3 Parsing

(**3. Parsing, p. 41**)

3.1 Context-Free Grammars

- Now that we have moved on to the **Parse** phase of our compiler, we can—thanks to lexical analysis—work with tokens (instead of character sequences).

- The **parser** analyses the incoming token stream and

  ① checks that the stream is indeed valid according to the **syntax rules** of the Tiger language, and

  ② builds a **parse tree** for the token stream.

- The parse tree is already suitable to **derive the actual meaning** of the source program. We thus need to be careful to build the correct parse tree for a given program.

- We will use **context-free grammars** (**CFGs**) to formally specify the syntax rules of our source language.

- Why not use REs for this task?

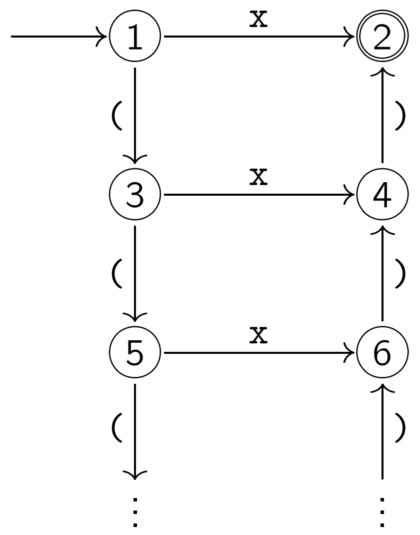
  [*An RE can describe valid sequences of symbols, namely the RE's language. Let's just use REs operating on tokens rather than single characters.*]

  - REs are simply not powerful enough to represent the **recursive** nature of the syntax of typical source languages.
  - **Example:** consider the simple syntax rule (this is actually a tiny CFG):

$$E \quad \rightarrow \quad \mathtt{x} \mid \mathtt{(}\ E\ \mathtt{)}$$

    Clearly, our intention is $L(E) = \{\texttt{"x"}, \texttt{"(x)"}, \texttt{"((x))"}, \texttt{"(((x)))"}, \ldots\}$.

- If we can define this language via an RE then there must be an equivalent DFA A with $L(A) = L(E)$.
  A would need to look similar to what is shown below:



- No DFA will be "deep enough" to accept all parenthesized inputs ("*A DFA cannot count.*").

• Parsing algorithms for CFGs therefore use a **stack** to remember which input tokens have already been consumed.

- **Definition CFG:**
  - A CFG is defined by a **set of productions** (**rules**) of the form

$$S \quad \rightarrow \quad \underbrace{S\ S\ \cdots\ S}_{\text{zero or more}}$$

  - Each **symbol** S is either a **terminal** ($=$ token) or a **non-terminal**.
  - Only non-terminals may appear on the left-hand sides (i.e., left of the $\rightarrow$) of productions.
  - One non-terminal symbol is marked as the **start symbol**.
  - **Example:** a CFG describing a straight-line programming language (start symbol S'):

$$\begin{array}{lll}
{}_0\, S' \rightarrow S\ \$ & {}_4\, E \rightarrow \texttt{id} & {}_8\, L \rightarrow E \\
{}_1\, S\ \rightarrow S\ \texttt{;}\ S & {}_5\, E \rightarrow \texttt{num} & {}_9\, L \rightarrow L\ \texttt{,}\ E \\
{}_2\, S\ \rightarrow \texttt{id}\ \texttt{:=}\ E & {}_6\, E \rightarrow E\ \texttt{+}\ E & \\
{}_3\, S\ \rightarrow \texttt{print}\ \texttt{(}\ L\ \texttt{)} & {}_7\, E \rightarrow \texttt{(}\ S\ \texttt{,}\ E\ \texttt{)} &
\end{array}$$

- **N.B.**: `$` is the end-of-file (EOF) symbol which marks the end of any input token stream.

3.1.1 Derivations (and Parse Trees)

- Just like an RE or DFA/NFA, a CFG describes a language.

- A **derivation** is like a proof that a given token sequence is **accepted** by a CFG G:

  ① Select the start symbol S of G as the next non-terminal E to expand.

  ② Replace the selected non-terminal E by *some* right-hand side of an E-production in G.

  ③ This right-hand side will, in general, contain more non-terminals. Choose of these non-terminals to be the new E, *goto* ②.
  If only terminals remain, *stop*.

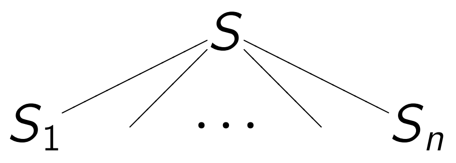
  The resulting token sequence is in $L(G)$, i.e., accepted by CFG G.

- **Example:** derive a token sequence from the start symbol S' of the CFG shown ealier (the next non-terminal chosen for replacement is underlined)[6]:

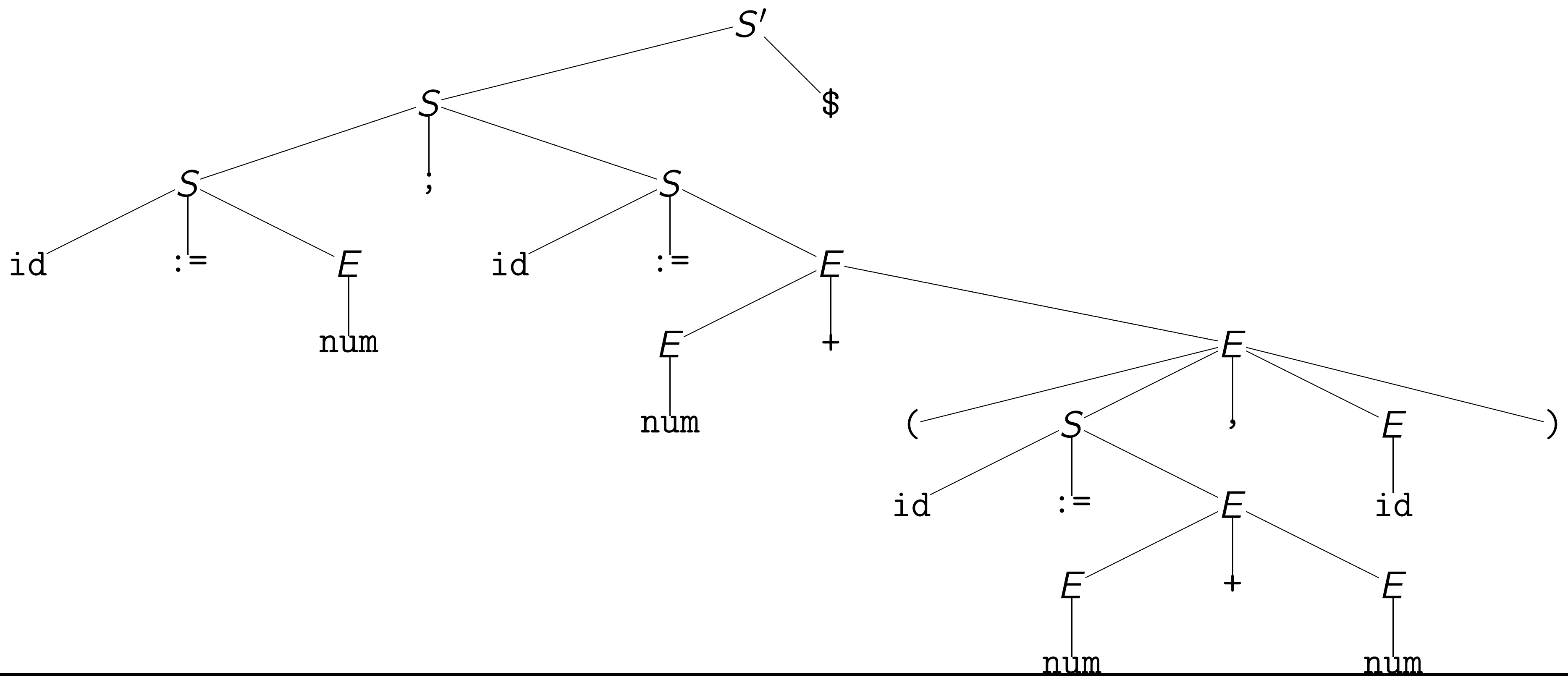
$$\begin{array}{l}
\underline{S'} \\
\underline{S}\ \$ \\
S\ ;\ \underline{S}\ \$ \\
\underline{S}\ ;\ \mathtt{id}\ \mathtt{:=}\ E\ \$ \\
\mathtt{id}\ \mathtt{:=}\ \underline{E}\ ;\ \mathtt{id}\ \mathtt{:=}\ E\ \$ \\
\mathtt{id}\ \mathtt{:=}\ \mathtt{num}\ ;\ \mathtt{id}\ \mathtt{:=}\ \underline{E}\ \$ \\
\mathtt{id}\ \mathtt{:=}\ \mathtt{num}\ ;\ \mathtt{id}\ \mathtt{:=}\ E + \underline{E}\ \$ \\
\mathtt{id}\ \mathtt{:=}\ \mathtt{num}\ ;\ \mathtt{id}\ \mathtt{:=}\ \underline{E} + (\ S\ ,\ E\ )\ \$ \\
\mathtt{id}\ \mathtt{:=}\ \mathtt{num}\ ;\ \mathtt{id}\ \mathtt{:=}\ \mathtt{id} + (\ \underline{S}\ ,\ E\ )\ \$ \\
\mathtt{id}\ \mathtt{:=}\ \mathtt{num}\ ;\ \mathtt{id}\ \mathtt{:=}\ \mathtt{id} + (\ \mathtt{id}\ \mathtt{:=}\ \underline{E}\ ,\ E\ )\ \$ \\
\mathtt{id}\ \mathtt{:=}\ \mathtt{num}\ ;\ \mathtt{id}\ \mathtt{:=}\ \mathtt{id} + (\ \mathtt{id}\ \mathtt{:=}\ E + E\ ,\ \underline{E}\ )\ \$ \\
\mathtt{id}\ \mathtt{:=}\ \mathtt{num}\ ;\ \mathtt{id}\ \mathtt{:=}\ \mathtt{id} + (\ \mathtt{id}\ \mathtt{:=}\ \underline{E} + E\ ,\ \mathtt{id}\ )\ \$ \\
\mathtt{id}\ \mathtt{:=}\ \mathtt{num}\ ;\ \mathtt{id}\ \mathtt{:=}\ \mathtt{id} + (\ \mathtt{id}\ \mathtt{:=}\ \mathtt{num} + \underline{E}\ ,\ \mathtt{id}\ )\ \$ \\
\mathtt{id}\ \mathtt{:=}\ \mathtt{num}\ ;\ \mathtt{id}\ \mathtt{:=}\ \mathtt{id} + (\ \mathtt{id}\ \mathtt{:=}\ \mathtt{num} + \mathtt{num}\ ,\ \mathtt{id}\ )\ \$
\end{array}$$

[6] We omit the semantic values associated with token types `id` and `num`.

- The derivation of a token sequence yields the **parse tree** associated with the derivation:
  - For each expansion of a production $S \rightarrow S_1 \; \cdots \; S_n$ build the parse tree fragment



- **Example:** parse tree associated with the derivation shown on previous slide:



- **Remarks:**
  - In any parse tree, the **start symbol** will always appears at the **root**.
  - The **front** of the parse tree, read from left to right, is the input token sequence.

3.1.2 Ambigiuous Grammars

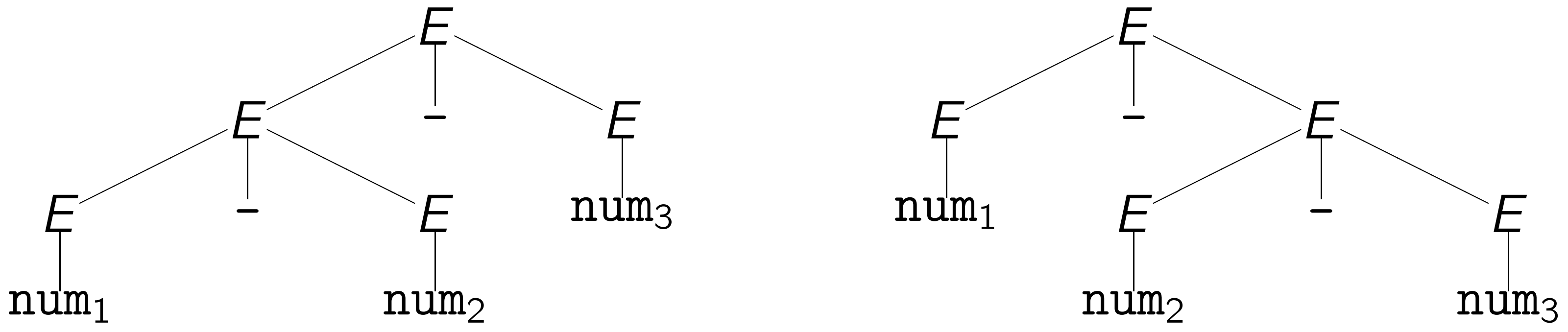
- Compilers use parse tree to **infer the meaning** of a given source program.
  - If we can build more than one parse tree (e.g., have more than one derivation) for a given token sequence, which meaning shall be the correct one?

- Grammars that associate more than one parse tree with a given token sequence are **ambigiuous**. Ambiguous grammars are *not* suitable for parsing.

- **Example:** consider the CFG G for expressions over the usual binary arithmetic operators (with their well-known mathematical semantics):

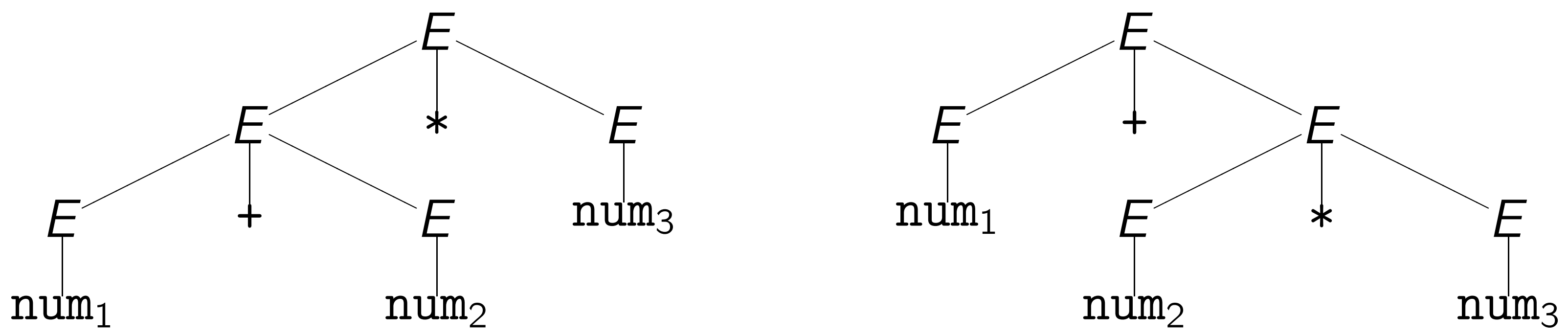
$$\begin{array}{lllllll} E & \to & \texttt{id} & \qquad & E & \to & E \texttt{ * } E \\ E & \to & \texttt{num} & & E & \to & E \texttt{ / } E \\ E & \to & \texttt{(} E \texttt{)} & & E & \to & E \texttt{ + } E \\ & & & & E & \to & E \texttt{ - } E \end{array}$$

  - G is ambiguous since we may derive two parse trees for the token sequence $\texttt{num}_1\texttt{-num}_2\texttt{-num}_3$:



Shall the binary operators **associate** *to the left or right?*

– G is ambigiuous since we may derive two parse trees for the token sequence $\texttt{num}_1\texttt{+num}_2\texttt{*num}_3$:



- The associated meanings would yield different machine code and thus different results.

