# Compilerconstructie

najaar 2012

http://www.liacs.nl/home/rvvliet/coco/

**Rudy van Vliet**
kamer 124 Snellius, tel. 071-527 5777
rvvliet(at)liacs.nl

werkcollege 9, dinsdag 27 november 2012

SLR Parsing / Backpatching



## FIRST

- Let $\alpha$ be string of grammar symbols
- FIRST($\alpha$) = set of terminals/tokens which begin strings derived from $\alpha$
- If $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then $\epsilon \in$ FIRST($\alpha$)
- Example

$$F \rightarrow (E) \mid \textbf{id}$$

FIRST($FT'$) = $\{(, \textbf{id}\}$

- When nonterminal has multiple productions, e.g.,

$$A \rightarrow \alpha \mid \beta$$

and FIRST($\alpha$) and FIRST($\beta$) are disjoint,
we can choose between these $A$-productions by looking at next input symbol



## Computing FIRST

Compute FIRST($X$) for all grammar symbols $X$:

- If $X$ is terminal, then FIRST($X$) = $\{X\}$
- If $X \rightarrow \epsilon$ is production, then add $\epsilon$ to FIRST($X$)
- Repeat adding symbols to FIRST($X$) by looking at productions

$$X \rightarrow Y_1 Y_2 \dots Y_k$$

(see book) until all FIRST sets are stable



## FIRST (Example)

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid \textbf{id} \end{aligned}$$

$$\begin{aligned} \text{FIRST}(E) &= \text{FIRST}(T) = \text{FIRST}(F) = \{(, \textbf{id}\} \\ \text{FIRST}(E') &= \{+, \epsilon\} \\ \text{FIRST}(T') &= \{*, \epsilon\} \end{aligned}$$



## FOLLOW

- Let $A$ be nonterminal
- FOLLOW($A$) is set of terminals/tokens that can appear immediately to the right of $A$ in sentential form:

$$\text{FOLLOW}(A) = \{a \mid S \stackrel{*}{\Rightarrow} \alpha A a \beta\}$$

- Compute FOLLOW($A$) for all nonterminals $A$
See book



## FIRST and FOLLOW (Example)

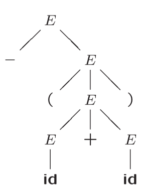
$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid \textbf{id} \end{aligned}$$

$$\begin{aligned} \text{FIRST}(E) &= \text{FIRST}(T) = \text{FIRST}(F) = \{(, \textbf{id}\} \\ \text{FIRST}(E') &= \{+, \epsilon\} \\ \text{FIRST}(T') &= \{*, \epsilon\} \\ \text{FOLLOW}(E) &= \text{FOLLOW}(E') = \{), \$\} \\ \text{FOLLOW}(T) &= \text{FOLLOW}(T') = \{+, ), \$\} \\ \text{FOLLOW}(F) &= \{*, +, ), \$\} \end{aligned}$$



## Parse Trees and Derivations

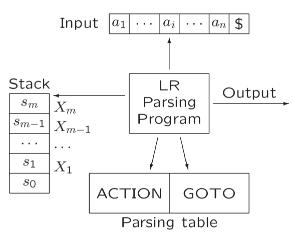
$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\textbf{id}+E) \underset{lm}{\Rightarrow} -(\textbf{id}+\textbf{id})$$




| | | |
|---|---|---|
| Leftmost derivation | ≈ | WLR construction tree |
| | ≈ | top-down parsing |
| Rightmost derivation | ≈ | WRL construction tree |
| Bottom-up parsing | ≈ | LRW construction tree |
| | ≈ | rightmost derivation in reverse |



## LR Parsing



## Simple LR Parsing

States are sets of LR(0) items

Production $A \to XYZ$ yields four items:

$$\begin{aligned} A &\to \cdot XYZ \\ A &\to X\cdot YZ \\ A &\to XY\cdot Z \\ A &\to XYZ\cdot \end{aligned}$$

Item indicates how much of production we have seen in input

LR(0) items are combined in sets

Canonical LR(0) collection is specific collection of item sets
These item sets are the states in LR(0) automaton,
a DFA that is used for making parsing decisions



## Closure of Item Sets

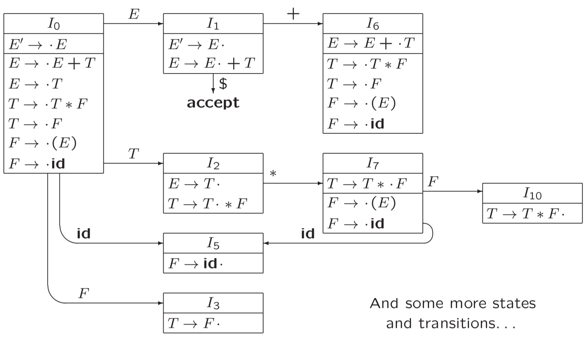
- – Consider $A \to \alpha \cdot B\beta$
  - We expect to see substring derivable from $B\beta$, with prefix derivable from $B$, by applying $B$-production
  - Hence, add $B \to \cdot\gamma$ for all $B \to \gamma$
- Let $I$ be item set
  1. Add every item in $I$ to CLOSURE($I$)
  2. Repeat

     If $A \to \alpha \cdot B\beta$ is in CLOSURE($I$) and $B \to \gamma$ is production, then add $B \to \cdot\gamma$ to CLOSURE($I$)

     until no more new items are added



## LR(0) Automaton (Example)




And some more states and transitions. . .



