input retraction

Implementing Transition Diagram

- Mapping transition diagrams into C code



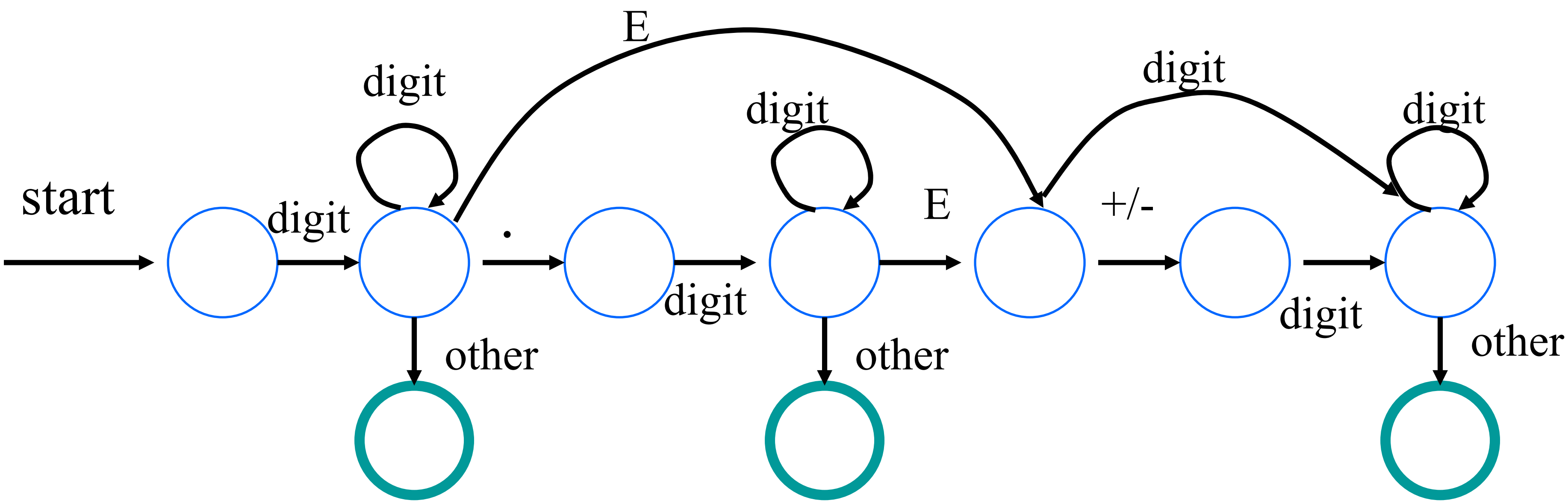

```
switch (state) {
…
case 9: c = nextchar();
        if (isletter( c) ) state = 10; else state =
failure();
        break;
case 10: ….
case 11: retract(1); insert(id); return;
```

Lexical analyzer loop

```
Token nexttoken() {
while (1) {
   switch (state) {
   case 0:       c = nextchar();
                 if (c is white space) state = 0;
                 else if (c == ‘<‘) state = 1;
                 else if (c == ‘=‘) state = 5;
                 …
   case 9:       c = nextchar();
                 if (isletter( c) ) state = 10; else state =fail();
                 break;
   case 10:      ….
   case 11:      retract(1); insert(id);
                 return;
```

RE with multiple accepting states

NUM = digit+ (.digit+)? (E(+|-)? digit+)?



Accepting integer
e.g. 12

Accepting float
e.g. 12.31

Accepting float
e.g. 12.31E4

RE with multiple accepting states

- Two ways to implement:
  - Implement it as multiple regular expressions.

    each with its own start and accepting states. Starting with the longest one first, if failed, then change the start state to a shorter RE, and re-scan. See example of Fig. 3.15 and 3.16 in the textbook.
  - Implement it as a transition diagram with multiple accepting states.

    When the transition arrives at the first two accepting states, just remember the states, but keep advancing until a failure is occurred. Then backup the input to the position of the last accepting state.

Transition Table and Driver

- The transition diagram can be naturally implemented by a transition table.

  Table[state] [c]

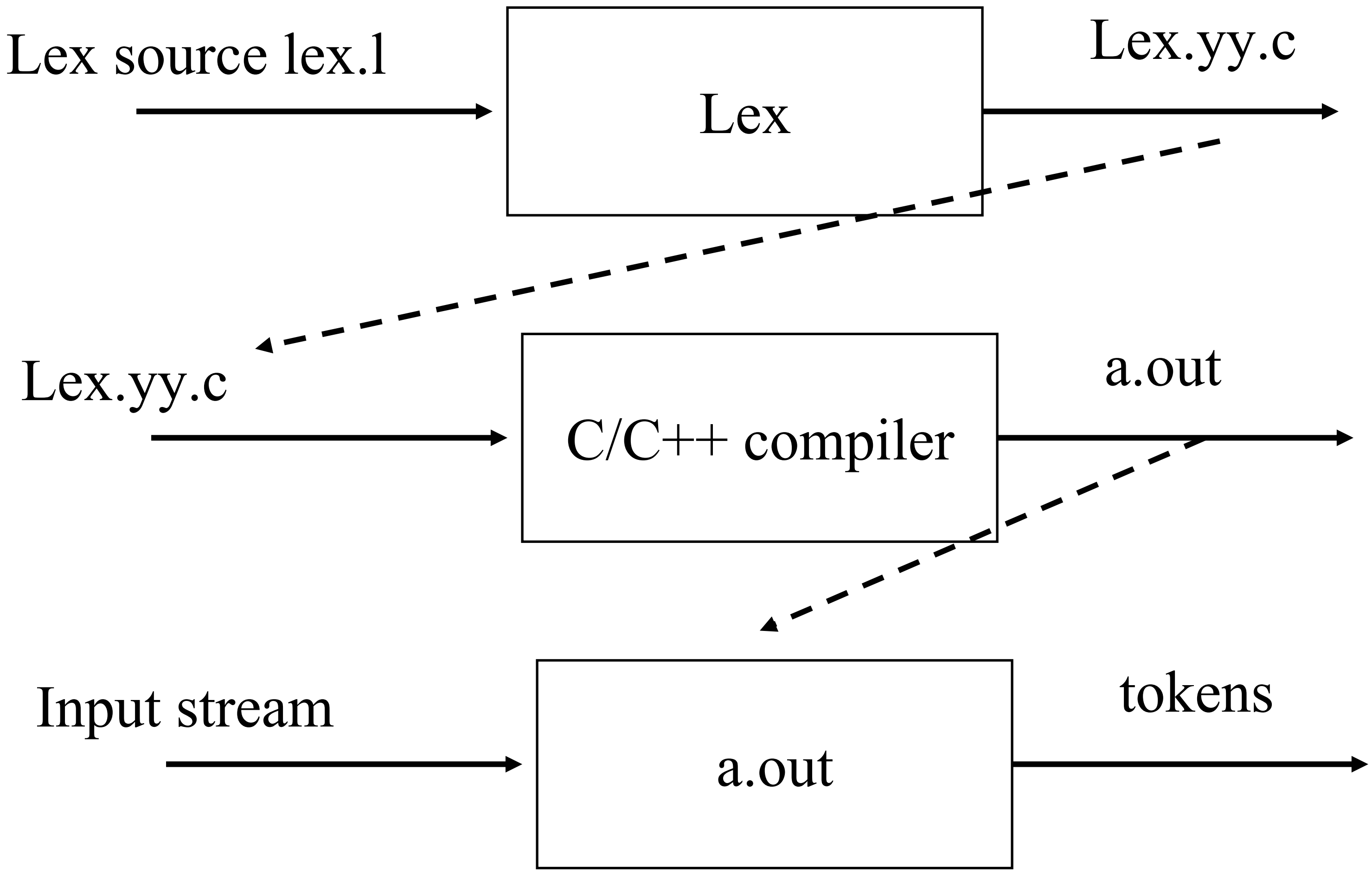
  The driver looks as follows:

```
While (not eof) {
      new_state = Table [current_state ] [c ]
      c = nextchar();
      current_state = new_state;
}
```

