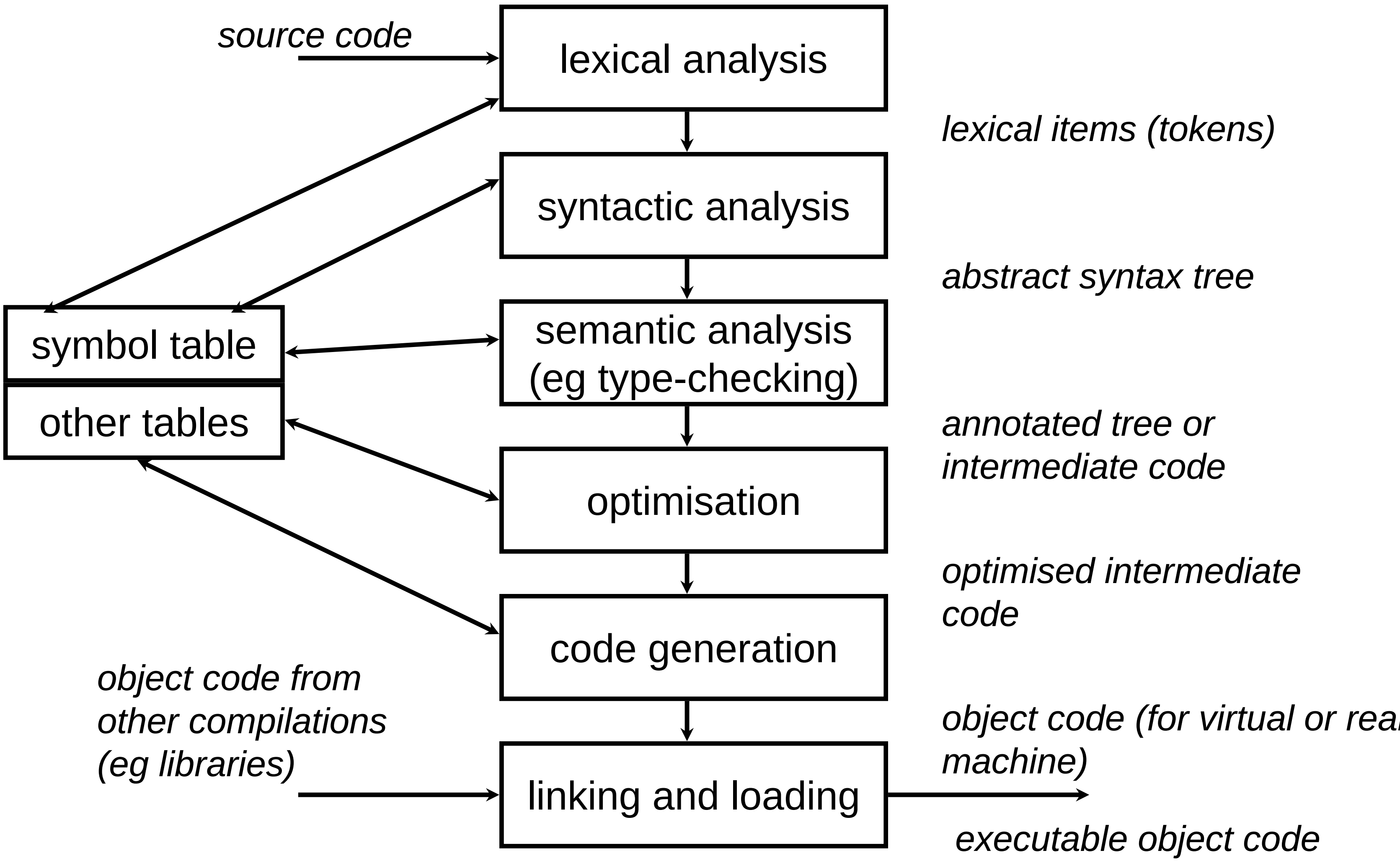
Session Plan

- Session 2: ***Language processing & lexical analysis***
  – language processing
    » what is lexical analysis
    » what is syntax analysis
  – lexical syntax (token) examples
  – lexical syntax (token) definition
    » regular expressions
  – implementation
  – tools

