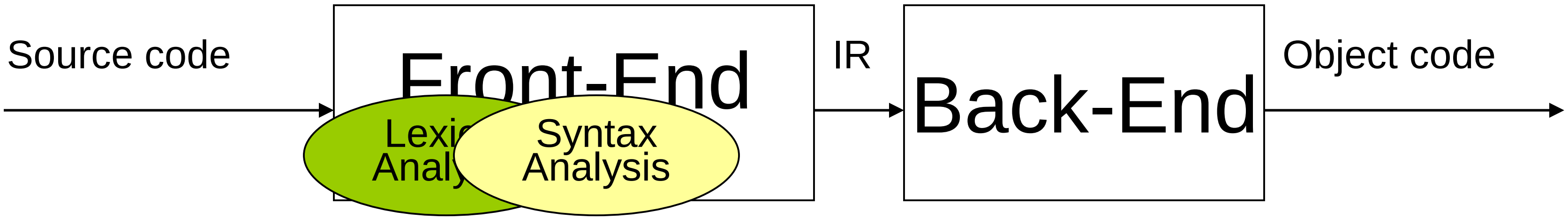
Compiler Design

Lecture 4:
Syntax Analysis

Dr. Momtazi
momtazi@aut.ac.ir

based on the slides of the course book

Introduction to Parsing (Syntax Analysis)



- Lexical analyzer reads characters of the input program and produces tokens.
  - Are they syntactically correct?
  - Are they valid sentences of the input language?

Outline

- **Context-free Grammars**
- Derivations
- Parse trees
- Ambiguity

- Next Lectures on Parsing:
  - Top-Down Parsing
  - Bottom-Up Parsing
  - Context Sensitive Analysis

Regular Expressions Limitation

- Not all languages can be described by Regular Expressions!! (Lecture 3)

- The descriptive power of regular expressions has limits:
  - REs cannot describe balanced or nested constructs:
    - E.g., set of all strings of balanced parentheses {(), (()), ((())), …}
  - REs cannot describe the set of all 0s followed by an equal number of 1s
    - E.g., {01, 0011, 000111, ...}

Regular Expressions Limitation

- Chomsky's hierarchy of Grammars:
  - 1. Phrase structured.
  - 2. Context Sensitive
    - number of Left Hand Side Symbols ≤ number of Right Hand Side Symbols
  - 3. Context-Free
    - The Left Hand Side Symbol is a non-terminal
  - 4. Regular
    - Only rules of the form: $A \to \varepsilon$, $A \to \alpha B$, $A \to B\alpha$

Regular Languages $\subset$ Context-Free Ls $\subset$ Cont.Sens.Ls $\subset$ Phr.Str.Ls

Expressing Syntax

- Context-free syntax is specified with a context-free grammar.

Recall (Lecture 3, slide 43):

- A context-free grammar, G, is a 4-tuple, $G=(S,N,T,P)$, where:

  S: starting symbol

  N: set of non-terminal symbols

  T: set of terminal symbols

  P: set of production rules

Expressing Syntax

- Example:

  Integer → *Integer digit* rule 1

  | *digit* rule 2

  - We can use the Integer grammar to create numbers; e.g.:

Rule	Sentential Form
-	*Integer*
1	*Integer digit*
2	*digit digit …*

  - Such a sequence of rewrites is called a derivation

*The process of discovering a derivation for some sentence is called **parsing**!*

