CS46K Programming Languages

Structure and Interpretation of Computer Programs

Chapter 9A Syntax Analysis – Context Free Grammar

LECTURE 12: CONTEXT FREE GRAMMAR

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

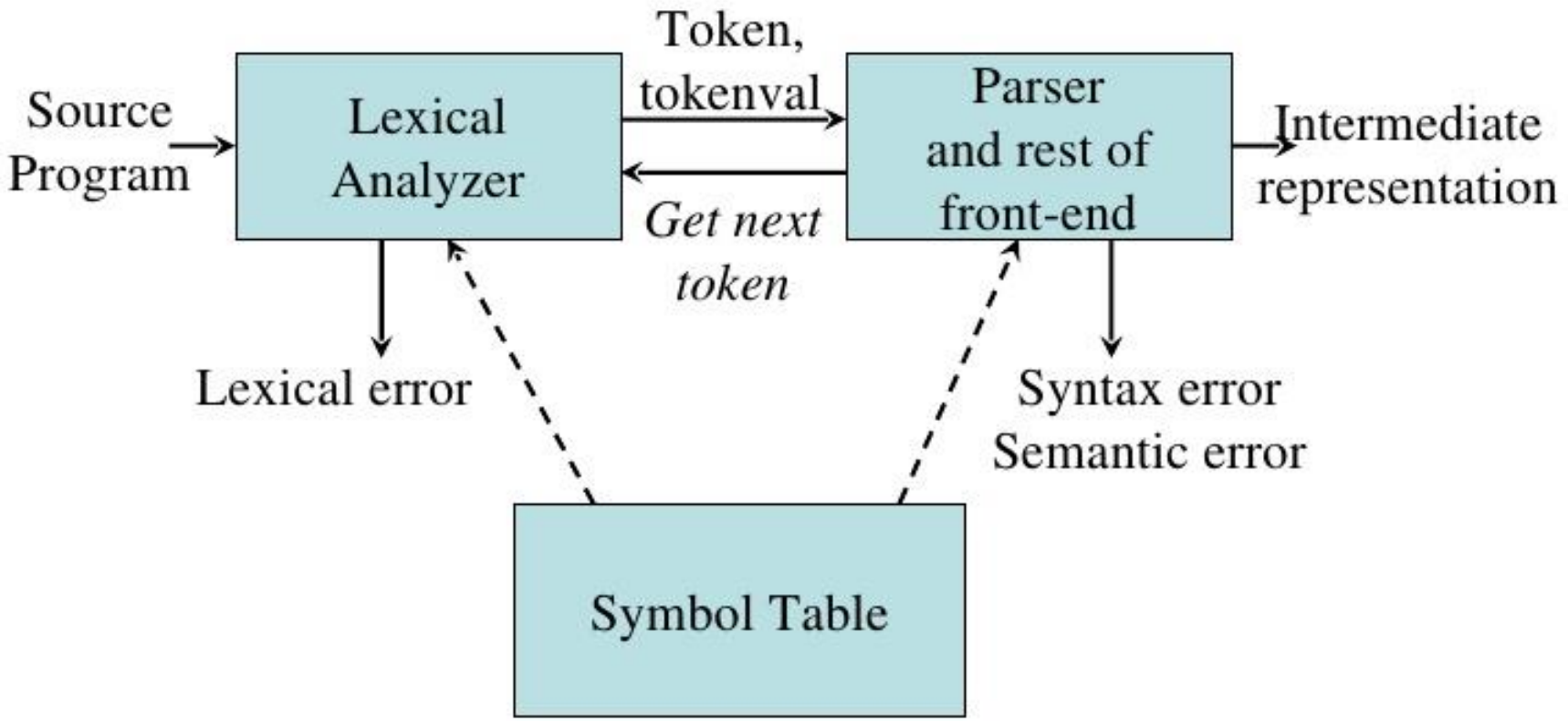
- Context Free Grammar
- Parser Design – Recursive Decent Parser Design
- Table Driven Parsing
- Bottom-Up Parsing

Parsing I

Overview

SECTION 1

Position of a Parser in the Compiler Model

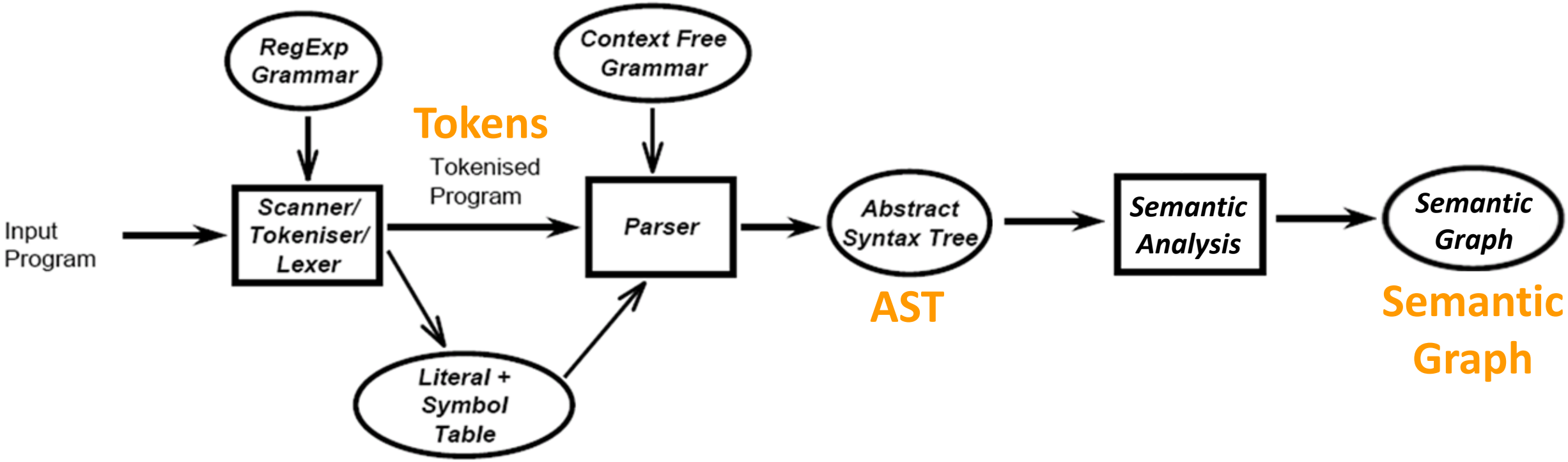


Parser is the Heart of a Typical Compiler

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**.

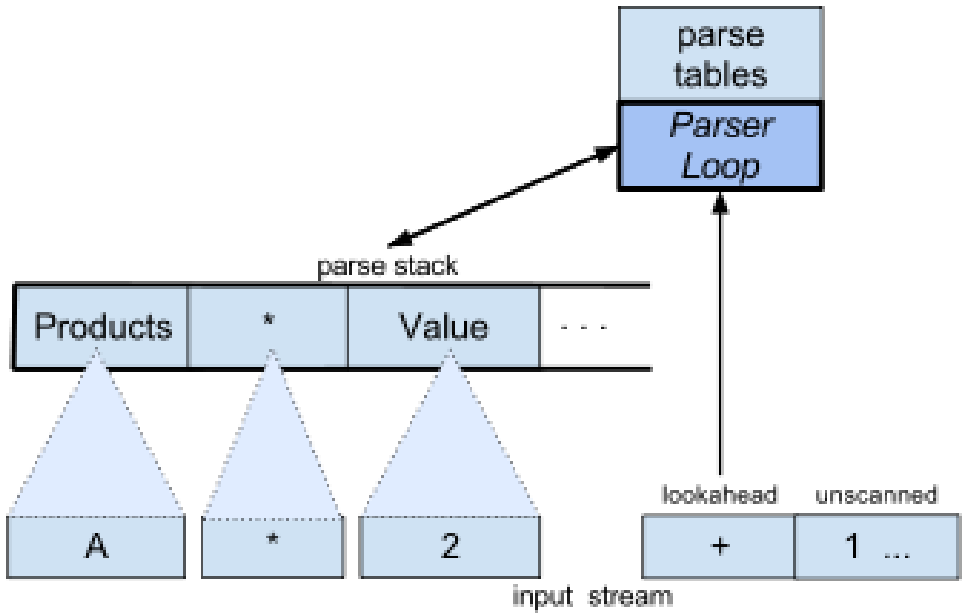
Systematic Compiler Front-End Generation

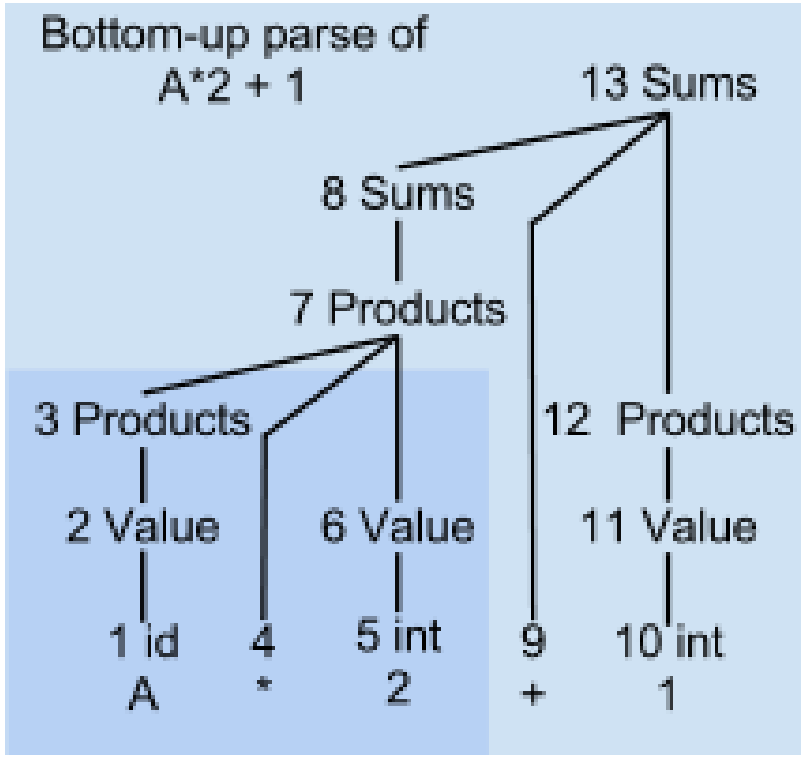


Semantic Graph:
AST with Additional Properties and Resolved References

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 union 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 U T)* that can be derived from **S**.
- Consider the linear grammar
  ({S, B}, {a, b}, S, {S→ aS, S → B, B → bB, B →λ }).
- A derivation using this grammar might look like this:
  $S \Longrightarrow aS \Longrightarrow aB \Longrightarrow abB \Longrightarrow abbB \Longrightarrow abb$
- 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 2

Parsing

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

- 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'.
- LR stands for 'Left-to-right, Rightmost derivation'

LL: L in Top-down Shape

LR: Reverse means bottom up

Parsing

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

- 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:

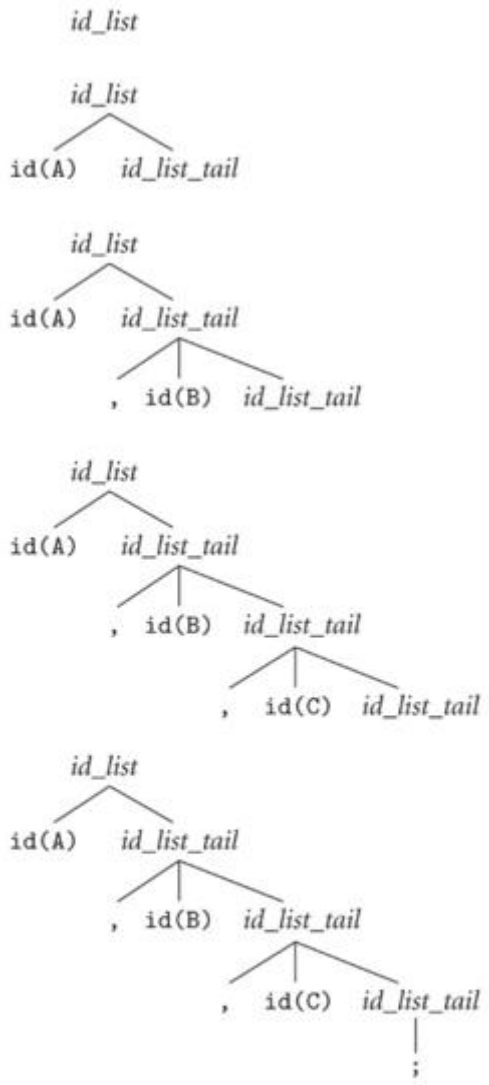
$$
\begin{aligned}
\textit{id\_list} &\longrightarrow \texttt{id}\ \textit{id\_list\_tail} \\
\textit{id\_list\_tail} &\longrightarrow \texttt{,}\ \texttt{id}\ \textit{id\_list\_tail} \\
\textit{id\_list\_tail} &\longrightarrow \texttt{;}
\end{aligned}
$$

- 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.



$id\_list \longrightarrow \texttt{id}\ id\_list\_tail$

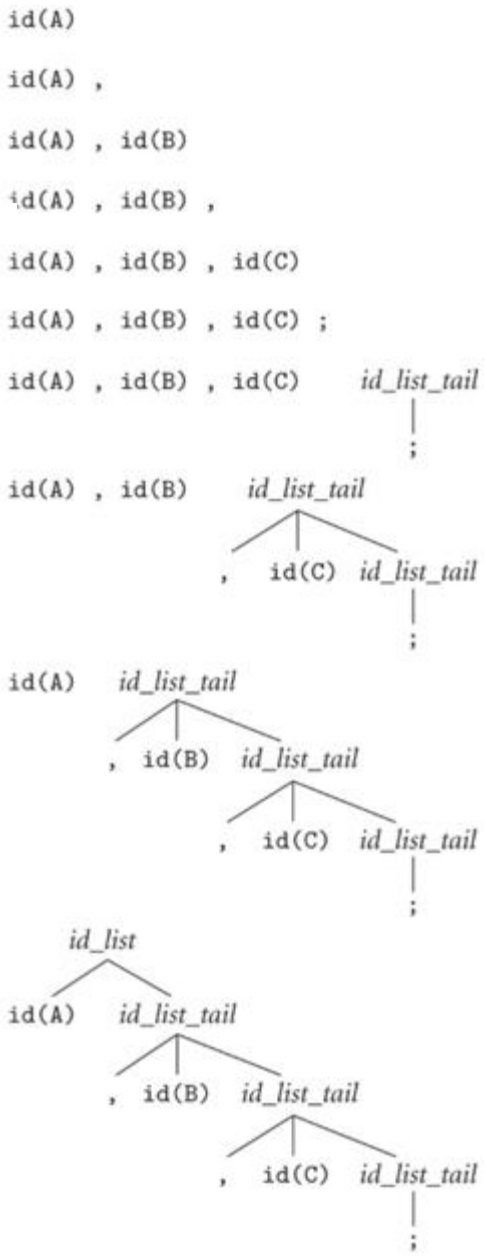
$id\_list\_tail \longrightarrow \texttt{,}\ \texttt{id}\ id\_list\_tail$

$id\_list\_tail \longrightarrow \texttt{;}$

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** → ;.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**.

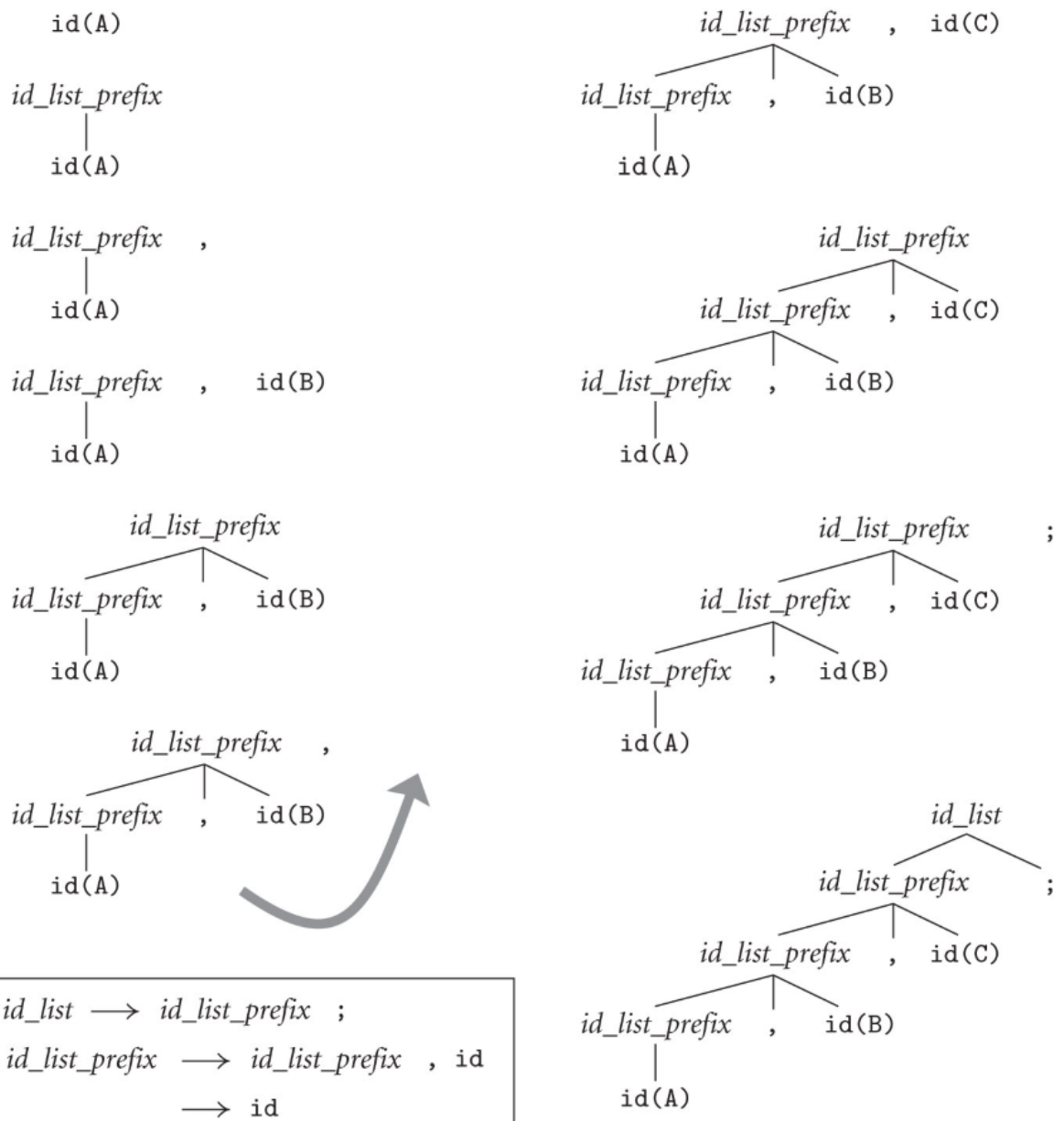
Bounding Space with a Bottom-Up Grammar

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:

$$
\begin{aligned}
\textit{id\_list} &\longrightarrow \textit{id\_list\_prefix} \ ; \\
\textit{id\_list\_prefix} &\longrightarrow \textit{id\_list\_prefix} \ , \ \texttt{id} \\
&\longrightarrow \texttt{id}
\end{aligned}
$$

- 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.



$$
\begin{aligned}
\textit{id\_list} &\longrightarrow \textit{id\_list\_prefix} \ ; \\
\textit{id\_list\_prefix} &\longrightarrow \textit{id\_list\_prefix} \ , \ \texttt{id} \\
&\longrightarrow \texttt{id}
\end{aligned}
$$

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).

