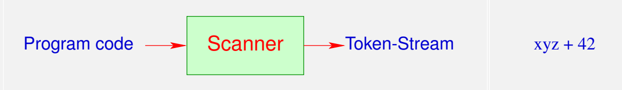
Topic:

Lexical Analysis

The Lexical Analysis



- A Token is a sequence of characters, which together form a unit.
- Tokens are subsumed in classes. For example:
  - → Names (Identifiers) e.g. `xyz`, `pi`, ...
  - → Constants e.g. `42`, `3.14`, `"abc"`, ...
  - → Operators e.g. `+`, ...
  - → Reserved terms e.g. `if`, `int`, ...

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*).

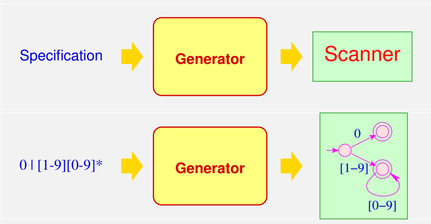
The Lexical Analysis

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.

The Lexical Analysis - Generating:

... in our case:



Specification of Token-classes: Regular expressions;
Generated Implementation: Finite automata + X

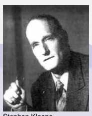
Chapter 1:

Basics: Regular Expressions

Regular Expressions

Basics

- Program code is composed from a finite alphabet Σ of input characters, e.g. Unicode
- The sets of textfragments of a token class is in general regular.
- Regular languages can be specified by regular expressions.


Stephen Kleene

Definition Regular Expressions

The set $\mathcal{E}_\Sigma$ of (non-empty) regular expressions is the smallest set $\mathcal{E}$ with:

- $\epsilon \in \mathcal{E}$ (ϵ a new symbol not from Σ);
- $a \in \mathcal{E}$ for all $a \in \Sigma$;
- $(e_1 \mid e_2), (e_1 \cdot e_2), {e_1}^* \in \mathcal{E}$ if $e_1, e_2 \in \mathcal{E}$.

Regular Expressions

... Example:

$$((a \cdot b^*) \cdot a)$$
$$(a \mid b)$$
$$((a \cdot b) \cdot (a \cdot b))$$

Attention:

- We distinguish between characters $a, 0, \$$,... and Meta-symbols $(, |,)$,...
- To avoid (ugly) parantheses, we make use of Operator-Precedences:

$$^* > \cdot > |$$

  and omit "$\cdot$"
- Real Specification-languages offer additional constructs:

$$e? \equiv (\epsilon \mid e)$$
$$e^+ \equiv (e \cdot e^*)$$

  and omit "ϵ"

Regular Expressions

Specification needs Semantics

...Example:

Specification	Semantics
$abab$	$\{abab\}$
$a \mid b$	$\{a, b\}$
ab^*a	$\{ab^na \mid n \geq 0\}$

For $e \in \mathcal{E}_\Sigma$ we define the specified language $[\![e]\!] \subseteq \Sigma^*$ inductively by:

$$\begin{array}{lcl} [\![\epsilon]\!] & = & \{\epsilon\} \\ [\![a]\!] & = & \{a\} \\ [\![e^*]\!] & = & ([\![e]\!])^* \\ [\![e_1|e_2]\!] & = & [\![e_1]\!] \cup [\![e_2]\!] \\ [\![e_1 \cdot e_2]\!] & = & [\![e_1]\!] \cdot [\![e_2]\!] \end{array}$$

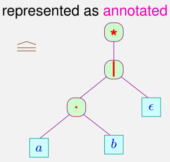
Keep in Mind:

- The operators $(\_)^*, \cup, \cdot$ are interpreted in the context of sets of words:

$$\begin{array}{lcl} (L)^* & = & \{w_1 \ldots w_k \mid k \geq 0, w_i \in L\} \\ L_1 \cdot L_2 & = & \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\} \end{array}$$

- Regular expressions are internally represented as annotated ranked trees:

$(ab|\epsilon)^*$ $\widehat{=}$



Inner nodes: Operator-applications;
Leaves: particular symbols or ϵ.