Parsing

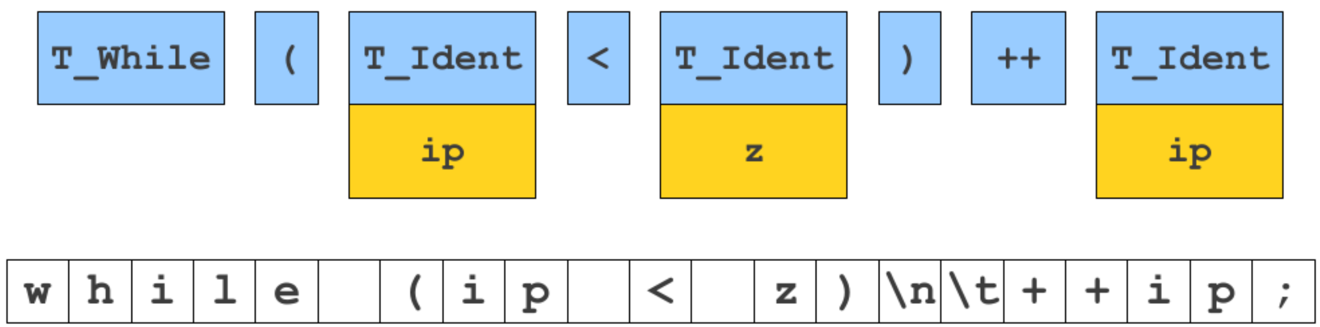
```
while (ip < z)
    ++ip;
```

Parsing

w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

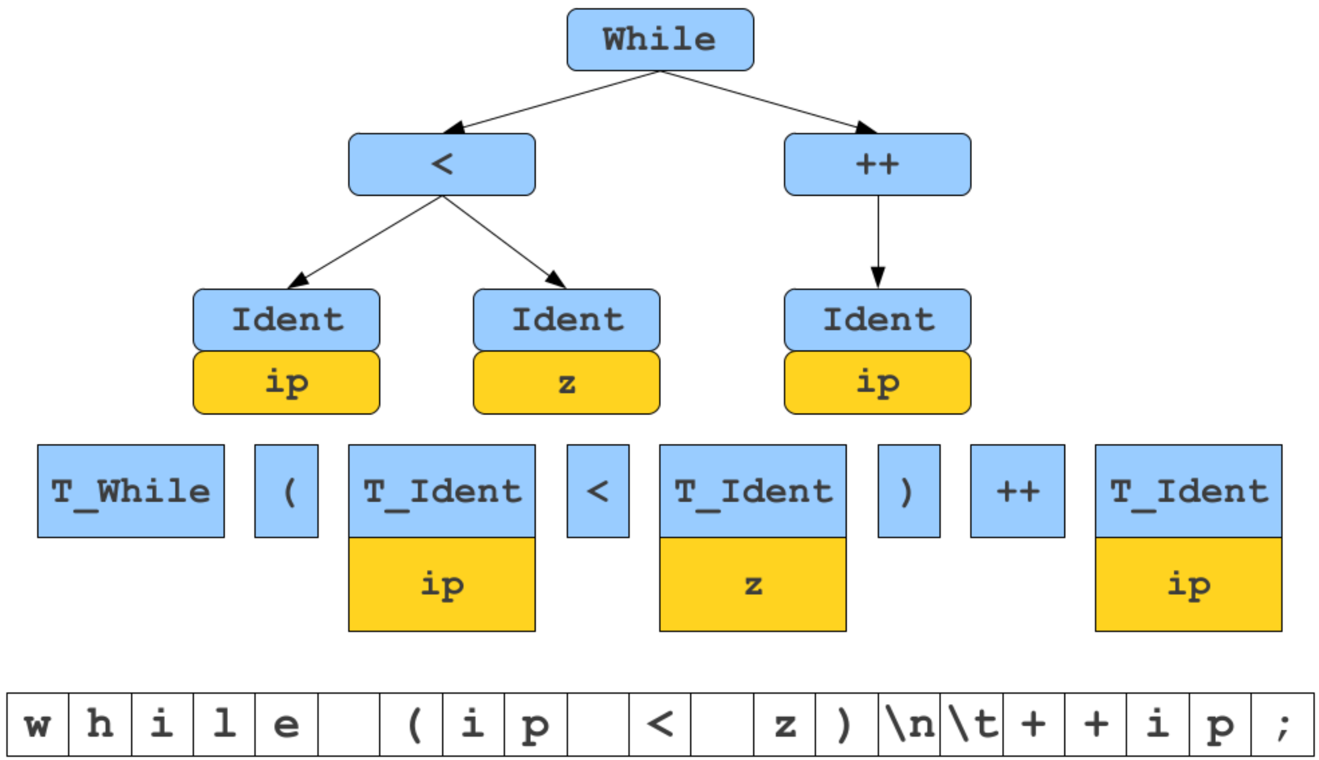
```
while (ip < z)
    ++ip;
```

Parsing




```
while (ip < z)
  ++ip;
```

Parsing




```
while (ip < z)
    ++ip;
```

Parsing

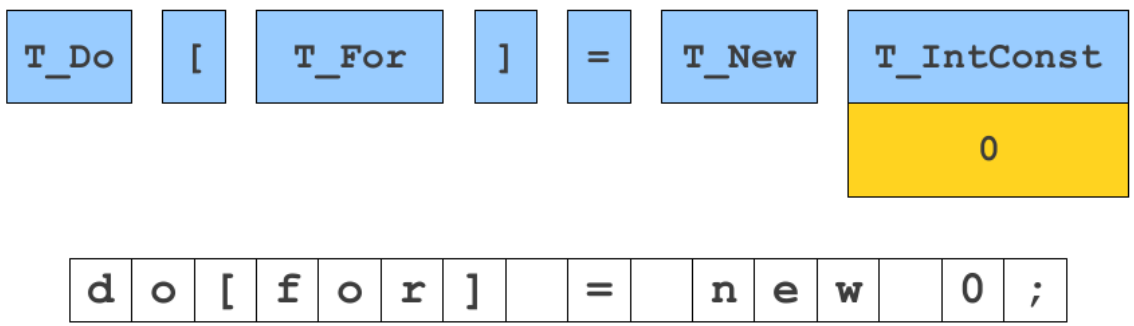
```
do[for] = new 0;
```

Parsing

d	o	[	f	o	r	]		=		n	e	w		0	;

```
do[for] = new 0;
```

Parsing




```
do[for] = new 0;
```

Parsing

T_Do	[	T_For	]	=	T_New	T_IntConst
						0

d	o	[	f	o	r	]		=		n	e	w		0	;

`do[for] = new 0;`

Grammar: Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

$$
\begin{aligned}
E &\rightarrow \mathtt{int} \\
E &\rightarrow E\ Op\ E \\
E &\rightarrow (E) \\
Op &\rightarrow + \\
Op &\rightarrow - \\
Op &\rightarrow * \\
Op &\rightarrow /
\end{aligned}
$$

Grammar: Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

$$
\begin{aligned}
E &\rightarrow \texttt{int} \\
E &\rightarrow E\ Op\ E \\
E &\rightarrow \texttt{(}E\texttt{)} \\
Op &\rightarrow \texttt{+} \\
Op &\rightarrow \texttt{-} \\
Op &\rightarrow \texttt{*} \\
Op &\rightarrow \texttt{/}
\end{aligned}
$$

$$
\begin{aligned}
& E \\
\Rightarrow\ & E\ Op\ E \\
\Rightarrow\ & E\ Op\ \texttt{(}E\texttt{)} \\
\Rightarrow\ & E\ Op\ \texttt{(}E\ Op\ E\texttt{)} \\
\Rightarrow\ & E\ \texttt{*}\ \texttt{(}E\ Op\ E\texttt{)} \\
\Rightarrow\ & \texttt{int * (}E\ Op\ E\texttt{)} \\
\Rightarrow\ & \texttt{int * (int}\ Op\ E\texttt{)} \\
\Rightarrow\ & \texttt{int * (int}\ Op\ \texttt{int)} \\
\Rightarrow\ & \texttt{int * (int + int)}
\end{aligned}
$$

