Context-Free Grammars

Lecture 7

http://webwitch.dreamhost.com/grammar.girl/

Outline

- Scanner vs. parser
  - Why regular expressions are not enough
- Grammars (context-free grammars)
  - grammar rules
  - derivations
  - parse trees
  - ambiguous grammars
  - useful examples
- Reading:
  - Sections 4.1 and 4.2

The Functionality of the Parser

- **Input:** sequence of tokens from lexer
- **Output:** parse tree of the program
  - parse tree is generated if the input is a legal program
  - if input is an illegal program, syntax errors are issued
- Note:
  - Instead of parse tree, some parsers produce directly:
    - abstract syntax tree (AST) + symbol table (as in P3), or
    - intermediate code, or
    - object code
  - In the following lectures, we'll assume that parse tree is generated.

Comparison with Lexical Analysis

Phase	*Input*	*Output*
Lexer	String of characters	String of tokens
Parser	String of tokens	Parse tree

Example

- **The program:**

  x * y + z

- **Input to parser:**

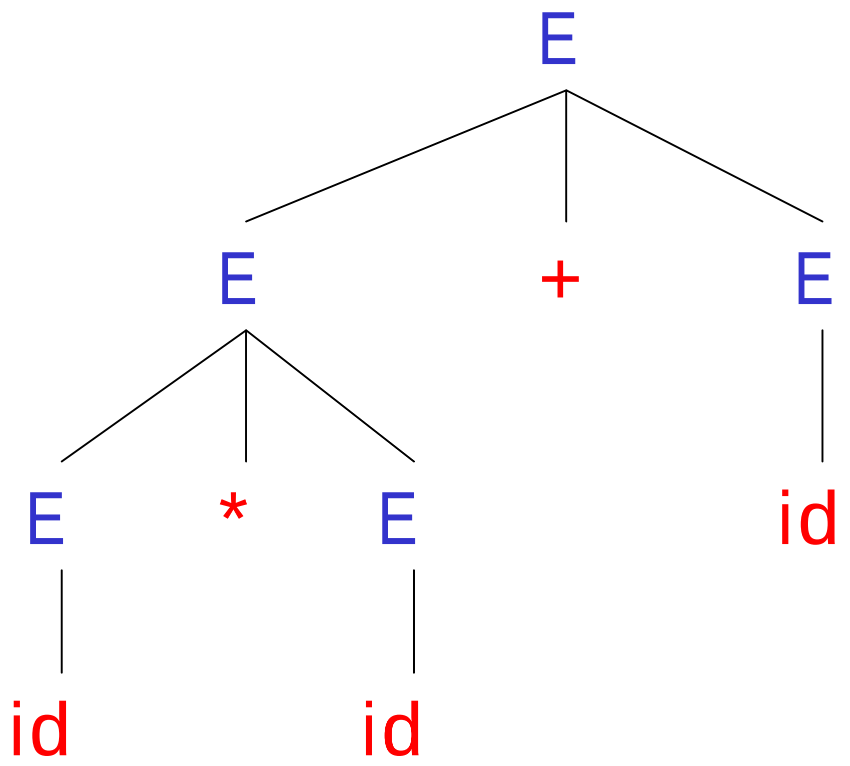
  ID TIMES ID PLUS ID

  we'll write tokens as follows:

  id * id + id

- **Output of parser:**

  the parse tree →



Why are regular expressions not enough?

TEST YOURSELF #1

- Write an automaton that accepts strings
  - “a”, “(a)”, “((a))”, and “(((a)))”
  - “a”, “(a)”, “((a))”, “(((a)))”, … “$(^k a)^k$”

Why are regular expressions not enough?

TEST YOURSELF #2

- What programs are generated by?

  digit+ ((“+” | “-” | “*” | “/”) digit+)*

- What important properties this regular expression fails to express?

What must parser do?

1. Recognizer: not all strings of tokens are programs
   - must distinguish between valid and invalid strings of tokens
2. Translator: must expose program structure
   - e.g., associativity and precedence
   - hence must return the parse tree

We need:

- A language for describing valid strings of tokens
  - **context-free grammars**
  - (analogous to regular expressions in the scanner)
- A method for distinguishing valid from invalid strings of tokens (and for building the parse tree)
  - **the parser**
  - (analogous to the state machine in the scanner)

We need context-free grammars (CFGs)

- Example: Simple Arithmetic Expressions
  - In English:
    - An integer is an arithmetic expression.
    - If exp_1 and exp_2 are arithmetic expressions, then so are the following:

      exp_1 - exp_2

      exp_1 / exp_2

      (exp_1)

- the corresponding CFG:

we'll write tokens as follows:

exp $\rightarrow$ INTLITERAL	E $\rightarrow$ intlit
exp $\rightarrow$ exp MINUS exp	E $\rightarrow$ E - E
exp $\rightarrow$ exp DIVIDE exp	E $\rightarrow$ E / E
exp $\rightarrow$ LPAREN exp RPAREN	E $\rightarrow$ (E)

