Compiler Construction

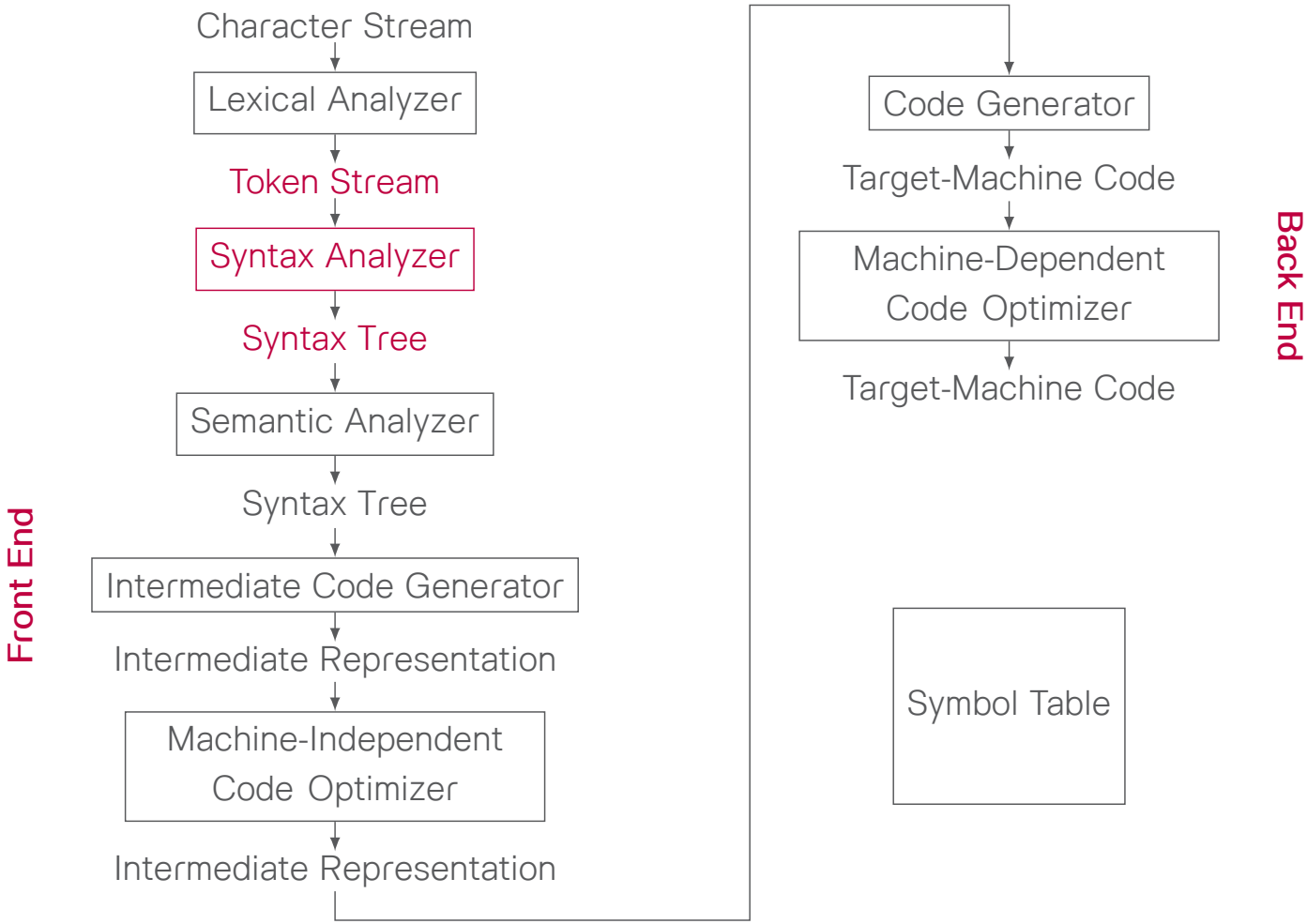
Chapter 3: Parser

Dittaya Wanvarie

Department of Mathematics and Computer Science
Chulalongkorn University

Second semester, 2024

Overview of compilation



Outline

Introduction

Parser

- Determine whether the input is valid (correct) according to the syntax (grammar)
- Report error and return diagnostic information to the user

Parser implementation

- Recursive descent parser - a hand-coded top-down parser
- LL(1) grammar with a lookahead parser - a table-driven top-down parser
- LR(1) grammar with a lookahead parser and a shift-reduce parser - a table-driven bottom-up parser

Why not regular expression?

Consider the following example: Recognizing an algebraic expression e.g. $a + b \times c$ and $e \div f \times g$

$$[a \ldots z]([a \ldots z]|[0 \ldots 9])^*((+|-|\times|\div)[a \ldots z]([a \ldots z]|[0 \ldots 9])^*)^*$$

- The RE matches both expressions
- However, there is no precedence enforcement

Consider the following example: matching parenthesis

- RE cannot check incorrectly matched parentheses, brackets

Context-Free Grammar

- Regular grammar cannot represent some constructs such as parentheses and blocks
- We have an efficient algorithm to parse a string in a context-free language
  - Cubic time to the size of the input program
  - Linear time to the size of the input in a subclass of context-free languages

Outline

Terminology

$$G = (V, \Sigma, R, S)$$

- **CFG**, G, is a set of production rules that defines a set of strings in the language.
- A **terminal symbol** is a token name (syntactic category).
  - Σ is the set of terminal symbols in G.
- A **sentence** is a string of terminal symbols that can be derived from the rules of a grammar.
- A **nonterminal symbol** is a variable in a production.
  - V is the set of variables in G.
- A **production** is a mapping function from a nonterminal symbol (in V) into a sequence of symbols (in $(V \cup \Sigma)^*$).
  - R is the set of production rules in G.
- S is the start variable.

Backus-Naur form

From the following CFG,

$$
\begin{aligned}
G &= (V, \Sigma, R, S) \\
V &= \{B\} \\
\Sigma &= \{(,)\} \\
R &= \{B \to (B), B \to ()B\} \\
S &= B
\end{aligned}
$$

We can write the grammar in its BNF as follow

```
<B> ::= ( <B> )
      |   ( ) <B>
```

- Variables are wrapped in angle brackets.

Terminology (Cont.)

- A **derivation** is a sequence of rewriting steps that begins with the grammar's start symbol and ends with a sentence in a language.
  - If we can find a valid derivation to the sentence, that sentence is in the language.
- A **sentence** is a string in a language, i.e. a sequence of nonterminal symbols in a language
- A **sentential form** is a string of symbols that occurs at one step in a valid derivation.
  - Any sentential form is in $(V \cup \Sigma)^*$
- A **parse tree** or a **syntax tree** is a graph that represents a derivation.
- A **rightmost derivation** is a derivation that rewrites nonterminal symbols from *right to left*.
- A **leftmost derivation** is a derivation the rewrites nonterminal symbols from *left to right*.

Notations

- Variables are in *italic* and usually begin with capital letters, e.g. $Expr$.
- Terminal symbols are written in **bold lowercase**, e.g. **id**. They are token types from the scanner.
- When we write only production rules, the variable on the **left hand side of the first rule** is the start variable.

Example grammar

1	*SheepNoise*	→	`baa` *SheepNoise*
2		|	`baa`

- Nonterminal symbol: *SheepNoise*
- Terminal symbols: **baa**
- Sentences: **baa**, **baa baa**
- Start symbol: *SheepNoise*

Derivation example: *SheepNoise* $\rightarrow^+$ **baa** and *SheepNoise* $\rightarrow^+$ **baa baa**

Rule	Sentential Form
	SheepNoise
2	`baa`

Rewrite with Rule 2

Rule	Sentential Form
	SheepNoise
1	`baa` *SheepNoise*
2	`baa baa`

Rewrite with Rule 1 Then 2

Complex example

1	*Expr*	→	(*Expr*)
2		|	*Expr Op Expr*
3		|	name

4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

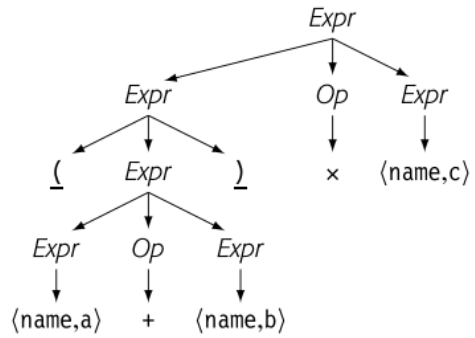
- Sentence: **a** + **b** × **c**
- Tokenized sentence: $\langle \textbf{name}, a \rangle + \langle \textbf{name}, b \rangle \times \langle \textbf{name}, c \rangle$

Precedence handling?

