

# 50006 - Compilers - Lecture 2

Oliver Killane

05/01/22

## Syntax Analysis

- **Syntax** The grammatical structure of the language expressed through rules. The compiler must determine if the program is syntactically correct.

Parser Generators tools used to generate the code to perform the analysis phases of a compiler from the language's formal specification (usually similar to **Bakus-Naur Form**).

- **Semantics** meaning associated with program. For example type-checking, or checking for memory safety.

**Compiler Generators/Compilers** are an active area of research. They generate the synthesis phase from a specification of the semantics of the source & target language.

These tools are promising but usually the code is written manually instead.

## Bakus-Naur Form

Also called **Backus Normal Form** is a context-free grammar used to specify the syntactic structure of a language.

$$stat \rightarrow 'if' '(' expr ')' stat 'else' stat$$

- **Context Free Grammar**

A context free grammar is a set of **Productions**. Associated with a set of tokens (terminals), a set of non-terminals and a start (non-terminal) symbol.

Each production is of the form:

$$\text{single non-terminal} \rightarrow \text{String of terminals \& non-terminals}$$

The simple LHS makes it a context-free grammar, more complex LHSs are possible in context-sensitive grammars.

- **Production**

Shows one valid way to expand a non-terminal symbol into a string of terminals & non-terminals.

$$expr \rightarrow '0'$$

$$expr \rightarrow '1'$$

$$expr \rightarrow expr + expr$$

$$expr \rightarrow '0' \mid '1' \mid expr + expr \quad \text{Can combine two productions for more concise representation.}$$

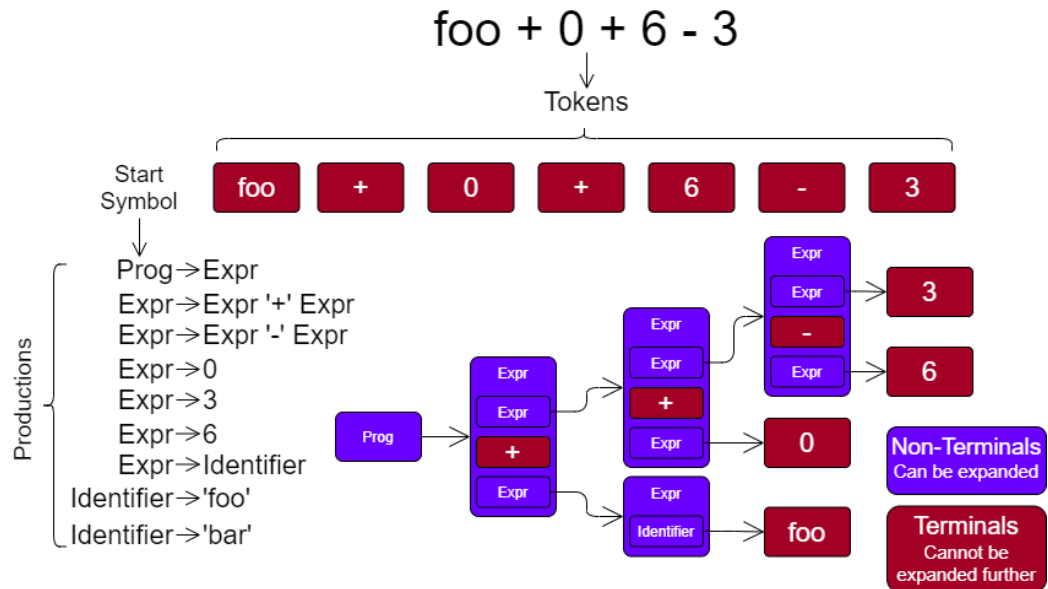
- **Terminals & Non-Terminals**

Symbols that cannot be further expanded, these are the tokens generated from lexical analysis (e.g brackets, identifiers, semicolons).

- Parse Tree

Shows how the string is derived from the start symbol.

This tree is a graphical proof that a given sentence is within the grammar. Parsing is the process of generating this.



We can express the grammar as a tuple:

$G = (S, P, t, nt)$  where S - start symbol, P - productions, t - terminals, nt - nonterminals, and  $S \in nt$

The input is entirely terminals, we use productions & pattern matching to analyse.

$\langle \text{Stmt} \rangle \rightarrow (\text{Id}) = \langle \text{Expr} \rangle ;$
$\langle \text{Stmt} \rangle \rightarrow \{ \langle \text{StmtList} \rangle \}$
$\langle \text{Stmt} \rangle \rightarrow \text{if } ( \langle \text{Expr} \rangle ) \langle \text{Stmt} \rangle$
$\langle \text{StmtList} \rangle \rightarrow \langle \text{Stmt} \rangle$
$\langle \text{StmtList} \rangle \rightarrow \langle \text{StmtList} \rangle \langle \text{Stmt} \rangle$
$\langle \text{Expr} \rangle \rightarrow (\text{Id})$
$\langle \text{Expr} \rangle \rightarrow (\text{Num})$
$\langle \text{Expr} \rangle \rightarrow \langle \text{Expr} \rangle \langle \text{Optr} \rangle \langle \text{Expr} \rangle$
$(\text{Id}) \rightarrow x$
$(\text{Id}) \rightarrow y$
$(\text{Num}) \rightarrow 0$
$(\text{Num}) \rightarrow 1$
$(\text{Num}) \rightarrow 9$
$(\text{Optr}) \rightarrow >$
$(\text{Optr}) \rightarrow +$

