Parsing

15-411/15-611 Compiler Design

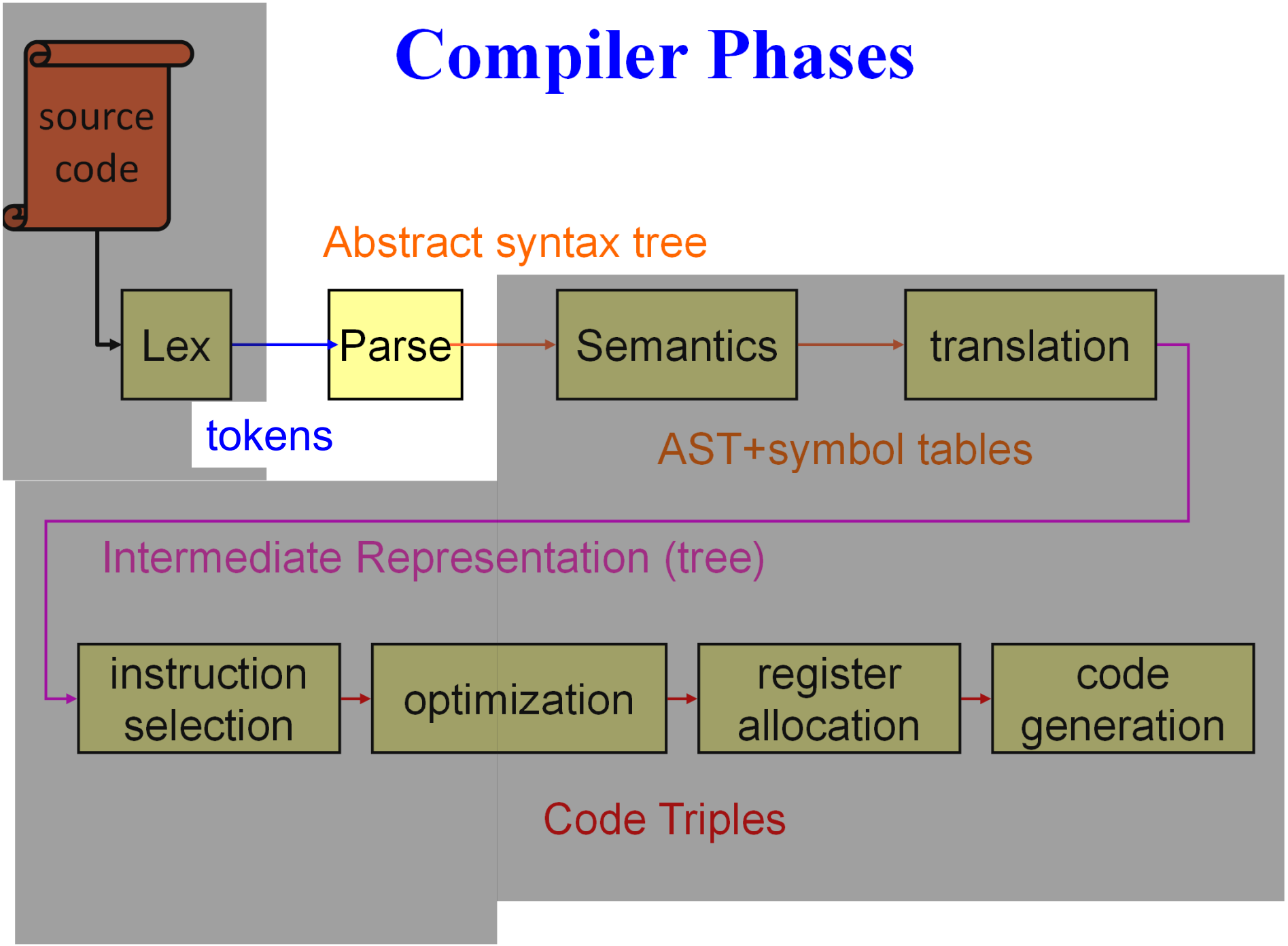
Seth Copen Goldstein

September 17, 2019

Today

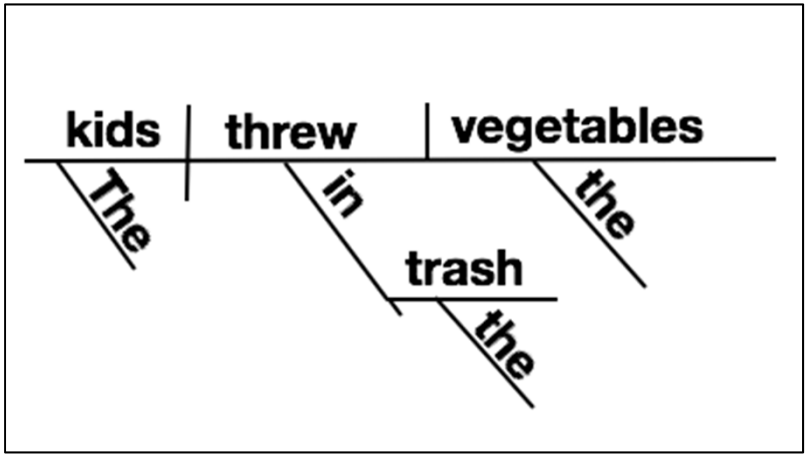
- Languages and Grammars
- Context Free Grammars
- Derivations & Parse Trees
- Ambiguity
- Top-down parsers
- FIRST, FOLLOW, and NULLABLE
- Bottom-up parsers