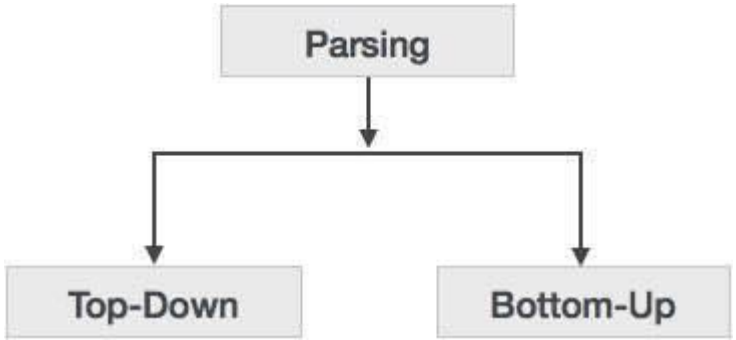
(Expression for Arithmetic, Example 2.8, not covered here)

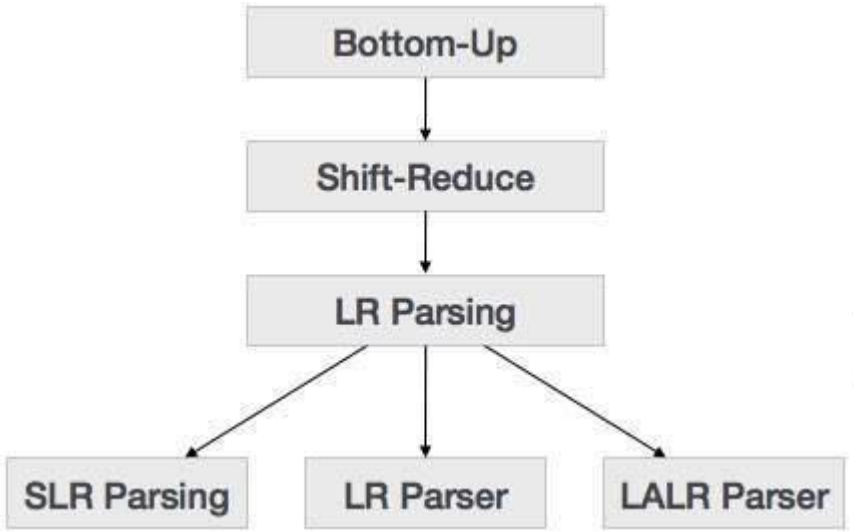
Example 2.8:

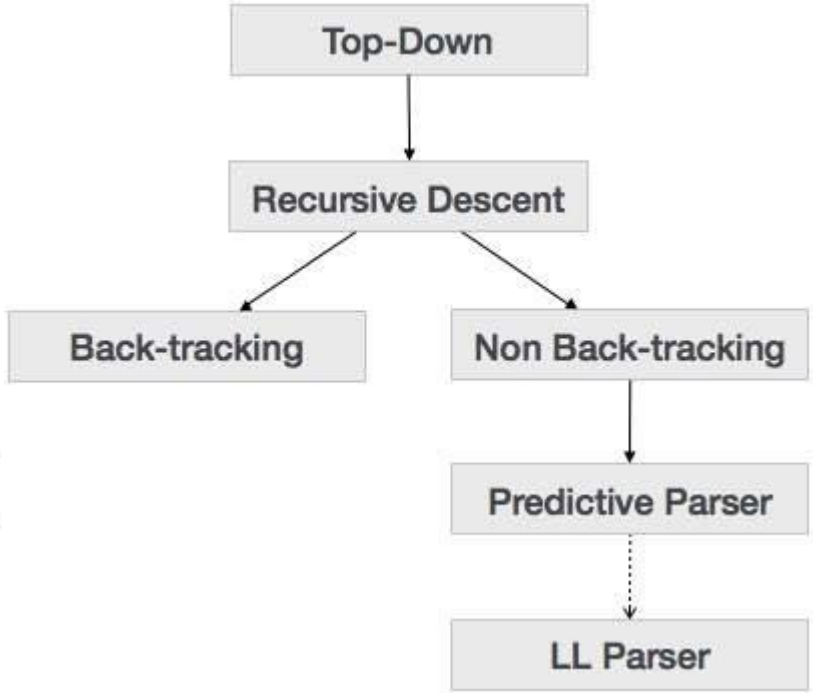
1. *expr* → *term* | *expr* *add_op* *term*
2. *term* → *factor* | *term* *mult_op* *factor*
3. *factor* → `id` | `number` | `-` *factor* | `(` *expr* `)`
4. *add_op* → `+` | `-`
5. *mult_op* → `*` | `/`

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

Types of Parsing







Parsing III

Recursive Descent Parsing

SECTION 3

Top-down Grammar for a Calculator

LL(1) Grammar for a Simple Calculator Language

Figure 2.16

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

- 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* ⟶ *term* | *expr add_op term*
2. *term* ⟶ *factor* | *term mult_op factor*
3. *factor* ⟶ `id` | `number` | `-` *factor* | `(` *expr* `)`
4. *add_op* ⟶ `+` | `-`
5. *mult_op* ⟶ `*` | `/`

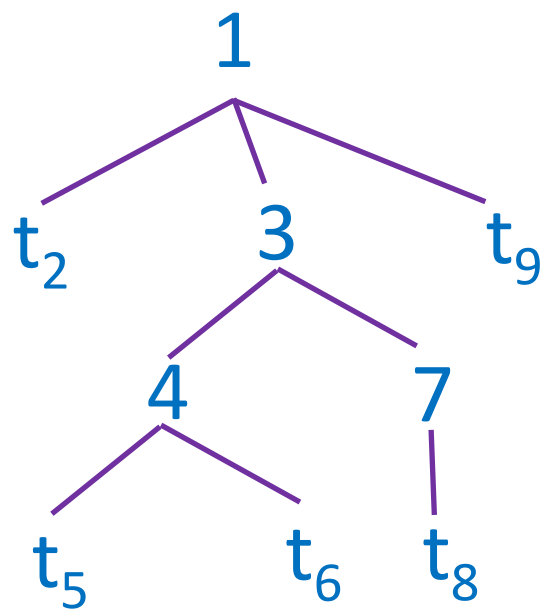
Figure 2.16:

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

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 -> Term | Term + Expr

  Term -> 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:

Expr -> Term | Term + Expr

Term -> int | int * Term | (Expr)

The token stream:

(int)

Next Token

RD Parsing Tree:

Expr

Recursive Descent Parsing Example

The grammar:

Expr -> Term | Term + Expr

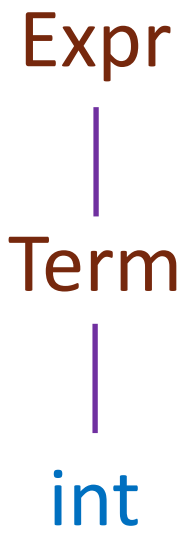
Term -> int | int * Term | (Expr)

The token stream:

(int)

Next Token

RD Parsing Tree:



Mismatch: It does not match!

Backtrack!

Recursive Descent Parsing Example

The grammar:

Expr -> Term | Term + Expr

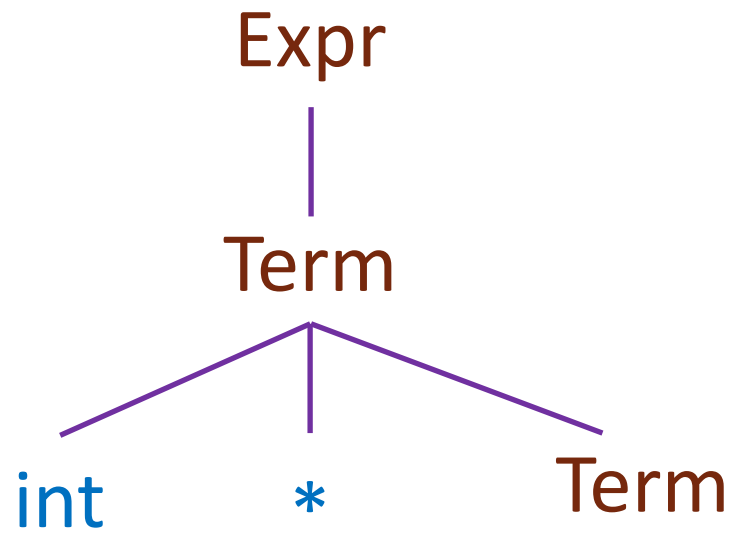
Term -> int | int * Term | (Expr)

The token stream:

(int)

Next Token

RD Parsing Tree:



Mismatch: It does not match!

Backtrack!

Recursive Descent Parsing Example

The grammar:

Expr -> Term | Term + Expr

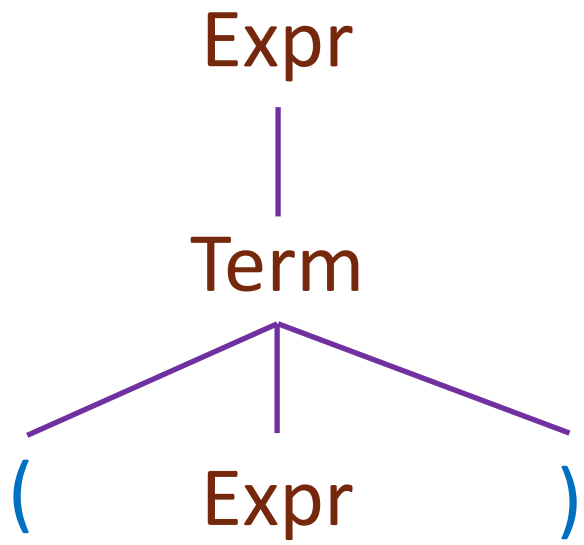
Term -> int | int * Term | (Expr)

The token stream:

(int)

Next Token

RD Parsing Tree:



Matched: Advance Input.

Recursive Descent Parsing Example

The grammar:

Expr -> Term | Term + Expr

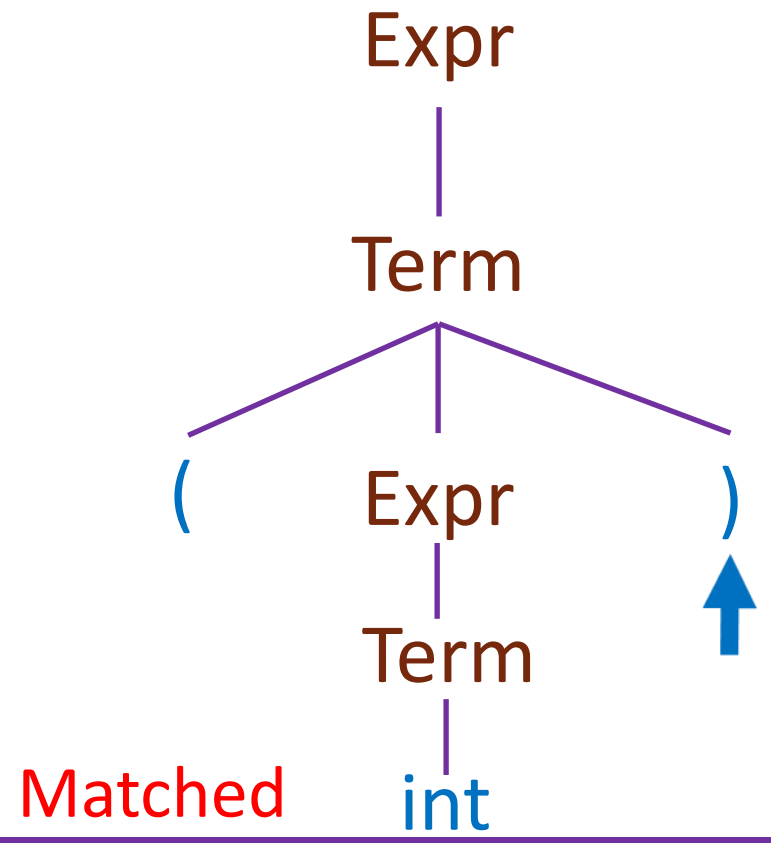
Term -> int | int * Term | (Expr)

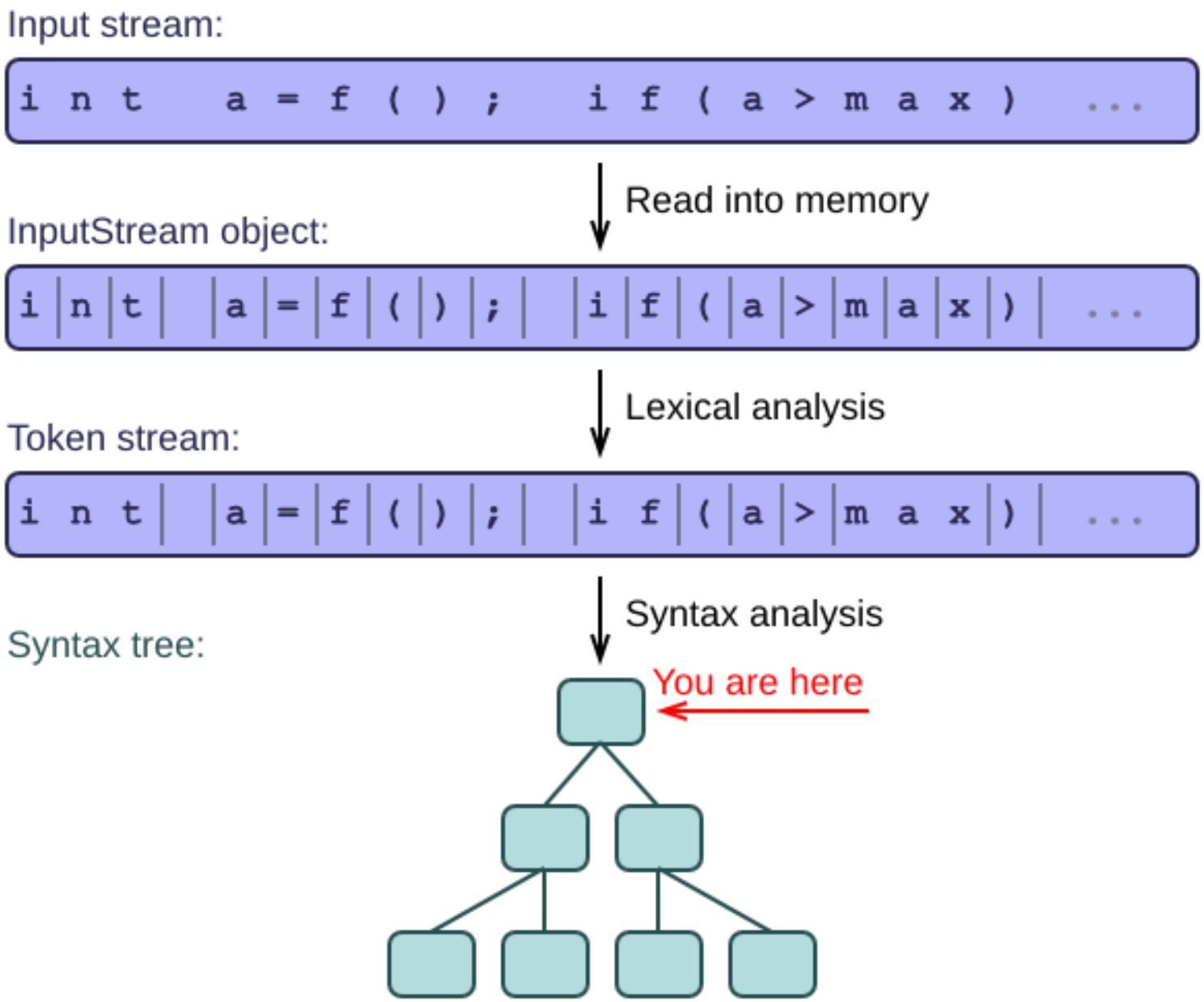
The token stream:

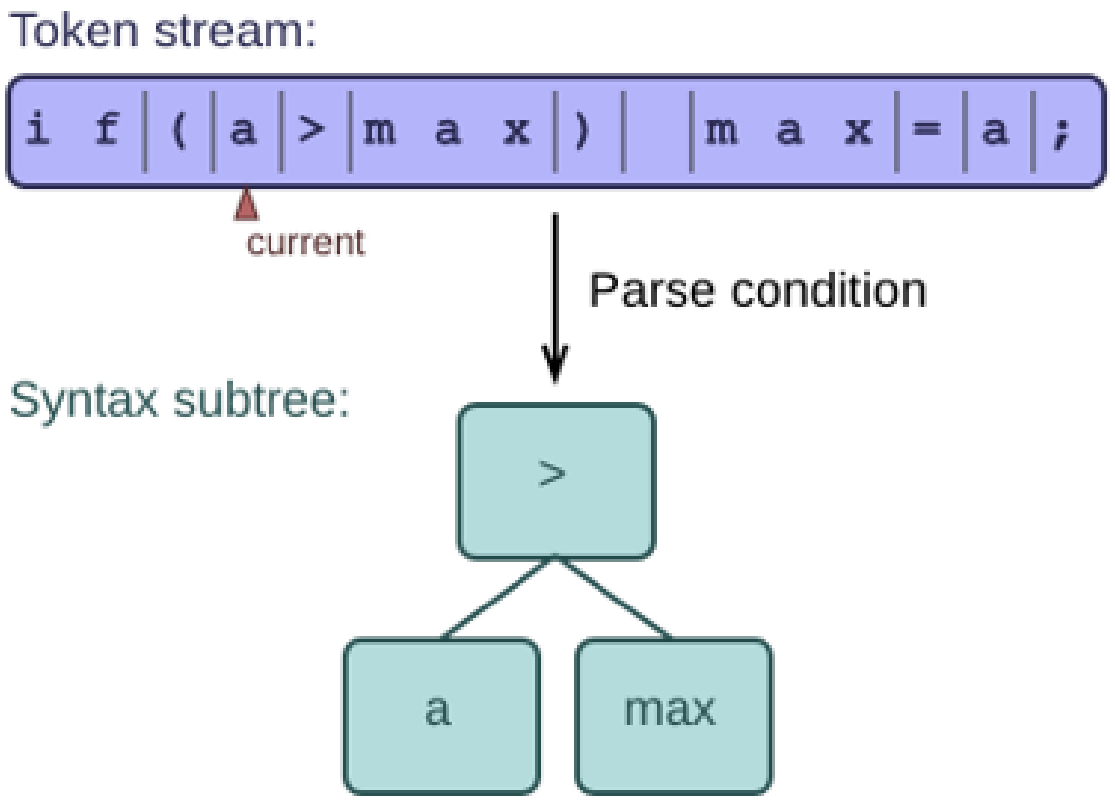
(int)