Complex example: derivations

Rule	Sentential Form
	Expr
2	*Expr Op Expr*
3	*Expr Op* name
6	*Expr* × name
1	(*Expr*) × name
2	(*Expr Op Expr*) × name
3	(*Expr Op* name) × name
4	(*Expr* + name) × name
3	(name + name) × name

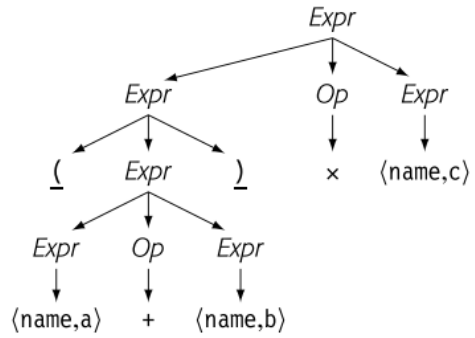
(a) Rightmost Derivation of (a + b) × c



(b) Corresponding Rightmost Parse Tree

Rule	Sentential Form
	Expr
2	*Expr Op Expr*
1	(*Expr*) *Op Expr*
2	(*Expr Op Expr*) *Op Expr*
3	(name *Op Expr*) *Op Expr*
4	(name + *Expr*) *Op Expr*
3	(name + name) *Op Expr*
6	(name + name) x *Expr*
3	(name + name) x name

(c) Leftmost Derivation of (a + b) × c



(d) Corresponding Leftmost Parse Tree

■ **FIGURE 3.1** Derivations of (a + b) × c.

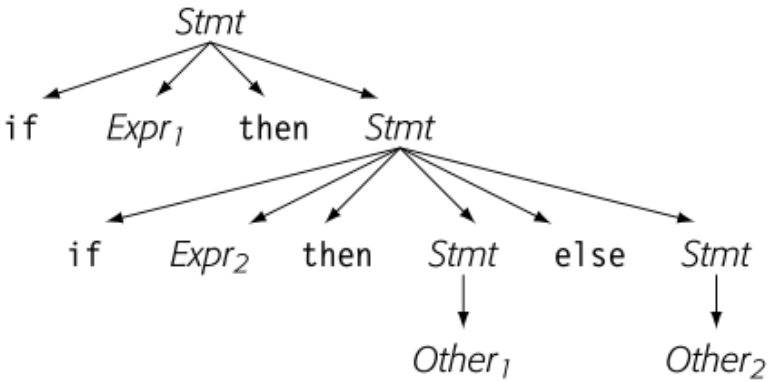
Ambiguous grammars

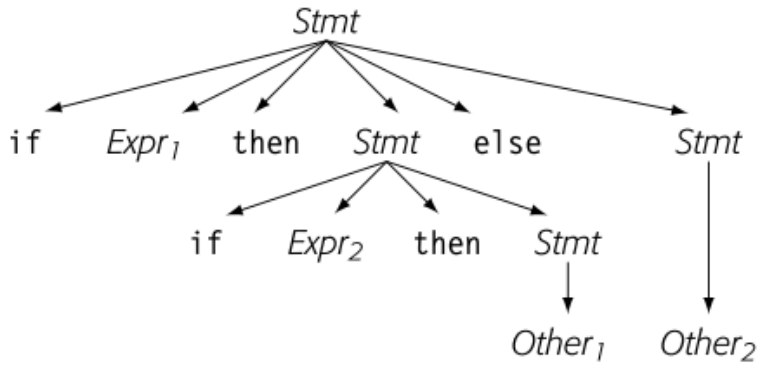
- A grammar is ambiguous if some strings in the language of that grammar have **more than one** rightmost (or leftmost) derivation trees.
- Either the leftmost or the rightmost derivation tree should be **identical** for any string in an **unambiguous** grammar.

Example of ambiguous grammar

1	*Stmt*	→	`if` *Expr* `then` *Stmt*
2		|	`if` *Expr* `then` *Stmt* `else` *Stmt*
3		|	*Other*

Sentence: **if** $Expr_1$ **then if** $Expr_2$ **then** $Other_1$ **else** $Other_2$





Question:

- If we enclose an if statement in a block, e.g. { and }, will it solve the ambiguity problem?
- If we put an end-of-statement symbol, e.g. a semicolon, will it solve the ambiguity problem?

If statement

Rewrite the grammar: there are 2 choices in the **then** part

1	*Stmt*	→	`if` *Expr* `then` *Stmt*
2		|	`if` *Expr* `then` *WithElse* `else` *Stmt*
3		|	*Other*
4	*WithElse*	→	`if` *Expr* `then` *WithElse* `else` *WithElse*
5		|	*Other*

- If there is an **else**, the next *Stmt* should match with the innermost **else**

Sentence: **if** $Expr_1$ **then if** $Expr_2$ **then** $Other_1$ **else** $Other_2$

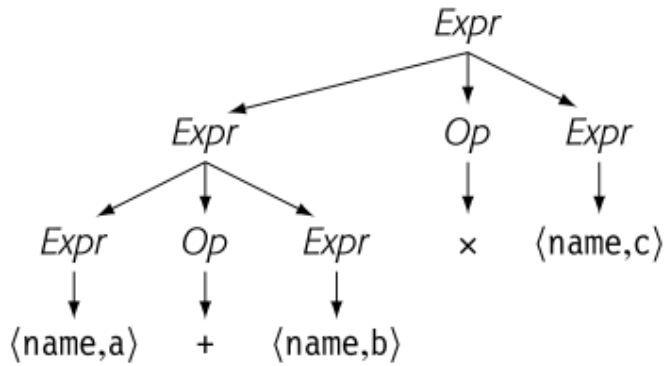
Rule	Sentential Form
	Stmt
1	`if` *Expr* `then` *Stmt*
2	`if` *Expr* `then if` *Expr* `then` *WithElse* `else` *Stmt*
3	`if` *Expr* `then if` *Expr* `then` *WithElse* `else` *Other*
5	`if` *Expr* `then if` *Expr* `then` *Other* `else` *Other*

Encoding meaning into structure

Sentence: **a** + **b** × **c**

Rule	Sentential Form
	Expr
2	*Expr Op Expr*
3	*Expr Op* name
6	*Expr* × name
2	*Expr Op Expr* × name
3	*Expr Op* name × name
4	*Expr* + name × name
3	name + name × name

Derivation of a + b × c



Corresponding Parse Tree

To evaluate this expression, we traverse the tree on **postorder**, i.e. operands-then-operator)

- However, this tree contradicts the classic rules of algebraic precedence

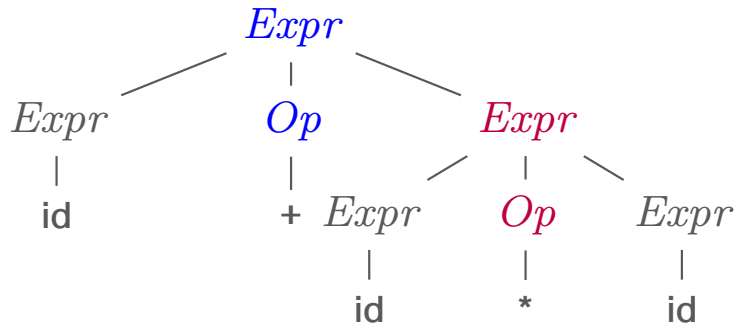
Handling precedence of operators

No precedence

$$
\begin{array}{lcl}
Expr & \rightarrow & (\ Expr\ ) \\
 & | & Expr\ Op\ Expr \\
 & | & \mathsf{id} \\
Op & \rightarrow & \mathsf{+} \\
 & | & \mathsf{-} \\
 & | & \mathsf{*} \\
 & | & \mathsf{/}
\end{array}
$$

Sentence: id + id * id

$$
\begin{array}{rcl}
Expr & \rightarrow & \underline{Expr}\ Op\ \underline{Expr} \\
 & \rightarrow & Expr\ Op\ Expr\ Op\ \underline{Expr} \\
 & \rightarrow & Expr\ Op\ Expr\ \underline{Op}\ \mathsf{id} \\
 & \rightarrow & Expr\ Op\ \underline{Expr}\ \mathsf{*}\ \mathsf{id} \\
 & \rightarrow & Expr\ \underline{Op}\ \mathsf{id}\ \mathsf{*}\ \mathsf{id} \\
 & \rightarrow & \underline{Expr}\ \mathsf{+}\ \mathsf{id}\ \mathsf{*}\ \mathsf{id} \\
 & \rightarrow & \mathsf{id}\ \mathsf{+}\ \mathsf{id}\ \mathsf{*}\ \mathsf{id}
\end{array}
$$



