1 Introduction

The purpose of this assignment is to write an interpreter for a small subset of the Lisp programming language. The interpreter should be able to perform simple arithmetic and comparisons using integer literals or variables. Two versions of the interpreter are required. One using the functional programming language ML and another using the object oriented programming language Java. The basis for both implementations are provided together with a test suite at `http://github.com/eivindgl/inf3110-assignment1`.

2 An Interpreter

Definition by wikipedia:

> In computer science, an interpreter normally means a computer program that executes, i.e. performs, instructions written in a programming language.

Most real world interpreters compiles source code into an efficient intermediate form and performs interpretation on that.

This interpreter will skip the intermediate form and execute a representation of the source code directly.

2.1 Handling Source Code

Source code is stored in simple text files on disk. The computer has no implicit understanding of the information residing in these files. They're understood as streams of bytes or characters. It follows that a preprocessing step is required before the interpreter may execute a line of source code. This preprocessing is typically divided into to distinct subtasks. Lexical analysis and parsing.

2.1.1 Lexical Analysis

This is the process of converting a sequence of characters into a sequence of tokens. A token may be a name, a number, some special keywords or a delimiter.

2.1.2 Parser

The parser creates a tree structure from the stream of tokens called a parse tree. Several checks may be performed on the finished parse tree. A statically

Figure 1: A stream of tokens

typed language will be type checked. A compiler will generate code on the basis of a parse tree. An interpreter may execute the program from this parse tree directly or perform further optimizations.

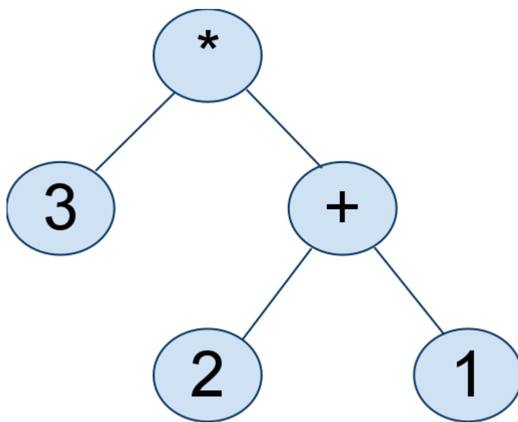

Figure 2: A parse tree created by a parser

3 Lisp

Lisp is one of the world's oldest programming languages. It's infamous for its use of parenthesis, but the language has pioneered many advanced features of programming. The name derives from "LISt Processing". Lists are one of the major data structures of the language. Lisp source code itself is made up of lists. As a result, Lisp programs can manipulate source code as a data structure, giving rise to the macro systems that allow programmers to create new syntax or even new domain-specific programming languages embedded in Lisp.

3.1 Scheme

You're about to write an interpreter for a subset of the Lisp dialect Scheme. Scheme is one of the two main dialects of Lisp. Scheme follows a minimalist design philosophy specifying a small standard core with powerful tools for

language extensions. Because of its expressiveness and compactness, it's a popular choice in education.

4 Lisp Syntax

This will only cover the constructs needed for this assignment.

All code and data are written as expressions in Lisp. When an expression is evaluated, it produces a value.

The biggest syntactical difference between Lisp and most other languages is the use of parenthesis and the syntax for function calls.

To illustrate with a function for numerical addition named *add* that has two parameters and returns the sum of those. A function call in most familiar languages would be something like this: *add(1,2)*. In Lisp it's *(add 1 2)*.

The difference is that the function name is moved within the parenthesis and that whitespace is used as a parameter delimiter.

Another difference is that there's no special support for infix operators, such as the most common mathematical operators.

4.1 Lists

A list in Lisp is written with its elements separated by whitespace and surrounded by parenthesis. The evaluation of a list is a function call. The first item in the list is the function name and the remainder of the list are the arguments to the function. Notice how this is true for the previous example: *(add 1 2)*.

4.2 Function Primitives

Only function primitives are allowed in this assignment. The logic of a primitive is implemented as a part of the interpreter, and not in the target language. Some primitives, like *if*, have a special behavior. They'll be described in this section. Other primitives, like +, are pretty straight forward and the test suite serves as a proper documentation for those.

- define
- quote
- if
- +

- -
- *
- >
- <
- =
- remainder
- quotient

4.3 Variable Definitions

Variables are created using the *define* primitive. This primitive accepts two arguments. The variable name and an expression whose value will be stored.

