Sub: Compiler Construction Syntax Analysis PART 3

Compiled for: 7th Sem, CE, DDU

Compiled by: Niyati J. Buch

Ref. : Compilers: Principles, Techniques, and Tools, 2nd Edition
Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

Topics Covered

- Error Recovery in Predictive Parsing
- Bottom-up parsing
  - Shift Reduce Parsing
  - Conflict

Error Recovery in Predictive Parsing

- Error recovery refers to the stack of a table-driven predictive parser, since it makes explicit the terminals and nonterminals that the parser hopes to match with the remainder of the input; the techniques can also be used with recursive-descent parsing.
- An error is detected during predictive parsing
  - when the terminal on top of the stack does not match the next input symbol
  - or when non terminal A is on top of the stack, a is the next input symbol, and M[A, a] is error (i.e., the parsing-table entry is empty).

Panic Mode

- Panic-mode error recovery is based on the idea of skipping over symbols on the input until a token in a selected set of synchronizing tokens appears.
- Its effectiveness depends on the choice of synchronizing set.
- The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice.
- The compiler designer must supply informative error messages that not only describe the error, they must draw attention to where the error was discovered.

Few Heuristics

1. As a starting point, place all symbols in FOLLOW(A) into the synchronizing set for nonterminal A. If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it is likely that parsing can continue.
2. It is not enough to use FOLLOW(A) as the synchronizing set for A.
   – For example, if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the FOLLOW set of the nonterminal representing expressions. A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped.

Few Heuristics

3. If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.
4. If a nonterminal can generate the empty string, then the production deriving can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.

Few Heuristics

5. If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.

EXAMPLE 2

$E \rightarrow TE'$	FIRST(E) = {(, id}	FOLLOW(E) = {), $}
$E' \rightarrow +TE' \mid \epsilon$	FIRST(E') = {+, ϵ}	FOLLOW(E') = {), $}
$T \rightarrow FT'$	FIRST(T) = {(, id}	FOLLOW(T) = {+,), $}
$T' \rightarrow *FT' \mid \epsilon$	FIRST(T') = {*, ϵ}	FOLLOW(T') ={+,), $}
$F \rightarrow (E) \mid id$	FIRST(F) = {(, id}	FOLLOW(F) = {*, +,), $}

	Input Symbols					
Non-Terminals	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

For each production $A \rightarrow \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to M[A, a].
2. If ∈ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \rightarrow \alpha$ to M[A, b]. If ∈ is in FIRST(α) and $ is in FOLLOW(A), add $A \rightarrow \alpha$ to M[A, $] as well.

EXAMPLE 2

$E \rightarrow TE'$	FIRST(E) = {(, id}	FOLLOW(E) = {), $}
$E' \rightarrow +TE' \mid \epsilon$	FIRST(E') = {+, ϵ}	FOLLOW(E') = {), $}
$T \rightarrow FT'$	FIRST(T) = {(, id}	FOLLOW(T) = {+,), $}
$T' \rightarrow *FT' \mid \epsilon$	FIRST(T') = {*, ϵ}	FOLLOW(T') ={+,), $}
$F \rightarrow (E) \mid id$	FIRST(F) = {(, id}	FOLLOW(F) = {*, +,), $}

	Input Symbols					
Non-Terminals	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

The parsing table with "synch" indicating synchronizing tokens obtained from the FOLLOW set of the non terminal in question.

EXAMPLE 2

E→TE'	FIRST(E) = {(, id}	FOLLOW(E) = {), $}
E'→+TE'|ϵ	FIRST(E') = {+, ϵ}	FOLLOW(E') = {), $}
T→FT'	FIRST(T) = {(, id}	FOLLOW(T) = {+,), $}
T'→*FT'|ϵ	FIRST(T') = {*, ϵ}	FOLLOW(T') ={+,), $}
F→(E)|id	FIRST(F) = {(, id}	FOLLOW(F) = {*, +,), $}

	Input Symbols					
Non-Terminals	id	+	*	(	)	$
E	E→TE'			E→TE'	synch	synch
E'		E'→+TE'			E'→ ϵ	E'→ ϵ
T	T→FT'			T→FT'		
T'		T'→ ϵ	T'→*FT'		T'→ ϵ	T'→ ϵ
F	F → id			F→(E)		

The parsing table with "synch" indicating synchronizing tokens obtained from the FOLLOW set of the non terminal in question.

EXAMPLE 2

E→TE'	FIRST(E) = {(, id}	FOLLOW(E) = {), $}
E'→+TE'|ϵ	FIRST(E') = {+, ϵ}	FOLLOW(E') = {), $}
T→FT'	FIRST(T) = {(, id}	FOLLOW(T) = {+,), $}
T'→*FT'|ϵ	FIRST(T') = {*, ϵ}	FOLLOW(T') ={+,), $}
F→(E)|id	FIRST(F) = {(, id}	FOLLOW(F) = {*, +,), $}

	Input Symbols					
Non-Terminals	id	+	*	(	)	$
E	E→TE'			E→TE'	synch	synch
E'		E'→+TE'			E'→ ϵ	E'→ ϵ
T	T→FT'	synch		T→FT'	synch	synch
T'		T'→ ϵ	T'→*FT'		T'→ ϵ	T'→ ϵ
F	F → id			F→(E)		

The parsing table with "synch" indicating synchronizing tokens obtained from the FOLLOW set of the non terminal in question.

