



CS49K Programming Languages

Chapter 2 Programming Language
Syntax - Sec. 2.2 – 2.3.1

LECTURE 3: SCANNERS AND OVERVIEW OF PARSERS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

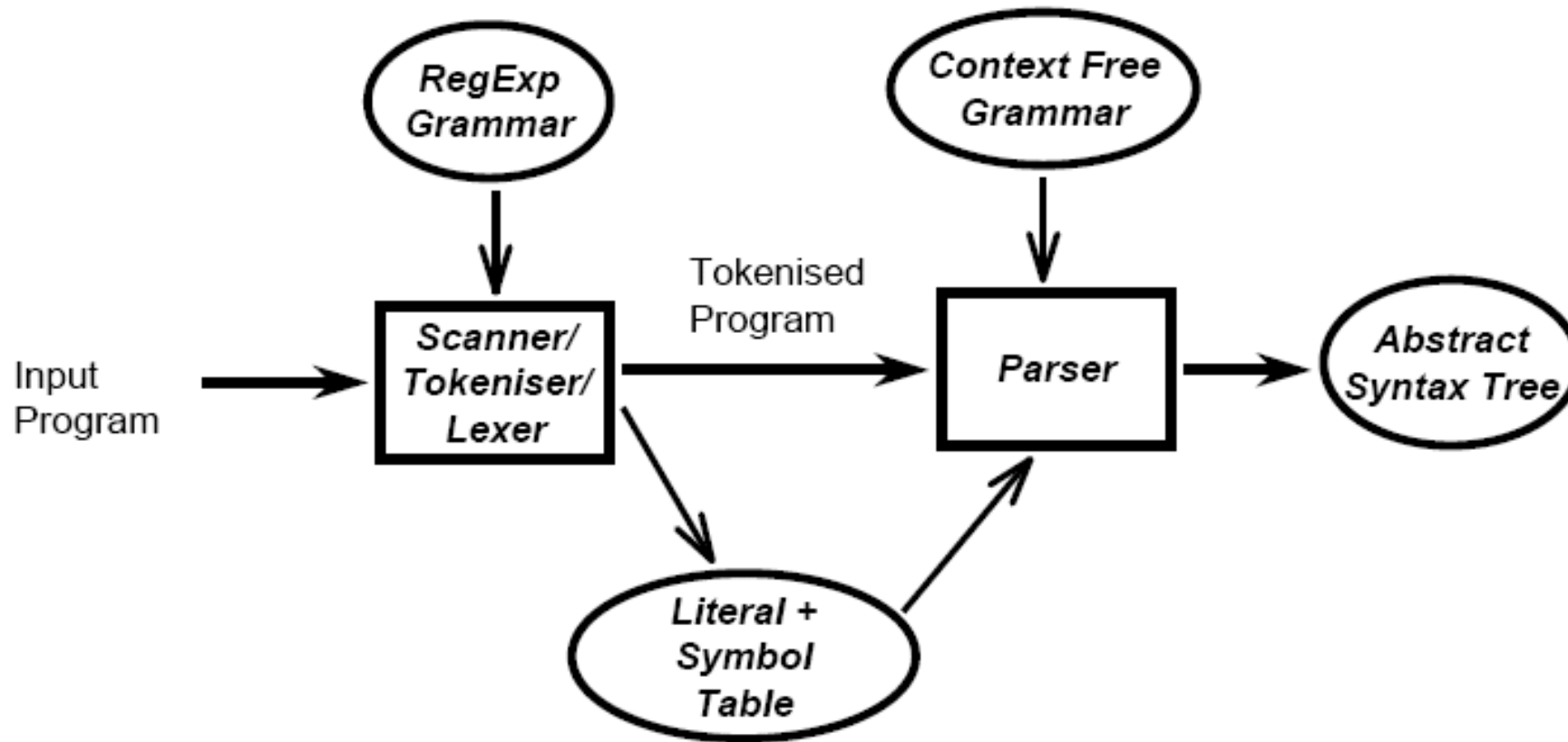
- Scanner (Tokenizer)
- From Regular Expression to Non-Deterministic Finite Automata
- From Non-Deterministic Finite Automata
- DFA Minimization
- Scanner Styles
- Overview of Parsers

Scanning I

Lexical Analyzer and Scanner

SECTION 1

Syntax Analysis



Calculator Language

Exemplary Language

C-Style Comment Line:

comment \longrightarrow */* (non-* | * non-/)^{*} ** /*
| // (non-newline)^{} newline*

assign \longrightarrow *:=*

plus \longrightarrow *+*

minus \longrightarrow *-*

times \longrightarrow ***

div \longrightarrow */*

lparen \longrightarrow *(*

rparen \longrightarrow *)*

id \longrightarrow *letter (letter | digit)^{*}*

except for **read** and **write**

number \longrightarrow *digit digit^{*} | digit^{*} (. digit | digit .) digit^{*}*



Rules with Terminal

Recall scanner is responsible for

- tokenizing source
- removing comments
- (often) dealing with pragmas (i.e., significant comments)
- saving text of identifiers, numbers, strings
- saving source locations (file, line, column) for error messages

Scanning

- Suppose we are building an ad-hoc (hand-written) scanner for Pascal:
 - We read the characters one at a time with look-ahead
- If it is one of the one-character tokens
 { () [] < > , ; = + - etc }
we announce that **token**
- If it is a ., we look at the next character
 - If that is a dot, we announce .
 - Otherwise, we announce . and reuse the look-ahead
 - (field selector a.b/range delimiter ..)

Scanning

- If it is a $<$, we look at the next character
 - if that is a $=$ we announce $<=$
 - otherwise, we announce $<$ and reuse the look-ahead, etc
- If it is a letter, we keep reading letters and digits and maybe underscores until we can't anymore
 - then we check to see if it is a reserve word

If it is a digit, we keep reading until we find a non-digit

- if that is not a . we announce an **integer**
- otherwise, we keep looking for a real number
- if the character after the . is not a digit we announce an integer and reuse the . and the look-ahead

This is a deterministic finite automaton (DFA)

- Lex, scangen, etc. build these things automatically from a set of regular expressions
- Specifically, they construct a machine that accepts the language

```
identifier | int const  
| real const | comment | symbol | ...
```

Scanning

- Scanners tend to be built three ways
 - ad-hoc
 - semi-mechanical pure DFA
(usually realized as nested case statements)
 - table-driven DFA
- Ad-hoc generally yields the fastest, most compact code by doing lots of special-purpose things, though good automatically-generated scanners come very close

```

skip any initial white space (spaces, tabs, and newlines)
if cur_char ∈ {'(', ')', '+', '-', '*'}
    return the corresponding single-character token
if cur_char = ':'
    read the next character
    if it is '=' then return assign else announce an error
if cur_char = '/'
    peek at the next character
    if it is '*' or '/'
        read additional characters until "*" or newline is seen, respectively
        jump back to top of code
    else return div
if cur_char = .
    read the next character
    if it is a digit
        read any additional digits
        return number
    else announce an error
if cur_char is a digit
    read any additional digits and at most one decimal point
    return number
if cur_char is a letter
    read any additional letters and digits
    check to see whether the resulting string is read or write
    if so then return the corresponding token
    else return id
else announce an error

```

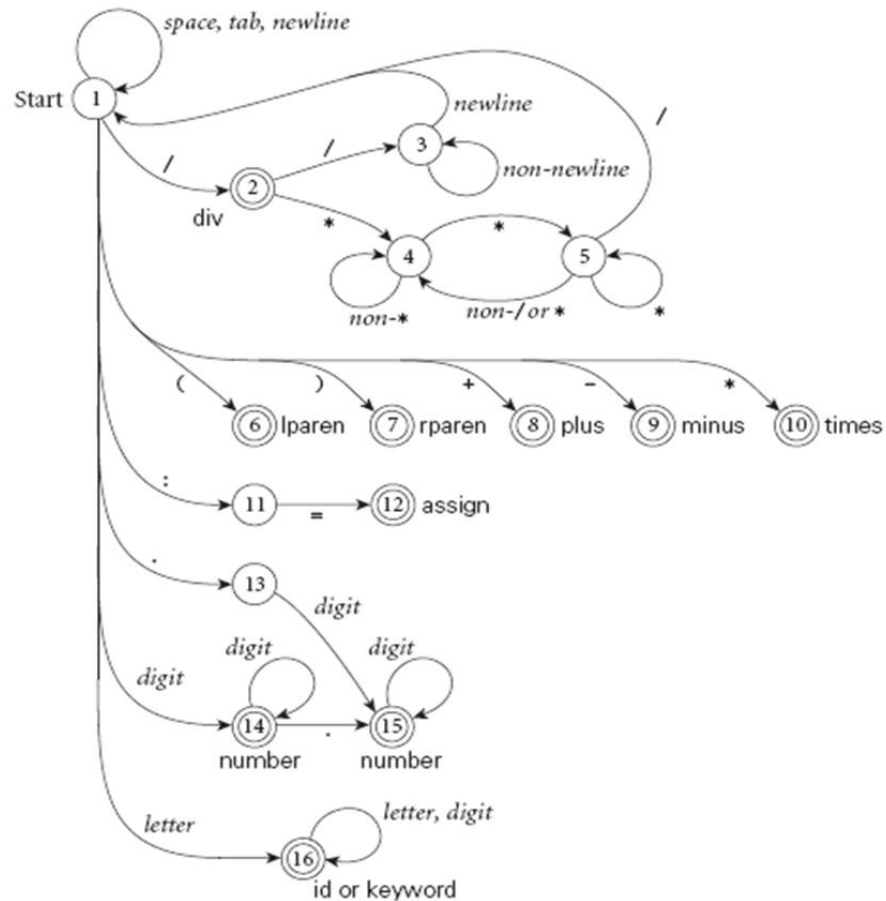
An Ad hoc Scanner

- After the scanner returns a **token** to the parser. When invoked again it repeats the algorithm from the beginning, using the next available characters of input (including any that were peeked at but not consumed the last time).
 - As a rule, we accept the longest possible token in each invocation of the scanner. Thus foobar is always foobar and never f or foo or foob. More to the point, in a language like C, 3.14159 is a real number and never 3, ., and 14159. White space is generally ignored, except to the extent that it separates tokens.
- Figure 2.5 could be extended to outline a scanner for a larger programming language. The result could create code in some implementation language.
- Production compilers often use ad hoc scanners. During development, however, it is usually preferable to build a scanner in a structured way, as an explicit representation of a finite automaton.
- **Finite automata** can be generated automatically from a set of regular expressions, making it easy to regenerate a scanner when token definitions change. [LEX/FLEX]

Figure 2.5 Outline of an ad hoc scanner for tokens in our calculator language.

Scanner for Calculator Tokens

in the form of a finite automaton (Case Study)



comment \rightarrow */* (non-* | * non-/)^{*} ** /*
 \quad *| // (non-newline)^{*} newline*

assign \rightarrow *:=*

plus \rightarrow *+*

minus \rightarrow *-*

times \rightarrow ***

div \rightarrow */*

lparen \rightarrow *(*

rparen \rightarrow *)*

id \rightarrow *letter (letter | digit)^{*}*
 except for **read** and **write**

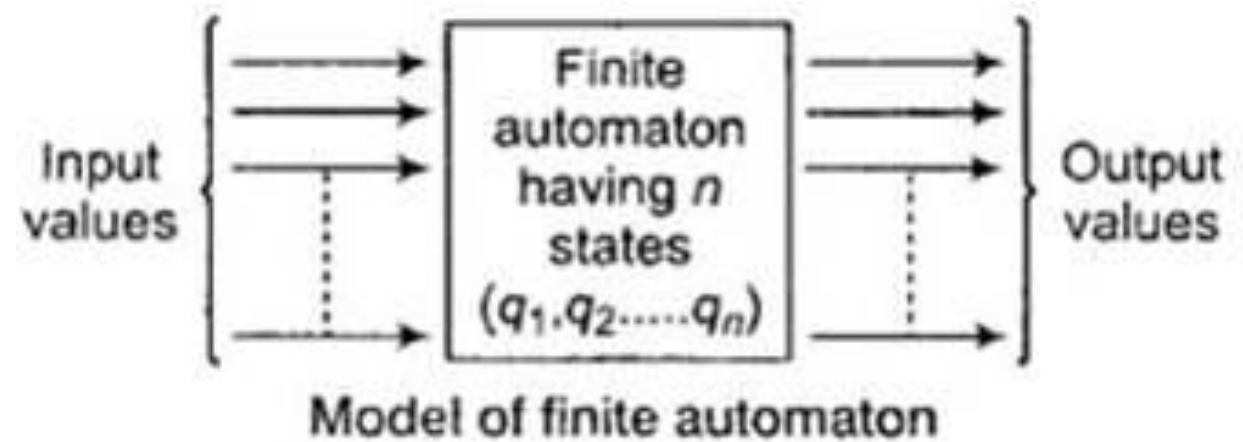
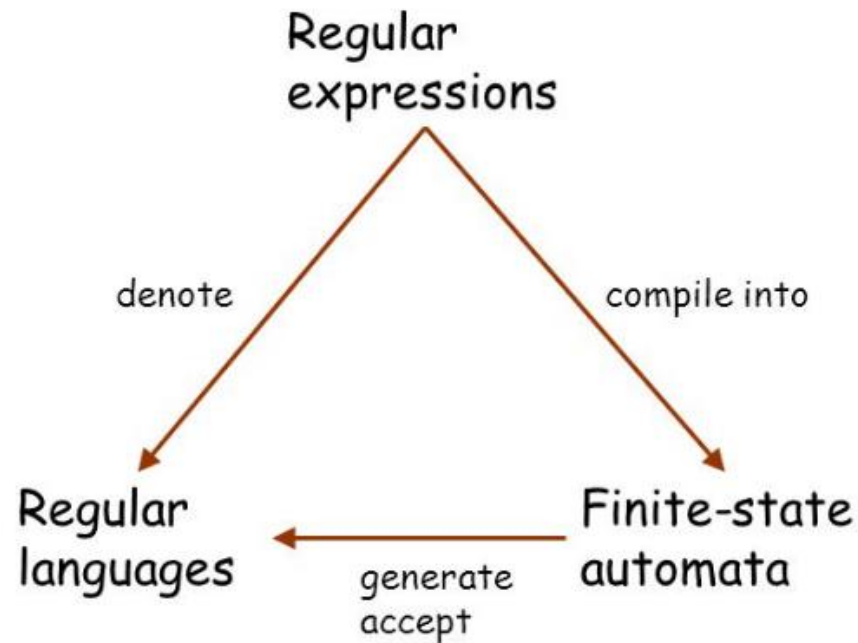
number \rightarrow *digit digit^{*} | digit^{*} (. digit | digit .) digit^{*}*