LEX – A Language for Specifying Lexical Analyzers

- Lex is a lexical analyzer generator
  - Implemented by Lesk and Schmidt of Bell Lab initially for Unix
  - Not only a table generator, but also allows "actions" to associate with RE's.
  - Lex is widely used in the Unix community
  - Lex is not efficient enough for production compilers, however, Flex (Fast Lex) is an improved version.

Lex Usage


Lex source lex.l
Lex
Lex.yy.c
Lex.yy.c
C/C++ compiler
a.out
Input stream
a.out
tokens

Lex Specification

- Declaration
  - Variables, constants
  - Regular definitions to define character class and auxiliary regular expressions
- Translation rules
  - Regular expression-1 {action 1}
- Auxiliary procedures
  - Routines needed by actions
  - Symbol table routines

Example

```
%{
#define THEN 5
#define ID      100
%}
WS      [ \t]+
letter  [A-Za-z]
digit   [0-9]
id      {letter}( {letter} | {digit})*
%%
{WS}    {}
{id}    {yylval = insert_id(); return(ID);}
%%
Insert_id() {…}
```

Definition section
%%
Rules section
%%
Subroutine section

To pass attributes to the parser, a global variable yylval is often used.

Overlapped Regular Expressions

- Lex allows RE’s to overlap. In the case of overlapped RE’s, two rules apply:
  - Longest possible match, for example, how to get “<=“ token rather than “<“ and “=“.
  - Order of rules. If two RE’s match the same string, the earlier rule is preferred. So ***if8*** is an identifier while ***if*** is a reserved word. (assuming if is defined as a token by RE).

Summary

- Learn how to specify and implement a lexical analyzer
- Precise specification: RE
- Map RE to transition diagram, and map transition diagrams to code
- LEX – A pattern-directed language and a scanner generator

Lookahead Operator

- In Lex/Flex, “/” is a special lookahead operator. It supports some PL constructs that need to look ahead beyond the end of a lexeme to determine a token. For example, a pattern “RE1 / RE2” matches RE1 only if it is followed by RE2.
- Example: how to distinguish IF(I) = 3 from a regular IF statement, such as IF(I) A=B or IF(I) THEN …

```
IF / \( .*\) {letter}    or
IF / {ws}”(“.*”)”{ws}{letter}
```

Special Variables/Procedures

- **yytext** where token text is stores
- **yyleng** length of the token text
- **yylineno** the current line number
- **yylex()** name of procedure for the lex generated scanner
- **yywrap** A user supplied function. It returns 1 when no more input to process, otherwise, return 0
- **yyin, yyout** input and output files

Additional Notes

- [^a-z] negate the char class will also match a new line "\n". This is true in Lex and Flex, but not necessary true in other implementation of RE.
- ECHO copies yytext to output
- If you use Lex, link with lex library –lf, if you use Flex, link with library –lfl.
- EOF is not handled by RE, it is signaled by having yylex() to return integer 0.

