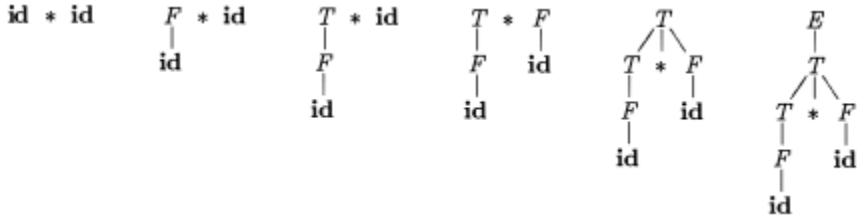
Bottom-Up Parsing

A bottom-up parser corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top). The sequence of snapshots



Key Idea

- The weakness of top-down *LL*(*k*) parsing techniques is that they must predict which production to use, having seen only the first *k* tokens in the right side.

- The more powerful techniques of bottom-up *LR*(*k*) parsing is able to postpone the decision until it has seen
  - input tokens corresponding to the *entire* right side of the production
  - and *k* lookahead tokens beyond

Bottom-up parsers are known as LR parsers. The L stands for scanning the input from left to right and R stands for producing a rightmost derivation. Most general form of bottom-up parsing is **Shift-Reduce Parsing**.

Bottom-up parsing as the process of reducing a string 'w' to the start symbol of the grammar. At each reduction step, a partial substring matching the right side of a production is replaced by the left side of the production. If that substring is chosen correctly at each step, we get the rightmost derivation in reverse order. The key decision during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

By definition, a reduction is the reverse of a step in a derivation. The goal of bottom-up parsing is therefore to construct a derivation in reverse. For the above parse the derivation will be

E => T => T * F => T * id => F * id => id * id

Consider the grammar,

$S \rightarrow aABe$

$A \rightarrow Abc|b$

$B \rightarrow d$

Consider the string w=abbcde

Right most derivation

S =>aABe
=> aAde
=>aAbcde
=> abbcde

Shift reduce

123456
abbcde => aAbcde // A→b is the handle 2
=>aAde // A → Abc is the handle 2
=>aABe // B → d is the handle 3
=> S // S → aABe is the handle 1

The trick appears to be scanning the input and find valid sentential form.

Handle

Handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step in the reverse of the rightmost derivation.

Matching substring may not be a handle always.

Formally, a **handle** of a right sentential form γ is a production A → β and a position of γ where the string β may be found and replaced by A to produce the previous right sentinel form of the rightmost derivation of γ. ie, s=>* αAw, then A → β in the position following α is the handle of αβw. The symbol 'w' to the right of the handle contains only terminal symbols.

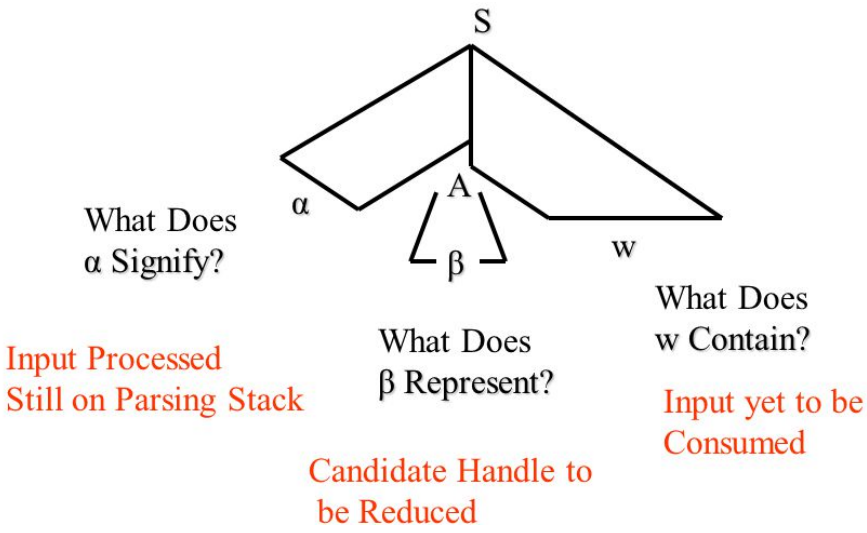
Consider the following grammar:

E → E + E | E * E | (E) | id

and a right-most derivation is as follows:

E =>E + E
=>E+ E * E
=>E + E * id3
=>E + id2 * id3
=>id1 + id2 * id3

Handles are underlined in each right sentential form.



Handle Pruning

A rightmost derivation in reverse can be obtained by "handle pruning".

If $\alpha Aw \Rightarrow_{rm} \alpha\beta w$, then $A \rightarrow \beta$ is the handle at position following α. Then reducing β to A can be considered as the process of removing children from A from the parse tree and this process is called **handle pruning**.

Two major problems of handle pruning are

1. locate the correct substring to be reduced
2. determine what production to use incase if more than one production with same substring on the right side.

Reduction made by shift-reduce parser

RIGHT-SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

Shift-reduce parser

Shift reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. As before, the handle always appears at

the top of stack just before it is identified as the handle. We use $ to mark the bottom of the stack and also the right end of the input.