Scanning II

Scanner Design 1: Regex to NFA

SECTION 2

Regular Expression, Regular Languages, Finite State Automata, and Finite State Machine



Conversion of Regular Expression to Finite State Automaton

▲ Starting state

Regular Expression

Finite Automaton

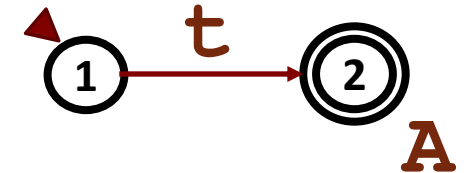
Regular Grammar Rules

Finite Automaton

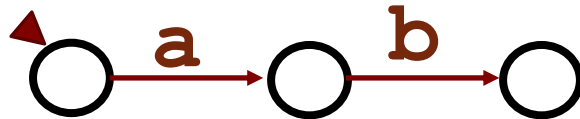
a



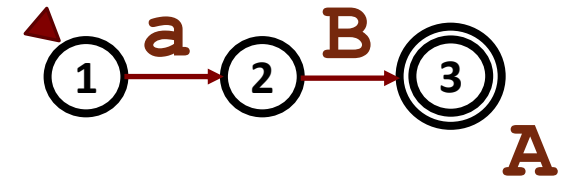
$A \rightarrow t$



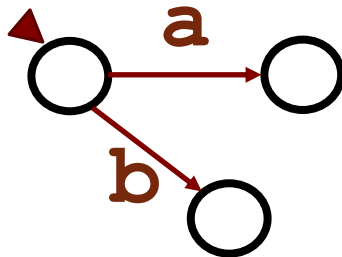
ab



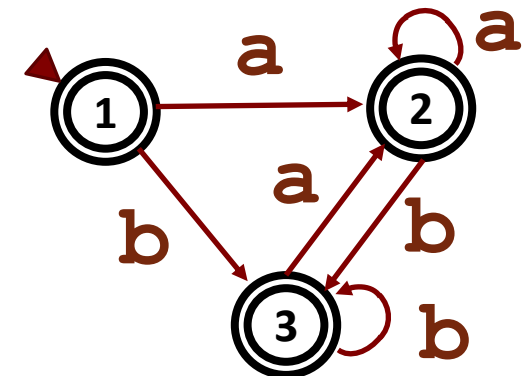
$A \rightarrow aB$



$a | b$



$A \rightarrow (a | b)^*$



a^*



Generating Finite Automaton

Three Step Conversion from Regular Expression to DFA

- The first step converts the regular expressions into a nondeterministic finite automaton (NFA).
- The second step is to convert nondeterministic finite automaton (NFA) to deterministic finite automaton (DFA).
- The third step is to minimize the DFA.

Non-deterministic Finite Automaton

The automaton has the desirable property that its actions are deterministic: in any given state with a given input character there is never more than one possible outgoing transition (arrow) labeled by that character. [DFA]

For NFA,

1. There may be more than one transition out of a given state labeled by a given character, and
2. There may be so-called epsilon transitions: arrows labeled but the empty string symbol, ϵ . The NFA is said to accept an input string(token) if there exists a path from the start state to a final state whose non-epsilon transitions are labeled, in order, by the characters of the token.

Constructing an NFA for a Given Regular Expression

A trivial regular expression consisting of a single character c is equivalent to a simple two-state NFA (in fact, a DFA), illustrated in part (a) of Figure 2.7 (next slide). Similarly, the regular expression ε is equivalent to a two-state NFA whose arc is labeled by ε .

Starting with this base we can use three sub-constructions, illustrated in parts (b) through (d) of the same figure, to build larger NFAs to represent the concatenation, alternation, or Kleene closure of the regular expressions represented by smaller NFAs. Each step preserves three invariants: there are no transitions into the initial state, there is a single final state, and there are no transitions out of the final state.

These invariants allow smaller machines to be joined into larger machines without any ambiguity about where to create the connections, and without creating any unexpected paths.

NFA for $d^*(.d \mid d.)d^*$

- To make these constructions concrete, we consider a small but nontrivial example—the decimal strings of **Example 2.3**.
- These consist of a string of decimal digits containing a single decimal point. With only one digit, the point can come at the beginning or the end: $(.d \mid d.)$, where for brevity we use d to represent any decimal digit. Arbitrary numbers of digits can then be added at the beginning or the end: $d^*(.d \mid d.)d^*$.
- Starting with this regular expression and using the constructions of Figure 2.7, we illustrate the construction of an equivalent NFA in Figure 2.8.

Example 2.3:

$number \rightarrow integer \mid real$

$integer \rightarrow digit\,digit^*$

$real \rightarrow integer\,exponent \mid decimal\,(exponent \mid \epsilon)$

$decimal \rightarrow digit^*(.digit \mid digit.)digit^*$

$exponent \rightarrow (e \mid E)(+ \mid - \mid \epsilon)integer$

$digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

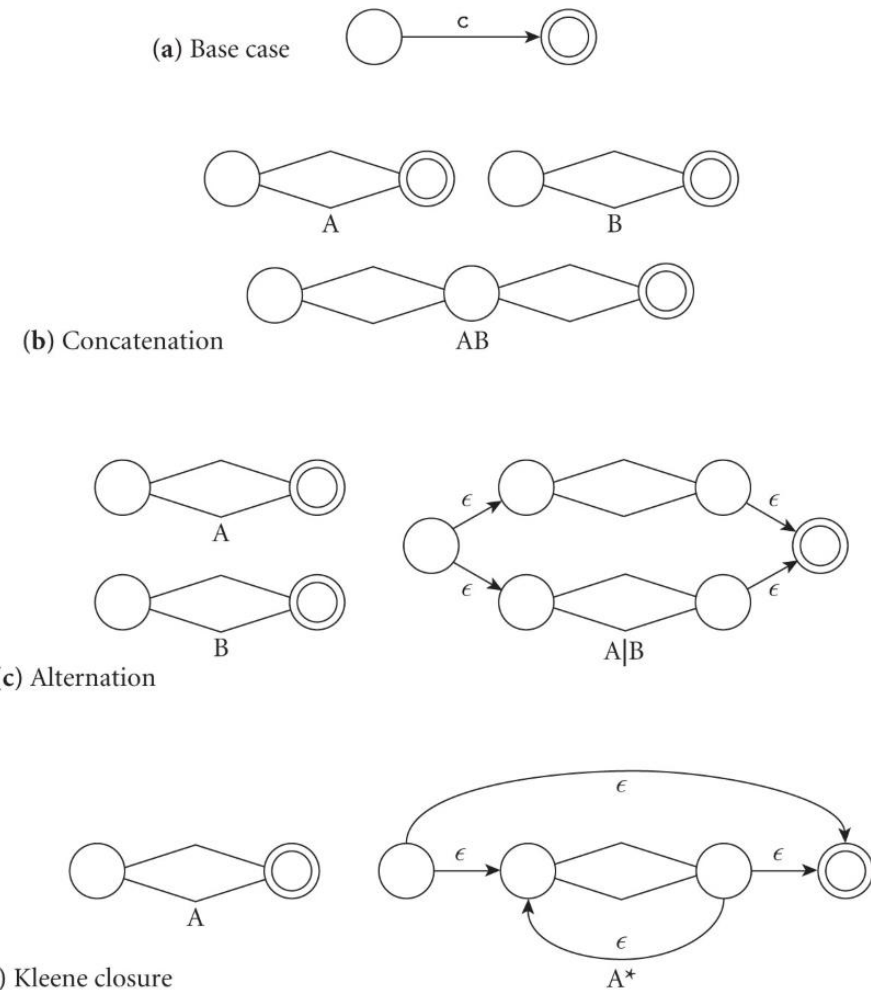


Figure 2.7 Construction of an NFA equivalent to a given regular expression. Part (a) shows the base case: the automaton for the single letter c . Parts (b), (c), and (d), respectively, show the constructions for concatenation, alternation, and Kleene closure. Each construction retains a unique start state and a single final state. Internal detail is hidden in the diamond-shaped center regions.

NFA Representation

decimal \longrightarrow *digit** (*.* *digit* | *digit* *.*) *digit**

- To avoid such a complex and time-consuming strategy, we can use a “set of subsets” construction to transform the NFA into an equivalent DFA.
- The key idea is for the state of the DFA after reading a given input to represent the set of states that the NFA might have reached on the same input.

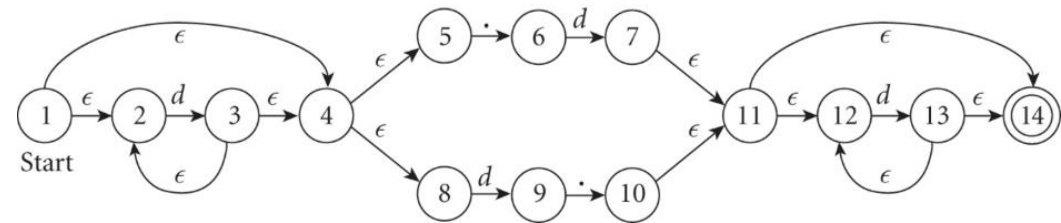
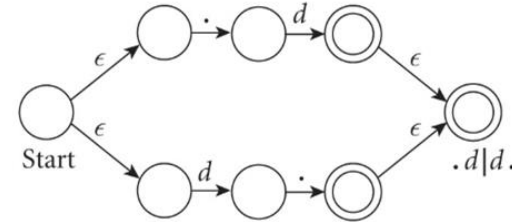
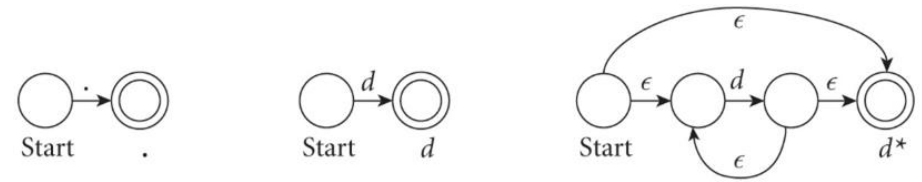


Figure 2.8 Construction of an NFA equivalent to the regular expression $d^*(.d \mid d.)d^*$. In the top row are the primitive automata for $.$ and d , and the Kleene closure construction for d^* . In the second and third rows we have used the concatenation and alternation constructions to build $.d$, $d.$, and $(.d \mid d.)$. The fourth row uses concatenation again to complete the NFA. We have labeled the states in the final automaton for reference in subsequent figures.

Scanning III

Scanner Design 2: NFA to DFA

SECTION 3

NFA to DFA Conversion

Problem Statement

- Let $X = (Q_x, \Sigma, \delta_x, q_0, F_x)$ be an NFA which accepts the language $L(X)$.
- We have to design an equivalent DFA $Y = (Q_y, \Sigma, \delta_y, q_0, F_y)$ such that $L(Y) = L(X)$.
- If a NFA and a corresponding DFA accept the same language, then the NFA is equivalent to the DFA.

NFA to DFA Conversion

Algorithm

Input – An NFA

Output – An equivalent DFA

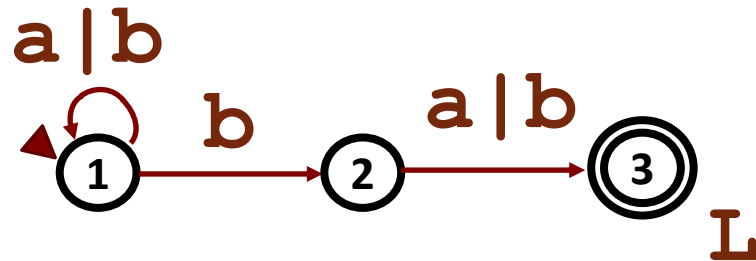
- **Step 1** – Create state table from the given NFA.
- **Step 2** – Create a blank state table under possible input alphabets for the equivalent DFA.
- **Step 3** – Mark the start state of the DFA by q_0 (Same as the NFA).
- **Step 4** – Find out the combination of States $\{Q_0, Q_1, \dots, Q_n\}$ for each possible input alphabet.
- **Step 5** – Each time we generate a new DFA state under the input alphabet columns, we have to apply step 4 again, otherwise go to step 6.
- **Step 6** – The states which contain any of the final states of the NFA are the final states of the equivalent DFA.

