Syntax Analysis Part 2

Dr. Ir. Bart Kienhuis
Computer Systems Group
LIACS

Follow Sets

- **Rules for Follow Sets**
  - First put $ (the end of input marker) in Follow(S) (S is the start symbol)
  - If there is a production A → aBb, (where a can be a whole string) **then** everything in FIRST(b) except for ε is placed in FOLLOW(B).
  - If there is a production A → aB, **then** everything in FOLLOW(A) is in FOLLOW(B)
  - If there is a production A → aBb, where FIRST(b) contains ε, **then** everything in FOLLOW(A) is in FOLLOW(B)

Bottom-Up Parsing

- LR methods (Left-to-right, Reftmost derivation)
  - LR(0), SLR, Canonical LR, LALR
- Other special cases:
  - Shift-reduce parsing
  - Operator-precedence parsing

3

Operator-Precedence Parsing

- Special case of shift-reduce parsing
- We will not further discuss (you can skip textbook section 4.6)

4

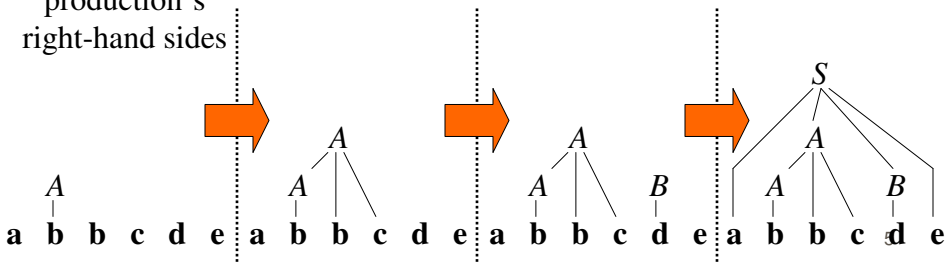
Shift-Reduce Parsing

Grammar:
$S \rightarrow \mathbf{a}\ A\ B\ \mathbf{e}$
$A \rightarrow A\ \mathbf{b}\ \mathbf{c} \mid \mathbf{b}$
$B \rightarrow \mathbf{d}$

These match production's right-hand sides

Reducing a sentence:
a b b c d e
a *A* **b c** d e
a *A* **d** e
a *A* *B* **e**
S

Shift-reduce corresponds to a rightmost derivation:
$S \Rightarrow_{rm} \mathbf{a}\ A\ B\ \mathbf{e}$
$\Rightarrow_{rm} \mathbf{a}\ A\ \mathbf{d}\ \mathbf{e}$
$\Rightarrow_{rm} \mathbf{a}\ A\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$
$\Rightarrow_{rm} \mathbf{a}\ \mathbf{b}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$



Handles

A *handle* is a substring of grammar symbols in a *right-sentential form* that matches a right-hand side of a production

Grammar:
$S \rightarrow \mathbf{a}\ A\ B\ \mathbf{e}$
$A \rightarrow A\ \mathbf{b}\ \mathbf{c} \mid \mathbf{b}$
$B \rightarrow \mathbf{d}$

a b b c d e
a *A* **b c** d e
a *A* **d** e
a *A* *B* **e**
S

Handle

a b b c d e
a *A* **b c** d e
a *A* *A* **e**
… ?

NOT a handle, because further reductions will fail (result is not a sentential form)

Stack Implementation of Shift-Reduce Parsing

Grammar:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow \mathbf{id}$

Stack	Input	Action
$	**id+id*id$**	shift
$**id**	**+id*id$**	reduce $E \rightarrow$ **id**
$*E*	**+id*id$**	shift
$*E*+	**id*id$**	shift
$*E*+**id**	***id$**	reduce $E \rightarrow$ **id**
$*E*+*E*	***id$**	shift (or reduce?)
$*E*+*E**	**id$**	shift
$*E*+*E***id**	**$**	reduce $E \rightarrow$ **id**
$*E*+*E***E*	**$**	reduce $E \rightarrow E * E$
$*E*+*E*	**$**	reduce $E \rightarrow E + E$
$*E*	**$**	accept

How to resolve conflicts?

