Principles of Compiler Construction

Prof. Wen-jun LI
School of Computer Science and Engineering
lnslwj@mail.sysu.edu.cn

Lecture 6. Operator Precedence Parsing (OPP)

1. Bottom-Up Parsing
2. Abstract Model for Shift-Reduce Parsing
3. Operator-Precedence Parser
4. What Have We discarded?
5. Does OPP Disappear?

1. Bottom-Up Parsing

- A motivating example
  - Construct a parse tree beginning at the leaves and working up towards the root.
  - For the grammar

$$E \rightarrow E + T \mid T$$

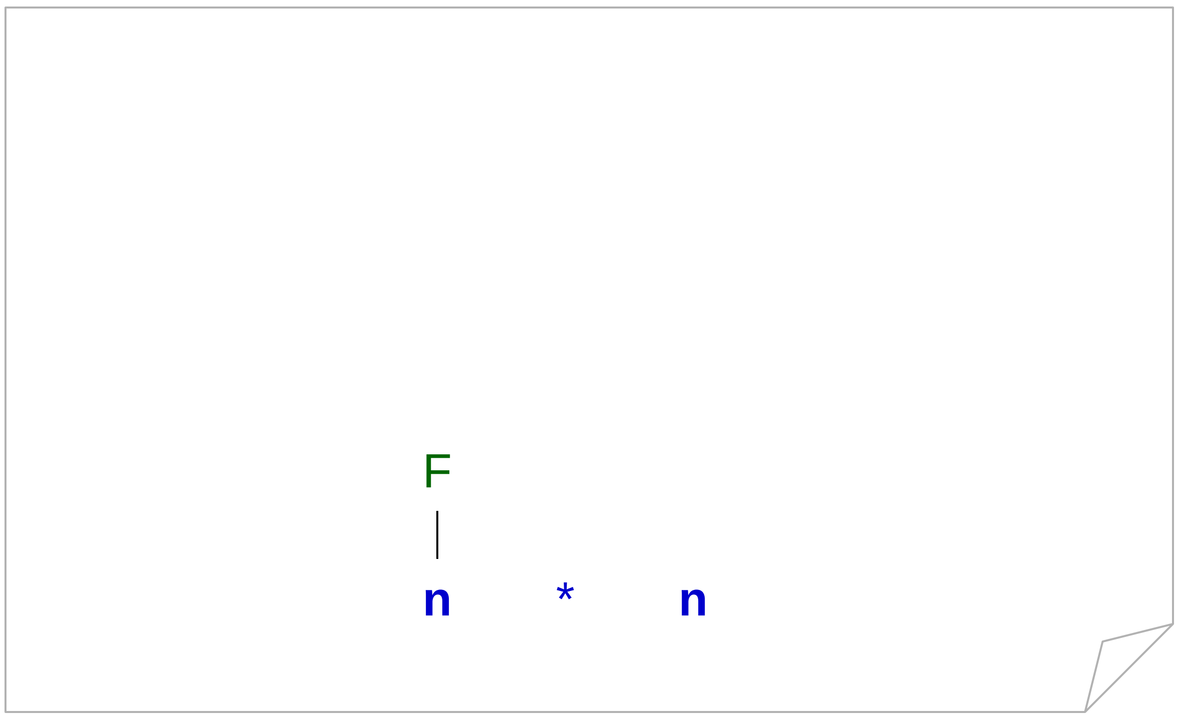
$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \mathbf{n}$$