Conversion Example

No-epsilon Case

A Language L is accepted by a NFA:

$L \rightarrow (a \mid b)^* b (a \mid b)$



State Transition Table

	a	b
1	1	1 2
2	3	3
3	-	-

NFA - State Transition Table

	a	b
1	1	1 2
2	3	3
3	-	-

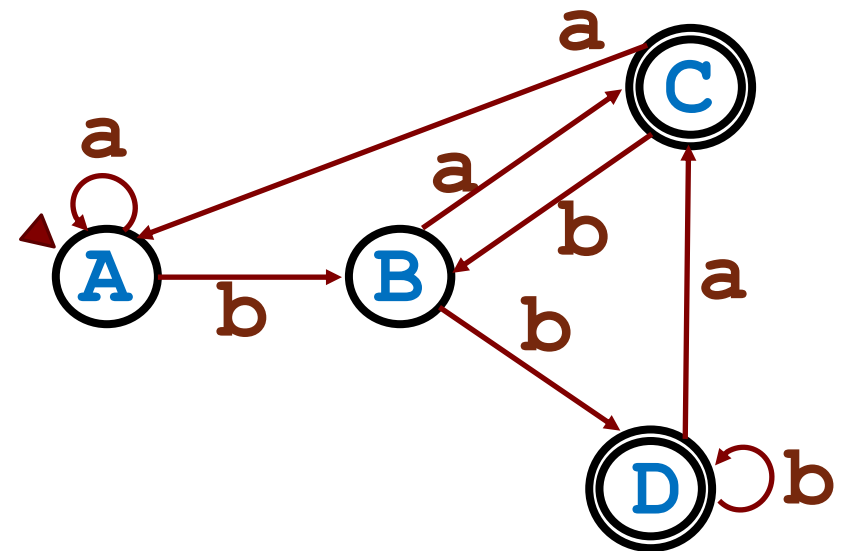
Target DFA - State Transition Table

DFA	NFA	a	b
A	1	1	1 2
B	1 2	1 3	1 2 3
C	1 3	1	1 2
D	1 2 3	1 3	1 2 3

DFA - State Transition Table

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

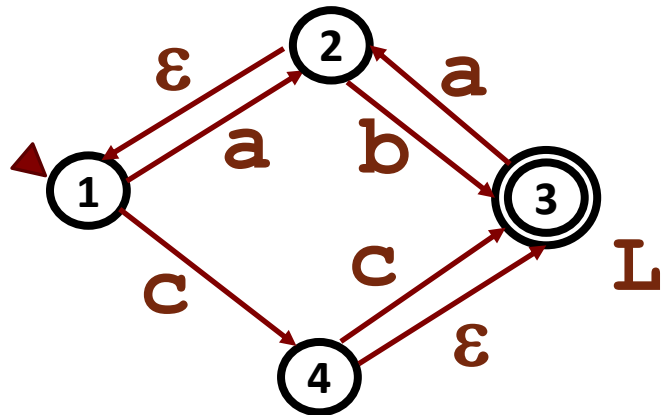
DFA



Conversion Example

With-epsilon Case

NFA with ϵ :



State Transition Table

	a	b	c	ϵ^*
1	2	-	4	1
2	-	3	-	2 1
3	2	-	-	3
4	-	-	3	4 3

State Transition Table

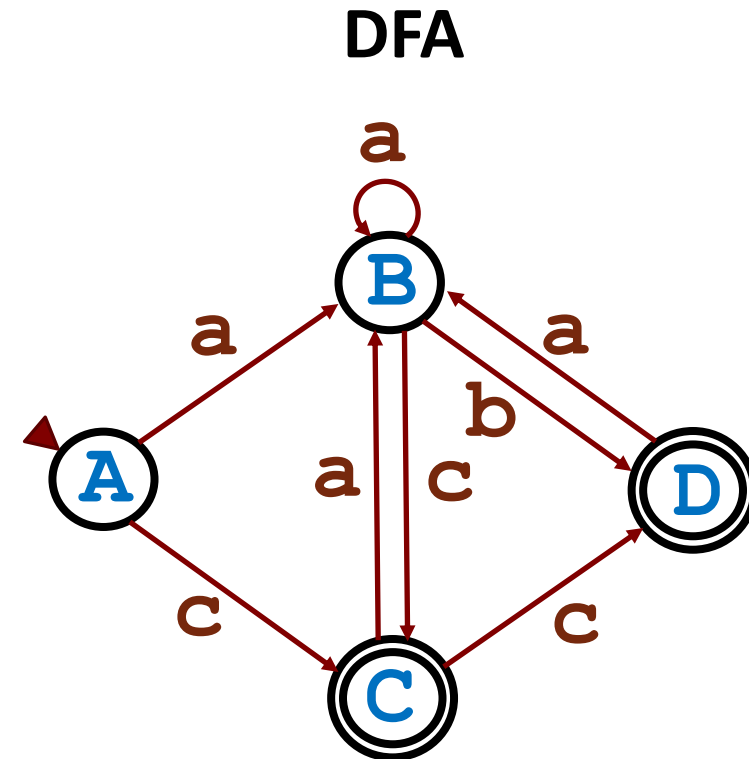
	a	b	c	ϵ^*
1	2	-	4	1
2	-	3	-	2 1
3	2	-	-	3
4	-	-	3	4 3

DFA State Transition Table

DFA	NFA	$a\epsilon^*$	$b\epsilon^*$	$c\epsilon^*$
A	1	2 1	-	4 3
B	2 1	2 1	3	4 3
C	4 3	2 1	-	3
D	3	2 1	-	-

DFA State Transition Table

DFA	NFA	a	b	c
◀ (A)	◀ (1)	(B)	-	⊙ (C)
(B)	(2) (1)	(B)	⊙ (D)	⊙ (C)
⊙ (C)	(4) ⊙ (3)	(B)	-	⊙ (D)
⊙ (D)	⊙ (3)	(B)	-	-



From NFA to DFA

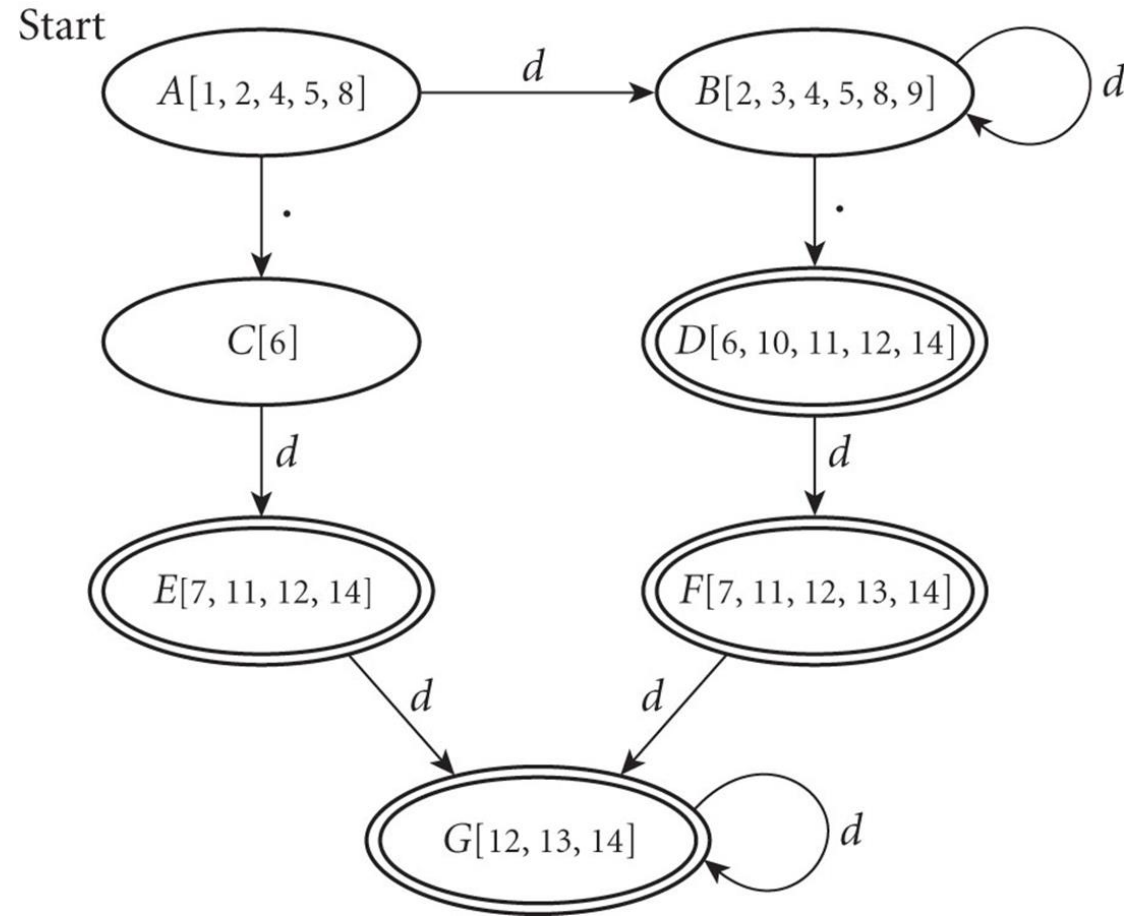


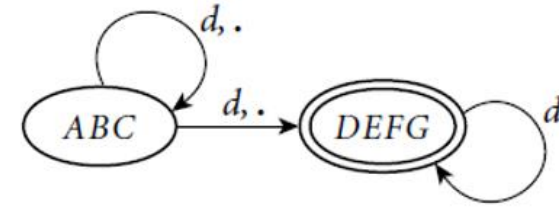
Figure 2.9 A DFA equivalent to the NFA at the bottom of [Figure 2.8](#). Each state of the DFA represents the set of states that the NFA could be in after seeing the same input.

Note: As long as you can find a DFA so that the NFA and the DFA accept the same inputs. Then, the two automaton are equivalent.

Scanning IV

Scanner Design 3: DFA Minimization

SECTION 4



Minimization of DFA I

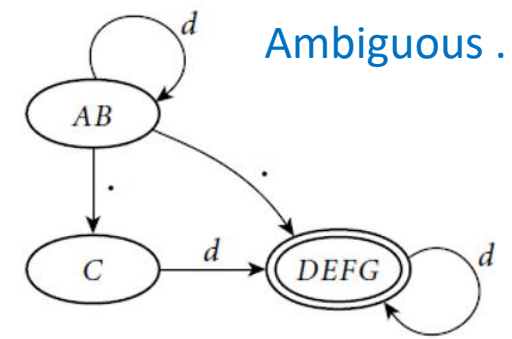
Starting from a regular expression we have now constructed an equivalent DFA.

Though this DFA has seven states, a smaller one should exist.

In particular, once we have seen both a **d** and a **.**, the only valid transitions are on **d**, and we ought to be able to make do with a single final state. We can formalize this intuition, allowing us to apply it to any DFA, via the following inductive construction.

Initially we place the states of the (not necessarily minimal) DFA into two equivalence classes: final states and nonfinal states. We then repeatedly search for an equivalence class χ and an input symbol **c** such that when given **c** as input, the states in χ make transitions to states in $k > 1$ different equivalence classes. We then partition χ into k classes in such a way that all states in a given new class would move to a member of the same old class on **c**.

Minimization of DFA II



When we are unable to find a class to partition in this fashion we are done. In our example, the original placement puts States D, E, F, and G in one class (final states) and States A, B, and C in another, as shown in the upper left of Figure 2.10.

Unfortunately, the start state has ambiguous transitions on both **d** and **.**. To address the **d** ambiguity, we split ABC into AB and C, as shown in the upper right.

New State AB has a self-loop on **d**; new State C moves to State DEFG. State AB still has an ambiguity on **.**, however, which we resolve by splitting it into States A and B, as shown at the bottom of the figure.

At this point there are no further ambiguities, and we are left with a four-state minimal DFA.

Minimization of DFA

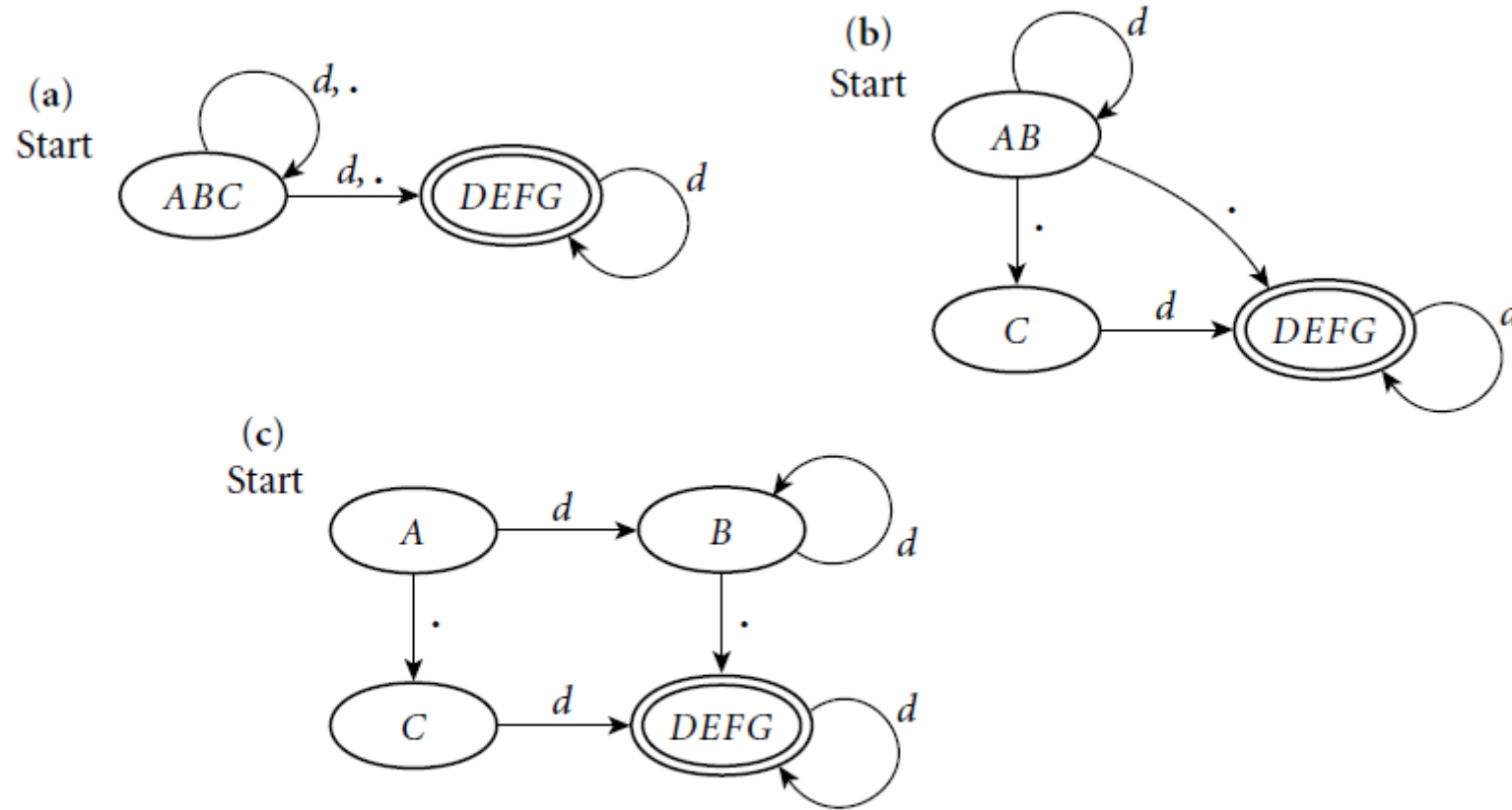


Figure 2.10 Minimization of the DFA of Figure 2.9. In each step we split a set of states to eliminate a transition ambiguity.