During a left to right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of stack. It then reduce β to the head of the appropriate production.

There are 4 basic actions for shift-reduce parser.

1. Shift – next input symbol is shifted onto the top stack
2. Reduce – Replaces a set of grammar symbols on the top of the stack with the LHS of a production rule.
3. Accept – parser announce the successful completion of parsing. (stack top – start symbol of grammar & input buffer has only $)
4. Error – discovers that a syntax error has occurred and call error recovery routine.

Steps in parsing input string **id1 + id2 * id3** given below.

There will be total 5 shift actions and 5 reduce actions and 1 accept action. Therefore total 11 actions.

Stack	Input	Action
$	$id_1 + id_2 * id_3$	shift
id_1	$+ id_2 * id_3$	reduce by E→id
$E	$+ id_2 * id_3$	shift
$E +	$id_2 * id_3$	shift
$E + id_2	$* id_3$	reduce by E→id
$E + E	$* id_3$	shift
$E + E*	id_3	shift
$E + E * id_3	$	reduce by E→id
$E + E * E	$	reduce by E→E * E
$E + E	$	reduce by E→ E + E
$E	$	accept

The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called **viable prefixes**.

Conflicts during shift-reduce parsing

2 conflicts can be occurred during shift reduce parsing.

1. Shift/reduce conflict
   The parser knows that the stack contents and next input symbol that unable to decide whether to shift or reduce.
   Eg: stack - $ E + E
   Input buffer - * id $
   Here is a conflict that whether to reduce E + E using E or to shift start symbol.
2. Reduce/reduce conflict

Unable to decide which of the several reductions to make.
Eg; Consider the production,
Expr →id
Parameter →id

Operator Precedence Parsing

Popular shift reduce parser works only in LR grammar and it is easy to parse if the LR grammar satisfy two conditions.

1. No production right side is ϵ
2. No two adjacent non-terminals.

The grammar with the later one is called operator grammar. Operator grammar is mainly used to define the mathematical operators in the compiler.

Eg: E → E + E | E – E | E * E | E / E | (E) | -E | **id** is an operator grammar.

Eg: E → E A E | (E) | -E | id

A → + | - | * | / is not an operator grammar, because of failure in 2nd condition.

Operator precedence grammar is the only grammar that can handle ambiguous grammar.

Disadvantage of Operator Precedence Parsing

1. It is very hard to handle tokens like minus sign which has two different precedence.
2. It's not sure that parser accept exactly the desired language.
3. Only small class of LR grammars can be parsed by this technique.

But due to its simplicity, numerous compilers are using this technique for parsing arithmetic expression.

3 precedence relations

3 disjoint precedence relations are between pairs of terminals.

RELATION	MEANING
$a <\cdot b$	a "yields precedence to" b
$a \doteq b$	a "has the same precedence as" b
$a \cdot> b$	a "takes precedence over" b

Eg: * •> + or + <•*

No adjacent non-terminals appear on right side of the productions implies that no right sentential form will have two adjacent non-terminals either. Thus right sentential form write as β0a1β1…..where ai is the terminal and βi is a nonterminal or ϵ.

The intention behind precedence relations is to delimit the handle of a right sentential form with <• marking the left end, = appearing in the interior of the handle and •> marking the right end.

Consider the right sentinel form **id + id * id** for the grammar E → E + E| E * E | id, the operator precedence/relation table will be

	id	+	*	$
id		•>	•>	•>
+	<•	•>	<•	•>
*	<•	•>	•>	•>
$	<•	<•	<•	

- **id** is having highest precedence than any operator.
- + •> +, because + is having left associativity.
- $ is having least precedence when compared with any operator.
- $ $ means succesfull.

The string with the precedence relations inserted is:

$ <• **id** •> + <• **id** •> * <• **id** •> $

The handle is found by the following process.

1. Scan the string from left end until the first •> is encountered. This occurs between first id and +.
2. Then scan backwards to left over any =’s until a <• is encountered. In the above example, we can scan backwards to $.
3. The handle contains everything to the left of the first •> and to the right of the <• encountered in step a, including any intervening or surrounding non-terminals.
4. Repeat these steps until starting symbol is obtained.

Example: consider the grammar

E → E + E | E – E | E * E | E / E | (E) | -E | **id**

and input string id + id * id

solution:

$ <• **id** •> + <• **id** •> * <• **id** •> $

Handle is first **id,** so, $ E + **id** * **id** $

$ + **id** * **id** $

$ <• + <• **id** •> * <• **id** •> $, handle is **id**

$ E + E * **id** $

$ + * **id** $

$ <• + <• * <• **id** •> $, handle is **id**

$ E + E * E $

$ <• + <• * •> $, handle is E * E

$ E + E $

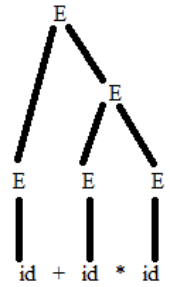
$ <• + •> $, handle is E + E

$ $

- Stack can be used for its implementation.
- Push input symbol to the stack.
- If the relation, <• or = hold between the stack top and input symbol, then the parser shift input symbol onto the stack.
- If the relation is •>, handle is found and reduction is applied.
- If no precedence relation is defined between a pair of terminals, then it is a syntactic error and go for error recovery routine.

