CS412/413

Introduction to
Compilers and Translators
Spring '00

Lecture 3: Syntactic Analysis

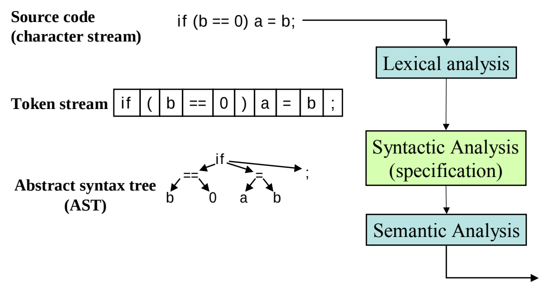
Outline

- Review of lexical analysis
- Context-Free Grammars (CFGs)
- Derivations
- Parse trees and abstract syntax
- Ambiguous grammars

Administration

- Homework 1 — due Monday
- Programming assignment 1 — due next Friday
- Everyone should have
  – a project group
  – a CSUGLAB account
  – received mail sent to cs412-students

Where we are

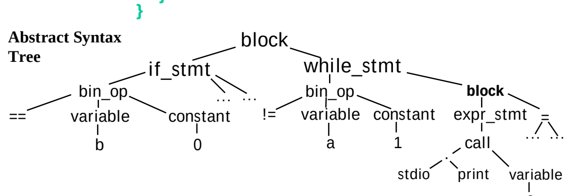


What is Syntactic Analysis?

Source code (token stream)

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

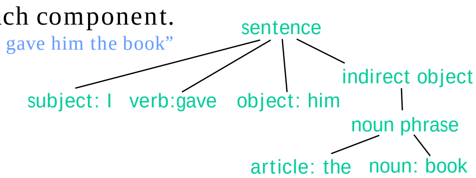
Abstract Syntax Tree



Parsing

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

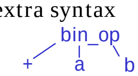
"I gave him the book"



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   (a) + (b)   ((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)

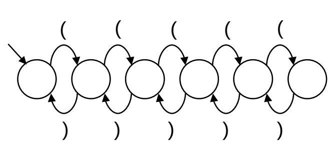
  ()   (())   ()()()   (())()((()()))

  ((   )(()) (()()

- 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



maximum depth: 5 parens

Context-Free Grammars

- A specification of the balanced-parenthesis language:

  $S \rightarrow (S) S$

  $S \rightarrow \varepsilon$

- The definition is recursive
- This is a **context-free grammar**
  - More expressive than regular expressions
  - $S = (S)\,\varepsilon = ((S)\,S)\,\varepsilon = ((\varepsilon)\,\varepsilon)\,\varepsilon = (())$

Definition of CFG

- Terminals
  - Token or ε
- Non-terminals
  - Syntactic variables
- 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

$S \rightarrow (S) S$
$S \rightarrow \varepsilon$

Sum grammar

$S \rightarrow E + S \mid E$

$E \rightarrow$ **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

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**]+))

Derivations

$S \rightarrow E + S \mid E$

$E \rightarrow \text{number} \mid (S)$

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

Derivation Example

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

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

$\boldsymbol{S} \rightarrow \boldsymbol{E} + S \rightarrow \boxed{(\boldsymbol{S})} + S \rightarrow (\boldsymbol{E} + S) + S$
$\rightarrow (1 + \boldsymbol{S}) + S \rightarrow (1 + \boldsymbol{E} + S) + S$
$\rightarrow (1 + 2 + \boldsymbol{S}) + S \rightarrow (1 + 2 + \boldsymbol{E}) + S$
$\rightarrow (1 + 2 + (\boldsymbol{S})) + S \rightarrow (1 + 2 + (\boldsymbol{E} + S)) + S$
$\rightarrow (1 + 2 + (3 + \boldsymbol{S})) + S$
$\rightarrow (1 + 2 + (3 + \boldsymbol{E})) + S$
$\rightarrow (1 + 2 + (3+4)) + \boldsymbol{S}$
$\rightarrow (1 + 2 + (3+4)) + \boldsymbol{E}$
$\rightarrow (1 + 2 + (3+4)) + 5$

replacement string

non-terminal being expanded

Constructing a derivation

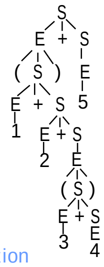
- Start from start symbol (S)
- Productions are used to derive a sequence of tokens from the start symbol
- For arbitrary strings α, β and γ and a production A → β
  A single step of derivation is

