Chapter 2 Context Free Grammars

The second phase of compilation is Syntactic Analysis or Parsing. In this phase, the program is analysed into its structural components - various sorts of declarations, statements, expressions, etc.

It is in this phase that structural errors are discovered. The parser (the process which performs the syntactic analysis) outputs syntax error messages.

The parser obtains tokens, one at a time, from the lexical analyser, performs the syntactic analysis of the structure of the program, and produces an "abstract syntax tree" which describes the structure of the program.

Language Definition

The structure of a computer language is usually described in terms of what is called a Context Free Grammar or Backus-Naur Form (BNF) of language definition.

A computer program is composed of tokens (Reserved words, constants, identifiers, special symbols, etc.)

The constructs of the language are composed of other constructs and tokens, and the relationship can be described by context free grammar rules such as:

```
S       →   E ";"
        |   "{" SS "}"
        |   "if" "(" E ")" S "else" S
        |   "if" "(" E ")" S

SS      →   ε
        |   SS S

E       →   IDENT
```

The symbol → means "expands to" or "generates", "|" means "or", and ε is just an explicit notation for the empty string, i.e. "nothing" (to make the fact that we have nothing more noticeable than not writing anything). In the above example, S means "Statement", SS means "Statement Sequence", IDENT means "Identifier".

A context free grammar is defined by:

- A finite set N of Nonterminal symbols, including a special symbol S called the start symbol. A nonterminal symbol represents the name for a kind of construct, and the start symbol represents the name of the overall construct.
- A finite set Σ of Terminal Symbols. A terminal symbol represents the name of a kind of token.
- A finite set of rules of the form $A \rightarrow B_1 \ldots B_n$, where A is a nonterminal, and $B_1, \ldots B_n$ are terminals or nonterminals. The notation $A \rightarrow \alpha \mid \beta \mid \gamma$, where $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$ (the set of sequences of terminal and nonterminal symbols), is an abbreviation for several rules with the same left hand side (in this case $A \rightarrow \alpha$, $A \rightarrow \beta$, and $A \rightarrow \gamma$).

Suppose we have a sequence of terminal and nonterminal symbols. We can generate a new sequence of symbols by performing what is called a "reduction". A reduction is a replacement of a sequence of symbols corresponding to the right hand side of a rule by the left hand side of the rule.

For example, suppose we start with the sequence $\alpha\ \beta\ \gamma$, and we have a rule $A \rightarrow \beta$, where $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$. Then we can reduce $\alpha\ \beta\ \gamma$ to the sequence of symbols $\alpha\ A\ \gamma$. We express this by the notation $\alpha\ \beta\ \gamma <= \alpha\ A\ \gamma$. (It is conventional to use lower case Greek letters for sequences of symbols.).

Similarly, we can perform an "expansion" if we do the replacement in the reverse direction, and replace a nonterminal corresponding to the left hand side of a rule by the symbols on the right hand side of the rule.

For example, suppose we start with the sequence $\alpha\ A\ \gamma$, and we have a rule $A \rightarrow \beta$, where $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$. Then we can expand $\alpha\ A\ \gamma$ to the sequence of symbols $\alpha\ \beta\ \gamma$. We express this by the notation $\alpha\ A\ \gamma => \alpha\ \beta\ \gamma$.

A program corresponds to input that can be matched to the start symbol (often something like "Program", "StatementList" or "ExternalDeclarationSequence"). If we perform the matching by starting with the input, and performing reductions, until we end up with the start symbol, then we are performing bottom up parsing. If we perform the matching by starting with the start symbol, and performing expansions, until we end up with the input, then we are performing top down parsing.

For example, if we have the above rules, the following input is a legal statement

```
{
a;
if ( a ) {
      a;
      a;
      }
else
      a;
}
```

This is because we can reduce this to S by the following sequence of reductions

{_a ; if (a) { a ; a ; } else a ; }

Reduce by SS → ε

<= { SS $\underline{\text{a}}$; if (a) { a ; a ; } else a ; }

Reduce by E → IDENT

<= { SS $\underline{\text{E ;}}$ if (a) { a ; a ; } else a ; }

Reduce by S → E ;

<= { $\underline{\text{SS S}}$ if (a) { a ; a ; } else a ; }

Reduce by SS → SS S

<= { SS if ($\underline{\text{a}}$) { a ; a ; } else a ; }

Reduce by E → IDENT

<= { SS if (E) {_a ; a ; } else a ; }

Reduce by SS → ε

<= { SS if (E) { SS $\underline{\text{a}}$; a ; } else a ; }

Reduce by E → IDENT

<= { SS if (E) { SS $\underline{\text{E ;}}$ a ; } else a ; }

Reduce by S → E ;

<= { SS if (E) { $\underline{\text{SS S}}$ a ; } else a ; }

Reduce by SS → SS S

<= { SS if (E) { SS $\underline{\text{a}}$; } else a ; }

Reduce by E → IDENT

<= { SS if (E) { SS $\underline{\text{E ;}}$ } else a ; }

Reduce by S → E ;

<= { SS if (E) { $\underline{\text{SS S}}$ } else a ; }

Reduce by SS → SS S

<= { SS if (E) $\underline{\text{\{ SS \}}}$ else a ; }

Reduce by S → { SS }

<= { SS if (E) S else $\underline{\text{a}}$; }

Reduce by E → IDENT

<= { SS if (E) S else $\underline{\text{E ;}}$ }

Reduce by S → E ;

