TDDD16 Compilers and Interpreters

TDDB44 Compiler Construction

LR Parsing, Part 2

Constructing Parse Tables

Parse table construction

Grammar conflict handling

Categories of LR Grammars and Parsers

Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2008.

An NFA Recognizing Viable Prefixes

- A.k.a. the ”*characteristic finite automaton*” for a grammar G
- States: LR(0) items (= context-free items) of extend. grammar
- Input stream: The grammar symbols on the stack
- Start state: [S' → –|.S]   Final state: [S' → –|S.]
- Transitions:
  - ”move dot across symbol” if symbol found next on stack:
    A → α.Bγ to A → αB.γ
    A → α.bγ to A → αb.γ
  - ε-transitions to LR(0)-items for nonterminal productions from items where the dot precedes that nonterminal:
    A → α.Bγ to B →.β
- (Example: see whiteboard)

Computing the Closure

For a set *I* of LR(0) items compute *Closure*(I):

1. *Closure*(*I*) := *I*
2. If ∃ [A→α.Bβ] in *Closure*(*I*)
   and ∃ production B → γ
   then add [B →.γ] to *Closure*(*I*)   (if not already there)
3. Repeat Step 2 until no more items can be added to *Closure*(*I*).

Remarks:

- *For s*=[A → α.Bγ], *Closure*(s) contains all NFA states reachable from *s* via ε-transitions, i.e., starting from which any substring derivable from Bβ could be recognized. A.k.a. ε-closure(*s*).
- Then apply the well-known subset construction to transform Closure-NFA -> DFA.
- DFA states will be sets unioning closures of NFA states

Representing Sets of Items

- Any item [A → α.β] can be represented by 2 integers:
  - production number
  - position of the dot within the RHS of that production
- The resulting sets often contain ”closure” items (where the dot is at the beginning of the RHS).
  - Can easily be reconstructed (on demand) from other (”kernel”) items
    - **Kernel items**: start state [S' → –|.S], plus all items where the dot is not at the left end.
  - Store only kernel items explicitly, to save space

GOTO Function and DFA States

Given: Set *I* of items, grammar symbol X

- GOTO(*I*, X) := ∪ $_{[A→α.Xβ] \text{ in } I}$ *Closure* ({ [A → αX.β] })
  - To become the state transitions in the DFA
- Now do the **subset construction** to obtain the DFA states:

  C := *Closure*({ [S' → –|.S] })   // C: Set of sets of NFA states
  repeat
   for each set of items *I* of *C*:
    for each grammar symbol X
     if (GOTO(*I*,X) is not empty and not in *C*)
      add GOTO(*I*,X) to *C*
  until no new states are added to *C* on a round

Resulting DFA

- (Example: see whiteboard)
- All states correspond to some viable prefix
- Final states: contain at least one item with dot to the right
  - recognized some handle → reduce *may* (*must*) follow
- Other states: handle recognition incomplete -> shift will follow
- The DFA is also called the GOTO graph (not the same as the LR GOTO Table!!).
- This automaton is deterministic as a FA (i.e., selecting transitions considering only input symbol consumption) but can still be nondeterministic as a pushdown automaton (e.g., in state I_1 above: to reduce or not to reduce?)

From DFA to parser tables: ACTION

1. For each DFA transition $I_i \rightarrow I_j$ reading a terminal a in Σ (thus, I_i contains items of kind [X→ α.aβ])
   - enter S j (shift, new state I_j) in ACTION
2. For each DFA final state I_i (containing a complete item [X→ α.])
   - enter R x (reduce, x = prod. rule number for X→ α) in ACTION[i, t] ...
     - LR(0) parser: for *all* t in Σ (all entries in row i)
     - SLR(1) parser: for all t in LA_{SLR}(i,[X→ α.]) = $FOLLOW_1$(X)
     - LALR(1) parser: for all t in LA_{LALR}(i,[X→ α.]) (see later)

ACTION table:

state	|--	,	a	b
0	X	X	S4	S5
1	A	S2	*	*
2	X	X	S4	S5
3	R1	R1	*	*
4	R3	R3	*	*
5	R4	R4	*	*
6	R2	R2	*	*

From DFA to parser tables: GOTO Table

1. For each DFA transition $I_i \rightarrow I_j$ reading nonterminal A (i.e., I_i contains an item [X → α.Aβ])
   - enter GOTO[i , A] = j

GOTO table:

state	L	E
0	1	6
1	*	*
2	*	3
3	*	*
4	*	*
5	*	*
6	*	*

Conflicts and Categories of LR Grammars and Parsers

Peter Fritzson, Christoph Kessler, IDA, Linköpings universitet, 2008.