Regular Expressions

Example: Identifiers in Java:

```
le = [a-zA-Z\_\$]
di = [0-9]
Id = {le} ({le} | {di})*

Float = {di}*(\.{di}|{di}\.){di}* ((e|E)(\+|\-)?{di}+)?
```

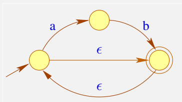
Remarks:

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

Chapter 2:
Basics: Finite Automata

Finite Automata

Example:



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

Finite Automata

Definition Finite Automata

A non-deterministic finite automaton (NFA) is a tuple $A = (Q, \Sigma, \delta, I, F)$ with:

Q	a finite set of states;
Σ	a finite alphabet of inputs;
$I \subseteq Q$	the set of start states;
$F \subseteq Q$	the set of final states and
δ	the set of transitions (-relation)

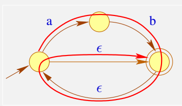
For an NFA, we reckon:

Definition Deterministic Finite Automata

Given $\delta : Q \times \Sigma \to Q$ a function and $|I| = 1$, then we call the NFA A deterministic (DFA).

Finite Automata

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



Finite Automata

Once again, more formally:

- We define the transitive closure δ^* of δ as the smallest set δ' with:

  $(p, \epsilon, p) \in \delta'$ and
  $(p, xw, q) \in \delta'$ if $(p, x, p_1) \in \delta$ and $(p_1, w, q) \in \delta'$.

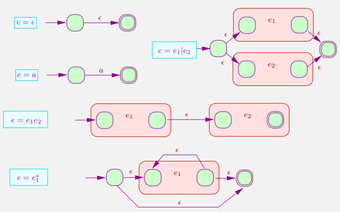
  δ^* characterizes for a path between the states p and q the words obtained by concatenating the labels along it.

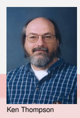
- The set of all accepting words, i.e. A's accepted language can be described compactly as:

$$\mathcal{L}(A) = \{w \in \Sigma^* \mid \exists i \in I, f \in F : (i, w, f) \in \delta^*\}$$

Chapter 3:
Converting Regular Expressions to NFAs

In Linear Time from Regular Expressions to NFAs





Thompson's Algorithm

Produces $\mathcal{O}(n)$ states for regular expressions of length n.

A formal approach to Thompson's Algorithm

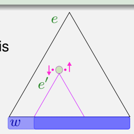
Berry-Sethi AlgorithmGlushkov Automaton

Produces exactly $n + 1$ states without ϵ-transitions and demonstrates → *Equality Systems* and → *Attribute Grammars*

Idea:

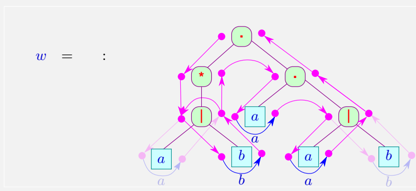
An automaton covering the syntax tree of a regular expression e tracks (conceptionally via markers "•"), which subexpressions e' are reachable consuming the rest of input w.

- markers contribute an entry or exit point into the automaton for this subexpression
- edges for each layer of subexpression are modelled after Thompson's automata



Berry-Sethi Approach

... for example:



Berry-Sethi Approach

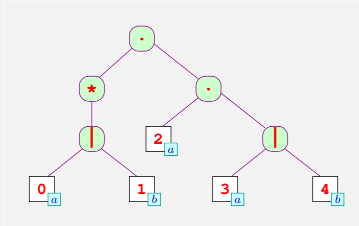
In general:

- Input is only consumed at the leaves.
- Navigating the tree does not consume input → ϵ-transitions
- For a formal construction we need identifiers for states.
- For a node n's identifier we take the subexpression, corresponding to the subtree dominated by n.
- There are possibly identical subexpressions in one regular expression.

$\Longrightarrow$ we enumerate the leaves ...