<= { SS $\underline{\text{if (E) S else S}}$ }

Reduce by S → if (E) S else S

<= { $\underline{\text{SS S}}$ }

Reduce by SS → SS S

<= $\underline{\text{\{ SS \}}}$

Reduce by S → { SS }

<= $\underline{\text{S}}$

Accept

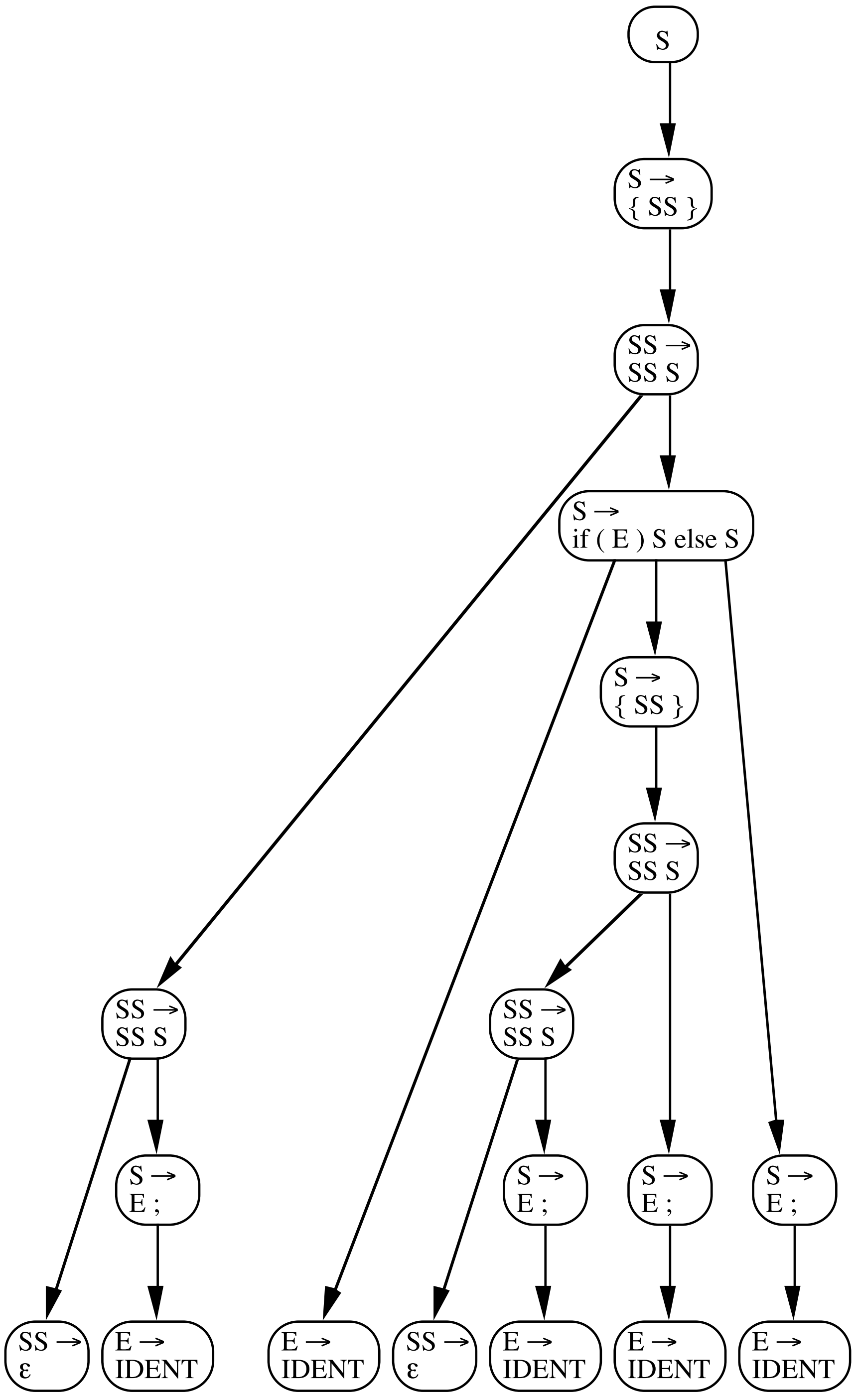
The sequence of symbols formed at each stage of the sequence is called a “sentential form”. (The terminology comes from the parsing of natural language, in which we are trying to parse a sentence.)

A parse tree can be built to represent the structure of the program. The root node is the start symbol. The internal nodes of the tree correspond to the rules used to reduce the program to the start symbol. There are also terminal nodes corresponding to the terminal symbols in the input that have attributes. Essentially we create the parse tree in a bottom up fashion, creating a new node whenever we perform a reduction.

Example

```
{
a;
if ( a ) {
     a;
     a;
     }
else
     a;
}
```

generates the parse tree shown below



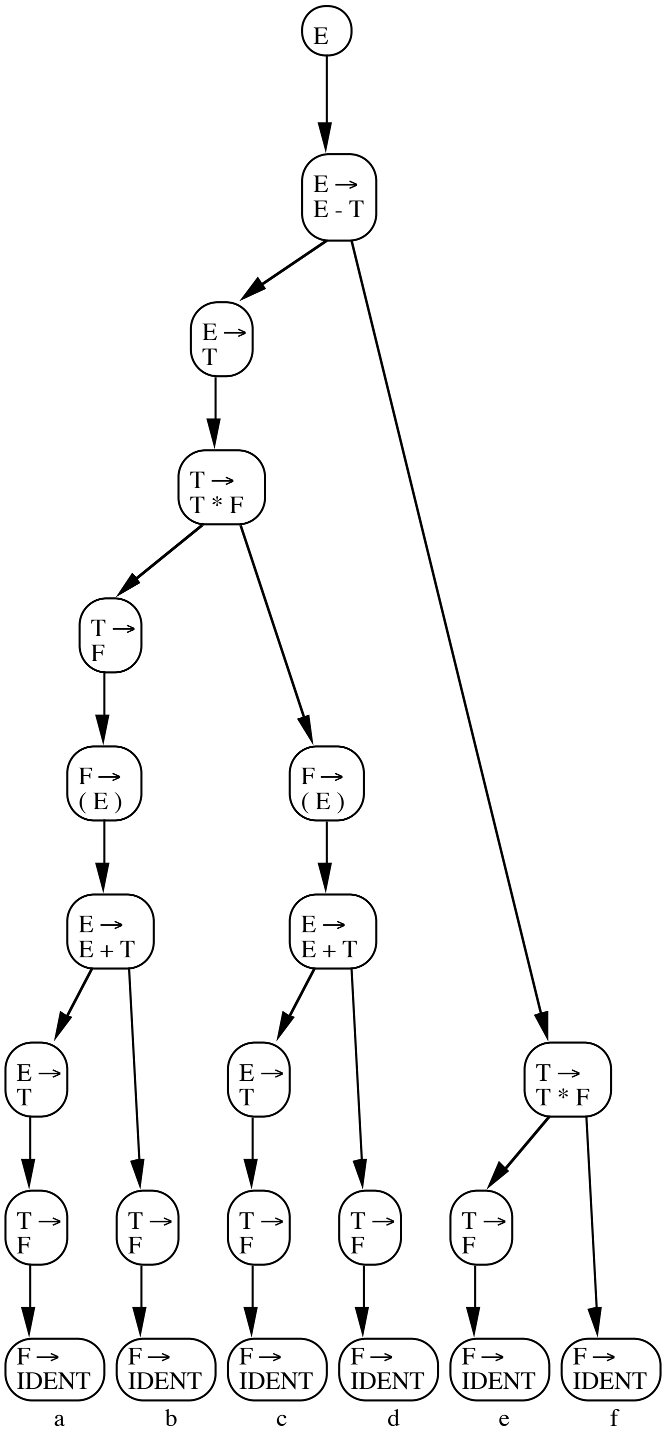
Sometimes nodes of the parse tree are irrelevant as far as the rest of the compiler is concerned. For example, the code generated for a parenthesised expression is the same as for an unparenthesised expression. The parentheses might have been important when performing parsing, in that they determined how operators and operands were grouped, but once the structure of the expression has been determined, they can be ignored. Similarly, nodes corresponding to precedence specifying rules, such as $E \rightarrow T$, $T \rightarrow F$ can be discarded.

