Compiler Construction

Lecture 6 - An Introduction to Bottom-Up Parsing

Bottom-up Parsing

- Bottom-up parsers parse a programs from the leaves of a parse tree, collecting the pieces until the entire parse tree is built all the way to the root.
- Bottom-up parsers emulate pushdown automata:
  - requiring both a state machine (to keep track of what you are looking for in the grammar) and a stack (to keep track of what you have already read in the program).
  - making it fairly easy to automate the process of creating the parser
  - ensuring that all context-free grammars can be parsed by this method.

Bottom-up parsers as shift-reduce parsers

- Bottom-up parsers are frequently called shift-reduce parsers because of their two basic operations:
  - A shift involves moving pushing the current input token onto the stack and fetching the next input token.
  - A reduce involves popping all the variables that comprise the right-sentential form for a nonterminal and replacing them on the stack with the equivalent nonterminal that appears on the left-hand side of that production.
  - While shifting involve pushing and reducing involve popping, do not think of them as equivalent: a shift also involve advancing the input token stream and a reduce involves zero or more pops followed by a push.

Bottom-up Parsing as an Emulation of Pushdown Automata

- Most bottom-up parsers are table-driven, with the table encoding the necessary information about the grammar.
- The parser decides what action to perform based on the combination of current state and current input token.
- A state in the machine which the computer is emulating reflects both what the machine has already parsed and that which it is expect to see in the input token stream.
- Several parser generators have been created based on this theoretical machine, the best known of which is ***YACC*** (***Y***et ***A***nother ***C***ompiler ***C***ompiler), is available on many UNIX system and its public domain lookalike ***Bison***.

LR(k) grammars

- Bottom-up grammars are referred to as LR(k) grammars:
  - – The first L indicates ***L***eft-to-Right scanning.
  - – The second L indicates ***R***ight-most derivation
  - – The k indicates k lookahead characters.
- There should be no need for anything more than a single lookahead, i.e, an LR(1) grammar.

An example - a LR(0) grammar

An LR(0) grammar does not use a lookahead character to determine the action that it will take - the current token will be used to determine the state into which it will go.

Consider the following grammar:

E ::= *E* **+** *T* | *T*

T ::= **+** *F* | **-** *F* | *F*

F ::= **id** | **const**

An example - a LR(0) grammar (continued)

Let’s write out our grammar and add to it a special first production with a special start symbol *S:*

1 S ::= E $ (indicates that the expression is followed by EOF)

2 E ::= E + T

3 E ::= T

4 T ::= +F

5 T ::= -F

6 T ::= F

7 F ::= id

8 F ::= const

The LR(0) parse table

		GOTO							
state	ACTION	+	-	id	const	$	E	T	F
0	s	4	5	6	7		1	2	3
1	s	8				12			
2	r3								
3	r6								
4	s			6	7				9
5	s			6	7				11
6	r7								
7	r8								
8	s	4	5	6	7			10	3
9	r4								
10	r2								
11	r5								
12	acc								

Tracing LR(0) parsing

There are 3 parsing operations:

Shift - moving a token and state onto the stack (we find the state using the GOTO table).

Reduce n - we pop enuogh items from the stack to form the right side of production n and then we push the nonterminal on its left side of production n on to thestack, together with thestate indicated by the GOTO table

Accept - we accept the program as completely and correctly parsed and terminate execution.

Tracing LR(0) parsing - an example

Example - the expression ***-27 + x***

5	-
0	$

We place the state 0 and the EOF marker **$** on the stack. The action for state 0 is ***shift***. We place the - and GOTO(0, -) = 5 on the stack

7	const
5	-
0	$

The action for state 5 is ***shift***. We place the constant on the stack together with GOTO(5, const) = 7.

11	F
5	-
0	$

The action for state 7 is reduce by production 8. Pop the const (and state 7). Push F and GOTO(5,F) = 11

Tracing LR(0) parsing - an example (continued)

2	T
0	$

The action for state 11 is reduce by production 5. Pop the - and F (along with states 5 and 11) and push the T together with GOTO(0,T) = 2

1	E
0	$

The action for state 2 is reduce by production 3. Pop the T (and state 2). Push the E and GOTO(0,E) = 1.

8	+
1	E
0	$

The action for state 1 is shift. We move the + onto the stack together with GOTO(1, +) = 8.

