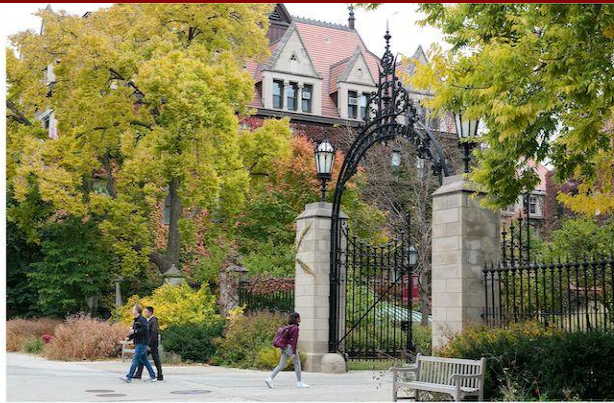


MPCS 51300 - Compilers

M3: Syntactical Analysis (Parsers)

Remote Students please mute your microphones, thank you.



Lamont Samuels

Copyright 2010, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Agenda

- Syntactical analysis overview
- Formal grammars: CFG
- Practical issues: ambiguity, left v. right recursion
- Top-down parsing

Syntactical Analysis (Parsing)

- **Goal:** Convert the token stream from the scanner into an abstract syntax tree and to verify the structure of the program is valid.

- Input source code: `if (x==y) x=45;`

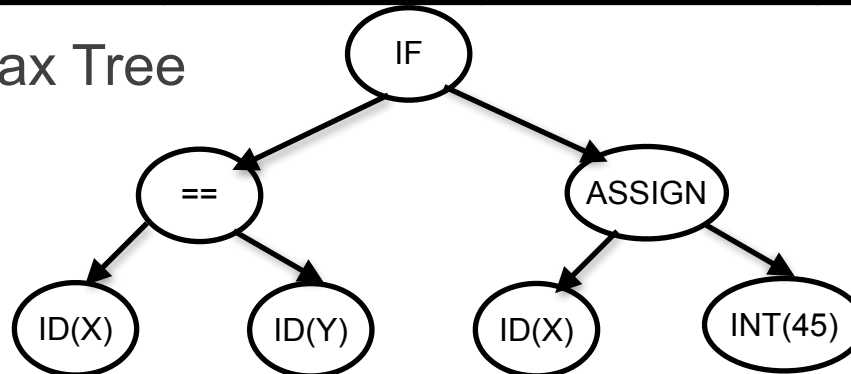
- Character Stream:

i	f	(x	=	y)	x	=	4	5	;
---	---	---	---	---	---	---	---	---	---	---	---

- Token Stream:

IF	LPAREN	ID(x)	EQ	ID(y)	RPAREN	ID(x)	ASSIGN	INT(45)	SCOLON
----	--------	-------	----	-------	--------	-------	--------	---------	--------