For example, if we have the grammar rules

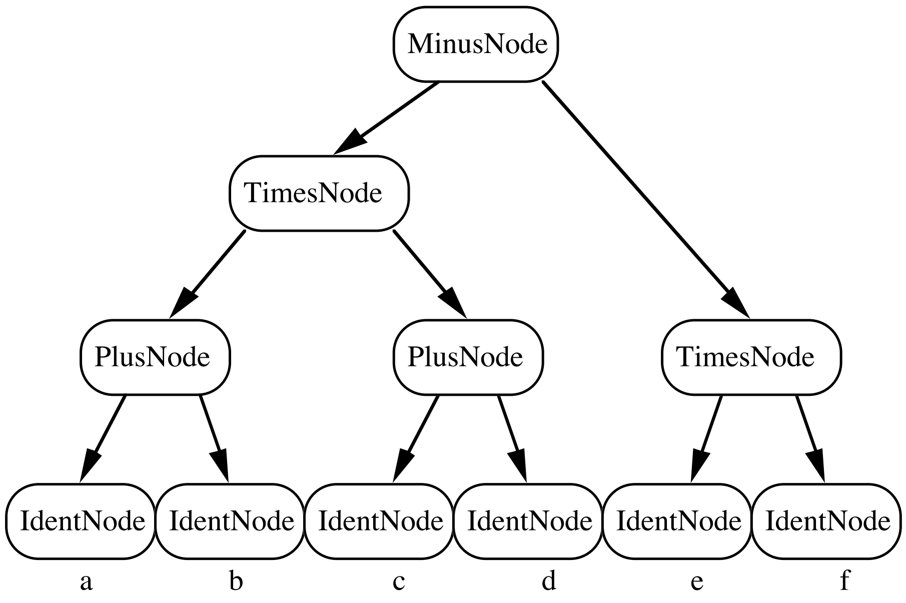
E	→	E **“+”** T | E **“-”** T | **“-”** T | T
T	→	T **“*”** F | T **“/”** F | F
F	→	**IDENT** | **NUMBER** | **“(”** E **“)”**

(E means Expression, T means Term, F means Factor).

Then the expression (a + b) * (c + d) - e * f generates the parse tree



but can be simplified to the abstract syntax tree



While it is a bit early to show you a CUP program, perhaps you can see how this is matched by the following code. The Java code enclosed in `{:...:}` represents the code to build a node of the tree.

```
Expr::=
        Expr:expr PLUS Term:term
        {:
        RESULT = new PlusNode( expr, term );
        :}
    |
        Expr:expr MINUS Term:term
        {:
        RESULT = new MinusNode( expr, term );
        :}
    |
        MINUS Term:term
        {:
        RESULT = new NegateNode( term );
        :}
    |
        Term:term
        {:
        RESULT = term;
        :}
    ;

Term::=
        Term:term TIMES Factor:factor
        {:
        RESULT = new TimesNode( term, factor );
        :}
    |
        Term:term DIVIDE Factor:factor
        {:
        RESULT = new DivideNode( term, factor );
        :}
    |
        Factor:factor
        {:
        RESULT = factor;
        :}
```

```
     ;

Factor::=
          LEFT Expr:expr RIGHT
          {:
          RESULT = expr;
          :}
     |
          NUMBER:value
          {:
          RESULT = new NumberNode( Integer.parseInt( value ) );
          :}
     |
          IDENT:ident
          {:
          RESULT = new IdentNode( ident );
          :}
     ;
```

Bottom Up (Shift-Reduce) Parsing

To perform a bottom up or shift-reduce parse, we use a stack of terminal and nonterminal symbols. We process the input from left to right. At each stage, the stack corresponds to the terminal symbols and constructs processed so far. The stack, concatenated with the remaining input yet to be processed, constitute a sentential form. Initially the stack contains just a “bottom of stack” marker, “$”. The stack is considered to grow from left to right.

We shift the input onto the stack, until we have a complete right hand side of a rule on the top of the stack. Whenever a complete right hand side of a rule appears on the top of the stack, we reduce it to its left hand side (i.e., replace it by the left hand side of the rule). We terminate when we shift the end of file marker (also represented by “$”) onto the stack and the stack contains “$ StartSymbol $”.

The decision on whether to shift or reduce, and if to reduce, which rule to reduce by, is made on the basis of the top portion of the stack, and the current token. The exact algorithm for making this decision varies for different forms of bottom up parsing, and the decision making is not trivial. A program called a parser generator is normally used to generate tables used to drive the parser.

The algorithm for bottom up parsing is shown below.

```
Make the stack “$”;

GetToken();

forever
     {
     if ( Stack is “$ StartSymbol $” )
          return;
     else if ( Stack and CurrToken imply
          have partial righthand side of rule on top of stack )
          Shift CurrToken onto stack;
          GetToken();
          }
     else if ( Stack and CurrToken imply
          have complete righthand side of rule on top of stack )
          Replace righthand side of rule on top of stack by lefthand side;
     else
          error();
     }
```

Example

Consider the grammar

E → E **“+”** T | E **“-”** T | **“-”** T | T

T → T “*” F | T “/” F | F

F → **IDENT** | **NUMBER** | “(” E “)”

and the input

a + b + c * d,

where a, b, c, d are all considered as identifiers. We can reduce the input to the start symbol E by

	$\underline{a} + b + c * d$	
		Reduce F → IDENT
<=	$\underline{F} + b + c * d$	
		Reduce T → F
<=	$\underline{T} + b + c * d$	
		Reduce E → T
<=	$E + \underline{b} + c * d$	
		Reduce F → IDENT
<=	$E + \underline{F} + c * d$	
		Reduce T → F
<=	$\underline{E + T} + c * d$	
		Reduce E → E + T
<=	$E + \underline{c} * d$	
		Reduce F → IDENT
<=	$E + \underline{F} * d$	
		Reduce T → F
<=	$E + T * \underline{d}$	
		Reduce F → IDENT
<=	$E + \underline{T * F}$	
		Reduce T → T * F
<=	$\underline{E + T}$	
		Reduce E → E + T
<=	$\underline{E}$	
		Accept

The bottom up parse of the input is shown below. $ represents both bottom of stack and end of input.

