**Lexical Analysis**

**Chapter 4:**

**Turning NFAs deterministic**

---

**Berry-Sethi Approach**

### ... for example:

![Diagram]

**Remarks:**
- This construction is known as Berry-Sethi- or Glushkov-construction.
- It is used for XML to define Content Models.
- The result may not be what we had in mind...
The expected outcome:

Remarks:
- Ingoing edges do not necessarily have the same label here
- But Berry-Sethi is rather directly constructed
- Anyway, we need a deterministic technique

⇒ Powerset-Construction

Powerset Construction

... for example:

Powerset Construction

... for example:

Powerset Construction

... for example:
Powerset Construction

... for example:

```
0
\[\begin{array}{c}
0 & 2 & 3 \\
\end{array}\]
\[\begin{array}{c}
1 & 4 \\
\end{array}\]
```

Theorem:

For every non-deterministic automaton \(A = (Q, \Sigma, \delta, I, F)\) we can compute a deterministic automaton \(\mathcal{P}(A)\) with

\[\mathcal{L}(A) = \mathcal{L}(\mathcal{P}(A))\]

Construction:

**States:** Powersons of \(Q'\)

**Start state:** \(\emptyset\)

**Final states:** \(Q' \subseteq Q \mid Q' \neq \emptyset\)

**Transitions:** \(\delta'(Q', a) = \{q \in Q \mid \exists p \in Q' : (p, a, q) \in \delta\}\).
**Theorem:**
For every non-deterministic automaton \( A = (Q, \Sigma, \delta, I, F) \) we can compute a deterministic automaton \( \mathcal{P}(A) \) with
\[
\mathcal{L}(A) = \mathcal{L}(\mathcal{P}(A))
\]

---

**Bummer!**
There are exponentially many powersets of \( Q \)

- Idea: Consider only contributing powersets. Starting with the set \( Q_p = \{I\} \) we only add further states by need ...
- i.e., whenever we can reach them from a state in \( Q_p \)
- Even though, the resulting automaton can become enormously huge ...
- which is (sort of) not happening in practice

---

**Bummer!**
There are exponentially many powersets of \( Q \)

- Idea: Consider only contributing powersets. Starting with the set \( Q_p = \{I\} \) we only add further states by need ...
- i.e., whenever we can reach them from a state in \( Q_p \)
- Even though, the resulting automaton can become enormously huge ...
- which is (sort of) not happening in practice

---

**... for example:**

---

**Bummer!**
There are exponentially many powersets of \( Q \)

- Idea: Consider only contributing powersets. Starting with the set \( Q_p = \{I\} \) we only add further states by need ...
- i.e., whenever we can reach them from a state in \( Q_p \)
- Even though, the resulting automaton can become enormously huge ...
- which is (sort of) not happening in practice

- Therefore, in tools like grep a regular expression’s DFA is never created!
- Instead, only the sets, directly necessary for interpreting the input are generated while processing the input
Powerset Construction

... for example:

a b a b

Powerset Construction

... for example:

a b a b

Powerset Construction

... for example:

a b a b

Powerset Construction

... for example:

a b a b
Remarks:

- For an input sequence of length $n$, maximally $O(n)$ sets are generated.
- Once a set/edge of the DFA is generated, they are stored within a hash-table.
- Before generating a new transition, we check this table for already existing edges with the desired label.

Summary:

**Theorem:**
For each regular expression $e$ we can compute a deterministic automaton $A = \mathcal{P}(A_e)$ with

$$\mathcal{L}(A) = [e]$$

Scanner design

Input (simplified):

- $e_1$, $e_2$, ..., $e_k$
- $\{\text{action}_1\}$, $\{\text{action}_2\}$, ..., $\{\text{action}_k\}$

Special information:
Scanner design

Input (simplified): a set of rules:

\[ \begin{align*}
  e_1 & \quad \{ \text{action}_1 \} \\
  e_2 & \quad \{ \text{action}_2 \} \\
  \vdots \\
  e_k & \quad \{ \text{action}_k \}
\end{align*} \]

Output: a program,

... reading a maximal prefix \( w \) from the input, that satisfies \( e_1 | \ldots | e_i \);

... determining the minimal \( i \), such that \( w \in [e_i] \);

... executing \( \text{action}_i \) for \( w \).

