



CSCI 2320: Principles of Programming Languages

Syntactic Analysis

Ch 3 (Tucker-Noonan)

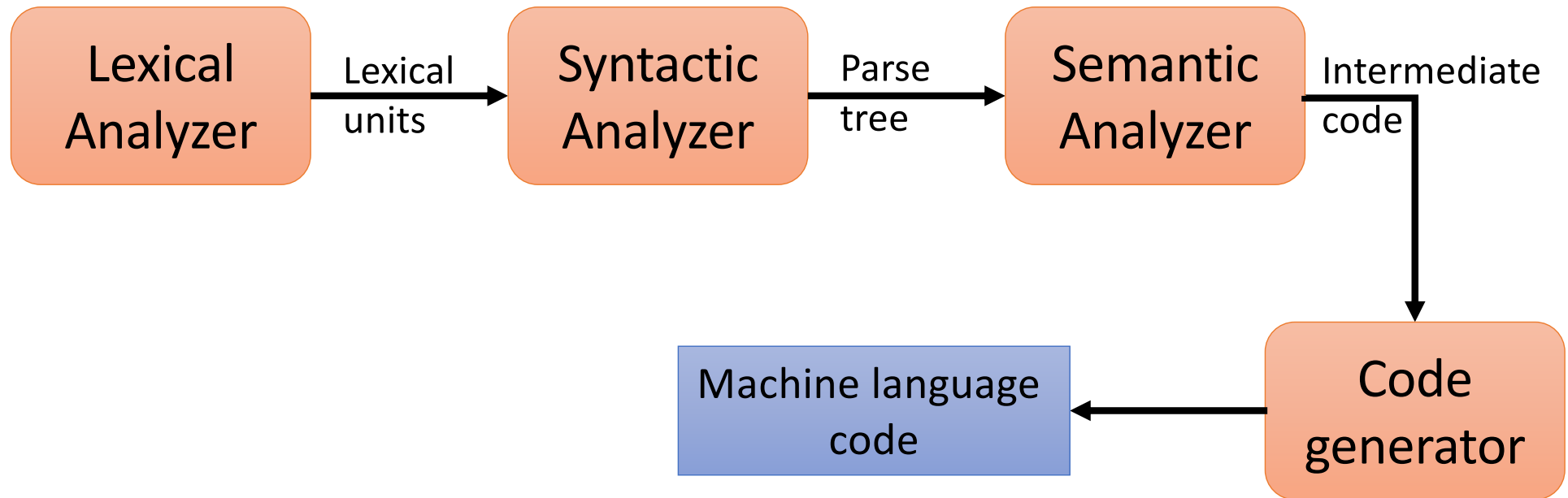
Handouts:

Canvas -> Modules -> Handouts -> Syntax and Semantics

Mohammad T. Irfan

Compiler

`i = 15 + 5 * 2;`

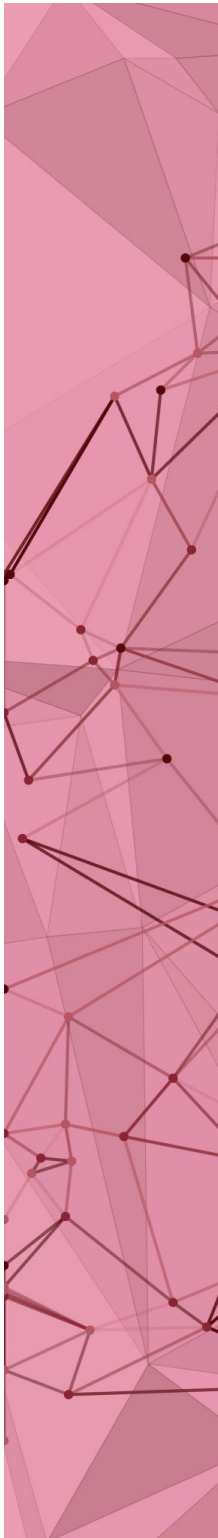


We can run it **later**

Lexical analysis



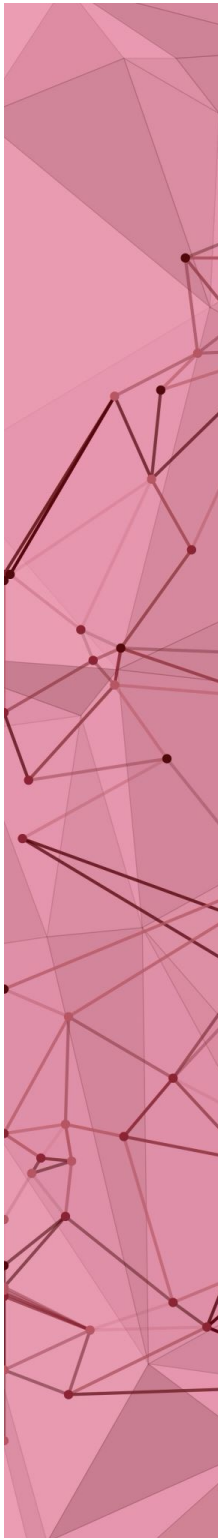
Great news: regular grammar/expression works



Syntactic analysis or parsing



Q: Will regular grammar/expression work?

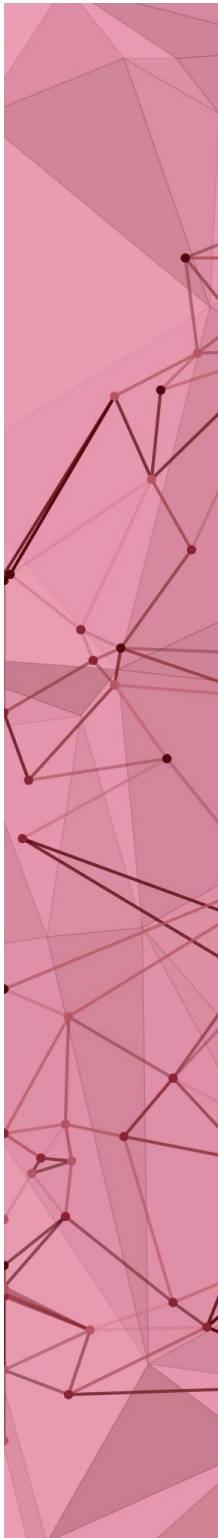


Parsing question

Inputs: Grammar G and a string of terminals x

Is x in L_G ?

L_G = Language corresponding to G
= Set of strings generated by G





Types of parsers



Top-down parsing: Recursive descent (RD)

RD parser for assignment stmt

Assignment \rightarrow Id = Expr;

Expr \rightarrow Term {AddOp Term}

AddOp \rightarrow + | -

Term \rightarrow Factor {MulOp Factor}

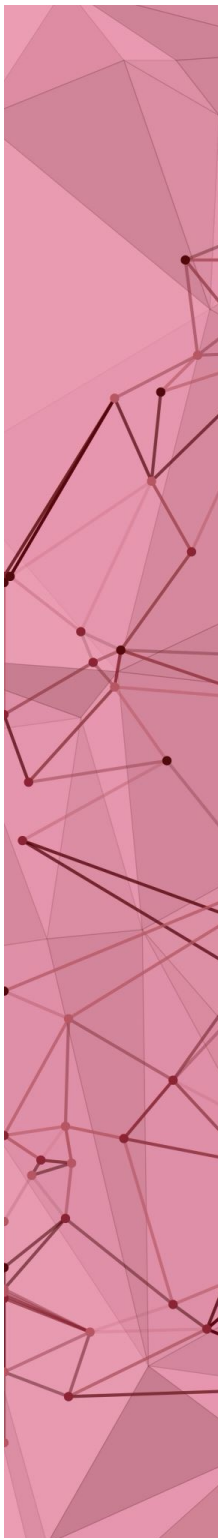
MulOp \rightarrow * | /

Factor \rightarrow [UnaryOp] Primary

UnaryOp \rightarrow -

Primary \rightarrow Id | IntLiteral | FloatLiteral | (Expr)

... <Lexical syntax for Id, IntLiteral, FloatLiteral> ...



