Topics in IT 1

Parsing and Pattern Recognition

Week 04
Principles of Parsing

College of Information Science and Engineering
Ritsumeikan University
this week

review of last week’s topics

principles of parsing
  • derivations
  • ambiguity

top-down and bottom-up parsing
last week: grammar types

<table>
<thead>
<tr>
<th>type</th>
<th>name</th>
<th>allowed rules</th>
<th>restrictions</th>
<th>recogniser</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>unrestricted</td>
<td>( \gamma \rightarrow \alpha )</td>
<td>none</td>
<td>turing</td>
</tr>
<tr>
<td>1</td>
<td>context-sensitive</td>
<td>( \alpha N \beta \rightarrow \alpha \gamma \beta )</td>
<td>( \gamma \neq \emptyset )</td>
<td>bounded turing</td>
</tr>
<tr>
<td>2</td>
<td>context-free</td>
<td>( N \rightarrow \alpha )</td>
<td></td>
<td>FSM + stack</td>
</tr>
<tr>
<td>3</td>
<td>regular</td>
<td>( N \rightarrow t M )</td>
<td></td>
<td>FSM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( N \rightarrow t )</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( N \rightarrow \epsilon )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>finite choice</td>
<td>( N \rightarrow t )</td>
<td></td>
<td>comparison</td>
</tr>
</tbody>
</table>

\( \alpha, \beta \in (V_N \cup V_T)^* \) any sequence of symbols
\( \gamma \in (V_N \cup V_T)^+ \) any non-empty sequence of symbols
\( N, M \in V_N \) any non-terminal symbol
\( t \in V_T \) any terminal symbol
principles of parsing

a language \( L \) is a set of sentences

a grammar for \( L \) is a device for generating those sentences

\[
\text{grammar} \rightarrow \text{sentences} \\
\text{hierarchical} \rightarrow \text{linear} \\
\text{structured} \rightarrow \text{unstructured}
\]

this is backwards for most of our needs; we might want to

- convert arithmetic expression into operation sequence
  - perform operations and print result
- convert natural language sentence to syntactic structure
  - emit same structure in a different language
- convert source program to semantic representation
  - optimise the program, then generate code
principles of parsing

parsing is the process of going in the opposite direction

unstructured → structured
linear → hierarchical
sentences → ???

let’s consider simple grammar, for arithmetic expressions

\[
\begin{align*}
\text{EXP} & \rightarrow \text{SUM} \\
\text{SUM}_a & \rightarrow \text{SUM} + \text{SUM} \\
\text{SUM}_b & \rightarrow \text{NUM} \\
\text{NUM} & \rightarrow [0 \ldots 9]
\end{align*}
\]

an expression is a sum
a sum is an addition ...
... or a single number
a number is a digit

([0 \ldots 9] represents any one of the digits: 0123456789)

from this we can generate sentences such as

\[
\begin{align*}
9 \\
2 + 3 + 4
\end{align*}
\]
productions

let’s list the rules involved in generating \( \text{EXP} \rightarrow 2 + 3 \)

<table>
<thead>
<tr>
<th>rule</th>
<th>generates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( \text{EXP} \rightarrow \text{SUM}_a )</td>
<td></td>
</tr>
<tr>
<td>2. ( \text{SUM}<em>a \rightarrow \text{SUM}</em>{b1} + \text{SUM}_{b2} )</td>
<td>( \ldots + \ldots )</td>
</tr>
<tr>
<td>3. ( \text{SUM}_{b1} \rightarrow \text{NUM}_1 )</td>
<td></td>
</tr>
<tr>
<td>4. ( \text{NUM}_1 \rightarrow [0 \ldots 9] )</td>
<td>2</td>
</tr>
<tr>
<td>5. ( \text{SUM}_{b2} \rightarrow \text{NUM}_2 )</td>
<td></td>
</tr>
<tr>
<td>6. ( \text{NUM}_2 \rightarrow [0 \ldots 9] )</td>
<td>3</td>
</tr>
</tbody>
</table>

three things are evident:

- a non-terminal can have many production rules (letter subscripts)
- a rule can be invoked more than once (number subscripts)
- the entire production takes the form of a tree
  - root node corresponds to the start symbol
  - interior nodes correspond to non-terminal symbols
  - leaf nodes correspond to terminal symbols
productions and derivations

(rule)

