# Grammars and Language Analysis

The vocabulary for Pascal contains words like for, x, ;, := or 1.

A sentence of the language consists of some sequence of words taken from this vocabulary.

For example, with the above vocabulary, we can compose the legal Pascal statement:

```
x:= 1
```

But not all possible sequences over a given vocabulary are legal, as the sequences need to meet certain syntactic and semantic restrictions.

For example, with the Pascal vocabulary, we can construct the sequence of words

```
x:= for 1
```

which is syntactically illegal.

A context free grammar provides a method for specifying which of the sentences are syntactically legal and which are syntactically illegal.

A context free grammar can be specified by a sequence of rules like the following (the rules are numbered for reference):

```
1)  program:    stmts '.'
2)  stmts:      stmt ';' stmts
3)  stmts:      stmt
4)  stmt:       ID ':=' expr
5)  expr:       NUM
6)  expr:       expr '+' NUM
```

Each rule consists of a left-hand side (LHS) and a right-hand side (RHS) separated by a colon.

The LHS contains a single symbol, whereas the RHS can consist of a (possibly empty) sequence of grammar symbols separated by spaces.

Each rule can be viewed as a rewrite rule, specifying that whenever we have an occurrence of a LHS symbol, it can be rewritten to the RHS.

For example, we can rewrite the LHS symbol program as follows:

```
    program ==> stmts .          by rule 1
            ==> stmt .           by rule 3
            ==> ID := expr .     by rule 4
            ==> ID := NUM .      by rule 5
```

Note that the above rewrite sequence or derivation terminates as we cannot rewrite ID, :=, NUM or . any further because they do not appear on the LHS of any rule.

Symbols like these terminate the derivation and are referred to as terminal symbols.

The other grammar symbols which occur on the LHS of grammar rules are referred to as nonterminal symbols. The terminal symbols correspond to words in the vocabulary.

The nonterminal symbols correspond to legal phrases or sentences formed using words from the vocabulary.

The terminal symbols are of two types:

terminals which stand for a single word in the vocabulary and terminals which stand for a class of multiple vocabulary words.

In the grammar, we denote terminals of the former type by enclosing the vocabulary word within quotes (for example, ':=' denotes the single vocabulary word :=),

and denote symbols of the latter type with a name which uses only upper-case letters (for example, ID denotes the class of all vocabulary words which are identifiers).

Not all the phrases of a language are sentences in the language; for example, a Pascal statement by itself (a phrase) is not a legal Pascal program (a sentence).

To ensure that a grammar describes only complete sentences, we distinguish one nonterminal called the start nonterminal from all others, and restrict all derivations to start from this distinguished start symbol.

We can now define a sentential form of a grammar G to be any sequence of grammar symbols (terminals or nonterminals) derived in 0 or more steps from the start symbol of G.

A sentence is a sentential-form which contains only terminal symbols.

The language defined by grammar G is the set of all sentences which can be derived from the start symbol of G.

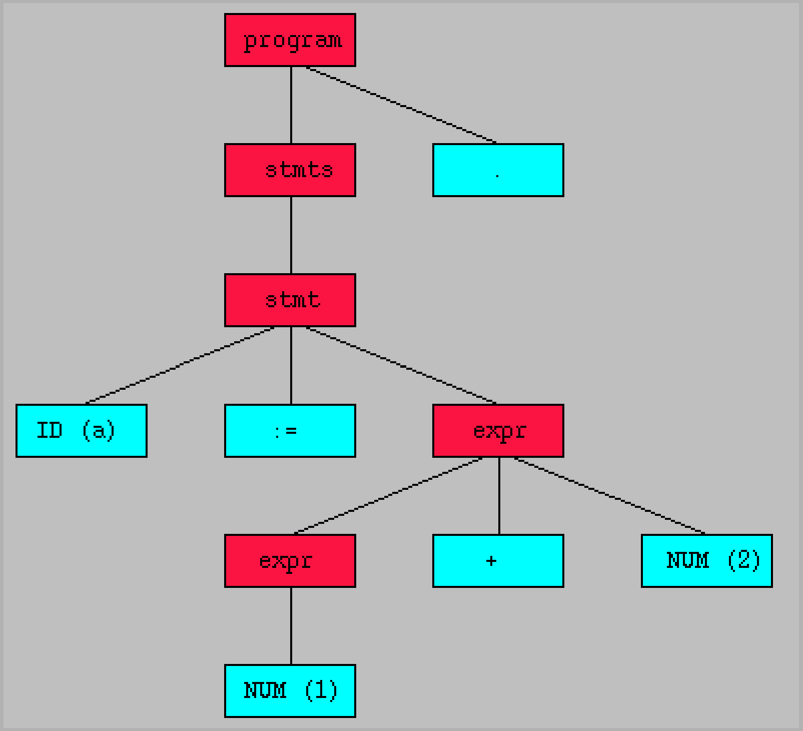
Note that a context-free grammar enforces only syntactic restrictions.

It does not enforce any semantic restrictions on the sentences to ensure that they make sense.

For example, a typical English grammar would allow the sentence

This page is swimming in the river

as grammatically correct, even though it does not make any sense.



The root of the parse tree corresponds to the start symbol. For each step in the derivation, when a LHS grammar nonterminal is replaced with the RHS of some grammar rule for that LHS, the RHS symbols are added as children to the node (shown in red) corresponding to the LHS symbol in the parse tree. After the derivation is complete, the leaves of the parse tree (shown in cyan) correspond to the terminal symbols in the sentence.

| | | |
|---|---|---|
| 1) | program: | stmts '.' |
| 2) | stmts: | stmt ';' stmts |
| 3) | stmts: | stmt |
| 4) | stmt: | ID ':=' expr |
| 5) | expr: | NUM |
| 6) | expr: | expr '+' NUM |

## Language Analysis

We can analyse the sentences of a language by extracting the grammatical structure of the sentence.

This is done by building an explicit or implicit parse tree for the sentence.

The parser can be organized so as to attempt to build a derivation of the sentence from the start symbol of the grammar.

This kind of parser is referred to as a top-down parser, because in terms of the parse tree, it is built from the top to the bottom.

Alternately, it can be organized so as to attempt to trace out a derivation in reverse, going from the sentence to the start symbol.

This kind of parser is referred to as a bottom-up parser, because in terms of the parse tree, it is built from the bottom to the top.

A top-down parser needs to make two kinds of decisions at each step:

1/ It needs to choose a nonterminal in the frontier of the current parse tree to expand next.

2/ In general, a nonterminal may have several rules associated with it. Hence the parser needs to decide which rule to use to expand the nonterminal it chose in (1).

A bottom-up parser needs to make similar decisions.

A common organization used in batch compilers is for the parser to call the scanner whenever it is needs another token.

Usually, the parser needs to look at one or more tokens to make its parsing decisions: these tokens which the parser is examining but not yet consumed are called lookahead tokens.

Typically, the parser needs to use a single lookahead token.

## Shift-Reduce Parsing

A shift-reduce parser uses a parse stack which (conceptually) contains grammar symbols.

During the operation of the parser, symbols from the input are shifted onto the stack.

If a prefix of the symbols on top of the stack matches the RHS of a grammar rule which is the correct rule to use within the current context, then the parser reduces the RHS of the rule to its LHS, replacing the RHS symbols on top of the stack with the nonterminal occurring on the LHS of the rule.

This shift-reduce process continues until the parser terminates, reporting either success or failure. It terminates with success when the input is legal and is accepted by the parser.

It terminates with failure if an error is detected in the input.

The parser is a stack automaton which may be in one of several discrete states.

A state is usually represented simply as an integer.

In reality, the parse stack contains states, rather than grammar symbols.

However, since each state corresponds to a unique grammar symbol, the state stack can be mapped onto the grammar symbol stack mentioned earlier.

So, for example, since you'll never have a "s3" following on two different symbols, you know when you're in state 3 that you've just seen a ":=", just as to get to states 5, 9 or 10 you'll have to have just seen an ID.

In practice, bison also maintains a stack for attributes (the $1, $2 etc.), so when there's (conceptually) an ID on the stack, there might actually be a "a" or "b" on the attribute stack.

## The operation of the parser is controlled by a couple of tables:

Action Table

The action table is a table with rows indexed by states and columns indexed by terminal symbols. When the parser is in some state s and the current lookahead terminal is t, the action taken by the parser depends on the contents of action[s][t], which can contain four different kinds of entries:

Shift s'

Shift state s' onto the parse stack.

Reduce r

Reduce by rule r. This is explained in more detail below.