Berry-Sethi Approach

... for example:



Berry-Sethi Approach (naive version)

Construction (naive version):

States: $\bullet r$, $r\bullet$ with r nodes of e;
Start state: $\bullet e$;
Final state: $e\bullet$;
Transitions: for leaves $r \equiv \boxed{i \mid x}$ we require: $(\bullet r, x, r\bullet)$.
The leftover transitions are:

r	Transitions
$r_1 \mid r_2$	$(\bullet r, \epsilon, \bullet r_1)$
	$(\bullet r, \epsilon, \bullet r_2)$
	$(r_1\bullet, \epsilon, r\bullet)$
	$(r_2\bullet, \epsilon, r\bullet)$
$r_1 \cdot r_2$	$(\bullet r, \epsilon, \bullet r_1)$
	$(r_1\bullet, \epsilon, \bullet r_2)$
	$(r_2\bullet, \epsilon, r\bullet)$

r	Transitions
r_1^*	$(\bullet r, \epsilon, r\bullet)$
	$(\bullet r, \epsilon, \bullet r_1)$
	$(r_1\bullet, \epsilon, \bullet r_1)$
	$(r_1\bullet, \epsilon, r\bullet)$
$r_1?$	$(\bullet r, \epsilon, r\bullet)$
	$(\bullet r, \epsilon, \bullet r_1)$
	$(r_1\bullet, \epsilon, r\bullet)$

Berry-Sethi Approach

Discussion:

- Most transitions navigate through the expression
- The resulting automaton is in general nondeterministic

⇒ Strategy for the sophisticated version:
Avoid generating ϵ-transitions

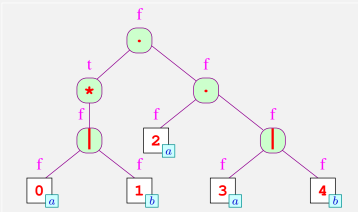
Idea:
Pre-compute helper attributes during D(epth)F(irst)S(earch)!

Necessary node-attributes:
first the set of read states below r, which may be reached first, when descending into r.
next the set of read states, which may be reached first in the traversal after r.
last the set of read states below r, which may be reached last when descending into r.
empty can the subexpression r consume ϵ ?

Berry-Sethi Approach: 1st step

empty$[r] = t$ if and only if $\epsilon \in [\![r]\!]$

... for example:



Berry-Sethi Approach: 1st step

Implementation:
DFS post-order traversal

for leaves $r \equiv \boxed{i \mid x}$ we find empty$[r] = (x \equiv \epsilon)$.

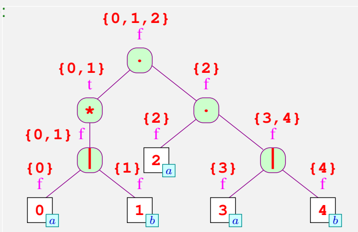
Otherwise:

$$\begin{aligned} \text{empty}[r_1 \mid r_2] &= \text{empty}[r_1] \vee \text{empty}[r_2] \\ \text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \wedge \text{empty}[r_2] \\ \text{empty}[r_1^*] &= t \\ \text{empty}[r_1?] &= t \end{aligned}$$

Berry-Sethi Approach: 2nd step

The may-set of first reached read states: The set of read states, that may be reached from $\bullet r$ (i.e. while descending into r) via sequences of ϵ-transitions:
first$[r] = \{i \text{ in } r \mid (\bullet r, \epsilon, \bullet\boxed{i \mid x}) \in \delta^*, x \neq \epsilon\}$

... for example:



Berry-Sethi Approach: 2nd step

Implementation:
DFS post-order traversal

for leaves $r \equiv \boxed{i \mid x}$ we find first$[r] = \{i \mid x \neq \epsilon\}$.

