CS49K Programming Languages

Chapter 2 Programming Language Syntax - Sec. 2.3.1 – 2.5

LECTURE 4: PARSER DESIGN

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

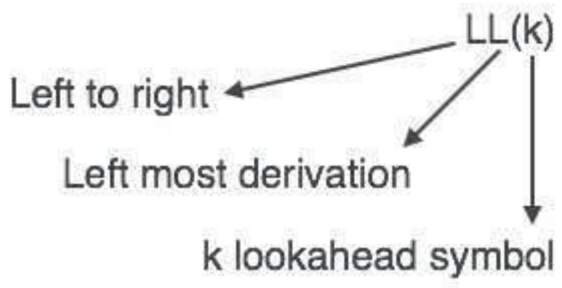
- LL Parsing
- LR Parsing
- Compiler-Compiler

LL Parsing I

Writing an LL(1) Grammar

SECTION 1

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:

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

Left Recursion

Elimination of Left Recursion

Left-Recursion

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

id, id, id, …., id;

Right-Recursion

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

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.

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

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

Left Factorization

$$
\begin{aligned}
stmt &\longrightarrow \boxed{\texttt{id}}\ \boxed{:=\ expr} \\
&\longrightarrow \boxed{\texttt{id}}\ \boxed{(\ argument\_list\ )}
\end{aligned}
$$

Common Left Factor

$$
\begin{aligned}
stmt &\longrightarrow \texttt{id}\ stmt\_list\_tail \\
stmt\_list\_tail &\longrightarrow \boxed{:=\ expr}\ |\ \boxed{(\ argument\_list\ )}
\end{aligned}
$$

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:

$$
\begin{aligned}
\textit{stmt} &\longrightarrow \texttt{if}\ \boxed{\textit{condition}}\ \boxed{\textit{then\_clause}}\ \boxed{\textit{else\_clause}} \mid \textit{other\_stmt} \\
\boxed{\textit{then\_clause}} &\longrightarrow \texttt{then}\ \textit{stmt} \\
\boxed{\textit{else\_clause}} &\longrightarrow \texttt{else}\ \textit{stmt} \mid \boxed{\epsilon}
\end{aligned}
$$

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

$$\begin{array}{lll} stmt & \longrightarrow & balanced\_stmt \mid unbalanced\_stmt \\ balanced\_stmt & \longrightarrow & \texttt{if}\ condition\ \texttt{then}\ balanced\_stmt\ \texttt{else}\ balanced\_stmt \\ & & \mid other\_stmt \\ unbalanced\_stmt & \longrightarrow & \texttt{if}\ condition\ \texttt{then}\ stmt \\ & & \mid \texttt{if}\ condition\ \texttt{then}\ balanced\_stmt\ \texttt{else}\ unbalanced\_stmt \end{array}$$

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 $\boxed{\text{if C2 then S1 else S2}}$

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:

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

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

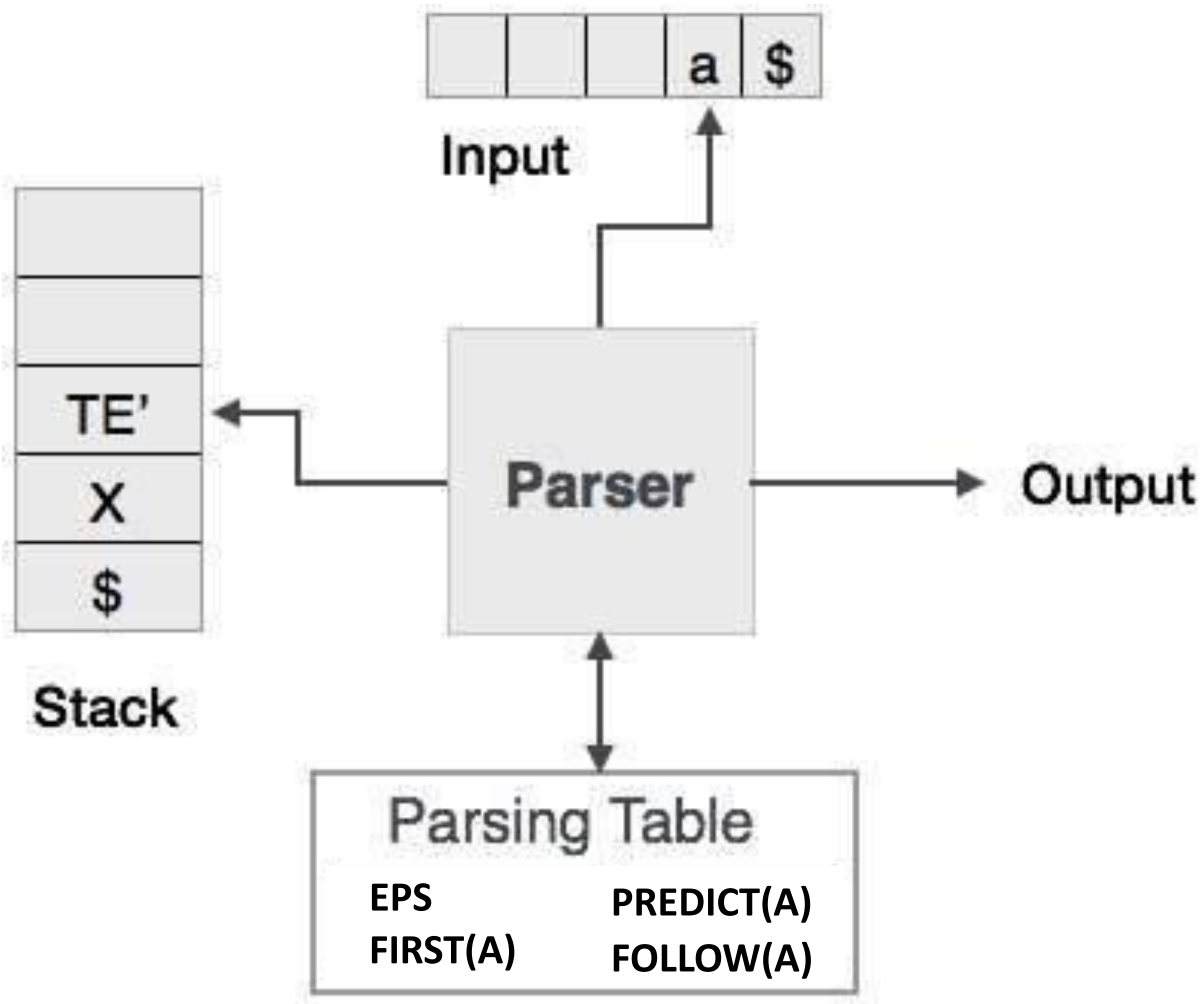
```
if A = B then ...
elsif A = C then ...
elsif A = D then ...
elsif A = E then ...
else ...
end
```

