LR(k) Parsing

CS 4100
Gordon Stewart
Ohio University

LL(1)

Some grammars are easy to parse by hand using an algorithm you saw in 3200: ***recursive descent***

Grammars in this class have parsing tables with no duplicate entries.

	FIRST	FOLLOW
E	num	
E’	+	$
T	num	+, $

	num	+	$
E	E -> T E’		
E’		E’ -> + T E’	E’ ->
T	T -> num		

We call such grammars

LL(1)

for “**L**eft-to-right Parse,

Leftmost derivation,

1 symbol of lookahead”

LL(k)

In general, an ***LL(k)*** grammar is one with no duplicate entries in its ***k-lookahead*** parsing table.

	num num	num +	+ num	+ +	…
E	…			…	
E’		…	…		
T	…				

But the size of such tables quickly grows out of hand (exponential: $(\#symbols)^k$)

LR(k)

LR(k) stands for "Left-to-right parse, rightmost derivation, k-token lookahead"

LR(k) (and its variants) is what's actually used by parser generators like Yacc, Menhir, etc.

Why are LR(k) parsers ***more powerful*** than LL(k)?

- LL(k) predicts which production to use just by looking at the next k input tokens
- LR(k) ***defers*** decisions regarding which production to apply until ***after*** the entire RHS of the production has been shifted onto a stack

LR(k)

An LR(k) grammar operates over

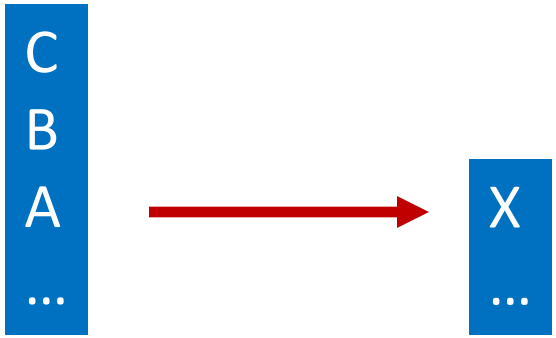
- a ***stack*** (of symbols – terminals or nonterminals) and
- the ***input***.

At every step, the LR(k) parser either

- ***shifts*** the next input token onto the stack, or
- ***reduces*** the topmost symbols on the stack

  Choose a production X -> A B C.

  Pop C, B, A from the stack and push X.



Whether to shift or reduce is determined by a DFA! (operating on the stack and first k tokens of the input)

LR(k) by Example

```
S -> S; S
S -> id := E
S -> print ( L )

E -> id
E -> num
E -> E + E
E -> (S, E)

L -> E
L -> L, E
```

[Appel Grammar 3.1]

```
a := 7;
b := c + (d := 5 + 6, d)
```

Is this string in the language?

Stack	*Input*	*Action*
1	a := 7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id_4	:= 7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id_4 :=6	7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id_4 :=6 num_{10}	; b := c + (d := 5 + 6 , d) $	*reduce* $E \rightarrow$ num
1 id_4 :=6 E_{11}	; b := c + (d := 5 + 6 , d) $	*reduce* $S \rightarrow$ id:=E
1 S_2	; b := c + (d := 5 + 6 , d) $	*shift*
1 S_2 ;3	b := c + (d := 5 + 6 , d) $	*shift*
1 S_2 ;3 id_4	:= c + (d := 5 + 6 , d) $	*shift*
1 S_2 ;3 id_4 :=6	c + (d := 5 + 6 , d) $	*shift*
1 S_2 ;3 id_4 :=6 id_{20}	+ (d := 5 + 6 , d) $	*reduce* $E \rightarrow$ id
1 S_2 ;3 id_4 :=6 E_{11}	+ (d := 5 + 6 , d) $	*shift*
1 S_2 ;3 id_4 :=6 E_{11} +16	(d := 5 + 6 , d) $	*shift*
1 S_2 ;3 id_4 :=6 E_{11} +16 (8	d := 5 + 6 , d) $	*shift*
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 id_4	:= 5 + 6 , d) $	*shift*
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 id_4 :=6	5 + 6 , d) $	*shift*
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 id_4 :=6 num_{10}	+ 6 , d) $	*reduce* $E \rightarrow$ num
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 id_4 :=6 E_{11}	+ 6 , d) $	*shift*
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 id_4 :=6 E_{11} +16	6 , d) $	*shift*
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 id_4 :=6 E_{11} +16 num_{10}	, d) $	*reduce* $E \rightarrow$ num
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 id_4 :=6 E_{11} +16 E_{17}	, d) $	*reduce* $E \rightarrow E + E$
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 id_4 :=6 E_{11}	, d) $	*reduce* $S \rightarrow$ id:=E
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 S_{12}	, d) $	*shift*
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 S_{12} ,18	d) $	*shift*
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 S_{12} ,18 id_{20}	) $	*reduce* $E \rightarrow$ id
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 S_{12} ,18 E_{21}	) $	*shift*
1 S_2 ;3 id_4 :=6 E_{11} +16 (8 S_{12} ,18 E_{21})22	$	*reduce* $E \rightarrow (S, E)$
1 S_2 ;3 id_4 :=6 E_{11} +16 E_{17}	$	*reduce* $E \rightarrow E + E$
1 S_2 ;3 id_4 :=6 E_{11}	$	*reduce* $S \rightarrow$ id:=E
1 S_2 ;3 S_5	$	*reduce* $S \rightarrow S; S$
1 S_2	$	*accept*

