Compiler Construction

ARNOLD MEIJSTER
A.MEIJSTER@RUG.NL

Conflicts in Parsing table

Grammar:

Grammar:

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

2: $S \rightarrow a\ S$

3: $S \rightarrow$

$I_0 = closure(\{S' \rightarrow \bullet\ S\ \$\})$:

$S' \rightarrow \bullet\ S\ \$$

$S \rightarrow \bullet\ a\ S$

$S \rightarrow \bullet$

$I_1 = \text{goto}(I_0, S)$:

$S' \rightarrow S \bullet \$$

$I_2 = \text{goto}(I_0, a)$:

$S \rightarrow a \bullet\ S$

$S \rightarrow \bullet\ a\ S$

$S \rightarrow \bullet$

$I_3 = \text{goto}(I_2, S)$:

$S \rightarrow a\ S\ \bullet$

Action Table:

	a	$
0	S, 2 R 3	R 3
1		Accept
2	S, 2 R 3	R 3
3	R 2	R 2

2 Shift-Reduce Conflicts

Idea: Choose **shift** because a is not in Follow(S)

SLR(1) Parsing

SLR(1) parsing makes a reduction by $A \rightarrow \alpha$ in state i if the current token is **a** and:

$A \rightarrow \alpha.$ in I_i

a is in Follow(A)

Simple LR (SLR) Parsing

Construct Action Table *action*, indexed by *states* × *terminals*, and Goto Table *goto*, indexed by *states* × *nonterminals*:

Construct $\{I_0, I_1, \ldots, I_n\}$, the set of LR(0) item sets of the grammar.
For each i, $0 \le i \le n$, do the following:

- If $A \to \alpha \bullet a\beta \in I_i$ and $goto(I_i, a) = I_j$ then set $action[i, a] = shift\ \ j$
- If $A \to \gamma \bullet \in I_i$ (A is not the start symbol) then
  for each $a \in FOLLOW(A)$, set $action[i, a] = reduce\ \ A \to \gamma$
- If $S' \to S \bullet \$ \in I_i$ then set $action[i, \$] = accept$
- If $goto(I_i, A) = I_j$ (A is a nonterminal) then set $goto[i, A] = j$

SLR(1) parsing table

$S' \to S\ \$$
$S \to a\ S$
$S \to$

	a	\$
0	S, 2 R 3	R 3
1		Accept
2	S, 2 R 3	R 3
3	R 2	R 2

Item Sets:

I_0	$= closure(\{S' \to \bullet\ S\})$	$S' \to \bullet\ S$ $S \to \bullet\ a\ S$ $S \to \bullet$
I_1	$= goto(I_0, S)$	$S' \to S \bullet$
I_2	$= goto(I_0, a)$	$S \to a \bullet\ S$ $S \to \bullet\ a\ S$ $S \to \bullet$
I_3	$= goto(I_2, S)$	$S \to a\ S\ \bullet$

$FOLLOW(S) = \{\$\}$

SLR Action Table:

	a	\$
0	S, 2	R 3
1		Acc
2	S, 2	R 3
3		R 2

Another example: SLR(1) Parsing

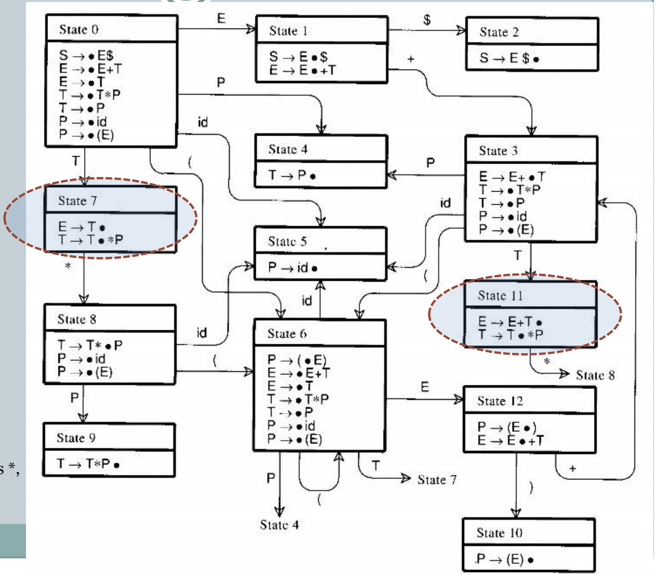
$S \to E\ \$$

$E \to E + T \mid T$

$T \to T * P \mid P$

$P \to id \mid (E)$

Is this an LR(0) Grammar?



The grammar is not LR(0):

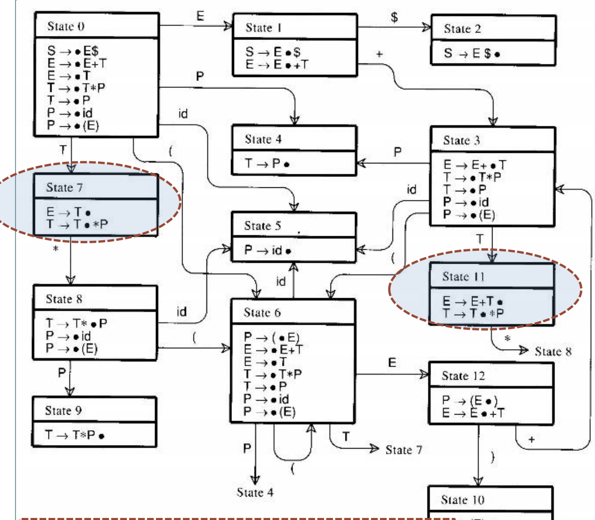
See states 7,11

Follow(E)={$,+,)}

So, the grammar is SLR(1):

In these states, shift if the lookahead is *, otherwise reduce.