Reading the CFG

- The grammar has five terminal symbols:
  - **intlit, -, /, (,)**
  - terminals of a grammar = tokens returned by the scanner.
- The grammar has one non-terminal symbol:
  - **E**
  - non-terminals describe valid sequences of tokens
- The grammar has four productions or rules,
  - each of the form: $E \rightarrow \alpha$
    - left-hand side = a single non-terminal.
    - right-hand side = either
      - a sequence of one or more terminals and/or non-terminals, or
      - ε (an empty production); again, the book uses symbol λ

Example, revisited

- Note:
  - a more compact way to write previous grammar:

$E \rightarrow$ intlit | E - E | E / E | (E)

or

```
E → intlit
  |  E - E
  |  E / E
  |  ( E )
```

A formal definition of CFGs

- A CFG consists of
  - A set of *terminals T*
  - A set of *non-terminals N*
  - A *start symbol S* (a non-terminal)
  - A set of *productions:*

$$X \rightarrow Y_1 Y_2 \mathrm{L} \; Y_n$$

$$\text{where } X \in N \text{ and } Y_i \in T \cup N \cup \{e\}$$

Notational Conventions

- In these lecture notes
  - Non-terminals are written upper-case
  - Terminals are written lower-case
  - The start symbol is the left-hand side of the first production

The Language of a CFG

The language defined by a CFG is the set of strings that can be derived from the start symbol of the grammar.

Derivation: Read productions as rules:

$$X \to Y_1 \ldots Y_n$$

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

Derivation: key idea

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

$$X \to Y_1 \mathrm{L}\ Y_n$$

3. Repeat (2) until there are no non-terminals in the string

Derivation: an example

CFG:

$E \rightarrow id$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

Is string id * id + id in the language defined by the grammar?

derivation:

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

Terminals

- Terminals are called because there are no rules for replacing them
- Once generated, terminals are permanent
- Therefore,
  terminals are the tokens of the language

The Language of a CFG (Cont.)

More formally, write

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

if there is a production

$$X_i \rightarrow Y_1 \ldots Y_m$$

The Language of a CFG (Cont.)

Write

$$X_1 \cdots X_n \overset{*}{\rightarrow} Y_1 \cdots Y_m$$

if

$$X_1 \cdots X_n \rightarrow \cdots \rightarrow \cdots \rightarrow Y_1 \cdots Y_m$$

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:

$$\left\{ a_1 \ldots a_n \mid S \xrightarrow{*} a_1 \ldots a_n \text{ and every } a_i \text{ is a terminal} \right\}$$

Examples

Strings of balanced parentheses $\{(^i)^i \mid i \geq 0\}$

The grammar:

$$\begin{array}{lll} S & \rightarrow & (S) \\ S & \rightarrow & e \end{array}$$

same as

$$\begin{array}{lll} S & \rightarrow & (S) \\ & \mid & e \end{array}$$

Arithmetic Example

Simple arithmetic expressions:

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

Some elements of the language:

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

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!
  - furthermore, we must handle errors gracefully
- Need an “implementation” of CFG’s,
  - i.e. the parser
  - we’ll create the parser using a parser generator
    - available generators: CUP, bison, yacc

More Notes

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

- Example:

$$
\begin{aligned}
E &\rightarrow E + E \\
  &\mid E - E \\
  &\mid \text{intlit}
\end{aligned}
$$

is not suitable for an LL(1) parser (a common kind of parser).
Stay tuned, you will soon understand why.

Derivations and Parse Trees

A *derivation* is a sequence of productions

$$S \to \mathrm{L} \to \mathrm{L} \to \mathrm{L}$$

A derivation can be drawn as a tree

- Start symbol is the tree's root
- For a production $X \to Y_1 \mathrm{L}\ Y_n$ add children $Y_1 \mathrm{L}\ Y_n$ to node X