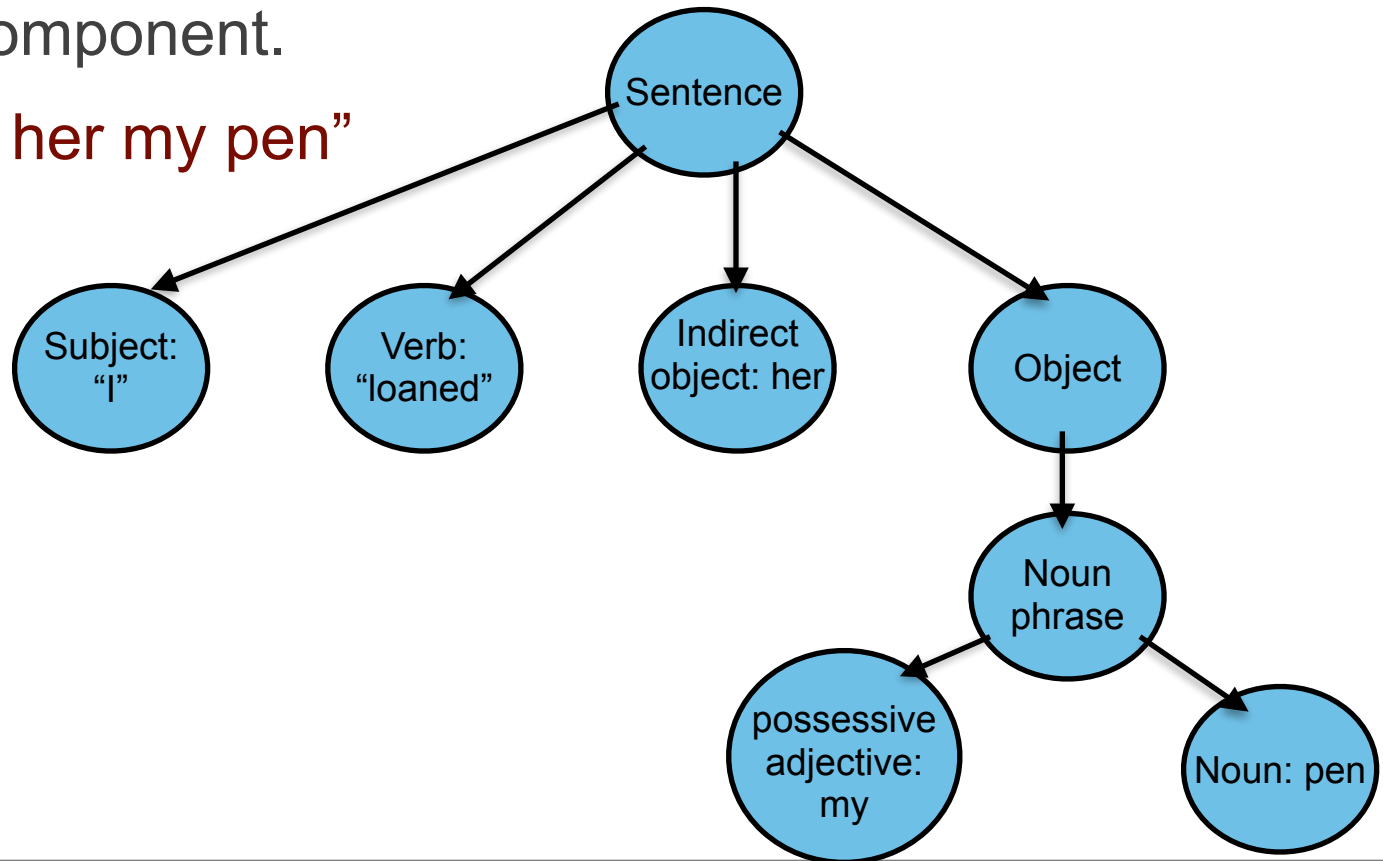
- Abstract Syntax Tree



Syntactical Analysis (Parsing)

- Determines whether a program (or sentence) is grammatically well-formed and identifies the function for each component.

“I loaned her my pen”



What Syntactical Analysis does not do?

- Type checking, variable declarations and initializations, function declarations, etc.

```
var a int  
b := 10  
c = foo(a)
```

- Deferred until semantic analysis

Specification of Language Syntax

- **Goal:** How can we specify the language syntax precisely and conveniently to make it easy to parse source code?
- Lexical Analysis: We used regular expressions to describe tokens
 - Made it easy to convert them to DFAs and simulated the DFAs to produce the tokens
- Why don't we just use regular expressions to specify programming language syntax?

Regular Expressions for Parsing?

- What if we wanted to add expression summation to the language

digits = [0-9]+
sum = (digits "+")* digits

- Defines sums of the form "34+93+1234"

- Now let's add parentheses to the language?

digits = [0-9]+
sum = expr "+" expr
expr = digits | "(" sum ")"

- But all of these are just abbreviations so it really looks like

digits = [0-9]+
expr = digits | "(" expr "+" expr ")"

- Now expand again:

expr = digits |
"(" (digits | "(" expr "+" expr ")")
"+" (digits | "(" expr "+" expr ")") ")"

Problem!! Regular expressions must
be finite and have no recursive structure

Limits to Regular Expressions

- Languages are not regular and cannot be described by regular expressions.
 - DFA has only a finite number of states so adding parenthesis would require some form of counting which is not doable with regular expressions.
- We need a way to specify nesting or specifying a recursive structure for various language constructs?
Grammars!