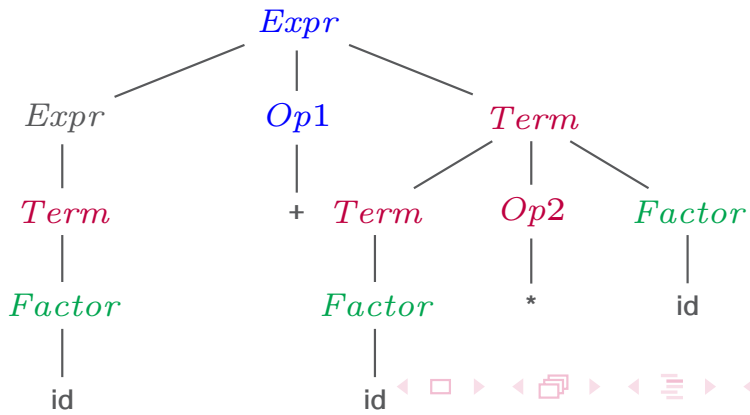
Handling precedence of operators

With precedence

$$
\begin{array}{lcl}
Expr & \rightarrow & Expr\ Op1\ Term \\
 & | & Term \\
Term & \rightarrow & Term\ Op2\ Factor \\
 & | & Factor \\
Factor & \rightarrow & \mathsf{id} \\
 & | & (\ Expr\ ) \\
Op1 & \rightarrow & \mathsf{+} \\
 & | & \mathsf{-} \\
Op2 & \rightarrow & \mathsf{*} \\
 & | & \mathsf{/}
\end{array}
$$

Sentence: id + id * id

$$
\begin{array}{rcl}
Expr & \rightarrow & \underline{Expr\ Op1\ Term} \\
 & \rightarrow & Expr\ Op1\ Term\ \underline{Op2\ Factor} \\
 & \rightarrow & Expr\ Op1\ \underline{Term\ Op2}\ \mathsf{id} \\
 & \rightarrow & Expr\ \underline{Op1\ Term}\ \mathsf{*}\ \mathsf{id} \\
 & \rightarrow & \underline{Expr\ Op1\ Factor}\ \mathsf{*}\ \mathsf{id} \\
 & \rightarrow & \underline{Expr\ Op1}\ \mathsf{id}\ \mathsf{*}\ \mathsf{id} \\
 & \rightarrow & \underline{Expr}\ \mathsf{+}\ \mathsf{id}\ \mathsf{*}\ \mathsf{id} \\
 & \rightarrow & \underline{Term}\ \mathsf{+}\ \mathsf{id}\ \mathsf{*}\ \mathsf{id} \\
 & \rightarrow & \underline{Factor}\ \mathsf{+}\ \mathsf{id}\ \mathsf{*}\ \mathsf{id} \\
 & \rightarrow & \mathsf{id}\ \mathsf{+}\ \mathsf{id}\ \mathsf{*}\ \mathsf{id}
\end{array}
$$



Handling precedence of operators

- Each level of precedence has its own variable
- Productions for variables with low precedence should start with the same or higher level variables.
  - The start variable has the lowest precedence.
  - Terminal symbols have the highest precedence.

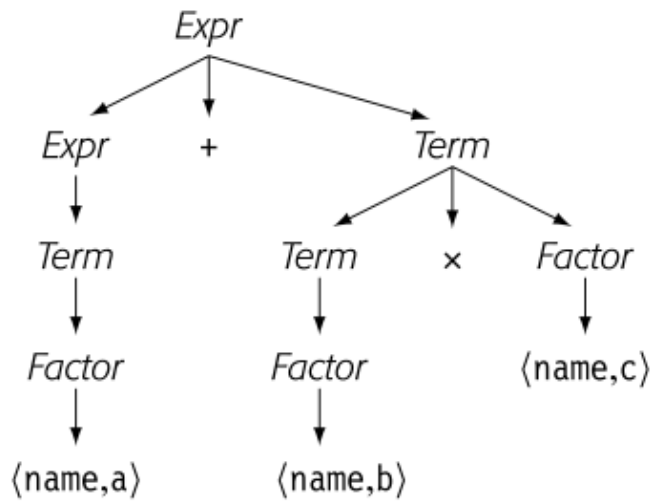
Expression grammar

0	*Goal*	→	*Expr*	5		|	*Term* ÷ *Factor*
1	*Expr*	→	*Expr* + *Term*	6		|	*Factor*
2		|	*Expr* - *Term*	7	*Factor*	→	(*Expr*)
3		|	*Term*	8		|	num
4	*Term*	→	*Term* × *Factor*	9		|	name

■ **FIGURE 3.2** The Classic Expression Grammar.

Rule	Sentential Form
0	*Expr*
1	*Expr* + *Term*
4	*Expr* + *Term* × *Factor*
9	*Expr* + *Term* × name
6	*Expr* + *Factor* × name
9	*Expr* + name × name
3	*Term* + name × name
6	*Factor* + name × name
9	name + name × name

Rightmost Derivation of a + b × c

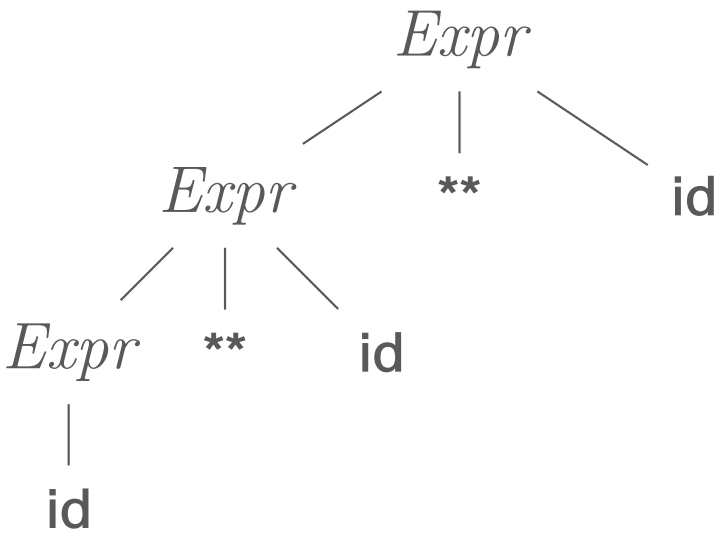


Corresponding Parse Tree

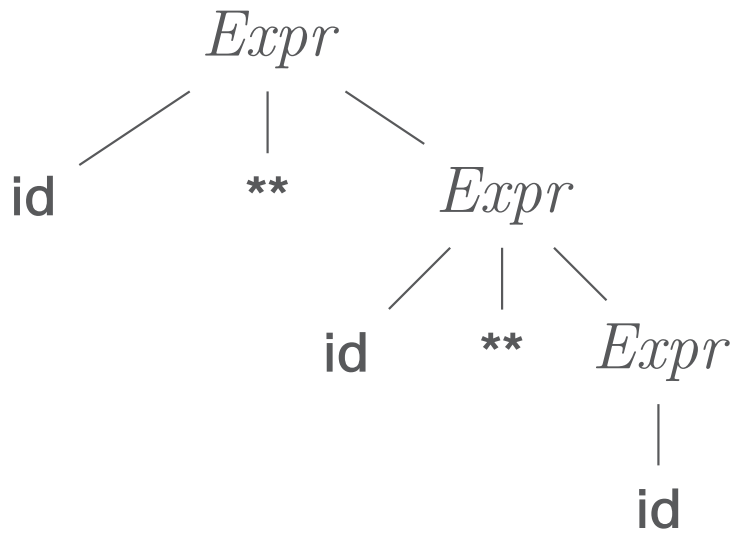
Handling Associativity of Operators

Sentence: **id ** id ** id**

$$Expr \rightarrow Expr \texttt{ ** id} \mid \texttt{id}$$



$$Expr \rightarrow \texttt{id ** } Expr \mid \texttt{id}$$



Outline

Parsing algorithms

A standard context-free grammar can be parsed in $O(n^3)$ of its input size.

- Earley's parsing algorithm
- CYK algorithm

We need a subset of languages that could be parsed in $O(n)$ of the input size.

- LL(1)
- LR(1)

Outline

Top-down parsing

Since we read the input from left to right, let's use the **leftmost** derivation.

```
root ← node for the start symbol, S
focus ← root
push null onto the stack
word ← NextWord()

while (true) do
    if (focus is a nonterminal) then
        pick next rule to expand focus, say A → β1 β2 β3 ... βn
        build nodes for β1 β2 β3 ... βn as children of focus
        push βn, βn-1, βn-2, ... β2 onto the stack
        focus ← β1
    else if (word matches focus) then
        word ← NextWord()
        focus ← pop from the stack
    else if (word = eof and focus = null)
        then accept the input and return root
    else backtrack
```

■ **FIGURE 3.3** A Leftmost, Top-Down Parsing Algorithm.

Top-Down Parsing

0	*Goal*	→	*Expr*	5		|	*Term* ÷ *Factor*
1	*Expr*	→	*Expr* + *Term*	6		|	*Factor*
2		|	*Expr* - *Term*	7	*Factor*	→	(*Expr*)
3		|	*Term*	8		|	num
4	*Term*	→	*Term* × *Factor*	9		|	name

■ **FIGURE 3.2** The Classic Expression Grammar.

