

# Compiler Construction

## SMD163

### Lecture 4: Introduction to Parsing

Viktor Leijon & Peter Jonsson with slides by Johan Nordlander.  
Contains material generously provided by Mark P. Jones

**COMPUTER SCIENCE  
AND ELECTRICAL ENGINEERING**  
LULEÅ UNIVERSITY OF TECHNOLOGY

1

## Syntax Analysis:

)	)		f	o	r		e	l	s	e		1	2	3
4	5	;		e	l	s	e		e	l	s	e	+	+

Lexical analysis

)	)	for	else	12345	;	else	else	++
---	---	-----	------	-------	---	------	------	----

?

2

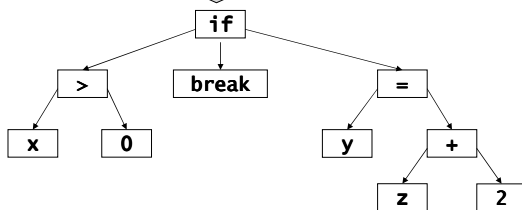
## Syntax Analysis:

i	f		x	>	0		t	h	e	n		b	r	e
a	k	;		e	l	s	e		y	=	z	+	2	;

Lexical analysis

if	x	>	0	then	break	;	else	y	=	z	+	2	;
----	---	---	---	------	-------	---	------	---	---	---	---	---	---

Parsing



3

## Review: Regular Languages

- ◆ Regular languages are formed from: single characters, the empty string, sequencing, alternatives, and repetition.
- ◆ Regular languages are good for describing lexical structure.
- ◆ We can recognize words in a regular language using simple, efficient finite automata.
- ◆ But the set of regular languages is quite limited.

4

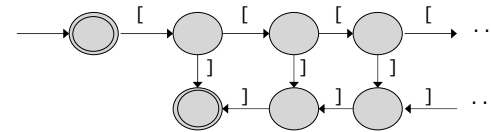
## A Non-Regular Language

- ◆ Brackets =  $\{\epsilon\} \cup \{ [b] \mid b \in \text{Brackets} \}$
- ◆ So the words in Brackets are  $\epsilon, [], [[]], [[[]]], [[[]]]], [[[]]]], \dots$
- ◆ Is this a regular language?
- ◆ Is there a regular expression  $r$  that describes all strings in Brackets?

5

## No!

- ◆ If Brackets is regular, then we can recognize it using a finite automaton.



- ◆ Suppose that we reach the same state  $s$  after either  $n$  or  $m$  open brackets  $[$ .
- ◆ Then we can reach an accept state from  $s$  after either  $n$  or  $m$  close brackets  $]$ .
- ◆ So, either the automaton accepts bad strings, or  $m=n$ .
- ◆ Hence we need infinitely many states ... not a regular language.

6

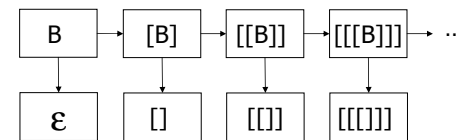
## Iteration vs Recursion:

- ◆ Regular expressions don't allow recursion ... just iteration.
- ◆ But it is easy enough to give a simple, recursive characterization for  $B \in \text{Brackets}$ :

$B \rightarrow \epsilon$       $B$  is empty  
 $B \rightarrow [ B ]$     $B$  consists of an opening  $[$ ,  
 another element of Brackets,  
 and then a closing  $]$ .

7

## Generating Brackets:



- ◆ The definition can be viewed as rewrite rules:  
 $B \rightarrow \epsilon$      replace each  $B$  with  $\epsilon$   
 $B \rightarrow [ B ]$    replace each  $B$  with  $[ B ]$
- ◆  $B$  derives  $s$  if the string  $s$  can be obtained from  $B$  by repeated replacement.

8

## Context-Free Grammars (CFGs):

- ◆ A context-free grammar  $(T, N, P, s)$  consists of:
  - A set  $T$  of terminal symbols;
  - A set  $N$  of nonterminal symbols;
  - A set  $P$  of productions, each of which is a rule of the form:  
 $n \rightarrow w$   
where  $n \in N$ , and  $w \in (T \cup N)^*$ ;
  - A start symbol  $s \in N$ .

9

## Brackets is a CFG:

- ◆ Here is a context-free grammar for Brackets:

Terminals: [ and ]  
Nonterminals: B  
Productions:  $B \rightarrow \epsilon$  and  $B \rightarrow [B]$   
Start symbol: B

- ◆ I.e., Brackets =  $(\{[, ]\}, \{B\}, \{B \rightarrow \epsilon, B \rightarrow [B]\}, B)$
- ◆ But what is the relationship between languages (i.e., sets of strings) and CFGs like the 4-tuple above?

10

## The Language of a CFG:

- ◆ Each context-free grammar  $(T, N, P, s)$  generates a language  $L$ : the set of all strings in  $T^*$  that can be derived from  $s$ .
- ◆ A language is context-free if it can be described by a context-free grammar.
- ◆ Brackets is a context-free language.

