INF3110/4110—Mandatory Exercise 1

To be delivered 21.10.2008

1 Grammar in ML and Java

Consider the simple language (GSLAN) defined by the grammar below. There are no declarations and there are only three variables (a, b, c), all of type `int`. `<number>` represents numbers of type `int`.

```
<prog> ::= <exp>
<ifelse> ::= if <boolexp> then <exp> else <exp> endif
<assign> ::= <var> := <exp>
<var> ::= a | b | c
<boolexp> ::= <exp> = <exp>
<exp> ::= ( <exp> {, <exp>}* ) | <assign> | <print> | <ifelse> |
                          <number> | <var> | <exp> + <exp>
<print> ::= print(<exp>)
```

In GSLAN, everything is an expression. Consider this example.

```
(
        a := 1,
        b := 2,
        print(if (c := a + b) = 3 then c+2 else c endif)
)
```

Each line in the example is executed in turn. The expressions in a list of comma-separated expressions enclosed in brackets are the statements in GSLAN. Here, the variable a is first set to 1, then b is set to 2 and then c+2 is printed if the assignment c := a + b assigns 3, else just c. If you think it is strange that an assignment returns something, and that a list of expressions is itself an expression, then know that it possible and legal in C and C++, and in Java assignments returns the value which was assigned.

To sum up the strange parts

- Assigments, such as a := 1+2, returns the value which was assigned to a.
- A list of expressions, such as (1, 2), returns the value of the last expression in the list. In this case 2.
- print just returns 0

Implementation in ML

You are going to write an interpreter for GSLAN in Standard ML. In order to do this, you must:

1. represent a data structure corresponding to a given program. To represent a program as a value in ML, you will typically need to declare datatypes for each of the non-terminals in the grammar. Then set up the tree-structure manually in ML consisting of the non-terminals and the terminals in the program.

2. write an interpret function. You interpret a program by calling a function `interpret` which takes a program and a state as input and gives a state as output.

To summarize, build the tree structure of a program manually and assign it to, for example, the variable myProgram as shown below. Then implement the interpret funtion which takes any program and the initial state (explained below) as input.

```
1 val myProgram = ...;
2 val interpret = fn : state * prog -> state
```

You will also need to write other functions which are called from the `interpret` function.

The program state

Since there are only three variables and no procedure calls, the program state should be quite simple to implement. You need to declare a datatype `state` which can hold three named values (including other variables if needed). [Hint: Two different ways of doing this is to use a record type or to use a list of pairs, but other solutions are also possible.]

In any case, you must also write two functions on the state: `getVal` which retrieves a variable value from the state, `putVal` which updates the state with a new value for a given variable:

```
1 val putVal = fn : var * int * state -> state
2 val getVal = fn : var * state -> int
```

Test run

The program listed above as an example should give the following output when interpret is run on the program tree:

```
1 5
2 val it = State {a=1,b=2,c=3,r=0} : state
```

The r variable is the return value of the outermost expression.

Implementation in Java

1. Represent a data structure corresponding to a given program. To represent a program as classes in Java, you will typically need to declare classes for each of the non-terminals in the grammar. Then set up the object-structure of the sample program listed above, so that you end up with an object program of the Class Program repressenting the non-terminal prog.

2. Write an interpreter. You interpret a program by calling a function interpret which is implemented in the Program class itself. The interpreter must implement the following interface. To clarify, the Program class itself, which is the root of the program tree created, is to have a function interpret on it, which runs the program.

It is a good idea to have interpret on the other tree-nodes of the program tree as well. Here is the interface which the Program class and other parts of the syntax tree must implement.

```
interface GslanInterpreter{
        void interpret();
}
```

2 Textprocessing in ML and Java

In word processing systems it is customary for lines to be filled and broken automatically to enhance the appearance of the text. This text is no exception. Input of the form

```
The heat bloomed     in december
 as the   carnival  season
            kicked into  gear.
Nearly helpless with sun and glare, I avoided Rios brilliant
sidewalks
  and glittering beaches,
panting in dark   corners
and waiting out the inverted southern summer.
```

would be transformed by filling to

```
The heat bloomed in december as the
carnival season kicked into gear.
Nearly helpless with sun and glare,
I avoided Rios brilliant sidewalks
and glittering beaches, panting in
dark corners and waiting out the
inverted southern summer.
```

To align the right-hand margin, the text is justified by adding extra inter-word spaces on all lines but the last.

```
The heat bloomed in december as the
carnival  season  kicked into gear.
Nearly helpless with sun and glare,
I  avoided Rios brilliant sidewalks
and  glittering beaches, panting in
dark  corners  and  waiting out the
inverted southern summer.
```

Implementation in Java

Implement the interface

```
interface TextFormater{
  String generateFormatedText(String text, int textWidth);
}
```

which takes any text and a text-width, and returns the text formatted to textWidth characters wide. Do not justify the text, only format the text to look like the second listing example above.

Since the purpose of this implementation is to compare object-orientation with functional programming in ML, you must use object-orientation in order to format the text. A natural choice of classes are: Character, which is the class for a character, Word, which is a class for any word, Space, which is the class for a space in between words, Line, which is a class for a line, and Text, which is the whole text. This way, a text contains a number of lines which contains a number of words and spaces where both contains a number of characters.

Implementation in ML

Implement the function

```
val formatText = fn : string * int -> string
```

which takes any text and a width as input, and produces a formatted text. Do not justify the text, only format the text to look like the second listing example above.

Treat the text as a string, and set it the following way in order to work with it.

```
val text1 = "The␣heat␣bloomed␣␣␣␣...";
```

As objects are the main artifacts in object-oriented programming, so are functions the main artifact of functional programming. Divide the task into smaller functions which each handle a simple task and combine them into more and more complex functions. The following functions should get you started.

```
val elem = fn : ''a * ''a list -> bool;
val listOfWords = fn : string -> string list
```

Here you have a function *elem* for determining whether a character is in a list, and a good intermediate step *listOfWords* which returns the list of words in its input string.

You are not allowed to use library functions such as String.tokens in order to split the text into words. You are encouraged, however, to use other library functions such as these if you find it usefull.

```
val explode = fn : string -> char list
val implode = fn : char list -> string
val hd = fn : 'a list -> 'a
val tl = fn : 'a list -> 'a list
```

Justification

After you have completed both implementations for the formatter. Add a feature to both implementations. This feature is the justification of each line. In the process of justification, each line of text gets its spaces increased so that the words spread out to touch each edge of the text.

Requirements and deliveries

For the ML implementations you are not allowed to use reference types (reference cells). The java implementations must be object-oriented. Email the files listed below to your group teacher.

- Gslan.sml
- Gslan.java
- TextProcessor.sml
- TextProcessor.java

Good luck!