	$\langle \text{Expr} \rangle$	$\langle \text{Optr} \rangle$	$\langle \text{Num} \rangle$	$\langle \text{Stmt} \rangle$
if (	<u><math>\langle \text{Expr} \rangle</math></u>			$\langle \text{Stmt} \rangle$
if (	<u><math>\langle \text{Expr} \rangle</math></u>	$\langle \text{Optr} \rangle$	<u><math>\langle \text{Expr} \rangle</math></u>	$\langle \text{Stmt} \rangle$
if (	<u><math>\langle \text{Id} \rangle</math></u>	$\langle \text{Optr} \rangle$	<u><math>\langle \text{Expr} \rangle</math></u>	$\langle \text{Stmt} \rangle$
if (	x	<u><math>\langle \text{Optr} \rangle</math></u>	<u><math>\langle \text{Expr} \rangle</math></u>	$\langle \text{Stmt} \rangle$
if (	x	>	<u><math>\langle \text{Expr} \rangle</math></u>	$\langle \text{Stmt} \rangle$
if (	x	>	<u><math>\langle \text{Num} \rangle</math></u>	$\langle \text{Stmt} \rangle$
if (	x	>	9	$\langle \text{Stmt} \rangle$
if (	x	>	9	{ <u><math>\langle \text{StmtList} \rangle</math></u> }
if (	x	>	9	{ <u><math>\langle \text{StmtList} \rangle</math></u> $\langle \text{Stmt} \rangle$ }
if (	x	>	9	{ <u><math>\langle \text{Stmt} \rangle</math></u> $\langle \text{Stmt} \rangle$ }
if (	x	>	9	{ <u><math>\langle \text{Id} \rangle = \langle \text{Expr} \rangle ;</math></u> $\langle \text{Stmt} \rangle$ }
if (	x	>	9	{ x = <u><math>\langle \text{Expr} \rangle</math></u> ; $\langle \text{Stmt} \rangle$ }
if (	x	>	9	{ x = <u><math>\langle \text{Num} \rangle</math></u> ; $\langle \text{Stmt} \rangle$ }
if (	x	>	9	{ x = 0 ; $\langle \text{Stmt} \rangle$ }
if (	x	>	9	{ x = 0 ; <u><math>(\text{Id}) = \langle \text{Expr} \rangle ;</math></u> $\langle \text{Expr} \rangle$ ; }
if (	x	>	9	{ x = 0 ; y = <u><math>\langle \text{Expr} \rangle \langle \text{Optr} \rangle \langle \text{Expr} \rangle</math></u> ; }
if (	x	>	9	{ x = 0 ; y = <u><math>(\text{Id}) \langle \text{Optr} \rangle \langle \text{Expr} \rangle</math></u> ; }
if (	x	>	9	{ x = 0 ; y = y + <u><math>\langle \text{Expr} \rangle</math></u> ; }
if (	x	>	9	{ x = 0 ; y = y + <u><math>\langle \text{Num} \rangle</math></u> ; }
if (	x	>	9	{ x = 0 ; y = y + 1 ; }

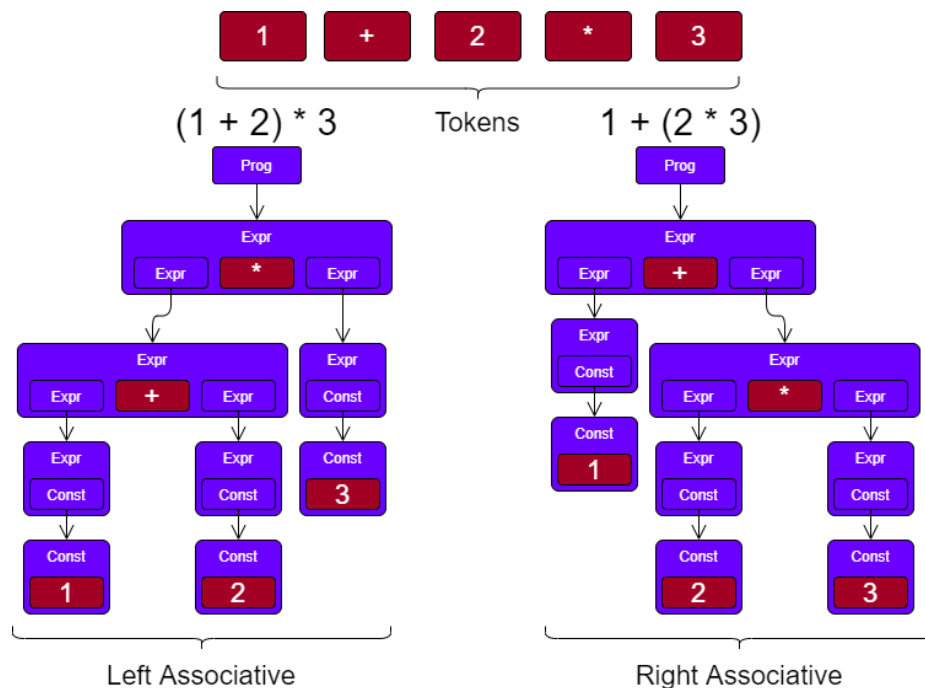
An example with a basic C-style if statement (sourced from wikipedia)

- Starting with the start symbol we can use the productions to replace each non-terminal with some string of terminals and non-terminals, continually expanding the non-terminals.
- A string derived that only consists of terminals is a **sentence** (cannot derive any further string of symbols).
- The **language** of a grammar is the set of all sentences that can be derived from the start symbol.

## Grammar Ambiguity

In some grammars there may be ambiguity (e.g multiple different productions can be applied to the same string, or the same production in different ways).

For example  $3 - 2 - 1$  can be  $(3 - 2) - 1$  or  $3 - (2 - 1)$ . This ambiguity results in multiple possible parse trees.



Often the language designer will specify how to deal with ambiguities (assigning operator precedence & associativity) using the grammar.

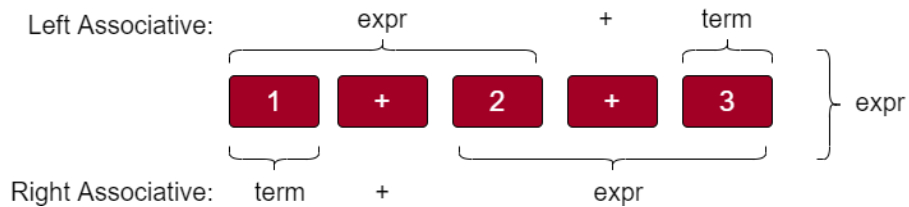
## Precedence and Associativity

Precedence determines which operators are applied first, and associativity how operators of the same precedence are applied.

## grammar for Associativity

Associativity can be enforced by using left or right recursive productions.

$term \rightarrow$	$const \mid ident$	Define a base term.
$expr \rightarrow$	$expr + term$	Left associative, the split is on the final +.
$expr \rightarrow$	$term + expr$	Right associative, the split is on the first +.