Grammar: Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

E → `int`
E → E Op E
E → `(`E`)`
Op → `+`
Op → `-`
Op → `*`
Op → `/`

E
⇒ E Op E
⇒ E Op `int`
⇒ `int` Op `int`
⇒ `int / int`

Grammar: Arithmetic Expressions

- A notational shorthand:

$$\mathbf{E} \rightarrow \texttt{int} \mid \mathbf{E}\ \mathbf{Op}\ \mathbf{E} \mid \texttt{(}\mathbf{E}\texttt{)}$$

$$\mathbf{Op} \rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}$$

$$\mathbf{E} \rightarrow \texttt{int}$$

$$\mathbf{E} \rightarrow \mathbf{E}\ \mathbf{Op}\ \mathbf{E}$$

$$\mathbf{E} \rightarrow \texttt{(}\mathbf{E}\texttt{)}$$

$$\mathbf{Op} \rightarrow \texttt{+}$$

$$\mathbf{Op} \rightarrow \texttt{-}$$

$$\mathbf{Op} \rightarrow \texttt{*}$$

$$\mathbf{Op} \rightarrow \texttt{/}$$

Grammar: Chemicals

$C_{19}H_{14}O_5S$

$Cu_3(CO_3)_2(OH)_2$

MnO_4^-

S^{2-}

Form → Cmp | Cmp Ion

Cmp → Term | Term Num | Cmp Cmp

Term → Elem | (Cmp)

Elem → H | He | Li | Be | B | C | …

Ion → + | - | IonNum + | IonNum -

IonNum → 2 | 3 | 4 | ...

Num → 1 | IonNum

Grammar: Chemicals

Form → Cmp | Cmp Ion

Cmp → Term | Term Num | Cmp Cmp

Term → Elem | (Cmp)

Elem → H | He | Li | Be | B | C | …

Ion → + | - | IonNum + | IonNum -

IonNum → 2 | 3 | 4 | …

Num → 1 | IonNum

Form
⇒ Cmp Ion
⇒ Cmp Cmp Ion
⇒ Cmp Term Num Ion
⇒ Term Term Num Ion
⇒ Elem Term Num Ion
⇒ Mn Term Num Ion
⇒ Mn Elem Num Ion
⇒ MnO Num Ion
⇒ MnO IonNum Ion
⇒ MnO_4 Ion
⇒ MnO_4^-

Grammar: Programming Languages

```
BLOCK   →  STMT
        |  { STMTS }

STMTS   →  ε
        |  STMT STMTS

STMT    →  EXPR;
        |  if (EXPR) BLOCK
        |  while (EXPR) BLOCK
        |  do BLOCK while (EXPR);
        |  BLOCK
        |  ...

EXPR    →  identifier
        |  constant
        |  EXPR + EXPR
        |  EXPR – EXPR
        |  EXPR * EXPR
        |  ...
```

Outline

- Context-free Grammars
- **Derivations**
- **Parse trees**
- Ambiguity

Derivations and Parse Trees

- Productions are treated as rewriting rules to generate a string

- Derivation steps:
  - At each step, we choose a non-terminal to replace
  - Different choices can lead to different derivations

Derivations and Parse Trees

- Two derivations are of interest:
  - Leftmost derivation:
    - At each step, replace the leftmost non-terminal
  - Rightmost derivation
    - At each step, replace the rightmost non-terminal

(we don't care about randomly-ordered derivations!)

Derivations and Parse Trees

- A parse tree is a graphical representation for a derivation

- Construction:
  - Start with the starting symbol (root of the tree)
  - For each sentential form:
    - Add children to the node corresponding to the left-hand-side symbol.
  - The leaves of the tree (read from left to right) constitute a sentential form

Example: Leftmost, Rightmost Derivation

E **-->** E + E

| E * E

| **-**E

| (E)

| **id**

Derivations for **–(id+id)**

E => **-**E

=> **-**(E)