- It is often possible to **transform** a CFG such that

① the accepted language does *not* change, and

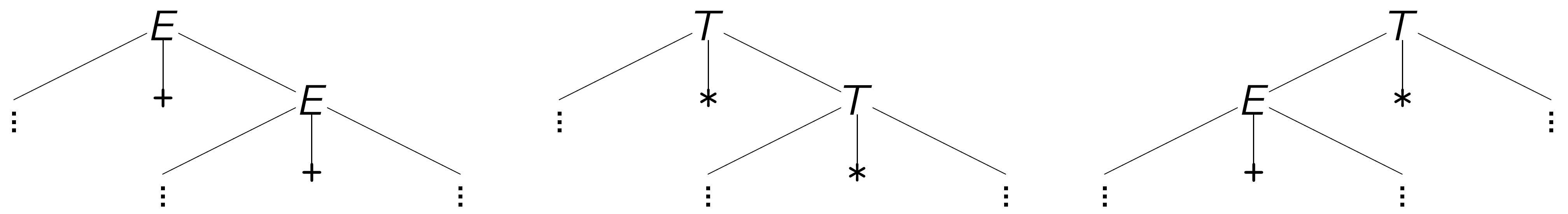
② any input token sequence is assigned exactly one parse tree.

- **Example:** transform the expression grammar G into a CFG G' such that the parse trees reflect usual mathematical semantics:

$$\begin{array}{lcl@{\qquad\qquad}lcl@{\qquad\qquad}lcl}
E & \to & E \texttt{+} T & T & \to & T \texttt{*} F & F & \to & \texttt{id} \\
E & \to & E \texttt{-} T & T & \to & T \texttt{/} F & F & \to & \texttt{num} \\
E & \to & T & T & \to & F & F & \to & \texttt{(} E \texttt{)}
\end{array}$$

(E, T, F stand for *expression*, *term*, *factor*, respectively.)

- **Note:** with G', we will *never* be able to derive parse trees shaped as shown below:



- The "layering" in the grammar expresses operator **priority**.
- The **left recursion** in the E, T-productions expresses operator **associativity**.

3.2 LR Parsing

(**3.3 LR Parsing, p. 56**)

- We will now discuss the LR(k) family of **parsing algorithms** which also lies at the core of modern parser program generators, like `yacc` or `bison`.

- An LR(k) parser

  L - reads the input token sequence from **left** to right,
  R - performs a **right-most** derivation during parsing,
  k - uses a **lookahead** of k tokens to guide itself.

- Performing a *right-most* derivation when reading the input *left* to right might sound odd at first, but:

  ① LR(k) decides which production to use *after* it has seen all tokens on the right-hand side of the production ($+k$ tokens more).
  ② Before this decision is made by the parser, it temporarily saves the tokens on a **stack**.

- **Overview LR parsing:**
  - The parser maintains its **input** (token sequence) and a **stack** (of CFG symbols).
  - While the parser reads the input it can perform two **actions**:

① **shift**:

Consume first token of input, push token onto the stack:

Stack	*Input*	*Action*
$S_1 \cdots S_n$	`id` + `num` $\cdots$ \$	*shift*
$S_1 \cdots S_n$ `id`	+ `num` $\cdots$ \$	

② **reduce** (production $S \to S_1 \cdots S_n$):

Pop $S_n, \cdots, S_1$ from the stack, then push S onto the stack:

Stack	*Input*	*Action*
$S_0\ S_1 \cdots S_n$	`id` + `num` $\cdots$ \$	*reduce* $S \to S_1 \cdots S_n$
$S_0\ S$	`id` + `num` $\cdots$ \$	

  - Shifting the \$ token is also called **accepting** (the parser has consumed all tokens).

Stack	*Input*	*Action*
	id := num ; id := id + (id := num + num , id) $	*shift*
id	:= num ; id := id + (id := num + num , id) $	*shift*
id :=	num ; id := id + (id := num + num , id) $	*shift*
id := num	; id := id + (id := num + num , id) $	*reduce* $E \rightarrow$ num
id := E	; id := id + (id := num + num , id) $	*reduce* $S \rightarrow$ id := E
S	; id := id + (id := num + num , id) $	*shift*
S ;	id := id + (id := num + num , id) $	*shift*
S ; id	:= id + (id := num + num , id) $	*shift*
S ; id :=	id + (id := num + num , id) $	*shift*
S ; id := id	+ (id := num + num , id) $	*reduce* $E \rightarrow$ id
S ; id := E	+ (id := num + num , id) $	*shift*
S ; id := E +	(id := num + num , id) $	*shift*
S ; id := E + (	id := num + num , id) $	*shift*
S ; id := E + (id	:= num + num , id) $	*shift*
S ; id := E + (id :=	num + num , id) $	*shift*
S ; id := E + (id := num	+ num , id) $	*reduce* $E \rightarrow$ num
S ; id := E + (id := E	+ num , id) $	*shift*
S ; id := E + (id := E +	num , id) $	*shift*
S ; id := E + (id := E + num	, id) $	*reduce* $E \rightarrow$ num
S ; id := E + (id := E + E	, id) $	*reduce* $E \rightarrow E + E$
S ; id := E + (id := E	, id) $	*reduce* $S \rightarrow$ id := E
S ; id := E + (S	, id) $	*shift*
S ; id := E + (S ,	id) $	*shift*
S ; id := E + (S , id	) $	*reduce* $E \rightarrow$ id
S ; id := E + (S , E	) $	*shift*
S ; id := E + (S , E)	$	*reduce* $E \rightarrow (S , E)$
S ; id := E + E	$	*reduce* $E \rightarrow E + E$
S ; id := E	$	*reduce* $S \rightarrow$ id := E
S ; S	$	*reduce* $S \rightarrow S ; S$
S	$	*accept* (*shift* $)

- **Remarks:**
  - The concatenation of stack and remaining input always represents the whole input.
  - Read bottom-up, the LR parser determines a right-most derivation of the input.

- The LR parser needs some control to decide when to shift and when to reduce.

> **Idea:** We *cannot* use a DFA applied to the input (DFAs/REs are too weak to handle CFGs), but we can **apply the DFA idea to the LR stack**.

  - Use a **parsing table** (transition matrix) $goto[d][S]$ to implement the DFA (d: DFA state, S: CFG symbol).
  - The DFA edges are labelled with the symbols that may occur on the LR stack, i.e., tokens *and* non-terminals.
  - As an optimization, with each symbol we push onto the stack we also push the current DFA state (later, when we pop the stack, we can easily determine which state we are in after the pops are done).
  - The transition matrix entries indicate which **action** the LR parser has to perform:

Matrix entry	**Action**
$s\ n$	*shift* (next state is n)
$r\ S \rightarrow S_1 \cdots S_k$	*reduce* by given rule
a	*accept*

- **LR parsing algorithm:**
  - **Given:** input token sequence $t_0\, t_1 \cdots t_n\, \$$, LR parsing table *goto*[][], empty LR stack[7].
  - **Output:** *accept* or *error*.

```
ip ← 0;          /* pointer into input token sequence */
push(1);         /* we start in state 1 */

while true do
    s ← top();
    switch goto[s][t_ip] do
        case s n:                      /* shift */
            push(t_ip);
            push(n);
            ip ← ip + 1;
        case r S → S_1 ··· S_k:        /* reduce */
            pop() 2 × k times;
            s' ← top();
            push(S);
            push(goto[s'][S]);
        case a:                        /* accept */
            return accept;
        otherwise
            return error;
```

[7] With the usual stack operations *push*(), *pop*(), *top*().