Compiler Phases



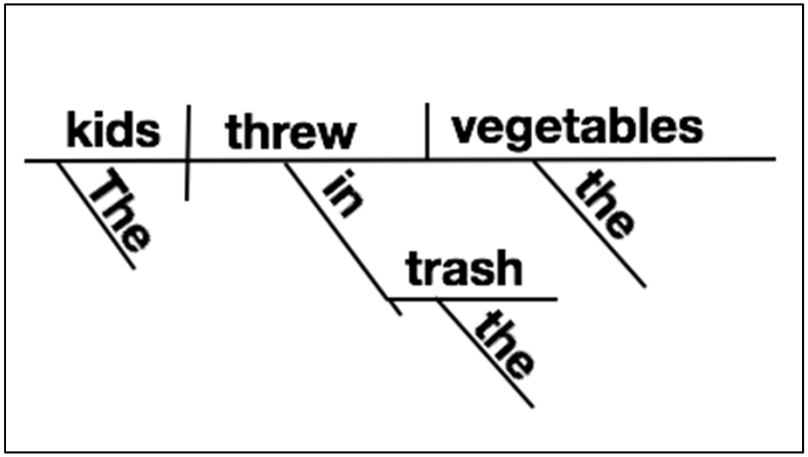
Languages

- Compiler translates from sequence of characters to an executable.
- A series of language transformations
- last week: characters → tokens
- today: tokens → “sentences”



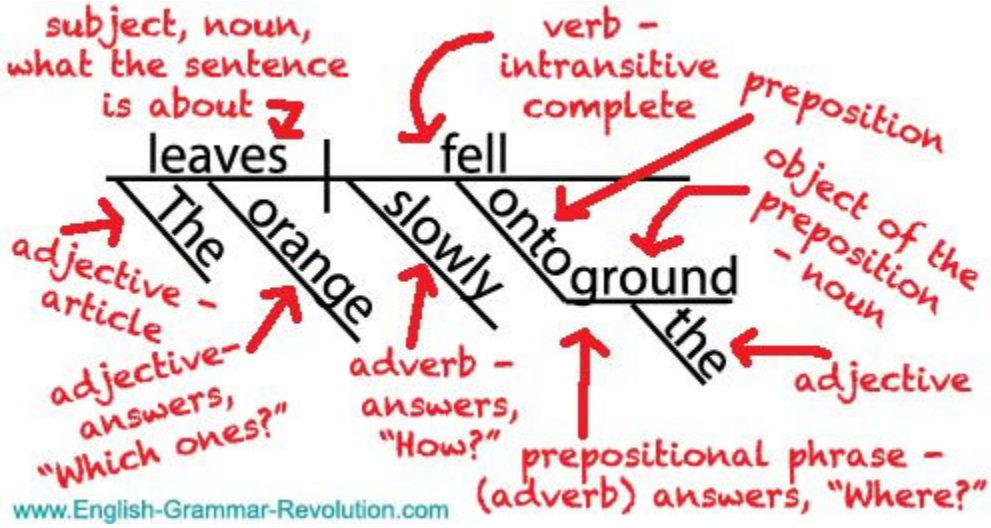
Languages

- Compiler translates from sequence of characters to an executable.
- A series of language transformations
- last week: characters → tokens
- today: tokens → parse trees



Languages

- Compiler translates from sequence of characters to an executable.
- A series of language transformations
- last week: characters → tokens
- today: tokens → “sentences”



Grammers and Languages

- A grammer, G, recognizes a language, L(G)
  - Σ set of terminal symbols
  - A set of non-terminals
  - S the start symbol, a non-terminal
  - P a set of productions
- Usually,
  - $\alpha, \beta, \gamma, \ldots$ strings of terminals and/or non-terminals
  - A, B, C, … are non-terminals
  - a, b, c, … are terminals
- General form of a production is: $\alpha \rightarrow \beta$

Derivation

- A sequence of applying productions starting with S and ending with *w*

$$S \rightarrow \gamma_1 \rightarrow \gamma_2 \ldots \rightarrow \gamma_{n-1} \rightarrow w$$

$$S \rightarrow^* w$$

- *L(G) are all the w that can be derived from S*

→ ε

Regular Grammar (NFA)

- Regular expressions and NFAs can be described by a regular grammar
- E.G.,

  $S \rightarrow aA$

  $A \rightarrow Sb$

  $S \rightarrow \varepsilon$

- An example derivation of aab:

→ ε

Regular Grammar (NFA)

- Regular expressions and NFAs can be described by a regular grammar
- E.G., a*bc*

$$S \rightarrow aS$$
$$S \rightarrow bA$$
$$A \rightarrow \varepsilon$$
$$A \rightarrow cA$$

- An example derivation of aabc:

$$S \rightarrow aS$$

→ ε

Regular Grammar (NFA)

- Regular expressions and NFAs can be described by a regular grammar
- E.G., a*bc*

  S → aS

  S → bA

  A → ε

  A → cA

- An example derivation of aabc:

  S → aS→ aaS

→ ε

Regular Grammar (NFA)

- Regular expressions and NFAs can be described by a regular grammar
- E.G., a*bc*

$$S \rightarrow aS$$
$$S \rightarrow bA$$
$$A \rightarrow \varepsilon$$
$$A \rightarrow cA$$

- An example derivation of aabc:

$$S \rightarrow aS \rightarrow aaS \rightarrow aabA$$

→ ε

Regular Grammar (NFA)

- Regular expressions and NFAs can be described by a regular grammar
- E.G., a*bc*

$$S \rightarrow aS$$
$$S \rightarrow bA$$
$$A \rightarrow \varepsilon$$
$$A \rightarrow cA$$

- An example derivation of aabc:

$$S \rightarrow aS \rightarrow aaS \rightarrow aabA \rightarrow aabcA$$

→ ε

Regular Grammar (NFA)

- Regular expressions and NFAs can be described by a regular grammar
- E.G., a*bc*

  S → aS

  S → bA

  A → ε

  A → cA

- An example derivation of aabc:

  S → aS→ aaS→ aabA→ aabcA → aabc

→ ε

Regular Grammar (NFA)

- Regular expressions and NFAs can be described by a regular grammar
- E.G., a*bc*

$$S \rightarrow aS$$
$$S \rightarrow bA$$
$$A \rightarrow \varepsilon$$
$$A \rightarrow cA$$

- Above is a right-regular grammar
- All rules are of form:

$$A \rightarrow a$$
$$A \rightarrow aB$$
$$A \rightarrow \varepsilon$$

→ ε

Regular Grammar (NFA)

- Regular expressions and NFAs can be described by a regular grammar
- right regular grammar:

  $A \rightarrow a$

  $A \rightarrow aB$

  $A \rightarrow \varepsilon$

- left regular grammar:

  $A \rightarrow a$

  $A \rightarrow Ba$

  $A \rightarrow \varepsilon$

- Regular grammars are either right-regular or left-regular.

