Compilers - Chapter 2: An introduction to syntax analysis (and a complete toy compiler)

- Lecturers:
  - Paul Kelly
  - Jamie Willis
  - Hongxiang Fan

Acknowledgements: Naranker Dulay and Chris Hankin, who have both contributed to the course design

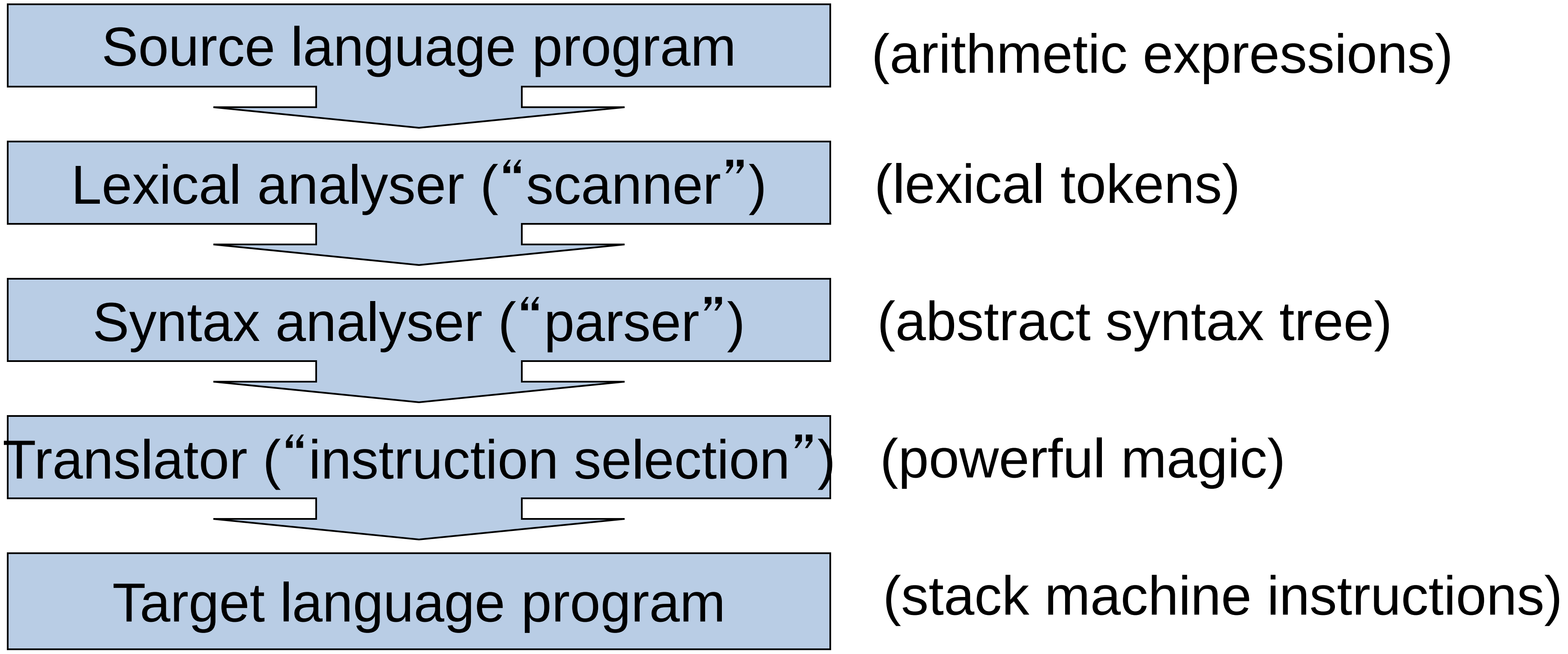
Plan:

- Complete compiler a few slides
- Using Haskell to keep code short
- Tutorial exercise shows how this translates to Java
- Contents:
  - Introduction to grammars
  - Parsing (also known as syntax analysis)
  - The abstract syntax tree
  - A simple instruction set (stack machine)
  - A code generator
  - A lexical analyser
- Compare: Engineering a Compiler Chapter 1, Dragon book, Chapter 2.

- Try it: working Haskell code can be found at: http://www.doc.ic.ac.uk/~phjk/CompilersCourse/SampleCode/Ex2-CodeGenInHaskell/SimpleCompilerV2.hs.

A complete example compiler

- Translation of a simple arithmetic expression language to stack machine code:



Haskell

- For clarity and conciseness, **Haskell** will be used to specify the data types and functions which make up the compiler:

```
compile :: [char] -> [instruction]
compile program
= translate(parse(scan program))
```

Walk the tree and generate instructions

“syntactic analysis” – Find the tree of nested eg if, for, while, statements

“lexical analysis” – Keywords, punctuation, identifiers

Syntax analysis (= parsing)

- The specification of a programming language consists of two parts:
  - **SYNTAX** —grammatical structure
  - **SEMANTICS** —meaning
- **SYNTAX** consists of rules for constructing "acceptable" utterances. To determine that a program is syntactically correct, you must determine how the grammatical rules were used to construct it.
  - Powerful tools ("parser generators") are available for generating analysis phases of your compiler, starting from a formal specification of the language's syntax. You should learn to use them.
- **SEMANTICS** is much harder to specify.
  - Much research has gone into "compiler generators", "compiler compilers" – that generate a compiler's synthesis phases from a semantic specification of the source/target language. There are promising tools but most people write the code manually.

Specifying syntactic rules

- Syntax is usually specified using a *context-free grammar*, often called *Backus-Naur Form* (BNF), or even *Backus Normal Form*.
- A sample BNF production:
  stat → 'if' '(' exp ')' stat 'else' stat
- Each production shows one valid way by which a non-terminal (LHS) can be expanded (RHS) into a string of terminals and non-terminals
- **Terminals:** 'if' , '(' , ')' , 'else'
- **Non-terminals:** stat, exp
- Only terminals appear in the final result (terminals are, in fact, just lexical tokens).

Using syntactic rules

- A sample BNF production:

  stat $\rightarrow$ 'if' '(' **exp** ')' **stat** 'else' **stat**

- Suppose we were confronted with:

  if (stuff$_1$) stuff$_2$ else stuff$_3$

- (where stuff$_1$, stuff$_2$ and stuff$_3$ are strings of terminals which we have not yet recognised)

- This ***looks*** like it was constructed using the production above. To prove that it ***is*** grammatically correct, we must show that stuff$_1$ can be derived from **exp**, and that stuff$_2$ and stuff$_3$ can each be derived (perhaps in different ways), from the non-terminal **stat**

More formally:

- A context-free grammar (CFG) consists of four components:
  - *S*: a non-terminal start symbol
  - *P*: a set of productions
  - *t*: a set of tokens (‘terminals’)
  - *nt*: a set of non-terminals
- Example: productions, $P = \left\{ \begin{array}{l} \textbf{bin} \rightarrow \textbf{bin} \text{ ‘+’ } \textbf{dig} \\ \textbf{bin} \rightarrow \textbf{bin} \text{ ‘–’ } \textbf{dig} \\ \textbf{bin} \rightarrow \textbf{dig} \\ \textbf{dig} \rightarrow \text{‘0’} \\ \textbf{dig} \rightarrow \text{‘1’} \end{array} \right\}$

Productions with the same LHS can have their RHS’s combined using ‘|’. In this example:

$$\begin{array}{l} \textbf{bin} \rightarrow \textbf{bin} \text{ ‘+’ } \textbf{dig} \mid \textbf{bin} \text{ ‘-’ } \textbf{dig} \mid \textbf{dig} \\ \textbf{dig} \rightarrow \text{‘0’} \mid \text{‘1’} \end{array}$$

- Terminals:
  - t = { ‘+’ , ‘-’ , ‘0’ , ‘1’ }
- Non-terminals:
  - nt = {bin, dig}
- We choose bin as the start symbol *S*
  - Strings of terminals can be **derived** using the grammar by beginning with the start symbol, and repeatedly replacing each non-terminal with the RHS from some corresponding production.
  - A string so derived which consists only of terminals is called a **sentence**.
  - The **language** of a grammar is the set of all sentences which can be derived from its start symbol.