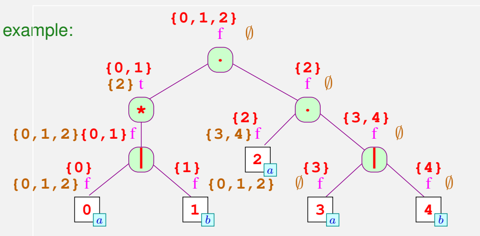
Otherwise:

$$\begin{aligned} \text{first}[r_1 \mid r_2] &= \text{first}[r_1] \cup \text{first}[r_2] \\ \text{first}[r_1 \cdot r_2] &= \begin{cases} \text{first}[r_1] \cup \text{first}[r_2] & \text{if } \text{empty}[r_1] = t \\ \text{first}[r_1] & \text{if } \text{empty}[r_1] = f \end{cases} \\ \text{first}[r_1^*] &= \text{first}[r_1] \\ \text{first}[r_1?] &= \text{first}[r_1] \end{aligned}$$

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 ϵ-transitions.
next$[r] = \{i \mid (r\bullet, \epsilon, \bullet\boxed{i \mid x}) \in \delta^*, x \neq \epsilon\}$

... for example:



Berry-Sethi Approach: 3rd step

Implementation:
DFS pre-order traversal

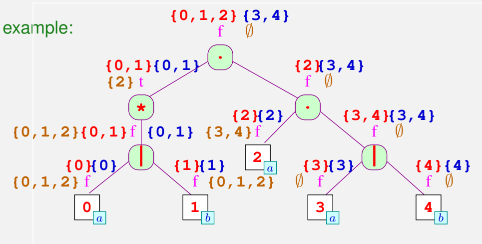
For the root, we find: next$[e] = \emptyset$
Apart from that we distinguish, based on the context:

r	Equalities
$r_1 \mid r_2$	next$[r_1]$ = next$[r]$
	next$[r_2]$ = next$[r]$
$r_1 \cdot r_2$	next$[r_1]$ = $\begin{cases} \text{first}[r_2] \cup \text{next}[r] & \text{if empty}[r_2] = t \\ \text{first}[r_2] & \text{if empty}[r_2] = f \end{cases}$
	next$[r_2]$ = next$[r]$
r_1^*	next$[r_1]$ = first$[r_1] \cup$ next$[r]$
$r_1?$	next$[r_1]$ = next$[r]$

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 ϵ-transitions only:
$\text{last}[r] = \{i \text{ in } r \mid (\boxed{i \mid x}\bullet, \epsilon, r\bullet) \in \delta^*, x \neq \epsilon\}$

... for example:



Berry-Sethi Approach: 4th step

Implementation:
DFS post-order traversal

for leaves $r \equiv \boxed{i \mid x}$ we find $\text{last}[r] = \{i \mid x \neq \epsilon\}$.

Otherwise:

$$\begin{array}{lll} \text{last}[r_1 \mid r_2] & = & \text{last}[r_1] \cup \text{last}[r_2] \\ \text{last}[r_1 \cdot r_2] & = & \begin{cases} \text{last}[r_1] \cup \text{last}[r_2] & \text{if } \text{empty}[r_2] = t \\ \text{last}[r_2] & \text{if } \text{empty}[r_2] = f \end{cases} \\ \text{last}[r_1^*] & = & \text{last}[r_1] \\ \text{last}[r_1?] & = & \text{last}[r_1] \end{array}$$

Berry-Sethi Approach: (sophisticated version)

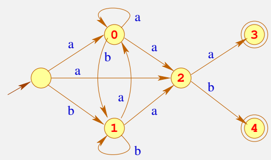
Construction (sophisticated version):
Create an automanton based on the syntax tree's new attributes:

States: $\{\bullet e\} \cup \{i\bullet \mid i$ a leaf not $\epsilon\}$
Start state: $\bullet e$
Final states: $\text{last}[e]$ if $\text{empty}[e] = f$
$\{\bullet e\} \cup \text{last}[e]$ otherwise
Transitions: $(\bullet e, a, i\bullet)$ if $i \in \text{first}[e]$ and i labled with a.
$(i\bullet, a, i'\bullet)$ if $i' \in \text{next}[i]$ and i' labled with a.

We call the resulting automaton A_e.

Berry-Sethi Approach

... for example:

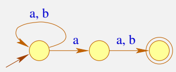


Remarks:

- 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...

