

Introduction to Parsing

Lecture 5

Prof. Necla CS 164

1

Outline

- Limitations of regular languages
- Parser overview
- Context-free grammars (CFG's)
- Derivations

Prof. Necla CS 164

2

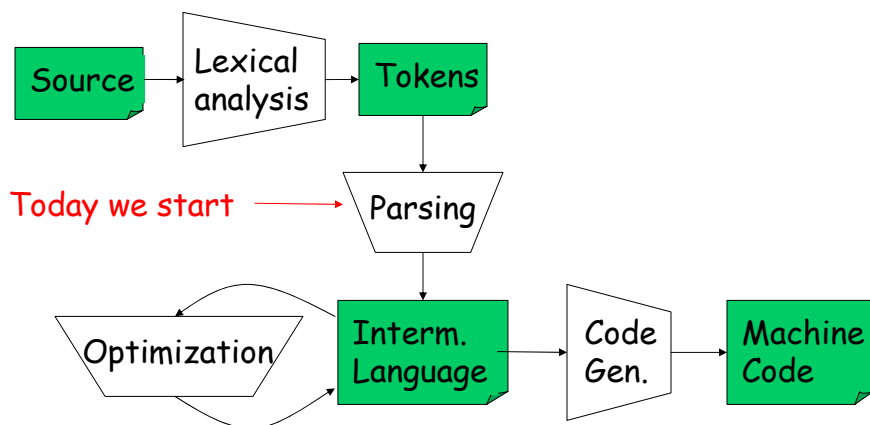
Languages and Automata

- Formal languages are very important in CS
 - Especially in programming languages
- Regular languages
 - The weakest formal languages widely used
 - Many applications
- We will also study context-free languages

Limitations of Regular Languages

- Intuition: A finite automaton that runs long enough must repeat states
- Finite automaton can't remember # of times it has visited a particular state
- Finite automaton has finite memory
 - Only enough to store in which state it is
 - Cannot count, except up to a finite limit
- E.g., language of balanced parentheses is not regular: $\{ ()^i \mid i \geq 0 \}$

Recall: The Structure of a Compiler



Prof. Neula CS 164

5

The Functionality of the Parser

- **Input:** sequence of tokens from lexer
- **Output:** parse tree of the program

Prof. Neula CS 164

6

Example

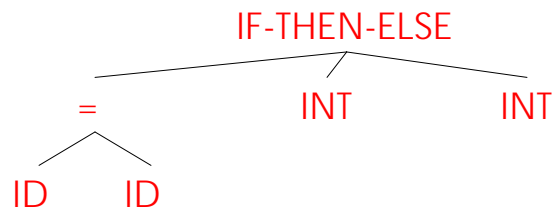
- Cool

if x = y then 1 else 2 fi

- Parser input

IF ID = ID THEN INT ELSE INT FI

- Parser output



Prof. Neula CS 164

7

Comparison with Lexical Analysis

<i>Phase</i>	<i>Input</i>	<i>Output</i>
Lexer	Sequence of characters	Sequence of tokens
Parser	Sequence of tokens	Parse tree

Prof. Neula CS 164

8

The Role of the Parser

- Not all sequences of tokens are programs . . .
- . . . Parser must distinguish between valid and invalid sequences of tokens

- We need
 - A language for describing valid sequences of tokens
 - A method for distinguishing valid from invalid sequences of tokens

Programming Language Structure

- Programming languages have recursive structure
- Consider the language of arithmetic expressions with integers, +, *, and ()
- An expression is either:
 - an integer
 - an expression followed by "+" followed by expression
 - an expression followed by "*" followed by expression
 - a '(' followed by an expression followed by ')'
- `int` , `int + int` , `(int + int) * int` are expressions

Notation for Programming Languages

- An alternative notation:

$E \rightarrow \text{int}$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

- We can view these rules as rewrite rules
 - We start with E and replace occurrences of E with some right-hand side
- $E \rightarrow E * E \rightarrow (E) * E \rightarrow (E + E) * E$
 $\rightarrow (\text{int} + \text{int}) * \text{int}$

Prof. Necula CS 164

11

Observation

- All arithmetic expressions can be obtained by a sequence of replacements
- Any sequence of replacements forms a valid arithmetic expression
- This means that we cannot obtain
 $(\text{int}))$
by any sequence of replacements. Why?
- This notation is a context free grammar

Prof. Necula CS 164

12

Context Free Grammars

- A CFG consists of
 - A set of *non-terminals* N
 - By convention, written with capital letter in these notes
 - A set of *terminals* T
 - By convention, either lower case names or punctuation
 - A *start symbol* S (a non-terminal)
 - A set of *productions*
- Assuming $E \in N$
 - $E \rightarrow \varepsilon$, or
 - $E \rightarrow Y_1 Y_2 \dots Y_n$ where $Y_i \in N \cup T$