4.4 Quote

A quote applies to its succeeding expression. The value of a quoted expression is its value, unevaluated. The use cases for this primitive will become apparent in the next assignment.

4.5 If

The if primitive accepts two or three arguments:

- A test expression.
- A *when-true* expression.
- A *when-false* expression (optional).

If the test expression evaluates to the boolean value *false*, then the *when-false* expression should be evaluated. The *when-true* expression should be evaluated for every other value of the test expression.

It's required that exactly one of the branch expressions is evaluated. If the optional *when-false* expression is missing and the *test* expression evaluates to *false*, then the value of the expression is *VOID*.

5 Java Implementation

The basis for the Java implementation resides in the folder "java". It's recommended to use an IDE. The readme file in the java base folder contains instructions for loading the project into the Eclipse IDE. Remember to run "ant jar" to rebuild the application before running the test suite.

Due to the nature of Java, the base code will contain a large number of files. The files that needs to be changed are located in these packages:

- no.uio.ifi.inf3110.lisp_interpreter.sexp
- no.uio.ifi.inf3110.lisp_interpreter.sexp.primitives

You should also familiarize yourself with the following packages:

- no.uio.ifi.inf3110.lisp_interpreter.util
- no.uio.ifi.inf3110.lisp_interpreter.main
- no.uio.ifi.inf3110.lisp_interpreter.exceptions
- no.uio.ifi.inf3110.lisp_interpreter.sexp.views

The Main class accepts a scheme filename as a command line argument that will be loaded and interpreted.

6 ML Implementation

The basis for the ML implementation resides in the folder "ml". You should try to familiarize yourself with every file in the base folder. The only files you'll need to change are **"sexp.sml"** and **"eval.sml"**.

6.1 Compilation Manager

There's one file named sources.cm that lists all the dependencies for the ML project. This file functions as a ML equivalent of the Makefile. To load all the required files in a sml repl, start sml with the *sources.cm* file as an argument. To load the scheme repl, type *Init.repl();* at the prompt. This will run the repl function located in the Init structure (see init.sml).

```
$ sml sources.cm
Standard ML of New Jersey v110.69 [built: Mon Jun  8 23:24:21 2009]
[scanning sources.cm]
[library $SMLNJ-BASIS/basis.cm is stable]
[library $SMLNJ-LIB/Util/smlnj-lib.cm is stable]
- Init.repl();
```

```
[autoloading]
[autoloading done]
$ (+ 1 2 3)
6
$
```

There are two convenience files for running the interpreter and running a test file.

```
$ sml sources.cm run-repl.sml
$ sml sources.cm run-test.sml <path-to-test-file>
```

7 Test Suite

There's a number of test files located in the *tests* folder that contains tests of the functionality both interpreters should support. A test is a line with a lisp expression followed by a comment with the expected result. For example:

```
(+ 1 2 3) ; 6

(+ 1 2 x) ; exception
```

The first test tells the test suite that the expression *(+ 1 2 3)* should evaluate to the value *6*.

The expected value of the second test is *exception*. This tells the interpreter that the expression is illegal and should result in an error message that contains the word (ignoring case) exception. The expression is illegal because the variable x is not previously defined.

The test suite compares the actual output of a tested file with the expected output defined in the test file. This works reasonably well as long as each evaluated expression results in exactly on line of output. If the output is longer (for example a Java stack trace spanning multiple lines), then the test suite will issue a warning and skip that particular test file altogether.

Here are some example invocations of the test suite.

```
# runs every test with both implementations
$ ./test-suite
```

```
# as above, with pretty colors
$ ./test-suite --colors


# runs every test with the java implementation
$ ./test-suite --only=java

# runs every test with the ml implementation
$ ./test-suite --only=ml

# runs a single testfile with both implementations
$ ./test-suite --single=tests/oblig1/add.scm

# runs a single testfile with the java implementation
$ ./test-suite --single=tests/oblig1/add.scm --only=java

# prints help
$ ./test-suite --help
```

Both base implementations have enough functionality to pass the arithmetic addition test. You're task is to extend the implementations so every other test will pass.

```
$ ./test-suite --single=tests/oblig1/add.scm
```

8 Final Remarks

You're expected to work on simpler practice problems and read a book or an online tutorial about the ML language before solving this exercise.

Don't be afraid to ask for help if you're stuck.

The assignment is due within the 17th of September. Delivery is done by email to eivindgl@student.matnat.uio.no.