Accept

Terminate the parse with success, accepting the input.

Error

Signal a parse error.

Goto Table

The goto table is a table with rows indexed by states and columns indexed by nonterminal symbols.

When the parser is in state s immediately after reducing by rule N, then the next state to enter is given by goto[s][N].

The current state of a shift-reduce parser is the state on top of the state stack. The detailed operation of such a parser is as follows:

1 Initialize the parse stack to contain a single state s0, where s0 is the distinguished initial state of the parser.

2 Use the state s on top of the parse stack and the current lookahead t to consult the action table entry action[s][t]:

If the action table entry is shift s' then push state s' onto the stack and advance the input so that the lookahead is set to the next token.

If the action table entry is reduce r and rule r has m symbols in its RHS, then pop m symbols off the parse stack. Let s' be the state now revealed on top of the parse stack and N be the LHS nonterminal for rule r.
Then consult the *goto* table and push the state *given* by *goto*[s'][N] onto the stack.
The lookahead token is not changed by this step.

If the action table entry is accept, then terminate the parse with success.

If the action table entry is error, then signal an error.

3 Repeat step (2) until the parser terminates.

For example, consider the following simple grammar

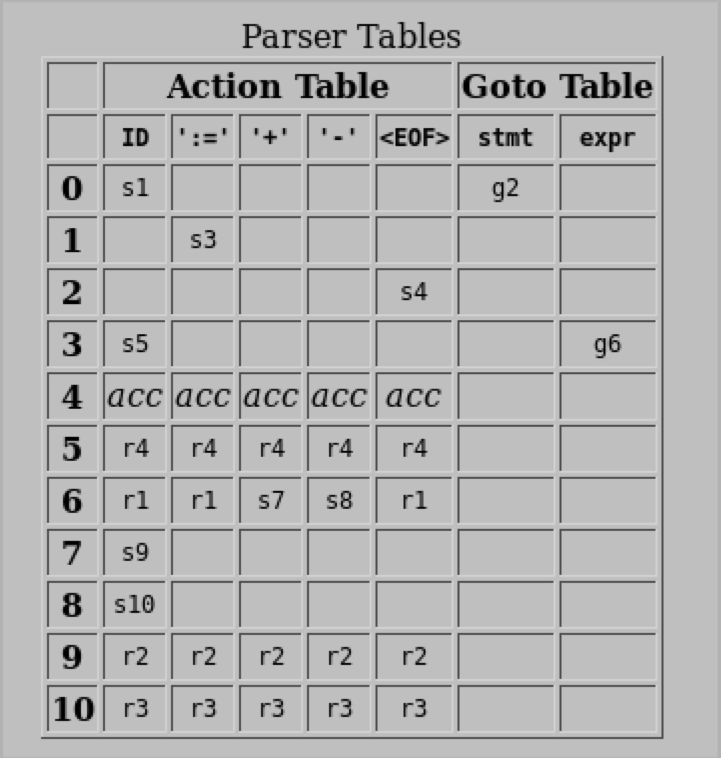
```
0)   $S: stmt <EOF>
1)   stmt:    ID ':=' expr
2)   expr:    expr '+' ID
3)   expr:    expr '-' ID
4)   expr:    ID
```

which describes assignment statements like a:= b + c - d

(Rule 0 is a special augmenting production added to the grammar).

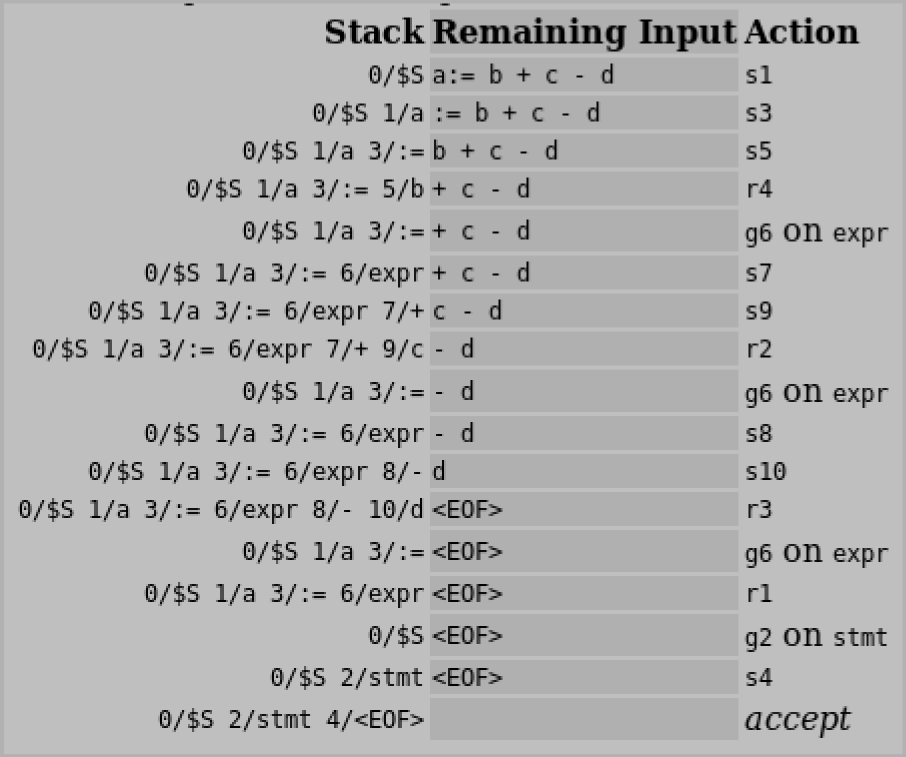
sn denotes shift n,
rn denotes reduce n,
acc denotes accept
blank entries denote error entries

input a:= b + c - d



Parser Tables

| | Action Table | | | | | Goto Table | |
|---|---|---|---|---|---|---|---|
| | ID | ':=' | '+' | '-' | <EOF> | stmt | expr |
| **0** | s1 | | | | | g2 | |
| **1** | | s3 | | | | | |
| **2** | | | | | s4 | | |
| **3** | s5 | | | | | | g6 |
| **4** | *acc* | *acc* | *acc* | *acc* | *acc* | | |
| **5** | r4 | r4 | r4 | r4 | r4 | | |
| **6** | r1 | r1 | s7 | s8 | r1 | | |
| **7** | s9 | | | | | | |
| **8** | s10 | | | | | | |
| **9** | r2 | r2 | r2 | r2 | r2 | | |
| **10** | r3 | r3 | r3 | r3 | r3 | | |



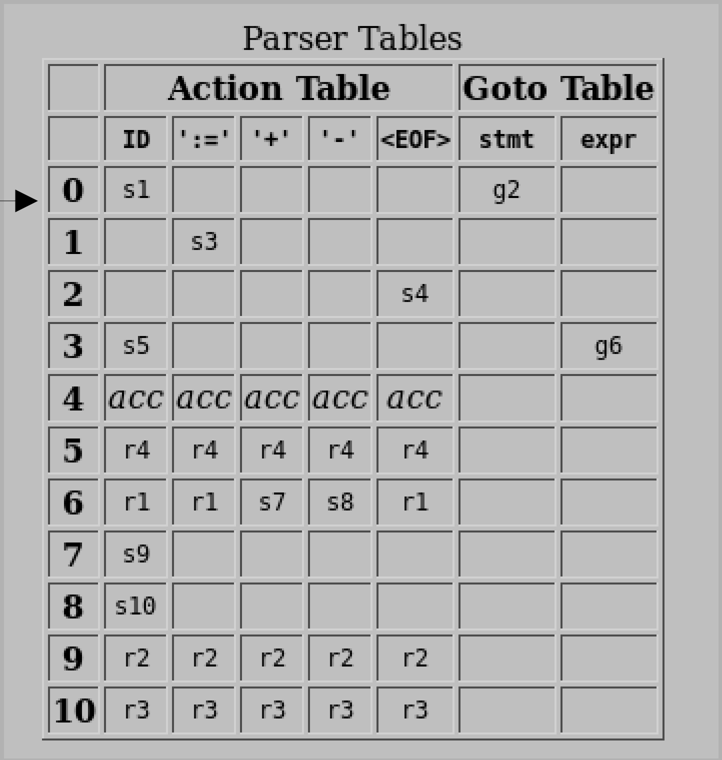
| Stack | Remaining Input | Action |
|---|---|---|
| 0/$S | a:= b + c - d | s1 |
| 0/$S 1/a | := b + c - d | s3 |
| 0/$S 1/a 3/:= | b + c - d | s5 |
| 0/$S 1/a 3/:= 5/b | + c - d | r4 |
| 0/$S 1/a 3/:= | + c - d | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | + c - d | s7 |
| 0/$S 1/a 3/:= 6/expr 7/+ | c - d | s9 |
| 0/$S 1/a 3/:= 6/expr 7/+ 9/c | - d | r2 |
| 0/$S 1/a 3/:= | - d | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | - d | s8 |
| 0/$S 1/a 3/:= 6/expr 8/- | d | s10 |
| 0/$S 1/a 3/:= 6/expr 8/- 10/d | <EOF> | r3 |
| 0/$S 1/a 3/:= | <EOF> | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | <EOF> | r1 |
| 0/$S | <EOF> | g2 on stmt |
| 0/$S 2/stmt | <EOF> | s4 |
| 0/$S 2/stmt 4/<EOF> | | *accept* |

0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID

Each stack entry is shown as a state number followed by the symbol which caused the transition to that state.