1. EXP $\rightarrow$ SUM $a$
2. SUM $a$ $\rightarrow$ SUM $b_1$ + SUM $b_2$
3. SUM $b_1$ $\rightarrow$ NUM $1$
4. NUM $1$ $\rightarrow$ 2
5. SUM $b_2$ $\rightarrow$ NUM $2$
6. NUM $2$ $\rightarrow$ 3

each step in the production of the sentence is represented in the tree

the tree is a recursive structure that describes

- the final sentence
- how the final sentence is generated from the grammar
- the role of each lexeme in the final sentence

this is called the derivation of the sentence
parsing

given
- a sentence in a language, and
- the grammar for that language

we would like to find a tree describing its derivation
- because this encodes the syntactic meaning of the sentence

finding the derivation: possibility 1
- recursively generate all sentences (and their derivations)
- compare each generated sentence with the input
- if equal, we know the derivation of the sentence

this *exhaustive search* is extremely inefficient
parsing

finding the derivation: possibility 2

- infer the correct derivation, given
  - the final sentence
  - and the language grammar

considering only derivations that can generate the final sentence

this is parsing

- sentence × grammar → derivation
the parser is often generated from the language grammar

- the grammar is implicit in the parser’s implementation
- any complete sentence can be provided as the parser’s input
- the derivation tree for that sentence is the parser’s output
top-down vs. bottom-up parsing

sentence × grammar → derivation

there are two possibilities:

1. convert the start symbol into the input sentence
   • the derivation tree is constructed *top-down*
   • from root (start symbol) to leaves (terminal symbols)

2. convert the input sentence into the start symbol
   • the derivation tree is constructed *bottom-up*
   • from leaves (terminal symbols) to root (start symbol)
top-down parsing

begin with a sentential form (SF) equal to the start symbol

repeat {
    choose a non-terminal in the SF
    expand the non-terminal to create a new SF
    discard any SF whose terminals contradict the input
} until the SF is equal to the input sentence
top-down parsing example

begin with the start symbol

\[ \text{EXP} \rightarrow \text{SUM} \]

which has only one rule, so must be a \text{SUM}

\[ \text{EXP} \rightarrow \text{SUM} \]
top-down parsing example

we now have to choose between two possibilities:

\[
\begin{align*}
\text{EXP} & \rightarrow \text{SUM} \rightarrow \text{NUM} \\
\text{EXP} & \rightarrow \text{SUM} \rightarrow \text{NUM + SUM}
\end{align*}
\]

we can explore both possibilities

- in parallel (as shown here), or
- one at a time
  - back-tracking to try the other if we choose the wrong one
top-down parsing example

we can expand both **NUMs** (which match the input ‘2’)

\[
\begin{align*}
\text{EXP} & \rightarrow \text{SUM} \rightarrow 2 \\
\text{EXP} & \rightarrow \text{SUM} \rightarrow 2 + \text{SUM}
\end{align*}
\]

the first possibility now contradicts the input ‘2+3’

- the production is finished, but the entire sentence has not been matched

we *reject the first possibility*, and continue with the second

- which does not contradict the input
top-down parsing example

rejecting the first possibility leaves us with

\[ \text{EXP} \rightarrow 2 + \text{SUM} \]

in which \text{SUM} has to be expanded
top-down parsing example

we can again expand the **SUM** in two ways

\[ \text{EXP} \rightarrow 2 + \text{NUM} \]
\[ \text{EXP} \rightarrow 2 + \text{NUM} + \text{SUM} \]

in which the **NUM** is easy to replace with ‘3’

\[ \text{EXP} \rightarrow 2 + 3 \]
\[ \text{EXP} \rightarrow 2 + 3 + \text{SUM} \]

but now the second possibility contradicts the input ‘2+3’

- there is no second ‘+’ following ‘3’ in the input sentence
- *reject* the second possibility, and continue with the first
the remaining sentential form

\[ \text{EXP} \rightarrow 2 + 3 \]

contains no non-terminal symbols

- the production is finished, and
- we are at the end of the input sentence

we can accept this possibility

the path we recorded as we descended recursively through the grammar is the derivation of the input sentence

this strategy is called top-down, recursive-descent parsing
another example

S → NP  VP
NP → N       NP → A  NP
VP → V  NP
N → jack       N → jane
V → likes       V → knows
A → tall       A → short
A → handsome   A → pretty

tall jack likes short jane
another example

S

NP

A
NP

N
VP

V

A
NP

N
tall
jack
likes
short
jane
bottom-up parsing

reduce the input sentence to the start symbol

- ‘reduction’ is the opposite of ‘production’

begin with a sentential form (SF) equal to the input sentence
then use the grammar’s rules ‘backwards’:
  
  repeat {
    find symbol(s) in the SF corresponding to a rule’s right-hand-side
    replace those symbol(s) with the left-hand-side non-terminal
  } until only the start symbol remains

