Script generated by TTT

Title: Simon: Compilerbau (29.04.2013)

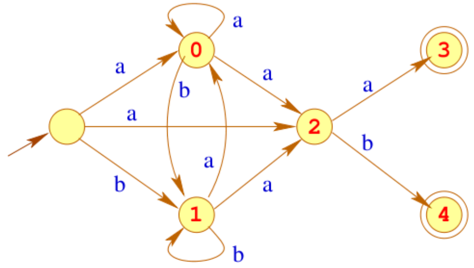
Date: Mon Apr 29 14:17:08 CEST 2013

Duration: 94:24 min

Pages: 63

Berry-Sethi Approach

... for example:



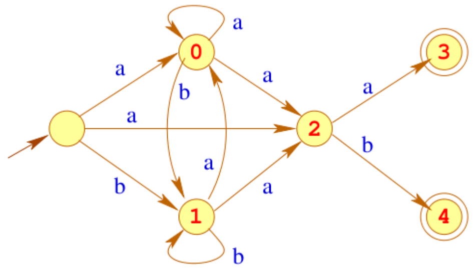
Remarks:

- 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

Berry-Sethi Approach

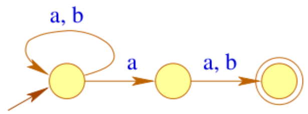
... for example:



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...

The expected outcome:



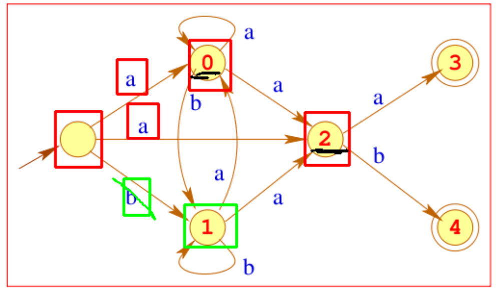
Remarks:

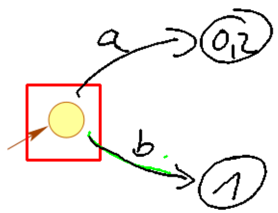
- ingoing edges do not necessarily have the same label here
- but Berry-Sethi is rather directly constructed
- Anyway, we need a deterministic technique

⇒ Powerset-Construction

Powerset Construction

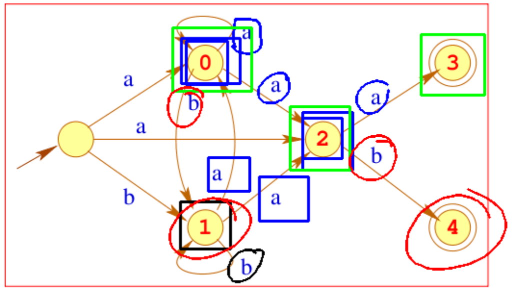
... for example:

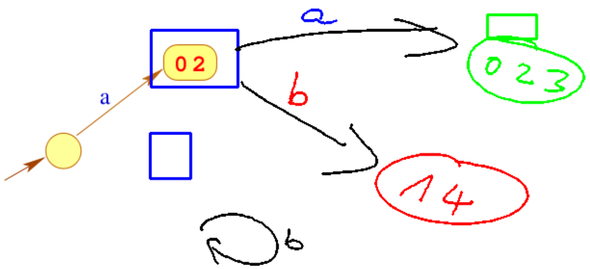




Powerset Construction

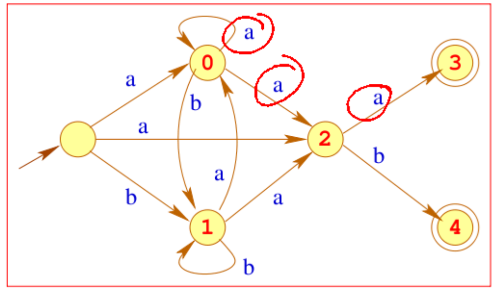
... for example:

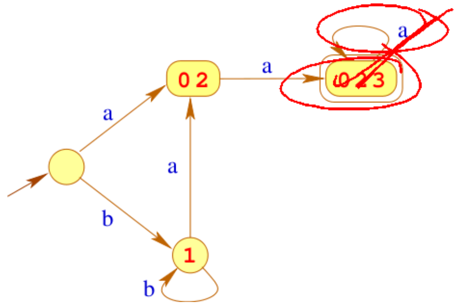




Powerset Construction

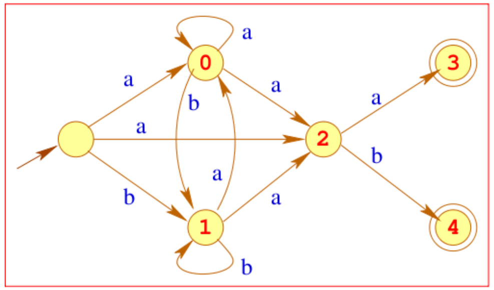
... for example:

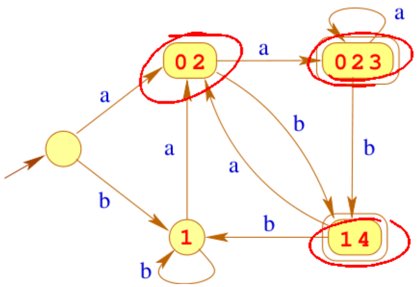




Powerset Construction

... 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

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

Theorem:

For every non-deterministic automaton $A = (Q, \Sigma, \delta, I, F)$ we can compute a deterministic automaton $\mathcal{P}(A)$ with

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

48 / 150

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

Bummer!
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 $Q_\mathcal{P}$
- Even though, the resulting automaton can become enormously huge
  ... which is (sort of) not happening in practice

Powerset Construction

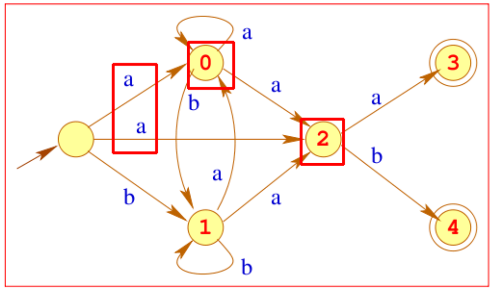
Bummer!
There are exponentially many powersets of Q

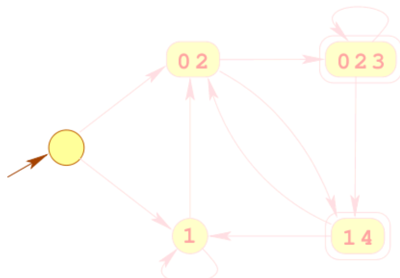
- Idea: Consider only contributing powersets. Starting with the set $Q_\mathcal{P} = \{I\}$ we only add further states by need ...
- i.e., whenever we can reach them from a state in $Q_\mathcal{P}$
- Even though, the resulting automaton can become enormously huge
  ... which is (sort of) not happening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

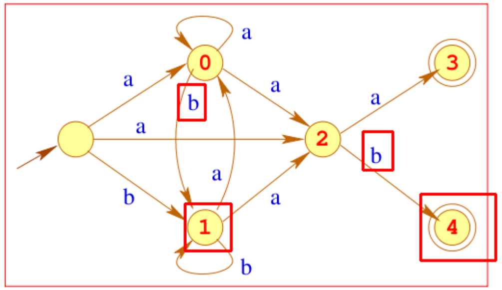


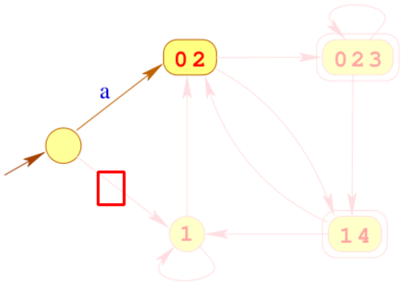


Powerset Construction

... for example:

a	b	a	b

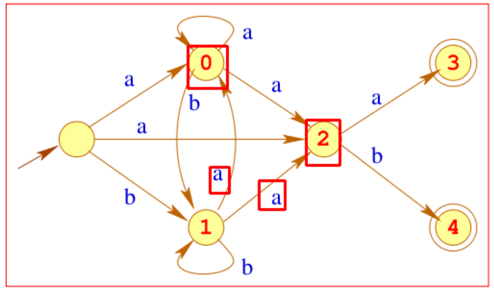


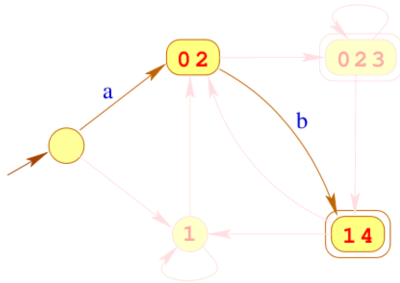


Powerset Construction

... for example:

a	b	a	b

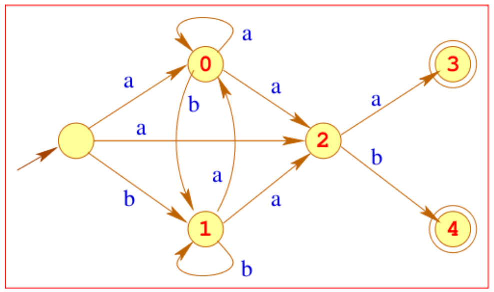


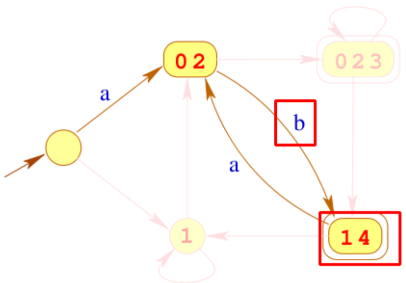


Powerset Construction

... for example:

a	b	a	b

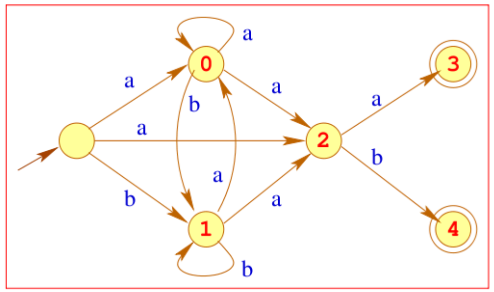


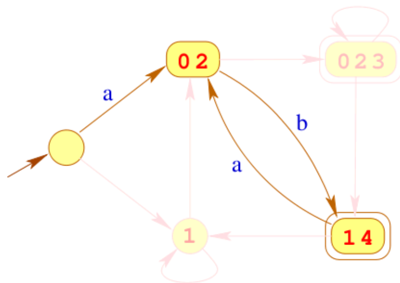


Powerset Construction

... 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:

e_1 { action$_1$ } → Special information

e_2 { action$_2$ }

...

e_k { action$_k$ }

Scanner design

Input (simplified): a set of rules:

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

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:

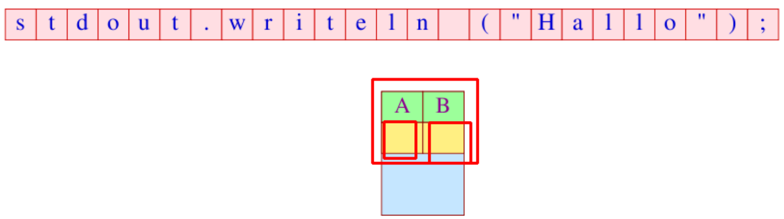
$$\begin{array}{lcl} F_1 & = & \{q \in F \mid q \cap \textsf{last}[e_1] \neq \emptyset\} \\ F_2 & = & \{q \in (F \backslash F_1) \mid q \cap \textsf{last}[e_2] \neq \emptyset\} \\ & \ldots & \\ F_k & = & \{q \in (F \backslash (F_1 \cup \ldots \cup F_{k-1})) \mid q \cap \textsf{last}[e_k] \neq \emptyset\}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

- 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):

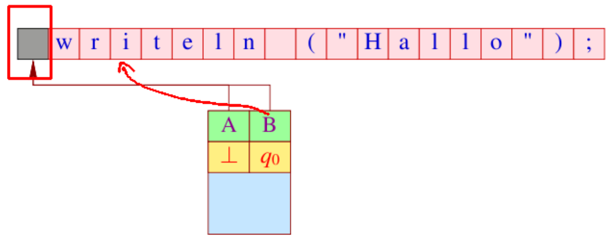
- The scanner manages two pointers $\langle A, B\rangle$ and the related states $\langle q_A, q_B\rangle$...
- Pointer A points to the last position in the input, after which a state $q_A \in F$ was reached;
- Pointer B tracks the current position.