EXAMPLE 2

$E \rightarrow TE'$	FIRST(E) = {(, id}	FOLLOW(E) = {), $}
$E' \rightarrow +TE' \mid \epsilon$	FIRST(E') = {+, ϵ}	FOLLOW(E') = {), $}
$T \rightarrow FT'$	FIRST(T) = {(, id}	FOLLOW(T) = {+,), $}
$T' \rightarrow *FT' \mid \epsilon$	FIRST(T') = {*, ϵ}	FOLLOW(T') ={+,), $}
$F \rightarrow (E) \mid id$	FIRST(F) = {(, id}	FOLLOW(F) = {*, +,), $}

	Input Symbols					
Non-Terminals	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

The parsing table with "synch" indicating synchronizing tokens obtained from the FOLLOW set of the non terminal in question.

EXAMPLE 2

$E \to TE'$	FIRST(E) = {(, id}	FOLLOW(E) = {), $}
$E' \to +TE' \mid \epsilon$	FIRST(E') = {+, ϵ}	FOLLOW(E') = {), $}
$T \to FT'$	FIRST(T) = {(, id}	FOLLOW(T) = {+,), $}
$T' \to *FT' \mid \epsilon$	FIRST(T') = {*, ϵ}	FOLLOW(T') ={+,), $}
$F \to (E) \mid id$	FIRST(F) = {(, id}	FOLLOW(F) = {*, +,), $}

	Input Symbols					
Non-Terminals	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$	synch	synch	$F \to (E)$	synch	synch

The parsing table with "synch" indicating synchronizing tokens obtained from the FOLLOW set of the non terminal in question.

Erroneous input **) id * + id**

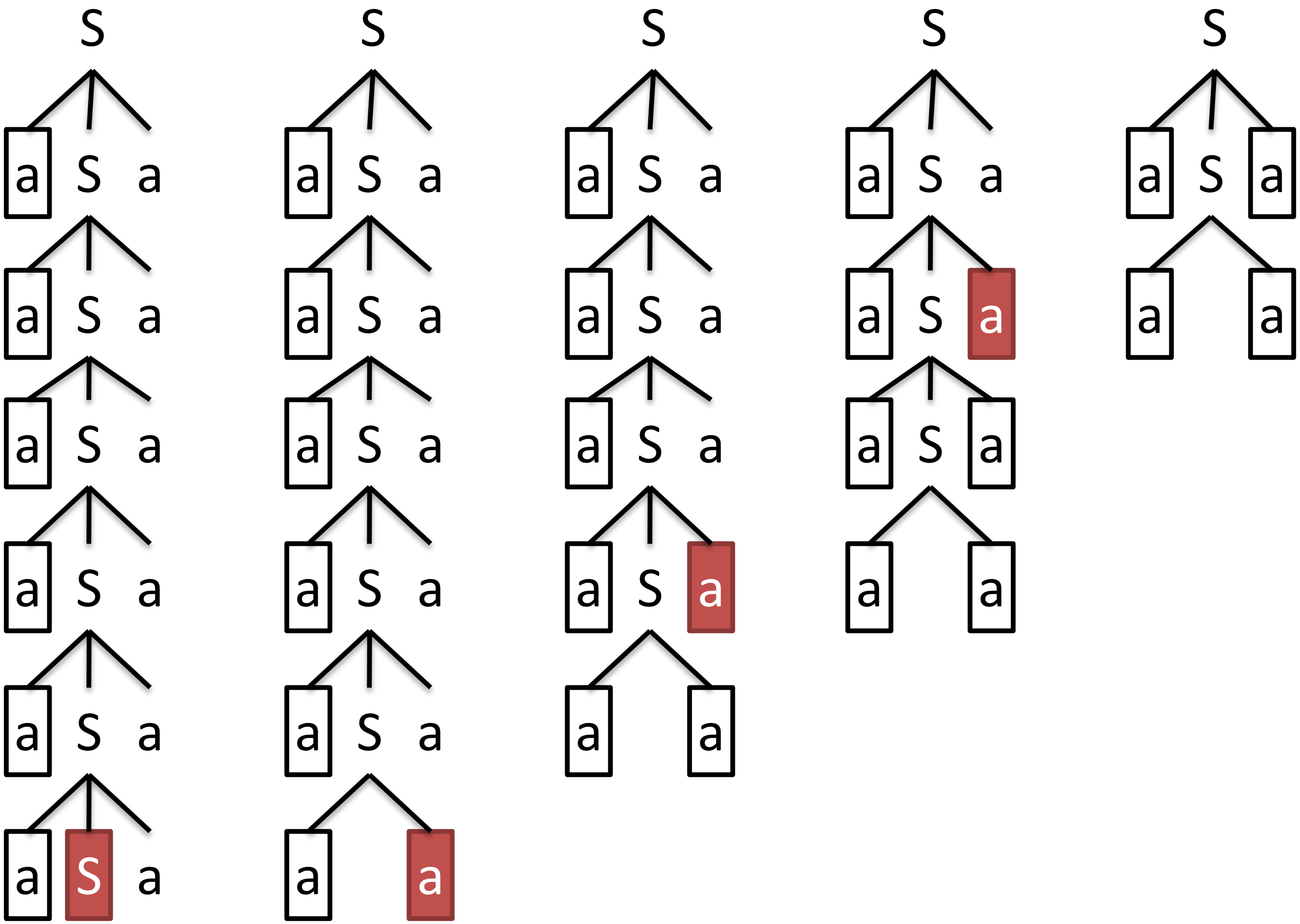
The parser and error recovery mechanism

	Input Symbols					
Non-Terminals	id	+	*	(	)	$
E	E→TE'			E→TE'	synch	synch
E'		E'→+TE'			E'→ ϵ	E'→ ϵ
T	T→FT'	synch		T→FT'	synch	synch
T'		T'→ ϵ	T'→*FT'		T'→ ϵ	T'→ ϵ
F	F → id	synch	synch	F→(E)	synch	synch

