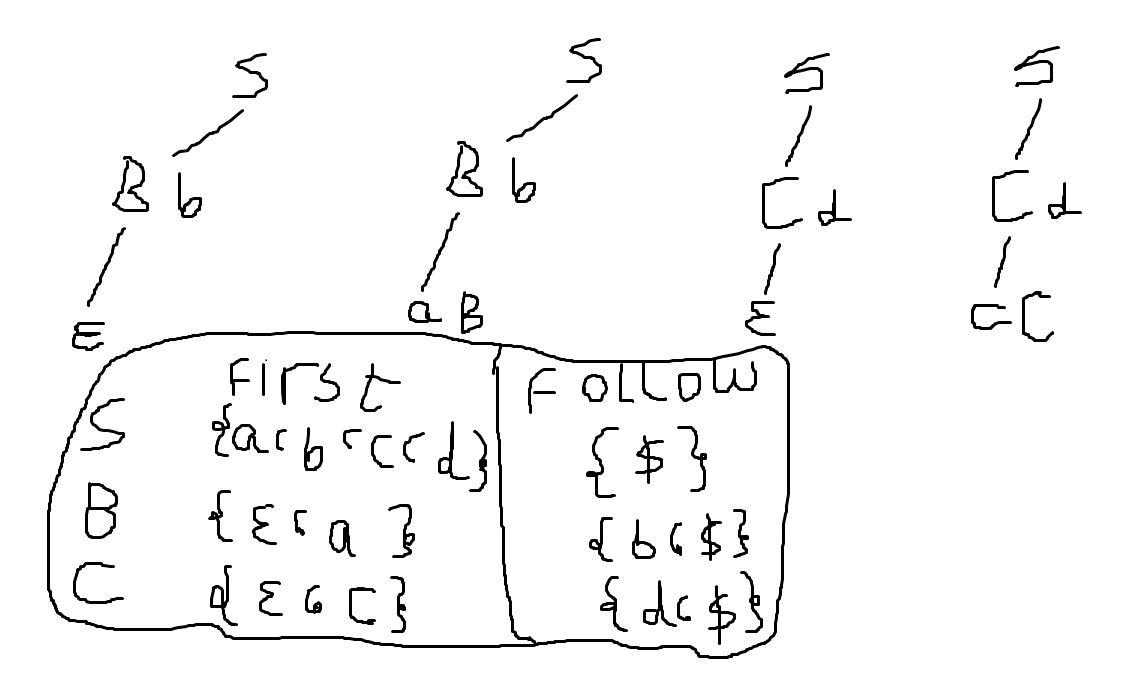


|  |  |  |
| --- | --- | --- |
| Non-terminals | First | Follow |
| S | { a, b, c, d } | { $ } |
| B | { a, 0 } | { b, $ } |
| C | { c, 0 } | { d, $ } |

For explanation:



How to store grammar: using a map has arrays values called grammar

[

“S”: {“Bb”, “Cd”},

“B”: {“ab”, “0”},

“C”: {“cC”, “0”}

]

Make file has non-terminals sorted in reverse to the grammar order:

C

B

S

Make file has terminals:

a

b

c

d

0

Algorithm to find first:

1. Create a map called first-table.
2. Read the first line from the non-terminals file and take its non-terminal.
3. Add the non-terminal as a key to the map.
4. Access the value of this non-terminal in the grammar map, loop over its elements
5. Read the first character and apply the function isTerminal
6. If terminal, then push it to the value of the non-terminal in the first-table map
7. If non-terminal, then access the value of this non-terminal from first-table
8. Loop over this value (array) if not 0 push it the non-terminal we are currently reading from non-terminals file. If 0 then, read the next character from grammar map and test using isTerminal, if terminal do the previous steps (push the value), if not, get the next character until the character end then push 0 (in case there is no terminal)
9. Repeat the reading for the next non-terminals

Algorithm to find follow:

1. Create a map called follow-table.
2. Read the first line of the non-terminals.
3. Create a key for this non-terminal.
4. Access this non-terminal from grammar map, and see if non-terminal exists in the elements, yes: get its next character and test it using isTerminal, if yes then push it to follow-table, if no then get its first and push it if it’s not 0, if 0 then get the next character and see if it’s terminal push, if not then get its first.
5. Access the rest of keys from the grammar table and test whether it has this non-terminal or not, if it has do the same as above.
6. Get the follow of next non-terminal in the parent non-terminal

