CA4003 - Compiler Construction

Bottom Up Parsing

David Sinclair

Bottom Up Parsing

LL(1) parsers have enjoyed a bit of a revival thanks to JavaCC.

LL(k) parsers must predict which production rule to use having seen the first k tokens of the right-hand side.

- LL(k) stands for "left to right parse, leftmost derivation using k token lookahead".

A more powerful approach is LR(k) parsing.

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

Many programming languages require an LR(1) grammar and parser.

- LR(1) is the norm since $k > 1$ generates lots of internal states.
- Complier generators such as SableCC, Yacc and Bison support LR grammars.

LR Parse Engine

An LR parser has a *stack* and an *input*.
The first k tokens of the *input* are the *lookahead*.

The LR engine *shifts* tokens from the *input* to the *stack* and when the tokens on top of the *stack* match the righthand-side of a production rule, pop the tokens off the *stack* and push the lefthand-side of the production rule onto the *stack*. For example, if the *stack* contains ABC and there is a production rule .
$X \rightarrow ABC$, then pop C, B and A and push X onto the *stack*.

The *stack* is initially empty and when the parser *shifts* the end-of-file marker, \$, and the *input* has a valid derivation, the parser enters the *accepting state*.

- Otherwise the parser ends in an *error state*.

Valid LR Parse Actions

The valid actions of an LR parse engine are:

shift(n)	Advance input by one token and push state n onto the stack.
reduce(k)	Pop of the number of symbols on the righthand-side of rule k. Let X be the lefthand-side symbol of rule k. In the state now on top of the stack, look up X in the state table to get "goto n" Push state n onto the stack.
accept	Stop parsing and report success.
error	Stop parsing and report failure.

What is the *state table*?

LR Parsing Engine and a DFA

The *state table* is a 2 dimensional table (states $\times$ symbols).

Each entry in the *state table* contains an action.
This table is implemented as a DFA.

- A DFA is not powerful enough to parse a context-free grammar, but it can manage the stack in the LR parse engine.
- The edges of the DFA are the symbols and the states represent the stack.

Consider the following grammar with S' as the start state.

0	$S' \rightarrow S\$$	4	$E \rightarrow \text{id}$		
1	$S \rightarrow S\ ;\ S$	5	$E \rightarrow \text{num}$	8	$L \rightarrow E$
2	$S \rightarrow \text{id} := E$	6	$E \rightarrow E + E$	9	$L \rightarrow L\ ,\ E$
3	$S \rightarrow \text{print}\ (\ L\ )$	7	$E \rightarrow (\ S\ ,\ E\ )$		

LR Parsing Table

	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				

LR Parsing Example

Let's parse the following sentence using the previous LR parse table.

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

The numeric subscripts in the stack are the DFA states.

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 := num
$_1 S_2$	`; b := c + ( d := 5 + 6 , d ) $`	*shift*
$_1 S_2$;$_3$	`b := c + ( d := 5 + 6 , d ) $`	*shift*

LR Parsing Example [2]

Stack	Input	Action
$_1 S_2$;$_3$ 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*

LR Parsing Example [3]

Stack	Input	Action
$_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11} +_{16} (_8\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ \mathrm{num}_{10}$	, d) $	*reduce* $E \rightarrow \mathrm{num}$
$_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11} +_{16} (_8\ \mathrm{id}_4\ :=_6\ E_{11}\ +_{16}\ E_{17}$	, d) $	*reduce* $E \rightarrow E + E$
$_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11} +_{16} (_8\ \mathrm{id}_4\ :=_6\ E_{11}$	, d) $	*reduce* $S \rightarrow \mathrm{id} := E$
$_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11} +_{16} (_8\ S_{12}$	, d) $	*shift*
$_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11} +_{16} (_8\ S_{12}\ ,_{18}$	d) $	*shift*
$_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11} +_{16} (_8\ S_{12}\ ,_{18}\ \mathrm{id}_{20}$	) $	*reduce* $E \rightarrow \mathrm{id}$
$_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11} +_{16} (_8\ S_{12}\ ,_{18}\ E_{21}$	) $	*shift*
$_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11} +_{16} (_8\ S_{12}\ ,_{18}\ E_{21}\ )_{22}$	$	*reduce* $E \rightarrow (S, E)$
$_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11} +_{16}\ E_{17}$	$	*reduce* $E \rightarrow E + E$
$_1\ S_2\ ;_3\ \mathrm{id}_4\ :=_6\ E_{11}$	$	*reduce* $S \rightarrow \mathrm{id} := E$
$_1\ S_2\ ;_3\ S_5$	$	*reduce* $S \rightarrow S; S$
$_1\ S_2$	$	*accept*

LR(0) Parsing

So the key to LR parsing is generating the LR parse table. Once you have this table, parsing a sentence becomes mechanical.

