CSCI 2320: Principles of Programming Languages

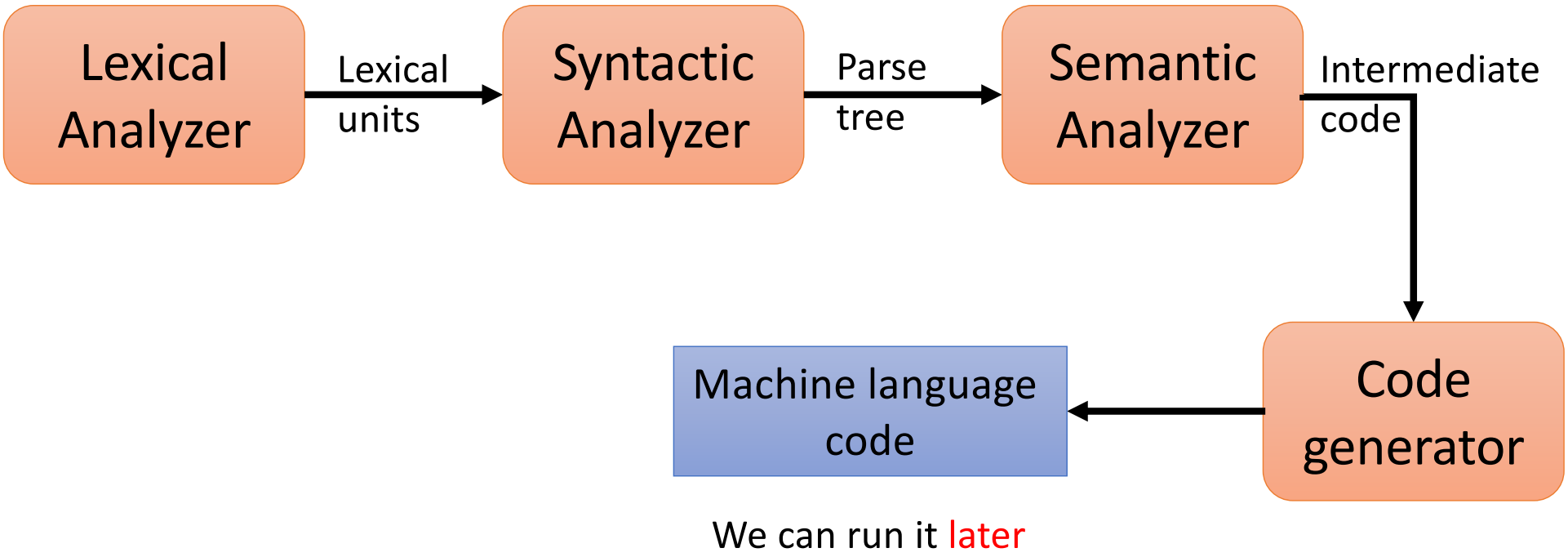
Syntactic Analysis

Ch 3 (Tucker-Noonan)
Handouts:
Canvas -> Modules -> Handouts -> Syntax and Semantics

