Script **generated by TTT**

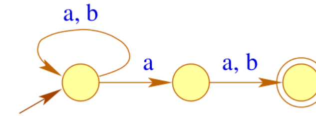
Title: Petter: Compilerbau (02.05.2019)

Date: Thu May 02 14:16:26 CEST 2019

Duration: 92:57 min

Pages: 31

The expected outcome:



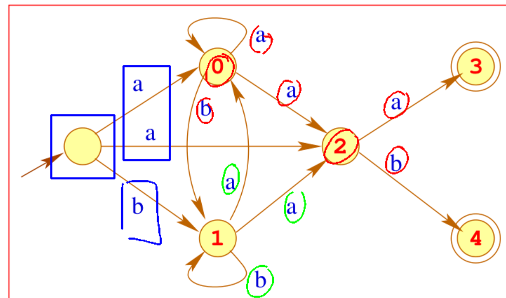
Remarks:

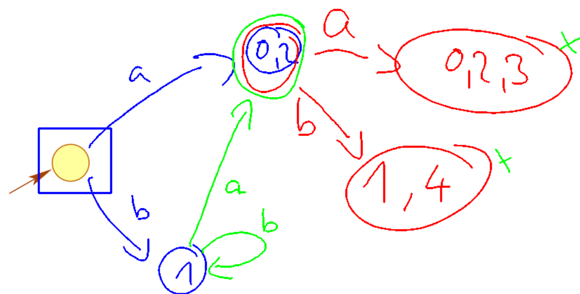
- ideal automaton would be even more compact (→ *Antimirov automata*, *Follow Automata*)
- but Berry-Sethi is rather directly constructed
- Anyway, we need a deterministic version

⇒ Powerset-Construction

Powerset Construction

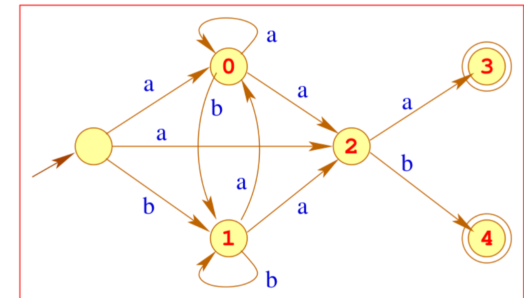
... for example:

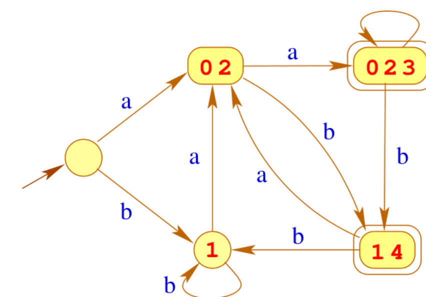




Powerset Construction

... 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

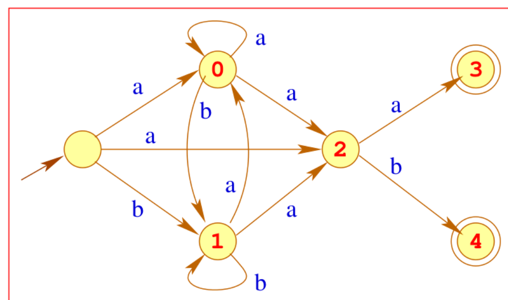
- Idea: Consider only contributing powersets. Starting with the set $Q_{\mathcal{P}} = \{I\}$ we only add further states by need ...
- i.e., whenever we can reach them from a state in $Q_{\mathcal{P}}$
- However, the resulting automaton can become enormously huge ... which is (sort of) not happening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

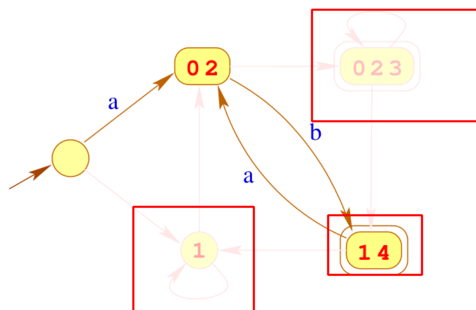
- Therefore, in tools like `grep` a regular expression's DFA is never created!
- Instead, only the sets, directly necessary for interpreting the input are generated while processing the input

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

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

Scanner design

Input (simplified): a set of rules:

e_1 { $\texttt{action}_1$ }
e_2 { $\texttt{action}_2$ }
...
e_k { $\texttt{action}_k$ }

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 $\texttt{action}_i$ 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:

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

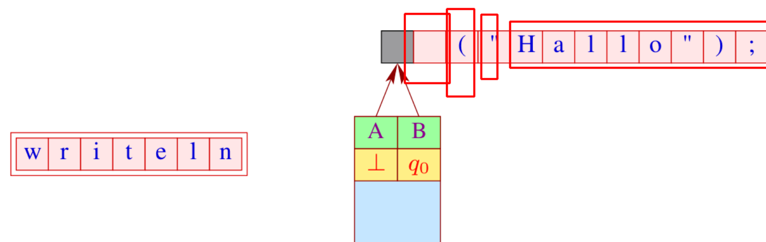
- For input w we find: $\delta^*(q_0, w) \in F_i$ iff the scanner must execute $\texttt{action}_i$ for w