STACK	INPUT	ACTION/REMARK
$	id + id * id $	$ <• id, push
$ id	+ id * id $	id •> + Reduce E → id, pop
$	+ id * id $	$ <• +, push
$ +	id * id $	+ <• id, push
$ + id	* id $	id •> * Reduce E → id, pop
$ +	* id $	+ <• *, push
$ + *	id $	* <• id, push
$ + * id	$	id •> $ Reduce E → id, pop
$ + *	$	* •> $ Reduce E → E * E, pop
$ +	$	+ •> $ Reduce E → E + E, pop
$	$	Accept

The parse tree will be



The input string is w$, the initial stack is $ and a table holds precedence relations between certain terminals

Algorithm:

```
set p to point to the first symbol of w$ ;
repeat forever
   if ( $ is on top of the stack and p points to $ ) then return
   else {
      let a be the topmost terminal symbol on the stack and let b be the symbol
       pointed to by p;
      if ( a <· b or a =· b ) then {          /* SHIFT */
         push b onto the stack;
         advance p to the next input symbol;
     }
     else if ( a ·> b ) then                  /* REDUCE */
         repeat pop stack
         until ( the top of stack terminal is related by <· to the terminal most
                                recently popped );
      else error();
   }
```

Example: consider the grammar id + id + id. Even though it is ambiguous, according to precedence rule and associativity, only one parse tree will generate.

Operator precedence relations from associativity and precedence

1. If operator Θ1 has higher precedence than operator Θ2, make Θ1 •> Θ2 and Θ2 <•Θ1.
2. If Θ1 and Θ2 are operators of equal precedence, then make Θ1 •> Θ2 and Θ2 •>Θ1 if operators are left associative.
3. (=) $ <• ($ <• id (<• ( ) •> $
   (<• id id •>)) •>)

	+	-	*	/	^	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
^	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			>	>
(	<	<	<	<	<	<	<	=	
)	>	>	>	>	>			>	>
$	<	<	<	<	<	<	<		

Handling unary operator

Logical negation ¬

Usually unary operators are having higher precedence than binary operators. And having right associativity also. We make ¬ •> Θ.

Operator Precedence function

If there are 'n' operators then the size of relation table is $O(n^2)$.

Operator precedence table is encoded by two precedence functions *f* & *g*.

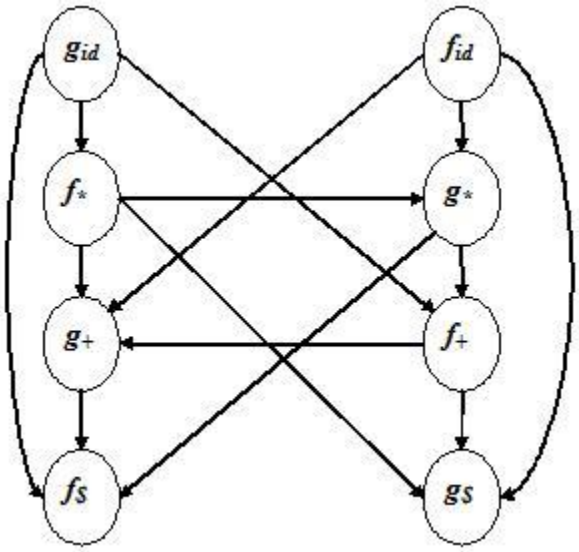
1. $f(a) < g(a)$ whenever a<•b
2. $f(a) = g(a)$ whenever a = b
3. $f(a) > g(a)$ whenever a•>b

Thus, the precedence relation between a and b can be determined by a numerical comparison between f(a) and g(b).

The disadvantage of operator precedence table is the size of the table is $O(n^2)$, where n is the number of operators. This size can be reduced to 2n by using precedence function table. The precedence function table is constructed from a graph with nodes are the function of operators based on the longest length.

For the below precedence table, the directed graph and precedence function table also given.

	id	+	*	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	



$f_{id} \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$

$g_{id} \rightarrow f_* \rightarrow g_* \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$

	+	*	id	$
f	2	4	4	0
g	1	3	5	0

Consider another precedence function table which contain most common operators.

	+	-	*	/	^	(	)	id	$
f	2	2	4	4	4	0	6	6	0
g	1	1	3	3	5	5	0	5	0

For * <• id, identify f(*) < g(id).

We can minimize the space complexity.

Drawback – it is unable to detect syntactic errors.

Eg: f(id) , g(id), from the precedence function we get the relation, 6 > 5.

Precedence function can be used when the input string is correct.

Algorithm for constructing precedence function:

Input. An operator precedence matrix.

Output. Precedence functions representing the input matrix, or an indication that none exist.

Method.

