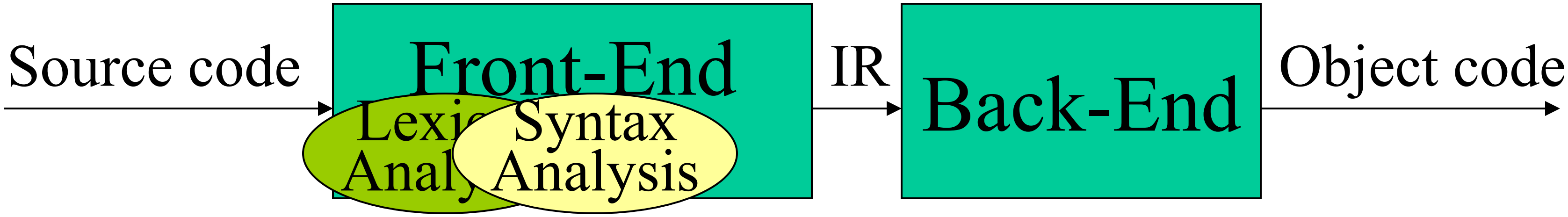
Lecture 7: Introduction to Parsing (Syntax Analysis)



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!!

(Lecture 3, Slide 7)

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 $\leq$ 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: $A\rightarrow\varepsilon$, $A\rightarrow a$, $A\rightarrow pB$ are allowed.

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

Expressing Syntax

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

  Recall (Lect.3, slide 3): 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**
| *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:

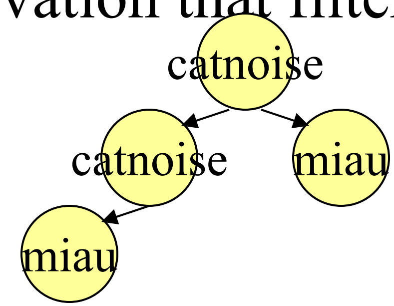
- At each step, we choose a non-terminal to replace.
- Different choices can lead to different derivations.

Two derivations are of interest:

- <u>Leftmost derivation</u>: at each step, replace the leftmost non-terminal.
- <u>Rightmost derivation</u>: at each step, replace the rightmost non-terminal

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

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. | /

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

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
- 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".

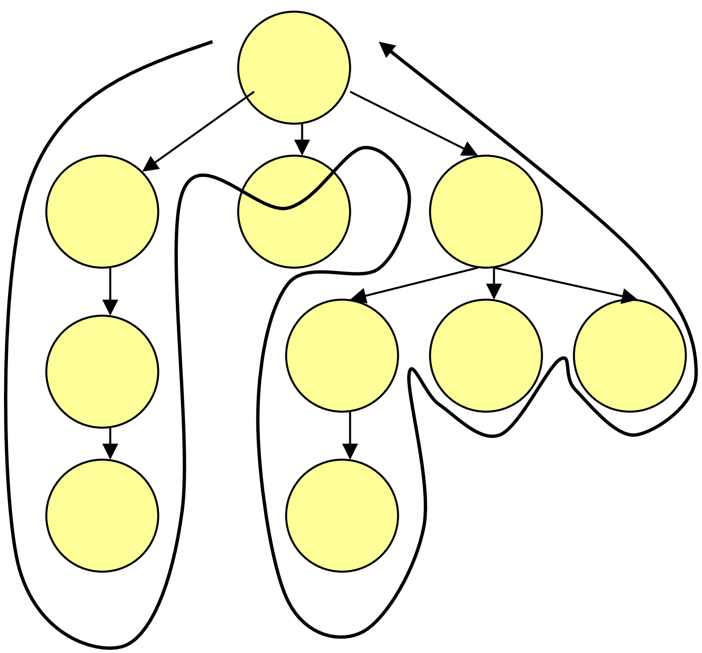
Parsing techniques

- Top-down parsers:
  - 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 …

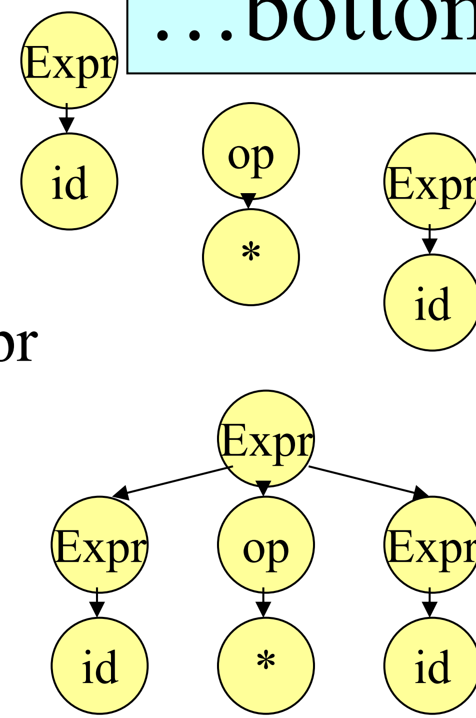
Has an analogy with two special cases of depth-first traversals:

- 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.

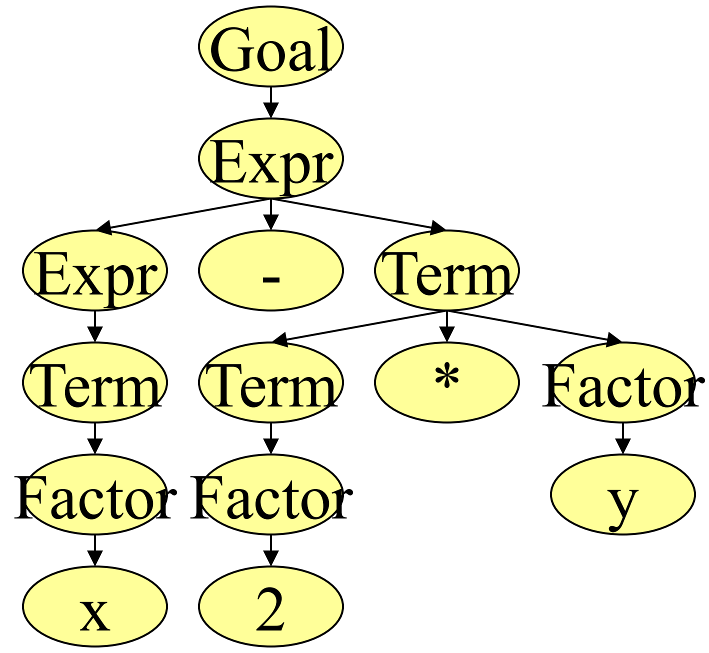
Example:

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

Example: Parse *x-2*y*

Steps (one scenario from many)

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 |



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!

Conclusion

- The parser's task is to analyse the input program as abstracted by the scanner.
- <u>Next time</u>: Top-Down Parsing
- <u>Reading</u>: Aho2, Sections 4.1, 4.2, 4.3.1, 4.3.2, (see also pp.56-60); Aho1, pp. 160-175; Grune pp.34-40, 110-115; Hunter pp. 21-44; Cooper pp.73-89.
- <u>Exercises</u>: Aho1 267-268; Hunter pp. 44-46.