11

## Why "context-free"?

- ◆ Because the productions in a context-free grammar can be expanded at any point in a derivation.
- ◆ Incidentally, "context-free" does not necessarily mean "easy to parse" ...

12

## Why Bother With Regular Exprs?

- ◆ If CFGs can express everything that regular expressions can, why do we bother with regular expressions?
- ◆ Because regular expressions are:
  - Easier to understand (perhaps);
  - Easier to recognize;
  - Plenty good enough for many applications!

13

## A Language of Expressions:

- ◆ Many computer languages are naturally described as context-free languages.
- ◆ A simple language of expressions:

$E \rightarrow n$  (n is an integer literal)  
 $E \rightarrow E + E$   
 $E \rightarrow E - E$   
 $E \rightarrow (E)$

14

## Deriving Expressions:

For example:  $3-(2+1)$  is an expression

$E \rightarrow E + E$   
 $\rightarrow E + (E)$   
 $\rightarrow E + (E - E)$   
 $\rightarrow E + (E - 1)$   
 $\rightarrow E + (2 - 1)$   
 $\rightarrow 3 + (2 - 1)$

15

## Deriving Expressions:

For example:  $3-(2+1)$  is an expression

$E \rightarrow E - E$   
 $\rightarrow E + (E)$   
 $\rightarrow E + (E - E)$   
 $\rightarrow E + (E - 1)$   
 $\rightarrow E + (2 - 1)$   
 $\rightarrow 3 + (2 - 1)$

16

## Deriving Expressions:

For example:  $3-(2+1)$  is an expression

$$\begin{aligned} E &\rightarrow E - E \\ &\rightarrow E - (E) \\ &\rightarrow E + (E - E) \\ &\rightarrow E + (E - 1) \\ &\rightarrow E + (2 - 1) \\ &\rightarrow 3 + (2 - 1) \end{aligned}$$

17

## Deriving Expressions:

For example:  $3-(2+1)$  is an expression

$$\begin{aligned} E &\rightarrow E - E \\ &\rightarrow E - (E) \\ &\rightarrow E - (E + E) \\ &\rightarrow E + (E - 1) \\ &\rightarrow E + (2 - 1) \\ &\rightarrow 3 + (2 - 1) \end{aligned}$$

18

## Deriving Expressions:

For example:  $3-(2+1)$  is an expression

$$\begin{aligned} E &\rightarrow E - E \\ &\rightarrow E - (E) \\ &\rightarrow E - (E + E) \\ &\rightarrow E - (E + 1) \\ &\rightarrow E + (2 - 1) \\ &\rightarrow 3 + (2 - 1) \end{aligned}$$

19

## Deriving Expressions:

For example:  $3-(2+1)$  is an expression

$$\begin{aligned} E &\rightarrow E - E \\ &\rightarrow E - (E) \\ &\rightarrow E - (E + E) \\ &\rightarrow E - (E + 1) \\ &\rightarrow E - (2 + 1) \\ &\rightarrow 3 + (2 - 1) \end{aligned}$$

20

## Deriving Expressions:

For example:  $3-(2+1)$  is an expression

$$\begin{aligned} E &\rightarrow E - E \\ &\rightarrow E - (E) \\ &\rightarrow E - (E + E) \\ &\rightarrow E - (E + 1) \\ &\rightarrow E - (2 + 1) \\ &\rightarrow 3 - (2 + 1) \end{aligned}$$

21

## Deriving Expressions:

For example:  $3-(2+1)$  is an expression

$$\begin{aligned} E &\rightarrow E + E \\ &\rightarrow 3 + E \\ &\rightarrow 3 + (E) \\ &\rightarrow 3 + (E - E) \\ &\rightarrow 3 + (2 - E) \\ &\rightarrow 3 + (2 - 1) \end{aligned}$$

22

## Deriving Expressions:

For example:  $3-(2+1)$  is an expression

$$\begin{aligned} E &\rightarrow E - E \\ &\rightarrow 3 + E \\ &\rightarrow 3 + (E) \\ &\rightarrow 3 + (E - E) \\ &\rightarrow 3 + (2 - E) \\ &\rightarrow 3 + (2 - 1) \end{aligned}$$

23

## Deriving Expressions:

For example:  $3-(2+1)$  is an expression

$$\begin{aligned} E &\rightarrow E - E \\ &\rightarrow 3 - E \\ &\rightarrow 3 + (E) \\ &\rightarrow 3 + (E - E) \\ &\rightarrow 3 + (2 - E) \\ &\rightarrow 3 + (2 - 1) \end{aligned}$$

24

## Deriving Expressions:

For example:  $3-(2+1)$  is an expression

$$\begin{aligned} E &\rightarrow E - E \\ &\rightarrow 3 - E \\ &\rightarrow 3 - (E) \\ &\rightarrow 3 + (E - E) \\ &\rightarrow 3 + (2 - E) \\ &\rightarrow 3 + (2 - 1) \end{aligned}$$

