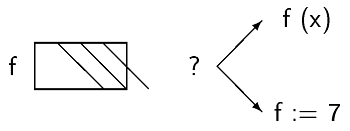
Chapter 1

Bottom Up Parsing

The key difficulty with top-down parsing is the requirement that the grammar satisfy the LL(1) property. You will recall that this entailed knowing, when you are facing the token that *begins* a production, which of many potential alternatives is the correct path to pursue. This can be a serious limitation. For example, suppose the parser is trying to match a statement, and knows only that the next token is an identifier. With only this information, the parser cannot tell whether the remainder of the statement will be an assignment or a function call:



Such issues can sometimes be finessed by factoring the grammar, as we have been able to do with Plat. However, there are situations where such a factoring is not possible, or at least not desirable. In such cases a more powerful parsing technique is called for.

We can begin our description of bottom up parsing techniques by asking why the LL(1) restriction is necessary at all. When faced with the choice of several possibilities, why not simply investigate all the alternatives in parallel? That is, if our goal is to match the nonterminal A and we have the following production:

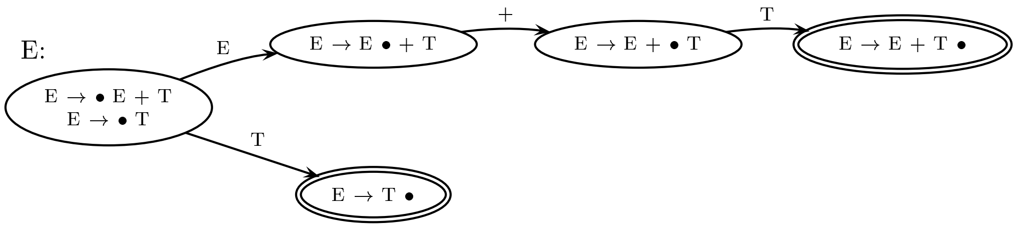
$$\mathsf{A} ::= \alpha \mid \beta \mid \gamma$$

Why can we not try matching α, β and γ in parallel, and see which one turns out to be correct? This idea is not as far fetched as it might at first seem. After all, this is exactly what a non-deterministic finite automata will do. A NDFA pursues all possible paths, abandoning those that cannot be moved forward, until only a single path emerges as the correct choice.

1.	S	::=	E	4.	T	::=	T * F
2.	E	::=	E + T	5.		|	F
3.		|	T	6.	F	::=	*id*
				7.		|	(E)

Figure 1.1: Our Sample Grammar for Bottom Up Parsing

Our belief that this might be a useful avenue to pursue is strengthened by noticing that the right hand sides of productions look very much like regular expressions. Regular expressions, we know, can always be converted into finite automata. In this fashion a finite automata for the nonterminal E in our sample grammar (Figure 1.1) could be easily described as follows:

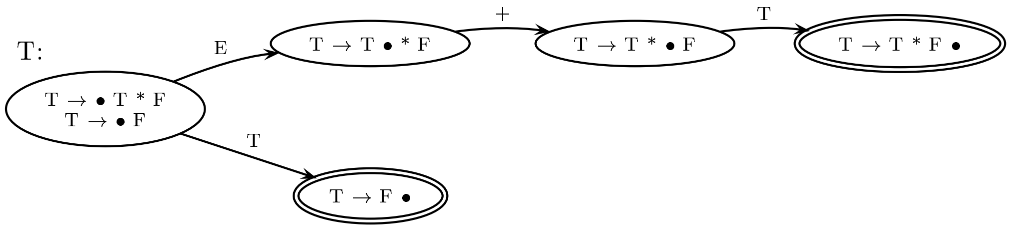


The states have been labelled in an unusual fashion, in order to more clearly illustrate the relationship between the state and the original productions. The dot is used to represent a cursor, or current position. The second state corresponds to having seen something that matches an E, and looking for a plus sign. The next state means we have seen both the E and the plus, and are looking for a T, and so on. There are two final states, which indicate success at matching an entire production.

