50006 - Compilers - Lecture 2

Oliver Killane

05/01/22

Lecture Recording

Lecture recording is available here

Syntax Analysis

- **Syntax** The grammatrical 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.

$$\textit{stat} \rightarrow \text{'if'} \ \ \text{'('} \ \textit{expr} \ \text{')'} \ \textit{stat} \ \text{'else'} \ \textit{stat}$$

- **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.

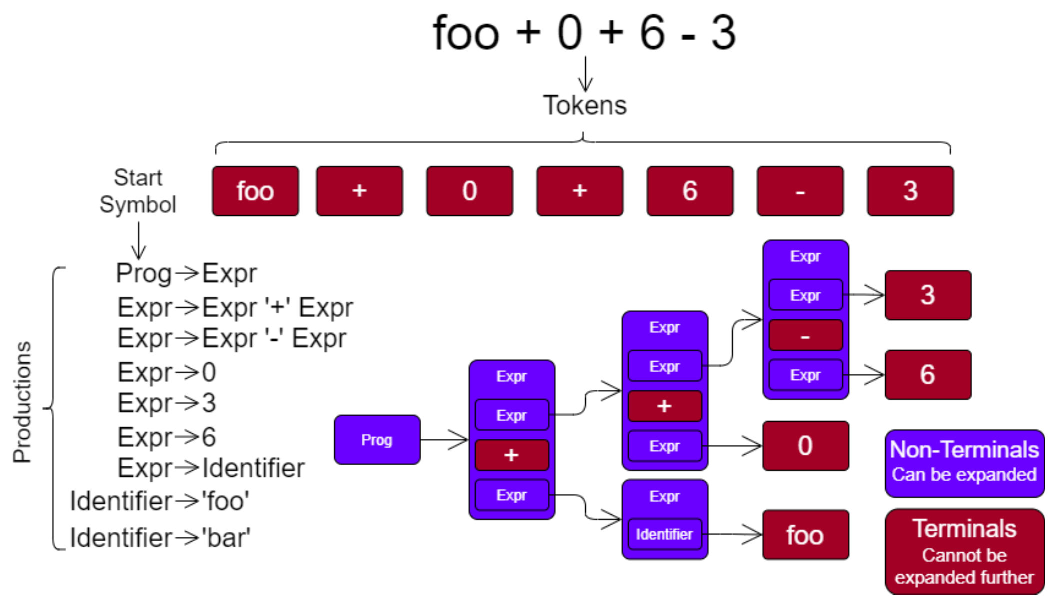
- **Production**
  Shows one valid way to expand a non-terminal symbol into a string of terminals & non-terminals.

  $\textit{expr} \rightarrow$ '0'
  $\textit{expr} \rightarrow$ '1'
  $\textit{expr} \rightarrow$ $\textit{expr} + \textit{expr}$
  $\textit{expr} \rightarrow$ '0' | '1' | $\textit{expr} + \textit{expr}$ 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.

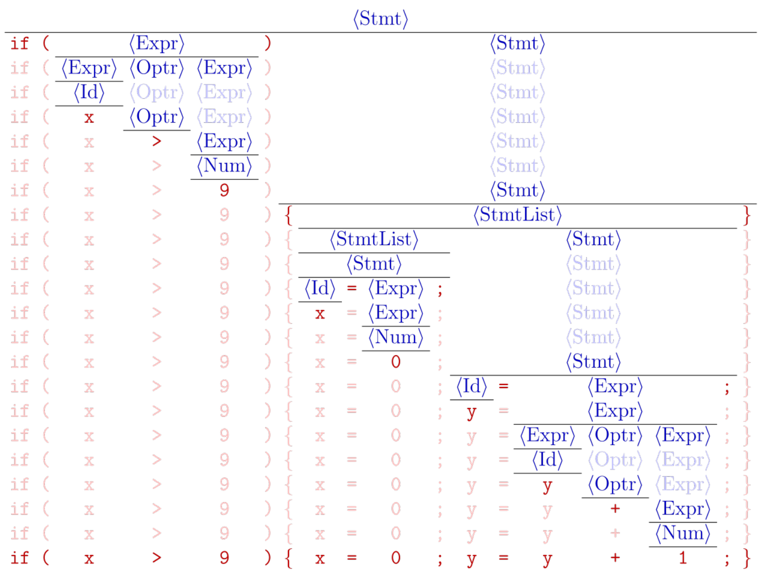


We can express the grammar as a tuple:

$G = (S, P, t, nt)$ 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.