25

## Deriving Expressions:

For example:  $3-(2+1)$  is an expression

$$\begin{aligned} E &\rightarrow E - E \\ &\rightarrow 3 - E \\ &\rightarrow 3 - (E) \\ &\rightarrow 3 - (E + E) \\ &\rightarrow 3 + (2 - E) \\ &\rightarrow 3 + (2 - 1) \end{aligned}$$

26

## Deriving Expressions:

For example:  $3-(2+1)$  is an expression

$$\begin{aligned} E &\rightarrow E - E \\ &\rightarrow 3 - E \\ &\rightarrow 3 - (E) \\ &\rightarrow 3 - (E + E) \\ &\rightarrow 3 - (2 + E) \\ &\rightarrow 3 + (2 - 1) \end{aligned}$$

27

## Deriving Expressions:

For example:  $3-(2+1)$  is an expression

$$\begin{aligned} E &\rightarrow E - E \\ &\rightarrow 3 - E \\ &\rightarrow 3 - (E) \\ &\rightarrow 3 - (E + E) \\ &\rightarrow 3 - (2 + E) \\ &\rightarrow 3 - (2 + 1) \end{aligned}$$

28

## One Word, Many Derivations:

- ◆ We derived the same expression in two different ways:
  - In a right-most derivation, we always replace the right-most nonterminal.
  - In a left-most derivation, we always replace the left-most nonterminal.
- ◆ There are many other choices between these two extremes.

29

## Deriving Expressions:

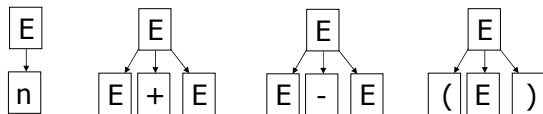
$E \rightarrow E - E$	$E \rightarrow E + E$
$\rightarrow E - E + E$	$\rightarrow E - E + E$
$\rightarrow E - E + 1$	$\rightarrow 3 - E + E$
$\rightarrow E - 2 + 1$	$\rightarrow 3 - 2 + E$
$\rightarrow 3 - 2 + 1$	$\rightarrow 3 - 2 + 1$

- ◆ Both derivations show that  $3-2+1$  is an expression.
- ◆ But this time there is a fundamental difference between them...

30

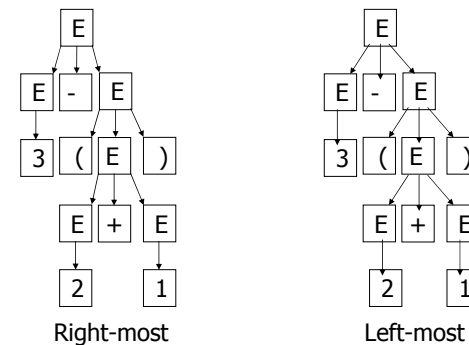
## Productions in Graphical Form:

- ◆ To understand the essential structure of a derivation, we will use a graphical notation to represent productions in a grammar:



31

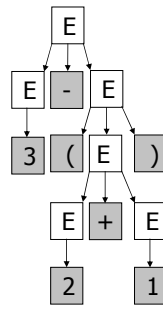
## $3-(2+1)$ Revisited:



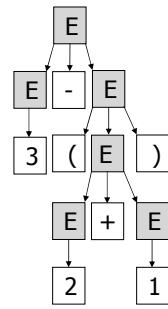
But the end result is the same!

32

## Parse Trees:



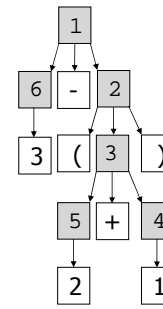
Leaves are terminals



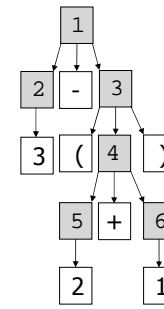
Interior nodes are nonterminals

33

## Right-most vs Left-most:



Right-most



Left-most

The only real difference between them is the order in which we construct the nodes ...

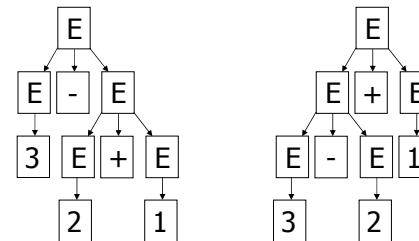
34

## CFGs and Parse Trees:

- ◆ Context-free grammars don't just define languages (i.e., sets of strings) ...
- ◆ ... they really define sets of trees.
- ◆ The strings in the corresponding context-free language can be recovered from the leaf nodes of the trees.
- ◆ Parsing works in reverse: start with a string, and try to construct the tree.

35

## 3-2+1 Revisited:



Fundamentally Different Structures!

36

## Ambiguity:

- ◆ A grammar is ambiguous if there is a string  $w$  in the corresponding language with more than one parse tree.
- ◆ Our grammar for expressions is ambiguous because the string "3-2+1" has two distinct parse trees.

