City University London

IN2009
Language Processors

Week 2
Language Processing & Lexical Analysis

Christian Cooper

Before we start…

- Last week you were asked to derive **`interp()`** and **`maxArgs()`**.
- Here is *one* possible implementation...

`Interp` class: Method `interp()`

```
static void interp(Stm s) {
   Table t = interpStm(s, new Table ("", 0, null));
 }
static Table interpStm (Stm s, Table t) {
  if (s instanceof PrintStm) {
    return interpPrintedExps(((PrintStm) s).exps, t);
  } else if (s instanceof AssignStm){
    AssignStm as = (AssignStm) s;
    IntAndTable it  = interpExp (as.exp, t);
    return update (it.t, as.id, it.i);
  } else if (s instanceof CompoundStm) {
    CompoundStm cs = (CompoundStm) s;
    return interpStm (cs.stm2, interpStm (cs.stm1, t));
  } else {
    System.out.println ("error in interpStm");
    return t;
  }
}
```


Method `interpPrintedExps()`

```
static Table interpPrintedExps (ExpList e, Table t) {
  if (e instanceof PairExpList) {
    PairExpList pe = (PairExpList) e;
    IntAndTable it = interpExp (pe.head, t);
    System.out.print (it.i);
    System.out.print (" ");
    return interpPrintedExps (pe.tail, it.t);
  } else if (e instanceof LastExpList) {
    IntAndTable it = interpExp (((LastExpList) e).head,t);
    System.out.println (it.i);
    return it.t;
  } else {
    System.out.println ("error in interpPrintedExps");
    return t;
  }
}
```


Method `intAndTable()`

```
static IntAndTable interpExp (Exp e, Table t) {
  if (e instanceof IdExp) {
    return new IntAndTable (lookup (t,((IdExp) e).id),t);
  } else if (e instanceof NumExp) {
    return new IntAndTable (((NumExp) e).num, t);
  } else if (e instanceof OpExp) {
    OpExp oe = (OpExp) e;
    IntAndTable it1 = interpExp (oe.left, t);
    IntAndTable it2 = interpExp (oe.right, it1.t);
    if (oe.oper == OpExp.Plus) {
      return new IntAndTable (it1.i+it2.i, it2.t);
    } else if (oe.oper == OpExp.Minus) {
      return new IntAndTable (it1.i-it2.i, it2.t);
    }
…
```


Method `intAndTable()`

```
…

else if (oe.oper == OpExp.Times) {
      return new IntAndTable (it1.i*it2.i, it2.t);
    } else if (oe.oper == OpExp.Div)
      return new IntAndTable (it1.i/it2.i, it2.t);
    } else {
      throw new Error ("interpExp: oper not 
  recognised");
    }
    else if (e instanceof EseqExp)
      return interpExp (((EseqExp) e).exp, interpStm 
  (((EseqExp) e).stm, t));
    else
      throw new Error ("interExp: exp not recognised");
  }
```

Method `maxargs(Stm s)`

```
static int maxargs(Stm s) {
  if (s instanceof PrintStm) {
    return Math.max (maxargs (((PrintStm) s).exps),
             length (((PrintStm) s).exps));
  } else if (s instanceof AssignStm)
     return maxargs (((AssignStm) s).exp);
  } else if (s instanceof CompoundStm)
     return Math.max (maxargs (((CompoundStm) s).stm1),
             maxargs (((CompoundStm) s).stm2));
  } else {
    System.out.println("maxargs(Stm): unrecognised Stm");
    return 0;
  }
}
```


method `maxargs(ExpList e)`

```
static int maxargs (ExpList e) {
   if (e instanceof PairExpList)
     return Math.max
       (maxargs (((PairExpList) e).head),
        maxargs (((PairExpList) e).tail));
   else if (e instanceof LastExpList)
     return maxargs (((LastExpList) e).head);
   else {
     System.out.println
       ("maxargs(ExpList): unrecognised ExpList");
     return 0;
   }
 }
```

