Script **generated by TTT**

Title: Petter: Compilerbau (24.05.2018)

Date: Thu May 24 14:18:07 CEST 2018

Duration: 89:24 min

Pages: 33

Left Recursion

Theorem:

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

Proof:

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

Assumption: G is $LL(k)$

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

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...
$S \to b \mid S\,a\,b$
Alternative idea: Regular Expressions
$S \to (\,b\,a\,)^*\,b$

Right-Regular Context-Free Parsing

Recurring scheme in programming languages: Lists of sth...
$S \to b \mid S\,a\,b$
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

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 $M^L_{\langle G \rangle}$

Reinhold Heckmann

Definition:

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

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
  $\to$ 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;
boolean expect(Set E){
    if ({ϵ, next} ∩ E = ∅){
        cerr << "Expected" << E << "found" << next;
        return false;
    }
    return true;
}
void parse(){
    next = scan();
    if (!expect(First₁(S))) exit(0);
    S();
    if (!expect({EOF})) exit(0);
}
```


Idea 2: Recursive Descent RLL Parsers:

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

```
void A(){
    generate(α)
}
```

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

$$
\begin{array}{lcl}
generate(r_1 \ldots r_k) & = & generate(r_1) \\
 & & \texttt{if (!expect(}\mathsf{First}_1(r_2)\texttt{)) exit(0);} \\
 & & generate(r_2) \\
 & & \vdots \\
 & & \texttt{if (!expect(}\mathsf{First}_1(r_k)\texttt{)) exit(0);} \\
 & & generate(r_k) \\
generate(\epsilon) & = & \texttt{;} \\
generate(a) & = & \texttt{consume();} \\
generate(A) & = & \texttt{A();}
\end{array}
$$

Idea 2: Recursive Descent RLL Parsers:

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

Idea 2: Recursive Descent RLL Parsers:

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

112 / 288

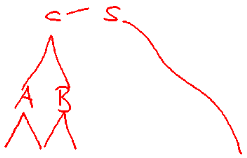
