Introduction to Parsing

Lecture 5

Outline

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

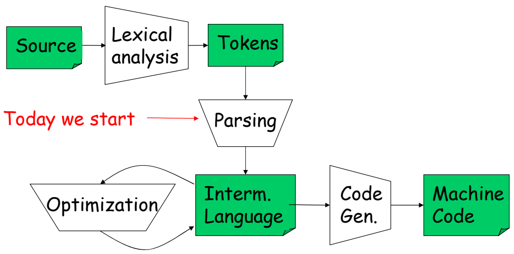
Languages and Automata

- 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: $\{ (^i)^i \mid i \geq 0\}$

Recall: The Structure of a Compiler



The Functionality of the Parser

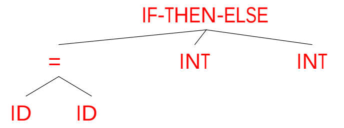
- **Input:** sequence of tokens from lexer
- **Output:** parse tree of the program

Example

- Cool

  if x = y then 1 else 2 fi
- Parser input

  IF ID = ID THEN INT ELSE INT FI
- Parser output


IF-THEN-ELSE

= INT INT

ID ID

Comparison with Lexical Analysis

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

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 → int
  E → E + E
  E → E * E
  E → (E)
- We can view these rules as rewrite rules
  - We start with E and replace occurrences of E with some right-hand side
- E → E * E → (E) * E → (E + E) * E
  → (int + int) * int

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

  (int))

  by any sequence of replacements. Why?

- This notation is a context free grammar

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 $E \in N$

$E \to \varepsilon$, or

$E \to Y_1 Y_2 \ldots Y_n$ where $Y_i \in N \cup T$

Examples of CFGs

Simple arithmetic expressions:

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

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

---

The Language of a CFG

Read productions as replacement rules:

$X \to Y_1 \ldots Y_n$

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

$X \to \varepsilon$

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

$$X \to Y_1 \ldots Y_n$$

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

The Language of a CFG (Cont.)

More formally, write

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

if there is a production

$$X_i \to Y_1 \ldots Y_m$$

The Language of a CFG (Cont.)

Write

$$X_1 \ldots X_n \to^* Y_1 \ldots Y_m$$

if

$$X_1 \ldots X_n \to \ldots \to \ldots \to Y_1 \ldots Y_m$$

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:

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

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)

Cool Example

A fragment of COOL:

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

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

$S \rightarrow \ldots \rightarrow \ldots$

A derivation can be drawn as a tree

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

Derivation Example

- Grammar

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

- String

$$\mathrm{id} * \mathrm{id} + \mathrm{id}$$

Derivation Example (Cont.)

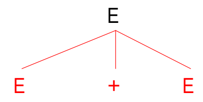
$$\begin{aligned} & E \\ \rightarrow\ & E+E \\ \rightarrow\ & E*E+E \\ \rightarrow\ & \mathrm{id}*E+E \\ \rightarrow\ & \mathrm{id}*\mathrm{id}+E \\ \rightarrow\ & \mathrm{id}*\mathrm{id}+\mathrm{id} \end{aligned}$$

Derivation in Detail (1)

E

$$E$$

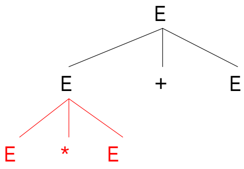
Derivation in Detail (2)



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

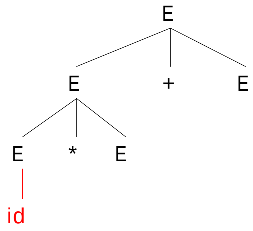
Derivation in Detail (3)

E
→ E+E
→ E ∗ E+E



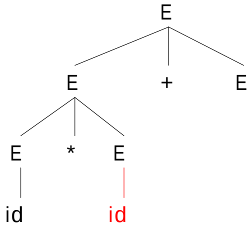
Derivation in Detail (4)

E
→ E+E
→ E ∗ E+E
→ id ∗ E + E



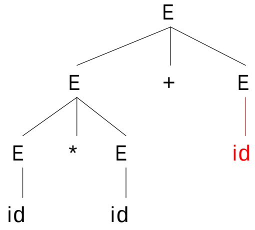
Derivation in Detail (5)

E
→ E+E
→ E ∗ E+E
→ id ∗ E + E
→ id ∗ id + E



Derivation in Detail (6)

E
→ E+E
→ E ∗ E+E
→ id ∗ E + E
→ id ∗ id + E
→ id ∗ id + id



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 !

Left-most and Right-most Derivations

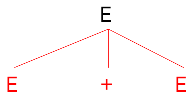
- The example is a *left-most* derivation
  - At each step, replace the left-most non-terminal
- There is an equivalent notion of a *right-most* derivation

E
→ E+E
→ E+id
→ E ∗ E + id
→ E ∗ id + id
→ id ∗ id + id

Right-most Derivation in Detail (1)

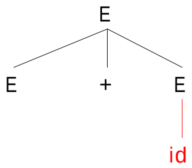
E

Right-most Derivation in Detail (2)



E
→ E+E

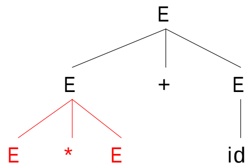
Right-most Derivation in Detail (3)



E
→ E+E
→ E+id

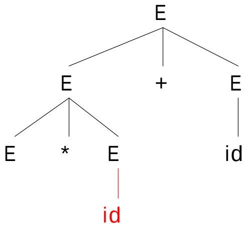
Right-most Derivation in Detail (4)

E
→ E+E
→ E+id
→ E ∗ E + id



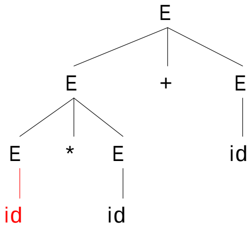
Right-most Derivation in Detail (5)

E
→ E+E
→ E+id
→ E ∗ E + id
→ E ∗ id + id



Right-most Derivation in Detail (6)

E
→ E+E
→ E+id
→ E ∗ E + id
→ E ∗ id + id
→ id ∗ id + id



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