						CFG symbol							
	`id`	`num`	`print`	`;`	`,`	`+`	`:=`	`(`	`)`	`$`	S	E	L
1	$s4$		$s7$								2		
2				$s3$						a			
3	$s4$		$s7$								5		
4							$s6$						
5				$r\ S \rightarrow S$`;`S	$r\ S \rightarrow S$`;`S					$r\ S \rightarrow S$`;`S			
6	$s20$	$s10$						$s8$				11	
7								$s9$					
8	$s4$		$s7$								12		
9	$s20$	$s10$						$s8$				15	14
10				$r\ E \rightarrow$ `num`	$r\ E \rightarrow$ `num`	$r\ E \rightarrow$ `num`			$r\ E \rightarrow$ `num`	$r\ E \rightarrow$ `num`			
11				$r\ S \rightarrow$ `id:=`E	$r\ S \rightarrow$ `id:=`E	$s16$				$r\ S \rightarrow$ `id:=`E			
12				$s3$	$s18$								
13				$r\ S \rightarrow$ `print(`L`)`	$r\ S \rightarrow$ `print(`L`)`					$r\ S \rightarrow$ `print(`L`)`			
14					$s19$				$s13$				
15					$r\ L \rightarrow E$				$r\ L \rightarrow E$				
16	$s20$	$s10$						$s8$				17	
17				$r\ E \rightarrow E$`+`E	$r\ E \rightarrow E$`+`E	$s16$			$r\ E \rightarrow E$`+`E	$r\ E \rightarrow E$`+`E			
18	$s20$	$s10$						$s8$				21	
19	$s20$	$s10$						$s8$				23	
20				$r\ E \rightarrow$ `id`	$r\ E \rightarrow$ `id`	$r\ E \rightarrow$ `id`			$r\ E \rightarrow$ `id`	$r\ E \rightarrow$ `id`			
21									$s22$				
22				$r\ E \rightarrow$ `(`S`,`E`)`	$r\ E \rightarrow$ `(`S`,`E`)`	$r\ E \rightarrow$ `(`S`,`E`)`			$r\ E \rightarrow$ `(`S`,`E`)`	$r\ E \rightarrow$ `(`S`,`E`)`			
23					$r\ L \rightarrow L$`,`E	$s16$			$r\ L \rightarrow L$`,`E				

Stack	*Input*	*Action*
$_1$	id := num ; id := id + (id := num + num , id) $	*shift*
$_1$ id$_4$	:= num ; id := id + (id := num + num , id) $	*shift*
$_1$ id$_4$:=$_6$	num ; id := id + (id := num + num , id) $	*shift*
$_1$ id$_4$:=$_6$ num$_{10}$	; id := id + (id := num + num , id) $	*reduce* *E* → num
$_1$ id$_4$:=$_6$ *E*$_{11}$	; id := id + (id := num + num , id) $	*reduce* *S* → id := *E*
$_1$ *S*$_2$	; id := id + (id := num + num , id) $	*shift*
$_1$ *S*$_2$;$_3$	id := id + (id := num + num , id) $	*shift*
$_1$ *S*$_2$;$_3$ id$_4$	:= id + (id := num + num , id) $	*shift*
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$	id + (id := num + num , id) $	*shift*
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ id$_{20}$	+ (id := num + num , id) $	*reduce* *E* → id
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ *E*$_{11}$	(id := num + num , id) $	*shift*
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$	(id := num + num , id) $	*shift*
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$ ($_8$	id := num + num , id) $	*shift*
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$ ($_8$ id$_4$	:= num + num , id) $	*shift*
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$ ($_8$ id$_4$:=$_6$	num + num , id) $	*shift*
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$ ($_8$ id$_4$:=$_6$ num$_{10}$	+ num , id) $	*reduce* *E* → num
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$ ($_8$ id$_4$:=$_6$ *E*$_{11}$	+ num , id) $	*shift*
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$ ($_8$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$	num , id) $	*shift*
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$ ($_8$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$ num$_{10}$	, id) $	*reduce* *E* → num
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$ ($_8$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$ *E*$_{17}$	, id) $	*reduce* *E* → *E* + *E*
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$ ($_8$ id$_4$:=$_6$ *E*$_{11}$	, id) $	*reduce* *S* → id := *E*
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$ ($_8$ *S*$_{12}$	, id) $	*shift*
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$ ($_8$ *S*$_{12}$,$_{18}$	id) $	*shift*
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$ ($_8$ *S*$_{12}$,$_{18}$ id$_{20}$	) $	*reduce* *E* → id
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$ ($_8$ *S*$_{12}$,$_{18}$ *E*$_{21}$	) $	*shift*
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$ ($_8$ *S*$_{12}$,$_{18}$ *E*$_{21}$)$_{22}$	$	*reduce* *E* → (*S* , *E*)
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ *E*$_{11}$ +$_{16}$ *E*$_{17}$	$	*reduce* *E* → *E* + *E*
$_1$ *S*$_2$;$_3$ id$_4$:=$_6$ *E*$_{11}$	$	*reduce* *S* → id := *E*
$_1$ *S*$_2$;$_3$ *S*$_5$	$	*reduce* *S* → *S* ; *S*
$_1$ *S*$_2$	$	*accept* (*shift* $)

3.3 LR Parsing Table Generation

- We are still missing a method of deriving an **LR parsing table** from a given CFG. This is what we discuss next.
  - In what follows, we will consider this example CFG (start symbol S):

$_1\ S \rightarrow A\ C\ \$$	$_4\ A \rightarrow \texttt{a}\ B\ C\ \texttt{d}$	$_7\ B \rightarrow \texttt{b}\ B$
$_2\ C \rightarrow \texttt{c}$	$_5\ A \rightarrow B\ Q$	$_8\ B \rightarrow \texttt{d}$
$_3\ C \rightarrow$	$_6\ A \rightarrow$	$_9\ Q \rightarrow \texttt{q}$

- The LR parsing algorithm imitates a DFA. So, given a CFG, how can we characterize the **state** the LR parser is in currently?
  - When parsing starts, the LR parser tries to derive the whole input from the right-hand side of the start symbol. No token has been consumed yet.
    We indicate this state via the **LR item** shown below:

$$S \quad \rightarrow \quad \bullet\ A\ C\ \$$$

– The dot • (i.e., the current parser position) is just before non-terminal A. The start of input must be derivable from non-terminal A.
  We indicate this by **adding the corresponding items** to our state:

$$\begin{array}{lll} S \to & \bullet & A\ C\ \texttt{\$} \\ A \to & \bullet & \texttt{a}\ B\ C\ \texttt{d} \\ A \to & \bullet & B\ Q \\ A \to & \bullet & \end{array}$$

– As long as • is just before a non-terminal (here: B), keep adding the corresponding items (*closure*):

$$\begin{array}{lll} S \to & \bullet & A\ C\ \texttt{\$} \\ A \to & \bullet & \texttt{a}\ B\ C\ \texttt{d} \\ A \to & \bullet & B\ Q \\ A \to & \bullet & \\ B \to & \bullet & \texttt{b}\ B \\ B \to & \bullet & \texttt{d} \end{array}$$

– Once the closure is stable, the LR state is complete.