Language processing



Lexical analysis

Straightline example program:

```
a := 5+3;
b := (print(a, a-1), 10*a);
print (b)
```

Lexical analysis converts text stream into a token stream, where tokens are the most basic symbols (words and punctuation):

a	**:=**	**5**	**+**	**3**	**;**	**b**	**:=**	**(**	**print**	**(**	**a**	**,**	**a**	**-**	**1**	**)**	**,**	**10**	*****	**a**	**)**	**;**

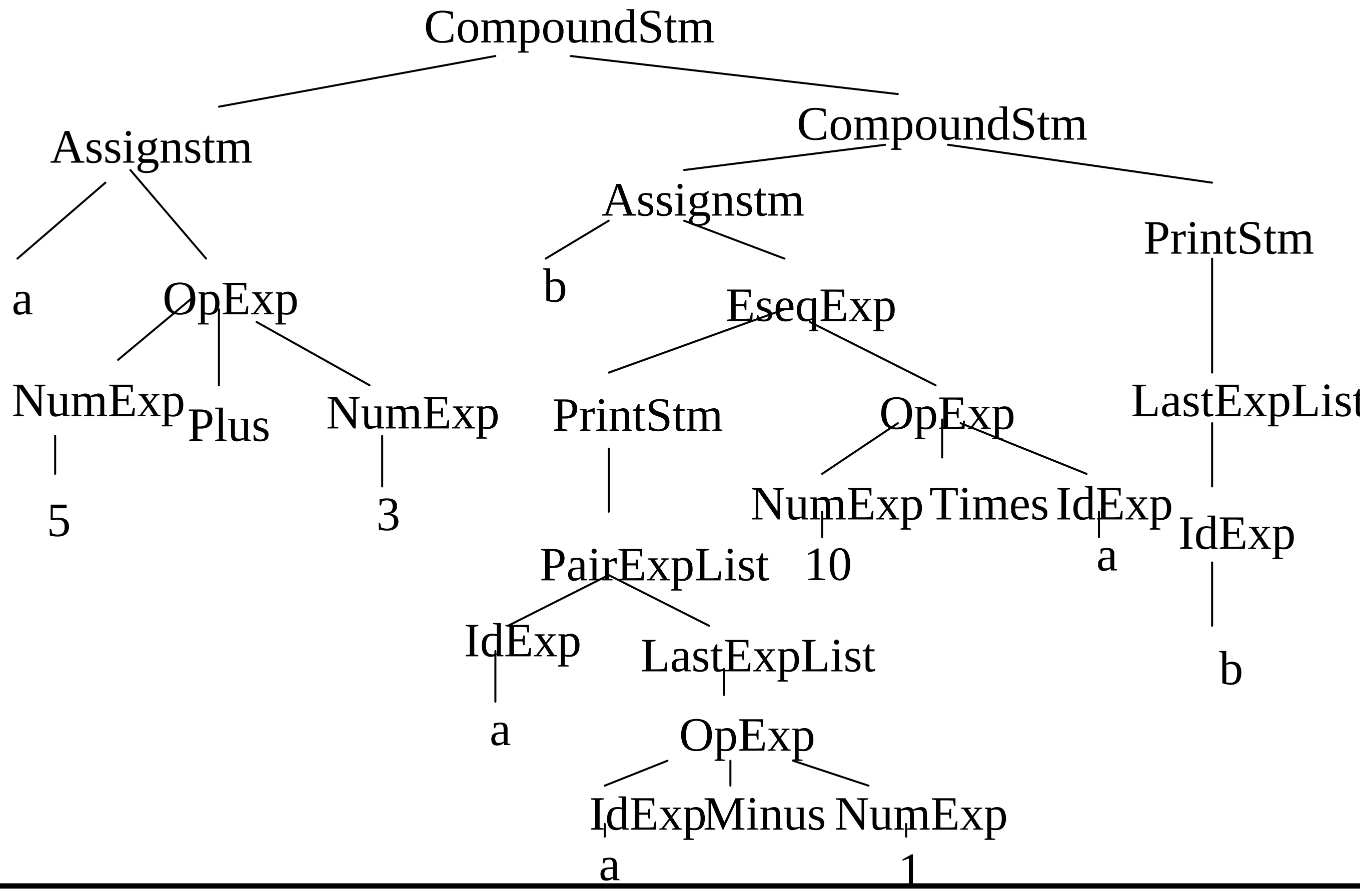
...