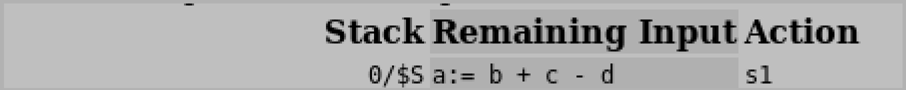
1/

input a := b + c - d



Parser Tables

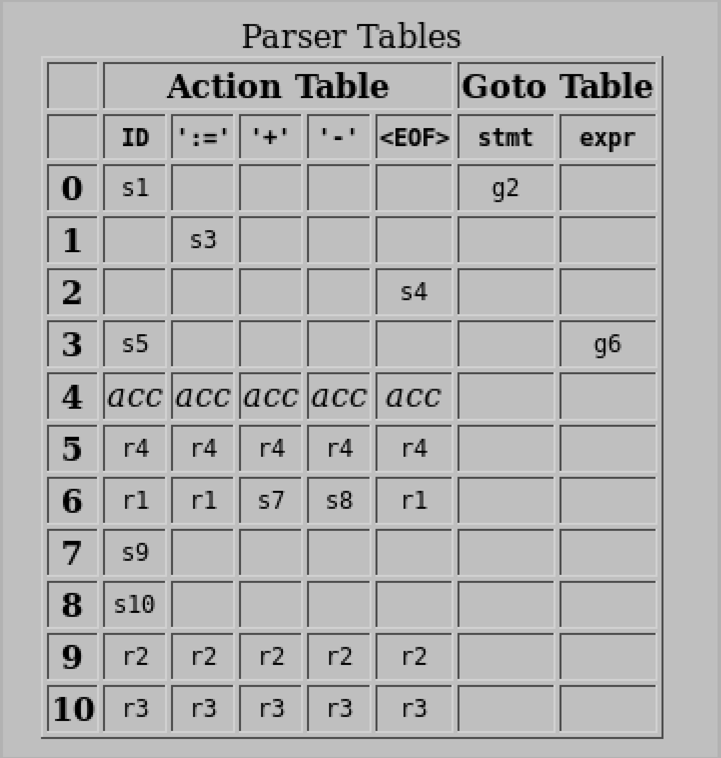
|  | Action Table |  |  |  |  | Goto Table |  |
|---|---|---|---|---|---|---|---|
|  | ID | ':=' | '+' | '-' | <EOF> | stmt | expr |
| **0** | s1 |  |  |  |  | g2 |  |
| **1** |  | s3 |  |  |  |  |  |
| **2** |  |  |  |  | s4 |  |  |
| **3** | s5 |  |  |  |  |  | g6 |
| **4** | *acc* | *acc* | *acc* | *acc* | *acc* |  |  |
| **5** | r4 | r4 | r4 | r4 | r4 |  |  |
| **6** | r1 | r1 | s7 | s8 | r1 |  |  |
| **7** | s9 |  |  |  |  |  |  |
| **8** | s10 |  |  |  |  |  |  |
| **9** | r2 | r2 | r2 | r2 | r2 |  |  |
| **10** | r3 | r3 | r3 | r3 | r3 |  |  |



| Stack | Remaining Input | Action |
|---|---|---|
| 0/$S | a:= b + c - d | s1 |

1 Initialize the parse stack to contain a single state s0, where s0 is the distinguished initial state of the parser.

input a := b + c - d



Parser Tables

| | Action Table | | | | | Goto Table | |
|---|---|---|---|---|---|---|---|
| | ID | ':=' | '+' | '-' | <EOF> | stmt | expr |
| **0** | s1 | | | | | g2 | |
| **1** | | s3 | | | | | |
| **2** | | | | | s4 | | |
| **3** | s5 | | | | | | g6 |
| **4** | *acc* | *acc* | *acc* | *acc* | *acc* | | |
| **5** | r4 | r4 | r4 | r4 | r4 | | |
| **6** | r1 | r1 | s7 | s8 | r1 | | |
| **7** | s9 | | | | | | |
| **8** | s10 | | | | | | |
| **9** | r2 | r2 | r2 | r2 | r2 | | |
| **10** | r3 | r3 | r3 | r3 | r3 | | |

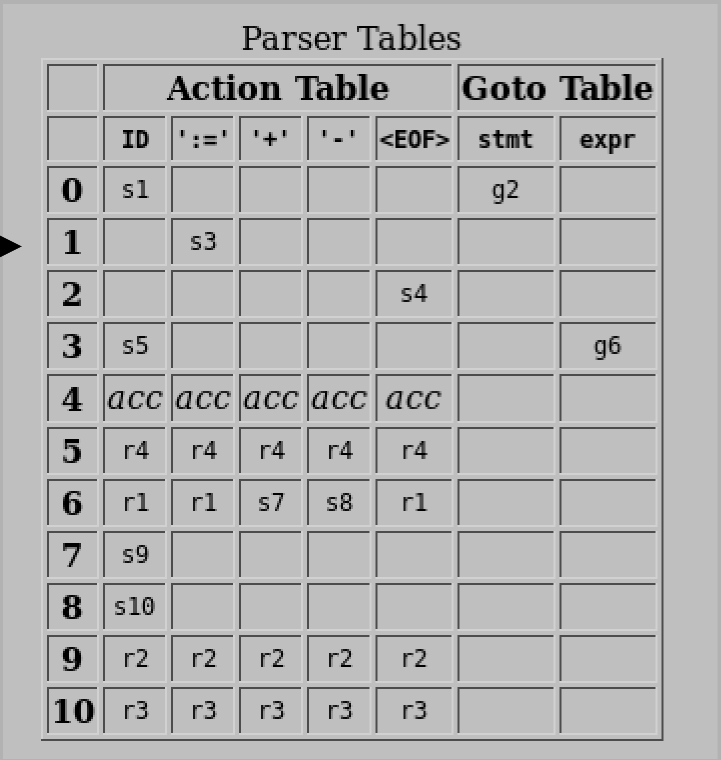
| Stack | Remaining Input | Action |
|---|---|---|
| 0/$S | a:= b + c - d | s1 |

Use the state s on top of the parse stack and the current lookahead t to consult the action table entry action[s][t]:

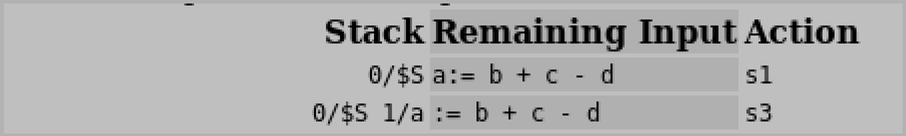
If the action table entry is shift s' then push state s' onto the stack and advance the input so that the lookahead is set to the next token.