Expressiveness

- Restrictions on production rules limit expressiveness of grammars.
- No restrictions allow a grammar to recognize all recursively enumerable languages
- A bit too expressive for our uses ☺
- Regular grammars cannot recognize $a^n b^n$
- We need something more expressive

Chomsky Hierarchy

Class	Language	Automaton	Form	"word" problem	Example
0	Recursively Enumerable	Turing Machine	any	undecidable	Post's Corresp. problem
1	Context Sensitive	Linear-Bounded TM	$\alpha A\beta \rightarrow \alpha\gamma\beta$	PSPACE-complete	$a^n b^n c^n$
2	Context Free	Pushdown Automata	$A \rightarrow \alpha$	cubic	$a^n b^n$
3	Regular	NFA	$A \rightarrow a$ $A \rightarrow aB$	linear	$a^* b^*$

Today

- Languages and Grammars
- Context Free Grammars
- Derivations & Parse Trees
- Ambiguity
- Top-down parsers
- FIRST, FOLLOW, and NULLABLE
- Bottom-up parsers

Context-Free Grammar

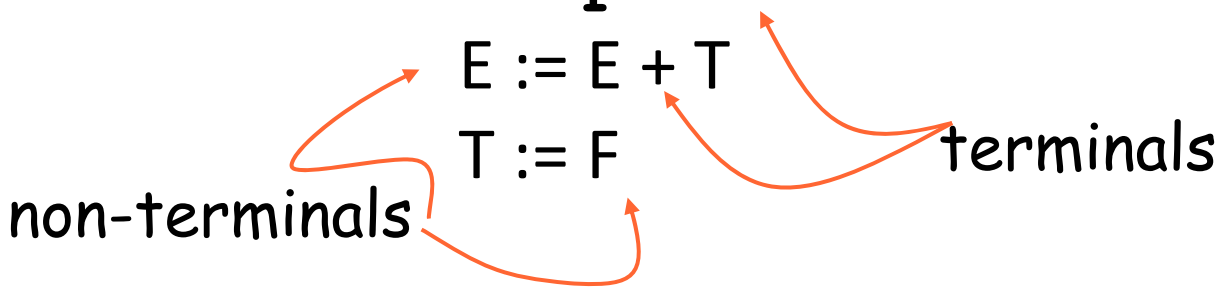
- A context-free grammar, G, is described by:
  - Σ, a set of terminals (which are just the set of possible tokens from the lexer)
    e.g., **`if`**, **`then`**, **`while`**, **`id`**, **`int`**, **`string`**, …
  - A, a set of non-terminals.
    Non-terminals are syntactic variables which define sets of strings in the language
    e.g., stmt, expr, term, factor, vardecl, …
  - S
  - P

Context-Free Grammar

- A context-free grammar, G, is described by:
  - Σ, a set of terminals …
  - A, a set of non-terminals.
  - S, S $\in$ A, the start symbol
    The set of strings derived from S are the valid string in the language.
  - P, set of productions that specify how terminals and non-terminals combine to form strings in the language
    a production, p, has the form: A$\rightarrow \alpha$

Context-Free Grammar

- A context-free grammar, G, is described by:
  - Σ, a set of terminals …
  - A, a set of non-terminals.
  - S, S $\in$ A, the start symbol
  - P, set of productions …
    a production, p, has the form: : A$\rightarrow \alpha$
  - E.g.,: S := E
    S := **print** E
    E := E + T
    T := F



What makes a grammar CF?

- Only one NT on left-hand side $\rightarrow$ context-free
- What makes a grammar context-sensitive?
- $\alpha A\beta \rightarrow \alpha\gamma\beta$ where
  - α or β may be empty,
  - but γ is not-empty
- Are context-sensitive grammars useful for compiler writers?

Matching Parenthesis

$S \rightarrow (S)$

$S \rightarrow SS$

$S \rightarrow \varepsilon$

Simple Grammar of Expressions

S := Exp

Exp := Exp + Exp

Exp := Exp - Exp

Exp := Exp * Exp

Exp := Exp / Exp

Exp := **`id`**

Exp := **`int`**

Describes a language of expressions. e.g.: 2+3*x

Derivations

- A sequence of step in which a non-terminal is replaced by its right-hand side.

1 S := Exp
2 Ex
3 Ex
4 Exp:= Exp * Exp
5 Exp:= Exp / Exp
6 Exp:= **id**
7 Exp:= **int**

There are possibly many derivations determined by the NT chosen to expand.

S

by 3 ⇒ Exp * $\mathbf{id}_x$

by 2 ⇒ Exp + Exp * $\mathbf{id}_x$

by 7 ⇒ $\mathbf{int}_2$ + Exp * $\mathbf{id}_x$

by 7 ⇒ $\mathbf{int}_2$ + $\mathbf{int}_3$ * $\mathbf{id}_x$

do leftmost derivation

Leftmost Derivations

- Leftmost derivation: leftmost NT always chosen

```
1  S   := Exp
2  Exp:= Exp + Exp
3  Exp:= Exp - Exp
4  Exp:= Exp * Exp
5  Exp:= Exp / Exp
6  Exp:= id
7  Exp:= int
```