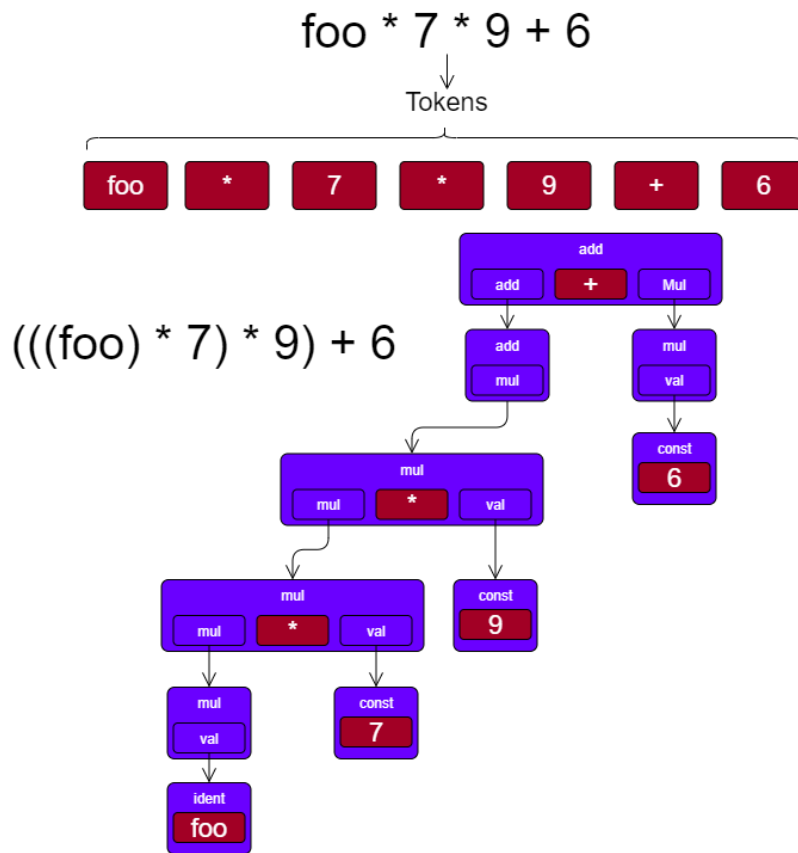


## Grammar for Precedence

We can *layer* our grammar such that some symbols are parsed first.

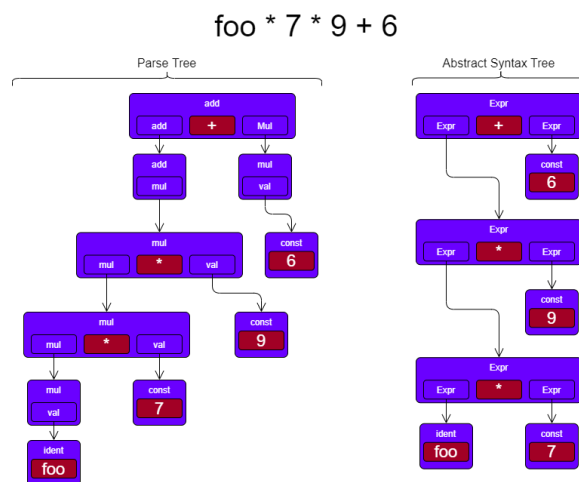
$add \rightarrow$	$add + mul \mid add - mul \mid mul$
$mul \rightarrow$	$mul * val \mid mul / val \mid val$
$val \rightarrow$	$const \mid ident$

By splitting the expression into an add and multiply stage (both left associative), the second layer (*mul*) has higher precedence. To add more levels of precedence we can use more layers.



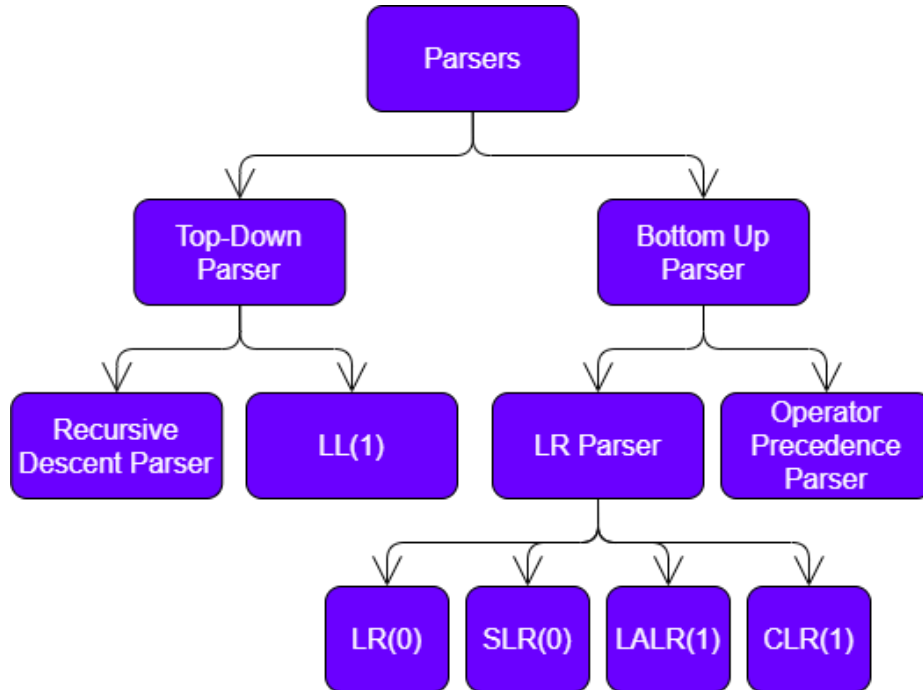
## Parse Tree vs Abstract Syntax Tree

The abstract syntax tree has similar structure, but does not need much of the extra information (layers for expressions used to enforce precedence for example).



# Parsers

Parsers check the grammar is correct & construct an **AST**.



## Top-Down Parsing

Also called predictive parsing.