2/

input a := b + c - d


Parser Tables

| | Action Table | | | | | Goto Table | |
|---|---|---|---|---|---|---|---|
| | ID | ':=' | '+' | '-' | <EOF> | stmt | expr |
| **0** | s1 | | | | | g2 | |
| **1** | | s3 | | | | | |
| **2** | | | | | s4 | | |
| **3** | s5 | | | | | | g6 |
| **4** | *acc* | *acc* | *acc* | *acc* | *acc* | | |
| **5** | r4 | r4 | r4 | r4 | r4 | | |
| **6** | r1 | r1 | s7 | s8 | r1 | | |
| **7** | s9 | | | | | | |
| **8** | s10 | | | | | | |
| **9** | r2 | r2 | r2 | r2 | r2 | | |
| **10** | r3 | r3 | r3 | r3 | r3 | | |



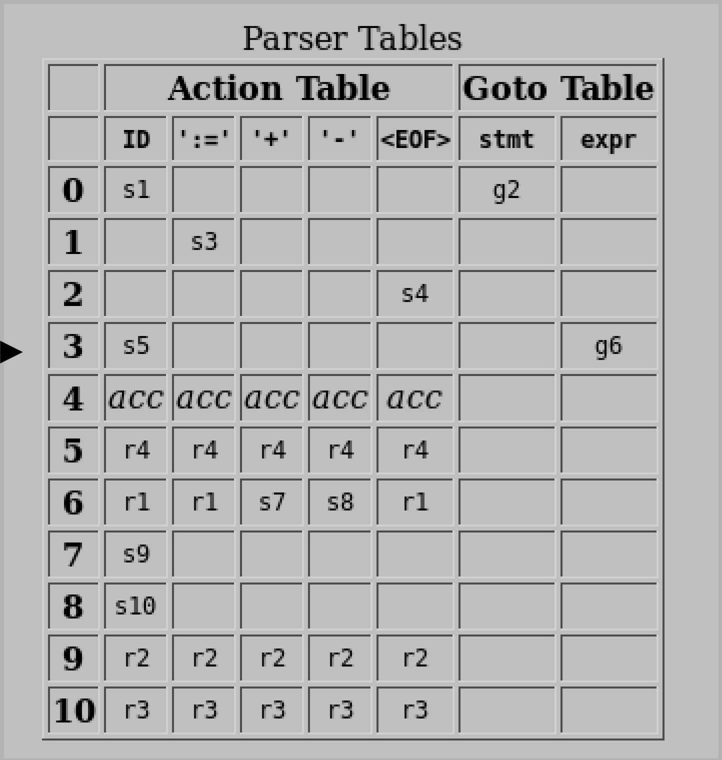
| Stack | Remaining Input | Action |
|---|---|---|
| 0/$S | a:= b + c - d | s1 |
| 0/$S 1/a | := b + c - d | s3 |

Use the state s on top of the parse stack and the current lookahead t to consult the action table entry action[s][t]:

If the action table entry is shift s' then push state s' onto the stack and advance the input so that the lookahead is set to the next token.

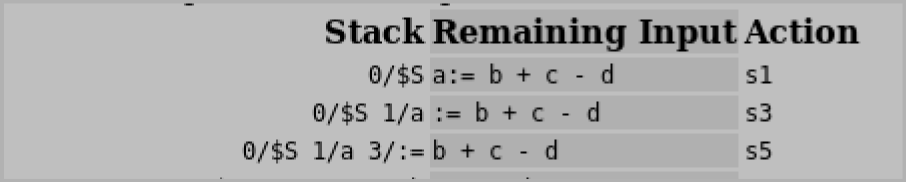
3/

input a := b + c - d



Parser Tables

| | Action Table | | | | | Goto Table | |
|---|---|---|---|---|---|---|---|
| | ID | ':=' | '+' | '-' | <EOF> | stmt | expr |
| 0 | s1 | | | | | g2 | |
| 1 | | s3 | | | | | |
| 2 | | | | | s4 | | |
| 3 | s5 | | | | | | g6 |
| 4 | *acc* | *acc* | *acc* | *acc* | *acc* | | |
| 5 | r4 | r4 | r4 | r4 | r4 | | |
| 6 | r1 | r1 | s7 | s8 | r1 | | |
| 7 | s9 | | | | | | |
| 8 | s10 | | | | | | |
| 9 | r2 | r2 | r2 | r2 | r2 | | |
| 10 | r3 | r3 | r3 | r3 | r3 | | |



| Stack | Remaining Input | Action |
|---|---|---|
| 0/$S | a:= b + c - d | s1 |
| 0/$S 1/a | := b + c - d | s3 |
| 0/$S 1/a 3/:= | b + c - d | s5 |

Use the state s on top of the parse stack and the current lookahead t to consult the action table entry action[s][t]:

If the action table entry is shift s' then push state s' onto the stack and advance the input so that the lookahead is set to the next token.

input a := b + c - d

Parser Tables

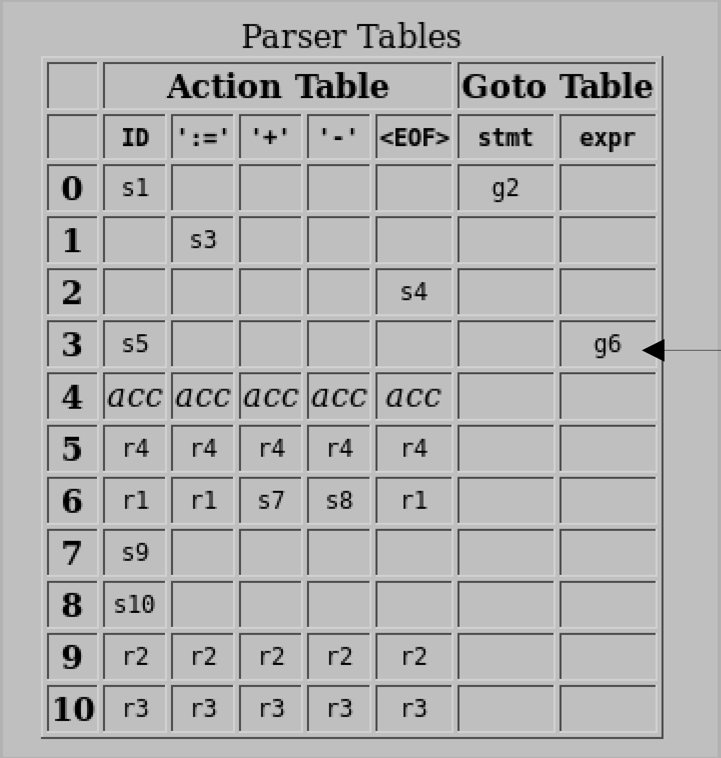
| | Action Table | | | | | Goto Table | |
|---|---|---|---|---|---|---|---|
| | ID | ':=' | '+' | '-' | <EOF> | stmt | expr |
| **0** | s1 | | | | | g2 | |
| **1** | | s3 | | | | | |
| **2** | | | | | s4 | | |
| **3** | s5 | | | | | | g6 |
| **4** | *acc* | *acc* | *acc* | *acc* | *acc* | | |
| **5** | r4 | r4 | r4 | r4 | r4 | | |
| **6** | r1 | r1 | s7 | s8 | r1 | | |
| **7** | s9 | | | | | | |
| **8** | s10 | | | | | | |
| **9** | r2 | r2 | r2 | r2 | r2 | | |
| **10** | r3 | r3 | r3 | r3 | r3 | | |

| Stack | Remaining Input | Action |
|---|---|---|
| 0/$S | a:= b + c - d | s1 |
| 0/$S 1/a | := b + c - d | s3 |
| 0/$S 1/a 3/:= | b + c - d | s5 |
| 0/$S 1/a 3/:= 5/b | + c - d | r4 |

| | | |
|---|---|---|
| 0) | $S: stmt <EOF> | |
| 1) | stmt: | ID ':=' expr |
| 2) | expr: | expr '+' ID |
| 3) | expr: | expr '-' ID |
| 4) | expr: | ID |

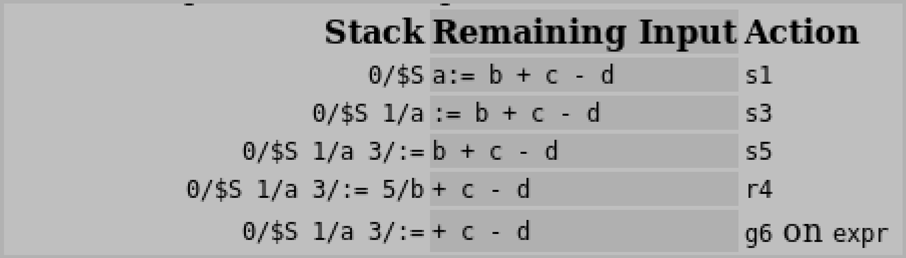
If the action table entry is reduce r and rule r has m symbols in its RHS, then pop m symbols off the parse stack. Let s' be the state now revealed on top of the parse stack and N be the LHS nonterminal for rule r.
Then consult the goto table and push the state given by goto[s'][N] onto the stack.
The lookahead token is not changed by this step.