Implementation:

Idea:

- Create the DFA \( \mathcal{P}(A_e) = (Q, \Sigma, \delta, q_0, F) \) for the expression \( e = (e_1 \mid \ldots \mid e_k) \);
- Define the sets:
  \[ \begin{align*}
    F_1 & = \{ q \in F \mid q \cap \text{last}[e_1] \neq \emptyset \} \\
    F_2 & = \{ q \in (F \setminus F_1) \mid q \cap \text{last}[e_2] \neq \emptyset \} \\
    \vdots \\
    F_k & = \{ q \in (F \setminus (F_1 \cup \ldots \cup F_{k-1})) \mid q \cap \text{last}[e_k] \neq \emptyset \}
  \end{align*} \]
- For input \( w \) we find: \( \delta^*(q_0, w) \in F_i \) iff the scanner must execute \( \text{action}_i \) for \( w \).

Implementation (cont'd):

- The scanner manages two pointers \( \langle A, B \rangle \) and the related states \( \langle q_A, q_B \rangle \).
- Pointer \( A \) points to the last position in the input, after which a state \( q_A \in F \) was reached;
- Pointer \( B \) tracks the current position.

```c
stdout.writeLatin("Hello")
```


Implementation:

Idea (cont'd):
- The current state being $q_B = \emptyset$, we consume input up to position $A$ and reset:
  
  \[
  B := A; \quad A := \bot; \\
  q_B := q_0; \quad q_A := \bot
  \]

```
writeln ("Hello"');
```

Implementation:

Idea (cont'd):
- The current state being $q_B = \emptyset$, we consume input up to position $A$ and reset:
  
  \[
  B := A; \quad A := \bot; \\
  q_B := q_0; \quad q_A := \bot
  \]

```
writeln ("Hello"');
```

Implementation:

Idea (cont'd):
- The current state being $q_B = \emptyset$, we consume input up to position $A$ and reset:
  
  \[
  B := A; \quad A := \bot; \\
  q_B := q_0; \quad q_A := \bot
  \]

```
writeln ("Hello"');
```

Extension:

States

- Now and then, it is handy to differentiate between particular scanner states.
- In different states, we want to recognize different token classes with different precedences.
- Depending on the consumed input, the scanner state can be changed.

Example:

Comments

Within a comment, identifiers, constants, comments, ... are ignored
Input (generalized): a set of rules:

\[
\begin{align*}
\text{state} & \{ 
\text{e}_1 & \{ \text{action}_1 & \text{yybegin(state}_1) \}; \\
\text{e}_2 & \{ \text{action}_2 & \text{yybegin(state}_2) \}; \\
\ldots & \\
\text{e}_k & \{ \text{action}_k & \text{yybegin(state}_k) \}; \\
\} \\
\text{The statement } & \text{yybegin (state}_i); \text{ resets the} \\
\text{current state to } & \text{state}_i. \\
\text{The start state is called (e.g. flex JFlex) } & \text{YYINITIAL.}
\end{align*}
\]

... for example:

\[
\begin{align*}
\text{YYINITIAL} & \{ \\
\text{COMMENT} & \{ \\
\text{"/x"} & \{ \text{yybegin(COMMENT);} \}; \\
\text{"*"} & \{ \text{yybegin(YYINITIAL);} \}; \\
\ldots & \{ \text{yybegin(YYINITIAL);} \}; \\
\} \} \\
\text{yybegin (STATE);} & \text{ switches between different scanners.}
\end{align*}
\]

Remarks:

- "." matches all characters different from \"\n\".
- For every state we generate the scanner respectively.
- Comments might be directly implemented as (admittedly overly complex) token-class.
- Scanner-states are especially handy for implementing preprocessors, expanding special fragments in regular programs.