Context-free grammar - example

- Question: what is the language *L*(G) of our example grammar G = (S,P,t,nt) ?

S = **bin**

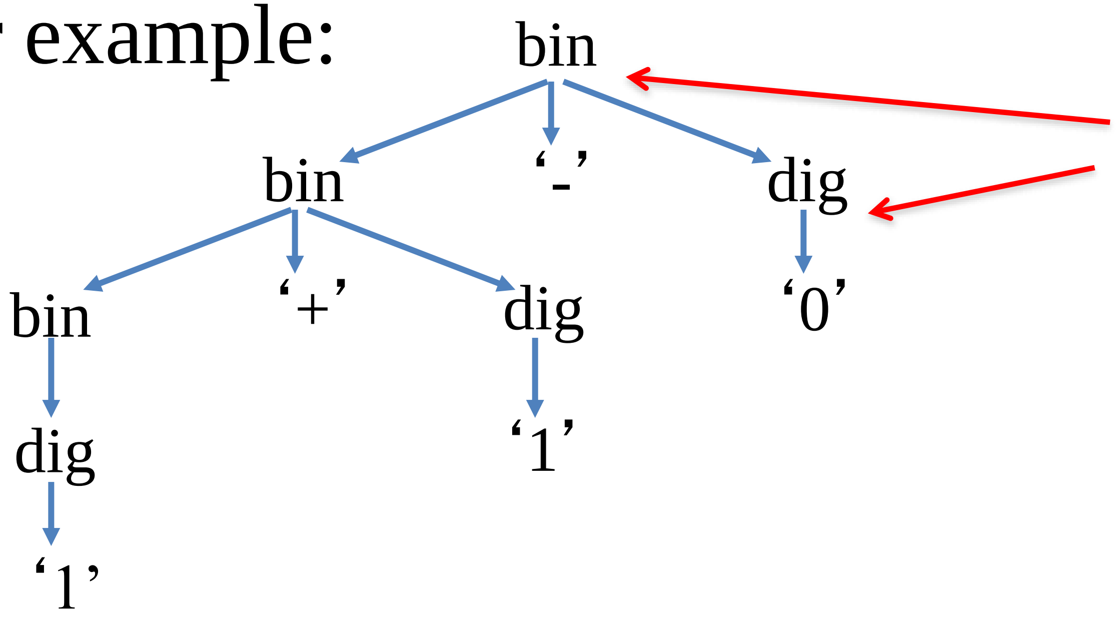
P =
$$\left\{\begin{array}{l}
\textbf{bin} \rightarrow \textbf{bin} \text{ '+' } \textbf{dig} \\
\textbf{bin} \rightarrow \textbf{bin} \text{ '–' } \textbf{dig} \\
\textbf{bin} \rightarrow \textbf{dig} \\
\textbf{dig} \rightarrow \text{'0'} \\
\textbf{dig} \rightarrow \text{'1'}
\end{array}\right.$$

t = { '+' , '-' , '0' , '1' }

nt = {bin, dig}

The Parse Tree

- The **parse tree** shows pictorially how the string is derived from the start symbol.
- For example:



Each "branch" corresponds precisely to a production in the grammar

```
bin → bin '+' dig
bin → bin '-' dig
bin → dig
dig → '0'
dig → '1'
```

- The parse tree is a graphical proof showing the steps in the derivation of the string.
- **Parsing** is the process of finding this proof.

Parse trees as proofs

- L(G) contains “1”
  - Proof:

$$\begin{tikzcd}
\text{bin} \arrow[d] \\
\text{dig} \arrow[d] \\
\text{‘1’}
\end{tikzcd}$$

- L(G) contains “1+1”
  - Proof:

$$\begin{tikzcd}
& \text{bin} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{bin} \arrow[d] & \text{‘+’} & \text{dig} \arrow[d] \\
\text{dig} \arrow[d] & & \text{‘1’} \\
\text{‘1’} & &
\end{tikzcd}$$

- L(G) contains “1+1-0”
  - Proof:

$$\begin{tikzcd}
& & \text{bin} \arrow[dl] \arrow[d] \arrow[dr] & \\
& \text{bin} \arrow[dl] \arrow[d] \arrow[dr] & \text{‘-’} & \text{dig} \arrow[d] \\
\text{bin} \arrow[d] & \text{‘+’} & \text{dig} \arrow[d] & \text{‘0’} \\
\text{dig} \arrow[d] & & \text{‘1’} & \\
\text{‘1’} & & &
\end{tikzcd}$$

- L(G) contains “1+1-0+1-0”
  - Proof:

$$\begin{tikzcd}
& & & & \text{bin} \arrow[dl] \arrow[d] \arrow[dr] & \\
& & & \text{bin} \arrow[dl] \arrow[d] \arrow[dr] & \text{‘-’} & \text{dig} \arrow[d] \\
& & \text{bin} \arrow[dl] \arrow[d] \arrow[dr] & \text{‘+’} & \text{dig} \arrow[d] & \text{‘0’} \\
& \text{bin} \arrow[dl] \arrow[d] \arrow[dr] & \text{‘-’} & \text{dig} \arrow[d] & \text{‘1’} & \\
\text{bin} \arrow[d] & \text{‘+’} & \text{dig} \arrow[d] & \text{‘0’} & & \\
\text{dig} \arrow[d] & & \text{‘1’} & & & \\
\text{‘1’} & & & & &
\end{tikzcd}$$

Ambiguity

- A grammar is **ambiguous** if the language it generates contains strings which can be generated in *two different ways* – that is, there exists a string with two *different parse trees*
- Example:

```
exp → exp ‘+’ exp |
      exp ‘–’ exp | const | ident
```

- Consider the string “**9 + a – 2**”.

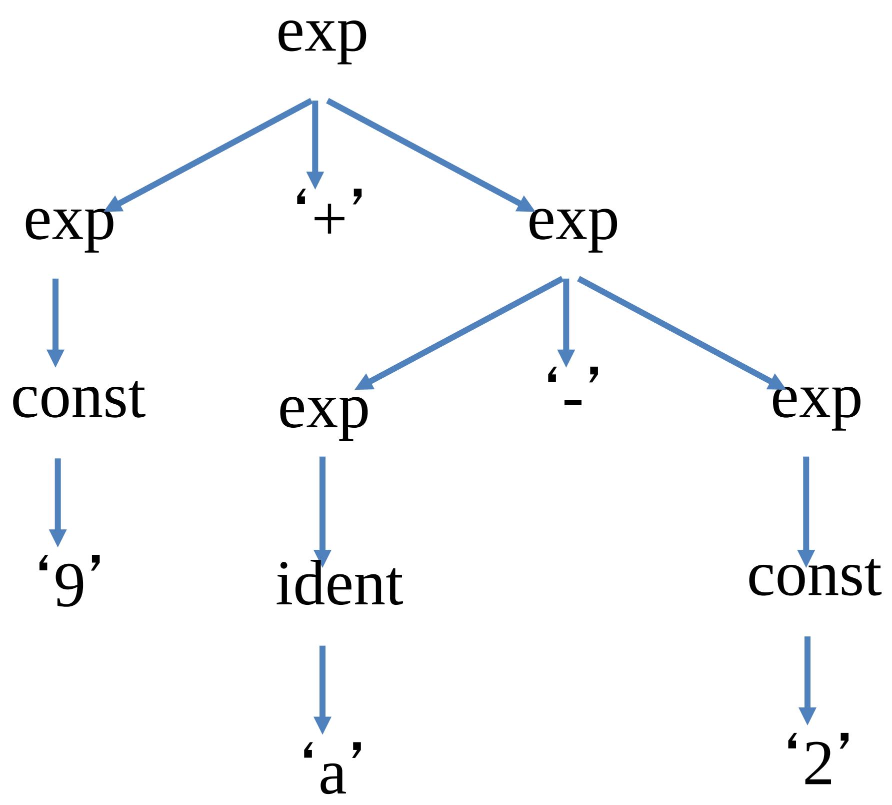
Ambiguity…

- The string “`9 + a – 2`” has two different parse trees according to our example grammar….

```
exp → exp ‘+’ exp |
      exp ‘–’ exp | const | ident
```

