Outline

- Context-Free Grammars (CFGs)
- Derivations
- Parse trees and abstract syntax
- Ambiguous grammars

Where we are

Source code (character stream)

if (b == 0) a = b;

Token stream

if (b == 0) a = b;

Abstract syntax tree (AST)

if_stmt

bin_op
t

variable

constant

0

while_stmt

block

Expression

expr_stmt

bin_op
t

variable

constant

1

Semantic Analysis

Lexical analysis

Syntactic Analysis (specification)

What is Syntactic Analysis?

Source code (token stream)

{ if (b == 0) a = b;
  while (a != 1) {
    stdio.print(a);
    a = a - 1;
  }
}

Abstract Syntax Tree
**Parsing**

- Parsing: recognizing whether a program (or sentence) is grammatically well-formed & identifying the function of each component.

  - “I gave him the book”
  - sentence
  - subject: I
  - verb: gave
  - indirect object: him
  - noun phrase
  - article: the
  - noun: book

---

**Overview of Syntactic Analysis**

- Input: stream of tokens
- Output: abstract syntax tree
- Implementation:
  - Parse token stream to traverse concrete syntax (**parse tree**)
  - During traversal, build abstract syntax tree
  - Abstract syntax tree removes extra syntax
    \[ a + b \approx (a) + (b) \approx ((a)+(b)) \]

---

**What Parsing doesn’t do**

- Doesn’t check many things: type agreement, variables declared, variables initialized, etc.
  
  - `int x = true;`
  - `int y;`
  - `z = f(y);`
- Deferred until semantic analysis

---

**Specifying Language Syntax**

- First problem: how to describe language syntax precisely and conveniently
- Last time: can describe tokens using regular expressions
- Regular expressions easy to implement, efficient (by converting to DFA)
- Why not use regular expressions (on tokens) to specify programming language syntax?
Limits of REs

• Programming languages are not regular -- cannot be described by regular exprs
• Consider: language of all strings that contain balanced parentheses (easier than PLs)
  () (()) ()()() ()()((()()))

• Problem: need to keep track of number of parentheses seen so far: unbounded counting

Need more power!

• RE = DFA
• DFA has only finite number of states; cannot perform unbounded counting

Context-Free Grammars

• A specification of the balanced-parenthesis language:
  \[ S \rightarrow ( S ) S \]
  \[ S \rightarrow \varepsilon \]

• The definition is recursive
• A context-free grammar
  – More expressive than regular expressions
    – \[ S = (S) \varepsilon = ((S) S) \varepsilon = ((\varepsilon) \varepsilon) \varepsilon = (\varepsilon) \]

If a grammar accepts a string, there is a derivation of that string using the productions of the grammar

Definition of CFG

• Terminals
  – Token or \( \varepsilon \)
  \[ S \rightarrow ( S ) S \]
• Non-terminals
  – Syntactic variables
  \[ S \rightarrow \varepsilon \]
• Start symbol
  – A special nonterminal is designated \( S \)
• Productions
  – Specify how non-terminals may be expanded to form strings
    – LHS: single non-terminal, RHS: string of terminals or non-terminals
• Vertical bar is shorthand for multiple prod’ns
**RE is subset of CFG**

Regular Expression defn of real numbers:
- *digit* → [0-9]
- *posint* → digit+
- *int* → -? posint
- *real* → int . (ε | posint)

- RE symbolic names are only shorthand: no recursion, so all symbols can be fully expanded:
  - real → -? [0-9]+ . (ε | ([0-9]+))

**Sum grammar**

\[
S \rightarrow E + S \mid E \\
E \rightarrow \text{number} \mid (S) \\
\]

- 4 productions
- 2 non-terminals (S, E)
- 4 terminals: (, ), +, number
- start symbol S

**Derivation Example**

- Start from start symbol (S)
- Productions are used to derive a sequence of tokens from the start symbol
- For arbitrary strings α, β and γ and a production A → β
  - A single step of derivation is αAγ ⇒ αβγ
  - i.e., substitute β for an occurrence of A
  - (S + E) + E → (E + S + E)+E