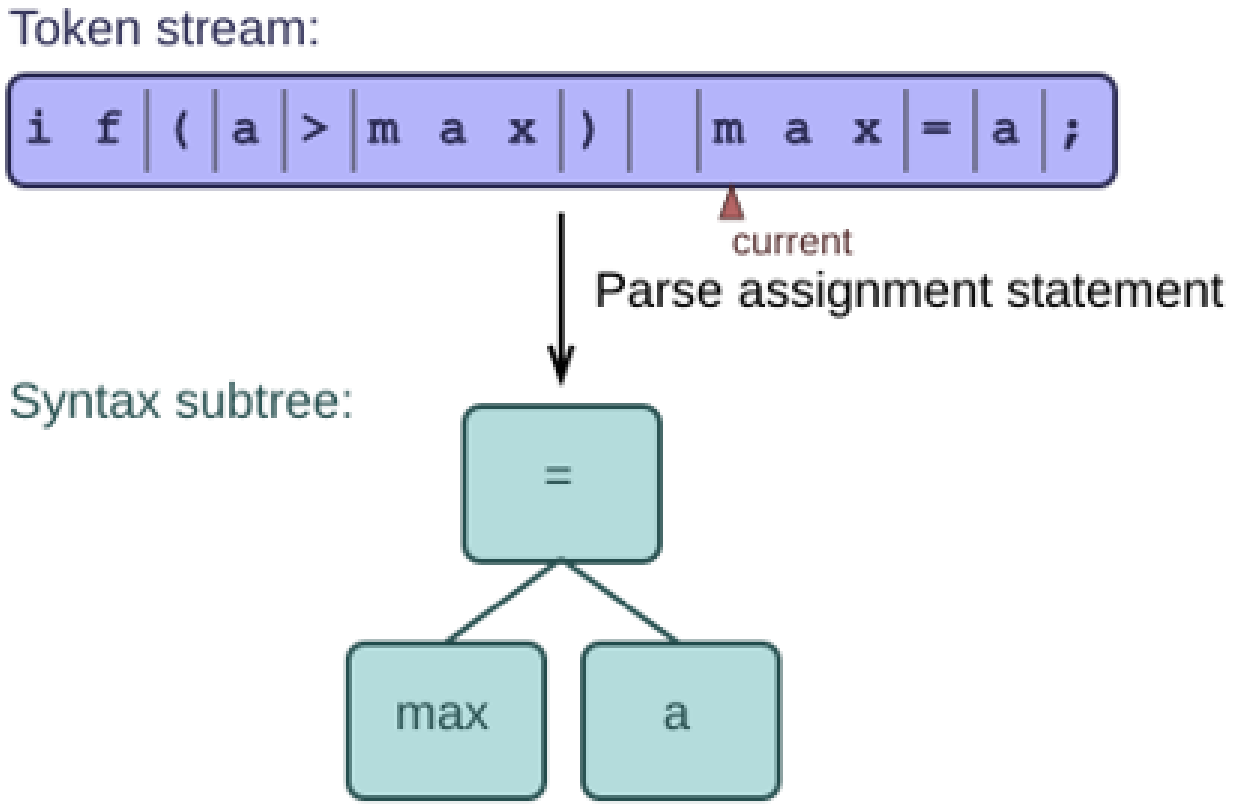
Next Token

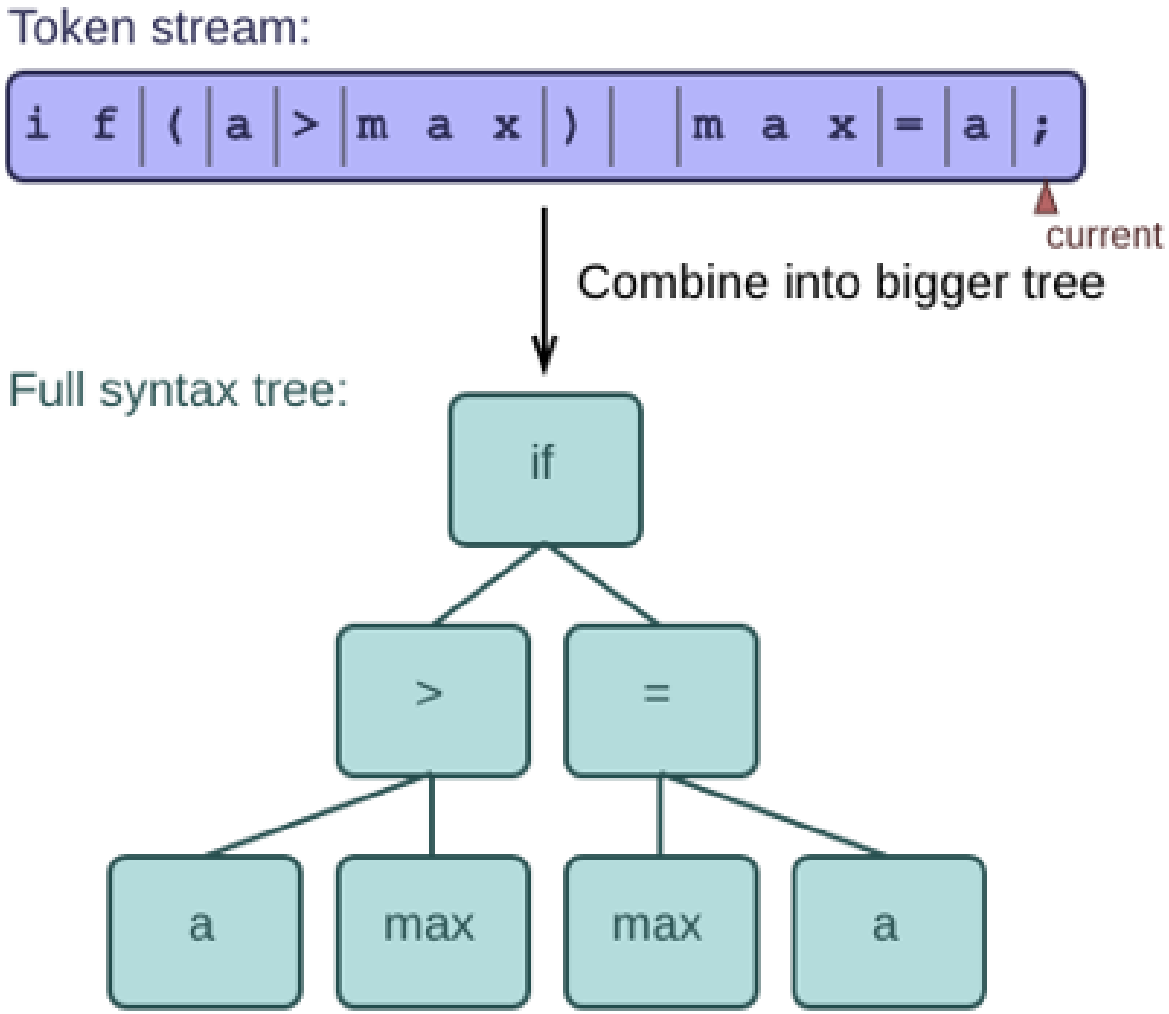
RD Parsing Tree:




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:
You are here


Token stream:
i f (a > m a x) m a x = a ;
current
Parse condition
Syntax subtree:
>
a
max


Token stream:
i f (a > m a x) m a x = a ;
current
Parse assignment statement
Syntax subtree:
=
max
a


Token stream:
i f (a > m a x) m a x = a ;
current
Combine into bigger tree
Full syntax tree:
if
>
=
a
max
max
a

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

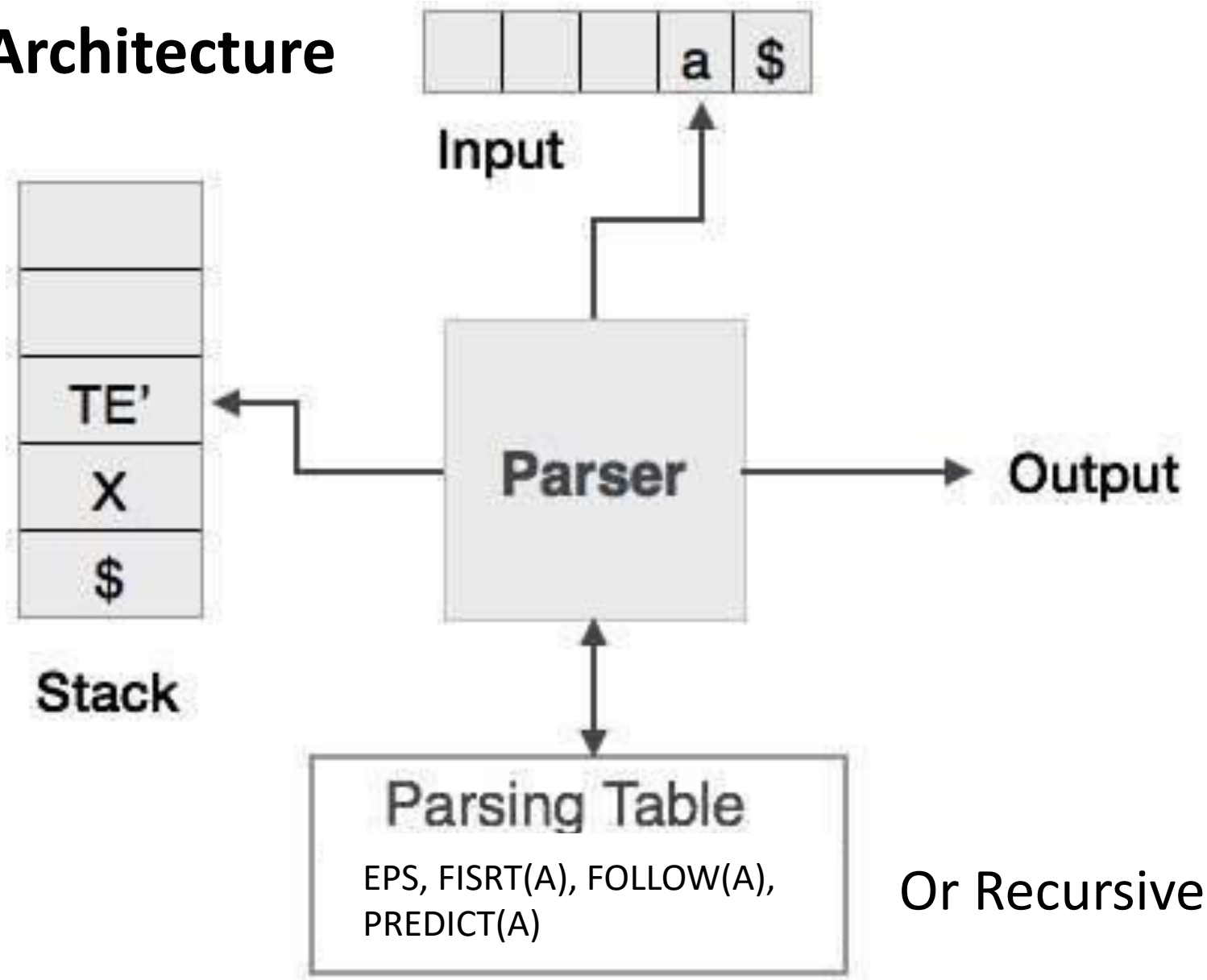
SECTION 4

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.

Handwritten recursive descent parsers are most often used when the language to be parsed is relatively simple, or when a parser-generator tool is not available.

Parser Architecture



Or Recursive Procedures

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
```

$$program \longrightarrow stmt\_list \; \$\$$$

Recursive

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

$$stmt\_list \longrightarrow stmt \; stmt\_list \mid \epsilon$$

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

$$stmt \longrightarrow \texttt{id} := expr \mid \texttt{read id} \mid \texttt{write} \; expr$$

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

$$expr \longrightarrow term \; term\_tail$$

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

$$term\_tail \longrightarrow add\_op \; term \; term\_tail \mid \epsilon$$

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

$$term \longrightarrow factor \; factor\_tail$$

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

$$factor\_tail \longrightarrow mult\_op \; factor \; factor\_tail \mid \epsilon$$

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

$$factor \longrightarrow \texttt{(} \; expr \; \texttt{)} \mid \texttt{id} \mid \texttt{number}$$

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

$$add\_op \longrightarrow \texttt{+} \mid \texttt{-}$$

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

$$mult\_op \longrightarrow \texttt{*} \mid \texttt{/}$$

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

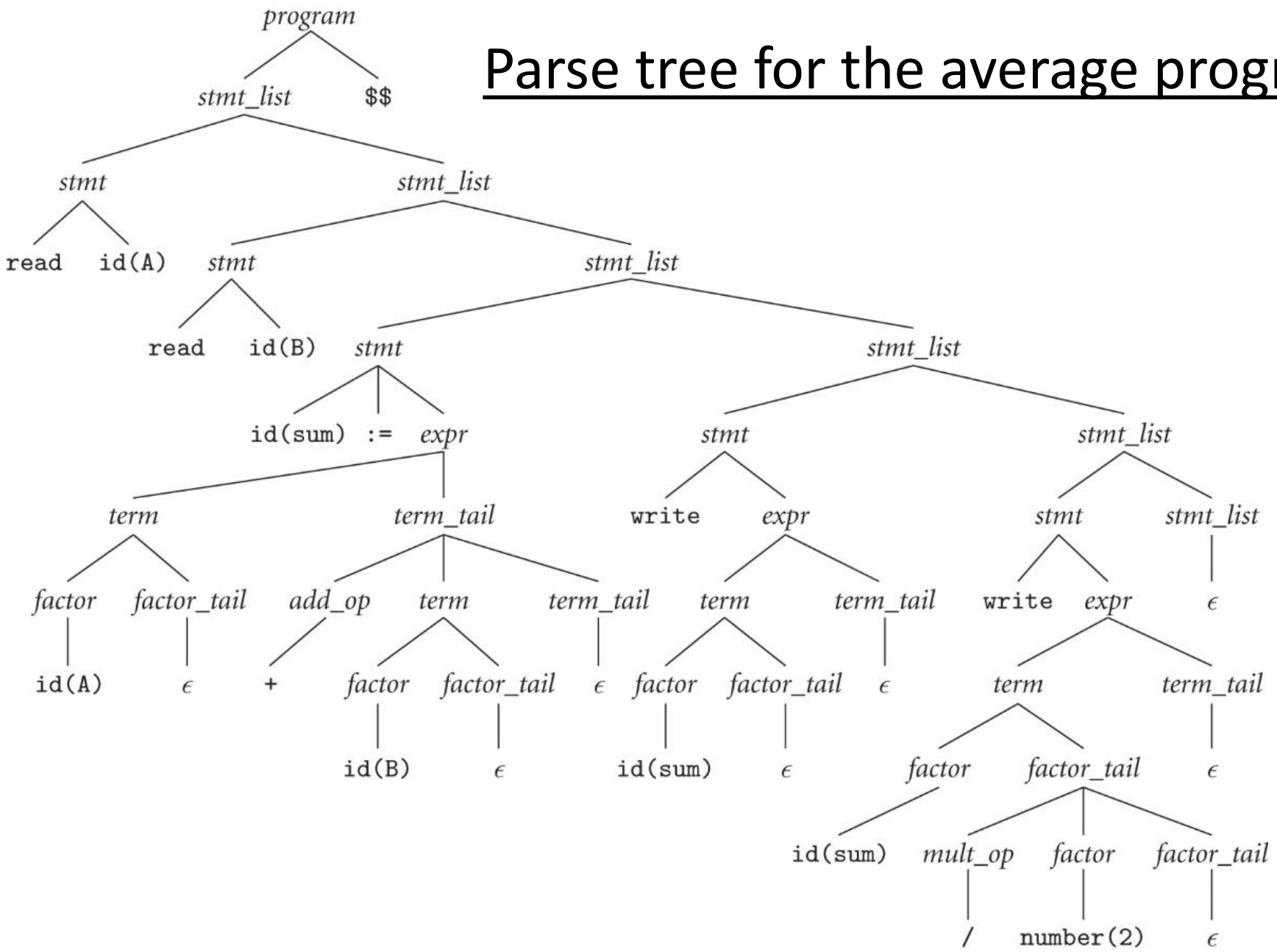
LL Parsing

- 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



Parsing V

Building a Recursive Descent Parser with Tool

SECTION 5

A simple calc.py Recursive Decent Parser

- Download from this lecture's Sample code.
- You should find a folder named "recursive descent". In this folder, there should have 3 files:
  - `calc.py`
  - `lex.py`
  - `parse.py`

Features

- Very simple, Pythonic code with pretty decent performance
- Pretty-ish error messages (through ParseError.print)
- Self-modifying parsers/lexers
- Lazy parsing support for interactive use

Usage

- A very simple calculator example (from which this code is taken) is provided in **calc.py**.
- The **lexer** is constructed with a list of token name/regular expression pairs, specified with Python's regex syntax. A transformation function can optionally be provided, allowing the token to hold any Python value (like the **NUMBER** example above).
- The parser works by translating an **EBNF**-like grammar directly into a recursive descent parser.

calc.py

- Calculator language Token Table.
- Build a lexer named **lexer** based on the Token table
- Calculator language parsing rule list.
- Build a parser named **parser** based on the parsing rules with top level node (‘expr’)
- REPL loop – read from console. Evaluate the tokens using parser.parse(). Then print the parsing results.

calc.py

```
# Building lexer
import decimal
import lex
import parse
num_type = decimal.Decimal

# Lexer tokens. These tokens define all valid input to the parser.
# Whitespace is ignored
table = {
    'PLUS':       r'\+',
    'MINUS':      r'-',
    'TIMES':      r'\*',
    'DIVIDE':     r'/',
    'POWER':      r'\^',
    'NUMBER':     (r'[0-9]+(\.[0-9]*)?|\.[0-9]+',
        lambda t: t.copy(value=num_type(t.value))),
    'LPAREN':     r'\(',
    'RPAREN':     r'\)',
    'WHITESPACE': (r'[ \t\n]+', lambda t: None),
}
lexer = lex.Lexer(table)
```

calc.py

```
def reduce_binop(p):  # reduce arithmetic production rules
    r = p[0]
    for item in p[1]:
        if   item[0] == '+': r = r + item[1]
        elif item[0] == '-': r = r - item[1]
        elif item[0] == '*': r = r * item[1]
        elif item[0] == '/': r = r / item[1]
    return r
```

calc.py

```
# Building parser

# Parse rules. Each rule is a list, with the first element being the
# rule name, and each item after one of the p
rules = [
    ['atom', 'NUMBER', ('LPAREN expr RPAREN', lambda p: p[1])],
    ['factor', ('atom POWER factor', lambda p: p[0] ** p[2]), 'atom',
        ('MINUS factor', lambda p: -p[1])],
    ['term', ('factor ((TIMES|DIVIDE) factor)*', reduce_binop)],
    ['expr', ('term ((PLUS|MINUS) term)*', reduce_binop)],
]

try:
    parser = parse.Parser(rules, 'expr')
except parse.ParseError as e:
    e.print()
    raise
```

Lexer

Class **LexError**: # Lexical Error class

msg : error message

info : error information

Class **Info**: # Lexer's information

filename: program file name

lineno: line number

textpos: text position

column: column location

length: length

lex.py

```
import copy
import re

class LexError(SyntaxError):
    def __init__(self, msg, info=None):
        self.msg = msg
        self.info = info

# Info means basically filename/line number, used for reporting errors
class Info:
    def __init__(self, filename, lineno=1, textpos=0, column=0, length=0):
        self.filename = filename
        self.lineno = lineno
        self.textpos = textpos
        self.column = column
        self.length = length
    def __str__(self):
        return 'Info("%s", %s, %s, %s)' % (self.filename, self.lineno,  \
                                           self.column, self.length)
```

Lexer

Class Token: # token representation

type: token type
value: token value
info: token information

- This Token class is used to represent the data of the tokenization results.

lex.py

