Script generated by TTT

Title: Petter: Compilerbau (25.04.2016)

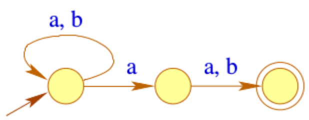
Date: Mon Apr 25 14:29:36 CEST 2016

Duration: 89:13 min

Pages: 53

Chapter 4:
Turning NFAs deterministic

The expected outcome:



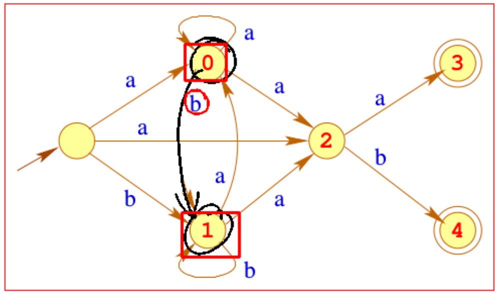
Remarks:

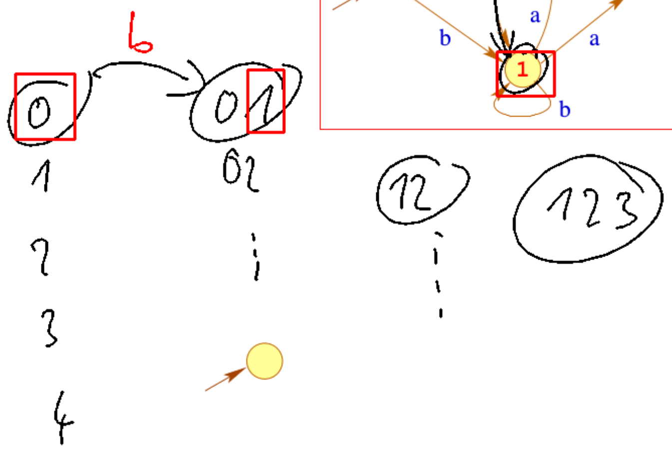
- ideal automaton would be even more compact
  (→ *Antimirov automata, Follow Automata*)
- but Berry-Sethi is rather directly constructed
- Anyway, we need a deterministic version

⇒ Powerset-Construction

Powerset Construction

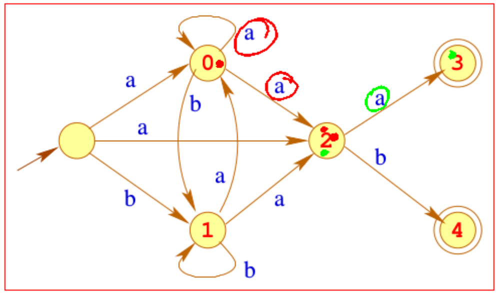
... for example:

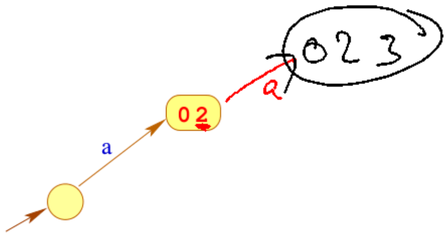




Powerset Construction

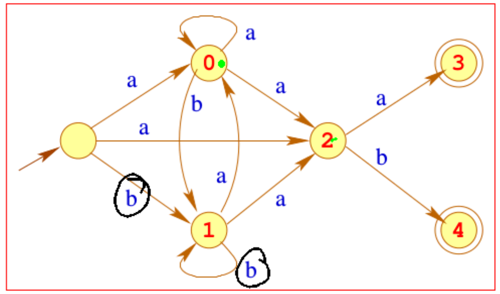
... for example:

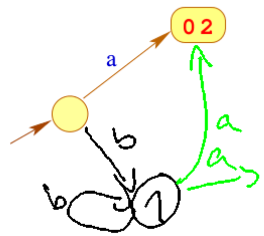




Powerset Construction

... for example:

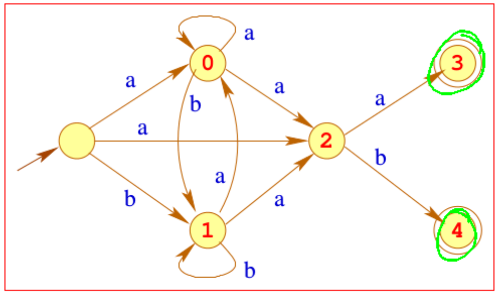


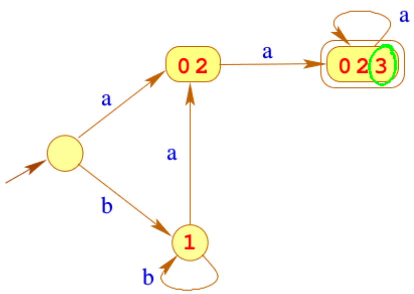


46 / 282

Powerset Construction

... for example:

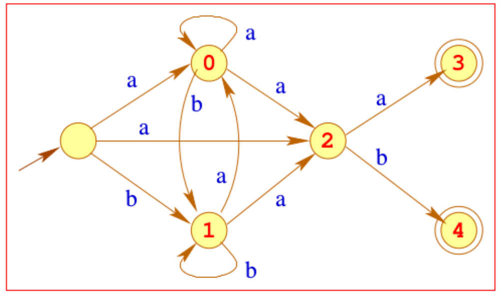


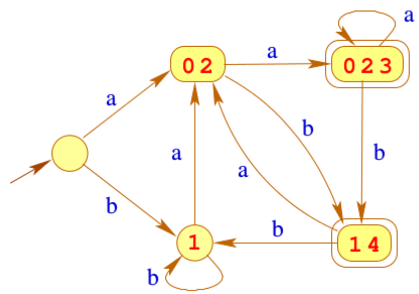


46 / 282

Powerset Construction

... for example:





46 / 282

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

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

Powerset Construction

Observation:

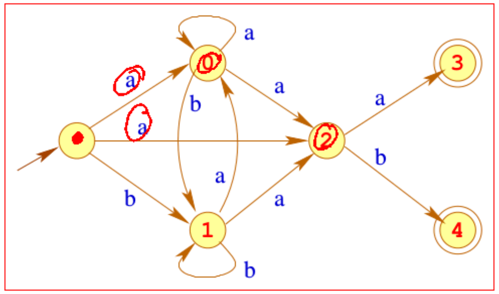
There are exponentially many powersets of Q

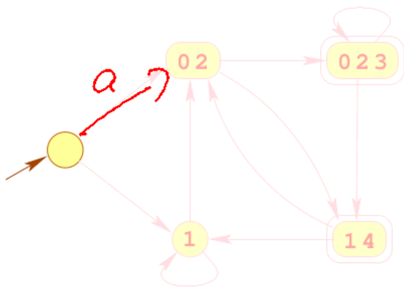
- Idea: Consider only contributing powersets. Starting with the set $Q_{\mathcal{P}} = \{I\}$ we only add further states by need ...
- i.e., whenever we can reach them from a state in $Q_{\mathcal{P}}$
- However, the resulting automaton can become enormously huge ... which is (sort of) not happening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

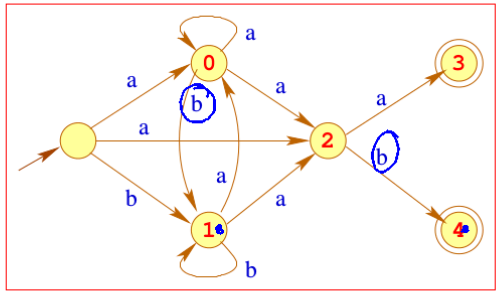


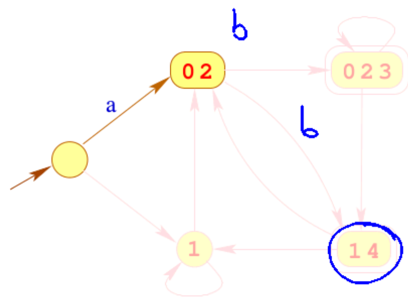


Powerset Construction

... for example:

a	~~b~~	a	b

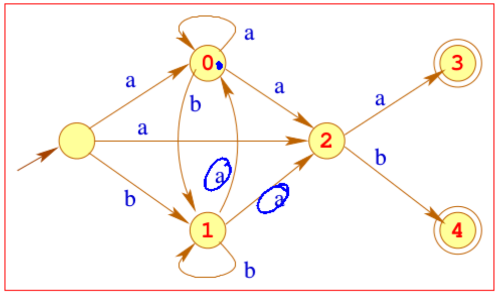


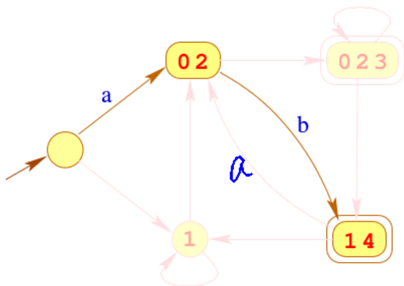


Powerset Construction

... for example:

a	b	a	b

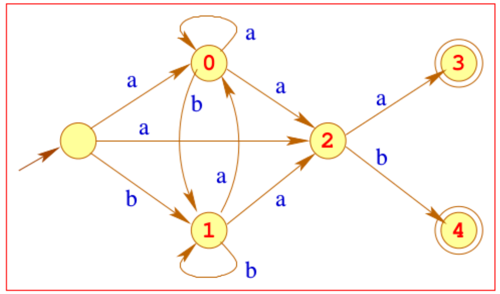


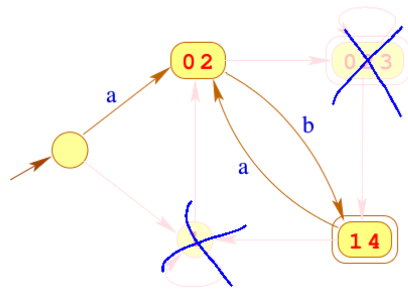


Powerset Construction

... 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

Scanner design

Input (simplified): a set of rules:

$$\begin{array}{ll} e_1 & \{\ \texttt{action}_1\ \} \\ e_2 & \{\ \texttt{action}_2\ \} \\ \ldots & \\ e_k & \{\ \texttt{action}_k\ \} \end{array}$$

yytext()

out print("<codes>" "data")

Scanner design

if

Input (simplified): a set of rules:

$$\begin{array}{lll} e_1 & & \{ \texttt{action}_1 \} \\ e_2 & & \{ \texttt{action}_2 \} \\ & \ldots & \\ e_k & & \{ \texttt{action}_k \} \end{array}$$

id

while

if

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.

Scanner design

whilea

Input (simplified): a set of rules:

$$\begin{array}{lll} e_1 & & \{ \texttt{action}_1 \} \\ e_2 & & \{ \texttt{action}_2 \} \\ & \ldots & \\ e_k & & \{ \texttt{action}_k \} \end{array}$$

while

id

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 $\boxed{e = (e_1 \mid \ldots \mid e_k)}$;
- Define the sets:

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

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

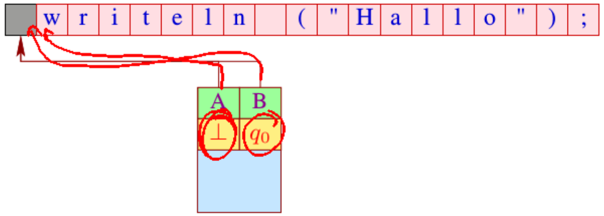
s	t	d	o	u	t	.	w	r	i	t	e	l	n		(	"	H	a	l	l	o	"	)	;

A	B

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.

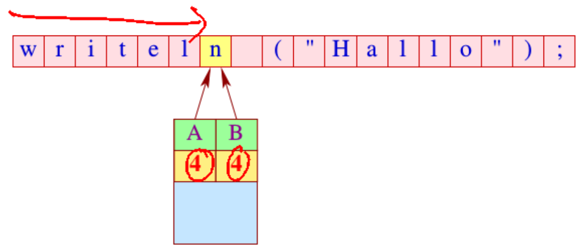


Implementation:

Idea (cont'd):