1. Create symbols f_a and g_a for each a that is a terminal or \$.
2. Partition the created symbols into as many groups as possible, in such a way that if $a \doteq b$, then f_a and g_b are in the same group. Note that we may have to put symbols in the same group even if they are not related by $\doteq$. For example, if $a \doteq b$ and $c \doteq b$, then f_a and f_c must be in the same group, since they are both in the same group as g_b. If, in addition, $c \doteq d$, then f_a and g_d are in the same group even though $a \doteq d$ may not hold.
3. Create a directed graph whose nodes are the groups found in (2). For any a and b, if $a <\cdot b$, place an edge from the group of g_b to the group of f_a. If $a \cdot> b$, place an edge from the group of f_a to that of g_b. Note that an edge or path from f_a to g_b means that $f(a)$ must exceed $g(b)$; a path from g_b to f_a means that $g(b)$ must exceed $f(a)$.
4. If the graph constructed in (3) has a cycle, then no precedence functions exist. If there are no cycles, let $f(a)$ be the length of the longest path beginning at the group of f_a; let $g(a)$ be the length of the longest path from the group of g_a. □

Error Recovery in operator precedence parsing

There are two points in the parsing process at which an operator precedence parser can discover syntactic errors:

1. If no precedence relation holds between the terminal on top of stack and the current input.
2. If a handle has been found, but there is no production with this handle as a right side.

Handling errors during reductions

As there is no production to reduce by, no semantic action are taken; a diagnostic message is printed. To determine what diagnostic should say, the routine handling case must decide what production the right side being popped "looks like".

Eg: Suppose abc is popped, and there is no production right side containing of a, b and c together with zero or more terminals. Then we might consider if deletion of one of a, b and c yields a legal right side. If there is a right side *aEcE,* we might issue the diagnostic

illegal b on line

We might also consider changing or inserting a terminal. Thus, if *abEdc* were a right side, we might issue a diagnostic

missing d on line

If +, *, -, / is reduced, it checks that non-terminals appear on both sides. If not, it issues the diagnostic

missing operand

If id is reduced, it checks that there is no non-terminal to the right or left. If there is, it can warn

missing operator

If ( ) is reduced, it checks that there is a non-terminal between the parenthesis. If not, it can say,

no expression between parenthesis

Handling Shift/Reduce errors

We may find that no relation holds between the top stack symbol and first input symbol. Suppose a and b are the two top stack symbols (b is at the top), c and d are the next two input symbols, and there is no precedence relations between b and c. To recover, we must modify the stack, input or both. We may change symbols, insert symbols onto the input or stack, or delete symbols from the input or stack. If we insert or change, we must be careful that we do not get into an infinite loop. For each blank entry in the precedence matrix we must specify an error recovery routine; the same routine could be used in several places.

Consider the precedence matrix with error entries:

	id	(	)	$
id	e3	e3	·>	·>
(	<·	<·	≐	e4
)	e3	e3	·>	·>
$	<·	<·	e2	e1

The substance of these error handling routines is as follows:

```
e1: /* called when whole expression is missing */
    insert id onto the input
    issue diagnostic: "missing operand"

e2: /* called when expression begins with a right parenthesis */
    delete ) from the input
    issue diagnostic: "unbalanced right parenthesis"

e3: /* called when id or ) is followed by id or ( */
    insert + onto the input
    issue diagnostic: "missing operator"

e4: /* called when expression ends with a left parenthesis */
    pop ( from the stack
    issue diagnostic: "missing right parenthesis"
```

Consider error handling for input **id** +). The first action taken is to shift **id**, reduce it to *E* and then shift +.

STACK	INPUT
$ E+	) $

Since + •>) a reduction is called for, and handle is +. Error handler issues *missing operand* and does the reduction anyway.

STACK	INPUT
$ E	) $

There is no precedence relation between $ and), e2 is the diagnostic routine *unbalanced right parenthesis* to be printed and removes the right parenthesis from the input.

STACK	INPUT
$ E	$

LR parsers

Advantages

1. LR parsers can be constructed to recognize virtually all programming language constructs for which CFG can be written.
2. LR parser is the most general nonbacktracking shift reduce parsing method known.
3. The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

4. LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

Drawback

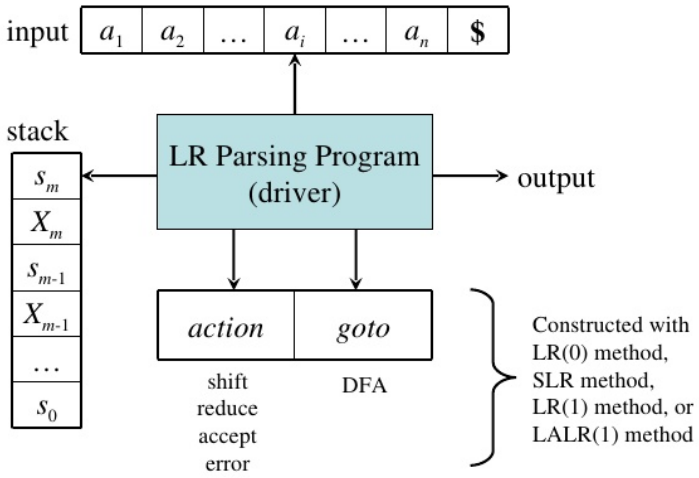
1. Too much work is required to construct an LR parser by hand for a typical programming language grammar.

