Script **generated by TTT**

Title: Petter: Compilerbau (26.04.2018)

Date: Thu Apr 26 14:14:21 CEST 2018

Duration: 96:55 min

Pages: 38

Berry-Sethi Approach: (sophisticated version)

Construction (sophisticated version):

Create an automanton based on the syntax tree's new attributes:

States: $\{\bullet e\} \cup \{i\bullet \mid i \text{ a leaf}\}$

Start state: $\bullet e$

Final states: $\text{last}[e]$ if $\text{empty}[e] = f$

$\{\bullet e\} \cup \text{last}[e]$ otherwise

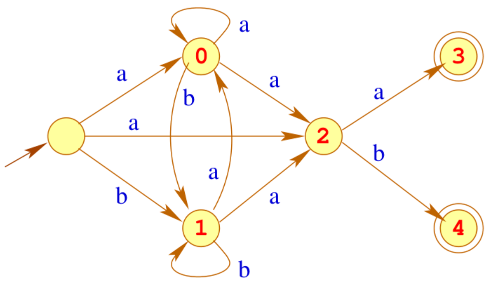
Transitions: $(\bullet e, a, i\bullet)$ if $i \in \text{first}[e]$ and i labled with a.

$(i\bullet, a, i'\bullet)$ if $i' \in \text{next}[i]$ and i' labled with a.

We call the resulting automaton A_e.

Berry-Sethi Approach

... for example:

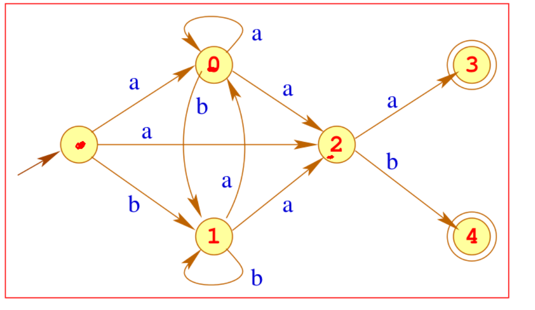


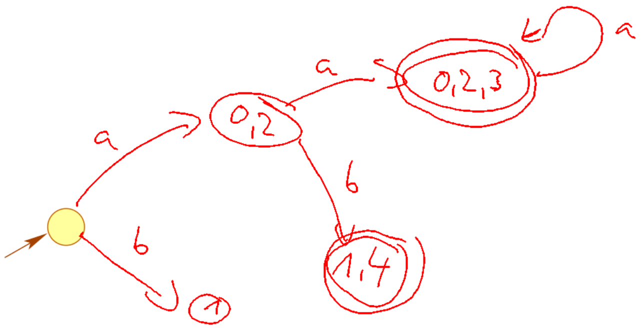
Remarks:

- 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...

Powerset Construction

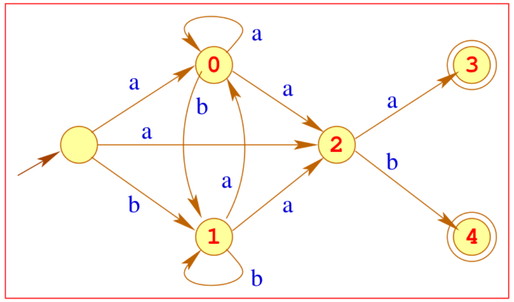
... for example:

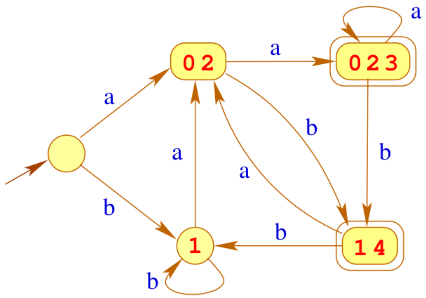




Powerset Construction

... 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

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

Powerset Construction

Observation:

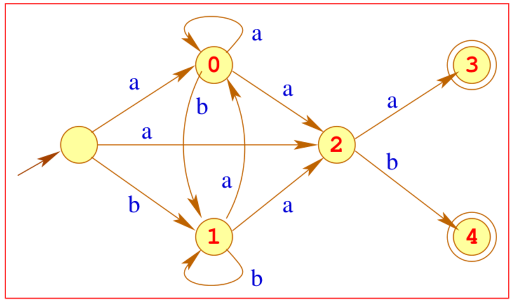
There are exponentially many powersets of Q