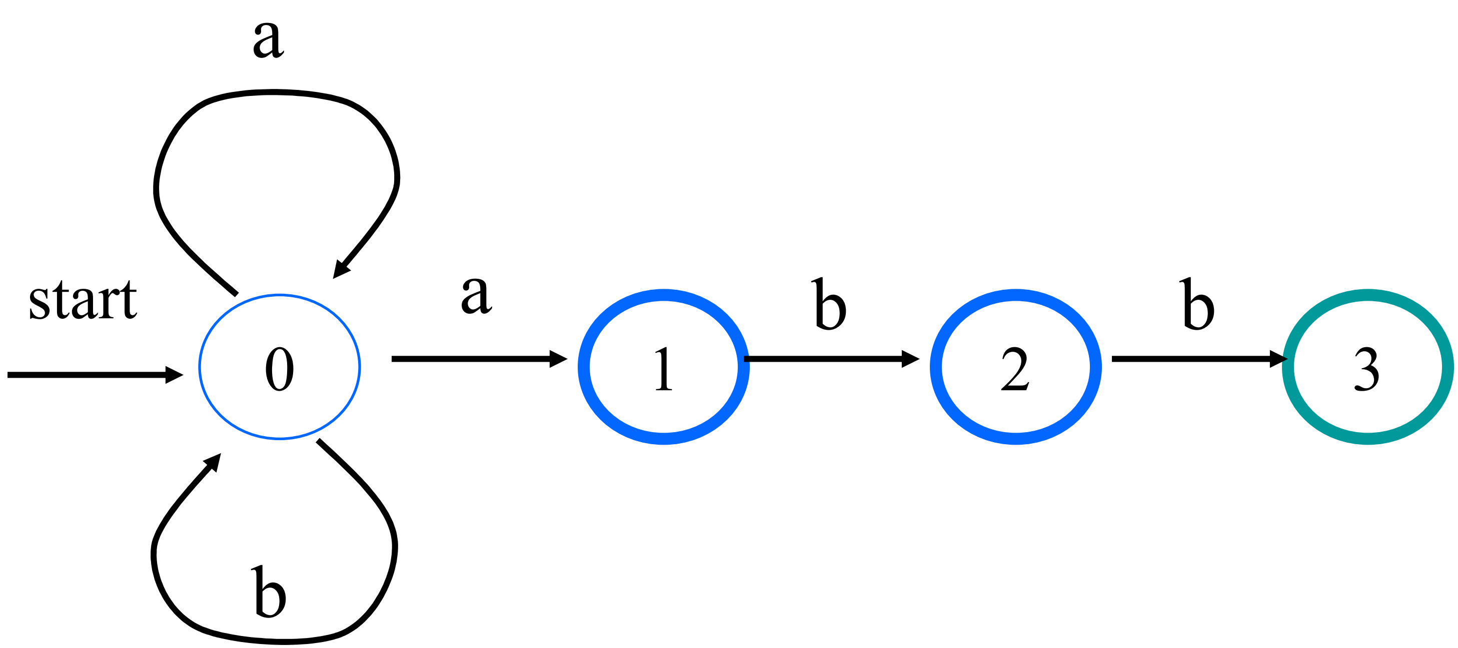
Lexical Analyzer Generator

- Lexical analyzer generator is to transform RE into a state transition table (i.e. Finite Automation)
- Theory of such transformation
- Some practical consideration

Finite Automata

- Transition diagram is finite automation
- Nondeterministic Finite Automation (NFA)
  - A set of states
  - A set of input symbols
  - A transition function, *move(),* that maps state-symbol pairs to sets of states.
  - A start state S0
  - A set of states F as accepting (Final) states.

Example



The set of states = {0,1,2,3}
Input symbol = {a,b}
Start state is S0, accepting state is S3

Transition Function

- Transition function can be implemented as a transition table.

State	Input Symbol	
	a	b
0	{0,1}	{0}
1	--	{2}
2	--	{3}

- Non-deterministic Finite Automata (NFA)
  - An NFA accepts an input string *x* iff there is a path in the transition graph from the start state to some accepting (final) states.
  - The language defined by an NFA is the set of strings it accepts
- Deterministic Finite Automata (DFA)
- A DFA is a special case of NFA in which
  - There is no e-transition
  - Always have unique successor states.

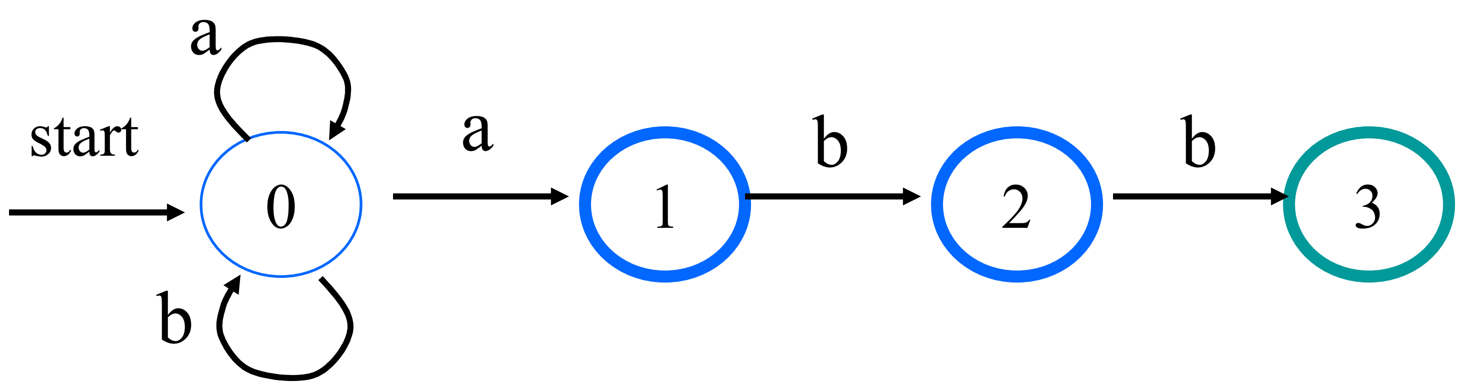
– How to simulate a DFA

```
s = s0; c := nextchar;
while ( c <> eof) do
        s := move(s, c);
        c := nextchar;
end
if (s in F) then return “yes”
```

Conversion of NFA to DFA

- Why?
  - DFA is difficult to construct directly from RE's
  - NFA is difficult to represent in a computer program and inefficient to compute
- Conversion algorithm: subset construction
  - The idea is that each DFA state corresponds to a set of NFA states.
  - After reading input a1, a2, …, an, the DFA is in a state that represents the subset T of the states of the NFA that are reachable from the start state.

NFA to DFA conversion



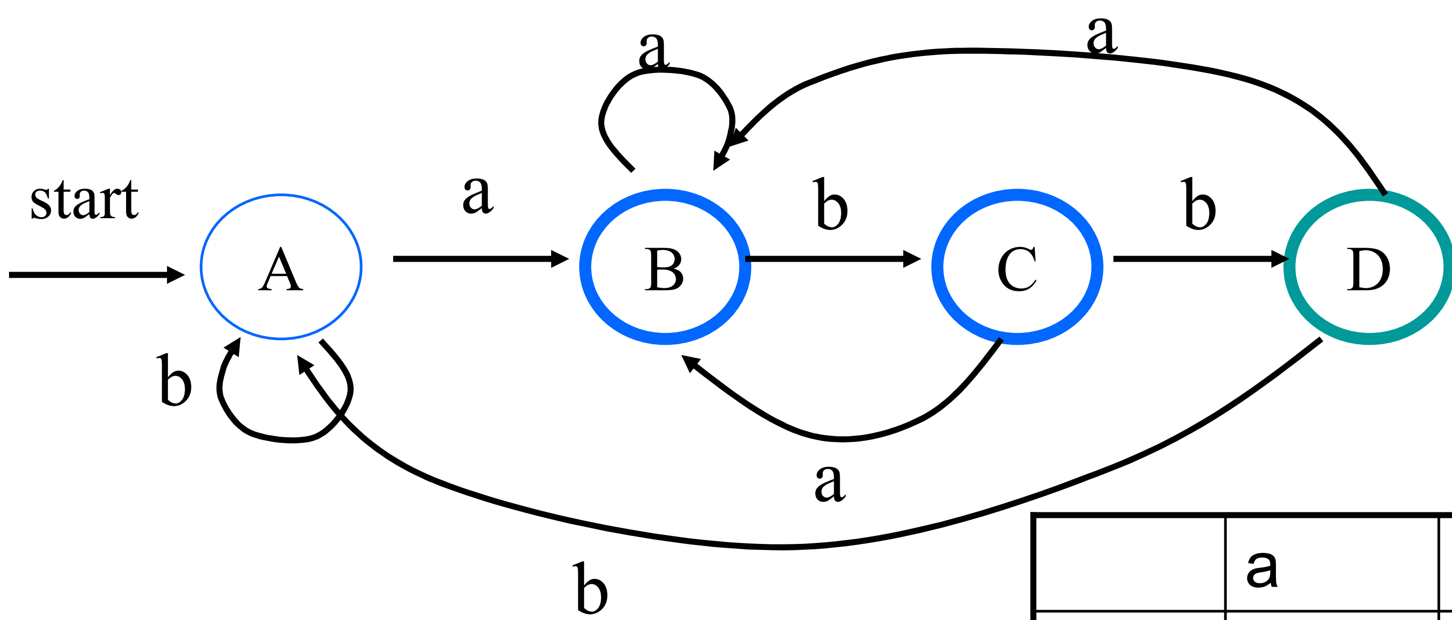
$(0,a) = \{0,1\}$
$(0,b) = \{0\}$
$(\{0,1\}, a) = \{0,1\}$
$(\{0,1\}, b) = \{0,2\}$
$(\{0,2\}, a) = \{0,1\}$
$(\{0,2\}, b) = \{0,3\}$