Stack	**Input Remaining**	**Action**
---->$_{\text{grows}}$		
$	a + b + c * d $	Shift IDENT
$ $\underline{\text{Id}}$	+ b + c * d $	Reduce F → IDENT
$ $\underline{\text{F}}$	+ b + c * d $	Reduce T → F
$ $\underline{\text{T}}$	+ b + c * d $	Reduce E → T
$ E	+ b + c * d $	Shift +
$ E +	b + c * d $	Shift IDENT
$ E + $\underline{\text{Id}}$	+ c * d $	Reduce F → IDENT
$ E + $\underline{\text{F}}$	+ c * d $	Reduce T → F
$ $\underline{\text{E + T}}$	+ c * d $	Reduce E → E + T
$ E	+ c * d $	Shift +
$ E +	c * d $	Shift IDENT
$ E + $\underline{\text{Id}}$	* d $	Reduce F → IDENT
$ E + $\underline{\text{F}}$	* d $	Reduce T → F
$ E + T	* d $	Shift *
$ E + T *	d $	Shift IDENT
$ E + T * $\underline{\text{Id}}$	$	Reduce F → IDENT
$ E + $\underline{\text{T * F}}$	$	Reduce T → T * F
$ $\underline{\text{E + T}}$	$	Reduce E → E + T
$ $\underline{\text{E}}$	$	Shift $
$\underline{\text{\$ E \$}}$		Accept

LALR(1) Parsing

In many forms of bottom up parsing, we place “states” on the stack, rather than terminal and nonterminal symbols. A state indicates all the possible rules we may be in the process of analysing, where we are up to in our analysis, and perhaps the possible terminal symbols that can follow the application of the rule. A state summarises all the information in the stack that is relevant in deciding whether to shift or reduce, and if to reduce, which rule to reduce by. While states are complicated objects, there are only a finite number of possible states, and these states can be precomputed by the parser generator, and referred to by number, when parsing is performed.

An action table is created to indicate the action to perform for a given state and input token. The action table is a mapping indexed by the top of stack state and current token. It returns a value of the form:

- SHIFT State
- REDUCE Rule
- ACCEPT
- ERROR

There is also a goto table created to indicate the state to push on when a reduction is performed. The goto table is a mapping indexed by the top of stack state uncovered after removal of the states corresponding to the right hand side of the rule, and the nonterminal corresponding to the left hand side to be pushed onto the stack. It returns the new state to be pushed onto the stack.

These tables are used to drive the parser. This form of bottom up parsing is called LR parsing. The parsing algorithm is shown below.

```
Start with a stack of the form
     state 0
     --->grows
where state 0 corresponds to having seen no input;

GetToken();

forever {
     /*
     Index the action table by
          the top of stack state
          and the current token
     to get the action to perform
     */
     CurrAction =  Action[ TopStack(), CurrToken ];
     switch ( CurrAction.ActionSort ) {
          case SHIFT:
               /*
               Push the indicated state onto the stack
               */
               Push( CurrAction.ShiftState );
               /*
               Get the next token
               */
               GetToken();
               break;
          case REDUCE:
               Perform code associated with CurrAction.ReduceRule
                    (e.g., build node of tree corresponding to
                    the grammar rule);
               /*
               Pop the states corresponding to
                    the righthand side of the rule
                    off the stack
               */
               Pop CurrentAction.ReduceRule.RHS.length()
                    states off the stack;
               /*
               Index the goto table by
                    the state uncovered on the top of stack
                    and the nonterminal corresponding to
                         the lefthand side of the rule
               and push the state found in the goto table
                    onto the stack;
               */
               Push( GoTo[ TopStack(),
                    CurrentAction.ReduceRule.LHS ] );
               break;
          case ACCEPT:
               /*
               Complete parsing
               */
               return;
          case ERROR:
               /*
               Generate an error message
               */
               error();
               break;
          }
     }
```

Example

Consider the grammar

```
StmtList →
         ε
     |
         StmtList Stmt "\n"
     |
         StmtList error "\n"
     |
         StmtList "\n"

Stmt →
         IDENT "=" Expr
     |
         Expr

Expr →
         Expr "+" Term
     |
         Expr "-" Term
     |
         "-" Term
     |
         Term

Term →
         Term "*" Factor
     |
         Term "/" Factor
     |
         Factor

Factor →
         "(" Expr ")"
     |
         NUMBER
     |
         IDENT
```

This grammar generates the LALR(1) action table shown below.

State Meaning	**State**	**$**	**err**	**(**	**)**	**\n**	**+**	**-**	*****	**/**	**=**	**Num**	**Id**
	0	r1	r1	r1		r1		r1				r1	r1
StmtList	**1**	s11	s12	s8		s4		s9				s2	s3
Number	**2**				r15	r15	r15	r15	r15	r15			
Ident	**3**					r16	r16	r16	r16	r16	s27		
StmtList \n	**4**	r4	r4	r4		r4		r4				r4	r4
Expr	**5**					r6	s23	s22					
Factor	**6**				r13	r13	r13	r13	r13	r13			
Term	**7**				r10	r10	r10	r10	s17	s18			
(	**8**			s8				s9				s2	s15
-	**9**			s8								s2	s15
StmtList Stmt	**10**					s14							
$ StmtList $	**11**	Acc											
StmtList error	**12**					s13							
StmtList error \n	**13**	r3	r3	r3		r3		r3				r3	r3
StmtList Stmt \n	**14**	r2	r2	r2		r2		r2				r2	r2
Ident	**15**				r16	r16	r16	r16	r16	r16			
- Term	**16**				r9	r9	r9	r9	s17	s18			
Term *	**17**			s8								s2	s15
Term /	**18**			s8								s2	s15
Term / Factor	**19**				r12	r12	r12	r12	r12	r12			
Term * Factor	**20**				r11	r11	r11	r11	r11	r11			
(Expr	**21**				s24		s23	s22					
Expr -	**22**			s8								s2	s15
Expr +	**23**			s8								s2	s15
(Expr)	**24**				r14	r14	r14	r14	r14	r14			
Expr + Term	**25**				r7	r7	r7	r7	s17	s18			
Expr - Term	**26**				r8	r8	r8	r8	s17	s18			
Ident =	**27**			s8				s9				s2	s15
Ident = Expr	**28**					r5	s23	s22					

s_n means shift state n, r_n means reduce by rule n. Acc means accept, and a blank means an error. $ is used to represent the end of text.