Prof. Necula CS 164

13

Examples of CFGs

Simple arithmetic expressions:

$$\begin{aligned} E &\rightarrow \text{int} \\ E &\rightarrow E + E \\ E &\rightarrow E * E \\ E &\rightarrow (E) \end{aligned}$$

- One non-terminal: E
- Several terminals: $\text{int}, +, *, (,)$
 - Called terminals because they are never replaced
- By convention the non-terminal for the first production is the start one

Prof. Necula CS 164

14

The Language of a CFG

Read productions as replacement rules:

$$X \rightarrow Y_1 \dots Y_n$$

Means X can be replaced by $Y_1 \dots Y_n$

$$X \rightarrow \varepsilon$$

Means X can be erased (replaced with empty string)

Key Idea

1. Begin with a string consisting of the start symbol "S"
2. Replace any non-terminal X in the string by a right-hand side of some production

$$X \rightarrow Y_1 \dots Y_n$$

3. Repeat (2) until there are only terminals in the string

The Language of a CFG (Cont.)

More formally, write

$$X_1 \dots X_{i-1} X_i X_{i+1} \dots X_n \rightarrow X_1 \dots X_{i-1} Y_1 \dots Y_m X_{i+1} \dots X_n$$

if there is a production

$$X_i \rightarrow Y_1 \dots Y_m$$

The Language of a CFG (Cont.)

Write

$$X_1 \dots X_n \rightarrow^* Y_1 \dots Y_m$$

if

$$X_1 \dots X_n \rightarrow \dots \rightarrow \dots \rightarrow Y_1 \dots Y_m$$

in 0 or more steps

The Language of a CFG

Let G be a context-free grammar with start symbol S . Then the language of G is:

$$\{ a_1 \dots a_n \mid S \rightarrow^* a_1 \dots a_n \text{ and every } a_i \text{ is a terminal} \}$$

Examples:

- $S \rightarrow 0$ also written as $S \rightarrow 0 \mid 1$
 $S \rightarrow 1$
Generates the language $\{ "0", "1" \}$
- What about $S \rightarrow 1 A$
 $A \rightarrow 0 \mid 1$
- What about $S \rightarrow 1 A$
 $A \rightarrow 0 \mid 1 A$
- What about $S \rightarrow \varepsilon \mid (S)$

Arithmetic Example

Simple arithmetic expressions:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

Some elements of the language:

id		id + id
(id)		id * id
(id) * id		id * (id)

Prof. Necula CS 164

21

Cool Example

A fragment of COOL:

$$\begin{aligned} \text{EXPR} &\rightarrow \text{if EXPR then EXPR else EXPR fi} \\ &| \text{while EXPR loop EXPR pool} \\ &| \text{id} \end{aligned}$$

Prof. Necula CS 164

22

Cool Example (Cont.)

Some elements of the language

id

if id then id else id fi

while id loop id pool

if while id loop id pool then id else id

if if id then id else id fi then id else id fi

Notes

The idea of a CFG is a big step. But:

- Membership in a language is "yes" or "no"
 - we also need parse tree of the input
- Must handle errors gracefully
- Need an implementation of CFG's (e.g., bison)

More Notes

- Form of the grammar is important
 - Many grammars generate the same language
 - Tools are sensitive to the grammar

- Note: Tools for regular languages (e.g., flex) are also sensitive to the form of the regular expression, but this is rarely a problem in practice

Derivations and Parse Trees

A *derivation* is a sequence of productions

$$S \rightarrow \dots \rightarrow \dots$$

A derivation can be drawn as a tree

- Start symbol is the tree's root
- For a production $X \rightarrow Y_1 \dots Y_n$ add children Y_1, \dots, Y_n to node X

Derivation Example

- Grammar

$$E \rightarrow E+E \mid E * E \mid (E) \mid id$$

- String

$$id * id + id$$

Derivation Example (Cont.)

$$\begin{aligned} & E \\ \rightarrow & E+E \\ \rightarrow & E * E+E \\ \rightarrow & id * E + E \\ \rightarrow & id * id + E \\ \rightarrow & id * id + id \end{aligned}$$

Derivation in Detail (1)

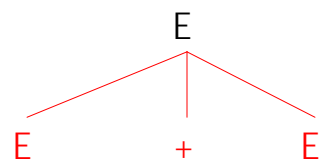
E

E

Prof. Necula CS 164

29

Derivation in Detail (2)