Example: SLR(1) Parsing



S→E $
E→E + T | T
T→T * P | P
P→id | (E)

State	Lookahead					
	+	*	ID	(	)	$
0			S	S		
1	S					A
2						
3			S	S		
4	R5	R5			R5	R5
5	R6	R6			R6	R6
6			S	S		
7	R3	S			R3	R3
8			S	S		
9	R4	R4			R4	R4
10	R7	R7			R7	R7
11	R2	S			R2	R2
12	S				S	

The grammar is SLR(1)

Exercise

Consider the following grammar:

0: $S \rightarrow E$

1: $E \rightarrow 1\ E$

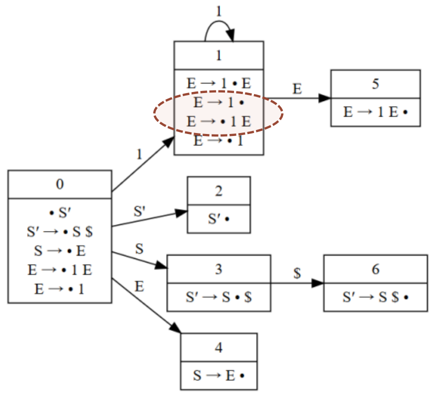
2: $E \rightarrow 1$

Show that the grammar can be parsed by an SLR parser but not by an LR(0) parser.

LR(0) automaton

- (0) $S \rightarrow E$
- (1) $E \rightarrow 1\ E$
- (2) $E \rightarrow 1$

State	\$	1	S'	S	E
0		shift(1)	2	3	4
1	reduce($E \rightarrow \mathbf{1}$)	shift(1) reduce($E \rightarrow \mathbf{1}$)			5
2	accept	accept			
3	shift(6)				
4	reduce($S \rightarrow E$)	reduce($S \rightarrow E$)			
5	reduce($E \rightarrow \mathbf{1}\ E$)	reduce($E \rightarrow \mathbf{1}\ E$)			
6	reduce($S' \rightarrow S\ \$$)	reduce($S' \rightarrow S\ \$$)			



SLR(1) tables

- (0) $S \rightarrow E$
- (1) $E \rightarrow 1\ E$
- (2) $E \rightarrow 1$

State	$	1
0		shift(1)
1	reduce($E \rightarrow \mathbf{1}$)	shift(1) reduce($E \rightarrow \mathbf{1}$)
2	accept	accept
3	shift(6)	
4	reduce($S \rightarrow E$)	reduce($S \rightarrow E$)
5	reduce($E \rightarrow \mathbf{1}\ E$)	reduce($E \rightarrow \mathbf{1}\ E$)
6	reduce($S' \rightarrow S$ $)	reduce($S' \rightarrow S$ $)

State	$	1	$	S'	S	E
0		shift(1)		2	3	4
1	reduce($E \rightarrow \mathbf{1}$)	shift(1)				5
2			accept			
3	shift(6)					
4	reduce($S \rightarrow E$)					
5	reduce($E \rightarrow \mathbf{1}\ E$)					
6			reduce($S' \rightarrow S$ $)			

Note that 1 is not in the follow set of E.

Another shift/reduce conflict: non-SLR(1)

$S \to L=R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

I_0: $S' \to .S$
$S \to .L=R$
$S \to .R$
$L \to .*R$
$L \to .id$
$R \to .L$

I_1: $S' \to S.$

I_2: $S \to L.=R$
$R \to L.$

I_3: $S \to R.$

I_4: $L \to *.R$
$R \to .L$
$L \to .*R$
$L \to .id$

I_5: $L \to id.$

I_6: $S \to L=.R$
$R \to .L$
$L \to .*R$
$L \to .id$

I_7: $L \to *R.$

I_8: $R \to L.$

I_9: $S \to L=R.$

Problem

FOLLOW(R)={=,$}

= → shift 6
→ reduce by $R \to L$

shift/reduce conflict

Action[2,=] = shift 6

Action[2,=] = reduce by $R \to L$

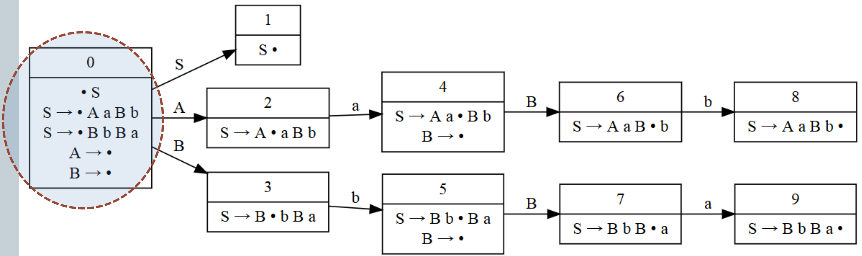
[$S \Rightarrow L=R \Rightarrow *R=R$] so follow(R) contains =

Deficiencies of SLR(1) parsing

SLR(1) treats all occurrences of a RHS on stack as identical. Only a few of these reductions may lead to a successful parse.

Example:

$$S \rightarrow A\,\mathrm{a}\,A\,\mathrm{b} \qquad A \rightarrow$$
$$S \rightarrow B\,\mathrm{b}\,B\,\mathrm{a} \qquad B \rightarrow$$



Reduce/Reduce conflicts in parse tables

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

I_0: $S' \rightarrow .S$

$S \rightarrow .AaAb$

$S \rightarrow .BbBa$

$A \rightarrow .$

$B \rightarrow .$

Problem

FOLLOW(A)={a,b}

FOLLOW(B)={a,b}

a $\rightarrow$ $\boxed{\text{reduce by } A \rightarrow \varepsilon}$

a $\rightarrow$ reduce by $B \rightarrow \varepsilon$

reduce/reduce conflict

b $\rightarrow$ reduce by $A \rightarrow \varepsilon$

b $\rightarrow$ $\boxed{\text{reduce by } B \rightarrow \varepsilon}$

reduce/reduce conflict

Since *Follow* (*A*) = *Follow*(*B*), we have a reduce/reduce conflict in state 0.