Ambiguity…

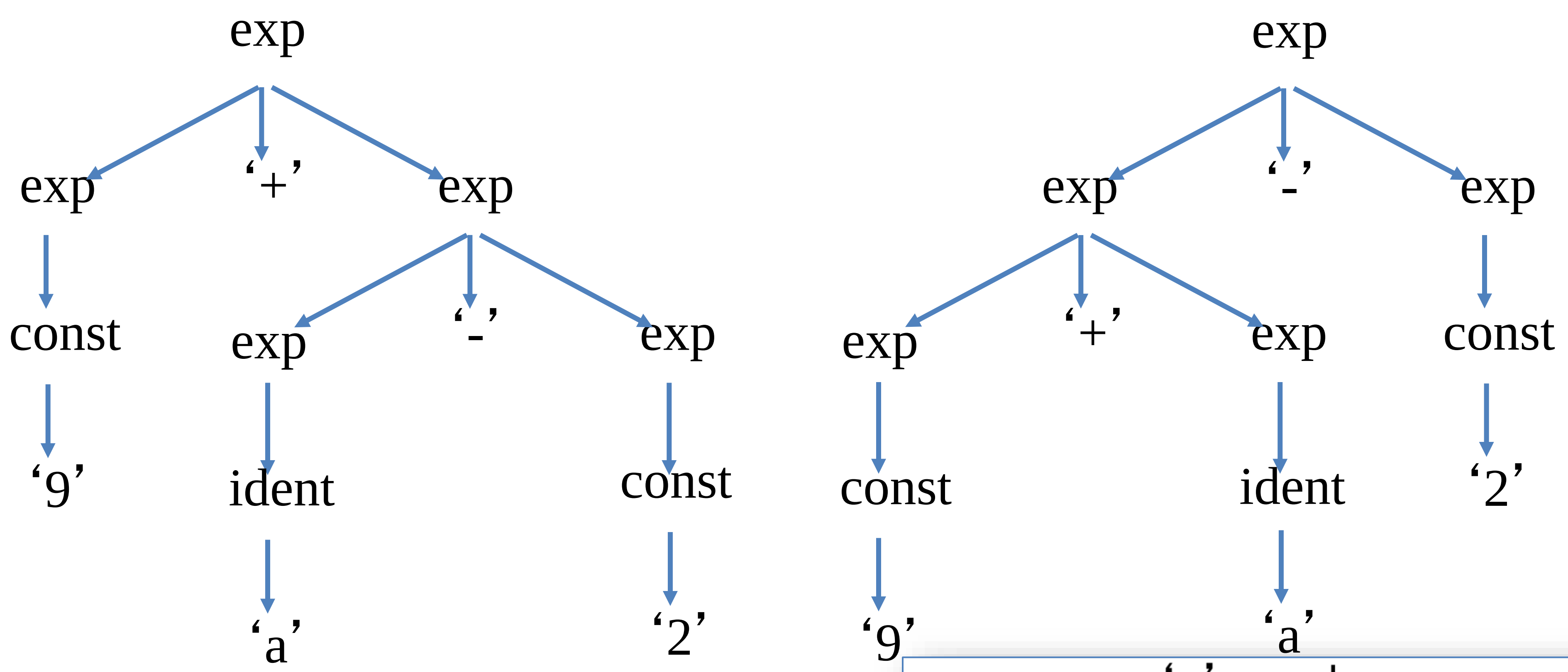
- The string "9 + a – 2" has two different parse trees according to our example grammar:



exp → exp '+' exp |
        exp '–' exp | const | ident

Ambiguity…

- The string “9 + a – 2” has two different parse trees according to our example grammar:



exp $\rightarrow$ **exp ‘+’ exp** |
exp ‘–’ exp | **const** | **ident**

Associativity

- Associativity concerns how sequences like "9+a-2" are parsed
- For example, what is the right interpretation of
  "2 – 3 – 4" ?
- **left-associativity**: (2 – 3) – 4
- **right-associativity**: 2 – (3 – 4)
- The choice is a matter for the language designer, who must take into account intuitions and convenience.

Right associativity applies to arithmetic only in unusual languages (e.g. APL). However it is just right, for example, for lists in Haskell:

1 : 2 : 3 : [ ].

Precedence

- What is the right interpretation of 9 + 5 * 2 ?
- Normally we assume that " * " has higher **precedence** than " + ":

9 + 5 * 2	9 + (5 * 2)
5 * 2 + 9	(5 * 2) + 9
9 – 5 / 2	9 – (5 / 2)
5 / 2 + 9	(5 / 2) + 9
12 + 5 * 2 + 9	(12 + (5 * 2)) + 9

- In fact there can be even higher levels of precedence:
  - 12 + 5 * 2 ^ 15 + 9 = 12 + (5 * (2 ^ 15)) + 9

For our example language

- All our operators are left-associative
- "*" and "/" have higher precedence than "+" and "–".

A useful way to think about precedence is to consider each precedence level separately: at the lowest level we have

term op_1 term op_2 … op_n term

(where *op* is a level 1 operator, "+" or "–").

Each term may be a constant (or an identifier), or may be a similar sequence composed from higher-precedence operators.

An unambiguous grammar for arithmetic expressions

- This grammar avoids ambiguity by expressing associativity and precedence explicitly. It does it by splitting `exp` into two layers, `exp` and `term`:

```
exp → exp + term |
      exp - term |
      term
term → term * factor |
       term / factor |
       factor
factor → const | ident
```

(Term level – a list of terms separated by low-precedence operators)

(Factor level – a list of factors, separated by high-precedence operators)

- Now consider an example: “9+5*2”. Is it possible to find two parse trees?

Parse tree with unambiguous grammar

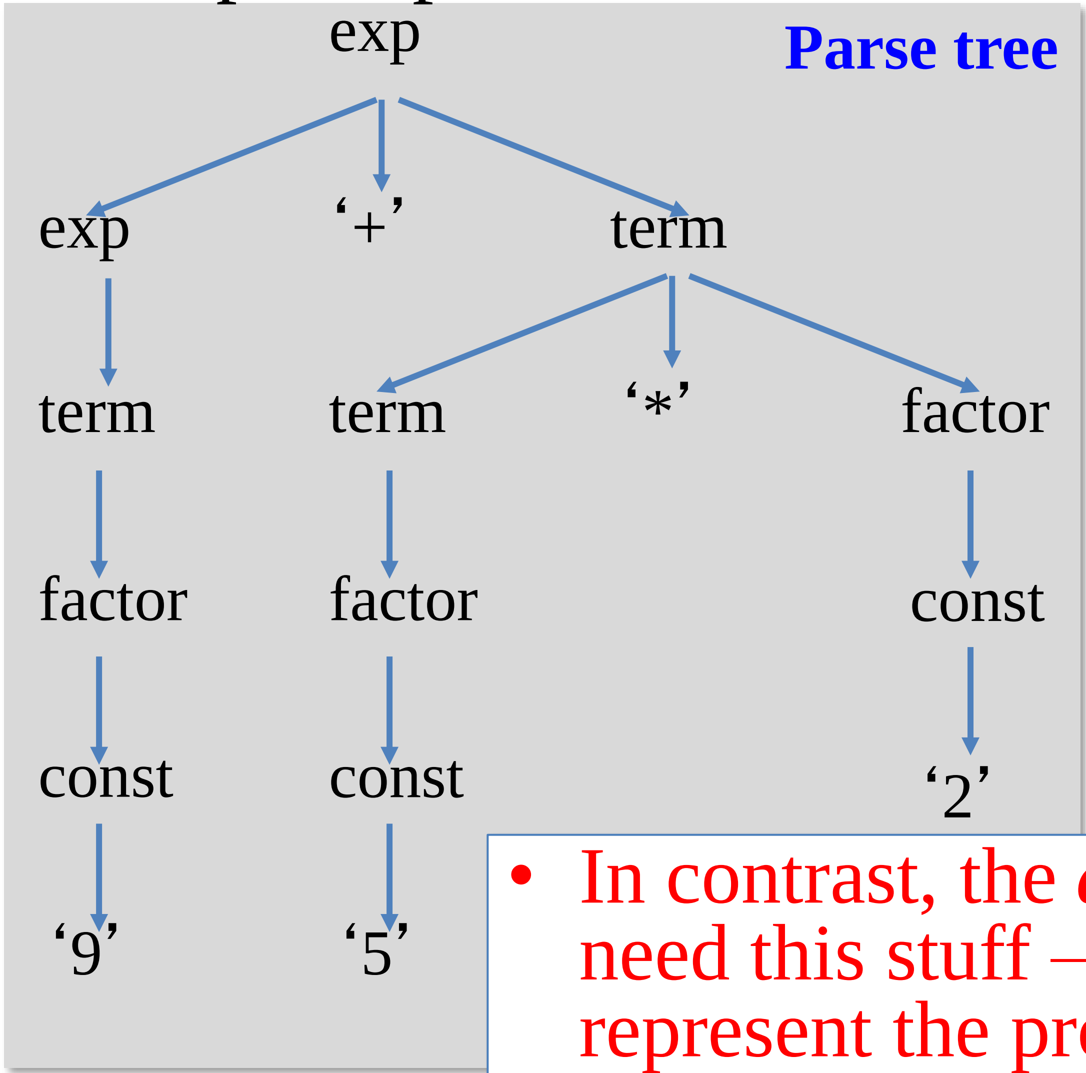
- With the new grammar the parse tree for “9 + 5 * 2” is:

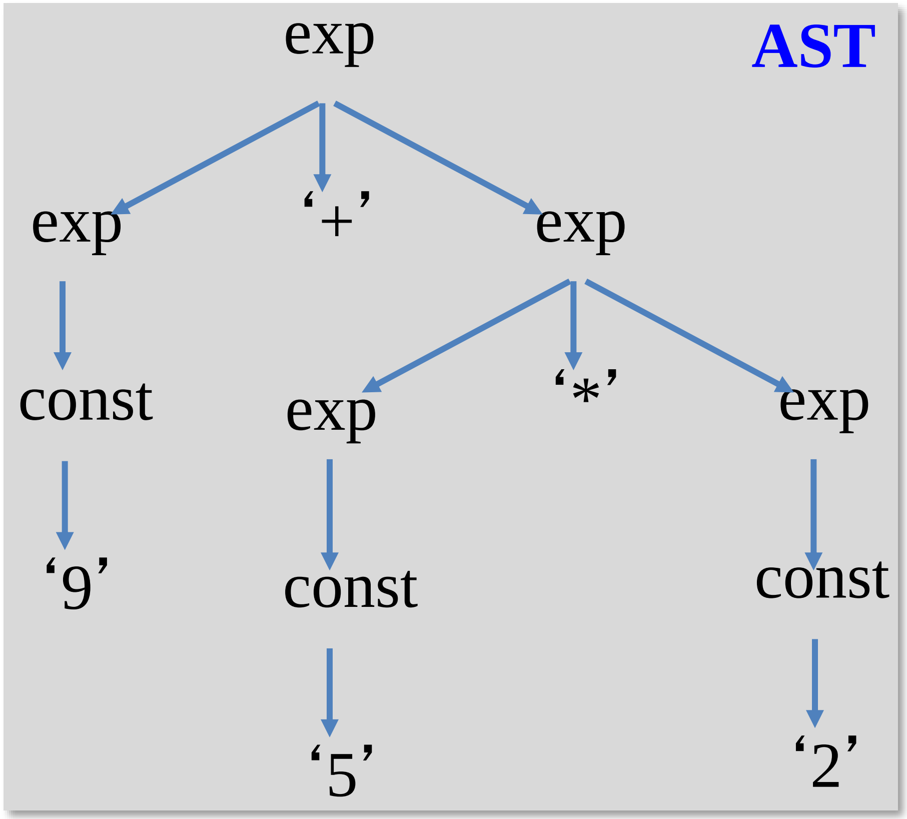
$$\begin{tikzcd}
 & \text{exp} \arrow[dl] \arrow[d] \arrow[dr] & & \\
\text{exp} \arrow[d] & \text{‘+’} & \text{term} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{term} \arrow[d] & \text{term} \arrow[d] & \text{‘*’} & \text{factor} \arrow[d] \\
\text{factor} \arrow[d] & \text{factor} \arrow[d] & & \text{const} \arrow[d] \\
\text{const} \arrow[d] & \text{const} \arrow[d] & & \text{‘2’} \\
\text{‘9’} & \text{‘5’} & &
\end{tikzcd}$$

exp → exp + term |
 exp - term |
 term
term → term * factor |
 term / factor |
 factor
factor → const | ident

Parse tree versus Abstract Syntax Tree

- The parse tree is rather complicated because of the need to capture precedence etc:





- In contrast, the ***abstract*** syntax tree doesn't need this stuff – the tree we ***build*** to represent the program can be simpler

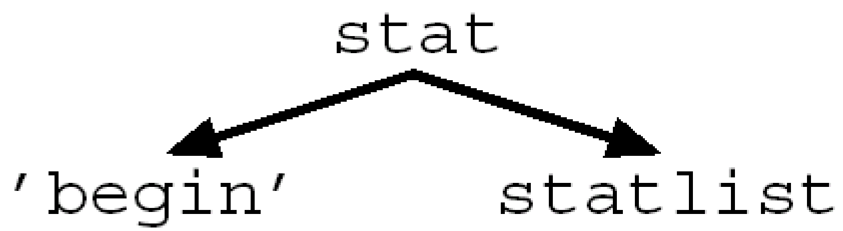
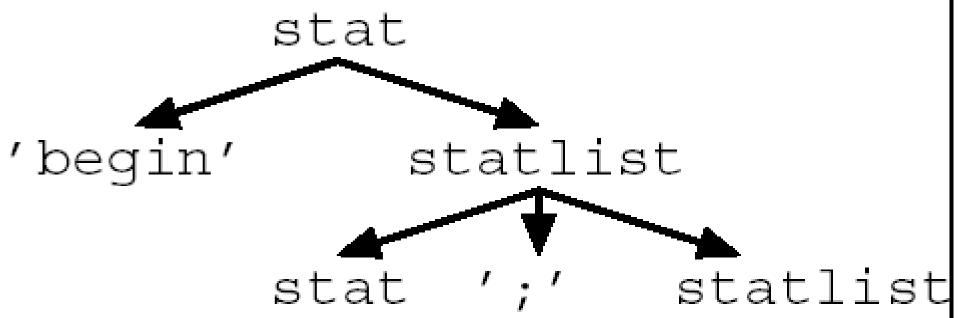
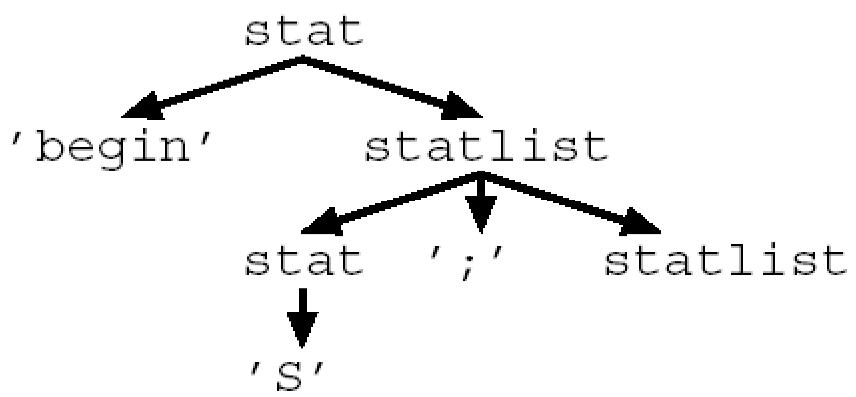
Parsers