3 techniques for constructing an LR parsing table for a grammar.

1. Simple LR (SLR) – easy to implement, but less powerful of the three. It may fail to produce a parsing table for certain grammars on which other methods succeed.
2. Canonical LR – most powerful and most expensive.
3. Look ahead LR (LALR) – intermediate in power and cost between the other two. LALR method will work on most programming language grammars and with some effort can be implemented effectively.

LR Parsing Algorithm

It consists of an input, an output, a stack, a driver program and a parsing table that has two parts (action and goto). The driver program is the same for all LR parsers; only the parsing table changes from one parser to another. The parsing program reads characters from an input buffer one at a time. The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2….X_ms_m$, where sm is on top. Each X_i is a grammar symbol and each si is a symbol called a *state*. Each state symbol summarizes the information contained in the stack below it, and the combination of the start symbol on top of the stack and the current input symbol are used to index the parsing table and determine the shift reduce parsing decisions.



The parsing table consists of two parts, a parsing action function *action* and a goto function *goto*.

The driver program consults action [s_m, a_i], sm, the state currently on top of stack and a_i the current input. The parsing action table entry for the state s_m and input a_i, which have four values;

1. Shift s, where s is a state
2. Reduce by a grammar production A → β
3. Accept
4. Error

The function goto takes a state and grammar symbol as arguments and produces a state. The goto function of a parsing table constructed from a grammar G using SLR, canonical LR or LALR method is the transition function of a deterministic finite automaton that recognizes the viable prefixes of G.

Example : consider the 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***

	Action Table						Goto Table		
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			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

si means shift and stack state i
rj means reduce by production numbered j
acc means accept
blank means error

Example:
Consider the input **id * id + id,** the sequence of stack and input contents is

stack	**input**	**action**	**output**
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

LR parsing program code

```
set ip to point to the first symbol of w$;
repeat forever begin
    let s be the state on top of the stack and
        a the symbol pointed to by ip;
    if action[s, a] = shift s' then begin
        push a then s' on top of the stack;
        advance ip to the next input symbol
    end
    else if action[s, a] = reduce A → β then begin
        pop 2*|β| symbols off the stack;
        let s' be the state now on top of the stack;
        push A then goto[s', A] on top of the stack;
        output the production A → β
    end
    else if action[s, a] = accept then
        return
    else error()
end
```

Consider the input **id + (id * id)** for the above parse table.

STACK	INPUT	ACTIONS
0	id+(id*id)$	Shift 5
0id5	+(id*id)$	Reduce r6, F → id
0F3	+(id*id)$	Reduce r4, T → F
0T2	+(id*id)$	Reduce r2, E → T
0E1	+(id*id)$	Shift 6
0E1+6	(id*id)$	Shift 4
0E1+6(4	id*id)$	Shift 5
0E1+6(4id5	*id)$	Reduce r6, F → id
0E1+6(4F3	*id)$	Reduce r4, T → F
0E1+6(4T2	*id)$	Shift 7
0E1+6(4T2*7	id)$	Shift 5
0E1+6(4T2*7id5	)$	Reduce r6, F → id
0E1+6(4T2*7F10	)$	Reduce r3, T → T * F
0E1+6(4T2	)$	Reduce r2, E → T
0E1+6(4E8	)$	Shift 11
0E1+6(4E8)11	$	Reduce r5, F → (E)
0E1+6F3	$	Reduce r4, T → F
0E1+6T9	$	Reduce r1, E → E + T
0E1	$	Acc

Difference between LL and LR grammar

A grammar for which we can construct a parsing table is said to be an LR grammar. In order for a grammar to be LR it is sufficient that a left-to-right shift reduce parser be able to recognize handles when they appear on top of stack. An LR parser does not have to scan the entire stack to know when the handle appears on top. It is a remarkable fact that if it is possible to recognize a handle knowing only the grammar symbols on the stack, then there is a finite automaton that can, by reading the grammar symbol on the stack from top to bottom, determine that handle, if any, is on the top of stack. The goto function of an LR parsing table is essentially such a finite automation.

A grammar that can be parsed by an LR parser examining upto *k* input symbols on each move is called an *LR(k)* grammar.

For a grammar to be LR(k), we must be able be recognize the occurrence of the right side of a production, having seen all of what is derived from that right side with k input symbols of lookahead. This requirement is far less stringent than that for LL(k) grammars where we must be able to recognize the use of a production seeing only the first k symbols of what its right side derives. Thus, LR grammars are describe more languages than LL grammars.

Constructing SLR parsing Table

How to construct LR parsing table from a grammar?

3 methods.

1. SLR method
2. Canonical LR method
3. LALR method

SLR is the weakest of the three in terms of the number of grammars for which it succeeds but is the easiest to implement. LR parser using an LSR parsing table is an SLR parser. A grammar for which an SLR parser can be constructed is said to be an SLR grammar.