Tracing LR(0) parsing - an example (continued)

6	id
8	+
1	E
0	$

The action for state 8 is shift. We move the id and GOTO(8, id) = 6 onto the stack.

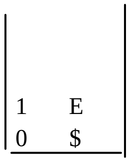
3	F
8	+
1	E
0	$

The action for state 6 is reduce by production 7. We pop the id and state 6. We push F and GOTO(8, F) = 3

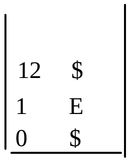
10	T
8	+
1	E
0	$

The action for state 3 is reduce by production 6. We pop the F and state 3. We push T and GOTO(8, T) = 10.

Tracing LR(0) parsing - an example (continued)



The action for state 10 is reduce by production 2. We pop the T (and state10), the + (and state8) and the E (and state1). We push the E and GOTO(0,E) = 1.



The action for state 1 is shift. We push the $ and GOTO (1,E) = 12 onto the stack.

The action for state 12 is accept. The only item on the stack (excluding the $s) is E, which is the start symbol in our expression grammar

---

Right sentential forms

- A right sentential form is a partially formed sentence (or program). It can contain the variables on the right- hand side of a production or phrases derived from it.
- Right sentential forms are derived from the rightmost derivation.
- Formally, if S $=>^{*}$ β, then β is a right sentential form.

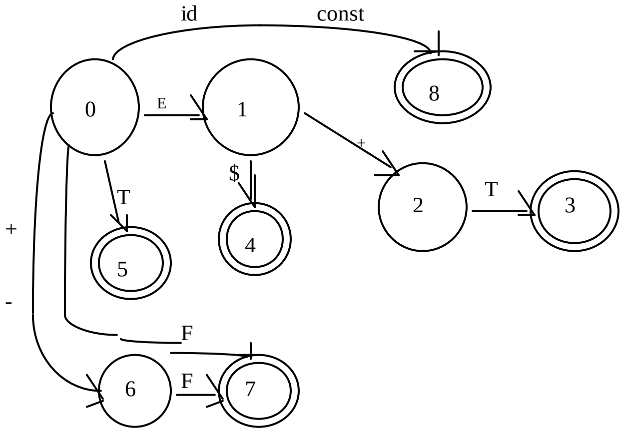
Handles

- In performing a reduce operation, we must decide which variables in a right-sentential form will be popped and replaced on the stack by the nonterminal on the production's left-hand side. These variables are collectively called the ***handle***.
- If A => β, then β would be handle for the production.

Items

- An item is a production, with a dot added to it indicating how much of the production has been matched up so far.
- Example:
  - E ::= . E + T ***nothing in the production has been matched yet.***
  - E ::= E + . T ***we have matched the E and the +***

What we would expect to the State Machine to look like



Constructing the State Machine

- We already know that processing context-free languages requires a pushdown automaton.
- As we prepare to match tokens in the item

  S ::= .E$

  we have no way of knowing what collection of tokens represent E
- We will have to consider all possible ways of representing an expression:

  E ::= .E + T

  E ::= .T

Constructing the State Machine (continued)

- Since matches a collection of tokens to E may mean matching it to T, we must know what to look for here as well:

  T ::= .+ F

  T ::= . - F

  T ::= .F

Constructing the State Machine (continued)

- Since matches a collection of tokens to T may mean matching it to F, we must know what to look for here as well:

  F ::= .id

  F ::= .const

  Since we know exactly how to match id and const to tokens (since they are terminals), we don't need any additional items.

Constructing the State Machine's Initial State

0

S ::= . E$

State 0 always contains an item showing the special start symbol deriving the regular start symbol followed by EOF