Derivation Example

- Grammar

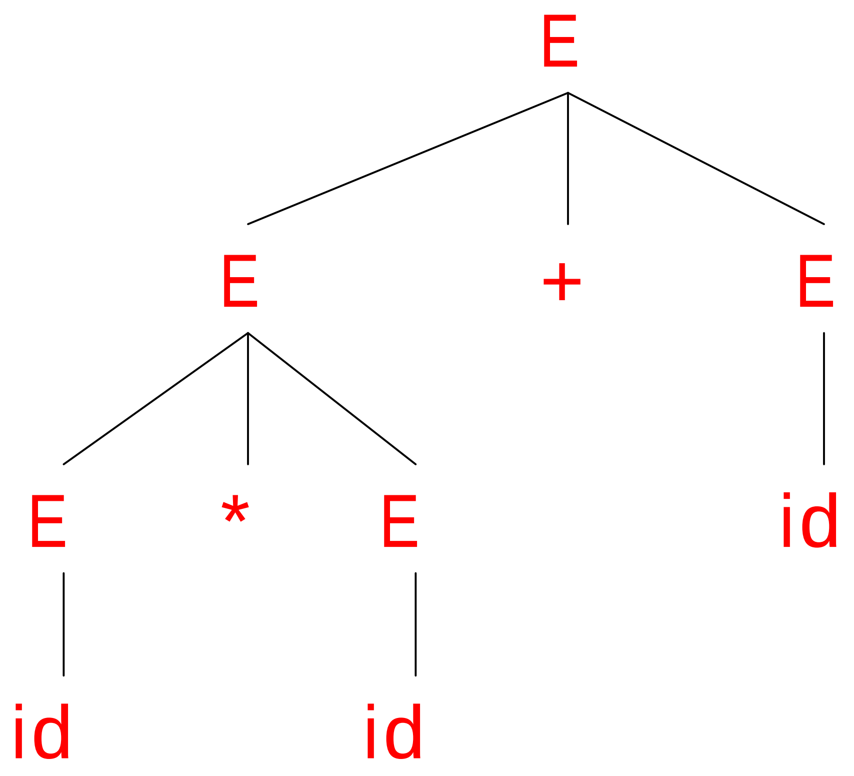
$$E \rightarrow E+E \mid E * E \mid (E) \mid id$$

- String

$$id * id + id$$

Derivation Example (Cont.)

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



Derivation in Detail (1)

id * id + id

E

E

Notes on Derivations

- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes

- An in-order traversal of the leaves is the original input

- The parse tree shows the association of operations, the input string does not

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

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

Right-most Derivation in Detail (1)

id * id + id

E

E

Derivations and Parse Trees

- Note that right-most and left-most derivations have the same parse tree
- 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*, (because we need to build the AST)

- A derivation defines a parse tree
  - But one parse tree may have many derivations

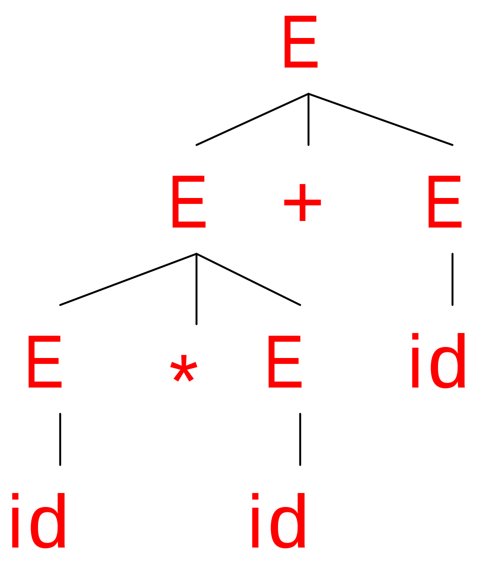
- Left-most and right-most derivations are important in parser implementation

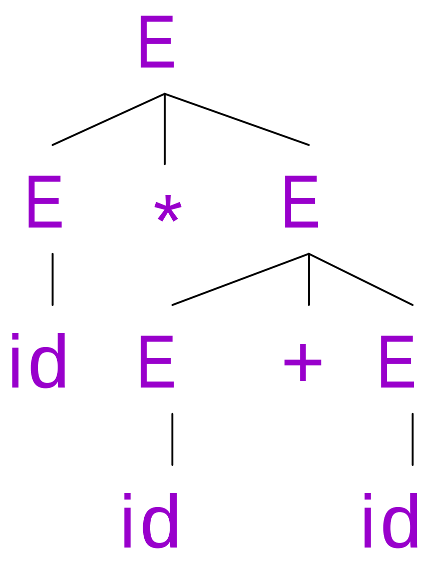
Ambiguity

- Grammar $E \rightarrow E+E \mid E * E \mid (E) \mid id$
- String $id * id + id$

Ambiguity (Cont.)

This string has two parse trees





TEST YOURSELF #3

Question 1:

- for each of the two parse trees, find the corresponding left-most derivation

Question 2:

- for each of the two parse trees, find the corresponding right-most derivation

Ambiguity (Cont.)

- A grammar is *ambiguous* if for some string (the following three conditions are equivalent)
  - it has more than one parse tree
  - if there is more than one right-most derivation
  - if there is more than one left-most derivation
- Ambiguity is **BAD**
  - Leaves meaning of some programs ill-defined

Dealing with Ambiguity

- There are several ways to handle ambiguity

- Most direct method is to rewrite grammar unambiguously

$$E \rightarrow E' + E \mid E'$$

$$E' \rightarrow \mathrm{id} * E' \mid \mathrm{id} \mid (E)$$

- Enforces precedence of * over +