Python code for smaller version

Expr \rightarrow Term $\{ (+ \mid -) \text{ Term} \}$

Term \rightarrow Factor $\{ (* \mid /) \text{ Factor} \}$

Factor \rightarrow IntLiteral

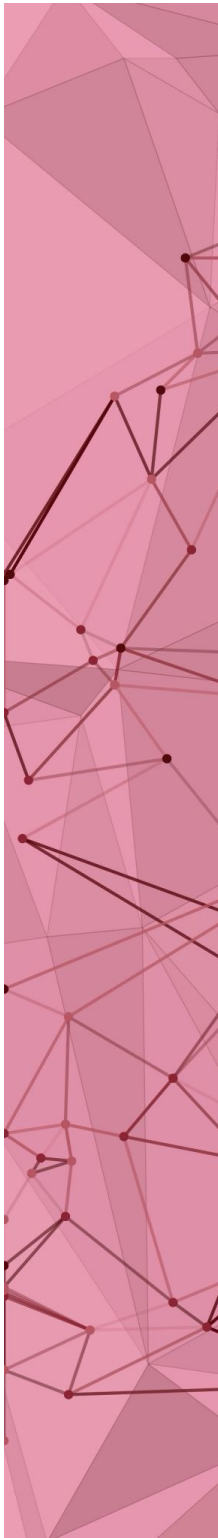
... <Lexical syntax for IntLiteral> ...

Download code from Canvas -> Modules -> Code -> Parsing



Requirements for RD parser

1. Know the FIRST set for each non-terminal
2. Do "left factoring"
3. Remove left recursions (why?)

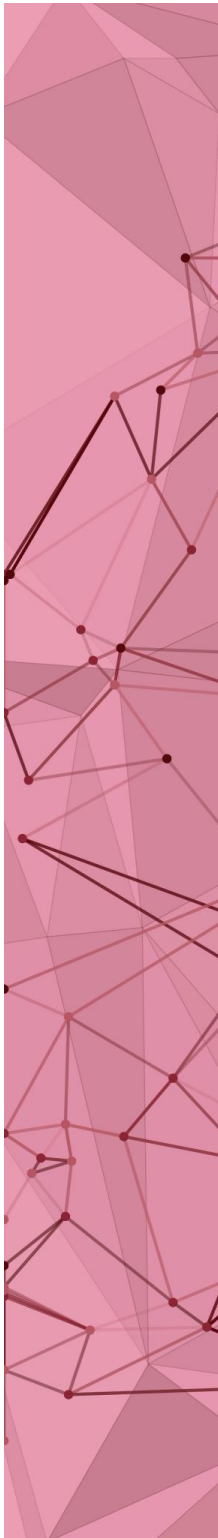


FIRST set

$\text{FIRST}(X)$ = the set of all terminal symbols (or tokens) that can occur as the leftmost symbol in a string derived from X .

For a terminal symbol a , $\text{FIRST}(a) = \{a\}$

Find FIRST sets of non-terminal symbols in C Lite.



Concrete Syntax: C Lite

$Program \rightarrow Type \text{ main } () \{ Declarations Statements \}$

$Declarations \rightarrow \{ Declaration \}$ 

$Declaration \rightarrow Type Identifier [[Integer]] \{ , Identifier [[Integer]] \} ;$

$Type \rightarrow \text{int} \mid \text{bool} \mid \text{float} \mid \text{char}$

$Statements \rightarrow \{ Statement \}$

$Statement \rightarrow ; \mid Block \mid Assignment \mid IfStatement \mid WhileStatement$

$Block \rightarrow \{ Statements \}$

$Assignment \rightarrow Identifier [[Expression]] = Expression ;$

$IfStatement \rightarrow \text{if } (Expression) Statement [\text{else } Statement]$

$WhileStatement \rightarrow \text{while } (Expression) Statement$

Concrete Syntax (cont...)