Mohammad T. Irfan

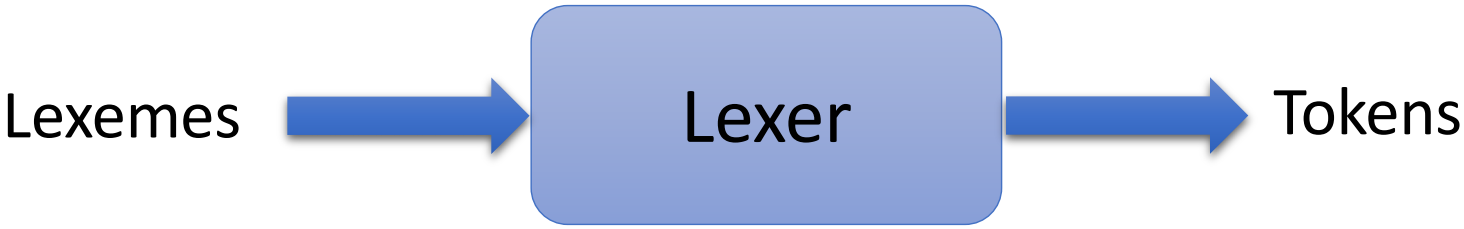
Compiler

```
i = 15 + 5 * 2;
```



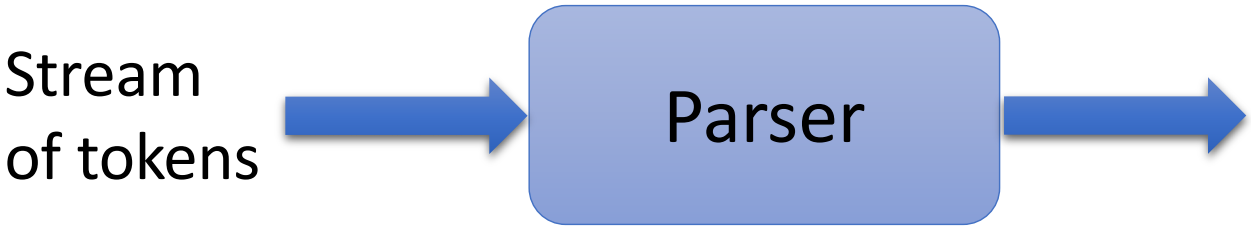
We can run it later

Lexical analysis



Great news: regular grammar/expression works

Syntactic analysis or parsing



Q: Will regular grammar/expression work?

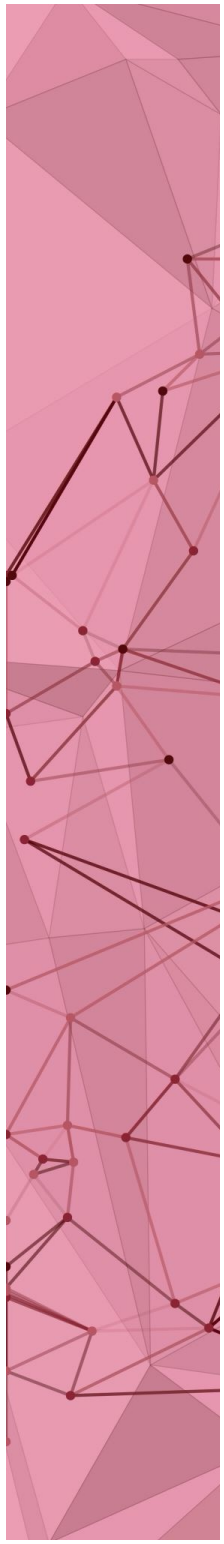
Parsing question

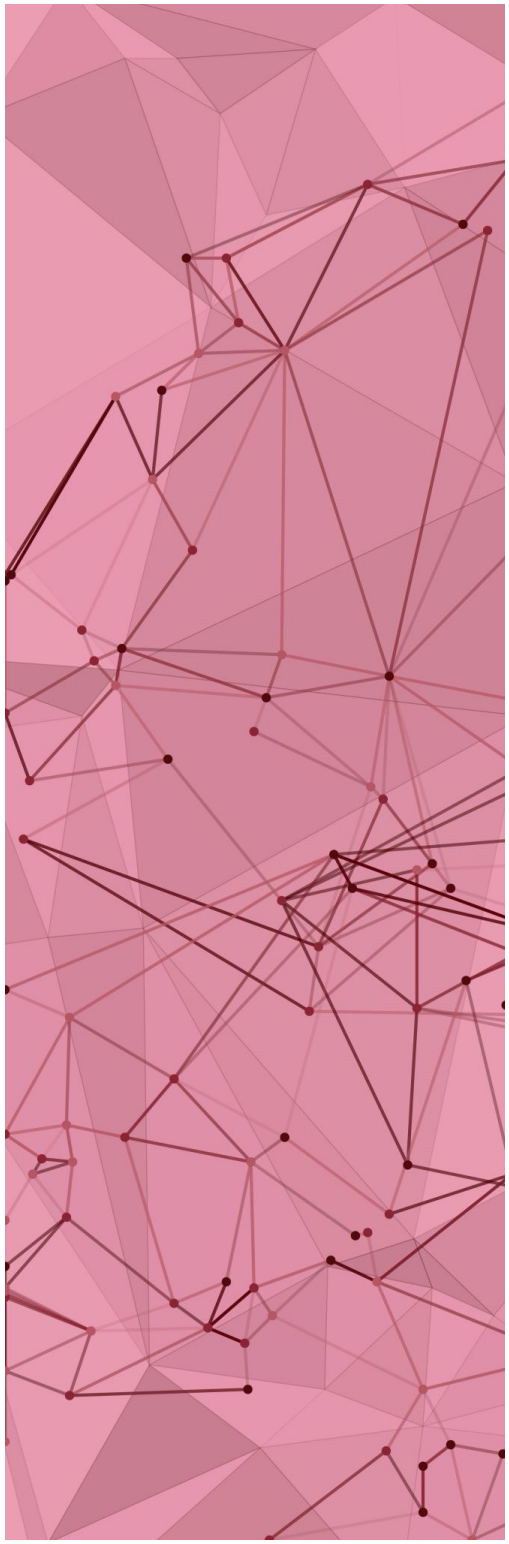
Inputs: Grammar G and a string of terminals **x**

