

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.

$$expr \rightarrow '0'$$

$$expr \rightarrow '1'$$

$$expr \rightarrow expr + expr$$

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

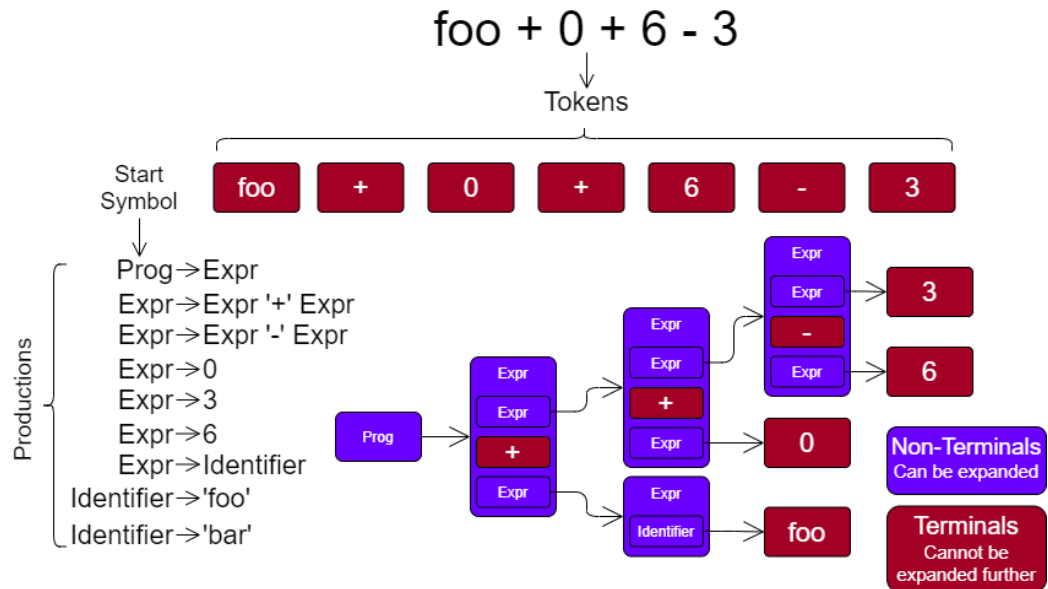
- **Terminals & Non-Terminals**

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.

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

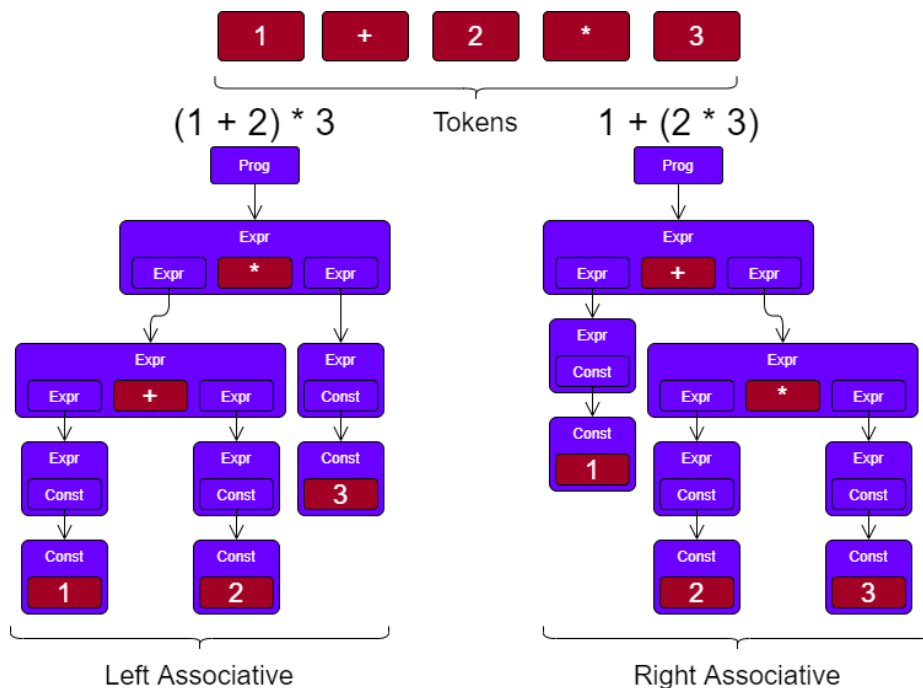
An example with a basic C-style if statement (sourced from wikipedia)

- 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.

