Regular expressions vs Grammars

Regular expressions are not powerful enough to describe the syntax of programming languages.

A simple generalization can be made: Use names for regular expressions and allow theses names to appear in expressions (recursively).

expr → INT | expr '+' expr | '(' expr ')'
Grammar

A context-free grammar has four components, \( G = (N, T, P, S) \):
- \( N \) – finite set of nonterminal symbols
- \( T \) – finite set of terminal symbols (tokens)
- \( P \) – finite set of productions (rules)
- \( S \) – the start symbol (one of the nonterminals in \( N \)).

\( N \) and \( T \) are disjoint sets.

Each production has a left part that is a nonterminal symbol and a right part that is a regular expression with terminal and nonterminal symbols.

There are three operations: alternative (\( \mid \)), concatenation, and iteration (\( * \)) in order of increasing precedence.

Parentheses are used for grouping.

Derivation

Derivation of "while (k <= n) \{sum = sum + k; k = k + 1;\}"

\[ \text{statement} \Rightarrow \text{whileStmt} \Rightarrow \text{'while'} (\text{'(' expr ')' statement} \Rightarrow \text{'while'} (\text{'(' lessEqual ')'} \text{statement} \Rightarrow \text{'while'} (\text{'(' expr 'lessEqual expr ')'} \text{statement} \Rightarrow \text{'while'} (\text{'(' ID 'lessEqual expr ')'} \text{statement} \Rightarrow \text{'while'} (\text{'(' ID 'lessEqual ID ')'} \text{compoundStmt} \Rightarrow \ldots \]

All the strings in the derivation are called sentential forms.
The final string is called a sentence.

Parse tree

\[ \text{statement} \]

Derivation of strings

A string belongs to a language \( \mathcal{L}(G) \) if it can be derived from the start symbol. Derivation (härledning):
- Start with the start symbol
- Replace a nonterminal \( X \) using the right-hand side of a production for \( X \):
  - if there are alternatives (\( \mid \)), choose one
  - if there are iterations (\( * \)), repeat any number of times
- Continue replacing nonterminals in this way until there are only terminal symbols left
- The resulting string belongs to the language

\[ \text{while ( ID <= ID ) \{ ID = ID + ID ; ID = ID + INT ; \}} \]
Derivations

A derivation has the form

$$\gamma_0 \Rightarrow \gamma_1 \cdots \Rightarrow \gamma_n , \quad n \geq 0$$

where each $\gamma_i$ is a string of terminal and non-terminal symbols obtained from the previous one by the use of one production.

When there is such a derivation we write

$$\gamma_0 \Rightarrow^* \gamma_n$$

The language $\mathcal{L}(G)$

The language $\mathcal{L}(G)$, where $G = (N, T, P, S)$, is the set of all strings that can be derived from $S$ using the rules in $P$. $G$ generates $\mathcal{L}(G)$.

$$\mathcal{L}(G) = \{ w \in T^* | S \Rightarrow^* w \}$$

$T^*$ denotes the set of all strings with symbols from $T$. $G$ is finite but $\mathcal{L}(G)$ is usually an infinite set.

Leftmost derivation (vänsterhäradning)

Grammar $G_{expr}$

$$\text{expr} \rightarrow \text{expr} \ '+' \ \text{expr}$$
$$\text{expr} \rightarrow \text{expr} \ '*' \ \text{expr}$$
$$\text{expr} \rightarrow \text{INT}$$

The leftmost nonterminal is replaced in each derivation step.

Derive $\text{INT} + \text{INT} * \text{INT}$. Parse tree

Another leftmost derivation

Grammar $G_{expr}$

$$\text{expr} \rightarrow \text{expr} \ '+' \ \text{expr}$$
$$\text{expr} \rightarrow \text{expr} \ '*' \ \text{expr}$$
$$\text{expr} \rightarrow \text{INT}$$

Derive $\text{INT} + \text{INT} * \text{INT}$. Parse tree
Ambiguous grammar

A grammar is *ambiguous* if there is a string having more than one parse tree.

A grammar is *unambiguous* if there is no string having more than one parse tree.

If a grammar is ambiguous we should try to find a grammar generating the same language that is unambiguous. In the current case we would prefer a parse tree that respects operator precedences.