Grammars

- A grammar is a precise, and declarative specification of syntactic structure of programming languages
- The format (i.e., notation) of grammars is normally specified using Extended Backus-Naur Form (EBNF)
 - A set of **rewriting rules** (also called **productions**)
 $\text{Stmt} \rightarrow \text{if Expr then Stmt else Stmt}$
 $\text{Expr} \rightarrow \text{Expr} + \text{Expr} \mid \text{Expr} * \text{Expr} \mid (\text{Expr}) \mid \text{id}$
 Vertical bar is shorthand for multiple productions
 - A set of **non-terminals** (appears on the LHS of a production) and a set of **terminals** (token from the alphabet)
 non-terminals = Stmt, Expr
 terminals = **if, then, else, +, *, (,), id**
 - Can specify lists using recursion
 $\text{Block} \rightarrow \{ \text{Stmt-list} \}$
 $\text{StmtBlock} \rightarrow \text{Stmt} \mid \text{Stmt} ; \text{StmtBlock}$

Context-Free Grammars

- Regular expressions with recursion (i.e., more expressive than regular expressions)
- Defined by the following (T, N, P, S) :
 - T is set of terminals
 - N is set of non-terminals
 - P is set of productions (rewriting rules)
 - S is the start symbol (belongs to N)
- Example:

$G=(T,N,P,S)$

EBNF

$T = \{ +, *, (,), \text{id} \}$

$N = \{ E \}$

$P = \{ E \rightarrow E + E, \\ E \rightarrow E * E, E \rightarrow (E), E \rightarrow \text{id} \}$

$S=E$

$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$

CFG Example

- Sum grammar on integers

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

$S \rightarrow E + S$	4 productions
$S \rightarrow E$	2 non-terminals (S,E)
$E \rightarrow \text{INT}$	4 terminals: (,), +, INT
$E \rightarrow (S)$	Start symbol S

- Each context-free grammar defines a context-free language L , which contains all sentences of terminal symbols derived from repeated application of productions from the starting symbol.
 - Example language sentences from the Sum grammar
(1 + 2), 2, 4 + 21, ((3+3)+5)

Derivations

- We can show if a sentence is part of a language by performing a **derivation**
 - Starting with the start symbol, repeatedly replace a non-terminal (using a production) on its right hand side.

$E \Rightarrow E * E$
 $\Rightarrow id + E$
 $\Rightarrow id * (E + E)$
 $\Rightarrow id * (id + E)$
 $\Rightarrow id * (id + id)$

CFG
 $E \rightarrow E + E$
 $E \rightarrow E * E$
 $E \rightarrow (E)$
 $E \rightarrow id$

- The intermediate forms ($id + E$, $id * (id + E)$, etc.) always contain non-terminals.

Derivation Order

- Can choose to apply productions in any order.
 - For some arbitrary strings α , and γ and a production $A \rightarrow \beta$, a single step of a derivation is
$$\alpha A \gamma \Rightarrow \alpha \beta \gamma \text{ (substitute } \beta \text{ for an occurrence of } A)$$
- Two standard orders: **leftmost** derivation and **rightmost** derivation
 - Leftmost derivation: at each step, the leftmost non-terminal is replaced
$$\begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow \mathbf{id} + E \\ &\Rightarrow \mathbf{id} * \mathbf{id} \end{aligned}$$
 - Rightmost derivation: at each step, the rightmost non-terminal is replaced
$$\begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow E + \mathbf{id} \\ &\Rightarrow \mathbf{id} * \mathbf{id} \end{aligned}$$

Derivation Example

$$\begin{aligned} S &\rightarrow E + S \mid E \\ E &\rightarrow \text{INT} \mid (S) \end{aligned}$$

- Derive $((34+3)+4)+9$

Left-most derivation

$$\begin{aligned} S &\Rightarrow E + S \\ &\Rightarrow (S) + S \\ &\Rightarrow (E + S) + S \\ &\Rightarrow ((S) + S) + S \\ &\Rightarrow ((E+S) + S) + S \\ &\Rightarrow ((34+S) + S) + S \\ &\Rightarrow ((34+E) + S) + S \\ &\Rightarrow ((34+3) + S) + S \\ &\Rightarrow ((34+3) + E) + S \\ &\Rightarrow ((34+3) + 4) + S \\ &\Rightarrow ((34+3) + 4) + E \\ &\Rightarrow ((34+3) + 4) + 9 \end{aligned}$$