Constructing the State Machine's Initial State

0

S ::= . E $

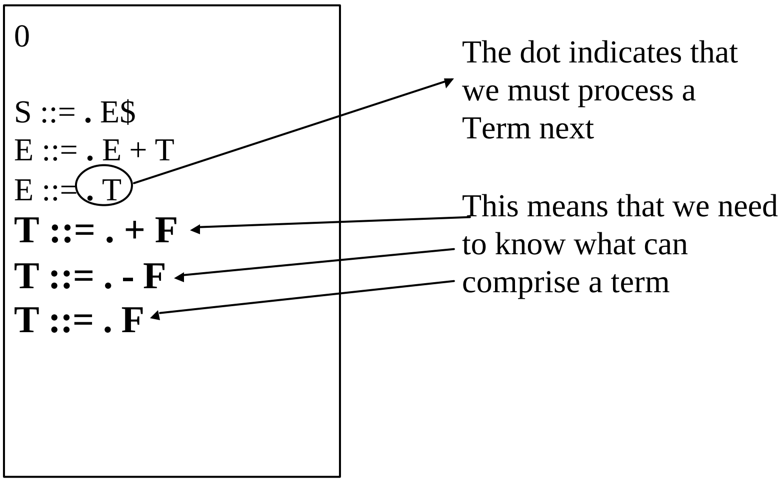
E ::= . E + T

E ::= . T

The dot indicates that we must process an Expression next

This means that we need to know what can comprise an expression

Constructing the State Machine’s Initial State



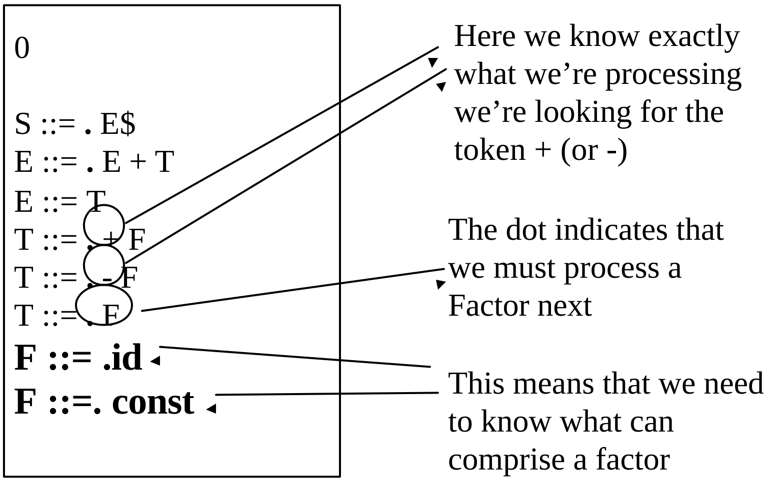
0

S ::= **.** E$
E ::= **.** E + T
E ::= **.** T
T ::= . + F
T ::= . - F
T ::= . F

The dot indicates that we must process a Term next

This means that we need to know what can comprise a term

Constructing the State Machine’s Initial State



0

S ::= **.** E$
E ::= **.** E + T
E ::= T
T ::= **.** + F
T ::= **.** - F
T ::= **.** F
F ::= .id
F ::=. const

Here we know exactly what we’re processing we’re looking for the token + (or -)

The dot indicates that we must process a Factor next

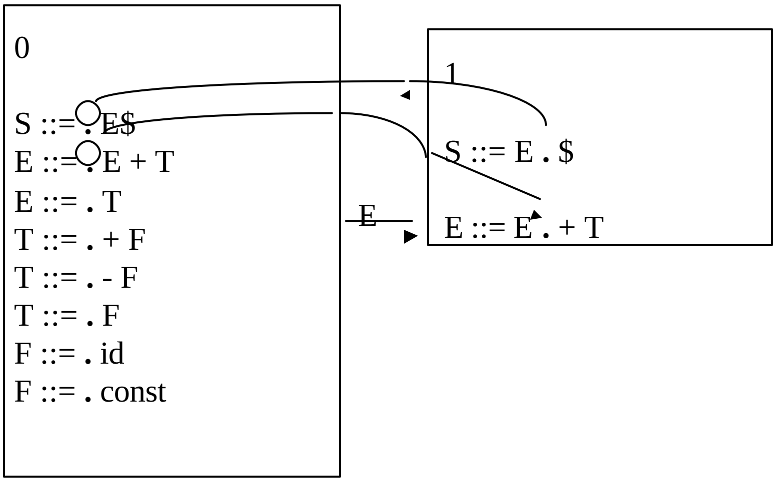
This means that we need to know what can comprise a factor

The LR(0) State Machine

0

S ::= . E$
E ::= . E + T
E ::= . T
T ::= . + F
T ::= . - F
T ::= . F
F ::= . id
F ::= . const