Topdown-Parsing

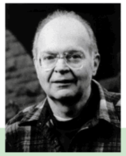
Discussion

- 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 $\mathsf{First}_1(\_)$ sets are not disjoint any more,

Chapter 4:
Bottom-up Analysis

Shift-Reduce Parser





Donald Knuth

Idea:
We *delay* the decision whether to reduce until we know, whether the input matches the right-hand-side of a rule!

Construction: Shift-Reduce parser M_G^R

- The input is shifted successively to the pushdown.
- Is there a complete right-hand side (a handle) atop the pushdown, it is replaced (reduced) by the corresponding left-hand side

Shift-Reduce Parser

Example:

$$\begin{array}{lcl} S & \to & A\,B \\ A & \to & a \\ B & \to & b \end{array}$$

The pushdown automaton:

States: $q_0, f, a, b, A, B, S;$
Start state: q_0
End state: f

q_0	a	$q_0\,a$
a	ϵ	A
A	b	$A\,b$
b	ϵ	B
$A\,B$	ϵ	S
$q_0\,S$	ϵ	f

Shift-Reduce Parser

Observation:

- The sequence of reductions corresponds to a reverse rightmost-derivation for the input
- To prove correctnes, we have to prove:

$$(\epsilon, w) \vdash^* (A, \epsilon) \qquad \text{iff} \qquad A \to^* w$$

- The shift-reduce pushdown automaton M_G^R is in general also non-deterministic
- For a deterministic parsing-algorithm, we have to identify computation-states for reduction

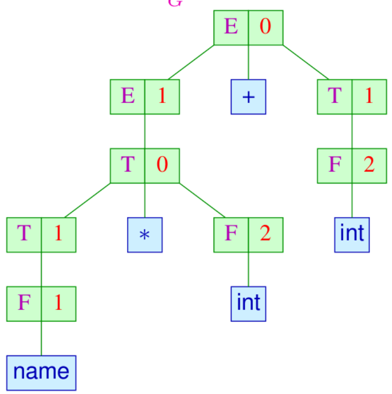
$\Longrightarrow$ LR-Parsing

Reverse Rightmost Derivations in Shift-Reduce-Parsers

Idea: Observe *reverse rightmost*-derivations of M_G^R!

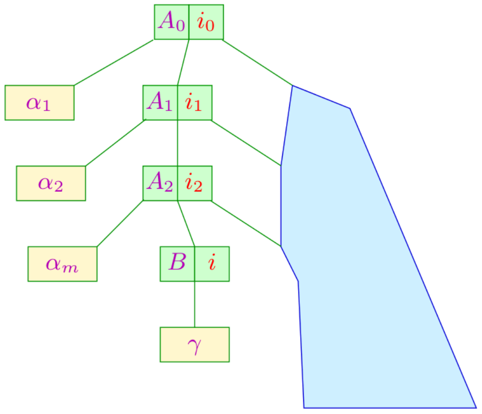
Input:
counter $*\ 2 + 40$

Pushdown:
$(\ q_0\ )$



Bottom-up Analysis: Viable Prefix

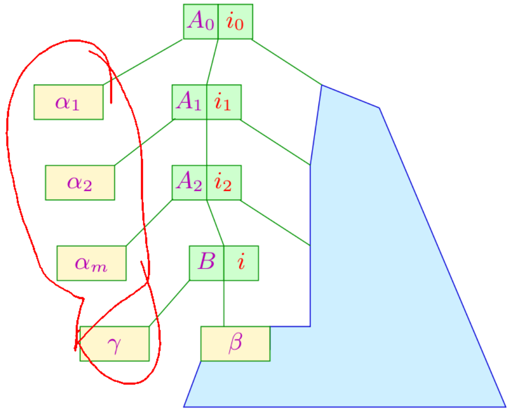
$\alpha\,\gamma$ is viable for $[B \to \gamma\bullet]$ iff $S \to^*_R \alpha\, B\, v$



... with $\alpha = \alpha_1 \ldots \alpha_m$

Bottom-up Analysis: Admissible Items

The item $[B \to \gamma \bullet \beta]$ is called admissible for α' iff $S \to^*_R \alpha\, B\, v$ with $\alpha' = \alpha\,\gamma$:



... with $\alpha = \alpha_1 \ldots \alpha_m$

Characteristic Automaton

Observation:

The set of viable prefixes from $(N \cup T)^*$ for (admissible) items can be computed from the content of the shift-reduce parser's pushdown with the help of a finite automaton:

States: Items
Start state: $[S' \to \bullet\, S]$
Final states: $\{[B \to \gamma\bullet] \mid B \to \gamma \in P\}$
Transitions:

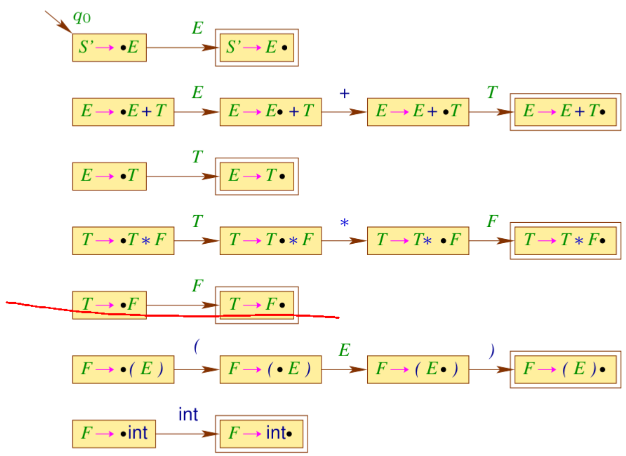
(1) $([A \to \alpha \bullet X\, \beta], X, [A \to \alpha\, X \bullet \beta]), \quad X \in (N \cup T), A \to \alpha\, X\, \beta \in P;$
(2) $([A \to \alpha \bullet B\, \beta], \epsilon, [B \to \bullet\, \gamma]), \quad A \to \alpha\, B\, \beta\, ,\ B \to \gamma \in P;$

The automaton $c(G)$ is called characteristic automaton for G.

Characteristic Automaton

For example:

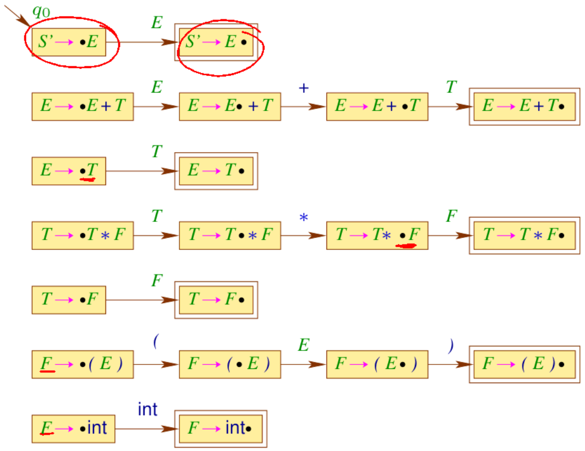
$$\begin{array}{lcl} E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (\ E\ ) \mid \text{int} \end{array}$$



Characteristic Automaton

For example:

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

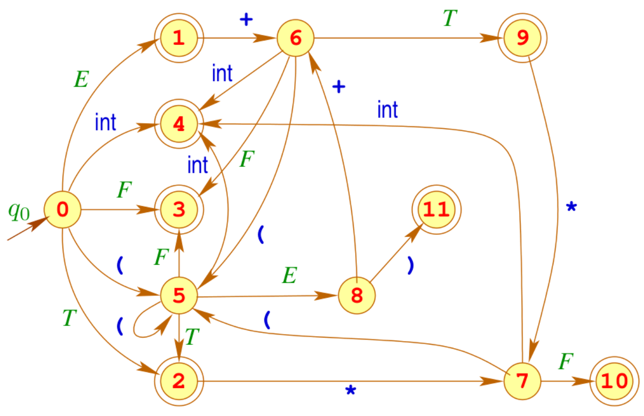


Canonical LR(0)-Automaton

The canonical $LR(0)$-automaton $LR(G)$ is created from $c(G)$ by:

1. performing arbitrarily many ϵ-transitions after every consuming transition
2. performing the powerset construction

... for example:



Canonical LR(0)-Automaton

Observation:

The canonical $LR(0)$-automaton can be created directly from the grammar.
Therefore we need a helper function δ_ϵ^* (ϵ-closure)

$$\delta_\epsilon^*(q) = q \cup \{[B \to \bullet\,\gamma] \mid B \to \gamma \in P,\ [A \to \alpha \bullet B'\,\beta'] \in q,\ B' \to^* B\,\beta\}$$

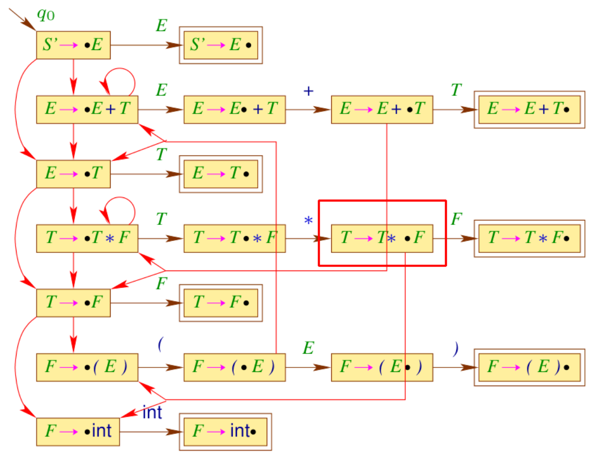
We define:

- States: Sets of items;
- Start state: $\delta_\epsilon^*\,\{[S' \to \bullet\, S]\}$
- Final states: $\{q \mid [A \to \alpha\,\bullet] \in q\}$
- Transitions: $\delta(q, X) = \delta_\epsilon^*\,\{[A \to \alpha\, X \bullet \beta] \mid [A \to \alpha \bullet X\,\beta] \in q\}$

Characteristic Automaton

For example:

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



Canonical LR(0)-Automaton

Observation:

The canonical $LR(0)$-automaton can be created directly from the grammar.