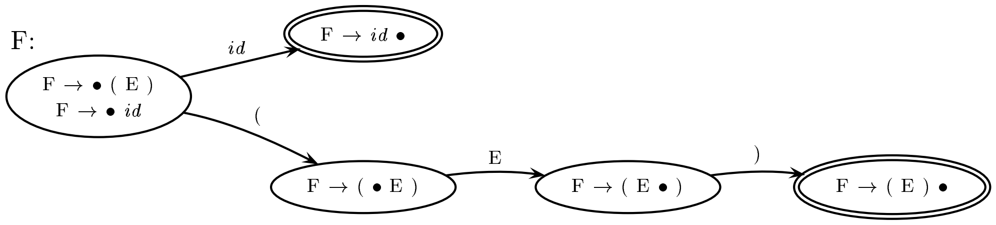
There is one major difficulty with this recognizer. The problem is the presence of arcs labelled with nonterminal symbols. In the automata we constructed for lexical analyzers, arcs were always labelled with terminals. This made it easy to imagine how such an automata could be used: simply traverse arcs until one can no longer go forward, then see if you are in a final state. Since a nonterminal, such as E, is never going to be found in the input stream, how is one going to traverse the arc labelled with this symbol?

One possible solution is to change slightly the interpretation placed on the concept of "being in a state". Imagine we have a state S and an arc that begins at S labelled with a value X. If we imagine ourselves "in" state S, rather than thinking this represents the situation where "if we see an X I am going to traverse this arc", instead think of the interpretation "when I'm in this state I'm *expecting* to see an X."

If we find ourselves in the start state in the automata given above, we are "expecting" to see either an E or a T. Ignoring the former for the moment, what would it mean to see a T? It would mean finding something that matched the specifications for a T. But what types of objects are those? The answer is that they are objects that can be recognized by the finite automata for T values:



To complete the picture, we can make a similar automata for S and F values:



Having separate automata for each production is somewhat clumsy. We can tie these automata together using epsilon arcs. You will recall that an epsilon arc "costs" nothing to traverse. So the rule is, whenever there is an arc that leads from a state that is labelled with a nonterminal, add an epsilon arc to the recognizer for that nonterminal. Self looping epsilon arcs (arcs that go nowhere and cost nothing to traverse) make little sense, and so can be omitted. The result is a non-determanistic finite automata, such as that shown in Figure 1.2. The states have been relabelled with numbers, to make the subsequent discussion easier.

But what sort of thing is this automata we have created? We know, for example, that is cannot be a recognizer for the language in question, since finite automata cannot recognize context free languages. An indeed, if we try simulating the actions of the automata on a sample input, we see that it does not recognize the language. Instead, it is a recognizer for *productions.*

To see this, imagine the input is the expression `x + y`. Our automata begins in state 0. But because of the epsilon productions, this is really the combined state (0,2,7,12). The lexical analyzer reports that the input is an identifier token. The only arc labelled with id leads us out of state 12 and into state 13. We follow this arc, and move the lexical analyzer on to the next token.

Here we come to a sudden halt. The lexical analyzier reports that the next token is a plus sign. But there is no arc labelled with a plus sign leading from the present state,

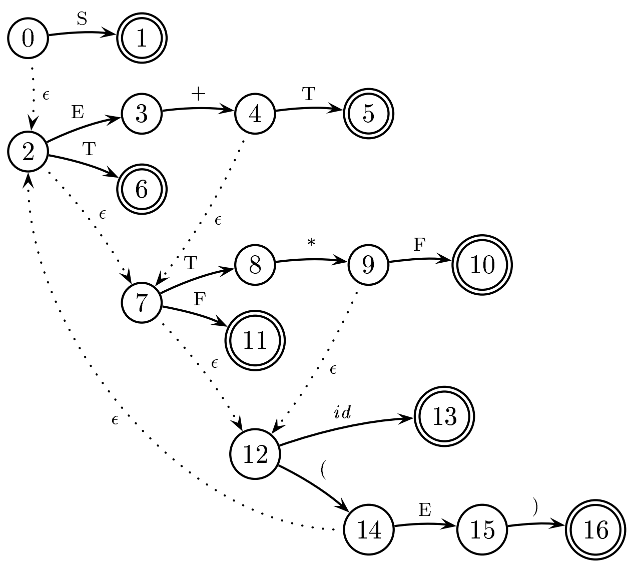


Figure 1.2: The combined NDFA for the grammar in Figure 1.1

which is state number 13. Indeed, there are no arcs labelled with anything leading from the present state. We seem to be stoped cold, after having barely started, with no obvious progress having been made.

But indeed we *have* discovered something important, if we remember that this portion of the recognizer came from the nonterminal F. The state we find ourselves in, state 13, was the final state for the automata that was designed to recognize production 6 from our grammar. We have thus succeeded in finding something that matches the description of an F. This tells us that a bit of the parse tree, indeed the bottom and left most part of the parse tree, will involve the nonterminal F.