An LR(0) item of a grammar G is a production G with a dot at some position of the right side. The production A → XYZ yields the four items

A → • XYZ

A → X•YZ

A → XY•Z

A → XYZ•

The production A → ϵ generates only one item A → •.

The central idea in the SLR method is first to construct from the grammar a deterministic finite automation to recognize viable prefixes.

We define an augmented grammar and two functions, *closure* and *goto*.

If G is a grammar with start symbol S, then G', the augmented grammar for G, is G with a new start symbol S' and production S' → S. the purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input.

The closure operation

If *I* is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:

1. Initially, every item in *I* is added to *closure (I)*.
2. If $A \rightarrow \alpha \bullet B\beta$ is in *closure(I)* and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \bullet\gamma$ to *I*, if it is not already here. We apply this rule until no more new items can be added to *closure(I)*.

Consider the augmented grammar

$E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

If I is the set of one item {[E'→•E]}, then closure(I) contain the items

$E' \rightarrow \bullet E$
$E \rightarrow \bullet E + T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet T * F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet(E)$
$F \rightarrow \bullet id$

Computation of Closure

```
function closure ( I )
begin
        J := I;
        repeat
                for each item A → α.Bβ in J and each production
                        B→γ of G such that B→.γ is not in J do
                                add B→.γ to J
                until no more items can be added to J
                return J
end
```

There are two classes of items.

1. Kernel items, which include the initial item, S' → •S, and all items whose dots are not at the left end.
2. Non-kernel items, which have their dots at the left end.

The Goto Operation

The second function is goto(I) where I is a set of items and X is a grammar symbol. Goto(I,X) is defined to be the closure of the set of all items [A→αX•β] such that [A→α•Xβ] is in I.

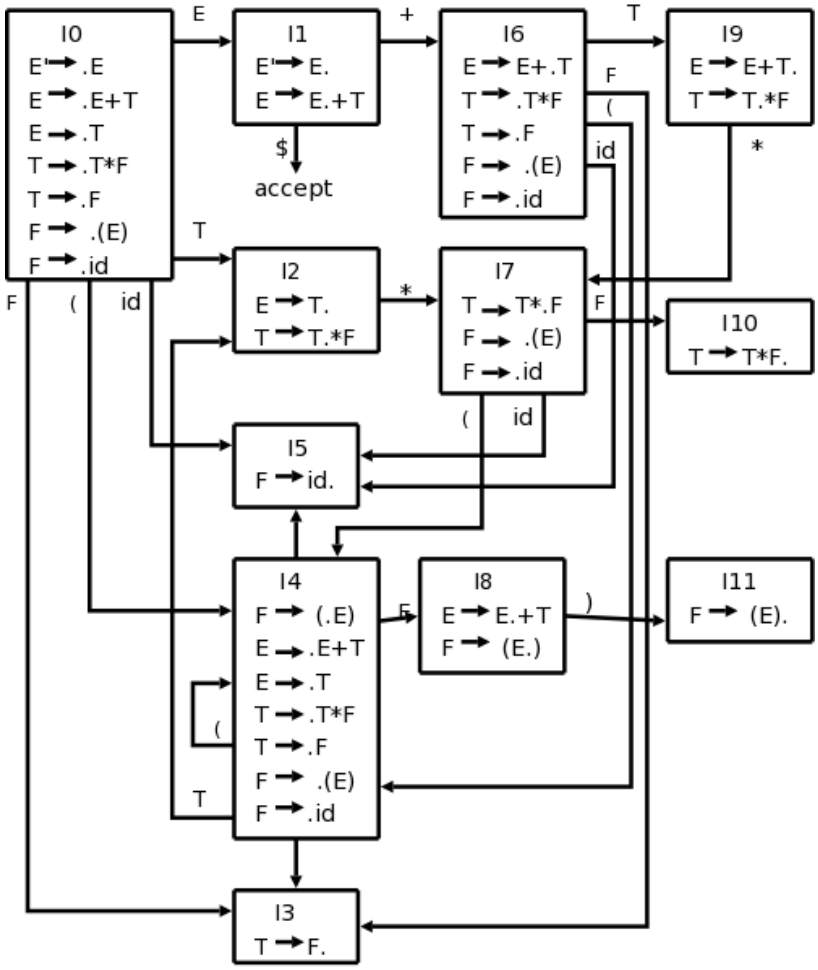
If I is a set of two items {[E'→E•], [E→E•+T]} then goto(I,+) consists of

