Script **generated by TTT**

Title: Petter: Compilerbau (01.06.2017)

Date: Thu Jun 01 14:15:02 CEST 2017

Duration: 88:57 min

Pages: 28

Lookahead Sets

for example...

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

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

... we obtain:

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

Fast Computation of Lookahead Sets

Observation:

- The form of each inequality of these systems is:

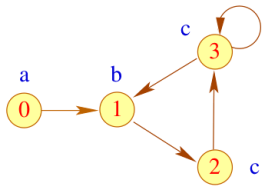
$$x \sqsupseteq y \quad \text{resp.} \quad x \sqsupseteq d$$

for variables x, y und $d \in \mathrm{D}$.

- Such systems are called pure unification problems
- Such problems can be solved in linear space/time.

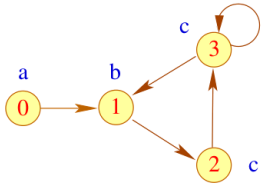
for example: $\mathrm{D} = 2^{\{a,b,c\}}$

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



Fast Computation of Lookahead Sets

Frank DeRemer & Tom Pennello



Proceeding:

- Create the Variable Dependency Graph for the inequality system.

Item Pushdown Automaton as LL(1)-Parser

back to the example: $S \to \epsilon \mid a\,S\,b$

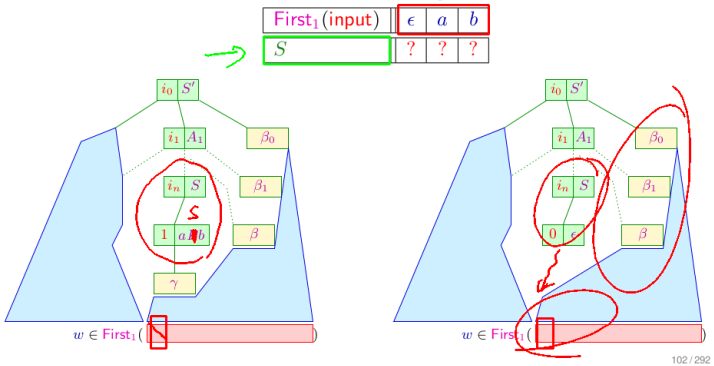
The transitions in the according Item Pushdown Automaton:

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	$[S' \to S \bullet]$

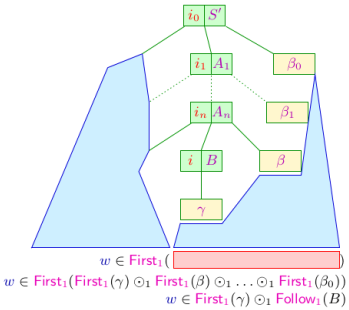
Conflicts arise between transations $(0, 1)$ or $(3, 4)$ resp..

Item Pushdown Automaton as LL(1)-Parser

... in detail: $S \to \epsilon^{\,0} \mid a\,S\,b^{\,1}$



Item Pushdown Automaton as LL(1)-Parser



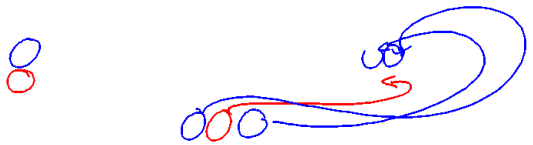
Item Pushdown Automaton as LL(1)-Parser

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

... for example: $S \to \epsilon^{\,0} \mid a\,S\,b^{\,1}$



Item Pushdown Automaton as LL(1)-Parser

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

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:

	ϵ	a	b
S	0	1	0

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

... for example: $S \to \epsilon^{\,0} \mid a\,S\,b^{\,1}$

$$\mathrm{First}_1(S) = \{\epsilon, a\} \qquad \mathrm{Follow}_1(S) = \{b, \epsilon\}$$

S-rule 0 : $\mathrm{First}_1(\epsilon) \odot_1 \mathrm{Follow}_1(S) = \{b, \epsilon\}$

S-rule 1 : $\mathrm{First}_1(aSb) \odot_1 \mathrm{Follow}_1(S) = \{a\}$

Item Pushdown Automaton as LL(1)-Parser

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

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:

	ϵ	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

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

Left Recursion

Theorem:

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

Proof:

Let $A \to A\,\beta \mid \alpha \quad \in P$
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 $A \to A\,\beta \mid \alpha \quad \in P$
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...
$S \to b \quad | \quad S\,a\,b$
Alternative idea: Regular Expressions
$S \to (\ b\ a\ )^*\ b$

Right-Regular Context-Free Parsing

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

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

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

. . . 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
  → 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:

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

Idea 2: Recursive Descent RLL Parsers:

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

Idea 2: Recursive Descent RLL Parsers:

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

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

Construction:

In general, we create an automaton $M_G^R = (Q, T, \delta, q_0, F)$ with:

- $Q = T \cup N \cup \{q_0, f\}$ (q_0, f fresh);
- $F = \{f\}$;
- Transitions:

$$\begin{array}{lll} \delta = & \{(q, x, q\,x) \mid q \in Q, x \in T\} \cup & // \text{ Shift-transitions} \\ & \{(q\,\alpha, \epsilon, q\,A) \mid q \in Q, A \to \alpha \in P\} \cup & // \text{ Reduce-transitions} \\ & \{(q_0\,S, \epsilon, f)\} & // \text{ finish} \end{array}$$

Example-computation:

$$\begin{array}{lllllll} (q_0, & a\,b) & \vdash & (q_0\,a, & b) & \vdash & (q_0\,A, \quad b) \\ & & \vdash & (q_0\,A\,b, & \epsilon) & \vdash & (q_0\,A\,B, \quad \epsilon) \\ & & \vdash & (q_0\,S, & \epsilon) & \vdash & (f, \quad \epsilon) \end{array}$$

Shift-Reduce Parser

Observation:

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

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

- 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

Shift-Reduce Parser

Observation:

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

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

- 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

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