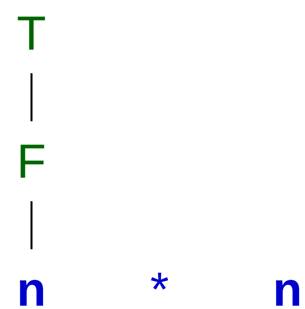
- Begin with leaves: **n * n**

n * **n**

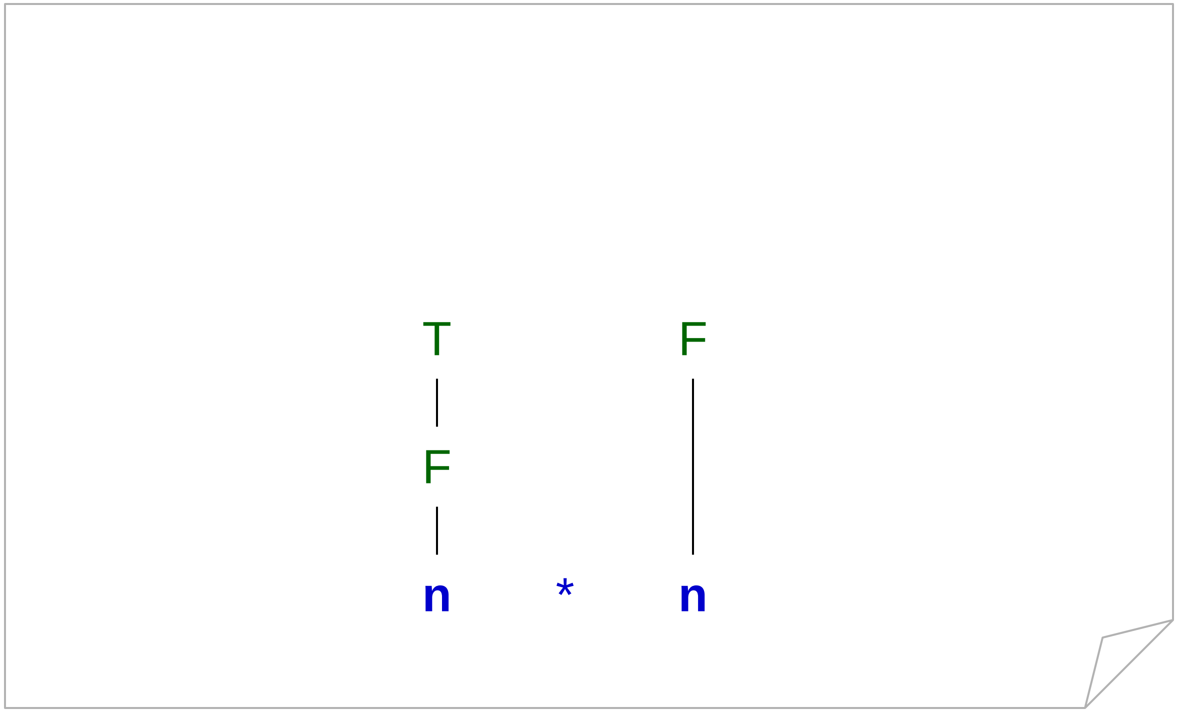
- The 1st **n** is the handle, after reduction we have the right sentential form: **F * n**



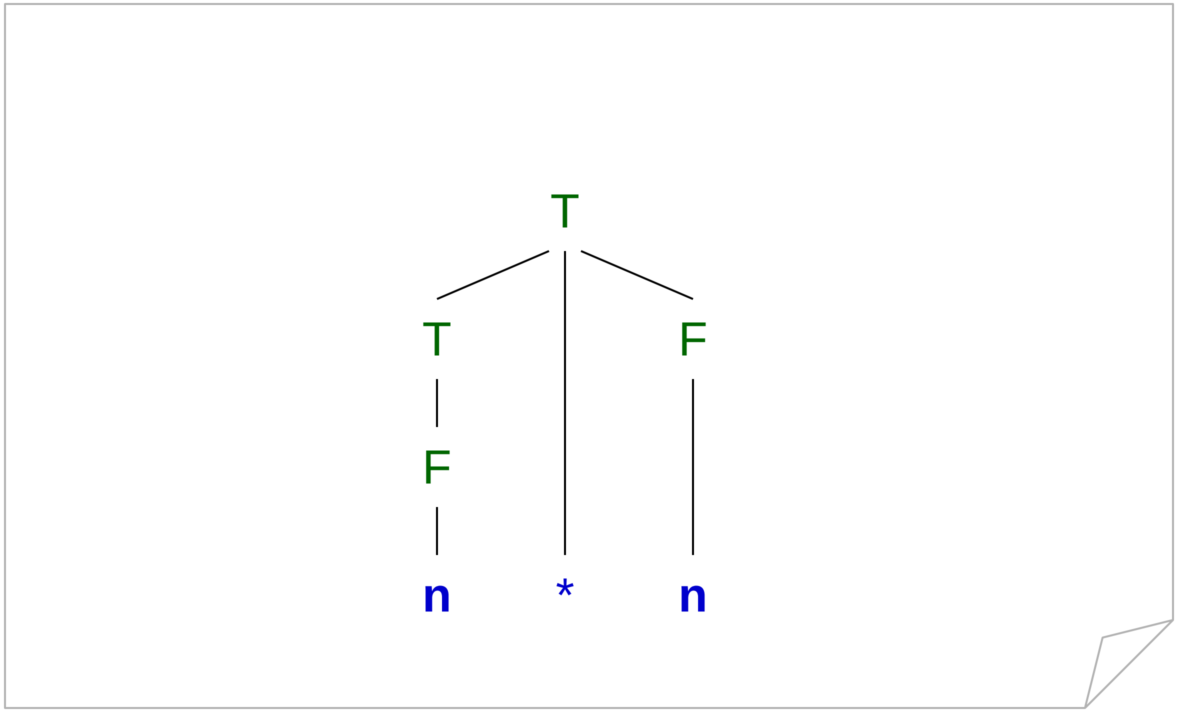
- **F** is the handle and we have: **T * n**