Conflict Examples in LR Grammars

- **Shift – Reduce conflict:**
  - E → id + E (shift +)
    | id (reduce id)
- **Reduce – Reduce conflict:**
  - E → id (reduce id)
    Pcall → id (reduce id)
- (Shift – Accept conflict)
  - S' → L (accept)
    L → L , E (shift ,)

Conflicts in LR Grammars

Observe conflicts in DFA (GOTO graph) kernels or at the latest when filling the ACTION table.

- **Shift-Reduce conflict**
  - A DFA accepting state has an outgoing transition, i.e. contains items [X→α.] and [Y→β.Zγ] for some Z in N∪Σ
- **Reduce-Reduce conflict**
  - A DFA accepting state can reduce for multiple nonterminals i.e. contains at least 2 items [X→α.] and [Y→β.], X != Y
- **(Shift/Reduce-Accept conflict)**
  - A DFA accepting state containing [S'→S.|--] contains another item [X→αS.] or [X→αS.bβ]

Only for LR(0) grammars there are no conflicts.

Handling Conflicts in LR Grammars

(Overview):

- Use lookahead
  - if lucky, the LR(0) states + a few fixed lookahead sets are sufficient to eliminate all conflicts in the LR(0)-DFA
    - SLR(1), LALR(1)
  - otherwise, use LR(1) items [X→α.β, a] (a is look-ahead) to build new, larger NFA/DFA
    - expensive (many items/states → very large tables)
  - if still conflicts, may try again with *k*>1 → even larger tables
- Rewrite the grammar (factoring / expansion) and retry...
- If nothing helps, re-design your language syntax
  - Some grammars are not LR(*k*) for any constant *k* and cannot be made LR(*k*) by rewriting either

Look-Ahead (LA) Sets

- For a LR(0) item $[X \rightarrow \alpha.\beta]$ in DFA-state I_i, define
  lookahead set LA(I_i, $[X \rightarrow \alpha.\beta]$) (a subset of Σ)

For SLR(1), LALR(1) etc., the LA sets only differ for reduce items:

- **For SLR(1):**
  $LA_{SLR}(I_i, [X \rightarrow \alpha.]) = \{ a \text{ in } \Sigma: S' \Rightarrow^* \beta X a \gamma \} = FOLLOW_1(X)$
  for all I_i with $[X \rightarrow \alpha.]$ in I_i
  - depends on nonterminal X only, not on state I_i
- **For LALR(1):**
  $LA_{LALR}(I_i, [X \rightarrow \alpha.]) = \{ a \text{ in } \Sigma: S' \Rightarrow^* \beta X a w$ and the LR(0)-DFA started in I_0 reaches I_i after reading $\beta\alpha \}$
  - usually a subset of $FOLLOW_1(X)$, i.e. of SLR LA set

Made it simple: Is my grammar SLR(1) ?

- Construct the (LR(0)-item) characteristic NFA and its equivalent DFA (= GOTO graph) as above.
- Consider all conflicts in the DFA states:
  - **Shift-Reduce:** [X → α.] [Y → β.bγ] ... —b→ ...

    Consider all pairs of conflicting items [X→α.], [Y→β.bγ]:
    If b in $FOLLOW_1(X)$ for any of these → not SLR(1).
  - **Reduce-Reduce:** [X → α.] [Y → β.] ...

    Consider all pairs of conflicting items [X→α.], [Y→β.]:
    If $FOLLOW_1(X)$ intersects with $FOLLOW_1(Y)$ → not SLR(1)
  - (**Shift-Accept**: similar to Shift-Reduce)

Example: L-Values in C Language

- L-values on left hand side of assignment.
  Part of a C grammar:
  1. S' → S
  2. S → L = R
  3. | R
  4. L → *R
  5. | **id**
  6. R → L
- Avoids that R (for R-values) appears as LHS of assignments
- But *R = ... is ok.
- This grammar is LALR(1) but not SLR(1):

Example (cont.)