Therefore we need a helper function δ^*_ϵ (ϵ-closure)

$$\delta^*_\epsilon(q) = q \cup \{[B \to \bullet\, \gamma] \mid B \to \gamma \in P, \; [A \to \alpha \bullet B'\, \beta'] \in q, \; B' \to^* B\, \beta\}$$

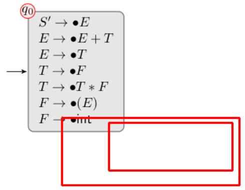
We define:

States: Sets of items;
Start state: $\delta^*_\epsilon\, \{[S' \to \bullet\, S]\}$
Final states: $\{q \mid [A \to \alpha \bullet] \in q\}$
Transitions: $\delta(q, X) = \delta^*_\epsilon\, \{[A \to \alpha\, X \bullet\, \beta] \mid [A \to \alpha \bullet X\, \beta] \in q\}$

Canonical LR(0)-Automaton

For example:

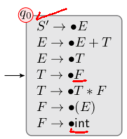
$$\begin{array}{lcll} S' & \to & E & \\ E & \to & E+T & \mid\; T \\ T & \to & T*F & \mid\; F \\ F & \to & (\;E\;) & \mid\; \text{int} \end{array}$$



Canonical LR(0)-Automaton

For example:

$$\begin{array}{lcll} S' & \to & E & \\ E & \to & E+T & \mid\; T \\ T & \to & T*F & \mid\; F \\ F & \to & (\;E\;) & \mid\; \text{int} \end{array}$$



LR(0)-Parser

Idea for a parser:

- The parser manages a viable prefix $\alpha = X_1 \ldots X_m$ on the pushdown and uses $LR(G)$, to identify reduction spots.
- It can reduce with $A \to \gamma$, if $[A \to \gamma \bullet]$ is admissible for α

Optimization:

We push the states instead of the X_i in order not to process the pushdown's content with the automaton anew all the time.
Reduction with $A \to \gamma$ leads to popping the uppermost $|\gamma|$ states and continue with the state on top of the stack and input A.

Attention:

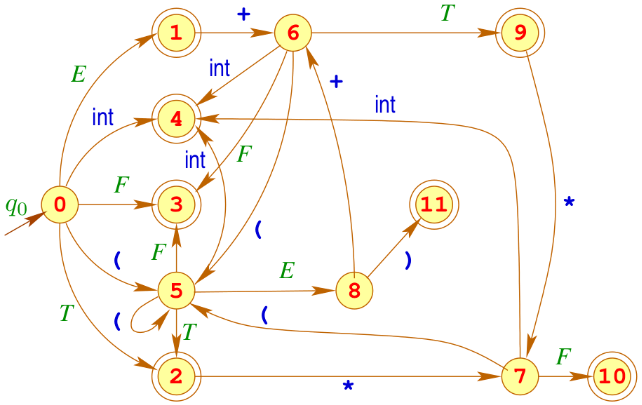
This parser is only deterministic, if each final state of the canonical $LR(0)$-automaton is conflict free.

Canonical LR(0)-Automaton

The canonical $LR(0)$-automaton $LR(G)$ is created from $c(G)$ by:

1. performing arbitrarily many ϵ-transitions after every consuming transition
2. performing the powerset construction

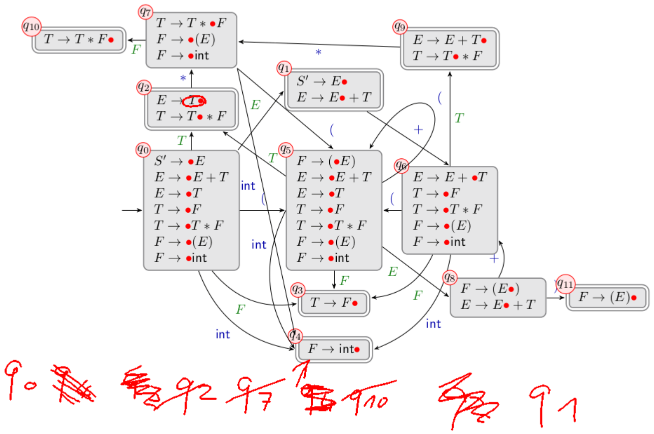
... for example:



Canonical LR(0)-Automaton

For example:

$$\begin{array}{lllll} S' & \to & E & & \\ E & \to & E+T & \mid & T \\ T & \to & T*F & \mid & F \\ F & \to & (\,E\,) & \mid & \text{int} \end{array}$$



