The Lexical Analysis

Topic:
Lexical Analysis

The Lexical Analysis - Siever

Classified tokens allow for further pre-processing:
- Dropping irrelevant fragments e.g. Spacing, Comments,...
- Collecting Pragmas, i.e. directives for the compiler, often implementation dependent, directed at the code generation process, e.g. OpenMP-Statements;
- Replacing of Tokens of particular classes with their meaning / internal representation, e.g.
  - Constants;
  - Names: typically managed centrally in a Symbol-table, maybe compared to reserved terms (if not already done by the scanner) and possibly replaced with an index or internal format (→ Name Mangling).

The Lexical Analysis - Generating:

Discussion:
Scanner and Siever are often combined into a single component, mostly by providing appropriate callback actions in the event that the scanner detects a token.
Scanners are mostly not written manually, but generated from a specification.

Regular Expressions

Definition Regular Expressions

The set \( \mathcal{E} \) of (non-empty) regular expressions is the smallest set \( \mathcal{E} \) with:
- \( x \in \mathcal{E} \) [a new symbol not from \( \Sigma \)];
- \( a \in \mathcal{E} \) for all \( a \in \Sigma \);
- \( e_1 \cdot e_2 \in \mathcal{E} \), \( e_1, e_2 \in \mathcal{E} \).

Regular Expressions

Specification needs Semantics

For \( x \in \Sigma^* \) we define the specified language \( L(x) \subseteq \Sigma^* \) inductively by:
- \( \varepsilon \in L(x) \)
- \( [x] = \{ \varepsilon \} \)
- \( [x^+] = \{ x \} \cup [x^+] \)
- \( [x^*] = \{ \varepsilon \} \cup [x^+] \)
- \( [x \cdot y] = \{ x \} \cdot \{ y \} \)

Regular Expressions

Attention:
- We distinguish between characters \( a, b, \ldots \) and Meta-symbols \( (\ldots) \).
- To avoid (ugly) parantheses, we make use of Operator-Precedences:
  - Addition \( + \) is right-associative:
  - and omit “\( \cdot \)".

Regular Expressions

... Example:
- \( a \cdot b^* \cdot a \)
- \( (a | b) \cdot (a | b) \)

Attention:
- Specifiers-languages offer additional constructs:
  - \( e? \equiv (\varepsilon | e) \)
  - \( e+ \equiv (e \cdot e^*) \)
- and omit “\( \varepsilon \)"

Regular Expressions

Keep in Mind:
- The operators \( (\ldots), (\ldots) \) are interpreted in the context of sets of words:
  - \( L(x) \equiv \{ w \in L \mid \exists w \in L \} \)
  - \( L(x)^* \equiv \{ w_1 \cdot w_2 \mid w_1 \in L, w_2 \in L \} \)
- Regular expressions are internally represented as annotated ranked trees:
  - \( (ab)^* \)

Inner nodes: Operator-applications;
Leaves: particular symbols or \( \varepsilon \).
Lexical Analysis

Chapter 2:
Basics: Finite Automata

Finite Automata

Example:

\[ \begin{array}{c}
A = (Q, \Sigma, \delta, I, F) \\
Q = \{q_1, q_2, q_3\} \\
\Sigma = \{a, b, c\} \\
I = \emptyset \\
F = \{q_3\} \\
\delta(q_1, a) = q_2 \\
\delta(q_2, b) = q_3 \\
\delta(q_3, c) = q_1
\end{array} \]

Nodes: States; Edges: Transitions; Lables: Consumed input;

Finite Automata

- Computations are paths in the graph.
- Accepting computations lead from \( I \) to \( F \).
- An accepted word is the sequence of labels along an accepting computation.

