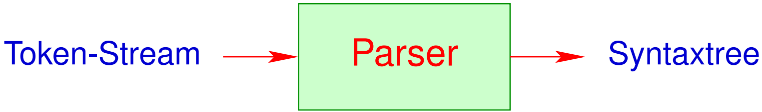
Topic:

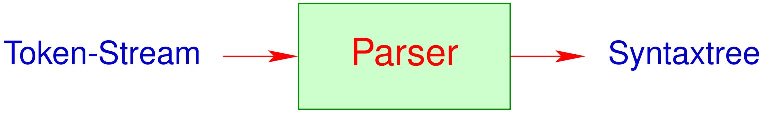
Syntactic Analysis

Syntactic Analysis



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

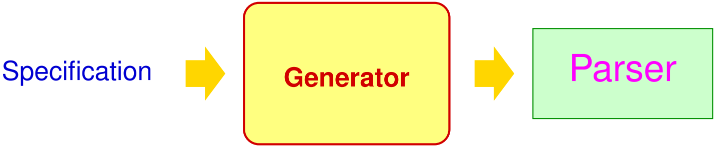
Syntactic Analysis



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

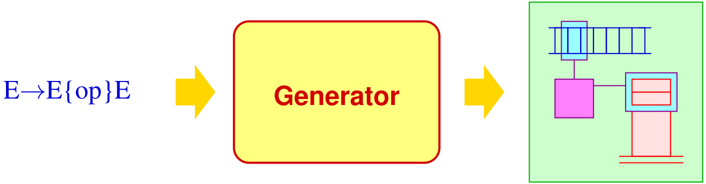
Discussion:

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

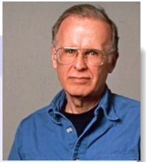
Basics: Context-free Grammars

- 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...

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:

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

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:

$$\begin{array}{lcl} S & \rightarrow & a\,S\,b \\ S & \rightarrow & \epsilon \end{array}$$

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

Conventions

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

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

... for example:

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

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

Conventions:

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

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

... a practical example:

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

... a practical example:

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

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:

$$E \rightarrow E+E \mid E*E \mid (\ E\ ) \mid \text{name} \mid \text{int}$$

$$\begin{array}{lcllll} E & \rightarrow & E+T & \mid & T & \\ T & \rightarrow & T*F & \mid & F & \\ F & \rightarrow & (\ E\ ) & \mid & \text{name} & \mid \text{int} \end{array}$$

Both grammars describe the same language

Pair of grammars:

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

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

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.

$$\underline{E}$$

... for example:

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:

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

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:

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

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:

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

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:

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

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:

$$\begin{array}{lcl} \underline{E} & \to & \underline{E} + T \\ & \to & \underline{T} + T \\ & \to & T * \underline{F} + T \\ & \to & \underline{T} * \textsf{int} + T \\ & \to & \underline{F} * \textsf{int} + T \end{array}$$

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:

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

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:

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

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:

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

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:

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

Definition

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

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

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

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: $\quad \to^*$

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:

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

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:

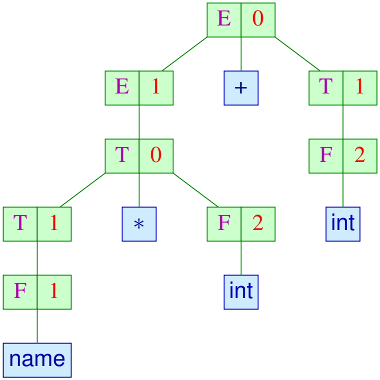
The order, in which disjunct fragments are rewritten is not relevant.

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

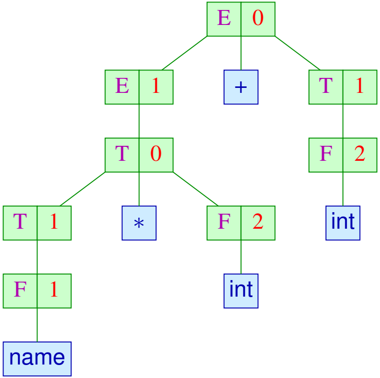
Attention:

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

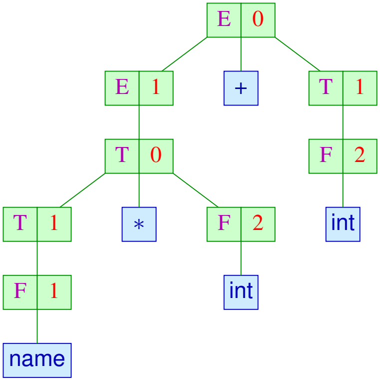
Special Derivations

... for example:



Special Derivations

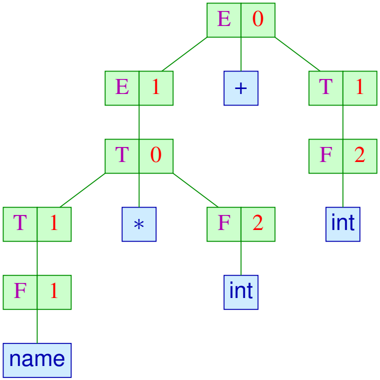
... for example:



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

Special Derivations

... for example:

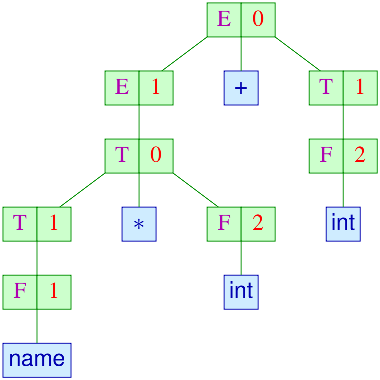


Leftmost derivation: $(E,0)\ (E,1)\ (T,0)\ (T,1)\ (F,1)\ (F,2)\ (T,1)\ (F,2)$

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

Special Derivations

... for example:



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

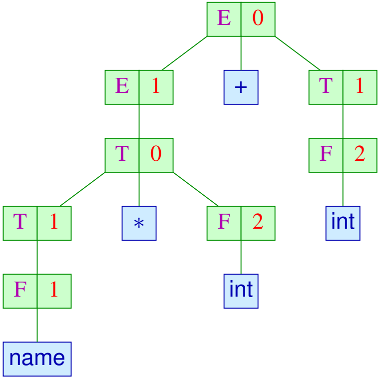
Rightmost derivation: $(E,0)\ (T,1)\ (F,2)\ (E,1)\ (T,0)\ (F,2)\ (T,1)\ (F,1)$

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 $\text{yield}(t)$.

... for example:



gives rise to the concatenation: $\text{name} * \text{int} + \text{int}$.

Unique Grammars

Definition:

Grammar G is called unique, if for every $w \in T^*$ there is maximally one derivation tree t of S with $\mathsf{yield}(t) = w$.

... in our example:

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

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

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.

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.

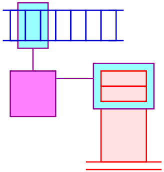
- Leftmost derivations correspond to a top-down reconstruction of the syntax tree.
- Reverse rightmost derivations correspond to a bottom-up reconstruction of the syntax tree.

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: $0, 2$

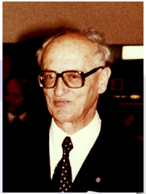
0	a	11
1	a	11
11	b	2
12	b	2

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

Definition: Pushdown Automaton

Friedrich Bauer



Klaus Samelson

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

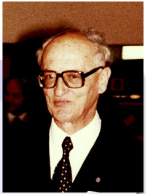
- Q a finite set of states;
- T an input alphabet;
- $q_0 \in Q$ the start state;
- $F \subseteq Q$ the set of final states and
- $\delta \subseteq Q^+ \times (T \cup \{\epsilon\}) \times Q^*$ a finite set of transitions

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

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

... 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

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

... 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

$$\begin{array}{rcl} (0\,, \quad a\,a\,a\,b\,b\,b) & \vdash & (1\,1\,, \quad a\,a\,b\,b\,b) \\ & \vdash & (1\,1\,1\,, \quad a\,b\,b\,b) \end{array}$$

... 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

$$
\begin{array}{rcrl}
(0\,, & a\,a\,a\,b\,b\,b) & \vdash & (1\,1\,, \quad a\,a\,b\,b\,b) \\
 & & \vdash & (1\,1\,1\,, \quad a\,b\,b\,b) \\
 & & \vdash & (1\,1\,1\,1\,, \quad b\,b\,b)
\end{array}
$$

... 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

$$
\begin{aligned}
(0\,,\quad a\,a\,a\,b\,b\,b) &\vdash (1\,1\,,\quad a\,a\,b\,b\,b) \\
&\vdash (1\,1\,1\,,\quad a\,b\,b\,b) \\
&\vdash (1\,1\,1\,1\,,\quad b\,b\,b) \\
&\vdash (1\,1\,2\,,\quad b\,b)
\end{aligned}
$$

... 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

$$
\begin{array}{rcrl}
(0\,, \quad a\,a\,a\,b\,b\,b) & \vdash & (1\,1\,, & a\,a\,b\,b\,b) \\
 & \vdash & (1\,1\,1\,, & a\,b\,b\,b) \\
 & \vdash & (1\,1\,1\,1\,, & b\,b\,b) \\
 & \vdash & (1\,1\,2\,, & b\,b) \\
 & \vdash & (1\,2\,, & b)
\end{array}
$$

... 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

$$
\begin{array}{rcrl}
(0\,, & a\,a\,a\,b\,b\,b) & \vdash & (1\,1\,, \quad a\,a\,b\,b\,b) \\
 & & \vdash & (1\,1\,1\,, \quad a\,b\,b\,b) \\
 & & \vdash & (1\,1\,1\,1\,, \quad b\,b\,b) \\
 & & \vdash & (1\,1\,2\,, \quad b\,b) \\
 & & \vdash & (1\,2\,, \quad b) \\
 & & \vdash & (2\,, \quad \epsilon)
\end{array}
$$

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

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

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

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

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 on 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.

