Outline

- Read Chapter 3 (especially 3.4)

Shift-Reduce Parsing

- *Shift-reduce parsing*
  - A.K.A.: *bottom-up parsing*
  - A.K.A.: *LR(k)* - Left-to-right parse, Rightmost derivation, k-token lookahead
- Parser has *stack*.
- Based on stack contents and next input token, one of two action performed:
  1. *Shift* - push next input token onto top of stack.
  2. *Reduce* - choose production ($X \to ABC$); pop off RHS (C, B, A); push LHS (X).
- Stack is initially empty.
- Parser points to beginning of input stream.
- If $ is shifted, then input stream has been parsed successfully.

Shift-Reduce Parsing

How does parser know when to shift or reduce?

- DFA: applied to stack contents, not input stream
- Each state corresponds to contents of stack at some point in time.
- Edges labelled with terms/non-terms that can appear on stack.

Example

Grammar:

1 A $\rightarrow$ S EOF
2 S $\rightarrow$ *(* L *)*
3 S $\rightarrow$ *id* = *num*
4 L $\rightarrow$ L*;* S
5 L $\rightarrow$ S

Input:

$$(a = 4; b = 5) \rightarrow (ID_a = NUM_4; ID_b = NUM_5)$$

```
         input: ( ID = NUM ; ID = NUM )
                |
0        stack:
        action: shift
```

Example

```
      input:  ( ID = NUM ; ID = NUM )
                |
 1    stack:  (
     action:  shift
-----------------------------------------
      input:  ( ID = NUM ; ID = NUM )
                   |
 2    stack:  ( ID
     action:  shift
-----------------------------------------
      input:  ( ID = NUM ; ID = NUM )
                     |
 3    stack:  ( ID =
     action:  shift
-----------------------------------------
      input:  ( ID = NUM ; ID = NUM )
                         |
 4    stack:  ( ID = NUM
     action:  reduce 3
```

Example

5

```
   input:  ( ID = NUM ; ID = NUM )
                      |
   stack:  ( S
  action:  reduce 5
```

6

```
   input:  ( ID = NUM ; ID = NUM )
                      |
   stack:  ( L
  action:  shift
```

7

```
   input:  ( ID = NUM ; ID = NUM )
                        |
   stack:  ( L ;
  action:  shift
```

8

```
   input:  ( ID = NUM ; ID = NUM )
                           |
   stack:  ( L; ID
  action:  shift
```

Example

9

```
 input: ( ID = NUM ; ID = NUM )
                       |
 stack: ( L ; ID =
action: shift
```

10

```
 input: ( ID = NUM ; ID = NUM )
                               |
 stack: ( L ; ID = NUM
action: reduce 3
```

11

```
 input: ( ID = NUM ; ID = NUM )
                               |
 stack: ( L ; S
action: reduce 4
```

12

```
 input: ( ID = NUM ; ID = NUM )
                               |
 stack: ( L
action: shift
```

Example

13

```
 input: ( ID = NUM ; ID = NUM )
                               |
 stack: ( L )
action: reduce 2
```

14

```
 input: ( ID = NUM ; ID = NUM )
                               |
 stack: S
action: ACCEPT
```

Shift-Reduce Parsing/Bottom-Up Parsing/LR(1)

- Shift-reduce parsing can parse more grammars than top-down predictive parsing.
- *Shift-reduce parsing* has stack and input.
- Based on stack contents and next input token, one of two action performed:
  1. *Shift* - push next input token onto top of stack.
  2. *Reduce* - choose production (X $\rightarrow$ ABC); pop off RHS (C, B, A); push LHS (X).
- If $ is shifted, then input stream has been parsed successfully.

LR(k) Parsing

Can generalize to case where parser makes decision based on stack contents and next k tokens. LR(k):

- Left-to-right parse
- right-most derivation
- k-symbol lookahead

LR(k) parsing, $k > 1$, rarely used in compilation:

- DFA too large: need transition for every sequence of k terminals.
- Most programming languages can be described by LR(1) grammars.

Shift-Reduce Parsing DFA

Parser uses DFA to make shift/reduce decisions:

- Each state corresponds to contents of stack at some point in time.
- Edges labeled with terminals/non-terminals.

Rather than scanning entire stack to determine current DFA state, parser can remember state reached for each stack element.