Implementation:

Idea (cont'd):

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

$$B := A; \quad A := \perp;$$
$$q_B := q_0; \quad q_A := \perp$$



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

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    { }
            }
```


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$_k$); }
}

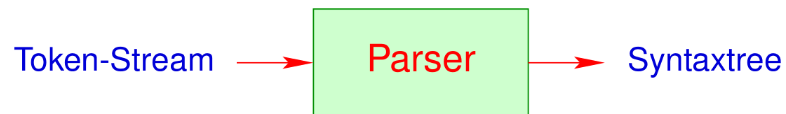
- The statement yybegin (state$_i$); resets the current state to state$_i$.
- The start state is called (e.g.flex JFlex) YYINITIAL.

... for example:

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

56 / 287

Syntactic Analysis



- 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 of the hierarchical structure: contextfree grammars
Generated implementation: Pushdown automata + X

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...

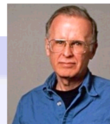
Basics: Context-free Grammars

- 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...

Definition: Context-Free Grammar

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

Noam Chomsky



John Backus

- 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:

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

... for example:

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

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

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: $A, B, C, ...,$ ⟨exp⟩, ⟨stmt⟩, ...;
- terminals are: $a, b, c, ...,$ int, name, ...;

... a practical example:

S	$\to$	⟨stmt⟩
⟨stmt⟩	$\to$	⟨if⟩ | ⟨while⟩ | ⟨rexp⟩;
⟨if⟩	$\to$	if (⟨rexp⟩) ⟨stmt⟩ else ⟨stmt⟩
⟨while⟩	$\to$	while (⟨rexp⟩) ⟨stmt⟩
⟨rexp⟩	$\to$	int | ⟨lexp⟩ | ⟨lexp⟩ = ⟨rexp⟩ | ...
⟨lexp⟩	$\to$	name | ...

... a practical example:

S	$\to$	⟨stmt⟩
⟨stmt⟩	$\to$	⟨if⟩ | ⟨while⟩ | ⟨rexp⟩;
⟨if⟩	$\to$	if (⟨rexp⟩) ⟨stmt⟩ else ⟨stmt⟩
⟨while⟩	$\to$	while (⟨rexp⟩) ⟨stmt⟩
⟨rexp⟩	$\to$	int | ⟨lexp⟩ | ⟨lexp⟩ = ⟨rexp⟩ | ...
⟨lexp⟩	$\to$	name | ...

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 $j \geq 0$).

stmt , (stmt)-> (rexp)

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:

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

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:

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

Definition

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

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

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

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:

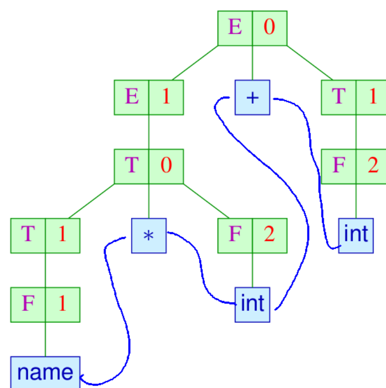
$$\mathcal{L}(G) = \{w \in T^* \mid S \to^* w\}$$

Derivation Tree

Derivations of a symbol are represented as derivation trees:

... for example:

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



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:

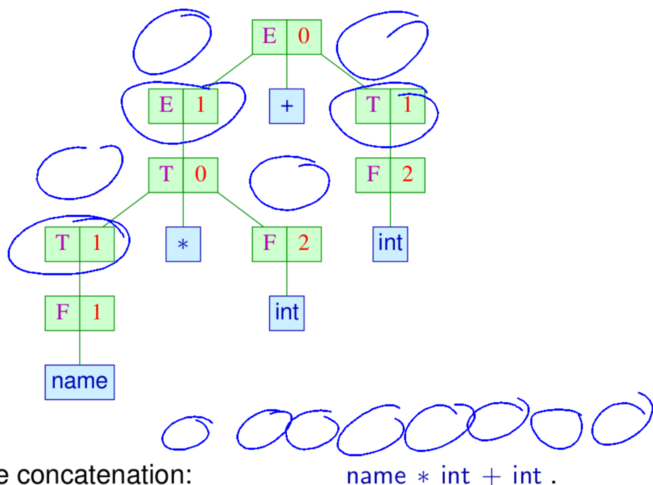
In contrast to arbitrary derivations, we find special ones, always rewriting the leftmost (or rather rightmost) occurance of a nonterminal.

- 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

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 yield$(t) = w$.

... in our example:

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

E	$\to$	$E{+}T^{\,0}$	$T^{\,1}$	
T	$\to$	$T{*}F^{\,0}$	$F^{\,1}$	
F	$\to$	$(\,E\,)^{\,0}$	name ^{1}	int ^{2}

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.

- 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.

Chapter 2:
Basics of Pushdown Automata