input a := b + c - d

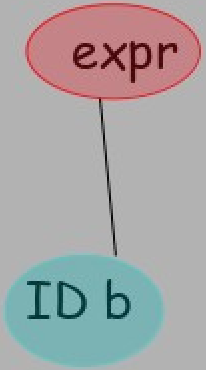


Parser Tables

| | Action Table | | | | | Goto Table | |
|---|---|---|---|---|---|---|---|
| | ID | ':=' | '+' | '-' | <EOF> | stmt | expr |
| **0** | s1 | | | | | g2 | |
| **1** | | s3 | | | | | |
| **2** | | | | | s4 | | |
| **3** | s5 | | | | | | g6 |
| **4** | *acc* | *acc* | *acc* | *acc* | *acc* | | |
| **5** | r4 | r4 | r4 | r4 | r4 | | |
| **6** | r1 | r1 | s7 | s8 | r1 | | |
| **7** | s9 | | | | | | |
| **8** | s10 | | | | | | |
| **9** | r2 | r2 | r2 | r2 | r2 | | |
| **10** | r3 | r3 | r3 | r3 | r3 | | |



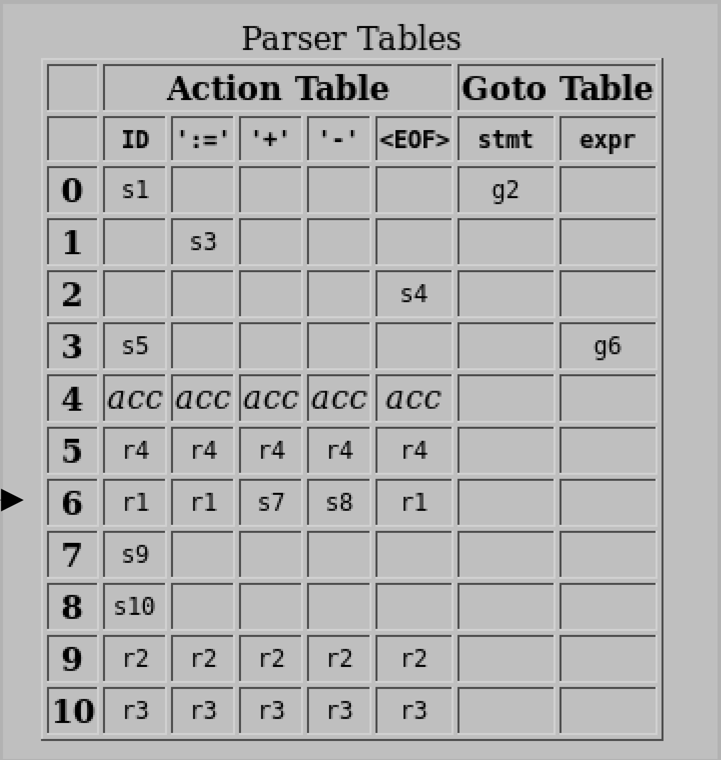
| Stack | Remaining Input | Action |
|---|---|---|
| 0/$S | a:= b + c - d | s1 |
| 0/$S 1/a | := b + c - d | s3 |
| 0/$S 1/a 3/:= | b + c - d | s5 |
| 0/$S 1/a 3/:= 5/b | + c - d | r4 |
| 0/$S 1/a 3/:= | + c - d | g6 on expr |



0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID

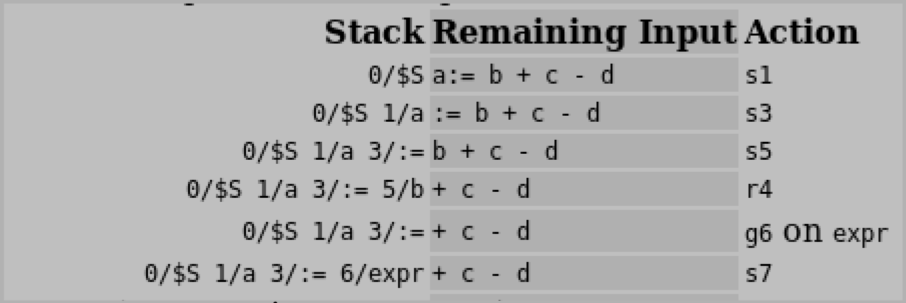
If the action table entry is reduce r and rule r has m symbols in its RHS, then pop m symbols off the parse stack. Let s' be the state now revealed on top of the parse stack and N be the LHS nonterminal for rule r.
Then consult the goto table and push the state given by goto[s'][N] onto the stack.
The lookahead token is not changed by this step.

input a := b + c - d



Parser Tables

| | Action Table | | | | | Goto Table | |
|---|---|---|---|---|---|---|---|
| | ID | ':=' | '+' | '-' | <EOF> | stmt | expr |
| **0** | s1 | | | | | g2 | |
| **1** | | s3 | | | | | |
| **2** | | | | | s4 | | |
| **3** | s5 | | | | | | g6 |
| **4** | *acc* | *acc* | *acc* | *acc* | *acc* | | |
| **5** | r4 | r4 | r4 | r4 | r4 | | |
| **6** | r1 | r1 | s7 | s8 | r1 | | |
| **7** | s9 | | | | | | |
| **8** | s10 | | | | | | |
| **9** | r2 | r2 | r2 | r2 | r2 | | |
| **10** | r3 | r3 | r3 | r3 | r3 | | |

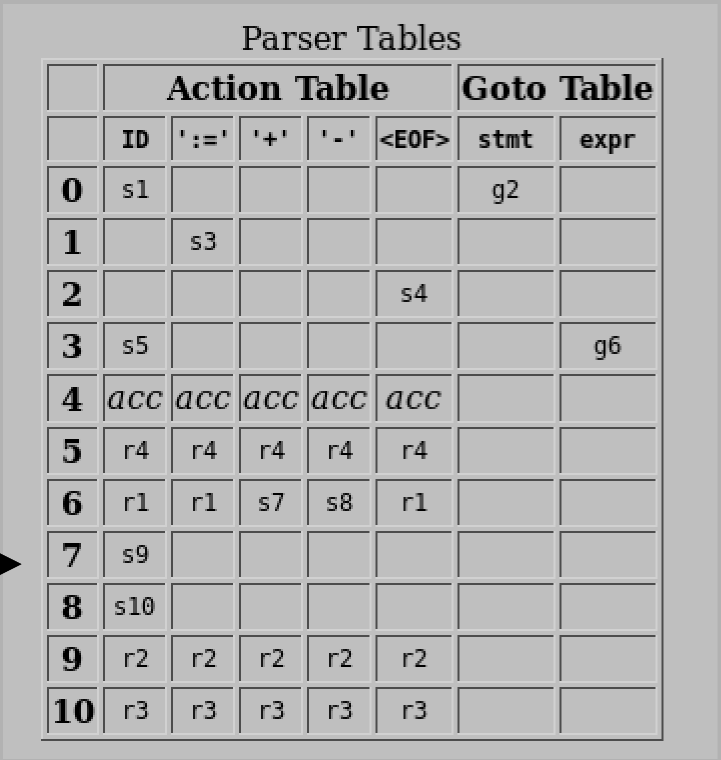


| Stack | Remaining Input | Action |
|---|---|---|
| 0/$S | a:= b + c - d | s1 |
| 0/$S 1/a | := b + c - d | s3 |
| 0/$S 1/a 3/:= | b + c - d | s5 |
| 0/$S 1/a 3/:= 5/b | + c - d | r4 |
| 0/$S 1/a 3/:= | + c - d | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | + c - d | s7 |

0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID

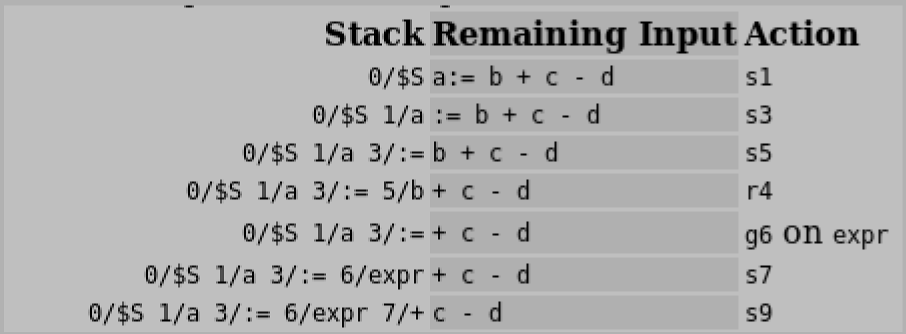
If the action table entry is reduce r and rule r has m symbols in its RHS, then pop m symbols off the parse stack. Let s' be the state now revealed on top of the parse stack and N be the LHS nonterminal for rule r.
Then consult the goto table and push the state given by goto[s'][N] onto the stack.
The lookahead token is not changed by this step.

