TDDB29 Compilers and Interpreters
TDDB44 Compiler Construction

LR Parsing

Updated/New slide material 2007:

LR parsing concept
Parser table construction

Conflict handling

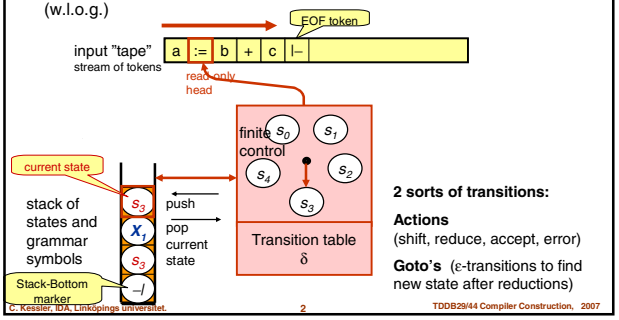
Using a parser generator
Parse Tree Generation

Christoph Kessler, IDA,
Linköpings universitet, 2007.

Pushdown Automaton for LR-Parsing

Finite-state pushdown automaton

- Stack contains alternatingly states and symbols in $N \cup \Sigma$ (w.l.o.g.)



Configurations of the LR-Parser

- Configuration = (stack contents, remaining input)
  = (s_0 X_1 s_1 ... X_{m-1} s_{m-1} X_m s_m a_i a_{i+1} ... a_n)
  current state (ToS)
- **Shift:** read current input symbol a_i and push it with new state s
  |= (s_0 X_1 s_1 ... X_{m-1} s_{m-1} X_m s_m a_i s a_{i+1} ... a_n)
- **Reduce:** read ε, pop $2r$ stack symbols for handle X_{m-r+1}...X_m, push LHS nonterminal + new state (see below)
- **Invariants**:
  - Nonterminals on stack + remaining input
    (X_1 ... X_{m-1} X_m a_i a_{i+1} ... a_n)
    is a rightmost-derived sentential form of G.
  - State on top of stack represents a viable prefix of G
    - Needs to be reconstructed after a reduce, using the GOTO table

Example: A SLR(1) Grammar

- Terminals: ,
  a
  b
- Nonterminals: <list> (or L) (is also the start symbol)
  <element> (or E)
- Productions:
  1. <list> -> <list> , <element>
  2. | <element>
  3. <element> -> a
  4. | b

Example (cont.): Extend Grammar with new start symbol

- Terminals: |– (end-of-input symbol)
  ,
  a
  b
- Nonterminals: <SYS> (or S') (new start symbol)
  <list> (or L)
  <element> (or E)
- Productions:
  0. <SYS> -> <list> |–
  1. <list> -> <list> , <element>
  2. | <element>
  3. <element> -> a
  4. | b

Example: Tables (given); Parsing input string a,b

0. S' -> L |--
1. L -> L , E
2. | E
3. E -> a
4. | b

Step	Stack	Input	Table entries
1	--| 0	**a , b** |--	ACTION[0, **a**] = S4

S = Shift
4 = successor state

ACTION table:

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

GOTO table:

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

Example: Tables (given); Parsing input string a,b

0. S' -> L |--
1. L -> L , E
2. | E
3. E -> a
4. | b

Step	Stack	Input	Table entries
1	--| 0	**a , b** |--	ACTION[0, **a**] = S4
2	--| 0**a**4	**, b** |--	ACTION[4, ,] = R3 (E -> a)

R = Reduce
3 = production rule (comment)

ACTION table:

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

GOTO table:

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

Example: Tables (given); Parsing input string a,b

0. S' -> L |--
1. L -> L , E
2. | E
3. E -> a
4. | b

Step	Stack	Input	Table entries
1	--| 0	**a , b** |--	ACTION[0, **a**] = S4
2	--| 0**a**4	**, b** |--	ACTION[4, ,] = R3 (E -> a)
	--| 0E	**, b** |--	GOTO[0, E] = 6

Reconstruct successor state after a Reduce from top stack items via the GOTO table

ACTION table:

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

GOTO table:

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

Example: Tables (given); Parsing input string a,b

0. S' -> L |--
1. L -> L , E
2. | E
3. E -> a
4. | b

Step	Stack	Input	Table entries
1	--| 0	**a , b** |--	ACTION[0, **a**] = S4
2	--| 0**a**4	**, b** |--	ACTION[4, ,] = R3 (E -> a)
	--| 0E	**, b** |--	GOTO[0, E] = 6
3	--| 0E6	**, b** |--	ACTION[6, ,] = R2 (L -> E)

... and so on ...

