CS412/413

Introduction to Compilers
Radu Rugina

Lecture 8: Bottom-up Parsing
11 Feb 04

Shift-reduce Parsing

- Parsing actions: is a sequence of shift and reduce operations
- Parser state: a stack of terminals and non-terminals (grows to the right)
- Current derivation step = always stack+input

Derivation step	stack	unconsumed input
(1+2+(3+4))+5 ←		(1+2+(3+4))+5
(E+2+(3+4))+5 ←	(E	+2+(3+4))+5
(S+2+(3+4))+5 ←	(S	+2+(3+4))+5
(S+E+(3+4))+5 ←	(S+E	+(3+4))+5

Shift-reduce Actions

- Parsing is a sequence of shifts and reduces
- Shift : move look-ahead token to stack

stack	input	action
(	1+2+(3+4))+5	shift 1
(1	+2+(3+4))+5	

- Reduce : Replace symbols γ from top of stack with non-terminal symbol X, corresponding to production $X \rightarrow \gamma$ (pop γ, push X)

stack	input	action
(<u>S+E</u>	+(3+4))+5	reduce S→ S+E
(S	+(3+4))+5	

Shift-reduce Parsing

S → S + E | E
E → num | (S)

derivation	stack	input stream	action
(1+2+(3+4))+5 ←		(1+2+(3+4))+5	shift
(1+2+(3+4))+5 ←	(	1+2+(3+4))+5	shift
(1+2+(3+4))+5 ←	(1	+2+(3+4))+5	reduce E→num
(E+2+(3+4))+5 ←	(E	+2+(3+4))+5	reduce S → E
(S+2+(3+4))+5 ←	(S	+2+(3+4))+5	shift
(S+2+(3+4))+5 ←	(S+	2+(3+4))+5	shift
(S+2+(3+4))+5 ←	(S+2	+(3+4))+5	reduce E→num
(S+E+(3+4))+5 ←	(S+E	+(3+4))+5	reduce S→ S+E
(S+(3+4))+5 ←	(S	+(3+4))+5	shift
(S+(3+4))+5 ←	(S+	(3+4))+5	shift
(S+(3+4))+5 ←	(S+(	3+4))+5	shift
(S+(3+4))+5 ←	(S+(3	+4))+5	reduce E→num

LR Parsing Engine

- Basic mechanism:
  - Use a set of parser states
  - Use a stack with alternating symbols and states
    - E.g: $_1$ ($_6$ S $_{10}$ + $_5$
  - Use a parsing table to:
    - Determine what action to apply (shift/reduce)
    - Determine the next state
- The parser actions can be precisely determined from the table

---

The LR Parsing Table

	Terminals	Non-terminals
State	Next action and next state	Next state
	Action table	Goto table

- Algorithm: look at entry for current state S and input terminal C

  If Table[S,C] = s(S') then shift:
  push(C), push(S')

  If Table[S,C] = X→**α** then reduce:
  pop(2*|**α**|), S'=top(), push(X), push(Table[S',X])

LR Parsing Table Example

	(	)	id	,	$	S	L
1	s3		s2			g4	
2	S→id	S→id	S→id	S→id	S→id		
3	s3		s2			g7	g5
4					accept		
5		s6		s8			
6	S→(L)	S→(L)	S→(L)	S→(L)	S→(L)		
7	L→S	L→S	L→S	L→S	L→S		
8	s3		s2			g9	
9	L→L,S	L→L,S	L→L,S	L→L,S	L→L,S		

LR(k) Grammars

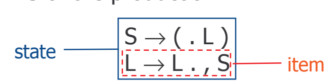
- LR(k) = Left-to-right scanning, Right-most derivation, k look-ahead characters
- Main cases: LR(0), LR(1), and some variations (SLR and LALR(1))
- Parsers for LR(0) Grammars:
  - Determine the actions without any lookahead symbol
  - will help us understand shift-reduce parsing

Building LR(0) Parsing Tables

- To build the parsing table:
  - Define states of the parser
  - Build a DFA to describe the transitions between states
  - Use the DFA to build the parsing table
- Each LR(0) state is a set of LR(0) items:
  - An LR(0) item: $X \to \alpha . \beta$, where $X \to \alpha \beta$ is a production in the grammar
  - The LR(0) items keep track of the progress on all of the possible upcoming productions
  - The item $X \to \alpha . \beta$ abstracts the fact that the parser already matched the string α at the top of the stack