Find handles to reduce

7

Conflicts

- Shift-reduce and reduce-reduce conflicts are caused by
  - The limitations of the LR parsing method (even when the grammar is unambiguous)
  - Ambiguity of the grammar

8

Shift-Reduce Parsing: Shift-Reduce Conflicts

Ambiguous grammar:
$S \rightarrow$ **if** E **then** S
| **if** E **then** S **else** S
| **other**

Resolve in favor of shift, so **else** matches closest **if**

Stack	Input	Action
$…	…**$**	…
$…**if** E **then** S	**else**…**$**	shift or reduce?

9

Shift-Reduce Parsing: Reduce-Reduce Conflicts

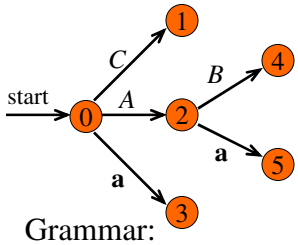
Grammar:
$C \rightarrow A\ B$
$A \rightarrow$ **a**
$B \rightarrow$ **a**

Resolve in favor of reduce $A \rightarrow$ **a**, otherwise we're stuck!

Stack	Input	Action
$	**aa$**	shift
$a	**a$**	reduce $A \rightarrow$ **a** or $B \rightarrow$ **a** ?

10

LR(k) Parsers: Use a DFA for Shift/Reduce Decisions



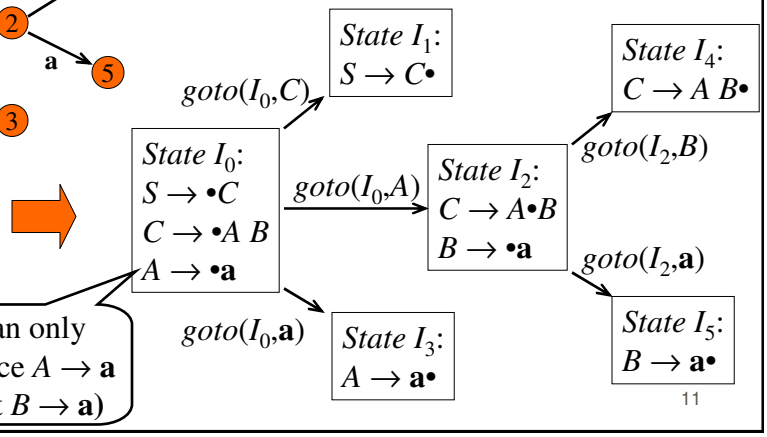
Grammar:

$S \rightarrow C$

$C \rightarrow A\ B$

$A \rightarrow \mathbf{a}$

$B \rightarrow \mathbf{a}$



Can only reduce $A \rightarrow \mathbf{a}$ (not $B \rightarrow \mathbf{a}$)

11

DFA for Shift/Reduce Decisions

The states of the DFA are used to determine if a handle is on top of the stack

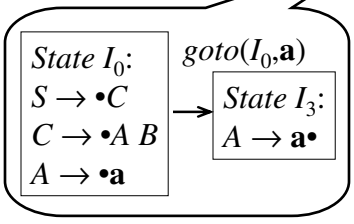
Grammar:

$S \rightarrow C$

$C \rightarrow A\ B$

$A \rightarrow \mathbf{a}$

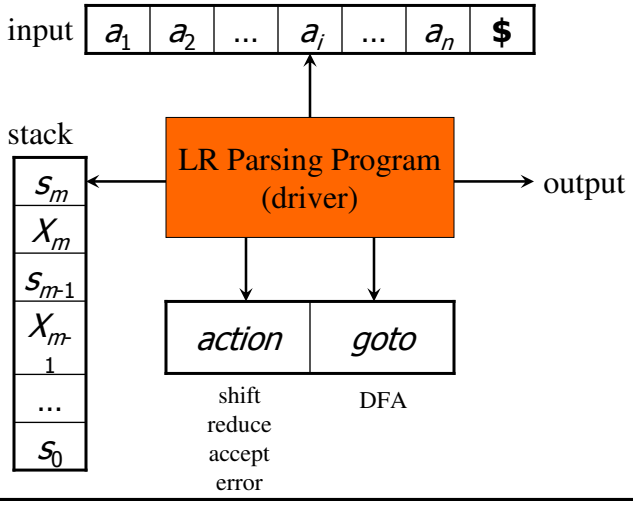
$B \rightarrow \mathbf{a}$



Stack	Input	Action
\$ 0	**aa**\$	start in state 0
\$ 0	**aa**\$	shift (and goto state 3)
\$ 0 **a** 3	**a**\$	reduce $A \rightarrow \mathbf{a}$ (goto 2)
\$ 0 *A* 2	**a**\$	shift (goto 5)
\$ 0 *A* 2 **a** 5	\$	reduce $B \rightarrow \mathbf{a}$ (goto 4)
\$ 0 *A* 2 *B* 4	\$	reduce $C \rightarrow AB$ (goto 1)
\$ 0 *C* 1	\$	reduce $S \rightarrow C$
\$ 0 *S* 1	\$	accept

12

Model of an LR Parser



LR Parsing

Configuration (= LR parser state):

$$(s_0\, X_1\, s_1\, X_2\, s_2 \ldots X_m\, s_m,\quad a_i\, a_{i+1} \ldots a_n\, \$)$$

stack | input

If $action[s_m,a_i]$ = shift s, then push a_i, push s, and advance input:

$$(s_0\, X_1\, s_1\, X_2\, s_2 \ldots X_m\, s_m\, a_i\, s,\quad a_{i+1} \ldots a_n\, \$)$$

If $action[s_m,a_i]$ = reduce $A \rightarrow \beta$ and $goto[s_{m-r},A] = s$ with $r=|\beta|$ then pop $2r$ symbols, push A, and push s:

$$(s_0\, X_1\, s_1\, X_2\, s_2 \ldots X_{m-r}\, s_{m-r}\, A\, s,\quad a_i\, a_{i+1} \ldots a_n\, \$)$$

If $action[s_m,a_i]$ = accept, then stop

If $action[s_m,a_i]$ = error, then attempt recovery

Example LR Parse Table

Grammar:
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow$ **id**

Shift & goto 5

Reduce by production #1

	action						*goto*		
state	**id**	**+**	*****	**(**	**)**	**$**	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s1 1				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

15

Example LR Parsing

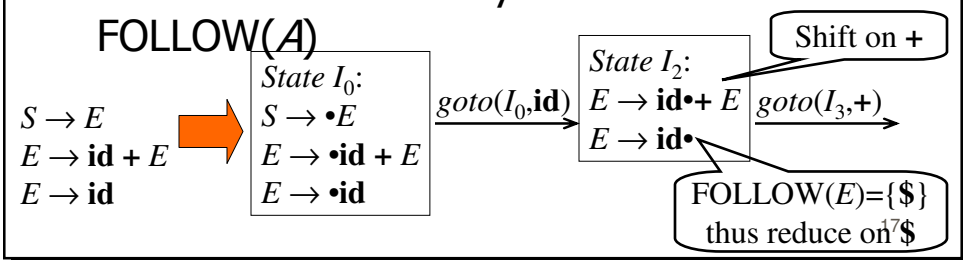
Grammar:
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow$ **id**

Stack	Input	Action
$ 0	**id*id+id$**	shift 5
$ 0 **id** 5	***id+id$**	reduce 6 goto 3
$ 0 *F* 3	***id+id$**	reduce 4 goto 2
$ 0 *T* 2	***id+id$**	shift 7
$ 0 *T* 2 * 7	**id+id$**	shift 5
$ 0 *T* 2 * 7 **id** 5	**+id$**	reduce 6 goto 10
$ 0 *T* 2 * 7 *F* 10	**+id$**	reduce 3 goto 2
$ 0 *T* 2	**+id$**	reduce 2 goto 1
$ 0 *E* 1	**+id$**	shift 6
$ 0 *E* 1 + 6	**id$**	shift 5
$ 0 *E* 1 + 6 **id** 5	**$**	reduce 6 goto 3
$ 0 *E* 1 + 6 *F* 3	**$**	reduce 4 goto 9
$ 0 *E* 1 + 6 *T* 9	**$**	reduce 1 goto 1
$ 0 *E* 1	**$**	accept

