TDDD55 Compilers and Interpreters

TDDB44 Compiler Construction

LR Parsing, Part 1

LR parsing concept

Using a parser generator

Parse Tree Generation

Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2011.

What is LR-parsing?

- L – Left-to-right scanning
- R – Rigthmost derivation in reverse, i.e., bottom-up parsing
  - 12 $<=_{rm}$ <digit> 2 $<=_{rm}$ <no> 2 $<=_{rm}$ <no> <digit> $<=_{rm}$ <no> $<=_{rm}$ <number>
- LR(1) – LR parsing with 1 token lookahead
- LR(k) – LR parsing with k tokens lookahead
- LR-parsing is the most general nonbacktracking shift-reduce parsing method known
- An LR-parser detects a syntactic error as soon as possible

LR-Grammar Definition

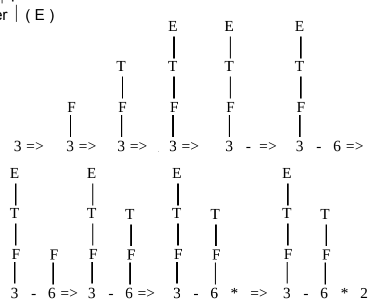
A grammar for which a unique LR-table can be constructed is called an *LR grammar (LR(0), SLR(1),* LALR(1), LR(1), ...).

- No ambiguous grammars are LR grammars.
- There are unambiguous grammars which are not LR grammars.
- The state at the top of the stack contains all the information we need.

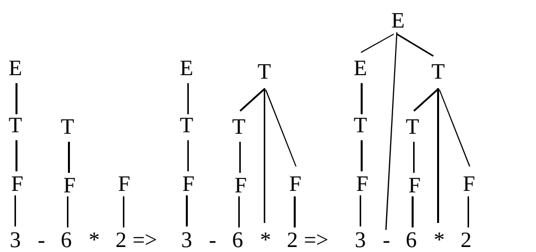
LR-Parsing is Bottom-up Parsing (from Lecture 5)

- Example: **Bottom-up** parsing with input: 3 - 6 * 2

E → E - T | T
T → T * F | F
F → Integer | (E)



Bottom-up Parsing cont.

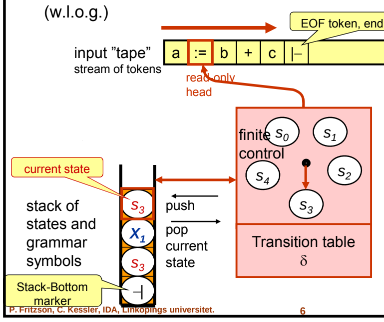


110a

Pushdown Automaton for LR-Parsing

Finite-state pushdown automaton

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



2 sorts of transitions:

Actions
(shift, reduce, accept, error)

Goto's (ε-transitions to find new state after reductions)

Configurations of the LR-Parser

- Configuration = (stack contents, remaining input)

  $= (s_0\ X_1\ s_1\ \ldots\ X_{m-1}\ s_{m-1}\ X_m\ s_m\quad a_i\ a_{i+1}\ \ldots\ a_n\ )$

  current state (TopOfStack)

- Shift: read current input symbol a_i and push it with new state s

  $\models\ (s_0\ X_1\ s_1\ \ldots\ X_{m-1}\ s_{m-1}\ X_m\ s_m\quad a_i\ s\ a_{i+1}\ \ldots\ a_n\ )$

- Reduce: read ε, pop $2r$ stack symbols for handle $X_{m-r+1} \ldots X_m$, push LHS nonterminal + new state (see below)

- **Invariants**:
  - Nonterminals on stack + remaining input $(X_1\ \ldots\ X_{m-1}\ X_m\ a_i\ a_{i+1}\ \ldots\ a_n\ )$ 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: An 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	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	--| 0a4	, **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	--| 0a4	, **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)

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	$\boxed{\text{R2}}$	*	*

GOTO table:

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

... and so on ...

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**,**2**b**5	|--	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	--| 0L1	|--	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	*	*

Small Exercise on LR-parsing

Rightmost Derivation, Handle (from Lecture 3)

- **Reverse rightmost derivation**
  - 12 $<=_{rm}$ <digit> 2 $<=_{rm}$ <no> 2 $<=_{rm}$ <no> <digit> $<=_{rm}$ <no> $<=_{rm}$ <number>
- **Handles**
  Consist of two parts:
  - 1. A production $A \rightarrow \beta$
  - 2. A position
  - If $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha \beta w$, the production $A \rightarrow \beta$ together with the position after α is a **handle** of $\alpha \beta w$.