Rule	Sentential Form	Input
	Expr	↑ name + name × name
1	*Expr* + *Term*	↑ name + name × name
3	*Term* + *Term*	↑ name + name × name
6	*Factor* + *Term*	↑ name + name × name
9	name + *Term*	↑ name + name × name
→	name + *Term*	name ↑ + name × name
→	name + *Term*	name + ↑ name × name
4	name + *Term* × *Factor*	name + ↑ name × name
6	name + *Factor* × *Factor*	name + ↑ name × name
9	name + name × *Factor*	name + ↑ name × name
→	name + name × *Factor*	name + name ↑ × name
→	name + name × *Factor*	name + name × ↑ name
9	name + name × name	name + name × ↑ name
→	name + name × name	name + name × name ↑

■ **FIGURE 3.4** Leftmost, Top-Down Parse of a + b × c with Oracular Choice.

Multiple choices and inconsistent selection

- Step 2: *Expr*, Rule 1 or 2 or 3
- Step 3: *Expr*, Rule 1 or 2 or 3
- ...

Left recursion

Bad selection may lead to infinite recursion

Rule	Sentential Form	Input
	Expr	↑ name + name × name
1	*Expr* + *Term*	↑ name + name × name
1	*Expr* + *Term* + *Term*	↑ name + name × name
1	...	↑ name + name × name

In a **left**most derivation, **left** recursion leads to an infinite loop.

Eliminating (direct) left recursion

Transform a left recursive grammar into a right recursive grammar.

$$
\begin{array}{lcl}
Fee & \rightarrow & Fee\ \alpha \\
 & | & \beta
\end{array}
$$

$$
\begin{array}{lcl}
Fee & \rightarrow & \beta\ Fee' \\
Fee' & \rightarrow & \alpha\ Fee' \\
 & | & \epsilon
\end{array}
$$

Transformed expression grammar

Original Grammar			Transformed Grammar		
Expr	→	*Expr* + *Term*	*Expr*	→	*Term Expr′*
	|	*Expr* - *Term*	*Expr′*	→	+ *Term Expr′*
	|	*Term*		|	- *Term Expr′*
				|	ϵ
Term	→	*Term* × *Factor*	*Term*	→	*Factor Term′*
	|	*Term* ÷ *Factor*	*Term′*	→	× *Factor Term′*
	|	*Factor*		|	÷ *Factor Term′*
				|	ϵ

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr′*
2	*Expr′*	→	+ *Term Expr′*
3		|	- *Term Expr′*
4		|	ϵ
5	*Term*	→	*Factor Term′*

6	*Term′*	→	× *Factor Term′*
7		|	÷ *Factor Term′*
8		|	ϵ
9	*Factor*	→	(*Expr*)
10		|	num
11		|	name

■ **FIGURE 3.5** Right-Recursive Variant of the Classic Expression Grammar.

Left recursion elimination algorithm

For any direct left recursion

- We rewrite the rule with its right recursion transformation

For any indirect left recursion

impose an order on the nonterminals, $A_0, A_1, \ldots, A_i$

for $i \leftarrow 0$ *to* n *do*

  for $s \leftarrow 0$ *to* $i\text{-}1$ *do*

    replace each production $A_i \rightarrow A_s\,\gamma$ *with* $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,
      where $A_i \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ *are the current productions for* A_i

  eliminate any direct left recursion on A_i *using the transformation*

■ **FIGURE 3.6** Algorithm for Removal of Indirect Left Recursion.

Left recursion elimination example

0	$Goal_0$	$\rightarrow$	A_1
1	A_1	$\rightarrow$	B_2 a
2		|	a
3	B_2	$\rightarrow$	A_1 b

Original Grammar

0	$Goal_0$	$\rightarrow$	A_1
1	A_1	$\rightarrow$	B_2 a
2		|	a
3	B_2	$\rightarrow$	a b C_3
4	C_3	$\rightarrow$	a b C_3
5		|	ϵ

Transformed Grammar

Left recursion elimination example

Consider the following grammar

$$\begin{aligned} S &\rightarrow A\,\mathbf{a} \mid \mathbf{b} \\ A &\rightarrow A\,\mathbf{c} \mid S\,\mathbf{d} \mid \epsilon \end{aligned}$$

- A has one direct recursive rule and one indirect left recursive rule
- S has one indirect recursive rule

General rules for eliminating left recursion

- Construct a subset of grammar with no left recursion.
- Add a new production rule to the set and eliminating left recursion caused by the newly added rule.

Parsing steps

When the top of stack is

- A nonterminal symbol, **pick a rule**
- A terminal symbol, match with the input
- An epsilon, pop the symbol from the stack without matching any input

When it leads to a dead end

1. We may try picking a different rule ← backtracking
2. If there is no other possible rule, report an error

We can avoid backtracking if we choose the correct rule from the first try

Left Factoring

When using a predictive parser, two production rules should not have common prefix.

$$\begin{array}{lcl} Stmt & \rightarrow & \textbf{if } Expr \textbf{ then } Stmt \textbf{ else } Stmt \\ & | & \textbf{if } Expr \textbf{ then } Stmt \end{array}$$

We cannot choose the correct production rule until we see the **else** token. The common prefix is $\underline{\textbf{if } Expr \textbf{ then } Stmt}$.

By left factoring, a new grammar is

$$\begin{array}{lcl} Stmt & \rightarrow & \textbf{if } Expr \textbf{ then } Stmt\ S' \\ S' & \rightarrow & \textbf{else } Stmt \mid \epsilon \end{array}$$

Backtrack-free grammar

For each input character, the parser must always predict the correct rule.

- This parser is a **predictive** parser.
- The grammar is called a **predictive** grammar.
- E.g. LL(1) parsers

Causes of backtracking

- The grammar is left-recursive but we are using a left-to-right top-down parsing.
- Common prefix

FIRST set

To make a predictive parser, we must determine the possible set of rules given the current variable and the input symbol.

```
for each α ∈ (T ∪ eof ∪ ε) do
    FIRST(α) ← α

for each A ∈ NT do
    FIRST(A) ← ∅

while (FIRST sets are still changing) do

    for each p ∈ P, of the form A → β1 β2 ... βk do
        rhs ← FIRST(β1) − {ε}
        trailing ← true

        for i ← 1 to k-1 do
            if ε ∈ FIRST(βi)
                then rhs ← rhs ∪ (FIRST(βi+1) − {ε})
                else
                    trailing ← false
                    break

        if trailing and ε ∈ FIRST(βk) then
            rhs ← rhs ∪ {ε}

        FIRST(A) ← FIRST(A) ∪ rhs
```

■ **FIGURE 3.7** Computing FIRST Sets for Symbols in a Grammar.

$FIRST$ set

- $FIRST(\alpha)$ is the set of **terminal** symbols that can appear **at the start** of the word derived from α.

	num	name	+	-	×	÷	(	)	eof	ϵ
FIRST	num	name	+	-	×	÷	(	)	eof	ϵ

	Productions			FIRST Set
9	*Factor*	→	(*Expr*)	{ (}
10		|	num	{ num }
11		|	name	{ name }

	Expr	*Expr'*	*Term*	*Term'*	*Factor*
FIRST	(, name, num	+, -, ϵ	(, name, num	×, ÷ , ϵ	(, name, num

	Productions			FIRST Set
6	*Term'*	→	× *Factor Term'*	{ × }
7		|	÷ *Factor Term'*	{ ÷ }
8		|	ϵ	{ ϵ }

For a variable A, we can choose the rule $A \longrightarrow \alpha$ only when

- The current top of stack is A
- The current input is in $FIRST(\alpha)$

When should we use the ϵ-production?

With ϵ-production

Given the current input **a** and the current variable A, we will choose the ϵ-production when

- **a** is not in $FIRST(A)$
- **a** can be found immediate right after A, or **a** is in $FOLLOW(A)$

```
for each A ∈ NT do
    FOLLOW(A) ← ∅
FOLLOW(S) ← { eof }
while (FOLLOW sets are still changing) do
    for each p ∈ P of the form A → β1 β2 β3 ... βk do
        TRAILER ← FOLLOW(A)
        for i ← k down to 1 do
            if βi ∈ NT then
                FOLLOW(βi) ← FOLLOW(βi) ∪ TRAILER
                if ϵ ∈ FIRST(βi) then
                    TRAILER ← TRAILER ∪ (FIRST(βi) − ϵ)
                else TRAILER ← FIRST(βi)
            else TRAILER ← { βi }  // βi ∈ T
```

FIGURE 3.8 Computing FOLLOW Sets for Nonterminal Symbols.

$FOLLOW$ set

- $FOLLOW(A)$ is the set of **terminal** symbols that can appear immediate right **after** the derivation of A.

	Expr	*Expr'*	*Term*	*Term'*	*Factor*
FOLLOW	eof,)	eof,)	eof, +, -,)	eof, +, -,)	eof, +, -, ×, ÷,)

$START$ set

Set of possible starting terminal symbols of the variable

- $FIRST$ set if there is no ϵ-production for the variable
- $FIRST$ set except ϵ and $FOLLOW$ set, otherwise

	Production		START Set
Goal	→	*Expr*	{ (, name, num }
Expr	→	*Term Expr'*	{ (, name, num }
Expr'	→	+ *Term Expr'*	{ + }
	|	- *Term Expr'*	{ - }
	|	ϵ	{ eof,) }
Term	→	*Factor Term'*	{ (, name, num }
Term'	→	× *Factor Term'*	{ × }
	|	÷ *Factor Term'*	{ ÷ }
	|	ϵ	{ eof, +, -,) }
Factor	→	(*Expr*)	{ (}
	|	num	{ num }
	|	name	{ name }

■ **FIGURE 3.9** START Sets for the Right-Recursive Expression Grammar.