- We then label each item of the state with an LR action: *shift*, *reduce, goto*, and *accept* (shifting $):

$$\left[\begin{array}{llll}
S \rightarrow & \bullet & A\ C\ \$ & \textit{goto}\ 2 \\
\hline
A \rightarrow & \bullet & \mathtt{a}\ B\ C\ \mathtt{d} & \textit{shift}\ 3 \\
A \rightarrow & \bullet & B\ Q & \textit{goto}\ 4 \\
A \rightarrow & \bullet & & \textit{reduce}\ A \rightarrow \\
B \rightarrow & \bullet & \mathtt{b}\ B & \textit{shift}\ 5 \\
B \rightarrow & \bullet & \mathtt{d} & \textit{shift}\ 6
\end{array}\right]_{\textcircled{1}}$$

  - *shift n*: consume token after •, then goto state *n* (in state *n*: • moved past the token).
  - *goto n*: a right-hand side of the non-terminal after • has just been recognized (reduced), goto state *n* (in state *n*: move • past the non-terminal).
  - *reduce*: • is at the end of a right-hand side, the parser is ready to reduce.
  - *accept*: • would move past $.

- Now simply continue to compute LR states:

$$\left[\begin{array}{lllll} S & \to & \bullet & A\ C\ \$ & \textit{goto}\ 2 \\ \hline A & \to & \bullet & \mathtt{a}\ B\ C\ \mathtt{d} & \textit{shift}\ 3 \\ A & \to & \bullet & B\ Q & \textit{goto}\ 4 \\ A & \to & \bullet & & \textit{reduce}\ A \to \\ B & \to & \bullet & \mathtt{b}\ B & \textit{shift}\ 5 \\ B & \to & \bullet & \mathtt{d} & \textit{shift}\ 6 \end{array}\right]_{\textcircled{1}}$$

$$\left[\begin{array}{llllll} S & \to & A & \bullet & C\ \$ & \textit{goto}\ 7 \\ \hline C & \to & & \bullet & \mathtt{c} & \textit{shift}\ 8 \\ C & \to & & \bullet & & \textit{reduce}\ C \to \end{array}\right]_{\textcircled{2}}$$

$$\left[\begin{array}{llllll} A & \to & \mathtt{a} & \bullet & B\ C\ \mathtt{d} & \textit{goto}\ 9 \\ \hline B & \to & & \bullet & \mathtt{b}\ B & \textit{shift}\ 5 \\ B & \to & & \bullet & \mathtt{d} & \textit{shift}\ 6 \end{array}\right]_{\textcircled{3}}$$

  - Now complete the computation for states ④ . . . ⑮!

3.3.1 Shift/Reduce Conflicts

- In some states, it is not obvious for the parser if *shifting* or *reducing* is the appropriate action: the LR parser table would allow both alternatives.
  - Consider this simple expression CFG and its LR states:

$$\begin{array}{llll} S \to E\ ;\ \$ & \qquad & E \to T \\ E \to T + E & \qquad & T \to \mathtt{x} \end{array}$$

$$\left[\begin{array}{ll} S \to\ \bullet\ E\ ;\ \$ & \textit{goto}\ 2 \\ \hline E \to\ \bullet\ T + E & \textit{goto}\ 3 \\ E \to\ \bullet\ T & \textit{goto}\ 3 \\ T \to\ \bullet\ \mathtt{x} & \textit{shift}\ 4 \end{array}\right]_{\textcircled{1}}$$

$$\left[\begin{array}{ll} S \to E\ \bullet\ ;\ \$ & \textit{shift}\ 5 \end{array}\right]_{\textcircled{2}}$$

$$\left[\begin{array}{ll} T \to \mathtt{x}\ \bullet & \textit{reduce}\ T \to \mathtt{x} \end{array}\right]_{\textcircled{4}}$$

$$\left[\begin{array}{ll} E \to T + E\ \bullet & \textit{reduce}\ E \to T + E \end{array}\right]_{\textcircled{7}}$$

$$\left[\begin{array}{ll} E \to T +\ \bullet\ E & \textit{goto}\ 7 \\ \hline E \to\ \bullet\ T + E & \textit{goto}\ 3 \\ E \to\ \bullet\ T & \textit{goto}\ 3 \\ T \to\ \bullet\ \mathtt{x} & \textit{shift}\ 4 \end{array}\right]_{\textcircled{6}}$$

$$\left[\begin{array}{ll} E \to T\ \bullet\ + E & \textit{shift}\ 6 \\ E \to T\ \bullet & \textit{reduce}\ E \to T \end{array}\right]_{\textcircled{3}}$$

$$\left[\begin{array}{ll} S \to E\ ;\ \bullet\ \$ & \textit{accept} \end{array}\right]_{\textcircled{5}}$$

- There is an apparent conflict in state ③:

$$\left[\begin{array}{lll} E \to T & \bullet\ + E & \textit{shift } 6 \\ E \to T & \bullet & \textit{reduce } E \to T \end{array}\right]_{③}$$

- Shall we shift the + token or reduce via rule $E \to T$[8]?
- Clearly, if the situation is as follows

  2+3+4 • ;

  we want to reduce $E \to T$, but in a case like this

  2+3 • +4;

  we want to shift the +.

> **SLR Rule:** choose *reduce* if the next token is among the tokens that can **follow** the non-terminal (here: E) in the input.

[8]Remember, *reducing* does not consume any token.

- Resolving shift/reduce conflicts with the help of **follow**ing tokens is called **SLR Parsing** (Simple LR).
- The resulting SLR parsing table thus is

CFG symbol

	`x`	`+`	`;`	`$`	E	T
1	$s4$				2	3
2			$s5$			
3		$s6$	$r\ E \to T$			
4	$r\ T \to$ `x`	$r\ T \to$ `x`	$r\ T \to$ `x`	$r\ T \to$ `x`		
5				a		
6	$s4$				7	3
7	$r\ E \to T$+E	$r\ E \to T$+E	$r\ E \to T$+E	$r\ E \to T$+E		

- To resolve such conflicts we need information about the *set of tokens which may possibly follow* some given non-terminal S. We call this set $FOLLOW(S)$.

  Note: in the example above we have $FOLLOW(E) = \{;\}$.

- To compute *FOLLOW*(S), we will additionally need information about the *set of tokens which may occur first* when we derive a non-terminal S. We call this set *FIRST*(S)[9].

  - Consider the CFG

$$\begin{array}{lll} Z \to \mathtt{z} & Y \to & X \to Y \\ Z \to X\ Y\ Z & Y \to \mathtt{y} & X \to \mathtt{x} \end{array}$$

  - *FIRST*(Y) $= \{\mathtt{y}\}$.
  - *FIRST*(X) $= \{ \quad ? \quad \}$.
  - To compute *FIRST*(Z), is it sufficient to compute *FIRST*(X) and then simply say *FIRST*(Z) $=$ *FIRST*(X)?
  - Similarly, to compute *FOLLOW*(X), can we simply say *FOLLOW*(X) $=$ *FIRST*(Y)?

[9] Later, we will need *FIRST*(S) anyway to build the more sophisticated LR(1) parsing tables.