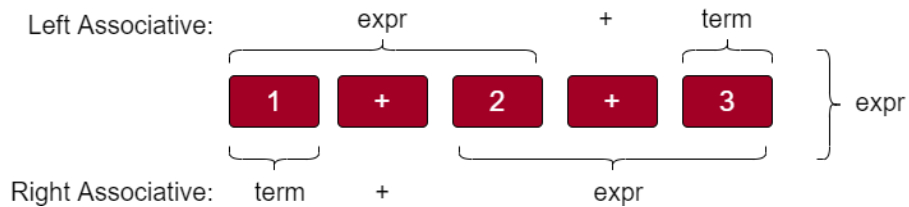
Precedence and Associativity

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 +.

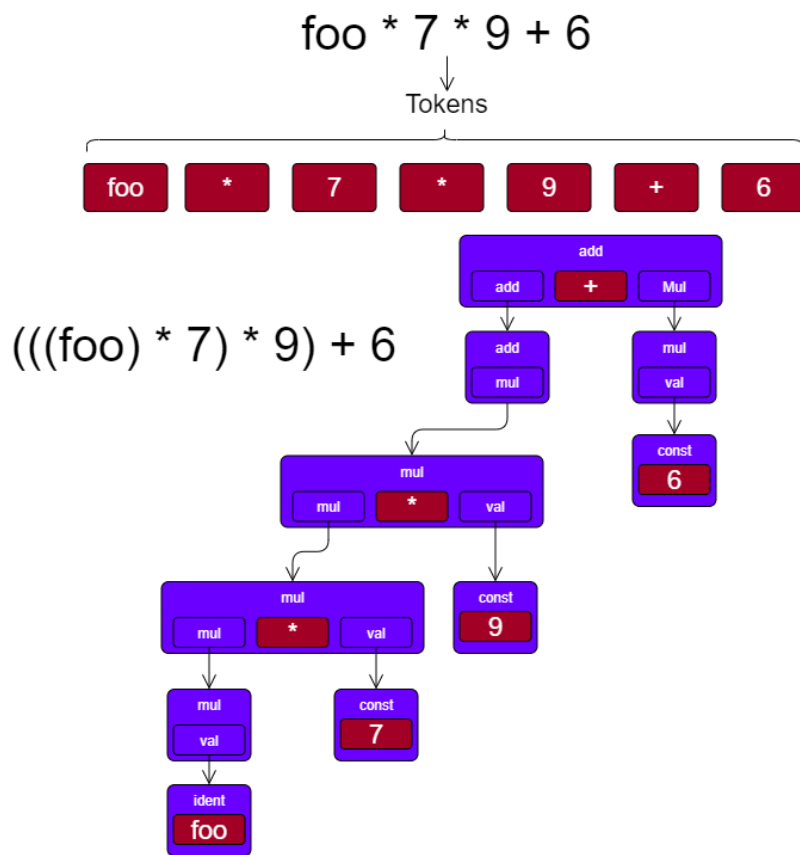


Grammar for Precedence

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

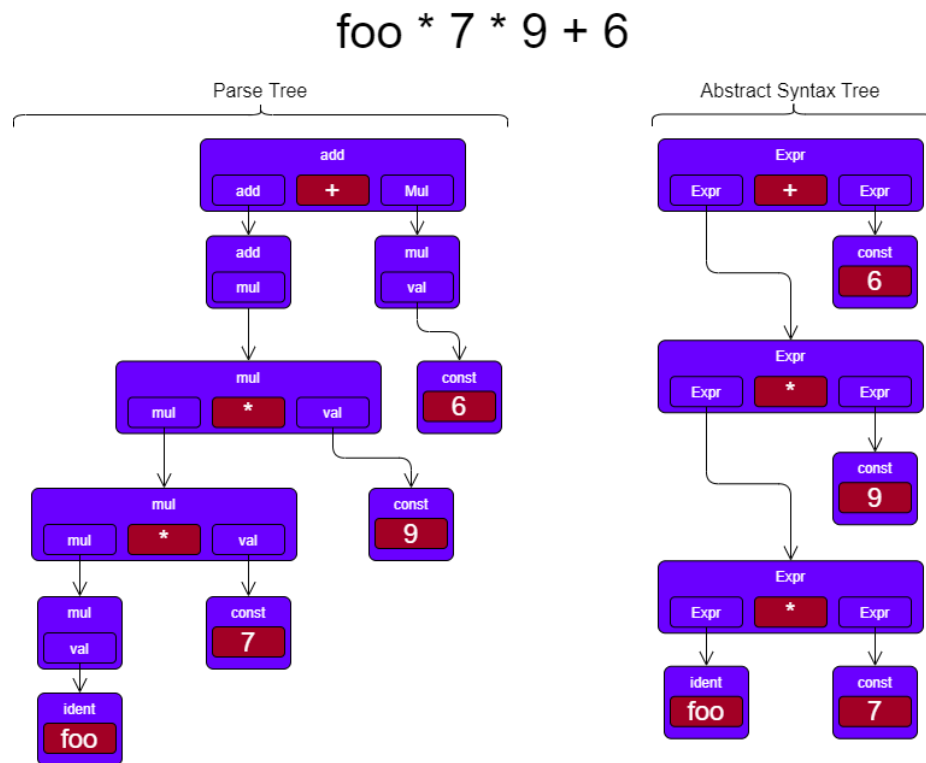
$add \rightarrow$	$add + mul \mid add - mul \mid mul$
$mul \rightarrow$	$mul * val \mid mul / val \mid val$
$val \rightarrow$	$const \mid ident$