⟨Stmt⟩	→	⟨Id⟩ = ⟨Expr⟩ ;
⟨Stmt⟩	→	{ ⟨StmtList⟩ }
⟨Stmt⟩	→	`if` (⟨Expr⟩) ⟨Stmt⟩
⟨StmtList⟩	→	⟨Stmt⟩
⟨StmtList⟩	→	⟨StmtList⟩ ⟨Stmt⟩
⟨Expr⟩	→	⟨Id⟩
⟨Expr⟩	→	⟨Num⟩
⟨Expr⟩	→	⟨Expr⟩ ⟨Optr⟩ ⟨Expr⟩
⟨Id⟩	→	x
⟨Id⟩	→	y
⟨Num⟩	→	0
⟨Num⟩	→	1
⟨Num⟩	→	9
⟨Optr⟩	→	>
⟨Optr⟩	→	+



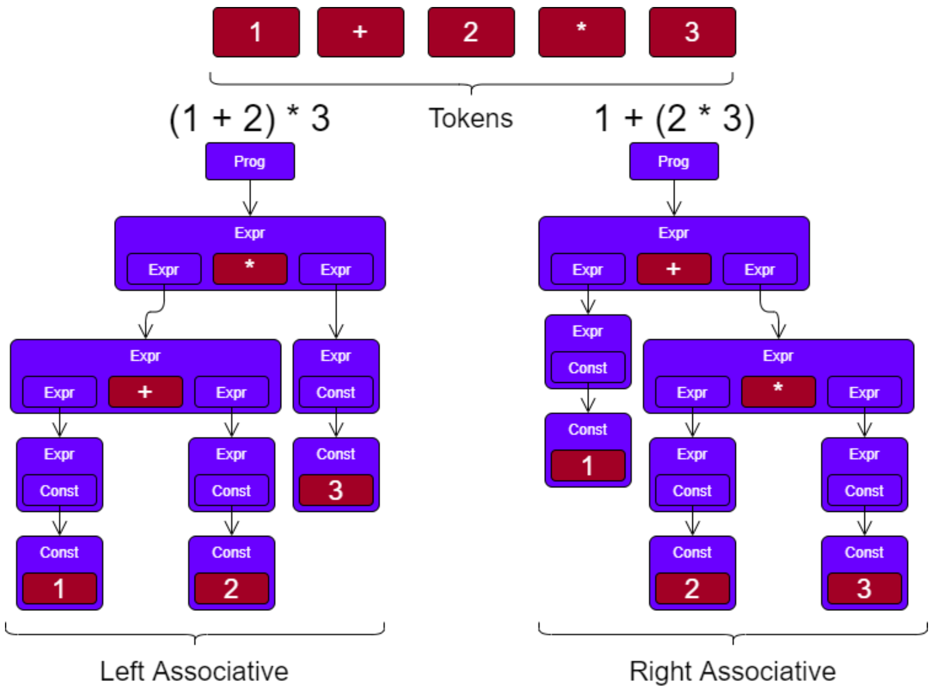
An example with a basic C-style if statement (sourced from wikipedia)

- 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 dervbied 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.

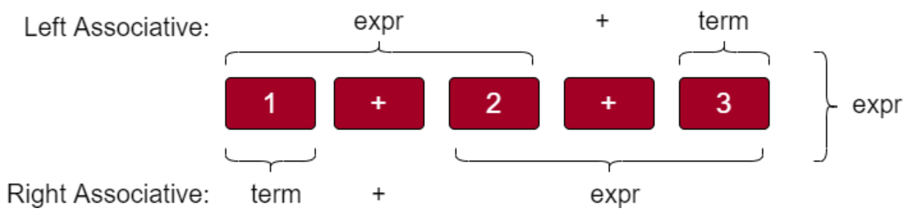
Precedence and Associativity

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 $+$.