Definition: Deterministic Pushdown Automaton

The pushdown automaton M is deterministic, if every configuration has maximally one successor configuration.

This is exactly the case if for distinct transitions (γ_1, x, γ_2), $(\gamma_1', x', \gamma_2') \in \delta$ we can assume:
Is γ_1 a suffix of γ_1', then $x \neq x' \wedge x \neq \epsilon \neq x'$ is valid.

Definition: Deterministic Pushdown Automaton

The pushdown automaton M is deterministic, if every configuration has maximally one successor configuration.

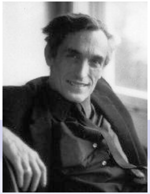
This is exactly the case if for distinct transitions (γ_1, x, γ_2), $(\gamma_1', x', \gamma_2') \in \delta$ we can assume:
Is γ_1 a suffix of γ_1', then $x \neq x' \;\wedge\; x \neq \epsilon \neq x'$ is valid.

... for example:

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

... this obviously holds

Pushdown Automata



M. Schützenberger

A. Öttinger

Theorem:

For each context free grammar $G = (N, T, P, S)$
a pushdown automaton M with $\mathcal{L}(G) = \mathcal{L}(M)$ can be built.

The theorem is so important for us, that we take a look at two constructions for automata, motivated by both of the special derivations:

- M_G^L to build Leftmost derivations
- M_G^R to build reverse Rightmost derivations

Chapter 3:

Top-down Parsing

Item Pushdown Automaton

Construction: Item Pushdown Automaton M_G^L

- Reconstruct a Leftmost derivation.
- Expand nonterminals using a rule.
- Verify successively, that the chosen rule matches the input.

$\Longrightarrow$ The states are now Items (= rules with a bullet):

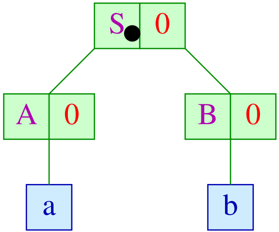
$$[A \to \alpha \bullet \beta]\,, \qquad A \to \alpha\,\beta \;\in\; P$$

The bullet marks the spot, how far the rule is already processed

Item Pushdown Automaton – Example

Our example:

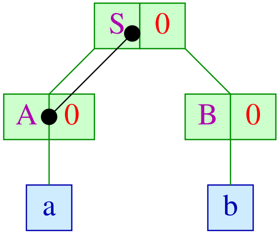
$$S \rightarrow A\,B^{0} \qquad A \rightarrow a^{0} \qquad B \rightarrow b^{0}$$



Item Pushdown Automaton – Example

Our example:

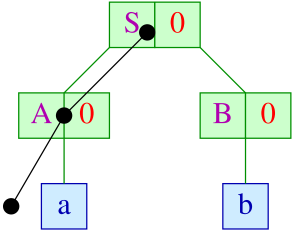
$$S \rightarrow A\,B^0 \qquad A \rightarrow a^0 \qquad B \rightarrow b^0$$



Item Pushdown Automaton – Example

Our example:

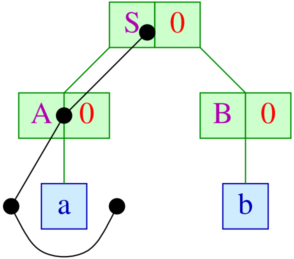
$$S \rightarrow A\,B^0 \qquad A \rightarrow a^0 \qquad B \rightarrow b^0$$



Item Pushdown Automaton – Example

Our example:

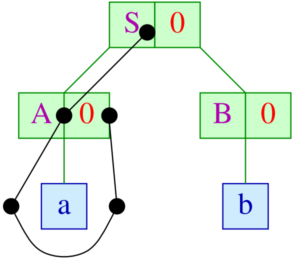
$$S \rightarrow A\,B^0 \qquad A \rightarrow a^0 \qquad B \rightarrow b^0$$



Item Pushdown Automaton – Example

Our example:

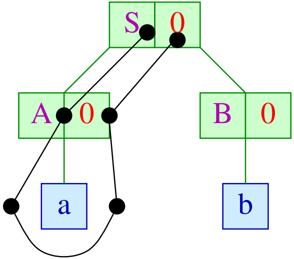
$$S \rightarrow A\,B^0 \qquad A \rightarrow a^0 \qquad B \rightarrow b^0$$



Item Pushdown Automaton – Example

Our example:

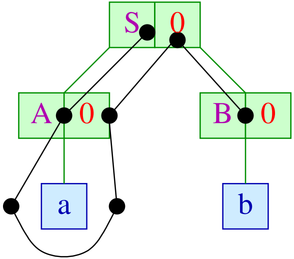
$$S \rightarrow A\,B^0 \qquad A \rightarrow a^0 \qquad B \rightarrow b^0$$



Item Pushdown Automaton – Example

Our example:

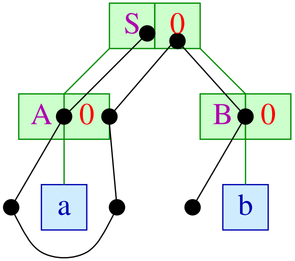
$$S \rightarrow A\,B^0 \qquad A \rightarrow a^0 \qquad B \rightarrow b^0$$



Item Pushdown Automaton – Example

Our example:

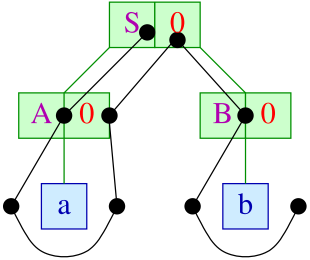
$$S \rightarrow A\,B^0 \qquad A \rightarrow a^0 \qquad B \rightarrow b^0$$



Item Pushdown Automaton – Example

Our example:

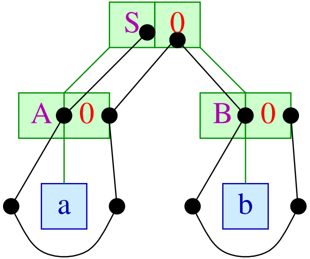
$$S \rightarrow A\,B^0 \qquad A \rightarrow a^0 \qquad B \rightarrow b^0$$



Item Pushdown Automaton – Example

Our example:

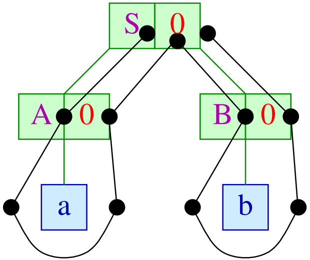
$$S \rightarrow A\,B^0 \qquad A \rightarrow a^0 \qquad B \rightarrow b^0$$



Item Pushdown Automaton – Example

Our example:

$$S \rightarrow A\,B^0 \qquad A \rightarrow a^0 \qquad B \rightarrow b^0$$



Item Pushdown Automaton – Example

We add another rule $S' \to S\,\$$ for initialising the construction:

Start state: $[S' \to \bullet\ S\ \$]$
End state: $[S' \to S\ \bullet\ \$]$
Transition relations:

$[S' \to \bullet\ S\ \$]$	ϵ	$[S' \to \bullet\ S\ \$]\ [S \to \bullet\ A\,B]$
$[S \to \bullet\ A\,B]$	ϵ	$[S \to \bullet\ A\,B]\ [A \to \bullet\ a]$
$[A \to \bullet\ a]$	a	$[A \to a\,\bullet]$
$[S \to \bullet\ A\,B]\ [A \to a\,\bullet]$	ϵ	$[S \to A\ \bullet\ B]$
$[S \to A\ \bullet\ B]$	ϵ	$[S \to A\ \bullet\ B]\ [B \to \bullet\ b]$
$[B \to \bullet\ b]$	b	$[B \to b\,\bullet]$
$[S \to A\ \bullet\ B]\ [B \to b\,\bullet]$	ϵ	$[S \to A\,B\,\bullet]$
$[S' \to \bullet\ S\ \$]\ [S \to A\,B\,\bullet]$	ϵ	$[S' \to S\ \bullet\ \$]$

Item Pushdown Automaton

The item pushdown automaton M_G^L has three kinds of transitions:

Expansions: $([A \to \alpha \bullet B\,\beta], \epsilon, [A \to \alpha \bullet B\,\beta]\,[B \to \bullet\,\gamma])$ for $A \to \alpha\,B\,\beta,\ B \to \gamma \in P$

Shifts: $([A \to \alpha \bullet a\,\beta], a, [A \to \alpha\,a \bullet \beta])$ for $A \to \alpha\,a\,\beta \in P$

Reduces: $([A \to \alpha \bullet B\,\beta]\,[B \to \gamma \bullet], \epsilon, [A \to \alpha\,B \bullet \beta])$ for $A \to \alpha\,B\,\beta,\ B \to \gamma \in P$

Items of the form: $[A \to \alpha \bullet]$ are also called complete

The item pushdown automaton shifts the bullet around the derivation tree ...