Scanning V

Scanner Styles and pragmas

SECTION 5

Scanner Design (Scanner Code)

- We can implement a scanner that explicitly captures the “circles-and-arrows” structure of DFA in either of two main ways.
- Implementation 1: one embeds the automaton in the control flow of the program using **gotos** or nested **case (switch)** statements.
- Implementation 2: Table and Driver.

As a general rule handwritten automata tend to use nested case statements, while automatically generated automata use tables.

Scanning

- Writing a pure DFA as a set of nested case statements is a surprisingly useful programming technique
 - though it's often easier to use **perl**, **awk**, **sed**
 - for details see Figure 2.11
- Table-driven DFA is what **lex** and **scangen** produce
 - **lex (flex)** in the form of **C** code
 - **scangen** in the form of numeric tables and a separate driver (for details see Figure 2.12)

Nested **case** Statement Automaton

- Scanning through the character stream with look-ahead.
- Use case statement as the dispatcher of the next state for the state machine(automaton).
- The outer case statement covers the states of the finite automaton.
- The inner case statements cover the transitions out of each state. Most inner clauses simply set a new state. Some return from the scanner with the current token.

```
state := 1                                -- start state
loop
  read cur_char
  case state of
    1 : case cur_char of
          ' ', '\t', '\n' : ...
          'a'...'z' :      ...
          '0'...'9' :      ...
          '>' :             ...
          ...
    2 : case cur_char of
          ...
          ...
    n: case cur_char of
          ...
```

Scanning and Look-ahead

- identifiers: keyword | variable | method_name | class_name | ...
- In general, upcoming characters that a scanner must examine in order to make a decision are known as its look-ahead. In the next lecture, we will see a similar notion of look-ahead tokens in parsing.
- In messier languages, a scanner may need to look an arbitrary distance ahead.

Scanning

- In messier cases, you may not be able to get by with any fixed amount of look-ahead. In Fortran, for example, we have

```
DO 5 I = 1,25    loop  
DO 5 I = 1.25    assignment
```

- Here, we need to remember we were in a potentially final state, and save enough information that we can back up to it, if we get stuck later

Scanning

- Note that the rule about longest-possible tokens means you return only when the next character can't be used to continue the current token
 - the next character will generally need to be saved for the next token
- In some cases, you may need to peek at more than one character of look-ahead in order to know whether to proceed
 - In Pascal, for example, when you have a 3 and you see a dot
 - do you proceed (in hopes of getting 3.14)?
 - or
 - do you stop (in fear of getting 3..5)?

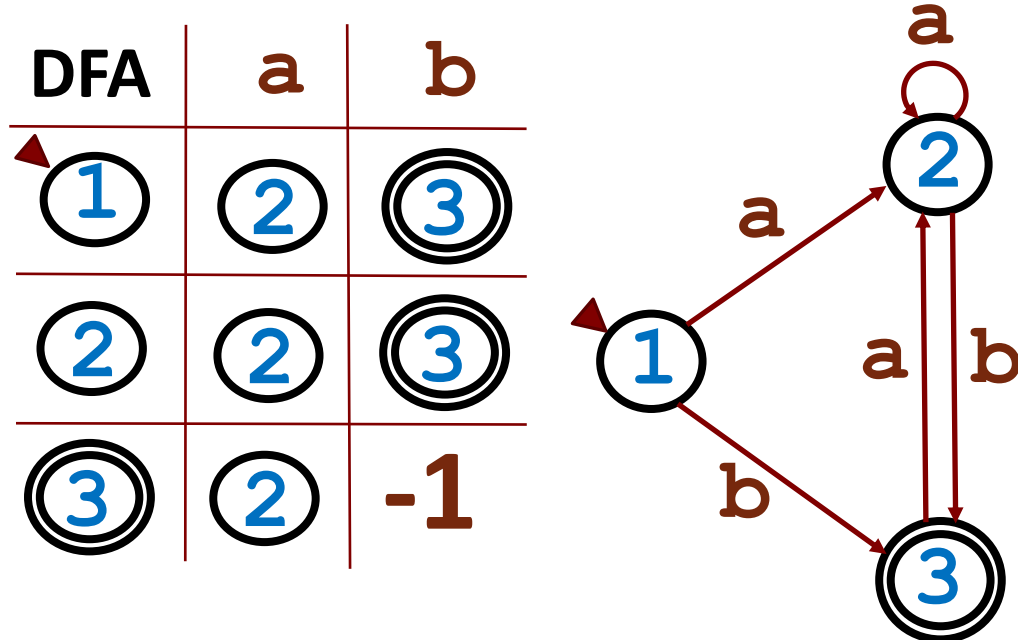
Table-Driven Scanning

- In the preceding subsection we sketched how control flow – **a loop and nested case statements** – can be used to represent a finite automaton.
- An alternative approach represents the automaton as a data structure: a **two-dimensional transition table**.

Table-Driven Automaton

Table-Driven Scanner is an Application of Table Driven Automaton

DFA



```

#define isFinal(s) ((s)<0)
int scanner(){
    char ch;
    int currState = 1;
    while(TRUE){
        ch = NextChar();
        if (ch == EOF) return 0;
        currState = T[currState, ch];
        if (isFinal(currState)){
            return 1;    /* success */
        }
    }
}
  
```


A Driver Program for Table-Driven Scanner

- Outer repeat loop is used for state transition management. Outer loop is also used to filter out the comments and white-space characters.
- Inner loop is used for character scanning and next-state update.
- move: perform state transition. (no break on this case.)
- recognize: token generation. (following move)
- error: lexical error handler.

Figure 2.11 Driver for a table-driven scanner, with code to handle the ambiguous case in which one valid token is a prefix of another, but some intermediate string is not.

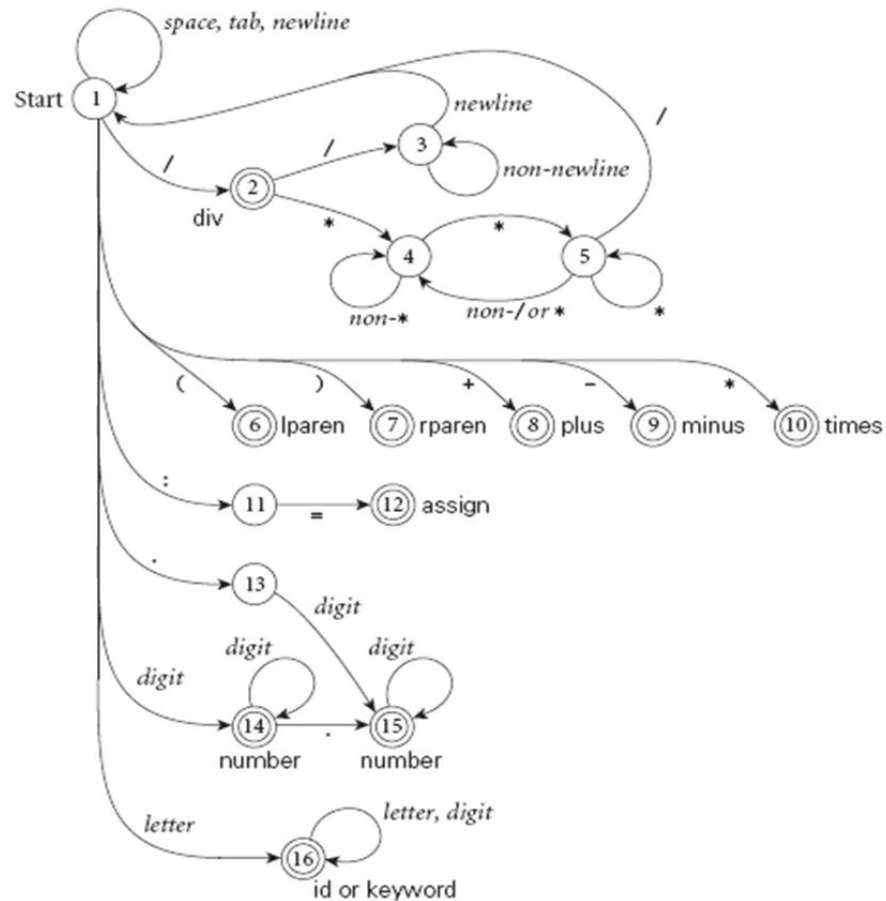
```
state = 0 .. number_of_states
token = 0 .. number_of_tokens
scan_tab : array [char, state] of record
    action : (move, recognize, error)
    new_state : state
token_tab : array [state] of token      -- what to recognize
keyword_tab : set of record
    k_image : string
    k_token : token
-- these three tables are created by a scanner generator tool

tok : token
cur_char : char
remembered_chars : list of char
repeat
    cur_state : state := start_state
    image : string := null
    remembered_state : state := 0      -- none
    loop
        read cur_char
        case scan_tab[cur_char, cur_state].action
            move:
                if token_tab[cur_state] ≠ 0
                    -- this could be a final state
                    remembered_state := cur_state
                    remembered_chars := ε
                    add cur_char to remembered_chars
                    cur_state := scan_tab[cur_char, cur_state].new_state
            recognize:
                tok := token_tab[cur_state]
                unread cur_char      -- push back into input stream
                exit inner loop
            error:
                if remembered_state ≠ 0
                    tok := token_tab[remembered_state]
                    unread remembered_chars
                    remove remembered_chars from image
                    exit inner loop
                -- else print error message and recover; probably start over
                append cur_char to image
        -- end inner loop
    until tok ∉ {white_space, comment}
    look image up in keyword_tab and replace tok with appropriate keyword if found
    return (tok, image)
```

 filtering

Scanner for Calculator Tokens

in the form of a finite automaton (Case Study)



comment \rightarrow */* (non-* | * non-/)^{*} ** /*
 \quad *| // (non-newline)^{*} newline*

assign \rightarrow *:=*

plus \rightarrow *+*

minus \rightarrow *-*

times \rightarrow ***

div \rightarrow */*

lparen \rightarrow *(*

rparen \rightarrow *)*

id \rightarrow *letter (letter | digit)^{*}*
 except for **read** and **write**

number \rightarrow *digit digit^{*} | digit^{*} (. digit | digit .) digit^{*}*

State	Current input character														
	<i>space, tab</i>	<i>newline</i>	<i>/</i>	<i>*</i>	<i>(</i>	<i>)</i>	<i>+</i>	<i>−</i>	<i>:</i>	<i>=</i>	<i>.</i>	<i>digit</i>	<i>letter</i>	<i>other</i>	
1	17	17	2	10	6	7	8	9	11	−	13	14	16	−	div
2	−	−	3	4	−	−	−	−	−	−	−	−	−	−	
3	3	18	3	3	3	3	3	3	3	3	3	3	3	3	
4	4	4	4	5	4	4	4	4	4	4	4	4	4	4	lparen
5	4	4	18	5	4	4	4	4	4	4	4	4	4	4	
6	−	−	−	−	−	−	−	−	−	−	−	−	−	−	
7	−	−	−	−	−	−	−	−	−	−	−	−	−	−	rparen
8	−	−	−	−	−	−	−	−	−	−	−	−	−	−	plus
9	−	−	−	−	−	−	−	−	−	−	−	−	−	−	minus
10	−	−	−	−	−	−	−	−	−	−	−	−	−	−	times
11	−	−	−	−	−	−	−	−	−	12	−	−	−	−	assign
12	−	−	−	−	−	−	−	−	−	−	−	−	−	−	
13	−	−	−	−	−	−	−	−	−	−	−	15	−	−	
14	−	−	−	−	−	−	−	−	−	−	15	14	−	−	number
15	−	−	−	−	−	−	−	−	−	−	−	15	−	−	number
16	−	−	−	−	−	−	−	−	−	−	−	16	16	−	identifier
17	17	17	−	−	−	−	−	−	−	−	−	−	−	−	white_space
18	−	−	−	−	−	−	−	−	−	−	−	−	−	−	comment

Figure 2.12 Scanner tables for the calculator language. These could be used by the code of [Figure 2.11](#). States are numbered as in [Figure 2.6](#), except for the addition of two states—17 and 18—to “recognize” white space and comments. The right-hand column represents table token tab; the rest of the figure is scan tab. Dashes indicate no way to extend the current token. Table keyword tab (not shown) contains the strings read and write.

Pragmas

Significant Comments

In this case they are best processed by the parser or semantic analyzer.