- Transition table for LR(1) or LR(0) DFA:

	Terminals $(T_1, T_2, \ldots, T_n)$	Non-Terminals $(N_1, N_2, \ldots, N_n)$
1	*actions*	*actions*
2	sn $\rightarrow$ shift n	gz $\rightarrow$ goto z
3	rk $\rightarrow$ reduce k	
$\vdots$	a $\rightarrow$ accept	
n	$\rightarrow$ error	

Parsing Algorithm

Look up DFA state on top of stack, next terminal in input:

- shift(n):
  - Advance input by one.
  - Push input token on stack with n (the new state).
- reduce(k):
  - Pop stack as many times as number of symbols on RHS of rule k.
  - Let X be LHS of rule k
  - In state now on top of stack, look up X to get goto(z)
  - Push X on stack with z (the new state).
- accept $\rightarrow$ stop, report success.
- error $\rightarrow$ stop, report syntax error.

To understand LR(k) parsing, first focus on LR(0) parser construction using an example.

LR(0) Parsing

$1\ S' \rightarrow S\ \$$
$2\ S \rightarrow (\ L\ )$
$3\ S \rightarrow\ \mathrm{x}$
$4\ L \rightarrow S$
$5\ L \rightarrow L, S$

Initially, stack empty, input contains 'S' string followed by a '\$':

1 $\boxed{\begin{array}{l} S' \rightarrow .S\$ \\ S \rightarrow .(\ L\ ) \\ S \rightarrow .\mathrm{x} \end{array}}$

- Combination of production and '.' called LR(0) *item*.
- '.' specifies parser position.
- Three items represent *closure* of: $S' \rightarrow .S\ \$$
- Closure adds more items to a set when dot exists to left of a non-terminal.

LR(0) Parsing

$1\ S' \rightarrow S\ \$$

$2\ S \rightarrow (\ L\ )$

$3\ S \rightarrow\ \mathrm{x}$

$4\ L \rightarrow S$

$5\ L \rightarrow L, S$

LR(0) states:

DFA Table Entry Computation

To compute transition table from state diagram perform the following:

- ${}^{i}\boxed{S' \to S.\$} \Rightarrow$ table[i, \$] = a.
- ${}^{i}\boxed{} \xrightarrow{T} {}^{j}\boxed{}$, Terminal $T \Rightarrow$ table[i, T] = sj.
- ${}^{i}\boxed{} \xrightarrow{N} {}^{j}\boxed{}$, Non-Terminal $N \Rightarrow$ table[i, N] = gj.
- ${}^{i}\boxed{A \to \gamma.} \Rightarrow$ table[i, T] = rk, for all terminals T.

Transition Table

	(	)	x	,	$	S'	S	L
1								
2								
3								
4								
5								
6								
7								
8								
9								

No duplicate entries $\Rightarrow$ grammar is LR(0)

Using Transition Table

$1\; S' \rightarrow S\,\$$ $\qquad$ $3\; S \rightarrow \text{x}$ $\qquad$ $5\; L \rightarrow L, S$

$2\; S \rightarrow (L)$ $\qquad$ $4\; L \rightarrow S$

	(	)	x	,	$	S'	S	L
1	s3		s2				g9	
2	r3	r3	r3	r3	r3			
3	s3		s2				g5	g4
4		s6		s7				
5	r4	r4	r4	r4	r4			
6	r2	r2	r2	r2	r2			
7	s3		s2				g8	
8	r5	r5	r5	r5	r5			
9					a			

STACK	INPUT	ACTION
1	(x , x) $	shift 3
1 (3	x , x) $	shift 2
1 (3 x2	, x) $	reduce 3
1 (3 S5	, x) $	reduce 4
1 (3 L4	, x) $	shift 7
1 (3 L4 ,7	x) $	shift 2
1 (3 L4 ,7 x2	) $	reduce 4
1 (3 L4 ,7 S8	) $	reduce 5
1 (3 L4	) $	shift 6
1 (3 L4)6	$	reduce 2
1 S9	$	accept

Another Example

$1\ S' \to E\ \$$

$2\ E \to T + E$

$3\ E \to T$

$4\ T \to \text{x}$

LR(0) states:

Another Example - SLR

Transition Table:

	+	x	\$	S'	E	T
1		s3			g2	g4
2			a			
3	r4	r4	r4			
4	s5/r3	r3	r3			
5		s3			g6	g4
6	r2	r2	r2			