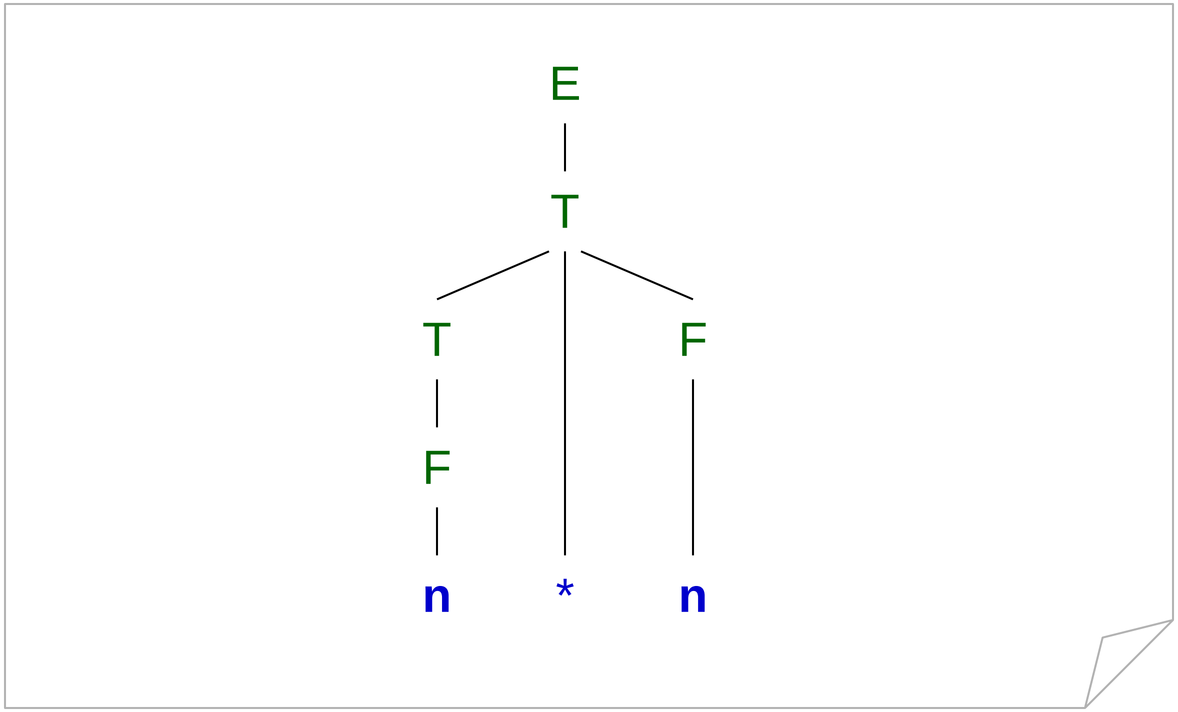
- **n** is the handle in **T * n**, we have: **T * F**