Left factoring

11	*Factor*	→	`name`
12		|	`name` [*ArgList*]
13		|	`name` (*ArgList*)
15	*ArgList*	→	*Expr MoreArgs*
16	*MoreArgs*	→	, *Expr MoreArgs*
17		|	ϵ

11	*Factor*	→	`name` *Arguments*
12	*Arguments*	→	[*ArgList*]
13		|	(*ArgList*)
14		|	ϵ

Common prefix is one cause of backtracking

- If two rules share some symbols in their $START$, these rules have common prefix
- In other words, all rules should have disjoint $START$ set should be disjoint

Outline

Recursive-descent parser

- One function for one variable
- Choose production rule
  - Simple: try applying rule in order and backtrack if failed
  - Backtrack-free: conditioned on the $START$ set

Example: `TermPrime()` function

	Production			START Set
6	*Term′*	→	× *Factor Term′*	{ × }
7		|	÷ *Factor Term′*	{ ÷ }
8		|	ϵ	{ eof, +, -,) }

Expression grammar

```
Main()  /* Goal → Expr */
    word ← NextWord();
    if (Expr())
        then if (word = eof)
            then report success;
            else Fail();

Fail()
    report syntax error;
    attempt error recovery or exit;

Expr()  /* Expr → Term Expr' */
    if (Term())
        then return ExprPrime();
        else Fail();

ExprPrime()
    /* Expr' → + Term Expr' | - Term Expr' */
    if (word = + or word = -) then
        word ← NextWord();
        if (Term())
            then return ExprPrime();
            else Fail();
    /* Expr' → ϵ */
    else if (word = ) or word = eof)
        then return true;
        else Fail();

Term()  /* Term → Factor Term' */
    if (Factor())
        then return TermPrime();
        else Fail();
```

```
TermPrime()
    /* Term' → × Factor Term' | ÷ Factor Term' */
    if (word = × or word = ÷) then
        word ← NextWord();
        if (Factor())
            then return TermPrime();
            else Fail();
    /* Term' → ϵ */
    else if (word = + or word = - or
             word = ) or word = eof)
        then return true;
        else Fail();

Factor()
    /* Factor → ( Expr ) */
    if (word = ( ) then
        word ← NextWord();
        if (not Expr())
            then Fail();
        if (word ≠ ) )
            then Fail();
        word ← NextWord();
        return true;
    /* Factor → num | name */
    else if (word = num or
            word = name) then
            word ← NextWord();
            return true;

    else Fail();
```

■ **FIGURE 3.10** Recursive-Descent Parser for Expressions.

Outline

LL(1) grammars

A grammar G is LL(1) if and only if whenever

- $A \rightarrow \alpha$ and

$A \rightarrow \beta$ are two distinct productions of G and the following conditions hold

- $START(\alpha)$ and $START(\beta)$ are disjoint.

LL(k) grammar can be parsed by a predictive parser with a Left-to-right, Leftmost derivation, with k symbols if lookahead.

Skeleton of an LL(1) parser

```
word ← NextWord()
push eof onto Stack
push the start symbol, S, onto Stack

while( true ) do
    focus ← top of Stack
    if (focus = eof and word = eof)
        then report success and break from the loop
    else if (focus ∈ T or focus = eof) then
        if focus matches word then
            pop Stack
            word ← NextWord()
        else report an error looking for the symbol in focus
    else    // focus is a nonterminal
        if Table[focus, word] is A → β1 β2 ... βk then
            pop Stack
            for i ← k to 1 by -1 do
                if (βi ≠ ε) then
                    push βi onto Stack
        else report an error expanding focus
```

■ **FIGURE 3.11** The Skeleton LL(1) Parser.

Table-driven parser

- Recursive-descent parser is hand written
- We may automate parser generation using a parsing table
  - Replace the conditions for rules in a recursive-descent parsing with a parse table, which is a mapping between start symbol and a rule choice

Example of LL(1) parse table

	eof	+	-	×	÷	(	)	num	name
Goal						0		0	0
Expr						1		1	1
Expr′	4	2	3				4		
Term						5		5	5
Term′	8	8	8	6	7		8		
Factor						9		10	11

■ **FIGURE 3.12** LL(1) Parse Table for the Right-Recursive Expression Grammar.

Example LL(1) parsing

Rule	Stack	Input
—	eof *Goal*	↑ name + name × name
0	eof *Expr*	↑ name + name × name
1	eof *Expr′ Term*	↑ name + name × name
5	eof *Expr′ Term′ Factor*	↑ name + name × name
11	eof *Expr′ Term′* name	↑ name + name × name
→	eof *Expr′ Term′*	name ↑ + name × name
8	eof *Expr′*	name ↑ + name × name
2	eof *Expr′ Term* +	name ↑ + name × name
→	eof *Expr′ Term*	name + ↑ name × name
5	eof *Expr′ Term′ Factor*	name + ↑ name × name
11	eof *Expr′ Term′* name	name + ↑ name × name
→	eof *Expr′ Term′*	name + name ↑ × name
6	eof *Expr′ Term′ Factor* ×	name + name ↑ × name
→	eof *Expr′ Term′ Factor*	name + name × ↑ name
11	eof *Expr′ Term′* name	name + name × ↑ name
→	eof *Expr′ Term′*	name + name × name ↑
8	eof *Expr′*	name + name × name ↑
4	eof	name + name × name ↑

(a) Actions of the LL(1) Parser on a + b × c

Rule	Stack	Input
—	eof *Goal*	↑ name + ÷ name
0	eof *Expr*	↑ name + ÷ name
1	eof *Expr′ Term*	↑ name + ÷ name
5	eof *Expr′ Term′ Factor*	↑ name + ÷ name
11	eof *Expr′ Term′* name	↑ name + ÷ name
→	eof *Expr′ Term′*	name ↑ + ÷ name
8	eof *Expr′*	name ↑ + ÷ name
2	eof *Expr′ Term* +	name ↑ + ÷ name
→	eof *Expr′* *Term*	name + ↑ ÷ name

syntax error at this point

(b) Actions of the LL(1) Parser on x + ÷ y

■ **FIGURE 3.13** Example LL(1) Parses.

Parse table construction

```
for each nonterminal A do
    Table[A,eof] ← error
    for each terminal w do
        Table[A,w] ← error

    for each production p of the form A → β do
        for each terminal w ∈ START(A → β) do
            Table[A,w] ← p

        if eof ∈ START(A → β)
            then Table[A,eof] ← p
```

FIGURE 3.14 LL(1) Table-Construction Algorithm.

Exercise

From the following grammar

$$
\begin{array}{lcl}
\textit{Expr} & \rightarrow & \textit{Term Expr'} \\
\textit{Expr'} & \rightarrow & \underline{+}\ \textit{Term Expr'} \\
 & \rightarrow & \epsilon \\
\textit{Term} & \rightarrow & \textit{Factor Term'} \\
\textit{Term'} & \rightarrow & \underline{\times}\ \textit{Factor Term'} \\
 & \rightarrow & \epsilon \\
\textit{Factor} & \rightarrow & \underline{(}\ \textit{Expr}\ \underline{)} \\
 & \rightarrow & \textbf{id}
\end{array}
$$

Construct a top-down parsing table

1. Find $FIRST$, $FOLLOW$, and $START$ set of each variable
2. Fill the parsing table

Is this grammar LL(1)?

Exercise

From the following grammar

$$
\begin{array}{lcl}
\textit{Stmt} & \rightarrow & \textbf{if } \textit{Expr} \textbf{ then } \textit{Stmt Stmt'} \\
 & \rightarrow & \textbf{other} \\
\textit{Stmt'} & \rightarrow & \textbf{else } \textit{Stmt} \\
 & \rightarrow & \epsilon \\
\textit{Expr} & \rightarrow & \textbf{other}
\end{array}
$$

Construct a top-down parsing table. Is this grammar LL(1)?

Outline

Bottom-up parsing

Tree building from leafs to the root, i.e.there are derivation steps

$$S \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \cdots \gamma_{n-1} \rightarrow \gamma_n = \textbf{sentence}$$

From (sentential form) γ_i and the grammar

- If there is a production $A \rightarrow \beta$ at location k in γ_i
- Replace the occurrence β that ends at position k in γ_i with A
- The result sentential form is γ_{i-1}
- Each γ_i should have exactly one handle
  - Otherwise, the grammar is not predictive

Shift-reduce parser

```
push INVALID on to the stack
word ← NextWord()
repeat until (top of stack = S and word = eof) do
    if the top of stack is a handle A → β then
        // reduce β to A
        pop |β| symbols off the stack
        push A onto the stack
    else if (word ≠ eof) then
        // shift word onto the stack
        push word onto the stack
        word ← NextWord()
    else  // parser needs to shift, but is out of input
        throw a syntax error
report success
```

■ **FIGURE 3.15** A Simple Shift-Reduce Parser.