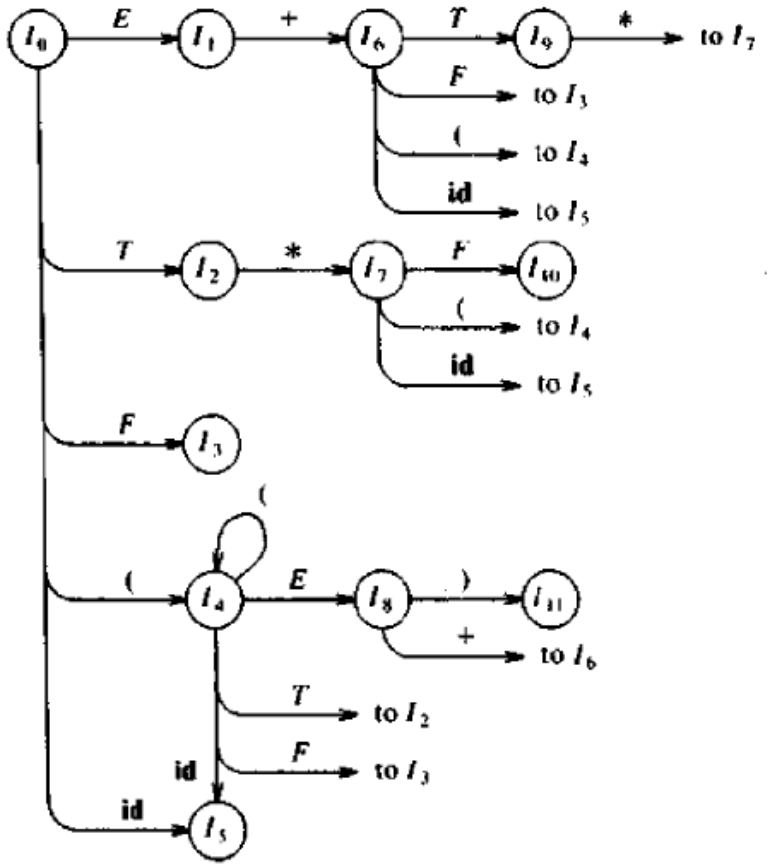
$E \rightarrow E + \bullet T$
$T \rightarrow \bullet T * F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet(E)$
$F \rightarrow \bullet id$

The set of items construction

```
procedure items(G');
begin
    C := {closure({[S' → ·S]})};
    repeat
        for each set of items I in C and each grammar symbol X
            such that goto(I, X) is not empty and not in C do
                add goto(I, X) to C
    until no more sets of items can be added to C
end
```

Transition Diagram





Construction of SLR parsing table

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for G'.

2. State i is constructed from I_i. The parsing actions for state i are determined as follows:

   (a) If $[A \to \alpha \cdot a\beta]$ is in I_i and GOTO$(I_i, a) = I_j$, then set ACTION$[i, a]$ to "shift j." Here a must be a terminal.

   (b) If $[A \to \alpha \cdot]$ is in I_i, then set ACTION$[i, a]$ to "reduce $A \to \alpha$" for all a in FOLLOW(A); here A may not be S'.

   (c) If $[S' \to S \cdot]$ is in I_i, then set ACTION$[i, \$]$ to "accept."

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$.
4. All entries not defined by rules (2) and (3) are made "error."
5. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow \cdot S]$.

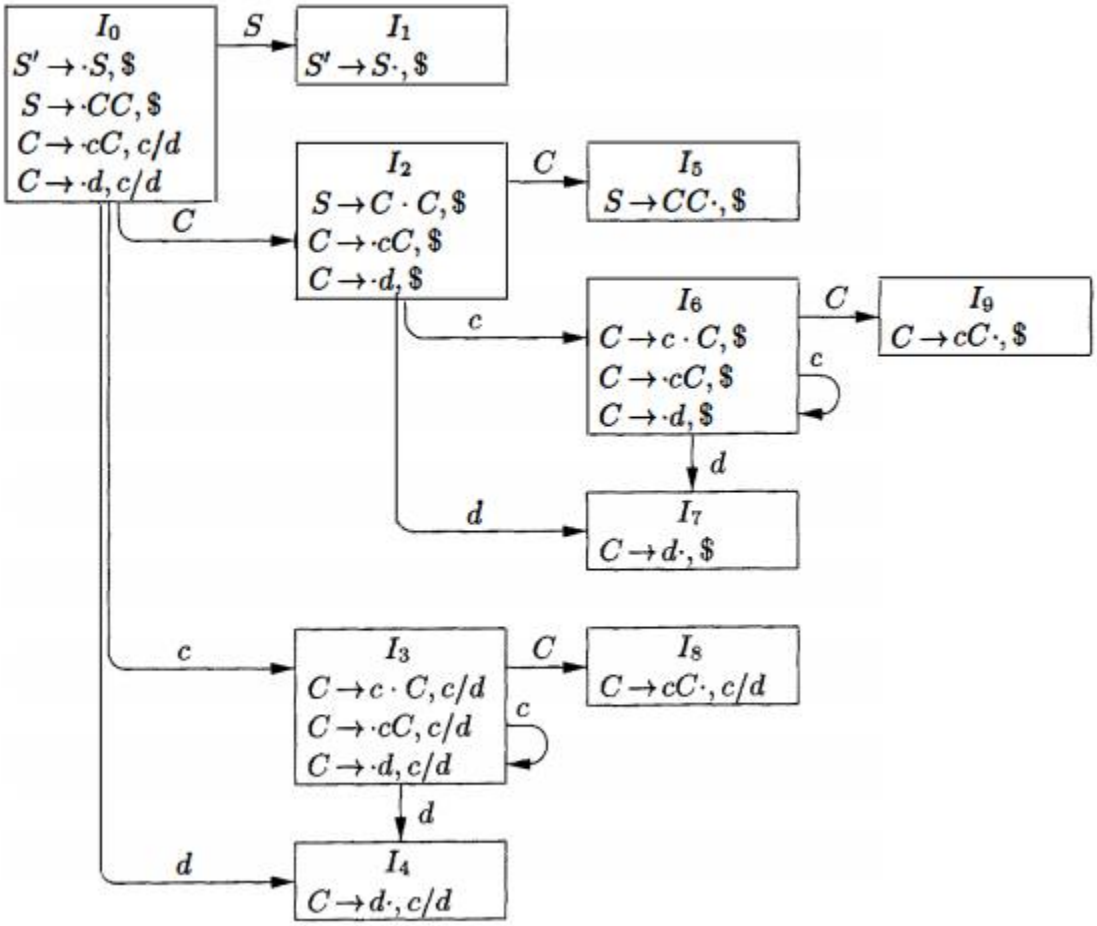
Constructing Canonical LR parsing Tables

