Assignment No.8

AIM:

Implement the top-down parsing using predictive parsing technique.

THEORY:

Predictive parser

In many cases, by carefully writing a grammar, eliminating left recursion from it, and left factoring the resulting grammar, we can obtain a grammar that can be parsed by a recursive-dcscent parser that needs no backtracking, i.e., a predictive parser.

To construct a predictive parser, we must know, given the current input symbol a and the nonterminal A to be expanded, which one of the alternatives of production A → α1| α2|…| αk is the unique alternative that derives a string beginning with a. That is, the proper alternative must be detectable by looking at only the first symbol it derives. Flow-of-control constructs in most programming languages, with their distinguishing keywords, are usually detectable in this way.

For example, if we have the following productions

stmt → **if** *expr* **then** *stmt* **else** *stmt*
| **while** *expr* **do** *stmt*
| **begin** *stmt_list* **end**

then the keywords ***if, while***, and ***begin*** tell us which alternative is the only one that could possibly succeed if we are to find a statement.

Transition diagram for predictive parser:

In parser, there is one diagram for each nonterminal. The labels of edges are tokens and non-terminals. A transition on a token (terminal) means we should take that transition if that token is the next input symbol. A transition on a non-terminal A is a call of the procedure for A.

To construct the transition diagram of a predictive parser from a grammar, first eliminate left recursion from the grammar and then left factor the grammar. Then for each nonterminal A do the following

- Create an initial and final state
- For each production A → X1X2...Xn, create a path from initial top the final state with edges labeled X1, X2, ... Nn.

Predictive parser behaves as follows:

- It begins in the start state from the start symbol. If after some action, it is in state *s* with an edge labeled by terminal *a* to state *t* and if the next input symbol is *a* then the parser moves the input cursor one position right and goes to state *t*.
- If edge is labeled by a nonterminal *A* then parser goes to the start state for *A* without moving the input cursor. If it ever reaches the final state for *A*, it immediately goes to state *t*, in effect having "read" A from the input during the time it moved from state s to t.
- If there is an edge from *s* to *t* labeled Ɛ, then from state s the parser immediately goes to state without advancing the input.
- A predictive parsing program based on a transition diagram attempts to match terminal symbols against the input, and makes a potentially recursive procedure call whenever it has to follow an edge labelled by a nonterminal.

Figure contains a collection of transition diagram for given grammar. The only ambiguities concern whether or not to take an epsilon edge. If we interpret the edges out of the initial state for E' as saying take the transition on + whenever that is the next input and take the transition on epsilon otherwise, and make the analogous assumption for T', then the ambiguity is removed, and we can write a predictive parsing program for grammar

Example:

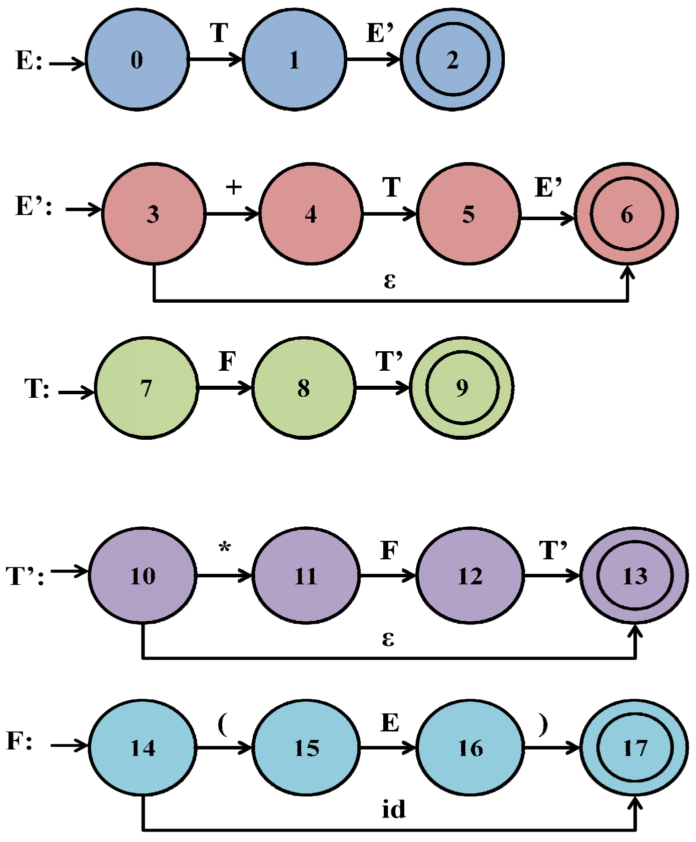
E → T E’

E’ → +T E’ | ε

T → F T’

T’ → *F T’ | ε

F → id | (E)



Transition diagrams can be simplified by substituting diagrams in one another. For example, in Fig. a, the call of E' on itself has been replaced by a jump to the beginning of the diagram for E' .

Figure b shows an equivalent transition diagram for E'. We may then substitute the diagram of Fig. b for the transition on E' in the transition diagram for E yielding the diagram of Fig.c Observe that the first and third nodes in Fig.c arc equivalent and we merge them resulting in fig d. The same techniques apply to the diagrams for T and T'.