- **T * F** is the handle



- **T** is the handle

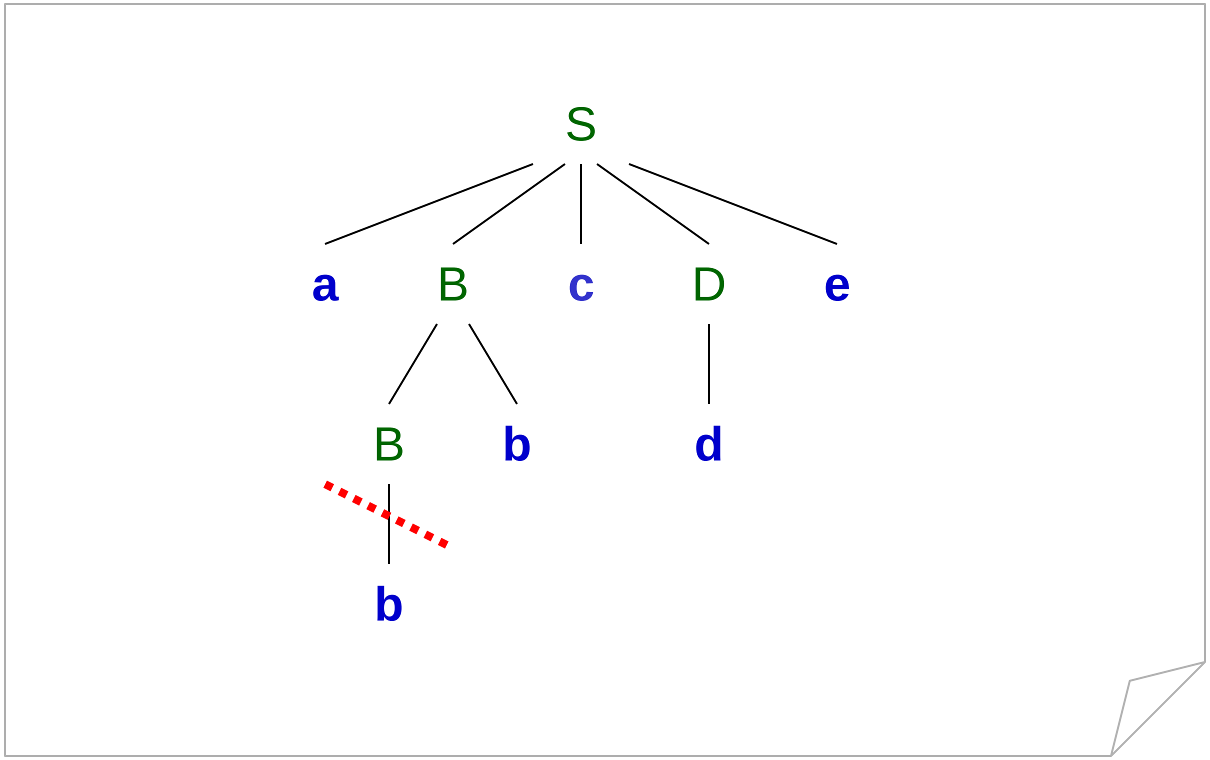


One More Example

- For the grammar

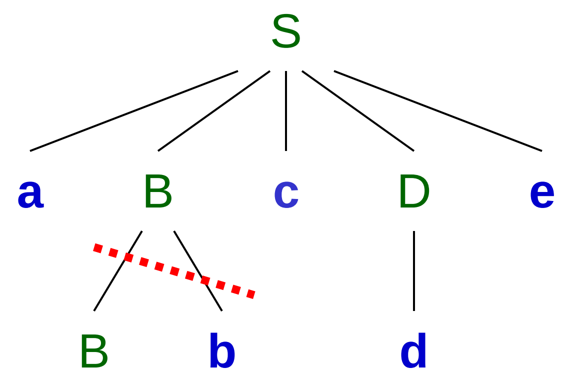
$$S \rightarrow \mathbf{a}\, B\, \mathbf{c}\, D\, \mathbf{e}$$
$$B \rightarrow B\, \mathbf{b} \mid \mathbf{b}$$
$$D \rightarrow \mathbf{d}$$

- Reduction = handle (**b**) pruning



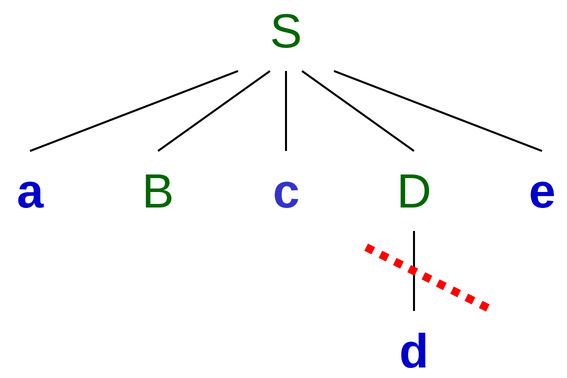
Right sentential form: **a b b c d e**

- **Bb** is the handle



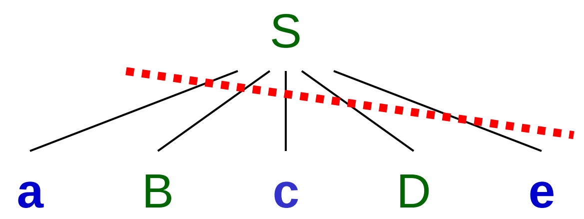
Right sentential form: **a B b c d e**

- **d** is the handle



Right sentential form: **a B c d e**

- **aBcDe** is the handle



Right sentential form: <u>**a B c D e**</u>

- Successfully reduced to the root

S

Right sentential form: **S**

Review

- Reducable string in a sentential form
  - Handle of a **right** sentential form
    - If $S \Rightarrow^*_{rm} \alpha A \omega \Rightarrow_{rm} \alpha\beta\omega$, then
    - production $A \rightarrow \beta$ in the position following α is a handle of $\alpha\beta\omega$.
  - Properties
    - The substring to the right of the handle (i.e. ω in the previous definition) must contain only terminals.
      - Note that it is a right-most (rm) derivation.

Review (cont')

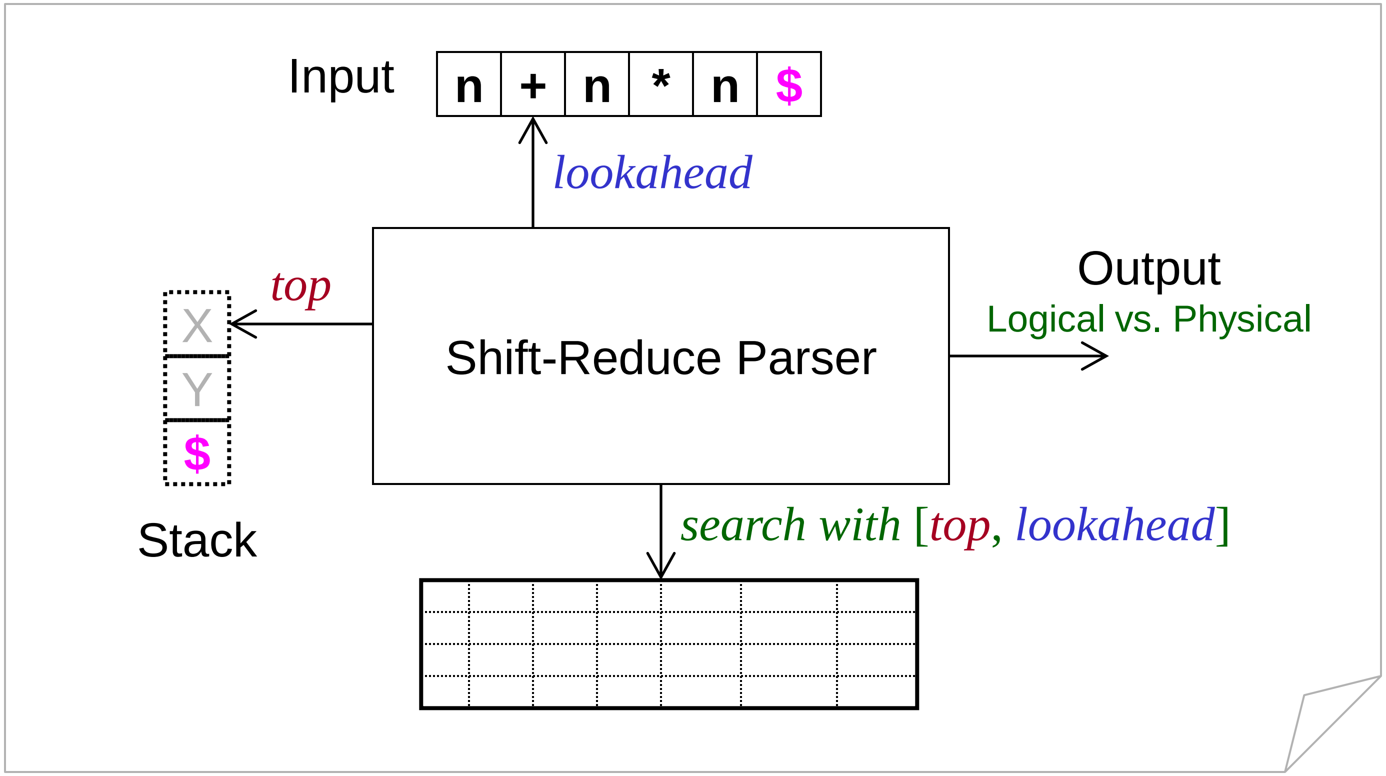
- Handle in a parse tree
  - A left-most two-level subtree.
  - Not practical. Only helpful to understand the concept.
- Critical problem in bottom-up parsing
  - How to find a reducable string in a sentential form?
    - LR parsing: handle (left-most simple phrase)
    - OPP: left-most prime phrase