Item Pushdown Automaton

Discussion:

- The expansions of a computation form a leftmost derivation
- Unfortunately, the expansions are chosen nondeterministically

- For proving correctness of the construction, we show that for every Item $[A \to \alpha \bullet B\,\beta]$ the following holds:

$$([A \to \alpha \bullet B\,\beta], w) \vdash^* ([A \to \alpha\,B \bullet \beta], \epsilon) \qquad \text{iff} \qquad B \to^* w$$

- LL-Parsing is based on the item pushdown automaton and tries to make the expansions deterministic ...

Item Pushdown Automaton

Example: $S' \to S\,\$ \qquad S \to \epsilon \mid a\,S\,b$

The transitions of the according Item Pushdown Automaton:

0	$[S' \to \bullet\, S\,\$]$	ϵ	$[S' \to \bullet\, S\,\$]\,[S \to \bullet]$
1	$[S' \to \bullet\, S\,\$]$	ϵ	$[S' \to \bullet\, S\,\$]\,[S \to \bullet\, a\,S\,b]$
2	$[S \to \bullet\, a\,S\,b]$	a	$[S \to a \bullet S\,b]$
3	$[S \to a \bullet S\,b]$	ϵ	$[S \to a \bullet S\,b]\,[S \to \bullet]$
4	$[S \to a \bullet S\,b]$	ϵ	$[S \to a \bullet S\,b]\,[S \to \bullet\, a\,S\,b]$
5	$[S \to a \bullet S\,b]\,[S \to \bullet]$	ϵ	$[S \to a\,S \bullet b]$
6	$[S \to a \bullet S\,b]\,[S \to a\,S\,b \bullet]$	ϵ	$[S \to a\,S \bullet b]$
7	$[S \to a\,S \bullet b]$	b	$[S \to a\,S\,b \bullet]$
8	$[S' \to \bullet\, S\,\$]\,[S \to \bullet]$	ϵ	$[S' \to S \bullet \$]$
9	$[S' \to \bullet\, S\,\$]\,[S \to a\,S\,b \bullet]$	ϵ	$[S' \to S \bullet \$]$

Item Pushdown Automaton

Example: $S' \to S\,\$ \qquad S \to \epsilon \mid a\,S\,b$

The transitions of the according Item Pushdown Automaton:

0	$[S' \to \bullet\, S\,\$]$	ϵ	$[S' \to \bullet\, S\,\$]\,[S \to \bullet]$
1	$[S' \to \bullet\, S\,\$]$	ϵ	$[S' \to \bullet\, S\,\$]\,[S \to \bullet\, a\,S\,b]$
2	$[S \to \bullet\, a\,S\,b]$	a	$[S \to a \bullet S\,b]$
3	$[S \to a \bullet S\,b]$	ϵ	$[S \to a \bullet S\,b]\,[S \to \bullet]$
4	$[S \to a \bullet S\,b]$	ϵ	$[S \to a \bullet S\,b]\,[S \to \bullet\, a\,S\,b]$
5	$[S \to a \bullet S\,b]\,[S \to \bullet]$	ϵ	$[S \to a\,S \bullet b]$
6	$[S \to a \bullet S\,b]\,[S \to a\,S\,b \bullet]$	ϵ	$[S \to a\,S \bullet b]$
7	$[S \to a\,S \bullet b]$	b	$[S \to a\,S\,b \bullet]$
8	$[S' \to \bullet\, S\,\$]\,[S \to \bullet]$	ϵ	$[S' \to S \bullet \$]$
9	$[S' \to \bullet\, S\,\$]\,[S \to a\,S\,b \bullet]$	ϵ	$[S' \to S \bullet \$]$

Conflicts arise between the transitions $(0, 1)$ and $(3, 4)$, resp..

Topdown Parsing

Problem:

Conflicts between the transitions prohibit an implementation of the item pushdown automaton as deterministic pushdown automaton.

Topdown Parsing

Problem:

Conflicts between the transitions prohibit an implementation of the item pushdown automaton as deterministic pushdown automaton.

Idea 1: GLL Parsing

For each conflict, we create a virtual copy of the complete configuration and continue computing in parallel.

Topdown Parsing

Problem:

Conflicts between the transitions prohibit an implementation of the item pushdown automaton as deterministic pushdown automaton.

Idea 1: GLL Parsing

For each conflict, we create a virtual copy of the complete configuration and continue computing in parallel.

Idea 2: Recursive Descent & Backtracking

Depth-first search for an appropriate derivation.

Topdown Parsing

Problem:

Conflicts between the transitions prohibit an implementation of the item pushdown automaton as deterministic pushdown automaton.

Idea 1: GLL Parsing

For each conflict, we create a virtual copy of the complete configuration and continue computing in parallel.

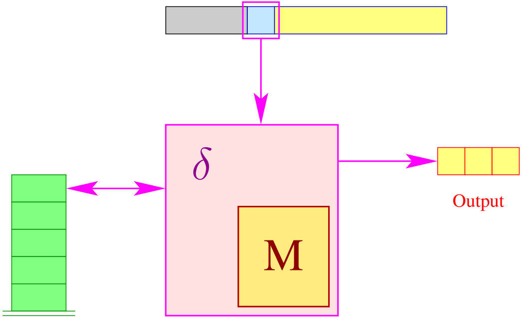
Idea 2: Recursive Descent & Backtracking

Depth-first search for an appropriate derivation.

Idea 3: Recursive Descent & Lookahead

Conflicts are resolved by considering a lookup of the next input symbols.

Structure of the $LL(1)$-Parser:



- The parser accesses a frame of length 1 of the input;
- it corresponds to an item pushdown automaton, essentially;
- table $M[q, w]$ contains the rule of choice.

Topdown Parsing

Idea:

- Emanate from the item pushdown automaton
- Consider the next input symbol to determine the appropriate rule for the next expansion
- A grammar is called $LL(1)$ if a unique choice is always possible

Topdown Parsing

Idea:

- Emanate from the item pushdown automaton
- Consider the next input symbol to determine the appropriate rule for the next expansion
- A grammar is called $LL(1)$ if a unique choice is always possible


Philip Lewis


Richard Stearns

Definition:

A reduced grammar is called $LL(1)$, if for each two distinct rules $A \to \alpha$, $A \to \alpha' \in P$ and each derivation $S \to_L^* u\, A\, \beta$ with $u \in T^*$ the following is valid:

$$\text{First}_1(\alpha\, \beta) \cap \text{First}_1(\alpha'\, \beta) = \emptyset$$

Topdown Parsing

Example 1:

$$\begin{array}{lcl} S & \rightarrow & \text{if (} E \text{) } S \text{ else } S \quad | \\ & & \text{while (} E \text{) } S \quad | \\ & & E \text{ ;} \\ E & \rightarrow & \text{id} \end{array}$$

is $LL(1)$, since $\text{First}_1(E) = \{\text{id}\}$

Topdown Parsing

Example 1:

$$
\begin{array}{lcl}
S & \rightarrow & \text{if (} E \text{) } S \text{ else } S \quad | \\
  &   & \text{while (} E \text{) } S \quad | \\
  &   & E \text{ ;} \\
E & \rightarrow & \text{id}
\end{array}
$$

is $LL(1)$, since $\text{First}_1(E) = \{\text{id}\}$

Example 2:

$$
\begin{array}{lcl}
S & \rightarrow & \text{if (} E \text{) } S \text{ else } S \quad | \\
  &   & \text{if (} E \text{) } S \quad | \\
  &   & \text{while (} E \text{) } S \quad | \\
  &   & E \text{ ;} \\
E & \rightarrow & \text{id}
\end{array}
$$

... is not $LL(k)$ for any $k > 0$.