Topic:

Syntactic Analysis

- Syntactic analysis tries to integrate Tokens into larger program units.
Syntactic Analysis

Token-Stream → Parser → Syntaxtree

- Syntactic analysis tries to integrate Tokens into larger program units.
- Such units may possibly be:
  - Expressions;
  - Statements;
  - Conditional branches;
  - Loops;

Discussion:

In general, parsers are not developed by hand, but generated from a specification:

Specification → Generator → Parser

Chapter 1:
Basics of contextfree Grammars

Specification of the hierarchical structure: contextfree grammars

Generated implementation: Pushdown automata + X
Basics: Context-free Grammars

- Programs of programming languages can have arbitrary numbers of tokens, but only finitely many Token-classes.
- This is why we choose the set of Token-classes to be the finite alphabet of terminals $T$.
- The nested structure of program components can be described elegantly via context-free grammars...

**Definition:**
A context-free grammar (CFG) is a 4-tuple $G = (N, T, P, S)$ with:

- $N$ the set of nonterminals,
- $T$ the set of terminals,
- $P$ the set of productions or rules, and
- $S \in N$ the start symbol.

Conventions

The rules of context-free grammars take the following form:

$$A \rightarrow \alpha \quad \text{with} \quad A \in N, \quad \alpha \in (N \cup T)^*$$

... for example:

$$S \rightarrow aSb$$
$$S \rightarrow \epsilon$$

Specified language: $\{a^n b^n | n \geq 0\}$

Conventions

The rules of context-free grammars take the following form:

$$A \rightarrow \alpha \quad \text{with} \quad A \in N, \quad \alpha \in (N \cup T)^*$$

... for example:

$$S \rightarrow aSb$$
$$S \rightarrow \epsilon$$

Specified language: $\{a^n b^n | n \geq 0\}$

Conventions:

In examples, we specify nonterminals and terminals in general implicitly:

- nonterminals are: $A, B, C, ...$ (exp), (stmt) ...
- terminals are: $a, b, c, ...$ int, name, ...
... further examples:

\[
\begin{align*}
S & \rightarrow \text{stmt} \\
\text{stmt} & \rightarrow \text{if} \mid \text{while} \mid \text{rexp} \\
\text{if} & \rightarrow \text{if} (\text{rexp}) \text{stmt} \text{else} \text{stmt} \\
\text{while} & \rightarrow \text{while} (\text{rexp}) \text{stmt} \\
\text{rexp} & \rightarrow \text{int} \mid \text{lexp} \mid \text{rexp} = \text{rexp} \\
\text{lexp} & \rightarrow \text{name} \mid ... \\
\end{align*}
\]

Further conventions:

- For every nonterminal, we collect the right hand sides of rules and list them together.
- The \( j \)-th rule for \( A \) can be identified via the pair \((A, j)\) (with \( j \geq 0 \)).

Further grammars:

\[
\begin{align*}
E & \rightarrow E + E \mid E \ast E \mid (E) \mid \text{name} \mid \text{int} \\
E & \rightarrow E + T \mid T \\
T & \rightarrow T \ast F \mid F \\
F & \rightarrow (E) \mid \text{name} \mid \text{int} \\
\end{align*}
\]

Both grammars describe the same language.
Grammars are term rewriting systems. The rules offer feasible rewriting steps. A sequence of such rewriting steps $\alpha_0 \rightarrow \ldots \rightarrow \alpha_m$ is called derivation.

... for example:

$$E \rightarrow E + T$$

$\rightarrow T + T$

$\rightarrow T * E + T$

$\rightarrow T * \text{int} + T$

---

Grammars are term rewriting systems. The rules offer feasible rewriting steps. A sequence of such rewriting steps $\alpha_0 \rightarrow \ldots \rightarrow \alpha_m$ is called derivation.