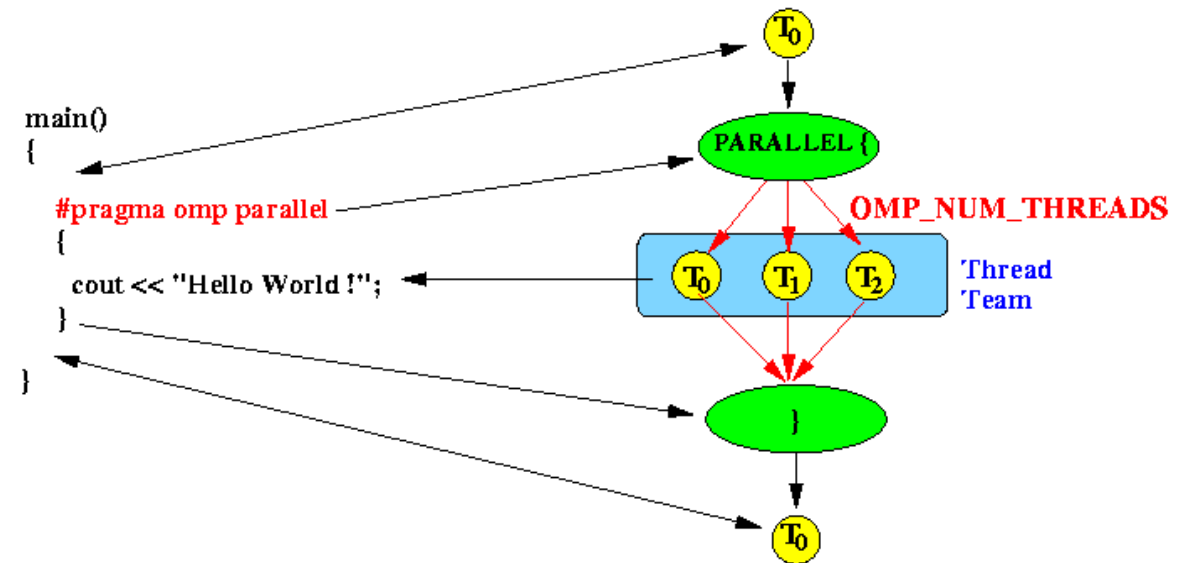
Pragmas that serve as directives may:

- Turn various kinds of run-time checks (e.g., pointer or subscript checking) on or off
- Turn certain code improvements on or off (e.g., on in inner loops to improve performance; off otherwise to improve compilation speed)
- Enable or disable performance profiling (statistics gathering to identify program bottlenecks)

Pragmas

Significant Comments

- **Pragmatics** is a subfield of linguistics and semiotics that studies the ways in which context contributes to meaning.
- In computer programming, a **directive pragma** (from "pragmatic") is a language construct that specifies how a compiler (or other translator) should process its input.



Note: Open Multi-Processing

Pragmas

Significant Comments

Pragmas that serve (merely) as hints provide the compiler with information about the source program that may allow it to do a better job:

- Variable *x* is very heavily used (it may be a good idea to keep it in a register).
- Subroutine *F* is a pure function: its only effect on the rest of the program is the value it returns.
- Subroutine *S* is not (indirectly) recursive (its storage may be statically allocated).
- 32 bits of precision (instead of 64) suffice for floating-point variable *x*.

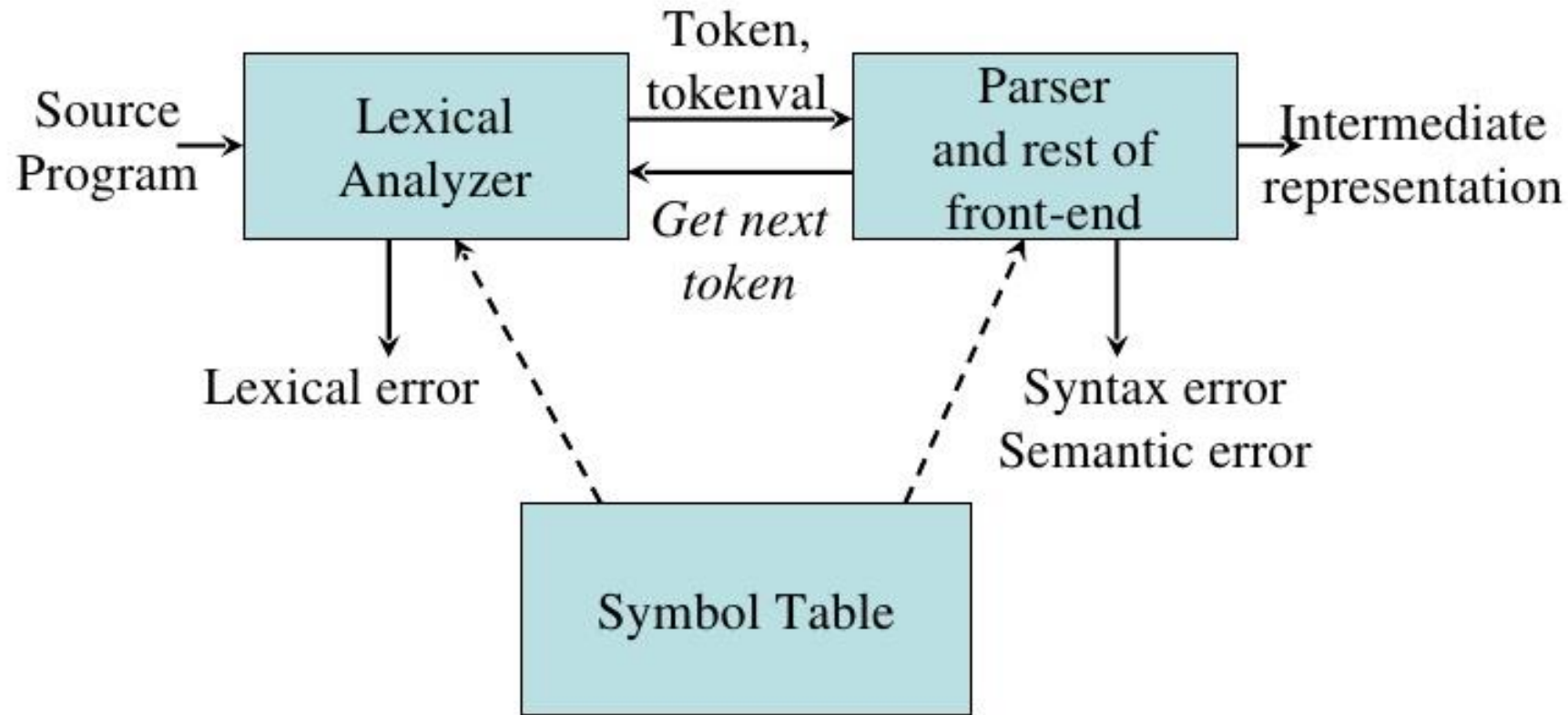
The compiler may ignore these in the interest of simplicity, or in the face of contradictory information.

Parsing I

Overview

SECTION 6

Position of a Parser in the Compiler Model

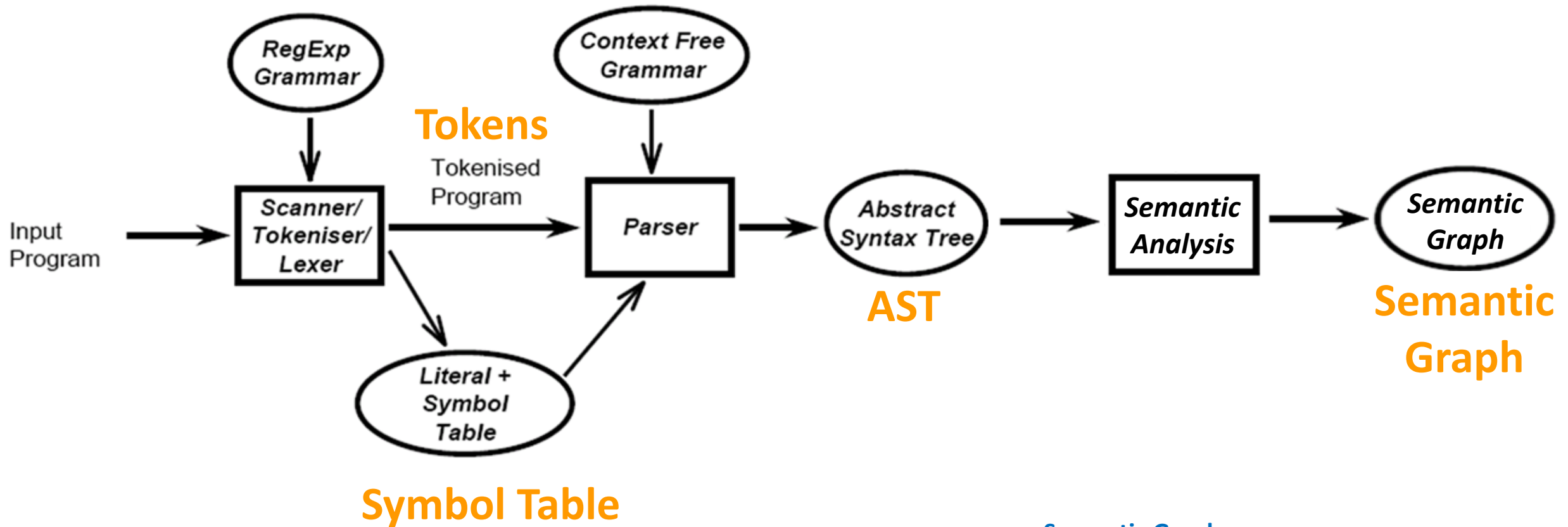


Parser is the Heart of a Typical Compiler

Symbol Table, Syntax Tree, Semantic Analysis

- Parser
 - calls the scanner to obtain the tokens of the input program,
 - assembles the tokens together into a syntax tree, and
 - passes the tree to the later phases of the compiler, which perform semantic analysis and code generation and improvement.
- In effect, the parser is “in charge” of the entire compilation process; this style of compilation is sometimes referred to as **syntax-directed translation**.

Systematic Compiler Front-End Generation

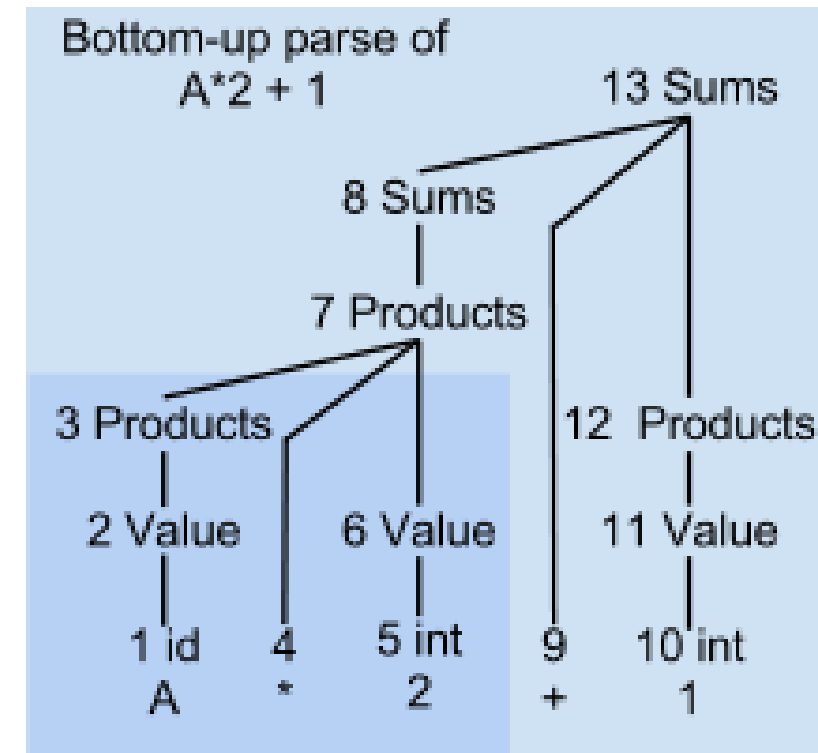
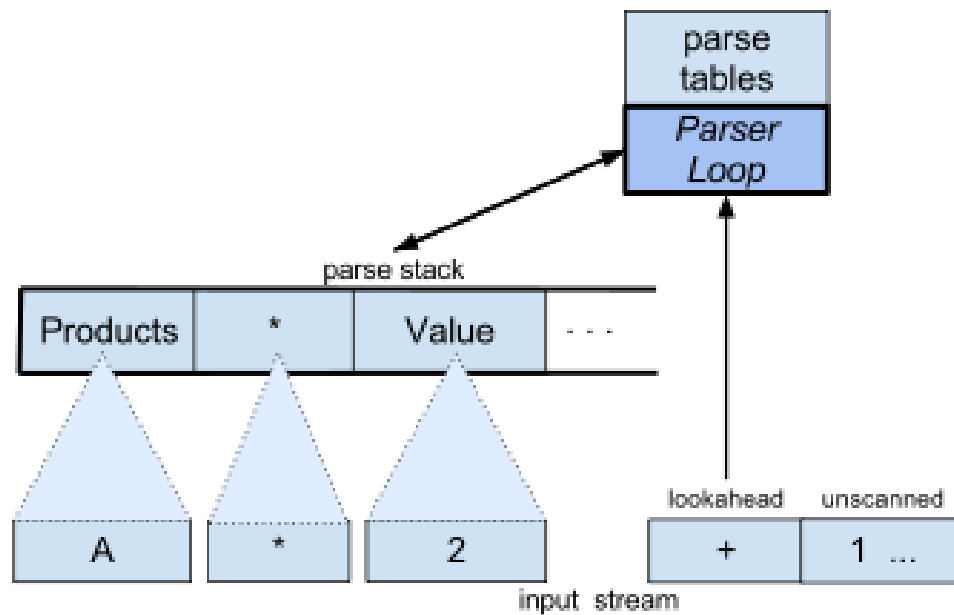


Semantic Graph:

AST with Additional Properties
and Resolved References

Parsing

Putting Tokens into a Parsing Tree (Parser is a language recognizer.)