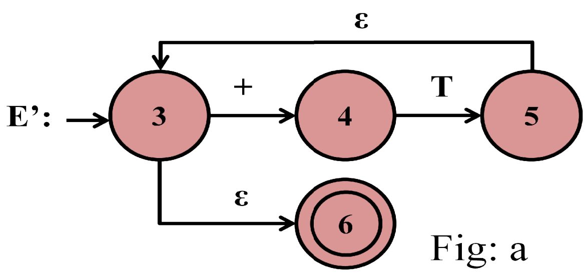


Fig: a

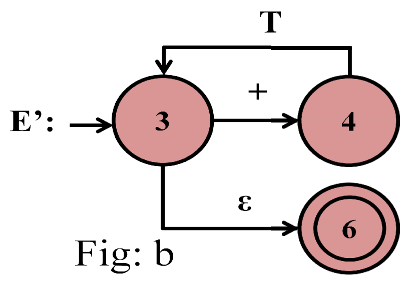


Fig: b

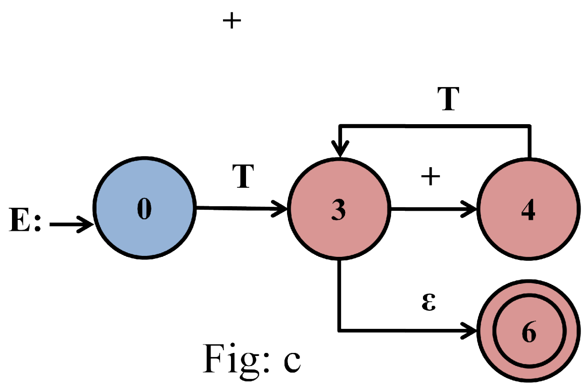


Fig: c

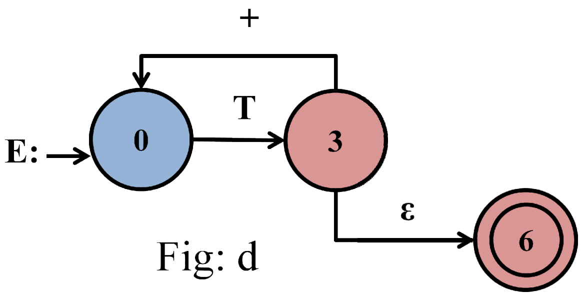
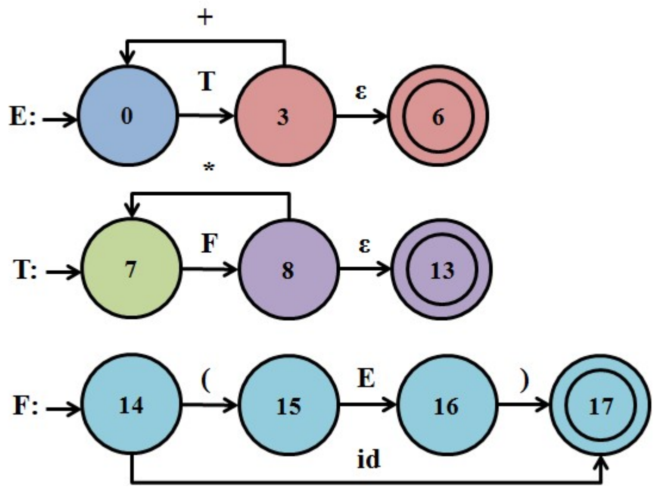


Fig: d

Now how to parse the input string using predictive parser: Here input string is id+id

- Step 1: First input symbol is id, hence input cursor points to id. So consider the first transition diagram that is for non-terminal E. There is transition from state 0 to 3 on non-terminal T, so parser goes to start state for T without moving input cursor. In transition diagram for T, there is transition from state 7 to 8 on non-terminal F, so parser goes to start state for F without moving input cursor. In transition diagram for F, there is state 14 with an edge labeled by terminal id to state 17 and the input symbol is also id so the parser moves the input cursor to next input symbol + and goes to final state 17. Now parser goes back from state 17 to 8 and from 8 to final state 13 on symbol Ɛ. From state 13, parser goes back to state 3.

- Step 2: Now input cursor is pointing to +, hence parser goes from state 3 to 0 on input symbol +. Now the parser moves the input cursor to the next input symbol id.
- Step 3: Parser repeat the **step 1** for input symbol id and finally goes to final state 6 from state 3 on Ɛ.



PROGRAM:

INPUT:

1. Enter string: i+i*i+i
2. Enter string: i+i*

OUTPUT:

1. Output is

```
Call E
Call T
Call F
Terminal i
Call t
Call e
Terminal +
Call T
Call F
Terminal i
Call t
Terminal *
Call F
Terminal i
Call t
Call e
Terminal +
Call T
Call F
Terminal i
Call t
Call e
Valid input
```

2. Output is

```
Call E
Call T
Call F
Terminal i
Call t
Call e
Terminal +
Call T
Call F
Terminal i
Call t
Terminal *
Call F
Invalid input
```

CONCLUSION:

Thus the recursive descent parsing which requires backtracking is implemented.

REFERENCES:

- Compilers - Principles, Techniques and Tools - A.V. Aho, R. Shethi and J. D. Ullman (Pearson Education)