UNIVERSITY OF THE WEST INDIES
Department of Computing
COMP3652–Language Processors
Sem. I, 2015
Lecturer: Dr. Daniel Coore

**Assignment 2: Due Thursday, Nov. 12, 2015**

# Introduction

In this assignment, you will be asked to (eventually) implement an interpreter for *HPL+*, a language whose description accompanies this assignment.

Some classes have been provided in whole for you. These are mainly the classes that involve graphics rendering and the computation for scaling and combining painter frames. In addition, the top level driver loop has also been provided for you, although you may need to modify it if you do extraordinary things with your intermediate representation. When using the given driver loop, you should type a full stop on a line as the only character on the line, and press ENTER. All the input preceding that will be collected as a single string and sent to the interpreter for execution.

In some cases, classes have been started for you, with the rest of them left up to you to complete. You should try to avoid modifying the portions that have been given to you – unless you really know what you are doing.

The classes have been placed into three packages: `hpl.lang, hpl.sys, hpl.values`.

- The `hpl.lang` package is intended to contain the parser and scanner classes, and all the intermediate representation classes.
- The `hpl.sys` package contains the top level driver class, as well as any system wide utility classes.
- The `hpl.values` package contains the implementation of the `Painter` heirarchy of classes. If you expand on the notion of an *HPL+* value, then those classes should also be placed into this package.

# Syntax Processing

The *HPL+* language description document describes the core syntax of the *HPL+* language. Use it to guide you in doing this problem. Some sample *HPL+* programs can also be made available to resolve any potential ambiguities in the language description.

For this assignment, you may use either *jflex* or *jlex* to do your assignment. The former is preferred because it is better supported. A `build.xml` script has been provided for you for use with your favourite IDE (it was used with NetBeans) which will invoke *jflex*, then *cup* before attempting to compile all the class files. If your IDE cannot use it directly, feel free to adapt it as necessary.

# Abstract Syntax Tree Representation

Since HPL has two (main) types of expressions – painter expressions and arithmetic expressions – there are two categories of expressions that need to be represented within the AST for an HPL program. The classes representing these categories are called AIRExp (for Arithmetic Intermediate Representation of an Expression) and PIRExp (for Painter Intermediate Representation of an Expression). Note that AIRExp could also be further broken down into integer and fractional expressions (you may implement this refinement if you choose). The class PIRStatement represents *HPL+* statements, and is the parent class of all *HPL+* statement intermediate representations. The parent class for the AIRExp, PIRExp and PIRStatement classes is called ASTNode. As given, it is little more than a placeholder class that provides a parent type (class) for all intermediate representations of HPL forms.

One important role that the ASTExp class serves is to organise expressions by the arity of their operators. Specifically, the classes ASTVar<*T*>, ASTUnaryExp<*T*>, and ASTBinaryExp<*T*> represent expressions that are, respectively: a variable reference, a unary operation, or a binary operation. The type parameter $T$ represents the type of expression that is represented, and it must be a subclass of ASTExp<*T*>. (Don't worry if the apparently self-referential type parameter confuses you, it takes a while to become comfortable with it, and you won't need to make up any new type relations of your own for this assignment). The benefit of this design of class hierarchy is that you do not need to have a visitor method for each operation that may be available in the language.

Ordinarily, you would have to include, in your visitor interface, a method for each operation (e.g. addition, subtraction, etc) that is explicitly captured by your grammar. Usually, the implementations of those methods differ only in how the operator is treated. To capture this pattern, the ASTExp$< T >$ class requires a visit method that accepts a generic visitor of type ASTVisitor$< E, S, T >$. This visitor must implement methods to handle all of the generic types of expressions (i.e. variable reference, unary and binary operations). The AIRVisitor$< S, T >$ interface has been completely specified, and the ArithEvaluator class is a complete implementation of it. By referencing their source files, you should be able to see the benefit of this approach. For example, observe how the one method for handling binary expressions takes the place of several methods, one for each operation.

Some of the ASTNode subclasses have already been implemented for you, such as ASTExp$< T >$, the entire AIRExp$< T >$ hierarchy of subclasses and PIRSequence. You will need to create representations for the remaining forms of HPL statements and expressions in order to build a working interpreter though.