```
class Token:
    def __init__(self, type, value, info=None):
        self.type = type
        self.value = value
        self.info = info
    def copy(self, type=None, value=None, info=None):
        c = copy.copy(self)
        if type is not None:  c.type = type
        if value is not None: c.value = value
        if info is not None:  c.info = info
        return c
    def __repr__(self):
        return 'Token(%s, %r, info=%s)' % (self.type, self.value, self.info)
```

Parser

Class **parser**:

See source code

Demo Program

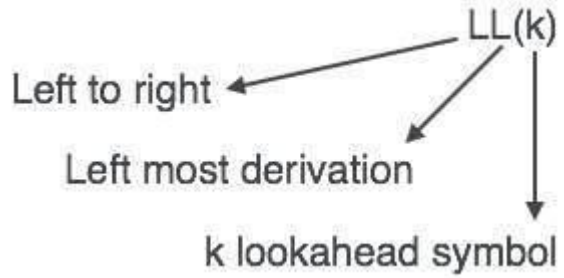
- Run the demonstration program

>>> python calc.py

LL Parsing I

Writing an LL(1) Grammar

SECTION 6



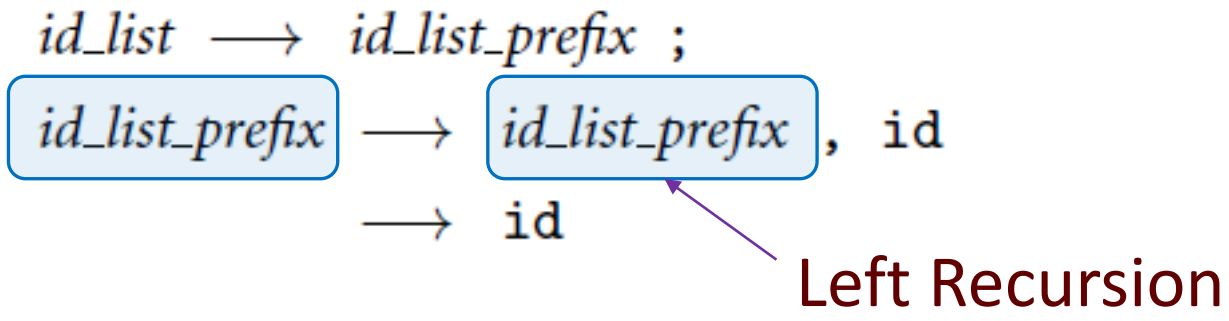
Writing an LL(1) Grammar

- When designing a recursive-descent parser, one has to acquire a certain facility in writing and modifying LL(1) grammars. The two most common obstacles to "LL(1)-ness" are left recursion and common prefixes.
- Transformation of a grammar from non-LL(1)-ness to LL(1)-ness:
  - Elimination of Left-Recursion
  - Left-Factorization

A grammar G is LL(1) if $A \rightarrow \alpha \mid \beta$ are two distinct productions of G:
- for no terminal, both α and β derive strings beginning with a.
- at most one of α and β can derive empty string.
- if $\beta \rightarrow t$, then α does not derive any string beginning with a terminal in FOLLOW(A).

Left Recursion

- A grammar is said to be left recursive if there is a nonterminal A such that A $\Rightarrow^+$ A α for some α.
- The trivial case occurs when the first symbol on the right-hand side of a production is the same as the symbol on the left-hand side.
- The following grammar cannot be parsed top-down:



Elimination of Left Recursion

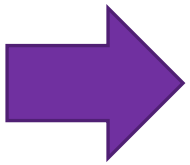
Left-Recursion

id_list ⟶ *id_list_prefix* ;

id_list_prefix ⟶ *id_list_prefix* , `id`

⟶ `id`

id, id, id, …., id;



Right-Recursion

id_list ⟶ `id` *id_list_tail*

id_list_tail ⟶ , `id` *id_list_tail*

id_list_tail ⟶ ;

id, id, id, …., id;

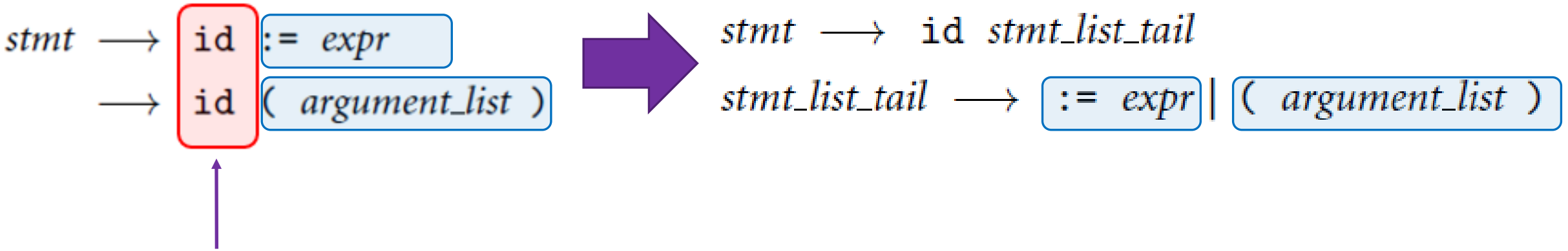
Common Prefixes

Common prefixes occur when two different productions with the same left-hand side begin with the same symbol or symbols.

$$
\begin{aligned}
stmt &\longrightarrow \texttt{id} := expr \\
&\longrightarrow \texttt{id} \; (\; argument\_list \;) \qquad \text{-- procedure call}
\end{aligned}
$$

Both left recursion and common prefixes can be removed from a grammar mechanically.

Left Factorization



Common Left Factor

Note:
A -> a b
A -> a c
Converted to
A -> a B
B -> b | c

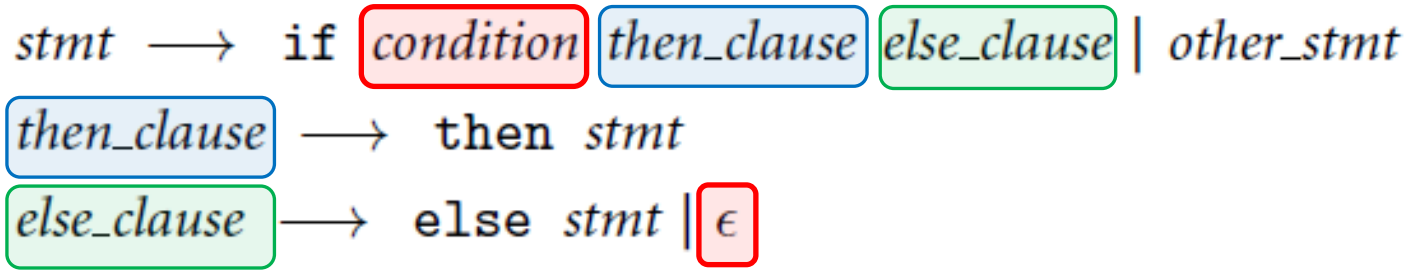
Note:

That eliminating left recursion and common prefixes does NOT make a grammar LL

- there are infinitely many non-LL LANGUAGES, and the mechanical transformations work on them just fine
- the few that arise in practice, however, can generally be handled with heuristics.

Parsing a Dangling Else

The best known example of a “not quite LL” construct arises in languages like Pascal, in which the else part of an if statement is optional. The natural grammar fragment:



is ambiguous (and thus neither LL nor LR); it allows the else in **if C1 then if C2 then S1 else S2** to be paired with either then.

It may belong to first or second **if**.

Removal of Dangling Else

stmt ⟶ *balanced_stmt* | *unbalanced_stmt*

balanced_stmt ⟶ `if` *condition* `then` *balanced_stmt* `else` *balanced_stmt*
| *other_stmt*

unbalanced_stmt ⟶ `if` *condition* `then` *stmt*
| `if` *condition* `then` *balanced_stmt* `else` *unbalanced_stmt*

This can be parse bottom-up but not top-down.

balanced_stmt comes first. This means **else** go with closer **if**.

bottom-up

if C1 then if C2 then S1 else S2

The fact that **else_clause** → else **stmt** comes before **else_clause** → ε ends up pairing the else with the nearest then, as desired.

Removal of Ambiguity

- The usual approach, whether top-down OR bottom-up, is to use the ambiguous grammar together with a disambiguating rule that says
  - else goes with the closest then or
  - more generally, the first of two possible productions is the one to predict (or reduce)

End Markers for Structured Statements

- Many other Algol-family languages (including Modula, Modula-2, and Oberon, all more recent inventions of Pascal's designer, Niklaus Wirth) require explicit end markers on all structured statements. The grammar fragment for if statements in Modula-2 looks something like this:

$$\begin{aligned} \textit{stmt} &\longrightarrow \texttt{IF}\ \textit{condition}\ \textit{then\_clause}\ \textit{else\_clause}\ \texttt{END} \mid \textit{other\_stmt} \\ \textit{then\_clause} &\longrightarrow \texttt{THEN}\ \textit{stmt\_list} \\ \textit{else\_clause} &\longrightarrow \texttt{ELSE}\ \textit{stmt\_list} \mid \epsilon \end{aligned}$$

The addition of the END eliminates the ambiguity.

The Need for elseif

Ambiguous

```
if A = B then ...
else if A = C then ...
else if A = D then ...
else if A = E then ...
else ...
```

With end markers this becomes

```
if A = B then ...
else if A = C then ...
else if A = D then ...
else if A = E then ...
else ...
end end end end
```

With elseif

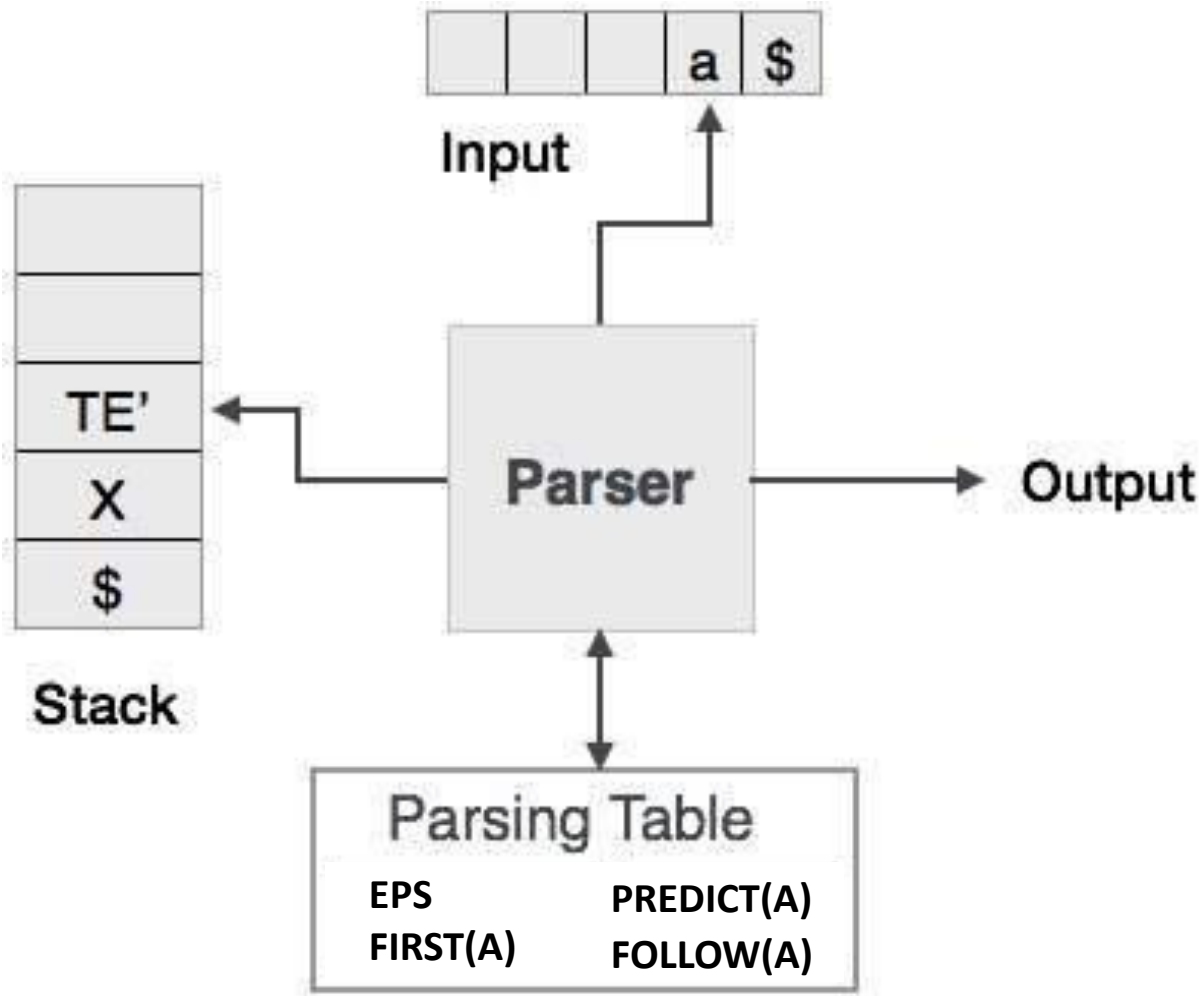
```
if A = B then ...
elsif A = C then ...
elsif A = D then ...
elsif A = E then ...
else ...
end
```

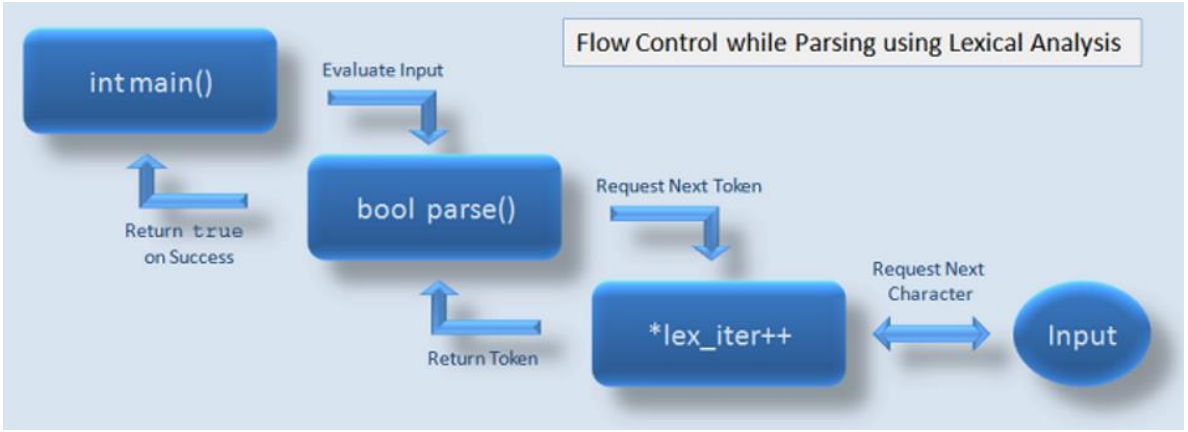
LL Parsing II

Overview of Table-Driven Top-down Parser

SECTION 7

Table-Driven Top-Down Parser





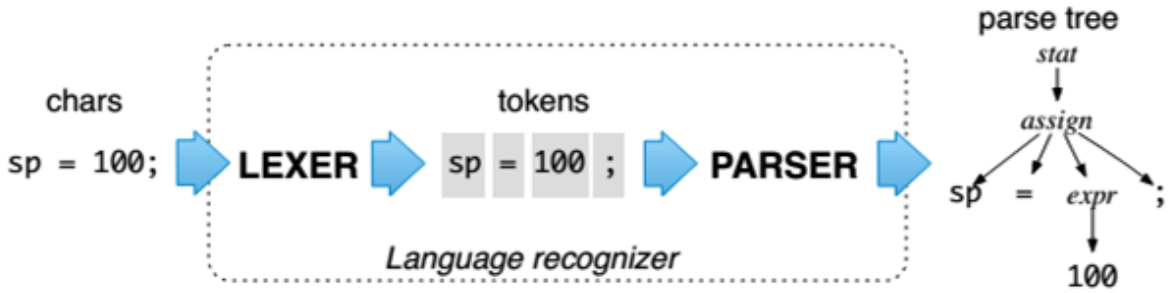


Table-driven LL parsing

- you have a big loop in which you repeatedly look up an action in a two-dimensional table based on current leftmost non-terminal and current input token. The actions are

  (1) match a terminal

  (2) predict a production

  (3) announce a syntax error

Driver and Table for Top-Down Parsing

In a recursive descent parser, each arm of a case statement corresponds to a production, and contains parsing routine and match calls corresponding to the symbols on the right-hand side of that production.

At any given point in the parse, if we consider the calls beyond the program counter (the ones that have yet to occur) in the parsing routine invocations currently in the **call stack**, we obtain a list of the symbols that the parser expects to see between here and the end of the program.

A table-driven top-down parser maintains an explicit stack containing this same list of symbols.

Driver Table –Driven LL(1) Parser

- The code is language independent. It requires a language-dependent parsing table, generally produced by an automatic tool. For the calculator grammar of Figure 2.16, the table appears in Figure 2.20.

Figure 2.19