S

by 1 ⇒ Exp

by 4 ⇒ Exp * Exp

by 2 ⇒ Exp + Exp * Exp

by 7 ⇒ $\texttt{int}_2$ + Exp * Exp

by 7 ⇒ $\texttt{int}_2$ + $\texttt{int}_3$ * Exp

by 6 ⇒ $\texttt{int}_2$ + $\texttt{int}_3$ * $\texttt{id}_x$

Rightmost Derivations

- Rightmost derivation: rightmost NT always chosen

1 S := Exp
2 Exp:= Exp + Exp
3 Exp:= Exp - Exp
4 Exp:= Exp * Exp
5 Exp:= Exp / Exp
6 Exp:= **id**
7 Exp:= **int**

S

by 1 $\Rightarrow$ Exp

by 4 $\Rightarrow$ Exp * Exp

by 6 $\Rightarrow$ Exp * $\mathbf{id}_x$

by 2 $\Rightarrow$ Exp + Exp * $\mathbf{id}_x$

by 7 $\Rightarrow$ Exp + $\mathbf{int}_3$ * $\mathbf{id}_x$

by 7 $\Rightarrow$ $\mathbf{int}_2$ + $\mathbf{int}_3$ * $\mathbf{id}_x$

Parse Trees

- symbols in rhs are children of NT being rewritten

S

by 1 ⇒ Exp

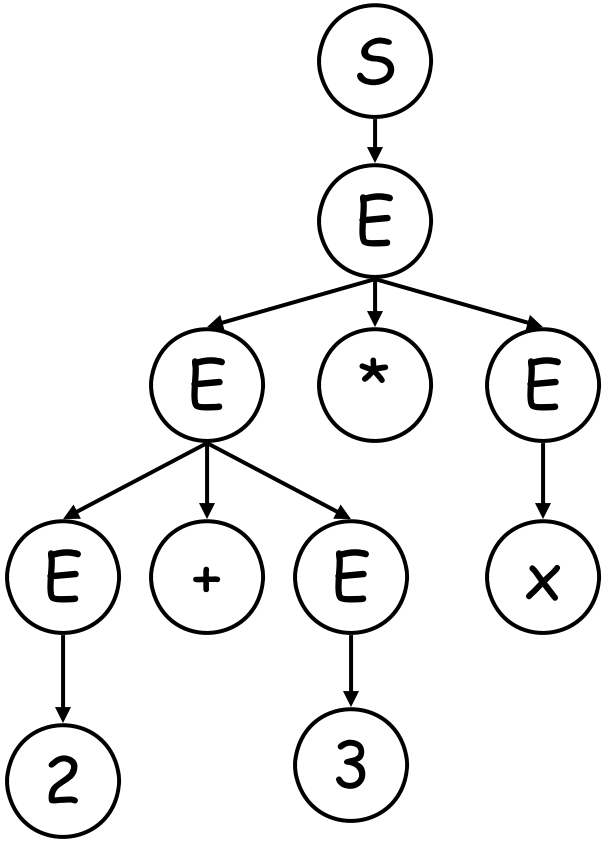
by 4 ⇒ Exp * Exp

by 2 ⇒ Exp + Exp * Exp

by 7 ⇒ $\texttt{int}_2$ + Exp * Exp

by 7 ⇒ $\texttt{int}_2$ + $\texttt{int}_3$ * Exp

by 6 ⇒ $\texttt{int}_2$ + $\texttt{int}_3$ * $\texttt{id}_x$



Parse Trees

- parse tree for rightmost derivation

S

by 1 ⇒ Exp

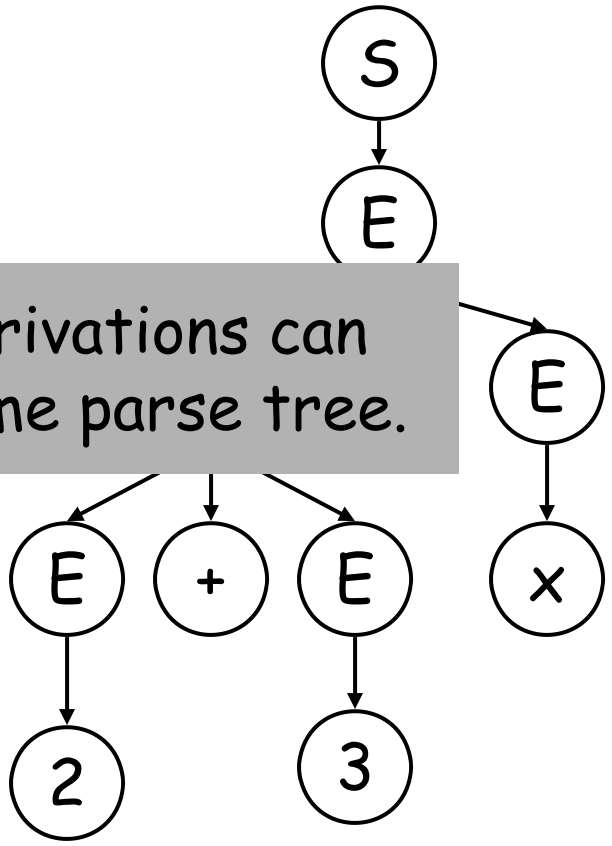
by 4 ⇒ Exp *

by 6 ⇒ Exp *

by 2 ⇒ Exp + Exp * $\mathbf{id}_x$

by 7 ⇒ Exp + $\mathbf{int}_3$ * $\mathbf{id}_x$

by 7 ⇒ $\mathbf{int}_2$ + $\mathbf{int}_3$ * $\mathbf{id}_x$



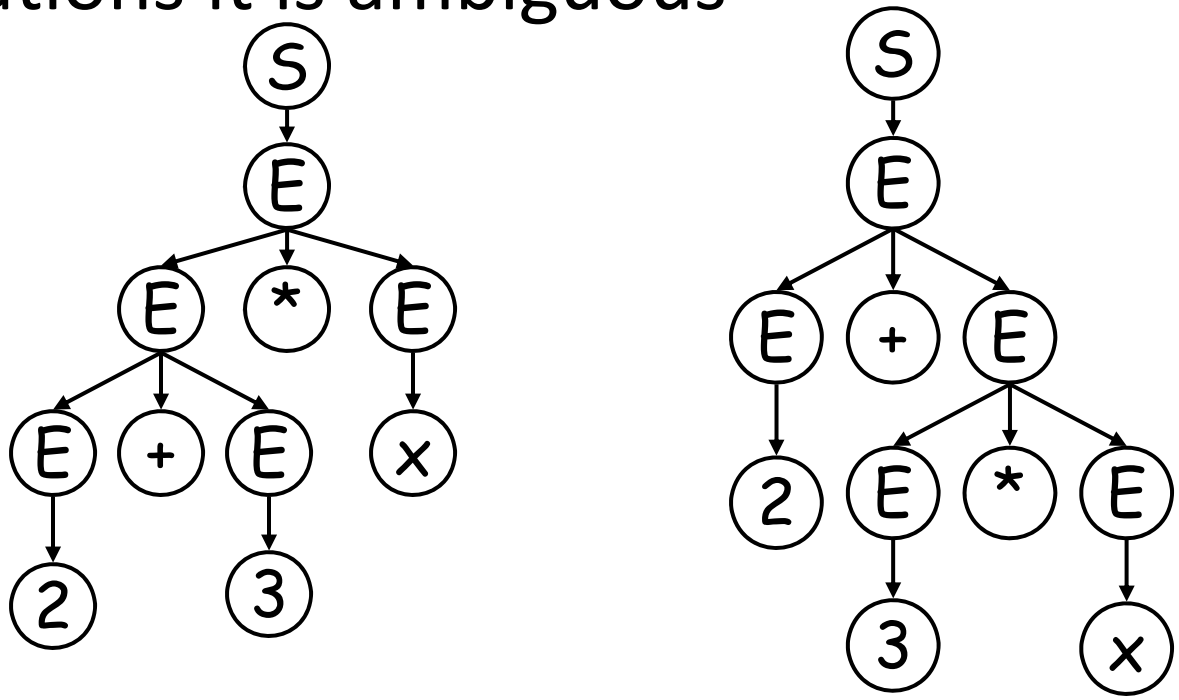
Different derivations can lead to the same parse tree.

What about different parse trees for same sentence?

Ambiguous Grammars

- A gra … a sentence with >1 parse trees. or,
- If grammer has >1 leftmost (rightmost) derivations it is ambiguous

What does ambiguity point out?



Converting Expression Grammar

- Adding precedence with more non-terminals
- One for each level of precedence:
  - (+, -) exp
  - (*, /) term
  - (**`id`**, **`int`**) factor
  - Make sure parse derives sentences that respect the precedence
  - Make sure that extra levels of precedence can be bypassed, i.e., “x” is still legal

A Better Exp Grammar

1 S := Exp
2 Exp := Exp + Term
3 Exp := Exp - Term
4 Exp := Term
5 Term := Term * Factor
6 Term := Term / Factor
7 Term := Factor
8 Factor := **id**
9 Factor := **int**

S
by 1 ⇒ Exp
by 2 ⇒ Exp + Term
by 4 ⇒ Term + Term
by 7 ⇒ Factor + Term
by 9 ⇒ $\mathbf{int}_2$ + Term
by 5 ⇒ $\mathbf{int}_2$ + Term * Factor
by 7 ⇒ $\mathbf{int}_2$ + Factor * Factor
by 9 ⇒ $\mathbf{int}_2$ + $\mathbf{int}_3$ * Factor
by 8 ⇒ $\mathbf{int}_2$ + $\mathbf{int}_3$ * $\mathbf{id}_x$

What is the parse tree?

Another Ambiguous Grammer

```
S := if E then S
   | if E then S else S
   | other
```

- What is the parse tree for:
  if E **then if** E **then** S **else** S?
- What is the language designers intention?
- Is there a context-free solution?

Dangling Else Grammar

S := matchedS

| unmatchedS

unmatchedS := **`if`** E **`then`** S

| if E then matchedS else unmatchedS

matchedS := **`if`** E **`then`** matchedS **`else`** matchedS

Later we will see another way to do this.

- Is this clearer?
- What is parse tree for: **`if`** E **`then if`** E **`then`** S **`else`** S?

A primitive robot

Swing := Back Swing Forward

|

Back := back-1-inch

Forward := forward-2-inchs

- What is L(Swing)?

A primitive robot

S := B S F
|
B := b
F := f

- What is L(Swing)?
- What is the parse tree for “bbff”

Parsing a CFG

- Top-Down
  - start at root of parse-tree
  - pick a production and expand to match input
  - may require backtracking
  - if no backtracking required, predictive
- Bottom-up
  - start at leaves of tree
  - recognize valid prefixes of productions
  - consume input and change state to match
  - use stack to track state

Top-down Parsers

- Starts at root of parse tree and recursively expands children that match the input
- In general case, may require backtracking
- Such a parser uses recursive descent.
- When a grammar does not require backtracking a predictive parser can be built.

A Predictive Parser

```
S  :=  B S F
   |
B  :=  b
F  :=  f
```

Idea is for parser to do something besides recognize legal sentences.

```
S() {
        if match('b') -> B(); S(); F(); action();
        else return;
}
B() {   mustMatch('b');  action(); return;}
F() {   mustMatch('f');  action();  return;}
```

Top-Down parsing

- Start with root of tree, i.e., S
- Repeat until entire input matched:
  - pick a non-terminal, A, and pick a production $A \rightarrow \gamma$ that can match input, and expand tree
  - if no such rule applies, backtrack
- Key is obviously selecting the right production

Top-down for Exp Grammar

1	S := E
2	E := E + T
3	E := E - T
4	E := T
5	T := T * F
6	T := T / F
7	T := F
8	F := id
9	F := int

	S	$\texttt{int}_2 - \texttt{int}_3 * \texttt{id}_x$
by 1 ⇒	E	$\texttt{int}_2 - \texttt{int}_3 * \texttt{id}_x$

Top-down for Exp Grammar

1	S := E
2	E := E + T
3	E := E - T
4	E := T
5	T := T * F
6	T := T / F
7	T := F
8	F := **id**
9	F := **int**

S	**int**$_2$ - **int**$_3$ * **id**$_x$
by 1 ⇒ E	**int**$_2$ - **int**$_3$ * **id**$_x$
by 2 ⇒ E + T	**int**$_2$ - **int**$_3$ * **id**$_x$
by 4 ⇒ T + T	**int**$_2$ - **int**$_3$ * **id**$_x$
by 7 ⇒ F + T	**int**$_2$ - **int**$_3$ * **id**$_x$
by 9 ⇒ **int**$_2$ + T	**int**$_2$ - **int**$_3$ * **id**$_x$

Must backtrack here!