Stack	**Input**	**Remarks**
$ E	) id * + id $	Error so skip the symbol
$ E	id * + id $	E → TE'
$ E' T	id * + id $	T → FT'
$ E' T' F	id * + id $	F → id
$ E' T' id	id * + id $	MATCH so pop
$ E' T'	* + id $	T'→*FT’
$ E' T' F*	* + id $	MATCH so pop
$ E' T' F	+ id $	Error so skip the symbol
$ E' T' F	id $	F → id
$ E' T' id	id $	MATCH so pop
$ E' T'	$	T'→ ϵ
$ E'	$	E'→ ϵ
$	$	

G: S →aSa | aa
input: aaaaaa

Recursive Descent Parser



Predictive Parser

G: S → aSa | aa

- FIRST(S) = {a}
- FOLLOW(S) = {$, a}

	Input Symbols	
Non-Terminals	a	$
S	S→aSa S →aa	

For each production A → α of the grammar, do the following:

1. For each terminal a in FIRST(α), add A → α to M[A, a].
2. If ∈ is in FIRST(α), then for each terminal b in FOLLOW(A), add A → α to M[A, b]. If ∈ is in FIRST(α) and $ is in FOLLOW(A), add A → α to M[A, $] as well.

	STACK	INPUT	ACTION
1	$S	aaaaaa$	S →aSa
2	$aSa	aaaaaa$	MATCH, so pop
3	$aS	aaaaa$	S →aSa
4	$aaSa	aaaaa$	MATCH, so pop
5	$aaS	aaaa$	S →aSa
6	$aaaSa	aaaa$	MATCH, so pop
7	$aaaS	aaa$	S →aSa
8	$aaaaSa	aaa$	MATCH, so pop
9	$aaaaS	aa$	S →aSa
10	$aaaaaSa	aa$	MATCH, so pop
11	$aaaaaS	a$	S →aSa
12	$aaaaaaSa	a$	MATCH, so pop
13	$aaaaaaS	$	STACK not empty so backtrack to 11
14	$aaaaaS	a$	S →aa
15	$aaaaaaa	a$	MATCH, so pop
16	$aaaaaa	$	STACK not empty so backtrack to 9

	STACK	INPUT	ACTION
16	$aaaaaa	$	STACK not empty so backtrack to 9
17	$aaaaS	aa$	S →aa
18	$aaaaaa	aa$	MATCH, so pop
19	$aaaaa	a$	MATCH, so pop
20	$aaaa	$	STACK not empty so backtrack to 7
21	$aaaS	aaa$	S →aa
22	$aaaaa	aaa$	MATCH, so pop
23	$aaaa	aa$	MATCH, so pop
24	$aaa	a$	MATCH, so pop
24	$aa	$	STACK not empty so backtrack to 5
25	$aaS	aaaa$	S →aa
26	$aaaa	aaaa$	MATCH, so pop
27	$aaa	aaa$	MATCH, so pop
28	$aa	aa$	MATCH, so pop
29	$a	a$	MATCH, so pop
30	$	$	

There should be no back tracking in predictive parsing

	STACK	INPUT	ACTION
16	$aaaaaa	$	STACK not empty so backtrack to 9
17	$aaaaS	aa$	S → aa
18	$aaaaaa	aa$	MATCH, so pop
19	$aaaaa	a$	MATCH, so pop
20	$aaaa	$	STACK not empty so backtrack
21	$aaaS	aaa$	S → aa
22	$aaaaa	aaa$	MATCH, so pop
23	$aaaa	aa$	MATCH, so pop
24	$aaa	a$	MATCH, so pop
24	$aa	$	STACK not empty so backtrack to 5
25	$aaS	aaaa$	S → aa
26	$aaaa	aaaa$	MATCH, so pop
27	$aaa	aaa$	MATCH, so pop
28	$aa	aa$	MATCH, so pop
29	$a	a$	MATCH, so pop
30	$	$	

Mayday…
Mayday…
This is
incorrect.
WHY??

	STACK	INPUT	ACTION
16	\$aaaaaa	\$	STACK not empty so backtrack to 9
17	\$aaaaS	aa\$	S → aa
18	\$aaaaaa	aa\$	MATCH, so pop
19	\$aaaaa	a\$	MATCH, so pop
20	\$aaaa	\$	STACK not emp
21	\$aaaS	aaa\$	S → aa
22	\$aaaaa	aaa\$	MATCH, so
23	\$aaaa	aa\$	MATCH, so
24	\$aaa	a\$	MATCH, so
24	\$aa	\$	STACK
25	\$aaS	aaaa\$	S → aa
26	\$aaaa	aaaa\$	MATCH, so pop
27	\$aaa	aaa\$	MATCH, so pop
28	\$aa	aa\$	MATCH, so pop
29	\$a	a\$	MATCH, so pop
30	\$	\$	

WHY??
The grammar in not LL(1) so should not use predictive parsing.

Why is it not LL(1)??
Because it is not left factored grammar.