LR(0) parser has a shift-reduce conflict in kernel of state I_2:

- I_0 = { [S'→.S], [S→.L=R], [S→.R], [L→.*R], [L→.**id**], R→.L] }
- I_1 = { [S'->S.] }
- I_2 = { [S->L.=R], [R->L.] } — Shift = or reduce to R?
- I_3 = { [S->R.] }
- I_4 = { [L->*.R], [R->.L], [L->.*R], [L->.**id**] }
- I_5 = { [L->**id**.] }
- I_6 = { [S->L=.R], [R->.L], [L->.*R], L->.**id**] }
- I_7 = { [L->*R.] }
- I_8 = { [R->L.] }
- I_9 = { [S->L=R.] }

$FOLLOW_1(R)$ = { |– , = } → SLR(1) still shift-reduce conflict in I_2
as = does not disambiguate

Example (cont.)

- I_0 = { [S'->.S], [S->.L=R], [S->.R], [L->.*R], [L->.**id**], R->.L] }
- I_1 = { [S'->S.] }
- I_2 = { [S->L.=R], [R->L.] }
- I_3 = { [S->R.] }
- I_4 = { [L->*.R], [R->.L], [L->.*R], [L->.**id**] }
- I_5 = { [L->**id**.] }
- I_6 = { [S->L=.R], [R->.L], [L->.*R], L->.**id**] }
- I_7 = { [L->*R.] }
- I_8 = { [R->L.] }
- I_9 = { [S->L=R.] }

LA_{LALR} (I_2, [R->L.]) = { |– } → LALR(1) parser is conflict-free
as computation path I_0...I_2 does not really allow = following R.
= can only occur after R if "*R" was encountered before.

LALR(1) Parser Construction

Method 1: (simple but not practical)

1. Construct the LR(1) items (see later). (If there is already a conflict, stop.)
2. Look for sets of LR(1) items that have the same kernel, and merge them.
3. Construct the ACTION table as for LR(1).
   If a conflict is detected, the grammar is not LALR(1).
4. Construct the GOTO function:
   For each merged $J = I_1 \cup I_2 \cup ... \cup I_r$,
   the kernels of GOTO(I_1,X), ..., GOTO(I_r,X) are identical because the kernels of I_1,...,I_r are identical.
   Set GOTO(J, X) := $\bigcup$ { I: I has the same kernel as GOTO(I_1,X) }

Method 2: (details see textbook)

1. Start from LR(0) items and construct kernels of DFA states I_0, I_1, ...
2. Compute lookahead sets by propagation along the GOTO(I_j,X) edges (fixed point iteration).

Solve Conflicts by Rewriting the Grammar

- Eliminate Reduce-Reduce Conflict:

Factoring

S → (A) | (B)
A → char | integer | ident
B → float | double | ident

[A → ident .]
[B → ident .]
...

factor ident

S → (A) | (B) | (C)
A → char | integer
B → float | double
C → ident

- Eliminate Shift-Reduce Conflict: (one token lookahead: '(')

Inline-Expansion

S → (A) | OptY (B)
OptY → Y | ε
Y → ...
A → ...
B → ...

[S → . (A)]
[S → . OptY (B)]
[OptY → .Y]
[OptY → .ε]
[OptY → ε .]
[Y → ...] ...

expand OptY

S → (A) | (B)
| Y (B)
Y → ...
A → ...
B → ...

LR(*k*) Grammar - Formal Definition

p.116

- Let G' be the augmented grammar for G
(i.e., extended by new start symbol S'
and production rule S' –> S |--)
- G is called a **LR(*k*) grammar** if
  - S' $_{rm}$=>* αXw $_{rm}$=> αβw and
  - S' $_{rm}$=>* γYx $_{rm}$=> αβy and
  - w[1:*k*] = y[1:*k*]

imply that α = γ and X = Y and x = y = w.

i.e., considering at most *k* symbols after the handle, in each rightmost derivation the handle can be localized and the production to be applied can be determined.