16

SLR Grammars

- SLR (Simple LR): a simple extension of LR(0) shift-reduce parsing
- SLR eliminates some conflicts by populating the parsing table with reductions $A \rightarrow \alpha$ on symbols in FOLLOW(A)



SLR Parsing Table

- Reductions do not fill entire rows
- Otherwise the same as LR(0)

1. $S \rightarrow E$
2. $E \rightarrow$ **id** + E
3. $E \rightarrow$ **id**

	id	+	$	E
0	s2			1
1			acc	
2		s3	r3	
3	s2			4
4			r2	

Shift on +

FOLLOW(E)={$}
thus reduce on $

SLR Parsing

- An LR(0) state is a set of LR(0) items
- An LR(0) item is a production with a • (dot) in the right-hand side
- Build the LR(0) DFA by
  - *Closure operation* to construct LR(0) items
  - *Goto operation* to determine transitions
- Construct the SLR parsing table from the DFA
- LR parser program uses the SLR parsing table to determine shift/reduce operations

19

LR(0) Items of a Grammar

- An *LR(0) item* of a grammar G is a production of G with a • at some position of the right-hand side
- Thus, a production
  $A \rightarrow X\ Y\ Z$
  has four items:
  $[A \rightarrow \bullet\ X\ Y\ Z]$
  $[A \rightarrow X \bullet\ Y\ Z]$
  $[A \rightarrow X\ Y \bullet\ Z]$
  $[A \rightarrow X\ Y\ Z \bullet]$
- Note that production $A \rightarrow \varepsilon$ has one item $[A \rightarrow \bullet]$

20

Constructing the set of LR(0) Items of a Grammar

1. The grammar is augmented with a new start symbol S' and production $S' \rightarrow S$
2. Initially, set $C = closure(\{[S' \rightarrow \bullet S]\})$ (this is the start state of the DFA)
3. For each set of items $I \in C$ and each grammar symbol $X \in (N \cup T)$ such that $goto(I,X) \notin C$ and $goto(I,X) \neq \emptyset$, add the set of items $goto(I,X)$ to C
4. Repeat 3 until no more sets can be added to C

21

The Closure Operation for LR(0) Items

1. Start with $closure(I) = I$
2. If $[A \rightarrow \alpha \bullet B\beta] \in closure(I)$ then for each production $B \rightarrow \gamma$ in the grammar, add the item $[B \rightarrow \bullet\gamma]$ to I if not already in I
3. Repeat 2 until no new items can be added

22

The Closure Operation (Example)

closure({[*E*' → •*E*]}) =

{ [*E*' → • *E*] } ⇒ { [*E*' → • *E*]
[*E* → • *E* + *T*]
[*E* → • *T*] } ⇒ { [*E*' → • *E*]
[*E* → • *E* + *T*]
[*E* → • *T*]
[*T* → • *T* * *F*]
[*T* → • *F*] } ⇒ { [*E*' → • *E*]
[*E* → • *E* + *T*]
[*E* → • *T*]
[*T* → • *T* * *F*]
[*T* → • *F*]
[*F* → • (*E*)]
[*F* → • **id**] }

Add [*E*→•γ]

Add [*T*→•γ]

Add [*F*→•γ]

Grammar:

E → *E* + *T* | *T*
T → *T* * *F* | *F*
F → (*E*)
F → **id**

23

The Goto Operation for LR(0) Items

1. For each item [*A*→α•*X*β] ∈ *I*, add the set of items *closure*({[*A*→α*X*•β]}) to *goto*(*I*,*X*) if not already there
2. Repeat step 1 until no more items can be added to *goto*(*I*,*X*)
3. Intuitively, *goto*(*I*,*X*) is the set of items that are valid for the viable prefix γ*X* when *I* is the set of items that are valid for γ

24

The Goto Operation (Example 1)