- Compute *FIRST*(), *FOLLOW*(), *nullable*()[10] for a given CFG G.

```
$FIRST(S) \leftarrow \varnothing$, $FOLLOW(S) \leftarrow \varnothing$    for all symbols $S$ in $G$;
$nullable(S) \leftarrow false$                    for all symbols $S$ in $G$;
$FIRST(\mathtt{x}) \leftarrow \{\mathtt{x}\}$                      for all terminals $\mathtt{x}$ in $G$;
repeat
    foreach $X \rightarrow Y_1 \cdots Y_k$ do
        if $k = 0$ or $nullable(Y_1) \wedge \cdots \wedge nullable(Y_k)$ then
            $nullable(X) \leftarrow true$;
        $FIRST(X) \leftarrow FIRST(X) \cup FIRST(Y_1)$;
        $FOLLOW(Y_k) \leftarrow FOLLOW(Y_k) \cup FOLLOW(X)$;
        foreach $i \leftarrow 2 \dots k$ do
            $FOLLOW(Y_{i-1}) \leftarrow FOLLOW(Y_{i-1}) \cup FIRST(Y_i)$;
            /* $X \rightarrow \varepsilon \cdots \varepsilon Y_i \cdots$ */
            if $nullable(Y_1) \wedge \cdots \wedge nullable(Y_{i-1})$ then
                $FIRST(X) \leftarrow FIRST(X) \cup FIRST(Y_i)$;
            /* $X \rightarrow \cdots Y_{i-1} \varepsilon \cdots \varepsilon$ */
            if $nullable(Y_i) \wedge \cdots \wedge nullable(Y_k)$ then
                $FOLLOW(Y_{i-1}) \leftarrow FOLLOW(Y_{i-1}) \cup FOLLOW(X)$;
            /* $X \rightarrow \cdots Y_{i-1} \varepsilon \cdots \varepsilon Y_j \cdots$ */
            foreach $j \leftarrow i+1 \dots k$ do
                if $nullable(Y_i) \wedge \cdots \wedge nullable(Y_{j-1})$ then
                    $FOLLOW(Y_{i-1}) \leftarrow FOLLOW(Y_{i-1}) \cup FIRST(Y_j)$;
until $FIRST()$, $FOLLOW()$, $nullable()$ stable;
```

[10] *nullable*(S) = *true* if ε is derivable from S.

- **Example:**

$Z \to \texttt{z}$	$Y \to$	$X \to Y$
$Z \to X\ Y\ Z$	$Y \to \texttt{y}$	$X \to \texttt{x}$

we start the algorithm with the initial assumption

	nullable()	*FIRST*()	*FOLLOW*()
X	f		
Y	f		
Z	f		

In the first iteration we find

	nullable()	*FIRST*()	*FOLLOW*()
X	t	x, y	
Y	t	y	z
Z	f	z	

and the third iteration already gives the stable result

	nullable()	*FIRST*()	*FOLLOW*()
X	t	x, y	x, y, z
Y	t	y	x, y, z
Z	f	x, y, z	

- **Example:** for the expression CFG G

$$\begin{array}{lll} S \to E\ \$ & \quad & T \to T * F \\ E \to E + T & & T \to F \\ E \to T & & F \to (\ E\ ) \\ & & F \to \texttt{num} \end{array}$$

we get

	nullable()	*FIRST*()	*FOLLOW*()
S	f	(, `num`	
E	f	(, `num`	+,), $
T	f	(, `num`	*, +,), $
F	f	(, `num`	*, +,), $

which we use to resolve the shift/reduce conflicts in two LR states associated with G:

$$\left[\begin{array}{ll} E \to T\ \bullet & \textit{reduce } E \to T \\ E \to T\ \bullet * F & \textit{shift } 9 \end{array} \right]_{③} \quad \left[\begin{array}{ll} E \to E + T\ \bullet & \textit{reduce } E \to E + T \\ E \to T\ \bullet * F & \textit{shift } 9 \end{array} \right]_{⑪}$$

Since $* \notin \textit{FOLLOW}(E)$ we reduce for any token $\neq *$, but shift for $*$.

- The SLR parsing table algorithm already leads to useful LR parsers for quite a wide range of input CFGs.

- SLR however is not sufficient to parse typical constructs to be found in even moderately complex programming language grammars.
  - Consider the CFG G

$$\begin{array}{llll} {}_0\, S' \to S\ \$ & & {}_3\, E \to V \\ {}_1\, S \to V\ \texttt{=}\ E & & {}_4\, V \to \texttt{id} \\ {}_2\, S \to E & & {}_5\, V \to \texttt{*}\ E \end{array}$$

  - G captures the essence of C-style variable assignments in which a dereference operator `*` may be applied to (pointer-typed) variables. G accepts input token sequences like

```
x = y        *x = y        x = **y
```

– To construct G's SLR parsing table, we compute LR items (only states ① … ③ shown here):

$$\left[\begin{array}{lcll} S' & \rightarrow & \bullet\ S\ \$ & \textit{goto}\ 2 \\ \hline S & \rightarrow & \bullet\ V = E & \textit{goto}\ 3 \\ S & \rightarrow & \bullet\ E & \textit{goto}\ 4 \\ V & \rightarrow & \bullet\ \texttt{id} & \textit{shift}\ 5 \\ V & \rightarrow & \bullet\ *\ E & \textit{shift}\ 6 \\ E & \rightarrow & \bullet\ V & \textit{goto}\ 3 \end{array}\right]_{①}
\qquad
\left[\begin{array}{ll} S' \rightarrow S\ \bullet\ \$ & \textit{accept} \end{array}\right]_{②}
\qquad
\left[\begin{array}{ll} S \rightarrow V\ \bullet\ = E & \textit{shift}\ 7 \\ E \rightarrow V\ \bullet & \textit{reduce}\ E \rightarrow V \end{array}\right]_{③}$$

and the associated *FOLLOW*() sets to resolve any shift/reduce conflicts which might occur:

	nullable()	*FIRST*()	*FOLLOW*()
S	f	`x`, `*`	\$
E	f	`x`, `*`	`=`, \$
V	f	`x`, `*`	`=`, \$

– According to the SLR rule, we resolve the shift/reduce conflict in state ③ by choosing *reduce* $E \rightarrow V$ when we see tokens in $\mathit{FOLLOW}(E) = \{=, \$\}$ in the input.

- The resulting SLR parsing table thus is

CFG symbol

	`id`	`*`	`=`	`$`	S	E	V
1	$s5$	$s6$			2	4	3
2				a			
3	$r\,E \rightarrow V$	$r\,E \rightarrow V$	$r\,E \rightarrow V$	$r\,E \rightarrow V$			
4	$r\,S \rightarrow E$	$r\,S \rightarrow E$	$r\,S \rightarrow E$	$r\,S \rightarrow E$			
5	$r\,V \rightarrow \texttt{id}$	$r\,V \rightarrow \texttt{id}$	$r\,V \rightarrow \texttt{id}$	$r\,V \rightarrow \texttt{id}$			
6						8	3
7						9	3
8	$r\,V \rightarrow * E$	$r\,V \rightarrow * E$	$r\,V \rightarrow * E$	$r\,V \rightarrow * E$			
9	$r\,S \rightarrow V = E$	$r\,S \rightarrow V = E$	$r\,S \rightarrow V = E$	$r\,S \rightarrow V = E$			

– When we feed the input `id = *id` into the resulting LR parser, we observe the following actions:

Stack	*Input*	*Action*
$_1$	`id = * id $`	*shift*
$_1$ `id` $_5$	`= * id $`	*reduce* $V \to$ `id`
$_1 V _3$	`= * id $`	*reduce* $E \to V$
$_1 E _4$	`= * id $`	*reduce* $S \to E$
$_1 S _2$	`= * id $`	*error*

CFG symbol

	`id`	`*`	`=`	`$`	S	E	V
1	$s5$	$s6$			2	4	3
2				a			
3	$r\,E \to V$	$r\,E \to V$	$r\,E \to V$	$r\,E \to V$			
4	$r\,S \to E$	$r\,S \to E$	$r\,S \to E$	$r\,S \to E$			
5	$r\,V \to$ `id`	$r\,V \to$ `id`	$r\,V \to$ `id`	$r\,V \to$ `id`			
6						8	3
7						9	3
8	$r\,V \to * E$	$r\,V \to * E$	$r\,V \to * E$	$r\,V \to * E$			
9	$r\,S \to V = E$	$r\,S \to V = E$	$r\,S \to V = E$	$r\,S \to V = E$			

- The problem here is that

  ① it may make sense to reduce via $E \to V$ when we see a = token in the input (since = $\in$ *FOLLOW*(E)), but

  ② this might not be the correct choice in *all* situations.