Another example of failure of SLR(1)

Elem→(List, Elem)

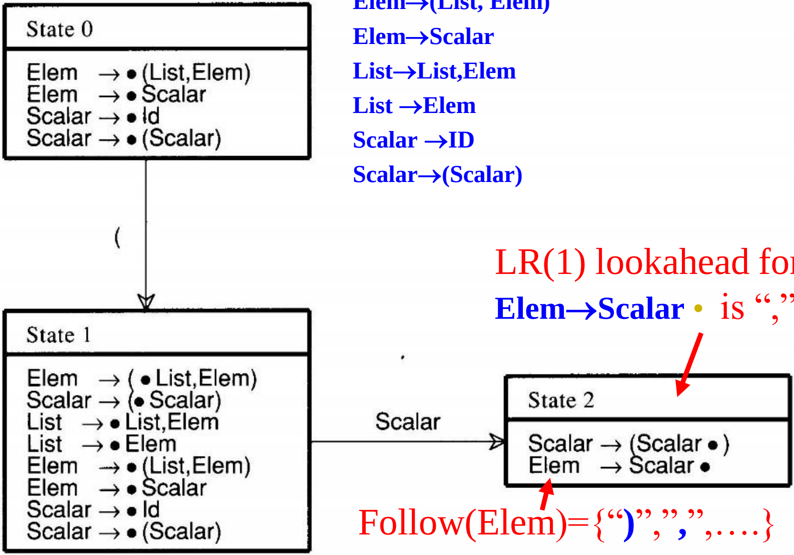
Elem→Scalar

List→List,Elem

List →Elem

Scalar →ID

Scalar→(Scalar)



LR(1) lookahead for **Elem→Scalar •** is “,”

Follow(Elem)={“)”,”,”,….}

LR(1) items

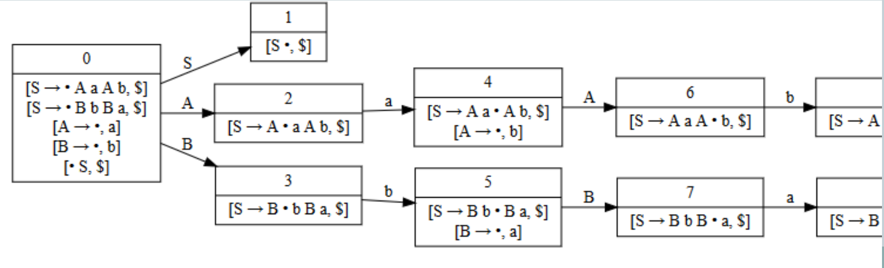
- To make a correct choice between reductions, the states need to carry more information.
- Extra information is put into a state by including a terminal symbol as a second component in an item.
- LR(1) item: $A \rightarrow \alpha.\beta$, **a** where **a** is the look-ahead of the item
  - Of course. The symbol a must be from Follow(A).
  - The 1 in LR(1) refers to the length of the second component, so an LR(2) item would be something like $A \rightarrow \alpha.\beta$, **a b**
- The look-ahead has no effect in an item of the form $[A \rightarrow \alpha.\beta, a]$, where β is not $\in$.
- But an item of the form $[A \rightarrow \alpha., a]$ calls for a reduction by $A \rightarrow \alpha$ only if the next input symbol is a.

LR(1) parsing

Construct LR(1) items of the form $A \rightarrow \alpha \cdot \beta, a$, which means:

The production $A \rightarrow \alpha\beta$ can be applied when the next token on input stream is a.

$$S \rightarrow AaAb \qquad A \rightarrow$$
$$S \rightarrow BbBa \qquad B \rightarrow$$



Construction of LR(1) parse tables

1. Construct the collection of LR(1) item sets: $\{I_0,\ldots,I_n\}$
2. Create the parsing action table as follows
   - If a is a terminal, $A\to\alpha.a\beta,b$ in I_i and $goto(I_i,a)=I_j$ then action[i,a] = ***shift j***.
   - If $A\to\alpha.,a$ is in I_i, then action[i,a] = ***reduce*** $\boldsymbol{A\to\alpha}$ where $A\neq S'$.
   - If $S'\to S.,\$$ is in I_i, then action[i,$] = ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not LR(1).
3. Create the parsing goto table
   - for all non-terminals A, if $goto(I_i,A)=I_j$ then goto[i,A]=j
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser contains $S'\to.S,\$$

LR(1) and LALR(1) parsing

LR(1) parsing: Parse tables built using LR(1) item sets.

LALR(1) parsing: *Look Ahead* LR(1)

Merge LR(1) item sets; then build parsing table.

Typically, LALR(1) parsing tables are much smaller than LR(1) parsing table.

LALR PARSING: Cores

A *core* is a set of LR(0) items that is obtained from a set of LR(1)-items by ignoring the look ahead information.

For example: Let s1 and s2 be two states in an LR(1) parser.