Top-down for Exp Grammar

1	S := E
2	E := E + T
3	E := E - T
4	E := T
5	T := T * F
6	T := T / F
7	T := F
8	F := id
9	F := int

S	int_2 - int_3 * id_x
by 1 ⇒ E	int_2 - int_3 * id_x
by 2 ⇒ E + T	int_2 - int_3 * id_x
by 4 ⇒ T + T	int_2 - int_3 * id_x
by 7 ⇒ F + T	int_2 - int_3 * id_x
by 9 ⇒ int_2 + T	int_2 - int_3 * id_x
by 3 ⇒ E - T	int_2 - int_3 * id_x
by 4 ⇒ T - T	int_2 - int_3 * id_x
by 7 ⇒ F - T	int_2 - int_3 * id_x
by 9 ⇒ int_2 - T	int_2 - int_3 * id_x
by 5 ⇒ int_2 - T * F	int_2 - int_3 * id_x

Top-down for Exp Grammar

1	S := E
2	E := E + T
3	E := E - T
4	E := T
5	T := T * F
6	T := T / F
7	T := F
8	F := id
9	F := int

	S	int$_2$ - int$_3$ * id$_x$
by 1 ⇒	E	int$_2$ - int$_3$ * id$_x$
by 2 ⇒	E + T	int$_2$ - int$_3$ * id$_x$
by 4 ⇒	T + T	int$_2$ - int$_3$ * id$_x$
by 7 ⇒	F + T	int$_2$ - int$_3$ * id$_x$
by 9 ⇒	int$_2$ + T	int$_2$ - int$_3$ * id$_x$
by 3 ⇒	E - T	int$_2$ - int$_3$ * id$_x$
by 4 ⇒	T - T	int$_2$ - int$_3$ * id$_x$
by 7 ⇒	F - T	int$_2$ - int$_3$ * id$_x$
by 9 ⇒	int$_2$ - T	int$_2$ - int$_3$ * id$_x$
		nt$_2$ - int$_3$ * id$_x$

What kind of derivation is this parsing?

Top-down for Exp Grammar

1	S := E
2	E := E + T
3	E := E - T
4	E := T
5	T := T * F
6	T := T / F
7	T := F
8	F := **id**
9	F := **int**

	S	**int**$_2$ - **int**$_3$ * **id**$_x$
by 1 ⇒	E	**int**$_2$ - **int**$_3$ * **id**$_x$
by 2 ⇒	E + T	**int**$_2$ - **int**$_3$ * **id**$_x$
by 2 ⇒	E + E + T	**int**$_2$ - **int**$_3$ * **id**$_x$
by 2 ⇒	E + E + E + T	**int**$_2$ - **int**$_3$ * **id**$_x$

Will not terminate! Why?

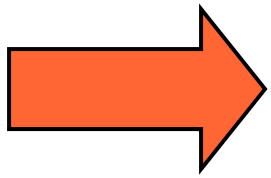
grammar is left-recursive

What should we do about it?

Eliminate left-recursion

Does this work?

1	S := E
2	E := E + T
3	E := E - T
4	E := T
5	T := T * F
6	T := T / F
7	T := F
8	F := **id**
9	F := **int**



1	S := E
2	E := T + E
3	E := T - E
4	E := T
5	T := F * T
6	T := F / T
7	T := F
8	F := **id**
9	F := **int**

It is right recursive, but also right associative!

Eliminating Left-Recursion

- Given 2 productions:

  $A := A\ \alpha \mid \beta$

  Where neither α nor β start with A

  (e.g., For example, E := E + T | T)

  α β

- Make it right-recursive:

$$A := \beta\ R$$
$$R := \alpha\ R \mid$$

R is right recursive

- Extends to general case.

Rewriting Exp Grammar

```
1   S   := E
2   E   := E + T
3   E   := E - T
4   E   := T
5   T   := T * F
6   T   := T / F
7   T   := F
8   F   := id
9   F   := int
```

```
1   S  := E
2'  E' := + T E'
3'  E' := - T E'
4'  E' :=
5'  T' := * F T'
6'  T' := / F T'
7'  T' :=
8   F  := id
9   F  := int
```

Is this legible?

```
2   E := T E'

5   T := F T'
```

Try again

1	S := E
2	E := T E′
2′	E′ := + T E′
3′	E′ := - T E′
4′	E′ :=
5	T := F T′
5′	T′ := * F T′
6′	T′ := / F T′
7′	T′ :=
8	F := id
9	F := int

S	●int$_2$ - int$_3$ * id$_x$
by 1 ⇒ E	●int$_2$ - int$_3$ * id$_x$
by 2 ⇒ T E′	●int$_2$ - int$_3$ * id$_x$
by 5 ⇒ F T′ E′	●int$_2$ - int$_3$ * id$_x$
by 9 ⇒ 2 T′ E′	int$_2$ ●- int$_3$ * id$_x$
by 7′ ⇒ 2 E′	int$_2$ ●- int$_3$ * id$_x$
by 3′ ⇒ 2 - T E′	int$_2$ - ●int$_3$ * id$_x$
by 5 ⇒ 2 - F T′ E′	int$_2$ - ●int$_3$ * id$_x$
by 9 ⇒ 2 - 3 T′ E′	int$_2$ - int$_3$ ●* id$_x$
by 5′ ⇒ 2 - 3 * F T′ E′	int$_2$ - int$_3$ * ●id$_x$
	int$_3$ * id$_x$●
	int$_3$ * id$_x$●
	int$_3$ * id$_x$●

Unlike previous time we tried this, it appears that only one production applies at a time. I.e., no backtracking needed. Why?