- Shift: push terminals into a working stack
- Reduce: pop a sequence of terminal and non-terminal symbols that match the right side of a production rule

Shift-reduce parser

- Shift: push terminals onto a working stack
- Reduce: pop a sequence of terminal and non-terminal symbols on the **top of stack** that match the right side of a production rule

e.g.
$$\begin{aligned} S &\rightarrow \mathbf{a}\ AB\ \mathbf{e} \\ A &\rightarrow A\mathbf{bc} \mid \mathbf{b} \\ B &\rightarrow \mathbf{d} \end{aligned}$$

1. a**b**bcde
2. aA**bc**de
3. aA**d**e
4. **a**AB**e**
5. S

$$\begin{aligned} S &\Rightarrow \mathbf{a}A\underline{B}\mathbf{e} \\ &\Rightarrow \mathbf{a}\underline{A}\mathbf{de} \\ &\Rightarrow \mathbf{a}\underline{A}\mathbf{bcde} \\ &\Rightarrow \mathbf{abbcde} \end{aligned}$$

We obtain a right-most derivation in reverse

Handles

Loosely speaking, a handle is

- a substring that matches the right side of a production
- that reduces to the head of the production rule

Strictly speaking, a handle is is

- the pair $\langle A \to \beta, k \rangle$ for transition from γ_i to γ_{i+1}
- We will rewrite β with A

i.e.

- If $S \Rightarrow^* \alpha A w \Rightarrow \alpha \beta w$, then
- $\langle A \to \beta, w \rangle$ in the position following α is a handle of $\alpha \beta w$

Handle usage

We need to find a handle in each step of the parsing

Right-sentential form	Handle
1. $a\underline{\mathbf{b}}\mathbf{bcde}$	1. $A \rightarrow \mathbf{b}$ at position 2
2. $a\underline{A\mathbf{bc}}\mathbf{de}$	2. $A \rightarrow A\mathbf{bc}$ at position 4
3. $aA\underline{\mathbf{d}}\mathbf{e}$	3. $B \rightarrow \mathbf{d}$ at position 3
4. $\underline{\mathbf{a}AB\mathbf{e}}$	4. $S \rightarrow \mathbf{a}A\mathbf{be}$ at position 4
5. S	

We need an efficient algorithm to find a handle.

Shift-reduce parsing algorithm

We need at least 2 components

- Stack
- Input buffer

Parsing operations

- Shift (or push) input symbols onto the stack
- until a handle β is on the top of stack and the lookahead symbol is k
- Then, reduce β to the head of the production rule
  - For a handle $\langle A \rightarrow \beta, k \rangle$
  - Pop β and push A onto the stack
- Parsing is complete when the stack contains the start symbol and the input is empty

Why stack?

Consider the following cases

1. $S \Rightarrow^* \alpha A\mathbf{z} \Rightarrow^* \alpha\beta \underline{B}\mathbf{yz} \Rightarrow^* \alpha\beta\underline{\gamma}\mathbf{yz}$

Stack	Input
$\$\alpha\beta\underline{\gamma}$	**yz**$
$\$\alpha\beta\underline{B}$	**yz**$
$\$\alpha\beta B\mathbf{y}$	**z**$

2. $S \Rightarrow^* \alpha Bx A z \Rightarrow^* \alpha \underline{B} xyz \Rightarrow^* \alpha\underline{\gamma}xyz$

Stack	Input
$\$\alpha\underline{\gamma}$	**xyz**$
$\$\alpha\underline{B}$	**xyz**$
$\$\alpha B\mathbf{x}$	**yz**$
$\$\alpha Bx\mathbf{y}$	**z**$

LR(1) parser

LR(1) language is a subset of context-free languages

```
push ⟨INVALID, INVALID⟩ onto the stack
push ⟨start symbol, s0⟩ onto the stack
word ← NextWord()
while (true) do
    state ← state from pair at top of stack
    if Action[state,word] = "reduce A → β" then
        pop |β| pairs from the stack
        state ← state from pair at top of stack
        push ⟨A, Goto[state, A]⟩ onto the stack
    else if Action[state,word] = "shift si" then
        push ⟨word, si⟩ onto the stack
        word ← NextWord()
    else if Action[state,word] = "accept" and word = eof
        then break
    else throw a syntax error
report success /* executed the "accept" case */
```

■ **FIGURE 3.16** The Skeleton LR(1) Parser.

Left-to-right scan, **R**everse rightmost derivation, **1** symbol lookhead.

Example: LR(1) parse table

1	*Goal*	→	*List*
2	*List*	→	*List Pair*
3		|	*Pair*
4	*Pair*	→	(*List*)
5		|	( )

(a) Parentheses Grammar

	Action Table			Goto Table	
State	eof	(	)	*List*	*Pair*
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 7	s 8	5	6
4	r 2	r 2			
5		s 7	s 10		9
6		r 3	r 3		
7		s 7	s 12	11	6
8	r 5	r 5			
9		r 2	r 2		
10	r 4	r 4			
11		s 7	s 13		9
12		r 5	r 5		
13		r 4	r 4		

(b) *Action* and *Goto* Tables for Parentheses Grammar

■ **FIGURE 3.17** The Parentheses Grammar.

Example: LR(1) parsing steps

Iteration	State	Word	Stack	Handle	Action
0	–	(	$ ⟨*Goal* 0⟩	–*none*–	–
1	0	(	$ ⟨*Goal* 0⟩	–*none*–	*shift 3*
2	3	)	$ ⟨*Goal* 0⟩ ⟨(3⟩	–*none*–	*shift 8*
3	8	eof	$ ⟨*Goal* 0⟩ ⟨(3⟩ ⟨) 8⟩	( )	*reduce 5*
4	2	eof	$ ⟨*Goal* 0⟩ ⟨*Pair* 2⟩	*Pair*	*reduce 3*
5	1	eof	$ ⟨*Goal* 0⟩ ⟨*List* 1⟩	*List*	*accept*

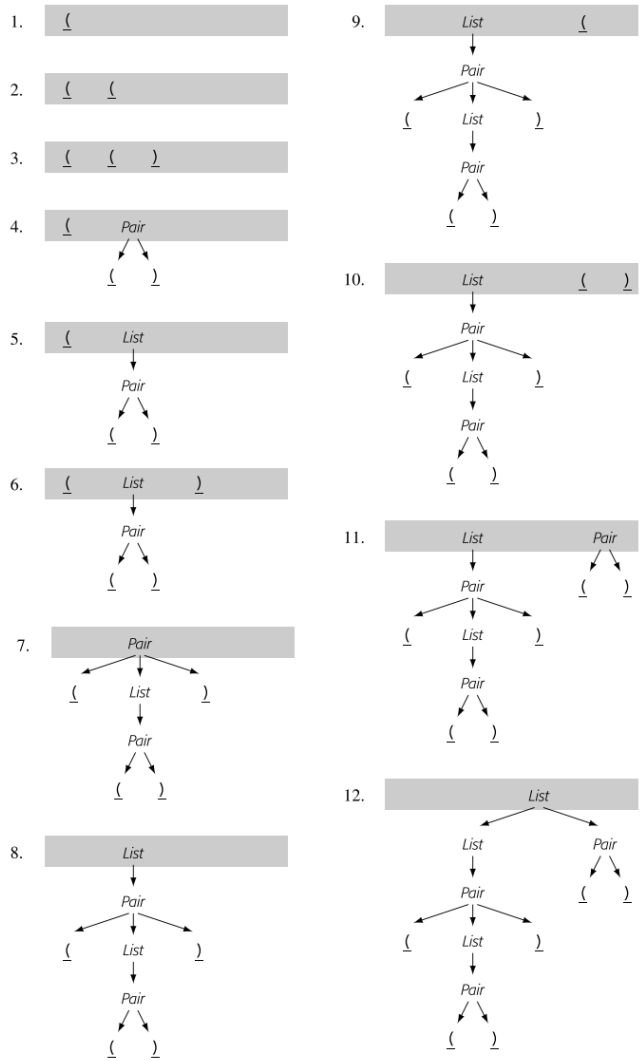
■ **FIGURE 3.18** States of the LR(1) Parser on ( ).