## Possible Actions in SLR Parsing

For state $i$ and input symbol $a$,

- if $[A \to \alpha \cdot a\beta]$ is in $I_i$ and GOTO$(I_i, a) = I_j$ then shift $j$ is possible
  ($a$ must be terminal, not \$)
- if $[A \to \alpha\cdot]$ is in $I_i$ and $a \in$ FOLLOW($A$), then reduce $A \to \alpha$ is possible ($A$ may not be $S'$)
- if $[S' \to S\cdot]$ is in $I_i$ and $a = \$$, then accept is possible

If conflicting actions result from this, then grammar is not SLR(1)



## Behaviour of LR Parser

LR parser configuration is pair (stack contents, remaining input):

$$(s_0 s_1 s_2 \ldots s_m, a_i a_{i+1} \ldots a_n\$)$$

which represents right-sentential form

$$X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n$$

1. If ACTION$[s_m, a_i]$ = shift $s$, then push $s$ and advance input:
   $$(s_0 s_1 s_2 \ldots s_m s, a_{i+1} \ldots a_n\$)$$
2. If ACTION$[s_m, a_i]$ = reduce $A \to \beta$, where $|\beta| = r$, then pop $r$ symbols. If GOTO$[s_{m-r}, A] = s$, then push $s$:
   $$(s_0 s_1 s_2 \ldots s_{m-r} s, a_i a_{i+1} \ldots a_n\$)$$
3. If ACTION$[s_m, a_i]$ = accept, then stop
4. If ACTION$[s_m, a_i]$ = error, then call error recovery routine



## SLR Parsing Table (Example)

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to$ **id**

| State | ACTION | | | | | | GOTO | | |
|---|---|---|---|---|---|---|---|---|---|
| | **id** | + | * | ( | ) | \$ | $E$ | $T$ | $F$ |
| 0 | s5 | | | s4 | | | 1 | 2 | 3 |
| 1 | | s6 | | | | acc | | | |
| 2 | | r2 | s7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | s5 | | | s4 | | | 8 | 2 | 3 |
| 5 | | r6 | r6 | | r6 | r6 | | | |
| 6 | s5 | | | s4 | | | | 9 | 3 |
| 7 | s5 | | | s4 | | | | | 10 |
| 8 | | s6 | | | s11 | | | | |
| 9 | | r1 | s7 | | r1 | r1 | | | |
| 10 | | r3 | r3 | | r3 | r3 | | | |
| 11 | | r5 | r5 | | r5 | r5 | | | |

Blank means error

| Line | Stack | Symbols | Input | Action |
|---|---|---|---|---|
| (1) | 0 | \$ | **id** $*$ **id**\$ | shift to 5 |
| (2) | 05 | \$**id** | $*$**id**\$ | reduce by $F \to$ **id** |
| (3) | 03 | \$$F$ | $*$**id**\$ | . . . |



## Exercise

(Derived from problem 1b from exam, 29 January 2002)

- Determine FIRST and FOLLOW for nonterminals
- Construct LR(0) automaton
- Construct parsing table
- Parse the string $p\ q + -\ p$



## 6.7 Backpatching

- Code generation problem:
  - Labels (addresses) that control must go to may not be known at the time that jump statements are generated
- One solution:
  - Separate pass to bind labels to addresses
- Other solution: backpatching
  - Generate jump statements with empty target
  - Add such statements to a list
  - Fill in labels when proper label is determined

## Backpatching

- Synthesized attributes *B.truelist*, *B.falselist*, *S.nextlist* containing lists of jumps

- Three functions

  1. *makelist*$(i)$ creates new list containing index $i$
  2. *merge*$(p_1, p_2)$ concatenates lists pointed to by $p_1$ and $p_2$
  3. *backpatch*$(p, i)$ inserts $i$ as target label for each instruction on list pointed to by $p$



## Translation Scheme for Backpatching

(Boolean Expressions)

| | |
|---|---|
| $B \to B_1 \| \| M B_2$ | { *backpatch*($B_1$.*falselist*, $M$.*instr*); $B$.*truelist* = *merge*($B_1$.*truelist*, $B_2$.*truelist*); } $B$.*falselist* = $B_2$.*falselist*; |
| $B \to B_1 \&\& M B_2$ | { *backpatch*($B_1$.*truelist*, $M$.*instr*); $B$.*truelist* = $B_2$.*truelist*; $B$.*falselist* = *merge*($B_1$.*falselist*, $B_2$.*falselist*); } |
| $B \to (\ B_1\ )$ | { $B$.*truelist* = $B_1$.*truelist*; $B$.*falselist* = $B_1$.*falselist*; } |
| $B \to E_1$ **rel** $E_2$ | { $B$.*truelist* = *makelist*(*nextinstr*); $B$.*falselist* = *makelist*(*nextinstr* + 1); *gen*('`if`' $E_1$.*addr* **rel**.*op* $E_2$.*addr* '`goto _`'); *gen*('`goto _`'); } |
| $M \to \epsilon$ | { $M$.*instr* = *nextinstr*; } |