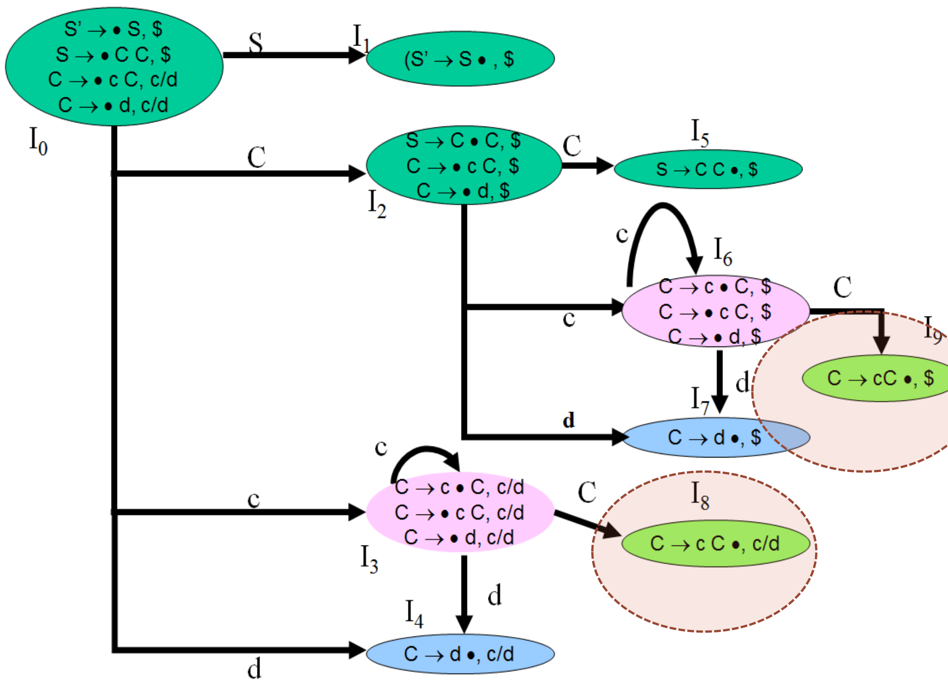
S1 = {C ->c.C, c/d; C -> .cC, c/d; C -> .d, c/d}
S2 = {C ->c.C, $; C -> .cC, $; C -> .d, $}

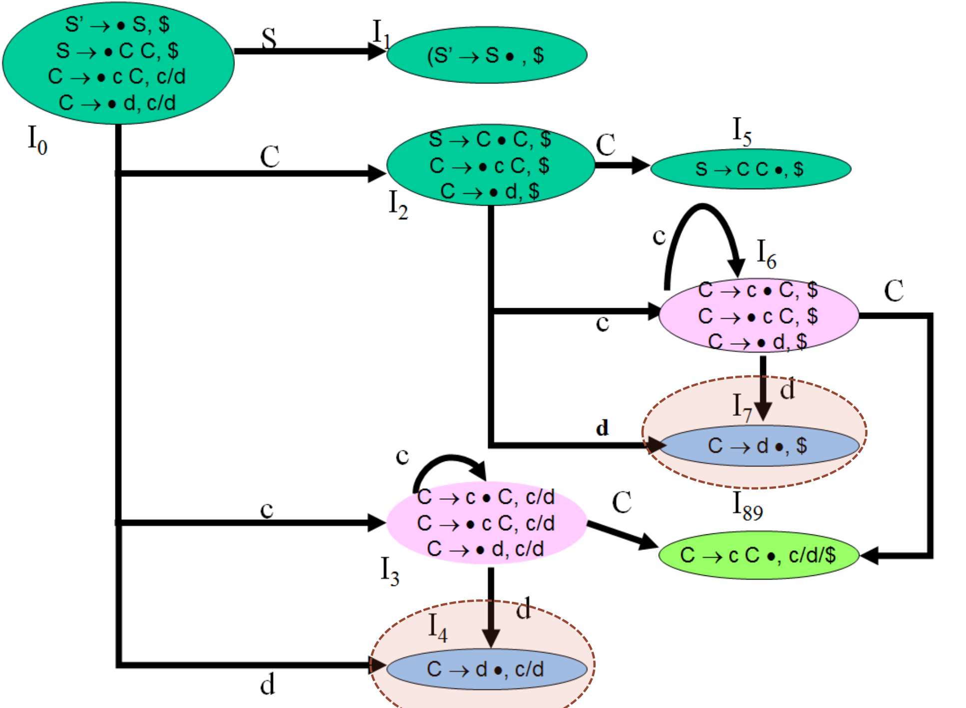
These two states have the same core S12 consisting of only the production rules without any look ahead information.