Lookahead Sets

Definition: First$_1$-Sets

For a set $L \subseteq T^*$ we define:

$$\text{First}_1(L) \;=\; \{\epsilon \mid \epsilon \in L\} \cup \{u \in T \mid \exists\, v \in T^* \;:\; uv \in L\}$$

Example: $S \to \epsilon \quad | \quad a\,S\,b$

$\text{First}_1(\llbracket S \rrbracket)$
ϵ
$a\,b$
$a\,a\,b\,b$
$a\,a\,a\,b\,b\,b$
$\dots$

Lookahead Sets

Definition: First_1-Sets

For a set $L \subseteq T^*$ we define:

$$\mathsf{First}_1(L) \;=\; \{\epsilon \mid \epsilon \in L\} \cup \{u \in T \mid \exists\, v \in T^* : \; uv \in L\}$$

Example: $S \to \epsilon \quad | \quad a\,S\,b$

$\mathsf{First}_1(\llbracket S \rrbracket)$
ϵ
$a\,b$
$a\,a\,b\,b$
$a\,a\,a\,b\,b\,b$
$\ldots$

$\equiv$ the yield's prefix of length 1

Lookahead Sets

Arithmetics:

$\text{First}_1(\_)$ is distributive with union and concatenation:

$$\begin{array}{lcl}
\text{First}_1(\emptyset) & = & \emptyset \\
\text{First}_1(L_1 \cup L_2) & = & \text{First}_1(L_1) \cup \text{First}_1(L_2) \\
\text{First}_1(L_1 \cdot L_2) & = & \text{First}_1(\text{First}_1(L_1) \cdot \text{First}_1(L_2)) \\
 & := & \text{First}_1(L_1) \odot_1 \text{First}_1(L_2)
\end{array}$$

$\odot_1$ being $1-$ concatenation

Lookahead Sets

Arithmetics:

$\text{First}_1(\_)$ is distributive with union and concatenation:

$$\begin{array}{lll} \text{First}_1(\emptyset) & = & \emptyset \\ \text{First}_1(L_1 \cup L_2) & = & \text{First}_1(L_1) \cup \text{First}_1(L_2) \\ \text{First}_1(L_1 \cdot L_2) & = & \text{First}_1(\text{First}_1(L_1) \cdot \text{First}_1(L_2)) \\ & := & \text{First}_1(L_1) \odot_1 \text{First}_1(L_2) \end{array}$$

$\odot_1$ being $1 -$ concatenation

Definition: 1-concatenation

Let $L_1, L_2 \subseteq T \cup \{\epsilon\}$ with $L_1 \neq \emptyset \neq L_2$. Then:

$$L_1 \odot_1 L_2 = \begin{cases} L_1 & \text{if } \epsilon \notin L_1 \\ (L_1 \backslash \{\epsilon\}) \cup L_2 & \text{otherwise} \end{cases}$$

If all rules of G are productive, then all sets $\text{First}_1(A)$ are non-empty.

Lookahead Sets

For $\alpha \in (N \cup T)^*$ we are interested in the set:

$$\text{First}_1(\alpha) \;=\; \text{First}_1(\{w \in T^* \mid \alpha \rightarrow^* w\})$$

Lookahead Sets

For $\alpha \in (N \cup T)^*$ we are interested in the set:

$$\text{First}_1(\alpha) \;=\; \text{First}_1(\{w \in T^* \mid \alpha \to^* w\})$$

Idea: Treat ϵ separately: $\text{First}_1(A) = F_\epsilon(A) \cup \{\epsilon \mid A \to^* \epsilon\}$

- Let $\quad \text{empty}(X) = \text{true} \quad$ iff $\quad X \to^* \epsilon$.
- $F_\epsilon(X_1 \ldots X_m) = F_\epsilon(X_1) \cup \ldots \cup F_\epsilon(X_j)$ if $\neg\text{empty}(X_j) \;\wedge\; \bigwedge_{i=1}^{j-1} \text{empty}(X_i)$

Lookahead Sets

For $\alpha \in (N \cup T)^*$ we are interested in the set:

$$\text{First}_1(\alpha) \;=\; \text{First}_1(\{w \in T^* \mid \alpha \to^* w\})$$

Idea: Treat ϵ separately: $\text{First}_1(A) = F_\epsilon(A) \cup \{\epsilon \mid A \to^* \epsilon\}$

- Let $\text{empty}(X) = \text{true}$ iff $X \to^* \epsilon$.
- $F_\epsilon(X_1 \ldots X_m) = \bigcup_{i=1}^{j} F_\epsilon(X_i)$ if $\neg\text{empty}(X_j) \;\wedge\; \bigwedge_{i=1}^{j-1} \text{empty}(X_i)$

Lookahead Sets

For $\alpha \in (N \cup T)^*$ we are interested in the set:

$$\text{First}_1(\alpha) \;=\; \text{First}_1(\{w \in T^* \mid \alpha \rightarrow^* w\})$$

Idea: Treat ϵ separately: $\text{First}_1(A) = F_\epsilon(A) \cup \{\epsilon \mid A \rightarrow^* \epsilon\}$

- Let $\quad \text{empty}(X) = \text{true} \quad$ iff $\quad X \rightarrow^* \epsilon$.
- $F_\epsilon(X_1 \ldots X_m) = \bigcup_{i=1}^{j} F_\epsilon(X_i)$ if $\neg\text{empty}(X_j) \;\wedge\; \bigwedge_{i=1}^{j-1} \text{empty}(X_i)$

We characterize the ϵ-free First_1-sets with an inequality system:

$$\begin{array}{lllll} F_\epsilon(a) & = & \{a\} & \text{if} & a \in T \\ F_\epsilon(A) & \supseteq & F_\epsilon(X_j) & \text{if} & A \rightarrow X_1 \ldots X_m \;\in P, \quad \text{empty}(X_1) \wedge \ldots \wedge \text{empty}(X_{j-1}) \end{array}$$

Lookahead Sets

for example...

$$\begin{array}{rcllll} E & \rightarrow & E + T & \mid & T & \\ T & \rightarrow & T * F & \mid & F & \\ F & \rightarrow & (\ E\ ) & \mid & \mathsf{name} & \mid \ \mathsf{int} \end{array}$$

with $\mathsf{empty}(E) = \mathsf{empty}(T) = \mathsf{empty}(F) = \mathsf{false}$

Lookahead Sets

for example...

$$\begin{array}{lcllll} E & \to & E + T & \mid & T & \\ T & \to & T * F & \mid & F & \\ F & \to & (\ E\ ) & \mid & \mathsf{name} & \mid\ \mathsf{int} \end{array}$$

with $\mathsf{empty}(E) = \mathsf{empty}(T) = \mathsf{empty}(F) = \mathsf{false}$

... we obtain:

$$\begin{array}{lcl@{\qquad}lcl} F_\epsilon(S') & \supseteq & F_\epsilon(E) & F_\epsilon(E) & \supseteq & F_\epsilon(E) \\ F_\epsilon(E) & \supseteq & F_\epsilon(T) & F_\epsilon(T) & \supseteq & F_\epsilon(T) \\ F_\epsilon(T) & \supseteq & F_\epsilon(F) & F_\epsilon(F) & \supseteq & \{\ (\ , \mathsf{name}, \mathsf{int}\} \end{array}$$

Fast Computation of Lookahead Sets

Observation:

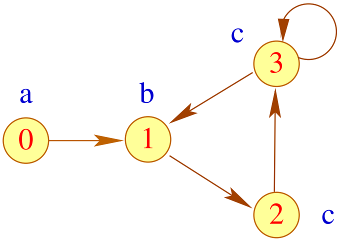
- The form of each inequality of these systems is:

$$x \sqsupseteq y \qquad \text{resp.} \qquad x \sqsupseteq d$$

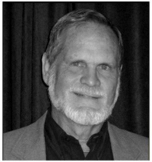
  for variables x, y und $d \in \mathbb{D}$.
- Such systems are called pure unification problems
- Such problems can be solved in linear space/time.

for example: $\mathbb{D} = 2^{\{a,b,c\}}$

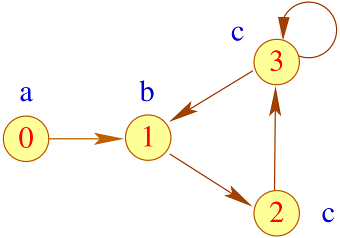
$$\begin{array}{lll}
x_0 \supseteq \{a\} & & \\
x_1 \supseteq \{b\} & x_1 \supseteq x_0 & x_1 \supseteq x_3 \\
x_2 \supseteq \{c\} & x_2 \supseteq x_1 & \\
x_3 \supseteq \{c\} & x_3 \supseteq x_2 & x_3 \supseteq x_3
\end{array}$$



Fast Computation of Lookahead Sets



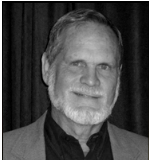
Frank DeRemer
& Tom Pennello



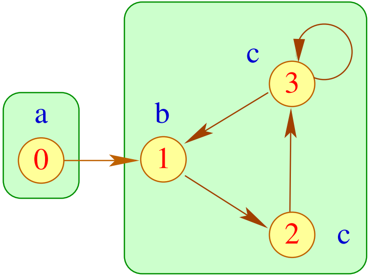
Proceeding:

- Create the Variable Dependency Graph for the inequality system.