Each box is a lexical item or token. A possible representation:

ID(a) ASSIGN NUM(5) PLUS NUM(3) SEMI ID(b) ASSIGN LEFTPAREN KEYPRINT LEFTPAREN ID(a) COMMA ID(a) MINUS NUM(1) RIGHTPAREN COMMA NUM(10) TIMES ID(a) RIGHTPAREN SEMI ...

Syntax analysis

Converts a token stream into a useful abstract representation (here an abstract syntax tree):



Lexical tokens

Type	Examples				
ID	`foo`	`n14`	`last`		
NUM	`73`	`0`	`00`	`515`	`082`
REAL	`66.1`	`.5`	`10.`	`1e67`	`5.5e-10`
IF	`if`				
COMMA	`,`				
LPAREN	`(`				
ASSIGN	`:=`				

- Some tokens eg ID NUM REAL have *semantic values* attached to them, eg ID(n14), NUM(515)
- Reserved words: Tokens e.g. IF VOID RETURN constructed from alphanumeric characters, cannot be used as identifiers

Example informal specification: Identifiers in C or Java

An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter. Upper- and lowercase letters are different. If the input stream has been divided into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token. Blanks, tabs, newlines and comments are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords and symbols.

Formal specifications of tokens

- Approach:
  – Specify lexical tokens using the formal language of regular expressions
  – Implement lexical analysers (lexers) using deterministic finite automata (DFA)
  – fortunately, there are automatic conversion tools…
- Languages
  – A *language* is a set of *strings*
  – A string is a finite sequence of *symbols*
  – Symbols are taken from a finite *alphabet*

Regular expressions

- **Symbol**: for each symbol **a** in the alphabet of the language, the regular expression (regex) **a** denotes the language containing just the string a
- **Alternation**: Given 2 regular expressions M and N then M | N is a new regex. A string is in lang(M|N) if it is lang(M) or lang(N). The lang(**a|b**) = {a,b} contains the 2 strings a and b.
- **Concatenation**: Given 2 regexes M and N then M•N is a new regex. A string is in lang(M•N) if it is the concatenation of 2 strings α and β s.t. α in lang(M) and β in lang(N). Thus regex **(a|b)•a** = {aa,ba} defines the language containing the 2 strings aa and ba

Regular expressions

- **Epsilon**: The regex ε represents the language whose only string is the empty string. Thus (a•b)|ε represents the language { `""`,`"ab"` }
- **Repetition**: Kleene closure of M is M*
  String in M* if it is the concatenation of ≥ 0 strings, all in M. Thus ((a|b)•a)* represents the infinite set {`""`, `"aa"`, `"ba"`, `"aaaa"`, `"baaa"`, `"aaba"`, `"baba"`, `"aaaaaa"`,…}

Examples

- (0|1)*•0
- b*(abb*)*(a|ε)
- (a|b)*aa(a|b)*
- Conventions: omit • and ε, assume Kleene closure binds tighter than • binds tighter than |
- ab | c means (a•b)|c
- (a |) means (a|ε)

Abbreviations (extensions)

- [abcd] means (a | b | c | d)
- [b-g] means [bcdefg]
- [^b-g] or ~[b-g] means everything *but* [bcdefg]
- [b-gM-Qkr] means [bcdefgMNOPQkr]
- *M*? means (*M* | ε)
- *M*+ means *M(M*)*

- NB: a lexical specification should be *complete*.