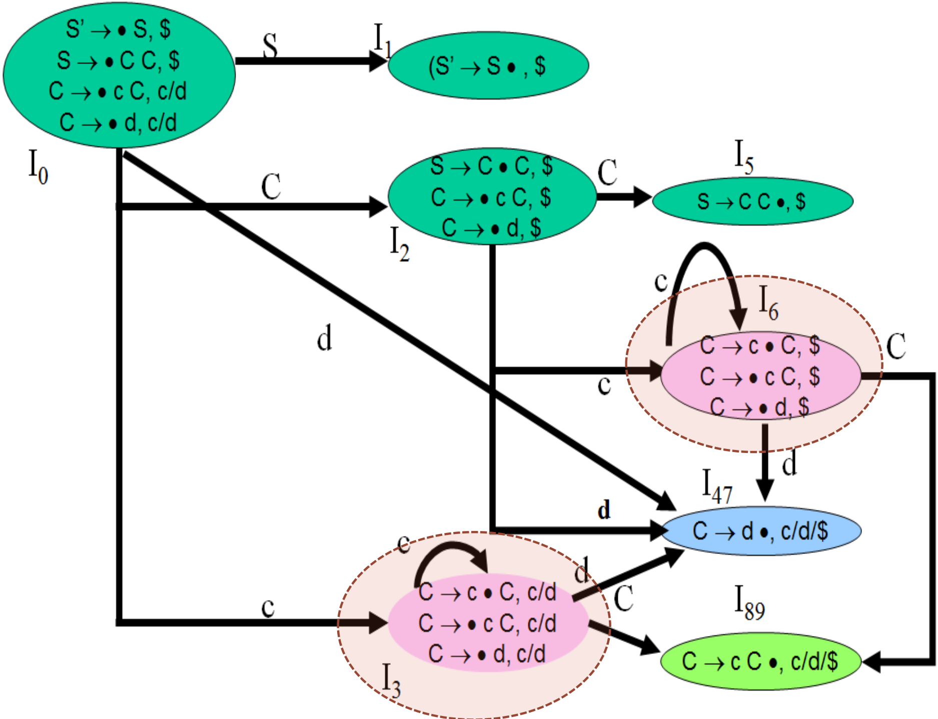
S12 = {C ->c.C; C -> .cC; C -> .d}

LALR(1) CONSTRUCTION:

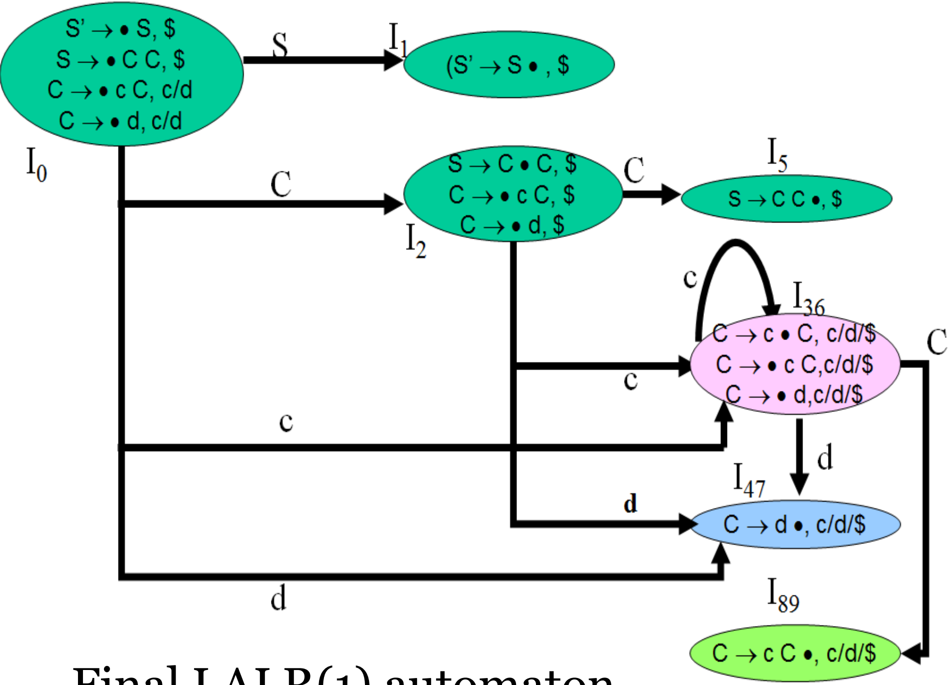
1. Construct the set of LR (1) items.
2. Merge the sets with common core together as one set, if no conflict (shift-shift or shift-reduce) arises.
3. If a conflict arises it implies that the grammar is not LALR.
4. The parsing table is constructed from the collection of merged sets of items using the same algorithm for LR (1) parsing.


S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
(S' → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
I6
C → c • C, $
C → • c C, $
C → • d, $
c
C
I9
C → cC •, $
d
I7
d
C → d •, $
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
c
I3
C
I8
C → c C •, c/d
d
I4
C → d •, c/d
d


S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I_0
S
I_1
(S' → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I_2
C
I_5
S → C C •, $
c
I_6
C → c • C, $
C → • c C, $
C → • d, $
C
c
d
I_7
C → d •, $
d
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I_3
c
C
I_{89}
C → c C •, c/d/$
d
I_4
C → d •, c/d
d



S' → • S, $
S → • C C, $
C → • c C, c/d
C → • d, c/d
I0
S
I1
(S' → S • , $
C
S → C • C, $
C → • c C, $
C → • d, $
I2
C
I5
S → C C •, $
c
I6
C → c • C, $
C → • c C, $
C → • d, $
C
d
c
d
I47
d
C → d •, c/d/$
c
C → c • C, c/d
C → • c C, c/d
C → • d, c/d
I3
d
C
I89
C → c C •, c/d/$
c



