The expected outcome:

![Diagram](image1)

Remarks:

- Ideal automaton would be even more compact (→ Antimirov automata, Follow Automata)
- But Berry-Sethi is rather directly constructed
- Anyway, we need a deterministic version

Powerset Construction

... for example:

![Diagram](image2)
Power set Construction

... for example:

```
1 0 2 3
```

```
4 0 2 3
```

Power set Construction

... for example:
Power Set Construction

**Theorem:**
For every non-deterministic automaton $A = (Q, \Sigma, \delta, I, F)$ we can compute a deterministic automaton $P(A)$ with

$$\mathcal{L}(A) = \mathcal{L}(P(A))$$

**Construction:**

- **States:** Powdersets of $Q$;
- **Start state:** $I$;
- **Final states:** $\{Q' \subseteq Q | Q' \cap F \neq \emptyset\}$;
- **Transitions:**

$$(P(Q'), \delta) = \{ x \in Q | \exists p \in Q' \quad p \cdot x \in \delta \}.$$ 

**Observation:**
There are exponentially many powdersets of $Q$

- **Idea:** Consider only contributing powdersets. 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 $Q_P$
- However, the resulting automaton can become enormously huge ...
  which is (sort of) not happening in practice

- **Idea:** Consider only contributing powdersets. 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 $Q_P$
- However, 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
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): a set of rules:

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

Lexical Analysis

Chapter 5: Scanner design
Scanner design

Input (simplified): a set of rules:

\[ e_1 \quad \{ \text{action}_1 \} \]
\[ e_2 \quad \{ \text{action}_2 \} \]
\[ \cdots \quad \{ \text{action}_k \} \]
\[ e_k \]

Output: a program,

\[ \text{id} \]
\[ \text{if} \]

... reading a maximal prefix \( w \) from the input, that satisfies \( e_1 \mid \ldots \mid e_k \);
... determining the minimal \( i \), such that \( w \in [e_i] \);
... executing \( \text{action}_i \) for \( w \).

Implementation:

Idea:

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

Scanner design

Input (simplified): a set of rules:

\[ e_1 \quad \{ \text{action}_1 \} \]
\[ e_2 \quad \{ \text{action}_2 \} \]
\[ \cdots \quad \{ \text{action}_k \} \]
\[ e_k \]

Output: a program,

\[ \text{id} \]
\[ \text{if} \]

... reading a maximal prefix \( w \) from the input, that satisfies \( e_1 \mid \ldots \mid e_k \);
... determining the minimal \( i \), such that \( w \in [e_i] \);
... executing \( \text{action}_i \) for \( w \).

Implementation:

Idea (cont’d):

- The scanner manages two pointers \( (A, B) \) and the related states \( (q_A, q_B) \);
- 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.

stdout. writein (" Hallo ");

\[
\begin{align*}
\text{stdout} & . \ \text{writein} \ (" \text{Hallo} \") ; \\
A & \quad \text{B} \\
& \quad \\
\end{align*}
\]
Implementation:

Idea (cont’d):
- The scanner manages two pointers \(A, B\) and the related states \((q_A, q_B)\).
- 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.

```
writeln("Hello"):  
```

Implementation:

Idea (cont’d):
- The current state being \(q_B = \emptyset\), we consume input up to position \(A\) and reset:
  - \(B := A;\)
  - \(A := \perp;\)
  - \(q_B := q_0;\)
  - \(q_A := \perp;\)

```
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:

\[
\text{(state)\{ \\
\quad x_1 \quad \{ \text{action}_1 \quad \text{yybegin(state}_1); \} \\
\quad x_2 \quad \{ \text{action}_2 \quad \text{yybegin(state}_2); \} \\
\quad \ldots \\
\quad x_k \quad \{ \text{action}_k \quad \text{yybegin(state}_k); \} \\
\}\}
\]

- The statement `yybegin(state_i);` resets the current state to `state_i`.
- The start state is called (e.g., `flex JFlex`) **YYINITIAL**.

... for example:

\[
\text{(YYINITIAL) \{ \\
\quad "/\" \quad \{ \text{yybegin} \text{COMMENT}; \} \\
\quad "\\*="/\" \quad \{ \text{yybegin} \text{YYINITIAL}; \} \\
\quad . \quad \{ \text{n} \} \\
\}\}
\]

### Syntactic Analysis

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

### Topic:

Syntactic Analysis

- Such units may possibly be:
  -> Expressions;
  -> Statements;
  -> Conditional branches;
  -> loops; ...
Discussion:

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

![Diagram]

Specification  $\rightarrow$ Generator  $\rightarrow$ Parser

Discussion:

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

$$E \rightarrow E[\text{op}]E$$

![Diagram]

Specification of the hierarchical structure: context-free grammars
Generated implementation: Pushdown automata + $X$

Syntactic Analysis

Chapter 1:
Basics of Context-free Grammars

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...
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: Context-Free Grammar**

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

![Noam Chomsky](image1.png)  ![John Backus](image2.png)

Conventions

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

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

... for example:

$$S \rightarrow aSa$$
$$S \rightarrow \varepsilon$$

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

Conventions

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

- nonterminals are: $A, B, C, ..., \text{[exp]}, \text{[stmt]}, ...$
- terminals are: $a, b, c, ..., \text{[int]}, \text{[name]}, ...$

... for example:

$$S \rightarrow aSb$$
$$S \rightarrow \varepsilon$$

Specified language: $\{a^n b^n \mid n \geq 0\}$
... a practical example:

\[
\begin{align*}
S & \rightarrow \text{stmt} \\
\text{stmt} & \rightarrow (\text{if}) \mid (\text{while}) \mid (\text{exp})_1 \\
\text{if} & \rightarrow \text{if} ( (\text{exp}) ) (\text{stmt}) \text{ else } (\text{stmt}) \\
\text{while} & \rightarrow \text{while} ( (\text{exp}) ) (\text{stmt}) \\
\text{exp} & \rightarrow \text{int} \mid (\text{exp}) \mid (\text{exp}) = (\text{exp}) \mid \cdots \\
\text{exp} & \rightarrow \text{name} \mid \cdots
\end{align*}
\]

More 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 \((A, j)\).

Pair of grammars:

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

Both grammars describe the same language

Pair of grammars:

\[
\begin{align*}
E & \rightarrow E + E^0 \mid E * E^1 \mid (E)^2 \mid \text{name}^3 \mid \text{int}^4 \\
E & \rightarrow E + T^0 \mid T^1 \\
T & \rightarrow T * F^0 \mid F^1 \\
F & \rightarrow (E)^0 \mid \text{name}^1 \mid \text{int}^2
\end{align*}
\]

Both grammars describe the same language
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: \( E \)

---

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: \( E \rightarrow E + T \)

---

Pair of grammars:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>name</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \rightarrow E + E )</td>
<td>( E * E )</td>
<td>( ( E ) )</td>
<td>name</td>
<td>int</td>
</tr>
<tr>
<td>( E \rightarrow E + T )</td>
<td>( T )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( T \rightarrow T + T )</td>
<td>( T )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( F \rightarrow ( E ) )</td>
<td>name</td>
<td>int</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Both grammars describe the same language
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:

$$
E \rightarrow E + T \\
\rightarrow T + T \\
\rightarrow T \star F + T
$$

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:

$$
E \rightarrow E + T \\
\rightarrow T + T \\
\rightarrow T \star E + T \\
\rightarrow T \star \text{int} + T
$$

Definition

The derivation relation $\rightarrow$ 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.