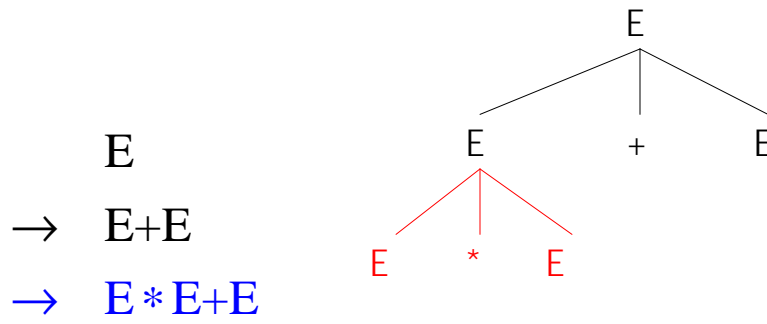


E
→ E+E

Prof. Necula CS 164

30

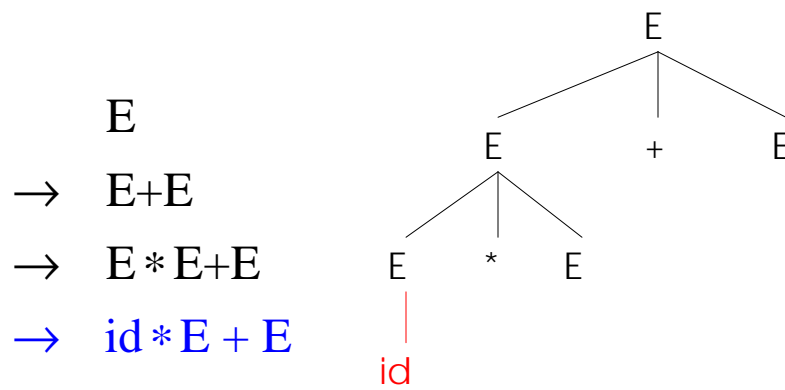
Derivation in Detail (3)



Prof. Necula CS 164

31

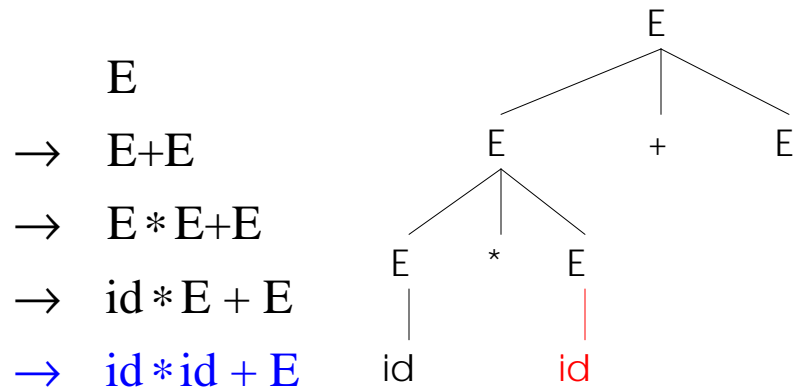
Derivation in Detail (4)



Prof. Necula CS 164

32

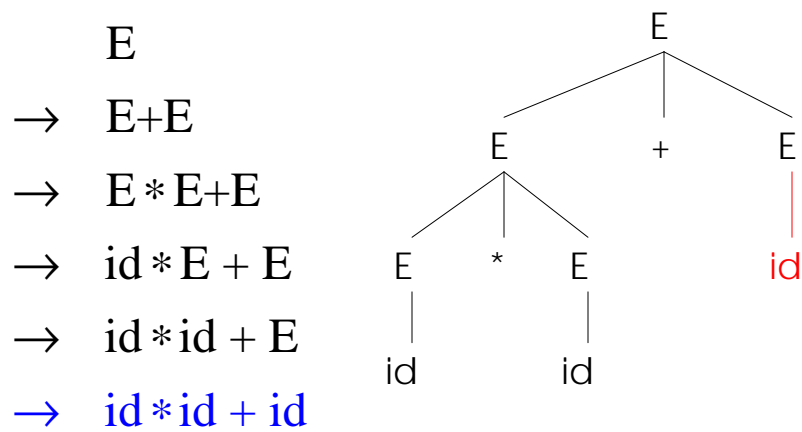
Derivation in Detail (5)



Prof. Necula CS 164

33

Derivation in Detail (6)



Prof. Necula CS 164

34

Notes on Derivations

- A parse tree has
 - Terminals at the leaves
 - Non-terminals at the interior nodes
- A left-right traversal of the leaves is the original input
- The parse tree shows the association of operations, the input string does not !