Lexical Analysis

Chapter 3:
Converting Regular Expressions to NFAs

Regular Expressions

Example: Identifiers in Java:

- \( le = \{a-zA-Z_\$\} \)
- \( di = \{0-9\} \)
- \( Id = \{le\} (\{le\} | \{di\})^* \)

Remarks:
- "\( le \)" and "\( di \)" are token classes.
- Defined Names are enclosed in "\{\}".
- Symbols are distinguished from Meta-symbols via "\"",

Finite Automata

- Thompson's automata characterizes for a path between the states \( p \) and \( q \).
- The set of all accepting words, i.e. \( A \)'s accepted language can be described compactly as:

\[ L(A) = \{ w \in \Sigma^* | \exists i \in I, f \in F : (i, w, f) \in \delta^* \} \]

In Linear Time from Regular Expressions to NFAs

Thompson's Algorithm

- Produces \( O(n) \) states for regular expressions of length \( n \).

A formal approach to Thompson's Algorithm

Berry-Sethi Approach

- Produces exactly \( n+1 \) states without \(-\)transitions and demonstrates \( \rightarrow \) Equality Systems and \( \rightarrow \) Attribute Grammars

Idea:
- An automaton covering the syntax tree of a regular expression \( \alpha \) tracks (conceptually via markers \( \star \)), which subexpressions \( \gamma \) are reachable consuming the rest of input \( \alpha \).
- Marks: Exit or start point of automaton for this subexpression.
- Edges: For each layer of subexpression are modelled after Thompson's automata

\[ w = \]

... for example:

Berry-Sethi Algorithm

Glushkov Automaton

- Produces exactly \( n+1 \) states without -transitions and demonstrates \( \rightarrow \) Equality Systems and \( \rightarrow \) Attribute Grammars
In general:
- Input is only consumed at the leaves.
- Navigating the tree does not consume input \(\rightarrow\) transitions
- For a formal construction we need identifiers for states.
- For a node \(i\)'s identifier we take the subexpression, corresponding to the subtree dominated by \(i\).

There are possibly identical subexpressions in one regular expression.

### Berry-Sethi Approach

#### 1st step

- Implementation:
  - DFS post-order traversal for leaves \(r \equiv i \cdot x\) we find
  
  \[
  \text{empty}[r] = \begin{cases} 
  \top & \text{if } x = \epsilon \\
  \bot & \text{otherwise}
  \end{cases}
  \]

  Otherwise:
  
  \[
  \begin{align*}
  \text{empty}[r_1 \cup r_2] &= \text{empty}[r_1] \lor \text{empty}[r_2] \\
  \text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \land \text{empty}[r_2] \\
  \text{empty}[r_1?] &= \text{empty}[r_1] \\
  \text{empty}[r_1^*] &= \top
  \end{align*}
  \]

#### 2nd step

The may-set of first reached read states: The set of read states, that may be reached from \(\epsilon\) (i.e. while descending into \(r\)) via sequences of \(\cdot\)-transitions:

\[
\text{first}[r] = \{ i \mid r \cdot \epsilon, r \}
\]

For example:

\[
\begin{array}{c|c}
\text{transitions} & t \\
\hline
\{ 0 \} & \{ 1 \}
\end{array}
\]

#### 3rd step

The may-set of next read states: The set of read states reached after reading \(r\), that may be reached next via sequences of \(\cdot\)-transitions.

\[
\text{next}[r] = \{ i \mid r \cdot i \}
\]

For example:

\[
\begin{array}{c|c}
\text{transitions} & t \\
\hline
\{ 0 \} & \{ 1 \}
\end{array}
\]

### Berry-Sethi Approach (naive version)

#### Construction (naive version):

- States: \(e\), \(r\) with \(r\) nodes of \(\cdot\).
- Start state: \(e\).
- Final state: \(r\).
- Transitions: for leaves \(r \equiv i \cdot x\) we require: \(x, \epsilon, x\).