Before looking at how to generate a LR(1) parse table, we will start with a LR(0) parse table.

- LR(0) makes it decisions by solely looking at the stack. There is no lookahead.
- Many of the ideas we will develop for LR(0) can be used for other LR parsers.

Consider the following grammar:

0 $S' \rightarrow S\$$
1 $S \rightarrow (\ L\ )$
2 $S \rightarrow x$
3 $L \rightarrow S$
4 $L \rightarrow L\ ,\ S$

LR(0) Parsing [2]

Initially we will have an empty *stack* and the *input* will have a complete sentence S followed by the end-of-input marker, \$. We denote this as:

$$S' \rightarrow .S\$$$

where the dot indicates the current parser position.
A production rule combined with a parser position, dot, is called a *LR(0) item*.
In this state, an input that can be parsed not only begins with S but also with any any possible right-hand side of an S-production.

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

We have labelled this state as state 1. A state is a set of *LR(0) items*

LR(0) Parsing [3]

Shift actions
If we shift an x is state 1, only the second production rule is affected. We can ignore the other rules. The effect of shifting an x symbol will be to move the dot past the x. This will give us a new state.

$$\boxed{S \rightarrow x.}^{2}$$

If we shift a (symbol we will obviously get $S \rightarrow (.L)$, but we also need to consider any input that can be derived from L, such as $L \rightarrow .L$, S and $L \rightarrow .S$.
Since there is a dot just before the S symbol we also need to include anything that S can produce.

LR(0) Parsing [4]

This will give us a new state:

$$\boxed{\begin{array}{l} S \rightarrow (\ .L\ ) \\ L \rightarrow .L\ ,\ S \\ L \rightarrow .S \\ S \rightarrow .\ (\ L\ ) \\ S \rightarrow .x \end{array}}^{3}$$

Goto actions

When we perform a reduce action what we have done is popped a sequence of symbols of the stack that match the RHS of a production rule. Now we need to figure out goto action for the non-terminal on the LHS of the production rule.

In state 1 if we parse a string of tokens derived from S the effect is to move the dot past the S symbol. This gives us a new state:

$$\boxed{S \rightarrow S.\$}^{4}$$

LR(0) Parsing [5]

Reduce actions

In state 2 of our example we have $S \rightarrow x.$, where the dot is at the end of an *item*. This means that the top of the stack must contain the complete RHS of a production rule and is ready to be reduced. In our case the production rule is $S \rightarrow x$, and the x can be reduced to S. We already have figured out what state the S will go to.

Thus the basic operations we have performed are:

`Closure(I)` which adds more items to a set of items, I, when there is a dot to the left of a non-termial, and

`Goto(I,X)` which moves the dot past the symbol x in all items.

LR(0) Parsing [6]

```
Closure(I) =
  repeat
    for any item A → α.Xβ in I
      for any production X → γ
        I ← I ∪ {X → .γ}
  until I does not change
  return I
```

```
Goto(I, X) =
  Set J to the empty set
  for any item A → α.Xβ in I
    add A → αX.β to J
  return Closure(J)
```

Let T be the set of states seen so far and E be the set of shift or goto edges found so far.

```
Set T to {Closure({S' → S$})}
Set E to the empty set
repeat
  for each state I in T
    for each item A → α.Xβ in I
      let J be Goto(I, X)
      T ← T ∪ {J}
      E ← E ∪ {I --X--> J}
until E and T do not change in this repeat iteration
```


LR(0) Parsing [7]

The set of **reduce** actions is calculated by:

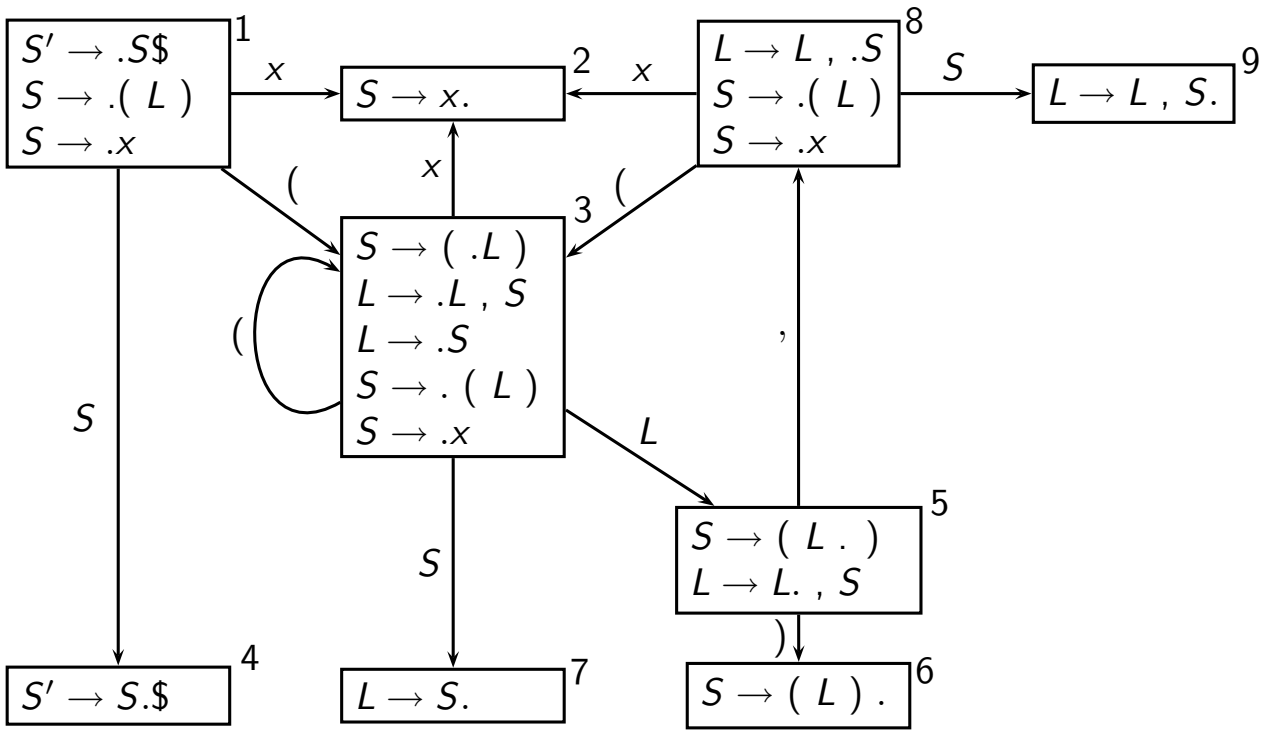
```
Set R to the empty set
for each state I in T
  for each item A → α. in I
    R ← R ∪ {(I, A → α)}
```

To build the LR(0) state table:

- For each edge $I \xrightarrow{X} J$ where X is a terminal, we put a **shift** J action at (I, X).
- For each edge $I \xrightarrow{X} J$ where X is a non-terminal, we put a **goto** J action at (I, X).
- For each state containing $S' \rightarrow S.\$$, we put an **accept** action at $(I, \$x)$.
- For each state containing $A \rightarrow \gamma.$ (rule n with a dot), we put a **reduce** n action at (I, Y) for every token Y.
- All remaining states have **error** actions.

LR(0) Example

Back to our example grammar, we get the following LR(0) states.



LR(0) Example [2]

And this gives rise to the following LR(0) parsing table.

	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

Another Example

Consider the following grammar.