Now comes the key insight. We got to where we are because earlier we were in a state (state 7, to be precise) where we were hoping to see a nonterminal F. We have now seen such a value. So let us *back up* the finite automata along the path we have travelled, and then go forward along an arc labelled with F. The latter is legal, since indeed we have seen an F value in the input. We back up to the combined state (0,2,7,12), then move forward along the arc labelled F, which takes us to state 11.

The algorithm that embodies these ideas is shown in Figure 1.3. A stack is used by the algorithm to remember the path that has been followed in traversing the finite automata. Backing up can then be performed by simply popping values from the stack. Note that the body of the algorithm consists of two major tasks, either a shift in the automata (which consumes a lexical token), or a reduce (which corresponds to having recognized a production).

```
Push start state on to stack
Repeat until starting production is reduced
    Let S be the current state (top of stack)
    Let x be the current token
    do one of the following:

    shift:
        traverse arc from S labelled x
        advance lexical analyzer to next token
        push new state on to stack

    reduce:
        if in final state for some production:
        let n be the number of symbols on
        right hand side of production.
        pop stack n times, moving backwards through automata
        move forward along arc labelled with
        nonterminal for production
        push new stack on to stack

if cannot do either, report parse error
```

Figure 1.3: The Basic LR shift/reduce algorithm

Note that in performing the latter, we must back up to the point where we *started* to look for the corresponding production. To determine this point, we simply count the number of symbols in the right hand side of the production. Each symbol, whether terminal or nonterminal, corresponds to an arc. To back up three symbols, for example, three locations are popped from the stack.

There are many issues left unanswered by this description of the process. A significant omission is the question of what should happen if it is possible to both shift and reduce, or if two final states in the NDFA map into one state in the DFA, resulting in two possible reduction actions for the same state. We will return to these questions shortly, after we complete this example.

Before returning to our example, let us convert the NDFA we built in Figure 1.2 into a DFA. The result is shown in Figure 1.4. The states have been renumbered, but the final states for each production are preserved, and are shown in the Figure. The steps taken by the algorithm in Figure 1.3 in recognzing the input `x + y` are as follows:

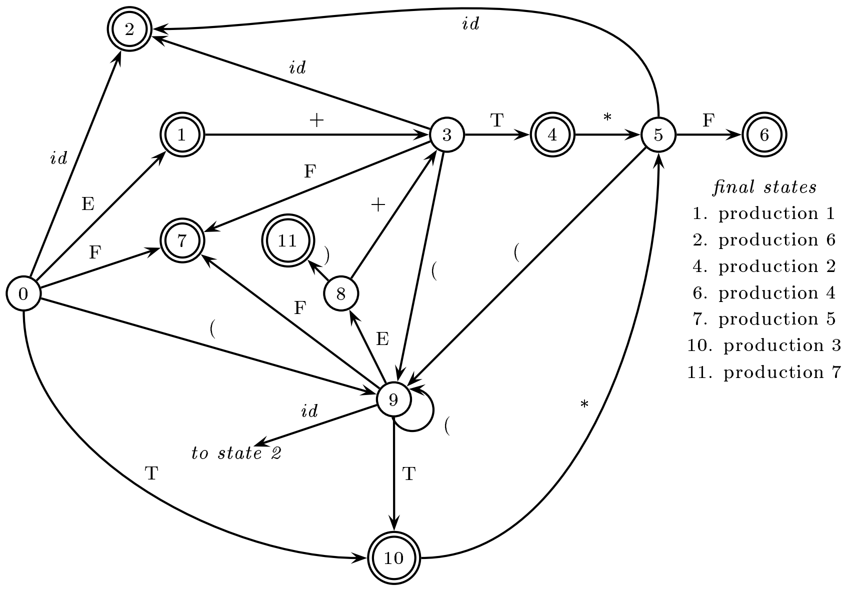


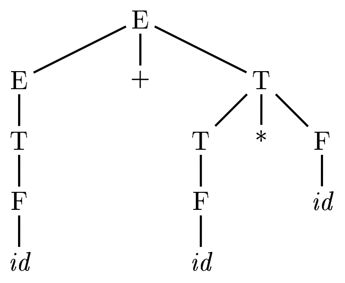
Figure 1.4: The DFA corresponding to Figure 1.2.

stack	input	action
0	`x + y * z`	shift into state 2
0,2	`+ y * z`	reduce F ::= id
0,7	`+ y * z`	reduce T ::= F
0,10	`+ y * z`	reduce E ::= T
0,1	`+ y * z`	shift into state 3
0,1,3	`y * z`	shift into state 2
0,1,3,2	`* z`	reduce F ::= id
0,1,3,7	`* z`	reduce T ::= F
0,1,3,4	`* z`	shift into state 5
0,1,3,4,5	`z`	shift into state 2
0,1,3,4,5,2	`<end>`	reduce F ::= id
0,1,3,4,5,6	`<end>`	reduce T ::= T * F
0,1,3,4	`<end>`	reduce E ::= E + T
0,1	`<end>`	reduce S ::= E
		finished

Parsing ends when we have successfully matched the first production, S ::= E, finding ourself in the end state for this production, state 1 in this example, and there are no more input tokens.