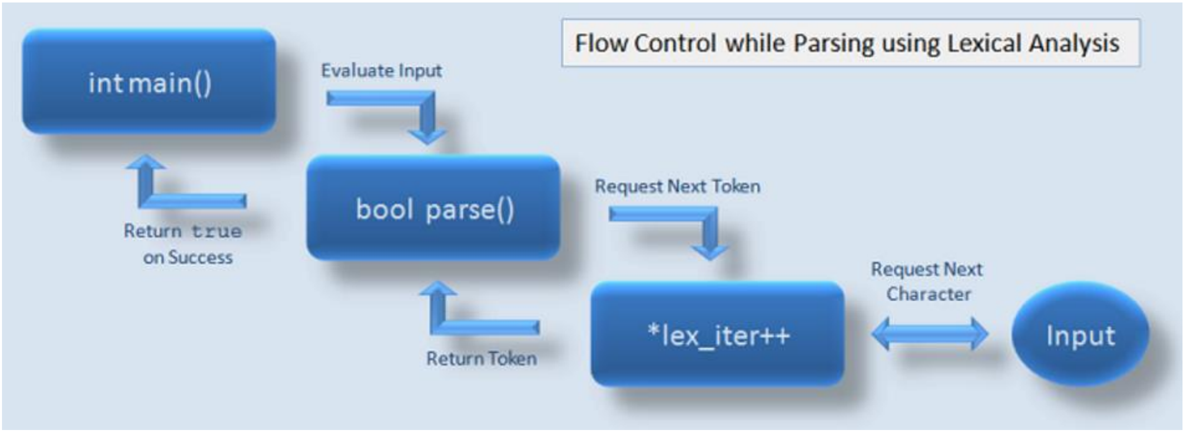
LL Parsing II

Overview of Table-Driven Top-down Parser

SECTION 2

Table-Driven Top-Down Parser





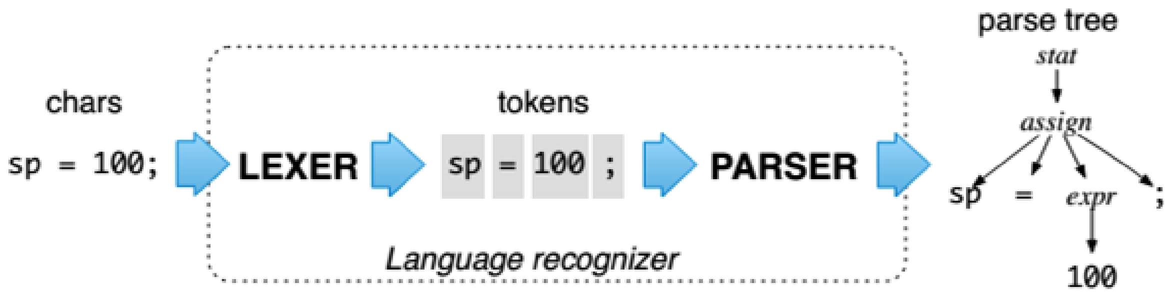


Table-driven LL parsing

- 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)
```

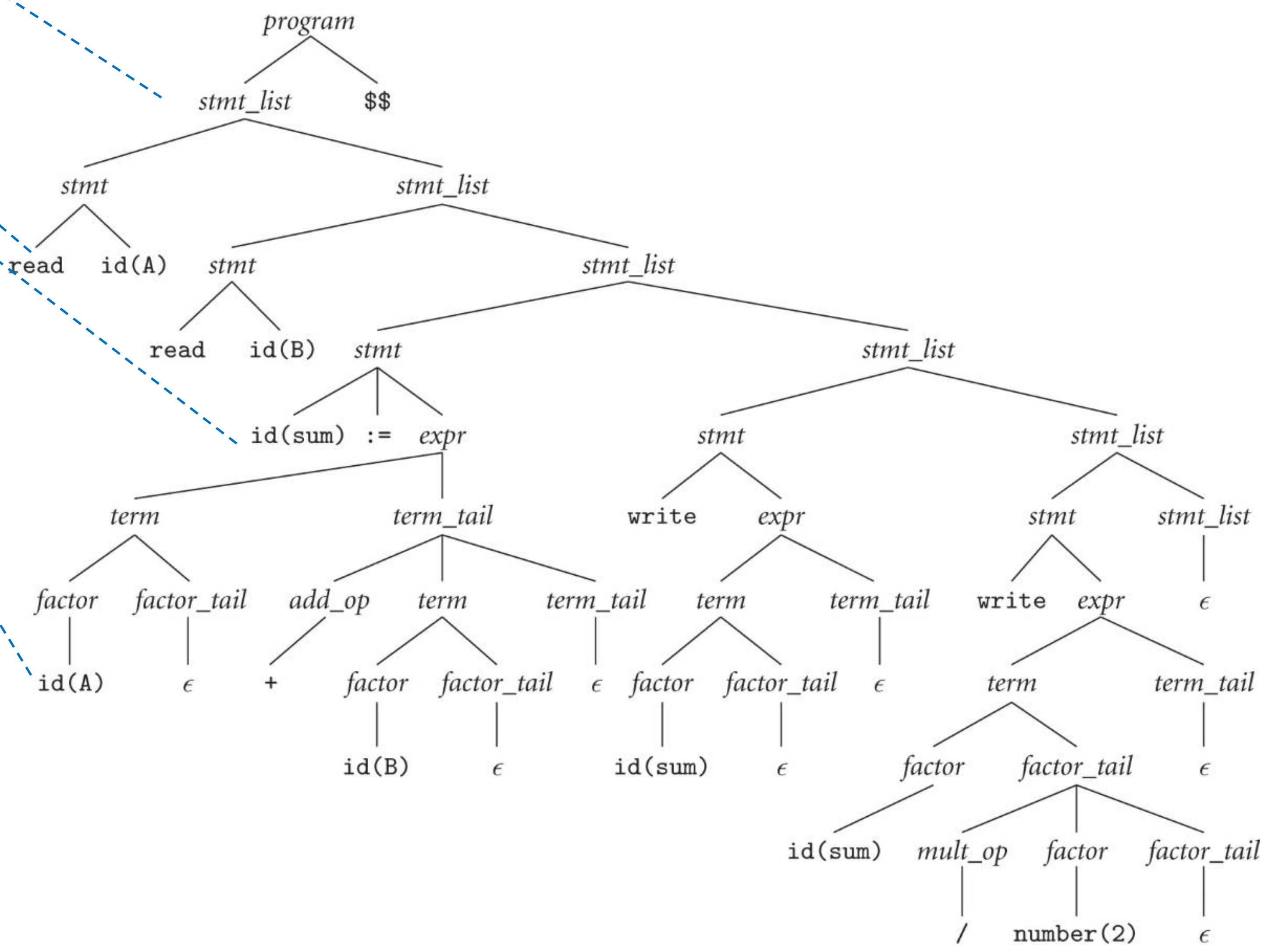
Top-of-stack nonterminal	Current input token											
	`id`	`number`	`read`	`write`	`:=`	`(`	`)`	`+`	`-`	`*`	`/`	`$$`
program	1	–	1	1	–	–	–	–	–	–	–	1
stmt_list	2	–	2	2	–	–	–	–	–	–	–	3
stmt	4	–	5	6	–	–	–	–	–	–	–	–
expr	7	7	–	–	–	7	–	–	–	–	–	–
term_tail	9	–	9	9	–	–	9	8	8	–	–	9
term	10	10	–	–	–	10	–	–	–	–	–	–
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 3

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}
    ∪ (if α =>* ε THEN {ε} ELSE NULL)
  - FOLLOW(A) == {a : S →$^+$ α A a β}
    ∪ (if S →* α A THEN {ε} ELSE NULL)
  - Predict (A → X_1 … X_m) == (FIRST (X_1 … X_m) - {ε}) ∪
    (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 4

Example: Calculator Language

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

$$
\begin{aligned}
S &\rightarrow exp\ \mathtt{STOP} \\[1ex]
exp &\rightarrow term\ exptail \\
exptail &\rightarrow \epsilon \mid \mathtt{OPA}\ term\ exptail \\[1ex]
term &\rightarrow sfactor\ termtail \\
termtail &\rightarrow \epsilon \mid \mathtt{OPM}\ factor\ termtail \\[1ex]
sfactor &\rightarrow \mathtt{OPA}\ factor \mid factor \\
factor &\rightarrow \mathtt{NUM} \mid \mathtt{LP}\ exp\ \mathtt{RP}
\end{aligned}
$$

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	

- Apply Rule 1 for PREDICT in four places

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.

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		$\boxed{\$}$
exp		STOP,RP
exptail	OPA	
term		
termtail	OPM	
sfactor	OPA	
factor	NUM, LP	

S $ default

S -> exp $\boxed{\text{STOP}}$
exp -> term exptail
exptail -> ε | OPA term exptail
term -> sfactor termtail
termtail -> ε | OPM factor termtail
sfactor -> OPA factor | factor
factor -> NUM | LP exp $\boxed{\text{RP}}$

- 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

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)

- Apply Rule 2 for FOLLOW to *term*

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 -> $\boxed{\varepsilon}$ | 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

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

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

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

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).

- Apply Rule 2 for PREDICT from the bottom up

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 ∅ )
```