## Syntax Related Problems

**Problem 1 [5]**
Complete the file named HPLLexer as an input for *jflex* (or *jflex*) so that the generated lexer (which should be in a file named HPLLexer.java) correctly tokenizes *HPL+* programs.

**Problem 2 [20]**
Complete the file named HPLParser as an input to *cup*, so that it can scan and parse *HPL+*programs. Some of the types of forms that you will have to worry about include: def-painter, paint, function call, frame, and argument lists.

In specifying your parser, you are not permitted to use the operator precedence forms provided by *cup*; you should, instead, create grammars that are unambiguous that also implement the correct operator precedences. The arithmetic operations have already been implemented for you; you are primarily responsible for the painter commands (and possibly the logic expressions).

Make sure that the classes generated from these specification files are called `HPLLexer.java` and `HPLParser.java`, respectively. Starter versions of these files have been provided for you.

When you have a correct grammar, you should be able to parse the example files that were presented (although they may not yet function as they ought).

**Problem 3 [10]**
Create all the remaining intermediate representations (subclasses of `PIRExp` and `PIRStatement`) necessary to build an abstract syntax tree (AST) representation for an *HPL+* program. Keep to the convention of naming `Painter` expressions with the prefix `PIR`. In particular, you will need to build intermediate representations for function definition.

## Traversing The Intermediate Representation

While traversing the intermediate representation, you will need to maintain context. Unlike the simpler algebraic interpreter that you have seen before now, the context for an *HPL+* evaluator is more complex than a single environment. Just think about the parameters given to functions, and you will see that we need to give consideration to the fact that there are numerical parameters and painter parameters available to the function simultaneously. One way to do this is to create a parent class for all runtime data types and store them all in the same environment. This makes sense if the different types can be combined to produce one of the types of the operands, or some other type. However, in *HPL+*, the data types are kept separate by the syntax. Non-painter typed expressions are confined to specific contexts, and therefore it is possible to manage all of the necessary state as separate components.

In other words, we can maintain an environment for each type of nameable object. In *HPL+*, this means one each for painters, numbers, and functions. In addition to the environments, we will also need to maintain the current frame, with respect to which `paint` statements must be interpreted.

**Problem 4 [10]**
Implement the interface `HPLContext` so that it can properly support an *HPL+* interpreter.

The `HPLVisitor` interface is intended to capture the functionality that must be supported by any object that is meant to traverse an AST for an HPL program. Naturally, the methods contained within this interface depend upon the intermediate representation classes that you have created (see description of the visitor design pattern in the text, if you are unclear on this).

The `HPLEvaluator` class is intended to be an *HPL+* interpreter. It should traverse a program's AST, passing down an instance of `HPLEnvironment` for evaluation of subexpressions, and combining the resulting `Painter` values in the appropriate manner for each node. It is nearly complete, but it is missing the implementation of the methods to handle function definition and invocation.

**Problem 5 [15]**
Based on the intermediate representations that you have created, complete the definition of the `HPLVisitor` interface. Complete the implementation of the HPL interpreter, `HPLEvaluator`, as a class that implements the `HPLVisitor` interface. In implementing function call and definition, you might want to look at the classes in the package `hpl.values`, particularly at the `HPLFunction` and `CompoundPainter` classes. You should also implement the method `HPLEvaluator.mkInitialContext()` to return an instance of your class that implements the `HPLContext` interface.

# HPL Extensions

**Problem 6 [15]**
[**Optional**] Extend *HPL+* to support the `if` statement. In order to accomplish this, you will have to expand the parser to handle logic expressions. You should implement this in the same style that the arithmetic expressions were handled.

When you are finished, you should be able to run recursive functions, and create some very intricate patterns with relatively simple *HPL+* code fragments. (Have fun!).

# Turning in your work

You should use *OurVLE* to turn in your work. You should zip up your entire project from the directory containing the `src` directory, maintaining the directory structure, and submit everything as a single **zip** file to the repository. (When I unzip your file, I should see at least 2 directories: `src`, `examples`, appear in the directory where I unzipped your file.) Please ensure that you have retained the directory structure, because automated tools may be used to extract your submission from the zip file, and it would be unfortunate if your submission failed to compile because you forgot to put the Java files in the correct directories.