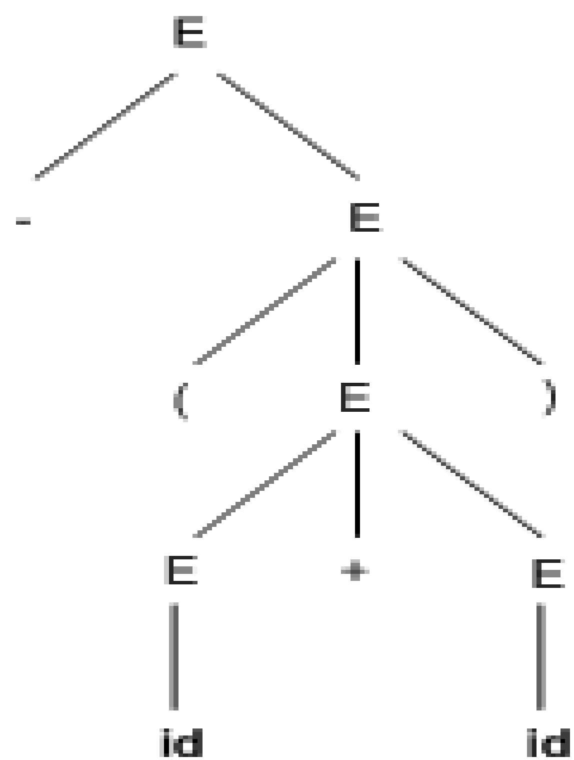
=> **-**(E+E)

=> **-**(**id**+E)

=>**-**(**id**+**id**)

Parse Trees

- **-(id**+**id)**
  - E => **-**E => **-**(E) => **-**(E+E) => **-**(**id**+E)=>**-**(**id**+**id**)



Grammar: Programming Languages

E → int
E → E Op E
E → (E)
Op → +
Op → -
Op → *
Op → /

E	E
⇒ E Op E	⇒ E Op E
⇒ int Op E	⇒ E Op (E)
⇒ int * E	⇒ E Op (E Op E)
⇒ int * (E)	⇒ E Op (E Op int)
⇒ int * (E Op E)	⇒ E Op (E + int)
⇒ int * (int Op E)	⇒ E Op (int + int)
⇒ int * (int + E)	⇒ E * (int + int)
⇒ int * (int + int)	⇒ int * (int + int)

Example: Leftmost, Rightmost Derivation

1. Goal → Expr
2. Expr → Expr op Expr
3. | number
4. | id
5. Op → +
6. | -
7. | *
8. | /

x-2*y

Derivations and Precedence

- The derivation of the previous example give rise to different parse trees:
  - $x – (2*y)$
  - $(x – 2)*y$.
- The two derivations point out a problem with the grammar: it has no notion of precedence (or implied order of evaluation).
- To add precedence: force parser to recognise high-precedence subexpressions first.

Outline

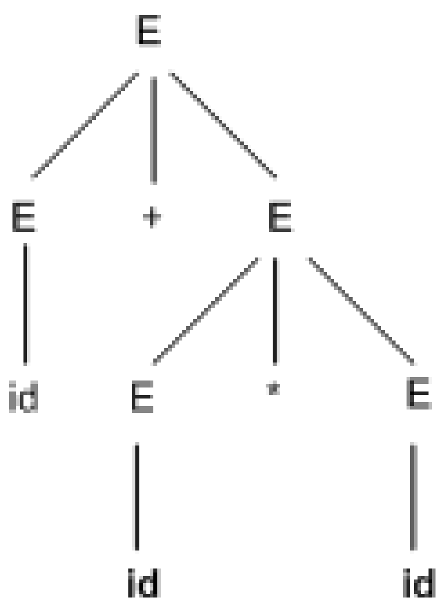
- Context-free Grammars
- Derivations
- Parse trees
- **Ambiguity**

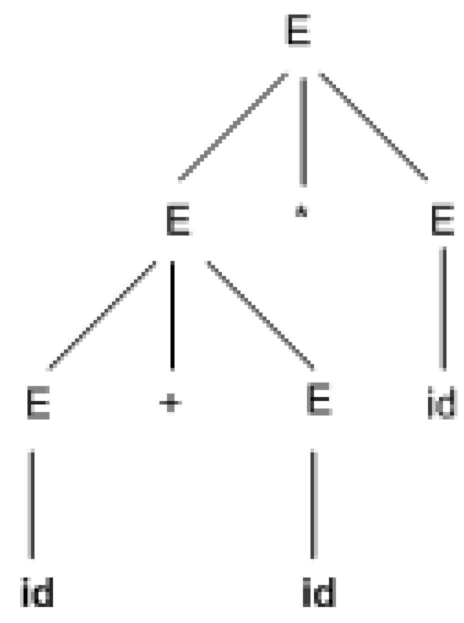
Ambiguity

- A grammar that produces more than one parse tree for some sentence is ambiguous.
  - If a grammar has more than one leftmost derivation for a single sentential form, the grammar is ambiguous.
  - If a grammar has more than one rightmost derivation for a single sentential form, the grammar is ambiguous.