Fast Computation of Lookahead Sets



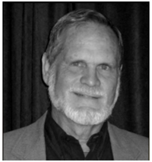
Frank DeRemer
& Tom Pennello



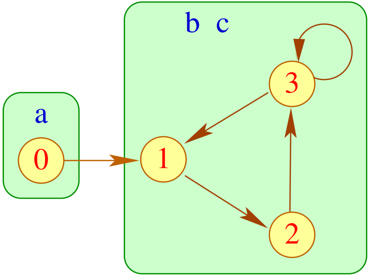
Proceeding:

- Create the Variable Dependency Graph for the inequality system.
- Whithin a Strongly Connected Component ($\rightarrow$ Tarjan) all variables have the same value

Fast Computation of Lookahead Sets



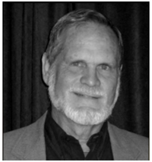
Frank DeRemer
& Tom Pennello



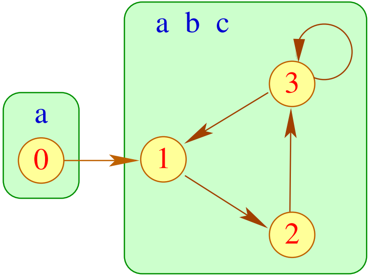
Proceeding:

- Create the Variable Dependency Graph for the inequality system.
- Whithin a Strongly Connected Component ($\rightarrow$ Tarjan) all variables have the same value
- Is there no ingoing edge for an SCC, its value is computed via the smallest upper bound of all values within the SCC

Fast Computation of Lookahead Sets



Frank DeRemer
& Tom Pennello



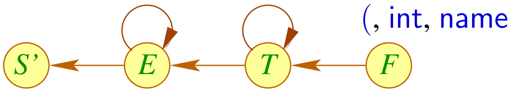
Proceeding:

- Create the Variable Dependency Graph for the inequality system.
- Whithin a Strongly Connected Component ($\rightarrow$ Tarjan) all variables have the same value
- Is there no ingoing edge for an SCC, its value is computed via the smallest upper bound of all values within the SCC
- In case of ingoing edges, their values are also to be considered for the upper bound

Fast Computation of Lookahead Sets

... for our example grammar:

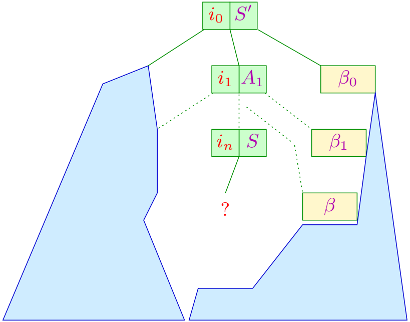
First$_1$:



Item Pushdown Automaton as LL(1)-Parser

context is relevant too: $S' \to S\,\$ \qquad S \to \epsilon^{\,0} \quad | \quad a\,S\,b^{\,1}$

$\text{First}_1(\text{input})$	$\$$	a	b
S	?	?	?

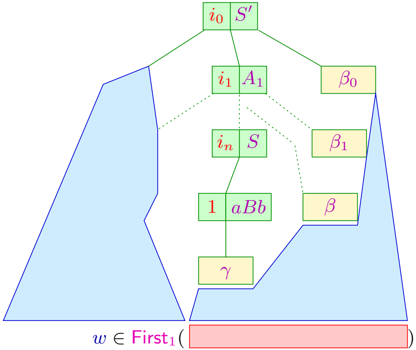


$w \in \text{First}_1($ $)$

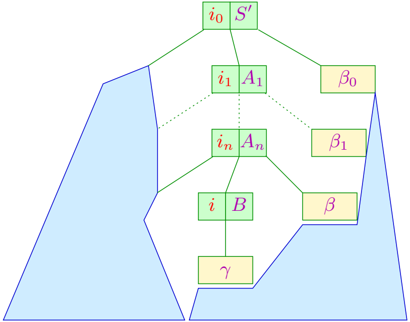
Item Pushdown Automaton as LL(1)-Parser

context is relevant too: $S' \to S\,\$ \qquad S \to \epsilon^{\,0} \;\;|\;\; a\,S\,b^{\,1}$

$\text{First}_1(\text{input})$	$\$$	a	b
S	?	?	?



Item Pushdown Automaton as LL(1)-Parser

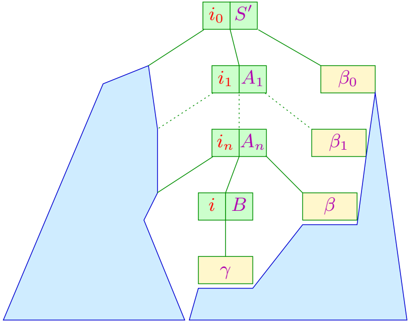


$w \in \mathsf{First}_1(\qquad)$

$w \in \mathsf{First}_1(\mathsf{First}_1(\gamma) \odot_1 \mathsf{First}_1(\beta) \odot_1 \ldots \odot_1 \mathsf{First}_1(\beta_0))$

$w \in \mathsf{First}_1(\gamma) \odot_1 \mathsf{Follow}_1(B)$

Item Pushdown Automaton as LL(1)-Parser



$w \in \text{First}_1(\quad)$

$w \in \text{First}_1(\text{First}_1(\gamma) \odot_1 \text{First}_1(\beta) \odot_1 \ldots \odot_1 \text{First}_1(\beta_0))$

$w \in \text{First}_1(\gamma) \odot_1 \text{Follow}_1(B)$

Inequality system for $\text{Follow}_1(B) = \text{First}_1(\beta) \odot_1 \ldots \odot_1 \text{First}_1(\beta_0)$

$$
\begin{array}{lcll}
\text{Follow}_1(S) & \supseteq & \{\$\} & \\
\text{Follow}_1(B) & \supseteq & F_\epsilon(X_j) & \text{if } A \rightarrow \alpha\, B\, X_1 \ldots X_m \in P,\ \text{empty}(X_1) \wedge \ldots \wedge \text{empty}(X_{j-1}) \\
\text{Follow}_1(B) & \supseteq & \text{Follow}_1(A) & \text{if } A \rightarrow \alpha\, B\, X_1 \ldots X_m \in P,\ \text{empty}(X_1) \wedge \ldots \wedge \text{empty}(X_m)
\end{array}
$$

Item Pushdown Automaton as LL(1)-Parser

Is G an $LL(1)$-grammar, we can index a lookahead-table with items and nonterminals:

LL(1)-Lookahead Table

We set $M[B,\ w]\ =\ i$ with $B \to \gamma^{\ i}$ if $w \in \text{First}_1(\gamma)\ \odot_1\ \text{Follow}_1(B)$

... for example: $S' \to S\,\$ \qquad S \to \epsilon^{\ 0} \quad | \quad a\,S\,b^{\ 1}$

Item Pushdown Automaton as LL(1)-Parser

Is G an $LL(1)$-grammar, we can index a lookahead-table with items and nonterminals:

LL(1)-Lookahead Table

We set $M[B,\ w]\ =\ i$ with $B \to \gamma^{\ i}$ if $w \in \text{First}_1(\gamma)\ \odot_1\ \text{Follow}_1(B)$

... for example: $S' \to S\ \$ \qquad S \to \epsilon^{\ 0} \quad | \quad a\,S\,b^{\ 1}$

$$\text{First}_1(S) = \{\epsilon, a\}$$

Item Pushdown Automaton as LL(1)-Parser

Is G an $LL(1)$-grammar, we can index a lookahead-table with items and nonterminals:

LL(1)-Lookahead Table

We set $M[B,\ w]\ =\ i$ with $B \to \gamma^{\,i}$ if $w \in \mathsf{First}_1(\gamma)\ \odot_1\ \mathsf{Follow}_1(B)$

... for example: $S' \to S\,\$ \qquad S \to \epsilon^{\,0} \quad | \quad a\,S\,b^{\,1}$

$$\mathsf{First}_1(S) = \{\epsilon, a\} \quad \mathsf{Follow}_1(S) = \{b, \$\}$$

Item Pushdown Automaton as LL(1)-Parser

Is G an $LL(1)$-grammar, we can index a lookahead-table with items and nonterminals:

LL(1)-Lookahead Table

We set $M[B,\ w] = i$ with $B \to \gamma^{\ i}$ if $w \in \text{First}_1(\gamma) \odot_1 \text{Follow}_1(B)$