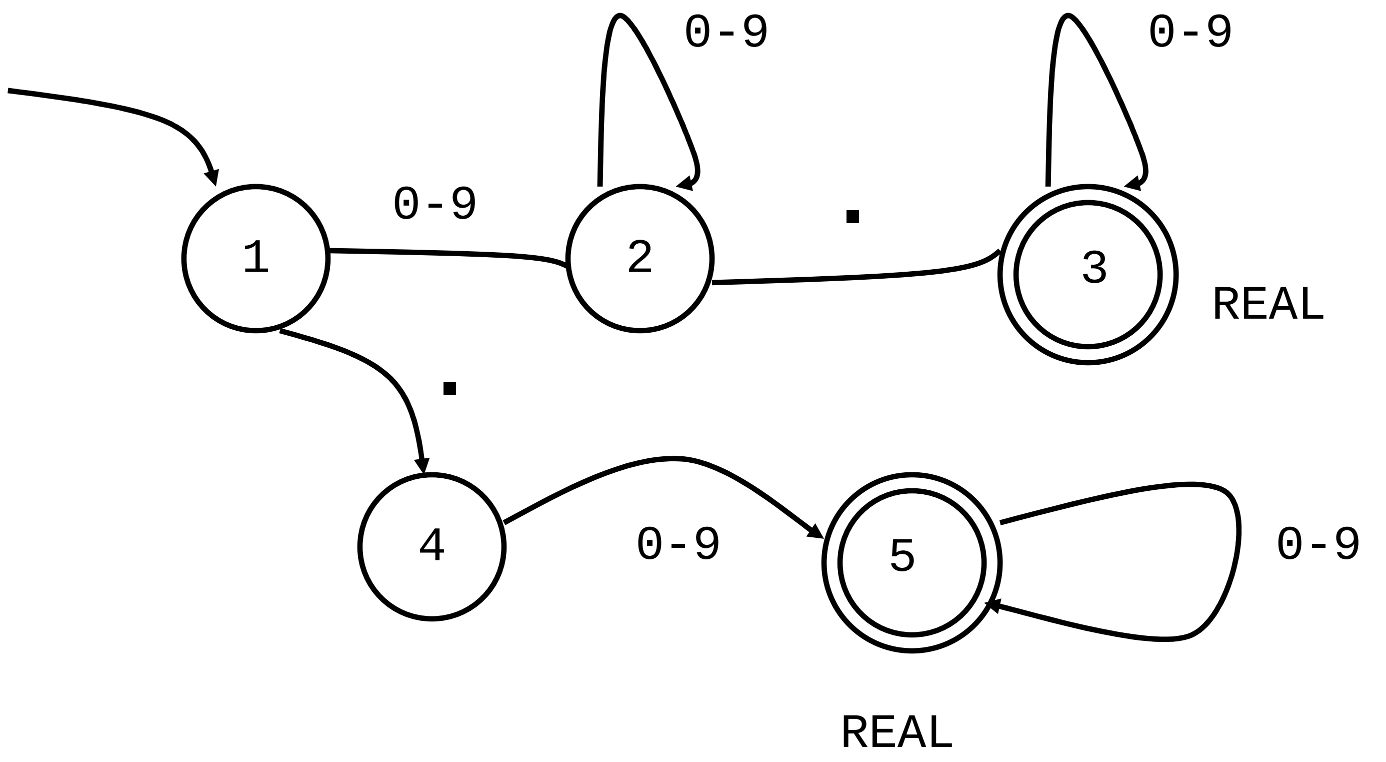
Regular expression summary

a or "a"	ordinary character, stands for itself
ε	the empty string
	another way to write the empty string
M | *N*	alternation
M • *N*	concatenation (often written simply as *MN*)
*M**	repetition (zero or more times)
M+	repetition (one or more times)
M?	Optional, zero or one occurrence of M
[a-zA-Z]	Character set alternation
["a"-"z""A"-"Z"]	Character set alternation (JavaCC form)
`.` or ~[]	Any single character (~[] is JavaCC form)
"\n" "\t" "\""	newline, tab, double quote (quoted special characters)
`"a.+*"`	quotation, string stands for itself (ie in this case `a.+*`)

Regular expressions for some tokens

`(" "|"\n"|"\t")`	***no token; whitespace; ignore***
`if`	IF
`[a-z][a-z0-9]*`	ID
`[0-9]+`	NUM
`([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)`	REAL
`("--"[a-z]*"\n")`	***comment starting --; ignore***
`.`	***error***

Finite automaton



(From Appel Figure 2.3)