- SLR is completely indifferent to differing parsing situations (i.e., different lookahead tokens) since it bases its shift/reduce conflict resolution on the global *FOLLOW*() sets.

- We can improve LR parser table generation considerably if we consider the next **lookahead token** while we are computing LR states.

  – The resulting LR parsing tables are called **LR(1)** (*left-to-right parse, right derivation, 1 lookahead token*).
  – The resulting LR(1) parsing tables contain more states than SLR tables. LR(1) can generate parsers for a strictly larger class of CFGs than SLR.

- An **LR(1) item** takes the form

$$(S \to S_1 \ \cdots \ S_{i-1} \bullet \ S_i \ S_{i+1} \ \cdots \ S_k, x)$$

  - x is the **lookahead token** (the token following the tokens derived from S_k). In this configuration, the parser expects to see input derivable from $S_i \ \cdots \ S_k \ x$.
  - When we compute the closure of an LR(1) item, we add an item for every possible lookahead token.
  - In the configuration above, the possible lookahead tokens are the elements of[11]

$$FIRST(S_{i+1} \ \cdots \ S_k \ x)$$

[11] $FIRST(S_1 \ S_2) = \begin{cases} FIRST(S_1) \cup FIRST(S_2) & \text{if } nullable(S_1) \\ FIRST(S_1) & \text{otherwise.} \end{cases}$

- **Example:** compute closure of LR(1) item $(S' \to \bullet\ S\ \$, ?)$:[12]

$$\left[\begin{array}{llll}
(S' & \to & \bullet\ S\ \$ & , ?) \\ \hline
S & \to & \bullet\ V = E & , \$) \\
S & \to & \bullet\ E & , \$) \\
E & \to & \bullet\ V & , \$) \\
V & \to & \bullet\ \texttt{id} & , \$) \\
V & \to & \bullet\ *\ E & , \$) \\
V & \to & \bullet\ \texttt{id} & , =) \\
V & \to & \bullet\ *\ E & , =)
\end{array}\right] \text{①}$$

  - **Note:** $\mathit{FIRST}(\$\ \ ?) = \{\$\}$, $\mathit{FIRST}(= E\ \$) = \{=\}$.
  - You can find the set of LR(1) items for this grammar on page 66. (Typo: replace $T \to \bullet * E$ by $V \to \bullet * E$ in state ⑥.)
  - When we determine the LR(1) parsing table entries, we set

$$\mathit{goto}[s][\mathtt{x}] = r\ S \to S_1\ \cdots\ S_k$$

for the LR(1) item $(S \to S_1\ \cdots\ S_k\ \bullet, \mathtt{x})$ in state s. Otherwise, table generation is like in the SLR case.

[12]For the start symbol S', the lookahead token is immaterial since we never shift (*look beyond*) \$.

- In the LR(1) case, the former problematic state ③ now looks like

$$\left[\begin{array}{lll} (S \to V \bullet = E & , \$) & \textit{shift}\ 4 \\ (E \to V \bullet & , \$) & \textit{reduce}\ E \to V \end{array}\right]_{③}$$

- With the LR(1) parsing table (also see page 66), we now observe the following LR parsing trace for the input `id = *id`:

Stack	*Input*	*Action*
${}_1$	`id = * id $`	*shift*
${}_1$ `id` ${}_8$	`= * id $`	*reduce* $V \to$ `id`
${}_1\, V\, {}_3$	`= * id $`	*shift*
${}_1\, V\, {}_3 = {}_4$	`* id $`	*shift*
${}_1\, V\, {}_3 = {}_4 * {}_{13}$	`id $`	*shift*
${}_1\, V\, {}_3 = {}_4 * {}_{13}$ `id` ${}_{11}$	`$`	*reduce* $V \to$ `id`
${}_1\, V\, {}_3 = {}_4 * {}_{13}\, V\, {}_7$	`$`	*reduce* $E \to V$
${}_1\, V\, {}_3 = {}_4 * {}_{13}\, E\, {}_{14}$	`$`	*reduce* $V \to * E$
${}_1\, V\, {}_3 = {}_4\, V\, {}_7$	`$`	*reduce* $E \to V$
${}_1\, V\, {}_3 = {}_4\, E\, {}_9$	`$`	*reduce* $S \to V = E$
${}_1\, S_2$	`$`	*accept*

3.4 Parser Generators

3.4 Using Parser Generators, p. 69

- The construction of an LR(1) (or the more compact LALR(1) variant) parsing table from a given CFG is tedious but simple enough to be done automatically.

- **Parser generators** like `yacc`[13], or the GNU project's `bison`

  ① read the description of a CFG G,

  ② internally construct the LALR(1) parsing table for G, and then

  ③ output a table-driven C program (with entry point `yyparse()`) which implements an LALR(1) parser for G, see slide 89.

- How can `yyparse()` return anything useful but `Parse OK` or `Parse Error`?

  – By attaching **semantic actions** to each CFG rule which are executed whenever a *reduce* action for that rule is executed (see next chapter).

[13] Yet Another Compiler Compiler

- GNU `bison` parser skeleton:

`bison` parser skeleton

```
parser declarations
%%
CFG rules and semantic actions
%%
program fragments, helper functions, . . .
```

  - **Example:** the expression CFG

$$\begin{array}{lllllllll} E & \to & \texttt{id} & E & \to & E \texttt{ * } E & E & \to & E \texttt{ + } E \\ E & \to & \texttt{num} & E & \to & E \texttt{ / } E & E & \to & E \texttt{ - } E \\ E & \to & \texttt{(} E \texttt{)} & & & & & & \end{array}$$

may bewritten as shown on the next slide to be able to feed it into `bison`:

bison **expression CFG file** expr.y

```
%token ID NUM LPAREN RPAREN TIMES DIV PLUS MINUS

%%

E : ID
  | NUM
  | LPAREN E RPAREN
  | E TIMES E
  | E DIV E
  | E PLUS E
  | E MINUS E
  ;

%%
```

– bison can write a C header file containing definitions[14] for the declared tokens (%token) of the CFG.

– A lexer can include this header file and then return properly encoded tokens to the parser.

[14]The written C header file includes a series of `#define ID 258`, `#define NUM 259`, ... or a C enum like `enum yytokentype { ID = 258, NUM = 259, ... }`

- Now let us feed `expr.y` into `bison`:

`bison` **invocation**

```
$ bison -d expr.y
expr.y: warning: 16 shift/reduce conflicts
$ █
```

- Remember from slide 82 that the expression grammar was highly ambiguous:
  - We neither incorporated knowledge about **operator precedence** ($*$ binds tighter than $+$, ...),
  - nor made **operator associativity** explicit (arith. operators associate to the left).

- LR(k) table generation fails for any ambiguous CFG: some parser states will have conflicting associated actions.