Example in LR Parsing (Bottom UP Parsing)

Terminology

- Context-Free Grammar (CFG) – Push-Down Automata
- Symbols
 - Terminals (tokens)
 - Non-terminals
- Production
- Derivations (left-most and right-most - canonical)
- Parse Trees
- Sentential Form

Note: A sentential form is the start symbol S of a grammar or any string in $(V \cup T)^*$ that can be derived from S .

Sentential Forms

- A sentential form is the start symbol **S** of a grammar or any string in $(V \cup T)^*$ that can be derived from **S**.
- Consider the linear grammar $(\{S, B\}, \{a, b\}, S, \{S \rightarrow aS, S \rightarrow B, B \rightarrow bB, B \rightarrow \lambda\})$.
- A derivation using this grammar might look like this:
$$S \Rightarrow aS \Rightarrow aB \Rightarrow abB \Rightarrow abbB \Rightarrow abb$$
- Each of $\{S, aS, aB, abB, abbB, abb\}$ is a sentential form.
- Because this grammar is linear, each sentential form has at most **one** variable. Hence there is never any choice about which variable to expand next.

Parsing

- By analogy to RE and DFAs, a context-free grammar (CFG) is a generator for a context-free language (CFL)
 - a parser is a language recognizer $O(n^3)$
- There is an infinite number of grammars for every context-free language
 - not all grammars are created equal. However, there are large classes of grammars for which we can build parsers that run in linear time.

Parsing

- It turns out that for any CFG we can create a parser that runs in $O(n^3)$ time
- There are two well-known parsing algorithms that permit this
 - Early's algorithm
 - Cooke-Younger-Kasami (CYK) algorithm
- $O(n^3)$ time is clearly unacceptable for a parser in a compiler
 - too slow

Parsing II

Parser Styles

SECTION 7

Class	Direction of scanning	Derivation discovered	Parse tree construction	Algorithm used
LL	left-to-right	left-most	top-down	predictive
LR	left-to-right	right-most	bottom-up	shift-reduce

Parsing

- Fortunately, there are large classes of grammars for which we can build parsers that run in linear time
 - The two most important classes are called LL and LR

• LL stands for 'Left-to-right, Leftmost derivation'. **LL**: L in Top-down Shape

• LR stands for 'Left-to-right, Rightmost derivation' **LR**: Reverse means bottom up

Class	Direction of scanning	Derivation discovered	Parse tree construction	Algorithm used
LL	left-to-right	left-most	top-down	predictive
LR	left-to-right	right-most	bottom-up	shift-reduce

Parsing

- LL parsers are also called 'top-down', or 'predictive' parsers & LR parsers are also called 'bottom-up', or 'shift-reduce' parsers
- There are several important sub-classes of LR parsers
 - **SLR (Simple LR)**: a type of LR parser with small parse tables and a relatively simple parser generator algorithm.
 - **LALR (Look-Ahead LR)**: an LALR parser or Look-Ahead LR parser is a simplified version of a canonical LR parser

Top-down and Bottom-up Parsing

- Consider the following grammar for a comma separated list of identifiers, terminated by a semicolon:

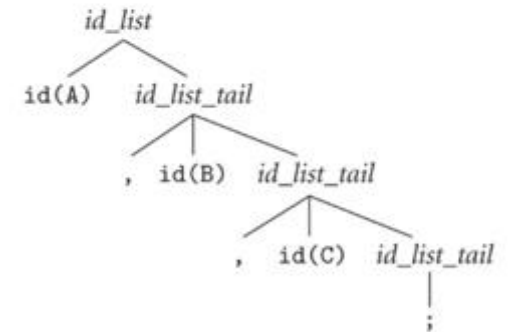
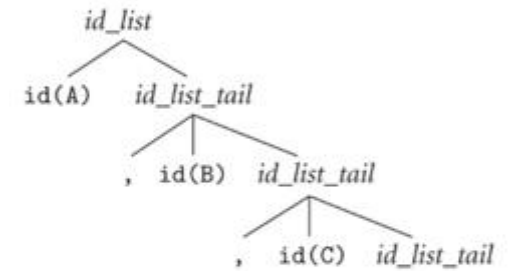
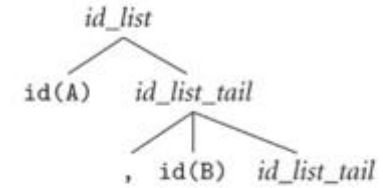
$$id_list \longrightarrow id\ id_list_tail$$
$$id_list_tail \longrightarrow ,\ id\ id_list_tail$$
$$id_list_tail \longrightarrow ;$$

- These productions can be parsed by top-down/bottom-up parsers.

Top-down Parser - LL

of the input string A, B, C; Grammar appears at lower left.

- Top-down parser begins by predicting that the root of the tree (id list) will be replaced by **id id_list_tail**.
Note: If the scanner produced something different, the parser would announce a syntax error.
- The parser then moves down into the first nonterminal child and predicts that id list tail will be replaced by , **id id_list_tail**.
- To make this prediction it needs to peek at the upcoming token (a comma), which allows it to choose between the two possible expansions for **id_list_tail**.
- It then matches the comma and the id and moves down into the next **id_list_tail**. In a similar, recursive fashion, the top-down parser works down the tree, left-to-right, predicting and expanding nodes and tracing out a left-most derivation of the fringe of the tree.

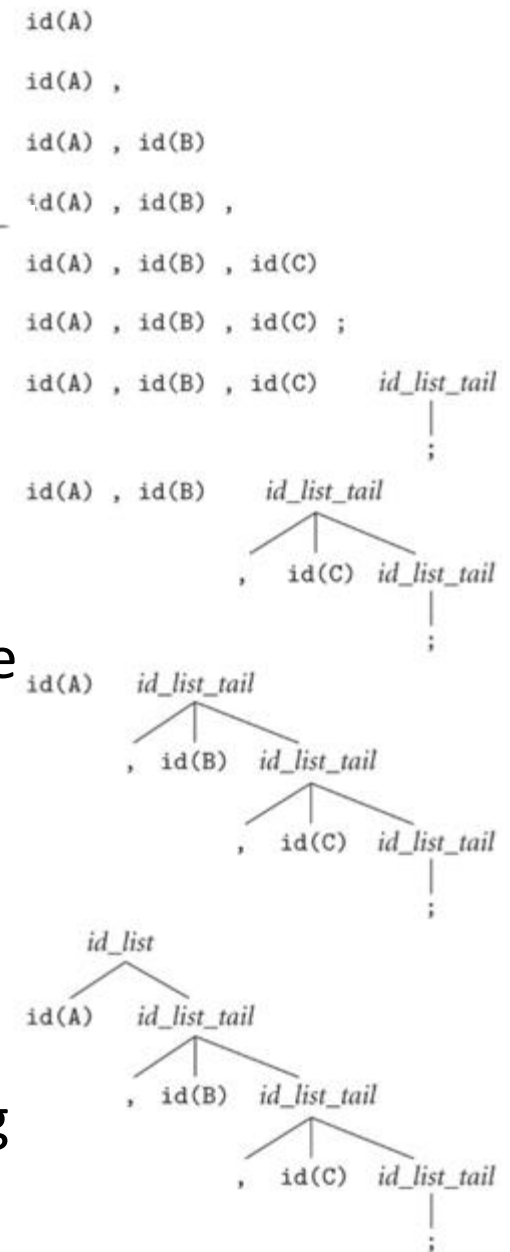


$id_list \rightarrow id\ id_list_tail$
$id_list_tail \rightarrow ,\ id\ id_list_tail$
$id_list_tail \rightarrow ;$

Bottom-Up Parsing - LR

of the input string **A, B, C**; Grammar appears at lower left.

- The bottom-up parser begins by noting that the left-most leaf of the tree is an **id**.
- The next leaf is a comma and the one after that is another **id**.
- The parser continues in this fashion, shifting new leaves from the scanner into a forest of partially completed parse tree fragments, until it realizes that some of those fragments constitute a complete right-hand side.
- **In this grammar, that doesn't occur until the parser has seen the semicolon**—the right-hand side of **id_list_tail** \rightarrow ;. With this right-hand side in hand, the parser reduces the semicolon to an **id_list_tail**.
- It then reduces **, id id_list_tail** into another **id_list_tail**. After doing this one more time it is able to reduce **id id_list_tail** into the root of the parse tree, **id_list**.



Canonical Derivation

- At no point does the bottom-up parser predict what it will see next. Rather, it shifts tokens into its forest until it recognizes a **right-hand side**, which it then reduces to a **left-hand side**.
- Because of this behavior, bottom-up parsers are sometimes called **shift-reduce parsers**. Moving up the figure, from bottom to top, we can see that the shift-reduce parser traces out a right-most derivation, in reverse.
- Because bottom-up parsers were the first to receive careful formal study, rightmost derivations are sometimes called **canonical**.

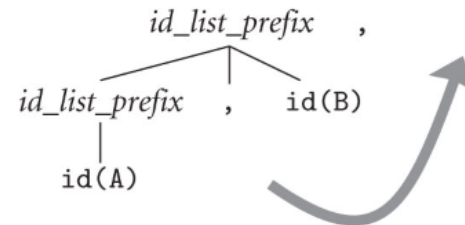
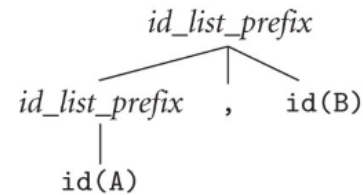
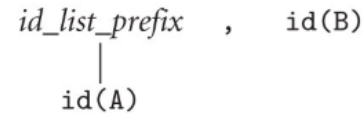
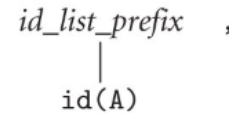
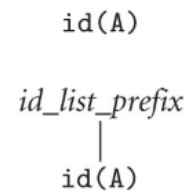
Bounding Space with a Bottom-Up Grammar

using a grammar (lower left) that allows lists to be collapsed incrementally.

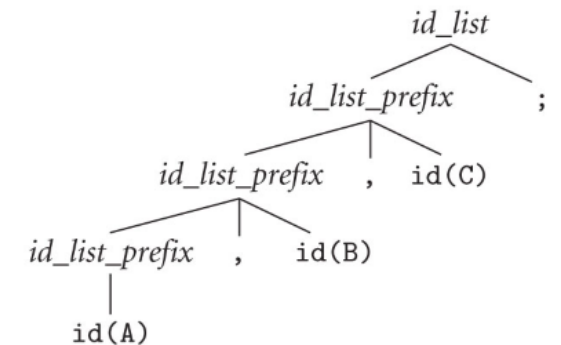
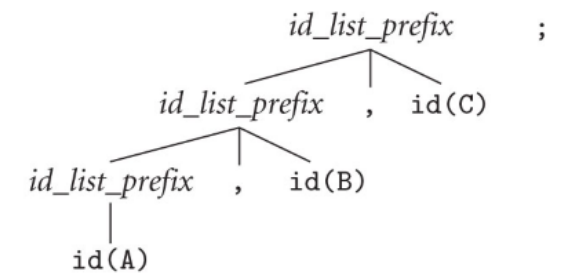
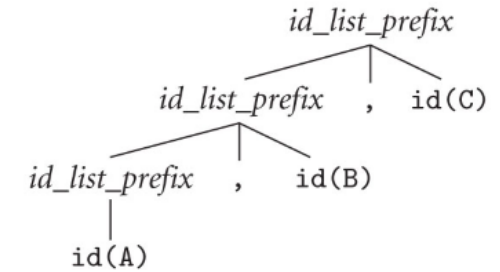
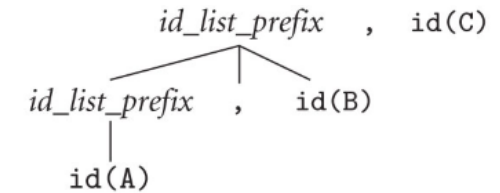
- We can use an alternative grammar to avoid shifting, that allows the parser to reduce prefixes of the **id_list** into non-terminals as it goes along:

$id_list \rightarrow id_list_prefix ;$
 $id_list_prefix \rightarrow id_list_prefix , id$
 $\rightarrow id$

- This grammar cannot be parsed top-down, because when we see an **id** on the input and we're expecting an **id_list_prefix**, we have no way to tell which of the two possible productions we should predict.
- This grammar works well for bottom-up.