The general form of an item becomes [A → α•β, a], where A → αβ is a production and a is a terminal or the right end marker $. This is called ***LR(1) item***. The *1* refers to the length of the second component, called the *lookahead* of the item. The lookahead has no effect in an item of the form [A → α•β, a], where β is not ϵ, but an item of the form [A → α•, a] calls for the production by A → α only if the next input symbol is *a*.

```
function closure (I);
begin
    repeat
        for each item [A → α·Bβ, a] in I,
            each production B → γ in G',
            and each terminal b in FIRST(βa)
            such that [B → ·γ, b] is not in I do
                add [B → ·γ, b] to I;
    until no more items can be added to I;
    return I
end;

function goto (I, X);
begin
    let J be the set of items [A → αX·β, a] such that
        [A → α·Xβ, a] is in I;
    return closure (J)
end;
```

```
procedure items(G');
begin
    C := {closure({[S' → ·S, $]})};
    repeat
        for each set of items I in C and each grammar symbol X
            such that goto(I, X) is not empty and not in C do
                add goto(I, X) to C
    until no more sets of items can be added to C
end
```


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

STATE	ACTION			GOTO	
	c	*d*	$	*S*	*C*
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Algorithm for the construction of canonical LR parsing table

1. Construct $C' = \{I_0, I_1, \cdots, I_n\}$, the collection of sets of LR(1) items for G'.
2. State i of the parser is constructed from I_i. The parsing action for state i is determined as follows.
   (a) If $[A \to \alpha \cdot a\beta, b]$ is in I_i and GOTO$(I_i, a) = I_j$, then set ACTION$[i, a]$ to "shift j." Here a must be a terminal.
   (b) If $[A \to \alpha \cdot, a]$ is in I_i, $A \neq S'$, then set ACTION$[i, a]$ to "reduce $A \to \alpha$."
   (c) If $[S' \to S \cdot, \$]$ is in I_i, then set ACTION$[i, \$]$ to "accept."

   If any conflicting actions result from the above rules, we say the grammar is not LR(1). The algorithm fails to produce a parser in this case.
3. The goto transitions for state i are constructed for all nonterminals A using the rule: If GOTO$(I_i, A) = I_j$, then GOTO$[i, A] = j$.
4. All entries not defined by rules (2) and (3) are made "error."
5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S, \$]$.

Constructing LALR parsing table

This method is often used in practice because the tables obtained by it are considerably smaller than the canonical LR tables. The SLR and LALR tables for a grammar always have the same number of states. The canonical LR is having much larger size.

Consider I4, I7 states in previous, the LR(0) item is similar, but the lookaheads are different. Reduction in I4 will only happen at terminal c and d. Reduction in I7 will only happen at $.

Similarly I3,I6 and I8,I9 are having similar LR(0) items with different lookaheads.

An easy, but space consuming LALR table construction is done by

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items.

2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.

3. Let $C' = \{J_0, J_1, \ldots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i in the same manner as in Algorithm 4.56. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).

4. The GOTO table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is, $J = I_1 \cap I_2 \cap \cdots \cap I_k$, then the cores of $\text{GOTO}(I_1, X)$, $\text{GOTO}(I_2, X), \ldots, \text{GOTO}(I_k, X)$ are the same, since $I_1, I_2, \ldots, I_k$ all have the same core. Let K be the union of all sets of items having the same core as $\text{GOTO}(I_1, X)$. Then $\text{GOTO}(J, X) = K$.

$$
\begin{array}{ll}
I_{36}: & C \to c \cdot C,\ c/d/\$ \\
 & C \to \cdot cC,\ c/d/\$ \\
 & C \to \cdot d,\ c/d/\$
\end{array}
$$

I_4 and I_7 are replaced by their union:

$$I_{47}: \quad C \to d \cdot,\ c/d/\$$$

and I_8 and I_9 are replaced by their union:

$$I_{89}: \quad C \to cC \cdot,\ c/d/\$$$

The LALR action and goto functions for the condensed sets of items are shown

STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		