ACTION table:

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

GOTO table:

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

Example: Tables (given); Parsing input string a,b

0. S' -> L |--
1. L -> L , E
2. | E
3. E -> a
4. | b

Step	Stack	Input	Table entries
1	--| 0	**a , b** |--	ACTION[0, **a**] = S4
2	--| 0**a**4	**, b** |--	ACTION[4, ,] = R3 (E -> a)
	--| 0E	**, b** |--	GOTO[0, E] = 6
3	--| 0E6	**, b** |--	ACTION[6, ,] = R2 (L -> E)
	--| 0L	**, b** |--	GOTO[0, L] = 1
4	--| 0L1	**, b** |--	ACTION[1, ,] = S2
5	--| 0L1,2	**b** |--	ACTION[2, **b**] = S5
6	--| 0L1,2b5	|--	ACTION[5, |--] = R4 (E -> b)
	--| 0L1,2E	|--	GOTO[2, E] = 3
7	--| 0L1,2E3	|--	ACTION[3, |--] = R1 (L ->L,E)
	--| 0L	|--	GOTO[0, L] = 1
8	--| 1	|--	ACTION[1, |--] = A (accept)

ACTION table:

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

GOTO table:

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

Handle, Viable Prefix

117a

- Consider a rightmost derivation $S \Rightarrow^*_{rm} \beta X u \Rightarrow_{rm} \beta\alpha u$ for a context-free grammar *G*.
- α is called **handle** of the right sentential form $\beta\alpha u$
- Each prefix of $\beta\alpha$ is called a **viable prefix** of *G*.

Example: Grammar *G* with productions { S -> aSb | c }

- Right sentential forms: e.g. c, acb, aSb, aaaaaSbbbbb, ...
- For c: Handle: c Viable prefixes: ε, c
- For acb: c ε, a, ac
- For aSb: aSb ε, a, aS, aSb
- For aaSbb: aSb ε, a, aa, aaS, aaSb
- ...

Recognizing Handles

117b

How to recognize if a handle appears as the top elements on the stack?

- **Naive approach**: Examine the entire stack (e.g. from top to bottom, or vice versa) at every step
  - Leads to unnecessarily long worst-case parsing time
  - Need to actually store grammar symbols on stack
- **Idea**: Incremental handle recognition
  - Keep information about partially recognized handles (= viable prefixes) on top of stack, encoded in state
    - –> Characteristic automaton (NFA) to recognize viable pr.
    - –> DFA by subset construction with ε-closure
    - –> Parser tables (ACTION / GOTO)

A NFA Recognizing Viable Prefixes

118a

- A.k.a. the "*characteristic finite automaton*" for a grammar G
- States: LR(0) items (= context-free items) of ext. grammar
- Input stream: The grammar symbols on the stack
- Start state: [S' -> –I.S] Final state: [S' -> –IS.]
- 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

118b

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

118c

- 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'->–I.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

120a

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

- GOTO(*I*, X) := $\bigcup_{[A->\alpha.X\beta] \text{ in } I}$ *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'-> –I.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

120b

- (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.
- 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[i, a]
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 $LA_{SLR}(i,[X→\alpha.]) = FOLLOW_1(X)$
     - LALR(1) parser: for all t in $LA_{LALR}(i,[X→\alpha.])$ (see later)
   - Collision with an already existing S or R entry? Conflict!!
3. For each DFA state containing [S' → S.I--]
   - enter A in ACTION[i, I--] (accept). NB - Conflict? (as in 2.)

ACTION table:

state	I--	,	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

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

GOTO table:

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

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 $[X \rightarrow \alpha.]$ and $[Y \rightarrow \beta.Z\gamma]$ for some Z in $N \cup \Sigma$
- **Reduce-Reduce conflict**
  - A DFA accepting state can reduce for multiple nonterminals i.e. contains at least 2 items $[X \rightarrow \alpha.]$ and $[Y \rightarrow \beta.]$, X != Y
- **(Shift/Reduce-Accept conflict)**
  - A DFA accepting state containing [S'→S.I--] contains another item $[X \rightarrow \alpha S.]$ or $[X \rightarrow \alpha S.b\beta]$

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

Conflict examples

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

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 $[X \rightarrow \alpha.\beta, a]$ 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

Lookahead Sets

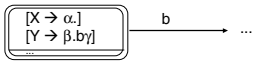
- For a LR(0) item $[X \rightarrow \alpha.\beta]$ in DFA-state I_i, define
  lookahead set $LA(I_i, [X \rightarrow \alpha.\beta])$ (a subset of Σ)

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
  - depends on state I_i

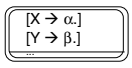
Made it simple: Is my grammar SLR(1) ?