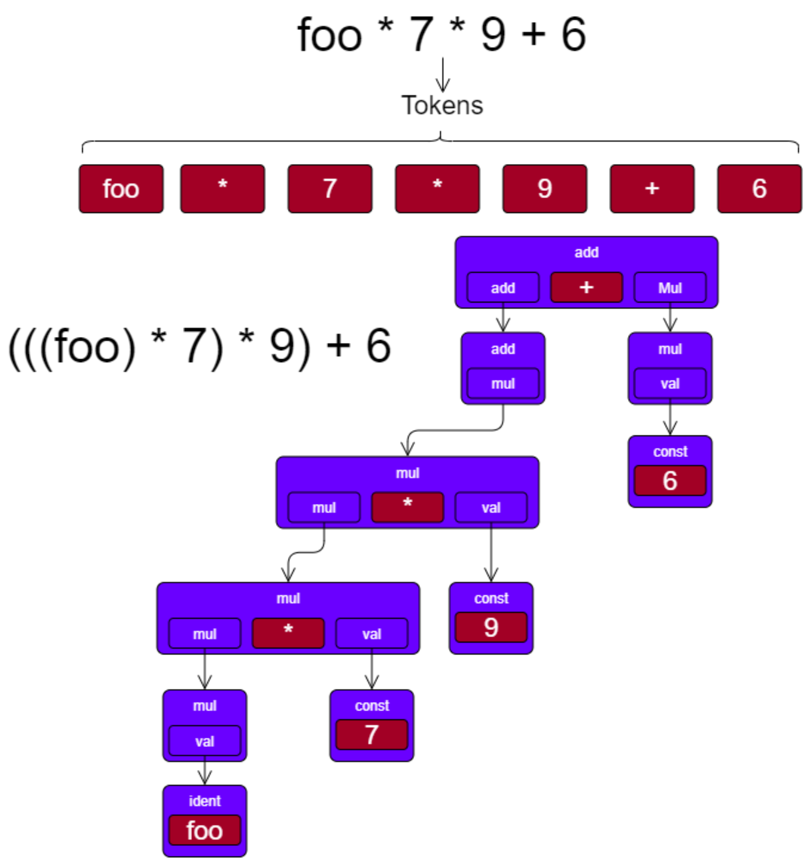


Grammar for Precedence

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

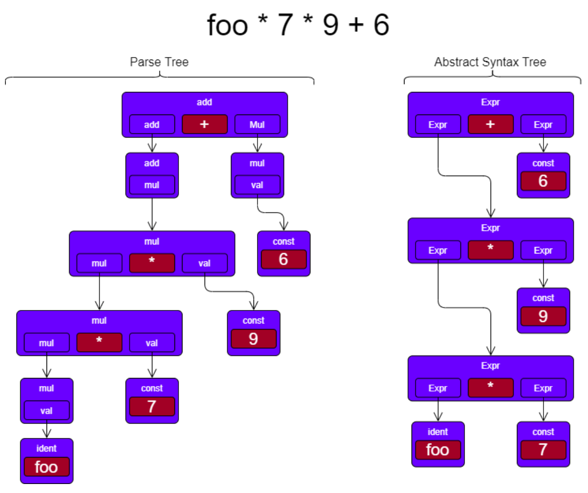
$$\begin{aligned} add &\rightarrow add + mul \mid add - mul \mid mul \\ mul &\rightarrow mul * val \mid mul/val \mid val \\ val &\rightarrow const \mid ident \end{aligned}$$

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.



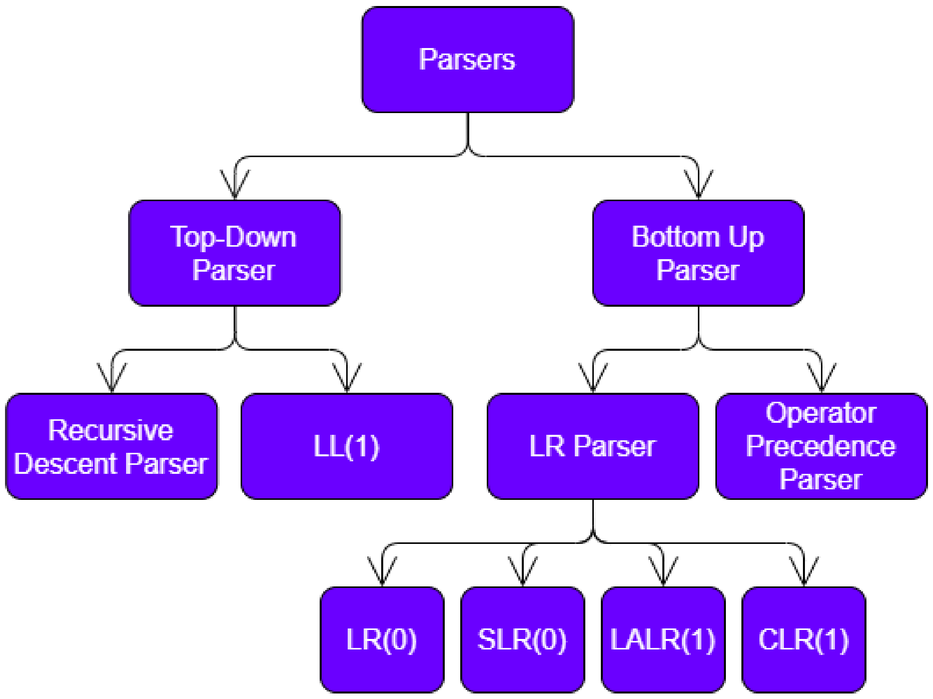
Parse Tree vs Abstract Syntax Tree

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

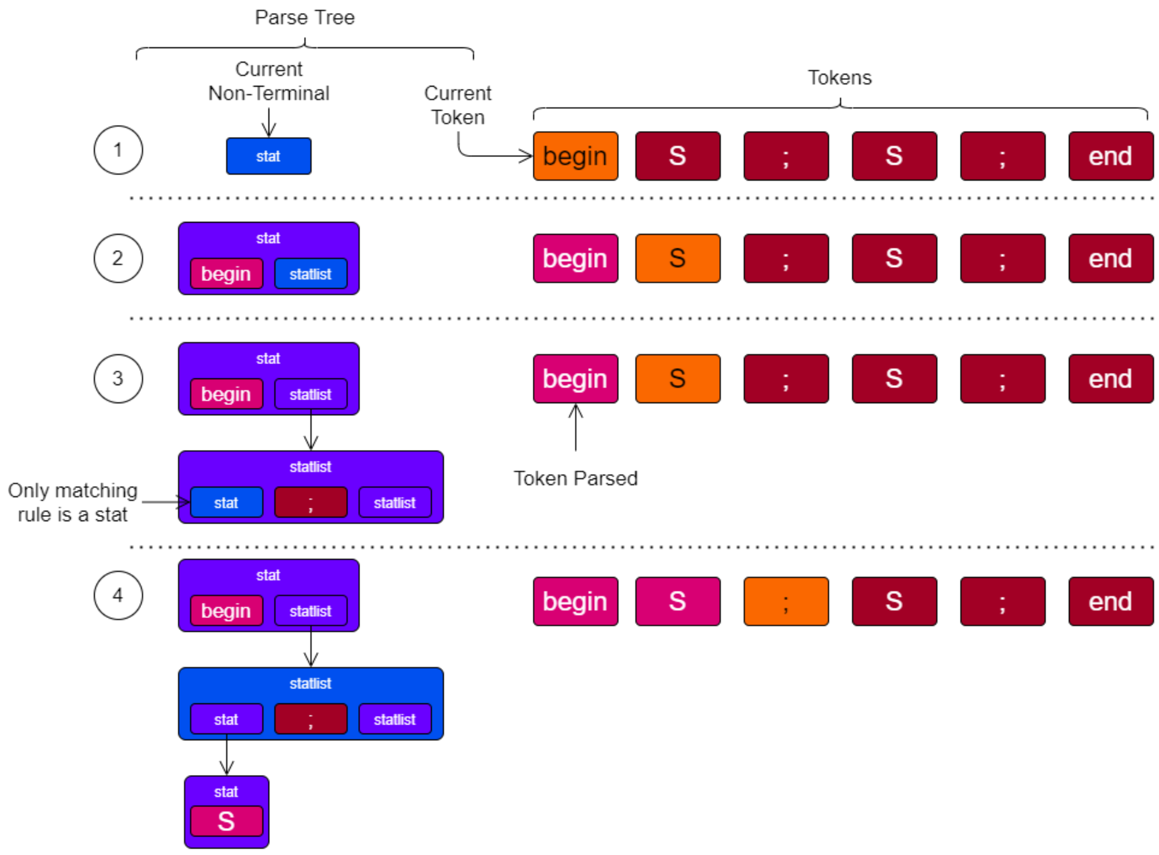
  By using the production rules & the current token we can predict the next production rule, and use this to either:

  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 occured/the program is syntactically invalid)
- We are using the grammar *left* $\rightarrow$ *right*.

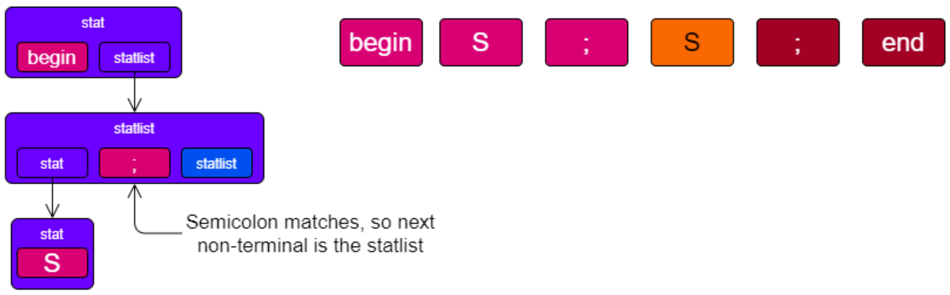
For example with the grammar:

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

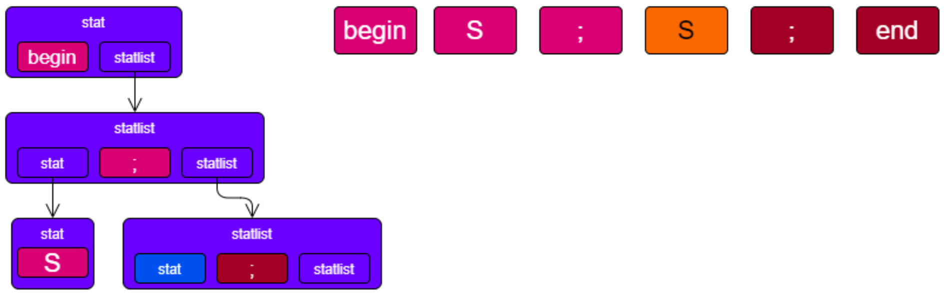
Start symbol is *stat*.



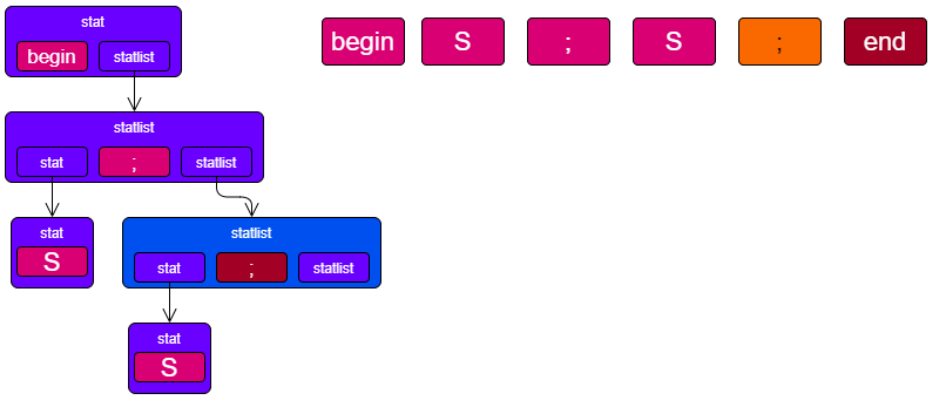
5


stat
begin
statlist
statlist
stat
;
statlist
stat
S
Semicolon matches, so next non-terminal is the statlist
begin
S
;
S
;
end