$G: S \rightarrow aSa \mid aa$

*Predictive Parser

- Let's try to make this grammar LL(1)
- Common left factor a is observed.
- Making the grammar left-factored,

  $S \rightarrow aS'$

  $S' \rightarrow Sa \mid a$
- Again indirect left factoring is observed,

  $S \rightarrow aS'$

  $S' \rightarrow aA$

  $A \rightarrow S'a \mid \in$

*Predictive Parser

G: $S \rightarrow aSa \mid aa$

- Let’s try to make this grammar LL(1)
- Common left factor a is observed.
- Making the grammar left-factored,

  $S \rightarrow aS'$

  $S' \rightarrow Sa \mid a$
- Again indirect left factoring is observed,

  $S \rightarrow aS'$

  $S' \rightarrow aA$

  $A \rightarrow S'a \mid \in$

*Predictive Parser

G: S →aSa | aa

- Let’s try to make this grammar LL(1)
- Common left factor a is observed.
- Making the grammar left-factored,

  S → aS'

  S' →Sa | a
- Again indirect left factoring is observed,

  S → aS'

  S' →aA

  A →S'a | ∈

	Input Symbols	
Non-Terminals	a	$
S	S →aS'	
S'	S' →aA	
A	A →S'a A → ∈	A → ∈

G: S →aSa | aa

*Predictive Parser

- Let’s try to make this grammar LL(1)
- Common left factor a is observed.
- Making the grammar left-factored,

  S → aS'

  S' →Sa | a
- Again indirect left factoring is observed,

  S → As'

  S' →aA

  A →S'a | ∈

	Input Symbols	
Non-Terminals	a	$
S	S →aS'	
S'	S' →aA	
A	A →S'a A → ∈	A → ∈

NOT LL(1)

As Look-ahead of one symbol is not sufficient.

Bottom Up Parsing

Bottom Up Parsing

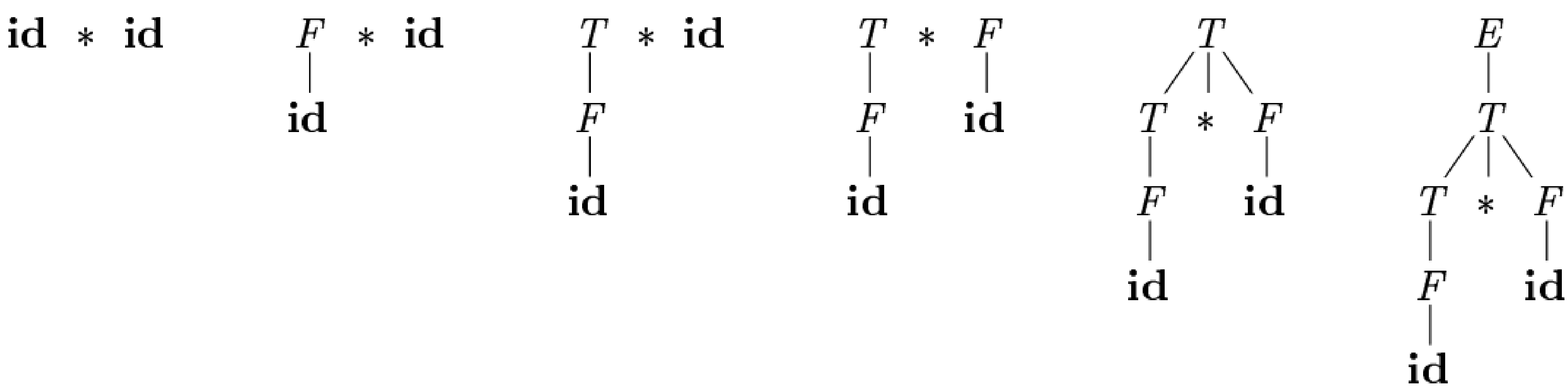
- A **bottom-up parse** corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).
- It is convenient to describe parsing as the process of building parse trees, although a front end may in fact carry out a translation directly without building an explicit tree.