Example LR(0) State

- An LR(0) item is a production from the language with a separator "." somewhere in the RHS of the production



- Sub-string before "." is already on stack (beginnings of possible γ's to be reduced)
- Sub-string after "." : what we might see next

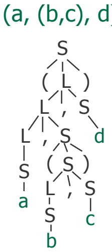
LR(0) Grammar

- Nested lists:

  S → (L) | id
  L → S | L , S

- Examples
  - (a, b, c)
  - ((a,b), (c,d), (e,f))
  - (a, (b,c,d), ((f,g)))

Parse tree for (a, (b,c), d)



Start State & Closure

- Start state
  - Augment grammar with production S ' → S $
  - Start state of DFA has empty stack: S ' → . S $
- Closure of a parser state:
  - Start with Closure(S) = S
  - Then for each item in S:

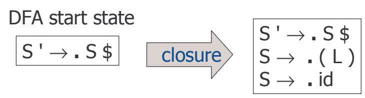
    $X \to \alpha . Y \beta$

    add the items for all the productions $Y \to \gamma$ to the closure of S:

    $Y \to . \gamma$

Closure Example

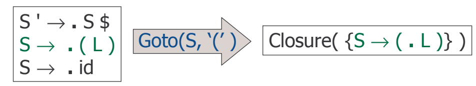
S → (L) | id
L → S | L, S



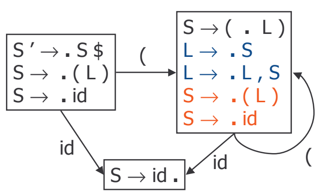
– set of possible productions to be reduced next
– Added items have the "." located at the beginning: no symbols for these items on the stack yet

The Goto Operation

- Goto operation = describes transitions between parser states, which are sets of items
- Algorithm: for a state S and a symbol Y
  - S' = {X → α Y . β | X → α . Y β ∈ S}
  - Goto(S, Y) = Closure(S')



Goto: Terminal Symbols

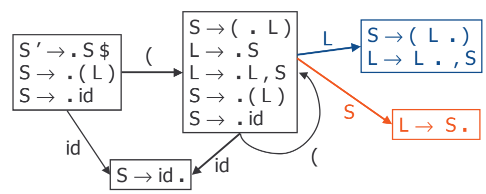


Grammar:

S → (L) | id
L → S | L , S

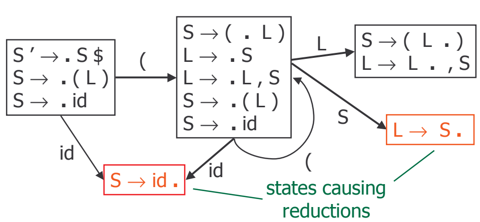
In new state, include all items that have appropriate input symbol just after dot, advance dot in those items, and take closure.

Goto: Non-terminal Symbols



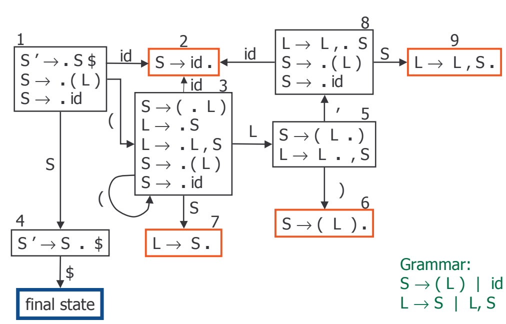
(same algorithm for transitions on non-terminals)

Applying Reduce Actions



- Pop RHS off stack, replace with LHS X (X→γ), then rerun DFA (e.g. (x))

Full DFA



Parsing Example: ((a),b)

S → (L) | id
L → S | L, S

derivation	stack	input	action
((a),b) ←	${}_1$	((a),b)	shift, goto 3
((a),b) ←	${}_1 (_3$	(a),b)	shift, goto 3
((a),b) ←	${}_1 (_3 (_3$	a),b)	shift, goto 2
((a),b) ←	${}_1 (_3 (_3 a_2$	),b)	reduce S→id
((S),b) ←	${}_1 (_3 (_3 S_7$	),b)	reduce L→S
((L),b) ←	${}_1 (_3 (_3 L_5$	),b)	shift, goto 6
((L),b) ←	${}_1 (_3 (_3 L_5)_6$	,b)	reduce S→(L)
(S,b) ←	${}_1 (_3 S_7$	,b)	reduce L→S
(L,b) ←	${}_1 (_3 L_5$	,b)	shift, goto 8
(L,b) ←	${}_1 (_3 L_5 ,_8$	b)	shift, goto 9
(L,b) ←	${}_1 (_3 L_5 ,_8 b_2$	)	reduce S→id
(L,S) ←	${}_1 (_3 L_5 ,_8 S_9$	)	reduce L→L , S
(L) ←	${}_1 (_3 L_5$	)	shift, goto 6
(L) ←	${}_1 (_3 L_5)_6$		reduce S→(L)
S	${}_1 S_4$	$	done