Lookahead

- How to pick right production?
- Lookahead in input stream for guidance
- General case: arbitrary lookahead required
- Luckily, many context-free grammers can be parsed with limited lookahead
- If we have $A \rightarrow \alpha \mid \beta$, then we want to correctly choose either $A \rightarrow \alpha$ or $A \rightarrow \beta$
- define FIRST(α) as the set of tokens that can be first symbol of α, i.e.,

  $a \in \text{FIRST}(\alpha)$ iff $\alpha \rightarrow^* a\gamma$ for some γ

Lookahead

- How to pick right production?
- If we have $A \rightarrow \alpha \mid \beta$, then we want to correctly choose either $A \rightarrow \alpha$ or $A \rightarrow \beta$
- define FIRST(α) as the set of tokens that can be first symbol of α, i.e.,

  $a \in$ FIRST(α) iff $\alpha \rightarrow^* a\gamma$ for some γ
- If $A \rightarrow \alpha \mid \beta$ we want:

  FIRST(α) $\cap$ FIRST(β) = $\varnothing$
- If that is always true, we can build a predictive parser.

FIRST sets

- We use next k characters in input stream to guide the selection of the proper production.
- Given: A := α | β we want next input character to decide between α and β.
- FIRST(α) = set of terminals that can begin any string derived from α.
- IOW: **a** $\in$ FIRST(α) iff $\alpha \Rightarrow^*$ **a**γ for some γ

- FIRST(α) $\cap$ FIRST(β) = $\varnothing \rightarrow$ no backtracking needed

Computing FIRST(α)

- Given X := A B C, FIRST(X) = FIRST(A B C)
- Can we ignore B or C?
- Consider:

```
A := a
   |
B := b
   | A
C := c
```

Computing FIRST(α)

- Given X := A B C, FIRST(X) = FIRST(A B C)
- Can we ignore B or C?
- Consider:

```
A := a
   |
B := b
   | A
C := c
```

- FIRST(X) must also include FIRST(C)
- IOW:
  - Must keep track of NTs that are nullable
  - For nullable NTs, determine FOLLOWS(NT)

nullable(A)

- nullable(A) is true if A can derive the empty string
- For example:

```
B := X Y b
X := x
   |  Y Y
Y :=
```

In this case, nullable(X) = nullable(Y) = true

nullable(B) = false

FOLLOW(A)

- FOLLOW(A) is the set of terminals that can immediately follow A in a sentential form.
- I.e.,
  a $\in$ FOLLOW(A) iff S $\Rightarrow^*$ αAaβ for some α and β

Building a Predictive Parser

- We want to know for each non-terminal which production to choose based on the next input character.
- Build a table with rows labeled by non-terminals, A, and columns labeled by terminals, a. We will put the production, A := α , in (A, a) iff
  - FIRST(α) contains a or
  - nullable(α) and FOLLOW(A) contains a

The table for the robot

S := B S F

|

B := b

F := f

	FIRST	FOLLOW	nullable
S	b	$	yes
B	b	b,f	no
F	f	f,$	no

	b	f	$
S			
B			
F			

The table for the robot

S := B S F

|

B := b

F

	FIRST	FOLLOW	nullable
S	b	$	yes
B	b	b,f	no
F	f	f,$	no

FIRST(BSF) = b

nullable(ε)=true and FOLLOW(S) = $

	b	f	$
S	S:=BSF		S:=
B	B:=b		
F		F:=f	

Table f

	FIRST	FOLLOW	nullable
S	id, int	$	
E	id, int	$	
E'	+, -	$	yes
T	id, int	+,-,$	
T'	/, *	+,-,$	yes
F	id, int	/, *,$	

```
1    S := E
2    E := T E'
2'   E' := + T E'
3'   E' := - T E'
4'   E' :=
5    T := F T'
5'   T' := * F T'
6'   T' := / F T'
7'   T' :=
8    F := id
9    F := int
```

	+	-	*	/	id	int	$
S							
E							
E'							
T							
T'							
F							

Table f

	FIRST	FOLLOW	nullable
S	id, int	$	
E	id, int	$	
E'	+, -	$	yes
T	id, int	+,-,$	
T'	/, *	+,-,$	yes
F	id, int	/, *,$	

1	S := E
2	E := T E'
2'	E' := + T E'
3'	E' := - T E'
4'	E' :=
5	T := F T'
5'	T' := * F T'
6'	T' := / F T'
7'	T' :=
8	F := **id**
9	F := **int**

	+	-	*	/	id	int	$
S					:=E	:=E	
E					:=TE'	:=TE'	
E'	:=+TE'	:=-TE'					:=
T					:=FT'	:=FT'	
T'	:=	:=	:=*FT'	:=/FT'			:=
F					:=id	:=int	

Using the Table

- Each row in the table becomes a function
- For each input token with an entry: Create a series of invocations that implement the production, where
  - a non-terminal is eaten
  - a terminal becomes a recursive call
- For the blank cells implement errors

Example function

	+	-	*	/	id	int	$
S					:=E	:=E	
E					:=TE'	:=TE'	
E'	:=+TE'	:=-TE'			:=TE'	:=TE'	:=
T							
T'	:=	:=	:=*FT				
F					:=id	:=int	

How to handle errors?

```
Eprime() {
   switch (token) {
   case PLUS:     eat(PLUS); T(); Eprime(); break;
   case MINUS:    eat(MINUS); T(); Eprime(); break;
   case ID:       T(); Eprime();
   case INT:      T(); Eprime();
   default:       error();
  }
```