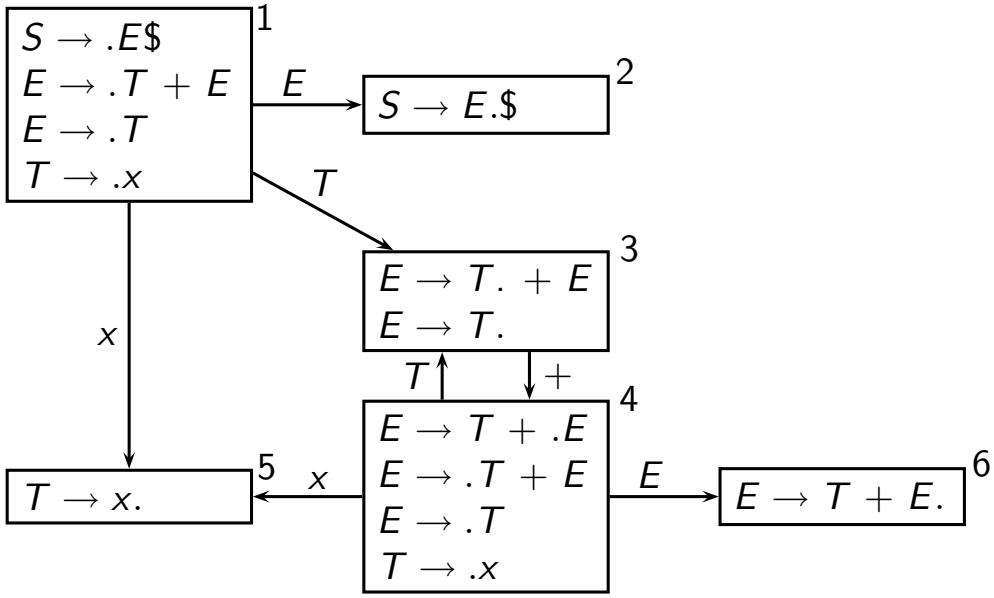
0 $S \rightarrow E\$$

1 $E \rightarrow T + E$

2 $E \rightarrow T$

3 $T \rightarrow x$

The LR(0) states are:



Another Example [2]

The corresponding LR(0) parsing table is:

	x	+	$	E	T
1	s5			g2	g3
2			a		
3	r2	s4,r2	r2		
4	s5			g6	g3
5	r3	r3	r3		
6	r1	r1	r1		

We have a conflict in state 3 on the + symbol.

This is not an LR(0) grammar since it cannot be parsed by an LR(0) parser.

Conflicts

There are 2 types of conflicts that can occur in a parsing table:

shift-reduce
: This is where the same state has two actions, a **shift** action and a **reduce** action. Some compiler generators may resolve this by favouring the **shift** action over the **reduce** action. Sometimes this is acceptable but you should generally try to avoid *shift-reduce* conflicts.

reduce-reduce
: This is where the same state has two **reduce** actions. Which **reduce** action should be used? *Reduce-reduce* should always be avoided. They are the sign of an ill-defined grammar.

SLR Parsers

SLR stands for *simple LR*, and surprisingly it is more powerful than an LR(0) parser.

The parser construction for an SLR is nearly identical to an LR(0) parser except that the generation of **reduce** actions depends upon the `FOLLOW` set.

The algorithm for **reduce** actions in an SLR parsing table.

Set R to the empty set
for each state I in T
  for each item $A \rightarrow \alpha.$ in I
    for each token in `FOLLOW`(A)
      $R \leftarrow R \cup \{(I, X, A \rightarrow \alpha)\}$

SLR Parsers [2]

The corresponding SLR parsing table is:

	x	+	$	E	T
1	s5			g2	g3
2			a		
3		s4	r2		
4	s5			g6	g3
5		r3	r3		
6			r1		

prod. rule 2: $E \rightarrow T$
$\texttt{FOLLOW(E)} = \{\$\}$

prod. rule 3: $T \rightarrow x$
$\texttt{FOLLOW(T)} = \{+, \$\}$

- SLR parsing tables have fewer **reduce** actions than LR(0) parsing tables.
- Many useful programming languages belong to the class of SLR grammars.

LR(1) Parsing

The LR(1) parsing algorithm is more powerful than the LR(0) and SLR parsing algorithms.

The algorithm for constructing an LR(1) parsing table is similar to the LR(0) algorithm, except:

- The concept of an *item* in LR(1) is expanded to a *lookahead symbol*.
- An LR(1) item is $(A \rightarrow \alpha.\beta, x)$ where x is the lookahead symbol.
  - The sequence α is on the top of the stack.
  - The input starts with a string that can be derived from βx.
- An LR(1) state is a set of LR(1) items that are constructed using versions of `Closure` and `Goto` that incorporate the lookahead symbol.