NUM \rightarrow [ 0 \ldots 9 ]
SUM \rightarrow NUM
SUM \rightarrow SUM + NUM
EXP \rightarrow SUM
bottom-up parsing example

we can consider bottom-up parsing as top-down parsing of a ‘reflected’ grammar, where we start with the sentence and apply productions until only the start symbol remains

original grammar:

\[
\begin{align*}
\text{a } & \quad \text{EXP} \rightarrow \text{SUM} \\
\text{SUM} & \rightarrow \text{SUM} + \text{NUM} \\
\text{SUM} & \rightarrow \text{NUM} \\
\text{NUM} & \rightarrow [0 \ldots 9] \\
\end{align*}
\]

reflected grammar:

\[
\begin{align*}
[0 \ldots 9] & \rightarrow \text{NUM} \\
\text{NUM} & \rightarrow \text{SUM} \\
\text{SUM} + \text{NUM} & \rightarrow \text{SUM} \\
\text{SUM} & \rightarrow \text{EXP} \\
\end{align*}
\]

work from sentence back towards start symbol:

\[
\begin{align*}
2 + 3 + 4 \\
\text{NUM} + 3 + 4 \\
\text{SUM} + 3 + 4 \\
\text{SUM} + \text{NUM} + 4 \\
\text{SUM} + 4 \\
\text{SUM} + \text{NUM} \\
\text{SUM} \\
\text{EXP} \\
\end{align*}
\]
ambiguity

our original expression grammar

\[
\begin{align*}
\text{EXP} & \rightarrow \text{SUM} \quad \text{an expression is a sum} \\
\text{SUM}_a & \rightarrow \text{SUM} + \text{SUM} \quad \text{a sum is an addition} \\
\text{SUM}_b & \rightarrow \text{NUM} \quad \ldots \text{or a single number} \\
\text{NUM} & \rightarrow [0 \ldots 9] \quad \text{a number is a digit}
\end{align*}
\]

can generate ‘2 + 3 + 4’ in two different ways

in particular, when expanding

\[
\text{SUM} \rightarrow \text{SUM}_a + \text{SUM}_b
\]
do we choose to expand \( \text{SUM}_a \) or \( \text{SUM}_b \) into \( \text{SUM} + \text{SUM} \)?

• choosing the leftmost (\( \text{SUM}_a \)) is called \textit{left-recursion}
• choosing the rightmost (\( \text{SUM}_b \)) is called \textit{right-recursion}
ambiguity
since generation is ambiguous, parsing is also ambiguous

depending on whether $\text{SUM} + \text{SUM}$ is left- or right-recursive

in the case of addition, this does not matter

but if we change $+$ to $-$ then the ambiguity becomes significant

• the two derivations mean different things (have different values)

• left-recursion is required for most arithmetic operators
  – because they are mostly left-associative
avoiding ambiguity
modifying the grammar slightly will fix the problem

for left-associative subtraction:

\[
\begin{align*}
\text{EXP} & \rightarrow \text{DIFF} \\
\text{DIFF} & \rightarrow \text{NUM} \\
\text{DIFF} & \rightarrow \text{DIFF} - \text{NUM} \\
\end{align*}
\]

\[ x - y - z = (x - y) - z \]

for right-associative subtraction:

\[
\begin{align*}
\text{EXP} & \rightarrow \text{DIFF} \\
\text{DIFF} & \rightarrow \text{NUM} \\
\text{DIFF} & \rightarrow \text{NUM} - \text{DIFF} \\
\end{align*}
\]

\[ x - y - z = x - (y - z) \]

neither alternative is correct for all operators

- most \textit{arithmetic} operators are left-associative
- \textit{assignment} operators are usually right-associative
enforcing precedence

we can enforce precedence in a similar way

(1) $\text{EXP} \rightarrow \text{IDENTIFIER} = \text{EXP}$
(2) $\text{EXP} \rightarrow \text{SUM}$
(3) $\text{SUM} \rightarrow \text{SUM} + \text{PROD}$
(4) $\text{SUM} \rightarrow \text{PROD}$
(5) $\text{PROD} \rightarrow \text{PROD} \ast \text{NUM}$
(6) $\text{PROD} \rightarrow \text{NUM}$
(7) $\text{NUM} \rightarrow [0 \ldots 9]$

this grammar ensures:

- assignment has lower precedence than arithmetic (1 and 2)
- multiplication has higher precedence than addition (3 and 4)
- assignment is right-associative (1)
- both multiplication and addition are left-associative (3 and 5)
unavoidable ambiguity

we cannot always eliminate ambiguity by modifying the grammar

(1) \text{STATEMENT} \rightarrow \text{if ( EXP ) STATEMENT}

(2) \text{STATEMENT} \rightarrow \text{if ( EXP ) STATEMENT else STATEMENT}

only one of the following prints correct information about y:

\begin{align*}
\text{if (x < y)} & \quad \text{if (x < y)} \\
\text{if (y < z)} & \quad \text{if (y < z)} \\
\text{puts("z is largest");} & \quad \text{puts("z is largest");} \\
\text{else} & \quad \text{else} \\
\text{puts("y is largest");} & \quad \text{puts("y is <= x");}
\end{align*}

depending on which of the ‘if’s includes the ‘else’

(in the C language, the version on the left is correct

\begin{itemize}
\item because \text{if-else} has ‘higher precedence’ than \text{if})
grammars generate the sentences of a language
  • sentences can be generated by recursive enumeration

parsing recognises an input sentence belonging to a language
  • according to the grammar that generates the language
  • reconstructing the tree of rules that generate the sentence
  • this tree is called the derivation of the sentence
top-down parsing turns the start symbol into the input sentence
  • the grammar is used to predict terminal symbols
  • predictions discarded when they do not match the input sentence

bottom-up parsing turns the input sentence into the start symbol
  • the input sentence is used to predict grammar rules
  • predictions discarded when the grammar has no corresponding rule

‘reflecting’ the grammar gives rules that describe how to do this
summary

ambiguity arises when there is more than one way to generate a sentence

- which means there is more than one way to parse it, too

ambiguity can often be avoided by careful design of the grammar

careful design of the grammar can also enforce operator precedence

some ambiguity cannot be avoided by modifying the grammar
homework

please study the following sections of the handout:

- Sections 3.6, 3.6.1, 3.6.2, 3.6.3 (finite automata)

review (or try to understand)

- what a non-deterministic finite state automaton (NFA) is, and
- how a NFA can be used to recognise sentences of this language:

  \[
  S \rightarrow aS \\
  S \rightarrow aB \\
  B \rightarrow bB \\
  B \rightarrow b
  \]
**glossary**

**accept** — to determine that a sentence can be generated by a grammar and terminate the parsing process successfully.

**back-track** — (in parsing) after rejecting the derivation under construction, go back to the most recent non-terminal with an unexplored alternative expansion and continue the search from there. (It will be necessary to rewind the input to the appropriate position, since any terminal symbols recently encountered will have caused the corresponding lexemes to be skipped.)

**bottom-up parsing** — a parsing strategy that begins with the input sentence and tries to reduce it to the start symbol.

**derivation** — the tree-structured path taken through the production rules to produce a given sentence.

**exhaustive search** — finding a solution to a problem by enumerating all possible candidate solutions and checking whether each candidate provides a viable solution.
infer — to deduce or conclude information from evidence and reasoning, rather than from explicit statements.

leftmost derivation — a derivation produced by expanding the leftmost (first) non-terminal in each successive sentential form.

parsing — the process of finding a derivation, according to some given grammar, for a given input sentence.

recursive descent — a top-down parsing strategy in which each non-terminal in a rule is recursively tested from left to right, and in which each terminal encountered must match the next lexeme present in the input sentence. (Left-recursive rules in a grammar will cause a recursive-descent parser to enter an infinite loop.

recursive enumeration — the process of generating all possible sentences in a language by recursively expanding every non-terminal in every rule, emitting sentences when no non-terminals remain, until all possible paths have been followed.
**recursive structure** — a data structure in which parts have the same properties as the whole.

**reduce** — to remove one or more symbols from a sentential form, replacing them with the single left-hand side non-terminal that represents them. Reducing a sentential form moves it one step closer to the start symbol within its derivation.

**reject** — determine that a derivation is not possible given the input sentence and grammar, and to discard it from the set of derivations being considered.

**rightmost derivation** — a derivation produced by expanding the rightmost (last) non-terminal in each successive sentential form.

**top-down parsing** — a parsing strategy that begins at the start symbol and then tries to generate the input sentence by recursively enumerating the sentences of the language, truncating the search space whenever a non-terminal in the grammar does not match the next lexeme in the input sentence.