... for example:

$$E \rightarrow E + T$$

$\rightarrow T + T$

$\rightarrow T * E + T$

$\rightarrow T * \text{int} + T$

$\rightarrow \text{name} * \text{int} + T$

$\rightarrow \text{name} * \text{int} + F$

$\rightarrow \text{name} * \text{int} + \text{int}$

---

**Definition**

A derivation $\rightarrow$ is a relation on words over $N \cup T$, with

$$\alpha \rightarrow \alpha' \iff \alpha = \alpha_1 A \alpha_2 \land \alpha' = \alpha_1 B \alpha_2 \text{ for an } A \rightarrow B \in P$$
Derivation

Grammars are term rewriting systems. The rules offer feasible rewriting steps. A sequence of such rewriting steps $\alpha_0 \rightarrow \ldots \rightarrow \alpha_m$ is called derivation.

... for example:

$E \rightarrow E + T$
$\rightarrow T + T$
$\rightarrow T * F + T$
$\rightarrow T * int + T$
$\rightarrow F * int + T$
$\rightarrow name * int + T$
$\rightarrow name * int + F$
$\rightarrow name * int + int$

Definition

A derivation $\rightarrow$ is a relation on words over $N \cup T$, with

$\alpha \rightarrow \alpha'$ iff $\alpha = \alpha_1 A \alpha_2 \land \alpha' = \alpha_1 \beta \alpha_2$ for an $A \rightarrow \beta \in P$

The reflexive and transitive closure of $\rightarrow$ is denoted as: $\rightarrow^*$

Derivation

Remarks:

- The relation $\rightarrow$ depends on the grammar
- In each step of a derivation, we may choose:
  - a spot, determining where we will rewrite.
  - a rule, determining how we will rewrite.
- The language, specified by $G$ is:

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

Attention:

The order, in which disjunct fragments are rewritten is not relevant.

Derivation tree

Derivations of a symbol are represented as derivation tree:

... for example:

A derivation tree for $A \in N$:

inner nodes: rule applications
root: rule application for $A$
leaves: terminals or $\epsilon$
Special Derivations

Attention:
In contrast to arbitrary derivations, we find special ones, always rewriting the leftmost (or rather rightmost) occurrence of a nonterminal.

- These are called leftmost (or rather rightmost) derivations and are denoted with the index $L$ (or $R$ respectfully).
- Leftmost (or rightmost) derivations correspond to a left-to-right (or right-to-left) preorder-DFS-traversal of the derivation tree.
- Reverse rightmost derivations correspond to a left-to-right postorder-DFS-traversal of the derivation tree.

**Leftmost derivation:**
$(E, 0) (E, 1) (T, 0) (T, 1) (F, 1) (F, 2) (T, 1) (F, 2)$

**Rightmost derivation:**
$(E, 0) (T, 1) (F, 2) (E, 1) (T, 0) (F, 2) (T, 1) (F, 1)$

**Reverse rightmost derivation:**
$(F, 1) (T, 1) (F, 2) (T, 0) (E, 1) (F, 2) (T, 1) (E, 0)$

... for example:
Unique grammars

The concatenation of leaves of a derivation tree \( t \) are often called \( \text{yield}(t) \).

... for example:

```
  E 0
   +
  E 1
 /  |
T 0  F 2
 / | |
T 1 F 2
 / | |
F 1 name
  |
  int
```

gives rise to the concatenation: \( \text{name} = \text{int} + \text{int} \).

Conclusion:

- A derivation tree represents a possible hierarchical structure of a word.
- For programming languages, only those grammars with a unique structure are of interest.
- Derivation trees are one-to-one corresponding with leftmost derivations as well as (reverse) rightmost derivations.
- Leftmost derivations correspond to a top-down reconstruction of the syntax tree.
- Reverse rightmost derivations correspond to a bottom-up reconstruction of the syntax tree.

Unique grammars

**Definition:**

Grammar \( G \) is called unique, if for every \( w \in T^* \) there is maximally one derivation tree \( t \) of \( S \) with \( \text{yield}(t) = w \).

... in our example:

```
E \rightarrow E + E^0 | E \ast E^1 | (E)^2 | \text{name}^3 | \text{int}^4
E \rightarrow E + T^0 | T^1
T \rightarrow T \ast F^0 | F^1
F \rightarrow (E)^0 | \text{name}^1 | \text{int}^2
```

The first one is ambiguous, the second one is unique.

Syntactic Analysis

Chapter 2: Basics of pushdown automata