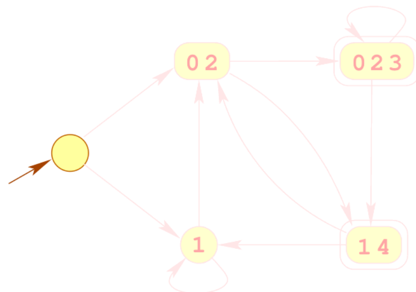
- Idea: Consider only contributing powersets. Starting with the set $Q_{\mathcal{P}} = \{I\}$ we only add further states by need ...
- i.e., whenever we can reach them from a state in $Q_{\mathcal{P}}$
- However, the resulting automaton can become enormously huge ... which is (sort of) not happening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

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.

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

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

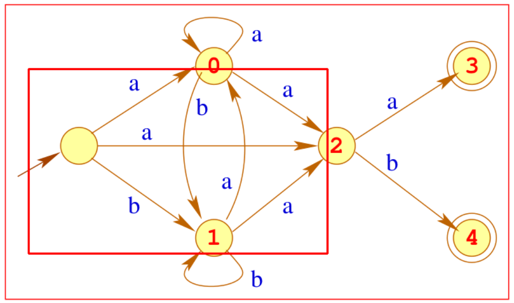
Chapter 5:

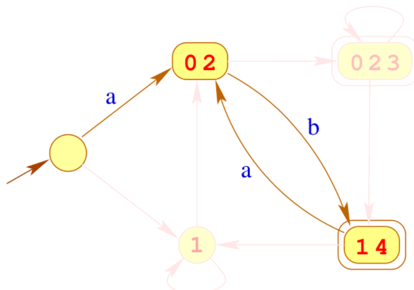
Scanner design

Powerset Construction

... for example:

a	b	a	b

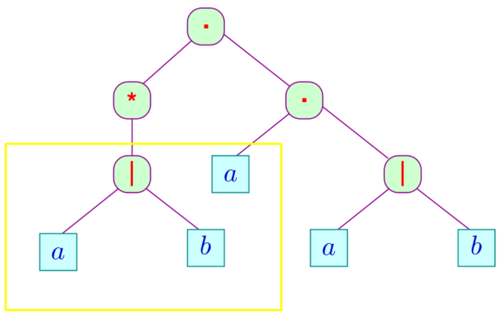




Berry-Sethi Approach

... for example:

$(a|b)^*a(a|b)$



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.

⟹ we enumerate the leaves ...

Berry-Sethi Approach: (sophisticated version)

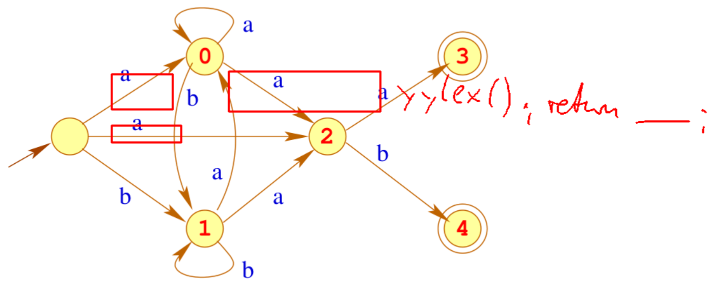
Construction (sophisticated version):
Create an automanton based on the syntax tree's new attributes:

States: $\{\bullet e\} \cup \{i\bullet \mid i$ a leaf$\}$
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...

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:

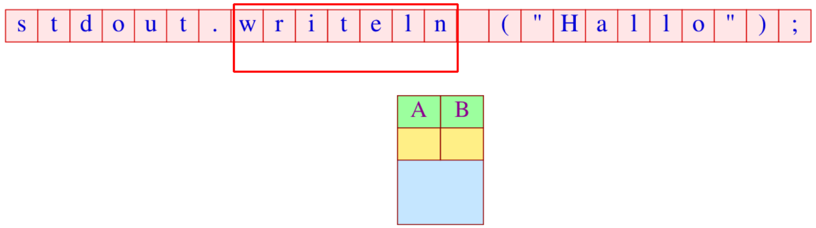
$$\begin{array}{lcl} F_1 & = & \{q \in F \mid q \cap \text{last}[e_1] \neq \emptyset\} \\ F_2 & = & \{q \in (F \backslash F_1) \mid q \cap \text{last}[e_2] \neq \emptyset\} \\ & \ldots & \\ F_k & = & \{q \in (F \backslash (F_1 \cup \ldots \cup F_{k-1})) \mid q \cap \text{last}[e_k] \neq \emptyset\} \end{array}$$

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



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 $\texttt{state}_i$.
- The start state is called (e.g.flex JFlex) `YYINITIAL`.