Left-Factoring

- Predictive parsers need to make a choice based on the next terminal.
- Consider:

```
S := if E then S else S
   | if E then S
```

- When looking at **`if`**, can't decide
- so left-factor the grammar

```
S := if E then S X
X := else S
   |
```

Top-Down Parsing

- Can be constructed by hand
- LL(k) grammars can be parsed
  - Left-to-right
  - Leftmost-derivation
  - with k symbols lookahead
- Often requires
  - left-factoring
  - Elimination of left-recursion

Bottom-up parsers

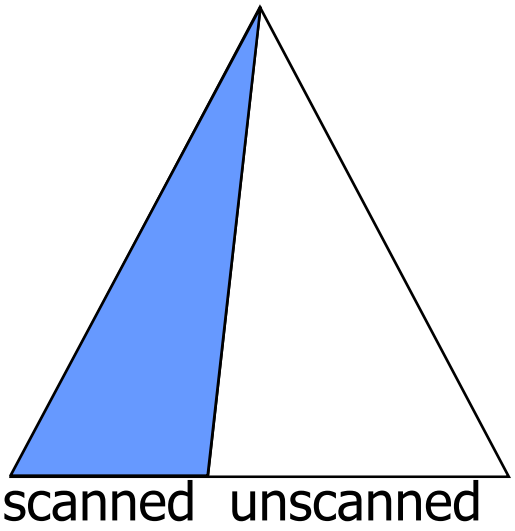
- What is the inherent restriction of top-down parsing, e.g., with LL(k) grammars?

Bottom-up parsers

- What is the inherent restriction of top-down parsing, e.g., with LL(k) grammars?
- Bottom-up parsers use the entire right-hand side of the production
- LR(k):
  - Left-to-right parse,
  - Rightmost derivation (in reverse),
  - k look ahead tokens

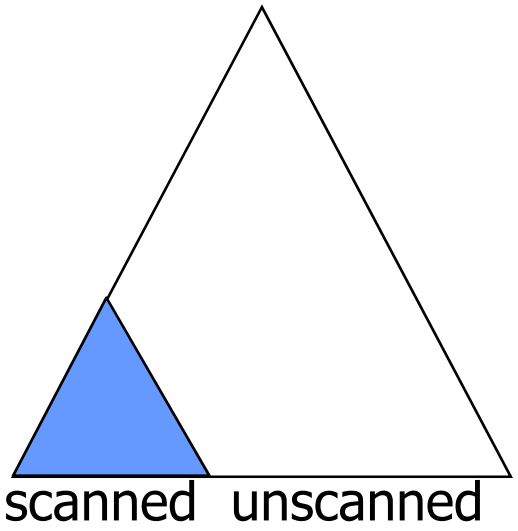
Top-down vs. Bottom-up

LL(k), recursive descent



Top-down

LR(k), shift-reduce



Bottom-up

Example - Top-down

```
S := X
X := X a
   | b
```

Is this grammar LL(k)?

How can we make it LL(k)?

```
S := X
X := b R
R := a R
   |
```

What about a bottom up parse?

Example - Bottom-up

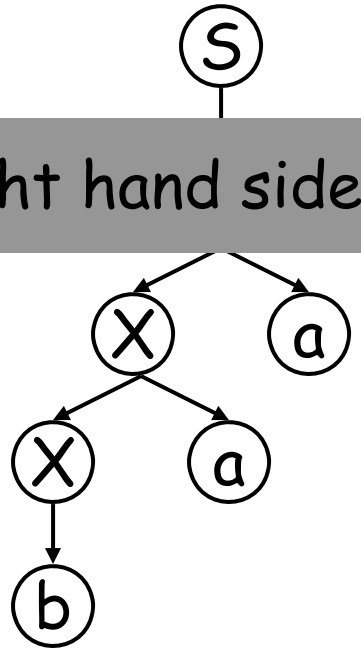
S := X
X := X a
| b

right-most derivation:

LR parser gets to look at an entire right hand side.

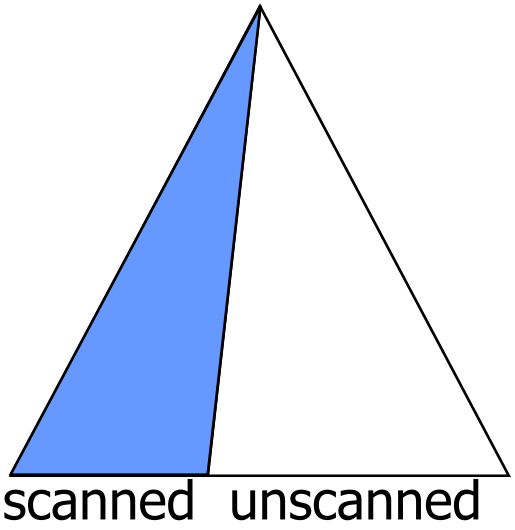
Left-to-Right, Rightmost in reverse

baa
Xaa
Xa
X
S



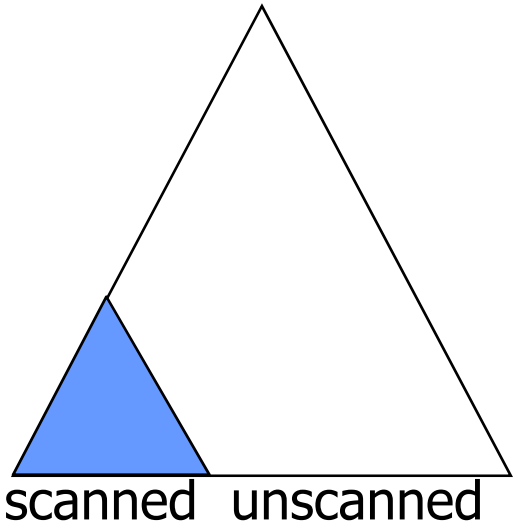
Top-down vs. Bottom-up

LL(k), recursive descent



Top-down

LR(k), shift-reduce



Bottom-up