Ambiguity

- Example: id+id*id





Ambiguity

- Example:

Stmt $\rightarrow$ if Expr then Stmt

| if Expr then Stmt else Stmt

| ...other…

- What are the derivations of:

if E1 then if E2 then S1 else S2

Eliminating Ambiguity

- Rewrite the grammar to avoid the problem
- Match each else to innermost unmatched if:
  1. Stmt $\rightarrow$ IfwithElse
  2. | IfnoElse
  3. IfwithElse $\rightarrow$ if Expr then IfwithElse else IfwithElse
  4. | … other stmts…
  5. IfnoElse $\rightarrow$ if Expr then Stmt
  6. | if Expr then IfwithElse else IfnoElse

Eliminating Ambiguity

- Rewrite the grammar to avoid the problem
- Match each else to innermost unmatched if:

1. Stmt → IfwithElse
2. | IfnoElse
3. IfwithElse → if Expr then IfwithElse else IfwithElse
4. | … other stmts…
5. IfnoElse → if Expr then Stmt
6. | if Expr then IfwithElse else IfnoElse

	Stmt
(2)	IfnoElse
(5)	if Expr then Stmt
(-)	if E1 then Stmt
(1)	if E1 then IfwithElse
(3)	if E1 then if Expr then IfwithElse else IfwithElse
(-)	if E1 then if E2 then IfwithElse else IfwithElse
(4)	if E1 then if E2 then S1 else IfwithElse
(4)	if E1 then if E2 then S1 else S2

Deeper Ambiguity

- Ambiguity usually refers to confusion in the CFG
- Overloading can create deeper ambiguity
  - E.g.: a=b(3) : b could be either a function or a variable.
- Disambiguating this one requires context:
  - An issue of type, not context-free syntax
  - Needs values of declarations
  - Requires an extra-grammatical solution

Deeper Ambiguity

- Resolving ambiguity:
  - Context-free ambiguity: rewrite the grammar
  - Context-sensitive ambiguity: check with other means: needs knowledge of types, declarations, … This is a language design problem

- Sometimes the compiler writer accepts an ambiguous grammar: parsing techniques may do the “right thing”.

Parsing Techniques

- Top-down parsers

- Bottom-up parsers

Parsing Techniques

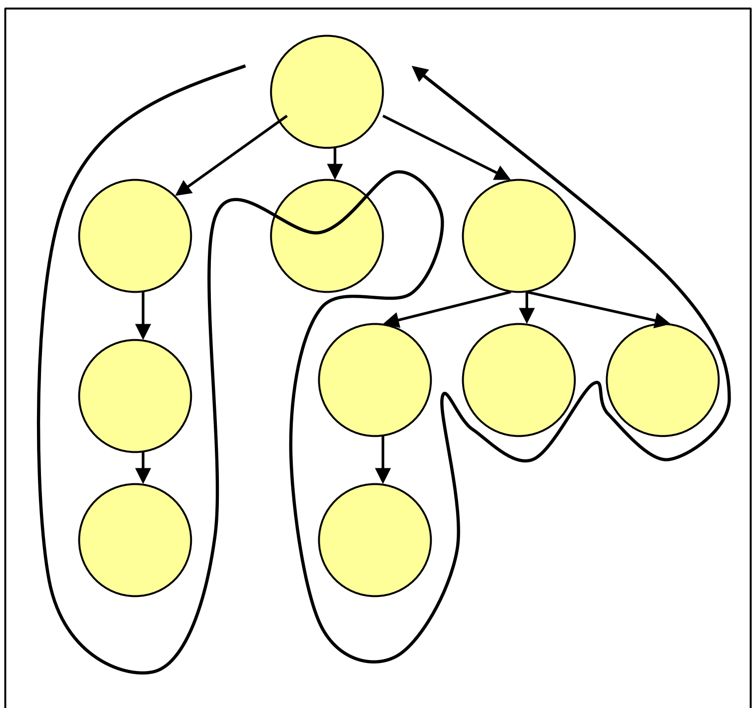
- Top-down parsers:
  - Construct the top node of the tree and then the rest in pre-order. (depth-first)
  - Pick a production & try to match the input; if you fail, backtrack.
  - Essentially, we try to find a **leftmost** derivation for the input string (which we scan left-to-right).
  - Some grammars are backtrack-free (predictive parsing).