Right-most derivation

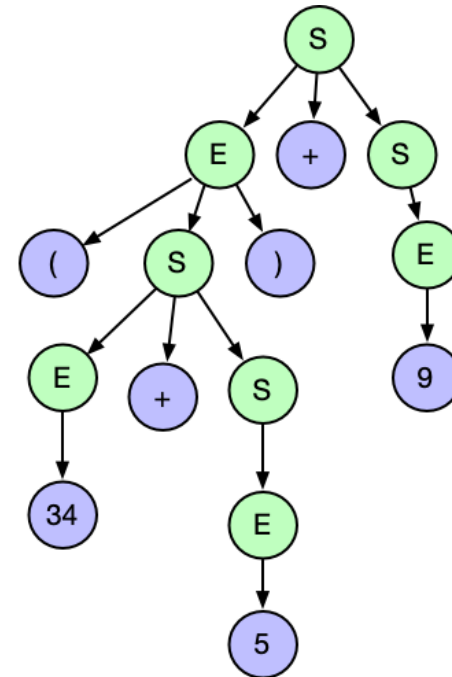
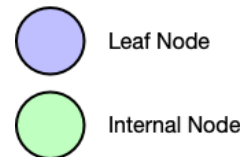
$$\begin{aligned} S &\Rightarrow E + S \\ &\Rightarrow E + E \\ &\Rightarrow E + 9 \\ &\Rightarrow (S) + 9 \\ &\Rightarrow (E + S) + 9 \\ &\Rightarrow (E + E) + 9 \\ &\Rightarrow (E + 4) + 9 \\ &\Rightarrow ((S) + 4) + 9 \\ &\Rightarrow ((E + S) + 4) + 9 \\ &\Rightarrow ((E + E) + 4) + 9 \\ &\Rightarrow ((E + 3) + 4) + 9 \\ &\Rightarrow ((34 + 3) + 4) + 9 \end{aligned}$$

Derivation to Parse Tree

- A parse tree is a tree representation of a derivation

(34 + 5) + 9 derivation

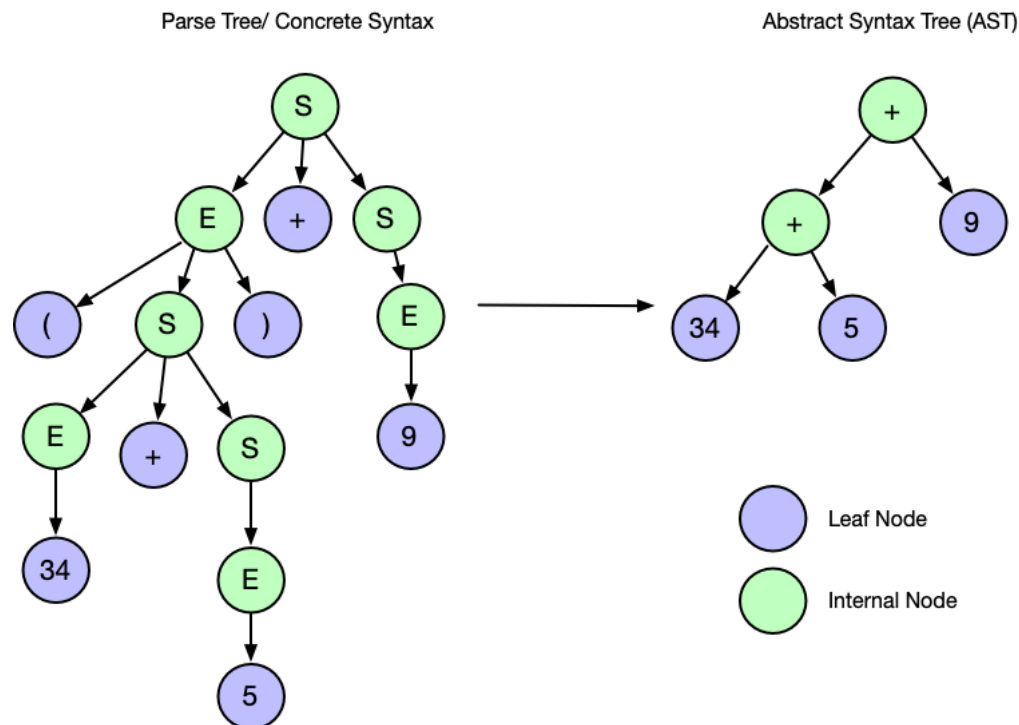
$S \Rightarrow E + S$
 $\Rightarrow (S) + S$
 $\Rightarrow (E + S) + S$
 $\Rightarrow (34 + S) + S$
 $\Rightarrow (34 + E) + S$
 $\Rightarrow (34 + 5) + S$
 $\Rightarrow (34 + 5) + E$
 $\Rightarrow (34 + 5) + 9$