1.1 Bottom Up versus Top Down

Consider the order in which productions are discovered by the technique just described. These are summarized below, along with the parse three for the sample expression.



F ::= *id*
T ::= F
E ::= T
F ::= *id*
T ::= F
F ::= *id*
T ::= T * F
E ::= E + T
S ::= E

There are two important features to note. The first is that the productions discovered are all from the parse tree, and furthermore are the productions in the parse tree read from the bottom to the top and left to right. Hence the term bottom up parsing.

The second key feature concerns *when* a production is recognized. Using this technique, productions are recognized after the fact, after all the input that matches the production has been seen already. The R in LR parsing describes this, as it denotes the fact that productions are recognized on the Right hand side, rather then on the Left hand side of LL parsing. This is rather like riding in a rear-facing seat in a train, in the sense that you see where you have been more clearly then where you are going. But this fact is important, and it is the key to understanding why LR parsers are more powerful than LL parsers. LR parsers do not predict what input they expect to see in the future, as do LL parsers, but rather simply classify the tokens they have already seen. Because LR parsers do not recognize productions until after all the matching tokens have been scanned, the decision that was so troubling in LL parsing, deciding which alternative to pursue, simply does not arize. We have achineved the goal we set out in the beginning of the chapter; LR parsers do pursue multiple paths in parallel, selecting the appropriate production only after all the necessary information is available.

Notice that left recursion is not a problem for an LR parser, as it is for an LL parser. Neither is right recursion. LR parsers work with a much larger set of grammars than do LL parsers. In fact, it can be demonstrated that if a context free grammar can be recognized deterministically at all (that is, without requiring the lexical analyzer to back up to previous tokens) then some form of LR parsing can recognize the grammar. (We will have more to say about the various different froms of LR parsing in a later section).

1.2 Adding Attributes to the Parse Tree

Recall that the goal of a parser is to verify that a parse tree exists, by making a systematic traversal of the parse tree. A top-down (or LL) parser traverses the tree from the top to the bottom, starting at the root node and considering each child node, left to right, in turn. In contrast, a LR parser performs the traversal walking over the parse tree in a pre-order fashion, bottom up and left to right. As you might expect, inherited attributes (information passed down the parse tree) is rather difficult to handle in this fashion, because the root node is not encountered until after all the child nodes have been processed. However, synthasized attributes (information passed up the parse tree) can be easily combined with the parsing process.

To do this, we modify the stack used in Algorithm 1.3 to hold not only state numbers, but also attribute information. Each entry in the stack will be a state/attribute pair. The stack is manipulated in both the shift and reduce actions. For a shift, the attribute will be whatever information is returned by the lexical analyzer for the token, typically just the text of the token. The combination of the state the arc moves into and the lexical analyzier attribute is then pushed on the stack.

Reductions are performed when a completed production has been recognized. To compute attributes, we simply add semantic actions to our description of productions. These actions are analogous to the semantic actions used in the lexical analysis generator described in Chapter **??**, and operate in the same fashion: when the pattern described by the production rule is found (this time, in its proper place in the parse tree), the semantic action is performed.

```
S ::= E
    { if (! lex.atEnd())
        throw new ParseException("expecting end of file");
      return; }

E ::= E + T
    { $$ = new BinaryNode("+", $1, $3); }

  | T
    { $$ = $1; }

T ::= T * F
    { $$ = new  BinaryNode("*", $1, $3); }

  | F
    { $$ = $1; }

F ::= id
    { $$ = new IdNode($1); }

  |  ( E )
    { $$ = $2; }
```

Figure 1.5: Semantic Actions for Sample Grammar

The semantic action can be any statement permitted by the underlying programming language (the language the parser is programmed in). In addition, the action can have access to the attribute values associated with the symbols in the right hand side of the production being reduced. Typically access is provided to these values through pseudo-variables, values that can be manipulated like variables but that need not be declared. A common convention is to use the dollar sign to indicate such a pseudo-variable, using $\$n$ to represent the attribute of the n^{th} symbol, and $$ to represent the value of the new attribute being generated.

