Compiler project tasks — part 3, due Monday 1 October

The goal in this phase is to construct the abstract syntax tree corresponding to the concrete parse tree that the parser produces.

Just as context-free grammars are used to describe concrete syntax, we use a formalism to describe abstract syntax: ASDL. You will find out about ASDL at the web page `http://www.cs.virginia.edu/zephyr/asdl.html`. You will also download the ASDL tool suite, version 1.2.

If your implementation language is Java, C, C++, SML, or Haskell, then it is supported by the ASDL tool suite, version 1.2; you should use the tool *asdlGen* to generate data-structure definitions and functions to read and write the data structures to file.

Convert the parse tree into the abstract syntax tree formed with these data-structure definitions. (If you are using a variant of *yacc*, you will add actions to grammar productions.)

Produce an output file with the pickled abstract syntax tree of the program being compiled; for instance, if you are compiling `primes.m`, you should produce `primes.absynpkl`. (Note that you should still be producing files from the earlier phases, `primes.tokens` and `primes.parse`.) The pickle files can be used to compare the results among students writing their project in different languages: if the files are not equal, at least one of the implementations is buggy. However, since pickle files are binary, it may be difficult to tell what went wrong.

If your implementation language is not one of the above, you may either write your own pickling code for our source language by hand, or extend *asdlGen* to generate code in your implementation language.

Abstract syntax of the source language

```
--description of the abstract syntax of ModulaX
--in ASDL v.1.2

module ModulaX
{
   program = (module)

   module = (identifier, block)

   block = (declaration*, stmt*)

   declaration = DECLCONST (identifier, typ?, expr)
               | DECLTYPE (typedeclaration)
               | DECLTYPEREC (typedeclaration*)
               | DECLVAR (identifier*, typ?, expr?)
               | DECLPROC (identifier, signature, block)

   typedeclaration = (identifier, typ)

   signature = (formal*, typ?)

   formal = (identifier*, mode?, typ)

   mode = VALUE | VAR

```

```
    stmt = STMTBLOCK (block)
         | STMTASSIGN (expr lhs, expr rhs)
         | STMTCALL (proccall)
         | STMTEXIT
         | STMTEVAL (expr)
         | STMTFOR (identifier index, expr from, expr to, expr? by, stmt*)
         | STMTIF (expr condition, stmt* thenpart, elsifclause*, elseclause?)
         | STMTLOOP (stmt*)
         | STMTREAD (string format, expr*)
         | STMTREPEAT (stmt*, expr condition)
         | STMTRETURN (expr?)
         | STMTWHILE (expr condition, stmt*)
         | STMTWRITE (string format, expr*)

    elsifclause = (expr condition, stmt*)

    elseclause = (stmt*)

    typ = TYPARRAY (typ* indextypes, typ elementtype)
        | TYPRECORD (field*)
        | TYPSUBRANGE (expr lowbound, expr highbound)
        | TYPID (identifier)
        | TYPREF (typ)

    field = (identifier, typ)

    expr = EXPRBINARY (expr, binary, expr)
         | EXPRUNARY (unary, expr)
         | EXPRID (identifier)
         | EXPRINT (int)
         | EXPRREAL (real)
         | EXPRTEXT (string)
         | EXPRRECSEL (expr, identifier)
         | EXPRARRAYREF (expr, expr*)
         | EXPRPROCCALL (proccall)
         | EXPRDEREF (expr)
         | EXPRNEW (typ)
         | EXPRNIL (typ)

    proccall = (expr, expr*)

    binary = OR | AND | EQUAL | NEQUAL | LESS | GREATER | LE | GE
           | ADD | SUB | MUL | DIV | MOD

    unary = NOT | UPLUS | UMINUS

    real = (int mantissa, int exp)
}
```