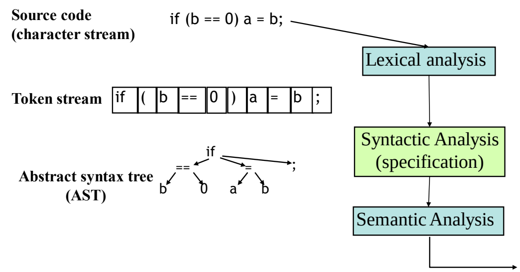
CS 412
Introduction to Compilers

Ross Tate
Cornell University

Lecture 4: Syntactic Analysis

Where we are

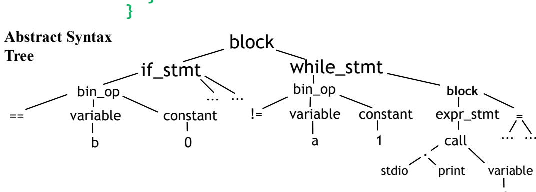


What is Syntactic Analysis?

Source code (token stream)

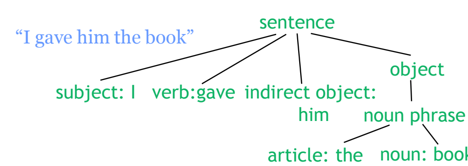
```
{
    if (b == (0)) a = b;
    while (a != 1) {
        stdio.print(a);
        a = a - 1;
    }
}
```

Abstract Syntax Tree



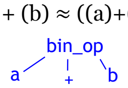
Parsing

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



Overview of Syntactic Analysis

- Input: stream of tokens
- Output: 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?

7

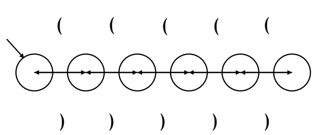
Limits of REs

- 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

8

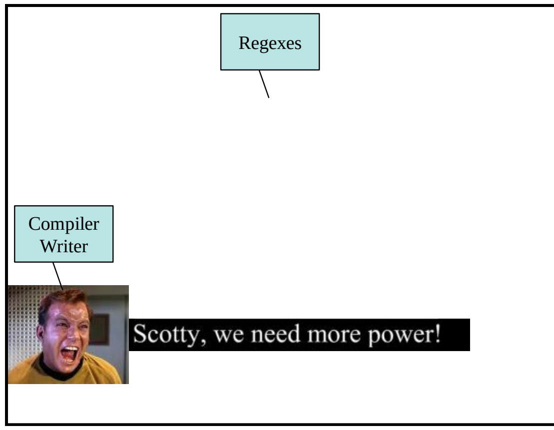
Limits of REs

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



maximum depth: 5 parens

9



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 = $(S)\ \varepsilon$ = $((S)\ S)\ \varepsilon$ = $((\varepsilon)\ \varepsilon)\ \varepsilon$ = $(())$
- If a grammar accepts a string, there is a *derivation* of that string using the productions of the grammar

11

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

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

12

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**]+))

Sum grammar

S → *E* + *S* | *E*
E → **number** | (*S*)

e.g. (1 + 2 + (3+4))+5

S → *E* + *S*
S → *E*
E → number
E → (*S*)

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

Derivation Example

S → *E* + *S* | *E*
E → number | (*S*)

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

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

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
  αAγ ⟹ αβγ
  – *i.e.*, substitute β for an occurrence of A
  – (**S** + E) + E → (**E** + **S** + E)+E
  (A = S, β = E + S)

Derivation Example

S → *E* + *S* | *E*
E → number | (*S*)

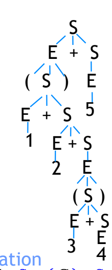
Derive (1+2+ (3+4))+5:

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

replacement string
non-terminal being expanded

Derivation ⟹ Parse Tree

Parse Tree



⇦ (1+2+ (3+4))+5

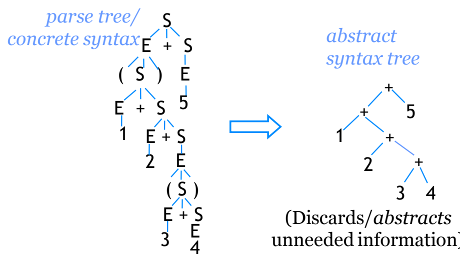
S → *E* + *S* | *E*
E → number | (*S*)

- 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 → ***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"



(Discards/*abstracts* unneeded information)

19

Derivation order

- Can choose to apply productions in any order; select any non-terminal
  E + S → 1 + S 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. $E + S \rightarrow E + E + S$

20

Example

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

- 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

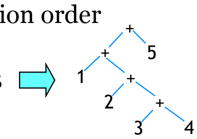
- Same parse tree: same productions chosen, diff. order

21

Associativity

- + operator associates to right in parse tree regardless of derivation order

(1+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

22

An Ambiguous Grammar

- Consider another grammar:

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

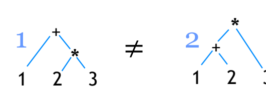
- Different derivations produce different parse trees: ambiguous grammar

23

Differing Parse Trees

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

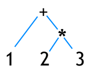
- Consider expression 1 + 2 * 3
- Derivation 1: $\boldsymbol{S} \rightarrow \boldsymbol{S} + S \rightarrow 1 + \boldsymbol{S} \rightarrow 1 + \boldsymbol{S} * S \rightarrow 1 + 2 * \boldsymbol{S} \rightarrow 1 + 2 * 3$
- Derivation 2: $\boldsymbol{S} \rightarrow S * \boldsymbol{S} \rightarrow \boldsymbol{S}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



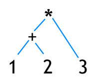
24

Impact of Ambiguity

- Different parse trees correspond to different evaluations!



= 7



= 9

- Meaning of program not well defined

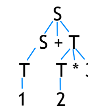
25

Eliminating Ambiguity

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

$S \rightarrow S + T \mid T$

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



- S/T separation enforces precedence
- Left-recursion : left-associativity

26

if-then-else

- How to write a grammar for if stmts?

  $S \rightarrow$ if (E) S else S

  $S \rightarrow$ if (E) S

  $S \rightarrow X = E \mid \ldots$

- Is this grammar ok?

27

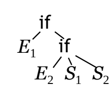
No—Ambiguous!

- How to parse?

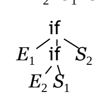
  if (E_1) if (E_2) S_1 else S_2

$S \rightarrow$ if (E) S
$S \rightarrow$ if (E) S else S
$S \rightarrow$ *other*

$S \rightarrow$ if (E) S
$\rightarrow$ if (E) if (E) S else S



$S \rightarrow$ if (E) S else S
$\rightarrow$ if (E) if (E) S else S



- *Which "if" is the "else" attached to?*

28

Grammar for Closest-if Rule

- Want to rule out if (E) if (E) S else S
- Problem: unmatched if may not occur as the "then" (consequent) clause of a containing "if"

statement $\rightarrow$ *matched* | *unmatched*
matched $\rightarrow$ if (E) *matched* else *matched*
  | *other*
unmatched $\rightarrow$ if (E) *statement*
  | if (E) *matched* else *unmatched*

29

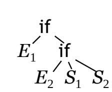
Greedy ANTLR

- How to parse?

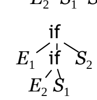
  if (E_1) if (E_2) S_1 else S_2

$S \rightarrow$ if (E) S
$S \rightarrow$ if (E) S else S
$S \rightarrow$ *other*

$S \rightarrow$ if (E) S
$\rightarrow$ if (E) if (E) S else S



$S \rightarrow$ if (E) S else S
$\rightarrow$ if (E) if (E) S else S



- *Which "if" is the "else" attached to?*

30

Greedy ANTLR

- ANTLR v4 grammar for if stmts:
  - $S \rightarrow$ if (E) S (else S)?
  - $S \rightarrow X = E$ | …
- Leftmost derivations
- Greedy derivations

31

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

32

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)

33