COMPILER (CSE 4120)

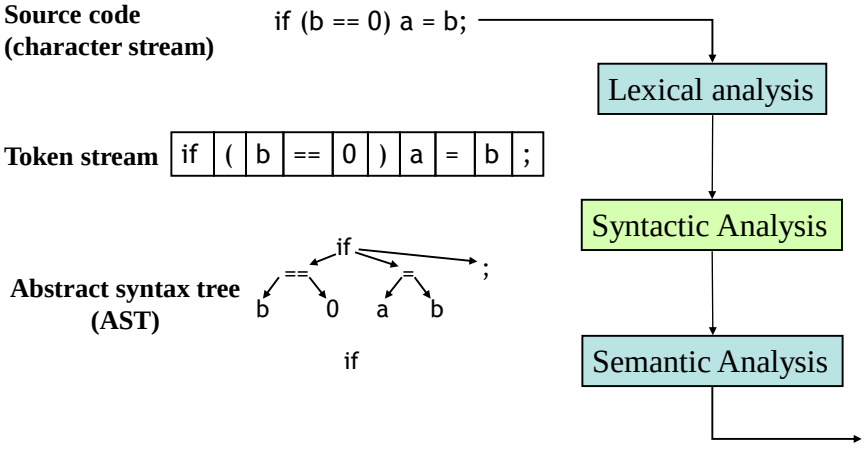
(Lecture 3: Parsing 1 – Syntactic Analysis)

Sungwon Jung

Mobile Computing & Data Engineering Lab
Dept. of Computer Science and Engineering
Sogang University
Seoul, Korea
Tel: +82-2-705-8930
Email : jungsung@sogang.ac.kr

Dept. of Computer Science at Sogang University

Where we are now?

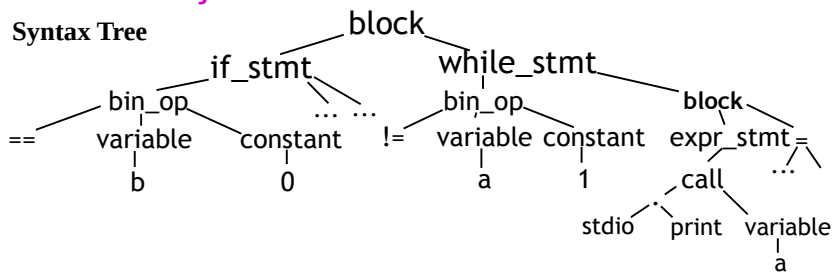


What is Syntactic Analysis?

Source code (token stream)

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

Syntax Tree

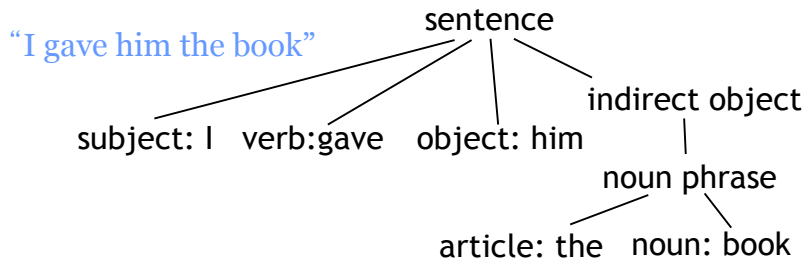


Overview of Syntactic Analysis

- Input:
  - stream of tokens
- Output:
  - abstract syntax tree
- Implementation:
  - Parse token stream to traverse concrete syntax
  - During traversal, build abstract syntax tree

Parsing

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



What Parsing Doesn't Do?

- Doesn't check many things:
  - type agreement, variables initialized
    - 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 Regular Expressions

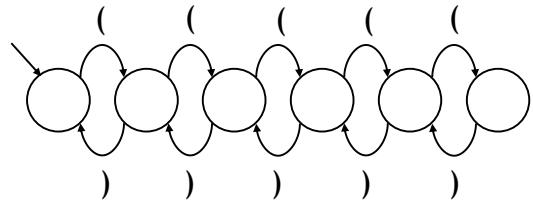
- Programming languages are not regular
  - cannot be described by regular expressions
- Consider: language of all strings that contain balanced parentheses

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

  (( )(()) (()()
- Problem:
  - Need to keep track of number of parentheses seen so far:
    - unbounded counting

Need more Power!

- Regular Expression = DFA
- DFA has only finite number of states; cannot perform unbounded counting



maximum depth: 5 parens

Context-Free Grammar

- 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
  - $S = (S)\,\varepsilon = ((S)\,S)\,\varepsilon = ((\varepsilon)\,\varepsilon)\,\varepsilon = (())$

Regular Expression is subset of CFG

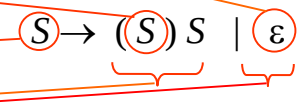
- Regular Expressions
  digit → [**0-9**]
  posint → *digit*+
  int → -? *posint*
  real → *int* (ε | (. *posint*))
- Symbolic names are shorthand
- All symbol names can be fully expanded: not recursive
  real → -? [**0-9**]+ (ε | (. [**0-9**]+))