$Expression \rightarrow Conjunction \{ \mid \mid Conjunction \}$
 $Conjunction \rightarrow Equality \{ \&\& Equality \}$
 $Equality \rightarrow Relation [EquOp Relation]$
 $EquOp \rightarrow == \mid !=$
 $Relation \rightarrow Addition [RelOp Addition]$
 $RelOp \rightarrow < \mid <= \mid > \mid >=$
 $Addition \rightarrow Term \{ AddOp Term \}$
 $AddOp \rightarrow + \mid -$
 $Term \rightarrow Factor \{ MulOp Factor \}$
 $MulOp \rightarrow * \mid / \mid \%$
 $Factor \rightarrow [UnaryOp] Primary$
 $UnaryOp \rightarrow - \mid !$
 $Primary \rightarrow Identifier [[Expression]] \mid Literal \mid (Expression) \mid$
 $Type (Expression)$

Note: Operators and punctuation symbols are part of "lexical syntax" (next)

Left factoring

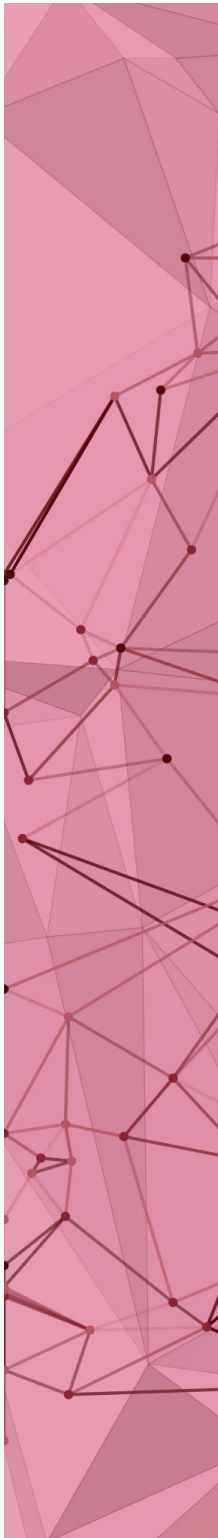
- $\text{IfStmt} \rightarrow \text{if Expr then Stmt}$
- $\text{IfStmt} \rightarrow \text{if Expr then Stmt else Stmt}$
- Why can't LL(1) parser deal with it?
- Solution

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$



$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$



Removing left recursion


- Example
- Algorithm (assume no cycle; i.e., no $A \xRightarrow{+} A$)

Nonterminals: A_1, A_2, \dots, A_n (ordered arbitrarily)

For $i = 1$ to n

For each $j < i$

Let $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$



Replace each $A_i \rightarrow A_j \gamma$ by

$A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$

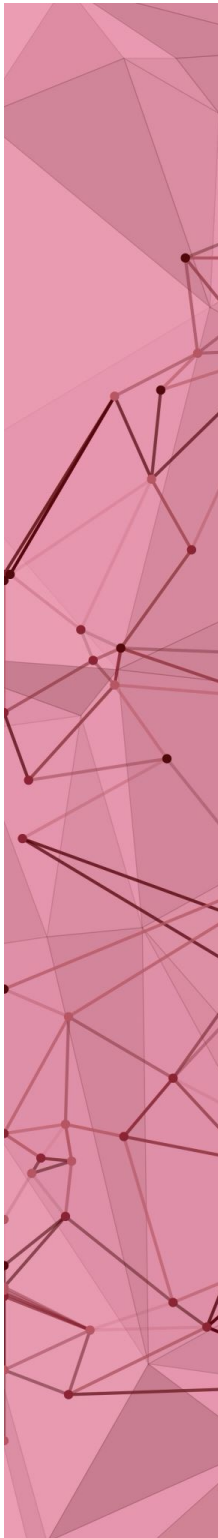
Eliminate left recursion from all A_i products