  $\alpha A \gamma \Rightarrow \alpha \beta \gamma$

  - *i.e.*, substitute β for an occurrence of A
  - $(\mathbf{S} + E) + E \rightarrow (\mathbf{E + S} + E) + E \qquad A = S, \beta = E + S$

Derivation ⇒ Parse Tree

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

(1+2+ (3+4))+5

$S \to E + S \mid E$

$E \to$ number | (S)

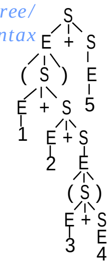
Derivation

S → **E** + S → (**S**) + S → (**E** + S)+ S → (1 + **S**)+S → (1 + **E** + S)+S → (1 + 2 + **S**)+S → ... → (1 + 2 + (**S**))+S→ (1 + 2 + (**E** + S))+S → ... → (1 + 2 + (3 + **E**))+S → ... → (1 + 2+ (3+4))+5

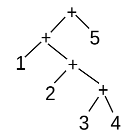
Parse Tree

- 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
- **Rightmost derivation**: find right-most non-terminal...etc.

Example

$S \to E + S \mid E$

$E \to$ number | (S)

- Left-most derivation

S →**E**+S →(**S**) + S → (**E** + S)+ S → (1 + **S**)+S → (1+**E**+S)+S → (1+2+**S**)+S → (1+2+**E**)+S → (1+2+(**S**))+S → (1+2+(**E**+S))+S → (1+2+(3+**S**))+S → (1+2+(3+**E**))+S → (1+2+(3+4))+**S** → (1+2+(3+4))+**E** → (1+2+(3+4))+5

- Right-most derivation

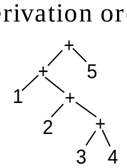
S →E+**S** →E+**E** → **E**+5 → (**S**)+5 → (E+**S**)+5 → (E+E+**S**)+5 → (E+E+**E**)+5 → (E+E+(**S**))+5 → (E+E+(E+**S**))+5 → (E+E+(E+**E**))+5 → (E+E+(**E**+4))+5 → (E+**E**+(3+4))+5→ (**E**+2+(3+4))+5 → (1+2+(3+4))+5

- 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 $S \to E + S$
- Consider another grammar:

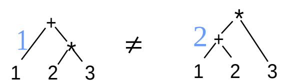
$$S \to S + S \mid S * S \mid \text{number}$$

- 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: $\boldsymbol{S} \rightarrow \boldsymbol{S} + S \rightarrow 1 + \boldsymbol{S} \rightarrow 1 + \boldsymbol{S} * S \rightarrow 1 + 2 * \boldsymbol{S} \rightarrow 1 + 2 * 3$
- Derivation 2: $\boldsymbol{S} \rightarrow S * \boldsymbol{S} \rightarrow \boldsymbol{S} * 3 \rightarrow S + \boldsymbol{S} * 3 \rightarrow \boldsymbol{S} + 2 * 3 \rightarrow 1 + 2 * 3$



Impact of Ambiguity

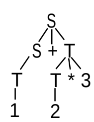
- Different parse trees correspond to different evaluations!
- Meaning of program ill-defined

Eliminating Ambiguity

- Often can eliminate ambiguity by adding non-terminals & allowing recursion only on right or left

$S \rightarrow S + T \mid T$

$T \rightarrow T * \textbf{num} \mid \textbf{num}$



- T non-terminal enforces precedence
- Left-recursion : left-associativity

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
- Read Appel 3.1, 3.2

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