- Input is derived from a start symbol.
- Parser takes tokens from left  $\rightarrow$  right, each only once.
- **For each step**

In each step the parser uses:

- the current token
- the current non-terminal being derived
- the current non-terminal's production rules

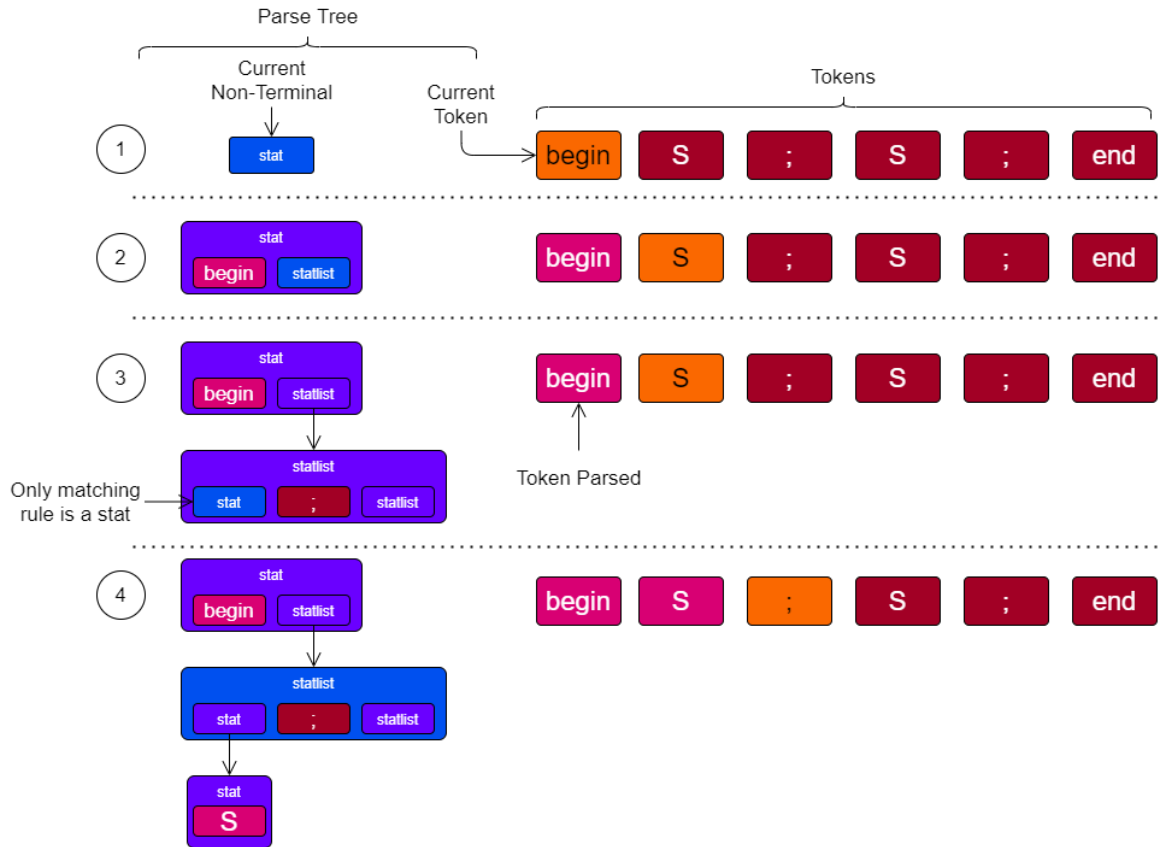
By using the production rules & the current token we can predict the next production rule, and use this to either:

1. Get another non-terminal to derive from, and potentially others for subsequent steps
  2. Get a terminal which should match the current token (or else an error has occurred/the program is syntactically invalid)
- We are using the grammar *left  $\rightarrow$  right*.