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

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

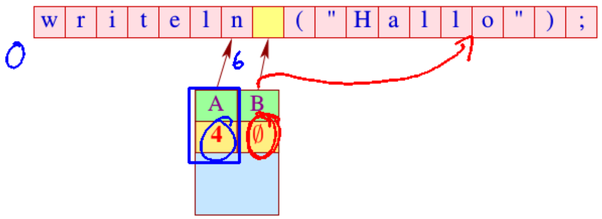


Implementation:

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:

$$\langle \texttt{state} \rangle \; \left\{ \begin{array}{lll} e_1 & \{ \; \texttt{action}_1 & \boxed{\texttt{yybegin}(\texttt{state}_1);} \; \} \\ e_2 & \{ \; \texttt{action}_2 & \texttt{yybegin}(\texttt{state}_2); \; \} \\ & \ldots & \\ e_k & \{ \; \texttt{action}_k & \texttt{yybegin}(\texttt{state}_k); \; \} \end{array} \right.$$

- 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 { }
           }
```


Topic:

Syntactic Analysis

Syntactic Analysis

Token-Stream → Parser → Syntaxtree

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

Syntactic Analysis

Token-Stream → Parser → Syntaxtree

- Syntactic analysis tries to integrate Tokens into larger program units.
- Such units may possibliy be:
  - → Expressions;
  - → Statements;
  - → Conditional branches;
  - → loops; ...

Discussion:

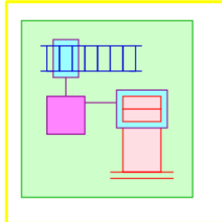
In general, parsers are not developed by hand, but generated from a specification:

Specification → **Generator** → Parser

Discussion:

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

$E \to E\{op\}E$ → **Generator** →



Specification of the hierarchical structure: contextfree grammars

Generated implementation: Pushdown automata + X

Chapter 1:

Basics of Contextfree Grammars

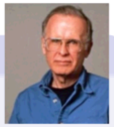
Basics: Context-free Grammars

- 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...


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:

$$\begin{array}{lll} S & \to & a\,S\,b \\ S & \to & \epsilon \end{array}$$

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:

$$\begin{array}{lll} S & \to & a\,S\,b \\ S & \to & \epsilon \end{array}$$

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: $a, b, c, ..., \texttt{int}, \texttt{name}, ...;$

... a practical example:

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

... a practical example:

$$
\begin{array}{lcl}
S & \to & \langle\text{stmt}\rangle \\
\langle\text{stmt}\rangle & \to & \langle\text{if}\rangle^{0} \;\mid\; \langle\text{while}\rangle^{1} \;\mid\; \langle\text{rexp}\rangle;^{2} \\
\langle\text{if}\rangle & \to & \text{if}\ (\ \langle\text{rexp}\rangle\ )\ \langle\text{stmt}\rangle\ \text{else}\ \langle\text{stmt}\rangle \\
\langle\text{while}\rangle & \to & \text{while}\ (\ \langle\text{rexp}\rangle\ )\ \langle\text{stmt}\rangle \\
\langle\text{rexp}\rangle & \to & \text{int} \;\mid\; \langle\text{lexp}\rangle \;\mid\; \langle\text{lexp}\rangle = \langle\text{rexp}\rangle \;\mid\; \dots \\
\langle\text{lexp}\rangle & \to & \text{name} \;\mid\; \dots
\end{array}
$$

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 $\boxed{(A, j)}$ (with $j \geq 0$).

$(\langle\text{stmt}\rangle, 1) \;\hat{=}\; \langle\text{stmt}\rangle \to \langle\text{while}\rangle$

Pair of grammars:

$$
\begin{array}{lcl}
E & \to & E{+}E \;\mid\; E{*}E \;\mid\; (\ E\ ) \;\mid\; \text{name} \;\mid\; \text{int}
\end{array}
$$

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

Both grammars describe the same language

Pair of grammars:

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

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

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: $\underline{E}$

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: $\underline{E} \quad \to \quad \underline{E} + T$

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:

$$\begin{array}{lll} \underline{E} & \to & \underline{E} + T \\ & \to & \underline{T} + T \end{array}$$

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:

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

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:

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

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:

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

$T \to T * F$

Definition

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

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

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:

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

Definition

The derivation 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: $\boxed{\to^*}$

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:

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

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.