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**. There are two types of parsers:

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:

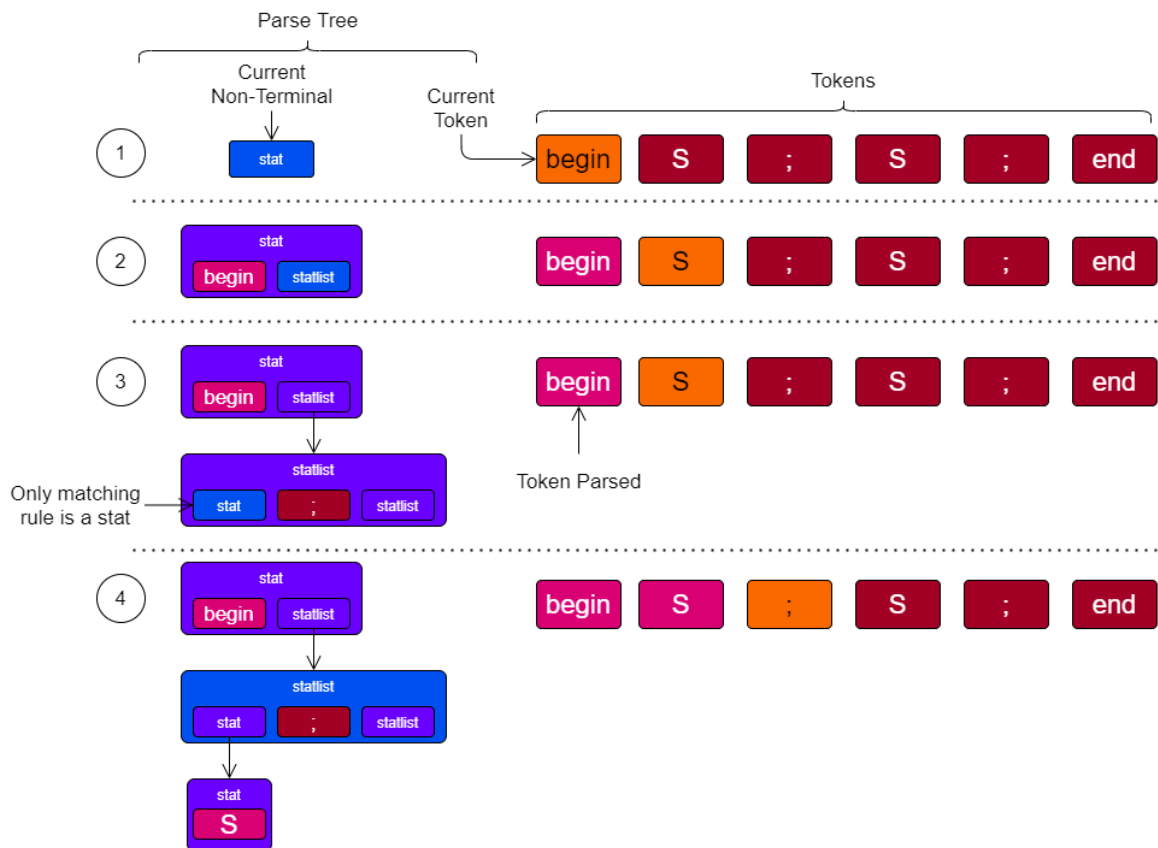
1. Get another non-terminal to derive from, and potentially others for subsequent steps