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.

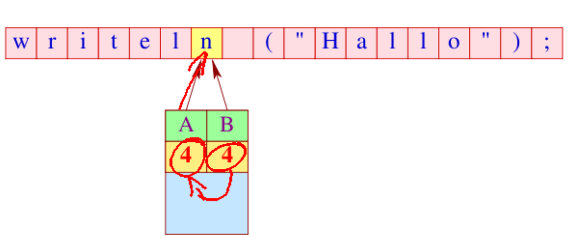


Implementation:

Idea (cont'd):

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

$$B := A; \qquad A := \bot;$$
$$q_B := q_0; \qquad q_A := \bot$$

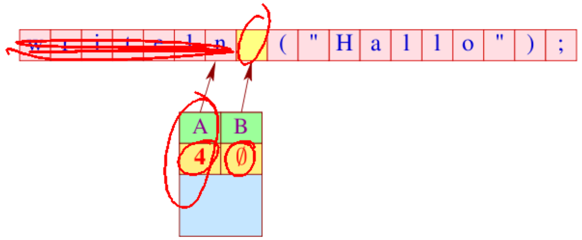


Implementation:

Idea (cont'd):

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

$$B := A; \qquad A := \bot;$$
$$q_B := q_0; \qquad q_A := \bot$$

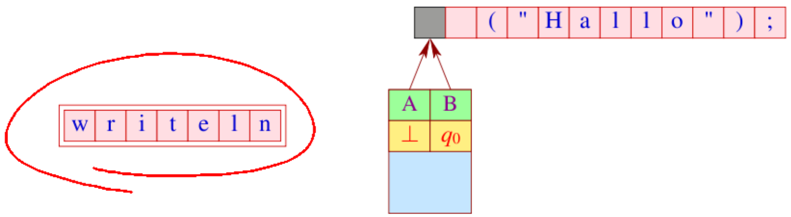


Implementation:

Idea (cont'd):

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

$$B := A; \qquad A := \bot;$$
$$q_B := q_0; \qquad q_A := \bot$$



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.

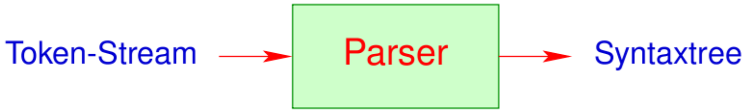
Topic:

Syntactic Analysis

Syntactic Analysis

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

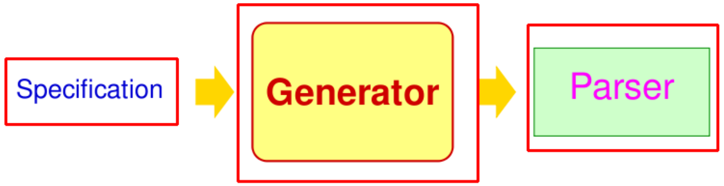
Syntactic Analysis



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

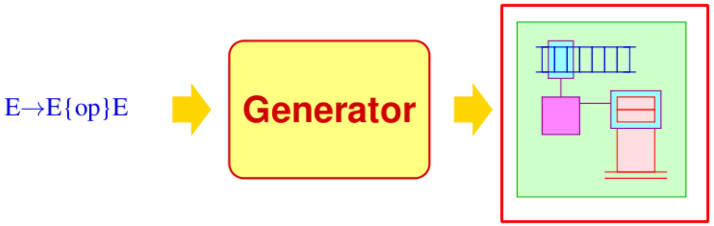
Discussion:

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



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

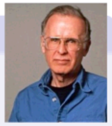
Chapter 1:

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...

Noam Chomsky



John Backus

Definition:

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:

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

Conventions

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

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

... for example:

$$S \rightarrow a\,S\,b$$
$$S \rightarrow \epsilon$$

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

Conventions

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

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

... for example:

$$S \rightarrow a\,S\,b$$
$$S \rightarrow \epsilon$$

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

Conventions:

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

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

... further examples:

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

... further examples:

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

Further 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 $j \geq 0$).

further 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

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

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:

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

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:

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

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:

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

Definition

A derivation $\to$ 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

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:

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

Definition

A derivation $\to$ 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 $A \to \beta \in P$

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

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:

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

Attention:

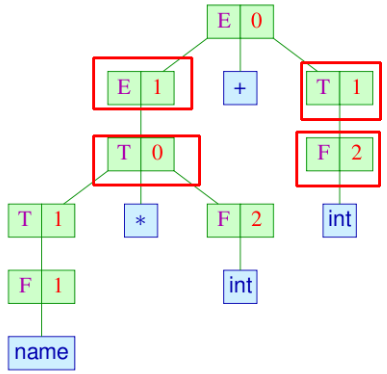
The order, in which disjunct fragments are rewritten is not relevant.

