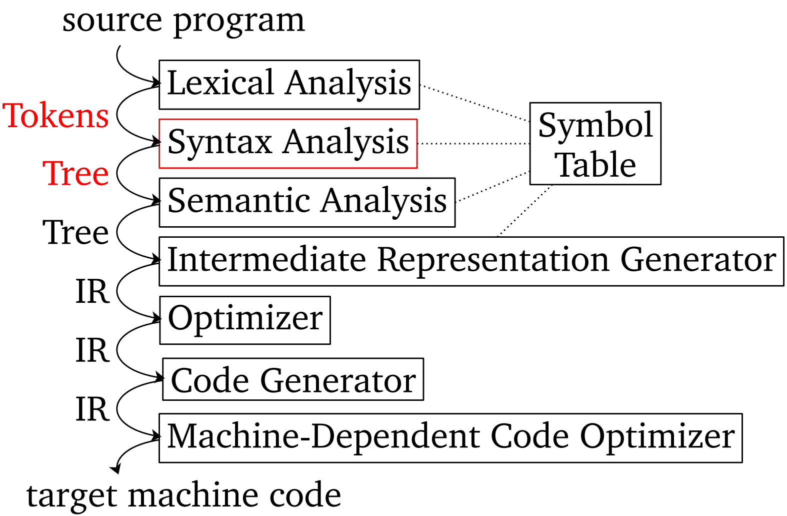
Bottom-Up Parsing

Eva Rose Kristoffer Rose

NYU Courant Institute
Compiler Construction (CSCI-GA.2130-001)
http://cs.nyu.edu/courses/fall14/CSCI-GA.2130-001/lecture-11.pdf

November 20, 2014

Second compilation phase



A Left Recursive Grammar

$$
\begin{aligned}
E &\to E + T \mid T \\
T &\to T * F \mid F \\
F &\to (E) \mid \texttt{id}
\end{aligned}
\tag{4.1}
$$

A Left Recursive Grammar

$$\begin{aligned} E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (\ E\ ) \mid \texttt{id} \end{aligned} \tag{4.1}$$

Bottom-Up Construction

Parser Categories

- **L** Left (to right)
- **R** (use) Rightmost derivation
- ***k*** look-ahead count

LR(0) Easy parsers—include SLR, LALR.

LR(1) More powerful, less practical:

- most general non-backtracking method known
- detects errors early*
- handles grammar superset of predictive parsers
- hard to hand-code
- hard to debug
- *error recovery problematic

Parser Categories

- **L** Left (to right)
- **R** (use) Rightmost derivation
- ***k*** look-ahead count
- **LR(0)** Easy parsers—include SLR, LALR.
- **LR(1)** More powerful, less practical:
  - most general non-backtracking method known
  - detects errors early*
  - handles grammar superset of predictive parsers
  - hard to hand-code
  - hard to debug
  - *error recovery problematic

Parser Categories

L	Left (to right)
R	(use) Rightmost derivation
k	look-ahead count
LR(0)	Easy parsers—include SLR, LALR.
LR(1)	More powerful, less practical:

- most general non-backtracking method known
- detects errors early*
- handles grammar superset of predictive parsers
- hard to hand-code
- hard to debug
- *error recovery problematic

Parser Categories

L Left (to right)
R (use) Rightmost derivation
k look-ahead count
LR(0) Easy parsers—include SLR, LALR.
LR(1) More powerful, less practical:

- most general non-backtracking method known
- detects errors early*
- handles grammar superset of predictive parsers
- hard to hand-code
- hard to debug
- *error recovery problematic

$LR(k)$

$LR(k)$ automaton:

- 2 basic moves: shift token, reduce production
- 2 other moves: accept, error

Concepts:

- An $LR(0)$ item is a rule with a dot at some position: $A \to .BCD, A \to B.CD, \ldots$
- An $LR(0)$ state is a set of items.
- Canonical $LR(0)$ collection: finite automaton used to make parsing decisions

$LR(k)$

$LR(k)$ automaton:

- 2 basic moves: shift token, reduce production
- 2 other moves: accept, error

Concepts:

- An $LR(0)$ item is a rule with a dot at some position: $A \to .BCD$, $A \to B.CD$, ...
- An $LR(0)$ state is a set of items.
- Canonical $LR(0)$ collection: finite automaton used to make parsing decisions

Closure Construction

For Grammar G need—

- Augmented grammar G'
  - Just add rules $S' \to S$
- closure function for set of items I, closure(I):
  1. add I to closure(I).
  2. if $A \to \alpha.B\beta$ in closure(I) then add $B \to .\gamma$ (for each possible γ)
- GOTO function

Example

1. $E' \to E$
2. $E \to E + T$
3. $E \to T$
4. $T \to T * F$
5. $T \to F$
6. $F \to (E)$
7. $F \to \texttt{id}$