- (A = S, β = E + S)
### Derivation ⇒ Parse Tree

**Parse Tree**
- Also called “concrete syntax”

**Parse Tree**
- Also called “concrete syntax”
- parse tree/concrete syntax
- abstract syntax tree

(Discards/abstracts unneeded information)

### Derivation order
- Can choose to apply productions in any order; select any non-terminal A
  \[ \alpha A \gamma \Rightarrow \alpha \beta \gamma \]
- Two standard orders: left- and right-most -- useful for different kinds of automatic parsing
- **Leftmost derivation**: In the string, find the left-most non-terminal and apply a production to it \( E + S \rightarrow 1 + S \)
- **Rightmost derivation**: find right-most non-terminal...etc. \( E + S \rightarrow E + E + S \)

### Example

\[
S \rightarrow E + S | E \\
E \rightarrow \text{number} | (S) \\
\]

- Left-most derivation
  \[
  S \rightarrow E + S \rightarrow (E + S) + S \rightarrow (1 + S) + S \rightarrow (1 + E + S) + S \rightarrow (1 + 2 + (3 + 4)) + S \\
  + S \rightarrow (1 + 2 + (3 + 4)) + S \rightarrow \ldots \rightarrow (1 + 2 + (3 + 4)) + 5
  \]
- Right-most derivation
  \[
  S \rightarrow E + S \rightarrow E + E \rightarrow E + 5 \rightarrow (S) + 5 \rightarrow (E + S) + 5 \rightarrow (E + E + S) + 5 \rightarrow (E + E + E + S) + 5 \rightarrow (E + E + E + E + S) + 5 \rightarrow \ldots \rightarrow (E + E + E + E + E + E + E + E + E + 5) \\
  \]
- Same parse tree: same productions chosen, diff. order
Ambiguous Grammars

- In example grammar, left-most and right-most derivations produced identical parse trees
- + operator associates to right in parse tree regardless of derivation order

\[(1+2+(3+4))+5\]

An Ambiguous Grammar

- + associates to right because of right-recursive production  \( S \rightarrow E + S \)
- Consider another grammar:

\[ S \rightarrow S + S \mid S * S \mid \text{number} \]

- Different derivations produce different parse trees: ambiguous grammar

Differing Parse Trees

\[ S \rightarrow S + S \mid S * S \mid \text{number} \]

- Consider expression  \( 1 + 2 * 3 \)
- Derivation 1: \( S \rightarrow S + S \rightarrow 1 + S \rightarrow 1 + S * S \rightarrow 1 + 2 * 3 \)
- Derivation 2: \( S \rightarrow S * S \rightarrow S * S \rightarrow S + S \rightarrow S + 2 * 3 \rightarrow 1 + 2 * 3 \)

Impact of Ambiguity

- Different parse trees correspond to different evaluations!
- Meaning of program not defined

\[ \begin{align*}
1 + 2 * 3 & = 7 \\
1 + 2 * 3 & = 9
\end{align*} \]
Eliminating Ambiguity

• Often can eliminate ambiguity by adding non-terminals & allowing recursion only on right or left
  \[ S \rightarrow S + T \mid T \]
  \[ T \rightarrow T * \text{num} \mid \text{num} \]
• \( T \) non-terminal enforces precedence
• Left-recursion: left-associativity

Limits of CFGs

• Syntactic analysis can’t catch all “syntactic” errors
• Example: C++
  \[
  \text{HashTable<Key,Value>} x;
  \]
  Need to know whether \( \text{HashTable} \) is the name of a type to understand syntax! Problem: “<”, “,” are overloaded
• Iota:
  \[
  f(4)[1][2] = 0;
  \]
• Difficult to write grammar for LHS of assign – may be easier to allow all exprs, check later

CFGs

• Context-free grammars allow concise specification of programming languages
• CFG specifies how to convert token stream to parse tree (if unambiguous!)
• Read Appel 3.1, 3.2

Next time: implementing a top-down parser (leftmost derivation)