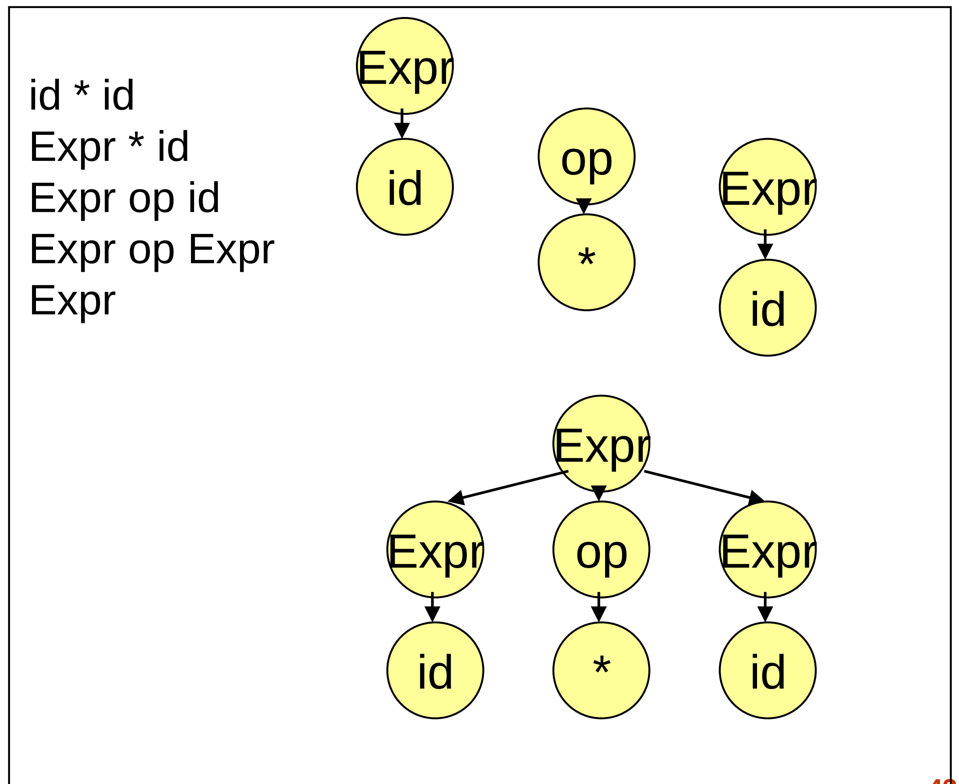
Parsing Techniques

- Bottom-up parsers:
  - Construct the tree for an input string, beginning at the leaves and working up towards the top (root).
  - Bottom-up parsing, using left-to-right scan of the input, tries to construct a **rightmost** derivation in reverse.
  - Handle a large class of grammars.

Top-down vs Bottom-up!

- Has an analogy with two special cases of depth-first traversals:
  - Pre-order: first traverse node x and then x's subtrees in left-to-right order. (action is done when we first visit a node)
  - Post-order: first traverse node x's subtrees in left-to-right order and then node x. (action is done just before we leave a node for the last time)





Summary

- The parser's task is to analyse the input program as abstracted by the scanner.

- Next Lecture: Top-Down Parsing

Reading

- Aho2, Sections 4.1; 4.2; 4.3.1; 4.3.2; (see also pp.56-60)
- Aho1, pp. 160-175
- Hunter, pp. 21-44
- Grune pp.34-40; 110-115
- Cooper, pp.73-89.

Question?