37

## Dealing with Ambiguity:

- ◆ Does it matter?
  - If any parse tree is as good (i.e., means the same) as any other, then just take whichever one we can get.
  - Example:  $3+2+1$
- ◆ If different parse trees have different meanings, then we need to choose between them:
  - Disambiguating rules (e.g., operator precedence and associativity);
  - Rewrite the grammar to avoid ambiguity.

38

## An Unambiguous Grammar:

- ◆ A simple language of expressions:

$E \rightarrow E + A$       Expressions

$E \rightarrow E - A$

$E \rightarrow A$

$A \rightarrow (E)$       Atoms

$A \rightarrow n$       ( $n$  is an integer literal)

39

## N.B.

- ◆ Notice that we can have multiple grammars describing the same language.
- ◆ It doesn't make sense to describe a context-free language as unambiguous; there may be both ambiguous and unambiguous grammars for it.

40

## Strategies for Parsing:

- ◆ We will focus on two different strategies that can be used to build parsers:
  - Top-down parsing;
  - Bottom-up parsing.
- ◆ We can use tools to build parsers, but we need to understand how they work so that we can debug the generated parsers ...

41

## Top-down Parsing:

- ◆ General Strategy:
  - Know which nonterminal you are trying to match at each stage.
  - Break down the token stream into pieces that match against productions.
  - Begin by looking for something that matches the start symbol.

42

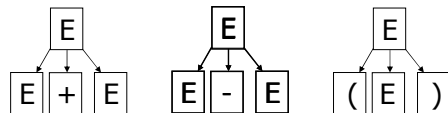
## Top-down Parsing:

Looking for E ... E

Tokens:

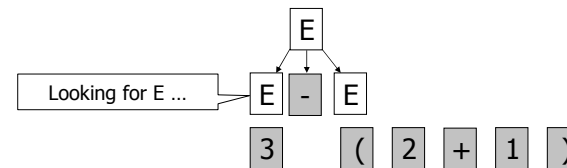
3 - ( 2 + 1 )

Patterns:

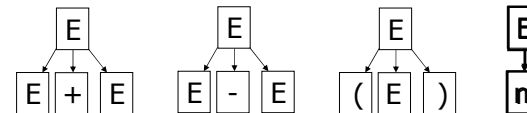


43

## Top-down Parsing:

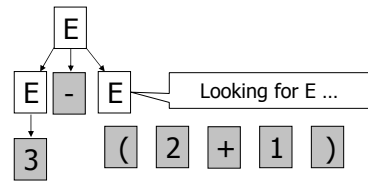


Patterns:

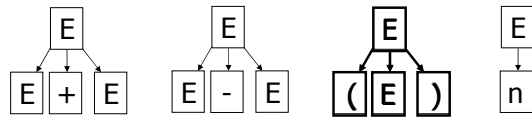


44

## Top-down Parsing:

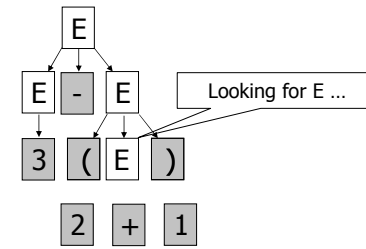


Patterns:

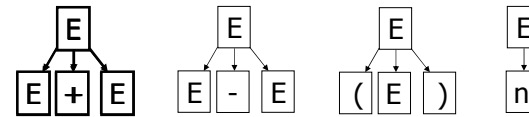


45

## Top-down Parsing:

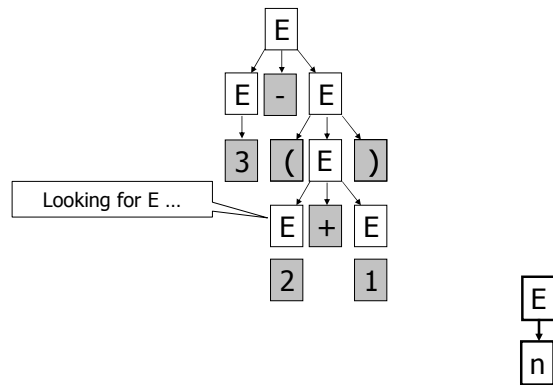


Patterns:



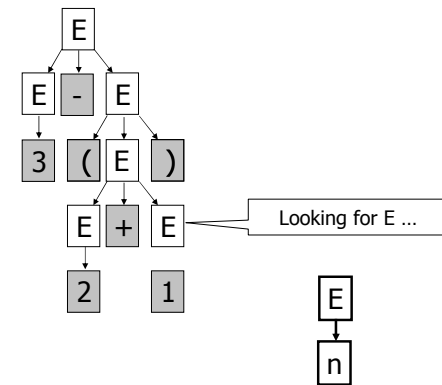
46

## Top-down Parsing:



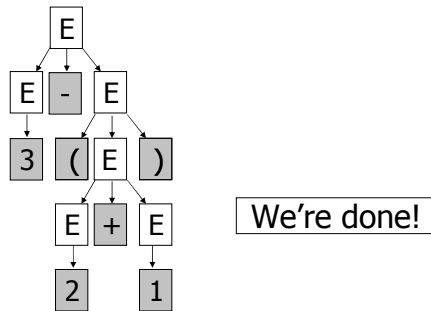
47

## Top-down Parsing:



48

## Top-down Parsing:



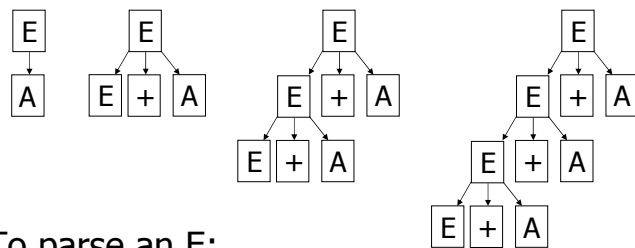
49

## Observations:

- ◆ The trick is in knowing which production to pick at each stage.
- ◆ We've been making our decisions in this example based on the token stream as a whole ...
- ◆ This is not realistic – we should aim to read the tokens from left to right only.

50

## Looking for an E?



To parse an E:

- parse an integer or (E)
- then parse zero or more + A

51

## Yet Another Grammar:

- ◆ We can define our language of expressions using the following alternative grammar:

$$E \rightarrow A E'$$

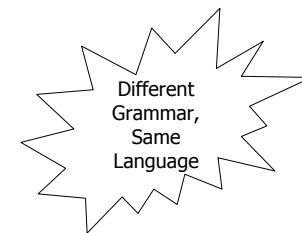
$$E' \rightarrow + A E'$$

$$E' \rightarrow - A E'$$

$$E' \rightarrow \epsilon$$

$$A \rightarrow (E)$$

$$A \rightarrow n$$



52

## Top-down Parsing:

Tokens: 3 - ( 2 + 1 )

Productions:            Looking for:  
 $E \rightarrow A E'$              $E \rightarrow \_A E'$   
  
 $E' \rightarrow + A E'$   
 $E' \rightarrow - A E'$   
 $E' \rightarrow \epsilon$   
  
 $A \rightarrow (E)$   
 $A \rightarrow n$

53

## Top-down Parsing:

Tokens: 3 - ( 2 + 1 )

Productions:            Looking for:  
 $E \rightarrow A E'$              $E \rightarrow \_A E'$   
  
 $E' \rightarrow + A E'$              $A \rightarrow \_n$   
 $E' \rightarrow - A E'$             Matches!  
 $E' \rightarrow \epsilon$   
  
 $A \rightarrow (E)$   
 $A \rightarrow n$

54

## Top-down Parsing:

Tokens: - ( 2 + 1 )

Productions:            Looking for:  
 $E \rightarrow A E'$              $E \rightarrow A\_E'$   
  
 $E' \rightarrow + A E'$   
 $E' \rightarrow - A E'$   
 $E' \rightarrow \epsilon$   
  
 $A \rightarrow (E)$   
 $A \rightarrow n$

55

## Top-down Parsing:

Tokens: - ( 2 + 1 )

Productions:            Looking for:  
 $E \rightarrow A E'$              $E \rightarrow A\_E'$   
  
 $E' \rightarrow + A E'$   
 $E' \rightarrow - A E'$   
 $E' \rightarrow \epsilon$   
  
 $A \rightarrow (E)$   
 $A \rightarrow n$

56

## Top-down Parsing:

Tokens:

- ( 2 + 1 )

Productions:

$E \rightarrow A E'$

$E' \rightarrow + A E'$

$E' \rightarrow - A E'$

$E' \rightarrow \epsilon$

$A \rightarrow (E)$

$A \rightarrow n$

Looking for:

$E \rightarrow A\_E'$

$E' \rightarrow \_ - A E'$

Matches!

57

## Top-down Parsing:

Tokens:

( 2 + 1 )

Productions:

$E \rightarrow A E'$

$E' \rightarrow + A E'$

$E' \rightarrow - A E'$

$E' \rightarrow \epsilon$

$A \rightarrow (E)$

$A \rightarrow n$

Looking for:

$E \rightarrow A\_E'$

$E' \rightarrow - \_ A E'$

58

## Top-down Parsing:

Tokens:

( 2 + 1 )

Productions:

$E \rightarrow A E'$

$E' \rightarrow + A E'$

$E' \rightarrow - A E'$

$E' \rightarrow \epsilon$

$A \rightarrow (E)$

$A \rightarrow n$

Looking for:

$E \rightarrow A\_E'$

$E' \rightarrow - \_ A E'$

$A \rightarrow \_ (E)$

Matches!