New states

$A = \{0\}$
$B = \{0,1\}$
$C = \{0,2\}$
$D = \{0,3\}$

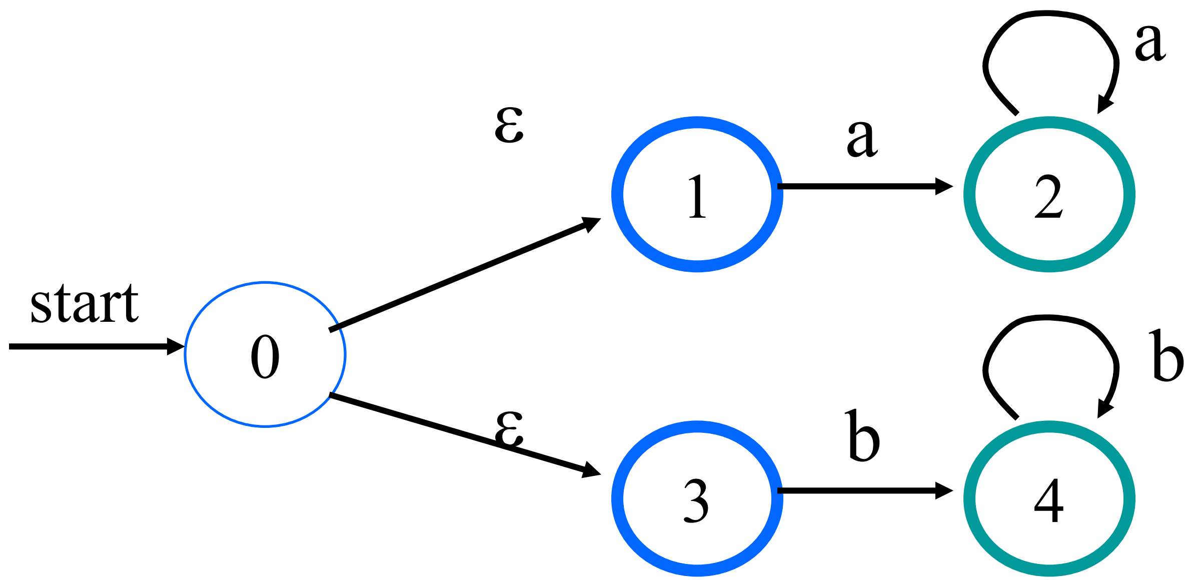
	a	b
A	B	A
B	B	C
C	B	D
D	B	A

NFA to DFA conversion (cont.)



	a	b
A	B	A
B	B	C
C	B	D
D	B	A

NFA to DFA conversion (cont.)



How about e-transition?

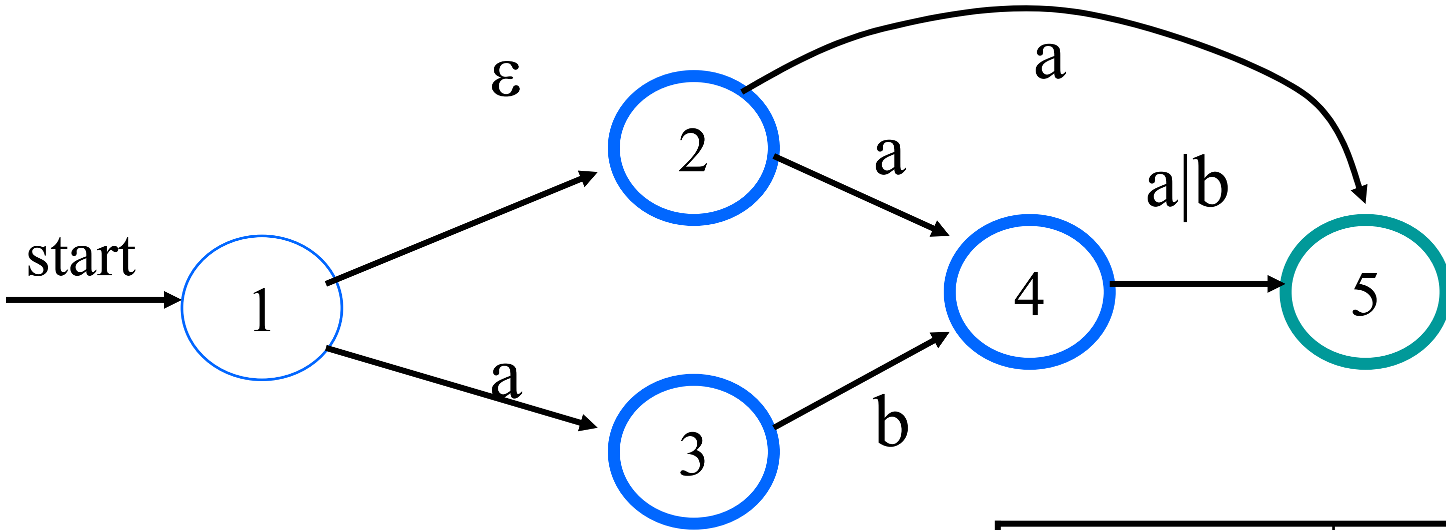
Due to e-transitions, we must compute e-closure(S) which is the set of NFA states reachable from NFA state S on e-transition, and e-closure(T) where T is a set of NFA states.

