**McMaster University**
**Department of Computing and Software**
**Dr. W. Kahl**

**SFWR ENG 3E03**
**Exercise Sheet 1**

# Design and Selection of Programming Languages

11 September 2005

## Exercise 1.1 — Lexing

The latest ISO standard for the C programming language, namely ISO/IEC 9899 with Technical Corrigenda 1 and 2, is available on-line at the following URL:

http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf

(a) Read the sections "5.1.1.2 Translation phases" and "6.4 Lexical elements", and point out potential portability problems allowed by the standard. Note that in particular most of the "Semantics" subsections are not relevant for basic lexing.

(b) Write a separate flex lexer implementing translation of trigraphs and line splicing as described in translation phases 1 and 2 in section 5.1.1.2.

(c) Adapt the toy lexer from the lecture (available on the course page) to C identifiers not containing universal character names.

(d) Add universal character names for at least one language of your choice.

(e) Further adapt the lexer to recognise C integer constants.

(f) Further adapt the lexer to recognise C character constants.

(g) Further adapt the lexer to recognise C comments. (Hint: Use "start conditions" in the lexer.)

## Exercise 1.2 — Expression Representation in Java

The following expression class definitions are available on the course page as Expression1.java:

```
class Operator {
  private char _op;
  public Operator (char c) { _op = c; }
}

abstract class Expression {} // Expression = Value + Variable + Binary

class Value extends Expression { // Value = int
  int intValue;
}
class Variable extends Expression { // Variable = String
  String name;
}
class Binary extends Expression { // Binary = Expression × Operator × Expression
  Operator op;
  Expression term1, term2;
}
```

(a) Add a *toString* method to these classes such that *e.toString*() produces an infix representation of any *Expression e* with **at least** the necessary parentheses for being able to apply higher-priority operators like * to argument expresssions constructed from lower-priority operators like +.

(b) Make the fields private and add constructors that ensure that only expressions with reasonable string representation can be constructed.

**Document** your definition of "reasonable string representation" in each case!

(c) The graphviz tool suite from AT&T includes, among others, the graph layout tools

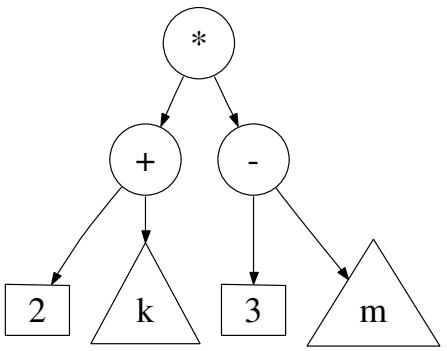
- dot for layout of directed, typically acyclic graphs — the main principle of the algorithm is to arrange nodes into "levels", and then reduce edge crossings, and
- neato for layout of undirected graphs using a "spring embedding" algorithm that understands nodes as electically charged and therefore repelling each other, and edges as springs of equal spring constants and lengths, and therefore produces always drawings with straight-line edges.

To the right is the the drawing that `dot` produces when invoked with

```
dot -Tps -o E1.ps E1.dot
```

on the following input file corresponding to the expression $(2 + k) * (3 - m)$:

```
digraph MT1a {
  node [fontsize="30"];
  edge [labeldistance="1",fontsize="30"];
  "+" [shape="circle"];
  "*" [shape="circle"];
  "-" [shape="circle"];
  "2" [shape="box"];
  "3" [shape="box"];
  "k" [shape="triangle"];
  "m" [shape="triangle"];
  "*" -> "+";
  "*" -> "-";
  "+" -> "k";
  "+" -> "2";
  "-" -> "3";
  "-" -> "m";
}
```




Extend the expression class with a *writeDotGraph* method that produces such a `dot` file.

**Note:** What should the interface of this method be? How do you distribute its task among the sub-classes?

***Document and justify your design decisions!*** Test your output with `dot`!

**Advanced:** Add a method *showDotGraph* that writes the `dot` file, invokes `dot` on it, and invokes a program to display the output of `dot` (if necessary in a different format).