Formal Definition of Context-free Grammar

- A grammar is (**T, N, S, P**)
  - **T** is the set of terminal (alphabet)
  - **N** is the set of non-terminal
  - **S** is the initial non-terminal
  - **P** is the set of production rules
    - **P** is a relation from **N** to $(\mathbf{N} \cup \mathbf{T})^*$

Definition of CFG

- Terminals
  - Symbols for strings or tokens (also ε)
- Non-terminals
  - Syntactic variables
- Start symbol
  - A special nonterminal is designated
- Production
  - Specifies how non-terminals may be expanded to form strings
  - LHS: single non-terminal, RHS: string of terminals and non-terminals



Sum Grammar

$S \rightarrow S + E \mid E$

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

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

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

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

Derivations

- $S \rightarrow S + E \mid E$
- $E \rightarrow$ number $\mid$ (S)

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

Derivation Example

$S \rightarrow E + S \mid E$

$E \rightarrow$ number $\mid$ (S)

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

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

replacement string

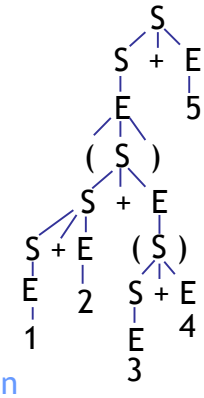
non-terminal being expanded

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
    - (S + E) + E → (S + E + E)+E A = S, β = S + E

Derivation ⇒ Parse Tree

Parse Tree



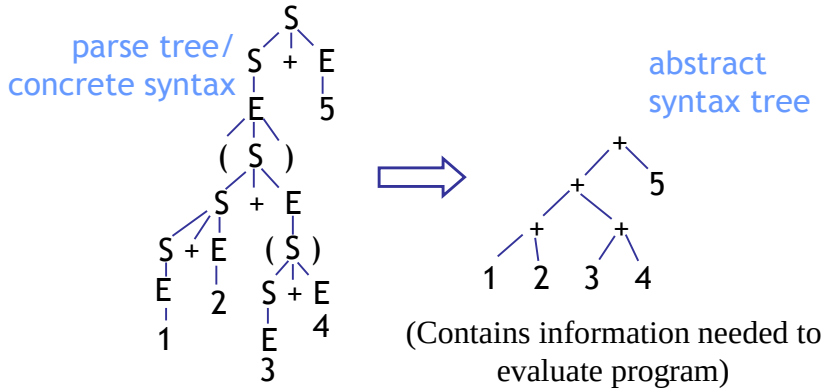
- Tree representation of the derivation
- Leaves of tree are terminals; in-order traversal yields string
- Internal nodes: non-terminals
- Loses information about order of derivation steps

Derivation

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

Parse Tree

- Also called "concrete syntax"



(Contains information needed to evaluate program)

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 automatic parsing
- Leftmost derivation
  - In the string, find the left-most non-terminal and apply a production to it

Example

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

- Left-most derivation

  $S \rightarrow S+E \rightarrow E+E \rightarrow (S)+E \rightarrow (S+E)+E \rightarrow (S+E+E)+E$
  $\rightarrow (E+E+E)+E \rightarrow (1+E+E)+E \rightarrow (1+2+E)+E \rightarrow (1+2+(S))+E$
  $\rightarrow (1+2+(S+E))+E \rightarrow (1+2+(E+E))+E$
  $\rightarrow (1+2+(3+E))+E \rightarrow (1+2+(3+4))+E \rightarrow (1+2+(3+4))+5$

- Right-most derivation

  $S \rightarrow S+E \rightarrow S+5 \rightarrow E+5 \rightarrow (S)+5 \rightarrow (S+E)+5 \rightarrow (S+(S))+5$
  $\rightarrow (S+(S+E))+5 \rightarrow (S+(S+4))+5 \rightarrow (S+(E+4))+5 \rightarrow (S+(3+4))+5$
  $\rightarrow (S+E+(3+4))+5 \rightarrow (S+2+(3+4))+5 \rightarrow (E+2+(3+4))+5$
  $\rightarrow (1+2+(3+4))+5$

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

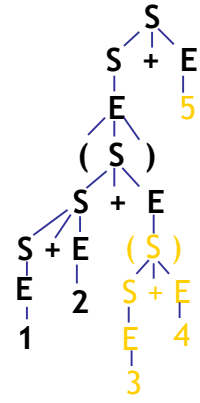
Top-down vs. Bottom-up Parsing

- We normally scan in tokens from left to right
- Left-most derivation reflects top-down parsing
  - Start with the start symbol
  - End with the string of tokens

$S \rightarrow S+E \rightarrow E+E \rightarrow (S)+E \rightarrow (S+E)+E \rightarrow (S+E+E)+E$
$\rightarrow (E+E+E)+E \rightarrow (1+E+E)+E \rightarrow (1+2+E)+E \rightarrow (1+2+(S))+E$
$\rightarrow (1+2+(S+E))+E \rightarrow (1+2+(E+E))+E \rightarrow (1+2+(3+E))+E$
$\rightarrow (1+2+(3+4))+E \rightarrow (1+2+(3+4))+5$