2. Abstract Model for Shift-Reduce Parsing

- Symmetric to LL(1) parsing
  - Table-driven approach
    - Parsing decision making based on a table
    - LL(1) parsing table vs.
      - Operator-precedence relation table
      - LR(0)/SLR(1)/LALR(1)/LR(1) parsing table
  - Explicit stack
    - Top-down parsing: $[\$S, \omega\$] \Rightarrow [\$, \$]$
    - Bottom-up parsing: $[\$, \omega\$] \Rightarrow [\$S, \$]$
  - Actions
    - Top-down parsing: match-derive
    - Bottom-up parsing: shift-reduce

An Abstract Model

- A shift-reduce parser



A Motivating Example

- Given the previous grammar

$$S \rightarrow \mathbf{a}\, B\, \mathbf{c}\, D\, \mathbf{e}$$

$$B \rightarrow B\, \mathbf{b} \mid \mathbf{b}$$

$$D \rightarrow \mathbf{d}$$

- For the previous sentence

a b b c d e

A Motivating Example (cont')

Step	Stack	Input	Action	Output
1	**$**	**a b b c d e $**	shift	
2	**$ a**	**b b c d e $**	shift	
3	**$ a b**	**b c d e $**	reduce	B $\rightarrow$ **b**
4	**$ a** B	**b c d e $**	shift	
5	**$ a** B **b**	**c d e $**	reduce	B $\rightarrow$ B **b**
6	**$ a** B	**c d e $**	shift	
7	**$ a** B **c**	**d e $**	shift	
8	**$ a** B **c d**	**e $**	reduce	D $\rightarrow$ **d**
9	**$ a** B **c** D	**e $**	shift	
10	**$ a** B **c** D **e**	**$**	reduce	S $\rightarrow$ **a** B **c** D **e**
11	**$** S	**$**	accept	

More Motivating Examples

- Given the **ambiguous** grammar for expressions

$$\mathrm{E} \rightarrow \mathrm{E} + \mathrm{E} \mid \mathrm{E} * \mathrm{E} \mid (\mathrm{E}) \mid \mathbf{n}$$

- For the following sentence

$$\mathbf{n} + \mathbf{n} * \mathbf{n}$$

More Motivating Examples (cont')