Example: e-closure (0) = {1,3}

Subset Construction Algorithm

```
Dstates := e-closure (s0)
While there is an unmarked state T in Dstates do
begin
      mark T;
      for each input symbol a do
      begin
         U := e-closure ( move(T,a) );
         if  U is not in Dstates then
               add U as an unmarked state to Dstates;
         Dtran [T, a] := U;
      end
end
```

Example



Dstates := ε-closure(1) = {1,2}

U:= ε-closure (move({1,2}, a)) = {3,4,5}

Add {3,4,5} to Dstates

U:= ε-closure (move({1,2}, b)) = {}

ε-closure (move({3,4,5}, a)) = {5}

ε-closure (move({3,4,5}, b)) = {4,5}

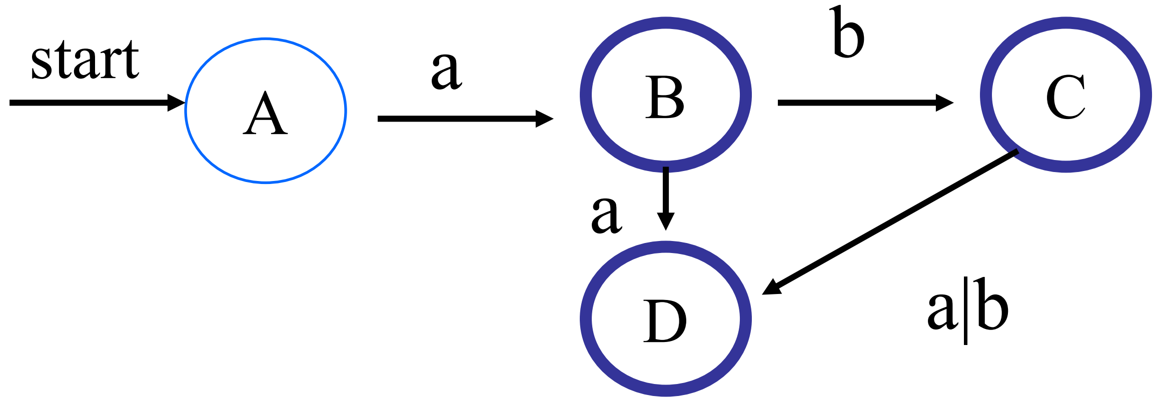
ε-closure (move({4,5}, a)) = {5}

ε-closure (move({4,5}, b)) = {5}

	a	b
A{1,2}	B	--
B{3,4,5}	D	C
C{4,5}	D	D
D{5}	--	--

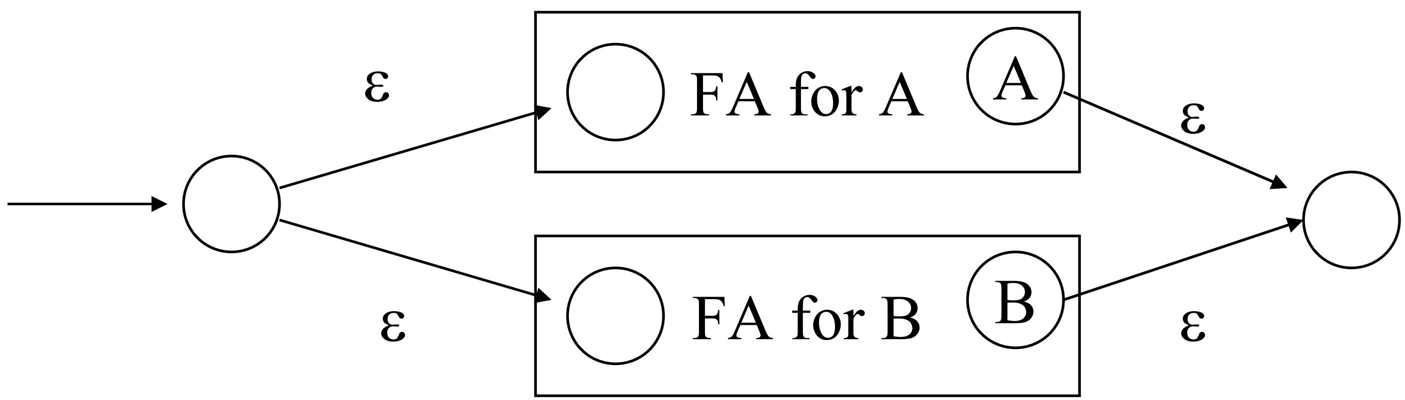
DFA after conversion

	a	b
A{1,2}	B	--
B{3,4,5}	D	C
C{4,5}	D	D
D{5}	--	--



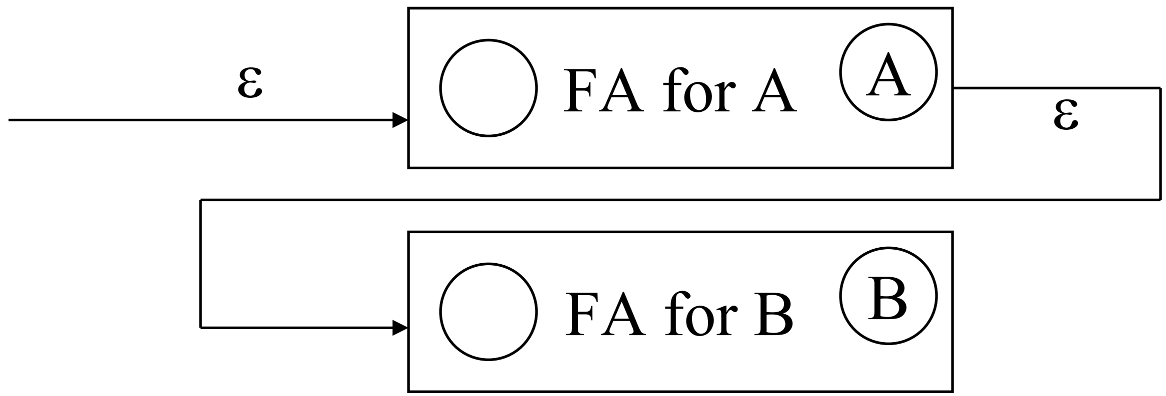
Map RE to NFA

- Three basic operations
  - All RE's are built out of atomic regular expressions by using concatenation, alternation and closure.
- Constructing A|B