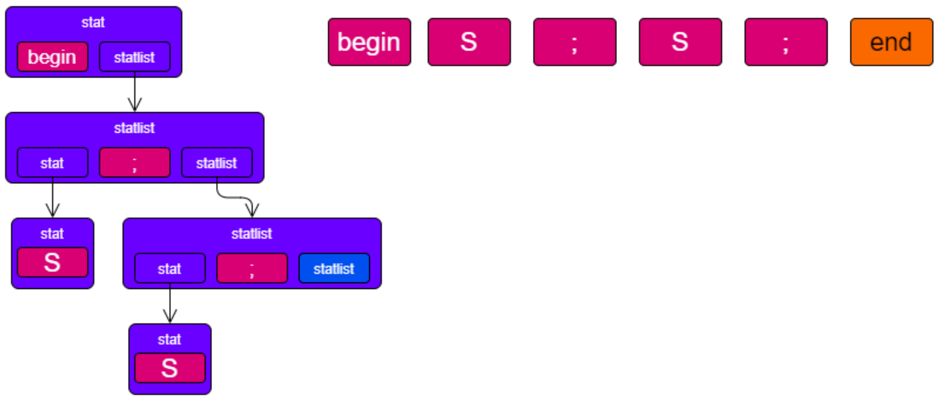
6


stat
begin
statlist
statlist
stat
;
statlist
stat
S
statlist
stat
;
statlist
begin
S
;
S
;
end

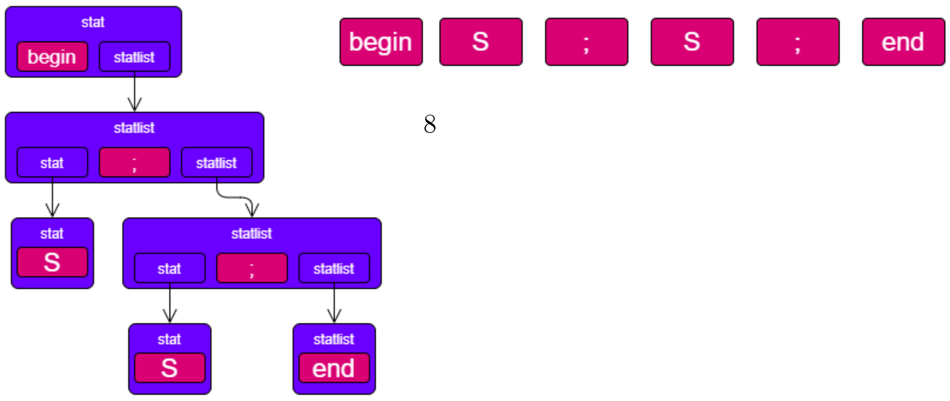
7


stat
begin
statlist
statlist
stat
;
statlist
stat
S
statlist
stat
;
statlist
stat
S
begin
S
;
S
;
end

8


stat
begin
statlist
statlist
stat
;
statlist
stat
S
statlist
stat
;
statlist
stat
S
begin
S
;
S
;
end

9


stat
begin
statlist
statlist
stat
;
statlist
stat
S
statlist
stat
;
statlist
stat
S
statlist
end
begin
S
;
S
;
end

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.

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

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.

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

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:

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

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.

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

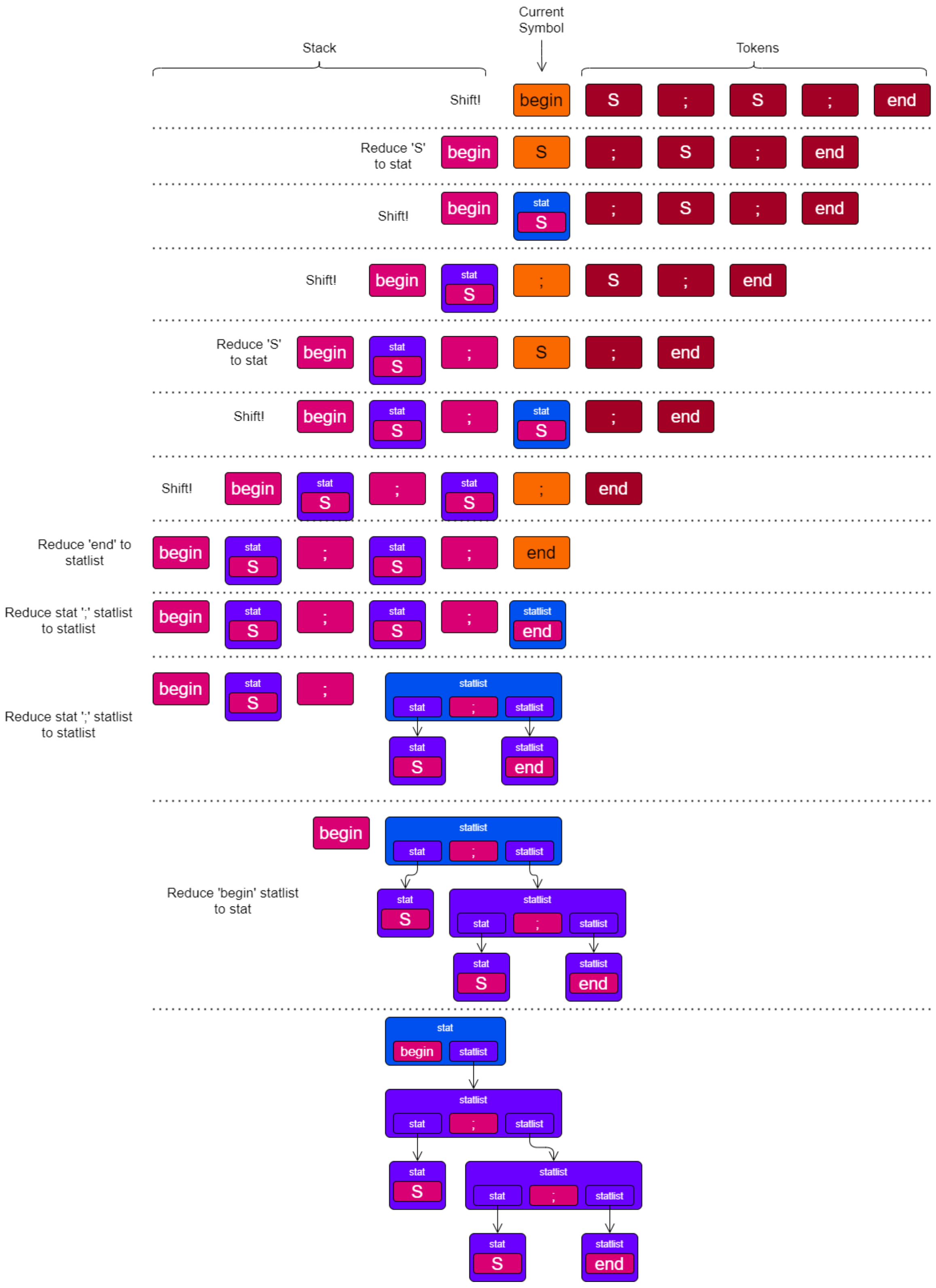
If attempting to parse this production:

```python
def parse_add(input):
    (add_parse_tree, rest) = parse_add(input)
    rest = parse_plus(rest) # infinite recursion
    (mul_parse_tree, rest) = parse_mul(rest)
    return (tree(add_parse_tree, mul_parse_tree), rest)
```

