Lecture 7: Introduction to Parsing (Syntax Analysis)

Lexical Analysis

Syntax Analysis

Lexical Analysis:

- Reads characters of the input program and produces tokens.

But: Are they syntactically correct? Are they valid sentences of the input language?

Today’s lecture

- context-free grammars,
- derivations,
- parse trees,
- ambiguity

Not all languages can be described by Regular Expressions!!

The descriptive power of regular expressions has limits:

- REs cannot be used to describe balanced or nested constructs: E.g., set of all strings of balanced parentheses {(), (()), ((())), …}, or the set of all 0s followed by an equal number of 1s, {01, 0011, 000111, ...}.

- In regular expressions, a non-terminal symbol cannot be used before it has been fully defined.

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: $A \to \varepsilon$, $A \to a$, $A \to pB$ are allowed.

Regular Languages ⊂ Context-Free Languages ⊂ Cont.Sens.Ls ⊂ Phr.Str.Ls

Expressing Syntax

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

  A grammar, G, is a 4-tuple G={S,N,T,P}, where:

  S is a starting symbol;
  N is a set of non-terminal symbols;
  T is a set of terminal symbols;
  P is a set of production rules.

Example:

$CatNoise \rightarrow CatNoise\ miau$	**rule 1**
$\mid miau$	**rule 2**

- We can use the CatNoise grammar to create sentences: E.g.:

Rule	**Sentential Form**
-	*CatNoise*
1	*CatNoise miau*
2	*miau miau*

- Such a sequence of rewrites is called a derivation

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

Derivations and Parse Trees

Derivation: a sequence of derivation steps:

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

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!)

A parse tree

A **parse tree** is a graphical representation for a derivation that filters out the choice regarding the replacement order.

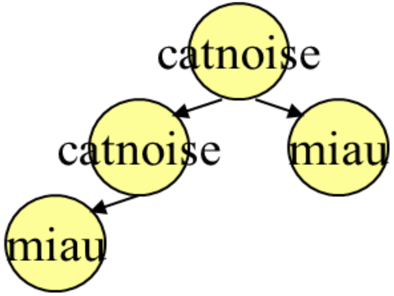
Construction:

start with the starting symbol (root of the tree);

for each sentential form:

- *add children nodes (for each symbol in the right-hand-side of the production rule that was applied) to the node corresponding to the left-hand-side symbol.*

The leaves of the tree (read from left to right) constitute a sentential form (fringe, or yield, or frontier, or ...)



Find leftmost, rightmost derivation & parse tree for: x-2*y

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

Find leftmost, rightmost derivation & parse tree for: x-2*y

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

Derivations and Precedence

- The leftmost and the rightmost derivation in the previous slide give rise to different parse trees. Assuming a standard way of traversing, the former will evaluate to *x – (2*y)*, but the latter will evaluate to *(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.

Ambiguity

A grammar that produces more than one parse tree for some sentence is ambiguous. Or:

- 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.

Example:

- Stmt → 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

Example:

- Stmt → 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 → 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 if
(4) E1 then if E2 then S1 else IfwithElse
(4) 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 → 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

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
- Resolving ambiguity:
  - if context-free: 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:

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).

