Where we are

Source code (character stream)

Token stream

Lexical analysis

Syntactic Analysis (specification)

Semantic Analysis

Where we are

Source code (character stream)

Lexical analysis

Token stream

Syntactic Analysis (specification)

Semantic Analysis

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

block

if_stmt

bin_op

variable

constant

==

block

while_stmt

bin_op

variable

constant

!=

block

expr_stmt

Abstract Syntax Tree

 block

 if_stmt

 bin_op

 variable

 constant

 ==

 block

 while_stmt

 bin_op

 variable

 constant

 !=

 block

 expr_stmt

 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

object

noun phrase: the

article:

ten: book

Overview of Syntactic Analysis

- Input: stream of tokens

- Output: 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 expressions

• 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

Limits of REs

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

Context-Free Grammars

• A specification of the balanced-parenthesis language:
  \[ S \rightarrow ( S ) S \]
  \[ S \rightarrow \epsilon \]
• The definition is recursive
• A context-free grammar
  – More expressive than regular expressions
  – \[ S = ( S ) \epsilon = (( S ) S) \epsilon = (( \epsilon ) \epsilon) \epsilon = (\epsilon) \]
• 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 \( \epsilon \)
  \[ S \rightarrow ( S ) S \]
• Non-terminals
  – Syntactic variables
  \[ S \rightarrow \epsilon \]
• 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 for 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)+)

Derivation Example

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

Derive \((1+2+(3+4))+5\):

\[ S \rightarrow E + S \rightarrow (S) + S \rightarrow (E + S) + S \]
\[ \rightarrow (1 + S) + S \rightarrow (1 + E + S) + S \]
\[ \rightarrow (1 + 2 + S) + S \rightarrow (1 + 2 + E) + S \]
\[ \rightarrow (1 + 2 + (S)) + S \rightarrow (1 + 2 + (E + S)) + S \]
\[ \rightarrow (1 + 2 + (3 + S)) + S \]
\[ \rightarrow (1 + 2 + (3 + E)) + S \]
\[ \rightarrow (1 + 2 + (3 + E)) + S \]
\[ \rightarrow (1 + 2 + (3 + E)) + E \]
\[ \rightarrow (1 + 2 + (3 + E)) + 5 \]

Constructing a derivation

• Start from start symbol: \( S \)

• Productions are used to derive a sequence of tokens from the start symbol

• For arbitrary strings \( \alpha, \beta \) and \( \gamma \)
  and a production \( A \rightarrow \beta \)
  A single step of derivation is
  \[ \alpha A \gamma \Rightarrow \alpha \beta \gamma \]
  \- i.e., substitute \( \beta \) for an occurrence of \( A \)

  \[ -(S + E) + E \rightarrow (E + S + E) + E \]
  \( (A = S, \beta = E + S) \)

Derivation \( \Rightarrow \) Parse Tree

• Tree representation of the derivation

• Leaves of tree are terminals;
  in-order traversal yields string

• Internal nodes: non-terminals

• No information about order of derivation steps

Derivation:

\[ S \rightarrow E + S \rightarrow (S) + S \rightarrow (E + S) + S \rightarrow (1 + S) + S \rightarrow (1 + E + S) + S \]
\[ \rightarrow (1 + 2 + S) + S \rightarrow (1 + 2 + E) + S \]
\[ \rightarrow (1 + 2 + (S)) + S \rightarrow (1 + 2 + (E + S)) + S \]
\[ \rightarrow (1 + 2 + (3 + S)) + S \]
\[ \rightarrow (1 + 2 + (3 + E)) + S \]
\[ \rightarrow (1 + 2 + (3 + E)) + E \]
\[ \rightarrow (1 + 2 + (3 + E)) + 5 \]

Sum grammar

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

\[ e.g. (1 + 2 + (3+4))+5 \]

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

4 productions

2 non-terminals: \( S, E \)

4 terminals: ( ), +, number

Start symbol \( S \)
Parse Tree

- Also called “concrete syntax”

![Parse Tree Diagram]

Derivation order

- Can choose to apply productions in any order; select any non-terminal
  \[ E + S \rightarrow 1 + S \text{ or } E + E + S \]
- 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

\[
S \rightarrow E + S | E
\]

