Script **generated by TTT**

Title: Petter: Compilerbau (04.05.2015)

Date: Mon May 04 14:20:08 CEST 2015

Duration: 95:37 min

Pages: 62

Chapter 5:
Scanner design

Scanner design

Input (simplified): a set of rules:

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

Scanner design

Input (simplified): a set of rules:

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

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:

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

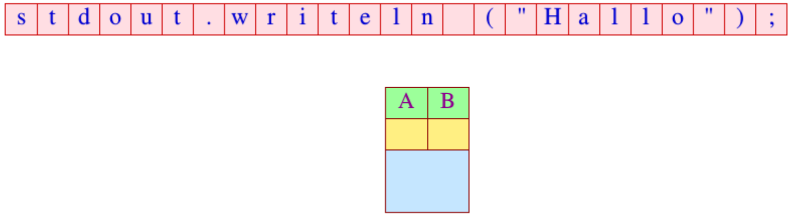
- For input w we find: $\delta^*(q_0, w) \in \boxed{F_{\boxed{i}}}$ iff the scanner must execute $\boxed{\texttt{action}_i}$ for w

---

Implementation:

Idea (cont'd):

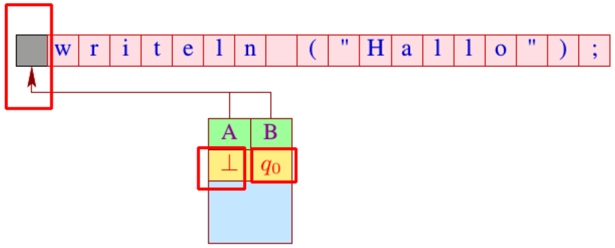
- The scanner manages two pointers $\langle A, B \rangle$ and the related states $\langle q_A, q_B \rangle$...
- Pointer A points to the last position in the input, after which a state $q_A \in F$ was reached;
- Pointer B tracks the current position.



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.

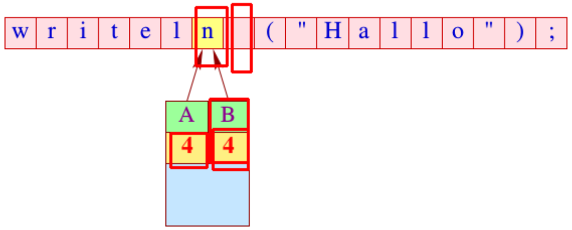


Implementation:

Idea (cont'd):

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

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

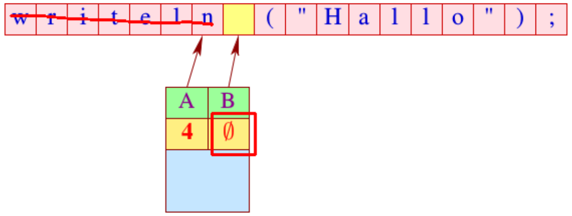


Implementation:

Idea (cont'd):

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

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

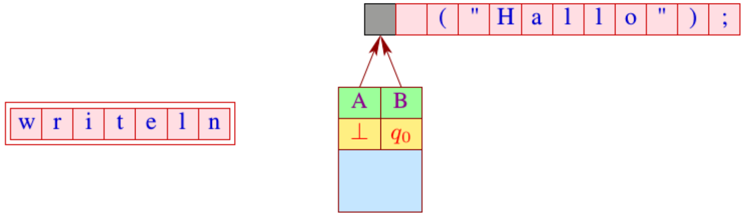


Implementation:

Idea (cont'd):

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

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



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:

$$\begin{array}{llll} \langle\texttt{state}\rangle & \{ & e_1 & \{\ \texttt{action}_1\quad \texttt{yybegin}(\texttt{state}_1);\ \} \\ & & e_2 & \{\ \texttt{action}_2\quad \texttt{yybegin}(\texttt{state}_2);\ \} \\ & & \ldots & \\ & & e_k & \{\ \texttt{action}_k\quad \texttt{yybegin}(\texttt{state}_k);\ \} \\ & \} & & \end{array}$$

- 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

Token-Stream → Parser → Syntaxtree

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

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

Syntactic Analysis

Chapter 1:

Basics of Contextfree Grammars

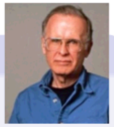
Basics: Context-free Grammars

- 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:

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

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

Conventions

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

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

... for example:

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

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, ..., int, name, ...;

... a practical example:

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

... a practical example:

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

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 (A, j) (with $j \geq 0$).