CS 412/413 Spring 2004 Introduction to Compilers 19

Reductions

- On reducing X→γ with stack αγ:
  - pop γ off stack, revealing prefix α and state
  - take single step in DFA from top state
  - push X onto stack with new DFA state
- Example:

((a),b) ←	${}_1 (_3 (_3$	a),b)	shift, goto 2
((a),b) ←	${}_1 (_3 (_3 a_2$	),b)	reduce S→id
((S),b) ←	${}_1 (_3 (_3 S_7$	),b)	reduce L→S

CS 412/413 Spring 2004 Introduction to Compilers 20

Build the Parsing Table

- States in the table = states in the DFA
- For a transition S → S' on terminal C:
  Shift(S') ⊆ Table[S,C]
- For a transition S → S' on non-terminal N:
  Goto(S') ⊆ Table[S,N]
- If S is a reduction state X → γ then:
  Reduce(X → γ) ⊆ Table[S,*]

CS 412/413 Spring 2004 Introduction to Compilers 21

Computed LR Parsing Table

	(	)	id	,	$	S	L
1	s3		s2			g4	
2	S→id	S→id	S→id	S→id	S→id		
3	s3		s2			g7	g5
4					accept		
5		s6		s8			
6	S→(L)	S→(L)	S→(L)	S→(L)	S→(L)		
7	L→S	L→S	L→S	L→S	L→S		
8	s3		s2			g9	
9	L→L,S	L→L,S	L→L,S	L→L,S	L→L,S		

CS 412/413 Spring 2004 Introduction to Compilers 22

LR(0) Summary

- LR(0) parsing recipe:
  Start with an LR(0) grammar
  Compute LR(0) states and build DFA:
  - Use the closure operation to compute states
  - Use the goto operation to compute transitions between states

  Build the LR(0) parsing table from the DFA
- This process can be automated, i.e. we can build parser generator tools

CS 412/413 Spring 2004 Introduction to Compilers 23

LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a single reduce action -- in those states, always reduce ignoring lookahead
- With more complex grammar, construction gives states with shift/reduce or reduce/reduce conflicts
- Need to use look-ahead to choose

ok	shift /reduce	reduce / reduce
L → L , S .	L → L , S . S → S . , L	L → S , L . L → S .

CS 412/413 Spring 2004 Introduction to Compilers 24

LR(0) Parsing Table

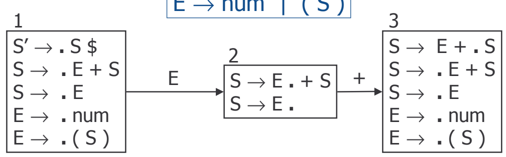
	(	)	id	,	$	S	L
1	s3		s2			g4	
2	S→id	S→id	S→id	S→id	S→id		
3	s3		s2			g7	g5
4					accept		
5		s6		s8			
6	S→(L)	S→(L)	S→(L)	S→(L)	S→(L)		
7	L→S	L→S	L→S	L→S	L→S		
8	s3		s2			g9	
9	L→L,S	L→L,S	L→L,S	L→L,S	L→L,S		

A Non-LR(0) Grammar

- Grammar for addition of numbers:
  S → S + E | E
  E → num | (S)
- Left-associative is LR(0)
- Right-associative version is not LR(0)
  S → E + S | E
  E → num | (S)

LR(0) Parsing Table

S → E + S | E
E → num | (S)



What to do in state 2?

	+	$	E
1			2
2	s3/S→E	S→E	

Next Time

- Learn about other kinds of LR parsing:
  - SLR = improved LR(0)
  - LR(1) = 1 character lookahead
  - LALR(1) = Look-Ahead LR(1)
- Basic ideas are the same as for LR(0)
  - Parser states with LR items
  - DFA with transitions between parser states
  - Parser table with shift/reduce/goto actions