Is **x** in L_G?

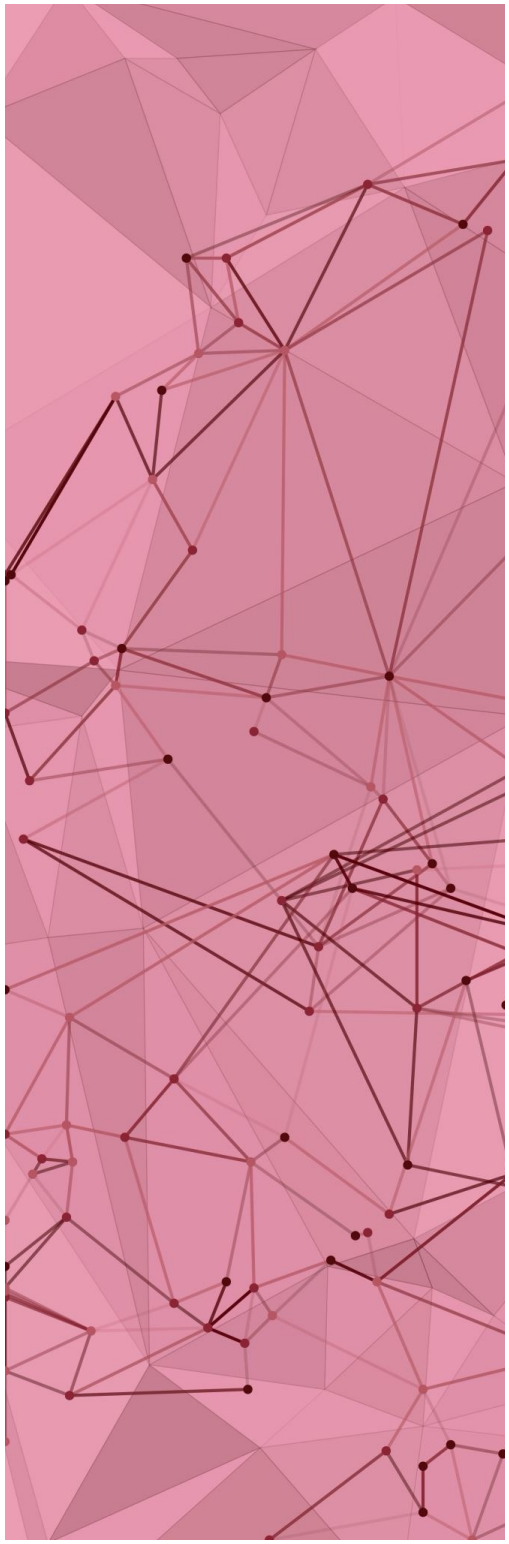
L_G = Language corresponding to G

= Set of strings generated by G





Types of parsers



Top-down parsing: Recursive descent (RD)

RD parser for assignment stmt

Assignment → Id = Expr;