input a := b + c - d



Parser Tables

| | Action Table | | | | | Goto Table | |
|---|---|---|---|---|---|---|---|
| | ID | ':=' | '+' | '-' | <EOF> | stmt | expr |
| **0** | s1 | | | | | g2 | |
| **1** | | s3 | | | | | |
| **2** | | | | | s4 | | |
| **3** | s5 | | | | | | g6 |
| **4** | *acc* | *acc* | *acc* | *acc* | *acc* | | |
| **5** | r4 | r4 | r4 | r4 | r4 | | |
| **6** | r1 | r1 | s7 | s8 | r1 | | |
| **7** | s9 | | | | | | |
| **8** | s10 | | | | | | |
| **9** | r2 | r2 | r2 | r2 | r2 | | |
| **10** | r3 | r3 | r3 | r3 | r3 | | |



| Stack | Remaining Input | Action |
|---|---|---|
| 0/$S | a:= b + c - d | s1 |
| 0/$S 1/a | := b + c - d | s3 |
| 0/$S 1/a 3/:= | b + c - d | s5 |
| 0/$S 1/a 3/:= 5/b | + c - d | r4 |
| 0/$S 1/a 3/:= | + c - d | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | + c - d | s7 |
| 0/$S 1/a 3/:= 6/expr 7/+ | c - d | s9 |

Use the state s on top of the parse stack and the current lookahead t to consult the action table entry action[s][t]:

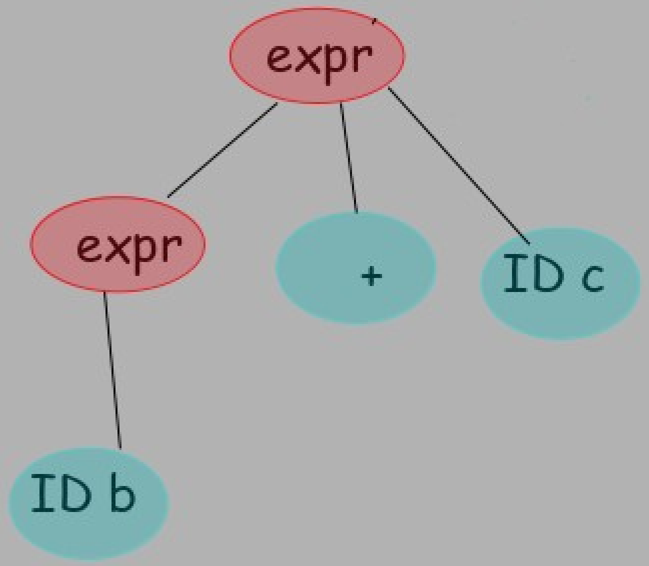
If the action table entry is shift s' then push state s' onto the stack and advance the input so that the lookahead is set to the next token.

input a := b + c - d

Parser Tables

| | Action Table | | | | | Goto Table | |
|---|---|---|---|---|---|---|---|
| | ID | ':=' | '+' | '-' | <EOF> | stmt | expr |
| **0** | s1 | | | | | g2 | |
| **1** | | s3 | | | | | |
| **2** | | | | | s4 | | |
| **3** | s5 | | | | | | g6 |
| **4** | *acc* | *acc* | *acc* | *acc* | *acc* | | |
| **5** | r4 | r4 | r4 | r4 | r4 | | |
| **6** | r1 | r1 | s7 | s8 | r1 | | |
| **7** | s9 | | | | | | |
| **8** | s10 | | | | | | |
| **9** | r2 | r2 | r2 | r2 | r2 | | |
| **10** | r3 | r3 | r3 | r3 | r3 | | |

| Stack | Remaining Input | Action |
|---|---|---|
| 0/$S | a:= b + c - d | s1 |
| 0/$S 1/a | := b + c - d | s3 |
| 0/$S 1/a 3/:= | b + c - d | s5 |
| 0/$S 1/a 3/:= 5/b | + c - d | r4 |
| 0/$S 1/a 3/:= | + c - d | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | + c - d | s7 |
| 0/$S 1/a 3/:= 6/expr 7/+ | c - d | s9 |
| 0/$S 1/a 3/:= 6/expr 7/+ 9/c | - d | r2 |



0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID

input a := b + c - d

Parser Tables

| | Action Table | | | | | Goto Table | |
|---|---|---|---|---|---|---|---|
| | ID | ':=' | '+' | '-' | <EOF> | stmt | expr |
| **0** | s1 | | | | | g2 | |
| **1** | | s3 | | | | | |
| **2** | | | | | s4 | | |
| **3** | s5 | | | | | | g6 |
| **4** | *acc* | *acc* | *acc* | *acc* | *acc* | | |
| **5** | r4 | r4 | r4 | r4 | r4 | | |
| **6** | r1 | r1 | s7 | s8 | r1 | | |
| **7** | s9 | | | | | | |
| **8** | s10 | | | | | | |
| **9** | r2 | r2 | r2 | r2 | r2 | | |
| **10** | r3 | r3 | r3 | r3 | r3 | | |

| Stack | Remaining Input | Action |
|---|---|---|
| 0/$S | a:= b + c - d | s1 |
| 0/$S 1/a | := b + c - d | s3 |
| 0/$S 1/a 3/:= | b + c - d | s5 |
| 0/$S 1/a 3/:= 5/b | + c - d | r4 |
| 0/$S 1/a 3/:= | + c - d | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | + c - d | s7 |
| 0/$S 1/a 3/:= 6/expr 7/+ | c - d | s9 |
| 0/$S 1/a 3/:= 6/expr 7/+ 9/c | - d | r2 |
| 0/$S 1/a 3/:= | - d | g6 on expr |

0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID

input a := b + c - d

Parser Tables

| | Action Table | | | | | Goto Table | |
|---|---|---|---|---|---|---|---|
| | **ID** | **':='** | **'+'** | **'-'** | **<EOF>** | **stmt** | **expr** |
| **0** | s1 | | | | | g2 | |
| **1** | | s3 | | | | | |
| **2** | | | | | s4 | | |
| **3** | s5 | | | | | | g6 |
| **4** | *acc* | *acc* | *acc* | *acc* | *acc* | | |
| **5** | r4 | r4 | r4 | r4 | r4 | | |
| **6** | r1 | r1 | s7 | s8 | r1 | | |
| **7** | s9 | | | | | | |
| **8** | s10 | | | | | | |
| **9** | r2 | r2 | r2 | r2 | r2 | | |
| **10** | r3 | r3 | r3 | r3 | r3 | | |

| Stack | Remaining Input | Action |
|---:|---|---|
| 0/$S | a:= b + c - d | s1 |
| 0/$S 1/a | := b + c - d | s3 |
| 0/$S 1/a 3/:= | b + c - d | s5 |
| 0/$S 1/a 3/:= 5/b | + c - d | r4 |
| 0/$S 1/a 3/:= | + c - d | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | + c - d | s7 |
| 0/$S 1/a 3/:= 6/expr 7/+ | c - d | s9 |
| 0/$S 1/a 3/:= 6/expr 7/+ 9/c | - d | r2 |
| 0/$S 1/a 3/:= | - d | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | - d | s8 |

input a := b + c - d

Parser Tables

| | Action Table | | | | | Goto Table | |
|---|---|---|---|---|---|---|---|
| | ID | ':=' | '+' | '-' | <EOF> | stmt | expr |
| **0** | s1 | | | | | g2 | |
| **1** | | s3 | | | | | |
| **2** | | | | | s4 | | |
| **3** | s5 | | | | | | g6 |
| **4** | *acc* | *acc* | *acc* | *acc* | *acc* | | |
| **5** | r4 | r4 | r4 | r4 | r4 | | |
| **6** | r1 | r1 | s7 | s8 | r1 | | |
| **7** | s9 | | | | | | |
| **8** | s10 | | | | | | |
| **9** | r2 | r2 | r2 | r2 | r2 | | |
| **10** | r3 | r3 | r3 | r3 | r3 | | |

| Stack | Remaining Input | Action |
|---|---|---|
| 0/$S | a:= b + c - d | s1 |
| 0/$S 1/a | := b + c - d | s3 |
| 0/$S 1/a 3/:= | b + c - d | s5 |
| 0/$S 1/a 3/:= 5/b | + c - d | r4 |
| 0/$S 1/a 3/:= | + c - d | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | + c - d | s7 |
| 0/$S 1/a 3/:= 6/expr 7/+ | c - d | s9 |
| 0/$S 1/a 3/:= 6/expr 7/+ 9/c | - d | r2 |
| 0/$S 1/a 3/:= | - d | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | - d | s8 |
| 0/$S 1/a 3/:= 6/expr 8/- | d | s10 |