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: $\underline{E}$

Pair of grammars:

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

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

Both grammars describe the same language

Derivation

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

... for example: $\underline{E} \rightarrow \underline{E} + T$

Derivation

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

... for example:

$$\begin{array}{lcl} \underline{E} & \rightarrow & \underline{E} + T \\ & \rightarrow & \underline{T} + T \end{array}$$

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:

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

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:

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

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:

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

Definition

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

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

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:

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

Definition

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

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

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

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:

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

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:

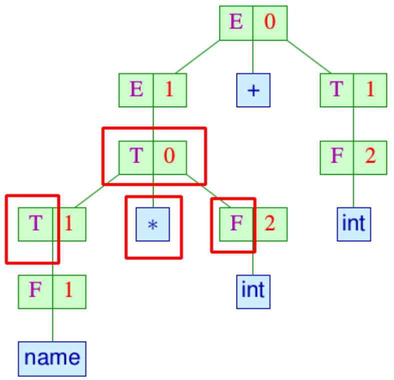
The order, in which disjunct fragments are rewritten is not relevant.

Derivation Tree

Derivations of a symbol are represented as derivation 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 $\boxed{(B, i)}$ correspond to right hand sides of the rule

Special Derivations

Attention:

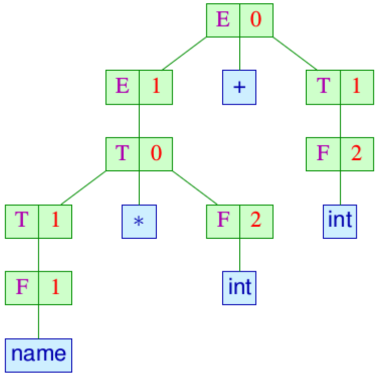
In contrast to arbitrary derivations, we find special ones, always rewriting the leftmost (or rather rightmost) occurance of a nonterminal.

- 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

---

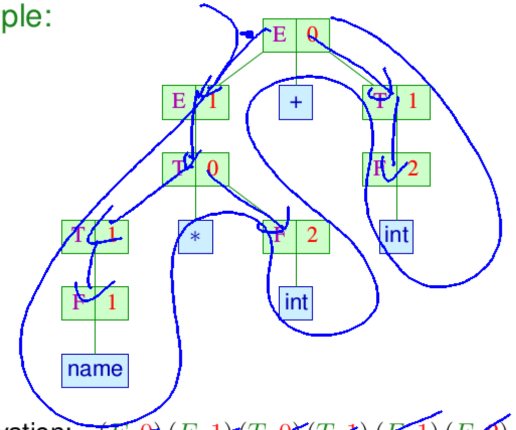
Special Derivations

... for example:



Special Derivations

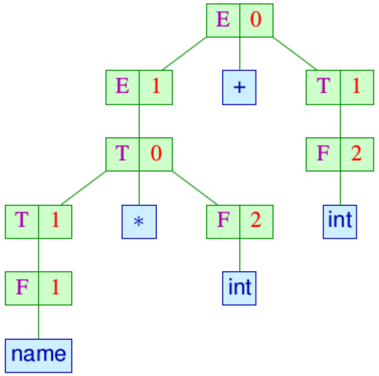
... for example:



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

Special Derivations

... for example:



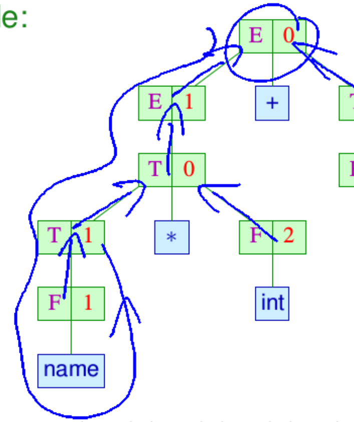
Leftmost derivation: $(E,0)\,(E,1)\,(T,0)\,(T,1)\,(F,1)\,(F,2)\,(T,1)\,(F,2)$

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:

$(E,0)\ (T,1)\ (F,2)\ (E,1)\ (T,0)\ (F,2)\ (T,1)\ (F,1)$

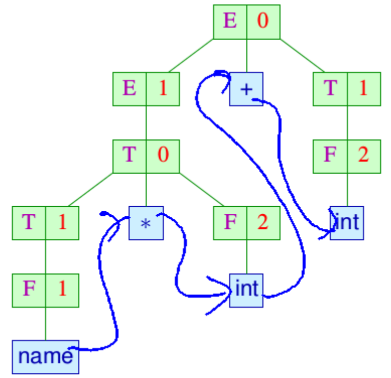
Reverse rightmost derivation:

~~$(F,1)$~~ ~~$(T,1)$~~ ~~$(F,2)$~~ ~~$(T,0)$~~ ~~$(E,1)$~~ ~~$(F,2)$~~ ~~$(T,1)$~~ ~~$(E,0)$~~

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:

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

$E \to E{+}T^{\,0} \mid T^{\,1}$

$T \to T{*}F^{\,0} \mid F^{\,1}$

$F \to (\,E\,)^{\,0} \mid \text{name}^{\,1} \mid \text{int}^{\,2}$

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.

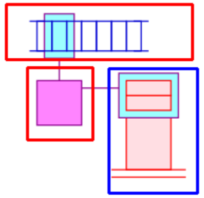
- Leftmost derivations correspond to a top-down reconstruction of the syntax tree.
- Reverse rightmost derivations correspond to a bottom-up reconstruction of the syntax tree.

Chapter 2:

Basics of Pushdown Automata

Basics of Pushdown Automata

Languages, specified by context free grammars are accepted by Pushdown Automata:



The pushdown is used e.g. to verify correct nesting of braces.

Example:

States: $0, 1, 2$
Start state: 0
Final states: $0, 2$

0	a	11
1	a	11
11	b	2
12	b	2

Example:

States: $0, 1, 2$
Start state: 0
Final states: $0, 2$

0	a	11
1	a	11
11	b	2
12	b	2

Conventions:

- We do not differentiate between pushdown symbols and states
- The rightmost / upper pushdown symbol represents the state
- Every transition consumes / modifies the upper part of the pushdown


Friedrich Bauer


Klaus Samelson

Definition: Pushdown Automaton

A pushdown automaton (PDA) is a tuple $M = (Q, T, \delta, q_0, F)$ with:

- Q a finite set of states;
- T an input alphabet;
- $q_0 \in Q$ the start state;
- $F \subseteq Q$ the set of final states and
- $\delta \subseteq Q^+ \times (T \cup \{\epsilon\}) \times Q^*$ a finite set of transitions

---


Friedrich Bauer


Klaus Samelson

Definition: Pushdown Automaton

A pushdown automaton (PDA) is a tuple $M = (Q, T, \delta, q_0, F)$ with:

- Q a finite set of states;
- T an input alphabet;
- $q_0 \in Q$ the start state;
- $F \subseteq Q$ the set of final states and
- $\delta \subseteq Q^+ \times (T \cup \{\epsilon\}) \times Q^*$ a finite set of transitions

We define computations of pushdown automata with the help of transitions; a particular computation state (the current configuration) is a pair:

$$(\gamma, w) \in Q^* \times T^*$$

consisting of the pushdown content and the remaining input.