Simple Example of id * id

E → E + T | T

T → T * F | F

F → (E) | id

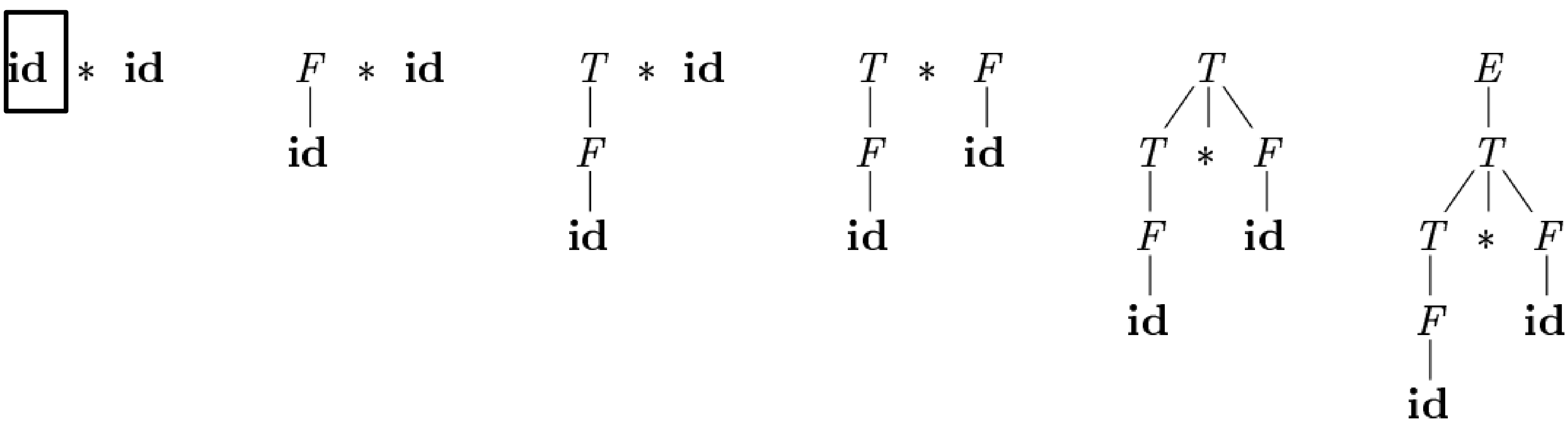


Simple Example of id * id

E → E + T | T

T → T * F | F

F → (E) | id



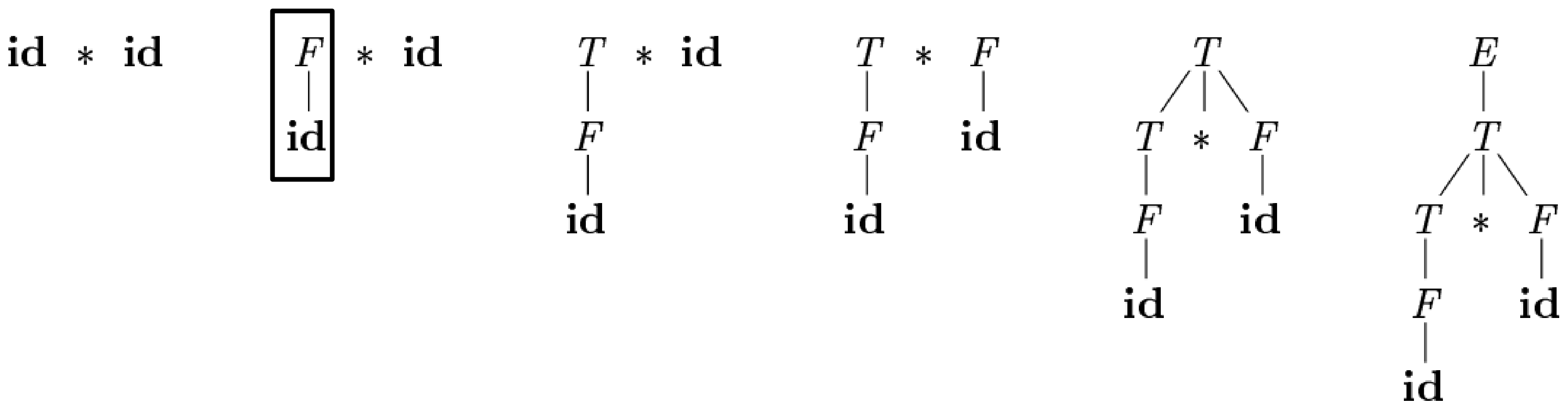
Simple Example of id * id

E → E + T | T

T → T * F | F

F → (E) | id

F → id



Simple Example of id * id

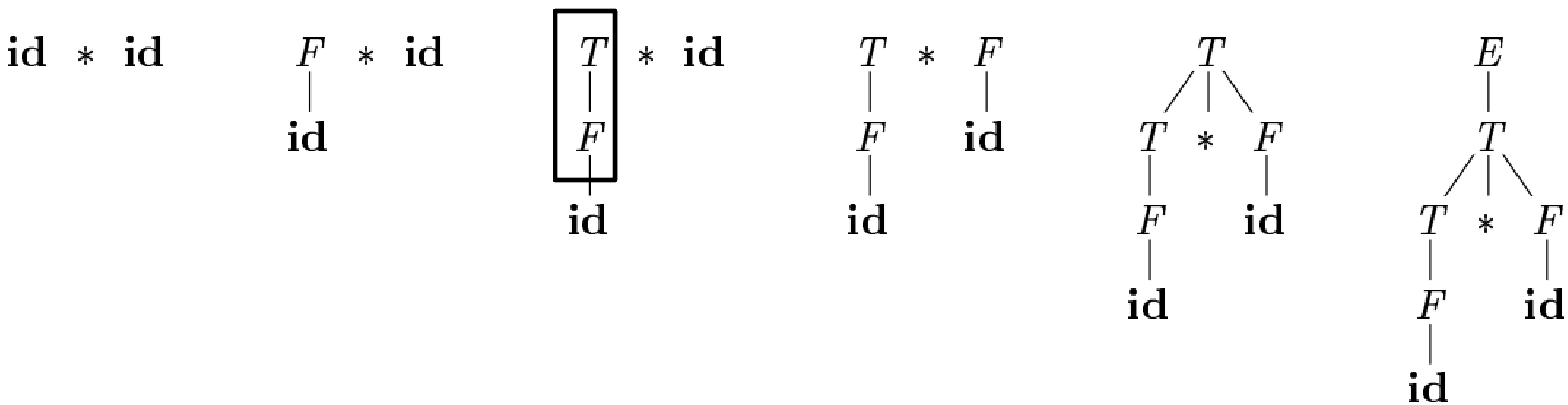
E → E + T | T

T → T * F | F

F → (E) | id

F → id

T → F



Simple Example of id * id

E → E + T | T

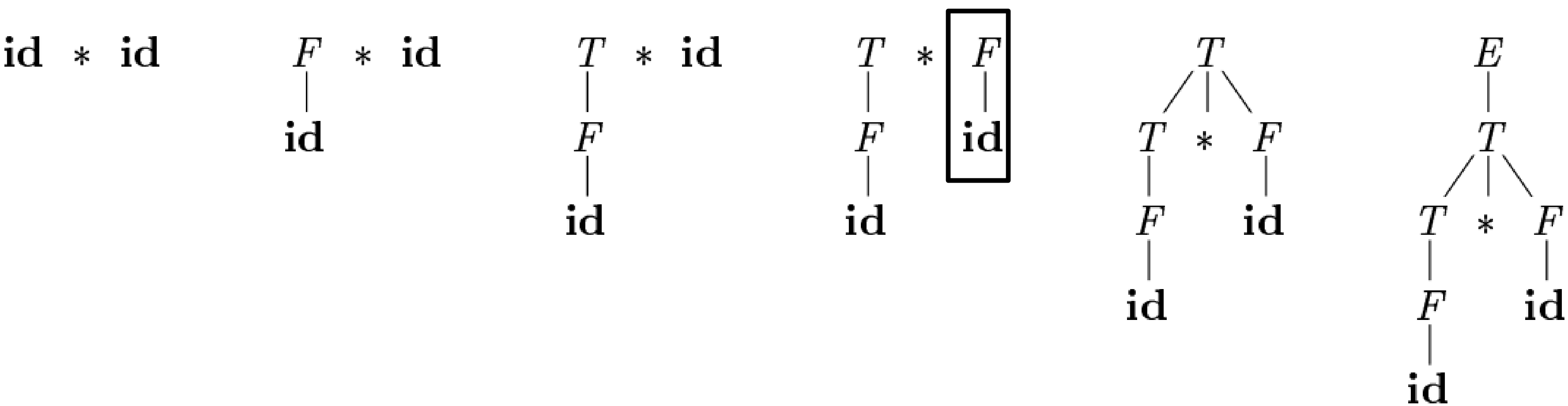
T → T * F | F

F → (E) | id

F → id

T → F

F → id



Simple Example of id * id

E → E + T | T

T → T * F | F

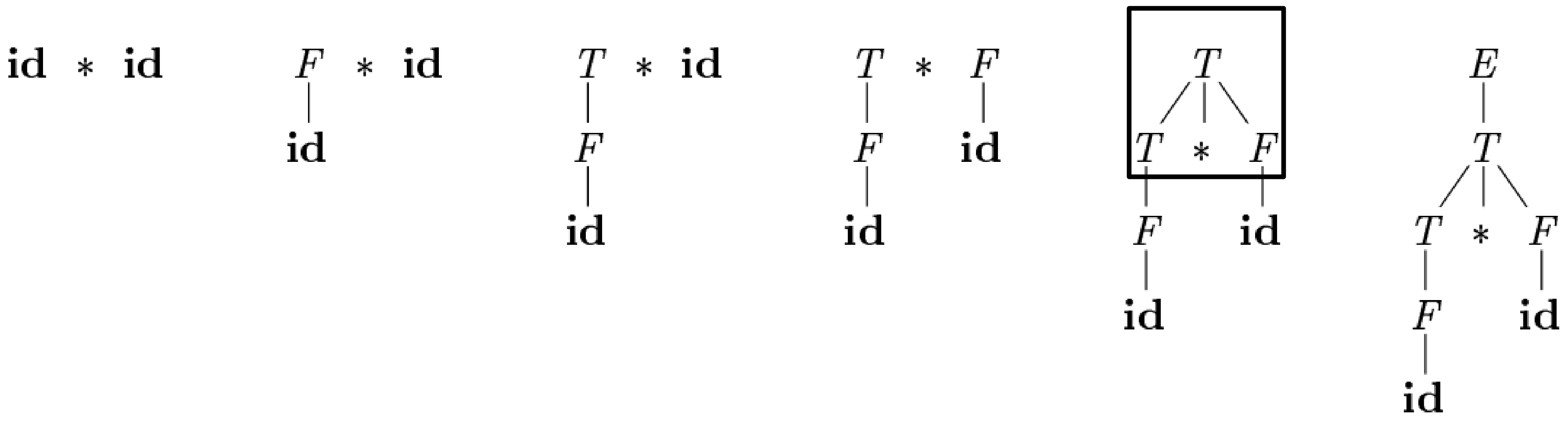
F → (E) | id

F → id

T → F

F → id

T → T * F



Simple Example of id * id

E → E + T | T

T → T * F | F

F → (E) | id

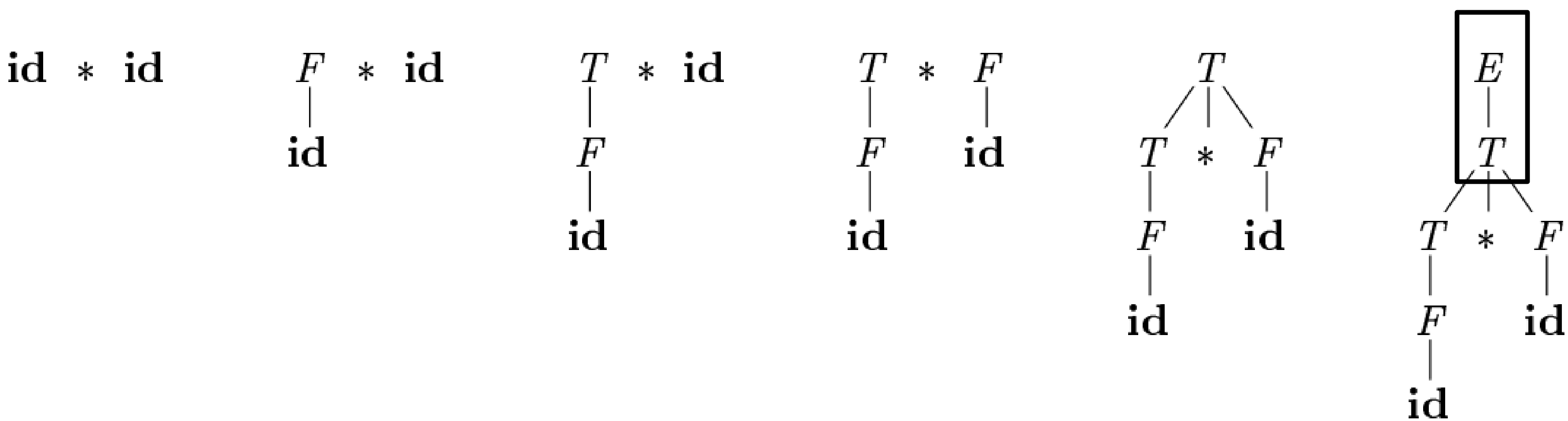
F → id

T → F

F → id

T → T * F

E → T



Simple Example of id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

$F \rightarrow id$

$T \rightarrow F$

$F \rightarrow id$

$T \rightarrow T * F$

$E \rightarrow T$

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id$$

Simple Example of id * id

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

$F \rightarrow id$

$T \rightarrow F$

$F \rightarrow id$

$T \rightarrow T * F$

$E \rightarrow T$

$E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id$

Simple Example of id * id

E → E + T | T

T → T * F | F

F → (E) | id

F → id

T → F

F → id

T → T * F

E → T

E ⇨ T ⇨ T * F ⇨ T * id ⇨ F * id ⇨ id * id

Yay...

Rightmost Derivative

in Reverse

Bottom-up Parsing

- **Bottom-up parsing** during a left-to-right scan of the input constructs a rightmost derivation in reverse.