$program \longrightarrow stmt\_list\ \texttt{\$\$}$	$\texttt{\$\$} \in \mathrm{FOLLOW}(stmt\_list)$
$stmt\_list \longrightarrow stmt\ stmt\_list$	
$stmt\_list \longrightarrow \epsilon$	$\mathrm{EPS}(stmt\_list) = \mathrm{true}$
$stmt \longrightarrow \texttt{id := } expr$	$\texttt{id} \in \mathrm{FIRST}(stmt)$
$stmt \longrightarrow \texttt{read id}$	$\texttt{read} \in \mathrm{FIRST}(stmt)$
$stmt \longrightarrow \texttt{write}\ expr$	$\texttt{write} \in \mathrm{FIRST}(stmt)$
$expr \longrightarrow term\ term\_tail$	
$term\_tail \longrightarrow add\_op\ term\ term\_tail$	
$term\_tail \longrightarrow \epsilon$	$\mathrm{EPS}(term\_tail) = \mathrm{true}$
$term \longrightarrow factor\ factor\_tail$	
$factor\_tail \longrightarrow mult\_op\ factor\ factor\_tail$	
$factor\_tail \longrightarrow \epsilon$	$\mathrm{EPS}(factor\_tail) = \mathrm{true}$
$factor \longrightarrow \texttt{(}\ expr\ \texttt{)}$	$\texttt{(} \in \mathrm{FIRST}(factor)$ and $\texttt{)} \in \mathrm{FOLLOW}(expr)$
$factor \longrightarrow \texttt{id}$	$\texttt{id} \in \mathrm{FIRST}(factor)$
$factor \longrightarrow \texttt{number}$	$\texttt{number} \in \mathrm{FIRST}(factor)$
$add\_op \longrightarrow \texttt{+}$	$\texttt{+} \in \mathrm{FIRST}(add\_op)$
$add\_op \longrightarrow \texttt{-}$	$\texttt{-} \in \mathrm{FIRST}(add\_op)$
$mult\_op \longrightarrow \texttt{*}$	$\texttt{*} \in \mathrm{FIRST}(mult\_op)$
$mult\_op \longrightarrow \texttt{/}$	$\texttt{/} \in \mathrm{FIRST}(mult\_op)$