... for example: $S' \to S\ \$ \qquad S \to \epsilon^{\ 0} \quad | \quad a\, S\, b^{\ 1}$

$$\text{First}_1(S) = \{\epsilon, a\} \quad \text{Follow}_1(S) = \{b, \$\}$$

$$\begin{array}{lrcl} S\text{-rule } 0: & \text{First}_1(\epsilon) & \odot_1 & \text{Follow}_1(S) = \{b, \$\} \\ S\text{-rule } 1: & \text{First}_1(aSb) & \odot_1 & \text{Follow}_1(S) = \{a\} \end{array}$$

Item Pushdown Automaton as LL(1)-Parser

Is G an $LL(1)$-grammar, we can index a lookahead-table with items and nonterminals:

LL(1)-Lookahead Table

We set $M[B,\ w]\ =\ i$ with $B \to \gamma^{\ i}$ if $w \in \text{First}_1(\gamma)\ \odot_1\ \text{Follow}_1(B)$

... for example: $S' \to S\ \$ \qquad S \to \epsilon^{\ 0} \quad | \quad a\ S\ b^{\ 1}$

$$\text{First}_1(S) = \{\epsilon, a\} \quad \text{Follow}_1(S) = \{b, \$\}$$

$$\begin{array}{lrcl} S\text{-rule } 0: & \text{First}_1(\epsilon) & \odot_1 & \text{Follow}_1(S) = \{b, \$\} \\ S\text{-rule } 1: & \text{First}_1(aSb) & \odot_1 & \text{Follow}_1(S) = \{a\} \end{array}$$

	$\$$	a	b
S	0	1	0

Item Pushdown Automaton as LL(1)-Parser

For example: $S' \to S\,\$ \qquad S \to \epsilon^{\,0} \mid a\,S\,b^{\,1}$

The transitions of the according Item Pushdown Automaton:

0	$[S' \to \bullet\, S\,\$]$	ϵ	$[S' \to \bullet\, S\,\$]\;[S \to \bullet]$
1	$[S' \to \bullet\, S\,\$]$	ϵ	$[S' \to \bullet\, S\,\$]\;[S \to \bullet\, a\,S\,b]$
2	$[S \to \bullet\, a\,S\,b]$	a	$[S \to a \bullet S\,b]$
3	$[S \to a \bullet S\,b]$	ϵ	$[S \to a \bullet S\,b]\;[S \to \bullet]$
4	$[S \to a \bullet S\,b]$	ϵ	$[S \to a \bullet S\,b]\;[S \to \bullet\, a\,S\,b]$
5	$[S \to a \bullet S\,b]\;[S \to \bullet]$	ϵ	$[S \to a\,S \bullet b]$
6	$[S \to a \bullet S\,b]\;[S \to a\,S\,b \bullet]$	ϵ	$[S \to a\,S \bullet b]$
7	$[S \to a\,S \bullet b]$	b	$[S \to a\,S\,b \bullet]$
8	$[S' \to \bullet\, S\,\$]\;[S \to \bullet]$	ϵ	$[S' \to S \bullet \$]$
9	$[S' \to \bullet\, S\,\$]\;[S \to a\,S\,b \bullet]$	ϵ	$[S' \to S \bullet \$]$

Lookahead table:

	$\$$	a	b
S	0	1	0

Left Recursion

Attention:

Many grammars are not $LL(k)$!

A reason for that is:

Definition

Grammar G is called left-recursive, if

$$A \rightarrow^{+} A\,\beta \qquad \text{for an} \quad A \in N \,,\; \beta \in (T \cup N)^{*}$$

Left Recursion

Attention:

Many grammars are not $LL(k)$!

A reason for that is:

Definition

Grammar G is called left-recursive, if

$$A \rightarrow^{+} A\,\beta \qquad \text{for an} \quad A \in N\,,\ \beta \in (T \cup N)^{*}$$

Example:

$$\begin{array}{lllllll} E & \rightarrow & E + T & \mid & T & & \\ T & \rightarrow & T * F & \mid & F & & \\ F & \rightarrow & (\ E\ ) & \mid & \text{name} & \mid & \text{int} \end{array}$$

... is left-recursive

Left Recursion

Theorem:

Let a grammar G be reduced and left-recursive, then G is not $LL(k)$ for any k.

Proof:

Let wlog. $A \to A\,\beta \mid \alpha \quad \in P$
and A be reachable from S

Assumption: G is $LL(k)$

Left Recursion

Theorem:

Let a grammar G be reduced and left-recursive, then G is not $LL(k)$ for any k.

Proof:

Let wlog. $A \to A\,\beta \mid \alpha \quad \in P$
and A be reachable from S

Assumption: G is $LL(k)$

$\Rightarrow$First$_k(\alpha\,\beta^n\,\gamma) \cap$
First$_k(\alpha\,\beta^{n+1}\,\gamma) = \emptyset$

Left Recursion

Theorem:

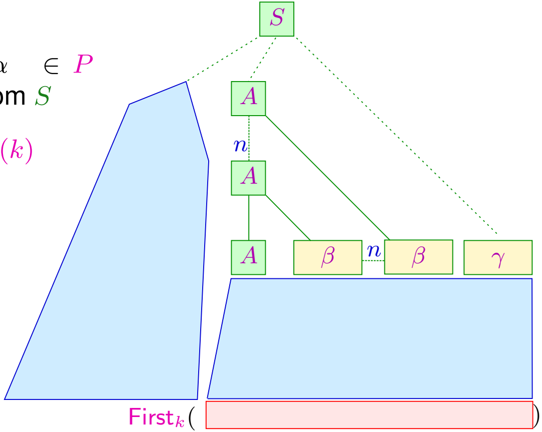
Let a grammar G be reduced and left-recursive, then G is not $LL(k)$ for any k.

Proof:

Let wlog. $A \to A\,\beta \mid \alpha \quad \in P$
and A be reachable from S

Assumption: G is $LL(k)$

$\Rightarrow \text{First}_k(\alpha\,\beta^n\,\gamma) \cap$
$\text{First}_k(\alpha\,\beta^{n+1}\,\gamma) = \emptyset$



Left Recursion

Theorem:

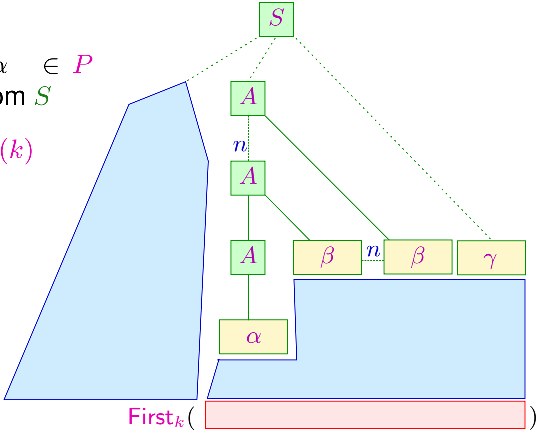
Let a grammar G be reduced and left-recursive, then G is not $LL(k)$ for any k.

Proof:

Let wlog. $A \to A\,\beta \mid \alpha \quad \in P$
and A be reachable from S

Assumption: G is $LL(k)$

$\Rightarrow \text{First}_k(\alpha\,\beta^n\,\gamma) \cap$
$\text{First}_k(\alpha\,\beta^{n+1}\,\gamma) = \emptyset$



Left Recursion

Theorem:

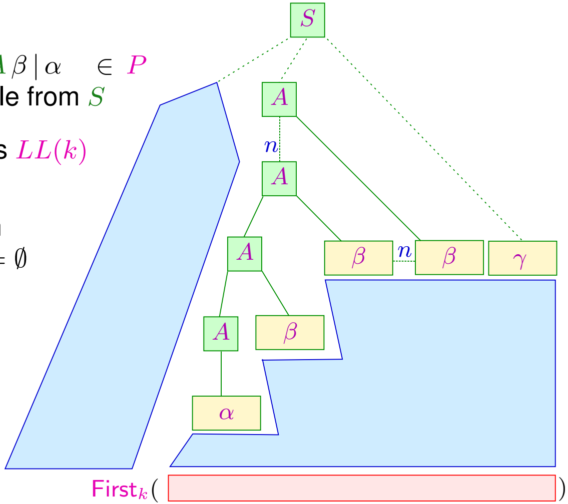
Let a grammar G be reduced and left-recursive, then G is not $LL(k)$ for any k.

Proof:

Let wlog. $A \to A\,\beta \mid \alpha \quad \in P$
and A be reachable from S

Assumption: G is $LL(k)$

$\Rightarrow \text{First}_k(\alpha\,\beta^n\,\gamma) \cap$
$\text{First}_k(\alpha\,\beta^{n+1}\,\gamma) = \emptyset$



Left Recursion

Theorem:

Let a grammar G be reduced and left-recursive, then G is not $LL(k)$ for any k.