Figure 1.5 shows our sample grammar augmented with semantic actions. Here the purpose of the semantic actions is to build an abstract syntax tree that represents the expression being parsed. For this purpose two procedures are used. The class IdNode represents an identifier node, while the class BinaryNode represents a binary operator. The constructor for the former takes as argument the name of the identifier (as yielded by the lexical analyzer) while the constructor for the latter takes as argument a string representing the operation being performed, and the abstract syntax trees for the two child nodes.

More here.

1.3 Tabular Representation in Java

The finite automata used by the LR parsing algorithm can be represented in Java using a variation on the tabular representation for finite automata described earlier in Chapter ??. There are two important differences from the earlier scheme. First, both terminals and nonterminals are considered as potential input values; in essence, nonterminals can be considered as a type of token. Second, the entries in the table must now encode either the possibility for a shift or for a reduce. If we use integer values for the table entries, we can use the positive/negative distinction to separate the two. A zero or positive value will represent a shift, which is just a normal transition in the finite automata. A negative value will represent a reduction, where the absolute value of the number represents the number of the production being reduced.

Using this encoding, the tabular representation for the automata shown in Figure 1.4 could be given as follows:

state	*id*	(	+	*	)	<end>	E	T	F
0	2	9	0	0	0	−1	1	10	7
1	0	0	3	0	0	0	0	0	0
2	0	0	−6	−6	0	−6	0	0	0
3	2	9	0	0	0	0	0	4	7
4	0	0	−2	5	0	−2	0	0	0
5	2	9	0	0	0	0	0	0	6
6	0	0	−4	−4	0	−4	0	0	0
7	0	0	−5	−5	0	−5	0	0	0
8	0	0	3	0	11	0	0	0	0
9	2	9	0	0	0	0	8	10	7
10	0	0	−3	5	0	−3	0	0	0
11	0	0	−7	−7	0	−7	0	0	0

Examining the algorithm described in Figure 1.3, there are 3 pieces of information we require for each production. With some effort, these can each be encoded as an integer value. The three values required represent the number of symbols on the right hand side of the production, the nonterminal the production is recognizing, and the semantic action to be performed when the production is reduced. The latter can be converted into an integer by enclosing all the semantic actions in a single large switch statement, and simply recording the case index corresponding to the appropriate action. Using this scheme, the productions for our sample grammar can be encoded as three integer arrays, as follows:

prodSize	0	1	3	1	3	1	1	3
prodTarget	0	S	E	E	T	T	F	F
prodAction	0	1	2	3	4	5	6	7

Using these tables, we can now describe how to rewrite in Java the pseudo-code algorithm presented in Figure 1.3. The class Parser embodies the parsing algorithm. Two data fields in this class are the transition table and the production table. These two fields can be declared as static, and initialized in the class body. This initialization is omitted from the figure. The actual parsing is performed by the method parse.

The stack used by the parse routine is maintained as a local variable inside the method parse. Shift and reduce actions are performed as directed by the input and the transition table. The values placed on the stack are state and attribute pairs, described by the following class:

```
class pair {
    public int state;
    public Object attribute;
    public pair (int s, Object a) { state = s; attribute = a; }
    }
```

The semantic actions will be implemented using a switch statement. Semantic actions are simply copied directly from the description file, making various trivial textual changes. References to the pseudo-variables $\$n$ will be replaced by index expressions for the array named attributes (the index value being reduced by one to reflect the fact that index values are one-based, while arrays in Java are zero-based). The pseudo-variable $$ will be replaced by the variable newAttribute. Each new semantic action is given a unique integer code.

The control over the major loop for the program can be simplified by reserving semantic action 0 to represent a parse error. Undefined entires in the tabular representation can then be set to "reduce" using this rule. In Exercise 1 we investigate another semantic action we might want to preassign, namely a default action for productions that do not provide any alternative. Figure 1.7 illustrates the encoding of the semantic actions of the rules shown earlier in Figure 1.5.

1.4 Building the DFA directly

In the beginning of this chapter we noted how each state in the nonterminal can be identified with a production and a cursor, or current location. This idea provides the insight necessary to see how one can construct the LR parser DFA directly from the grammar, rather than first constructing the nondetermanistic finite automata.