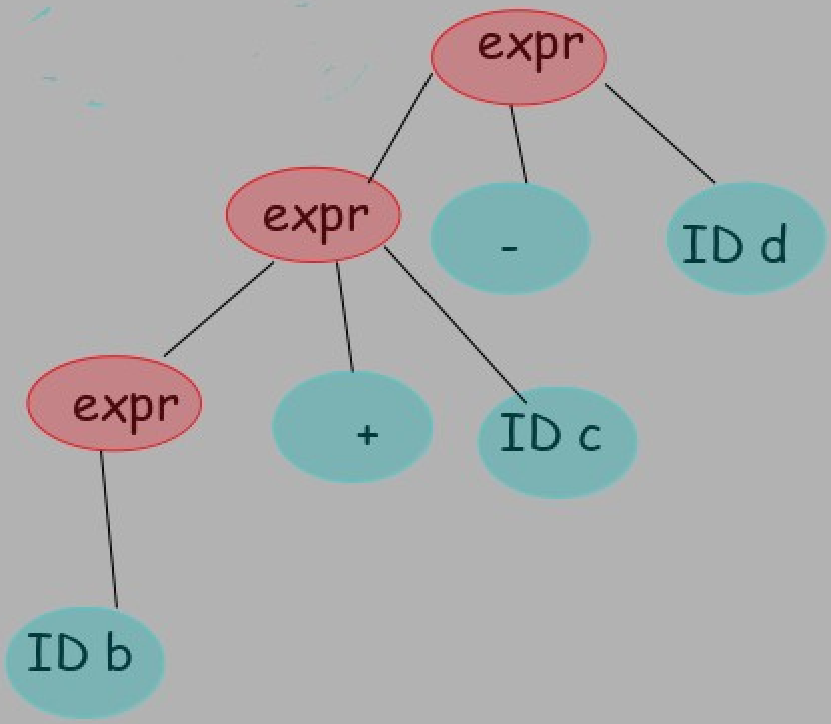
input a := b + c - d

Parser Tables

| | Action Table | | | | | Goto Table | |
|---|---|---|---|---|---|---|---|
| | ID | ':=' | '+' | '-' | <EOF> | stmt | expr |
| **0** | s1 | | | | | g2 | |
| **1** | | s3 | | | | | |
| **2** | | | | | s4 | | |
| **3** | s5 | | | | | | g6 |
| **4** | *acc* | *acc* | *acc* | *acc* | *acc* | | |
| **5** | r4 | r4 | r4 | r4 | r4 | | |
| **6** | r1 | r1 | s7 | s8 | r1 | | |
| **7** | s9 | | | | | | |
| **8** | s10 | | | | | | |
| **9** | r2 | r2 | r2 | r2 | r2 | | |
| **10** | r3 | r3 | r3 | r3 | r3 | | |

| Stack | Remaining Input | Action |
|---|---|---|
| 0/$S | a:= b + c - d | s1 |
| 0/$S 1/a | := b + c - d | s3 |
| 0/$S 1/a 3/:= | b + c - d | s5 |
| 0/$S 1/a 3/:= 5/b | + c - d | r4 |
| 0/$S 1/a 3/:= | + c - d | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | + c - d | s7 |
| 0/$S 1/a 3/:= 6/expr 7/+ | c - d | s9 |
| 0/$S 1/a 3/:= 6/expr 7/+ 9/c | - d | r2 |
| 0/$S 1/a 3/:= | - d | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | - d | s8 |
| 0/$S 1/a 3/:= 6/expr 8/- | d | s10 |
| 0/$S 1/a 3/:= 6/expr 8/- 10/d | <EOF> | r3 |

0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID


expr
expr
-
ID d
expr
+
ID c
ID b

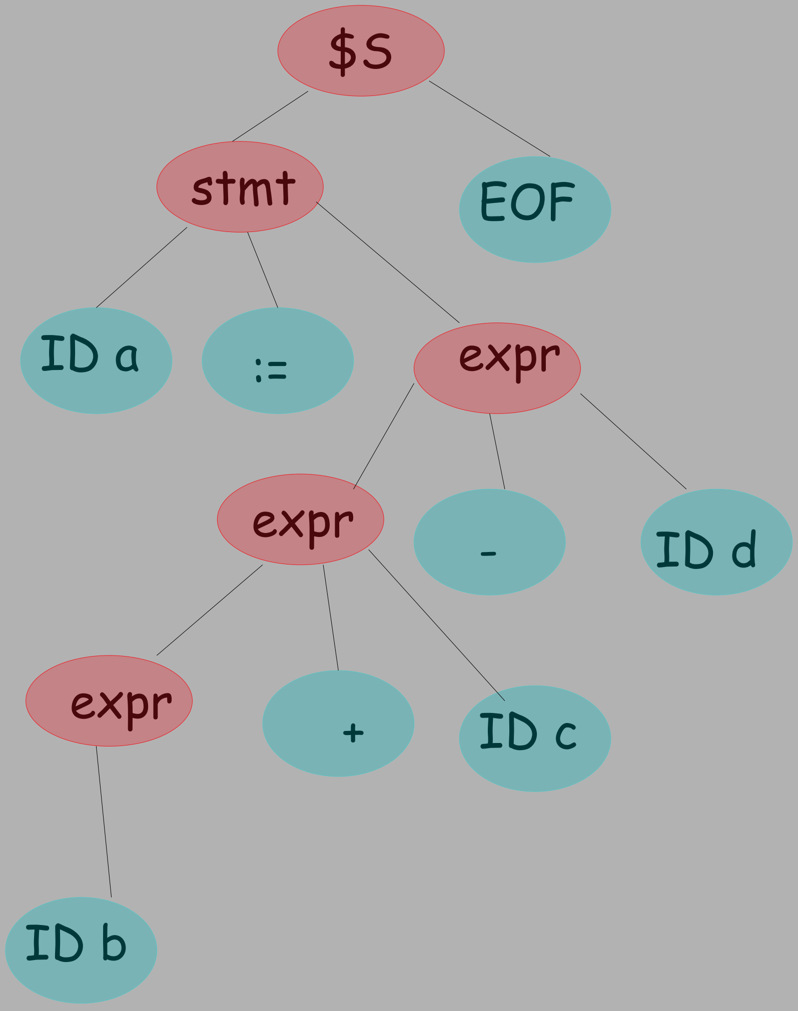
input a := b + c - d

Parser Tables

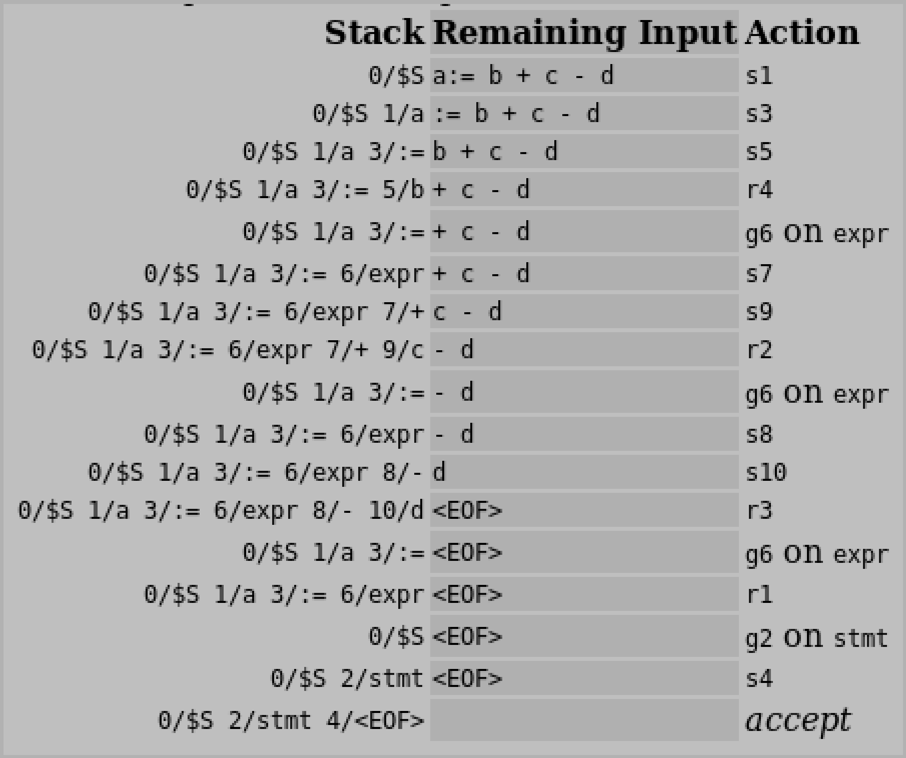
| | Action Table | | | | | Goto Table | |
|---|---|---|---|---|---|---|---|
| | ID | ':=' | '+' | '-' | <EOF> | stmt | expr |
| **0** | s1 | | | | | g2 | |
| **1** | | s3 | | | | | |
| **2** | | | | | s4 | | |
| **3** | s5 | | | | | | g6 |
| **4** | *acc* | *acc* | *acc* | *acc* | *acc* | | |
| **5** | r4 | r4 | r4 | r4 | r4 | | |
| **6** | r1 | r1 | s7 | s8 | r1 | | |
| **7** | s9 | | | | | | |
| **8** | s10 | | | | | | |
| **9** | r2 | r2 | r2 | r2 | r2 | | |
| **10** | r3 | r3 | r3 | r3 | r3 | | |