59

## Top-down Parsing:

Tokens:

2 + 1 )

Productions:

$E \rightarrow A E'$

$E' \rightarrow + A E'$

$E' \rightarrow - A E'$

$E' \rightarrow \epsilon$

$A \rightarrow (E)$

$A \rightarrow n$

Looking for:

$E \rightarrow A\_E'$

$E' \rightarrow - \_ A E'$

$A \rightarrow (\_ E)$

60

## Top-down Parsing:

Tokens:

2 + 1 )

Productions:

$E \rightarrow A E'$

$E' \rightarrow + A E'$

$E' \rightarrow - A E'$

$E' \rightarrow \epsilon$

$A \rightarrow (E)$

$A \rightarrow n$

Looking for:

$E \rightarrow A\_E'$

$E' \rightarrow -\_A E'$

$A \rightarrow (\_E)$

$E \rightarrow \_A E'$

61

## Top-down Parsing:

Tokens:

2 + 1 )

Productions:

$E \rightarrow A E'$

$E' \rightarrow + A E'$

$E' \rightarrow - A E'$

$E' \rightarrow \epsilon$

$A \rightarrow (E)$

$A \rightarrow n$

Looking for:

$E \rightarrow A\_E'$

$E' \rightarrow -\_A E'$

$A \rightarrow (\_E)$

$E \rightarrow \_A E'$

$A \rightarrow \_n$

Matches!

62

## Top-down Parsing:

Tokens:

+ 1 )

Productions:

$E \rightarrow A E'$

$E' \rightarrow + A E'$

$E' \rightarrow - A E'$

$E' \rightarrow \epsilon$

$A \rightarrow (E)$

$A \rightarrow n$

Looking for:

$E \rightarrow A\_E'$

$E' \rightarrow -\_A E'$

$A \rightarrow (\_E)$

$E \rightarrow \_A E'$

63

## Top-down Parsing:

Tokens:

+ 1 )

Productions:

$E \rightarrow A E'$

$E' \rightarrow + A E'$

$E' \rightarrow - A E'$

$E' \rightarrow \epsilon$

$A \rightarrow (E)$

$A \rightarrow n$

Looking for:

$E \rightarrow A\_E'$

$E' \rightarrow -\_A E'$

$A \rightarrow (\_E)$

$E \rightarrow A\_E'$

$E' \rightarrow \_+ A E'$

Matches!

64

## Top-down Parsing:

Tokens:

1 )

Productions:

$E \rightarrow A E'$

$E' \rightarrow + A E'$

$E' \rightarrow - A E'$

$E' \rightarrow \epsilon$

$A \rightarrow (E)$

$A \rightarrow n$

Looking for:

$E \rightarrow A\_E'$

$E' \rightarrow -\_A E'$

$A \rightarrow (\_E)$

$E \rightarrow A\_E'$

$E' \rightarrow +\_A E'$

65

## Top-down Parsing:

Tokens:

1 )

Productions:

$E \rightarrow A E'$

$E' \rightarrow + A E'$

$E' \rightarrow - A E'$

$E' \rightarrow \epsilon$

$A \rightarrow (E)$

$A \rightarrow n$

Looking for:

$E \rightarrow A\_E'$

$E' \rightarrow -\_A E'$

$A \rightarrow (\_E)$

$E \rightarrow A\_E'$

$E' \rightarrow +\_A E'$

$A \rightarrow \_n$

Matches!

66

## Top-down Parsing:

Tokens:

)

Productions:

$E \rightarrow A E'$

$E' \rightarrow + A E'$

$E' \rightarrow - A E'$

$E' \rightarrow \epsilon$

$A \rightarrow (E)$

$A \rightarrow n$

Looking for:

$E \rightarrow A\_E'$

$E' \rightarrow -\_A E'$

$A \rightarrow (\_E)$

$E \rightarrow A\_E'$

$E' \rightarrow +\_A E'$

67

## Top-down Parsing:

Tokens:

)

Productions:

$E \rightarrow A E'$

$E' \rightarrow + A E'$

$E' \rightarrow - A E'$

$E' \rightarrow \epsilon$

$A \rightarrow (E)$

$A \rightarrow n$

Looking for:

$E \rightarrow A\_E'$

$E' \rightarrow -\_A E'$

$A \rightarrow (E\_)$

Matches!

68

## Top-down Parsing:

Tokens:

Productions:

$E \rightarrow A E'$

$E' \rightarrow + A E'$

$E' \rightarrow - A E'$

$E' \rightarrow \epsilon$

$A \rightarrow (E)$

$A \rightarrow n$

Looking for:

$E \rightarrow A\_E'$

$E' \rightarrow - A\_E'$

69

## Top-down Parsing:

Tokens:

All the tokens gone!

Productions:

$E \rightarrow A E'$

$E' \rightarrow + A E'$

$E' \rightarrow - A E'$

$E' \rightarrow \epsilon$

$A \rightarrow (E)$

$A \rightarrow n$

We found an E!