Expr → Term {AddOp Term}

AddOp → + | -

Term → Factor {MulOp Factor}

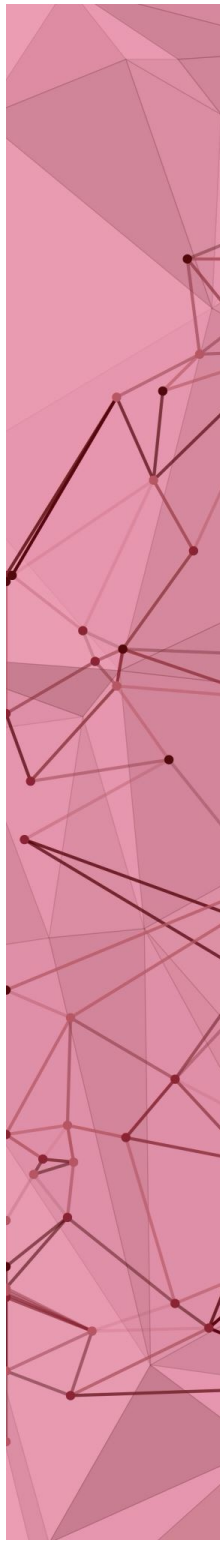
MulOp → * | /

Factor → [UnaryOp] Primary

UnaryOp → -

Primary → Id | IntLiteral | FloatLiteral | (Expr)

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



Python code for smaller version

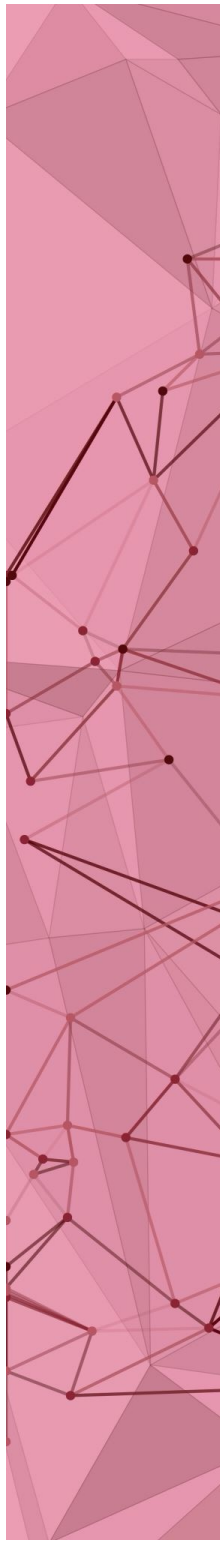
Expr → Term {(+|-) Term}

Term → Factor {(*|/) Factor}

Factor → IntLiteral

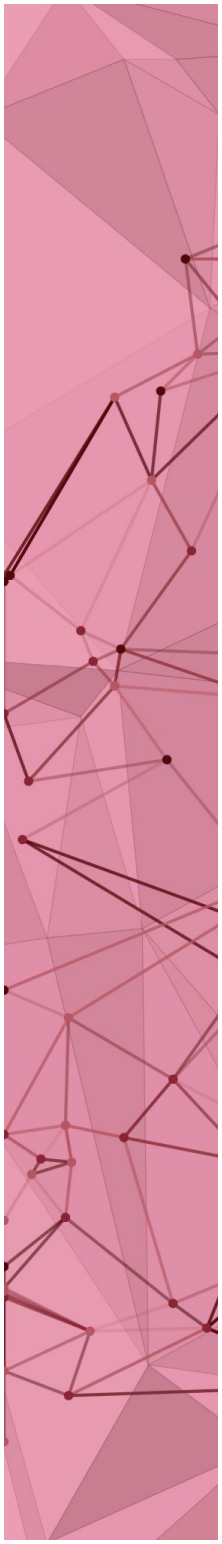
... <Lexical syntax for IntLiteral> ...

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?)



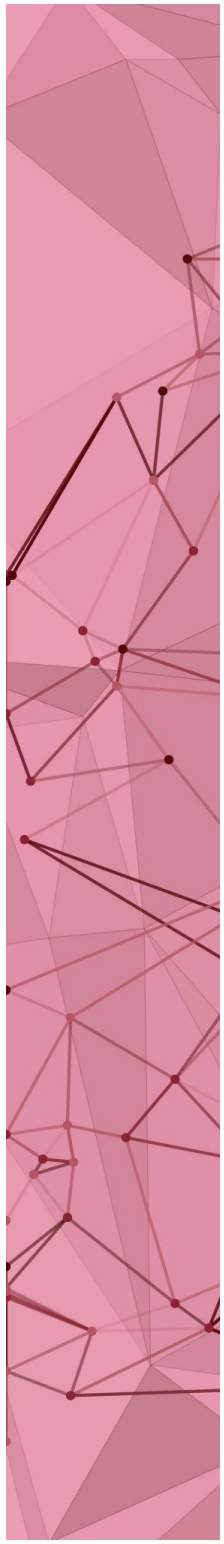
FIRST set

FIRST(α) = the set of all terminal symbols (or tokens) that can occur as the leftmost symbol in a string derived from α.