- We can use `bison` itself to analyze the conflicts:

`bison` **conflict analysis**

```
$ bison --verbose expr.y
expr.y: warning: 16 shift/reduce conflicts
$ more expr.output
```

expr.output

```
State 12 contains 4 shift/reduce conflicts.
State 13 contains 4 shift/reduce conflicts.
State 14 contains 4 shift/reduce conflicts.
State 15 contains 4 shift/reduce conflicts.


Grammar

    0 $accept: E $end

    1 E: ID
    2  | NUM
    3  | LPAREN E RPAREN
    4  | E TIMES E
    5  | E DIV E
    6  | E PLUS E
    7  | E MINUS E

[...]
```

expr.output (continued)

```
state 0
    0 $accept: . E $end

    ID      shift, and go to state 1
    NUM     shift, and go to state 2
    LPAREN  shift, and go to state 3

    E  go to state 4

state 1
    1 E: ID .

    $default  reduce using rule 1 (E)

state 2
    2 E: NUM .

    $default  reduce using rule 2 (E)

state 3
    3 E: LPAREN . E RPAREN

    ID      shift, and go to state 1
    NUM     shift, and go to state 2
    LPAREN  shift, and go to state 3

    E  go to state 5
```

expr.output continued

```
state 12

    4 E: E . TIMES E
    4  | E TIMES E .
    5  | E . DIV E
    6  | E . PLUS E
    7  | E . MINUS E

    TIMES  shift, and go to state 7
    DIV    shift, and go to state 8
    PLUS   shift, and go to state 9
    MINUS  shift, and go to state 10

    TIMES     [reduce using rule 4 (E)]
    DIV       [reduce using rule 4 (E)]
    PLUS      [reduce using rule 4 (E)]
    MINUS     [reduce using rule 4 (E)]
    $default  reduce using rule 4 (E)

```

expr.output continued

```
state 13

    4 E: E . TIMES E
    5  | E . DIV E
    5  | E DIV E .
    6  | E . PLUS E
    7  | E . MINUS E

    TIMES  shift, and go to state 7
    DIV    shift, and go to state 8
    PLUS   shift, and go to state 9
    MINUS  shift, and go to state 10

    TIMES     [reduce using rule 5 (E)]
    DIV       [reduce using rule 5 (E)]
    PLUS      [reduce using rule 5 (E)]
    MINUS     [reduce using rule 5 (E)]
    $default  reduce using rule 5 (E)

```

- We can clearly see the *shift*/*reduce* conflicts and that `bison` has decided to favor the *shift*. This is the `bison` default conflict resolution for ambiguous grammars.
  - Are we happy with `bison`'s default decision in this case?

– Trace the `bison` generated parser on input `num * num + num`:

Stack	*Input*	*Action*
$_0$	`num * num + num $`	*shift*
$_0$ `num` $_2$	`* num + num $`	*r* $E \to$ `num`
$_0\ E\ _4$	`* num + num $`	*shift*
$_0\ E\ _4 * _7$	`num + num $`	*shift*
$_0\ E\ _4 * _7$ `num` $_2$	`+ num $`	*r* $E \to$ `num`
$_0\ E\ _4 * _7\ E\ _{12}$	`+ num $`	*shift* ϟ
$_0\ E\ _4 * _7\ E\ _{12} + _9$	`num $`	*shift*
$_0\ E\ _4 * _7\ E\ _{12} + _9$ `num` $_2$	`$`	*r* $E \to$ `num`
$_0\ E\ _4 * _7\ E\ _{12} + _9\ E\ _{14}$	`$`	*r* $E \to E + E$

$$\left[\begin{array}{ll}
(E \to E \bullet * E & ,?) \\
(E \to E \bullet / E & ,?) \\
(E \to E \bullet + E & ,?) \\
(E \to E \bullet - E & ,?) \\
(E \to E * E \bullet & ,\$) \\
(E \to E * E \bullet & ,)) \\
(E \to E * E \bullet & ,*) \\
(E \to E * E \bullet & ,/) \\
(E \to E * E \bullet & ,+) \\
(E \to E * E \bullet & ,-)
\end{array}\right]_{⑫}$$

- For our example input of `num * num + num` the following two LR items of state 12 comprise the *shift*/*reduce* dilemma:

$$\left[\begin{array}{lllll} & & \vdots & & \\ (E \rightarrow E & & \bullet & + E & ,?) \\ (E \rightarrow E * E & & \bullet & & ,+) \\ & & \vdots & & \end{array}\right]_{⑫}$$

  - Here we want to choose the *reduce* because this interprets `*` as an operator with higher precedence as `+`.
  - So, to sensibly resolve such ambiguity, for an LR item of the form

$$\begin{array}{c} (E \rightarrow E \;\bullet\; \odot\; E, ?) \\ (E \rightarrow E \;\odot\; E \;\bullet, \otimes) \end{array}$$

    choose

    ① *reduce*, if the precedence of $\odot$ is higher than $\otimes$,

    ② *shift*, if the precedence is lower.

- For an LR item of the form

$$\begin{array}{c}(E \to E \ \bullet \ \odot \ E, ?) \\ (E \to E \ \odot \ E \ \bullet, \odot)\end{array}$$

  choose

  ① $\boxed{\phantom{\text{reduce}}}$, if operator $\odot$ is left-associative,

  ② $\boxed{\phantom{\text{reduce}}}$, if operator $\odot$ is right-associative,

  ③ $\boxed{\phantom{\text{reduce}}}$, if operator $\odot$ is not associative at all.

- In `bison`, the declarations `%left`, `%right`, and `%nonassoc` assign **associativity** to the specified token.
  The order in which these declarations appear in the `bison` input file specifies **precedence** (lower precedence comes first).

- The precedence of a rule is the precedence of the **rightmost** token appearing in it[15].

[15] This may be overridden by `bison`'s `%prec` directive. See **p. 74** and the `bison` manual.

- The resulting modified `bison` input file:

`bison` **expression CFG file** `expr.y`**, modified**

```
%token ID NUM LPAREN RPAREN TIMES DIV PLUS MINUS

%left PLUS MINUS
%left TIMES DIV

%%

E : ID
  | NUM
  | LPAREN E RPAREN
  | E TIMES E
  | E DIV E
  | E PLUS E
  | E MINUS E
  ;

%%
```

  – `bison` generates an LALR(1) parsing table for this grammar without any ambiguity problems.

3.5 Syntax vs. Semantics

- There still remain decisions to be made during program analysis for which *no* parsing technique whatsoever will be sufficiently clever.
  - **Example:** in a small language, we have *boolean* (operators `and`, `or`, `=`) and *arithmetic* expressions (operators `+`, `*`). Arithmetic operators bind tighter than boolean operators, such that

    ```
    x := a + b = c
    ```

    is parsed as `x := (a + b) = c`.
  - The `bison` grammar tries to separate boolean from arithmetic expressions as shown on the next slide. In some sense, the CFG tries to implement a bit of **type checking** in order to prevent acceptance of input like

    ```
    x := a + b and c
    ```

arith-bool.y

```
%token ID ASSIGN PLUS TIMES AND OR EQUAL

%left OR
%left AND
%left EQUAL
%left PLUS
%left TIMES

%%

stm   : ID ASSIGN ae
      | ID ASSIGN be
      ;

ae    : ae PLUS ae
      | ae TIMES ae
      | ID
      ;

be    : be OR be
      | be AND be
      | be EQUAL be
      | ae EQUAL ae
      | ID
      ;
```

– `bison` reports a *reduce/reduce* conflict for this CFG. Where/Why?

- For input such as

```
x := a
```

  there is no way to tell for the parser if the variable `a` is an arithmetic or a boolean variable (reduce `ae` $\rightarrow$ `ID` or reduce `be` $\rightarrow$ `ID`?).

- If there is **no syntactical criterion** to support a parsing decision, then either

  ① the grammar is severely malformed (fix the CFG!), or

  ② the decision has to be **postponed until semantical analysis**.

- Remark on ②: make the CFG more permissive, then reject type incorrect input such as

```
x := a + b and c
```

  during **semantical analysis**.

arith-bool.y **(modified)**

```
%token ID ASSIGN PLUS TIMES AND OR EQUAL

%left OR
%left AND
%left EQUAL
%left PLUS
%left TIMES

%%

stm   : ID ASSIGN e
      ;

e     : e PLUS e
      | e TIMES e
      | e AND e
      | e OR e
      | e EQUAL e
      | ID
      ;

%%
```