Duplicate entries $\Rightarrow$ grammar is NOT LR(0)
Can make grammar bottom-up parsable using more powerful parsing techniques: **SLR** (Simple LR)

- Use same LR(0) states.
- ${}^{i}\boxed{A \to \gamma.}$ $\Rightarrow$ table[i,T] = reduce(k), for all terminals $T \in$ follow(A)

Another Example - SLR

Transition Table:

	+	x	\$	S'	E	T
1		s3			g2	g4
2			a			
3	r4	r4	r4			
4	s5/r3	r3	r3			
5		s3			g6	g4
6	r2	r2	r2			

Follow Set Computation:

	nullable	first	follow
S'	no	x	
E	no	x	\$
T	no	x	+,\$

$1\ S' \rightarrow E\ \$$ $\qquad 3\ E \rightarrow T$

$2\ E \rightarrow T + E$ $\qquad 4\ T \rightarrow \text{ x}$

SLR Transition Table:

	+	x	\$	S'	E	T
1		s3			g2	g4
2			a			
3	r4		r4			
4	s5		r3			
5		s3			g6	g4
6			r2			

No duplicate entries $\Rightarrow$ grammar is SLR.

Yet Another Example

Sometimes grammar can't be parsed using SLR techniques.

$1\ S' \rightarrow S\ \$$	$3\ S \rightarrow E$	$5\ V \rightarrow$ x
$2\ S \rightarrow V$ = E	$4\ E \rightarrow V$	$6\ V \rightarrow *\ E$

This grammar is not SLR. Need more powerful parsing algorithm $\Rightarrow$ LR(1)

LR(1) Parsing

- LR(1) item consists of two components: ($A \rightarrow \alpha.\beta$, x)
  1. Production
  2. Lookahead symbol (x)
- α is on top of stack, head of input is string derivable from β x.

LR(0) closure computation

- Initial: $A \rightarrow \alpha.X$
- Add all items $X \rightarrow .\gamma$
- Repeat closure computation

LR(1) closure computation

- Initial: $A \rightarrow \alpha.X\beta,$ z
- Add all items ($X \rightarrow .\gamma$, ω) for each $\omega \in$ first(βz)
- Repeat closure computation

- shift, goto, accept table entries computed same way as LR(0)/SLR.
- reduce entries computed differently:

$$^{i}\boxed{A \rightarrow \gamma.\ ,\ \text{z}} \Rightarrow \text{table}[i,\text{z}] = \text{reduce}(k)$$

Yet Another Example - LR(1)

$1\ S' \rightarrow S\ \$$

$2\ S \rightarrow V = E$

$3\ S \rightarrow E$

$4\ E \rightarrow V$

$5\ V \rightarrow \mathrm{x}$

$6\ V \rightarrow *\ E$

LR(1) states:

Yet Another Example - LR(1)

	=	x	*	\$	S'	S	L	V
1		s11	s12			g2	g10	g3
2				a				
3	s4			r4				
4		s7	s8				g5	g6
5				r2				
6				r4				
7				r5				
8		s7	s8				g9	g6
9				r6				
10				r3				
11	r5			r5				
12		s11	s12				g13	g14
13	r6			r6				
14	r4			r4				

No duplicate entries $\Rightarrow$ grammar is LR(1)

LALR(1)

- Problem with LR(1) parsers: tables too large!
  - Can make smaller table by merging states whose items are identical except for look-ahead sets $\Rightarrow$ LALR(1) (Look-Ahead LR(1)).
  - LALR(1) transition table may contain shift-reduce/reduce-reduce conflicts where LR(1) table has none.

LALR(1)

Can make smaller table by merging states whose items are identical except for look-ahead sets ⇒ LALR(1) (Look-Ahead LR(1)).

	=	x	*	\$	S'	S	L	V
1		s11	s12			g2	g10	g3
2				a				
3	s4			r4				
4		s7	s8				g5	g6
5				r2				
6/14	r4			r4				
7/11	r5			r5				
8/12		s7/11	s8/12				g9/13	g6/14
9/13	r6			r6				
10				r3				

No conflicts ⇒ grammar is LALR(1).

Grammars

ML-YACC uses LALR(1) parsing because reasonable programming languages can be specified by an LALR(1) grammar.