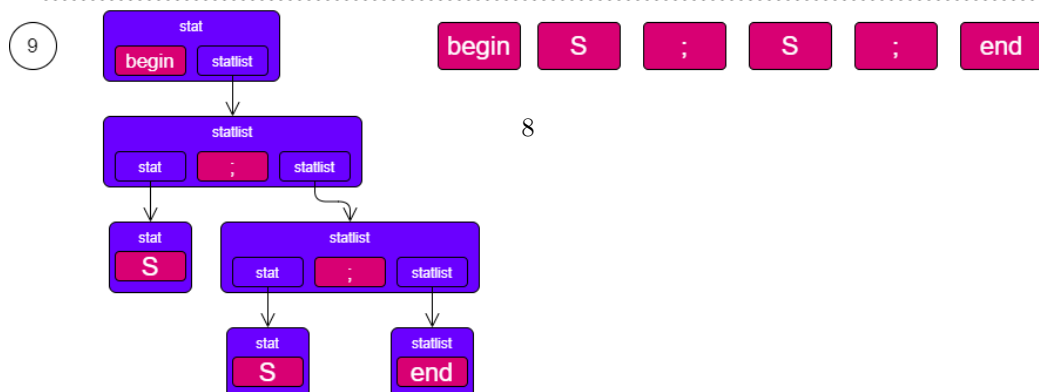
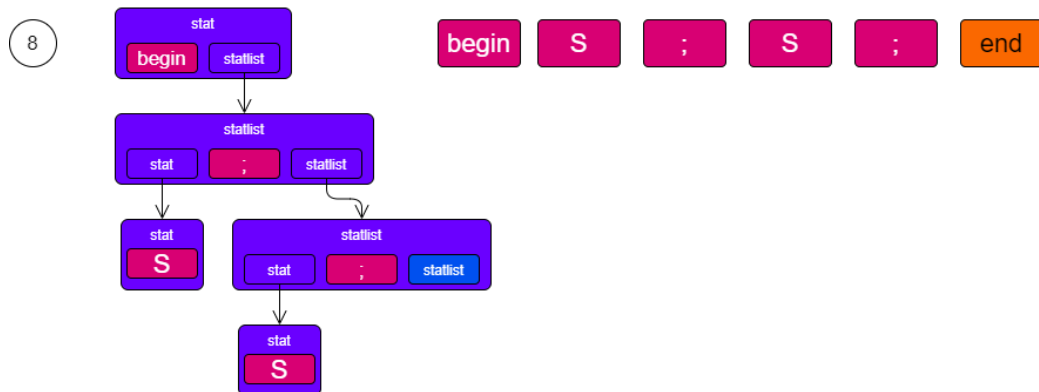
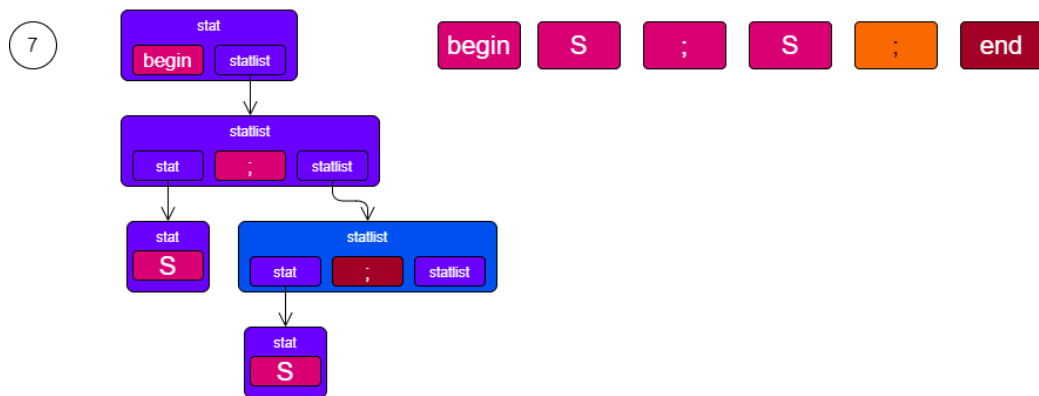
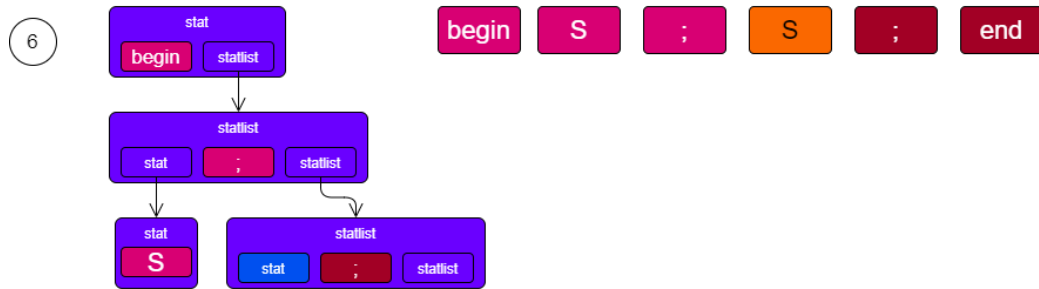
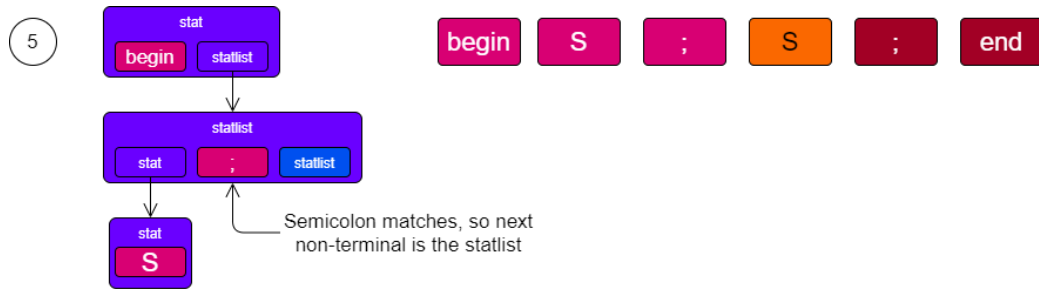
For example with the grammar:

$$\begin{aligned}
 stat &\rightarrow 'begin' \ statlist \\
 stat &\rightarrow 'S' \\
 statlist &\rightarrow 'end' \\
 statlist &\rightarrow stat \ ; \ statlist
 \end{aligned}$$

Start symbol is *stat*.







## Production Choice

We may have a grammar where we cannot determine which production for a non-terminal token to use based on the first symbol.

$$\begin{aligned} stat &\rightarrow \text{'loop' } statlist \text{'until' } expr \\ stat &\rightarrow \text{'loop' } statlist \text{'while' } expr \\ stat &\rightarrow \text{'loop' } statlist \text{'forever' } \end{aligned}$$

When we have token 'loop' we cannot determine which production to use. There are two methods to deal with this:

- **Delay the choice**

Delay creating this tree (from stat) until it is known which production matches.

It is still possible to create the statlist inside while doing so.

- **Modify the grammar**

Change the grammar to factor out the difference.

$$\begin{aligned} stat &\rightarrow \text{'loop' } statlist \text{ loopstat} \\ loopstat &\rightarrow \text{'until' } expr \\ loopstat &\rightarrow \text{'while' } expr \\ loopstat &\rightarrow \text{'forever' } \end{aligned}$$

However there are more difficult problems, which can be more easily fixed with bottom-up parsing.

## Left recursion

Right recursive grammars produce right recursive parse trees:

$$\begin{aligned} add &\rightarrow mul \text{'+' } add \\ add &\rightarrow mul \text{'-' } add \\ add &\rightarrow mul \\ mul &\rightarrow val \text{'*' } mul \\ mul &\rightarrow val \text{'/' } mul \\ mul &\rightarrow val \\ val &\rightarrow integer \\ val &\rightarrow identifier \end{aligned}$$

We consider this right-recursive as the recursion on productions is right of the symbol.

Top-down parsing implemented through **Recursive Descent Parsers** cannot do left recursive grammars. This is as it will result in an infinite recursion.

