Outline

Last Week:

- LR(0) Parsing:
  - Table construction
  - Using the tables to parse

Today:

- Beyond LR(0):
  - SLR
  - LR(1)
  - LALR(1)
- Error Recover Techniques

Reminder: Shift-Reduce Parsing

- Shift-reduce parsing can parse more grammars than 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 $\longrightarrow$ ABC); pop off RHS (C, B, A); push LHS (X).
- If \$ is shifted, then input stream has been parsed successfully.

Reminder: Using Transition Table

1 $S' \rightarrow S\ \$$ 3 $S \rightarrow$ x 5 $L \rightarrow L, S$
2 $S \rightarrow (L)$ 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 $S' \rightarrow E\ \$$ 3 $E \rightarrow T$ 4 $T \rightarrow$ x
2 $E \rightarrow T + E$

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.
- ${}^{i}\boxed{A \rightarrow \gamma.}$ $\Rightarrow$ 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	+,\$

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			

1 $S' \rightarrow E\ \$$ 3 $E \rightarrow T$
2 $E \rightarrow T + E$ 4 $T \rightarrow$ x

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 $S' \rightarrow S\,\$$	3 $S \rightarrow E$	5 $V \rightarrow$ x
2 $S \rightarrow V = E$	4 $E \rightarrow V$	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.

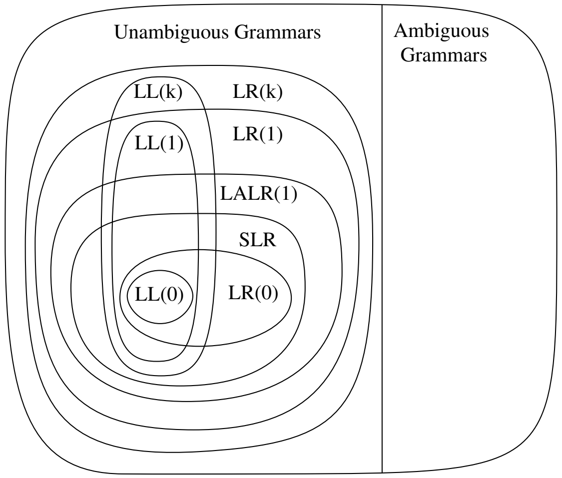
LALR(1)

Can make smaller table by merging states whose items are identical except for look-ahead sets $\Rightarrow$ 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 $\Rightarrow$ grammar is LALR(1).

Grammars



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

Error Recovery

Syntax Errors:

- A *Syntax Error* occurs when stream of tokens is an invalid string.
- In LL(k) or LR(k) parsing tables, blank entries refer to syntax errors.

How should syntax errors be handled?

1. Report error, terminate compilation $\Rightarrow$ not user friendly
2. Report error, *recover* from error, search for more errors $\Rightarrow$ better

Error Recovery

***Error Recovery*: process of adjusting input stream so that parsing may resume after syntax error reported.**

- Deletion of token types from input stream
- Insertion of token types
- Substitution of token types

Two classes of recovery:

1. *Local Recovery*: adjust input at point where error was detected.
2. *Global Recovery*: adjust input *before* point where error was detected.

These may be applied to both LL and LR parsing techniques.

LL Local Error Recovery

Consider LL(1) parsing context:

$$Z \rightarrow XYZ \qquad Y \rightarrow \text{c} \qquad X \rightarrow \text{a}$$
$$Z \rightarrow \text{d} \qquad Y \rightarrow \epsilon \qquad X \rightarrow \text{b}\,Y\,\text{e}$$

	nullable	first	follow
Z	no	a,b,d	
Y	yes	c	a,b,d,e
X	no	a,b	a,b,c,d

	a	b	c	d	e
Z	$Z \rightarrow XYZ$	$Z \rightarrow XYZ$		$Z \rightarrow d$	
Y	$Y \rightarrow \epsilon$	$Y \rightarrow \epsilon$	$Y \rightarrow c$	$Y \rightarrow \epsilon$	$Y \rightarrow \epsilon$
X	$X \rightarrow a$	$X \rightarrow bYe$			

LL Local Error Recovery

Local Recovery Technique: in function A(), delete token types from input stream until token type in follow(A) found ⇒ *synchronizing* token types.

```
datatype token = a | b | c | d | e;
val tok = ref(getToken());
fun advance() = tok := getToken();
fun eat(t) = if(!tok = t) then advance() else error();
...
and X() = case !tok of
       a => (eat(a))
     | b => (eat(b); Y(); eat(e))
     | c => (print "error!"; skipTo[a,b,c,d])
     | d => (print "error!"; skipTo[a,b,c,d])
     | e => (print "error!"; skipTo[a,b,c,d])

and skipTo(synchTokens) =
    if member(!tok, synchTokens) then ()
    else (eat(!tok); skipTo(synchTokens))
```


LR Local Error Recovery

Consider:

1 $E \rightarrow$ ID	3 $E \rightarrow (E)$	5 $ES \rightarrow ES \, ; E$
2 $E \rightarrow E + E$	4 $ES \rightarrow E$	

- Match a sequence of erroneous input tokens using the *error* token (a terminal).