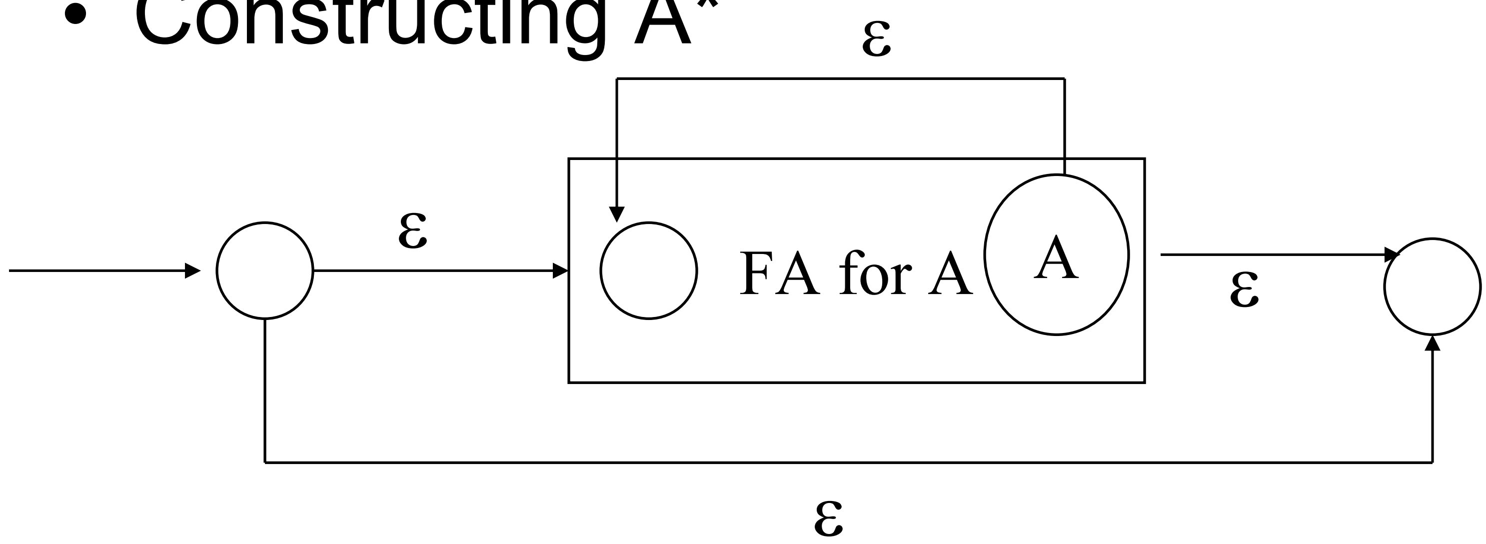
Map RE to NFA

- Constructing AB



Map RE to NFA

- Constructing A*

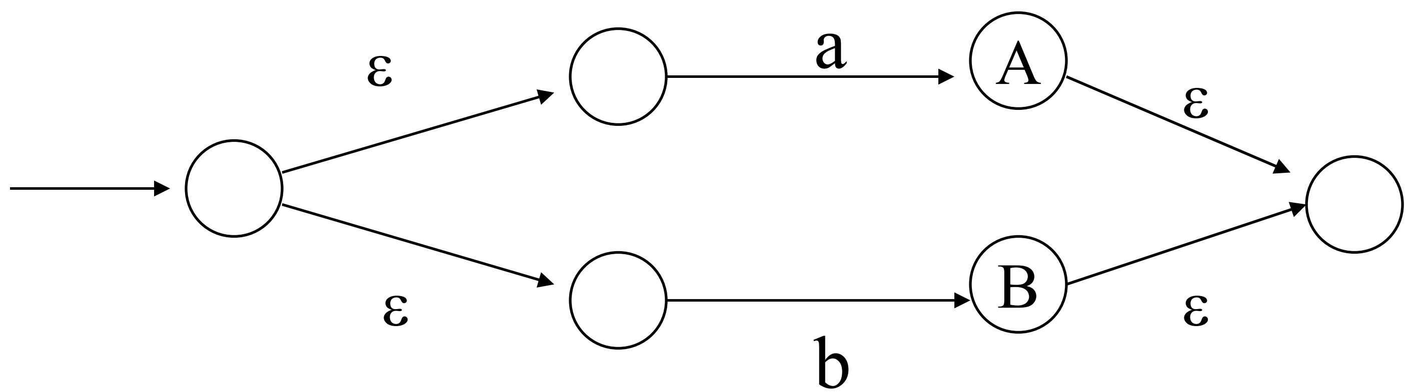


Example

Constructing NFA for regular expression

r = (a|b)*abb

Step 1: constructing a | b

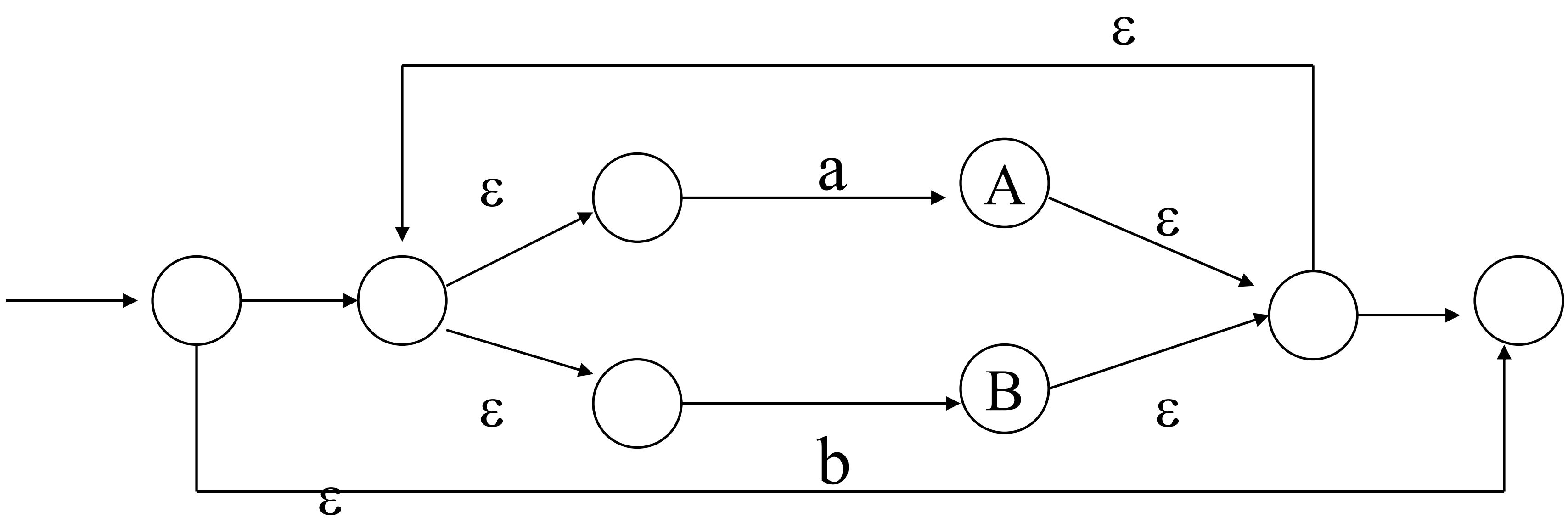


Example

Constructing NFA for regular expression

$r = (a|b)^*abb$

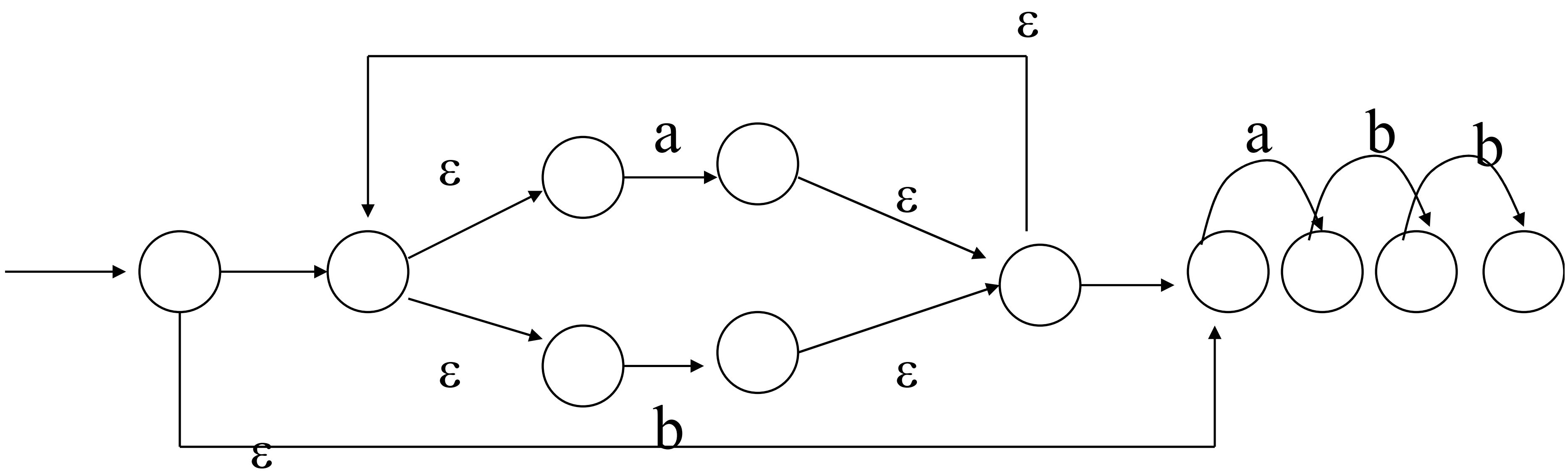
Step 2: constructing (a | b)*



Example

Constructing NFA for regular expression
r = (a|b)*abb

Step 3: catenate with abb



Simulation of NFA

- Given an NFA *N* and an input string *x*, determine whether N accepts *x*