Proof:

Let wlog. $A \to A\,\beta \,|\, \alpha \quad \in P$
and A be reachable from S

Assumption: G is $LL(k)$

$\Rightarrow$ $\text{First}_k(\alpha\,\beta^n\,\gamma) \cap$
$\text{First}_k(\alpha\,\beta^{n+1}\,\gamma) = \emptyset$

Case 1: $\beta \to^* \epsilon$ — Contradiction !!!
Case 2: $\beta \to^* w \neq \epsilon \Longrightarrow \text{First}_k(\alpha\,w^k\,\gamma) \cap \text{First}_k(\alpha\,w^{k+1}\,\gamma) \neq \emptyset$

Right-Regular Context-Free Parsing

Recurring scheme in programming languages: Lists of sth...

$S \to b \quad | \quad S\, a\, b$

Alternative idea: Regular Expressions

$S \to (\ b\ a\ )^*\ b$

Right-Regular Context-Free Parsing

Recurring scheme in programming languages: Lists of sth...

$S \rightarrow b \quad | \quad S\,a\,b$

Alternative idea: Regular Expressions

$S \rightarrow (\ b\ a\ )^*\ b$

Definition: Right-Regular Context-Free Grammar

A right-regular context-free grammar (RR-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 rules with regular expressions of symbols as rhs,
- $S \in N$ the start symbol

Right-Regular Context-Free Parsing

Recurring scheme in programming languages: Lists of sth...

$S \to b \quad | \quad S\, a\, b$

Alternative idea: Regular Expressions

$S \to (\ b\ a\ )^*\ b$

Definition: Right-Regular Context-Free Grammar

A right-regular context-free grammar (RR-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 rules with regular expressions of symbols as rhs,
- $S \in N$ the start symbol

Example: Arithmetic Expressions

$$\begin{array}{lll} S & \to & E \\ E & \to & T\ (\ +\ T\ )^* \\ T & \to & F\ (\ *\ F\ )^* \\ F & \to & (\ E\ )\ |\ \text{name}\ |\ \text{int} \end{array}$$

Idea 1: Rewrite the rules from G to $\langle G \rangle$:

$$\begin{array}{lcll}
A & \to & \langle \alpha \rangle & \text{if} \quad A \to \alpha \in P \\
\langle \alpha \rangle & \to & \alpha & \text{if} \quad \alpha \in N \cup T \\
\langle \epsilon \rangle & \to & \epsilon & \\
\langle \alpha^* \rangle & \to & \epsilon \mid \langle \alpha \rangle \langle \alpha^* \rangle & \text{if} \quad \alpha \in \mathsf{Regex}_{T,N} \\
\langle \alpha_1 \ldots \alpha_n \rangle & \to & \langle \alpha_1 \rangle \ldots \langle \alpha_n \rangle & \text{if} \quad \alpha_i \in \mathsf{Regex}_{T,N} \\
\langle \alpha_1 \mid \ldots \mid \alpha_n \rangle & \to & \langle \alpha_1 \rangle \mid \ldots \mid \langle \alpha_n \rangle & \text{if} \quad \alpha_i \in \mathsf{Regex}_{T,N}
\end{array}$$

…and generate the according LL(k)-Parser $M^L_{\langle G \rangle}$

Idea 1: Rewrite the rules from G to $\langle G \rangle$:

$$
\begin{array}{lcll}
A & \to & \langle \alpha \rangle & \text{if} \quad A \to \alpha \in P \\
\langle \alpha \rangle & \to & \alpha & \text{if} \quad \alpha \in N \cup T \\
\langle \epsilon \rangle & \to & \epsilon & \\
\langle \alpha^* \rangle & \to & \epsilon \mid \langle \alpha \rangle \langle \alpha^* \rangle & \text{if} \quad \alpha \in \text{Regex}_{T,N} \\
\langle \alpha_1 \ldots \alpha_n \rangle & \to & \langle \alpha_1 \rangle \ldots \langle \alpha_n \rangle & \text{if} \quad \alpha_i \in \text{Regex}_{T,N} \\
\langle \alpha_1 \mid \ldots \mid \alpha_n \rangle & \to & \langle \alpha_1 \rangle \mid \ldots \mid \langle \alpha_n \rangle & \text{if} \quad \alpha_i \in \text{Regex}_{T,N}
\end{array}
$$

…and generate the according LL(k)-Parser $M^L_{\langle G \rangle}$

Example: Arithmetic Expressions cont'd

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

Idea 1: Rewrite the rules from G to $\langle G \rangle$:

$$\begin{array}{lcll}
A & \to & \langle \alpha \rangle & \text{if} \quad A \to \alpha \in P \\
\langle \alpha \rangle & \to & \alpha & \text{if} \quad \alpha \in N \cup T \\
\langle \epsilon \rangle & \to & \epsilon & \\
\langle \alpha^* \rangle & \to & \epsilon \mid \langle \alpha \rangle \langle \alpha^* \rangle & \text{if} \quad \alpha \in \mathsf{Regex}_{T,N} \\
\langle \alpha_1 \ldots \alpha_n \rangle & \to & \langle \alpha_1 \rangle \ldots \langle \alpha_n \rangle & \text{if} \quad \alpha_i \in \mathsf{Regex}_{T,N} \\
\langle \alpha_1 \mid \ldots \mid \alpha_n \rangle & \to & \langle \alpha_1 \rangle \mid \ldots \mid \langle \alpha_n \rangle & \text{if} \quad \alpha_i \in \mathsf{Regex}_{T,N}
\end{array}$$

...and generate the according LL(k)-Parser $M^L_{\langle G \rangle}$

Example: Arithmetic Expressions cont'd

$$\begin{array}{lcl}
S & \to & E \\
E & \to & \langle T\,(\,+T)^* \rangle \\
T & \to & F\,(\,*\,F\,)^* \\
F & \to & (\,E\,) \mid \mathsf{name} \mid \mathsf{int} \\
\langle T\,(\,+T)^* \rangle & \to & T\,\langle (\,+T)^* \rangle
\end{array}$$

Idea 1: Rewrite the rules from G to $\langle G \rangle$:

$$
\begin{array}{lcll}
A & \to & \langle \alpha \rangle & \text{if} \quad A \to \alpha \in P \\
\langle \alpha \rangle & \to & \alpha & \text{if} \quad \alpha \in N \cup T \\
\langle \epsilon \rangle & \to & \epsilon & \\
\langle \alpha^* \rangle & \to & \epsilon \mid \langle \alpha \rangle \langle \alpha^* \rangle & \text{if} \quad \alpha \in \text{Regex}_{T,N} \\
\langle \alpha_1 \ldots \alpha_n \rangle & \to & \langle \alpha_1 \rangle \ldots \langle \alpha_n \rangle & \text{if} \quad \alpha_i \in \text{Regex}_{T,N} \\
\langle \alpha_1 \mid \ldots \mid \alpha_n \rangle & \to & \langle \alpha_1 \rangle \mid \ldots \mid \langle \alpha_n \rangle & \text{if} \quad \alpha_i \in \text{Regex}_{T,N}
\end{array}
$$

…and generate the according LL(k)-Parser $M^L_{\langle G \rangle}$

Example: Arithmetic Expressions cont'd

$$
\begin{array}{lcl}
S & \to & E \\
E & \to & \langle T \,(\, + T)^* \rangle \\
T & \to & F \,(\, * \, F \,)^* \\
F & \to & (\, E \,) \mid \text{name} \mid \text{int} \\
\langle T \,(\, + T)^* \rangle & \to & T \, \langle (\, + T)^* \rangle \\
\langle (\, + T)^* \rangle & \to & \epsilon \mid \langle \, + T \rangle \langle (\, + T)^* \rangle
\end{array}
$$

Idea 1: Rewrite the rules from G to $\langle G \rangle$:

$$\begin{array}{lcll}
A & \to & \langle\alpha\rangle & \text{if} \quad A \to \alpha \in P \\
\langle\alpha\rangle & \to & \alpha & \text{if} \quad \alpha \in N \cup T \\
\langle\epsilon\rangle & \to & \epsilon & \\
\langle\alpha^*\rangle & \to & \epsilon \mid \langle\alpha\rangle\langle\alpha^*\rangle & \text{if} \quad \alpha \in \text{Regex}_{T,N} \\
\langle\alpha_1 \ldots \alpha_n\rangle & \to & \langle\alpha_1\rangle \ldots \langle\alpha_n\rangle & \text{if} \quad \alpha_i \in \text{Regex}_{T,N} \\
\langle\alpha_1 \mid \ldots \mid \alpha_n\rangle & \to & \langle\alpha_1\rangle \mid \ldots \mid \langle\alpha_n\rangle & \text{if} \quad \alpha_i \in \text{Regex}_{T,N}
\end{array}$$