Suppose I = { $[E' \rightarrow \bullet E]$
$[E \rightarrow \bullet E + T]$
$[E \rightarrow \bullet T]$
$[T \rightarrow \bullet T * F]$
$[T \rightarrow \bullet F]$
$[F \rightarrow \bullet (E)]$
$[F \rightarrow \bullet \mathbf{id}]$ }

Then $goto(I,E)$
$= closure(\{[E' \rightarrow E \bullet, E \rightarrow E \bullet + T]\})$
$= \{ [E' \rightarrow E \bullet]$
$[E \rightarrow E \bullet + T] \}$

Grammar:
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E)$
$F \rightarrow \mathbf{id}$

25

The Goto Operation (Example 2)

Suppose I = { $[E' \rightarrow E \bullet]$, $[E \rightarrow E \bullet + T]$ }

Then $goto(I,+) = closure(\{[E \rightarrow E + \bullet T]\}) = \{ [E \rightarrow E + \bullet T]$
$[T \rightarrow \bullet T * F]$
$[T \rightarrow \bullet F]$
$[F \rightarrow \bullet (E)]$
$[F \rightarrow \bullet \mathbf{id}] \}$

Grammar:
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E)$
$F \rightarrow \mathbf{id}$

26

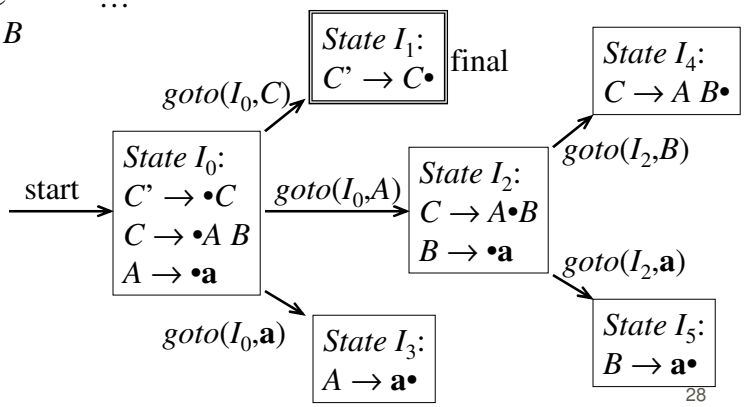
Constructing SLR Parsing Tables

1. Augment the grammar with $S' \rightarrow S$
2. Construct the set $C=\{I_0,I_1,\ldots,I_n\}$ of *LR(0) items*
3. If $[A \rightarrow \alpha \bullet a\beta] \in I_i$ and $goto(I_i,a)=I_j$ then set $action[i,a]$=shift j
4. If $[A \rightarrow \alpha \bullet] \in I_i$ then set $action[i,a]$=reduce $A \rightarrow \alpha$ for all $a \in$ FOLLOW(A) (apply only if $A \neq S'$)
5. If $[S' \rightarrow S \bullet]$ is in I_i then set $action[i,\$]$=accept
6. If $goto(I_i,A)=I_j$ then set $goto[i,A]=j$
7. Repeat 3-6 until no more entries added
8. The initial state i is the I_i holding item $[S' \rightarrow \bullet S]$

Example SLR Grammar and LR(0) Items

Augmented grammar:
1. $C' \rightarrow C$
2. $C \rightarrow A\ B$
3. $A \rightarrow \mathbf{a}$
4. $B \rightarrow \mathbf{a}$

$I_0 = closure(\{[C' \rightarrow \bullet C]\})$
$I_1 = \text{goto}(I_0,C) = closure(\{[C' \rightarrow C \bullet]\})$
…



Example SLR Parsing Table

State I_0:
$C' \rightarrow \bullet C$
$C \rightarrow \bullet A\ B$
$A \rightarrow \bullet \mathbf{a}$