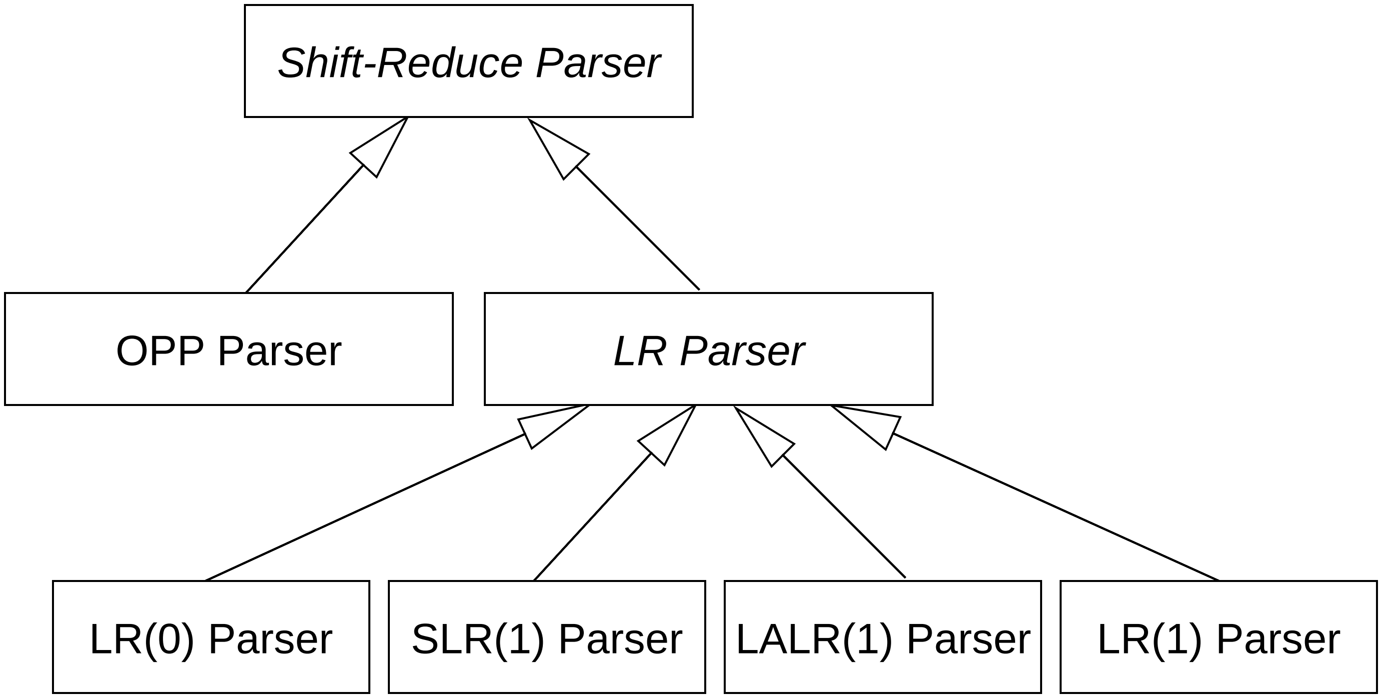
Step	Stack	Input	Action	Output
1	**$**	**n + n * n $**	shift	
2	**$ n**	**+ n * n $**	reduce	E $\rightarrow$ **n**
3	**$** E	**+ n * n $**	shift	
4	**$** E **+**	**n * n $**	shift	
5	**$** E **+ n**	*** n $**	reduce	E $\rightarrow$ **n**
6	**$** E **+** E	*** n $**	shift	
7	**$** E **+** E *****	**n $**	shift	
8	**$** E **+** E *** n**	**$**	reduce	E $\rightarrow$ **n**
9	**$** E **+** E ***** E	**$**	reduce	E $\rightarrow$ E ***** E
10	**$** E **+** E	**$**	reduce	E $\rightarrow$ E **+** E
11	**$** E	**$**	accept	

What Is Abstract in the Model?

- What is the form of the parsing table?
  - Operator-precedence relation table
  - LR parsing table
- What is the contents of the stack?
  - Operators
  - Parsing states
- How to determine a reducable substring on the top of the stack?
  - Left-most prime phrase
  - Handles