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

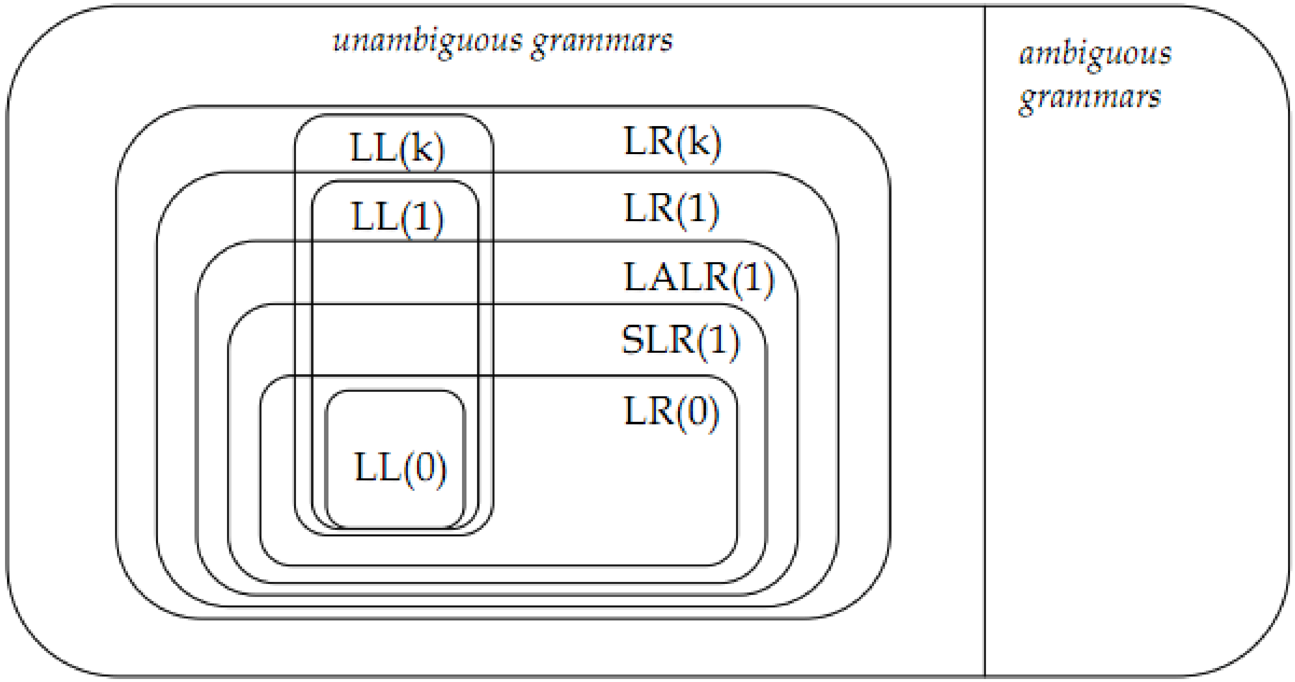
SECTION 5

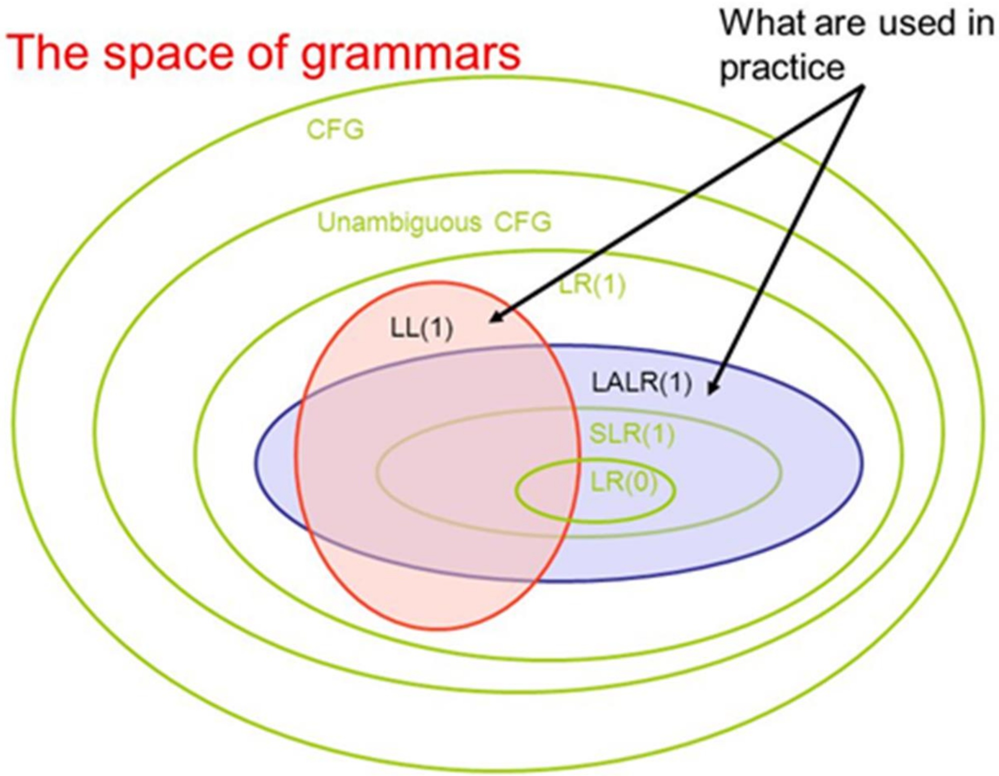
LR-Family Parsing

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

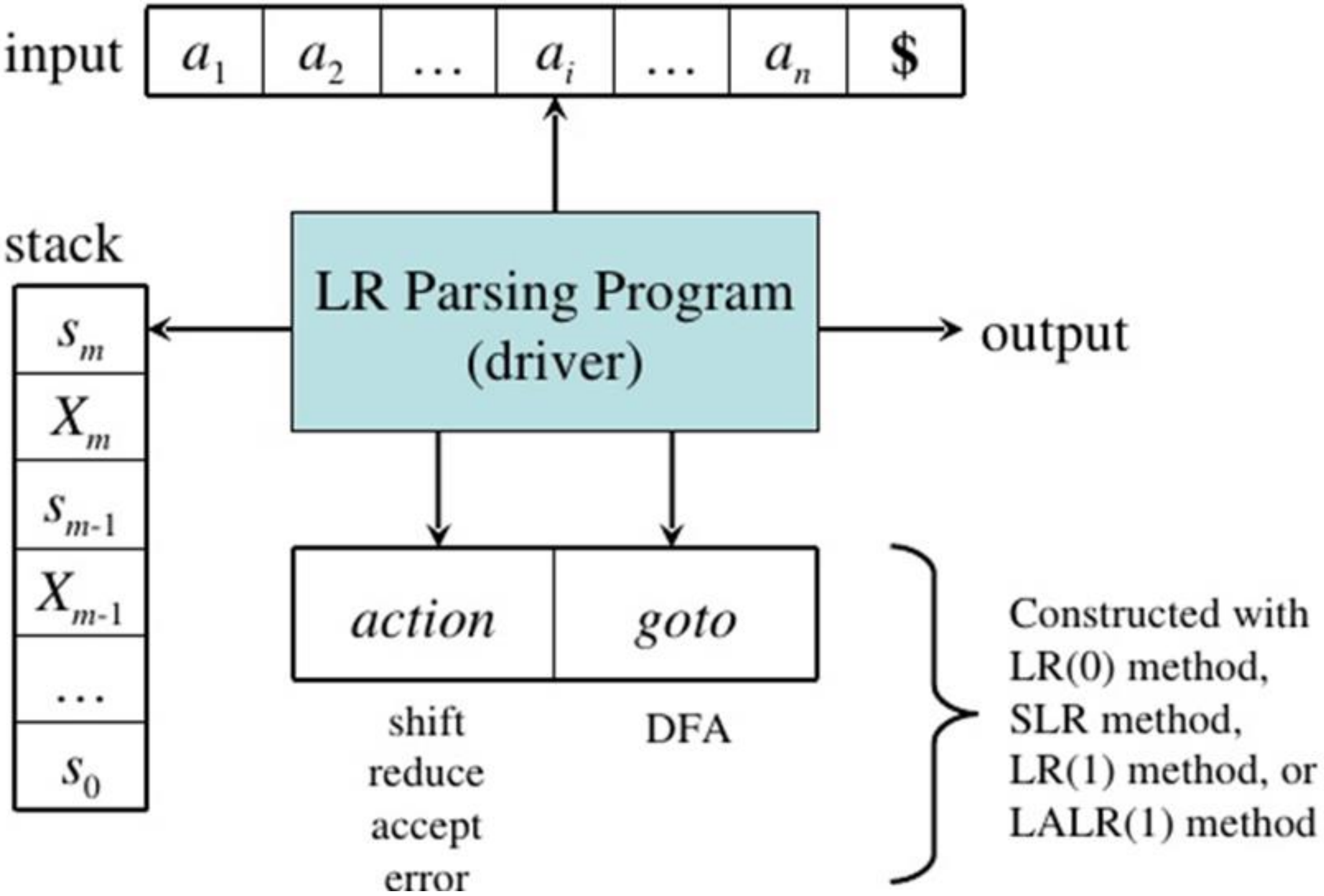
- A bottom-up parser works by maintaining a forest of **partially completed** subtrees of the parse tree, which it joins together whenever it recognizes the symbols on the right-hand side of some production used in the right-most derivation of the input string.
- It creates a new internal node and makes 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 6

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:

	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			

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) $	

	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			

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	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			

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	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			

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 7

Canonical Derivations

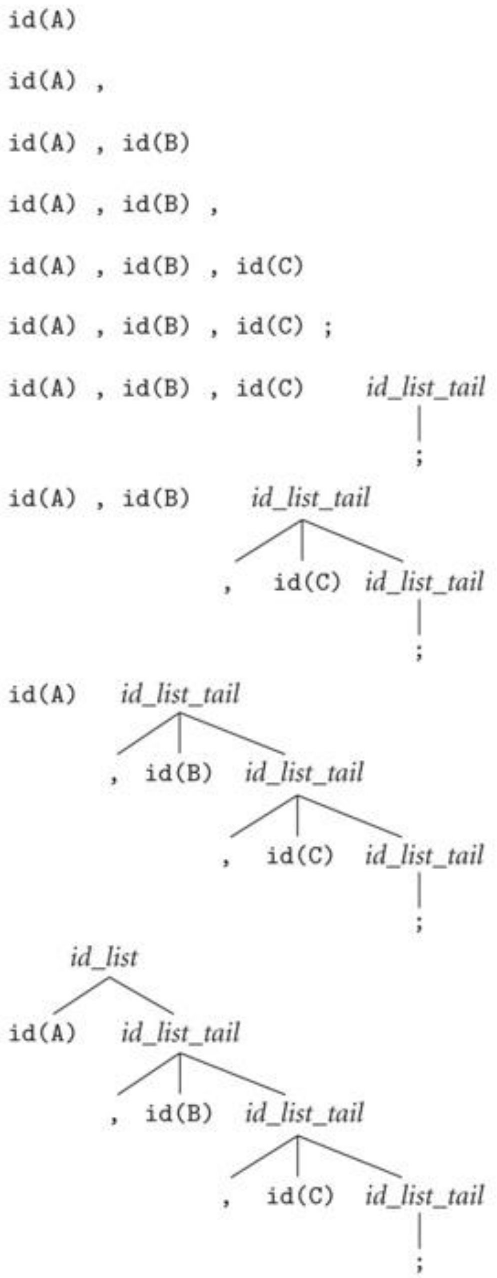
Stack Contents (Roots of Partial Trees):