Final LALR(1) automaton

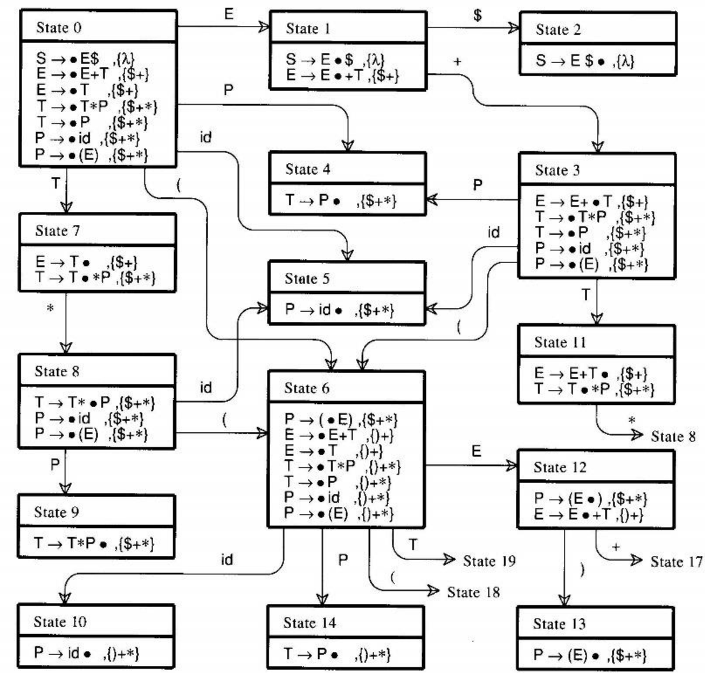
S→E $

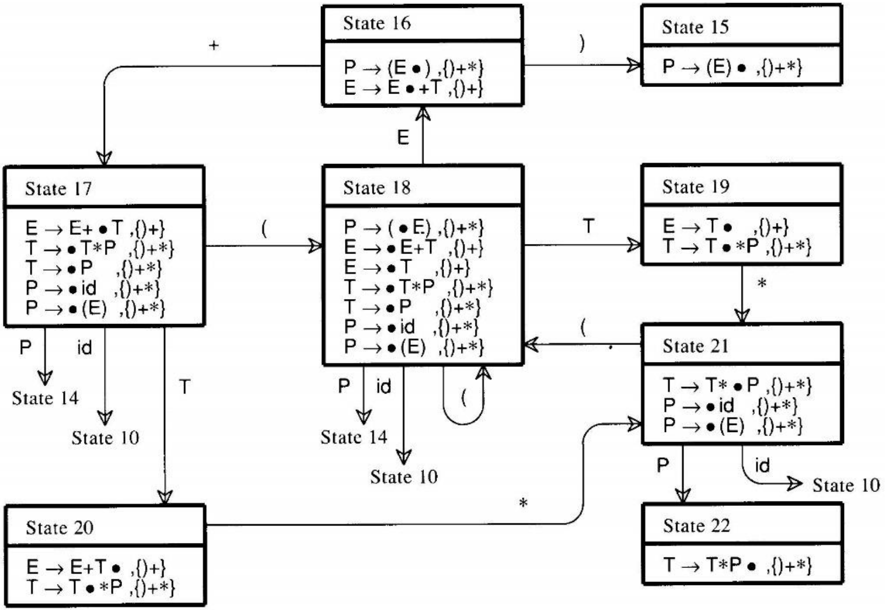
E→E + T | T

T→T * P | P

P→id | (E)

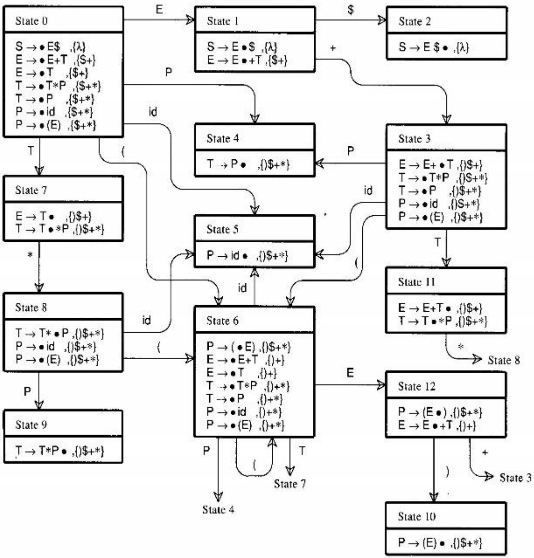
LR(1) automaton





State 16
P → (E •) ,{)+*}
E → E •+T ,{)+}
State 15
P → (E) • ,{)+*}
+
)
E
State 17
E → E+ •T ,{)+}
T → •T*P ,{)+*}
T → •P ,{)+*}
P → •id ,{)+*}
P → •(E) ,{)+*}
State 18
P → (•E) ,{)+*}
E → •E+T ,{)+}
E → •T ,{)+}
T → •T*P ,{)+*}
T → •P ,{)+*}
P → •id ,{)+*}
P → •(E) ,{)+*}
State 19
E → T • ,{)+}
T → T •*P ,{)+*}
(
T
*
State 21
T → T* •P ,{)+*}
P → •id ,{)+*}
P → •(E) ,{)+*}
P
id
State 14
State 10
T
State 14
State 10
State 10
State 20
E → E+T • ,{)+}
T → T •*P ,{)+*}
State 22
T → T*P • ,{)+*}

LALR(1) automaton for expressions



$S \rightarrow E\ \$$

$E \rightarrow E + T \mid T$

$T \rightarrow T * P \mid P$

$P \rightarrow id \mid (E)$

LALR(1) Cognate State	LR(1) States with Common Core
State 0	State 0
State 1	State 1
State 2	State 2
State 3	State 3, State 17
State 4	State 4, State 14
State 5	State 5, State 10
State 6	State 6, State 18
State 7	State 7, State 19
State 8	State 8, State 21
State 9	State 9, State 22
State 10	State 13, State 15
State 11	State 11, State 20
State 12	State 12, State 16

COMPARISON OF LR (1) AND LALR:

1. If LR (1) has shift-reduce conflict then LALR will also have it.
2. If LR (1) does not have shift-reduce conflict LALR will also not have it.
3. Any shift-reduce conflict which can be removed by LR (1) can also be removed by LALR.
4. For cases where there are no common cores SLR and LALR produce same parsing tables.

LALR: Example of R-R Conflict

S'-> S
S -> a A d
S -> b B d
S -> a B e
S -> b A e
A -> c
B -> c

I0 : S'-> .S , $
S-> . aAd, $
S-> . bBd, $
S-> . aBe, $
S-> . bAe, $

I1: S'-> S ., $

I2: S-> a . Ad, $
S-> a . Be, $
A-> .c, d
B->.c, e

I3: S-> b . Bd, $
S-> b . Ae, $
A->.c, e
B->.c,d

I4: S->aA.d, $

I5: S-> aB.e,$

<u>I6: A->c. , d ; B->c. , e</u>

I7: S->bB.d, $

I8: S->bA.e, $

<u>I9: B->c. ,d ; A->c. , e</u>

I10: S->aAd. , $

I11: S->aBe., $

I12: S->bBd., $

I13: S->aBe., $

The underlined items are problematic.