Grammar:  
Exp -> Term Expr

Expr -> + Term Expr | - Term Expr | 0

Term -> Factor Termp

Termp ->  \* Factor Termp |  / Factor Termp | 0

Factor -> ( Exp ) | Id

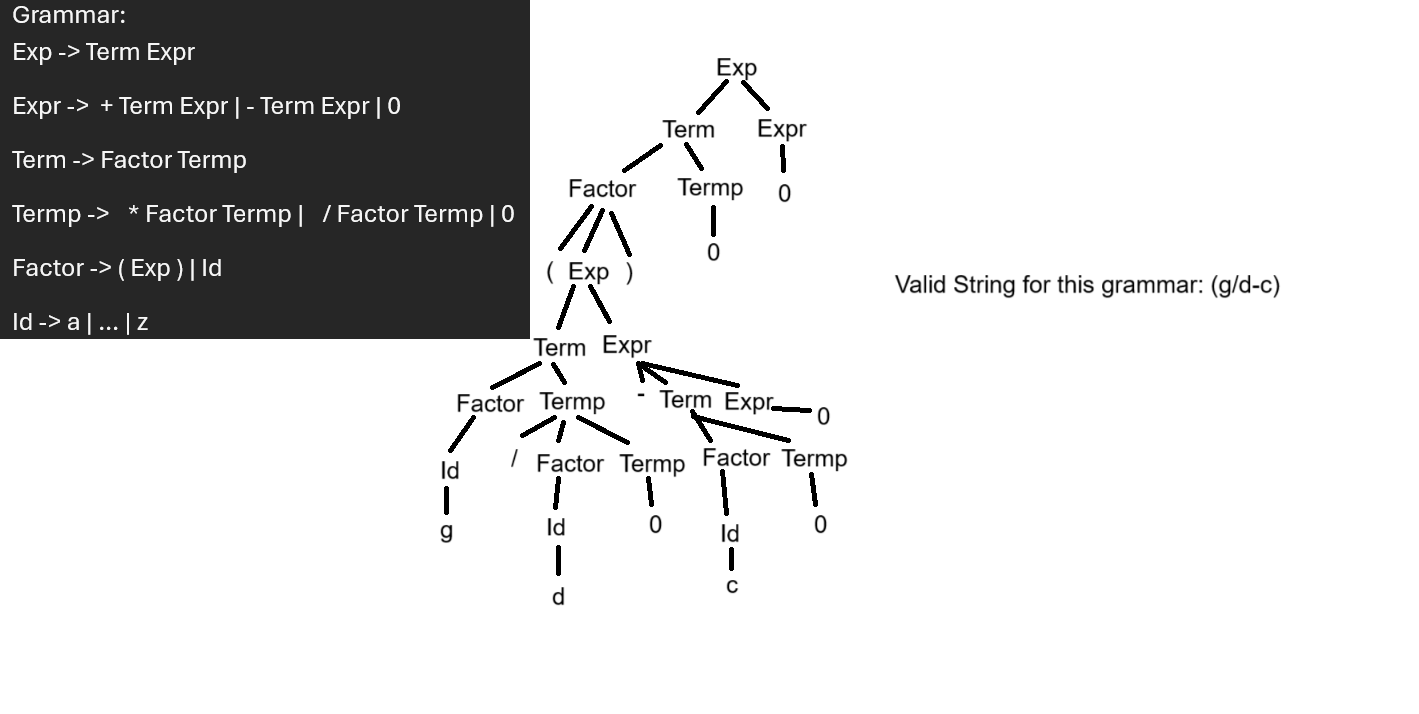
Id -> a | … | z

First-Follow table:

|  |  |  |
| --- | --- | --- |
| Non-Terminal | First | Follow |
| Exp | {(, a, …, z} | {$, )} |
| Expr | {+, -, 0} | {(, a, …, z, $, )} (follow of parent used here) |
| Term | {(, a, …, z} | {\*, /, +, -, $, )} (here also) |
| Termp | {\*, /, 0} | {(, a, …, z, +, -, $, )} |
| Factor | {(, a, …, z} | {(, a, …, z, \*, /, +, -, $, )} |
| Id | {a, …, z} | {\*, /, +, -, $, )} |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | ( | a, …, z | + | - | \* | / | ) $ |
| Exp | Term Expr | Term Expr |  |  |  |  |  |
| Expr | 0 | 0 | + Term Expr | - Term Expr |  |  | 0 0 |
| Term | Factor Termp | Factor Termp |  |  |  |  |  |
| Termp | 0 | 0 | 0 | 0 | \* Factor Termp | / Factor Termp | 0 0 |
| Factor | ( Exp ) | Id |  |  |  |  |  |
| Id |  | a | … | z |  |  |  |  |  |