- Informally, a **"handle"** is a substring that matches **the body of a production**, and whose **reduction represents one step along the reverse of a rightmost derivation**.

Handle during a parse of id * id

Right Sentential Form	Handle	Reducing Production
id * id	id	F → id
F * id	F	T → F
T * id	id	F → id
T * F	T * F	T → T * F
T	T	E → T

a "**handle**" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

Example

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

Input String : abbcde

Example

S → aABe

A → Abc | b

B → d

Input String : abbcde

S ⇨ aABe ⇨ aAde ⇨ aAbcde ⇨ abbcde

Right Sentential Form	Handle	Reducing Production

Example

S → aABe

A →Abc | b

B → d

Input String : abbcde

S ⇨ aABe ⇨aAde ⇨aAbcde ⇨abbcde

Right Sentential Form	Handle	Reducing Production
a**b**bcde	b	A→b
a**Abc**de	Abc	A →Abc
aA**d**e	d	B → d
aABe	**aABe**	S → aABE

Handle-pruning

- The process of discovering a handle & reducing it to the appropriate left-hand side is called **handle pruning**.
- Handle pruning forms the basis for a bottom-up parsing method.

Identifying the correct handle (substring) is always difficult.

Shift-Reduce Parser

- **Shift-reduce parsing** is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
- During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string of grammar symbols on top of the stack.
- It then reduces to the head of the appropriate production.
- The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is $.

Actions of shift-reduce parser

1. **Shift**
   – Shift the next input symbol onto the top of the stack.
2. **Reduce.**
   – The right end of the string to be reduced must be at the top of the stack.
   – Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. **Accept.**
   – Announce successful completion of parsing.
4. **Error.**
   – Discover a syntax error and call an error recovery routine.

Shift Reduce Parsing with a Stack

- **Two problems:**
  - locate a handle and
  - decide which production to use (if there are more than two candidate productions).
- **General Construction: using a stack:**
  - **“shift”** input symbols into the stack until a handle is found on top of it.
  - **“reduce”** the handle to the corresponding non-terminal.
  - other operations:
    - **“accept”** when the input is consumed and only the start symbol is on the stack,
    - **“error”**

Example 1
Input string: id1 * id2

E → E + T | T
T → T * F | F
F → (E) | id

STACK	INPUT	ACTION
$	id1 * id2 $	shift
$ id1	* id2 $	Reduce F → id
$ F	* id2 $	Reduce T → F
$ T	* id2 $	shift
$ T *	id2 $	shift
$ T * id2	$	Reduce F → id
$ T * F	$	Reduce T → T * F
$ T	$	Reduce E → T
$ E	$	Accept

The use of a stack in shift-reduce parsing is justified by an important fact: the handle will always eventually appear on top of the stack, never inside.

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Right Sentential Form	Handle	Reducing Production
id * id	**id**	F → id
F * id	**F**	T → F
T * id	**id**	F → id
T * F	**T * F**	T → T * F
T	**T**	E → T

STACK	INPUT	ACTION
$	id1 * id2 $	shift
$ id1	* id2 $	Reduce F → id
$ F	* id2 $	Reduce T → F
$ T	* id2 $	shift
$ T *	id2 $	shift
$ T * id2	$	Reduce F → id
$ T * F	$	Reduce T → T * F
$ T	$	Reduce E → T
$ E	$	Accept

Example 2

- For the grammar $S \rightarrow 0\ S\ 1 \mid 0\ 1$ indicate the handle in each of the following right-sentential forms:
  a) 000111
  b) 00S11

