Syntax analysis (Ch. 2 [M])

Takes a list of tokens, gives a syntax tree

context-free grammars

\[ N \rightarrow X_1 \ldots X_n \]

production (rule, or production rule)

one nonterminal symbol

on the LHS

\[ A \rightarrow a \]

\[ A \rightarrow aA \]
\[
\begin{align*}
B & \rightarrow \varepsilon \\
B & \rightarrow aB
\end{align*}
\]

where:
\[
\begin{align*}
s^* & \rightarrow \{\varepsilon\} \cup \{vw \mid v \in L(s), w \in L(s^*)\}
\end{align*}
\]

Each string in the language is a concatenation of any number of strings in the language of \(s\).

Up to this point, we have given examples of c.f. languages that are also regular languages, for example:

\[
\begin{align*}
S & \rightarrow \varepsilon \\
S & \rightarrow ax \\
S & \rightarrow ax^2
\end{align*}
\]

\[\{a^n b^n \mid n \geq 0\} = \varepsilon, ab, aabb, aaabbb, \ldots\]
In EBNF:

\[ T \rightarrow b^+ | aTa \]

### (2)
Several nonterminals: \( T \) and \( R \)

\[
\begin{align*}
T & \rightarrow R \\
T & \rightarrow aTa \\
R & \rightarrow b \\
R & \rightarrow bR
\end{align*}
\]

Start symbol: \( T \)

---

Context-free grammar, using regexp notation

You may use *+, *, ?
<table>
<thead>
<tr>
<th>Form of $s_i$</th>
<th>Productions for $N_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>$N_i \rightarrow$</td>
</tr>
<tr>
<td>a</td>
<td>$N_i \rightarrow a$</td>
</tr>
<tr>
<td>$s_js_k$</td>
<td>$N_i \rightarrow N_jN_k$</td>
</tr>
</tbody>
</table>
| $s_j|s_k$     | $N_i \rightarrow N_j$  
|              | $N_i \rightarrow N_k$  |
| $s_j*$       | $N_i \rightarrow N_jN_i$  
|              | $N_i \rightarrow$    |
| $s_j+$       | $N_i \rightarrow N_jN_i$  
|              | $N_i \rightarrow N_j$  |
| $s_j?$       | $N_i \rightarrow N_j$  
|              | $N_i \rightarrow$    |

Fig: From regular expression to context-free grammars
Syntactic categories: constructs of a (programming) language that differ in meaning. Three typical ones:

- Expressions: are evaluated to yield a value
- Commands: are executed to change memory or perform I/O
- Declarations: define properties of names used in other parts of the program

Expressions can be described by a regular expression:

\[ \text{num}((+|\times|\div)\text{num})^* \]

A grammar for the example expressions:

- \( Exp \rightarrow Exp + Exp \)
- \( Exp \rightarrow Exp - Exp \)
- \( Exp \rightarrow Exp * Exp \)
- \( Exp \rightarrow Exp / Exp \)
- \( Exp \rightarrow \text{num} \)
- \( Exp \rightarrow (Exp) \)

Each syntactic category is denoted by a non-terminal, such as \( Exp \) in the grammar above.

2, 5+2-3, 3-6*5, 4/5+2

(3-6)*5 ×
A grammar for (simple) statements

Stat → id := Exp

Stat → Stat ; Stat

Stat → if Exp then Stat else Stat
d two-way conditional

Stat → if Exp then Stat
d one-way conditional

Ex. 2, 3 [M] (front part)
Derivation $\Rightarrow$ or, sometimes $::=$

$n$ may be rewritten as

1. $\alpha \rightarrow_\beta \Rightarrow \alpha \rightarrow \beta$ if there is a production $N \rightarrow \gamma$

2. $\alpha \Rightarrow \alpha$ (reflexivity)

3. $\alpha \Rightarrow \gamma$ if there is a $\beta$ such that $\alpha \Rightarrow \beta$ and $\beta \Rightarrow \gamma$ (transitivity)

**Definition 3.1** Given a context-free grammar $G$ with start symbol $S$, terminal symbols $T$ and productions $P$, the language $L(G)$ that $G$ generates is defined to be the set of strings of terminal symbols that can be obtained by derivation from $S$ using the productions $P$, i.e., the set $\{w \in T^* \mid S \Rightarrow w\}$. 

For one step in a derivation:

$P \Rightarrow (P) \Rightarrow ((P)) \Rightarrow (((())))$ sentence
\[ T \rightarrow R \]
\[ T \rightarrow aTc \]
\[ R \rightarrow RbR \]

\[ T \Rightarrow aTc \Rightarrow aRc \Rightarrow ac \]

T

⇒ aTc
⇒ aaTcc
⇒ aaRrc
⇒ aaRbRcc
⇒ aaRbRbcc
⇒ aaRbRbRbcc
⇒ aaRbRbRbRbcc
⇒ aaRbRbRbRbcc
⇒ aaabbRbRbcc
⇒ aabbbbcc

neither a leftmost nor a rightmost derivation
a leftmost derivation
Syntax tree

This tree corresponds to two different derivations described before.