We know that initially we have not yet seen any input, and thus our objective is the starting production, and the cursor is to the left of the production, as in S ::= • E. Because the cursor is to the left of a nonterminal, we are expecting to match values that derive in the parse tree from that nonterminal. This is the reason why the state in tied by epsilon arcs, in the NDFA shown in Figure 1.2, to the recognizer for E values. But anything that matches an E must derive from one of the productions for nonterminal E. Rather than explicitly making the epsilon arcs, and then tying states together when the NDFA is made deterministic, we can simply relabel the start state with the combined productions, S := •

```
class Parser {
    static int transitionTable [ ] [ ];
    static int prodSize [ ], prodAction [ ], prodTarget[ ];

    public void function parse () throws ParseException {
        Stack stateStack = new Stack();
        int currentState = 0; // initial state
        stateStack.push (new pair(currentState, null));
        while (true) {
            int currentToken = lex.tokenType();
            int action = transitionTable[currentState][currentToken];
            if (action >= 0) { // shift
                stateStack.push(new pair(currentState, lex.tokenText()));
                currentState = action; // move to new state
                lex.nextLex(); // advance lexer to next symbol
                }
            else { // reduce
                action = - action;
                int n = prodSize[action];
                attributes = new Object[n];
                for (int i = n-1; i >= 0; i--) {
                    Pair p = stateStack.pop();
                    attributes[i] = p.attribute;
                    }
                    // find state we have reduced to
                Pair p = stateStack.peek();
                currentState = p.state;
                    // then advance
                currentState =
                    transitionTable[currentState][prodTarget[action]];
                Object newAttribute = null;
                switch (prodAction[action]) {
                    case 0:
                        // ...
                    }
                stateStack.push(new pair(currentState, newAttribute));
                }
            }
    }
}
```

Figure 1.6: The LR Parsing Algorithm rewritten in Java

```
switch (prodAction[action]) {
    case 0: // parse error
        throw new ParseException("parse error");

    case 1:         // production  S ::= E
        if (lex.tokenType() != -1)
            throw new ParseException("expecting end of file");
        return;

    case 2:         // production  E ::= E + T
        newAttribute = new BinaryNode("+", attribute[0], attribute[1]);
        break;

    case 3:         // production  E ::+ T
        newAttribute = attribute[0];
        break;

    case 4:         // production T ::= T * F
        newAttribute = new BinaryNode("*", attribute[0], attribute[1]);
        break;

    case 5:         // production T ::= F
        newAttribute = attribute[0];
        break;

    case 6:         // production F ::= id
        newAttribute = new IdNode(attribute[0]);
        break;

    case 7:      // production F ::= ( E )
        newAttribute = attribute[1];
        break;
}
```

Figure 1.7: Switch statement for semantic actions shown in Figure 1.5

```
Let S be a set of productions augmented with cursors
For each nonterminal X that follows a cursor
    add all productions X -> $\bullet$ $\alpha$ to S
Repeat until S ceases to grow
```

Figure 1.8: The state closure algorithm

E, E ::= • T, and E ::= • E + T. But again the cursor sits in front of a nonterminal symbol, namely T. This indicates that we are expecting to find a T object. Since a T must match one of the productions T ::= • F, or T ::= • T * F, we add these two productions to the set. But finally we again have a cursor in front of the nonterminal F, so we must add those two symbols to the set. The final combined label for the state is as follows:

picture

The reader should verify that this corresponds to the combined state (0,2,7,12) in the NDFA shown in Figure 1.2. All we have done is to simply bring together the states marked with epsilon arcs. The algorithm we have be using is called the *closure* algorithm, and is described in Figure 1.8.

The arcs that will lead from a set are easily discovered by simply enumerating the symbols that are to the right of the cursor marks. Traversing an arc will simply move the cursor over the symbol. Often two or more productions will have the same symbol to the right of the cursor, and these are combined into a single state. Performing this action on our sample grammar yields the following new states:

picture

The process of enumerating the new states that will follow a state is called the *goto* step. Following the goto, it is necessary to form a closure of each of the new sets. The closure of the state labelled with E ::= (• E), for example, will convert it into the following state:

picture

The process of performing closures and goto's is repeated for each state until no new states can be generated. The result will be the automata similar to that shown in Figure 1.4, but derived without the necessity of first forming the nondetermanistic automata. Final states in the automata are easily recognized, as they are states that contain a production in which the cursor appears on the far right of the production.

1.5 Varieties of LR parsers

1.6 A Parser Generator

Exercises

1. Reserving $$ = $1 as a default action.