```
terminal = 1 . . number_of_terminals
non_terminal = number_of_terminals + 1 . . number_of_symbols
symbol = 1 . . number_of_symbols
production = 1 . . number_of_productions

parse_tab : array [non_terminal, terminal] of record
      action : (predict, error)
      prod : production
prod_tab : array [production] of list of symbol
-- these two tables are created by a parser generator tool

parse_stack : stack of symbol

parse_stack.push(start_symbol)
loop
      expected_sym : symbol := parse_stack.pop()
      if expected_sym ∈ terminal
            match(expected_sym)                    -- as in Figure 2.17
            if expected_sym = $$ then return       -- success!
      else
            if parse_tab[expected_sym, input_token].action = error
                  parse_error
            else
                  prediction : production := parse_tab[expected_sym, input_token].prod
                  foreach sym : symbol in reverse prod_tab[prediction]
                        parse_stack.push(sym)
```

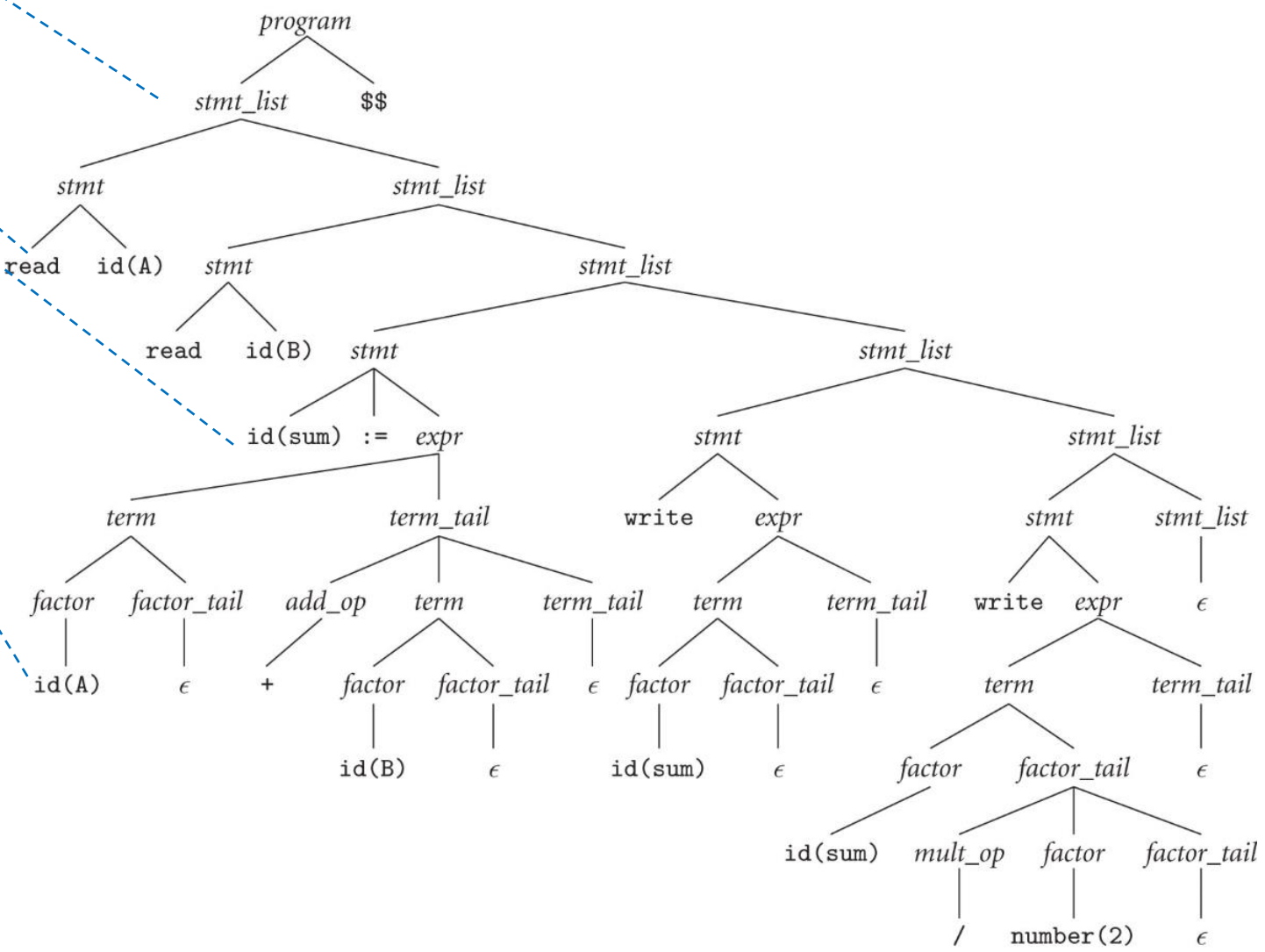
Top-of-stack nonterminal	Current input token											
	`id`	`number`	`read`	`write`	`:=`	`(`	`)`	`+`	`-`	`*`	`/`	`$$`
program	1	–	1	1	–	–	–	–	–	–	–	1
stmt_list	2	–	2	2	–	–	–	–	–	–	–	3
stmt	4	–	5	6	–	–	–	–	–	–	–	–
expr	7	7	–	–	–	7	–	–	–	–	–	–
term_tail	9	–	9	9	–	–	9	8	8	–	–	9
term	10	10	–	–	–	10	–	–	–	–	–	–
factor_tail	12	–	12	12	–	–	12	12	12	11	11	12
factor	14	15	–	–	–	13	–	–	–	–	–	–
add_op	–	–	–	–	–	–	–	16	17	–	–	–
mult_op	–	–	–	–	–	–	–	–	–	18	19	–

Figure 2.20

Table-driven parse of the "sum and average" program

- The parser iterates around a loop in which it pops the top symbol off the stack and performs the following actions. If the popped symbol is a terminal, the parser attempts to match it against an incoming token from the scanner.
- If the match fails, the parser announces a syntax error and initiates some sort of error recovery.
- If the popped symbol is a nonterminal, the parser uses that nonterminal together with the next available input token to index into a two-dimensional table that tells it which production to predict (or whether to announce a syntax error and initiate recovery).

Parse stack	Input stream	Comment
program	`read A read B ...`	initial stack contents
stmt_list `$$`	`read A read B ...`	predict *program* ⟶ *stmt_list* `$$`
stmt stmt_list `$$`	`read A read B ...`	predict *stmt_list* ⟶ *stmt stmt_list*
`read id` *stmt_list* `$$`	`read A read B ...`	predict *stmt* ⟶ `read id`
`id` *stmt_list* `$$`	`A read B ...`	match `read`
stmt_list `$$`	`read B sum := ...`	match `id`
stmt stmt_list `$$`	`read B sum := ...`	predict *stmt_list* ⟶ *stmt stmt_list*
`read id` *stmt_list* `$$`	`read B sum := ...`	predict *stmt* ⟶ `read id`
`id` *stmt_list* `$$`	`B sum := ...`	match `read`
stmt_list `$$`	`sum := A + B ...`	match `id`
stmt stmt_list `$$`	`sum := A + B ...`	predict *stmt_list* ⟶ *stmt stmt_list*
`id :=` *expr stmt_list* `$$`	`sum := A + B ...`	predict *stmt* ⟶ `id :=` *expr*
`:=` *expr stmt_list* `$$`	`:= A + B ...`	match `id`
expr stmt_list `$$`	`A + B ...`	match `:=`
term term_tail stmt_list `$$`	`A + B ...`	predict *expr* ⟶ *term term_tail*
factor factor_tail term_tail stmt_list `$$`	`A + B ...`	predict *term* ⟶ *factor factor_tail*
`id` *factor_tail term_tail stmt_list* `$$`	`A + B ...`	predict *factor* ⟶ `id`
factor_tail term_tail stmt_list `$$`	`+ B write sum ...`	match `id`
term_tail stmt_list `$$`	`+ B write sum ...`	predict *factor_tail* ⟶ ϵ
add_op term term_tail stmt_list `$$`	`+ B write sum ...`	predict *term_tail* ⟶ *add_op term term_tail*
`+` *term term_tail stmt_list* `$$`	`+ B write sum ...`	predict *add_op* ⟶ `+`
term term_tail stmt_list `$$`	`B write sum ...`	match `+`
factor factor_tail term_tail stmt_list `$$`	`B write sum ...`	predict *term* ⟶ *factor factor_tail*
`id` *factor_tail term_tail stmt_list* `$$`	`B write sum ...`	predict *factor* ⟶ `id`
factor_tail term_tail stmt_list `$$`	`write sum ...`	match `id`
term_tail stmt_list `$$`	`write sum write ...`	predict *factor_tail* ⟶ ϵ
stmt_list `$$`	`write sum write ...`	predict *term_tail* ⟶ ϵ
stmt stmt_list `$$`	`write sum write ...`	predict *stmt_list* ⟶ *stmt stmt_list*
`write` *expr stmt_list* `$$`	`write sum write ...`	predict *stmt* ⟶ `write` *expr*
expr stmt_list `$$`	`sum write sum / 2`	match `write`
term term_tail stmt_list `$$`	`sum write sum / 2`	predict *expr* ⟶ *term term_tail*
factor factor_tail term_tail stmt_list `$$`	`sum write sum / 2`	predict *term* ⟶ *factor factor_tail*
`id` *factor_tail term_tail stmt_list* `$$`	`sum write sum / 2`	predict *factor* ⟶ `id`
factor_tail term_tail stmt_list `$$`	`write sum / 2`	match `id`
term_tail stmt_list `$$`	`write sum / 2`	predict *factor_tail* ⟶ ϵ
stmt_list `$$`	`write sum / 2`	predict *term_tail* ⟶ ϵ
stmt stmt_list `$$`	`write sum / 2`	predict *stmt_list* ⟶ *stmt stmt_list*
`write` *expr stmt_list* `$$`	`write sum / 2`	predict *stmt* ⟶ `write` *expr*
expr stmt_list `$$`	`sum / 2`	match `write`
term term_tail stmt_list `$$`	`sum / 2`	predict *expr* ⟶ *term term_tail*
factor factor_tail term_tail stmt_list `$$`	`sum / 2`	predict *term* ⟶ *factor factor_tail*
`id` *factor_tail term_tail stmt_list* `$$`	`sum / 2`	predict *factor* ⟶ `id`
factor_tail term_tail stmt_list `$$`	`/ 2`	match `id`
mult_op factor factor_tail term_tail stmt_list `$$`	`/ 2`	predict *factor_tail* ⟶ *mult_op factor factor_tail*
`/` *factor factor_tail term_tail stmt_list* `$$`	`/ 2`	predict *mult_op* ⟶ `/`
factor factor_tail term_tail stmt_list `$$`	`2`	match `/`
`number` *factor_tail term_tail stmt_list* `$$`	`2`	predict *factor* ⟶ `number`
factor_tail term_tail stmt_list `$$`		match `number`
term_tail stmt_list `$$`		predict *factor_tail* ⟶ ϵ
stmt_list `$$`		predict *term_tail* ⟶ ϵ
`$$`		predict *stmt_list* ⟶ ϵ



Trace of the Parsing Process

- To keep track of the left-most non-terminal, you push the as-yet-unseen portions of productions onto a stack
  - for details see Figure 2.21
- The key thing to keep in mind is that the stack contains all the stuff you expect to see between now and the end of the program
  - what you predict you will see

LL Parsing III

Rules for Building Predict sets

SECTION 8

To Build the Predict Sets for the Calculator Language

- The algorithm to build predict sets is tedious (for a "real" sized grammar), but relatively simple
- It consists of three stages:

  (1) compute FIRST sets for symbols

  (2) compute FOLLOW sets for non-terminals
  (this requires computing FIRST sets for some strings)

  (3) compute **predict sets** or **table** for all productions

Set Definitions

Given any CFG in BNF (no alternation | or Kleene *, +), we can construct the following sets.

Each set contains only tokens, and FIRST(***A***), FOLLOW(***A***) and PREDICT(***A***) are defined for every non-terminal A in the language.

- EPS: All non-terminals that could expand to ε, the empty string. [All non-terminals which has leaf node of ε]
- FIRST(***A***): The set of tokens that could appear as the first token in an expansion of ***A***.
- FOLLOW(***A***): The set of tokens that could appear immediately after A in an expansion of S, the start symbol .
- PREDICT(***A***): The set of tokens that could appear next in a valid parse of string in the language, when the next non-terminal in the parse tree is *A*.

Each set is defined using mutual recursion.

To Build the Predict Sets for the Calculator Language

- Algorithm First/Follow/Predict:
  - FIRST(α) == {a : α →* a β}
    U (if α =>* ε THEN {ε} ELSE NULL)
  - FOLLOW(A) == {a : S →$^{+}$ α A a β}
    U (if S →* α A THEN {ε} ELSE NULL)
  - Predict (A → X_1 ... X_m) == (FIRST (X_1 ... X_m) - {ε}) U
    (if X_1, ..., X_m →* ε then FOLLOW (A) ELSE NULL)
- Details following…

Do we need FIRST set in parsing?

NO!

For top-town parsing, the only sets we really need are the PREDICT sets.

It turns out the FIRST sets are not necessary to computer PREDICT.

So (for now) we will forget about FIRST and just worry about the other three (EPS, PREDICT, FOLLOW).

EPS

Definition: EPS contains all non-terminals that could expand to ε, the empty string.

EPS contains:

1) Any non-terminals that has an epsilon production. // exampel: int main(){ /* code block here is ε */}
2) Any non-terminals that has a production containing only non-terminals that are all in EPS.

PREDICT

1. A → token … =>PREDICT(A) += {token}
2. A → B … =>PREDICT(A) += PREDICT(B)
3. A → ε | … =>PREDICT(A) += FOLLOW(A)

Definition: PREDICT(***A***) contains all tokens that could appear next on the token stream when we are expecting an ***A***.

PREDICT(***A***) contains:

1) Any token that appears as the leftmost symbol in a production rule for ***A***.
2) For every non-terminal ***B*** that appears as the leftmost symbol in a production rule for ***A***, every token in PREDICT(***B***).
3) If ***A*** ∈ EPS, every token FOLLOW(***A***).

FOLLOW

1. C → A token => FOLLOW(A) += {token}
2. C → A B => FOLLOW(A) += PREDICT(B)
3. B → … A => FOLLOW(A) += FOLLOW(B)

Definition: FOLLOW(***A***) contains all tokens that could come right after ***A*** is a valid parse.

FOLLOW(***A***) contains:

1. Any token that immediately follows ***A*** on any right-hand side of a production
2. For any non-terminal ***B*** that immediately follows ***A*** on any right-hand side of a production, every token in PREDICT(***B***)
3. For any non-terminal ***B*** such that ***A*** appears right-most in a right-hand side of a production of **B**, every token in FOLLOW(***B***).

There is also the special rule for the start symbol that FOLLOW(***S***) always contains ***$***, the end-of-file token.

LL Parsing IV

Building Predict Sets – an Example

SECTION 9

Example: Calculator Language

Different One from Text book (Another LL(1) Calculator Grammar)

$$
\begin{aligned}
S &\rightarrow exp\ \texttt{STOP} \\
exp &\rightarrow term\ exptail \\
exptail &\rightarrow \epsilon \mid \texttt{OPA}\ term\ exptail \\
term &\rightarrow sfactor\ termtail \\
termtail &\rightarrow \epsilon \mid \texttt{OPM}\ factor\ termtail \\
sfactor &\rightarrow \texttt{OPA}\ factor \mid factor \\
factor &\rightarrow \texttt{NUM} \mid \texttt{LP}\ exp\ \texttt{RP}
\end{aligned}
$$

Computing PREDICT for the calculator language

EPS = {}

Non-terminal	PREDICT	FOLLOW
S		$
exp		
exptail		
term		
termtail		
sfactor		
factor		

S -> exp STOP
exp -> term exptail
exptail -> ε | OPA term exptail
term -> sfactor termtail
termtail -> ε | OPM factor termtail
sfactor -> OPA factor | factor
factor -> NUM | LP exp RP

Computing PREDICT for the calculator language

EPS = {*exptail*, *termtail*}

Non-terminal	PREDICT	FOLLOW
S		$
exp		
exptail		
term		
termtail		
sfactor		
factor		

S -> exp STOP
exp -> term exptail
exptail -> ε | OPA term exptail
term -> sfactor termtail
termtail -> ε | OPM factor termtail
sfactor -> OPA factor | factor
factor -> NUM | LP exp RP

- First construct EPS; we just have to use Rule 1.

Computing PREDICT for the calculator language

EPS = {*exptail*, *termtail*}

Non-terminal	PREDICT	FOLLOW
S		$
exp		
exptail	OPA	
term		
termtail	OPM	
sfactor	OPA	
factor	NUM, LP	

S -> exp STOP
exp -> term exptail
exptail -> ε | OPA term exptail
term -> sfactor termtail
termtail -> ε | OPM factor termtail
sfactor -> OPA factor | factor
factor -> NUM | LP exp RP

Note: A | B can be considered as two separate two rules.

- Apply Rule 1 for PREDICT in four places

PREDICT Rule 1: Any token that appears as the leftmost symbol in a production rule for A.

Computing PREDICT for the calculator language

EPS = {*exptail*, *termtail*}

Non-terminal	PREDICT	FOLLOW
S		$
exp		STOP,RP
exptail	OPA	
term		
termtail	OPM	
sfactor	OPA	
factor	NUM, LP	

S $ default

S -> exp STOP
exp -> term exptail
exptail -> ε | OPA term exptail
term -> sfactor termtail
termtail -> ε | OPM factor termtail
sfactor -> OPA factor | factor
factor -> NUM | LP exp RP

- Apply Rule 1 for FOLLOW in two places

FOLLOW Rule 1: Any token that immediately follows A on any right-hand side of a production

Computing PREDICT for the calculator language

EPS = {*exptail*, *termtail*}

Non-terminal	PREDICT	FOLLOW
S		$
exp		STOP,RP
exptail	OPA	STOP,RP
term		
termtail	OPM	
sfactor	OPA	
factor	NUM, LP	

FOLLOW(exptail) += FOLLOW(exp)

S -> exp STOP
exp -> term exptail
exptail -> ε | OPA term exptail
term -> sfactor termtail
termtail -> ε | OPM factor termtail
sfactor -> OPA factor | factor
factor -> NUM | LP exp RP

- Apply Rule 3 for FOLLOW to *exptail*

FOLLOW RULE 3: For any non-terminal B such that A appears right-most in a right-hand side of a production of B, every token in FOLLOW(B).

Computing PREDICT for the calculator language

EPS = {*exptail*, *termtail*}

Non-terminal	PREDICT	FOLLOW
S		$
exp		STOP,RP
exptail	OPA,STOP,RP	STOP,RP
term		
termtail	OPM	
sfactor	OPA	
factor	NUM, LP	

PREDICT(exptail) += FOLLOW(exptail)

S -> exp STOP
exp -> term exptail
exptail -> ε | OPA term exptail
term -> sfactor termtail
termtail -> ε | OPM factor termtail
sfactor -> OPA factor | factor
factor -> NUM | LP exp RP

- Apply Rule 3 for PREDICT to *exptail*

PREDICT RULE 3: If A∈ EPS, every token FOLLOW(A).

Computing PREDICT for the calculator language

EPS = {*exptail*, *termtail*}

Non-terminal	PREDICT	FOLLOW
S		$
exp		STOP,RP
exptail	OPA,STOP,RP	STOP,RP
term		OPA,STOP,RP
termtail	OPM	
sfactor	OPA	
factor	NUM, LP	

FOLLOW(term) += PREDICT(extail)

S -> exp STOP
exp -> term exptail
exptail -> ε | OPA term exptail
term -> sfactor termtail
termtail -> ε | OPM factor termtail
sfactor -> OPA factor | factor
factor -> NUM | LP exp RP

- Apply Rule 2 for FOLLOW to *term*

FOLLOW Rule 2: For any non-terminal B that immediately follows A on any right-hand side of a production, every token in PREDICT(B)

Computing PREDICT for the calculator language

EPS = {*exptail*, *termtail*}

Non-terminal	PREDICT	FOLLOW
S		$
exp		STOP,RP
exptail	OPA,STOP,RP	STOP,RP
term		OPA,STOP,RP
termtail	OPM	OPA,STOP,RP
sfactor	OPA	
factor	NUM, LP	

FOLLOW(termtail) += FOLLOW(term)

S -> exp STOP
exp -> term exptail
exptail -> ε | OPA term exptail
term -> sfactor termtail
termtail -> ε | OPM factor termtail
sfactor -> OPA factor | factor
factor -> NUM | LP exp RP

FOLLOW RULE 3: For any non-terminal B such that A appears right-most in a right-hand side of a production of B, every token in FOLLOW(B).

- Apply Rule 3 for FOLLOW to *termtail*

Computing PREDICT for the calculator language

EPS = {*exptail*, *termtail*}

Non-terminal	PREDICT	FOLLOW
S		$
exp		STOP,RP
exptail	OPA,STOP,RP	STOP,RP
term		OPA,STOP,RP
termtail	OPM,OPA,STOP,RP	OPA,STOP,RP
sfactor	OPA	
factor	NUM, LP	

PREDICT(termtail) += FOLLOW(termtail)

S -> exp STOP
exp -> term exptail
exptail -> ε | OPA term exptail
term -> sfactor termtail
termtail -> ε | OPM factor termtail
sfactor -> OPA factor | factor
factor -> NUM | LP exp RP

- Apply Rule 3 for PREDICT to *termtail*

PREDICT RULE 3: If A∈ EPS, every token FOLLOW(A).

Computing PREDICT for the calculator language

EPS = {*exptail*, *termtail*}

Non-terminal	PREDICT	FOLLOW
S		$
exp		STOP,RP
exptail	OPA,STOP,RP	STOP,RP
term		OPA,STOP,RP
termtail	OPM,OPA,STOP,RP	OPA,STOP,RP
sfactor	OPA	OPM,OPA,STOP,RP
factor	NUM, LP	OPM,OPA,STOP,RP

S -> exp STOP
exp -> term exptail
exptail -> ε | OPA term exptail
term -> sfactor termtail
termtail -> ε | OPM factor termtail
sfactor -> OPA factor | factor
factor -> NUM | LP exp RP

FOLLOW(sfactor) += PREDICT(termtail)
FOLLOW(factor) += PREDICT(termtail)

- Apply Rule 2 for FOLLOW in two places

FOLLOW Rule 2: For any non-terminal B that immediately follows A on any right-hand side of a production, every token in PREDICT(B)

Computing PREDICT for the calculator language

$EPS = \{exptail, termtail\}$

PREDICT(sfactor)+= PREDICT(factor)
PREDICT(term)+= PREDICT(sfactor)
PREDICT(exp)+= PREDICT(term)
PREDICT(S)+= PREDICT(exp)

Non-terminal	PREDICT	FOLLOW
S	OPA,NUM,LP	$
exp	OPA,NUM,LP	STOP,RP
exptail	OPA,STOP,RP	STOP,RP
term	OPA,NUM,LP	OPA,STOP,RP
termtail	OPM,OPA,STOP,RP	OPA,STOP,RP
sfactor	OPA,NUM,LP	OPM,OPA,STOP,RP
factor	NUM, LP	OPM,OPA,STOP,RP

S -> exp STOP
exp -> term exptail
exptail -> ε | OPA term exptail
term -> sfactor termtail
termtail -> ε | OPM factor termtail
sfactor -> OPA factor | factor
factor -> NUM | LP exp RP

- Apply Rule 2 for PREDICT from the bottom up

PREDICT Rule 2: For every non-terminal B that appears as the leftmost symbol in a production rule for A, every token in PREDICT(B).

Computing PREDICT for the calculator language

EPS = {*exptail*, *termtail*}

Non-terminal	PREDICT	FOLLOW
S	OPA,NUM,LP	$
exp	OPA,NUM,LP	STOP,RP
exptail	OPA,STOP,RP	STOP,RP
term	OPA,NUM,LP	OPA,STOP,RP
termtail	OPM,OPA,STOP,RP	OPA,STOP,RP
sfactor	OPA,NUM,LP	OPM,OPA,STOP,RP
factor	NUM, LP	OPM,OPA,STOP,RP

- Check that none of the rules add anything to any set.

Tagging Set Elements

FIRST is easy to be found. Now, both FOLLOW and PREDICT sets are found.

For use in top-down parsing, every symbol in EPS and in any PREDICT set is tagged with an ε–production rule, as follows:

- Every non-terminal in EPS is tagged with the rule that gives in ε-production for that non-terminal.
- Every token in PREDICT(***A***) that was added from Rule 1 or Rule 2 on **Slide PREDICT-DEFINITION** is tagged with the production ***A*** that brought to that token.
- Every token in PREDICT(***A***) that was added from Rule 3 on **Slide PREDICT-DEFINITION** is tagged with the tag on ***A*** in EPS; that is, the rule that gives the ε-production of ***A***.

These tags indicate what the top-down parser should do when expect ***A*** and sees a token in PREDICT(***A***).