Finite automaton implementation

```
method Token lex() {  // example automaton for REAL only
    int state = 1; String text = “”;  char ch;
    while (true) {
            ch = nextchar();    // Get the next input character
            text = text + ch;    // Collect the text of the token
            if (state == 1)
                        if (ch >= ‘0’ && ch <= ‘9’) state = 2;
                        else if (ch == ‘.’) state = 4;
                        else lexerror(ch);
            else if (state == 2)
                        if (ch >= ‘0’  && ch <= ‘9’) state = 2;
                        else if (ch == ‘.’) state = 3;
                        else lexerror(ch);
…// see next slide
```

Finite automaton implementation (continued)

```
…// continued from previous slide

        else if (state == 3)
                if (ch >= ‘0’  && ch <= ‘9’) state = 3;
                else return new Token(REAL,new Double(text));
        else if (state == 4)
                if (ch >= ‘0’  && ch <= ‘9’) state = 5;
                else lexerror(ch)
        else if (state == 5)
                if ch >= ‘0’  && ch <= ‘9’) state = 5;
                else return new Token(REAL,new Double(text));
        else error (“Illegal state: shouldn’t happen”);
    }
}
```

JavaCC compiler-compiler

- Fortunately, tools can produce finite automata programs from regular expressions…

JavaCC lexical specification
(regular expressions and some action code)

JavaCC tool

text

Java program recognising regular expressions and executing code

tokens

JavaCC token regular expressions

- characters and strings must be quoted, eg:
  - “;” “int” “while” “\n” “\”hello\””
- character lists […] provide a shorthand for |, eg:
  - [“a”-”z”] matches “a” through “z”, [“a”,”e”,”i”,”o”,”u”] matches any single vowel, ~[“a”,”e”,”i”,”o”,”u”] any non-vowel, ~[] any character
- repetition with + and *, eg:
  - [“a”-”z”,”A”-”Z”]+ matches one or more letters
  - [“a”-”z”]([“0”-”9”])* matches a letter followed by zero or more digits
- shorthand with ? provides for optional expr, eg:
  - (“+”|”-”)?([“0”-”9”])+ matches signed and unsigned integers
- tokens can be named
  - TOKEN : { < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >}
  - TOKEN : {< LETTER: ["a"-"z", "A"-"Z"] > | < DIGIT: ["0"-"9"] >}
  - now <IDENTIFIER> can be used later in defining syntax

JavaCC lexical analysis example

```
/* file MyParser.jj */

PARSER_BEGIN(MyParser)
   class MyParser {}
PARSER_END(MyParser)

TOKEN : {
     < IF: "if" >
  |  < #DIGIT: ["0"-"9"] >
  |  < ID: ["a"-"z"] (["a"-"z"]|<DIGIT>)* >
  |  < NUM: (<DIGIT>)+ >
  |  < REAL: ( (<DIGIT>)+"."(<DIGIT>)* ) |
             ( (<DIGIT>)*"."(<DIGIT>)+ ) >
}

SKIP : {
     < "--" (["a"-"z"])* ("\n" | "\r" | "\r\n") >
  |  " " | "\t" | "\n" | "\r"
}
```

*# means can use **only** in TOKEN definitions*

JavaCC lexical analysis example

```
void Start() :
{Token t;}
{ ( ( t=<IF> | t=<ID> | t=<NUM> | t=<REAL> )
           { System.out.println("token found: “
                 + tokenImage[t.kind]
                 + "('"+t.image+"')"); }
  )* <EOF>
} /* end of file MyParser.jj */
```

```
/* file Main.java */
class Main {
  public static void main(String args[]) throws
                          ParseException {
    …
    MyParser parser = new MyParser(System.in);
    parser.Start();
  }
}
```

JavaCC introduction

- generates a combined lexical analyser and parser (a Java class)…here I've called the class MyParser
- this session we're learning about lexical analysis
  - JavaCC does this and it is the focus of our first example – we'll see more about complete JavaCC-generated parsers later
  - all you need to know now is that it also generates a Java method to recognize the things we've labelled under 'Start()'
  - to make it work,
    - » create an object…MyParser parser = new MyParser(*inputstream*)
    - » and then call the method…parser.Start()
- a class Token is also generated
  - field image is the string matched for the token
  - field kind can be used to index array tokenImage to see the token type

What you should do now…

- read chapter 2
  - don’t worry too much about finite automata stuff
  - you really do need to know all about regular expressions
  - think about how to represent real numbers with exponents using regular expressions