| Stack | Remaining Input | Action |
|---|---|---|
| 0/$S | a:= b + c - d | s1 |
| 0/$S 1/a | := b + c - d | s3 |
| 0/$S 1/a 3/:= | b + c - d | s5 |
| 0/$S 1/a 3/:= 5/b | + c - d | r4 |
| 0/$S 1/a 3/:= | + c - d | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | + c - d | s7 |
| 0/$S 1/a 3/:= 6/expr 7/+ | c - d | s9 |
| 0/$S 1/a 3/:= 6/expr 7/+ 9/c | - d | r2 |
| 0/$S 1/a 3/:= | - d | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | - d | s8 |
| 0/$S 1/a 3/:= 6/expr 8/- | d | s10 |
| 0/$S 1/a 3/:= 6/expr 8/- 10/d | <EOF> | r3 |
| 0/$S 1/a 3/:= | <EOF> | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | <EOF> | r1 |
| 0/$S | <EOF> | g2 on stmt |
| 0/$S 2/stmt | <EOF> | s4 |
| 0/$S 2/stmt 4/<EOF> | | *accept* |

input a:= b + c - d



0) $S: stmt <EOF>
1) stmt: ID ':=' expr
2) expr: expr '+' ID
3) expr: expr '-' ID
4) expr: ID



| Stack | Remaining Input | Action |
|---|---|---|
| 0/$S | a:= b + c - d | s1 |
| 0/$S 1/a | := b + c - d | s3 |
| 0/$S 1/a 3/:= | b + c - d | s5 |
| 0/$S 1/a 3/:= 5/b | + c - d | r4 |
| 0/$S 1/a 3/:= | + c - d | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | + c - d | s7 |
| 0/$S 1/a 3/:= 6/expr 7/+ | c - d | s9 |
| 0/$S 1/a 3/:= 6/expr 7/+ 9/c | - d | r2 |
| 0/$S 1/a 3/:= | - d | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | - d | s8 |
| 0/$S 1/a 3/:= 6/expr 8/- | d | s10 |
| 0/$S 1/a 3/:= 6/expr 8/- 10/d | <EOF> | r3 |
| 0/$S 1/a 3/:= | <EOF> | g6 on expr |
| 0/$S 1/a 3/:= 6/expr | <EOF> | r1 |
| 0/$S | <EOF> | g2 on stmt |
| 0/$S 2/stmt | <EOF> | s4 |
| 0/$S 2/stmt 4/<EOF> | | *accept* |

## Construction of Shift-Reduce Parsing Tables

The general idea of bottom-up parsing is to repeatedly match the RHS of some rule and reduce it to the rule's LHS.

To identify the matching RHS's, the parser needs to keep track of all possible rules which may match.

This is done by means of the parser state, where **each state keeps track of the set of rules the parser may currently be in, and how far along the parser may be within each rule.**

This idea of states will become clearer if we attempt to build the tables for a small example.

Consider the grammar

```
0)   $S: stmt <EOF>
1)   stmt:    ID ':=' expr
2)   expr:    expr '+' ID
3)   expr:    expr '-' ID
4)   expr:    ID
```

The input must be ultimately reducible to the augmenting nonterminal $S.

Hence the parser should initially be in rule 0; more specifically, it should be expecting the stmt in rule 0.

To show precisely which symbol is expected in a rule RHS, we define an item to be a rule, along with a position on the RHS specifying the next symbol to be expected in that RHS.

We denote an item as a rule with a dot . just before the next expected symbol.
Hence, returning to our example, the parser is initially expecting the item

```
0)   $S: . stmt <EOF>
```

However, if the parser is expecting to see a stmt, it could be at the beginning of any of the rules for stmt.
Hence the initial state should include the initial item for stmt. (The process of including these additional induced items is referred to as forming the closure of the state).

```
 State 0
0)   $S: . stmt <EOF>
1)   stmt:     . ID ':=' expr
```

Now if the parser sees an ID in state 0, then it can move the dot past any ID symbols in state 0. We get a new state; let's call it State 1:

```
State 1
1)   stmt:    ID . ':=' expr
```

If the parser has seen a stmt in state 0, then it can move the dot past any stmt symbols in state 0. We get a new state; let's call it State 2:

```
State 2
0)   $S: stmt . <EOF>
```

However since the dot is before the nonterminal expr, the parser could be in any of the rules for expr. Hence we need to include the rules for expr in a new state 3:

State 3

```
1)  stmt:   ID ':=' . expr
2)  expr:   . expr '+' ID
3)  expr:   . expr '-' ID
4)  expr:  . ID
```

```
0)  $S: stmt <EOF>
1)  stmt:   ID ':=' expr
2)  expr:   expr '+' ID
3)  expr:   expr '-' ID
4)  expr:   ID
```

We continue this process of following all possible transitions out of states until we cannot construct any new states.

The transitions on terminal symbols correspond to shift actions in the parser; the transitions on nonterminal symbols correspond to goto actions in the parser.

Note that the construction guarantees that **each state is entered by a unique grammar symbol**; that is why we can map a state stack into a symbol stack as mentioned earlier.

```
0)   $S: stmt <EOF>
1)   stmt:     ID ':=' expr
2)   expr:     expr '+' ID
3)   expr:     expr '-' ID
4)   expr:     ID
```

```
State 0
0)      $S: . stmt <EOF>
1)      stmt: . ID ':=' expr
        GOTO   2 on stmt
        SHIFT 1 on ID

State 1
1)      stmt: ID . ':=' expr
        SHIFT 3 on ':='

State 2
0)      $S: stmt . <EOF>
        SHIFT 4 on <EOF>

State 3
1)      stmt: ID ':=' . expr
2)      expr: . expr '+' ID
3)      expr: . expr '-' ID
4)      expr: . ID
        GOTO   6 on expr
        SHIFT 5 on ID

State 4
0)      $S: stmt <EOF> .

State 5
4)      expr: ID .
```

```
State 6
1)      stmt: ID ':=' expr .
2)      expr: expr . '+' ID
3)      expr: expr . '-' ID
        SHIFT 7 on '+'
        SHIFT 8 on '-'

State 7
2)      expr: expr '+' . ID
        SHIFT 9 on ID

State 8
3)      expr: expr '-' . ID
        SHIFT 10 on ID

State 9
2)      expr: expr '+' ID .

State 10
3)      expr: expr '-' ID .
```

Parser Tables

| | **Action Table** | | | | | **Goto Table** | |
|---|---|---|---|---|---|---|---|
| | **ID** | **':='** | **'+'** | **'-'** | **<EOF>** | **stmt** | **expr** |
| **0** | s1 | | | | | g2 | |
| **1** | | s3 | | | | | |
| **2** | | | | | s4 | | |
| **3** | s5 | | | | | | g6 |
| **4** | *acc* | *acc* | *acc* | *acc* | *acc* | | |
| **5** | r4 | r4 | r4 | r4 | r4 | | |
| **6** | r1 | r1 | s7 | s8 | r1 | | |
| **7** | s9 | | | | | | |
| **8** | s10 | | | | | | |
| **9** | r2 | r2 | r2 | r2 | r2 | | |
| **10** | r3 | r3 | r3 | r3 | r3 | | |