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



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
```

Design of the Action Table and the Goto Tables

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.

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 (State 0)

Basis

program -> ■ stmt_list $$

Closure

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

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.

- 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 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

read
stmt

Input Token

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

stmt -> read ■ id

Shift

A
read
stmt

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

stmt_list

parse Tree

stmt

read A

Our New State

program -> stmt_list ■ $$

stmt_list -> stmt_list ■ stmt

stmt -> ■ id := expr

stmt -> ■ read id

stmt -> ■ read id

(state 2)

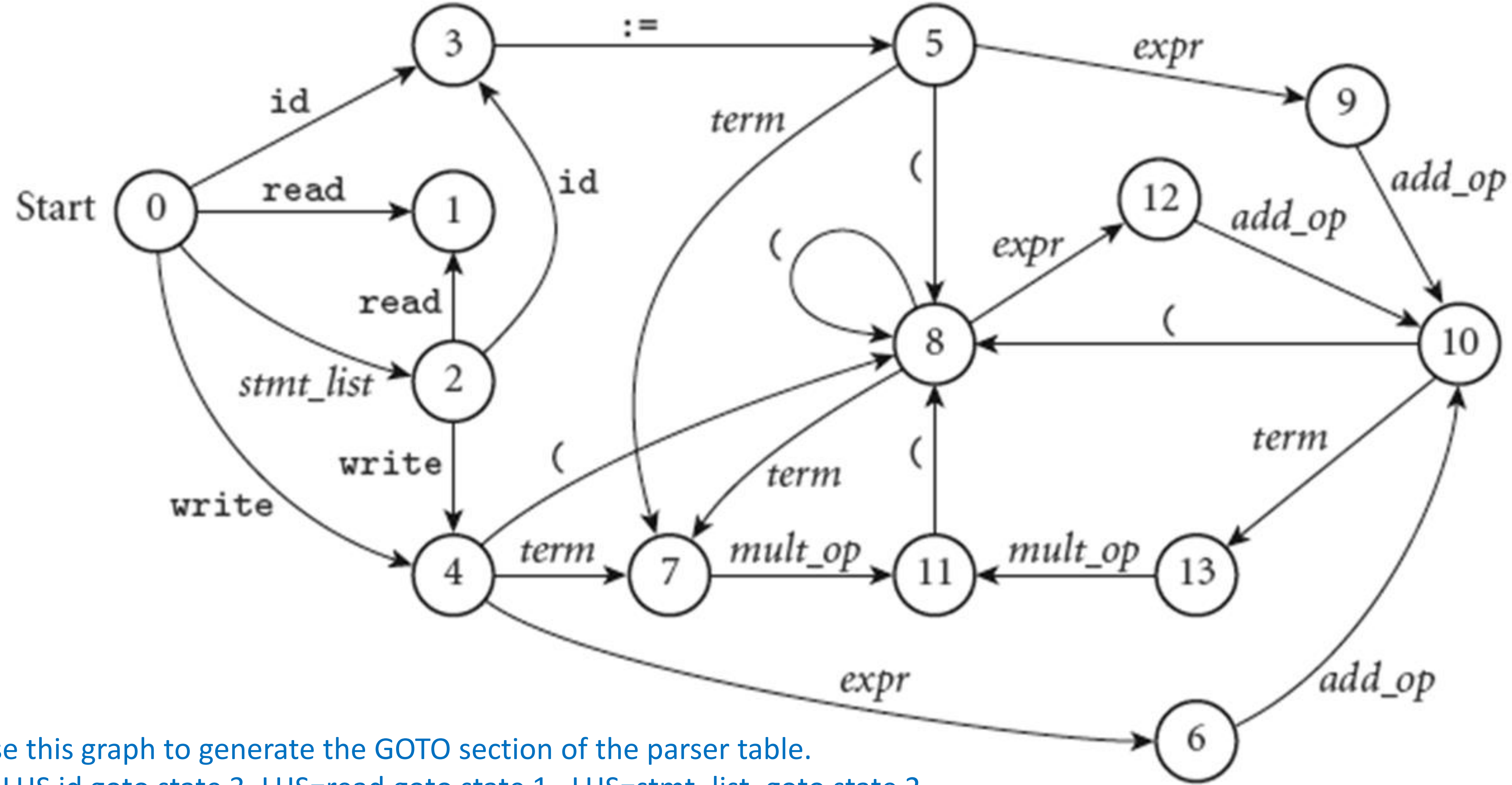
	State	Transitions
0.	*program* ⟶ • *stmt_list* `$$`	on *stmt_list* shift and goto 2
	stmt_list ⟶ • *stmt_list stmt*	
	stmt_list ⟶ • *stmt*	on *stmt* shift and reduce (pop 1 state, push *stmt_list* on input)
	stmt ⟶ • `id` `:=` *expr*	on `id` shift and goto 3
	stmt ⟶ • `read` `id`	on `read` shift and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