$$add \rightarrow add \text{'+' } mul$$

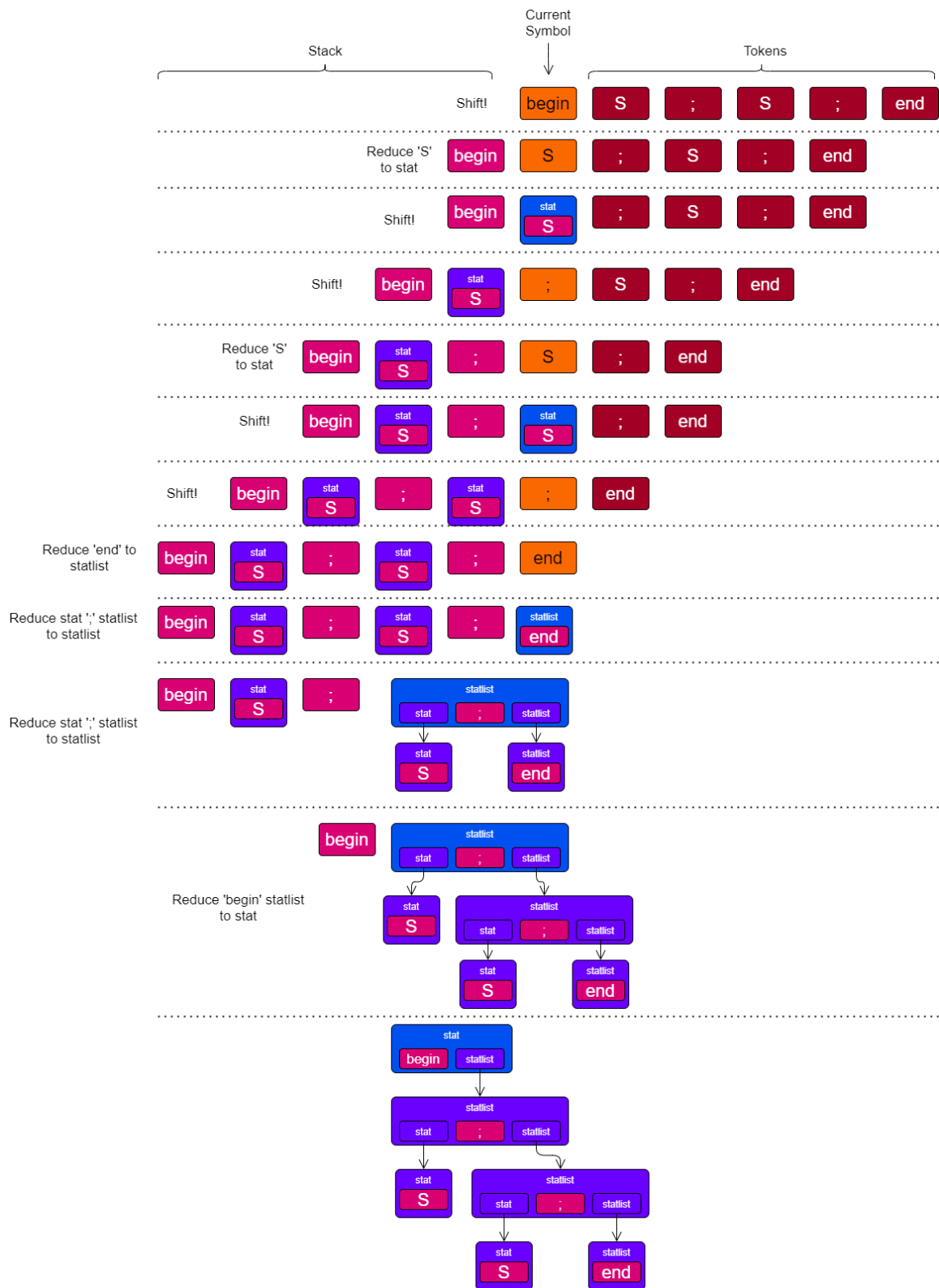
If attempting to parse this production:

```
1 def parse_add(input):
2     (add_parse_tree, rest) = parse_add(input)
3     rest = parse_plus(rest) # infinite recursion
4     (mul_parse_tree, rest) = parse_mul(rest)
5     return (tree(add_parse_tree, mul_parse_tree), rest)
```

## Bottom-up Parsing

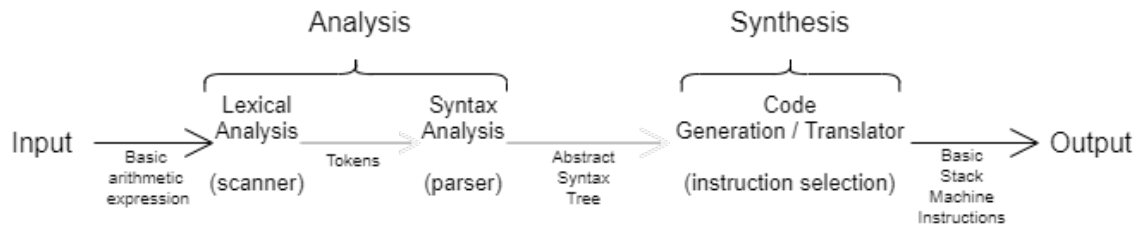
- The grammar's productions are used *right*  $\rightarrow$  *left*.
- Input is compared against the right hand side to produce a non-terminal on the left.
- Parsing is complete when the whole input is replaced by the start symbol.

Bottom up parsers are difficult to implement, so parser generators are recommended.



# Simple Complete Compiler

A very basic compiler written in Haskell to convert basic arithmetic expressions into instructions for a basic stack machine.



```

1 {-
2 Simple compiler example:
3 Arithmetic Expressions -> Stack Machine Instructions
4
5 Original Description:
6 "Compiling arithmetic expressions into code for a stack machine. This is not a
7 solution to Exercise 2 - it's an executable version of the code generator for
8 expressions, which is given in the notes. Build on it to yield a code generator
9 for statements.
10
11 Paul Kelly, Imperial College, 2003
12
13 Tested with Hugs (Haskell 98 mode), Feb 2001 version"
14
15 Changes:
16 - This version has been updated to work with Haskell version 8.6.5
17 - Use of where over let & general refactoring
18 - Fixed bug with execute (missing patterns for invalid stack instructions)
19 - New grammar to support multiplication and division
20
21 Grammar:
22 add    -> mul + add | mul - add | mul
23 mul    -> factor * mul | factor / mul | factor
24 factor -> number | identifier
25
26 + - (right associative, low precedence)
27 * / (right associative, high precedence)
28
29 Eg:
30 > tokenise "a+b*17"
31 [IDENT a,PLUS,IDENT b,MUL,NUM 17]
32
33 > parser (tokenise "a+b-17")
34 Plus (Ident a, Minus (Ident b, Num 17))
35
36 > parser (tokenise "3-3-3*77*a-4")
37 Minus (Num 3, Minus (Num 3, Minus (Mul (Num 3, Mul (Num 77, Ident a)), Num 4)))
38
39 > compile "a+b+c*7-3"
40 [PushVar a,PushVar b,PushVar c,PushConst 7,MulToS,PushConst 3,SubToS,AddToS,AddToS]
41
42 > translate (parser (tokenise "a+b/17"))
43 [PushVar a,PushVar b,PushConst 17,DivToS,AddToS]
44
  