```
-- EPS values and FIRST sets for all symbols:
    for all terminals c, EPS(c) := false; FIRST(c) := {c}
    for all nonterminals X, EPS(X) := if X ⟶ ϵ then true else false; FIRST(X) := ∅
    repeat
        ⟨outer⟩ for all productions X ⟶ Y_1 Y_2 ... Y_k,
            ⟨inner⟩ for i in 1..k
                add FIRST(Y_i) to FIRST(X)
                if not EPS(Y_i) (yet) then continue outer loop
            EPS(X) := true
    until no further progress

-- Subroutines for strings, similar to inner loop above:

    function string_EPS(X_1 X_2 ... X_n)
        for i in 1..n
            if not EPS(X_i) then return false
        return true

    function string_FIRST(X_1 X_2 ... X_n)
        return_value := ∅
        for i in 1..n
            add FIRST(X_i) to return_value
            if not EPS(X_i) then return

-- FOLLOW sets for all symbols:
    for all symbols X, FOLLOW(X) := ∅
    repeat
        for all productions A ⟶ α B β,
            add string_FIRST(β) to FOLLOW(B)
        for all productions A ⟶ α B
                or A ⟶ α B β, where string_EPS(β) = true,
            add FOLLOW(A) to FOLLOW(B)
    until no further progress

-- PREDICT sets for all productions:
    for all productions A ⟶ α
        PREDICT(A ⟶ α) := string_FIRST(α) ∪ (if string_EPS(α) then FOLLOW(A) else ∅)
```

program ⟶ *stmt_list* `$$`	`$$` ∈ FOLLOW(*stmt_list*)
stmt_list ⟶ *stmt* *stmt_list*	
stmt_list ⟶ ϵ	EPS(*stmt_list*) = true
stmt ⟶ `id :=` *expr*	`id` ∈ FIRST(*stmt*)
stmt ⟶ `read id`	`read` ∈ FIRST(*stmt*)
stmt ⟶ `write` *expr*	`write` ∈ FIRST(*stmt*)
expr ⟶ *term* *term_tail*	
term_tail ⟶ *add_op* *term* *term_tail*	
term_tail ⟶ ϵ	EPS(*term_tail*) = true
term ⟶ *factor* *factor_tail*	
factor_tail ⟶ *mult_op* *factor* *factor_tail*	
factor_tail ⟶ ϵ	EPS(*factor_tail*) = true
factor ⟶ `(` *expr* `)`	`(` ∈ FIRST(*factor*) and `)` ∈ FOLLOW(*expr*)
factor ⟶ `id`	`id` ∈ FIRST(*factor*)
factor ⟶ `number`	`number` ∈ FIRST(*factor*)
add_op ⟶ `+`	`+` ∈ FIRST(*add_op*)
add_op ⟶ `-`	`-` ∈ FIRST(*add_op*)
mult_op ⟶ `*`	`*` ∈ FIRST(*mult_op*)
mult_op ⟶ `/`	`/` ∈ FIRST(*mult_op*)

Easy Rules:
EPS Rule
FIRST rules
FOLLOW Rule 1
Predict Rule 1

Figure 2.22 “Obvious” facts about the LL(1) calculator grammar

FIRST

program {id, read, write, $$}
stmt_list {id, read, write}
stmt {id, read, write}
expr {(, id, number}
term_tail {+, -}
term {(, id, number}
factor_tail {*, /}
factor {(, id, number}
add_op {+, -}
mult_op {*, /}

FOLLOW

program ∅
stmt_list {$$}
stmt {id, read, write, $$}
expr {), id, read, write, $$}
term_tail {), id, read, write, $$}
term {+, -,), id, read, write, $$}
factor_tail {+, -,), id, read, write, $$}
factor {+, -, *, /,), id, read, write, $$}
add_op {(, id, number}
mult_op {(, id, number}

PREDICT

1. *program* ⟶ *stmt_list* $$ {id, read, write, $$}
2. *stmt_list* ⟶ *stmt stmt_list* {id, read, write}
3. *stmt_list* ⟶ ϵ {$$}
4. *stmt* ⟶ id := *expr* {id}
5. *stmt* ⟶ read id {read}
6. *stmt* ⟶ write *expr* {write}
7. *expr* ⟶ *term term_tail* {(, id, number}
8. *term_tail* ⟶ *add_op term term_tail* {+, -}
9. *term_tail* ⟶ ϵ {), id, read, write, $$}
10. *term* ⟶ *factor factor_tail* {(, id, number}
11. *factor_tail* ⟶ *mult_op factor factor_tail* {*, /}
12. *factor_tail* ⟶ ϵ {+, -,), id, read, write, $$}
13. *factor* ⟶ (*expr*) {(}
14. *factor* ⟶ id {id}
15. *factor* ⟶ number {number}
16. *add_op* ⟶ + {+}
17. *add_op* ⟶ - {-}
18. *mult_op* ⟶ * {*}
19. *mult_op* ⟶ / {/}

Figure 2.23 FIRST, FOLLOW, and PREDICT sets for the calculator language. EPS(A) is true iff A ∈ {stmt list, term tail, factor tail}.

Predict Ambiguity

- If any token belongs to the predict set of more than one production with the same LHS, then the grammar is not LL(1)
- A conflict can arise because
  - the same token can begin more than one RHS
  - it can begin one RHS and can also appear *after* the LHS in some valid program, and one possible RHS is ε

LR Parsing I

LR Parser and Bottom-UP Parsing

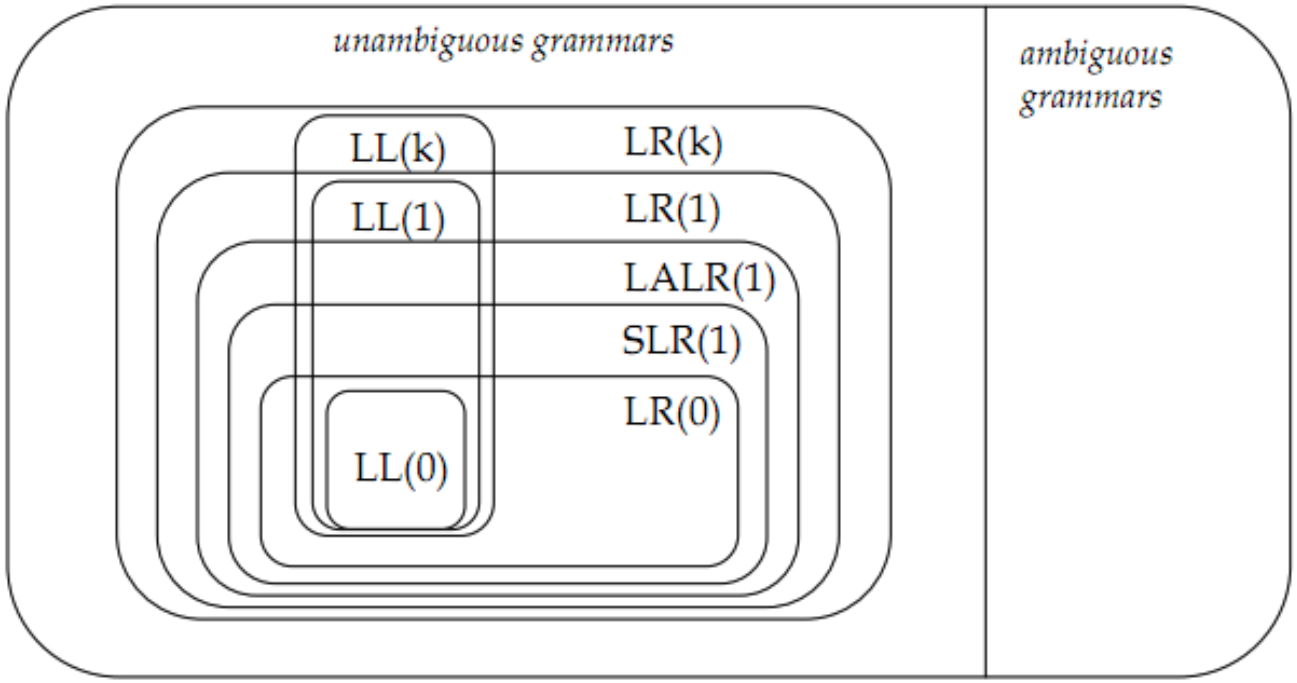
SECTION 10

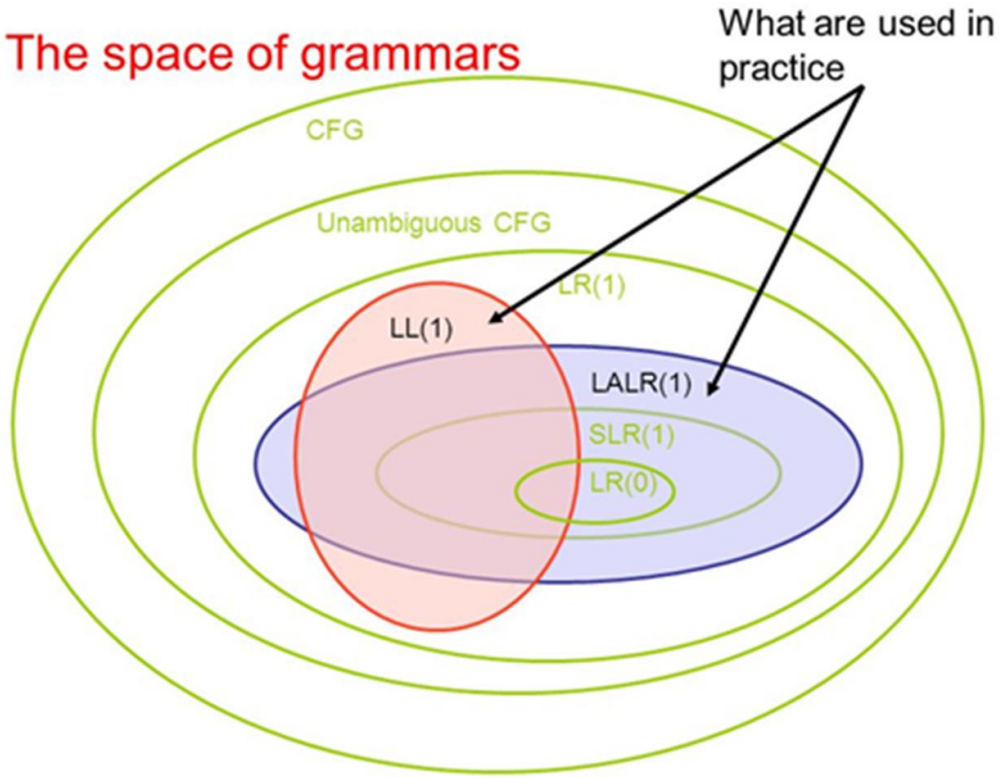
LR-Family Parsing

Note: Not in Textbook

- The shift-reduce method to be described here is called **LR-parsing**. There are a number of variants (hence the use of the term LR-family), but they all use the same driver. They differ only in the generated table. The L in LR indicates that the string is parsed from left to right; the R indicates that the reverse of a right derivation is produced.
- Given a grammar, we want to develop a deterministic bottom-up method for parsing legal strings described by the grammar. As in top-down parsing, we do this with a table and a driver which operates on the table.

LL VS LR Grammars





Note: LR(0)∈SLR(1) ∈LALR(1) ∈LR(1) ∈LR(k)

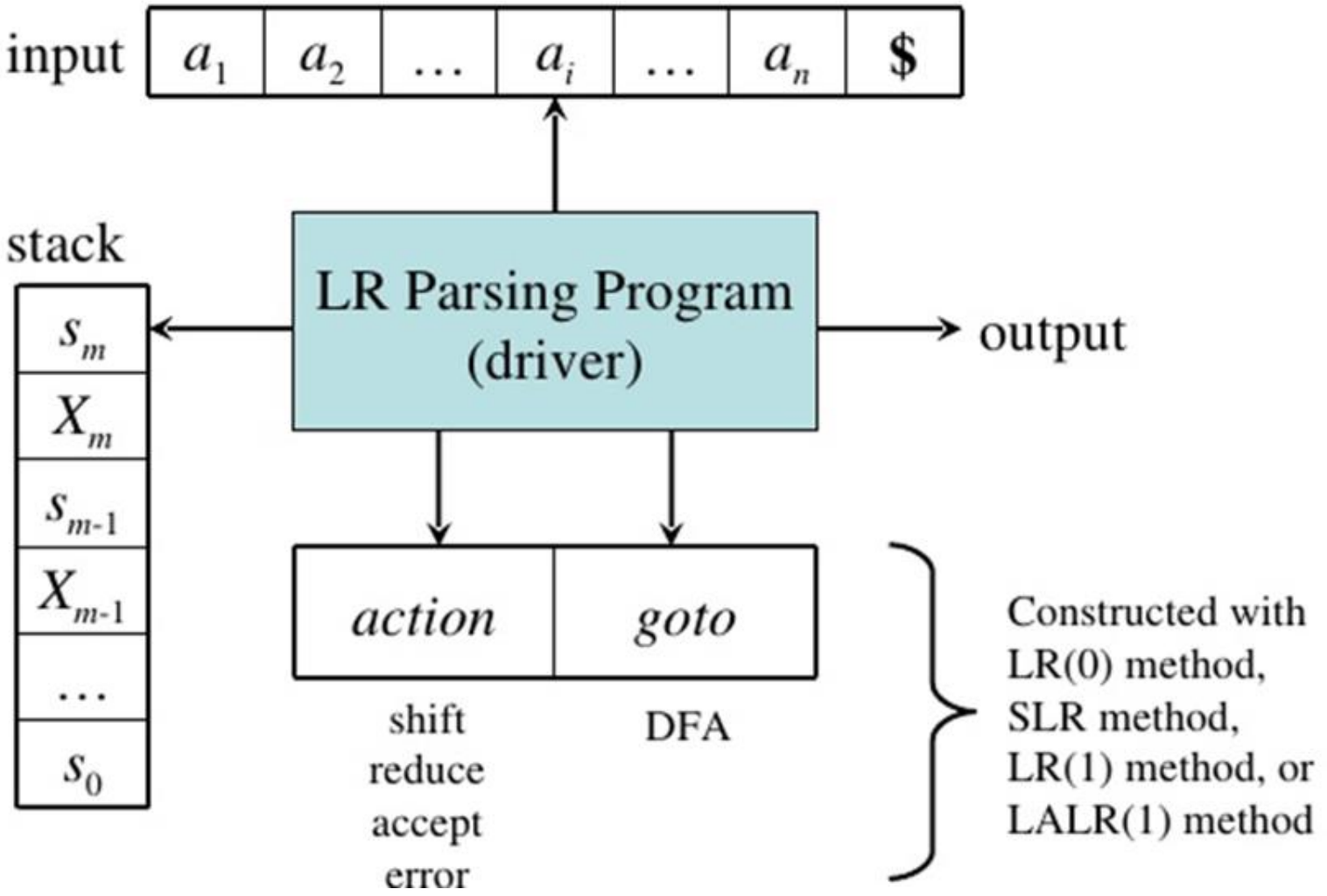
Bottom-Up Parsing

- A bottom-up parser works by maintaining a forest of **partially completed** subtrees of the parse tree, which it joins together whenever it recognizes the symbols on the right-hand side of some production used in the right-most derivation of the input string.
- It creates a new internal node and makes the roots of the joined-together trees the children of that node.

LR parsers are almost always Table-Driven

- **Like** a table-driven LL parser, an LR parser uses a big loop in which it repeatedly inspects a two-dimensional table to find out what action to take
- **Unlike** the LL parser, however, the LR driver has non-trivial state (like a DFA), and the table is indexed by current input token and current state
- The stack contains a record of what has been seen SO FAR (NOT what is expected)

Model of an LR Parser



1. E --> E + T
2. E --> T
3. T --> T * F
4. T --> F
5. F --> (E)
6. F --> id

Pre-Compiled Table

1. E --> E + T
2. E --> T
3. T --> T * F
4. T --> F
5. F --> (E)
6. F --> id

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

LR Parsing II

LR Parser Driver and Parsing Example

SECTION 11

Parser Driver - Actions

The driver reads the input and consults the table. The table has four different kinds of entries called actions:

- **Shift:** Shift is indicated by the "S#" entries in the table where **# is a new state**. When we come to this entry in the table, we shift the current input symbol followed by the indicated new state onto the stack.
- **Reduce:** Reduce is indicated by "R#" **where # is the number of a production**. The top of the stack contains the right-hand side of a production, the handle. Reduce by the indicated production, consult the GOTO part of the table to see the next state, and push the left-hand side of the production onto the stack followed by the new state.
- **Accept:** Accept is indicated by the "Accept" entry in the table. When we come to this entry in the table, we accept the input string. Parsing is complete.
- **Error:** The blank entries in the table indicate a syntax error. No action is defined.

Parser Driver - Algorithm

Algorithm LR Parser Driver
Initialize Stack to state 0
Append $ to end of input
While Action != Accept And Action != Error Do
Let Stack = $s_0\, x_1\, s_1 \ldots x_m\, s_m$ and remaining Input=$a_i\, a_{i+1} \ldots$ $
{**S**'s are **state numbers**; **x**'s are sequences of terminals and non-terminals}
Case Table [s_m, a_i] is // Combination of State and Input for CFSM
S#: Action : = Shift
R#: Action : = Reduce
Accept : Action : = Accept
Blank : Action : = Error
EndWhile

LR Parsing Example

Consider the following grammar, a subset of the assignment statement grammar:

1. E --> E + T
2. E --> T
3. T --> T * F
4. T --> F
5. F --> (E)
6. F --> id

Note: This is a set of LR left recursive grammar.

And, consider the table to be built by magic for the moment:

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

S: Shift, R:Reduce, B:Shift and Reduce

Example

We will use grammar and table to parse the input string, a * (b + c), and $; to understand the meaning of the entries in the table:

1. E --> E + T
2. E --> T
3. T --> T * F
4. T --> F
5. F --> (E)
6. F --> id

Step (1):
Parsing begins with state 0 on the stack and the input terminated by "$":

Stack	Input	Action
0	a * (b + c) $	

Consulting the table, across from **state 0** and under input **id**, is the action S5 which means to Shift (push) the input onto the stack and go to state 5.

Step (2):

Stack	Input	Action
(1) 0	a * (b + c) $	S5
(2) 0 id 5	* (b + c) $	

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

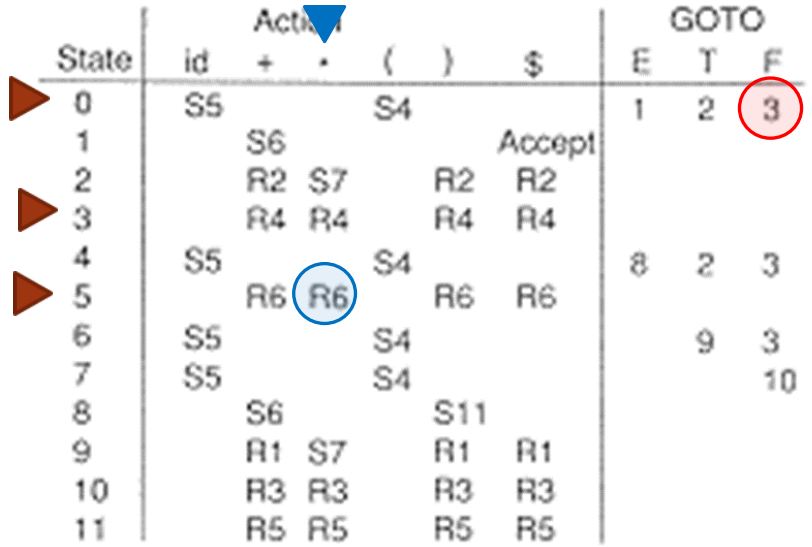
Example

The next step in the parse consults Table [5, *]. The entry across from state 5 and under input *, is the action R6 which means the right-hand side of production 6 is the handle to be reduced. We remove everything on the stack that includes the handle. Here, this is id 5. The stack now contains only 0. Since the left-hand side of production 6 will be pushed on the stack, consult the GOTO part of the table across from state 0 (the exposed top state) and under F (the left-hand side of the production). The entry there is 3. Thus, we push F 3 onto the stack.

1. E --> E + T
2. E --> T
3. T --> T * F
4. T --> F
5. F --> (E)
6. F --> id

Step (3)

Stack	Input	Action
(2) 0 id 5	* (b + c) $	R6
(2) 0		// id 5 popped out after reduction.
(3) 0 F 3	* (b + c) $	



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

Example

Now the top of the stack is state 3 and the current input is *. Consulting the entry at Table [3, *], the action indicated is R4, reduced using production 4. Thus, the right-hand side of production 4 is the handle on the stack. The algorithm says to pop the stack up to and including the F. That exposes state 0. Across from 0 and under the right-hand side of production 4 (the T) is state 2. We shift the T onto the stack followed by state 2.

1. E --> E + T
2. E --> T
3. T --> T * F
4. T --> F
5. F --> (E)
6. F --> id

Stack	Input	Action
(3) 0 F 3	* (b + c) $	R4
(3) 0 T 2	* (b + c) $	R4

Continuing,

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

Example

Continuing,

	Stack	Input	Action
(3)	0 T 3	* (b + c) $	R4
(4)	0 T 2	* (b + c) $	S7
	0 T 2 * 7	(b + c) $	S4
	0 T 2 * 7 (4	(b + c) $	S5
	0 T 2 * 7 (4 id 5	+ c) $	R6
	0 T 2 * 7 (4 F 3	+ c) $	R4
	0 T 2 * 7 (4 T 2	+ c) $	R2
	0 T 2 * 7 (4 E 8	+ c) $	S6
	0 T 2 * 7 (4 E 8 + 6	c $	S5
	0 T 2 * 7 (4 E 8 + 6 id 5	) $	R6
	0 T 2 * 7 (4 E 8 + 6 F 3	) $	R4
	0 T 2 * 7 (4 E 8 + 6 T 9	) $	R1
	0 T 2 * 7 (4 E 8)	) $	S11
	0 T 2 * 7 (4 E 8 11	$	R5
	0 T 2 * 7 F 10	$	R3
	0 T 2	$	R2
(19)	0 E 1	$	Accept

Step (19):

The parse is in state 1 looking at "$". The table indicates that this is the accept state. Parsing has thus completed successfully. By following the reduce actions in reverse, starting with R2, the last reduce action, and continuing until R6 the first reduce action, a parse tree can be created. Exercise 1 asks the reader to draw this parse tree.

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

LR Parsing III

The Calculator Language Example

SECTION 12

Canonical Derivations

Stack Contents (Roots of Partial Trees):

ε
id (A)
id (A) ,
id (A) , id (B)
id (A) , id (B) ,
id (A) , id (B) , id (C)
id (A) , id (B) , id (C) ;
id (A) , id (B) , id_list_tail
id (A) , id_list_tail
id (A) id_list_tail
id_list

Remaining Input:

A, B, C;
, B, C;
B, C;
, C;
C;
;

id_list ⟶ id *id_list_tail*
id_list_tail ⟶ , id *id_list_tail*
id_list_tail ⟶ ;

id(A)

id(A) ,

id(A) , id(B)

id(A) , id(B) ,

id(A) , id(B) , id(C)

id(A) , id(B) , id(C) ;

id(A) , id(B) , id(C) *id_list_tail*

;

id(A) , id(B) *id_list_tail*

, id(C) *id_list_tail*

;

id(A) *id_list_tail*

, id(B) *id_list_tail*

, id(C) *id_list_tail*

;

id_list

id(A) *id_list_tail*

, id(B) *id_list_tail*

, id(C) *id_list_tail*

;

Bottom-Up Grammar for the Calculator Language

- In our **id_list** example, no handles were found until the entire input had been shifted onto the stack. In general this will not be the case. We can obtain a more realistic example of an LR calculator language is shown in Figure 2.25.
- This version in Figure 2.25 is preferable (for bottom up) for two reasons:
  - First, it uses a left-recursive production for **stmt_list**. Left recursion allows the parser to collapse long statement lists as it goes along, rather than waiting until the entire list is on the stack and then collapsing it from the end.
  - Second, it uses left-recursive productions for **expr** and **term**. These productions capture left associativity while still keeping an operator and its operands together in the same right-hand side, something we were unable to do in a top-down grammar.

1. *program* ⟶ *stmt_list* `$$`
2. *stmt_list* ⟶ *stmt_list stmt*
3. *stmt_list* ⟶ *stmt*
4. *stmt* ⟶ `id :=` *expr*
5. *stmt* ⟶ `read id`
6. *stmt* ⟶ `write` *expr*
7. *expr* ⟶ *term*
8. *expr* ⟶ *expr add_op term*
9. *term* ⟶ *factor*
10. *term* ⟶ *term mult_op factor*
11. *factor* ⟶ `(` *expr* `)`
12. *factor* ⟶ `id`
13. *factor* ⟶ `number`
14. *add_op* ⟶ `+`
15. *add_op* ⟶ `-`
16. *mult_op* ⟶ `*`
17. *mult_op* ⟶ `/`

Figure 2.25 LR(1) grammar for the calculator language. Productions have been numbered for reference in future figures.

Left Recursive

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

Figure 2.16

Model a Parser with LR Items

Bottom-Up Parse of the "sum and average" Program

EXAMPLE 2.24