The LR (1) Parsing Table

	a…………………	d	e	
I_1 I_2 . . .		. . . .	. . . .	
I_6	…………………	**r6**	**r7**	
… . . .		. . . .	. . . .	
I_9	……………………	**r7**	**r6**	
.				

The LALR Parsing table

LR(1) Parsing table on reduction to the LALR parsing table

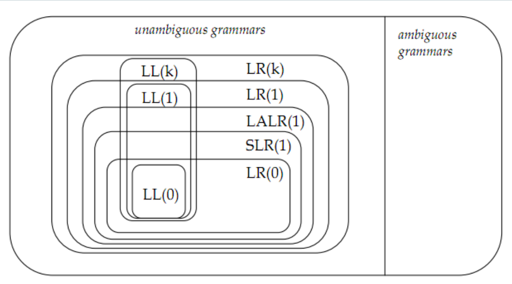
	a	d	e	
I_1 I_2 : . .		. : : .	. : : .	
I_{69}	....	**r6/r7**	**r7/r6**	
.. : : .		: . : .	.. : : .	
~~I_9~~	~~........................~~	~~r7~~	~~r6~~	
.				

Conclusion

- So, we find that the LALR may introduce reduce-reduce conflict whereas the corresponding LR (1) counterpart was void of it. This shows LALR is less powerful than LR (1).

$$LR(0) \subset SLR(1) \subset LALR(1) \subset LR(1).$$

$$LL(1) \not\subseteq SLR(1), \text{ but } LL(1) \subset LR(1).$$



Yacc/Bison

Yet **A**nother **C**ompiler **C**ompiler:

LALR(1) parser generator.

Grammar rules written in a specification (`.y`) file, analogous to the regular definitions in a `lex` specification file.

Yacc translates the specifications into a parsing function `yyparse()`.

```
          yacc
spec.y     →   spec.tab.c
```

`yyparse()` calls `yylex()` whenever input tokens need to be consumed.

`bison`: GNU variant of `yacc`.

Yacc/Bison input file

```
%{
  ... C headers (#include)
%}

... Yacc declarations:
        %token ...
%%
... Grammar rules with actions:

Expr:   Expr TOK_PLUS Expr
     |  Expr TOK_MINUS Expr
     ;
%%
... C support functions
```

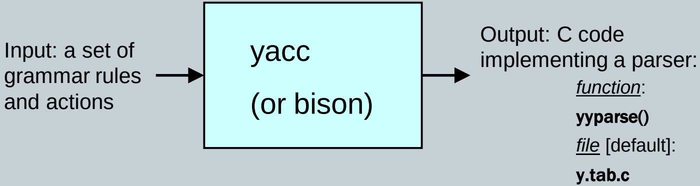
Plan for Wednesday

Theory: Old exam exercise
Practice: Crash course Flex/Bison

Crash course Yacc/Bison

Parser generator:

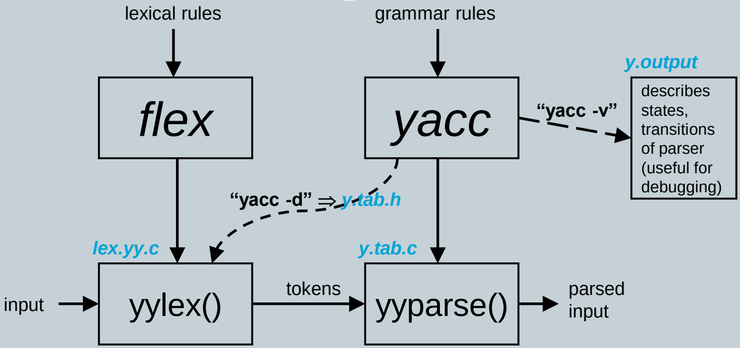
- Takes a specification for a context-free grammar.
- Produces code for a parser.



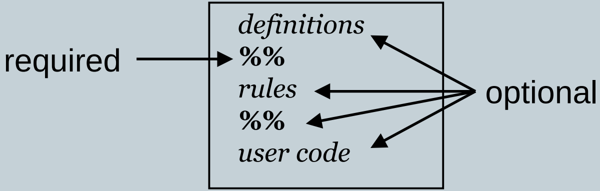
Scanner-Parser interaction

- Parser assumes the existence of the scanner **`int yylex()`**
- Scanner:
  - return value indicates the type of token found;
  - other values communicated to the parser using **`yytext`**, **`yylval`**
- Yacc/Bison determines integer representations for tokens:
  - Communicated to scanner in file **y.tab.h**
    - use "**yacc -d**" to produce **y.tab.h**
  - Token encodings:
    - "end of file" represented by '0' (zero);
    - a character literal: its ASCII value;
    - other tokens: assigned numbers ≥ 256.