Chapter 4:
Turning NFAs deterministic

The expected outcome:



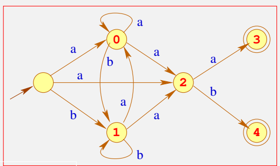
Remarks:

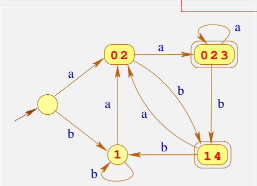
- ideal automaton would be even more compact
  (→ *Antimirov automata, Follow Automata*)
- but Berry-Sethi is rather directly constructed
- Anyway, we need a deterministic version

⇒ Powerset-Construction

Powerset Construction

... for example:





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: $\{Q' \subseteq Q \mid Q' \cap F \neq \emptyset\}$;
Transitions: $\delta_{\mathcal{P}}(Q', a) = \{q \in Q \mid \exists p \in Q' : (p, a, q) \in \delta\}$.

Powerset Construction

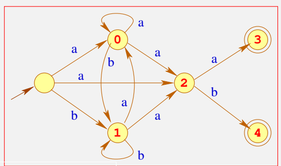
Observation:
There are exponentially many powersets of Q

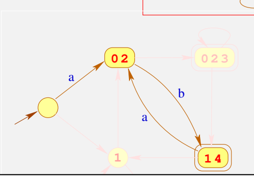
- Idea: Consider only contributing powersets. Starting with the set $Q_{\mathcal{P}} = \{I\}$ we only add further states by need ...
- i.e., whenever we can reach them from a state in $Q_{\mathcal{P}}$
- However, the resulting automaton can become enormously huge
  ... which is (sort of) not happening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

Powerset Construction

... for example:

a	b	a	b





Remarks:

- For an input sequence of length n, maximally $\mathcal{O}(n)$ 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

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

Chapter 5:
Scanner design

Scanner design

Input (simplified): a set of rules:

e_1 { `action`$_1$ }
e_2 { `action`$_2$ }
...
e_k { `action`$_k$ }

Output: a program,

... 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 `action`$_i$ for w.

Implementation:

Idea:

- Create the NFA $\mathcal{P}(A_e) = (Q, \Sigma, \delta, q_0, F)$ for the expression $e = (e_1 \mid \ldots \mid e_k)$;
- Define the sets:

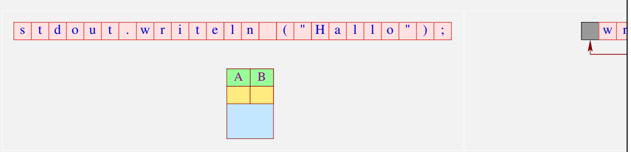
$$
\begin{aligned}
F_1 &= \{q \in F \mid q \cap \text{last}[e_1] \neq \emptyset\} \\
F_2 &= \{q \in (F \backslash F_1) \mid q \cap \text{last}[e_2] \neq \emptyset\} \\
&\ldots \\
F_k &= \{q \in (F \backslash (F_1 \cup \ldots \cup F_{k-1})) \mid q \cap \text{last}[e_k] \neq \emptyset\}
\end{aligned}
$$

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

Implementation:

Idea (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.

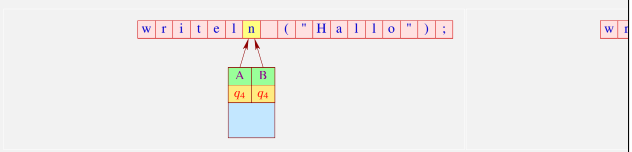


Implementation:

Idea (cont'd):

- The current state being $q_B = \emptyset$, we consume input up to position A and reset:

$$
\begin{aligned}
B &:= A; & A &:= \bot; \\
q_B &:= q_0; & q_A &:= \bot
\end{aligned}
$$



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:

⟨`state`⟩ { e_1 { `action`$_1$ `yybegin(state`$_1$`);` }
e_2 { `action`$_2$ `yybegin(state`$_2$`);` }
...
e_k { `action`$_k$ `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:

```
⟨YYINITIAL⟩   "/*"     { yybegin(COMMENT); }
⟨COMMENT⟩   { "*/"     { yybegin(YYINITIAL); }
              . | \n   { }
            }
```


Remarks:

- "`.`" matches all characters different from "`\n`".
- For every state we generate the scanner respectively.
- Method `yybegin (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.