Concrete Implementations

- Implementations of the abstract model for shift-reduce parsing

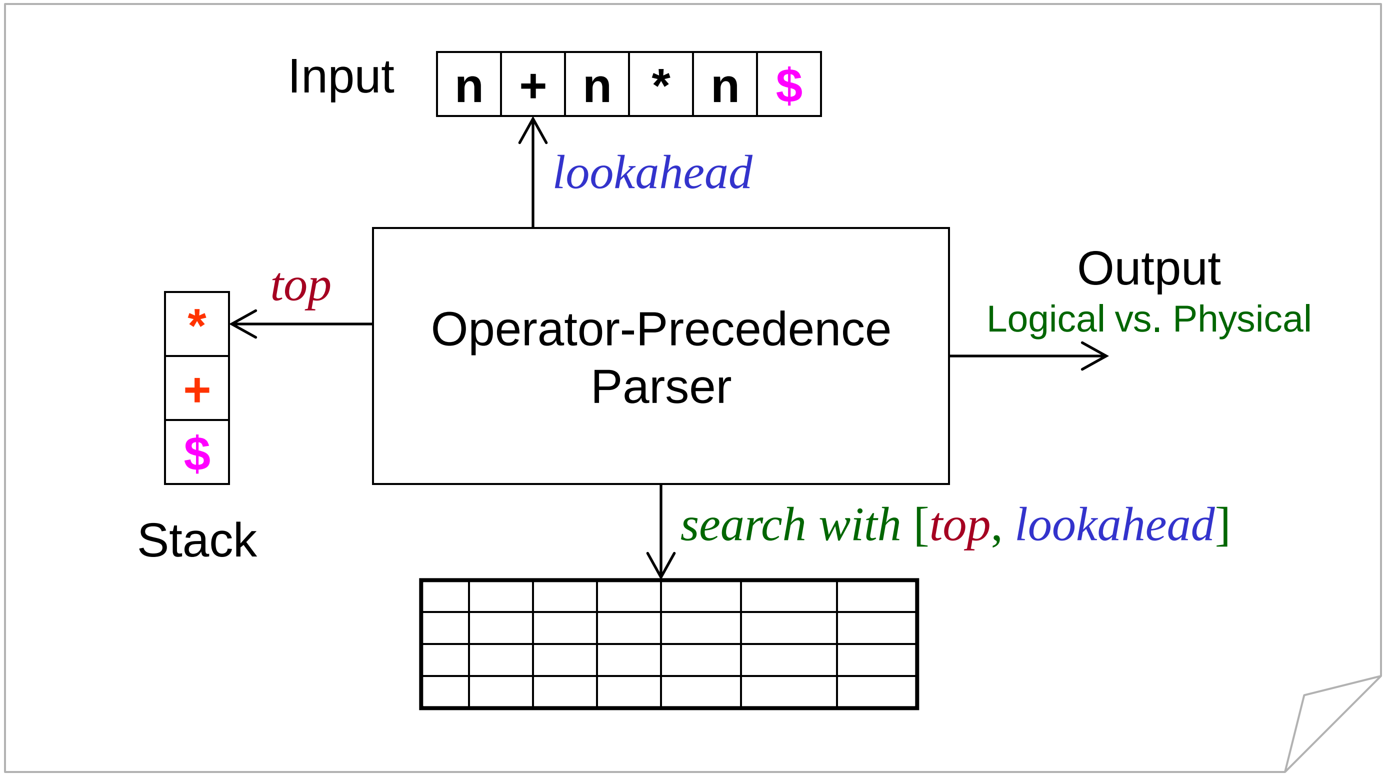


3. Operator-Precedence Parser

- Concrete implementations
  - Parsing table
    - Operator-precedence relations
  - Explicit stack
    - Operators
  - Reducable substring
    - Left-most prime phrases

A Concrete Model for OPP Parser

- Operator Precedence Parser



Operator Grammar

- A small but important class of grammars
  - There is no ε-production.
  - No production has two adjacent nonterminals.
- For example
  - Not an operator grammar

$$E \rightarrow E\,A\,E \mid (\,E\,) \mid -E \mid \mathbf{id}$$

$$A \rightarrow + \mid - \mid * \mid / \mid \wedge$$

  - An operator grammar

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \wedge E$$

$$\mid \; (\,E\,) \mid -E \mid \mathbf{id}$$

Operator-Precedence

- Three precedence relations
  - a $\prec$ b
  - a $\equiv$ b
  - a $\succ$ b
- Properties of the binary relation
  - Non-reflexive: “$+ \equiv +$” is not true.
  - Non-symmetric: “$+ \succ -$”, but “$- \succ +$”.
  - Non-transitive: “$> \succ <$” & “$< \succ =$”, but “$= \succ >$”
  - A partial relation

Operator-Precedence Ralation Table

	id	**+**	*****	**$**
id		$\succ$	$\succ$	$\succ$
+	$\prec$	$\succ$	$\prec$	$\succ$
*****	$\prec$	$\succ$	$\succ$	$\succ$
$	$\prec$	$\prec$	$\prec$	

Parsing Algorithm

```
initialize();
for (;;) {
    if (top == $ && lookahead == $)  accept();
    topOp = stack.topMostTerminal();
    if (topOp ⋖ lookahead || topOp ≡ lookahead) {  // shift
        stack.push(lookahead);
        lookahead = scanner.getNextToken();
    } else if (topOp ⋗ lookahead) {  // reduce
        do {
            topOp = stack.pop();
        } while (stack.topMostTerminal() ⋗ || ≡ topOp);
    } else  error();
}
```

Parsing a Sentence

Step	Stack	Input	Reference	Action	Output
1	**$**	**n + n * n $**	**$** $\prec$ **n**	shift	
2	**$ n**	**+ n * n $**	**n** $\succ$ **+**	reduce	E $\rightarrow$ **n**
3	**$** E	**+ n * n $**	**$** $\prec$ **+**	shift	
4	**$** E **+**	**n * n $**	**+** $\prec$ **n**	shift	
5	**$** E **+ n**	*** n $**	**n** $\succ$ *****	reduce	E $\rightarrow$ **n**
6	**$** E **+** E	*** n $**	**+** $\prec$ *****	shift	
7	**$** E **+** E *****	**n $**	***** $\prec$ **n**	shift	
8	**$** E **+** E *** n**	**$**	**n** $\succ$ **$**	reduce	E $\rightarrow$ **n**
9	**$** E **+** E ***** E	**$**	***** $\succ$ **$**	reduce	E $\rightarrow$ E ***** E
10	**$** E **+** E	**$**	**+** $\succ$ **$**	reduce	E $\rightarrow$ E **+** E
11	**$** E	**$**		accept	

More Practical OPP Table

	+	–	*	/	^	id	(	)	$\$$
+	$\succ$	$\succ$	$\prec$	$\prec$	$\prec$	$\prec$	$\prec$	$\succ$	$\succ$
–	$\succ$	$\succ$	$\prec$	$\prec$	$\prec$	$\prec$	$\prec$	$\succ$	$\succ$
*****	$\succ$	$\succ$	$\succ$	$\succ$	$\prec$	$\prec$	$\prec$	$\succ$	$\succ$
/	$\succ$	$\succ$	$\succ$	$\succ$	$\prec$	$\prec$	$\prec$	$\succ$	$\succ$
^	$\succ$	$\succ$	$\succ$	$\succ$	$\prec$	$\prec$	$\prec$	$\succ$	$\succ$
id	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$			$\succ$	$\succ$
(	$\prec$	$\prec$	$\prec$	$\prec$	$\prec$	$\prec$	$\prec$	$\equiv$	
)	$\succ$	$\succ$	$\succ$	$\succ$	$\succ$			$\succ$	$\succ$
$\$$	$\prec$	$\prec$	$\prec$	$\prec$	$\prec$	$\prec$	$\prec$		