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

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,

$$\mathit{stmt\_list} \longrightarrow \mathit{stmt}$$

would need to be replaced with

$$\mathit{stmt\_list} \longrightarrow \epsilon$$

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.

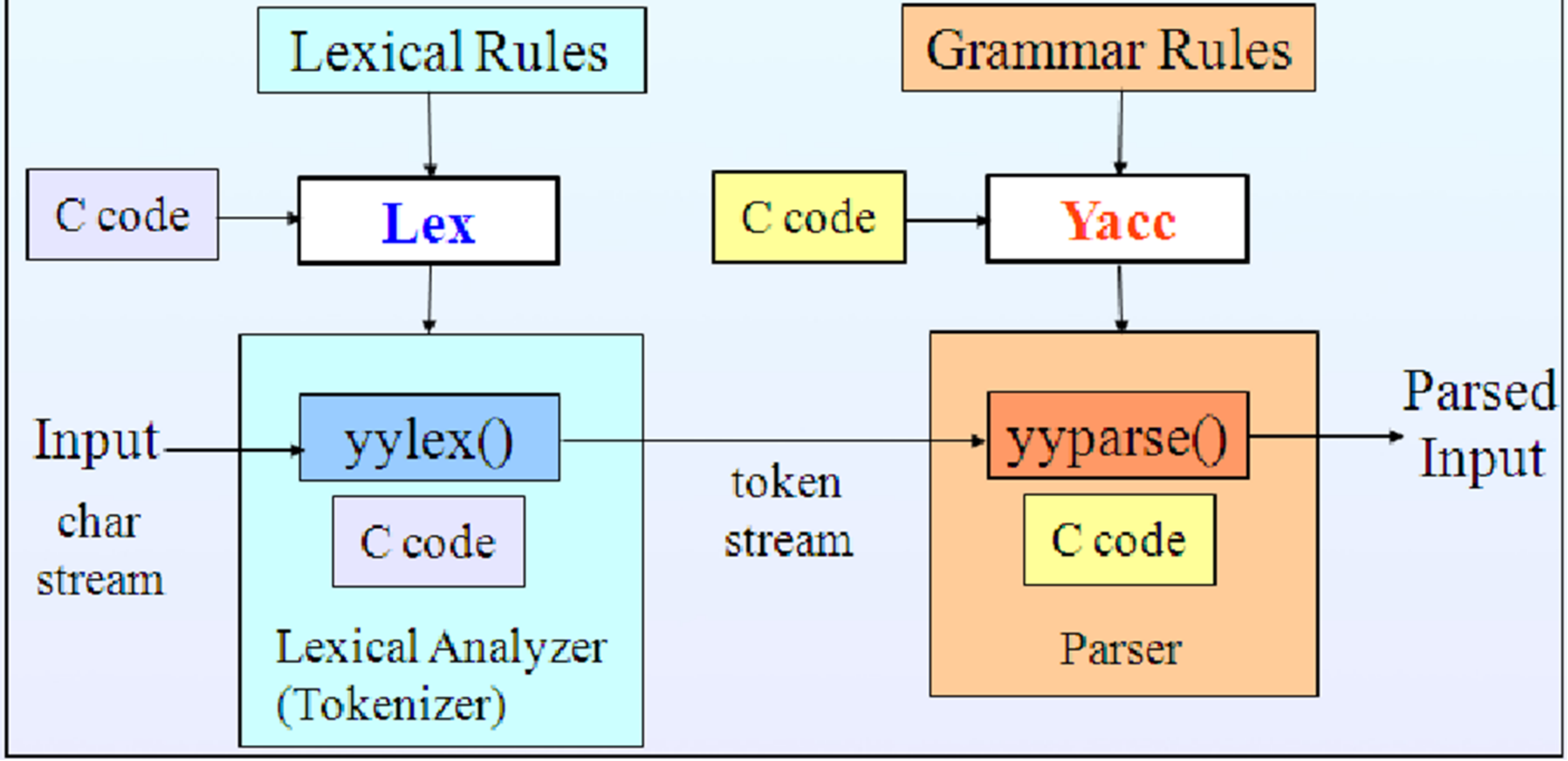
Overview for Compiler-Compiler

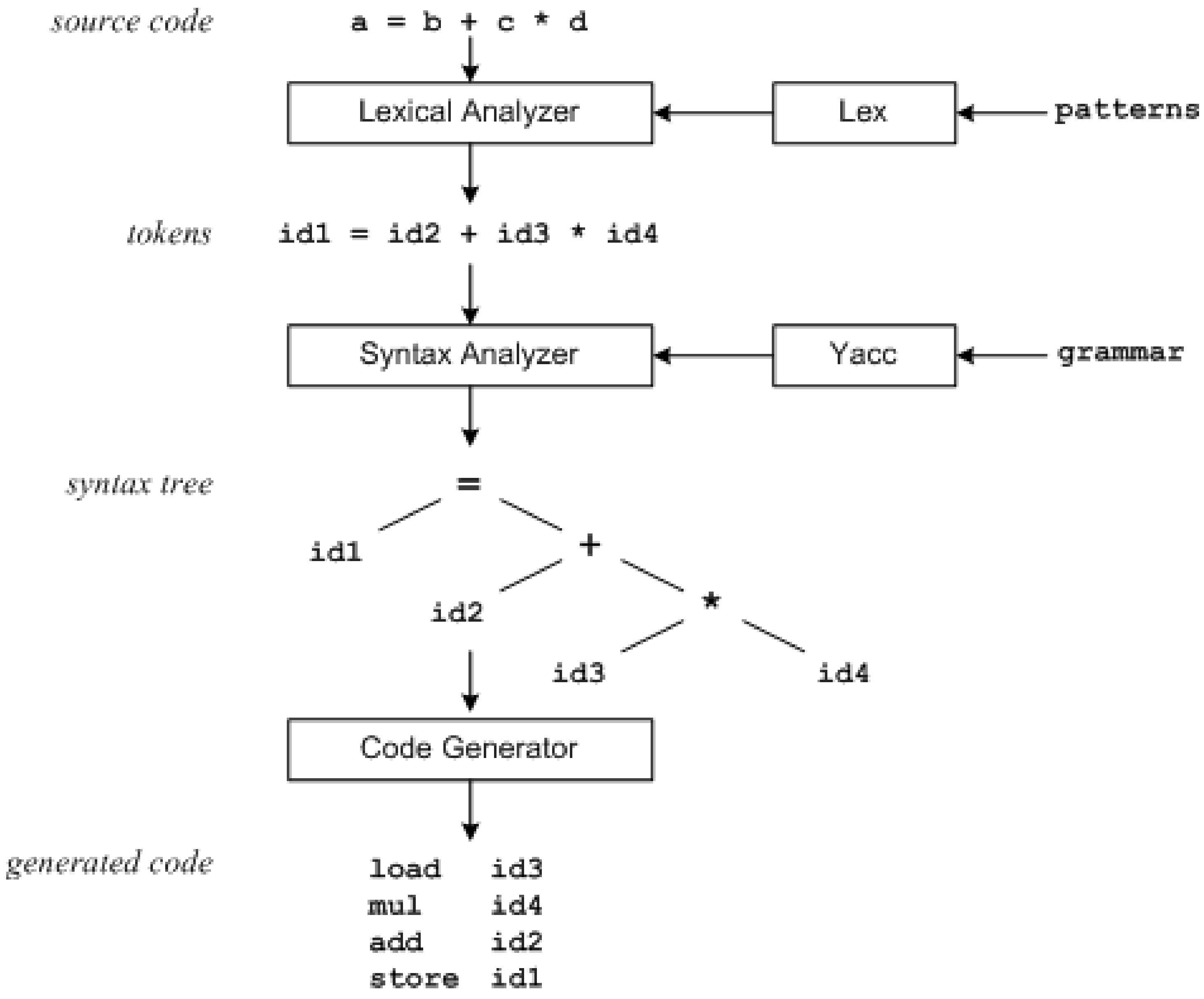
SECTION 9

Lex and Yacc

http://dinosaur.compilertools.net/

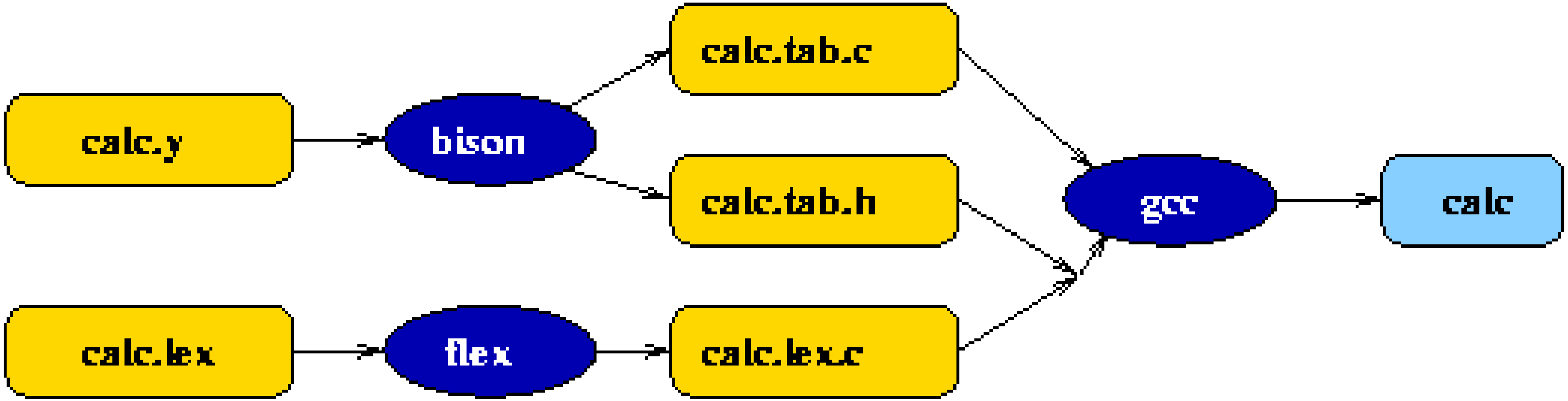
- **Lex** and **Yacc** generate C code for your analyzer & parser.



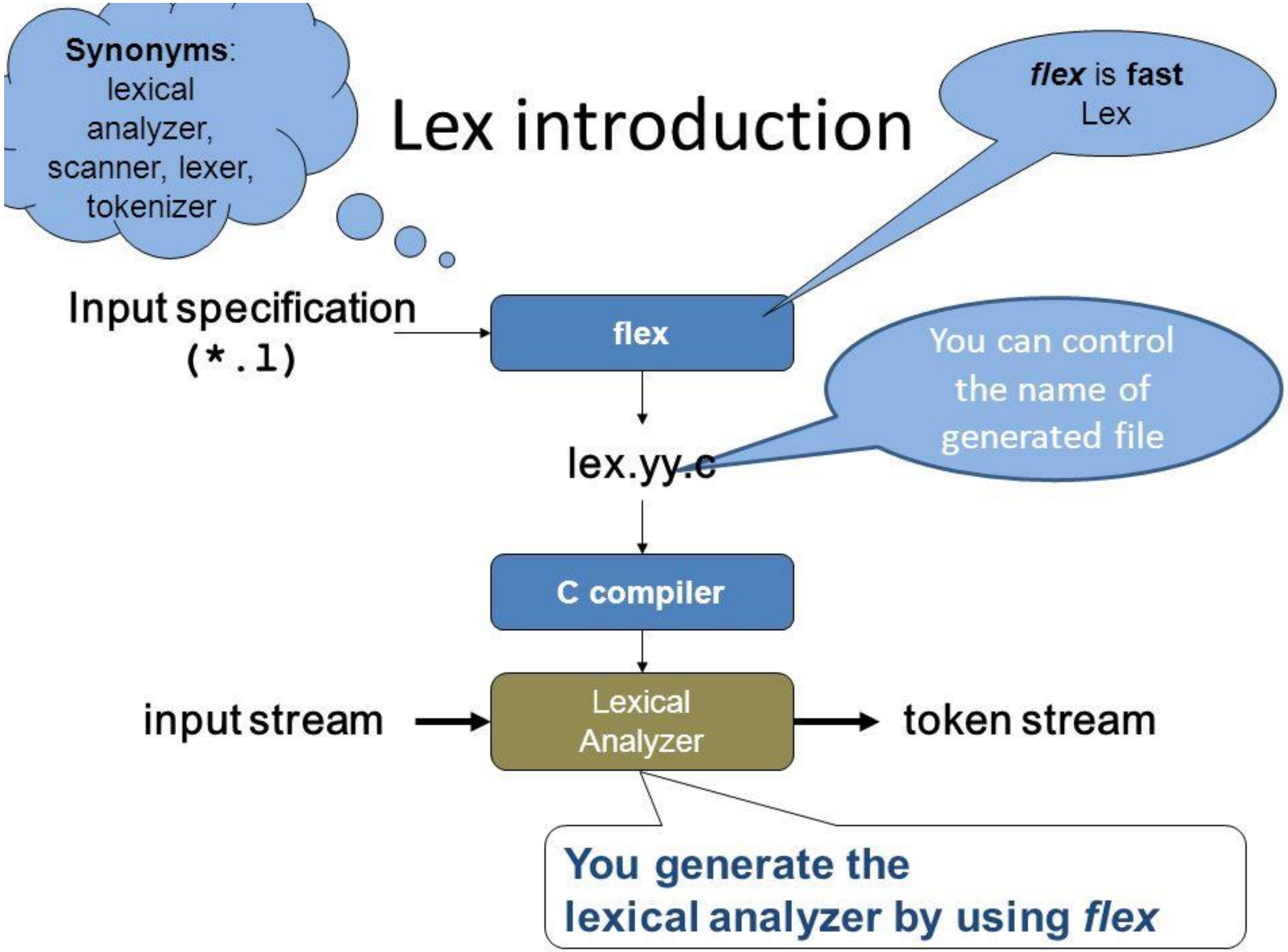

source code
a = b + c * d
Lexical Analyzer
Lex
patterns
tokens
id1 = id2 + id3 * id4
Syntax Analyzer
Yacc
grammar
syntax tree
=
id1
+
id2
*
id3
id4
Code Generator
generated code
load id3
mul id4
add id2
store id1

Calc Language Compiler Generation

flex(LEX), bison(yacc)

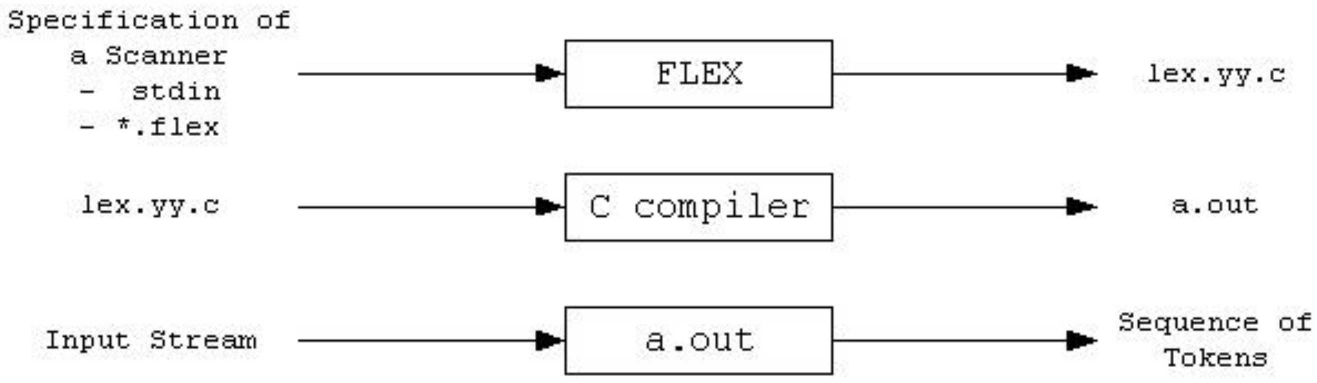


Lex introduction



flex - fast lexical analyzer generator

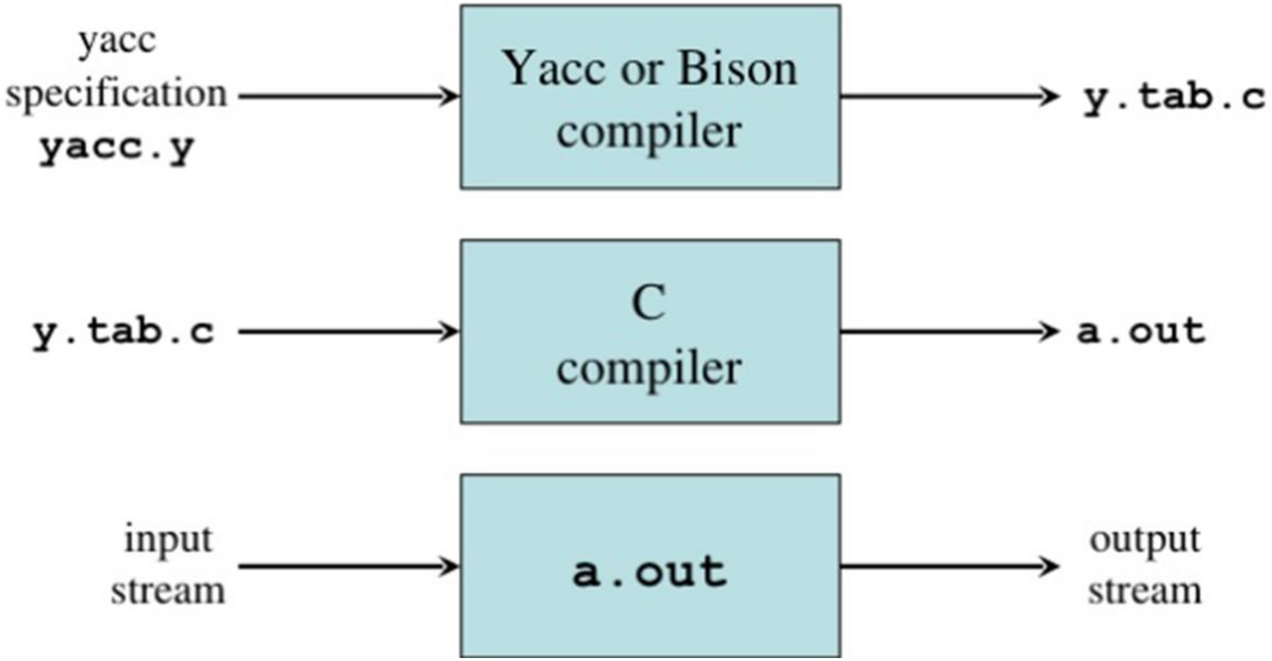
- Flex is a tool for generating scanners.
- Flex source is **a table of regular expressions** and corresponding program fragments.
- Generates lex.yy.c which defines a routine yylex()



Flex for Windows: http://gnuwin32.sourceforge.net/packages/flex.htm