... for example:

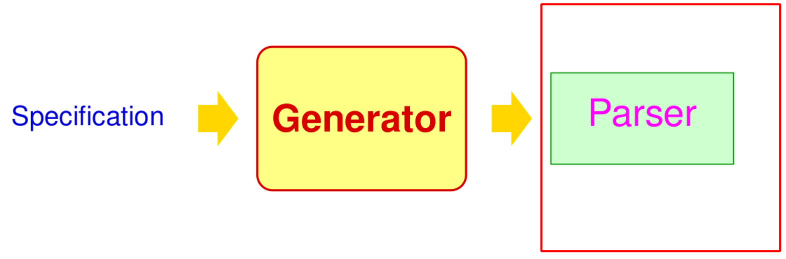
```
⟨YYINITIAL⟩    "/*"      { yybegin(COMMENT); }
⟨COMMENT⟩   {  "*/"      { yybegin(YYINITIAL); }
               . | \n    {  }
            }
```

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.

Discussion:

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



Chapter 1:

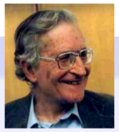
Basics of Contextfree Grammars

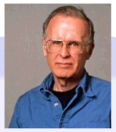
Basics: Context-free Grammars

- 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...


Noam Chomsky


John Backus

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

Conventions

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

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

Conventions

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

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

... for example:

$$S \to a\,S\,b$$
$$S \to \epsilon$$

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

Conventions:

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

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

... a practical example:

$$
\begin{array}{lcl}
S & \to & \langle\text{stmt}\rangle \\
\langle\text{stmt}\rangle & \to & \langle\text{if}\rangle \mid \langle\text{while}\rangle \mid \langle\text{rexp}\rangle; \\
\langle\text{if}\rangle & \to & \text{if} \ (\ \langle\text{rexp}\rangle \ ) \ \langle\text{stmt}\rangle \ \text{else} \ \langle\text{stmt}\rangle \\
\langle\text{while}\rangle & \to & \text{while} \ (\ \langle\text{rexp}\rangle \ ) \ \langle\text{stmt}\rangle \\
\langle\text{rexp}\rangle & \to & \text{int} \mid \langle\text{lexp}\rangle \mid \langle\text{lexp}\rangle = \langle\text{rexp}\rangle \mid ... \\
\langle\text{lexp}\rangle & \to & \text{name} \mid ...
\end{array}
$$

... a practical example:

$S \to \langle\text{stmt}\rangle$
$\langle\text{stmt}\rangle \to \langle\text{if}\rangle \mid \langle\text{while}\rangle \mid \langle\text{rexp}\rangle;$
$\langle\text{if}\rangle \to \text{if}\ (\ \langle\text{rexp}\rangle\ )\ \langle\text{stmt}\rangle\ \text{else}\ \langle\text{stmt}\rangle$
$\langle\text{while}\rangle \to \text{while}\ (\ \langle\text{rexp}\rangle\ )\ \langle\text{stmt}\rangle$
$\langle\text{rexp}\rangle \to \text{int} \mid \langle\text{lexp}\rangle \mid \langle\text{lexp}\rangle = \langle\text{rexp}\rangle \mid \ldots$
$\langle\text{lexp}\rangle \to \text{name} \mid \ldots$

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) (with $j \geq 0$).