```
S:= e-closure({s0}) ;  a := nextchar;
While a <> eof do begin
      S:= e-closure(move(S,a));
      a:= nextchar;
end
if (an accepting state s in S, return(yes)
otherwise return (no)
```

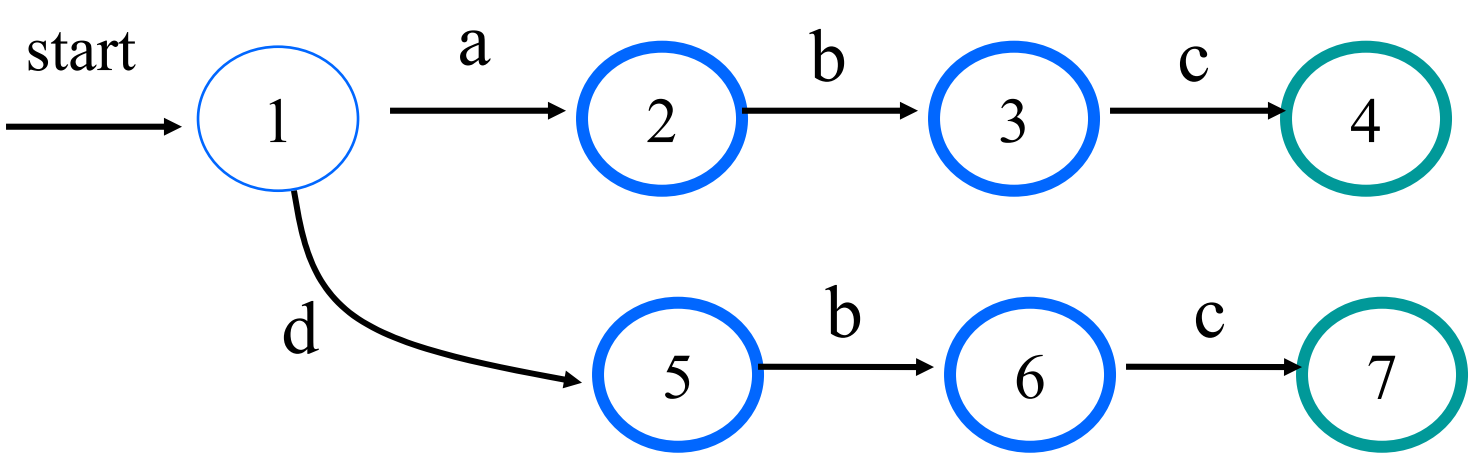
Time Space Tradeoffs

Given regular expression r, and string x.

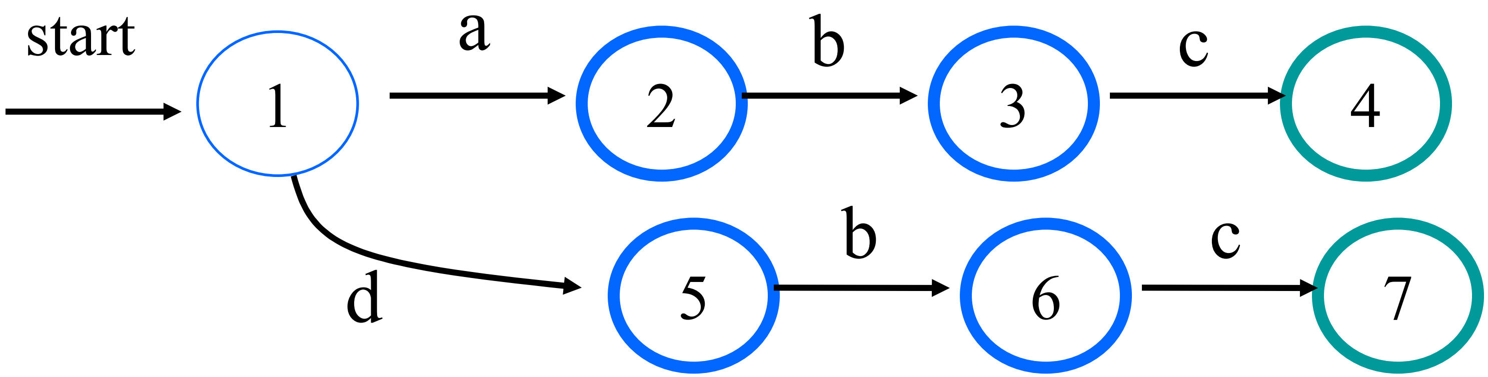
Automation	Space	Time
NFA	O (| r |)	O (|r| X |x|)
DFA	O (2 exp(|r|))	O(| x|)
Lazy transition	~NFA	~DFA

Optimizing Finite Automata

- Minimizing the number of states
  - For every DFA, there is a unique smallest equivalent DFA (that accept the same set of strings).