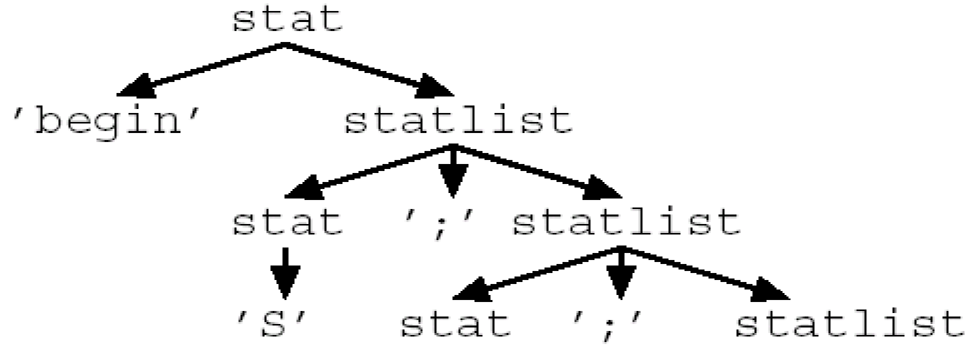
- The purpose of a parser (=syntax analyser) is to check that the input is grammatically correct, and to build an abstract syntax tree (AST) representing its structure.
- There are two general classes of parsing algorithms:
  - **top-down** or **predictive** (we will study the **recursive descent** algorithm)
  - **bottom-up** (also known as **shift-reduce** algorithms)

Top-down parsing

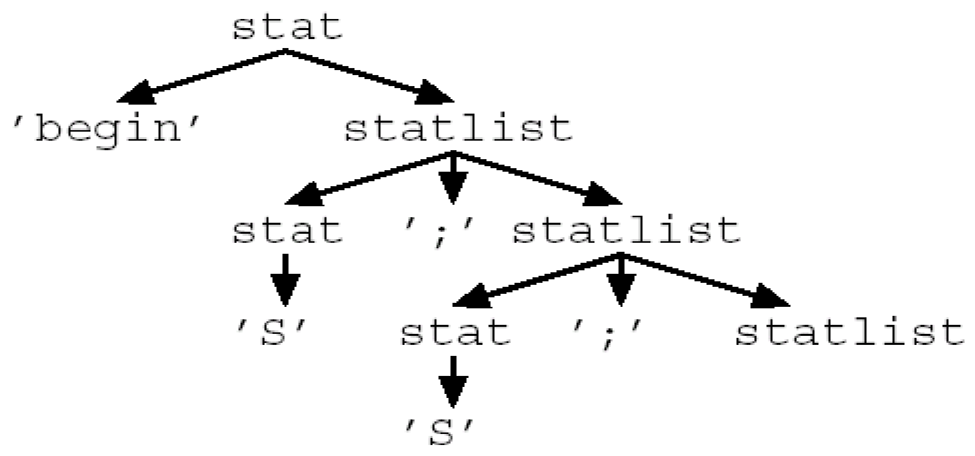
- Example:
  - stat → ‘`begin`’ statlist
  - stat → ‘`S`’
  - statlist → ‘`end`’
  - statlist → stat ‘`;`’ statlist
- Example input: “`begin S; S; end`”
- **Slogan**: *“Starting from the start symbol, search for a rule which rewrites the nonterminals to yield terminals consistent with the input”*
- The challenge in designing a top-down parser is to look at each terminal in turn, just once, and use it to decide, once and for all, which rule to use.

Top-down parsing – “the search for a derivation”

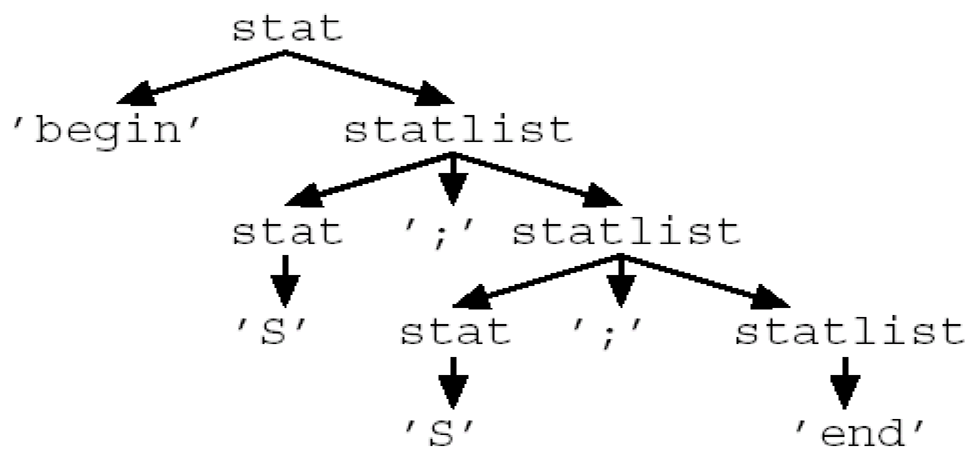
Parse tree so far	*Input*
stat	begin S; S; end
	$\underline{\texttt{begin}}$ S; S; end
	no match; persevere
 stat 'begin' statlist stat ';' statlist 'S'	$\underline{\texttt{begin S;}}$ S; end



no match; persevere



$\underline{\texttt{begin S; S;}}\ \texttt{end}$



$\underline{\texttt{begin S; S; end}}$

The final parse tree

All input consumed

Top-down parsing:

- Assume input is derived from start symbol (**`stat`** in our example)
- Examine each alternative production for `stat`
- Compare first unmatched input token with first symbol on RHS of each alternative production for `stat`
  - **If a matching production is found** (e.g. '**`begin`**') use it to rewrite
  - Repeat, using next input token to determine the production to be used for the next non-terminal
  - **If no match,** try a production which begins with a non-terminal (e.g. "**`stat ';' statlist`**")

At each step, one of the productions was chosen, and used from left-to-right, to replace a non-terminal in the parse tree by a RHS.

Q: what if we choose the wrong production?

- How might we choose the wrong production?

- Example:

  stat → ‘loop’ statlist ‘until’ exp

  stat → ‘loop’ statlist ‘forever’

- Can you see how to modify the grammar to avoid this problem?

Q: what if we choose the wrong production?

- How might we choose the wrong production?

- Example:

  stat → ‘loop’ statlist ‘until’ exp

  stat → ‘loop’ statlist ‘forever’

Instead:

  stat → ‘loop’ statlist stat2

  stat2 → ‘until’ exp

  stat2 → ‘forever’

- Not all such problems are as easy to cure as this one

Bottom-up parsing

- ("shift-reduce", as used in most parser generators)

In **top-down** parsing the grammar is used *left-to-right*: in trying to match against a non-terminal, each of the possible RHSs are tried in order to extend the parse tree correctly.

In **bottom-up** parsing, the input is compared against all the RHSs, to find where a string can be replaced by a non-terminal by using a production from *right-to-left*. Parsing succeeds when the whole input has been replaced by the start symbol.

Bottom-up parsing Example…

Recall the grammar:

stat → ‘begin’ statlist | ‘S’
(Rule A)

statlist → ‘end’ | stat ‘;’ statlist *(Rule B)*

Walkthrough of bottom-up parse:

Stack	*current symbol*	*Remaining input*	*Action*
	begin	S ; S ; end	shift
begin	$\underline{\text{S}}$	; S ; end	reduce, rule A
begin	stat	; S ; end	shift
begin stat	;	S ; end	shift
begin stat ;	$\underline{\text{S}}$	; end	reduce A
begin stat ;	stat	; end	shift
begin stat ; stat	;	end	shift
begin stat ; stat ;	$\underline{\text{end}}$		reduce B
begin stat ; $\underline{\text{stat ;}}$	$\underline{\text{statlist}}$		reduce B
begin $\underline{\text{stat ;}}$	$\underline{\text{statlist}}$		reduce B
$\underline{\text{begin}}$	$\underline{\text{statlist}}$		reduce A
	stat		finished!