Construction (I)

$$
\begin{aligned}
I &= \{[E' \to .E]\} \\
I_0 &= \mathrm{closure}(I) \\
&= \{[E' \to .E], [E \to .E{+}T], [E \to .T], [T \to .T{*}F], [T \to .F], \\
&\qquad [F \to .(E)], [F \to .\mathtt{id}]\}
\end{aligned}
$$

Construction (II)

For items I, $\forall x$ in a set of symbols following dot.

$$\text{GOTO}(I,x) = \text{closure}(\{[A \to \alpha x.\beta] \mid [A \to \alpha.x\beta] \in I\})$$

gives

$$I_1 = \text{GOTO}(I_0, E) = \{[E' \to E.], [E' \to E. + T]\}$$
$$I_2 = \text{GOTO}(I_0, T) = \{[E \to T.], [T \to T. * F]\}$$
$$I_3 = \text{GOTO}(I_0, F) = \{[T \to F.]\}$$
$$I_4 = \text{GOTO}(I_0, () = \{[F \to (.E)], [E \to .E + T], [E \to .T], [T \to .T * F], [T \to .F], [F \to .(E)], [F \to .\mathtt{id}]\}$$
$$I_5 = \text{GOTO}(I_0, \mathtt{id}) = \{[F \to \mathtt{id}.]\}$$

Algorithm

```
initialize C = { closure({[S'→.S]}) }
repeat for all I∈C, for all x,
  if GOTO(I,x) ≠ Ø and GOTO(I,x) ∉ C
    add GOTO(I,x) to C
```

In practice find repeating items, *i.e.*, `GOTO(I4,T) == I2`

- A final item is one with the `.` at the end.
- If all final items are in states by themselves then the grammar is $LR(0)$,
- otherwise there is a shift-reduce or reduce-reduce conflict in $LR(0)$.

SLR Trick

Use Follow Sets to decide when to reduce: $SLR(1)$.
If they overlap we can look 2 symbols ahead, *etc.*

Automaton

- 2 basic moves: shift, reduce
- 2 other moves: accept, error

Shift-Reduce

STACK	INPUT	ACTION
$\$$	$\texttt{id}_1 * \texttt{id}_2\ \$$	shift
$\$\ \texttt{id}_1$	$* \texttt{id}_2\ \$$	reduce by $F \to \texttt{id}$
$\$\ F$	$* \texttt{id}_2\ \$$	reduce by $T \to F$
$\$\ T$	$* \texttt{id}_2\ \$$	shift
$\$\ T\ *$	$\texttt{id}_2\ \$$	shift
$\$\ T\ *\ \texttt{id}_2$	$\$$	reduce by $F \to \texttt{id}$
$\$\ T\ *\ F$	$\$$	reduce by $T \to T * F$
$\$\ T$	$\$$	reduce by $E \to T$
$\$\ E$	$\$$	accept

LR components

- Input "pointer."
- Driver.
- Action/GOTO table.
- Stack.

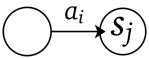
Configuration

Stack $s_0 \ldots s_m$ (both productions and tokens)

Input $a_1 \ldots a_n\$$

Moves. . .

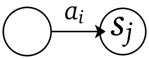
shift j consume symbol a_i, push s_j.



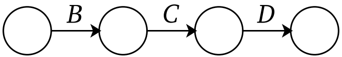
reduce $A \to \beta$ pop $|\beta|$ states from stack
push GOTO($s_{m-|\beta|}$, A) on stack.
With $A \to .BCD$:

Moves...

shift j consume symbol a_i, push s_j.



reduce $A \to \beta$ pop $|\beta|$ states from stack
push GOTO($s_{m-|\beta|}$, A) on stack.
With $A \to .BCD$:



Example

Example.

Remark

Do you see the bottom-up?

Universal LR Driver

```
push S0 onto empty stack
read first input token
repeat:
  let s = state on top of stack
  if action(s,a) = shift t:
    push t onto stack
    a := next symbol
  else if action(s,a) = reduce A→β
    pop |β| states off stack
    let t be new top of stack
    push GOTO(t,A) onto stack
    synthesize attributes for A→β
  else if action(s,a) = accept
    break
  else
    call error recovery
```

LR(1) Items

- Extend LR(0) items with look-ahead: $[A \to \alpha.\beta, a]$
- Start with $[S' \to .S, \$]$
- Example: $[E' \to .E, \$]$
  closure:
  $[E \to .E{+}T, \$, +], [E \to .T, \$, +], [T \to .T{*}F, \$, +, *], \ldots$

LR(1) Items

- Extend LR(0) items with look-ahead: $[A \to \alpha.\beta, a]$
- Start with $[S' \to .S, \$]$
- Example: $[E' \to .E, \$]$
  closure:
  $[E \to .E{+}T, \$, +], [E \to .T, \$, +], [T \to .T{*}F, \$, +, *], \ldots$