Using Yacc


lexical rules
grammar rules
flex
yacc
y.output
"yacc -v"
describes states, transitions of parser (useful for debugging)
"yacc -d" ⇒ y.tab.h
lex.yy.c
y.tab.c
input
yylex()
tokens
yyparse()
parsed input

yacc: input format



Definitions section

- Information about tokens:
  - token names:
    - declared using `%token`
    - single-character tokens don't have to be declared
- start symbol of grammar, using `%start` [optional]
- operator info:
    - precedence, associativity
- stuff to be copied verbatim into the output (e.g., declarations, `#includes`): enclosed in `%{ … }%`

Rules section

Grammar production		Yacc/Bison rule
$A \to B_1\ B_2\ \ldots\ B_m$ $A \to C_1\ C_2\ \ldots\ C_n$ $A \to D_1\ D_2\ \ldots\ D_k$	➡	`A : B1 B2 … Bm` `	C1 C2 … Cn` `	D1 D2 … Dk` ` ;`

- The RHS can have arbitrary actions, within `{ … }`:

```
A : B1 { printf("after B1\n"); x = 0; } B2 { x++; } B3
```

- Left-recursion more efficient than right-recursion:
  - `A : A x | …`   rather than   `A : x A | …`

C Section

- At a minimum, provide yyerror and main functions

```
int yyerror(char *errmsg) {
  fprintf(stderr, "%s\n", errmsg);
  exit(EXIT_FAILURE);
}

int main(){
  yyparse();
  return EXIT_SUCCESS;
}
```

Error Messages

- On finding an error, the parser calls a function

```
void yyerror(char *s); /* s points to error msg */
```

  - user-supplied, prints out error message.
  - In this course, we abort in `yyerror()`.

- More informative error messages:
  - `int yychar`: token number of token causing the error.
  - user program must keep track of line numbers, as well as any additional info desired.

Conflicts

- Conflicts arise when there is more than one way to proceed with parsing.

- Two types:
  - shift-reduce [default action: *shift*]
  - reduce-reduce [default: *reduce with the first rule listed*]

- Identifying conflicts:
  - use **y.output** to identify reasons for the conflict (yacc/bison –v).

Specifying Operator Properties

- Binary operators: **%left, %right, %nonassoc**:

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

Operators in the same group have the same precedence

Across groups, precedence increases going down.

- Unary operators: **%prec**
  - Changes the precedence of a rule to be that of the token specified:

```
%left '+'  '-'
%left  '*'  '/'
Expr: expr '+' expr
        | '-' expr %prec '*'
        | …
```

The rule for unary '–' has the same (high) precedence as '*'

Debugging the Parser

To trace the shift/reduce actions of the parser:

- when compiling:
  - `#define YYDEBUG`

- at runtime:
  - `set yydebug = 1    /* extern int yydebug; */`

Synthesized Attributes

Each nonterminal can “return” a value:

- The return value for a nonterminal X is “returned” to a rule that has X in its body, e.g.:

  A : … X …

  value “returned” by X

  X : …

- *This is different from the token values produced by the scanner!*

Attribute return values

- To access the value returned by the i^{th} symbol in a RHS, use **`$i`**
  - an action occurring in a rhs counts as a symbol!

$$\texttt{Sum : } \underbrace{\texttt{Term}}_{1}\ \underbrace{\texttt{Addop}}_{2}\ \underbrace{\texttt{\{op=\$2;\}}}_{3}\ \underbrace{\texttt{Term}}_{4}\ \underbrace{\texttt{\{ \$\$ = do\_op(\$1,op,\$4); \}}}_{5}$$

- The return value of a rule is **`$$`**
  - by default, the value of a rule is the value of its first symbol **`$1`**.
  - Default type for nonterminal return values is **`int`**.

Using other types

- **YYSTYPE** determines the data type of the values returned by the lexer.

- If the lexer returns different types depending on what is read, include a union:

```
%union {          // C feature, allows one memory area to
  char cval;      // be interpreted in different ways.
  char *sval;     // For Yacc/Bison, this is the type of yylval
  int ival;
}
```

- The union is placed at the top of your Yacc/Bison file (in the definitions section)

- Tokens and non-terminals should be defined using the union

Using other types - Example

- Definitions in Yacc/Bison file:

```
%union {
  float fval;
  int ival;
}
%token <ival> NUMBER
%token <fval> FNUMBER
%type  <fval> expression, term, factor
```

terminals

nonterminals

- Use union in rules in (f)lex file:

```
{DIGIT}+ { yylval.ival = atoi(yytext); return NUMBER;}
```

Summary of steps

The actual language-design process using Yacc/Bison, from grammar specification to a working compiler or interpreter, has these parts:

1. Formally specify the grammar in a form recognized by Bison (i.e., machine-readable BNF). For each grammatical rule in the language, describe the action that is to be taken when an instance of that rule is recognized. The action is described by a sequence of C statements.

2. Write a lexical analyzer to process input and pass tokens to the parser.

3. Write a controlling function (main) that calls the Bison-produced parser.

4. Write error-reporting routines.

Exercise: calculator

```
%{
#include "global.h"
#include "calc.tab.h"
#include <stdlib.h>

%}

white       [ \t]+

digit       [0-9]
integer     {digit}+
exponent    [eE][+-]?{integer}

real        {integer}("."{integer})?{exponent}?

%%

{white}     { /* We ignore white characters */ }
{real}      { yylval=atof(yytext); return(NUMBER); }
"\n"        { return(NEWLINE); }
.           { return *yytext;  }
```

Exercise: calculator

```
%token NUMBER NEWLINE
%left '-' '+'
%right NEG        /* negation--unary minus */

%start Input

%%  /****** grammar rules section ********/

Input : /* empty */
      | Input Line
      ;

Line  : NEWLINE
      | Expr NEWLINE { printf("Result: %f\n",$1); }
      ;

Expr  : Expr '+' Expr        { $$ = $1 + $3;      }
      | Expr '-' Expr        { $$ = $1 - $3;      }
      | '-' Expr  %prec NEG  { $$ = -$2;          }
      | '(' Expr ')'         { $$ = $2;           }
      | NUMBER
      ;
```

Exercise: calculator

- Extend the calculator such that it also accepts multiplication, division, exponentiation.
- Introduce variables, assignments, and expressions containing variables:
  - `let x = 5`