Bottom-up parsers

- Bottom-up parsers are somewhat complicated to construct by hand.
- However, they can be constructed automatically by *parser generators*
- **This is often the most practical way to build a parser, and doing this forms part of lab work associated with this course**
- Meanwhile we will look at top-down (in particular, recursive descent) parsing:
  - Recursive descent parsers are easy to construct by hand
  - But you sometimes have to modify your grammar first

Use textbooks:

Introductory:

- EaC Chapter 1
- Dragon book Ch.s 1 and 2.
- Appel Ch. 1

General grammar issues, top-down parsing

- EaC Chapter 3 sections 3.1-3.3
- Dragon Book pp.42, pp.60 and Ch. 4
- Appel Ch. 3.

Bottom-up parsing

- EaC Chapter 3 section 3.4
- Dragon Book Ch. 4, pp.233
- Appel Ch. 3 pp.57

Parser-generators

- EaC Section 3.5
- Dragon Book Ch. 4 pp.287
- Appel Ch. 3 pp.68
- Web documentation, eg search "ANTLR"

- Use your textbook to get an overall picture of what a compiler is and how a compiler is put together

- We will cover grammars and parsing in more detail later in the course

A complete (simple!) compiler…

- In the next few slides we'll see a complete – but very very simple – compiler
- **The input**: a string of characters representing an arithmetic expression
- **The output**: a sequence of instructions for a simple stack-based computer
- When these instructions are executed, the expression is evaluated

- By doing this in Haskell, the entire compiler will fit on a handful of slides

Recursive descent parsing in Haskell

- The easiest way to construct a top-down parser is to use "recursive descent".
- The idea is straightforward, and is best learned by doing it yourself – see tutorial exercise.
- **Using Haskell makes it possible to write an entire parser on two slides, as will be demonstrated:**
- **Example grammar:**

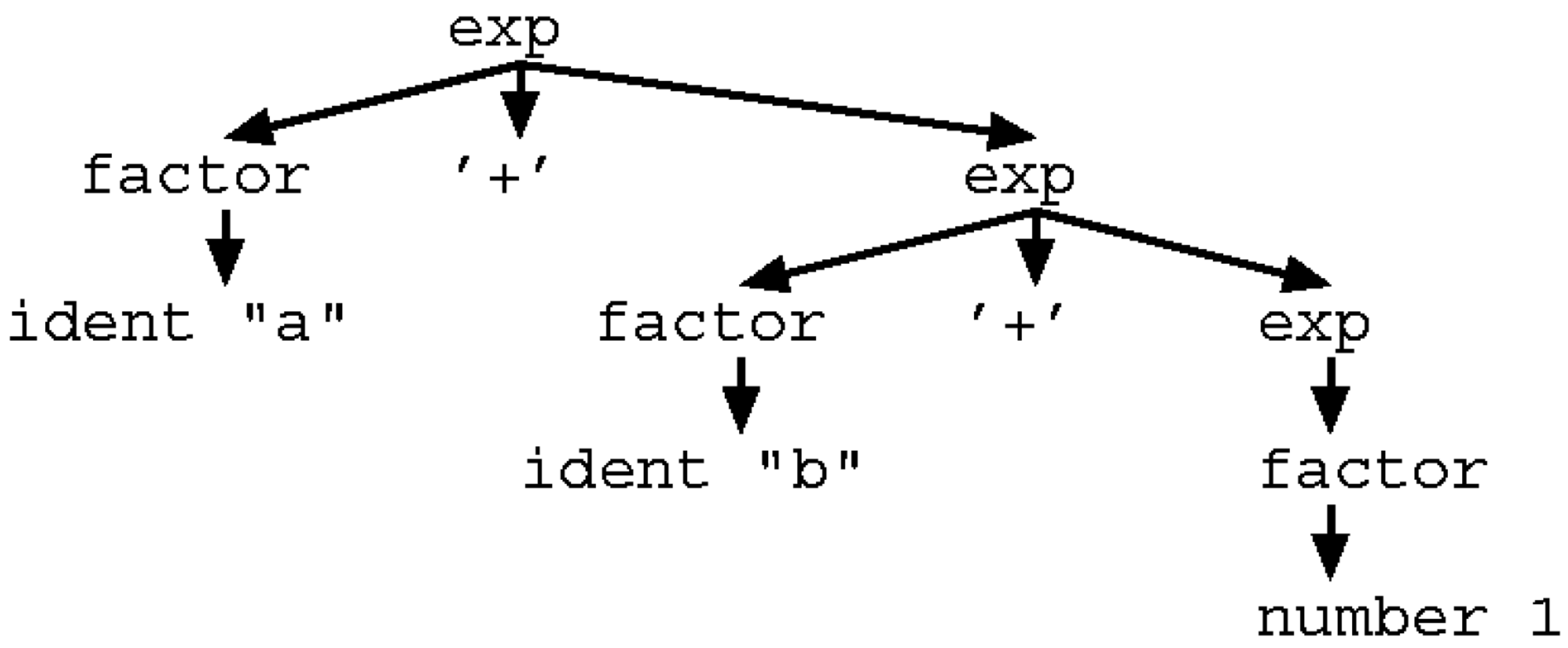
  exp $\rightarrow$ factor '+' exp | factor

  factor $\rightarrow$ number | identifier

Recursive descent parsing in Haskell…

- Example “`a+b+1`”
- Input to parser is Haskell list of lexical tokens:

  [IDENT ”a”, PLUS, IDENT ”b”, PLUS, NUM 1]
- Parse tree:

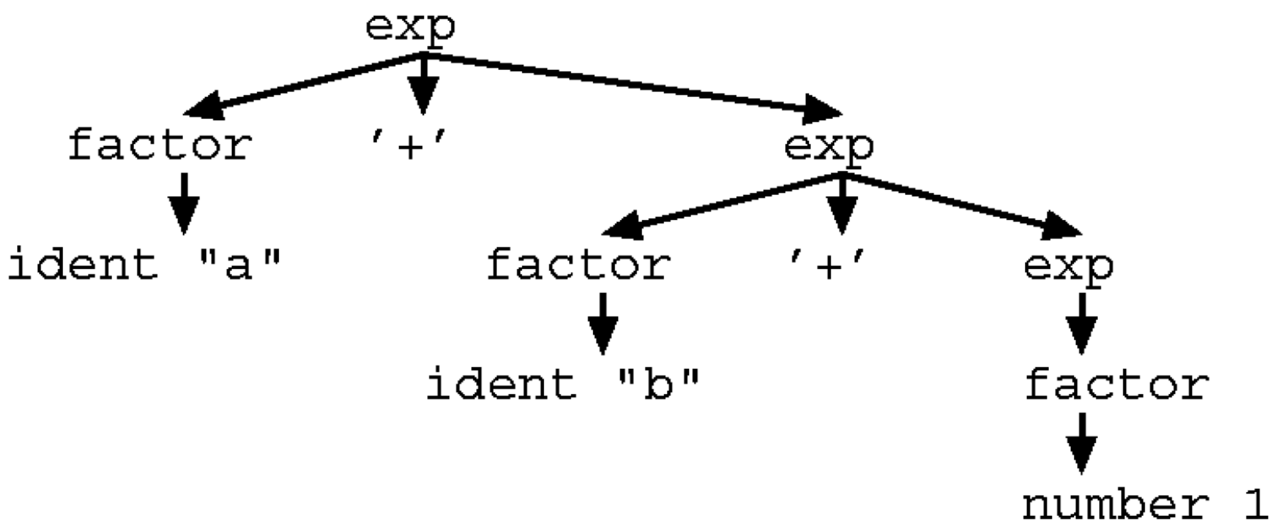


- Parser returns abstract syntax tree as Haskell data structure:

  Plus (Ident ”a”) (Plus (Ident ”b”) (Num 1))

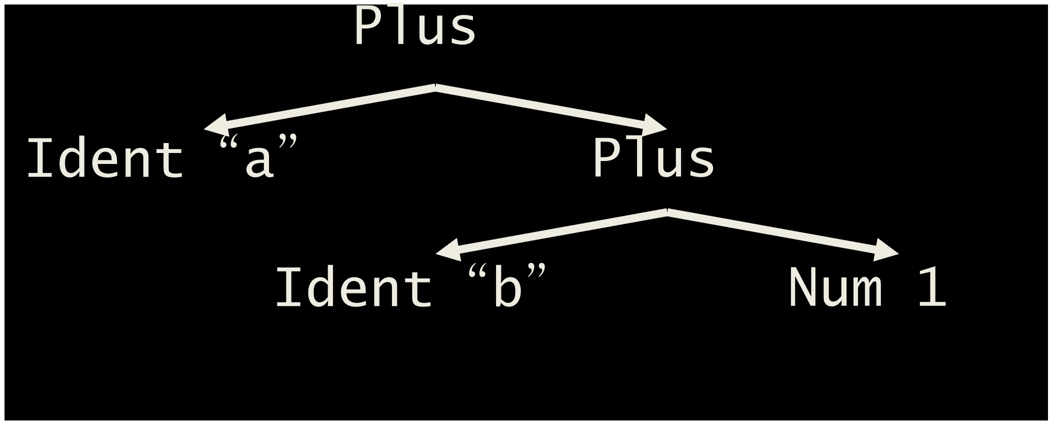
Abstract Syntax Tree (AST)

- The output of our parser is a simplified tree
- The parse tree:



- The AST:

Plus (Ident "a") (Plus (Ident "b") (Num 1))



Haskell data types

We need data type specifications for lexical tokens and the AST:

```
data Token
  = IDENT [Char] | NUM Int | PLUS

data Ast
  = Ident [Char] | Num Int | Plus Ast Ast
```

NOTE:

1. Haskell data declarations look a bit like BNF but the similarity is misleading.

2. The `Token` and `Ast` data types look similar, but observe that `Ast` is a tree while `Token` is not.

Lexical analysis and parsing in Haskell

- Let’s assume we have already written the lexical analyser/scanner (see later in the slides). It will have the type:

```
scan :: [Char] → [Token]
```

- The parser itself then takes a list of lexical tokens as input, and produces a AST as output:

```
parser :: [Token] → Ast
```

- If the parser finds a syntax error, it should produce a helpful error message explaining what was expected.

- **Principle**: a parse-function for each non-terminal
  - There are two non-terminals in the grammar, exp and factor

```
exp → factor '+' exp | factor
factor → number | identifier
```

  - The job of a parse-function is to **look** at the next token in the input, **choose** a production which matches, and **return** when it has found a sequence of tokens which can be derived from the given non-terminal:

```
parseExp :: [token] → (ast, [token])

parseFactor :: [token] → (ast, [token])
```

  - A parse-function returns two things:
    - The AST of the exp it has found
    - The remaining tokens after the expression has been consumed.

A simple parse function – base case

- The parse function for the non-terminal ‘factor’ is easy to write – just look at the next token and see whether it’s a number or a variable name:

```
exp → factor '+' exp | factor
factor → number | identifier
```

```
parseFactor ((NUM n):restoftokens)
  = (Num n, restoftokens)

parseFactor ((IDENT x):restoftokens)
  = (Ident x, restoftokens)
```

A simple parse function

- It’s sometimes useful to use “case” instead of pattern matching on the function arguments:

```
parseFactor (firstToken:restoftokens)
  = case firstToken of
      NUM n → (Num n, restoftokens)
      IDENT x → (Ident x, restoftokens)
      other →
       error “Number or identifier expected”
```

- The parse function for the non-terminal 'exp' is based directly on the definition – find a 'factor'; see if there's a '+', if so find another 'exp':

```
exp → factor '+' exp |
      factor
```

```
parseExp tokens
  = let
      (factortree, rest) = parseFactor tokens
    in
      case rest of








.
```

- The parse function for the non-terminal 'exp' is based directly on the definition – find a 'factor'; see if there's a '+', if so find another 'exp':

```
exp → factor '+' exp |
      factor
```

```
parseExp tokens
  = let
     (factortree, rest) = parseFactor tokens
    in
     case rest of
       (PLUS : rest2) →
          let
            (subexptree, rest3) =
          in
            (Plus factortree subexptree, rest3)

       othertokens → (factortree, othertokens)
```

- The parse function for the non-terminal 'exp' is based directly on the definition – find a 'factor'; see if there's a '+', if so find another 'exp':

```
exp → factor '+' exp |
      factor
```

```
parseExp tokens
  = let
      (factortree, rest) = parseFactor tokens
     in
      case rest of
        (PLUS : rest2) →
          let
            (subexptree, rest3) = parseExp rest2
          in
            (Plus factortree subexptree, rest3)

        othertokens → (factortree, othertokens)
```

The parser

```
parse tokens
  = let
      (tree, rest) = parseExp tokens
    in
     if null rest then
       tree
     else
       error "excess rubbish“
```

- First call parseExp
- ParseExp returns the Ast for the expression it finds, together with the remaining Tokens
- Then check there are no remaining tokens

The code generator

- Suppose we have a stack-based computer whose instruction set is represented by the following Haskell data type:

```
data Instruction
  = PushConst Int | PushVar [Char] | Add
```

- Here is the complete code generator for our language:

```
translate :: Ast → [Instruction]
translate (Num n) = [PushConst n]
translate (Ident x) = [PushVar x]
translate (Plus e1 e2)
  = translate e1 ++ translate e2 ++ [Add]
```

- It should be clear how this would have to be modified to handle other arithmetic operators like -, * and / .

Example

- Input: “10+5-a”
- scan “10+5-a”

  → [NUM 10, PLUS, NUM 5, MINUS, IDENT “a”]
- parse (scan “10+5-a”)

  → Plus (Num 10) (Minus (Num 5) (Ident “a”))
- translate (parse (scan “10+5-a”))

  →[PushConst 10,

  PushConst 5,

  PushVar “a”,

  Minus,

  Add]

(note that the grammar is right-recursive)

Example: code generation in Haskell

- Haskell works by *rewriting*
- At each step we select a term to rewrite and an equation to rewrite it with

```
translate :: Ast → [Instruction]
translate (Num n) = [PushConst n]
translate (Ident x) = [PushVar x]
translate (Plus e1 e2)
  = translate e1 ++ translate e2 ++ [Add]
```

- As we progress, the resulting list of instructions is concatenated

$= \underbrace{\text{translate (Plus (Num 10) (Minus (Num 5) (Ident “a”)))}}$

$= \underbrace{\text{translate (Num 10)}} \text{ ++ } \underbrace{\text{translate (Minus (Num 5) (Ident “a”))}} \text{ ++ Add}$

$= \underbrace{\text{translate (Num 10)}} \text{ ++ } \underbrace{\text{translate (Num 5)}} \text{ ++ } \underbrace{\text{translate (Ident “a”)}} \text{ ++ Minus ++ Add}$

= [PushConst 10, PushConst 5, PushVar “a”, Minus, Add]

Lexical analysis

- Our basic compiler for arithmetic expressions is now complete except for one small detail: the lexical analyser, often called the scanner. Lexical analysis is covered in great detail in the standard textbooks. But it's fairly easy to write one by hand:

```
scan :: [Char] -> [Token]
```

- where

```
data Token = PLUS | MINUS | TIMES | DIVIDE |
   NUM Int | IDENT [Char]
```

- The punctuation cases are easy:

```
scan [ ] = [ ]                                   (end of input)
scan (' ' :rest) = scan rest                     (skip spaces)
scan ('+' :rest) = PLUS : (scan rest)
scan ('-' :rest) = SUB : (scan rest)
```

- The cases of numbers and identifiers are a little more complicated. If a digit (a character between `0` and `9`) is found, we are at the beginning of a possibly-large number. We should collect the digits, convert them into a number, and return it with the NUM token:

```
 scan (ch:rest)
   | isDigit ch  = let (n, rest2) = convert (ch:rest)
                   in
                     (NUM n):(scan rest2)
```

- where convert is a function which collects the digits of a number, converts it to binary, and returns the remaining characters.