This grammar generates the LALR(1) goto table shown below.

State	**StmL**	**Stmt**	**E**	**T**	**F**	
0	s1					r1 = StmtList → ε
1		s10	s5	s7	s6	
2						r15 = Factor → Number
3						r16 = Factor → Ident
4						r4 = StmtList → StmtList \n
5						r6 = Stmt → Expr
6						r13 = Term → Factor
7						r10 = Expr → Term
8			s21	s7	s6	
9				s16	s6	
10						
11						
12						
13						r3 = StmtList → StmtList error \n
14						r2 = StmtList → StmtList Stmt \n
15						r16 = Factor → Ident
16						r9 = Expr → - Term
17					s20	
18					s19	
19						r12 = Term → Term / Factor
20						r11 = Term → Term * Factor
21						
22				s26	s6	
23				s25	s6	
24						r14 = Factor → (Expr)
25						r7 = Expr → Expr + Term
26						r8 = Expr → Expr - Term
27			s28	s7	s6	
28						r5 = Stmt → Ident = Expr

s_n means shift state n.

A state summarises all the relevant information for determining the action, that can be gleaned from the (conventional) stack of symbols. For example, state 20 represents a stack of the form "... Term * Factor", for which we know that we should reduce "Term * Factor" to "Term". State 25 represents a stack of the form "...Expr + Term", for which we should reduce "Expr + Term" to "Expr" if the current token is not a higher precedence operator, such as "*" or "/". If the current token is "*" or "/", we should shift, rather than reduce.

Using the table, we can perform a bottom up parse of the input

```
a + b + c * d \n
```

where we assume that a, b, c, d are all tokenised to IDENT by the lexical analyser. This is illustrated below.

Stack	**Input**	**Action**
$\$^0$	Id a	Reduce StmtList → ε
$\$^0$ SL^1		Shift Id^3
$\$^0$ SL^1 Id^3	+	Reduce F → Ident
$\$^0$ SL^1 F^6		Reduce T → F
$\$^0$ SL^1 T^7		Reduce E → T
$\$^0$ SL^1 E^5		Shift $+^{23}$
$\$^0$ SL^1 E^5 $+^{23}$	Id b	Shift Id^{15}
$\$^0$ SL^1 E^5 $+^{23}$ Id^{15}	+	Reduce F → Ident
$\$^0$ SL^1 E^5 $+^{23}$ F^6		Reduce T → F
$\$^0$ SL^1 E^5 $+^{23}$ T^{25}		Reduce E → E + T
$\$^0$ SL^1 E^5		Shift $+^{23}$
$\$^0$ SL^1 E^5 $+^{23}$	Id c	Shift Id^{15}
$\$^0$ SL^1 E^5 $+^{23}$ Id^{15}	*	Reduce F → Ident
$\$^0$ SL^1 E^5 $+^{23}$ F^6		Reduce T → F
$\$^0$ SL^1 E^5 $+^{23}$ T^{25}		Shift $*17$
$\$^0$ SL^1 E^5 $+^{23}$ T^{25} $*17$	Id d	Shift Id^{15}
$\$^0$ SL^1 E^5 $+^{23}$ T^{25} $*17$ Id^{15}	\n	Reduce F → Ident
$\$^0$ SL^1 E^5 $+^{23}$ T^{25} $*17$ F^{20}		Reduce T → T * F
$\$^0$ SL^1 E^5 $+^{23}$ T^{25}		Reduce E → E + T
$\$^0$ SL^1 E^5		Reduce S → E
$\$^0$ SL^1 S^{10}		Shift $\backslash n^{14}$
$\$^0$ SL^1 S^{10} $\backslash n^{14}$	$	Reduce StmtList → StmtList Stmt \n
$\$^0$ SL^1		Shift $\$^{11}$
$\$^0$ SL^1 $\$^{11}$		Accept

Terminal and nonterminal symbols have been appended to the states, to indicate the top of stack symbol that they would conventionally correspond to. In practice, only the states appear on the stack, but these symbols have been added for readablity.

Consider another example

```
a * ( - b ) \n
```

Stack	**Input**	**Action**
$\$^{0}$	Id a	Reduce StmtList → ε
$\$^{0}\ SL^{1}$		Shift Id^{3}
$\$^{0}\ SL^{1}\ Id^{3}$	*	Reduce F → Ident
$\$^{0}\ SL^{1}\ F^{6}$		Reduce T → F
$\$^{0}\ SL^{1}\ T^{7}$		Shift $*17$
$\$^{0}\ SL^{1}\ T^{7}\ *17$	(	Shift $(^{8}$
$\$^{0}\ SL^{1}\ T^{7}\ *17\ (^{8}$	-	Shift $-^{9}$
$\$^{0}\ SL^{1}\ T^{7}\ *17\ (^{8}\ -^{9}$	Id b	Shift Id^{15}
$\$^{0}\ SL^{1}\ T^{7}\ *17\ (^{8}\ -^{9}\ Id^{15}$	)	Reduce F → Ident
$\$^{0}\ SL^{1}\ T^{7}\ *17\ (^{8}\ -^{9}\ F^{6}$		Reduce T → F
$\$^{0}\ SL^{1}\ T^{7}\ *17\ (^{8}\ -^{9}\ T^{16}$		Reduce E → -T
$\$^{0}\ SL^{1}\ T^{7}\ *17\ (^{8}\ E^{21}$		Shift $)^{24}$
$\$^{0}\ SL^{1}\ T^{7}\ *17\ (^{8}\ E^{21}\ )^{24}$	\n	Reduce F → (E)
$\$^{0}\ SL^{1}\ T^{7}\ *17\ F^{20}$		Reduce T → T * F
$\$^{0}\ SL^{1}\ T^{7}$		Reduce E → T
$\$^{0}\ SL^{1}\ E^{5}$		Reduce S → E
$\$^{0}\ SL^{1}\ S^{10}$		Shift $\backslash n^{14}$
$\$^{0}\ SL^{1}\ S^{10}\ \backslash n^{14}$	$	Reduce StmtList → StmtList Stmt \n
$\$^{0}\ SL^{1}$		Shift $\$^{11}$
$\$^{0}\ SL^{1}\ \$^{11}$		Accept

A bottom up parse of the invalid input

```
        - a * b + c * - d \n
```

is shown below.