Right Sentential Form	Handle	Reducing Production
000111		
00S11		

Example 2

- For the grammar $S \rightarrow 0\ S\ 1 \mid 0\ 1$ indicate the handle in each of the following right-sentential forms:
  a) 000111
  b) 00S11

Right Sentential Form	Handle	Reducing Production
000111	01	$S \rightarrow 01$
00S11		

To generate 000111,

$S \rightarrow 0\underline{S}1$

$\rightarrow 00\underline{S}11$

$\rightarrow 000111$

Example 2

- For the grammar $S \rightarrow 0\ S\ 1 \mid 0\ 1$ indicate the handle in each of the following right-sentential forms:
  a) 000111
  b) 00S11

Right Sentential Form	Handle	Reducing Production
000111	01	$S \rightarrow 01$
00S11	0S1	$S \rightarrow 0S1$

To generate 00S11,

$S \rightarrow 0\underline{S}1$

$\rightarrow 00\underline{S}11$

Example 3

- For the grammar $S \rightarrow S\ S\ +\ |\ S\ S\ *\ |\ a$ indicate the handle in each of the following right-sentential forms:
  a) S S S + a * +
  b) S S + a * a +
  c) a a a * a + +

Right Sentential Form	Handle	Reducing Production
S S S + a * +		
S S + a * a +		
a a a * a + +		

Example 3

- For the grammar S → S S + | S S * | a indicate the handle in each of the following right-sentential forms:

  a) **S S S + a * +**

  b) S S + a * a +

  c) a a a * a + +

S → S <u>S</u> +
→ S S <u>S</u> * +
→ S <u>S</u> a * +
→ S S <u>S</u> + a * +

Right Sentential Form	Handle	Reducing Production
S S S + a * +	SS+	S → SS+
S S + a * a +		
a a a * a + +		

Example 3

- For the grammar S → S S + | S S * | a indicate the handle in each of the following right-sentential forms:

  a) S S S + a * +

  b) S S + a * a +

  c) a a a * a + +

$$
\begin{aligned}
S &\rightarrow S\ \underline{S}\ + \\
&\rightarrow \underline{S}\ a\ + \\
&\rightarrow S\ \underline{S}\ *\ a\ + \\
&\rightarrow \underline{S}\ a\ *\ a\ + \\
&\rightarrow S\ \underline{S}\ +\ a\ *\ a\ +
\end{aligned}
$$

Right Sentential Form	Handle	Reducing Production
S S S + a * +	SS+	S → SS+
S S + a * a +	SS+	S → SS+
a a a * a + +		

Example 3

- For the grammar S → S S + | S S * | a indicate the handle in each of the following right-sentential forms:
  a) S S S + a * +
  b) S S + a * a +
  c) a a a * a + +

S→S$\underline{S}$+
→SS$\underline{S}$++
→S$\underline{S}$a++
→SS$\underline{S}$*a++
→S$\underline{S}$a*a++
→$\underline{S}$aa*a++
→aaa*a++

Right Sentential Form	Handle	Reducing Production
S S S + a * +	SS+	S → SS+
S S + a * a +	SS+	S → SS+
a a a * a + +	a	S → a

Give bottom-up parse for the input string 000111 according to the grammar $S \rightarrow 0\,S\,1 \mid 0\,1$

STACK	INPUT	ACTION
$	0 0 0 1 1 1 $	

Give bottom-up parse for the input string 000111 according to the grammar S → 0 S 1 | 0 1

STACK	INPUT	ACTION
$	0 0 0 1 1 1 $	Shift
$ 0	0 0 1 1 1 $	Shift
$ 0 0	0 1 1 1 $	Shift
$ 0 0 0	1 1 1 $	Shift
$ 0 0 0 1	1 1 $	Reduce S → 0 1
$ 0 0 S	1 1 $	Shift
$ 0 0 S 1	1 $	Reduce S → 0 S 1
$ 0 S	1 $	Shift
$ 0 S 1	$	Reduce S → 0 S 1
$ S	$	Accept

Give bottom-up parse for the input string a a a * a + + according to the grammar S → S S + | S S * | a

STACK	INPUT	ACTION
$	a a a * a + + $	

Give bottom-up parse for the input string a a a * a + + according to the grammar S → S S + | S S * | a

STACK	INPUT	ACTION
$	a a a * a + + $	Shift
$ a	a a * a + + $	Reduce S → a
$ S	a a * a + + $	Shift
$ S a	a * a + + $	Reduce S → a
$ S S	a * a + + $	Shift
$ S S a	* a + + $	Reduce S → a
$ S S S	* a + + $	Shift
$ S S S *	a + + $	Reduce S → S S *
$ S S	a + + $	Shift
$ S S a	+ + $	Reduce S → a
$ S S S	+ + $	Shift
$ S S S +	+ $	Reduce S → S S +
$ S S	+ $	Shift
$ S S +	$	Reduce S → S S +
$ S	$	Accept

Shift Reduce Conflict

- There are some context-free grammars for which shift-reduce parsing cannot be used.
- Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack and also the next k input symbols, cannot decide whether to shift or to reduce (**a shift/reduce conflict**), or cannot decide which of several reductions to make (**a reduce/reduce conflict**).

Shift Reduce Conflict

- There are some syntactic constructs that give rise to such grammars.
- Technically, these grammars are not in the LR(k) class of grammars we refer to them as non-LR grammars.
  - The k in LR(k) refers to the number of symbols of lookahead on the input.
  - Grammars used in compiling usually fall in the LR(1) class, with one symbol of lookahead at most.
- An ambiguous grammar can never be LR.