Bottom-up Parsing

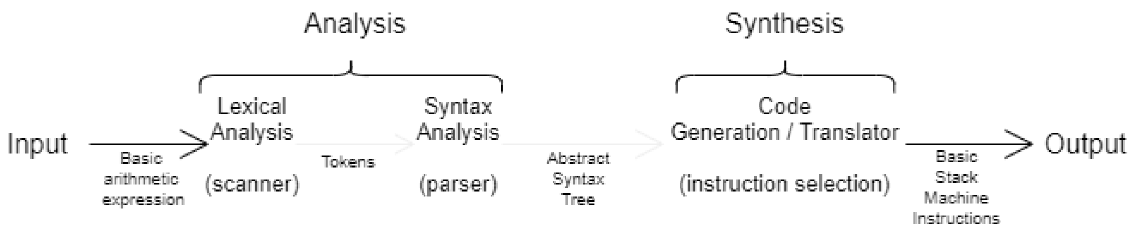
- The grammar's productions are used *right* $\rightarrow$ *left*.
- Input is compared against the right hand side to produce a non-terminal on the left.
- Parsing is complete when the whole input is replaced by the start symbol.

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


Current Symbol
Stack
Tokens
Shift!
begin
S
;
S
;
end
Reduce 'S' to stat
Shift!
stat
Reduce 'end' to statlist
statlist
Reduce stat ';' statlist to statlist
Reduce 'begin' statlist to stat