- **Left-most derivation**
  \[
  S \rightarrow E + S \rightarrow (E + S) + S \rightarrow (1 + S) + S \rightarrow (1 + 2 + (3 + E)) + S + S \rightarrow (1 + 2 + (3 + 4)) + S + S \rightarrow (1 + 2 + (3 + 4)) + S + S \rightarrow (1 + 2 + (3 + 4)) + 5
  \]

- **Right-most derivation**
  \[
  S \rightarrow E + S \rightarrow E + E \rightarrow E + 5 \rightarrow (E + S) + 5 \rightarrow (E + E + S) + 5 \rightarrow (E + E + E + S) + 5 \rightarrow (E + E + E + E + S) + 5 \rightarrow (E + E + E + E + E + S) + 5 \rightarrow (E + E + E + E + E + E + S) + 5 \rightarrow (E + E + E + E + E + E + S) + 5 \rightarrow (E + E + E + E + E + E + S) + 5 \rightarrow (E + E + E + E + S) + 5 \rightarrow (E + E + E + S) + 5 \rightarrow (E + E + S) + 5 \rightarrow (E + S) + 5 \rightarrow (1 + 2 + 3 + E) + 5 \rightarrow (1 + 2 + 3 + 5 + S) + S \rightarrow (1 + 2 + (3 + 4)) + S + S \rightarrow (1 + 2 + (3 + 4)) + S + S \rightarrow (1 + 2 + (3 + 4)) + 5
  \]

- **Same parse tree**: same productions chosen, diff. order

Associativity

- + operator associates to right in parse tree regardless of derivation order
  \[
  E + S \rightarrow 1 + S + 2 + 3 + 4 + 5
  \]

- + associates to right because of **right-recursive** production \( S \rightarrow E + S \)
- In the example grammar, leftmost and rightmost derivations produce identical parse trees

An Ambiguous Grammar

- Consider another grammar:
  \[
  S \rightarrow S + S \mid S \times S \mid \text{ number}
  \]

- Different derivations produce different parse trees: ambiguous grammar

Differing Parse Trees

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

![Differing Parse Trees Diagram]
Impact of Ambiguity
- Different parse trees correspond to different evaluations!
  \[
  \begin{align*}
  1 & 2 & 3 \\
  1 & 2 & 3 \\
  \end{align*}
  = 7 \\
  = 9
  \]
- Meaning of program not well defined

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

if-then-else
- How to write a grammar for if stmts?
  \[
  S \rightarrow \text{if} (E)\ S \text{ else } S \\
  S \rightarrow \text{if} (E)\ S \\
  S \rightarrow X = E \mid \ldots
  \]
- Is this grammar ok?

No—Ambiguous!
- How to parse?
  \[
  \begin{align*}
  S &\rightarrow \text{if} (E) S \\
  &\rightarrow \text{if} (E) \text{ if } (E) S \text{ else } S \\
  S &\rightarrow \text{if} (E) S \text{ else } S \\
  &\rightarrow \text{if} (E) \text{ if } (E) S \text{ else } S
  \end{align*}
  \]
- Which “if” is the “else” attached to?

Grammar for Closest-if Rule
- Want to rule out \(\text{if } (E) \text{ if } (E) S \text{ else } S\)
- Problem: unmatched if may not occur as the “then” (consequent) clause of a containing “if”

Greedy ANTLR
- How to parse?
  \[
  \begin{align*}
  S &\rightarrow \text{if} (E) S \\
  &\rightarrow \text{if} (E) \text{ if } (E) S \text{ else } S \\
  S &\rightarrow \text{if} (E) S \text{ else } S \\
  &\rightarrow \text{if} (E) \text{ if } (E) S \text{ else } S
  \end{align*}
  \]
- Which “if” is the “else” attached to?
Greedy ANTLR

• ANTLR v4 grammar for if stmts:
  \[ S \rightarrow \text{if } (E) \; S \; (\text{else } S)? \]
  \[ S \rightarrow X = E \mid \ldots \]

• Leftmost derivations
• Greedy derivations

Limits of CFGs

• Syntactic analysis can’t catch all “syntactic” errors
• Example: C++
  – HashTable<Key,Value> x;
• Need to know whether 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
  – Or a derivation preference is designated

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