LR(0)-Parser

Idea for a parser:

- The parser manages a viable prefix $\alpha = X_1 \ldots X_m$ on the pushdown and uses $LR(G)$, to identify reduction spots.
- It can reduce with $A \to \gamma$, if $[A \to \gamma\,\bullet]$ is admissible for α

Optimization:

We push the states instead of the X_i in order not to process the pushdown's content with the automaton anew all the time.
Reduction with $A \to \gamma$ leads to popping the uppermost $|\gamma|$ states and continue with the state on top of the stack and input A.

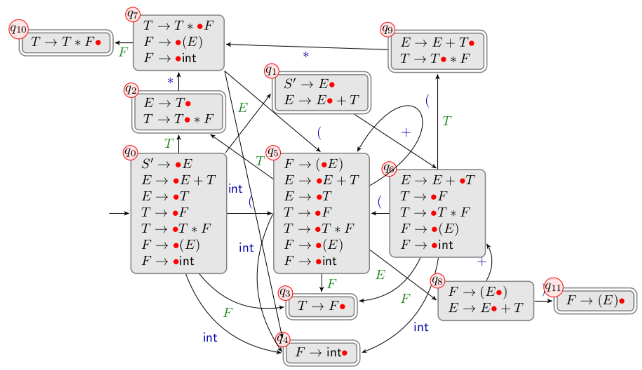
Attention:

This parser is only deterministic, if each final state of the canonical $LR(0)$-automaton is conflict free.

Canonical LR(0)-Automaton

For example:

$$\begin{array}{lllll} S' & \to & E & & \\ E & \to & E+T & \mid & T \\ T & \to & T*F & \mid & F \\ F & \to & (\,E\,) & \mid & \text{int} \end{array}$$



LR(0)-Parser

... for example:

$$q_1 = \{[S' \to E\bullet], [E \to E \bullet + T]\}$$

$$q_2 = \{[E \to T\bullet], [T \to T \bullet * F]\} \qquad q_9 = \{[E \to E + T\bullet], [T \to T \bullet * F]\}$$

$$q_3 = \{[T \to F\bullet]\} \qquad q_{10} = \{[T \to T * F\bullet]\}$$

$$q_4 = \{[F \to \text{int}\bullet]\} \qquad q_{11} = \{[F \to (E)\bullet]\}$$

The final states q_1, q_2, q_9 contain more than one admissible item

$\Rightarrow$ non deterministic!

LR(0)-Parser

Idea for a parser:

- The parser manages a viable prefix $\alpha = X_1 \ldots X_m$ on the pushdown and uses $LR(G)$, to identify reduction spots.
- It can reduce with $A \to \gamma$, if $[A \to \gamma\bullet]$ is admissible for α

Optimization:

We push the states instead of the X_i in order not to process the pushdown's content with the automaton anew all the time.
Reduction with $A \to \gamma$ leads to popping the uppermost $|\gamma|$ states and continue with the state on top of the stack and input A.

Attention:

This parser is only deterministic, if each final state of the canonical $LR(0)$-automaton is conflict free.