Another Form of OPP Table

	+	–	*	/	^	id	(	)	$
+	reduce	reduce	shift	shift	shift	shift	shift	reduce	reduce
–	reduce	reduce	shift	shift	shift	shift	shift	reduce	reduce
*****	reduce	reduce	reduce	reduce	shift	shift	shift	reduce	reduce
/	reduce	reduce	reduce	reduce	shift	shift	shift	reduce	reduce
^	reduce	reduce	reduce	reduce	shift	shift	shift	reduce	reduce
id	reduce	reduce	reduce	reduce	reduce			reduce	reduce
(	shift	shift	shift	shift	shift	shift	shift	shift	
)	reduce	reduce	reduce	reduce	reduce			reduce	reduce
$	shift	shift	shift	shift	shift	shift	shift		

Error Recovery in OPP

Phrase-Level Recovery

	+	–	*	/	^	id	(	)	$
+	reduce	reduce	shift	shift	shift	shift	shift	reduce	reduce
–	reduce	reduce	shift	shift	shift	shift	shift	reduce	reduce
*****	reduce	reduce	reduce	reduce	shift	shift	shift	reduce	reduce
/	reduce	reduce	reduce	reduce	shift	shift	shift	reduce	reduce
^	reduce	reduce	reduce	reduce	shift	shift	shift	reduce	reduce
id	reduce	reduce	reduce	reduce	reduce	error1	error1	reduce	reduce
(	shift	shift	shift	shift	shift	shift	shift	shift	error2
)	reduce	reduce	reduce	reduce	reduce	error1	error1	reduce	reduce
$	shift	shift	shift	shift	shift	shift	shift	error3	accept

Handling Overloading Operators

- Overloading unary and binary operators
  - **id – id * id**
  - **id * – id**
- Distinguish or resolve overloading by the scanner
  - Remember the previous processed token.

4. What Have We Discarded?

- Automatically table construction
  - Construct an operator-precedence relation table from an unambiguous grammar of expressions.
  - Trade-off: cost vs. cost
    - Writing an unambiguous grammar
    - Constructing an precedence relation table

Saving Table Space

- Precedence functions
  - Pack an $n \times n$ matrix into a $2 \times n$ matrix.
  - Save some space to store the parsing table.
  - But something is missing.
    - Each pair of operators must have a relation.

	+	–	*	/	^	id	(	)	$
f	2	2	4	4	4	6	0	6	0
g	1	1	3	3	5	5	5	0	0

5. Does OPP Disappear?

- OPP has disappeared from many modern compiler textbooks, e.g.
  - DBv2 (while DBv1 concerned with OPP)
  - Tiger book
  - The Art of Compiler Design
  - etc.

Reasons for Ignoring OPP

- Some reasons are:
  - OPP only handles very simple languages.
  - OPP isn't very good at discovering when a string isn't in the language.
  - Operator precedences have been incorporated into LR parsers, so writing an LR parser for expressions with many precedence levels isn't very hard.
  - Space isn't nearly as much an issue as it used to be.
  - OPP is often taught in introductory Data Structures courses.

But OPP Still Works

- Even in commercial applications, specially combination of OPP and recursive parsing
  - Some Sun **javac** compiler
    - Recursive parser parses expressions coarsely.
    - OPP parses the binary operators correctly.
  - Some open source Verilog simulator
    - OPP parses Verilog expressions as a separate section of a hand coded recursive descent parser.
    - Verilog expressions are complex and language is irregular and non context-free.
  - The early Symantec's THINK C compiler
  - The original C compiler at Bell Labs (reported)

Exercise 6.1

- Consider the grammar

$$S \rightarrow (L) \mid \mathbf{a}$$

$$L \rightarrow L, S \mid S$$

- We have the following operator-precedence relations for the grammar.
  - Show the detailed process of the parsing of the sentence **`(a, (a, a))`**, follow the style in the previous slides.

	a	(	)	,	$
a			$\succ$	$\succ$	$\succ$
(	$\prec$	$\prec$	$\equiv$	$\prec$	
)			$\succ$	$\succ$	$\succ$
,	$\prec$	$\prec$	$\succ$	$\succ$	
$	$\prec$	$\prec$			

Further Reading

- Dragon Book, 2^{nd} Edition (DBv2)
  - Comprehensive Reading:
    - Section 4.5 for the abstract model for shift-reduce parsing.
- Dragon Book, 1^{nd} Edition (DBv1)
  - Comprehensive Reading:
    - Section 4.6 for the OPP details.
  - Skip Reading:
    - Section 4.6 on precedence functions.

Enjoy the Course!