The leftover transitions are:

- For leaves \(r \equiv i \cdot x\) we find \(\text{first}[r] = \{ i \cdot x \neq \epsilon \}\)

#### Berry-Sethi Approach: 2nd step

Implementation:
- DFS post-order traversal

For leaves \(r \equiv i \cdot x\) we find

\[
\text{empty}[r] = \begin{cases} 
\top & \text{if } x = \epsilon \\
\bot & \text{otherwise}
\end{cases}
\]

Otherwise:

\[
\begin{align*}
\text{empty}[r_1 \cup r_2] &= \text{empty}[r_1] \lor \text{empty}[r_2] \\
\text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \land \text{empty}[r_2] \\
\text{empty}[r_1?] &= \text{empty}[r_1] \\
\text{empty}[r_1^*] &= \top
\end{align*}
\]

### Berry-Sethi Approach: 1st step

Implementation:
- DFS post-order traversal

For leaves \(r \equiv i \cdot x\) we find

\[
\text{empty}[r] = \begin{cases} 
\top & \text{if } x = \epsilon \\
\bot & \text{otherwise}
\end{cases}
\]

Otherwise:

\[
\begin{align*}
\text{empty}[r_1 \cup r_2] &= \text{empty}[r_1] \lor \text{empty}[r_2] \\
\text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \land \text{empty}[r_2] \\
\text{empty}[r_1?] &= \text{empty}[r_1] \\
\text{empty}[r_1^*] &= \top
\end{align*}
\]

### Berry-Sethi Approach: 2nd step

The may-set of first reached read states: The set of read states, that may be reached from \(x\) (i.e. while descending into \(r\)) via sequences of \(\cdot\)-transitions:

\[
\text{first}[r] = \{ i \mid r \cdot (x, \epsilon, x) \}
\]

#### Example:

\[
\begin{array}{c|c|c}
\text{transitions} & t \\
\hline
\{ 0 \} & \{ 1 \}
\end{array}
\]

### Berry-Sethi Approach: 3rd step

The may-set of next read states: The set of read states reached after reading \(r\), that may be reached next via sequences of \(\cdot\)-transitions.

\[
\text{next}[r] = \{ i \mid r \cdot i \}
\]

#### Example:

\[
\begin{array}{c|c|c}
\text{transitions} & t \\
\hline
\{ 0 \} & \{ 1 \}
\end{array}
\]

### Berry-Sethi Approach: 1st step

Implementation:
- DFS post-order traversal

For leaves \(r \equiv i \cdot x\) we find

\[
\text{empty}[r] = \begin{cases} 
\top & \text{if } x = \epsilon \\
\bot & \text{otherwise}
\end{cases}
\]

Otherwise:

\[
\begin{align*}
\text{empty}[r_1 \cup r_2] &= \text{empty}[r_1] \lor \text{empty}[r_2] \\
\text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \land \text{empty}[r_2] \\
\text{empty}[r_1?] &= \text{empty}[r_1] \\
\text{empty}[r_1^*] &= \top
\end{align*}
\]

### Berry-Sethi Approach: 2nd step

Implementation:
- DFS post-order traversal

For leaves \(r \equiv i \cdot x\) we find

\[
\text{empty}[r] = \begin{cases} 
\top & \text{if } x = \epsilon \\
\bot & \text{otherwise}
\end{cases}
\]

Otherwise:

\[
\begin{align*}
\text{empty}[r_1 \cup r_2] &= \text{empty}[r_1] \lor \text{empty}[r_2] \\
\text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \land \text{empty}[r_2] \\
\text{empty}[r_1?] &= \text{empty}[r_1] \\
\text{empty}[r_1^*] &= \top
\end{align*}
\]
Lexical Analysis

Chapter 4:
Turning NFAs deterministic

The expected outcome:

Remarks:
- Ideal automaton would be even more compact
- It is used for XML to define Content Models
- The result may not be, what we had in mind...