Optimizing Finite Automata



	a	b	c	d
1	2			5
2		3		
3			4	
4				
5		6		
6			7	
7				

State 2 and state 5
State 3 and 6
State 4 and 7

Are equivalent!

- We may begin by trying the most aggressive (or optimistic) merge by creating only two states: final state and non-final state. We then split the states.

- Algorithm:

  Repeat

  Let S be any merged state {s1, s2, ..,sn}, and c be any input symbol.

  Let t1, t2, … tn, be the successor state to {s1, .. sn} under c, if t1, .. tn do not all belong to the same merged state then split S into new states so that si and sj remain in the same merged state iff ti and tj are in the same merged state.

  Until no more splits are possible.

Optimizing Finite Automata

	a	b	c	d
1	2			5
2		3		
3			4	
4				
5		6		
6			7	
7				

Start with two states

Non-final = {1,2,3,5,6}

Final = {4,7}

Optimizing Finite Automata

	a	b	c	d
1	2			5
2		3		
3			4	
4				
5		6		
6			7	
7				

Start with two states
Non-final = {1,2,3,5,6}
Final = {4,7}

Split non-final as
{1} {2,3,5,6}
Final = {4,7}

For input b, the successor state for 2, and 3 are not in the same merged state, so we nee to split more

Optimizing Finite Automata

	a	b	c	d
1	2			5
2		3		
3			4	
4				
5		6		
6			7	
7				

Non-final is
{1} {2,3,5,6}
Final = {4,7}

Non-final is
{1} {2,5} {3,6}
Final = {4,7}

No more split! Job done.

Optimizing Finite Automata

	a	b	c	d
1	2			5
2		3		
3			4	
4				
5		6		
6			7	
7				

	a	b	c	d
A	B			B
B		C		
C			D	
D				

A = {1}
B = {2,5}
C = {3,6}
D = {4,7}

Optimizing Finite Automata

- Table Compaction
  - Two dimensional arrays provide fast access
  - Table size may be a concern (10KB to 100KB)
  - Table compression techniques
    - Compressing by eliminating redundant rows
    - Pair-compressed transition tables

- A typical transition table has many identical columns and some identical rows.

	a	b	c	…	1	2	…	=
1	1	1	1		2	2		5
2	1	1	1		2	2		5
3	2	2	2		3	3		4
4	3	3	3		4	4		4
5	2	2	2		3	3		4
6	3	3	3		4	4		4
7	4	4	4		3	3		6
8	4	4	4		3	3		6

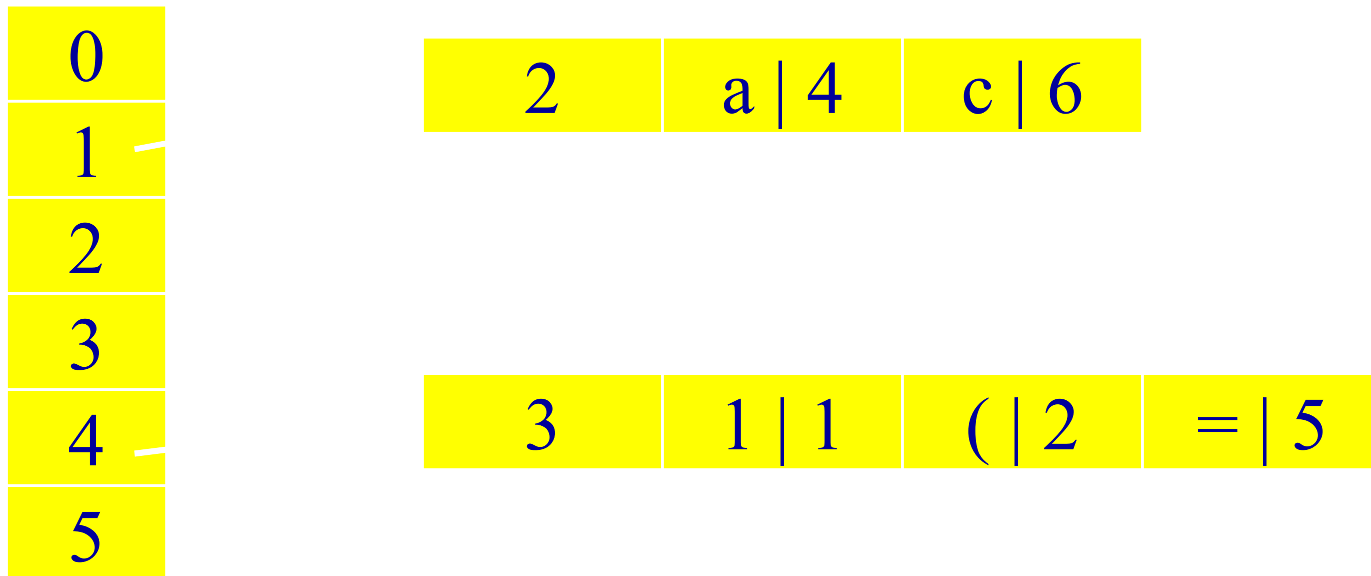
We may create a much smaller transition table with indirect row and column maps. Table is now accessed as T[rmap[s], cmap[c]].

a b c …. 1 2 3 …. =
1 1 1….. 2 2 2 ….. 3

1 1
2 1
3 2
4 3
5 4
6 3
7 5
8 5

1	2	5
2	3	4
3	4	4
2	4	7
4	3	6

Sparse table techniques



Summary

- Finite Automata, NFA, DFA
- Converting NFA to DFA – subset construction
- From RE to NFA – 3 basic operations
- Time space tradeoffs
- Optimizations: minimize the number of states and table compression

Summary

- Learn how to specify and implement a lexical analyzer
- Precise specification: RE
- Map RE to transition diagram, and map transition diagrams to code
- LEX – A pattern-directed language and a scanner generator

Assignment#2

- Using Lex/Flex to write a scanner for a subset of language C
- Define tokens in RE's
- Handle identifiers and interact with a symbol table
- Experience with RE for comments