Stack	**Input**	**Action**
$\$0$	-	Reduce StmtList → ε
$\$^0\ SL^1$		Shift $-^9$
$\$^0\ SL^1\ -^9$	Id a	Shift Id^{15}
$\$^0\ SL^1\ -^9\ Id^{15}$	*	Reduce F → Ident
$\$^0\ SL^1\ -^9\ F^6$		Reduce T → F
$\$^0\ SL^1\ -^9\ T^{16}$		Shift $*17$
$\$^0\ SL^1\ -^9\ T^{16}\ *17$	Id b	Shift Id^{15}
$\$^0\ SL^1\ -^9\ T^{16}\ *17\ Id^{15}$	+	Reduce F → Ident
$\$^0\ SL^1\ -^9\ T^{16}\ *17\ F^{20}$		Reduce T → T * F
$\$^0\ SL^1\ -^9\ T^{16}$		Reduce E → - T
$\$^0\ SL^1\ E^5$		Shift $+^{23}$
$\$^0\ SL^1\ E^5\ +^{23}$	Id c	Shift Id^{15}
$\$^0\ SL^1\ E^5\ +^{23}\ Id^{15}$	*	Reduce F → Ident
$\$^0\ SL^1\ E^5\ +^{23}\ F6$		Reduce T → F
$\$^0\ SL^1\ E^5\ +^{23}\ T^{25}$		Shift $*17$
$\$^0\ SL^1\ E^5\ +^{23}\ T^{25}\ *17$	-	Error (Cannot parse due to invalid input).

The derivation of the action and goto tables is nontrivial. It should nevertheless be obvious that, given the tables, it is a simple matter to perform a bottom up parse of the input.

States

A state corresponds to a set of "items".

An item is a rule we may be in the process of analysing, together with an indication of where we are up to in analysing the rule (represented by a "." inserted in the right hand side of the rule), and an indication of the tokens that could follow the construct corresponding to the rule.

For example, we could have the item

Rule with "."	**Follow symbols**
Term → Term . * Factor	) \n + - * /

Since there are only a finite number of possible rules, possible places in the rule, and possible follow symbols, there are only a finite number of possible items, and hence only a finite number of possible states. Thus it is possible to identify each state with a number.

There is a mapping from conventional stacks (in other words, sequences of symbols) to states. Stacks that can actually occur (what are called "viable prefixes" of "right sentential forms") map to nonempty states, while impossible stacks map to the empty set (which is not regarded as a state).

- **The state corresponding to a stack is intended to summarise all important information on the stack, about rules we might be in the process of analysing, necessary for deciding what action to perform.**
- **There is a state (State 0) that corresponds to an empty stack.**

- **The state corresponding to a stack α X can be computed from the state corresponding to α, and the symbol X.**
- **The function that performs this mapping is called the goto function.**
- **If Q is the state corresponding to stack α, then goto(Q, X) is the state corresponding to the stack α X.**
- **We can generate the state corresponding to X_1 X_2 ... X_n by computing**

  Q_1 = goto(State 0, X_1), Q_2 = goto(Q_1, X_2), ... Q_n = goto(Q_{n-1}, X_n).

It turns out to be convenient to augment our grammar, by adding an extra rule. Suppose S is the start symbol. CUP adds another rule S' $\rightarrow$ \$ S \$, where \$ represents the beginning and end of input. (Other parser generators add a rule S' $\rightarrow$ S instead, and there are minor differences with regard to the point at which the parse is accepted).

State 0, the state corresponding to the empty stack, is generated by taking the "closure" of the set { [S' $\rightarrow$ \$. S \$, {}] }. This represents having processed nothing, and expecting to process S, followed by end of input.

Goto(Q, X) is the state formed by taking the "closure" of

{ [A $\rightarrow$ α X . β, F] for which [A $\rightarrow$ α . X β, F] $\in$ Q }.

(If [A $\rightarrow$ α . X β, F] was possible before processing X, then [A $\rightarrow$ α X . β, F] must be possible after processing X.)

How do we take the closure of a set of items Q?

If there is already an item of the form [A $\rightarrow$ α . B β, F] in Q, and there is a rule of the form B $\rightarrow$ γ:

- **If β is present, add [B $\rightarrow$.γ, first(β)] to Q. (These follow symbols are said to be spontaneously generated from the item [A $\rightarrow$ α . B β, F]).**
- **If β is nullable (is not present, or can expand to nothing), add [B $\rightarrow$.γ, F] to Q. (These follow symbols are said to propagate from the item [A $\rightarrow$ α . B β, F]).**

We continue to do this until a complete pass through the items in Q adds no more items to Q.

(Because if the item [A $\rightarrow$ α . B β, F] is possible, we might be starting the construct B, and hence we might be starting γ. Thus taking the closure amounts to adding in the additional items that must be possible, as a consequence of the existing items.)

- **To generate all possible states, we can start with state 0, and keep processing states, generating goto(Q, X) for all states Q and symbols X, until a complete pass through the set of states generates no new states.**

The set of items in a state that have symbols to the left of the "." is called the **kernel set of items** of the state. This is effectively what you get by taking the goto of another state (selecting states to process, and shifting the "." over the symbol X), but not taking the closure. The full set of items can be derived from the kernel set. This used to be important in the days of machines with 64KB of memory. It meant that the states could be stored more concisely.

LALR(1) and LR(1) states

There are variations of the above algorithm, that generate different states, and hence different parsing tables. These include LALR(1) and LR(1) systems of bottom up parsing. (LR = Process from the Left, producing a Right parse. LALR = LookAhead LR. The parameter indicates the number of tokens of lookahead.).

The core of a state is the set of items, after discarding the follow symbols.

- **With what is called LR(1) parsing, states are considered to be the same if the items are the same, taking into account any differences in the follow symbols.**
- **With what is called LALR(1) parsing, states are considered to be the same if the items are the same, ignoring any differences in the follow symbols. We compute the states as for LR(1) parsing, but merge states that have the same "core".**
- **There is a many to 1 mapping from LR(1) states to LALR(1) states. An LALR(1) state is formed by merging LR(1) states with the same "core".**

Generation of the parsing tables

Sometimes we have a grammar that is not parsable by a particular system. Usually this is because the grammar is in fact ambiguous (there is more than one way of parsing the input). However, the reason could be more subtle than this. It could be that we have to look more than one token beyond the end of the rule to determine the rule to apply. In this case the grammar is not LR(1). If a grammar is LR(1), but not LALR(1), then the LALR(1) state does not summarise enough information about the stack to permit us to make the appropriate decision.

The action and goto tables are derived from the states.