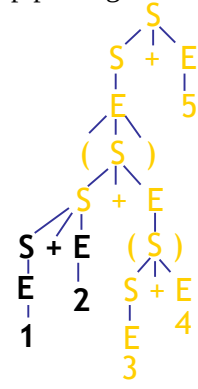
Top-down Parsing

- $S \rightarrow S+E \rightarrow E+E \rightarrow (S)+E \rightarrow (S+E)+E$
  $\rightarrow (S+E+E)+E \rightarrow (E+E+E)+E$
  $\rightarrow (1+E+E)+E \rightarrow (1+2+E)+E$ ...
- Entire tree above a token (2) has been expanded when encountered



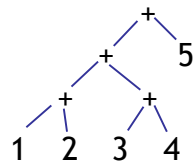
Bottom-up Parsing

- Right-most derivation reflects bottom-up parsing
  - Start with the tokens
  - End with the start symbol
- $(1+2+(3+4))+5 \leftarrow (E+2+(3+4))+5$
  $\leftarrow (S+2+(3+4))+5 \leftarrow (S+E+(3+4))+5$
  $\leftarrow (S+(3+4))+5 \leftarrow (S+(E+4))+5$
  $\leftarrow (S+(S+4))+5 \leftarrow (S+(S+E))+5$
  $\leftarrow (S+(S))+5 \leftarrow (S+E)+5$
  $\leftarrow (S)+5 \leftarrow E+5 \leftarrow S+E \leftarrow S$
- Advantage of bottom-up parsing:
  - Can select production based on more information



Ambiguous Grammars

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



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

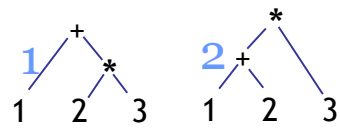
An Ambiguous Grammar

- + associates to left because of production
  - $S \rightarrow S + E$
- Consider another grammar:
  - $S \rightarrow S + S \mid S * S \mid$ number
- An ***ambiguous grammar*** is one that produce more than one left-most or more than one right-most derivation for some sentence

Differing Parse Trees

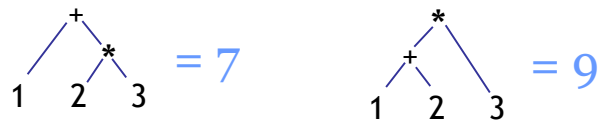
$$S \rightarrow S + S \mid S * S \mid \text{number}$$

- Consider expression 1 + 2 * 3
- Left-most Derivation 1:
  - $S \rightarrow S + S \rightarrow 1 + S \rightarrow 1 + S * S \rightarrow 1 + 2 * S \rightarrow 1 + 2 * 3$
- Left-most Derivation 2:
  - $S \rightarrow S * S \rightarrow S + S * S \rightarrow 1 + S * S \rightarrow 1 + 2 * S \rightarrow 1 + 2 * 3$



Impact of Ambiguity

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



Another Ambiguous Grammar

$E \rightarrow E + E \mid E - E$
$\mid E * E \mid E / E$
$\mid E \uparrow E \mid (E)$
$\mid -E \mid id$

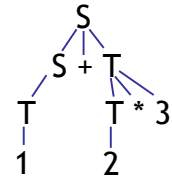
- Derivation 1:
  - $E \rightarrow E + E \rightarrow \mathbf{id} + E \rightarrow \mathbf{id} + E * E \rightarrow \mathbf{id} + \mathbf{id} * E \rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$
- Derivation 2:
  - $E \rightarrow E * E \rightarrow E + E * E \rightarrow \mathbf{id} + E * E \rightarrow \mathbf{id} + \mathbf{id} * E \rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$

Eliminating Ambiguity

- Often can eliminate ambiguity by adding non-terminals

  $S \rightarrow S + T \mid T$

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



- Enforces precedence, left-associativity

Eliminating Ambiguity

- Can disambiguate an ambiguous grammar by specifying the associativity and precedence of the arithmetic operators
  - - (Unary minus) : right-associative
  - ↑ : right-associative : a ↑ b ↑ c
    to mean a ↑ (b ↑ c) rather than (a ↑ b) ↑ c
  - * / : left-associative : a * b / c
    to mean (a * b) / c rather than a * (b / c)
  - + - : left-associative : a - b - c
    to mean (a - b) - c rather than a - (b - c)
- Example: a + -b ↑ c + d * e : (a + ((-b) ↑ c)) + (d * e)

Eliminating Ambiguity

- An Equivalent Unambiguous Grammar obeying the associativity and precedence rules:
  - Introduce one nonterminal for each precedence level
  - A subexpression that is essentially indivisible we shall call an **element**
  - An element is either a single identifier or a parenthesized expression
- $E \rightarrow E + T \mid E - T \mid T$
  $T \rightarrow T * F \mid T / F \mid F$
  $F \rightarrow G \uparrow F \mid G$
  $G \rightarrow -G \mid H$
  $H \rightarrow id \mid (E)$

Summary

- Context-free grammars allow concise specification of programming languages
- More powerful than regular expressions
- CFG converts token stream to parse tree