Table-driven LL(1) parser

Grammar format: BNF

- Remove left recursion (can't do EBNF loops)
- Do left factoring
- Augment the grammar with an artificial start symbol S as follows: $S \rightarrow \text{Actual_Start_Symbol } \$$



$\text{FIRST}(\alpha) \equiv \{a : \alpha \Rightarrow^* a \beta\} \cup (\text{if } \alpha \Rightarrow^* \epsilon \text{ then } \{\epsilon\} \text{ else } \emptyset)$

$\text{FOLLOW}(A) \equiv \{a : S \Rightarrow^+ \alpha A a \beta\} \cup (\text{if } S \Rightarrow^* \alpha A \text{ then } \{\epsilon\} \text{ else } \emptyset)$

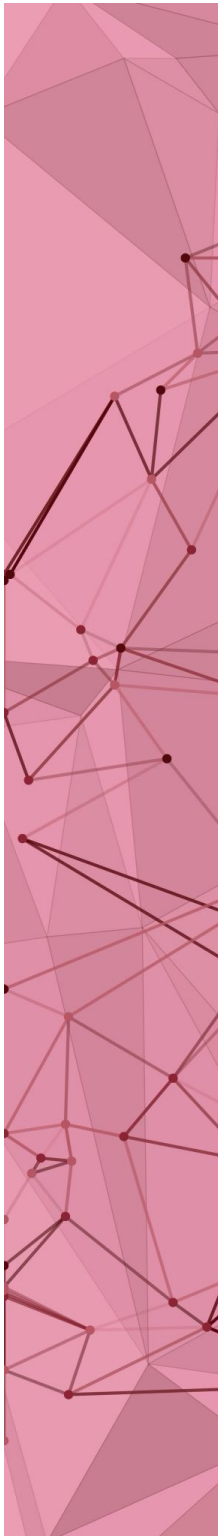
Algorithm

<https://www.cs.princeton.edu/courses/archive/spring20/cos320/LL1/>

Reference: handout (Prof. Irfan's notes)

Textbook reference: pg. 81

Caution: The textbook does not detail how the table is constructed





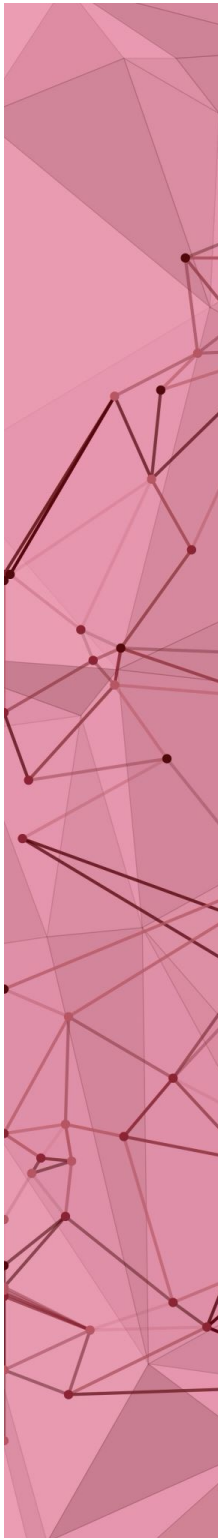
RD vs. Table-driven

Comparison

How to choose a parser?

Literature review

- NP-complete:
Given a CFG, is there an LL(1) parser?
- Impossibility example:
 $L_G = \{a^n 0 b^n \mid n \geq 1\} \cup \{a^n 1 b^{2n} \mid n \geq 1\}$
 - Why is LL(1) parsing impossible here?



Literature review

- Is there always a parser (not necessarily LL(1)) for *any* CFG?
- **CYK algorithm**: Cocke & Younger (1967) and Kasami (1965)
 - First parser for *any* CFG
 - Bottom-up parser: Dynamic prog. $O(n^3)$
- Earley's algorithm (1970)
 - Top-down: Dynamic prog. $O(n^3)$
- **Frost (2007)**: First top-down parser for any CFG; improved by Ridge (2014)