Example: LR(1) parsing steps

Iteration	State	Word	Stack	Handle	Action
0	—	(	$ ⟨*Goal* 0⟩	–*none*–	—
1	0	(	$ ⟨*Goal* 0⟩	–*none*–	*shift 3*
2	3	(	$ ⟨*Goal* 0⟩ ⟨(3⟩	–*none*–	*shift 7*
3	7	)	$ ⟨*Goal* 0⟩ ⟨(3⟩ ⟨(7⟩	–*none*–	*shift 12*
4	12	)	$ ⟨*Goal* 0⟩ ⟨(3⟩ ⟨(7⟩ ⟨) 12⟩	( )	*reduce 5*
5	6	)	$ ⟨*Goal* 0⟩ ⟨(3⟩ ⟨*Pair* 6⟩	*Pair*	*reduce 3*
6	5	)	$ ⟨*Goal* 0⟩ ⟨(3⟩ ⟨*List* 5⟩	–*none*–	*shift 10*
7	10	(	$ ⟨*Goal* 0⟩ ⟨(3⟩ ⟨*List* 5⟩ ⟨) 10⟩	(*List*)	*reduce 4*
8	2	(	$ ⟨*Goal* 0⟩ ⟨*Pair* 2⟩	*Pair*	*reduce 3*
9	1	(	$ ⟨*Goal* 0⟩ ⟨*List* 1⟩	–*none*–	*shift 3*
10	3	)	$ ⟨*Goal* 0⟩ ⟨*List* 1⟩ ⟨(3⟩	–*none*–	*shift 8*
11	8	eof	$ ⟨*Goal* 0⟩ ⟨*List* 1⟩ ⟨(3⟩ ⟨) 8⟩	( )	*reduce 5*
12	4	eof	$ ⟨*Goal* 0⟩ ⟨*List* 1⟩ ⟨*Pair* 4⟩	*List Pair*	*reduce 2*
13	1	eof	$ ⟨*Goal* 0⟩ ⟨*List* 1⟩	*List*	*accept*

■ **FIGURE 3.19** States of the LR(1) Parser on (( )) ( ).

Example: LR(1) parse tree



■ **FIGURE 3.20** The Sequence of Partial Parse Trees Built for (( )) ( ).

Example: LR(1) parsing error

Parsing **(())**

Iteration	State	Word	Stack	Handle	Action
0	—	(	$ ⟨*Goal* 0⟩	–*none*–	—
1	0	(	$ ⟨*Goal* 0⟩	–*none*–	*shift 3*
2	3	)	$ ⟨*Goal* 0⟩ ⟨(3⟩	–*none*–	*shift 8*
3	8	)	$ ⟨*Goal* 0⟩ ⟨(3⟩ ⟨) 8⟩	–*none*–	*error*

Outline

LR parser

LR(k) parsing technique

- L stands for the left-to-right scanning of the input
- R stands for constructing a right-most derivation in reverse
- k stands for the number of input symbols of lookahead that are used in making parsing decisions

LR parser

Pros

- Can recognize all programming language constructs for which context-free grammars can be written
- General but efficient shift-reduce parsing method
- The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers
- Can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input

Cons

- Constructing the parser by hand is difficult
  - However, we can use an LR parser generator instead

Example: Parenthesis grammar

1	*Goal*	→	*List*
2	*List*	→	*List Pair*
3		|	*Pair*
4	*Pair*	→	(*List*)
5		|	( )

The Parentheses Grammar

LR(1) items

For a production $A \rightarrow \beta\gamma$ and a lookahead symbol a

- A placeholder $\cdot$ is placed on the right-hand side of the production to create an item
- $[A \rightarrow \cdot\beta\gamma, a]$ indicates that an A would be valid and that recognizing a β next would be one step toward discovering an A.
- $[A \rightarrow \beta \cdot \gamma, a]$ indicates that the parser has progressed from the state $[A \rightarrow \cdot\beta\gamma, a]$ by recognizing β.
- $[A \rightarrow \beta\gamma\cdot, a]$ indicates that the parser has found $\beta\gamma$ in a context where an A followed by an a would be valid.

LR(1) items

[*Goal* → • *List*, eof]
[*Goal* → *List* •, eof]

[*List* → • *Pair*, eof]
[*List* → *Pair* •, eof]
[*List* → • *Pair*, ( ]
[*List* → *Pair* •, ( ]
[*List* → • *Pair*,)]
[*List* → *Pair* •,)]

[*List* → • *List Pair*, eof]
[*List* → *List* • *Pair*, eof]
[*List* → *List Pair* •, eof]
[*List* → • *List Pair*, ( ]
[*List* → *List* • *Pair*, (]
[*List* → *List Pair* •, ( ]
[*List* → • *List Pair*,)]
[*List* → *List* • *Pair*,)]
[*List* → *List Pair* •,)]

[*Pair* → • (*List*), eof]
[*Pair* → (• *List*), eof]
[*Pair* → (*List* •), eof]
[*Pair* → (*List*) •, eof]
[*Pair* → • (*List*), (]
[*Pair* → (• *List*), (]
[*Pair* → (*List* •), (]
[*Pair* → (*List*) •, (]
[*Pair* → • (*List*),)]
[*Pair* → (• *List*),)]
[*Pair* → (*List* •),)]
[*Pair* → (*List*) •,)]

[*Pair* → • ( ), eof]
[*Pair* → (•), eof]
[*Pair* → ( ) •, eof]
[*Pair* → • ( ), (]
[*Pair* → (•), (]
[*Pair* → ( ) •, (]
[*Pair* → • ( ),)]
[*Pair* → (•),)]
[*Pair* → ( ) •,)]

■ **FIGURE 3.21** LR(1) Items for the Parentheses Grammar.

Canonical collection

To construction a canonical collection, a collection of LR(1) items, we need to operations

- Closure: state transition without input
- Goto: state transition with input

```
closure(s)
    while (s is still changing) do
        for each item [A → β • C δ, a] ∈ s do
            lookahead ← δa
            for each production C → γ ∈ P do
                for each b ∈ FIRST( lookahead ) do
                    s ← s ∪ {[C → • γ, b]}
    return s
```

(a) The *Closure* Function

```
goto(s, x)
    t ← ∅
    for each item i ∈ s do
        if i is [α → β •x δ, a] then
            t ← t ∪ {[α → β x • δ, a]}
    return closure(t)
```

(b) The *Goto* Function

■ **FIGURE 3.22** Support Functions for the LR(1) Table Construction.

Construction of canonical collection

```
cc0 ← ∅
for each production of the form Goal → α do
    cc0 ← cc0 ∪ { [Goal → • α, eof] }
cc0 ←closure( cc0)
CC ← { cc0 }
while (new sets are still being added to CC) do
    for each unmarked set cci ∈ CC do
        mark cci as processed
        for each x following a • in an item in cci do
            temp ← goto(cci, x)
            if temp ∉ CC then
                CC ← CC ∪ {temp}
            record transition from cci to temp on x
```

■ **FIGURE 3.23** The Algorithm to Build the Canonical Collection of Sets of LR(1) Items.

Example: CC Construction

From CC_0, with (input

1	*Goal*	→	*List*
2	*List*	→	*List Pair*
3		|	*Pair*
4	*Pair*	→	(*List*)
5		|	( )

The Parentheses Grammar

$$\mathrm{CC}_0 = \left\{ \begin{array}{l} [Goal \rightarrow \bullet\ List, \texttt{eof}],\ [List \rightarrow \bullet\ List\ Pair, \texttt{eof}], \\ [List \rightarrow \bullet\ List\ Pair, (],\ [List \rightarrow \bullet\ Pair, \texttt{eof}], \\ [List \rightarrow \bullet\ Pair, (],\ [Pair \rightarrow \bullet\ (\ List\ ), \texttt{eof}], \\ [Pair \rightarrow \bullet\ (\ List\ ), (],\ [Pair \rightarrow \bullet\ (\ ), \texttt{eof}], \\ [Pair \rightarrow \bullet\ (\ ), (] \end{array} \right\}$$

$$\mathrm{CC}_3 = \left\{ \begin{array}{l} [Pair \rightarrow (\ \bullet\ List\ ), \texttt{eof}]\ [Pair \rightarrow (\ \bullet\ List\ ), (] \\ [Pair \rightarrow (\ \bullet\ ), \texttt{eof}]\ [Pair \rightarrow (\ \bullet\ ), (] \\ [List \rightarrow \bullet\ List\ Pair, (]\ [List \rightarrow \bullet\ List\ Pair,)] \\ [List \rightarrow \bullet\ Pair, (]\ [List \rightarrow \bullet\ Pair,)]\ [Pair \rightarrow \bullet\ (\ List\ ), (] \\ [Pair \rightarrow \bullet\ (\ List\ ),)]\ [Pair \rightarrow \bullet\ (\ ), (]\ [Pair \rightarrow \bullet\ (\ ),)] \end{array} \right\}$$

Example: CC Construction (cont.)