…and generate the according LL(k)-Parser $M^L_{\langle G\rangle}$

Example: Arithmetic Expressions cont'd

$$\begin{array}{lcl}
S & \to & E \\
E & \to & \langle T\,(\,+T)^*\rangle \\
T & \to & F\,(\,*\,F\,)^* \\
F & \to & (\,E\,) \mid \text{name} \mid \text{int} \\
\langle T\,(\,+T)^*\rangle & \to & T\,\langle(\,+T)^*\rangle \\
\langle(\,+T)^*\rangle & \to & \epsilon \mid \langle\,+T\rangle\langle(\,+T)^*\rangle \\
\langle\,+T\rangle & \to & +\,T
\end{array}$$

Idea 1: Rewrite the rules from G to $\langle G \rangle$:

$$
\begin{array}{lcll}
A & \to & \langle \alpha \rangle & \text{if} \quad A \to \alpha \in P \\
\langle \alpha \rangle & \to & \alpha & \text{if} \quad \alpha \in N \cup T \\
\langle \epsilon \rangle & \to & \epsilon & \\
\langle \alpha^* \rangle & \to & \epsilon \mid \langle \alpha \rangle \langle \alpha^* \rangle & \text{if} \quad \alpha \in \text{Regex}_{T,N} \\
\langle \alpha_1 \ldots \alpha_n \rangle & \to & \langle \alpha_1 \rangle \ldots \langle \alpha_n \rangle & \text{if} \quad \alpha_i \in \text{Regex}_{T,N} \\
\langle \alpha_1 \mid \ldots \mid \alpha_n \rangle & \to & \langle \alpha_1 \rangle \mid \ldots \mid \langle \alpha_n \rangle & \text{if} \quad \alpha_i \in \text{Regex}_{T,N}
\end{array}
$$

…and generate the according LL(k)-Parser $M^L_{\langle G \rangle}$

Example: Arithmetic Expressions cont'd

$$
\begin{array}{lcl}
S & \to & E \\
E & \to & \langle T\,(\,+T)^* \rangle \\
T & \to & \langle F\,(\,*\,F\,)^* \rangle \\
F & \to & (\,E\,) \mid \text{name} \mid \text{int} \\
\langle T\,(\,+T)^* \rangle & \to & T\, \langle (\,+T)^* \rangle \\
\langle (\,+T)^* \rangle & \to & \epsilon \mid \langle \,+T \rangle \langle (\,+T)^* \rangle \\
\langle \,+T \rangle & \to & +\,T
\end{array}
$$

Idea 1: Rewrite the rules from G to $\langle G \rangle$:

$$\begin{array}{llll}
A & \to & \langle\alpha\rangle & \text{if} \quad A \to \alpha \in P \\
\langle\alpha\rangle & \to & \alpha & \text{if} \quad \alpha \in N \cup T \\
\langle\epsilon\rangle & \to & \epsilon & \\
\langle\alpha^*\rangle & \to & \epsilon \mid \langle\alpha\rangle\langle\alpha^*\rangle & \text{if} \quad \alpha \in \mathsf{Regex}_{T,N} \\
\langle\alpha_1 \ldots \alpha_n\rangle & \to & \langle\alpha_1\rangle \ldots \langle\alpha_n\rangle & \text{if} \quad \alpha_i \in \mathsf{Regex}_{T,N} \\
\langle\alpha_1 \mid \ldots \mid \alpha_n\rangle & \to & \langle\alpha_1\rangle \mid \ldots \mid \langle\alpha_n\rangle & \text{if} \quad \alpha_i \in \mathsf{Regex}_{T,N}
\end{array}$$

…and generate the according LL(k)-Parser $M^L_{\langle G\rangle}$

Example: Arithmetic Expressions cont'd

$$\begin{array}{lll}
S & \to & E \\
E & \to & \langle T\,(\,+T)^*\rangle \\
T & \to & \langle F\,(\,*\,F\,)^*\rangle \\
F & \to & (\,E\,) \mid \mathsf{name} \mid \mathsf{int} \\
\langle T\,(\,+T)^*\rangle & \to & T\,\langle(\,+T)^*\rangle \\
\langle(\,+T)^*\rangle & \to & \epsilon \mid \langle\,+T\rangle\langle(\,+T)^*\rangle \\
\langle\,+T\rangle & \to & +T \\
\langle F\,(\,*\,F\,)^*\rangle & \to & F\,\langle(\,*\,F\,)^*\rangle \\
\langle(\,*\,F\,)^*\rangle & \to & \epsilon \mid \langle *\,F\rangle\,\langle(\,*\,F\,)^*\rangle \\
\langle\,*\,F\rangle & \to & *\,F
\end{array}$$

Definition:

An $RR\!-\!CFG$ G is called $RLL(1)$,
if the corresponding CFG $\langle G\rangle$ is an $LL(1)$ grammar.

Reinhold Heckmann

Discussion

- directly yields the table driven parser $M^L_{\langle G\rangle}$ for $RLL(1)$ grammars
- however: mapping regular expressions to recursive productions unnessessarily strains the stack
  $\rightarrow$ instead directly construct automaton in the style of Berry-Sethi

Idea 2: Recursive Descent RLL Parsers:

Recursive descent RLL(1)-parsers are an alternative to table-driven parsers; apart from the usual function `scan()`, we generate a program frame with the lookahead function `expect()` and the main parsing method `parse()`:

```
int next;
void expect(Set E){
    if ({ϵ, next} ∩ E = ∅){
        cerr << "Expected" << E << "found" << next;
        exit(0);
    }
    return ;
}
void parse(){
    next = scan();
    expect(First₁(S)) ;
    S();
    expect({EOF}) ;
}
```

Idea 2: Recursive Descent RLL Parsers:

For each $A \to \alpha \; \in P$, we introduce:

$$\begin{array}{l} \texttt{void A()\{} \\ \qquad generate(\alpha) \\ \} \end{array}$$

with the meta-program $generate$ being defined by structural decomposition of α:

$$\begin{array}{lcl} generate(r_1 \ldots r_k) & = & generate(r_1) \\ & & \texttt{expect}(\mathsf{First}_1(r_2))\;; \\ & & generate(r_2) \\ & & \vdots \\ & & \texttt{expect}(\mathsf{First}_1(r_k))\;; \\ & & generate(r_k) \\ generate(\epsilon) & = & ; \\ generate(a) & = & \texttt{next} = \texttt{scan}(); \\ generate(A) & = & \texttt{A}(); \end{array}$$

Idea 2: Recursive Descent RLL Parsers:

$$
\begin{array}{lcl}
\mathit{generate}(r^*) & = & \texttt{while} \; (\; \texttt{next} \in \mathsf{F}_\epsilon(r)) \; \{ \\
 & & \quad \mathit{generate}(r) \\
 & & \} \\
\mathit{generate}(r_1 \mid \ldots \mid r_k) & = & \texttt{switch}(\texttt{next}) \; \{ \\
 & & \quad \mathit{labels}(\mathsf{First}_1(r_1)) \; \mathit{generate}(r_1) \; \texttt{break} \; ; \\
 & & \quad \vdots \\
 & & \quad \mathit{labels}(\mathsf{First}_1(r_k)) \; \mathit{generate}(r_k) \; \texttt{break} \; ; \\
 & & \} \\
\mathit{labels}(\{\alpha_1, \ldots, \alpha_m\}) & = & \mathit{label}(\alpha_1) \colon \; \ldots \; \mathit{label}(\alpha_m) \colon \\
\mathit{label}(\alpha) & = & \texttt{case} \; \alpha \\
\mathit{label}(\epsilon) & = & \texttt{default}
\end{array}
$$

Discussion

- A practical implementation of an $RLL(1)$-parser via recursive descent is a straight-forward idea
- However, only a subset of the deterministic contextfree languages can be parsed this way.
- As soon as $\mathsf{First}_1(\_)$ sets are not disjoint any more,

Topdown-Parsing

Discussion

- A practical implementation of an $RLL(1)$-parser via recursive descent is a straight-forward idea
- However, only a subset of the deterministic contextfree languages can be parsed this way.
- As soon as $\text{First}_1(\_)$ sets are not disjoint any more,
  - Solution 1: For many accessibly written grammars, the alternation between right hand sides happens too early. Keeping the common prefixes of right hand sides joined and introducing a new production for the actual diverging sentence forms often helps.
  - Solution 2: Introduce *ranked* grammars, and decide conflicting lookahead always in favour of the higher ranked alternative
    $\to$ relation to LL parsing not so clear any more
    $\to$ not so clear for $\_^*$ operator how to decide
  - Solution 3: Going from $LL(1)$ to $LL(k)$
    The size of the occuring sets is rapidly increasing with larger k
    Unfortunately, even $LL(k)$ parsers are not sufficient to accept all deterministic contextfree languages. (regular lookahead $\to$ $LL(*)$)
- In practical systems, this often motivates the implementation of $k = 1$ only ...