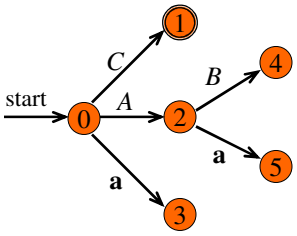
State I_1:
$C' \rightarrow C\bullet$

State I_2:
$C \rightarrow A\bullet B$
$B \rightarrow \bullet \mathbf{a}$

State I_3:
$A \rightarrow \mathbf{a}\bullet$

State I_4:
$C \rightarrow A\ B\bullet$

State I_5:
$B \rightarrow \mathbf{a}\bullet$



	a	$	C	A	B
0	s3		1	2	
1		acc			
2	s5				4
3	r3				
4		r2			
5		r4			

Grammar:
1. $C' \rightarrow C$
2. $C \rightarrow A\ B$
3. $A \rightarrow \mathbf{a}$
4. $B \rightarrow \mathbf{a}$

29

SLR and Ambiguity

- Every SLR grammar is unambiguous, but **not** every unambiguous grammar is SLR
- Consider for example the unambiguous grammar

$S \rightarrow L = R \mid R$
$L \rightarrow * R \mid \mathbf{id}$
$R \rightarrow L$

I_0:
$S' \rightarrow \bullet S$
$S \rightarrow \bullet L=R$
$S \rightarrow \bullet R$
$L \rightarrow \bullet *R$
$L \rightarrow \bullet \mathbf{id}$
$R \rightarrow \bullet L$

I_1:
$S' \rightarrow S\bullet$

I_2:
$S \rightarrow L\bullet=R$
$R \rightarrow L\bullet$

I_3:
$S \rightarrow R\bullet$

I_4:
$L \rightarrow *\bullet R$
$R \rightarrow \bullet L$
$L \rightarrow \bullet *R$
$L \rightarrow \bullet \mathbf{id}$

I_5:
$L \rightarrow \mathbf{id}\bullet$

I_6:
$S \rightarrow L=\bullet R$
$R \rightarrow \bullet L$
$L \rightarrow \bullet *R$
$L \rightarrow \bullet \mathbf{id}$

I_7:
$L \rightarrow *R\bullet$

I_8:
$R \rightarrow L\bullet$

I_9:
$S \rightarrow L=R\bullet$

action[2,=]=s6
action[2,=]=r5

no Has no SLR parsing table

30

LR(1) Grammars

- SLR too simple
- LR(1) parsing uses lookahead to avoid unnecessary conflicts in parsing table
- LR(1) item = LR(0) item + lookahead

LR(0) item:	LR(1) item:
$[A \rightarrow \alpha \bullet \beta]$	$[A \rightarrow \alpha \bullet \beta, a]$

31

LALR(1) Grammars

- LR(1) parsing tables have many states
- LALR(1) parsing (Look-Ahead LR) combines LR(1) states to reduce table size
- Less powerful than LR(1)
  - Will not introduce shift-reduce conflicts, because shifts do not use lookaheads
  - May introduce reduce-reduce conflicts, but seldom do so for grammars of programming languages

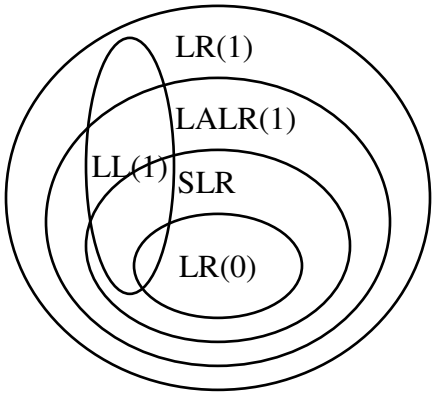
41

LL, SLR, LR, LALR Summary

- LL parse tables computed using FIRST/FOLLOW
  - Nonterminals × terminals → productions
  - Computed using FIRST/FOLLOW
- LR parsing tables computed using closure/goto
  - LR states × terminals → shift/reduce actions
  - LR states × terminals → goto state transitions
- A grammar is
  - LL(1) if its LL(1) parse table has no conflicts
  - SLR if its SLR parse table has no conflicts
  - LALR(1) if its LALR(1) parse table has no conflicts
  - LR(1) if its LR(1) parse table has no conflicts