$$CC_0 = \left\{ \begin{array}{l} [Goal \to \bullet\ List, \texttt{eof}] \\ [List \to \bullet\ List\ Pair, \texttt{eof}]\ [List \to \bullet\ List\ Pair, \underline{(}]\ [List \to \bullet\ Pair, \texttt{eof}]\ [List \to \bullet\ Pair, \underline{(}] \\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \texttt{eof}]\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \texttt{eof}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{(}] \end{array} \right\}$$

$$CC_1 = \left\{ \begin{array}{l} [Goal \to List \bullet, \texttt{eof}]\ [List \to List \bullet\ Pair, \texttt{eof}]\ [List \to List \bullet\ Pair, \underline{(}] \\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \texttt{eof}]\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \texttt{eof}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{(}] \end{array} \right\}$$

$$CC_2 = \left\{ [List \to Pair \bullet, \texttt{eof}]\ [List \to Pair \bullet, \underline{(}] \right\}$$

$$CC_3 = \left\{ \begin{array}{l} [Pair \to \underline{(}\ \bullet\ List\ \underline{)}, \texttt{eof}]\ [Pair \to \underline{(}\ \bullet\ List\ \underline{)}, \underline{(}]\ [Pair \to \underline{(}\ \bullet\ \underline{)}, \texttt{eof}]\ [Pair \to \underline{(}\ \bullet\ \underline{)}, \underline{(}] \\ [List \to \bullet\ List\ Pair, \underline{(}]\ [List \to \bullet\ List\ Pair, \underline{)}]\ [List \to \bullet\ Pair, \underline{(}]\ [List \to \bullet\ Pair, \underline{)}] \\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{)}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{)}] \end{array} \right\}$$

$$CC_4 = \left\{ [List \to List\ Pair \bullet, \texttt{eof}]\ [List \to List\ Pair \bullet, \underline{(}] \right\}$$

$$CC_5 = \left\{ \begin{array}{l} [List \to List \bullet\ Pair, \underline{(}]\ [List \to List \bullet\ Pair, \underline{)}]\ [Pair \to \underline{(}\ List \bullet\ \underline{)}, \texttt{eof}]\ [Pair \to \underline{(}\ List \bullet\ \underline{)}, \underline{(}] \\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{)}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{)}] \end{array} \right\}$$

$$CC_6 = \left\{ [List \to Pair \bullet, \underline{(}]\ [List \to Pair \bullet, \underline{)}] \right\}$$

$$CC_7 = \left\{ \begin{array}{l} [Pair \to \underline{(}\ \bullet\ List\ \underline{)}, \underline{(}]\ [Pair \to \underline{(}\ \bullet\ List\ \underline{)}, \underline{)}]\ [Pair \to \underline{(}\ \bullet\ \underline{)}, \underline{(}]\ [Pair \to \underline{(}\ \bullet\ \underline{)}, \underline{)}] \\ [List \to \bullet\ List\ Pair, \underline{(}]\ [List \to \bullet\ List\ Pair, \underline{)}]\ [List \to \bullet\ Pair, \underline{(}]\ [List \to \bullet\ Pair, \underline{)}] \\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{)}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{)}] \end{array} \right\}$$

$$CC_8 = \left\{ [Pair \to \underline{(}\ \underline{)}\ \bullet, \texttt{eof}]\ [Pair \to \underline{(}\ \underline{)}\ \bullet, \underline{(}] \right\}$$

$$CC_9 = \left\{ [List \to List\ Pair \bullet, \underline{(}]\ [List \to List\ Pair \bullet, \underline{)}] \right\}$$

$$CC_{10} = \left\{ [Pair \to \underline{(}\ List\ \underline{)}\ \bullet, \texttt{eof}]\ [Pair \to \underline{(}\ List\ \underline{)}\ \bullet, \underline{(}] \right\}$$

$$CC_{11} = \left\{ \begin{array}{l} [List \to List \bullet\ Pair, \underline{(}]\ [List \to List \bullet\ Pair, \underline{)}]\ [Pair \to \underline{(}\ List \bullet\ \underline{)}, \underline{(}]\ [Pair \to \underline{(}\ List \bullet\ \underline{)}, \underline{)}] \\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ List\ \underline{)}, \underline{)}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{(}]\ [Pair \to \bullet\ \underline{(}\ \underline{)}, \underline{)}] \end{array} \right\}$$

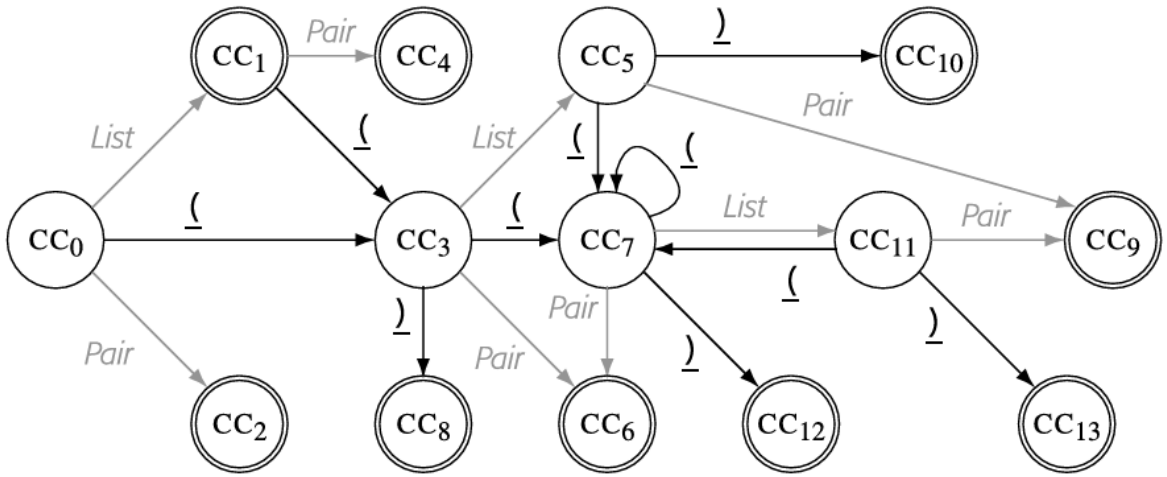
Filling parse tables

Shift, reduce, accept actions

for each $CC_i \in \mathcal{CC}$ *do*
  for each item $I \in CC_i$ *do*
    if I *is* $[A \rightarrow \beta \bullet \mathtt{c}\ \gamma, \mathtt{a}]$ *and* $goto(CC_i, \mathtt{c}) = CC_j$ *then*
      $Action[i, \mathtt{c}] \leftarrow$ "*shift* j"
    else if I *is* $[A \rightarrow \beta \bullet, \mathtt{a}]$ *then*
      $Action[i, \mathtt{a}] \leftarrow$ "*reduce* $A \rightarrow \beta$"
    else if I *is* $[Goal \rightarrow \beta \bullet, \mathtt{eof}]$ *then*
      $Action[i, \mathtt{eof}] \leftarrow$ "*accept*"
  for each $n \in NT$ *do*
    if $goto(CC_i, n) = CC_j$ *then*
      $Goto[i, n] \leftarrow j$

■ **FIGURE 3.26** LR(1) Table-Filling Algorithm.

Visualizing transition table



■ **FIGURE 3.27** Handle-Finding DFA for the Parentheses Grammar.

Exercise

Construct an LR(1) for the following grammar

$$S \rightarrow S + S \mid SS \mid (S) \mid a$$

Is this grammar LR(1)?

Outline

Conflicts types

1. Shift/reduce conflict
   - Can either shift or reduce
2. Reduce/reduce conflict
   - Have several possible reducible production rules

We can give preference to a rule over others to solve the conflict.

Shift/reduce conflict

Example

$$
\begin{aligned}
stmt \rightarrow\ & \textbf{if}\ expr\ \textbf{then}\ stmt \\
\mid\ & \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\
\mid\ & \textbf{other}
\end{aligned}
$$

We found a conflict in the following configuration

Stack	Input
\$. . . **if** $expr$ **then** $stmt$	**else** . . . \$

We can give preference to a rule over others to resolve the conflict.

Reduce/reduce conflict

Example

$$
\begin{aligned}
(1) \quad & stmt \rightarrow \textsf{id}(parameter\_list) \\
(2) \quad & stmt \rightarrow expr := expr \\
(3) \quad & parameter\_list \rightarrow paramter\_list, parameter \\
(4) \quad & parameter\_list \rightarrow parameter \\
(5) \quad & parameter \rightarrow \textsf{id} \\
(6) \quad & expr \rightarrow \textsf{id}(expr\_list) \\
(7) \quad & expr \rightarrow \textsf{id} \\
(8) \quad & expr\_list \rightarrow expr\_list, expr \\
(9) \quad & expr\_list \rightarrow expr
\end{aligned}
$$

We found a conflict in the following configuration

Stack	Input
$\$ \ldots \textbf{id}(\textbf{id}$	$, \textbf{id}) \ldots \$$

We can change to token **id** in production (1) to **procid** to differentiate (1) from (6)

Outline

Error report

- Current parsing algorithm halts when encounter the first error
- Finding all syntax errors may be more helpful (and common)
  - We need a synchronize symbol to resume from the error state

Error recovery

Top-down parsing

- Read new input tokens until the synchronize token (e.g. ;) is found

Bottom-up parsing

- Read new input tokens until the synchronize token (e.g. ;) is found
- Clean-up the stack upto to starting point of the error path (e.g. the starting symbol of a statement)