A grammar such that there exists a string in its language that has two distinct syntax trees is ambiguous.

\[ T \Rightarrow a \quad T \Rightarrow a \varepsilon \]

\[ T \Rightarrow a \varepsilon \quad T \Rightarrow a \varepsilon \]

\[ T \Rightarrow a T c \Rightarrow a \varepsilon T c \Rightarrow a a R c c \Rightarrow a a R b R c c \Rightarrow a a b R c c \Rightarrow a a b R b R c c \Rightarrow a a b b R c c \Rightarrow a a b b R b R c c \Rightarrow a a b b b R c c \Rightarrow a a b b b b c c \]
This grammar is a non-ambiguous version of

\[
\begin{align*}
T & \rightarrow R \\
T & \rightarrow aTc \\
R & \rightarrow bR \\
\end{align*}
\]

\[
\begin{align*}
T & \rightarrow R \\
T & \rightarrow aTc \text{, which is ambiguous} \\
R & \rightarrow RbR \\
\end{align*}
\]

Any grammar with productions like these:

\[
\begin{align*}
N & \rightarrow NaN \\
N & \rightarrow \beta \\
\end{align*}
\]

is ambiguous, \( \beta \beta \beta \beta \) as a contextual form.

Both left and right recursion.
Fig. 2.10  Fully reduced tree for the syntax tree in Fig. 2.7

Fig. 2.11  Preferred syntax tree for \(2 + 3 \times 4\) using Grammar 2.2, and the corresponding fully reduced tree

* Associates mildly more tightly than +
\begin{align*}
\text{Exp} & \rightarrow \text{Exp} + \text{Exp} \\
\text{Exp} & \rightarrow \text{num}
\end{align*}

We need to get rid of productions that are both left and right recursive.
Operator precedence & associativity

\[ E \rightarrow E \oplus E \]
\[ E \rightarrow \text{num} \]

\[ E \rightarrow E \oplus E' \]
\[ E' \rightarrow E' \text{ only left recursive} \]
\[ E' \rightarrow \text{num} \]

5 + 2 - 3 = \( (5+2) - 3 \) ?
5 \* 2/3 = \( (5 \* ?) / 3 \) ?
5 \* (2/3)

2 - 3 - 4 = \( 2 - (3-4) \) ?
(2-3)-4 = 2 - 1 - 4 = -5

By convention, - and / are left-associative, so

2 - 3 - 4 = (2-3)-4

left-associative
\[ E \rightarrow E' \oplus E \]
\[ E \rightarrow E' \]
\[ E' \rightarrow \text{num} \]

\( \oplus \) is a right associative operator

For example, the list constructor:

```
```

\[ E \rightarrow E' \oplus E' \]
\[ E \rightarrow E' \]
\[ E' \rightarrow \text{num} \]

Non-associative operators

E.g., \(<\) in Pascal is non-associative

\[ \text{ln } c , \ (3 < 4) < 5 \Rightarrow 1 < 5 \quad \text{true} \]

\( \uparrow \) is left associative
\[
E \rightarrow E + E' \\
E \rightarrow E - E' \\
E \rightarrow E' \\
E' \rightarrow \text{num}
\]

\[
E \rightarrow E + E' \\
E \rightarrow E' \oplus E \\
E \rightarrow E' \\
E' \rightarrow \text{num}
\]
\[
\begin{align*}
\text{Exp} & \rightarrow \text{Exp} + \text{Exp2} \\
\text{Exp} & \rightarrow \text{Exp} - \text{Exp2} \\
\text{Exp} & \rightarrow \text{Exp2} \\
\text{Exp2} & \rightarrow \text{Exp2} \times \text{Exp3} \\
\text{Exp2} & \rightarrow \text{Exp2} / \text{Exp3} \\
\text{Exp2} & \rightarrow \text{Exp3} \\
\text{Exp3} & \rightarrow \textbf{num} \\
\text{Exp3} & \rightarrow (\text{Exp})
\end{align*}
\]
\[
\begin{align*}
  \text{Stat} & \rightarrow \text{Stat2} ; \text{Stat} \\
  \text{Stat} & \rightarrow \text{Stat2} \\
  \text{Stat2} & \rightarrow \text{Matched} \\
  \text{Stat2} & \rightarrow \text{Unmatched} \\
  \text{Matched} & \rightarrow \text{if Exp then Matched else Matched} \\
  \text{Matched} & \rightarrow \text{id} := \text{Exp} \\
  \text{Unmatched} & \rightarrow \text{if Exp then Matched else Unmatched} \\
  \text{Unmatched} & \rightarrow \text{if Exp then Stat2}
\end{align*}
\]
The following grammar (for the language of balanced parentheses) is ambiguous:

\[ P \rightarrow \varepsilon \]
\[ P \rightarrow (P) \]
\[ P \rightarrow PP \]

Consider the string: \( ()()() \)

Note that the last production is both left and right recursive.
We can remove left recursion:

\[ P \rightarrow \varepsilon \]

\[ P \rightarrow (P) \]

\[ (())()()() \]

Remove the left recursion