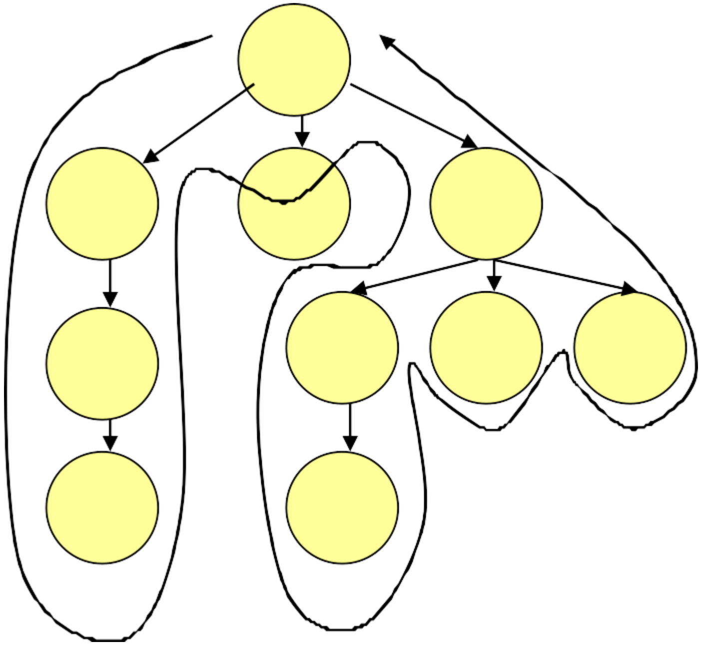
Bottom-up parsers

- 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 …

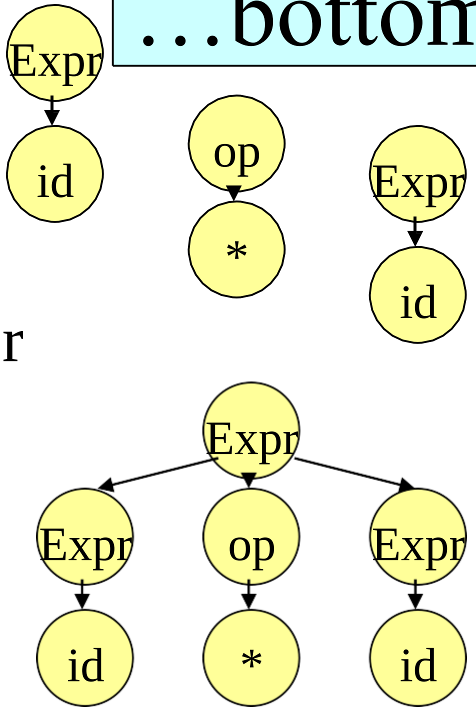
Has an analogy with two special cases of depth-first traversals:

- 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)



…bottom-up!

id * id
Expr * id
Expr op id
Expr op Expr
Expr



Top-Down Recursive-Descent Parsing

- 1. Construct the root with the starting symbol of the grammar.
- 2. Repeat until the fringe of the parse tree matches the input string:
  - Assuming a node labelled A, select a production with A on its left-hand-side and, for each symbol on its right-hand-side, construct the appropriate child.
  - When a terminal symbol is added to the fringe and it doesn't match the fringe, backtrack.
  - Find the next node to be expanded.

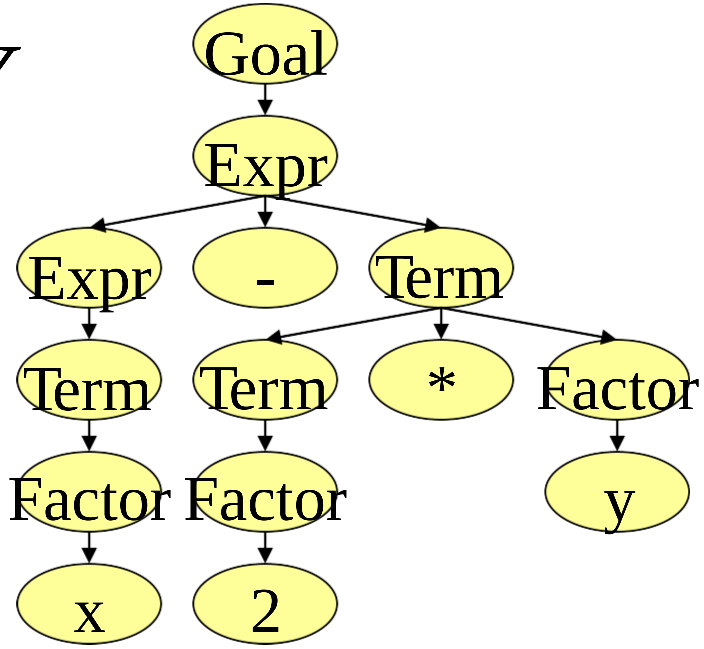
The key is picking the right production in the first step: that choice should be guided by the input string.