LR(1) Items

- Extend LR(0) items with look-ahead: $[A \to \alpha.\beta, a]$
- Start with $[S' \to .S, \$]$
- Example: $[E' \to .E, \$]$
  closure:
  $[E \to \ .E{+}T, \$, +], [E \to \ .T, \$, +], [T \to \ .T{*}F, \$, +, *], \ldots$

Algorithm (with Lookahead)

```
closure(I):
  repeat for all [A→α.Bβ,a] ∈ I,
         for all B→γ in G',
         for all b in FIRST(βa)
    add [B→.γ,b] to I
  until no more items can be added

goto(I,x):
  J = Ø
  for all [A→α.xβ,a] ∈ I
    add [A→αx.β,a] to J
  return closure(J)
```

Remarks

- Unlike LR(0)→SLR step no need to deal with ambiguous states of LR(1)
- For LALR(1), systematically *merge* certain LR(1) states ⇒ compact tables
- To build LALR from LR(0) consider *path that led to it* ⇒ efficient table construction

Remarks

- Unlike LR(0)→SLR step no need to deal with ambiguous states of LR(1)
- For LALR(1), systematically *merge* certain LR(1) states ⇒ compact tables
- To build LALR from LR(0) consider *path that led to it* ⇒ efficient table construction

Remarks

- Unlike LR(0)→SLR step no need to deal with ambiguous states of LR(1)
- For LALR(1), systematically *merge* certain LR(1) states ⇒ compact tables
- To build LALR from LR(0) consider *path that led to it* ⇒ efficient table construction

Summary

- LR(k) parsers are also called bottom-up parsers in contrast to LL(k) top-down parsers.
- LR(0) is efficient, LR($k > 0$) is not.
- SLR and LALR improve LR(0) in important ways but remain efficient.
- SLR(k) and LALR(k) are both less than LR(k).
- The main source of conflicts are shift-reduce and reduce-reduce conflicts;
  resolving in nonstandard ways allows for (some) ambiguous grammars

Summary

- LR(k) parsers are also called bottom-up parsers in contrast to LL(k) top-down parsers.
- LR(0) is efficient, LR($k > 0$) is not.
- SLR and LALR improve LR(0) in important ways but remain efficient.
- SLR(k) and LALR(k) are both less than LR(k).
- The main source of conflicts are shift-reduce and reduce-reduce conflicts;
  resolving in nonstandard ways allows for (some) ambiguous grammars

Summary

- LR(k) parsers are also called bottom-up parsers in contrast to LL(k) top-down parsers.
- LR(0) is efficient, LR($k > 0$) is not.
- SLR and LALR improve LR(0) in important ways but remain efficient.
- SLR(k) and LALR(k) are both less than LR(k).
- The main source of conflicts are shift-reduce and reduce-reduce conflicts;
  resolving in nonstandard ways allows for (some) ambiguous grammars

Summary

- LR(k) parsers are also called bottom-up parsers in contrast to LL(k) top-down parsers.
- LR(0) is efficient, LR($k > 0$) is not.
- SLR and LALR improve LR(0) in important ways but remain efficient.
- SLR(k) and LALR(k) are both less than LR(k).
- The main source of conflicts are shift-reduce and reduce-reduce conflicts;
  resolving in nonstandard ways allows for (some) ambiguous grammars

Summary

- LR(k) parsers are also called bottom-up parsers in contrast to LL(k) top-down parsers.
- LR(0) is efficient, LR($k > 0$) is not.
- SLR and LALR improve LR(0) in important ways but remain efficient.
- SLR(k) and LALR(k) are both less than LR(k).
- The main source of conflicts are shift-reduce and reduce-reduce conflicts;
  resolving in nonstandard ways allows for (some) ambiguous grammars

Questions?

1. Parsers (recap)
2. LR(0)
3. LR Parsing
4. LR(1) and LALR(1)
5. ARM Recap

ARM32 Instruction Subset

```
MOV reg,arg              reg = R0 | R1 | ... | R15 | SP | LR | PC

ADD reg,reg1,arg2        arg = #imm8 | reg | reg,LSL #imm5 | reg,LSR #imm5 | &label
SUB reg,reg1,arg2
MUL reg,reg1,arg2        immn = n-bit constant
AND reg,reg1,arg2
ORR reg,reg1,arg2
EOR reg,reg1,arg2

CMP reg,arg

B   label
Bcd label                cd = EQ | NE | GT | LT | GE | LE
BL  label

LDRb reg,mem             mem = [reg,arg]   b = B?
STRb reg,mem

LDMFD reg!,mreg          mreg = {reg,...,reg}
STMFD reg!,mreg
```

Questions?