- Identifiers can be dealt with in the same way:

```
scan (ch:rest)
  | isAlpha ch  = let (n, rest2) = getname (ch:rest)
                  in
                    (IDENT n):(scan rest2)
```

- where `getname` is a function which collects alphanumeric characters to form a variable name, and returns the remaining input.

- *Question:* How would `scan` have to be modified to handle multi-character punctuation (e.g. the assignment symbol ":=")?

- For completeness, here is an implementation of `convert` and `getname`:

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

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

Parsing

```
data Ast
  = Ident [Char] | Num Int |
     Plus Ast Ast | Minus Ast Ast
data Instruction
  = PushConst Int | PushVar [Char] | Add | Sub |

scan :: [Char] -> [Token]
parser :: [Token] -> Ast

parseExp :: [Token] -> (Ast, [Token])
parseFactor :: [Token] -> (Ast, [Token])

parser tokens
  = let (tree, rest) = parseExp tokens
    in if null rest then
         tree
       else error "excess rubbish"

parseFactor ((NUM n):restoftokens)
  = (Num n, restoftokens)

parseFactor ((IDENT x):restoftokens)
  = (Ident x, restoftokens)

parseExp tokens
  = let (factortree, rest) = parseFactor tokens
     in case rest of
       (PLUS : rest2) ->
         let (subexptree, rest3) = parseExp rest2
         in
           (Plus factortree subexptree, rest3)

       (MINUS : rest2) ->
         let (subexptree, rest3) = parseExp rest2
         in
           (Minus factortree subexptree, rest3)

       othertokens -> (factortree, othertokens)
```

Code generation

```
translate :: Ast -> [Instruction]

translate (Num n) = [PushConst n]
translate (Ident x) = [PushVar x]
translate (Plus e1 e2)   = translate e1 ++
                           translate e2 ++
                           [Add]
translate (Minus e1 e2) = translate e1 ++
                           translate e2 ++
                           [Sub]
```

Compiler

```
compiler :: [Char] -> [Instruction]

compiler input
  = translate (parser (scan input))
```

The complete compiler

We have now assembled all the components of a complete compiler for a very simple language

You can find a working Haskell implementation on the course web site

http://www.doc.ic.ac.uk/~phjk/CompilersCourse

Lexical analysis

```
data Token
  = IDENT [Char] | NUM Int | PLUS | MINUS

scan [] = []                              -- (end of input)
scan (' ':rest) = scan rest               -- (skip spaces)
scan ('+':rest) = PLUS : (scan rest)
scan ('-':rest) = MINUS : (scan rest)

scan (ch:rest)
  | isDigit ch  = let (n, rest2) = convert (ch:rest)
                  in
                    (NUM n):(scan rest2)

scan (ch:rest)
  | isAlpha ch  = let (n, rest2) = getname (ch:rest)
                  in
                    (IDENT n):(scan rest2)

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

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

Conclusion

- This concludes the introductory component of the course. We have seen how a complete compiler is constructed, although the source language was very simple and the translation was very naive. We will return to each aspect later.

- Suppose the grammar were left-recursive instead of right recursive: exp → exp '+' factor | factor
- What goes wrong?

```
parseExp tokens
  = let
      (subexptree, rest) = parseExp tokens
    in
      case rest of
        (PLUS : rest2) →
          let
            (factortree, rest3) = parseFactor rest2
          in
            (Plus subexptree factortree , rest3)

        othertokens → (factortree, othertokens)
```

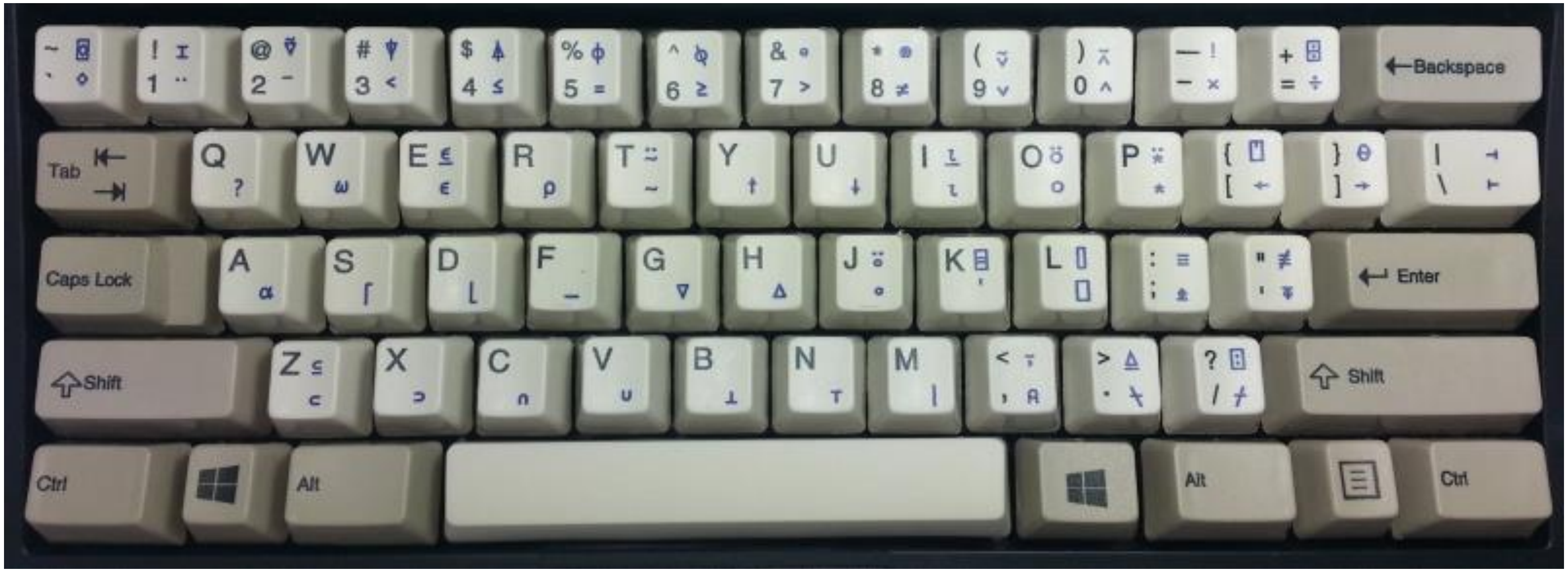
Extra slide for anticipated question

Feeding curiosity…

- "**Fast Context-Free Grammar Parsing Requires Fast Boolean Matrix Multiplication**", Lillian Lee, JACM 2002. So general, arbitrary CFGs can be parsed with the same complexity as Boolean matrix-matrix multiply – eg using Strassen's algorithm.
- Why might you need to parse general CFGs? Imagine a language where you can import new *syntax*! See "**Reliable and Automatic Composition of Language Extensions to C: The ableC Extensible Language Framework**", Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk, OOPSLA 2017.
- In fact you could embed one language directly into another. Eg mixed Python+PHP. And then automatically generate a compiler from the interpreters of the two languages. See "**Fine-grained Language Composition: A Case Study**", Edd Barrett, Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt, ECOOP 2016.

Feeding curiosity…

- Q: Why are *all* operators in APL *right*-associative?
- A: because APL has dozens of single-character operators, and no-one would be able to remember a more complicated rule
- APL programmers had to buy special keyboards:



- Game of life in APL:
```
life←{⊃1 ω ∨.∧ 3 4 = +/ +/ 1 0 ¯1 ∘.⊖ 1 0 ¯1 ⌽¨ ⊂ω}
R (life R) (life life R)
```

From https://tryapl.org/