$id_list \rightarrow id_list_prefix ;$
 $id_list_prefix \rightarrow id_list_prefix , id$
 $\rightarrow id$



Parsing

- Every LL(1) grammar is also LR(1), though right recursion in production tends to require very deep stacks and complicates semantic analysis
- Every CFL that can be parsed deterministically has an SLR(1) grammar (which is LR(1))
- Every deterministic CFL with the prefix property (no valid string is a prefix of another valid string) has an LR(0) grammar

Parsing

LR(1) LR Parser with One Token Look-Ahead

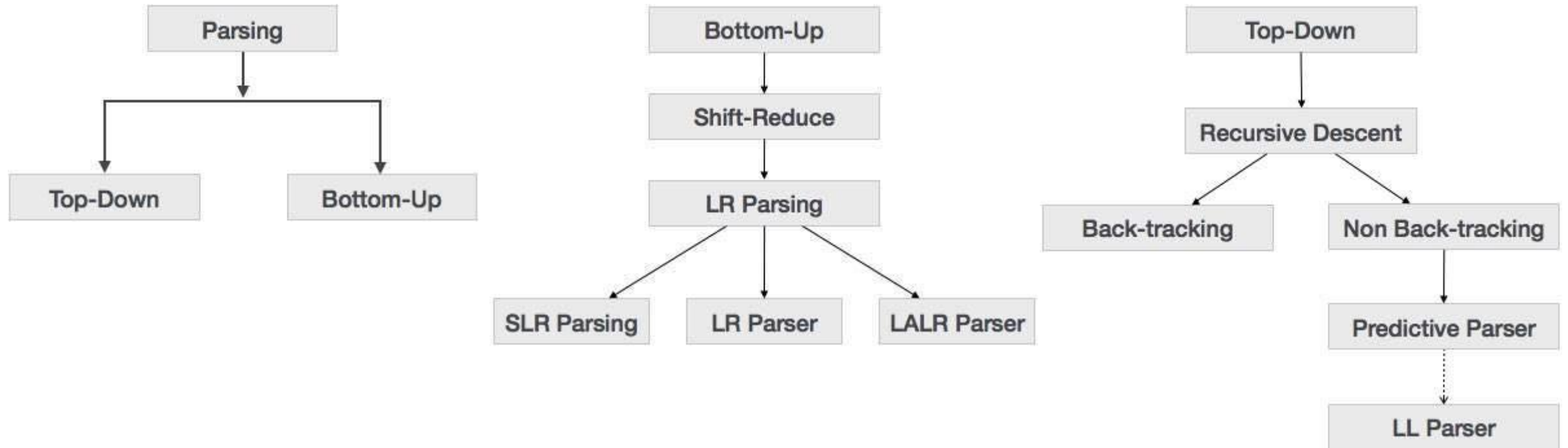
- You commonly see LL or LR (or whatever) written with a number in parentheses after it
 - This number indicates how many tokens of look-ahead are required in order to parse
 - Almost all real compilers use one token of look-ahead
- The expression grammar (with precedence and associativity) you saw before is LR(1), but not LL(1).
(Expression for Arithmetic, Example 2.8, not covered here)

Example 2.8:

1. $expr \longrightarrow \boxed{term} \mid expr \text{ add_op } \boxed{term}$
2. $term \longrightarrow factor \mid \boxed{term} \text{ mult_op } \boxed{factor}$
3. $factor \longrightarrow \boxed{id} \mid \text{number} \mid - \boxed{factor} \mid (expr)$
4. $add_op \longrightarrow + \mid -$
5. $mult_op \longrightarrow * \mid /$

Note: With Left-Recursion, it is not LL.

Types of Parsing



Parsing III

Recursive Descent Parsing

SECTION 8

Top-down Grammar for a Calculator

LL(1) Grammar for a Simple Calculator Language

Figure 2.16

program \rightarrow *stmt_list* $\$ \$$
stmt_list \rightarrow *stmt* *stmt_list* | ϵ
stmt \rightarrow *id* := *expr* | read *id* | write *expr*
expr \rightarrow *term* *term_tail*
term_tail \rightarrow *add_op* *term* *term_tail* | ϵ
term \rightarrow *factor* *factor_tail*
factor_tail \rightarrow *mult_op* *factor* *factor_tail* | ϵ
factor \rightarrow (*expr*) | *id* | *number*
add_op \rightarrow + | -
mult_op \rightarrow * | /

- Top-Down (predictive) parsing.
- The calculator allows value to be read into named (numeric) variables which is used in expressions.
- Expression written to the output.
- Control flow is linear. (no loop, if-statement, or jumps)
- The end-marker ($\$ \$$) pseudo token is produced by the scanner at the end of the input. This token allows the parser to terminate cleanly once it has seen the entire program.
- As in regular expressions, we use the symbol ϵ to denote the empty string. A production with ϵ on the right-hand side is sometimes called an **epsilon production**.

Recursive Descent

Make Parsing Predictively

- It is helpful to compare the *expr* portion of **Figure 2.16** to the expression grammar of **Example 2.8**.
- LR grammar is significantly more intuitive.
- It suffers from a problem similar to that of the *id_list* grammar: if we see an *id* on the input when expecting an *expr*. We have no way to tell which of the two possible productions to predict.
- Grammar in Figure 2.16 avoids the problem by merging the common prefixes of right-hand sides into a single production by using new symbols (***term_tail*** and ***factor_tail***)
- In effect, we have sacrificed grammatical elegance in order to be able to parse predictively.

Example 2.8:

1. $expr \rightarrow \boxed{term} \mid \boxed{expr} \text{ add_op } \boxed{term}$
2. $term \rightarrow \boxed{factor} \mid \boxed{term} \text{ mult_op } \boxed{factor}$
3. $factor \rightarrow \boxed{id} \mid \text{number} \mid - \boxed{factor} \mid (\text{expr})$
4. $add_op \rightarrow + \mid -$
5. $mult_op \rightarrow * \mid /$

Figure 2.16:

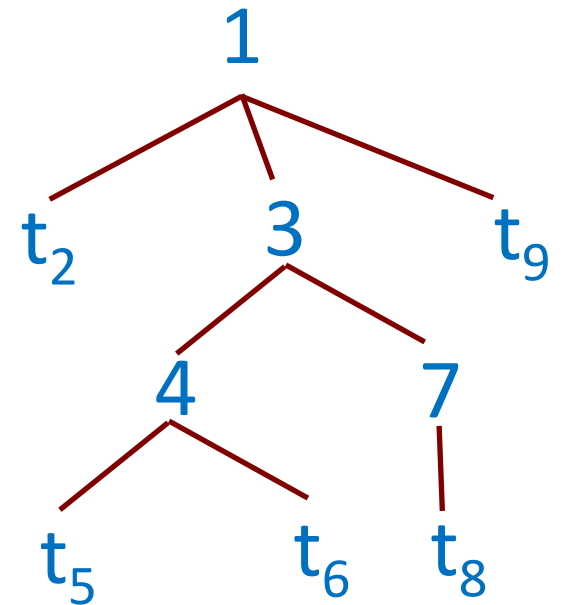
- ▲ $\boxed{program} \rightarrow \text{stmt_list } \$\$$
 $\text{stmt_list} \rightarrow \text{stmt stmt_list} \mid \epsilon$
 $\text{stmt} \rightarrow \text{id} := \text{expr} \mid \text{read id} \mid \text{write expr}$
 $\text{expr} \rightarrow \text{term term_tail}$
 $\text{term_tail} \rightarrow \text{add_op term term_tail} \mid \epsilon$
 $\text{term} \rightarrow \text{factor factor_tail}$
 $\text{factor_tail} \rightarrow \text{mult_op factor factor_tail} \mid \epsilon$
 $\text{factor} \rightarrow (\text{expr}) \mid \text{id} \mid \text{number}$
 $\text{add_op} \rightarrow + \mid -$
 $\text{mult_op} \rightarrow * \mid /$

Recursive Descent Parsing Example

- Top-down Parser
- Built from a set of mutual recursive procedures
- The structure of the resulting program closely mirrors that of the grammar it recognizes.
- The parse tree is constructed
 - From the top
 - From left to right
 - Backtrack when mismatches happen
- Terminals are seen in order of appearance in the token stream:

t_2 t_5 t_6 t_8 t_9

(Note: The number is the order stamp)



Recursive Descent Parsing Example

- Consider the grammar
Expr \rightarrow Term | Term + Expr
Term \rightarrow int | int * Term | (Expr)
- Token stream is: (int)
- Start with top-level non-terminal Expr
 - Try the rules for E in order

Recursive Descent Parsing Example

The grammar:

→ $\text{Expr} \rightarrow \text{Term} \mid \text{Term} + \text{Expr}$
 $\text{Term} \rightarrow \text{int} \mid \text{int} * \text{Term} \mid (\text{Expr})$

RD Parsing Tree:

Expr

The token stream:

(int)



Next Token

Recursive Descent Parsing Example

The grammar:

Expr \rightarrow Term | Term + Expr

→ Term \rightarrow int | int * Term | (Expr)

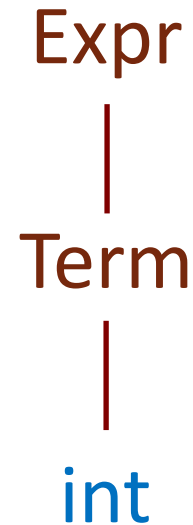
The token stream:

(int)



Next Token

RD Parsing Tree:



Mismatch: It does not match!
Backtrack!

Recursive Descent Parsing Example

The grammar:

Expr \rightarrow Term | Term + Expr

→ Term \rightarrow int | int * Term | (Expr)



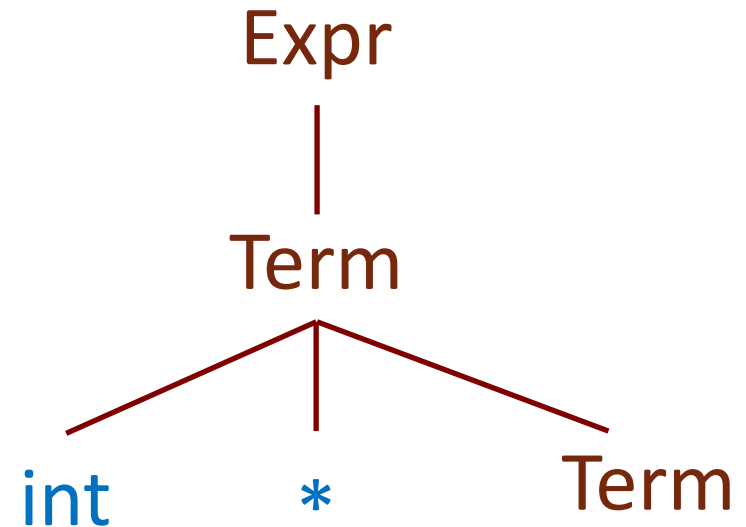
The token stream:

(int)



Next Token

RD Parsing Tree:



Mismatch: It does not match!
Backtrack!

Recursive Descent Parsing Example

The grammar:

Expr \rightarrow Term | Term + Expr

→ Term \rightarrow int | int * Term | (Expr)



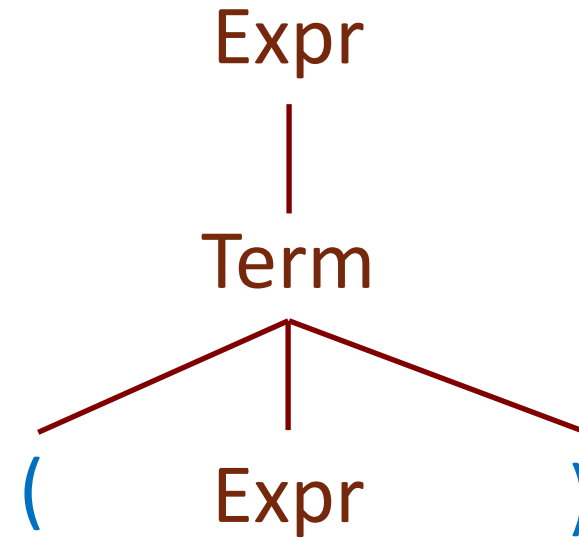
The token stream:

(int)



Next Token

RD Parsing Tree:



Matched: Advance Input.

Recursive Descent Parsing Example

The grammar:

$\text{Expr} \rightarrow \text{Term} \mid \text{Term} + \text{Expr}$

$\text{Term} \rightarrow \text{int} \mid \text{int} * \text{Term} \mid (\text{Expr})$

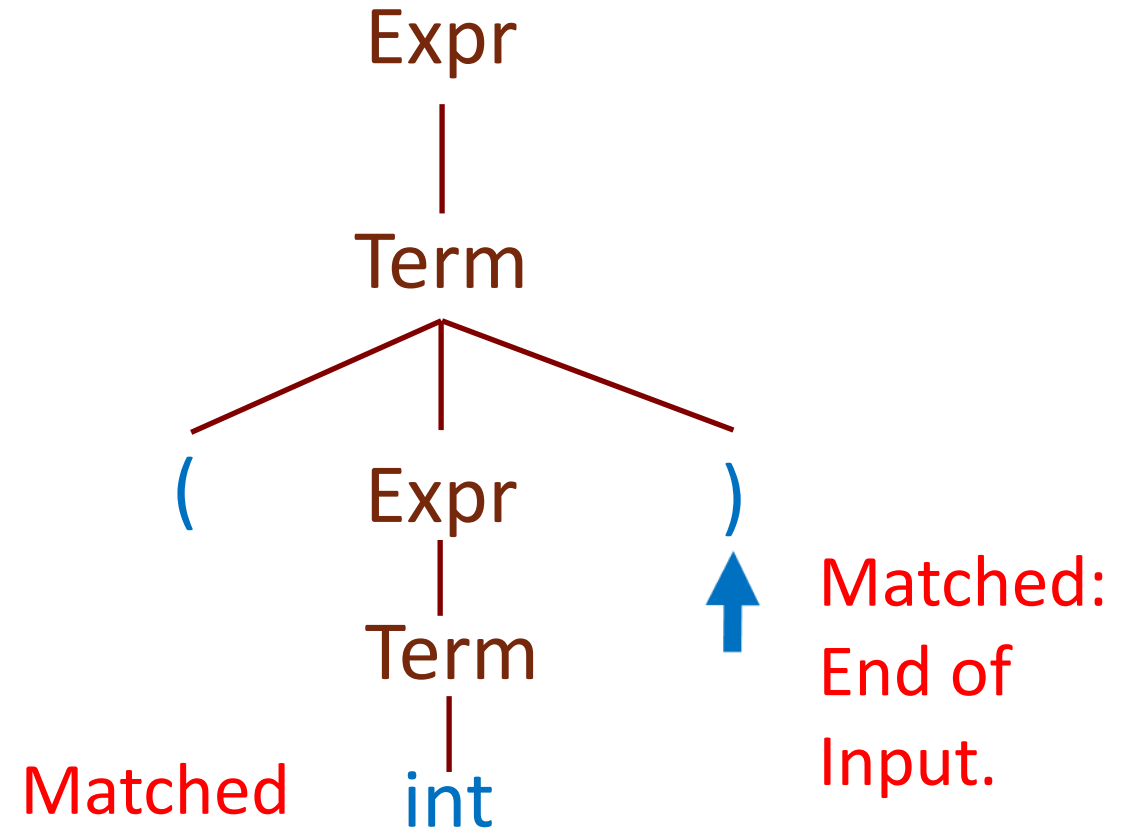
The token stream:

(int)



Next Token

RD Parsing Tree:



Input stream:

i n t a = f () ; i f (a > m a x) ...