Predictive Table:



Parsing using predictive table the valid string: (g/d-c)

.1 starting with Exp the starting rule trying to parse ‘(’:

**(**g/d-c)

Term Expr

Factor Termp

( Exp )

.2 first character match, so move to next character ‘g’ and keep going:

**(g**/d-c)

Term Expr

Factor Termp

( Exp )

Term Expr

Factor Termp

Id

**g**

.3 again match, move to next character ‘/’ and keep going:

**(g/**d-c)

Term Expr

Factor Termp

( Exp )

Term Expr

Factor Termp

Id / Factor Termp

**g**

.4 again match, move to next character ‘d’ and keep going:

**(g/d**-c)

Term Expr

Factor Termp

( Exp )

Term Expr

Factor Termp

Id / Factor Termp

**g** Id

d

.5 again match, move to next character ‘-’ and keep going:

**(g/d-**c)

Term Expr

Factor Termp

( Exp )

Term Expr

Factor Termp **-** Term Expr

Id **/** Factor Termp

**g** Id

d 0

.6 again match, move to next character ‘c’ and keep going:

**(g/d-c**)

Term Expr

Factor Termp

( Exp )

Term Expr

Factor Termp **-** Term Expr

Id **/** Factor Termp Factor Termp

**g** Id Id

d 0 **c**

.7 again match, move to next character ‘)’ and keep going:

**(g/d-c)**

Term Expr

Factor Termp

**(** Exp  **)**

Term Expr

Factor Termp **-** Term Expr

Id **/** Factor Termp Factor Termp 0

**g** Id Id 0

d 0 **c**

.8 again match, move to next character ‘$’ and keep going until no remaining non-terminals:

**(g/d-c)**

Term Expr

Factor Termp 0

( Exp  **)** 0

Term Expr

Factor Termp **-** Term Expr

Id **/** Factor Termp Factor Termp 0

**g** Id Id 0

d 0 **c**

.9 finally because we are at ‘$’ and there are no remaining non-terminals this string is valid, and LL(1) accepts it.

Requirements for LL(1):

Scanner, and getNextToken(): returns each token.

Use stack to save the non-terminal of the grammar.

getRule(token, non-terminal): returns the rule that parse to the given token and non-terminal from the predictive table in array. E.g., [`Term`, `Expr`]

equivalent predictive table:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | `(` | `a`, …, `z` | `+` | `-` | `\*` | `/` | `)` `$` |
| `Exp` | `Term` `Expr` | `Term` `Expr` | `~` | `~` | `~` | `~` | `~` `~` |
| `Expr` | `0` | `0` | `+` `Term` `Expr` | `-` `Term` `Expr` | `~` | `~` | `0` `0` |
| `Term` | `Factor` `Termp` | `Factor` `Termp` | `~` | `~` | `~` | `~` | `~` `~` |
| `Termp` | `0` | `0` | `0` | `0` | `\*` `Factor` `Termp` | `/` `Factor` `Termp` | `0` `0` |
| `Factor` | `(` `Exp` `)` | `Id` | `~` | `~` | `~` | `~` | `~` `~` |
| `Id` | `~` | `a` | … |`z` | `~` | `~` | `~` | `~` | `~` `~` |

Algorithm for LL(1):

// initialize the stack with “$”  
// read first token

// stack.push(getRule(first token, first non-terminal)) add the rule

// while stack top isn’t “$”:

// get the top array, and first element from it

// if it equals ~: return false

// if it’s terminal:  
 // if it doesn’t equal the token: return false

// if it equals token: remove this first element from the array, or pop the array if it has only one element. Make the pointer points to the next token.

// if it's non-terminal: stack.push(getRule(first token, this element))

// reaching here means the stack only has “$”. If getNextToken() gives “”: return true.