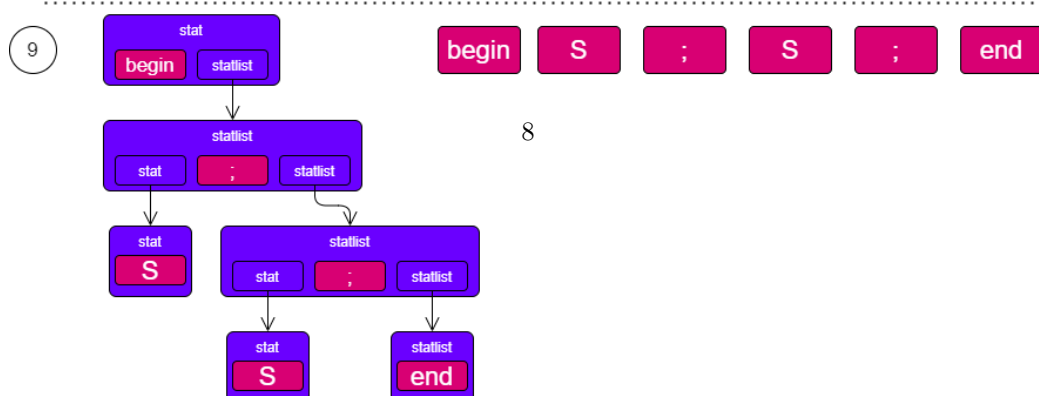
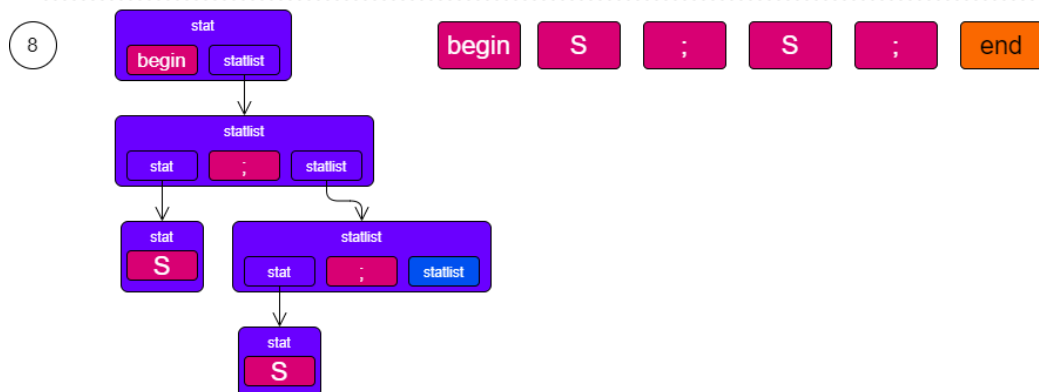
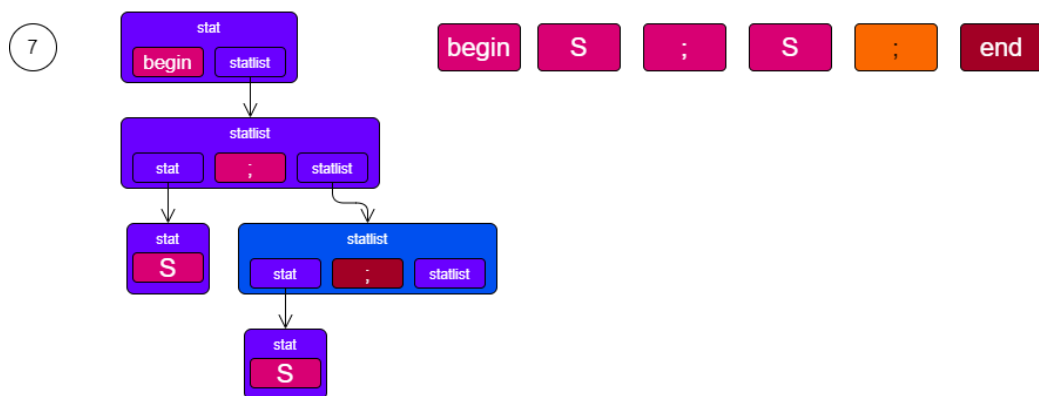
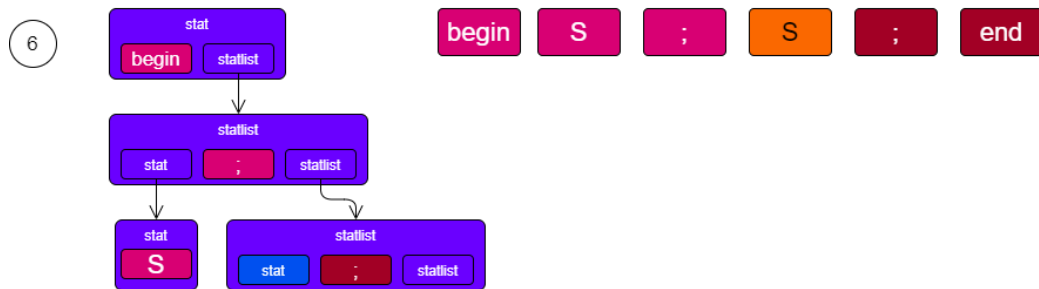
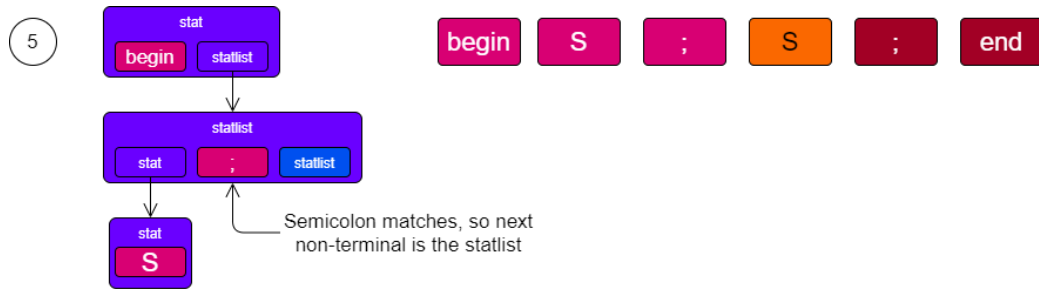
2. Get a terminal which should match the current token (or else an error has occurred/the program is syntactically invalid)
- We are using the grammar *left* \rightarrow *right*.