Read into memory

InputStream object:

i | n | t | a | = | f | (|) | ; | i | f | (| a | > | m | a | x |) | ...



Lexical analysis

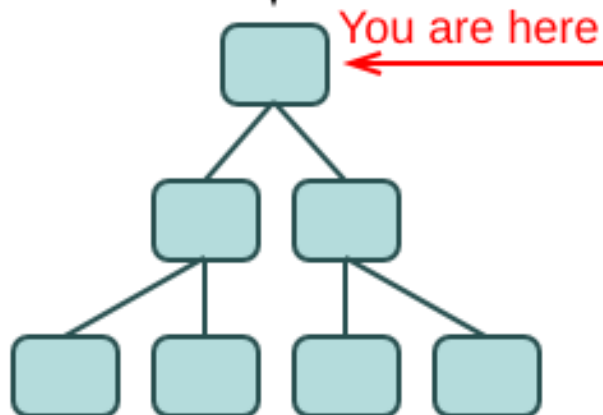
Token stream:

i n t | a = f () ; | i f (a > m a x) | ...



Syntax analysis

Syntax tree:



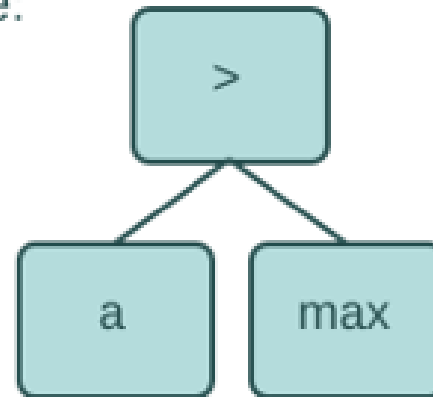
Token stream:



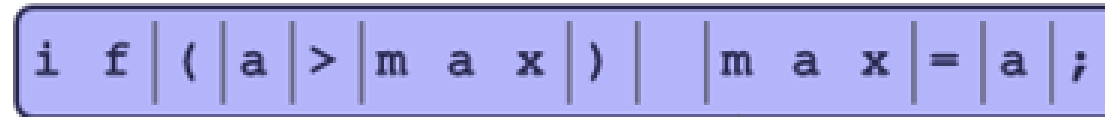
current

Parse condition

Syntax subtree:



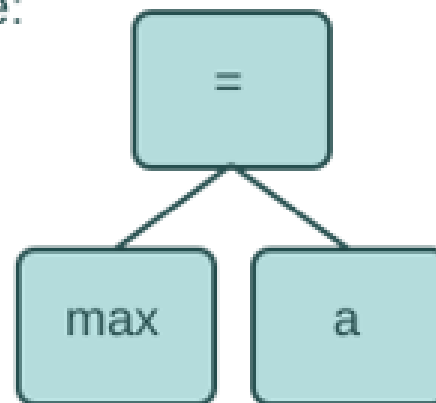
Token stream:



current

Parse assignment statement

Syntax subtree:

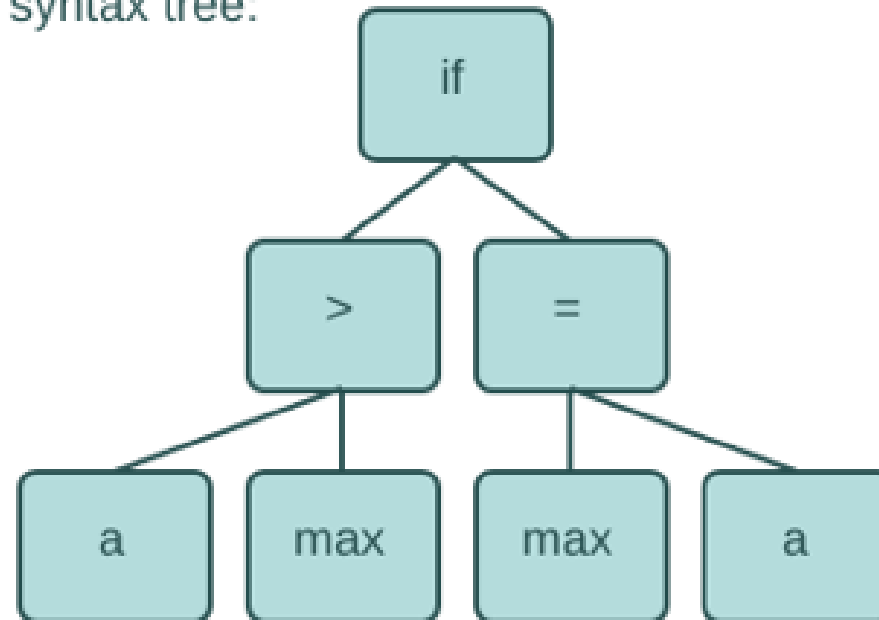


Token stream:



Combine into bigger tree

Full syntax tree:



Backtracking

When a syntax function fails, we want it to have no side effects.

- We want the input to be exactly the way it was before calling the function, because this makes it easier for other functions to have a go at parsing the same input.
- In addition, the function should delete any objects that it created, remove any symbols it entered into a symbol table, etc.

In other words, the function has to backtrack to an earlier situation.

Parsing IV

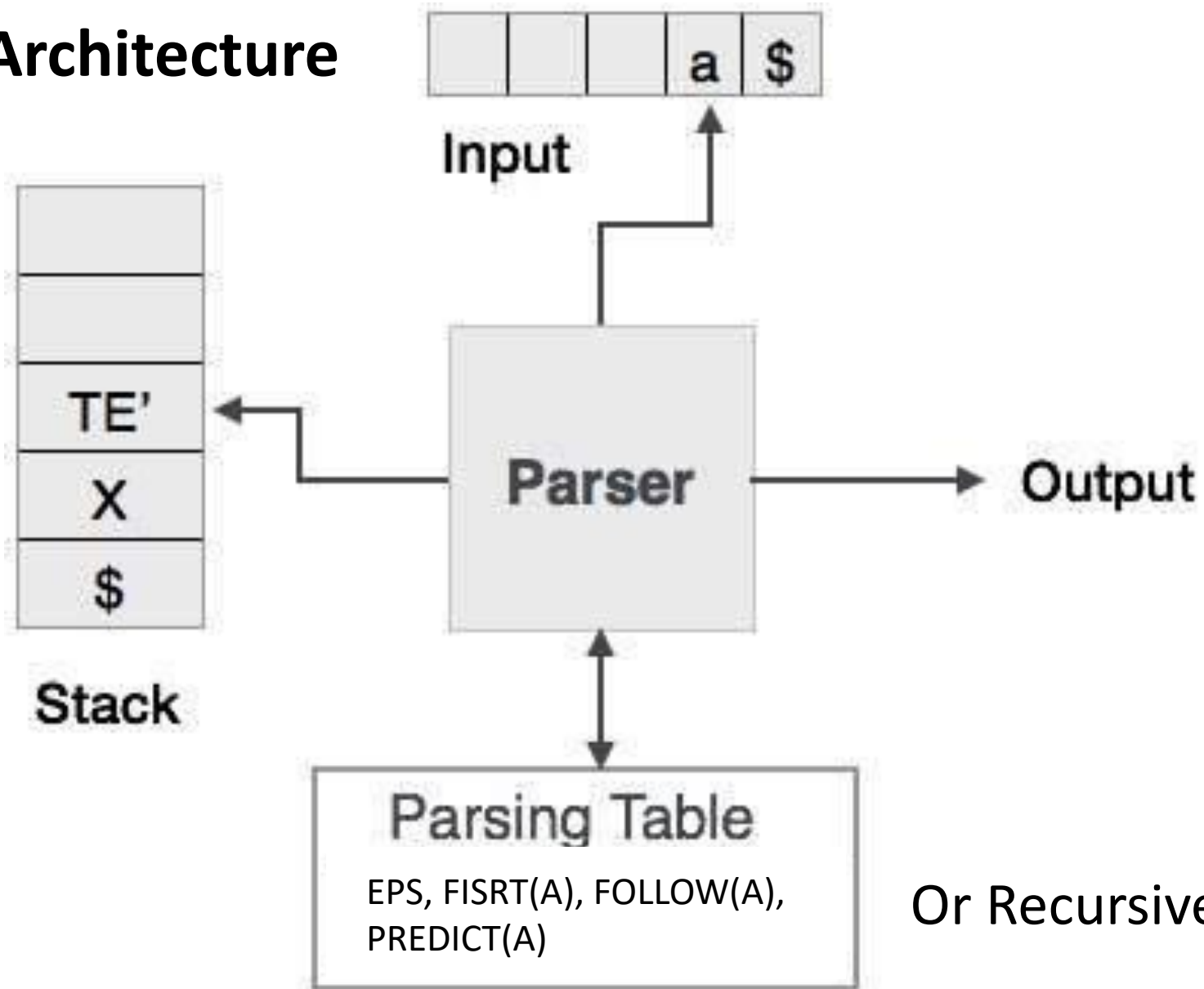
Building a Recursive Descent Parser

SECTION 9

Building a Recursive Descent Parser

- So how do we parse a string with our calculator grammar?
- We can formalize this process in one of two ways.
- The first is to build a recursive descent parser whose subroutines correspond, one-one, to the non-terminals of the grammar. Recursive descent parsers are typically constructed by hand, though the ANTLR parser generator constructs them automatically from an input grammar.
- The second approach, described in Section 2.3.3, is to build an LL parse table which is then read by a driver program. Table-driven parsers are almost always constructed automatically by a parser generator.
- These two options—**recursive descent** and **table-driven**—are reminiscent of the nested case statements and table-driven approaches to building a scanner that we saw in Sections 2.2.2 and 2.2.3. It should be emphasized that they implement the same basic parsing algorithm.

Parser Architecture



Recursive Descent Parser for the Calculator Language

- Pseudocode for a recursive descent parser for our calculator language appears in Figure 2.17.
- It has a subroutine for every nonterminal in the grammar.
- It also has a mechanism input token to inspect the next token available from the scanner and a subroutine (match) to consume and update this token, and in the process verify that it is the one that was expected (as specified by an argument).
- If match or any of the other subroutines sees an unexpected token, then a syntax error has occurred. For the time being let us assume that the parse error subroutine simply prints a message and terminates the parse.
- In Section 2.3.5 we will consider how to recover from such errors and continue to parse the remainder of the input. [\[Backtracking\]](#)

Recursive Procedures for Each Terminal

```

procedure match(expected)
  if input_token = expected then consume_input_token()
  else parse_error
    
```

Next Token

-- this is the start routine:

```

procedure program()
  case input_token of
    id, read, write : $$ :
      stmt_list()
      match($$)
    otherwise parse_error
    
```

$program \rightarrow stmt_list \ \$\$$

Recursive

```

procedure stmt_list()
  case input_token of
    id, read, write : stmt(); stmt_list()
    $$ : skip -- epsilon production
    otherwise parse_error
    
```

$stmt_list \rightarrow stmt \ stmt_list \mid \epsilon$

```

procedure stmt()
  case input_token of
    id : match(id); match(:=); expr()
    read : match(read); match(id)
    write : match(write); expr()
    otherwise parse_error
    
```

$stmt \rightarrow id := expr \mid read \ id \mid write \ expr$

```

procedure expr()
  case input_token of
    id, number, ( : term(); term_tail()
    otherwise parse_error
    
```

$expr \rightarrow term \ term_tail$

```

procedure term_tail()
  case input_token of
    +, - : add_op(); term(); term_tail()
    ), id, read, write : $$ :
      skip -- epsilon production
    otherwise parse_error
    
```

$term_tail \rightarrow add_op \ term \ term_tail \mid \epsilon$

```

procedure term()
  case input_token of
    id, number, ( : factor(); factor_tail()
    otherwise parse_error
    
```

$term \rightarrow factor \ factor_tail$

```

procedure factor_tail()
  case input_token of
    *, / : mult_op(); factor(); factor_tail()
    +, -, ), id, read, write : $$ :
      skip -- epsilon production
    otherwise parse_error
    
```

$factor_tail \rightarrow mult_op \ factor \ factor_tail \mid \epsilon$

```

procedure factor()
  case input_token of
    id : match(id)
    number : match(number)
    ( : match( ( ); expr(); match() )
    otherwise parse_error
    
```

$factor \rightarrow (\ expr \) \mid id \mid number$

```

procedure add_op()
  case input_token of
    + : match(+)
    - : match(-)
    otherwise parse_error
    
```

$add_op \rightarrow + \mid -$

```

procedure mult_op()
  case input_token of
    * : match(*)
    / : match(/)
    otherwise parse_error
    
```

$mult_op \rightarrow * \mid /$

LL Parsing

- Like the bottom-up grammar, this one captures associativity and precedence, but most people don't find it as pretty
 - for one thing, the operands of a given operator aren't in a RHS together!
 - however, the simplicity of the parsing algorithm makes up for this weakness
- How do we parse a string with this grammar?
 - by building the parse tree incrementally

LL Parsing

- Example (average program)

read A

read B

sum := A + B

write sum

write sum / 2

- We start at the top and predict needed productions on the basis of the current left-most non-terminal in the tree and the current input token

Parse tree for the average program