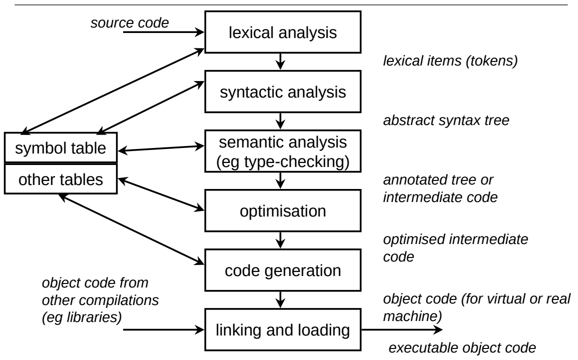

method `maxargs(Exp e)`

```
static int maxargs (Exp e) {
   if (e instanceof OpExp) {
     return Math.max
       (maxargs (((OpExp) e).left),
        maxargs (((OpExp) e).right));
   } else if (e instanceof EseqExp) }
     return Math.max
       (maxargs (((EseqExp) e).stm),
        maxargs (((EseqExp) e).exp));
   } else {
     /* it's an IdExp or a NumExp */
     return 0;
 }
```


Session Plan

Session 2
- Language processing
- Lexical analysis
- 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):

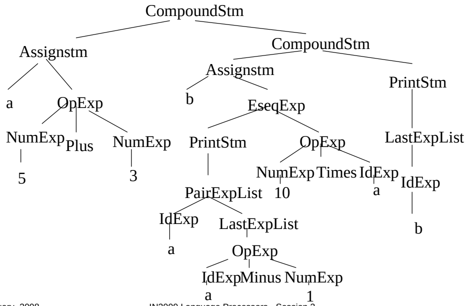
a	:=	5	+	3	;	b	:=	(	print	(	a	,	a	-	1	)	,	10	*	a	)	;	...

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."

Identifiers in C or Java

- *"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 for our sanity, there are automatic conversion tools…

Formal specifications of tokens

"a formal language is a language that is defined by precise mathematical or machine processable formulas" (Wikipedia)

- Languages
  - A language is a set of strings
  - A string is a finite sequence of symbols
  - Symbols are taken from a finite alphabet
- Many types of formal languages, at this stage we will consider **regular languages**.

Regular languages

- Regular languages can be:
  - Accepted by a **deterministic finite state machine**
  - Accepted by a **nondeterministic finite state machine**
  - Accepted by an alternating finite automaton
  - Described by a **regular expression**
  - Generated by a regular grammar
  - Generated by a prefix grammar
  - Accepted by a read-only Turing machine
  - Defined in monadic second-order logic

Regular expressions

- Regular expressions (regex, regexp) - strings that describe the syntax of other strings.
- These are simply *patterns*, constructed by syntactical rules and defining a set of strings.

Regular expressions

- There is a useful conne(x|ct)ion to real life software development here, too.
  - Regexps can be used in a number of programming languages and tools
  - TextPad allows for searching for text given a regular expression - much more powerful than plain text searching.
  - Java contains a class RegExp, too.
- On to the rules...

Regular expressions

Symbol

- For each symbol a in the alphabet of the language, the regexp a denotes the language containing just the string **a**

Regular expressions

Alternation

- Given 2 regular expressions M and N then M | N is a new regexp.
- 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**.

Regular expressions

Concatenation

- Given 2 regexes M and N then M•N is a new regexp.
- 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 regexp **(a|b)•a** = {aa,ba} defines the language containing the 2 strings **aa** and **ba**

Regular expressions

Epsilon

- The regexp ε represents the language whose only string is the empty string.
- Thus (a•b)|ε represents the language { `"","ab"` }

Regular expressions

Repetition

- *M* - Kleene closure* (or Kleene Star) of *M*
- A 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

Regular expression summary

["a"-"z""A"-"Z"]	Character set alternation (JavaCC)
~[]	Any single character (**`~[]`** is JavaCC form)
"\n" "\t" "\""	newline, tab, double quote (quoted special characters)
`"a.+*"`	quotation, string stands for itself (in this case **`a.+*`**)
[a-zA-Z]	Character set alternation

Regular expressions for some tokens

`(" "|"\n"|"\t")` ***no token; whitespace; ignore***

`if` **IF**

`a-z][a-z0-9]*` **ID**

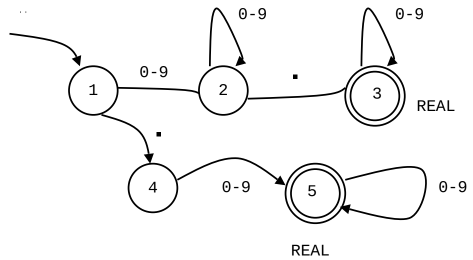
`[0-9]+` **NUM**

`([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 text of the token
    if (state == 1) {
      if (ch >= '0' && ch <= '9'){
        state = 2;
      } else if (ch == '.') {
        state = 4;
      } else {
        lexerror(ch);
      }
  }
…
```


Finite automaton implementation