- Leaves of a parse tree are terminals and internal nodes are non-terminals.
 - In-order traversal yields a sentence from the language
 - Non information about order of derivation steps (although we used left-most in the above example)

Parse Tree vs. Abstract Syntax Tree

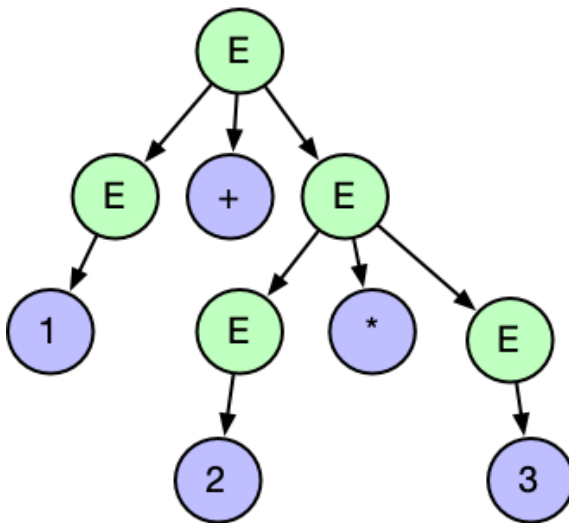
- A parse tree is also known as “concrete syntax”
- An AST is similar to a parse tree but discards/abstracts out unnecessary information



Ambiguous Grammars

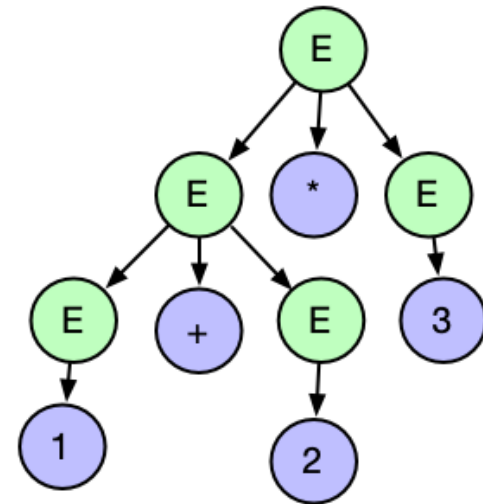
- A grammar is **ambiguous** if it can derive a sentence with two different parse trees (i.e., there's more than one leftmost (or rightmost) derivation).
- To see this, let's look at this grammar $E \rightarrow E + E \mid E * E \mid \text{INT}$
- Consider the expression: $1 + 2 * 3$

Leftmost Parse Tree



\neq

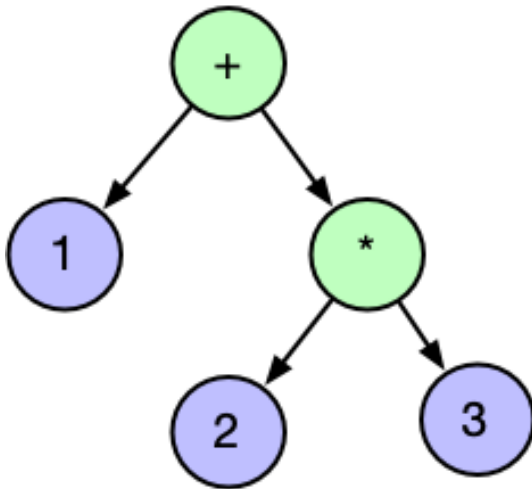
Rightmost Parse Tree



Ambiguous Grammar

- Different parse trees will evaluate to different results.

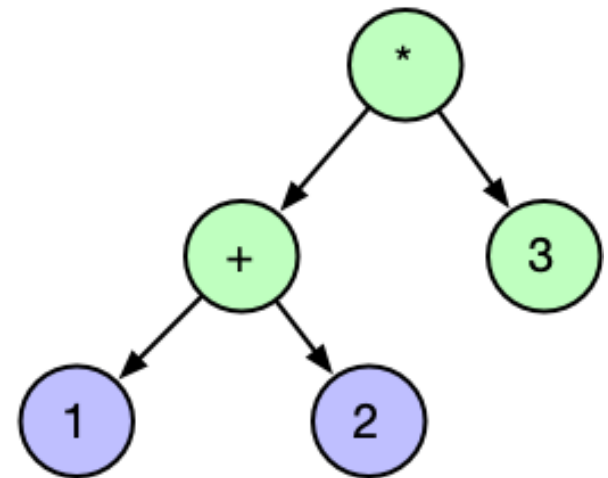
Leftmost AST



= 7

≠

Rightmost AST



= 9

How to Fix an Ambiguous Grammar?