## Exercise

(Derived from problem 6.7.1(a) from second edition book)

- Construct the parse tree for the following boolean expression:

  `a==b && (c==d || e==f)`

- Using the translation scheme of Fig. 6.43, translate the above expression.
  Show the true and false lists for each subexpression.
  You may assume the address of the first instruction generated is 100.

The semantic actions needed from Fig. 6.43 are listed in the previous slide. They are also listed in the next slide, with a different numbering of the variables. This numbering may be more useful for the exercise.



## Translation Scheme for Backpatching

(Boolean Expressions)

| | |
|---|---|
| $B_3 \to B_4 \| \| M_2 B_5$ | { *backpatch*($B_4$.*falselist*, $M_2$.*instr*); $B_3$.*truelist* = *merge*($B_4$.*truelist*, $B_5$.*truelist*); } $B_3$.*falselist* = $B_5$.*falselist*; |
| $B \to B_1 \&\& M_1 B_2$ | { *backpatch*($B_1$.*truelist*, $M_1$.*instr*); $B$.*truelist* = $B_2$.*truelist*; $B$.*falselist* = *merge*($B_1$.*falselist*, $B_2$.*falselist*); } |
| $B_2 \to (\ B_3\ )$ | { $B_2$.*truelist* = $B_3$.*truelist*; $B_2$.*falselist* = $B_3$.*falselist*; } |
| $B \to E_1$ **rel** $E_2$ | { $B$.*truelist* = *makelist*(*nextinstr*); $B$.*falselist* = *makelist*(*nextinstr* + 1); *gen*('`if`' $E_1$.*addr* **rel**.*op* $E_2$.*addr* '`goto _`'); *gen*('`goto _`'); } |
| $M \to \epsilon$ | { $M$.*instr* = *nextinstr*; } |



## Translation Scheme for Backpatching

(Flow-of-Control Statements)

| | |
|---|---|
| $S \to$ **if** $(B)\ M S_1$ | { *backpatch*($B$.*truelist*, $M$.*instr*); $S$.*nextlist* = *merge*($B$.*falselist*, $S_1$.*nextlist*); } |
| $S \to \{L\}$ | { $S$.*nextlist* = $L$.*nextlist*; } |
| $S \to A;$ | { $S$.*nextlist* = **null**; } |
| $M \to \epsilon$ | { $M$.*instr* = *nextinstr*; } |
| $L \to L_1 M S$ | { *backpatch*($L_1$.*nextlist*, $M$.*instr*); $L$.*nextlist* = $S$.*nextlist*; } |
| $L \to S$ | { $L$.*nextlist* = $S$.*nextlist*; } |



## Exercise

(Extension of problem 6.7.1(a) from second edition book)

- Construct the parse tree for the following 'program':

  `{ if (a==b && (c==d || e==f)) x=1; y=x+1 }`

- Using the translation scheme of Fig. 6.43 and Fig. 6.46, translate the above program.
  Show the next list for each statement or statement list.
  You may assume the address of the first instruction generated is 100.

The semantic actions needed from Fig. 6.46 are listed in the previous slide. They are also listed in the next slide, with a different numbering of the variables. This numbering may be more useful for the exercise.



## Translation Scheme for Backpatching

(Flow-of-Control Statements)

| | |
|---|---|
| $S_2 \to$ **if** $(B)\ M_4 S_3$ | { *backpatch*($B$.*truelist*, $M_4$.*instr*); $S_2$.*nextlist* = *merge*($B$.*falselist*, $S_3$.*nextlist*); } |
| $S \to \{L\}$ | { $S$.*nextlist* = $L$.*nextlist*; } |
| $S_3 \to A_1;$ | { $S$.*nextlist* = **null**; } |
| $M \to \epsilon$ | { $M$.*instr* = *nextinstr*; } |
| $L \to L_1 M_3 S_1$ | { *backpatch*($L_1$.*nextlist*, $M_3$.*instr*); $L$.*nextlist* = $S_1$.*nextlist*; } |
| $L_1 \to S_2$ | { $L_1$.*nextlist* = $S_2$.*nextlist*; } |



## En verder...

- Dinsdag 4 december: practicum over opdracht 4
- Maandag 10 december: inleveren opdracht 4
- Vrijdag 21 december, 10:00 – 13:00: tentamen
- Tevoren: vragenuur?

# Compiler constructie

werkcollege 9
SLR Parsing / Backpatching

Chapters for reading:
4.4.2, 4.5, 4.6, 6.7