- Construct the (LR(0)-item) characteristic NFA and its equivalent DFA (= GOTO graph) as above.
- Consider all conflicts in the DFA states:
  - **Shift-Reduce:**



Consider all pairs of conflicting items $[X \rightarrow \alpha.]$, $[Y \rightarrow \beta.b\gamma]$:
If b in $FOLLOW_1(X)$ for any of these ➔ not SLR(1).

  - **Reduce-Reduce:**



Consider all pairs of conflicting items $[X \rightarrow \alpha.]$, $[Y \rightarrow \beta.]$:
If $FOLLOW_1(X)$ intersects with $FOLLOW_1(Y)$ ➔ not SLR(1)

  - (**Shift-Accept**: similar to Shift-Reduce)

Example: L-Values in C

- 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 $I_0 \ldots I_2$ 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 $J = I_1 \cup I_2 \cup \ldots \cup I_r$,
   the kernels of GOTO(I_1,X), ..., GOTO(I_r,X) are identical because the kernels of $I_1,\ldots,I_r$ 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_i,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
  - $S' \underset{rm}{\Rightarrow}^* \alpha X w \underset{rm}{\Rightarrow} \alpha\beta w$ and
  - $S' \underset{rm}{\Rightarrow}^* \gamma Y x \underset{rm}{\Rightarrow} \alpha\beta y$ and
  - w[1:*k*] = y[1:*k*]

  imply that $\alpha = \gamma$ 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 $(N \cup \Sigma)^*$ 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 b b^n c
- S =>* a b^{n+1} B b^{n+1} c => a b^{n+1} b b^{n+1} c

By the LR(*k*) definition,

- α = a b^n β = b w = b^n c
- γ = a b^{n+1} β = b y = b^{n+1} c

Although w[1:*k*] = y[1:*k*], we have α != γ → grammar is not LR(*k*).

The handle cannot be localized with only limited lookahead size k

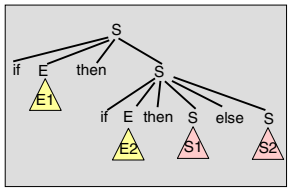
No ambiguous grammar is LR(*k*) for any fixed *k*

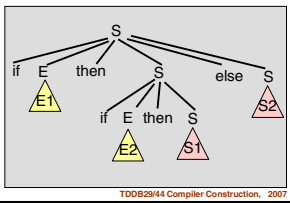
- S → if E then S
  | if E then S else S
  | *other statements*

...

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 if E then S 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

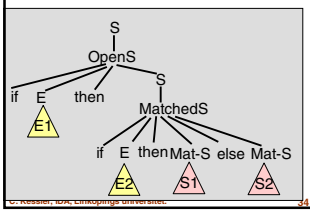
MatchedS → if E then MatchedS else MatchedS
| *other statements*

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 , alb] 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.

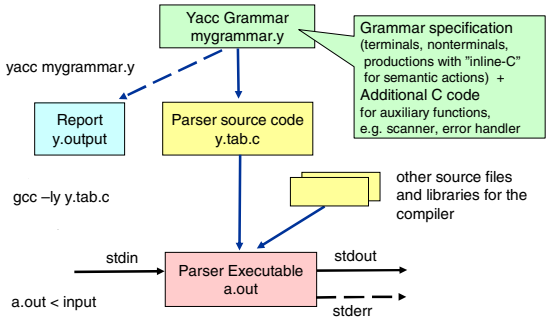
TDDB29 Compilers and Interpreters
TDDB44 Compiler Construction

Using a Parser Generator

Christoph Kessler, IDA,
Linköpings universitet, 2007.