Action(State Q, Terminal x) =

- **Shift goto(Q, x), if there is an item of the form $[\ A \rightarrow \beta\ .\ x\ \gamma,\ F\ ] \in Q$. Under these circumstances, goto(Q, x) will be nonempty.**
- **Reduce by Rule $A \rightarrow \beta$, if the item $[A \rightarrow \beta\ .\ ,\ F\ ] \in Q$, and $x \in F$.**
- **Accept, if the item $[\ S' \rightarrow \$\ S\ \$\ .\ ,\ \{\}\ ] \in Q$ and x = \$, the end of text symbol.**
- **Error, otherwise.**

GoTo(State Q, Nonterminal B) =

- **Shift goto(Q, B), if there is an item of the form $[\ A \rightarrow \alpha\ .\ B\ \beta,\ F\ ] \in Q$. Under these circumstances, goto(Q, B) will be nonempty.**
- **Error, otherwise.**

If a grammar is not parsable, then the action table has conflicts (i.e., more than one alternative action for a given state and terminal symbol). There are two kinds of conflicts - shift/reduce and reduce/reduce conflicts.

- **A shift/reduce conflict occurs for a state Q and current token x, if there is both an item of the form $[A \rightarrow \beta_1\ .\ x\ \gamma,\ F_1\ ] \in Q$ and an item of the form $[B \rightarrow \beta_2\ .\ ,\ F_2\ ] \in Q$, where $x \in F_2$.**
- **A reduce/reduce conflict occurs for a state Q and current token x, if there is both an item of the form $[A \rightarrow \beta_1\ .\ ,\ F_1\ ] \in Q$, where $x \in F_1$ and an item of the form $[B \rightarrow \beta_2\ .\ ,\ F_2\ ] \in Q$, where $x \in F_2$.**

(One of β_1 and β_2 will be a suffix of the other.)

- **In terms of power (ability to parse more grammars, and detect errors sooner), LALR(1) is less powerful than LR(1).**
- **More powerful parsing techniques have fewer non-error entries than less powerful techniques. Thus a more powerful technique detects an error in the input sooner.**

We must then make some decision on how to resolve the conflict (i.e. prefer one action over another), or use a more powerful method that does not generate conflicts.

Usually reduce/reduce conflicts represent a major error in the design of the grammar, and cannot be resolved. Shift/reduce conflicts are relatively common, and often relate to operator precedence or processing nested "if" statements with a single "else". The shift/reduce conflict for nested if statements is usually resolved by preferring the shift over the reduction. For operators, we have to take into account the precedence of the operators.

When parsing with tables generated by a less powerful method, we may perform more reductions before detecting an error, than we would with a more powerful method. However, no matter what method is used for generating the tables, we never perform a shift when the current token is in error.

The LR(1) method produces more states than the other methods. There is a many to one mapping of LR(1) states onto LALR(1) states. (For typical computer languages, there are hundreds of LALR(1) states, and thousands of LR(1) states. Each LALR(1) state for expressions splits into multiple LR(1) states, with one for each terminal that can follow the expression. For example, in Pascal "while Expr" will be followed by "do", and "if Expr" by "then". The LALR(1) states corresponding to having shifted "while Expr <" and "if Expr <" are the same (the closure of the item [Expr $\rightarrow$ Expr < . Expr]), but the LR(1) states are different.) An LALR(1) state is the union of all LR(1) states with the same "core" - the state formed from an LR(1) state by stripping the follow symbols from the component items.

LALR(1) parsing has less theoretical justification than LR(1) parsing, but LALR(1) parsing has an advantages in terms of the number of states, while having the decision making power of LR(1) parsing, for almost all practical grammars.

The LALR(1) state summarises all the information that is relevant in deciding whether to shift or reduce, and if to reduce, which rule to reduce by, given only 1 token lookahead and the possible rules we might be processing as information when making a decision. The LALR(1) parser detects an error at the first possible occasion given only the potential rules, and the current token, as information.

There is also a method of parsing called SLR(1) (Simple LR(1)) parsing. In this method of parsing, the items in the states do not include the follow symbols. We use the follow set of the lefthand side instead. SLR(1) parsing is less powerful than LALR(1), but it is easier to compute the tables.

The actions for an LALR(1) state with a given token, are essentially just equal to the merged actions of the component LR(1) states. If there is a conflict for an LR(1) state, then that conflict also occurs for the LALR(1) state it merges into. Any shift action that occurs for an LALR(1) state, occurs for all component LR(1) states, while a reduce action will only occur in some of the LR(1) states. A consequence of this is that using LR(1) instead of LALR(1), may decrease the number of reduce/reduce conflicts, but will not affect the existence of shift/reduce conflicts.

An Example

The above grammar, when processed by the CUP parser generator, generates the following LALR(1) states. The "." on the right hand side of the rule represents where we could be up to in parsing. The list of terminal symbols on the right indicate the symbols that could follow the application of the rule. After the set of items that make up a state, is the goto mapping, for symbols for which it is not empty.

Try and match the table entries to the information in the states.

For example, in State 17, we have the following information:

State 17

Factor → . (Expr)	) \n + - * /
Factor → . IDENT	) \n + - * /
Factor → . NUMBER	) \n + - * /
Term → Term * . Factor	) \n + - * /

(	Go to State	8
Factor	Go to State	20
IDENT	Go to State	15
NUMBER	Go to State	2

So, we shift state 8 if the current token is a "(", we shift state 15 if the current token is an identifier, we shift state 2 if the current token is a number, and if we get state 17 on the top of stack after reducing something to a Factor, we shift state 20.

In State 25, we have the following information:

State 25

Expr → Expr + Term .	) \n + -
Term → Term . * Factor	) \n + - * /
Term → Term . / Factor	) \n + - * /

*	Go to State	17
/	Go to State	18

So, we shift state 17 if the current token is a "*", shift state 18 if the current token is a "/", reduce by rule "Expr → Expr + Term" if the current token can follow this rule, namely ")", "\n", "+", "-".

It is important to be able to "read" the description of the set of states (generated as output by the CUP parser generator), when attempting to debug your grammar definition.