```
read A
read B
sum := A + B
write sum
write sum / 2
```

The key to success will be to figure out when we have reached the end of a right-hand side – that is, when we have a handle at the **top** of the parse stack.

The trick is to keep track of **the set of productions** we might be "in the middle of " at any particular time, together with an indication of where in those productions we might be.

Design of the Action Table and the Goto Tables

When we begin execution, the parse stack is empty and we are at the beginning of the production for **program**.

LR items

A Production Rule with ■ is Called an Item.

■ represents the location on top of the stack.

Set of Items		Basis
(State 0)	program -> ■ stmt_list	$$
Closure	stmt_list -> ■ stmt_list	stmt
	stmt_list -> ■ stmt	
	stmt -> ■ id := expr	
	stmt -> ■ read id	
	stmt -> ■ read id	

- The original item:
  program -> ■ stmt_list $$
  is called the basis of the list.
- The additional items are its **closure**.
- The list represents the initial state of the parser.
- As we shift and reduce, the **set of items** will change.
- If we reach a state in which some item has the ■ at the end of the right-hand side, we can reduce by that production.
- Otherwise, as in the current situation, we must shift.

Note: Since the ■ in this term is in from of a non-terminal – namely **stmt_list** – we may be about to see the yield of that nonterminal coming up on the input. This possibility implies that we may be at the beginning of some production with **stmt_list** on the left hand side. **stmt** is a nonterminal, we ma also be at the beginning of any production whose left hand side is stmt.

Note that if we need to shift, but the incoming token cannot follow the in any item of the current state, the a syntax error has occurred. We will consider error recovery in more details in Section C-2.3.5. **[Extra] (report error or backtracking)**

The operations of Shift(s), Reduce(r), and Shift and Reduce(b)

Input Token

```
read A
read B
sum := A + B
write sum
write sum / 2
```

stmt -> ■ read id

(state 1)

Shift

Input Token

```
read A
read B
sum := A + B
write sum
write sum / 2
```

stmt -> read ■ id

Shift

End of stmt

```
read A
read B
sum := A + B
write sum
write sum / 2
```

stmt -> read id ■

Reduce

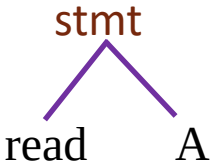
stmt_list -> ■ stmt

stmt_list -> stmt ■

(state 0')

Shift

parse Tree



Our New State

program -> stmt_list ■ $$
stmt_list -> stmt_list ■ stmt
stmt -> ■ id := expr
stmt -> ■ read id
stmt -> ■ read id

(state 2)

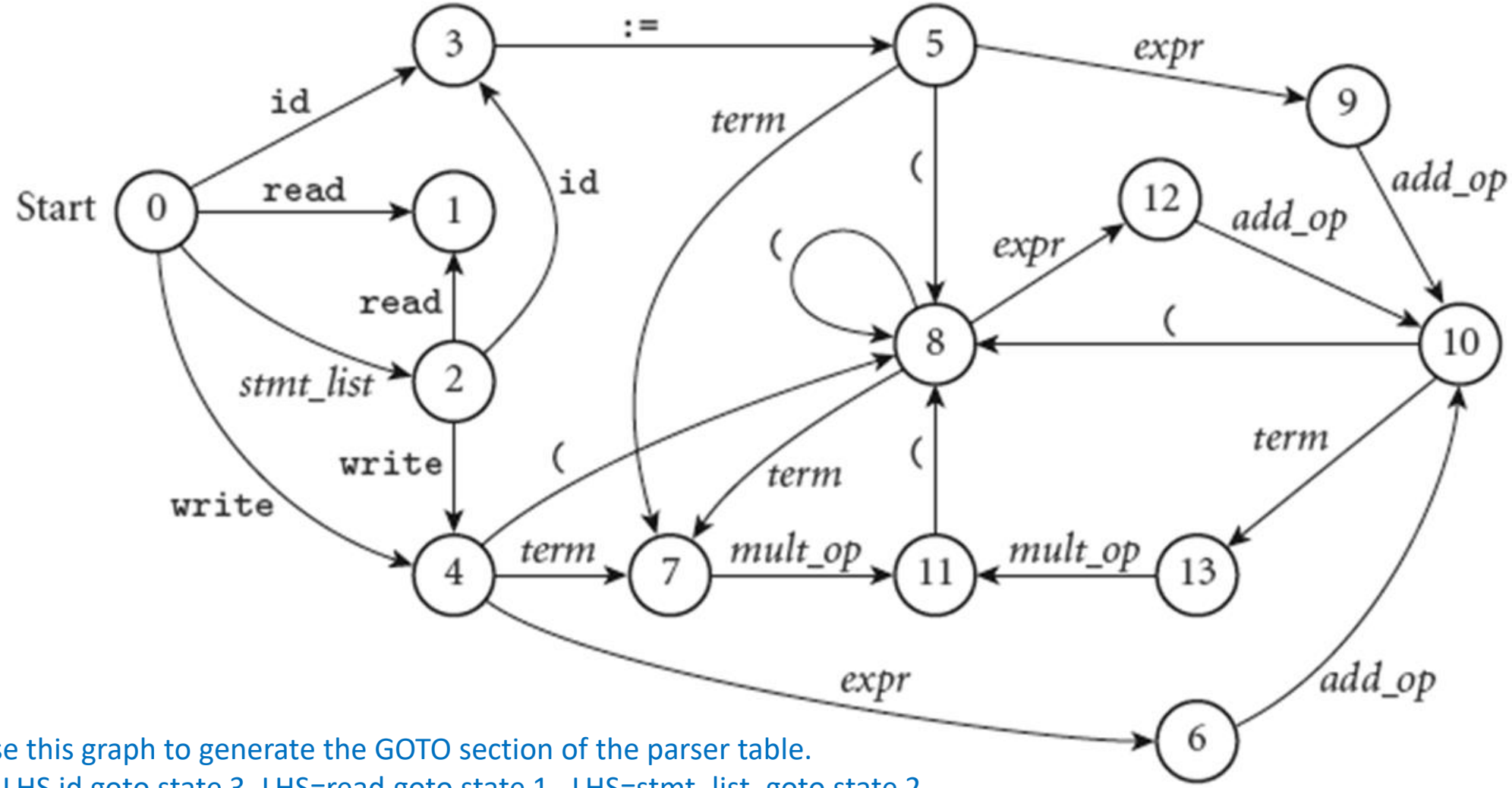
	State	Transitions
0.	*program* ⟶ • *stmt_list* **$$**	on *stmt_list* shift and goto 2
	stmt_list ⟶ • *stmt_list stmt*	
	stmt_list ⟶ • *stmt*	on *stmt* shift and reduce (pop 1 state, push *stmt_list* on input)
	stmt ⟶ • **id** := *expr*	on **id** shift and goto 3
	stmt ⟶ • **read id**	on **read** shift and goto 1
	stmt ⟶ • **write** *expr*	on **write** shift and goto 4
1.	*stmt* ⟶ **read** • **id**	on **id** shift and reduce (pop 2 states, push *stmt* on input)
2.	*program* ⟶ *stmt_list* • **$$**	on **$$** shift and reduce (pop 2 states, push *program* on input)
	stmt_list ⟶ *stmt_list* • *stmt*	on *stmt* shift and reduce (pop 2 states, push *stmt_list* on input)
	stmt ⟶ • **id** := *expr*	on **id** shift and goto 3
	stmt ⟶ • **read id**	on **read** shift and goto 1
	stmt ⟶ • **write** *expr*	on **write** shift and goto 4
3.	*stmt* ⟶ **id** • := *expr*	on := shift and goto 5
4.	*stmt* ⟶ **write** • *expr*	on *expr* shift and goto 6
	expr ⟶ • *term*	on *term* shift and goto 7
	expr ⟶ • *expr add_op term*	
	term ⟶ • *factor*	on *factor* shift and reduce (pop 1 state, push *term* on input)
	term ⟶ • *term mult_op factor*	
	factor ⟶ • (*expr*)	on (shift and goto 8
	factor ⟶ • **id**	on **id** shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ • **number**	on **number** shift and reduce (pop 1 state, push *factor* on input)
5.	*stmt* ⟶ **id** := • *expr*	on *expr* shift and goto 9
	expr ⟶ • *term*	on *term* shift and goto 7
	expr ⟶ • *expr add_op term*	
	term ⟶ • *factor*	on *factor* shift and reduce (pop 1 state, push *term* on input)
	term ⟶ • *term mult_op factor*	
	factor ⟶ • (*expr*)	on (shift and goto 8
	factor ⟶ • **id**	on **id** shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ • **number**	on **number** shift and reduce (pop 1 state, push *factor* on input)
6.	*stmt* ⟶ **write** *expr* •	on FOLLOW(*stmt*) = {**id**, **read**, **write**, **$$**} reduce
	expr ⟶ *expr* • *add_op term*	(pop 2 states, push *stmt* on input)
		on *add_op* shift and goto 10
	add_op ⟶ • +	on + shift and reduce (pop 1 state, push *add_op* on input)
	add_op ⟶ • -	on - shift and reduce (pop 1 state, push *add_op* on input)

Figure 2.26 CFSM for the calculator grammar (Figure 2.25). Basis and closure items in each state are separated by a horizontal rule. Trivial reduce-only states have been eliminated by use of "shift and reduce" transitions. *(continued)*

	State	Transitions
7.	*expr* ⟶ *term* •	on FOLLOW(*expr*) = {**id**, **read**, **write**, **$$**,), +, -} reduce
	term ⟶ *term* • *mult_op factor*	(pop 1 state, push *expr* on input)
		on *mult_op* shift and goto 11
	mult_op ⟶ • *	on * shift and reduce (pop 1 state, push *mult_op* on input)
	mult_op ⟶ • /	on / shift and reduce (pop 1 state, push *mult_op* on input)
8.	*factor* ⟶ (• *expr*)	on *expr* shift and goto 12
	expr ⟶ • *term*	on *term* shift and goto 7
	expr ⟶ • *expr add_op term*	
	term ⟶ • *factor*	on *factor* shift and reduce (pop 1 state, push *term* on input)
	term ⟶ • *term mult_op factor*	
	factor ⟶ • (*expr*)	on (shift and goto 8
	factor ⟶ • **id**	on **id** shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ • **number**	on **number** shift and reduce (pop 1 state, push *factor* on input)
9.	*stmt* ⟶ **id** := *expr* •	on FOLLOW(*stmt*) = {**id**, **read**, **write**, **$$**} reduce
	expr ⟶ *expr* • *add_op term*	(pop 3 states, push *stmt* on input)
		on *add_op* shift and goto 10
	add_op ⟶ • +	on + shift and reduce (pop 1 state, push *add_op* on input)
	add_op ⟶ • -	on - shift and reduce (pop 1 state, push *add_op* on input)
10.	*expr* ⟶ *expr add_op* • *term*	on *term* shift and goto 13
	term ⟶ • *factor*	on *factor* shift and reduce (pop 1 state, push *term* on input)
	term ⟶ • *term mult_op factor*	
	factor ⟶ • (*expr*)	on (shift and goto 8
	factor ⟶ • **id**	on **id** shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ • **number**	on **number** shift and reduce (pop 1 state, push *factor* on input)
11.	*term* ⟶ *term mult_op* • *factor*	on *factor* shift and reduce (pop 3 states, push *term* on input)
	factor ⟶ • (*expr*)	on (shift and goto 8
	factor ⟶ • **id**	on **id** shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ • **number**	on **number** shift and reduce (pop 1 state, push *factor* on input)
12.	*factor* ⟶ (*expr* •)	on) shift and reduce (pop 3 states, push *factor* on input)
	expr ⟶ *expr* • *add_op term*	on *add_op* shift and goto 10
	add_op ⟶ • +	on + shift and reduce (pop 1 state, push *add_op* on input)
	add_op ⟶ • -	on - shift and reduce (pop 1 state, push *add_op* on input)
13.	*expr* ⟶ *expr add_op term* •	on FOLLOW(*expr*) = {**id**, **read**, **write**, **$$**,), +, -} reduce
	term ⟶ *term* • *mult_op factor*	(pop 3 states, push *expr* on input)
		on *mult_op* shift and goto 11
	mult_op ⟶ • *	on * shift and reduce (pop 1 state, push *mult_op* on input)
	mult_op ⟶ • /	on / shift and reduce (pop 1 state, push *mult_op* on input)

Figure 2.26 *(continued)*

Note:
This table include the ACTION part and GOTO part for the LR Parser Table



Note: Use this graph to generate the GOTO section of the parser table.
State 0, LHS id goto state 3, LHS=read goto state 1, LHS=stmt_list, goto state 2

Figure 2.27 Pictorial representation of the CFSM of Figure 2.26. Reduce actions are not shown.

Top-of-stack state	Current input symbol																		
	sl	*s*	*e*	*t*	*f*	*ao*	*mo*	`id`	`lit`	`r`	`w`	`:=`	`(`	`)`	`+`	`-`	`*`	`/`	`$$`
0	s2	b3	–	–	–	–	–	s3	–	s1	s4	–	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–	b5	–	–	–	–	–	–	–	–	–	–	–
2	–	b2	–	–	–	–	–	s3	–	s1	s4	–	–	–	–	–	–	–	b1
3	–	–	–	–	–	–	–	–	–	–	–	s5	–	–	–	–	–	–	–
4	–	–	s6	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
5	–	–	s9	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
6	–	–	–	–	–	s10	–	r6	–	r6	r6	–	–	–	b14	b15	–	–	r6
7	–	–	–	–	–	–	s11	r7	–	r7	r7	–	–	r7	r7	r7	b16	b17	r7
8	–	–	s12	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
9	–	–	–	–	–	s10	–	r4	–	r4	r4	–	–	–	b14	b15	–	–	r4
10	–	–	–	s13	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
11	–	–	–	–	b10	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
12	–	–	–	–	–	s10	–	–	–	–	–	–	–	b11	b14	b15	–	–	–
13	–	–	–	–	–	–	s11	r8	–	r8	r8	–	–	r8	r8	r8	b16	b17	r8

Figure 2.28 SLR(1) parse table for the calculator language. Table entries indicate whether to shift (s), reduce (r), or shift and then reduce (b). The accompanying number is the new state when shifting, or the production that has been recognized when (shifting and) reducing. Production numbers are given in Figure 2.25. Symbol names have been abbreviated for the sake of formatting. A dash indicates an error. An auxiliary table, not shown here, gives the left-hand-side symbol and right-hand-side length for each production.