⇒ Powerset-Construction

Powerset Construction

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: Powersets of \( Q \);
Start state: \( I \);
Final states: \( \{Q' \subseteq Q \mid Q' \neq \emptyset \land Q' \not\supseteq F\} \);
Transitions:
\[
(\mathcal{P}(Q', a) = \{q \in Q' \mid q \in Q' \land (p, a, q) \in \delta \}).
\]

Powerset Construction

... for example:

Observation:
There are exponentially many powersets of \( Q \)

- Idea: Consider only contributing powersets. Starting with the set \( Q_m = \{\} \) we only add further states by need
- i.e., whenever we can reach them from a state in \( Q_m \)
- 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.

Powerset Construction

Berry-Sethi Approach: 4th step

The may-set of last reached read states: The set of read states, which may be reached last during the traversal of \( r \) connected to the root via \( \epsilon \)-transitions only:
\[
\text{last}[r] = \{i \in r : (i, \epsilon) \in \delta \}.
\]

Berry-Sethi Approach: 4th step

Implementation:
DFS post-order traversal

for leaves \( r = e \) we find \( \text{last}[r] = \{i : (i, \epsilon) \notin r\} \).

Otherwise:
\[
\text{last}(r_1 \cup r_2) = \text{last}(r_1) \cup \text{last}(r_2) \quad \text{if} \quad \text{empty}(r_2) = \epsilon
\]
\[
\text{last}(r_1 \cup r_2) = \text{last}(r_2) \quad \text{if} \quad \text{empty}(r_1) = \epsilon
\]
\[
\text{last}(r_1') = \text{last}(r_1)
\]

... for example:

Lexical Analysis

Berry-Sethi Approach: (sophisticated version)

Construction (sophisticated version):
Create an automaton based on the syntax tree's new attributes:

- States: \( \{\} \cup \{s \mid s \text{ a leaf } \} \)
- Start state: \( \epsilon \)
- Final states: \( \text{last}[\epsilon] \) if \( \text{empty}[\epsilon] = \epsilon \)
- Final states: \( \text{last}(r) \) otherwise
- Transitions:
  - \( (s, a, b) \) if \( b \in \text{last}(s) \) and \( b \) labeled with \( a \)
  - \( (s, a, b) \) if \( b \in \text{next}[s] \) and \( b \) labeled with \( a \)

We call the resulting automaton \( A_n \).

Powerset Construction

... for example:
Remarks:

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

Summary:

**Theorem:**
For each regular expression \( e \) we can compute a deterministic automaton \( A = P(A_e) \) with
\[ L(A) = \{ w \mid \exists \text{ some } i \text{ such that } w \in [e_i] \}. \]

Scanner design

**Input (simplified):** a set of rules:
\[ e_1 \{ \text{action}_1 \} \]
\[ e_2 \{ \text{action}_2 \} \]
\[ \ldots \]
\[ e_k \{ \text{action}_k \} \]

**Output:** a program,

\[ H \text{Hallo}\) \; \) ;( \) \) w \text{r i t l e n} \]

**Implementation:**

**Idea:**
Create the NFA \( P(A_e) = (Q, \Sigma, q_0, F) \) for the expression \( e = (e_1 | \ldots | e_k) \);
Define the sets:
\[ F_1 = \{ q \in F \mid \text{last}[q] \neq \emptyset \} \]
\[ F_2 = \{ q \in F \mid \text{last}[q_2] \neq \emptyset \} \]
\[ \vdots \]
\[ F_k = \{ q \in F \mid \text{last}[q_k] \neq \emptyset \} \]

For input \( w \) we find: \( \delta^*(q_0, w) \in F_i \) iff the scanner must execute \( \text{action}_i \) for \( w \)

**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 was reached.
- Pointer \( B \) tracks the current position.

**Implementation:**

**Idea:**
- The current state being \( q_0 = \emptyset \), we consume input up to position \( A \) and reset:
  \[ B := A; A := \perp; q_0 := q_1; q_4 := \perp \]

**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

Remarks:

* "." matches all characters different from "\n".
* For every state we generate the scanner respectively.
* Methods \( \text{yybegin} (\text{STATE}) \); switches between different scanners.
* 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.