- α is a string of terminal and/or nonterminal symbols.

For a terminal symbol a, FIRST(a) = {a}

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



Concrete Syntax: C Lite

Program → *Type* `main ( )` { *Declarations Statements* }

Declarations → { *Declaration* }

Declaration → *Type Identifier* [[*Integer*]] { , *Identifier* [[*Integer*]] } ;

Type → `int | bool | float | char`

Statements → { *Statement* }

Statement → ; | *Block* | *Assignment* | *IfStatement* | *WhileStatement*

Block → { *Statements* }

Assignment → *Identifier* [[*Expression*]] = *Expression* ;

IfStatement → `if` (*Expression*) *Statement* [`else` *Statement*]

WhileStatement → `while` (*Expression*) *Statement*

Blue color: EBNF meta-symbol

Concrete Syntax (cont...)

Expression → *Conjunction* { || *Conjunction* }

Conjunction → *Equality* { && *Equality* }

Equality → *Relation* [*EquOp Relation*]

EquOp → == | !=

Relation → *Addition* [*RelOp Addition*]

RelOp → < | <= | > | >=

Addition → *Term* { *AddOp Term* }

AddOp → + | −

Term → *Factor* { *MulOp Factor* }

MulOp → * | / | %

Factor → [*UnaryOp*] *Primary*

UnaryOp → − | !

Primary → *Identifier* [[*Expression*]] | *Literal* | (*Expression*) | *Type* (*Expression*)

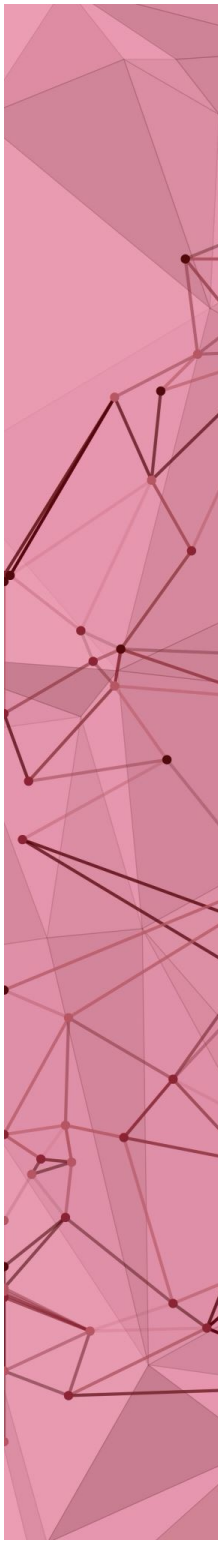
Note: Operators and punctuation symbols are part of "lexical syntax" (next)

Left factoring

- `IfStmt → if Expr then Stmt`
- `IfStmt → if Expr then Stmt else Stmt`
- Why can't LL(1) parser deal with it?
- Solution
  - Find the largest prefix α and factor it out

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

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



Removing left recursion

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

Nonterminals: $A_1, A_2, \ldots, A_n$ (ordered arbitrarily)

For i = 1 to n

    For each j < i

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

        Replace each $A_i \rightarrow A_j \gamma$ by
        $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$