Constructing The Next Set of States



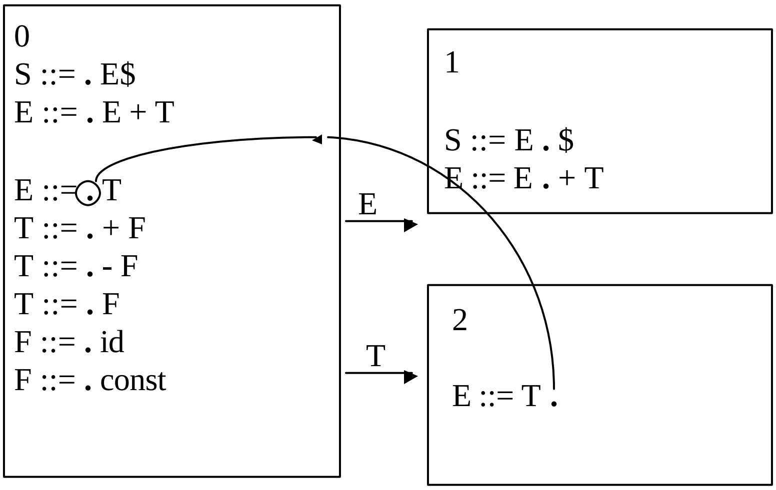
0

S ::= . E$
E ::= . E + T
E ::= . T
T ::= . + F
T ::= . - F
T ::= . F
F ::= . id
F ::= . const

1

S ::= E . $
E ::= E . + T

Constructing The Next Set of States



0
S ::= . E$
E ::= . E + T

E ::= . T
T ::= . + F
T ::= . - F
T ::= . F
F ::= . id
F ::= . const

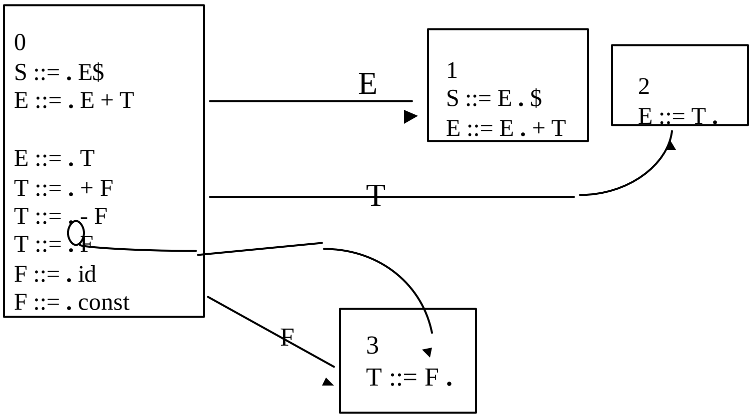
1

S ::= E . $
E ::= E . + T

2

E ::= T .

Constructing The Next Set of States



0
S ::= . E$
E ::= . E + T

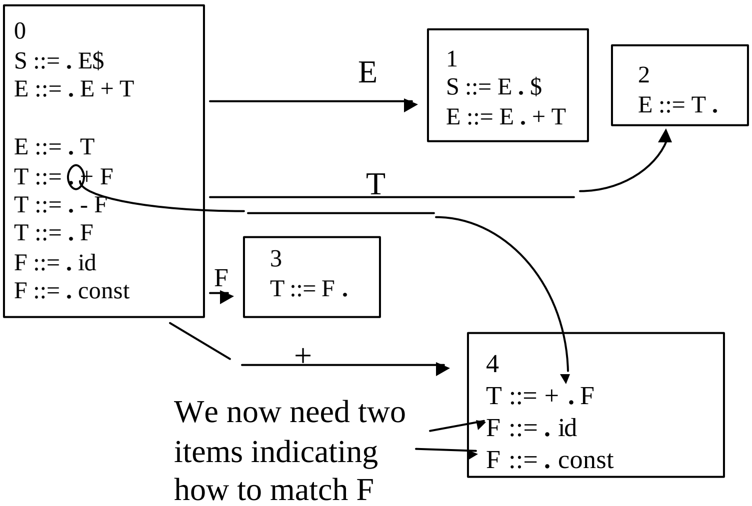
E ::= . T
T ::= . + F
T ::= . - F
T ::= . F
F ::= . id
F ::= . const