State 0

$START → $. StmtList $	$
StmtList → .	$ error (\n - NUMBER IDENT
StmtList → . StmtList error \n	$ error (\n - NUMBER IDENT
StmtList → . StmtList Stmt \n	$ error (\n - NUMBER IDENT
StmtList → . StmtList \n	$ error (\n - NUMBER IDENT

StmtList	Go to State	1

State 1

$START → $ StmtList . $	$
Expr → . - Term	\n + -
Expr → . Expr + Term	\n + -
Expr → . Expr - Term	\n + -
Expr → . Term	\n + -

Factor → . (Expr)		\n + - * /
Factor → . IDENT		\n + - * /
Factor → . NUMBER		\n + - * /
StmtList → StmtList . error \n		$ error (\n - NUMBER IDENT
StmtList → StmtList . Stmt \n		$ error (\n - NUMBER IDENT
StmtList → StmtList . \n		$ error (\n - NUMBER IDENT
Stmt → . Expr		\n
Stmt → . IDENT ASSIGN Expr		\n
Term → . Factor		\n + - * /
Term → . Term * Factor		\n + - * /
Term → . Term / Factor		\n + - * /
-	Go to State 9	
NUMBER	Go to State 2	
IDENT	Go to State 3	
\n	Go to State 4	
Expr	Go to State 5	
Factor	Go to State 6	
Term	Go to State 7	
(	Go to State 8	
Stmt	Go to State 10	
$	Go to State 11	
error	Go to State 12	

State 2

Factor → NUMBER .	) \n + - * /

State 3

Factor → IDENT .		\n + - * /
Stmt → IDENT . ASSIGN Expr		\n
ASSIGN	Go to State 27	

State 4

StmtList → StmtList \n .	$ error (\n - NUMBER IDENT

State 5

Expr → Expr . + Term		\n + -
Expr → Expr . - Term		\n + -
Stmt → Expr .		\n
+	Go to State 23	
-	Go to State 22	

State 6

Term → Factor .	) \n + - * /

State 7

Expr → Term .		) \n + -
Term → Term . * Factor		) \n + - * /
Term → Term . / Factor		) \n + - * /
*	Go to State 17	
/	Go to State 18	

State 8

Expr → . - Term	) + -

```
Expr → . Expr + Term                  ) + -
Expr → . Expr - Term                  ) + -
Expr → . Term                         ) + -
Factor → ( . Expr )                   ) \n + - * /
Factor → . ( Expr )                   ) + - * /
Factor → . IDENT                      ) + - * /
Factor → . NUMBER                     ) + - * /
Term → . Factor                       ) + - * /
Term → . Term * Factor                ) + - * /
Term → . Term / Factor                ) + - * /
(            Go to State    8
-            Go to State    9
Expr         Go to State    21
Factor       Go to State    6
IDENT        Go to State    15
NUMBER       Go to State    2
Term         Go to State    7
```

State 9

```
Expr → - . Term                       ) \n + -
Factor → . ( Expr )                   ) \n + - * /
Factor → . IDENT                      ) \n + - * /
Factor → . NUMBER                     ) \n + - * /
Term → . Factor                       ) \n + - * /
Term → . Term * Factor                ) \n + - * /
Term → . Term / Factor                ) \n + - * /
(            Go to State    8
Factor       Go to State    6
IDENT        Go to State    15
NUMBER       Go to State    2
Term         Go to State    16
```

State 10

```
StmtList → StmtList Stmt . \n         $ error ( \n - NUMBER IDENT
\n           Go to State    14
```

State 11

```
$START → $ StmtList $ .               $
```

State 12

```
StmtList → StmtList error . \n        $ error ( \n - NUMBER IDENT
\n           Go to State    13
```

State 13

```
StmtList → StmtList error \n .        $ error ( \n - NUMBER IDENT
```

State 14

```
StmtList → StmtList Stmt \n .         $ error ( \n - NUMBER IDENT
```

State 15

```
Factor → IDENT .                      ) \n + - * /
```

State 16

```
Expr → - Term .                       ) \n + -
```

Term → Term . * Factor	) \n + - * /
Term → Term . / Factor	) \n + - * /

*	Go to State	17
/	Go to State	18

State 17

Factor → . (Expr)	) \n + - * /
Factor → . IDENT	) \n + - * /
Factor → . NUMBER	) \n + - * /
Term → Term * . Factor	) \n + - * /

(	Go to State	8
Factor	Go to State	20
IDENT	Go to State	15
NUMBER	Go to State	2

State 18

Factor → . (Expr)	) \n + - * /
Factor → . IDENT	) \n + - * /
Factor → . NUMBER	) \n + - * /
Term → Term / . Factor	) \n + - * /

(	Go to State	8
Factor	Go to State	19
IDENT	Go to State	15
NUMBER	Go to State	2

State 19

Term → Term / Factor .	) \n + - * /

State 20

Term → Term * Factor .	) \n + - * /

State 21

Expr → Expr . + Term	) + -
Expr → Expr . - Term	) + -
Factor → (Expr .)	) \n + - * /

)	Go to State	24
+	Go to State	23
-	Go to State	22

State 22

Expr → Expr - . Term	) \n + -
Factor → . (Expr)	) \n + - * /
Factor → . IDENT	) \n + - * /
Factor → . NUMBER	) \n + - * /
Term → . Factor	) \n + - * /
Term → . Term * Factor	) \n + - * /
Term → . Term / Factor	) \n + - * /

(	Go to State	8
Factor	Go to State	6
IDENT	Go to State	15
NUMBER	Go to State	2
Term	Go to State	26

State 23

Expr → Expr + . Term	) \n + -
Factor → . (Expr)	) \n + - * /
Factor → . IDENT	) \n + - * /
Factor → . NUMBER	) \n + - * /
Term → . Factor	) \n + - * /
Term → . Term * Factor	) \n + - * /
Term → . Term / Factor	) \n + - * /

(	Go to State	8
Factor	Go to State	6
IDENT	Go to State	15
NUMBER	Go to State	2
Term	Go to State	25

State 24

Factor → (Expr) .	) \n + - * /

State 25

Expr → Expr + Term .	) \n + -
Term → Term . * Factor	) \n + - * /
Term → Term . / Factor	) \n + - * /

*	Go to State	17
/	Go to State	18

State 26

Expr → Expr - Term .	) \n + -
Term → Term . * Factor	) \n + - * /
Term → Term . / Factor	) \n + - * /

*	Go to State	17
/	Go to State	18

State 27

Expr → . - Term	\n + -
Expr → . Expr + Term	\n + -
Expr → . Expr - Term	\n + -
Expr → . Term	\n + -
Factor → . (Expr)	\n + - * /
Factor → . IDENT	\n + - * /
Factor → . NUMBER	\n + - * /
Stmt → IDENT ASSIGN . Expr	\n
Term → . Factor	\n + - * /
Term → . Term * Factor	\n + - * /
Term → . Term / Factor	\n + - * /

(	Go to State	8
-	Go to State	9
Expr	Go to State	28
Factor	Go to State	6
IDENT	Go to State	15
NUMBER	Go to State	2
Term	Go to State	7

State 28

Expr → Expr . + Term		\n + -
Expr → Expr . - Term		\n + -
Stmt → IDENT ASSIGN Expr .		\n
+	Go to State 23	
-	Go to State 22	