Pair of grammars:

E	$\to$	$E+E$	|	$E*E$	|	$(\ E\ )$	| name	| int

E	$\to$	$E+T$	|	T		
T	$\to$	$T*F$	|	F		
F	$\to$	$(\ E\ )$	|	name	|	int

Both grammars describe the same language

Derivation

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

... for example:

$$\begin{aligned} E &\to \underline{E} + T \\ &\to \underline{T} + T \\ &\to T * \underline{F} + T \\ &\to \underline{T} * \text{int} + T \\ &\to \underline{F} * \text{int} + T \\ &\to \text{name} * \text{int} + \underline{T} \\ &\to \text{name} * \text{int} + \underline{F} \\ &\to \text{name} * \text{int} + \text{int} \end{aligned}$$

Definition

The derivation relation $\to$ is a relation on words over $N \cup T$, with

$\alpha \to \alpha'$ iff $\alpha = \alpha_1\ A\ \alpha_2 \ \wedge\ \alpha' = \alpha_1\ \beta\ \alpha_2$ for an $A \to \beta \in P$

Derivation

Remarks:

- The relation $\to$ 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:

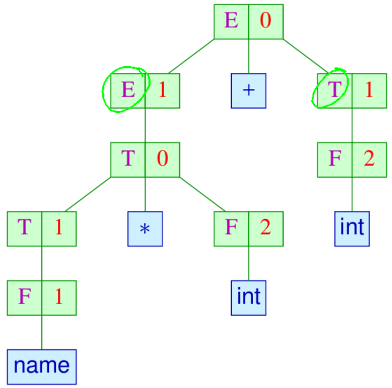
$$\mathcal{L}(G) = \{w \in T^* \mid S \to^* w\}$$

Derivation Tree

Derivations of a symbol are represented as derivation trees:

... for example:

$$
\begin{aligned}
\underline{E} &\to^0 \underline{E} + T \\
&\to^1 \underline{T} + T \\
&\to^0 T * \underline{F} + T \\
&\to^2 \underline{T} * \text{int} + T \\
&\to^1 \underline{F} * \text{int} + T \\
&\to^1 \text{name} * \text{int} + \underline{T} \\
&\to^1 \text{name} * \text{int} + \underline{F} \\
&\to^2 \text{name} * \text{int} + \text{int}
\end{aligned}
$$



A derivation tree for $A \in N$:

inner nodes: rule applications

root: rule application for A

leaves: terminals or ϵ

The successors of (B, i) correspond to right hand sides of the rule

Special Derivations

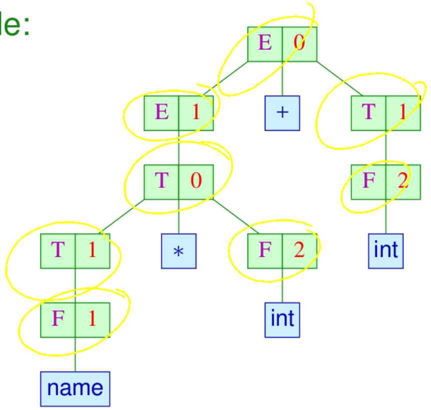
Attention:

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

- These are called leftmost (or rather rightmost) derivations and are denoted with the index L (or R respectively).
- Leftmost (or rightmost) derivations correspondt 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

Special Derivations

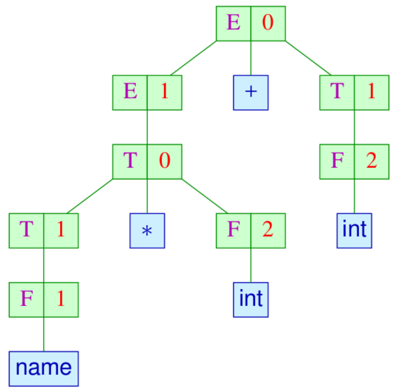
... for example:



Unique Grammars

The concatenation of leaves of a derivation tree t are often called yield(t) .

... for example:



gives rise to the concatenation: name $*$ int $+$ int .

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:

$$E \;\to\; E{+}E^{\,0} \;\mid\; E{*}E^{\,1} \;\mid\; (\,E\,)^{\,2} \;\mid\; \text{name}^{\,3} \;\mid\; \text{int}^{\,4}$$

$$\begin{array}{lcl} E & \to & E{+}T^{\,0} \;\mid\; T^{\,1} \\ T & \to & T{*}F^{\,0} \;\mid\; F^{\,1} \\ F & \to & (\,E\,)^{\,0} \;\mid\; \text{name}^{\,1} \;\mid\; \text{int}^{\,2} \end{array}$$

The first one is ambiguous, the second one is unique

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.

Chapter 2:
Basics of Pushdown Automata