Equivalent grammars

Two grammars, $G_1$ and $G_2$, are equivalent if they generate the same language, i.e. each sentence that can be derived using one of the grammars, can also be derived using the other grammar.

\[ L(G_1) = L(G_2) \]

An equivalent grammar

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} + \text{term} \\
\text{expr} & \rightarrow \text{term} \\
\text{term} & \rightarrow \text{term} * \text{factor} \\
\text{term} & \rightarrow \text{factor} \\
\text{factor} & \rightarrow \text{INT}
\end{align*}
\]

Derive INT + INT * INT.

\[
\begin{align*}
\text{expr} & \Rightarrow \text{expr} + \text{term} \\
\text{term} & \Rightarrow \text{term} + \text{factor} \\
\text{factor} & \Rightarrow \text{INT}
\end{align*}
\]

Other equivalent grammars

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} + \text{term} | \text{term} \\
\text{term} & \rightarrow \text{term} * \text{factor} | \text{factor} \\
\text{factor} & \rightarrow \text{INT}
\end{align*}
\]

\[
\begin{align*}
\text{expr} & \rightarrow \text{term} + \text{term}^* \\
\text{term} & \rightarrow \text{factor} + \text{factor}^* \\
\text{factor} & \rightarrow \text{INT}
\end{align*}
\]
Different notations for CFGs

Canonical form (kanonisk form)
- The simplest notation: no alternatives nor iterations are permitted.
- Useful when proving formal properties of grammars and implementing parser generators.

BNF – Backus-Naur form
- Includes a shorthand for writing several productions for the same nonterminal in the same rule.

EBNF – Extended Backus-Naur Form
- Additional shorthand notations are introduced: regular expressions can be written in the right-hand side of a rule, e.g., (...)*. Gives very compact grammars.
- Standard notation for describing the syntax for programming languages.
- Used in JavaCC.

Transformation to canonical form – iteration
A grammar rule containing iteration on the top level has the form

\[ X \rightarrow \gamma_1 \gamma_2^* \gamma_3 \]

Where \( \gamma_i \) is a regular expression on the alphabet \( N \cup T \).

It can be transformed into

\[ X \rightarrow \gamma_1 N \gamma_3 \]
\[ N \rightarrow \gamma_2 N \]
\[ N \rightarrow \epsilon \]

Where \( N \) is a new nonterminal symbol.

Transformation to canonical form – alternatives
A grammar rule containing an alternative on the top level has the form

\[ X \rightarrow \gamma_1 \mid \gamma_2 \]

Can be transformed into

\[ X \rightarrow \gamma_1 \]
\[ X \rightarrow \gamma_2 \]

Example
 expr \rightarrow \text{term} (('+\mid-') \text{term})^* can be transformed into

\[ expr \rightarrow \text{term} \]
\[ expr \rightarrow \text{term} \]
**EBNF**

In EBNF (JavaCC) other operators are permitted

- $\gamma^+$ is equivalent to $\gamma \gamma^*$
- $\gamma?$ and $[\gamma]$ are equivalent to $\gamma | \epsilon$

**Example, dangling else**

statement $\rightarrow$ ‘if’ expr ‘then’ statement ['else' statement] ‘fi’

This production introduces ambiguity. Consider

if e1 then if e2 then s1 else s2

if e1 then {if e2 then s1} else s2
if e1 then {if e2 then s1 else s2}

**Dangling else – remedies**

statement $\rightarrow$ ‘if’ expr ‘then’ statement ['else' statement] ‘fi’

Grammar $G_{expr}$

- expr $\rightarrow$ expr ‘+’ expr
- expr $\rightarrow$ expr ‘*’ expr
- expr $\rightarrow$ INT

The rightmost nonterminal is replaced in each derivation step.

Derive INT + INT * INT.

```
expr \Rightarrow expr ' +' expr
expr ' +' expr ' * ' expr \Rightarrow
expr ' +' expr ' * ' INT \Rightarrow
expr ' +' INT ' * ' INT \Rightarrow
INT ' + ' INT ' * ' INT
```
LL and LR parsers

Parser algorithms will be the topic of the next lecture.

▶ an LL parser constructs a leftmost derivation
▶ an LR parser constructs a rightmost derivation

Is \( L(G_{Stmt}) \) too large?

statement \( \rightarrow \) whileStmt | assignment | compoundStmt
whileStmt \( \rightarrow \) 'while' '(' expr ')' statement
assignment \( \rightarrow \) ID '=' expr ';'
compoundStmt \( \rightarrow \) '{' statement* '}'
expr \( \rightarrow \) lessEqual | add | ID | INT
lessEqual \( \rightarrow \) expr '<=' expr
add \( \rightarrow \) expr '+' expr

\( G_{Stmt} \) allows statements that would not be legal in Java.

\[
\begin{align*}
\text{while (a+b) \{ ... \}} \\
x = x <= y;
\end{align*}
\]

How can this be handled?

What is meant by context-free?