1
S ::= E . $
E ::= E . + T

2
E ::= T .

3
T ::= F .

Constructing The Next Set of States



0
S ::= . E$
E ::= . E + T

E ::= . T
T ::= . + F
T ::= . - F
T ::= . F
F ::= . id
F ::= . const

1
S ::= E . $
E ::= E . + T

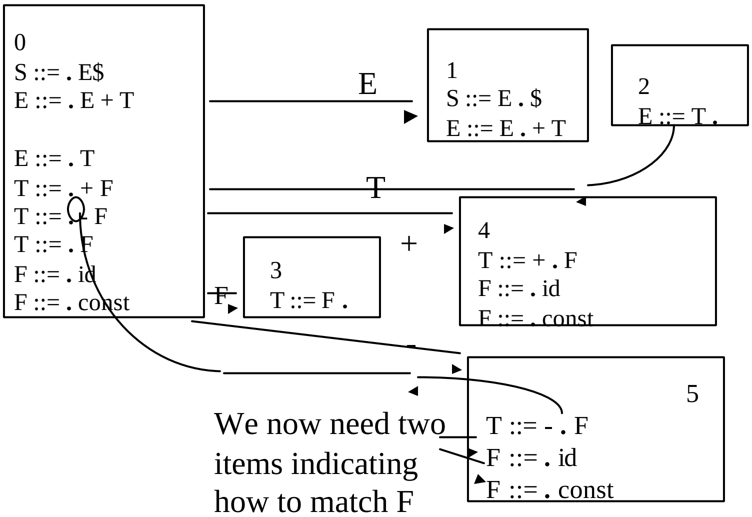
2
E ::= T .

3
T ::= F .

4
T ::= + . F
F ::= . id
F ::= . const

We now need two items indicating how to match F

Constructing The Next Set of States



0
S ::= . E$
E ::= . E + T

E ::= . T
T ::= . + F
T ::= . - F
T ::= . F
F ::= . id
F ::= . const

1
S ::= E . $
E ::= E . + T

2
E ::= T .

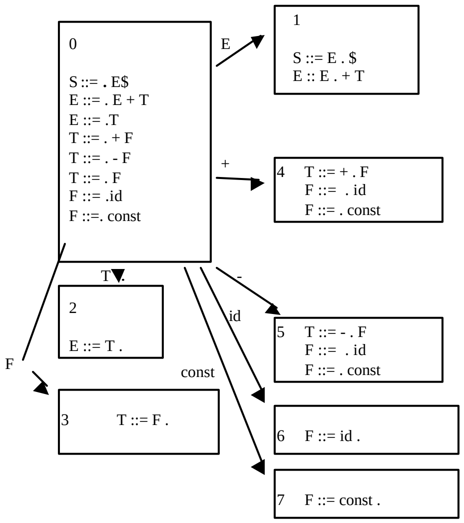
3
T ::= F .

4
T ::= + . F
F ::= . id
F ::= . const

5
T ::= - . F
F ::= . id
F ::= . const

We now need two items indicating how to match F

The LR(0) State Machine



0

S ::= . E$
E ::= . E + T
E ::= .T
T ::= . + F
T ::= . - F
T ::= . F
F ::= .id
F ::=. const

1

S ::= E . $
E :: E . + T

2

E ::= T .

3 T ::= F .

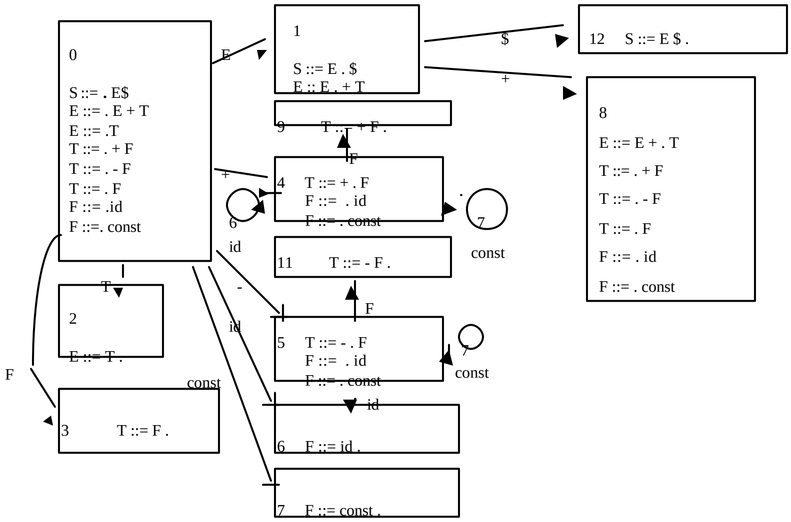
4 T ::= + . F
F ::= . id
F ::= . const

5 T ::= - . F
F ::= . id
F ::= . const

6 F ::= id .

7 F ::= const .

The LR(0) State Machine



0

S ::= . E$
E ::= . E + T
E ::= .T
T ::= . + F
T ::= . - F
T ::= . F
F ::= .id
F ::=. const

1

S ::= E . $
E :: E . + T

2

E ::= T .