```

```

45 > putStr (runAnimated [("a", 9)] [] (translate (parser (tokenise "100+a*3-17"))))
46 [100]
47 [9,100]
48 [3,9,100]
49 [27,100]
50 [17,27,100]
51 [10,100]
52 [110]
53 [110]
54
55 -}
56
57 import Data.Char ( isDigit , isAlpha , digitToInt )
58 import Text.Parsec (tokens , Stream (uncons))
59
60 — Token data type
61 data Token
62   = IDENT [Char] | NUM Int | PLUS | MINUS | MUL | DIV
63
64 — Ast (abstract syntax tree) data type
65 data Ast
66   = Ident [Char] | Num Int | Plus Ast Ast | Minus Ast Ast | Mul Ast Ast | Div Ast Ast
67
68 — Instruction data type
69 —
70 — PushConst pushes a given number onto the stack; AddToS takes the top
71 — two numbers from the top of the stack (ToS), and them and pushes the sum.
72 — (We have to invent new names to avoid clashing with MUL, Mul etc above)
73 data Instruction
74   = PushConst Int | PushVar [Char] | AddToS | SubToS | MulToS | DivToS
75
76 instance Show Token where
77   showsPrec p (IDENT name) = showString "IDENT " . showString name
78   showsPrec p (NUM num) = showString "NUM " . shows num
79   showsPrec p (PLUS) = showString "PLUS"
80   showsPrec p (MINUS) = showString "MINUS"
81   showsPrec p (MUL) = showString "MUL"
82   showsPrec p (DIV) = showString "DIV"
83
84 instance Show Ast where
85   showsPrec p (Ident name) = showString "Ident " . showString name
86   showsPrec p (Num num) = showString "Num " . shows num
87   showsPrec p (Plus e1 e2) = showString "Plus (" . shows e1 . showString ", " . shows
88     ↪ e2 . showString ")"
89   showsPrec p (Minus e1 e2) = showString "Minus (" . shows e1 . showString ", " .
90     ↪ shows e2 . showString ")"
91   showsPrec p (Mul e1 e2) = showString "Mul (" . shows e1 . showString ", " . shows
92     ↪ e2 . showString ")"
93   showsPrec p (Div e1 e2) = showString "Div (" . shows e1 . showString ", " . shows
94     ↪ e2 . showString ")"
95
96 instance Show Instruction where
97   showsPrec p (PushConst n) = showString "PushConst " . shows n
98   showsPrec p (PushVar name) = showString "PushVar " . showString name
99   showsPrec p AddToS = showString "AddToS"
100  showsPrec p SubToS = showString "SubToS"
101  showsPrec p MulToS = showString "MulToS"
102  showsPrec p DivToS = showString "DivToS"
103
104 — Parse the tokens (top-down) by parsing each expression to get a new parse

```

```

101 — tree and the rest of the tokens. No tokens should remain after parsing.
102 parser :: [Token] -> Ast
103 parser tokens
104   | null rest = tree
105   | otherwise = error "(parser) excess rubbish"
106   where
107     (tree, rest) = parseAdd tokens
108
109 parseAdd :: [Token] -> (Ast, [Token])
110 parseAdd tokens
111   = case rest of
112     (PLUS : rest2) -> let (subexptree, rest3) = parseAdd rest2 in (Plus multree
113       ↪ subexptree, rest3)
114     (MINUS : rest2) -> let (subexptree, rest3) = parseAdd rest2 in (Minus multree
115       ↪ subexptree, rest3)
116     othertokens -> (multree, othertokens)
117   where
118     (multree, rest) = parseMul tokens
119
120 parseMul :: [Token] -> (Ast, [Token])
121 parseMul tokens
122   = case rest of
123     (MUL : rest2) -> let (subexptree, rest3) = parseMul rest2 in (Mul factortree
124       ↪ subexptree, rest3)
125     (DIV : rest2) -> let (subexptree, rest3) = parseMul rest2 in (Div factortree
126       ↪ subexptree, rest3)
127     othertokens -> (factortree, othertokens)
128   where
129     (factortree, rest) = parseFactor tokens
130
131 parseFactor :: [Token] -> (Ast, [Token])
132 parseFactor ((NUM n):restoftokens) = (Num n, restoftokens)
133 parseFactor ((IDENT x):restoftokens) = (Ident x, restoftokens)
134 parseFactor [] = error "(parseFactor) Attempted to parse empty list"
135 parseFactor (t:_) = error $ "(parseFactor) error parsing token " ++ show t
136
137 — Lexical analysis – tokenisation
138 tokenise :: [Char] -> [Token]
139 tokenise [] = [] — (end of input)
140 tokenise (' ':rest) = tokenise rest — (skip spaces)
141 tokenise ('+':rest) = PLUS : (tokenise rest)
142 tokenise ('-':rest) = MINUS : (tokenise rest)
143 tokenise ('*':rest) = MUL : (tokenise rest)
144 tokenise ('/':rest) = DIV : (tokenise rest)
145 tokenise (ch:rest)
146   | isDigit ch = (NUM dn):(tokenise drest2)
147   | isAlpha ch = (IDENT an):(tokenise arest2)
148   where
149     (dn, drest2) = convert (ch:rest)
150     (an, arest2) = getname (ch:rest)
151 tokenise (c:_) = error $ "(tokenise) unexpected character " ++ [c]
152
153 getname :: [Char] -> ([Char], [Char]) — (name, rest)
154 getname = flip getname' []
155 where
156   getname' :: [Char] -> [Char] -> ([Char], [Char])
157   getname' [] chs = (chs, [])
158   getname' (ch : str) chs
159     | isAlpha ch = getname' str (chs++[ch])