LR(1) Parsing [2]

```
Closure(I) =
  repeat
    for any item (A → α.Xβ, z) in I
      for any production X → γ
        for any w ∈ FIRST(βz)
          I ← I ∪ {(X → .γ, w)}
  until I does not change
  return I
```

```
Goto(I, X) =
  set J to the empty set
  for any item (A → α.Xβ, z) in I
    J ← J ∪ (A → αX.β, z)
  return Closure(J)
```


LR(1) Parsing [3]

The set of **reduce** actions is calculated by:

```
Set R to the empty set
for each state I in T
  foreach item (A → α., z) in I
    R ← R ∪ {(I, z, A → α)}
```

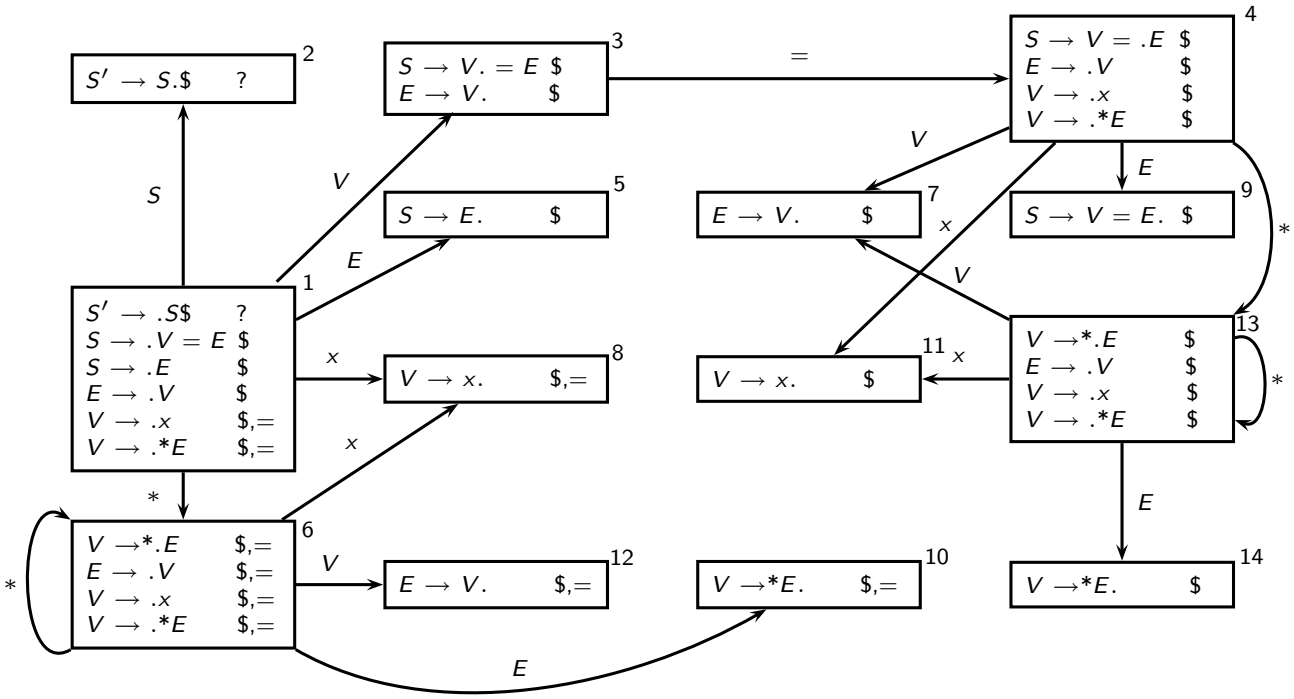
where the action $(I, z, A \rightarrow \alpha)$ means that in state I, on lookahead symbol z, apply reduction $A \rightarrow \alpha$.

Consider the following grammar which has pointer dereferencing (as in C):

0 $S' \rightarrow S\$$
1 $S \rightarrow V = E$
2 $S \rightarrow E$
3 $E \rightarrow V$
4 $V \rightarrow x$
5 $V \rightarrow * E$

LR(1) Parsing [4]

The LR(1) states for this grammar are:



LR(1) Parsing [5]

Building the LR(1) parsing table from the LR(1) state graph:

- If the dot is at the end of a production rule then a **reduce** action is placed in the parsing table at the row corresponding to the state and the column corresponding to the *lookahead* symbol.
- If the dot is to the left of a terminal or non-terminal symbol, the corresponding column for that state contains a **shift** or **goto** action respectively (the same as you would do with an LR(0) table).

LR(1) Parsing [6]

The corresponding LR(1) parsing table is:

	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s11	s13				g9	g7
5				r2			
6	s8	s6				g10	g12
7				r3			
8			r4	r4			
9				r1			
10			r5	r5			
11				r4			
12			r3	r3			
13	s11	s13				g14	g7
14				r5			

LALR(1) Parsing

AN LR(1) parsing table can have a large number of states. COBOL-85, for example, requires over 2,000,000 states.