46

LL, SLR, LR, LALR Grammars



47

Error Detection in LR Parsing

- Canonical LR parser uses full LR(1) parse tables and will never make a single reduction before recognizing the error when a syntax error occurs on the input
- SLR and LALR may still reduce when a syntax error occurs on the input, but will never shift the erroneous input symbol

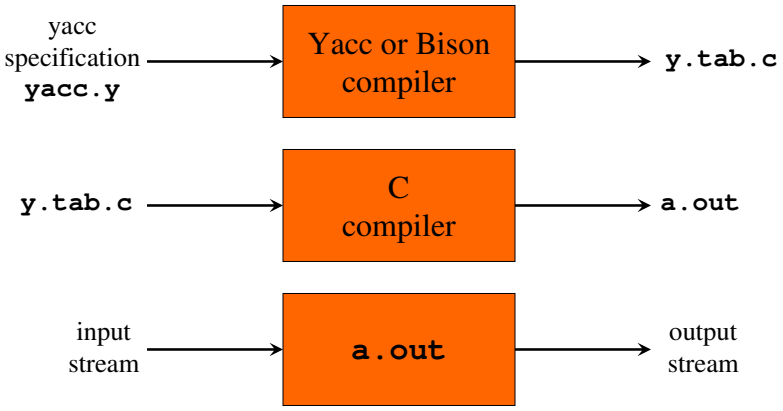
50

ANTLR, Yacc, and Bison

- *ANTLR* tool generates LL(*k*) parsers
- *Yacc* (Yet Another Compiler Compiler) generates LALR(1) parsers
- *Bison* (Yacc improved)

52

Creating an LALR(1) Parser with Yacc/Bison



53

Yacc Specification

- A *yacc specification* consists of three parts:
  yacc declarations, and C declarations in `%{ %}`
  `%%`
  translation rules
  `%%`
  user-defined auxiliary procedures
- *Translation rules* are grammar productions and actions:
  $production_1$ { $semantic\ action_1$ }
  $production_2$ { $semantic\ action_2$ }
  ...
  $production_n$ { $semantic\ action_n$ }

54

Writing a Grammar in Yacc

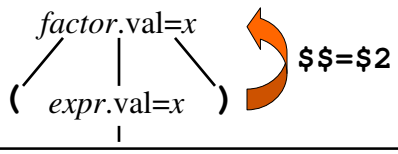
- Productions in Yacc are of the form
  Nonterminal : tokens/nonterminals { *action* }
  | tokens/nonterminals { *action* }
  ...
  ;
- Tokens that are single characters can be used directly within productions, e.g. `'+'`
- Named tokens must be declared first in the declaration part using
  `%token` *TokenName*

55

Synthesized Attributes

- Semantic actions may refer to values of the *synthesized attributes* of terminals and nonterminals in a production:
  $X : Y_1\ Y_2\ Y_3 \ldots Y_n$ { *action* }
  - `$$` refers to the value of the attribute of X
  - `$`i refers to the value of the attribute of Y_i
- For example

```
factor : '(' expr ')' { $$=$2; }
```



56

Example 1

```
%{ #include <ctype.h> %}
%token DIGIT
%%
line    : expr '\n'          { printf("%d\n", $1); }
        ;
expr    : expr '+' term      { $$ = $1 + $3; }
        | term               { $$ = $1; }
        ;
term    : term '*' factor    { $$ = $1 * $3; }
        | factor             { $$ = $1; }
        ;
factor  : '(' expr ')'       { $$ = $2; }
        | DIGIT              { $$ = $1; }
        ;
%%
int yylex()
{ int c = getchar();
  if (isdigit(c))
  { yylval = c-'0';
    return DIGIT;
  }
  return c;
}
```

Also results in definition of **`#define DIGIT xxx`**

Attribute of **`term`** (parent)

Attribute of **`factor`** (child)

Attribute of token (stored in **`yylval`**)