Remark: w, x, y in Σ* α, β, γ in (N ∪ Σ)* X, Y in N

Example: see whiteboard

Some grammars are not LR(*k*) for any fixed *k*

- Example: S → a B c
B → b B b
| b
  - describes language { a b^{2N+1} c : N >= 0 }
- This grammar is not LR(*k*) for any fixed *k*.

Proof: As k is fixed (constant), consider for any n > *k*:

- S =>* a b^n B b^n c => a b^n <u>b</u> b^n c
- S =>* a b^{n+1} B b^{n+1} c => a b^{n+1} <u>b</u> b ...

The handle cannot be localized with only limited lookahead size k

By the LR(*k*) definition,

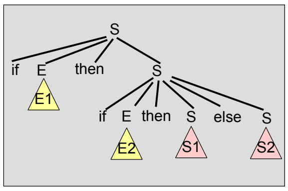
- α = a b^n β = b w = b^n c

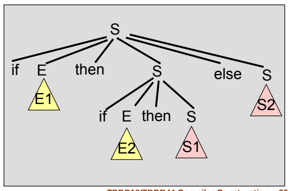
No ambiguous grammar is LR(*k*) for any fixed *k*

- S → if E then S
| if E then S else S
| *other statements*
...

is ambiguous – the following statement has 2 parse trees:

if E1 then if E2 then S1 else S2





(cont.)

- Consider situation
(configuration of shift-reduce parser)

--| ... if E then S else ... |--

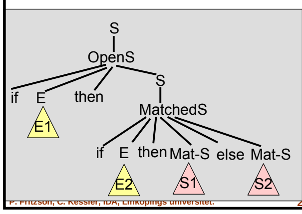
- Not clear whether to
  - shift else
(following production 2, i.e. if E then S is not handle), or
  - reduce handle <u>if E then S</u> to S (following production 1)
- Any fixed-size lookahead (else and beyond) does not help!
- Suggestion: Rewrite grammar to make it unambiguous

Rewriting the grammar...

S → MatchedS
| OpenS
MatchedS → if E then MatchedS else MatchedS
| *other statements*
OpenS → if E then S
| if E then MatchedS else OpenS
...

is no longer ambiguous

Impossible now to derive any sentential form containing an OpenS nonterminal from a MatchedS



Some grammars are not LR(*k*) for any fixed *k*

- Grammar with productions

  S → a S a | ε

  is unambiguous but not LR(*k*) for any fixed *k.* (Why?)

- An equivalent LR grammar for the same language is

  S → a a S | ε

LR(1) Items and LR(*k*) Items

LR(*k*) parser: Construction similar to LR(0) / SLR(1) parser, but plan for distinguishing between states for *k*>0 tokens **lookahead** already from the beginning

- States in the LR(0) GOTO graph may be split up

- LR(1) items:
  [A->α.β , a] for all productions A->αβ and all a in Σ
- Can be combined for lookahead symbols with equal behavior:
  [A->α.β , a|b] or [A->α.β , L] for a subset L of Σ
- Generalized to *k*>1:
  [A->α.β , $a_1 a_2 ... a_k$]

Interpretation of [A->α.β , a] in a state:

- If β not ε, ignore second component (as in LR(0))
- If β=ε, *i.e.* [A->α. , a], reduce only if next input symbol = a.

LR(1) Parser

- NFA start state is [S'->.S, |–]
- Modify computation of *Closure*(*I*), GOTO(*I*,X) and the subset computation for LR(1) items
  - Details see [ASU86, p.232] or [ALSU06, p.261]
- Can have many more states than LR(0) parser
  - Which may help to resolve some conflicts

Interesting to know...

- For each LR(*k*) grammar with some constant *k*>1 there exists an equivalent* grammar G' that is LR(1).

- For any LL(*k*) grammar there exists an equivalent LR(*k*) grammar (but not vice versa!)
  - e.g., language { $a^n b^n$: n>0 } U { $a^n c^n$: n > 0 }
    has a LR(0) grammar
    but no LL(*k*) grammar for any constant *k*.

- Some grammars are LR(0) but not LL(*k*) for any *k*
  - e.g., S → A b
    A → Aa | a (left recursion, could be rewritten)

* Two grammars are *equivalent* if they describe the same language.