Exercise Compiler Construction (7)

Hans de Nivelle

Due: 13.12.2011

In this week and in the next week, we are going to try to implement the type checking algorithm in the slides **typechecking.ps**. Download the file `intermediate2011.tar.gz` from the course homepage. It contains a class **tree.h**, which represents abstract syntax trees.
Trees are defined by the following recursive definition:

- If f is an identifier, c is a binary string, t is a tree type, then $f(c) : t$ is a tree.
- If f is an identifier, $t_1, \ldots, t_n$ are trees, c is a tree type, then $f(t_1, \ldots, t_n) : c$ is a tree.

Since the attribute of a tree can be pretty much everything, it is represented by a binary string, stored in an `std::vector<char>`. Only leaves (trees without subtrees) can have attributes.
Tree has a field **tt** that stores the type of the tree. If no type is given, by default the type of the tree is **unknown**.

1. In file **intermediate.cpp**, there is some code that
   (a) Constructs three tree types, **char**, **int** and **double**.
   (b) Constructs a few trees, **e**, **pi**, **one**, **two** and **three**.
   (c) Constructs a tree (**pi** $\times$ **zero**) + **two** and pretty prints it.

   Compile the code, and try to understand it. The classes are constructed in such a way that you never have to worry about memory management.

2. Our final goal is to implement the type checking algorithm on slides 15, etc. in **typechecking.ps**.

   Write a procedure that constructs a **list** of functions, See in **intermediate.cpp** how this done.

```
std::list< tree > functionlist( );
   // Should declare a few polymorphic functions on the
   // types int,char and double.
   // For example:
```

```
// + : func( double; double, double ).
// + : func( int; int, int ).
```

In practice, using a list of function declarations would be very inefficient, but it is fine for this exercise.

3. Write a procedure that constructs a list of possible conversions.

```
std::list< tree > conversions( );
   // For example conv_int_double : func( double, int ).
```

4. Write a function

```
std::list< tree >
possibleconversions( const std::list< tree > & conversionlist, tree t );
```

that constructs the list of all possible conversions of t. Make sure that the algorithm also terminates when circular conversions are possible.

I think that this was enough for today. The rest of the algorithm, we will do next week. We have to think about a suitable representation of $\prec$ as well.