An LALR(1) parser, *lookahead LR(1) parser*, can reduce the number of states by merging states that only differ by their lookahead sets.

- In our LR(1) example we could merge:
  - states 6 and 13
  - states 7 and 12
  - states 8 and 11
  - states 10 and 14

An LALR(1) parser for COBOL-85 only requires 1,720 states approximately.

LALR(1) Parsing [2]

The corresponding LALR(1) parsing table for our LR(1) example is:

	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s8	s6				g9	g7
5				r2			
6	s8	s6				g10	g7
7			r3	r3			
8			r4	r4			
9				r1			
10			r5	r5			

Sometimes, an LALR(1) parsing table may have reduce-reduce conflicts while the corresponding LR(1) parsing table does not.

Problems with if...then..else

Many programming language have an if...then..else construct.
$S \rightarrow$ if E then S else S
$S \rightarrow$ if E then S
$S \rightarrow$ other

This allows programs such as:
`if a then if b then s1 else s2`

But how should this be interpreted?

1. `if a then {if b then s1 else s2}`
2. `if a then {if b then s1} else s2`

The problem is there is a shift-reduce conflict.

$S \rightarrow$ if E then S .	else
$S \rightarrow$ if E then S . else S	(any)

Problems with if...then..else [2]

If we choose *shifting* we get interpretation 1 (which is what most programming languages intend).
If we choose *reducing* we get interpretation 2.

A better way is to introduce additional non-terminals into the grammar, M for matched statement and U for unmatched statement.

$S \rightarrow M$
$S \rightarrow U$
$M \rightarrow$ if E then M else M
$M \rightarrow$ other
$U \rightarrow$ if E then S
$S \rightarrow$ if E then M else U

Precedence Rules

Consider the ambiguous grammar:

1 $E \rightarrow$ id
2 $E \rightarrow$ num
3 $E \rightarrow E * E$
4 $S \rightarrow E/E$
5 $E \rightarrow E + E$
6 $E \rightarrow E - E$
7 $E \rightarrow (E)$

and the expression 2*3+4.

In one of the LR(1) states, two of the items will be:

$E \rightarrow E.\ +\ E$	(any)
$E \rightarrow E\ *\ E.$	$+$

and hence a *shift-reduce* conflict.

Precedence Rules [2]

Using our example expression, when the stack is E * E and we take the shift operation, the stack will become E * E + and subsequently E * E + E. The E + E will reduce by rule 5 and then the resulting E * E will reduce by rule 3.

This is not what we want. If we want * to bind more tightly than +, we need to reduce the E * E first by rule 3 and then shift. When we subsequently get E + E on the stack we then reduce by rule 5.

If we have an expression `x - y - z` and we want to force the programmer to explicitly bracket the expression, i.e. `(x - y) - z` or `x - (y - z)` then we need to make the minus operator *non-associative*.

Precedence Rules [3]

Some compiler generators, such as Yacc and Bison, have *precedence directives*. These directives assist in resolving shift-reduce conflicts.

For example, the following series of Yacc directives,

```
precedence nonassoc EQ, NEQ;
precedence left PLUS, MINUS;
precedence left TIMES, DIV;
precedence right EXP;
```

specify EXP is right-associative and has the highest precedence. TIMES and DIV are left-associative, and the precedence is lower than EXP but higher than PLUS, MINUS, EQ and NEQ. EQ and NEQ are not associative and have the lowest precedence.

The priority of a rule is determined by the precedence of the last token occurring on the right-hand side.

Left versus right recursion

Right Recursion:

- Needed for termination in predictive parsers.
- Requires more stack space.
- Gives rise to right associative operators.

Left Recursion:

- Works fine in bottom-up parsers.
- Limits the required stack space.
- Gives rise to left associative operators.

Rule of thumb:

- Right recursion for top-down parsers.
- Left recursion for bottom-up parsers.

Syntax vs Semantics

Consider a language that has both arithmetic and boolean expressions.

$E \rightarrow \text{id} := E$
$E \rightarrow \text{id}$
$E \rightarrow E \& E$
$E \rightarrow E = E$
$E \rightarrow E + E$

An expression `z = x + 2 & y` is syntactically correct, but makes no sense.

The determination as to whether an expression makes sense, or not, is the function of the *Semantic Analysis Phase*.