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.

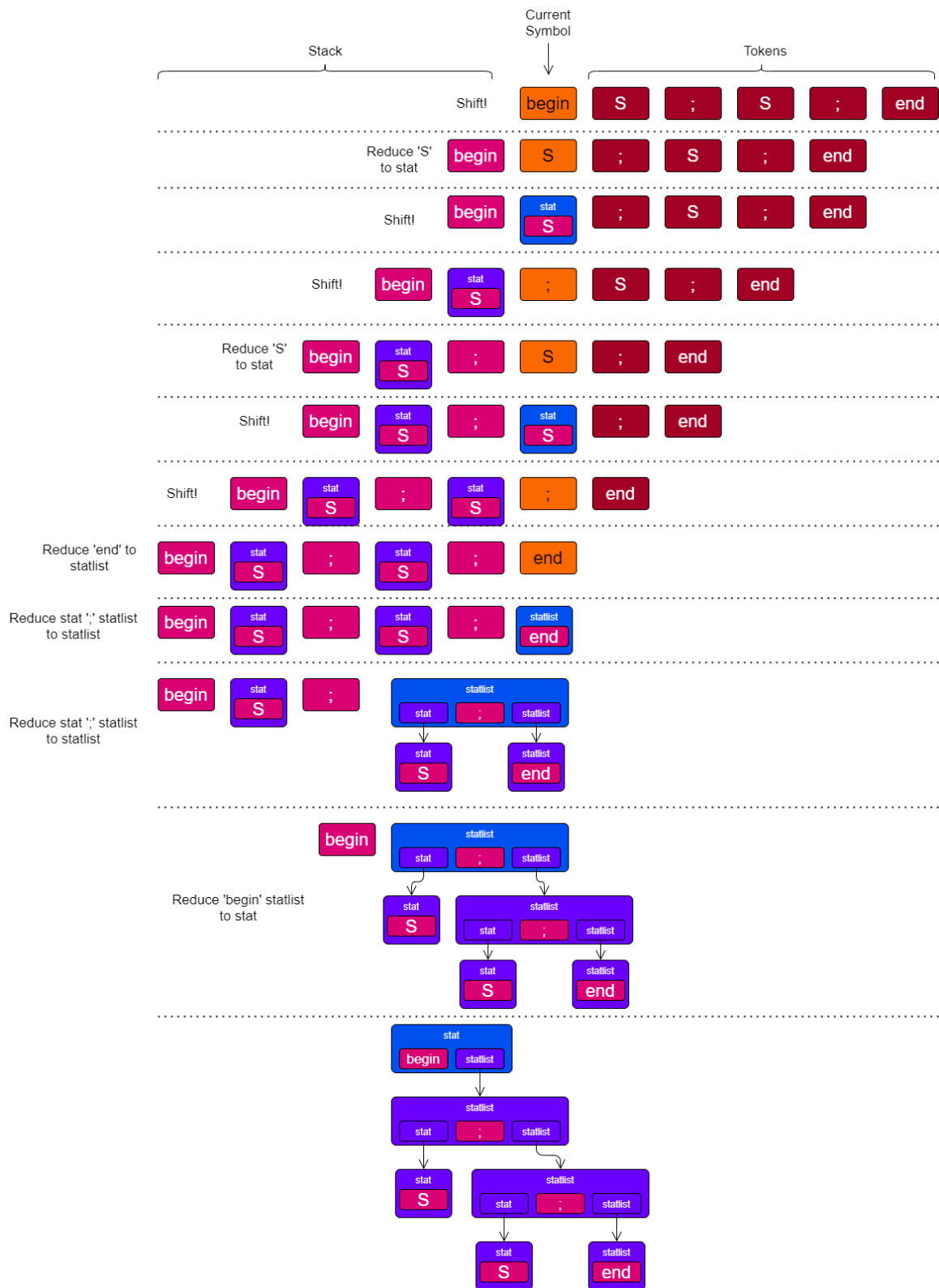
$$\begin{aligned} stat &\rightarrow \text{'loop' } statlist \text{ loopstat} \\ loopstat &\rightarrow \text{'until' } expr \\ loopstat &\rightarrow \text{'while' } expr \\ loopstat &\rightarrow \text{'forever' } \end{aligned}$$

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

Bottom-up Parsing

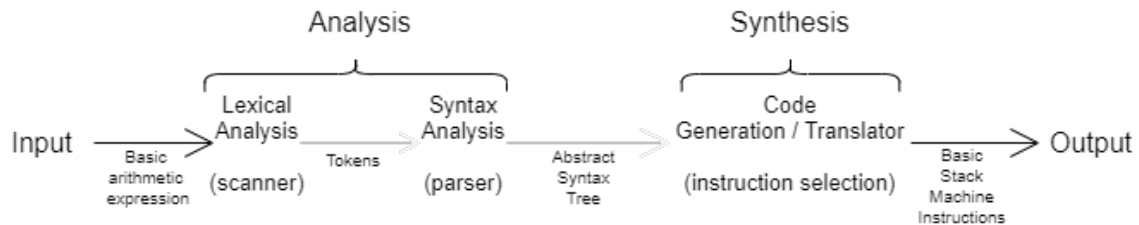
- The grammar's productions are used *right* \rightarrow *left*.
- Input is compared against the right hand side to produce a non-terminal on the left.
- Parsing is complete when the whole input is replaced by the start symbol.

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
22 Grammar:
23 add    -> mul + add | mul - add | mul
24 mul    -> factor * mul | factor / mul | factor
25 factor -> number | identifier
26
27 + - (left associative, low precedence)
28 * / (left associative, high precedence)
29
30 Eg:
31 > tokenise "a+b*17"
32 [IDENT a,PLUS,IDENT b,MUL,NUM 17]
33
34 > parser (tokenise "a+b-17")
35 Plus (Ident a, Minus (Ident b, Num 17))
36
37 > parser (tokenise "3-3-3*77*a-4")
38 Minus (Num 3, Minus (Num 3, Minus (Mul (Num 3, Mul (Num 77, Ident a)), Num 4)))
39
40 > compile "a+b+c*7-3"
41 [PushVar a,PushVar b,PushVar c,PushConst 7,MulToS,PushConst 3,SubToS,AddToS,AddToS]
42
43 > translate (parser (tokenise "a+b/17"))
44 [PushVar a,PushVar b,PushConst 17,DivToS,AddToS]
```

```

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

```

```

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

```

```

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

```

```

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

```