Example of a very crude lexical analyzer invoked by the parser

Dealing With Ambiguous Grammars

- By defining operator precedence levels and left/right associativity of the operators, we can specify ambiguous grammars in Yacc, such as $E \rightarrow E$**+**$E \mid E$**-**$E \mid E$*****$E \mid E$**/**$E \mid$ **(**E**)** $\mid$ **-**$E \mid$ **num**
- To define precedence levels and associativity in Yacc's declaration part:

```
%left '+' '-'
%left '*' '/'
%right UMINUS
```

Example 2

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines   : lines expr '\n'       { printf("%g\n", $2); }
        | lines '\n'
        | /* empty */
        ;
expr    : expr '+' expr         { $$ = $1 + $3; }
        | expr '-' expr         { $$ = $1 - $3; }
        | expr '*' expr         { $$ = $1 * $3; }
        | expr '/' expr         { $$ = $1 / $3; }
        | '(' expr ')'          { $$ = $2; }
        | '-' expr %prec UMINUS { $$ = -$2; }
        | NUMBER
        ;
%%
```

Double type for attributes and **yylval**

Example 2 (cont'd)

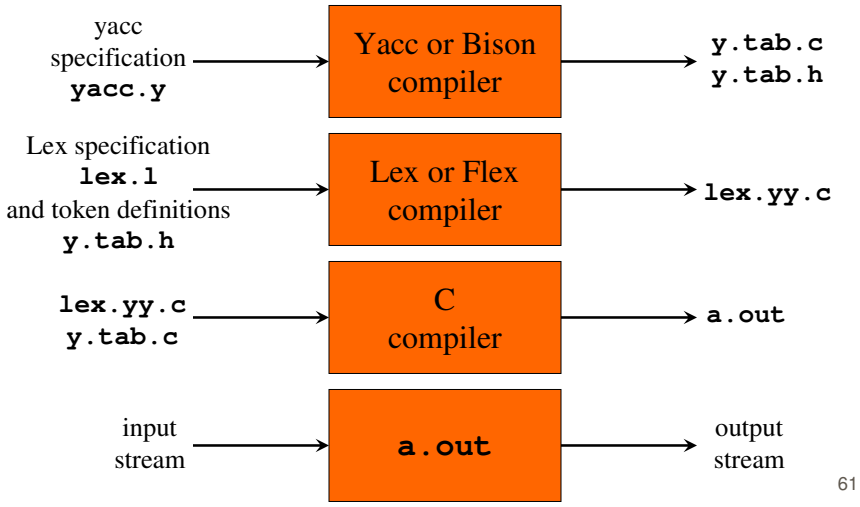
```
%%
int yylex()
{ int c;
  while ((c = getchar()) == ' ')
    ;
  if ((c == '.') || isdigit(c))
  { ungetc(c, stdin);
    scanf("%lf", &yylval);
    return NUMBER;
  }
  return c;
}
int main()
{ if (yyparse() != 0)
    fprintf(stderr, "Abnormal exit\n");
  return 0;
}
int yyerror(char *s)
{ fprintf(stderr, "Error: %s\n", s);
}
```

Crude lexical analyzer for fp doubles and arithmetic operators

Run the parser

Invoked by parser to report parse errors

Combining Lex/Flex with Yacc/Bison



Lex Specification for Example 2

```
%option noyywrap
%{
#include "y.tab.h"

extern double yylval;
%}
number  [0-9]+\.?|[0-9]*\.[0-9]+
%%
[ ]             { /* skip blanks */ }
{number}        { sscanf(yytext, "%lf", &yylval);
                  return NUMBER;
                }
\n|.            { return yytext[0]; }
```

Generated by Yacc, contains `#define NUMBER xxx`

Defined in `y.tab.c`

```
yacc -d example2.y
lex example2.l
gcc y.tab.c lex.yy.c
./a.out
```

```
bison -d -y example2.y
flex example2.l
gcc y.tab.c lex.yy.c
./a.out
```

Error Recovery in Yacc

```
%{
…
%}
…
%%
lines   : lines expr '\n'      { printf("%g\n", $2; }
        | lines '\n'
        | /* empty */
        | error '\n'           { yyerror("reenter last line: ");
                                 yyerrok;
                               }
        ;
…
```

Error production: set error mode and skip input until newline

Reset parser to normal mode

63