Derivation tree

Derivations of a symbol are represented as derivation tree:

... for example:

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



A derivation tree for $A \in N$:

inner nodes: rule applications

root: rule application for A

leaves: terminals or ϵ

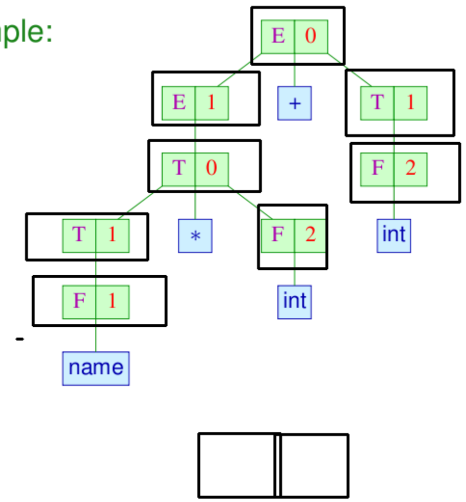
Special Derivations

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 correspond 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

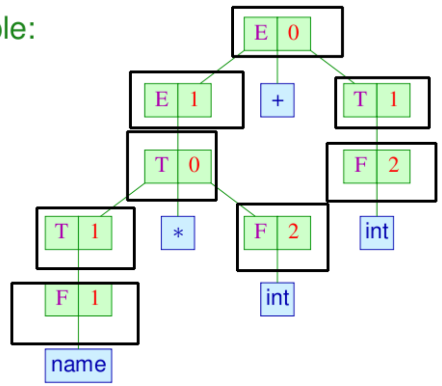
Special Derivations

... for example:



Special Derivations

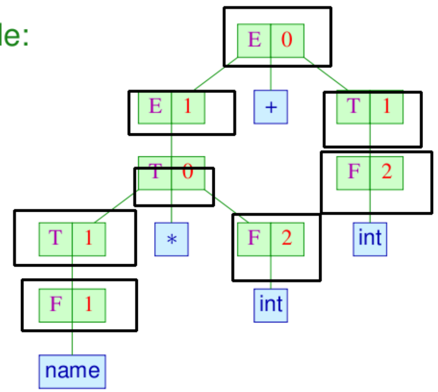
... for example:



Leftmost derivation:
$(E,0)\ (E,1)\ (T,0)\ (T,1)\ (F,1)\ (F,2)\ (T,1)\ (F,2)$
Rightmost derivation:
$(E,0)\ (T,1)\ (F,2)\ (E,1)\ (T,0)\ (F,2)\ (T,1)\ (F,1)$

Special Derivations

... for example:

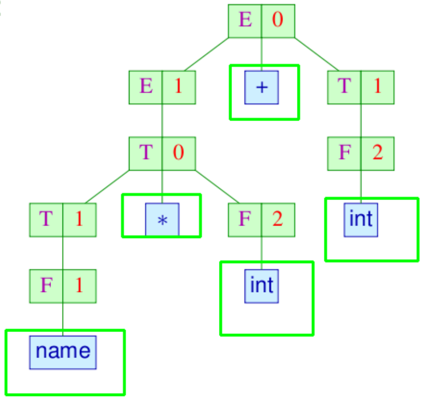


Leftmost derivation:
$(E,0)\ (E,1)\ (T,0)\ (T,1)\ (F,1)\ (F,2)\ (T,1)\ (F,2)$
Rightmost derivation:
$(E,0)\ (T,1)\ (F,2)\ (E,1)\ (T,0)\ (F,2)\ (T,1)\ (F,1)$
Reverse rightmost derivation:
$(F,1)\ (T,1)\ (F,2)\ (T,0)\ (E,1)\ (F,2)\ (T,1)\ (E,0)$

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:

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

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

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 intrerest.
- 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