  6 $E \rightarrow$ (error) 7 $ES \rightarrow$ error ; E

- In general, follow *error* with synchronizing lookahead token.
  1. Pop stack (if necessary) until a state is reached in which the action for the *error* token is *shift*.
  2. Shift the *error* token.
  3. Discard input symbols (if necessary) until a state is reached that has a non-error action in the current state.
  4. Resume normal parsing.

Global Error Recovery

Consider LR(1) parsing:

```
let type a := intArray[10] of 0 in ... end
```

Local Recovery Techniques would:

1. report syntax error at ':='
2. substitute '=' for ':='
3. report syntax error at '['
4. delete token types from input stream, synchronizing on 'in'

Global Recovery Techniques would substitute 'var' for 'type':

- Actual syntax error occurs *before* point where error was detected.
- ML-Yacc uses global error recovery technique ⇒ *Burke-Fisher*
- Other Yacc versions employ local recovery techniques.

Burke-Fisher

Suppose parser gets stuck at n^{th} token in input stream.

- Burke-Fisher repairer tries every *single-token-type* insertion, deletion, and substitution at all points between $(n-k)^{th}$ and n^{th} token.

  _____ _____ - - - - - _____ _____
  n-k   n-k+1   n-1   n

- Best repair: one that allows parser to parse furthest past n^{th} token.
- If languages has N token types, then:

  total # of repairs = deletions + insertions + substitutions

  total # of repairs = $(k) + (k+1)\,N + (k)\,(N-1)$

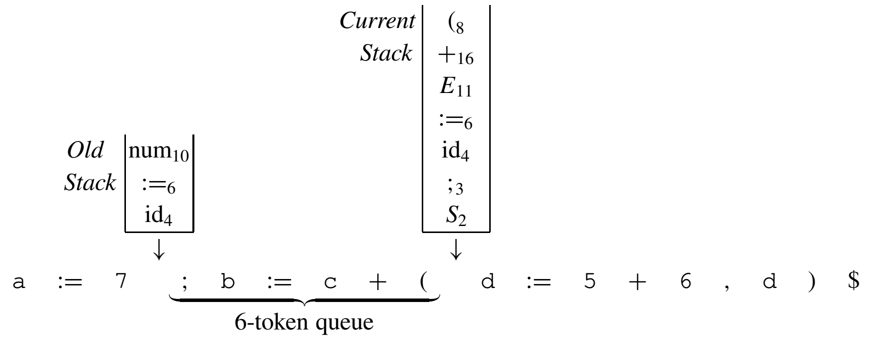
Burke-Fisher

In order to backup K tokens and reparse repaired input, 2 structures needed:

1. *k-length buffer/queue* - if parser currently processing n^{th} token, queue contains tokens $(n-k) \rightarrow (n-1)$. (ML-Yacc $k = 15$)
2. *old parse stack* - if parser currently processing n^{th} token, old stack represents stack state when parser was processing $(n-k)^{th}$ token.

- Whenever token shifted onto current stack, also put onto queue tail.
- Simultaneously, queue head removed, shifted onto old stack.
- Whenever token shifted onto either stack, appropriate reductions performed.

Burke-Fisher



- Semantic actions are only applied to old stack.
  - Not desirable if semantic actions affect lexical analysis.
  - Example: `typedef` in C.

(Figure from MCI/ML.)

Burke-Fisher

For each repair R that can be applied to token $(n-k) \rightarrow n$:

1. copy queue, copy n^{th} token
2. copy old parse stack
3. apply R to copy of queue or copy of n^{th} token
4. reparse queue copy (and copy of n^{th} token) from old stack copy
5. evaluate R

Choose best repair R, and apply.

Burke-Fisher in ML-YACC

Semantic Values

- Insertions need semantic values

```
%value ID {"bogus"}
%value INT {1}
%value STRING {"STRING")
```

Programmer-Specified Substitutions

- Some single token insertions and deletions are common.
- Some multiple token insertions and deletions are common.

```
%change EQ -> ASSIGN | SEMICOLON ELSE -> ELSE
           |  -> IN INT END
```