70

## Observations:

- ◆ By modifying the grammar, we have ensured that one of the following holds at each step:
  - There is only one production in the grammar for the nonterminal that we are looking for;
  - Or the token at the front of the input stream tells us which production applies;
  - Or the token at the front of the input stream matches the one we are looking for;
  - Or we've reached the end!

71

## Predictive Parsers:

- ◆ In general (but not always), we need to change the grammar to make this work.
- ◆ For some languages, there is no CFG that will make this work.
- ◆ But when it does work, and especially for small grammars, we can even write the parser by hand ...

72

## Recursive Descent Parsing:

```

Expr parseExpr() {
  A {
    Expr e = parseAtom();

    for (;;) {
      E' {
        if (getToken()=='+') {
          e = new AddExpr(e, parseAtom());
        } else if (getToken()=='-') {
          e = new SubExpr(e, parseAtom());
        } else {
          break;
        }
      }
    }
    return e;
  }
}

```

$E \rightarrow A E'$  Expressions

$E' \rightarrow + A E'$  Terms

$E' \rightarrow - A E'$

$E' \rightarrow \epsilon$

73

## Recursive Descent Parsing:

```

Expr parseAtom() {
  (E) {
    if (getToken()=='(') {
      nextToken();
      Expr e = parseExpr();
      if (getToken()!=')') {
        ... report error ...
      } else {
        nextToken();
      }
      return e;
    } else if (getToken()==INTEGER) {
      int val = getTokenAttribute();
      nextToken();
      return new IntExpr(val);
    } else {
      ... report error ...
    }
  }
  n {
  }
}

```

$A \rightarrow (E)$  Atoms

$A \rightarrow n$

74

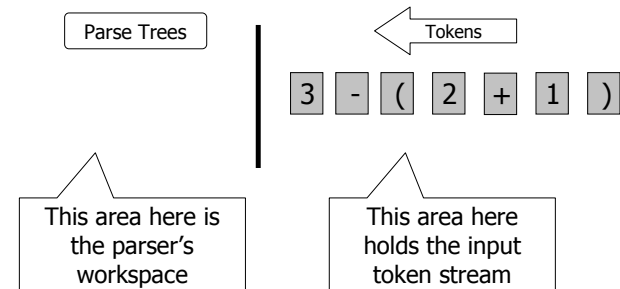
## Bottom-up Parsing:

### ◆ General Strategy:

- Read input from left to right.
- Maintain a collection of parse tree fragments.
- Every time we have a collection of fragments that belong together, combine them to make a bigger fragment.

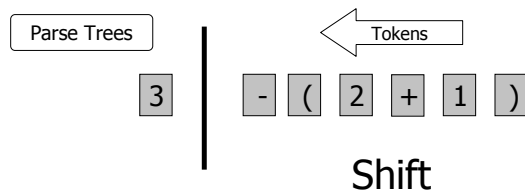
75

## Bottom-up Parsing:



76

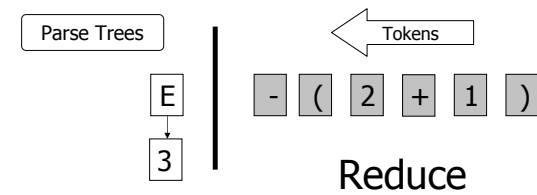
## Bottom-up Parsing:



A shift step occurs each time we move a terminal symbol across the red line from the input stream to the parser's workspace

77

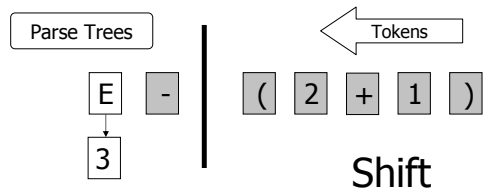
## Bottom-up Parsing:



A reduce step occurs each time we match the right hand side of a production up against the red line. We replace the corresponding entries in the parser's workspace with a section of parse tree.