```
state = 1 . . number_of_states
symbol = 1 . . number_of_symbols
production = 1 . . number_of_productions
action_rec = record
        action : (shift, reduce, shift_reduce, error)
        new_state : state
        prod : production

parse_tab : array [symbol, state] of action_rec
prod_tab : array [production] of record
        lhs : symbol
        rhs_len : integer
-- these two tables are created by a parser generator tool

parse_stack : stack of record
        sym : symbol
        st : state

parse_stack.push(⟨null, start_state⟩)
cur_sym : symbol := scan()                      -- get new token from scanner
loop
        cur_state : state := parse_stack.top().st   -- peek at state at top of stack
        if cur_state = start_state and cur_sym = start_symbol
                return                          -- success!
        ar : action_rec := parse_tab[cur_state, cur_sym]
        case ar.action
                shift:
                        parse_stack.push(⟨cur_sym, ar.new_state⟩)
                        cur_sym := scan()       -- get new token from scanner
                reduce:
                        cur_sym := prod_tab[ar.prod].lhs
                        parse_stack.pop(prod_tab[ar.prod].rhs_len)
                shift_reduce:
                        cur_sym := prod_tab[ar.prod].lhs
                        parse_stack.pop(prod_tab[ar.prod].rhs_len−1)
                error:
                        parse_error
```

Figure 2.29: Driver for a table-driven SLR(1) parser.

Parse stack	Input stream	Comment
0	`read A read B ...`	
0 `read` *1*	`A read B ...`	shift `read`
0	*stmt* `read B ...`	shift `id(A)` & reduce by *stmt* ⟶ `read id`
0	*stmt_list* `read B ...`	shift *stmt* & reduce by *stmt_list* ⟶ *stmt*
0 *stmt_list* *2*	`read B sum ...`	shift *stmt_list*
0 *stmt_list* *2* `read` *1*	`B sum := ...`	shift `read`
0 *stmt_list* *2*	*stmt* `sum := ...`	shift `id(B)` & reduce by *stmt* ⟶ `read id`
0	*stmt_list* `sum := ...`	shift *stmt* & reduce by *stmt_list* ⟶ *stmt_list stmt*
0 *stmt_list* *2*	`sum := A ...`	shift *stmt_list*
0 *stmt_list* *2* `id` *3*	`:= A + ...`	shift `id(sum)`
0 *stmt_list* *2* `id` *3* `:=` *5*	`A + B ...`	shift `:=`
0 *stmt_list* *2* `id` *3* `:=` *5*	*factor* `+ B ...`	shift `id(A)` & reduce by *factor* ⟶ `id`
0 *stmt_list* *2* `id` *3* `:=` *5*	*term* `+ B ...`	shift *factor* & reduce by *term* ⟶ *factor*
0 *stmt_list* *2* `id` *3* `:=` *5* *term* *7*	`+ B write ...`	shift *term*
0 *stmt_list* *2* `id` *3* `:=` *5*	*expr* `+ B write ...`	reduce by *expr* ⟶ *term*
0 *stmt_list* *2* `id` *3* `:=` *5* *expr* *9*	`+ B write ...`	shift *expr*
0 *stmt_list* *2* `id` *3* `:=` *5* *expr* *9*	*add_op* `B write ...`	shift `+` & reduce by *add_op* ⟶ `+`
0 *stmt_list* *2* `id` *3* `:=` *5* *expr* *9* *add_op* *10*	`B write sum ...`	shift *add_op*
0 *stmt_list* *2* `id` *3* `:=` *5* *expr* *9* *add_op* *10*	*factor* `write sum ...`	shift `id(B)` & reduce by *factor* ⟶ `id`
0 *stmt_list* *2* `id` *3* `:=` *5* *expr* *9* *add_op* *10*	*term* `write sum ...`	shift *factor* & reduce by *term* ⟶ *factor*
0 *stmt_list* *2* `id` *3* `:=` *5* *expr* *9* *add_op* *10* *term* *13*	`write sum ...`	shift *term*
0 *stmt_list* *2* `id` *3* `:=` *5*	*expr* `write sum ...`	reduce by *expr* ⟶ *expr add_op term*
0 *stmt_list* *2* `id` *3* `:=` *5* *expr* *9*	`write sum ...`	shift *expr*
0 *stmt_list* *2*	*stmt* `write sum ...`	reduce by *stmt* ⟶ `id :=` *expr*
0	*stmt_list* `write sum ...`	shift *stmt* & reduce by *stmt_list* ⟶ *stmt*
0 *stmt_list* *2*	`write sum ...`	shift *stmt_list*
0 *stmt_list* *2* `write` *4*	`sum write sum ...`	shift `write`
0 *stmt_list* *2* `write` *4*	*factor* `write sum ...`	shift `id(sum)` & reduce by *factor* ⟶ `id`
0 *stmt_list* *2* `write` *4*	*term* `write sum ...`	shift *factor* & reduce by *term* ⟶ *factor*
0 *stmt_list* *2* `write` *4* *term* *7*	`write sum ...`	shift *term*
0 *stmt_list* *2* `write` *4*	*expr* `write sum ...`	reduce by *expr* ⟶ *term*
0 *stmt_list* *2* `write` *4* *expr* *6*	`write sum ...`	shift *expr*
0 *stmt_list* *2*	*stmt* `write sum ...`	reduce by *stmt* ⟶ `write` *expr*
0	*stmt_list* `write sum ...`	shift *stmt* & reduce by *stmt_list* ⟶ *stmt_list stmt*
0 *stmt_list* *2*	`write sum / ...`	shift *stmt_list*
0 *stmt_list* *2* `write` *4*	`sum / 2 ...`	shift `write`
0 *stmt_list* *2* `write` *4*	*factor* `/ 2 ...`	shift `id(sum)` & reduce by *factor* ⟶ `id`
0 *stmt_list* *2* `write` *4*	*term* `/ 2 ...`	shift *factor* & reduce by *term* ⟶ *factor*
0 *stmt_list* *2* `write` *4* *term* *7*	`/ 2 $$`	shift *term*
0 *stmt_list* *2* `write` *4* *term* *7*	*mult_op* `2 $$`	shift `/` & reduce by *mult_op* ⟶ `/`
0 *stmt_list* *2* `write` *4* *term* *7* *mult_op* *11*	`2 $$`	shift *mult_op*
0 *stmt_list* *2* `write` *4* *term* *7* *mult_op* *11*	*factor* `$$`	shift `number(2)` & reduce by *factor* ⟶ `number`
0 *stmt_list* *2* `write` *4*	*term* `$$`	shift *factor* & reduce by *term* ⟶ *term mult_op factor*
0 *stmt_list* *2* `write` *4* *term* *7*	`$$`	shift *term*
0 *stmt_list* *2* `write` *4*	*expr* `$$`	reduce by *expr* ⟶ *term*
0 *stmt_list* *2* `write` *4* *expr* *6*	`$$`	shift *expr*
0 *stmt_list* *2*	*stmt* `$$`	reduce by *stmt* ⟶ `write` *expr*
0	*stmt_list* `$$`	shift *stmt* & reduce by *stmt_list* ⟶ *stmt_list stmt*
0 *stmt_list* *2*	`$$`	shift *stmt_list*
0	*program*	shift `$$` & reduce by *program* ⟶ *stmt_list* `$$`
[done]		

Figure 2.30: SLR parsing is based on Shift, Reduce, and also Shift & Reduce (for optimization)

LR Parsing IV

Other Topics

SECTION 13

LR Parsing Variants

- **LR(0)** states were created: no lookahead was used to create them. We did, however, consider the next input symbol(one symbol lookahead) when creating the table. If no lookahead is used to create the table, then the parser would be called an LR(0) parser. [Not very Useful]
- **LR(1)** tables for typical programming languages are massive.
- **SLR(1)** parsers recognize many, but not all, of the constructs in typical programming languages.
- **LALR(1):** There is another type of parser which recognizes almost as many constructs as an LR(1) parser. This is called a LALR(1) parser and is constructed by first constructing the LR(1) items and states and them merging many of them. Whenever two states are the same except for the lookahead symbol, they are merged. The first LA stands for Lookahead token is added to the item.

Note:

- It is important to note that the same driver is used to parse.
- It is the table generation that is different.

LR Parser Family

- The grammars are different. The have different Finite Automata to be mapped to. The parser tables are different.
- The simpler members of the LR family of parsers – LR(0), SLR(1), and LALR(1) all use the same automaton, called the Characteristic Finite State Machine (CFSM).
- Full LR parsers use a machine with (for most grammars) a much larger number of states. The difference between the algorithms lie in how they deal with states that contain a shift-reduce conflict.

Bottom Up Parsing Tables

Note: this has been demonstrated by a SLR parser example in a previous lecture.

- Like a table-driven LL(1) parser, an SLR(1), LALR(1) or LR(1) parser executes a loop in which it repeatedly inspects a two-dimensional table to find out what action to take.
- Instead of using the current input token and top-of-stack non-terminal to index into the table, an LR-family parser uses the current input token and the current parser state. **[ACTION section]**
- "Shift" table entries indicate the state that should be pushed.
- "Reduce" table entries indicate the number of states that should be popped and the non-terminal that should be pushed back onto the input stream, to be shifted by the state uncovered by the pops.
- There is always one popped state for every symbol on the right-hand side of the reducing production.
- The state to be pushed next can be found by indexing into the table using the uncovered state and the newly recognized non-terminal. **[GOTO section]**

Handling Epsilon Productions

The careful reader may have noticed that the grammar of Figure 2.25, in addition to using left-recursive rules for **stmt_list**, **expr**, and **term**, differs from the grammar of Figure 2.16 in one other way: it defines a **stmt_list** to be a sequence of one or more **stmt**s, rather than zero or more. (This means, of course, that it defines a different language.)

To capture the same language as Figure 2.16, production 3 in Figure 2.5,

$$stmt\_list \longrightarrow stmt$$

would need to be replaced with

$$stmt\_list \longrightarrow \epsilon$$

Note that it does in general make sense to have an empty statement list. In the calculator language it simply permits an empty program, which is admittedly silly. In real languages, however, it allows the body of a structured statement to be empty, which can be very useful.

CFSM with Epsilon Productions

- If we look at the CFSM for the calculator language, we discover that State 0 is the only state that needs to be changed in order to allow empty statement lists.

The item

$$stmt\_list \longrightarrow \bullet\ stmt$$

becomes

$$stmt\_list \longrightarrow \bullet\ \epsilon$$

which is equivalent to

$$stmt\_list \longrightarrow \epsilon\ \bullet$$

or simply

$$stmt\_list \longrightarrow \bullet$$

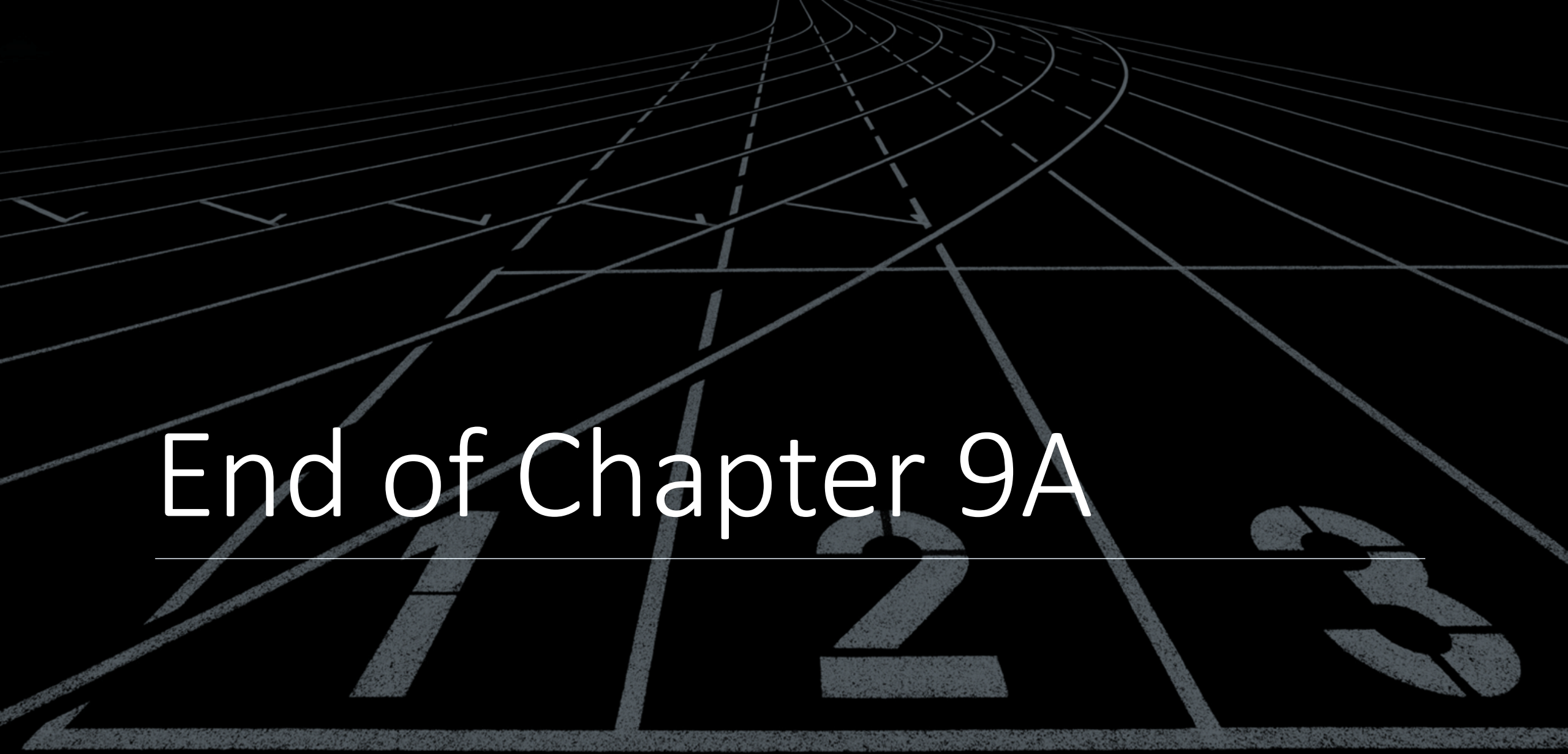
- The entire state is then

program ⟶ • *stmt_list* `$$`	on *stmt_list* shift and goto 2
stmt_list ⟶ • *stmt_list stmt*	
stmt_list ⟶ •	on `$$` reduce (pop 0 states, push *stmt_list* on input)
stmt ⟶ • `id :=` *expr*	on `id` shift and goto 3
stmt ⟶ • `read id`	on `read` shift and goto 1
stmt ⟶ • `write` *expr*	on `write` shift and goto 4

The look-ahead for item

$$stmt\_list \longrightarrow \bullet$$

is FOLLOW(stmt list), which is the end-marker, $$. Since $$ does not appear in the look-aheads for any other item in this state, our grammar is still SLR(1). It is worth noting that epsilon productions commonly prevent a grammar from being LR(0): if such a production shares a state with an item in which the dot precedes a terminal, we won't be able to tell whether to "recognize" ε without peeking ahead.



End of Chapter 9A