```

```

157 | otherwise = (chs, ch : str)
158
159 convert :: [Char] -> (Int, [Char])
160 convert = flip conv' 0
161 where
162   conv' [] n = (n, [])
163   conv' (ch : str) n
164     | isDigit ch = conv' str ((n*10) + digitToInt ch)
165     | otherwise = (n, ch : str)
166
167 — Translate — the code generator
168 translate :: Ast -> [Instruction]
169 translate (Num n) = [PushConst n]
170 translate (Ident x) = [PushVar x]
171 translate (Plus e1 e2) = translate e1 ++ translate e2 ++ [AddToS]
172 translate (Minus e1 e2) = translate e1 ++ translate e2 ++ [SubToS]
173 translate (Mul e1 e2) = translate e1 ++ translate e2 ++ [MulToS]
174 translate (Div e1 e2) = translate e1 ++ translate e2 ++ [DivToS]
175
176 compile :: [Char] -> [Instruction]
177 compile = translate . parser . tokenise
178
179 — Execute, run — simulate the machine running the stack instructions
180 —
181 — Note that this simple machine is too simple to be realistic;
182 — (1) 'execute' doesn't return the store, so no instruction can change it
183 — (2) 'run' forgets each instruction as it is executed, so can't do loops
184
185 — The state of the machine consists of a store (a set of associations
186 — between variable and their values), together with a stack:
187
188 type Stack = [Int]
189 type Store = [(Char, Int)]
190
191 — 'run' executes a sequence of instructions using a specified
192 — store, and starting from a given stack
193 —
194 run :: Store -> Stack -> [Instruction] -> Stack
195
196 run store stack [] = stack
197 run store stack (i : is) = run store (execute store stack i) is
198
199 — 'execute' applies a given instruction to the current state of the
200 — machine — ie the store and the stack
201 —
202 execute :: Store -> Stack -> Instruction -> Stack
203 execute store (a : b: rest) AddToS = ( (b+a) : rest )
204 execute store (a : b: rest) SubToS = ( (b-a) : rest )
205 execute store (a : b: rest) MulToS = ( (b*a) : rest )
206 execute store (a : b: rest) DivToS = ( (b `div` a) : rest )
207 execute store rest (PushConst n) = ( n : rest )
208 execute store rest (PushVar x) = ( n : rest )
209 where n = valueOf x store
210 execute store [x] instr = error $ "(execute) attempted to run " ++ show instr ++ "
211   ↪ with only" ++ show x ++ " on the stack"
212 execute store [] instr = error $ "(execute) attempted to run " ++ show instr ++ "
213   ↪ with an empty stack"
214
215 valueOf x [] = error ("no value for variable " ++ show x)
216 valueOf x ( (y,n) : rest ) = if x==y then n else valueOf x rest

```

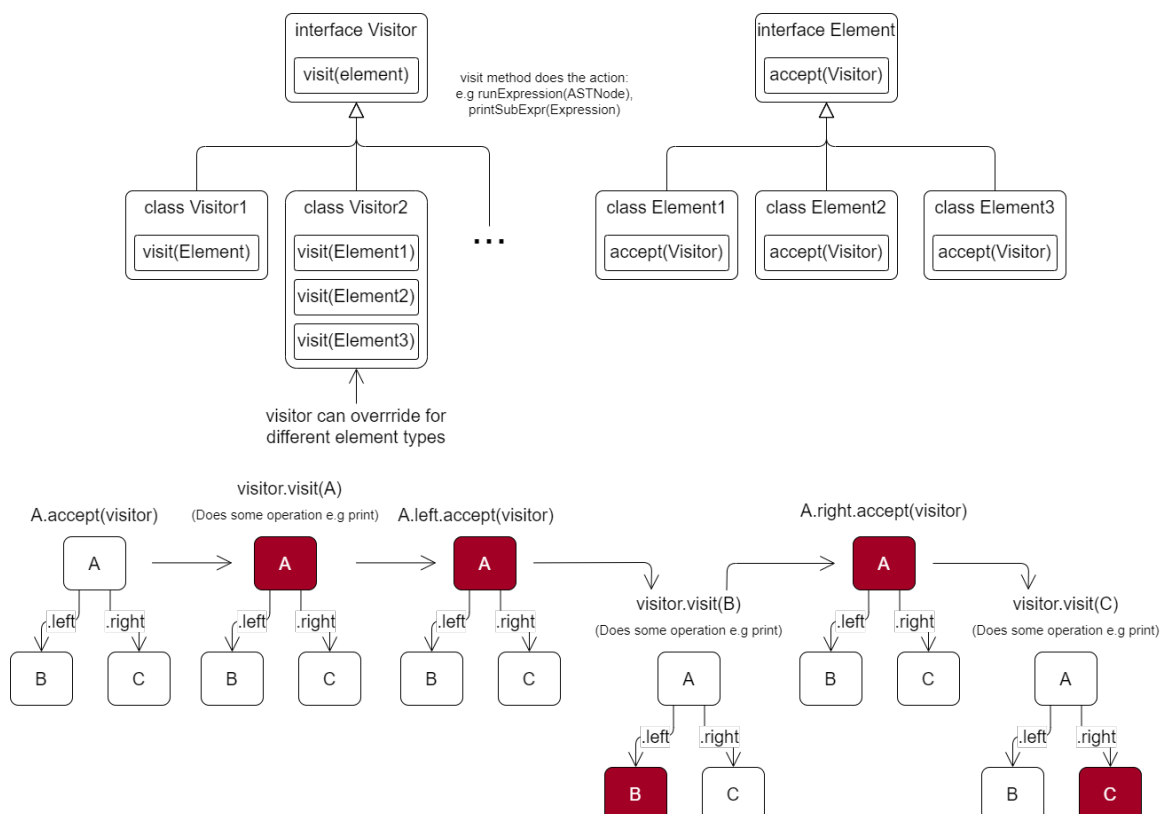


```

215
216 — runAnimated does what run does but shows the stack after each step:
217 —
218 runAnimated :: Store -> Stack -> [Instruction] -> [Char]
219 runAnimated store stack [] = show stack
220 runAnimated store stack (i : is) = show newstack ++ "\n" ++ runAnimated store
    ↪ newstack is
221 where
222     newstack = execute store stack i

```

## Visitor Pattern



The advantage of this pattern is that the data structure and operations are separated. This means new operations can be added easily by simply creating and passing a new visitor, which is then able to operate on the structure (e.g a tree as in the diagram).

Example usage could be: turtle operations on an **abstract syntax tree** of a turtle program (visitors for different colour, text styles, languages).