Prof. Necula CS 164

35

Left-most and Right-most Derivations

- The example is a *left-most* derivation
 - At each step, replace the left-most non-terminal
 - There is an equivalent notion of a *right-most* derivation
- E
- $\rightarrow E+E$
- $\rightarrow E+id$
- $\rightarrow E * E + id$
- $\rightarrow E * id + id$
- $\rightarrow id * id + id$

Prof. Necula CS 164

36

Right-most Derivation in Detail (1)

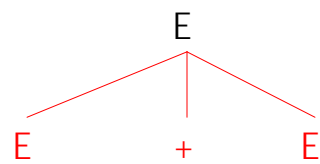
E

E

Prof. Necula CS 164

37

Right-most Derivation in Detail (2)

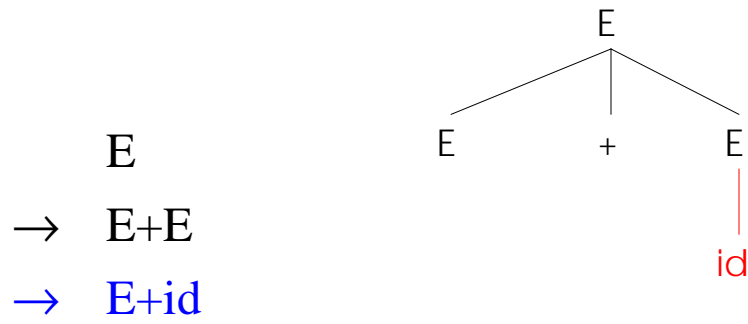


E
→ E+E

Prof. Necula CS 164

38

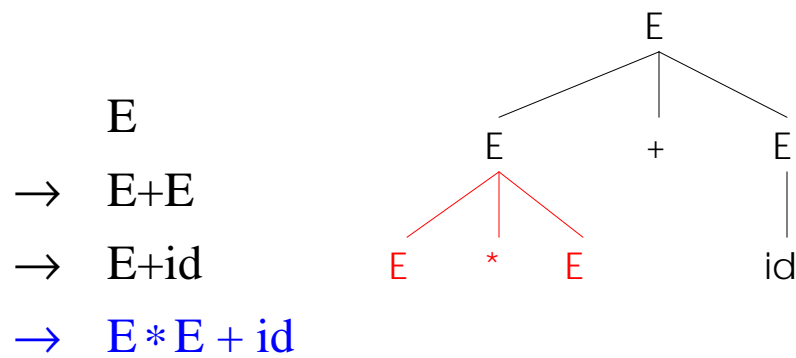
Right-most Derivation in Detail (3)



Prof. Necula CS 164

39

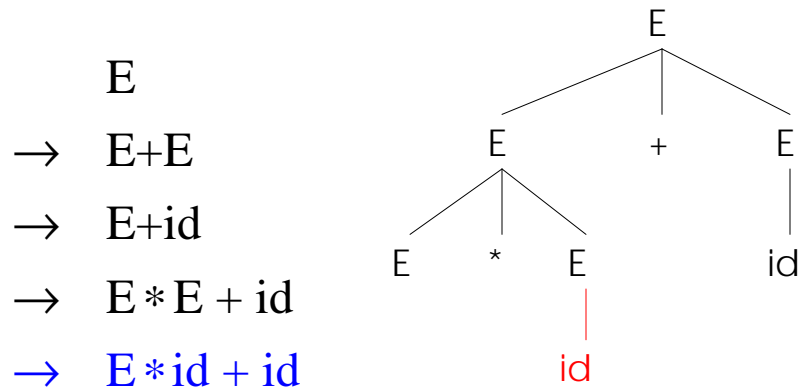
Right-most Derivation in Detail (4)



Prof. Necula CS 164

40

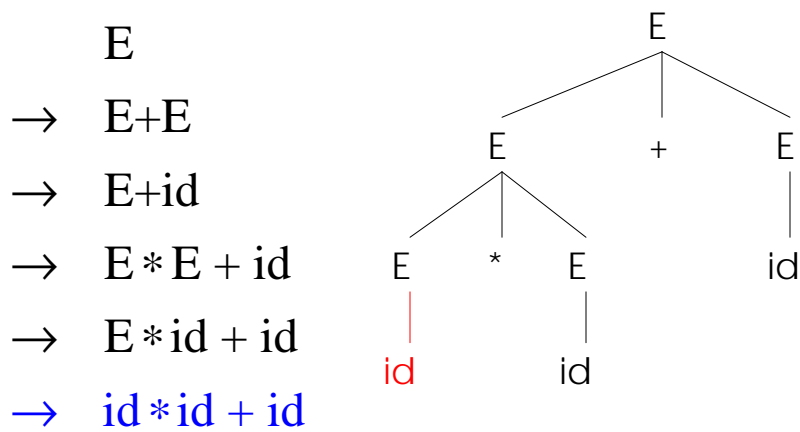
Right-most Derivation in Detail (5)



Prof. Necula CS 164

41

Right-most Derivation in Detail (6)



Prof. Necula CS 164

42

Derivations and Parse Trees

- Note that for each parse tree there is a left-most and a right-most derivation
- The difference is the order in which branches are added

Summary of Derivations

- We are not just interested in whether $s \in L(G)$
 - We need a parse tree for s
- A derivation defines a parse tree
 - But one parse tree may have many derivations
- Left-most and right-most derivations are important in parser implementation