Simple Complete Compiler

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

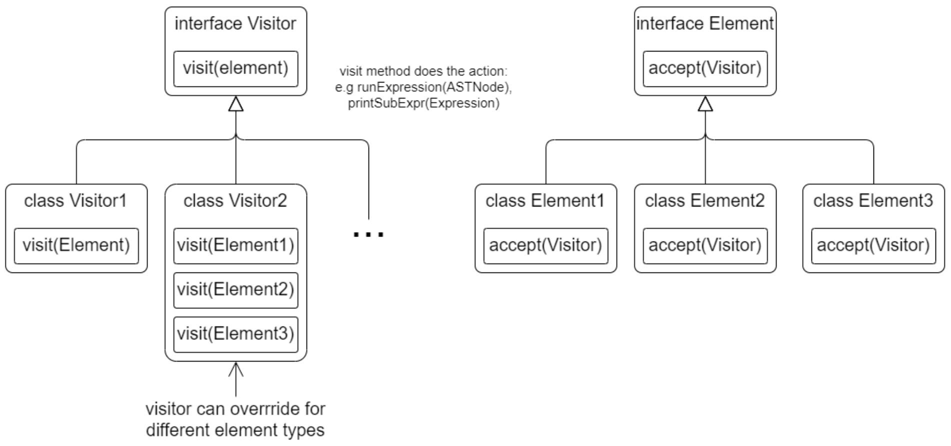


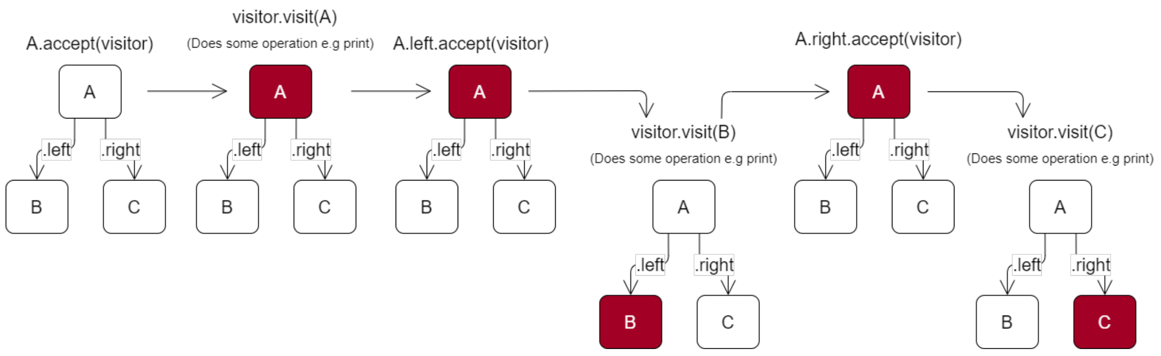
```
{-
Simple compiler example:
Arithmetic Expressions -> Stack Machine Instructions

Original Description:
"Compiling arithmetic expressions into code for a stack machine.  This is not a
solution to Exercise 2 - it's an executable version of the code generator for
expressions, which is given in the notes.  Build on it to yield a code generator
for statements.

Paul Kelly, Imperial College, 2003

Tested with Hugs (Haskell 98 mode), Feb 2001 version"

Changes:
- This version has been updated to work with Haskell version 8.6.5
- Use of where over let & general refactoring
- Fixed bug with execute (missing patterns for invalid stack instructions)
- New grammar to support multiplication and division

Grammar:
 add    -> mul + add | mul - add | mul
 mul    -> factor * mul | factor / mul | factor
 factor -> number | identifier

+ - (right associative, low precedence)
* / (right associative, high precedence)

Eg:
> tokenise "a+b*17"
[IDENT a,PLUS,IDENT b,MUL,NUM 17]

> parser (tokenise "a+b-17")
Plus (Ident a, Minus (Ident b, Num 17))

> parser (tokenise "3-3-3*77*a-4")
Minus (Num 3, Minus (Num 3, Minus (Mul (Num 3, Mul (Num 77, Ident a)), Num 4)))

> compile "a+b+c*7-3"
[PushVar a,PushVar b,PushVar c,PushConst 7,MulToS,PushConst 3,SubToS,AddToS,AddToS]

> translate (parser (tokenise "a+b/17"))
[PushVar a,PushVar b,PushConst 17,DivToS,AddToS]