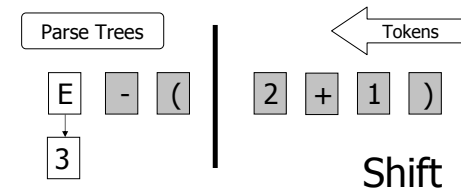
78

## Bottom-up Parsing:



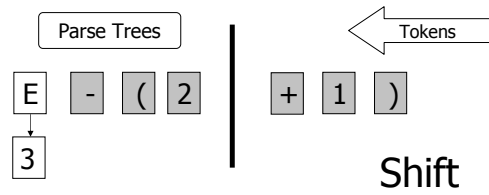
79

## Bottom-up Parsing:



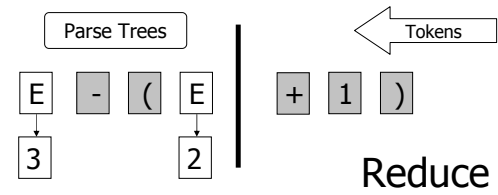
80

### Bottom-up Parsing:



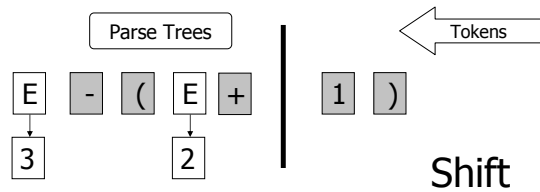
81

### Bottom-up Parsing:



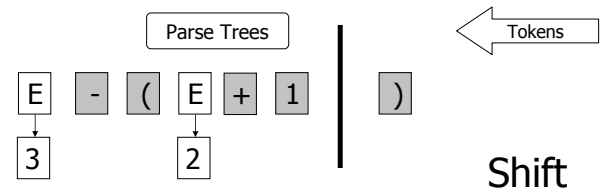
82

### Bottom-up Parsing:



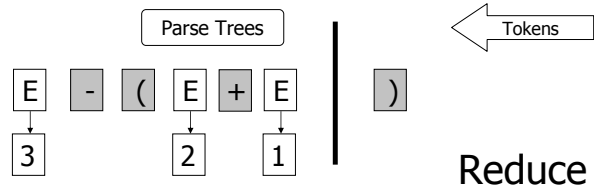
83

### Bottom-up Parsing:



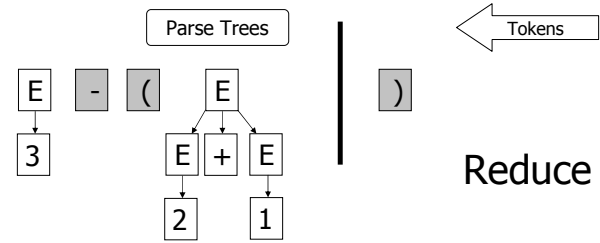
84

### Bottom-up Parsing:



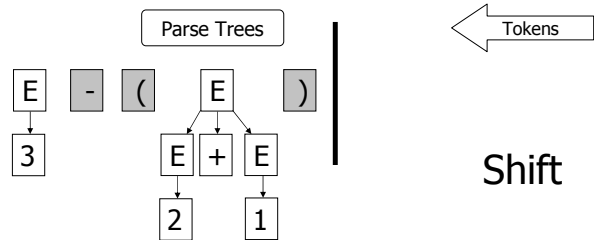
85

### Bottom-up Parsing:



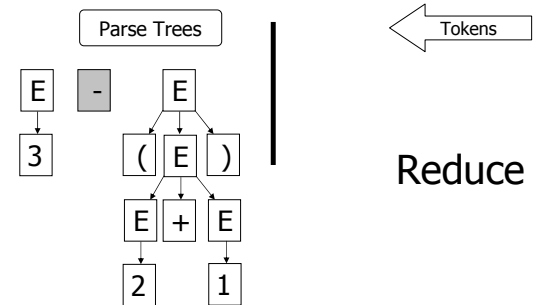
86

### Bottom-up Parsing:



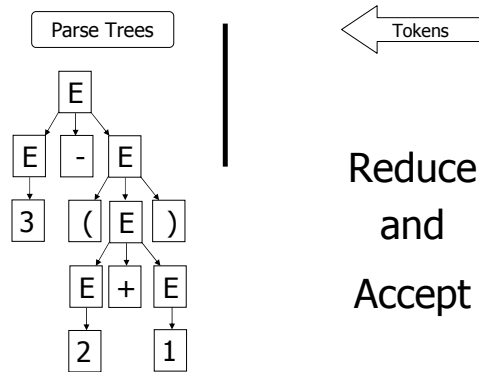
87

### Bottom-up Parsing:



88

## Bottom-up Parsing:



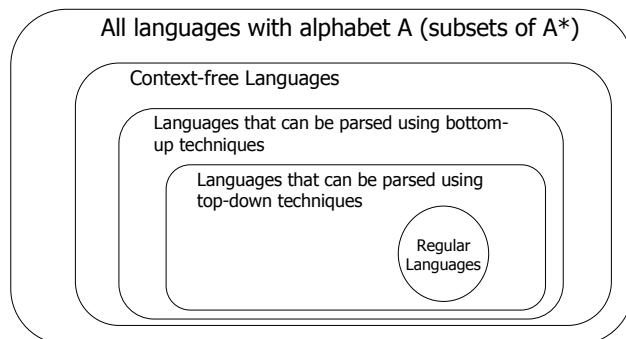
89

## Observations:

- ◆ We read the token stream left to right.
- ◆ The parser's workspace behaves like a stack.
- ◆ The trick is in knowing when to shift and when to reduce ...

90

## The Language Design Space:



91

## Summary:

- ◆ Context-free grammars are more powerful than regular expressions, and good for defining programming language syntax.
- ◆ Parsing is the process of constructing a parse tree from an input sequence of tokens.
- ◆ Ambiguous grammars leave the job of parsing under-specified. Extra information must be given to describe how ambiguities are resolved.
- ◆ Top-down and bottom-up strategies can be used to build parsers for CFGs.

92