Using a Parser Generator

- Example: UNIX Yacc, GNU Bison for LALR(1) grammars



Example Grammar for Yacc

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE char  /* char type for Yacc stack */
%}

%token ','
%token 'a'
%token 'b'

%%
list   : list ',' element     { printf("/%c", $3); }
       | element              { printf("%c", $1); }
       ;
element :  'a'                { $$ = 'A'; }
       |   'b'                { $$ = 'B'; }
       ;
%%
yylex() {  /* hand-crafted scanner for toy example */
  char c;
  while (1)
    switch (c = getchar()) {
      case ',':
      case 'a':
      case 'b': return c;
      case '\n':
      case EOF: return EOF;
      default: continue;  /* eat whitespace */
    }
}
```

Extra data field for each stack entry.
Can be used to store values (e.g. in an interpreter, as here) or pointers to tree nodes to construct an explicit tree representation (see Slides 114+115).

Semantic actions:
C code to be "pasted into the parser" and executed when reducing for a production.
Can access the extra data field of the production's stacked grammar symbols by $$, $1, $2, ...

Example (cont.) Yacc Report

```
state 0
      $accept : _list $end

      a  shift 3
      b  shift 4
      .  error

      list  goto 1
      element  goto 2

state 1
      $accept :  list_$end
      list :  list_, element

      $end  accept
      ,  shift 5
      .  error

state 2
      list :  element_   (2)

      .  reduce 2

state 3
      element :  a_   (3)

      .  reduce 3

state 4
      element :  b_   (4)

      .  reduce 4
```

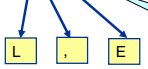
```
state 5
      list :  list ,_element

      a  shift 3
      b  shift 4
      .  error

      element  goto 6

state 6
      list :  list , element_   (1)

      .  reduce 1


5/127 terminals, 2/600 nonterminals
5/300 grammar rules, 7/1000 states
0 shift/reduce, 0 reduce/reduce conflicts reported
5/601 working sets used
memory: states,etc. 18/2000, parser 2/4000
3/3001 distinct lookahead sets
0 extra closures
5 shift entries, 1 exceptions
3 goto entries
0 entries saved by goto default
Optimizer space used: input 16/2000, output 10/4000
10 table entries, 4 zero
maximum spread: 98, maximum offset: 97
```

TDDB29 Compilers and Interpreters
TDDB44 Compiler Construction

Parse Tree Construction

in a LR-Parser using the Semantic Stack

Step	Stack	Input	Table entries
...	--I0 L1 , 2 E 3	...	...



Semantic stack now stores pointers to parse tree nodes

Christoph Kessler, IDA, Linköpings universitet, 2007.

Parse Tree Construction
Parsing input string a,b

0. S' -> L I--
1. L -> L , E
2. I E
3. E -> a
4. I b

Step	Stack	Input	Table entries
1	--I 0	**a , b** I--	ACTION[0, **a**] = S4
2	--I 0a4	**, b** I--	ACTION[4, ,] = R3 (E -> a)

Shift: Create a one-node tree containing the shifted symbol.

ACTION table:

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

GOTO table:

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

Example: Tables (given); Parsing input string a,b

0. S' -> L I--
1. L -> L , E
2. I E
3. E -> a
4. I b

Step	Stack	Input	Table entries
1	--I 0	**a , b** I--	ACTION[0, **a**] = S4
2	--I 0a4	**, b** I--	ACTION[4, ,] = R3 (E -> a)
	--I 0E	**, b** I--	GOTO[0, E] = 6

Reduce [X → α .]: Create a new tree node for X whose children are those former root nodes pointed to from the handle elements' semantic stack fields (in Yacc: $1, $2, ...)

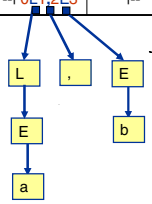
ACTION table:

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

Example: Tables (given); Parsing input string a,b

0. S' -> L I--
1. L -> L , E
2. I E
3. E -> a

Step	Stack	Input	Table entries
4	--I 0L1	**, b** I--	ACTION[1, ,] = S2
5	--I 0L1,2	**b** I--	ACTION[2, **b**] = S5
6	--I 0L1,2b5	I--	ACTION[5, I--] = R4 (E -> b)
	--I 0L1,2E	I--	GOTO[2, E] = 3
7	--I 0L1,2E3	I--	ACTION[3, I--] = R1 (L ->L,E)



During parsing: Forest of subtrees, roots pointed from the semantic stack

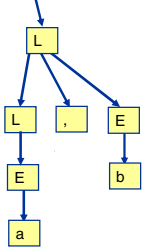
ACTION table:

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

Example: Tables (given); Parsing input string a,b

0. S' -> L I--
1. L -> L , E
2. I E
3. E -> a

Step	Stack	Input	Table entries
7	--I 0L1,2E3	I--	ACTION[3, I--] = R1 (L ->L,E)
	--I 0L	I--	GOTO[0, L] = 1



At accept [S' → S.I--]: Emit the parse tree computed for S.

ACTION table:

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