    Eliminate left recursion from all A_i products

No left recursion here

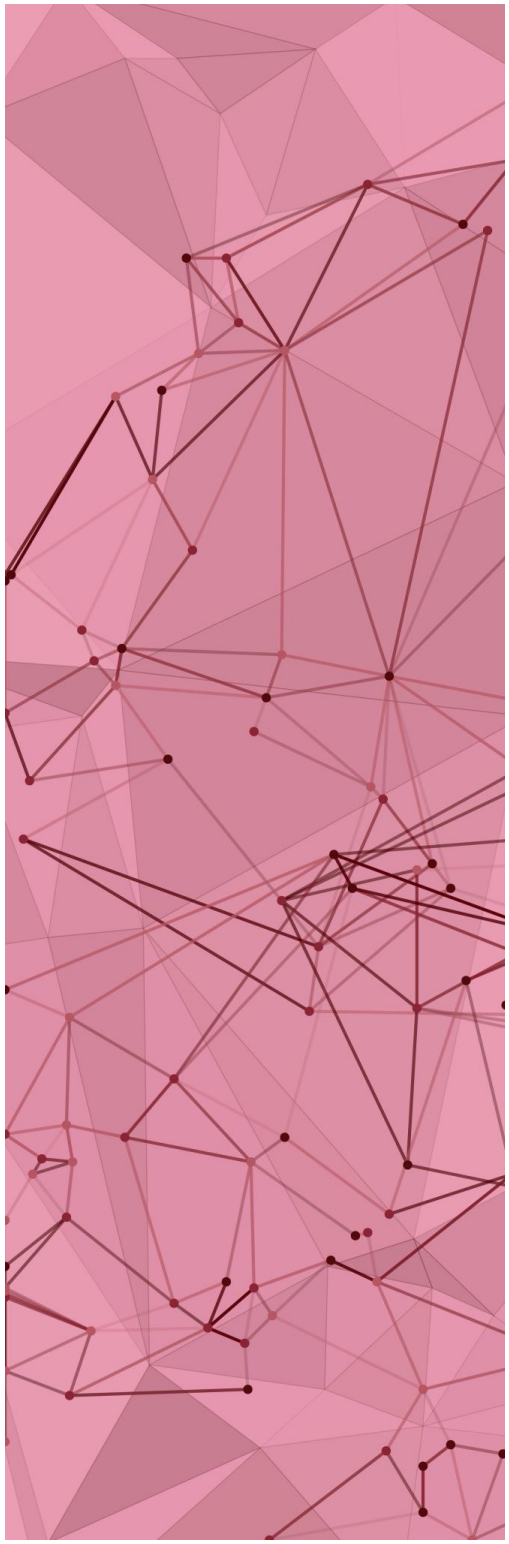
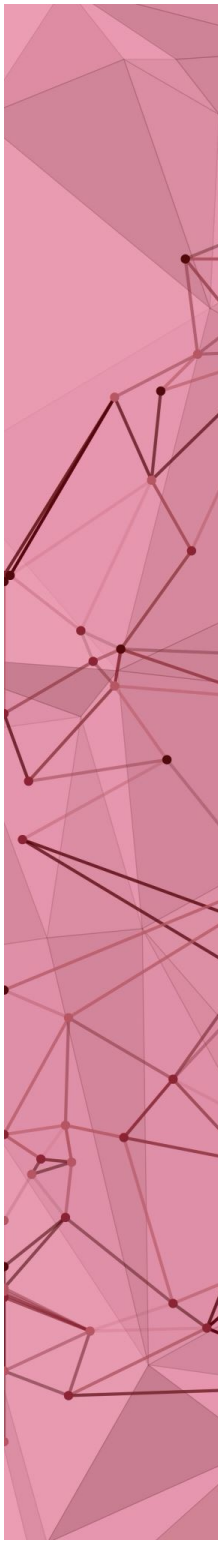