Example:

1. *Goal* → *Expr*
2. *Expr* → *Expr* + *Term*
3. | *Expr* – *Term*
4. | *Term*
5. *Term* → *Term* * *Factor*
6. | *Term* / *Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

Example: Parse *x-2*y*

Steps (one scenario from many)



Other choices for expansion are possible:

Rule	Sentential Form	Input
-	*Goal*	| x – 2*y
1	*Expr*	| x – 2*y
2	*Expr + Term*	| x – 2*y
2	*Expr + Term + Term*	| x – 2*y
2	*Expr + Term + Term + Term*	| x – 2*y
2	*Expr + Term + Term + … + Term*	| x – 2*y

- Wrong choice leads to non-termination!
- This is a bad property for a parser!
- Parser must make the right choice!

Example: Parse *x-2*y*

Rule	**Sentential Form**	**Input**
-	*Goal*	| x – 2*y
1	*Expr*	| x – 2*y
2	*Expr + Term*	| x – 2*y
4	*Term + Term*	| x – 2*y
7	*Factor + Term*	| x – 2*y
9	*id + Term*	| x – 2*y
Fail	*id + Term*	x | – 2*y
Back	*Expr*	| x – 2*y
3	*Expr – Term*	| x – 2*y
4	*Term – Term*	| x – 2*y
7	*Factor – Term*	| x – 2*y
9	*id – Term*	| x – 2*y
Match	*id – Term*	x – | 2*y
7	*id – Factor*	x – | 2*y
9	*id – num*	x – | 2*y
Fail	*id – num*	x – 2 | *y
Back	*id – Term*	x – | 2*y
5	*id – Term * Factor*	x – | 2*y
7	*id – Factor * Factor*	x – | 2*y
8	*id – num * Factor*	x – | 2*y
match	*id – num * Factor*	x – 2* | y
9	*id – num * id*	x – 2* | y
match	*id – num * id*	x – 2*y |

Example:

1. *Goal* → *Expr*
2. *Expr* → *Expr + Term*
3. | *Expr – Term*
4. | *Term*
5. *Term* → *Term * Factor*
6. | *Term / Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

Left-Recursive Grammars

- **Definition**: A grammar is left-recursive if it has a non-terminal symbol A, such that there is a derivation $A \Rightarrow Aa$, for some string a.
- A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop.
- **Eliminating left-recursion**: In many cases, it is sufficient to replace $A \rightarrow Aa \,|\, b$ with $A \rightarrow bA'$ and $A' \rightarrow aA' \,|\, \varepsilon$
- **Example**:

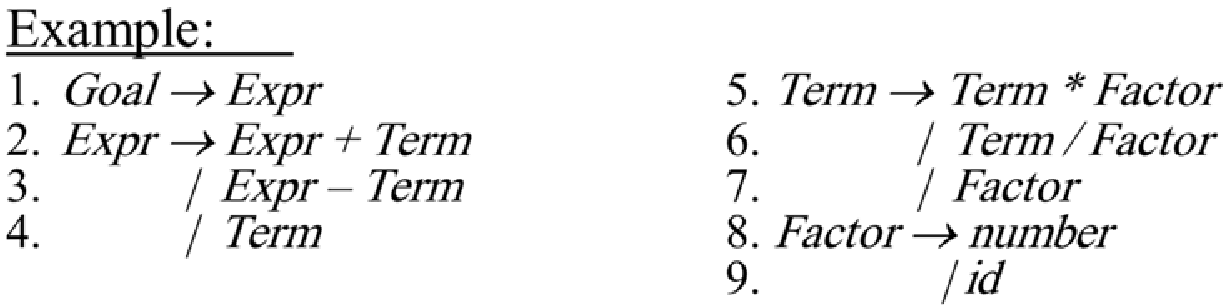
$$Sum \rightarrow Sum{+}number \,|\, number$$

would become:

$$Sum \rightarrow number \,|\, S'um$$

$$Sum' \rightarrow +number \;\; Sum' \,|\, \varepsilon$$

Eliminating Left Recursion



Example:

1. $Goal \rightarrow Expr$
2. $Expr \rightarrow Expr + Term$
3. $\quad | \; Expr - Term$
4. $\quad | \; Term$
5. $Term \rightarrow Term * Factor$
6. $\quad | \; Term \,/\, Factor$
7. $\quad | \; Factor$
8. $Factor \rightarrow number$
9. $\quad | \; id$

Applying the transformation to the Grammar of the Example we get:

$$Expr \rightarrow Term \,|\, Expr'$$

$$Expr' \rightarrow +Term\ Expr' \ \ | -Term\ Expr' \ \ | \varepsilon$$

$$Term \rightarrow \ Factor \,|\, Term'$$

$$Term' \rightarrow \ \ *Factor \ \ Term' \ \ | /Factor \ \ Term' \ \ | \varepsilon$$

($Goal \rightarrow Expr$ and $Factor \rightarrow number \,|\, id$ remain unchanged)

Non-intuitive, but it works!

Eliminating Left Recursion Algorithm

General algorithm: works for non-cyclic, no ε-productions grammars

```
1. Arrange the non-terminal symbols in order: $A_1, A_2, A_3, \ldots, A_n$
2. For i=1 to n do
        for j=1 to i-1 do
            I) replace each production of the form $A_i \rightarrow A_j\gamma$ with
            the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$
            where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current $A_j$ productions
            II) eliminate the immediate left recursion among the $A_i$
```

Example

E→ E+T|T
T→T*F|F
F→ (E)|id

Where are we?

- We can produce a top-down parser, but:
  - if it picks the wrong production rule it has to backtrack.
- **Idea**: look ahead in input and use context to pick correctly.
- How much lookahead is needed?
  - In general, an arbitrarily large amount.
  - Fortunately, most programming language constructs fall into subclasses of context-free grammars that can be parsed with limited lookahead.

Predictive Parsing

- Basic idea:
  - For any production $A \rightarrow a / b$ we would like to have a distinct way of choosing the correct production to expand.
- *FIRST* sets:
  - For any symbol A, *FIRST(A)* is defined as the set of terminal symbols that appear as the first symbol of one or more strings derived from A.

  E.g. for previous grammer: *FIRST(Expr')={+,-,ε}, {FIRST(Term')={*,/,ε}, FIRST(Factor)={number, id*
- The LL(1) property:
  - If $A \rightarrow a$ and $A \rightarrow b$ both appear in the grammar, we would like to have: $FIRST(a) \cap FIRST(b) = \emptyset$. This would allow the parser to make a correct choice with a lookahead of exactly one symbol!

Left Factoring

What if my grammar does not have the LL(1) property?

Sometimes, we can transform a grammar to have this property.

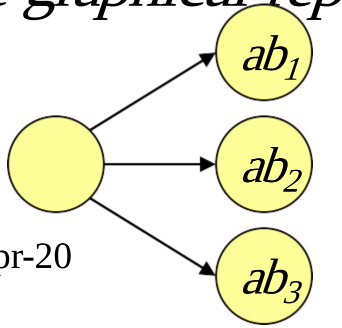
Algorithm:

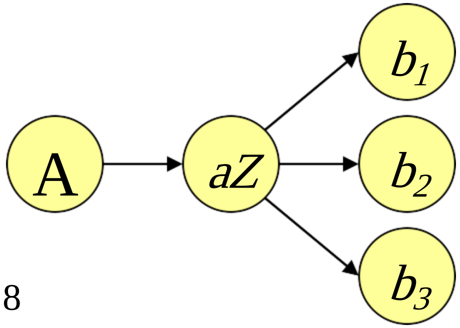
1. *For each non-terminal A, find the longest prefix, say a, common to two or more of its alternatives*
2. *if* $a \neq \varepsilon$ *then replace all the A productions,* $A \rightarrow ab_1|ab_2|ab_3|...|ab_n|\gamma$, *where* γ *is anything that does not begin with a, with* $A \rightarrow aZ\,|\,\gamma$ *and* $Z \rightarrow b_1|b_2|b_3|...|b_n$