3 T ::= F .

4 T ::= + . F
F ::= . id
F ::= . const

5 T ::= - . F
F ::= . id
F ::= . const

6 F ::= id .

7 F ::= const .

8

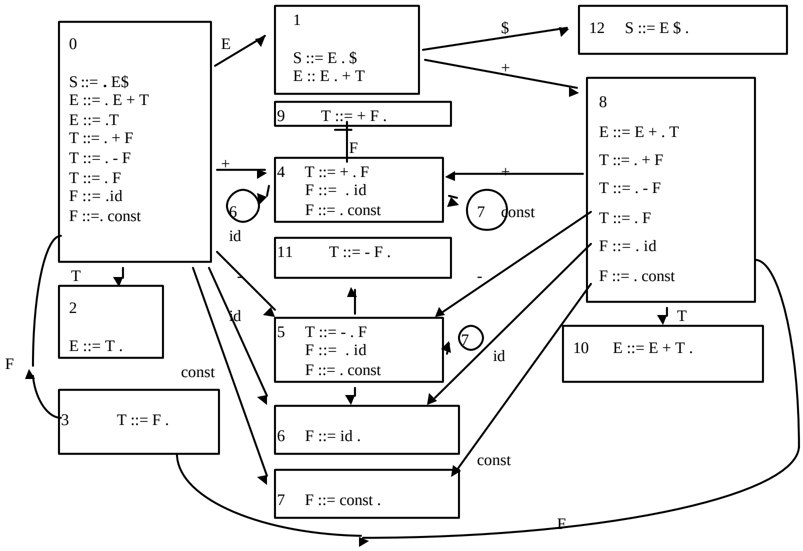
E ::= E + . T
T ::= . + F
T ::= . - F
T ::= . F
F ::= . id
F ::= . const

9 T ::= + F .

11 T ::= - F .

12 S ::= E $.

The LR(0) State Machine



0
S ::= . E$
E ::= . E + T
E ::= .T
T ::= . + F
T ::= . - F
T ::= . F
F ::= .id
F ::=. const

1
S ::= E . $
E :: E . + T

12 S ::= E $.

9 T ::= + F .

4 T ::= + . F
F ::= . id
F ::= . const

11 T ::= - F .

5 T ::= - . F
F ::= . id
F ::= . const

2
E ::= T .

3 T ::= F .

6 F ::= id .

7 F ::= const .

8
E ::= E + . T
T ::= . + F
T ::= . - F
T ::= . F
F ::= . id
F ::= . const

10 E ::= E + T .

The LR(0) parse table

					GOTO				
state	ACTION	+	-	id	const	$	E	T	F
0	s	4	5	6	7		1	2	3
1	s	8				12			
2	r3								
3	r6								
4	s			6	7				9
5	s			6	7				11
6	r7								
7	r8								
8	s	4	5	6	7			10	3
9	r4								
10	r2								
11	r5								
12	acc								

Follow the transitions to the next state

The LR(0) parse table

GOTO									
state	ACTION	+	-	id	const	$	E	T	F
0	s	4	5	6	7		1	2	3
1	s	8				12			
2	r3								
3	r6								
4	s			6	7				9
5	s			6	7				11
6	r7								
7	r8								
8	s	4	5	6	7			10	3
9	r4								
10	r2								
11	r5								
12	acc								

This is a “final” state because of the item E ::= T **.**

The LR Parser Driver

```
Perform the Action associated with the current state and token
REPEAT
    IF the Action is:
        Shift:          Shift the current token on the stack  with the
                        new state
        Reduce n:       Popall the variables of the right sentential form
                        together with the states.  Push the nonterminal
                        from the left side  of the production together
                        with GOTO(state, Nonterminal).
        Acecept         Clean up
        Error           Any error handling procedure

UNTIL Action for the current state and token is ACCEPT
```