... for example:

States: $0, 1, 2$
Start state: 0
Final states: $0, 2$

0	a	11
1	a	11
11	b	2
12	b	2

$$(0\,, \quad a\,a\,a\,b\,b\,b)$$

... for example:

States: $0, 1, 2$
Start state: 0
Final states: $0, 2$

0	a	11
1	a	11
11	b	2
12	b	2

$$(0\,, \quad a\,a\,a\,b\,b\,b) \quad \vdash \quad (1\,1\,, \quad a\,a\,b\,b\,b)$$

... for example:

States: $0, 1, 2$
Start state: 0
Final states: $0, 2$

0	a	11
1	a	11
11	b	2
12	b	2

$$
\begin{array}{rcr}
(0,\ aaabbb) & \vdash & (11,\ aabbb) \\
 & \vdash & (111,\ abbb)
\end{array}
$$

... for example:

States: $0, 1, 2$
Start state: 0
Final states: $0, 2$

0	a	11
1	a	11
11	b	2
12	b	2

$$
\begin{array}{rcr}
(0,\ aaabbb) & \vdash & (11,\ aabbb) \\
 & \vdash & (111,\ abbb) \\
 & \vdash & (1111,\ bbb)
\end{array}
$$

... for example:

States: $0, 1, 2$
Start state: 0
Final states: $0, 2$

0	a	11
1	a	11
11	b	2
12	b	2

$$
\begin{array}{rcr}
(0,\ aaabbb) & \vdash & (11,\ aabbb) \\
 & \vdash & (111,\ abbb) \\
 & \vdash & (1111,\ bbb) \\
 & \vdash & (112,\ bb)
\end{array}
$$

... for example:

$\{a^n b^n \mid n \geq 0\}$

States: $0, 1, 2$
Start state: 0
Final states: $0, 2$

0	a	11
1	a	11
11	b	~~2~~ 12
1~~2~~	b	~~2~~ ε

$$
\begin{array}{rcr}
(0,\ aaabbb) & \vdash & (11,\ aabbb) \\
 & \vdash & (111,\ abbb) \\
 & \vdash & (1111,\ bbb) \\
 & \vdash & (112,\ bb) \\
 & \vdash & (12,\ b) \\
 & \vdash & (2,\ \epsilon)
\end{array}
$$

A computation step is characterized by the relation $\vdash \;\subseteq\; (Q^* \times T^*)^2$ with

$$(\alpha\,\gamma,\, \boxed{x}\, w) \vdash (\alpha\,\gamma',\, \boxed{w}) \quad \text{for} \quad (\gamma,\, \boxed{x},\, \gamma') \in \delta$$

A computation step is characterized by the relation $\vdash \;\subseteq\; (Q^* \times T^*)^2$ with

$$(\alpha\,\gamma,\, x\, w) \vdash (\alpha\,\gamma',\, w) \quad \text{for} \quad (\gamma,\, x,\, \gamma') \in \delta$$

Remarks:

- The relation $\vdash$ depends of the pushdown automaton M
- The reflexive and transitive closure of $\vdash$ is denoted by $\vdash^*$
- Then, the language accepted by M is

$$\mathcal{L}(M) \;=\; \boxed{\{w \in T^*} \mid \exists\, f \in F :\; \boxed{(q_0},\boxed{w)}\boxed{\vdash^*}\boxed{(f},\boxed{\epsilon)}\}$$

A computation step is characterized by the relation $\vdash \;\subseteq\; (Q^* \times T^*)^2$ with

$$(\alpha\,\gamma,\, x\, w) \vdash (\alpha\,\gamma',\, w) \quad \text{for} \quad (\gamma,\, x,\, \gamma') \in \delta$$

Remarks:

- The relation $\vdash$ depends of the pushdown automaton M
- The reflexive and transitive closure of $\vdash$ is denoted by $\vdash^*$
- Then, the language accepted by M is

$$\mathcal{L}(M) \;=\; \{w \in T^* \mid \exists\, f \in F :\; (q_0, w) \vdash^* (f, \epsilon)\}$$

We accept with a final state together with empty input.