Repeat the above until no common prefixes remain

Example: $A \rightarrow ab_1\,|\,ab_2\,|\,ab_3$ would become $A \rightarrow aZ$ and $Z \rightarrow b_1|b_2|b_3$

Note the graphical representation:





Example

S→iEtS|iEtSeS|a
E→ b

Example

$Goal \rightarrow Expr$
$\boldsymbol{Expr \rightarrow Term + Expr}$
$\boldsymbol{\mid Term - Expr}$
$\boldsymbol{\mid Term}$

$\boldsymbol{Term \rightarrow Factor * Term}$
$\boldsymbol{\mid Factor \,/\, Term}$
$\boldsymbol{\mid Factor}$
$Factor \rightarrow number$
$\mid id$

We have a problem with the different rules for *Expr* as well as those for *Term*. In both cases, the first symbol of the right-hand side is the same (*Term* and *Factor*, respectively). E.g.:

$$FIRST(Term) = FIRST(Term) \cap FIRST(Term) = \{number, id\}.$$
$$FIRST(Factor) = FIRST(Factor) \cap FIRST(Factor) = \{number, id\}.$$

Applying left factoring:

$Expr \rightarrow Term\ Expr'$
$Expr' \rightarrow +Expr \mid - Expr \mid \varepsilon$

$Term \rightarrow Factor\ Term'$
$Term' \rightarrow *\ Term \mid /\ Term \mid \varepsilon$

$FIRST(+) = \{+\};\ FIRST(-) = \{-\};\ FIRST(\varepsilon) = \{\varepsilon\};$
$FIRST(-) \cap FIRST(+) \cap FIRST(\varepsilon) = = \emptyset$

$FIRST(*) = \{*\};\ FIRST(/) = \{/\};\ FIRST(\varepsilon) = \{\varepsilon\};$
$FIRST(*) \cap FIRST(/) \cap FIRST(\varepsilon) = = \emptyset$

Parsing Table

Example (cont.)

1. *Goal* → *Expr*
2. *Expr* → *Term Expr´*
3. *Expr´*→ + *Expr*
4. |- *Expr*
5. |*ε*
6. *Term* → *Factor Term´*
7. *Term´*→ * *Term*
8. |/ *Term*
9. |*ε*
10. *Factor* → *number*
11. |*id*

Rule	**Sentential Form**	**Input**
-	*Goal*	| x – 2*y
1	*Expr*	| x – 2*y
2	*Term Expr´*	| x – 2*y
6	*Factor Term´ Expr´*	| x – 2*y
11	*id Term´ Expr´*	| x – 2*y
Match	*id Term´ Expr´*	x | – 2*y
9	*id ε Expr´*	x | – 2*y
4	*id – Expr*	x | – 2*y
Match	*id – Expr*	x – | 2*y
2	*id – Term Expr´*	x – | 2*y
6	*id – Factor Term´ Expr´*	x – | 2*y
10	*id – num Term´ Expr´*	x – | 2*y
Match	*id – num Term´ Expr´*	x – 2 | *y
7	*id – num * Term Expr´*	x – 2 | *y
Match	*id – num * Term Expr´*	x – 2* | y
6	*id – num * Factor Term´ Expr´*	x – 2* | y
11	*id – num * id Term′ Expr´*	x – 2* | y
Match	*id – num * id Term´ Expr´*	x – 2*y |
9	*id – num * id Expr´*	x – 2*y |
5	*id – num * id*	x – 2*y |

The next symbol determines each choice correctly. No backtracking needed.

Conclusion

- Top-down parsing:
  - recursive with backtracking (not often used in practice)
  - recursive predictive

- Given a Context Free Grammar that doesn’t meet the LL(1) condition, it is undecidable whether or not an equivalent LL(1) grammar exists.
- Next time: Bottom-Up Parsing