1. <number> → <no>
2. <no> → <no> <digit>
3. | <digit>
4. digit> → 0|1|2|3|4|5|6|7|8|9

- Example: The handle of <u><no> 2</u> is the production <digit> → 2 and the position after <no> because:
  - <number> $\Rightarrow_{rm}$ <no> $\Rightarrow_{rm}$ <no> <digit> $\Rightarrow_{rm}$ <no> 2 $\Rightarrow_{rm}$ <digit> 2 $\Rightarrow_{rm}$ 12
- Informally: a handle is what we *reduce* to what and where to get the previous sentential form in a rightmost derivation.

Handle, Viable Prefix

- Consider a rightmost derivation $S =>^*_{rm} \beta X u =>_{rm} \beta \alpha u$ for a context-free grammar *G*.
- α is called a **handle** of the right sentential form $\beta\alpha u$, associated with the rule $X =>_{rm} \alpha$
- 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: Handle: c ε, a, ac
- For aSb: Handle: aSb ε, a, aS, aSb
- For aaSbb: Handle: aSb ε, a, aa, aaS, aaSb
- ...

Recognizing Handles

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

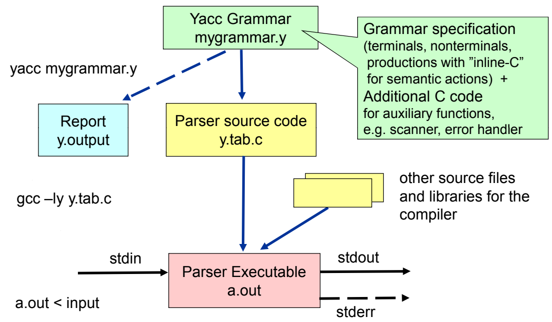
- A.k.a. the "*characteristic finite automaton*" for a grammar G
- States: LR(0) items (= context-free items) of extended grammar
- Input stream: The grammar symbols on the stack
- Start state: [S' → –|.S] Final state: [S' → –|S.]
- Transitions:
  - "move dot across symbol" if symbol found next on stack:
    $A \rightarrow \alpha.B\gamma$ to $A \rightarrow \alpha B.\gamma$
    $A \rightarrow \alpha.b\gamma$ to $A \rightarrow \alpha b.\gamma$
  - ε-transitions to LR(0)-items for nonterminal productions from items where the dot precedes that nonterminal:
    $A \rightarrow \alpha.B\gamma$ to $B \rightarrow .\beta$
- (Example and construction of DFA later, in Lecture 7)

Using a Parser Generator

Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2011.

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, ...

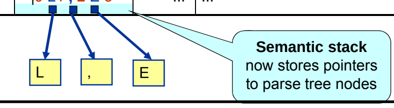
Example (cont.) Yacc Report

```
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
```


Parse Tree Construction

in an LR-Parser using the Semantic Stack

Step	Stack	Input	Table entries
...	--|0 L1 , 2 E 3	...	...



Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2011.

Construction of parse tree (Bottom-up)

- *Shift operations:*

  Create a *one-node tree* containing the shifted symbol.

- *Reduce operations:*

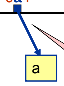
  When reducing a handle β to A (as in A→β), create a *new node A* whose "children" are those nodes that were created in the handle.

- During the analysis we have a forest of sub-trees. Each entry in the stack points to its corresponding sub-tree. When we *accept, the whole parse tree is completed.*

Parse Tree Construction 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	--| 0a4	, b |--	ACTION[4, ,] = R3 (E -> a)



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

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	*	*

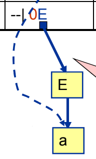
GOTO table:

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

Parse Tree Construction 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	--| 0a4	, b |--	ACTION[4, ,] = R3 (E -> a)
	--| 0E	, b |--	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, ...)

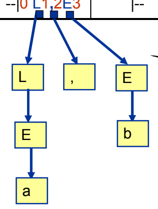
ACTION table:

state	|--	,	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	*	*

Parse Tree Construction Parsing input string a,b

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

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



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

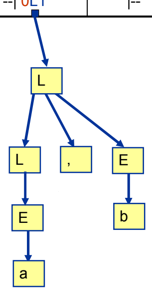
ACTION table:

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	*	*

Parse Tree Construction Parsing input string a,b

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

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



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

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	*	*

Small Exercise on Parse Tree Construction