FIGURE 3.18. Shift-reduce parse of a sentence. Numeric subscripts in the *Stack* are DFA state numbers; see Table 3.19.

In state 1, on input token **id**, shift and go to state 4

	id	num	print	;	,	+	:=	(	)	$	S	E	L
1	s4		s7								g2		
2				s3						a			
3	s4		s7								g5		
4							s6						
5				r1	r1					r1			
6	s20	s10						s8				g11	
7								s9					
8	s4		s7								g12		
9	s20	s10						s8				g15	g14
10				r5	r5	r5			r5	r5			
11				r2	r2	s16				r2			
12				s3	s18								
13				r3	r3					r3			
14					s19				s13				
15					r8				r8				
16	s20	s10						s8				g17	
17				r6	r6	s16			r6	r6			
18	s20	s10						s8				g21	
19	s20	s10						s8				g23	
20				r4	r4	r4			r4	r4			
21									s22				
22				r7	r7	r7			r7	r7			
23					r9	s16			r9				

After reducing an E, jump to state 23

TABLE 3.19. LR parsing table for Grammar 3.1.

In state 22, on input token **;** reduce using production 7

CONFLICTS

Shift/Reduce Conflicts

- In a given state, looking at next input token, can either ***shift*** the token and carry on, or ***reduce***
- ***Example:***

1) E -> num
2) E -> E + E
3) E -> E * E

"1 * 2 + 3"

1	shift
1 *	shift
1 * 2	shift

1	shift
1 *	shift
1 * 2	shift
1 * 2 +	shift

1	shift
1 *	shift
1 * 2	shift
E(1 * 2)	reduce(2)

I'm eliding the reductions by rule 1) here.

Shift/Reduce in Menhir

Calc

parser.mly
https://github.com/gstew5/cs4100-public/blob/master/calc-example/parser.mly

```
%inline binop:
| PLUS { BPlus }
| MINUS { BMinus }
| TIMES { BTimes }
| DIV { BDiv }

exp:
| LPAREN e = exp RPAREN { e }
| n = INTCONST { EInt n }
| e1 = exp b = binop e2 = exp { EBinop(b, e1, e2) }
```

```
$ make
ocamlbuild -use-menhir -use-ocamlfind calc.native
+ menhir --ocamlc 'ocamlfind ocamlc -package batteries' --infer
parser.mly
Warning: 4 states have shift/reduce conflicts.
Warning: 16 shift/reduce conflicts were arbitrarily resolved.
Finished, 18 targets (0 cached) in 00:00:00.
```

Shift/Reduce in Menhir

```
$ menhir --explain parser.mly
   4 shift/reduce conflicts …

$ less parser.conflicts
```

```
** In state 9, looking ahead at TIMES, reducing production
** exp -> exp MINUS exp
** is permitted because of the following sub-derivation:

exp TIMES exp // lookahead token appears
exp MINUS exp .

** In state 9, looking ahead at TIMES, shifting is permitted
** because of the following sub-derivation:

exp MINUS exp
          exp . TIMES exp
```

Fixing Shift/Reduce Conflicts

Calc

parser.mly
https://github.com/gstew5/cs4100-public/blob/master/calc-example/parser.mly

```
%left PLUS MINUS
%left TIMES DIV

…

%%

%inline binop:
| PLUS { BPlus }
| MINUS { BMinus }
| TIMES { BTimes }
| DIV { BDiv }

exp:
| LPAREN e = exp RPAREN { e }
| n = INTCONST { EInt n }
| e1 = exp b = binop e2 = exp { EBinop(b, e1, e2) }
```

TIMES, DIV higher precedence

Prefer exp MINUS exp
exp . TIMES exp (shift)

over reducing via
exp -> exp MINUS exp

Reduce/Reduce Conflics

- Given current lookahead and state, ***reduce/reduce*** conflict occurs when it's possible to reduce via multiple distinct productions

- ***shift/reduce*** conflicts can often be resolved using precedence directives

- ***reduce/reduce*** conflicts are more pernicious; refactor the grammar to remove them, if possible