Table-driven LL(1) parser

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 → Actual_Start_Symbol $



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

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

Programming Language Pragmatics by Michael Scott

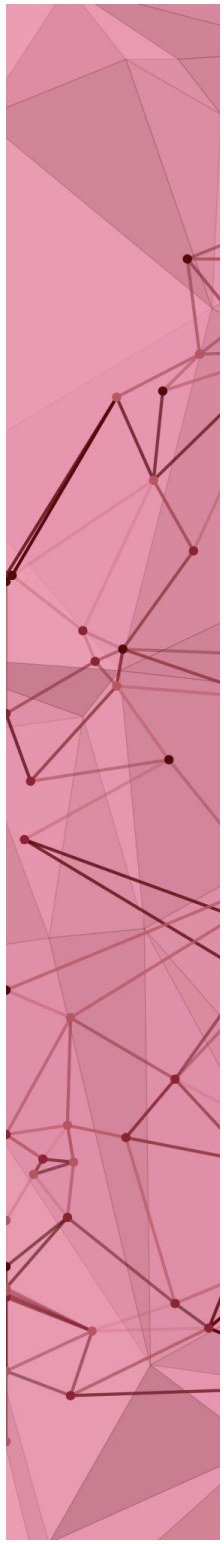
Table-driven LL(1) algorithm

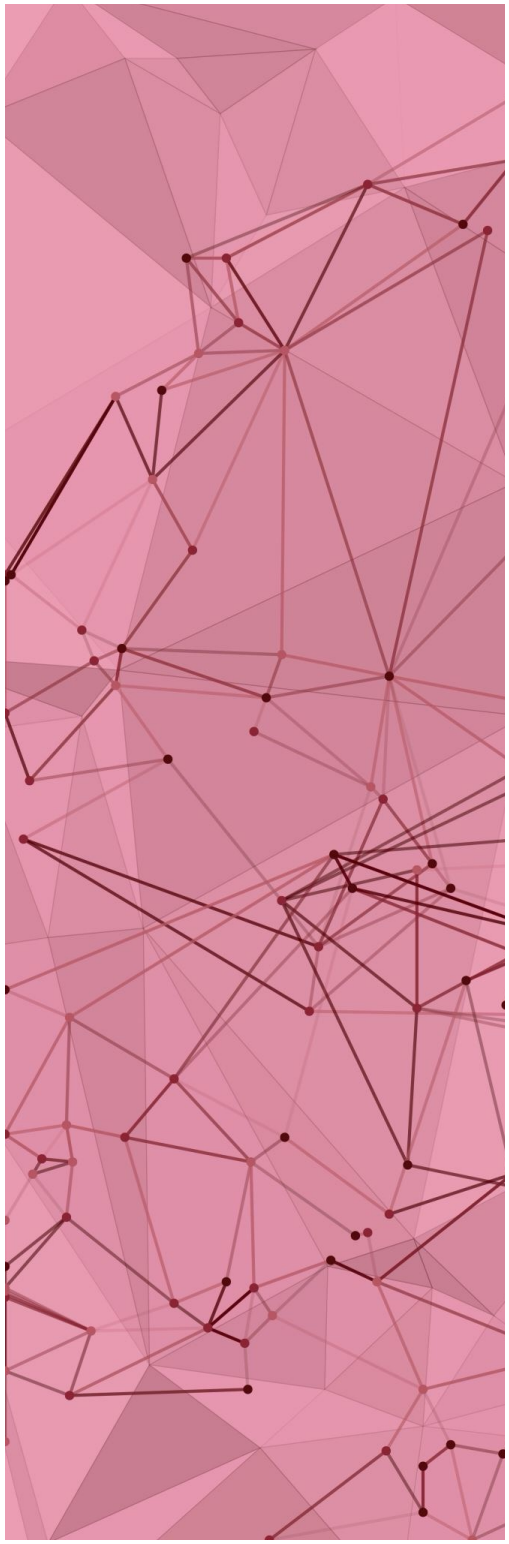
https://www.cs.princeton.edu/courses/archive/spring20/cos320/LL1/

Reference: handout (Prof. Irfan's notes)

Textbook reference: pg. 81

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





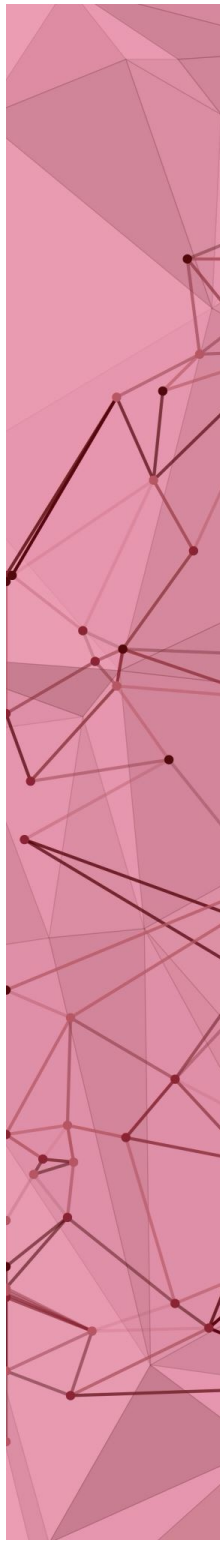
RD vs. Table-driven

Comparison

How to choose a parser?

Literature review

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