Win flex-bison: https://sourceforge.net/projects/winflexbison/

Parser generator

- Takes a specification for a context-free grammar.
- Produces code for a parser.

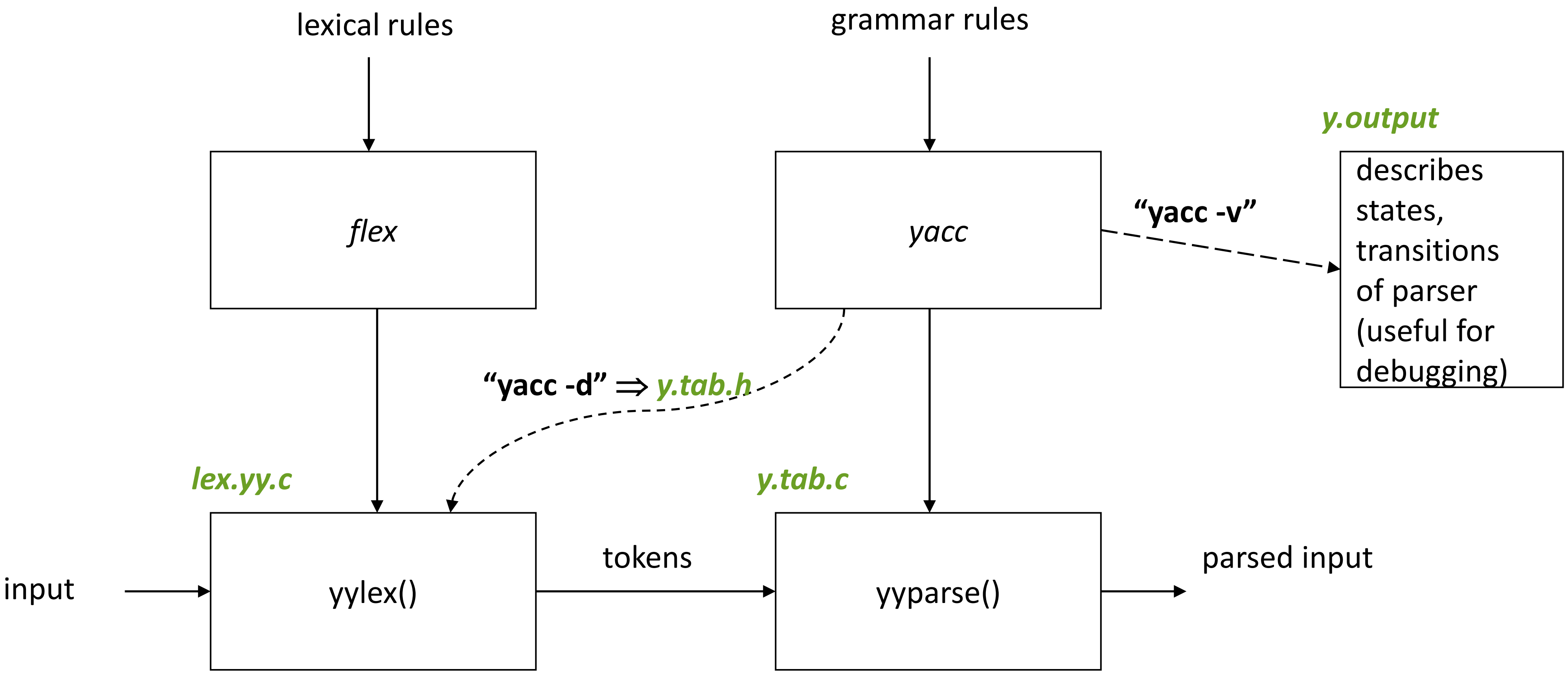


Bison for Windows: http://gnuwin32.sourceforge.net/packages/bison.htm

Scanner-Parser interaction

- Parser assumes the existence of a function '**int yylex()**' that implements the scanner.
- Scanner:
  - return value indicates the type of token found;
  - other values communicated to the parser using **yytext**, **yylval** (see man pages).
- Yacc determines integer representations for tokens:
  - Communicated to scanner in file **y.tab.h**
    - use "**yacc -d**" to produce **y.tab.h**
  - Token encodings:
    - "end of file" represented by '0';
    - a character literal: its ASCII value;
    - other tokens: assigned numbers ≥ 257.

Using Yacc



lexical rules
grammar rules
flex
yacc
"yacc -v"
y.output
describes states, transitions of parser (useful for debugging)
"yacc -d" ⇒ y.tab.h
lex.yy.c
y.tab.c
input
yylex()
tokens
yyparse()
parsed input

int yyparse()

Called once from main() [*user-supplied*]

Repeatedly calls yylex() until done:

- On syntax error, calls yyerror() [*user-supplied*]
- Returns 0 if all of the input was processed;
- Returns 1 if aborting due to syntax error.

Example:

```
int main() { return yyparse(); }
```

yacc: input format

A yacc input file has the following structure:

required

```
definitions
%%
rules
%%
user code
```

optional

Shortest possible legal yacc input:

```
%%
```

Bison Grammar Rules

Input Format for Bison (CFG)

```
%%          /* Bison grammar rules */
input     : /* allow empty input */
          | input line
          ;
line      : expr '\n'    { printf("Result is %f\n", $1); }
expr      : expr '+' term    { $$ = $1 + $3; }
          | expr '-' term    { $$ = $1 - $3; }
          | term             { $$ = $1; }
          ;
term      : term '*' factor { $$ = $1 * $3; }
          | term '/' factor { $$ = $1 / $3; }
          | factor           { $$ = $1; }
          ;
factor    : '(' expr ')'     { $$ = $2; }
          | NUMBER           { $$ = $1; }
          | '-' NUMBER       { $$ = -$2; }
          ;
```

Rules

<u>Grammar production</u>		<u>yacc rule</u>
$A \rightarrow B_1\ B_2 \ldots B_m$		$A \rightarrow B_1\ B_2 \ldots B_m$
$A \rightarrow C_1\ C_2 \ldots C_n$	➡	$\mid\ C_1\ C_2 \ldots C_n$
$A \rightarrow D_1\ D_2 \ldots D_k$		$\mid\ D_1\ D_2 \ldots D_k$
		; /* ';' optional, but advised */

- Rule RHS can have arbitrary **C code** embedded, within { … }. E.g.:

  A : B1 { printf("after B1\n"); x = 0; } B2 { x++; } B3

- Left-recursion more efficient than right-recursion:
  - A : A x | …   rather than   A : x A | …