▶ A canonical grammar is context-free if every production has the form \( A \rightarrow \gamma \), where \( A \) is a nonterminal and \( \gamma \) is a string of terminals and nonterminals.
▶ A production on the form \( \alpha A \beta \rightarrow \alpha \gamma \beta \), where \( \alpha \) and \( \beta \) are strings of terminals and nonterminals, is context-sensitive. \( A \) may be replaced by \( \gamma \) in the context \( \alpha \cdot \beta \).

Comparing CFGs to REs

<table>
<thead>
<tr>
<th></th>
<th>CFG</th>
<th>RE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphabet</td>
<td>terminal symbols (tokens)</td>
<td>characters</td>
</tr>
<tr>
<td>Language</td>
<td>sentences (sequences of tokens)</td>
<td>strings (sequences of characters)</td>
</tr>
<tr>
<td>Used for describing</td>
<td>syntax for programming languages</td>
<td>tokens</td>
</tr>
<tr>
<td>Expressive power</td>
<td>recursion</td>
<td>iteration</td>
</tr>
<tr>
<td>Recognizer</td>
<td>nondeterministic pushdown automaton (NFA with stack)</td>
<td>deterministic finite automata (DFA)</td>
</tr>
</tbody>
</table>
The Chomsky hierarchy

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Rule patterns</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular</td>
<td>$X \to aY$ or $X \to \epsilon$</td>
<td>3</td>
</tr>
<tr>
<td>context free</td>
<td>$X \to \gamma$</td>
<td>2</td>
</tr>
<tr>
<td>context sensitive</td>
<td>$\alpha X \beta \to \alpha \gamma \beta$</td>
<td>1</td>
</tr>
<tr>
<td>arbitrary</td>
<td>$\gamma \to \delta$</td>
<td>0</td>
</tr>
</tbody>
</table>

Regular grammars have the same describing power as regular expressions.
Type 2 and 3 are of practical use in compiler construction. The others are only of theoretical interest.

Unsolvable problems

▶ Given two context-free grammars, $G_1$ and $G_2$. Is $L((G_1)) = L((G_2))$?
▶ Given a context-free grammar, $G$. Is $G$ ambiguous?

Summary questions

▶ Define a small example language with a context-free grammar.
▶ What is a nonterminal symbol? A terminal symbol? A production? A start symbol?
▶ Given a grammar $G$, what is meant by the language $L(G)$?
▶ What is a derivation? A leftmost derivation?
▶ When are two grammars equivalent?
▶ What is the difference between a context-free grammar and a context-sensitive grammar?
▶ When should we use canonical form, and when EBNF?
▶ Translate an EBNF grammar to canonical form.
▶ Explain why context-free grammars are more powerful than regular expressions.
▶ Explain why tokens are usually defined by regular expressions rather than context-free grammars.
### Some terms

<table>
<thead>
<tr>
<th>abbr.</th>
<th>English</th>
<th>Svenska</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST</td>
<td>abstract syntax tree</td>
<td>abstrakt syntaxträd</td>
</tr>
<tr>
<td></td>
<td>terminal symbol</td>
<td>terminalsymbol</td>
</tr>
<tr>
<td></td>
<td>nonterminal symbol</td>
<td>icketerminalsymbol</td>
</tr>
<tr>
<td>CFG</td>
<td>context-free grammar</td>
<td>kontextfri grammatik</td>
</tr>
<tr>
<td></td>
<td>context-sensitive grammar</td>
<td>kontextberoende grammatik</td>
</tr>
<tr>
<td></td>
<td>derivation</td>
<td>härledning</td>
</tr>
<tr>
<td></td>
<td>canonical form</td>
<td>kanonisk form</td>
</tr>
<tr>
<td>BNF</td>
<td>Backus-Naur form</td>
<td></td>
</tr>
<tr>
<td>EBNF</td>
<td>Extended BNF</td>
<td>utvidgad BNF</td>
</tr>
<tr>
<td>RE</td>
<td>regular expression</td>
<td>reguljärt uttryck</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>the empty string</td>
<td>tomma strängen</td>
</tr>
<tr>
<td>DFA</td>
<td>deterministic finite automaton</td>
<td>deterministisk ändlig automat</td>
</tr>
</tbody>
</table>

### Readings

- F3: Context free grammars, derivations, ambiguity, EBNF
  Appel, chapter 3-3.1.
- F4: Predictive parsing. Recursive descent. LL grammars and parsing Left recursion and factorization.
  Appel, chapter 3.2
- Seminar 2. Grammars.
- Programming assignment 1. Build a scanner. Instructions is on the web.