```

```haskell
> putStr (runAnimated [("a", 9)] [] (translate (parser (tokenise "100+a*3-17"))))
[100]
[9,100]
[3,9,100]
[27,100]
[17,27,100]
[10,100]
[110]
[110]

-}

import Data.Char ( isDigit, isAlpha, digitToInt )
import Text.Parsec (tokens, Stream (uncons))

-- Token data type
data Token
  = IDENT [Char] | NUM Int | PLUS | MINUS | MUL | DIV

-- Ast (abstract syntax tree) data type
data Ast
  = Ident [Char] | Num Int | Plus Ast Ast | Minus Ast Ast | Mul Ast Ast | Div Ast Ast

-- Instruction data type
--
-- PushConst pushes a given number onto the stack; AddToS takes the top
-- two numbers from the top of the stack (ToS), and them and pushes the sum.
-- (We have to invent new names to avoid clashing with MUL, Mul etc above)
data Instruction
  = PushConst Int | PushVar [Char] | AddToS | SubToS | MulToS | DivToS

instance Show Token where
  showsPrec p (IDENT name) = showString "IDENT " . showString name
  showsPrec p (NUM num) = showString "NUM " . shows num
  showsPrec p (PLUS) = showString "PLUS"
  showsPrec p (MINUS) = showString "MINUS"
  showsPrec p (MUL) = showString "MUL"
  showsPrec p (DIV) = showString "DIV"

instance Show Ast where
  showsPrec p (Ident name) = showString "Ident " . showString name
  showsPrec p (Num num) = showString "Num " . shows num
  showsPrec p (Plus e1 e2) = showString "Plus (" . shows e1 . showString ", " . shows
      ↪ e2 . showString ")"
  showsPrec p (Minus e1 e2) = showString "Minus (" . shows e1 . showString ", " .
      ↪ shows e2 . showString ")"
  showsPrec p (Mul e1 e2) = showString "Mul (" . shows e1 . showString ", " . shows
      ↪ e2 . showString ")"
  showsPrec p (Div e1 e2) = showString "Div (" . shows e1 . showString ", " . shows
      ↪ e2 . showString ")"

instance Show Instruction where
  showsPrec p (PushConst n) = showString "PushConst " . shows n
  showsPrec p (PushVar name) = showString "PushVar " . showString name
  showsPrec p AddToS = showString "AddToS"
  showsPrec p SubToS = showString "SubToS"
  showsPrec p MulToS = showString "MulToS"
  showsPrec p DivToS = showString "DivToS"

-- Parse the tokens (top-down) by parsing each expression to get a new parse
-- tree and the rest of the tokens. No tokens should remain after parsing.
parser :: [Token] -> Ast
parser tokens
  | null rest = tree
  | otherwise = error "(parser) excess rubbish"
  where
    (tree, rest) = parseAdd tokens

parseAdd :: [Token] -> (Ast, [Token])
parseAdd tokens
  = case rest of
      (PLUS : rest2) -> let (subexptree, rest3) = parseAdd rest2 in (Plus multree
          ↪ subexptree, rest3)
      (MINUS : rest2) -> let (subexptree, rest3) = parseAdd rest2 in (Minus multree
          ↪ subexptree, rest3)
      othertokens -> (multree, othertokens)
  where
    (multree, rest) = parseMul tokens

parseMul :: [Token] -> (Ast, [Token])
parseMul tokens
  = case rest of
      (MUL : rest2) -> let (subexptree, rest3) = parseMul rest2 in (Mul factortree
          ↪ subexptree, rest3)
      (DIV : rest2) -> let (subexptree, rest3) = parseMul rest2 in (Div factortree
          ↪ subexptree, rest3)
      othertokens -> (factortree, othertokens)
  where
    (factortree, rest) = parseFactor tokens

parseFactor :: [Token] -> (Ast, [Token])
parseFactor ((NUM n):restoftokens) = (Num n, restoftokens)
parseFactor ((IDENT x):restoftokens) = (Ident x, restoftokens)
parseFactor [] = error "(parseFactor) Attempted to parse empty list"
parseFactor (t:_) = error $ "(parseFactor) error parsing token " ++ show t


-- Lexical analysis - tokenisation
tokenise :: [Char] -> [Token]
tokenise [] = []                    -- (end of input)
tokenise (' ':rest) = tokenise rest      -- (skip spaces)
tokenise ('+':rest) = PLUS : (tokenise rest)
tokenise ('-':rest) = MINUS : (tokenise rest)
tokenise ('*':rest) = MUL : (tokenise rest)
tokenise ('/':rest) = DIV : (tokenise rest)
tokenise (ch:rest)
  | isDigit ch  = (NUM dn):(tokenise drest2)
  | isAlpha ch  = (IDENT an):(tokenise arest2)
  where
    (dn, drest2) = convert (ch:rest)
    (an, arest2) = getname (ch:rest)
tokenise (c:_) = error $ "(tokenise) unexpected character " ++ [c]

getname :: [Char] -> ([Char], [Char]) -- (name, rest)
getname = flip getname' []
  where
      getname' :: [Char] -> [Char] -> ([Char], [Char])
      getname' [] chs = (chs, [])
      getname' (ch : str) chs
        | isAlpha ch = getname' str (chs++[ch])
        | otherwise  = (chs, ch : str)

convert :: [Char] -> (Int, [Char])
convert = flip conv' 0
  where
    conv' [] n = (n, [])
    conv' (ch : str) n
      | isDigit ch = conv' str ((n*10) + digitToInt ch)
      | otherwise  = (n, ch : str)

-- Translate - the code generator
translate :: Ast -> [Instruction]
translate (Num n) = [PushConst n]
translate (Ident x) = [PushVar x]
translate (Plus e1 e2) = translate e1 ++ translate e2 ++ [AddToS]
translate (Minus e1 e2) = translate e1 ++ translate e2 ++ [SubToS]
translate (Mul e1 e2) = translate e1 ++ translate e2 ++ [MulToS]
translate (Div e1 e2) = translate e1 ++ translate e2 ++[DivToS]

compile :: [Char] -> [Instruction]
compile = translate . parser . tokenise

-- Execute, run - simulate the machine running the stack instructions
--
-- Note that this simple machine is too simple to be realistic;
-- (1) 'execute' doesn't return the store, so no instruction can change it
-- (2) 'run' forgets each instruction as it is executed, so can't do loops

-- The state of the machine consists of a store (a set of associations
-- between variable and their values), together with a stack:

type Stack = [Int]
type Store = [([Char], Int)]

-- 'run' executes a sequence of instructions using a specified
-- store, and starting from a given stack
--
run :: Store -> Stack -> [Instruction] -> Stack

run store stack [] = stack
run store stack (i : is) = run store (execute store stack i) is

-- 'execute' applies a given instruction to the current state of the
-- machine - ie the store and the stack
--
execute :: Store -> Stack -> Instruction -> Stack
execute store (a : b: rest) AddToS        = ( (b+a) : rest )
execute store (a : b: rest) SubToS        = ( (b-a) : rest )
execute store (a : b: rest) MulToS        = ( (b*a) : rest )
execute store (a : b: rest) DivToS        = ( (b `div` a) : rest )
execute store rest          (PushConst n) = ( n : rest )
execute store rest          (PushVar x)   = ( n : rest )
  where n = valueOf x store
execute store [x] instr = error $ "(execute) attempted to run " ++ show instr ++ " with only" ++ show x ++ " on the stack"
execute store []  instr = error $ "(execute) attempted to run " ++ show instr ++ " with an empty stack"

valueOf x [] = error ("no value for variable "++show x)
valueOf x ( (y,n) : rest ) = if x==y then n else valueOf x rest
```

```

-- runAnimated does what run does but shows the stack after each step:
--
runAnimated :: Store -> Stack -> [Instruction] -> [Char]
runAnimated store stack [] = show stack
runAnimated store stack (i : is) = show newstack ++ "\n" ++ runAnimated store newstack is
  where
    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).