- Usually can eliminate ambiguity by rewriting grammar to include additional rules and allowing recursion only on the right or left

$$\begin{array}{lll} E \rightarrow E + T & T \rightarrow T * F & F \rightarrow \text{INT} \\ E \rightarrow T & T \rightarrow F & \end{array}$$

E for Expression
T for Term
F for Factor

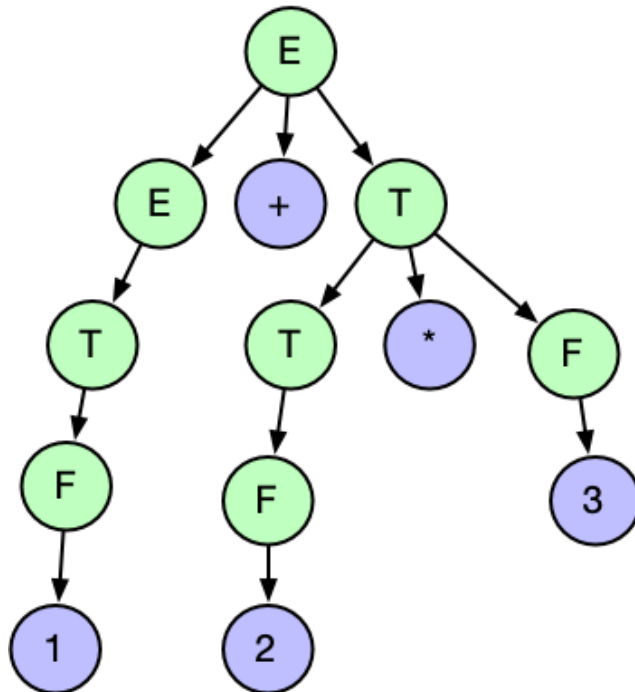
- Make * bind higher than + (i.e., * has higher precedence than +)
 - $1 + 2 * 3$ means $1 + (2 * 3)$ instead of $(1 + 2) * 3$
 - Build grammar from highest to lowest precedence
- Make the grammar use (right or left) recursion. In this case we use left-recursion \rightarrow left-associativity

How to Fix an Ambiguous Grammar?

$E \rightarrow E + T$ $T \rightarrow T * F$ $F \rightarrow \text{INT}$
 $E \rightarrow T$ $T \rightarrow F$

E for Expression
T for Term
F for Factor

Leftmost and Rightmost Parse Tree



At-home exercise:
Write out the derivation
steps for both leftmost
and rightmost to see
how this tree was
produced.

Parsing

- A **parser** is a program that given a sentence constructs a derivation for that sentence
 - If it can construct a derivation then it will **accept** the sentence as part of the language; otherwise error.
 - Parsers read their input from left-to-right but may construct the parse tree differently.
- **Top-down** parsers - construct the tree from root to leaves
 - Algorithms - recursive descent, predictive parsing, LL(1)
- **Bottom-down** parsers - construct the tree from leaves to root
 - Algorithms - shift-reduce, LR, SLR, LALR
 - LR algorithms are the most commonly used parsing algorithm in modern compilers.

Top-Down Parsing

- Construct parse tree by starting at the start symbol and “guess” at derivation step.
 - We can use the next input token to guide in guessing
- We can implement top-down parsing using **recursive descent**; however before we do this we must modify the grammar to be right recursive

Left Recursive Grammar

$E \rightarrow E + T$ $T \rightarrow T * F$ $F \rightarrow \text{INT}$
 $E \rightarrow T$ $T \rightarrow F$

Right Recursive Grammar

$E \rightarrow T E'$ $T \rightarrow F T'$ $F \rightarrow \text{INT}$
 $E' \rightarrow + T E'$ $T' \rightarrow * F T'$
 $E' \rightarrow \epsilon$ $T' \rightarrow \epsilon$

- Most top-down parsing algorithms don't handle left recursion very well.

Demo: Recursive Decent Parser

$$\begin{array}{lll} E \rightarrow T E' & T \rightarrow F T' & F \rightarrow \text{INT} \\ E' \rightarrow + T E' & T' \rightarrow * F T' & \\ E' \rightarrow \varepsilon & T' \rightarrow \varepsilon & \end{array}$$