Introduction to Parsing (Syntax Analysis)

Introduction

Lexical Analysis:

- Reads characters of the input program and produces tokens.

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

➔Now:

Context-free grammars,

Derivations,

Parse trees,

Ambiguity

Parsing: top-down and bottom-up.

Regular Expression

- The set of all strings of balanced parentheses {(), (()), ((())), …},
- The set of all 0s followed by an equal number of 1s, {01, 0011, 000111, ...}.
- **Not all languages can be described by Regular Expressions!!**

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

  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 → *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*

Derivation and Parsing

- Such a sequence of rewrites is called a **derivation**
- The process of discovering a derivation for some sentence is called **parsing**!

Derivations

Derivation: a sequence of derivation steps:
- At each step, we choose a <u>non-terminal</u> to <u>replace</u>.
- Different choices can lead to different derivations.

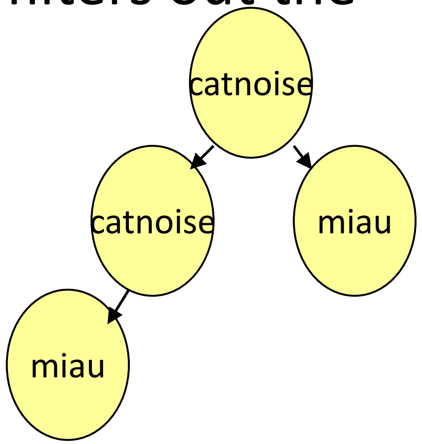
Two derivations are of interest:

- <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

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 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 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 $x - (2*y)$.
    - 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 sub-expressions 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.

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

- 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

- if E1 then if E2 then S1 else S2

	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”.

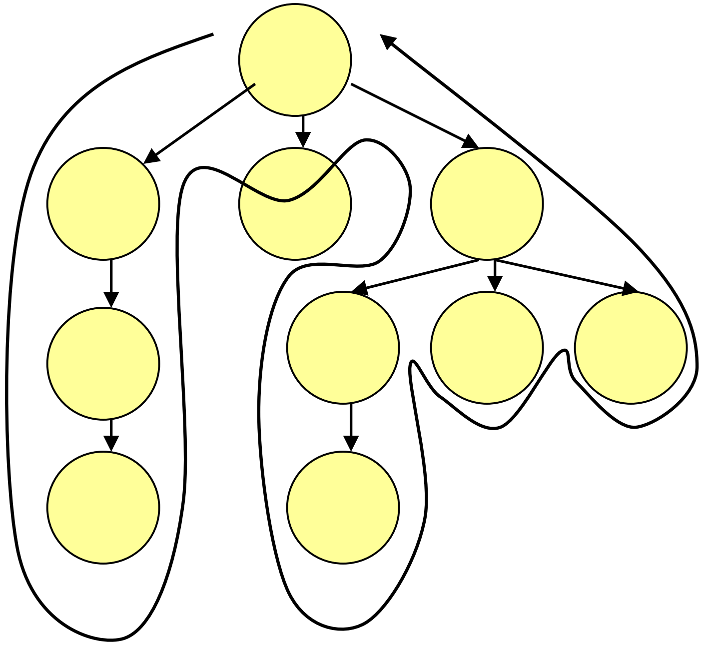
Parsing techniques

- 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 …

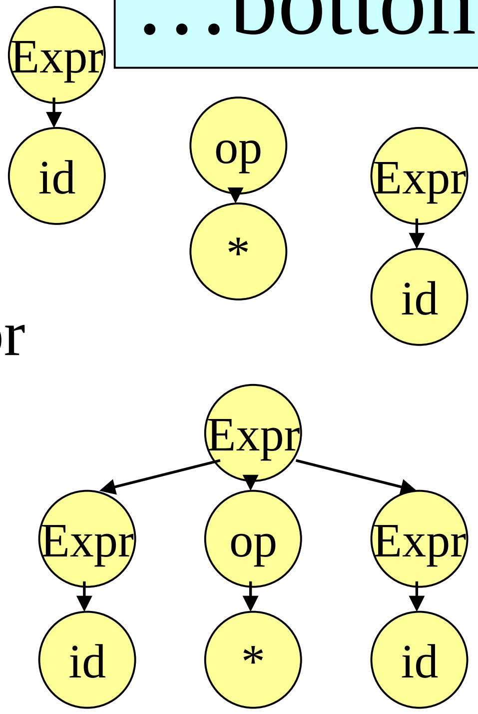
Has an analogy with two special cases of depth-first traversals:

- 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 Parsing

Next Class