```
…
  else if (state == 2) {
  if (ch >= '0'  && ch <= '9') {
      state = 2;
    } else if (ch == '.') {
      state = 3;
    } else {
      lexerror(ch);
    }
  }
…
```


Finite automaton implementation

```
…
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)
    }
  }
…
```


Finite automaton implementation

```
…
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");
  }
}
```

- Tedious and trivial to program!

JavaCC compiler-compiler

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

```
        JavaCC lexical specification
 (regular expressions and some action code)
                    |
                    v
               JavaCC tool
                    |
                    v
text      Java program recognising regular     tokens
  -----&gt;  expressions and executing code      -----&gt;
```


JavaCC token regexps

- 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

JavaCC token regexps

- 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 the class is called 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

JavaCC introduction

– 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…

- Re-read chapter 2
  - Don't worry too much about the finite automata stuff.
  - But **do** worry about regular expressions - you need to know all about these!
  - Think about how to represent real numbers with exponents using regular expressions… because:

Assessment 1, part 1

Preamble:

- Individual or declared pairwork (but no more than a pair, else trouble awaits)
- Hand in using Cityspace.
- Guard your work, don't risk plagiarism charges by leaving a USB key with your work around, or "sharing answers" etc.

Assessment 1, part 1

- Use JavaCC **regular expressions** to define precisely *integer literals* and *floating point literals*.
- In this context, 'literal' means the piece of text that appears in a program to denote a number (for example, the text `'3.142'` denotes the number 3.142).
- Implement and test your expressions using JavaCC (make your expressions readable and understandable).

Integer Literals

- An integer literal may be expressed as:
  - *binary*,
  - *decimal*,
  - *hexadecimal*, or
  - *octal* numerals.
- Each may optionally be suffixed with the character `L` to denote an integer of type *long*, and may be prefixed with a + or a `-` character to indicate sign.

Integer Literals

- A **decimal numeral** is either the single character `0`, or consists of a digit from `1` to `9`, optionally followed by one or more digits from `0` to `9`.
- A **binary numeral** consists of the leading characters `0b` or `0B` followed by one or more of the digits `0` or `1`.

Integer Literals

- A **hexadecimal numeral** consists of leading characters `0X` or `0x` followed by one or more hexadecimal digits.
- A *hexadecimal digit* is a digit from `0` to `9` or a letter from `a` through `f` or `A` through `F`.
- An **octal numeral** consists of a digit `0` followed by one or more of the digits `0` to `7`.

Integer Literals

- Examples of integer literals:

```
0
19960372
0xDadaCafe
0L
0777L
0xC0B0L
0x00FF00FF
0b00100110
0b110010L
4263556900031337111211331146 41
```


Floating Point Literals

- A floating point literal has the following parts:
  - a *whole-number part,*
  - a *decimal point* (represented by the period character `.`),
  - a *fractional part,*
  - an *exponent*, and
  - a *type suffix*. A type suffix is either the letter `d` (denoting *double* type) or `f` (denoting *float* type).

Floating Point Literals

- The exponent, if present, is indicated by the letter `e` followed by an optionally signed number.
- At least one digit, in either the whole number or the fraction part is required.
- One of the following is also required:
  - a decimal point,
  - an exponent, or
  - a float type suffix

Floating Point Literals

- All other parts are optional.
- Subject to the previous constraints, the the fractional-part and the number in the exponent are sequences of digits from `0` to `9` (i.e. decimal only).
- The whole-number part is a sequence of digits from `0` to `9` and may optionally be prefixed with a `+` or a `-` character to indicate sign.

Floating Point Literals

- Examples:

```
1e1f     2.f
.3f      0f
3.14f    6.022137e23f
1e1      2.
0.3      0.0
3.14     1e-9d
1e137    -5.56e4263
-42f
```

Advice

- Create regular expressions for each useful case (so perhaps one for binary integer literals, another for decimal integer literals, etc.), and build up larger expressions from smaller ones.
- Test your expressions regularly!
- Aim for a smaller number of entirely correct cases rather than lots of broken ones.

Next Lecture

- ***Parsing I (syntax analysis)***
- Monday 4th February, 2008
  - 11:00 - 12:50
  - C.348