Semester Project Proposal: SOOL Compiler

For our semester project, we aim to create a compiler for SOOL, a simple object-oriented language that is derived from Kim Bruce’s *Fundamentals of Object-Oriented Languages*. Our compiler would require the user to provide human-readable text as input. The input will be parsed, type-checked, and output as SPIM instructional code. Kim Bruce’s description of SOOL in Chapter 10 of *FOOL* is an informal and vague indication of what the language consists of. In this proposal, we include a more formal and concrete syntax that the user is required to use when writing their code. In the following sections we describe the various complexities related to our project.

**Source-to-source translation**

Our project will convert input programs written in an abbreviated SOOL syntax into SPIM instructions. Figure 10.3 in Bruce’s *FOOL* gives the following example of an input program:

```
Program PointExample;
    PointType = ObjectType {
        move: Integer X Integer -> Command;
        getx: Void -> Integer;
        gety: Void -> Integer
    };

    class Point {
        x: Integer = 0;
        y: Integer = 0;
        function move(dx:Integer,dy:Integer): Command is {
            x := x + dx;
            y := y + dy
        }
        function getx(): Integer is { return x }
        function gety(): Integer is { return y }
    };

    pt: PointType;
    {
        pt := new Point;
        pt <= move(3,2)
    }
```

This syntax is understandable and readable. We will use it as the syntax for the input programs and convert it to SPIM instructions.
Name and Type Analysis

For the scope of this project, we are aiming to implement SOOL as a statically-typed language with explicit type declarations. An entity for the program is defined when the user declares the identifier in a particular block. Later, if this identifier is used in any further computations, the compiler will make sure that it has been defined before it allows any computations on it. Therefore, our variable declarations and type declarations will be considered as defining occurrences of identifiers, and the scope of the identifier will be a C-style scope.

The abbreviated syntax of SOOL does not provide enough information to make proper judgements in type-checking. Hence, the abbreviated language will be converted to a formal AST by our compiler, which will supply the type information to make proper decisions and report errors.

Concrete Grammar

As mentioned earlier, the description of SOOL given in Bruce’s FOOL is vague and informal. The following concrete grammar gives a more formal description of the input syntax that the user is required to provide:

```plaintext
SOOL.con

Program : 'PROGRAM' id ';' Block .
Block : TypeDefs ConstDefs '{' Stmts 'return' Expr '}' / TypeDefs ConstDefs '{' Stmts '}' .
TypeDefs : TypeDef TypeDefs /
TypeDef : id '=' Type ';' .
ConstDefs : ConstDef ConstDefs /
ConstDef : id '::' Type ';' / ClassDef .
ClassDef : 'class' id '{' ClassVarDefs FunctionDefs'}' ';' .
ClassVarDefs : ClassVarDef ClassVarDefs /
ClassVarDef : id '::' Type '=' Expr ';' .
FunctionDefs : FunctionDef FunctionDefs /
FunctionDef : 'function' id '(' Formals ')' ':' Type 'is' '{' Block '}' .
Formals : FormalList /
Formal : id '::' Type .
Type : TypeConst / id / FuncType / ObjectType .
TypeConst : 'Integer' / 'Void' / 'Command' .
FuncType : ArgTypeList '->' Type .
ArgTypeList : Type / Type 'x' ArgTypeList .
ObjectType : 'ObjectType' '{' FuncTypeList '}' .
FuncTypeList : id '::' FuncType ';' FuncTypeList / id '::' FuncType .
Expr : id / Number / Nil / FuncCall / FunctionDef / ClassDef / 'new' Expr / MethodCall / InstAccess / SubclassDef .
Nil : 'nil' .
FuncCall : Expr '()' ActualList .
Actuals : ActualList /
ActualList : Expr ',' ActualList / Expr .
```
MethodCall :   Expr '!=' Expr .
SubclassDef :   'class' 'inherits' Expr 'modifies' Labels '{' 
   ClassVarDefs FunctionDefs '}' .
Labels :   LabelList / .
LabelList :   Id ',' LabelList / Id .
Stmts :   StmtList / .
StmtList :   Stmt ';' StmtList / Stmt .
Stmt :   Assignment / Conditional / Iteration .
Assignment :   Expr ':=' Expr .
Conditional :   'if' Expr 'then' '{' Stmts '}' else '{' Stmts'}' .
Iteration :   'while' Expr 'do' '{' Stmts'}' .

The concrete grammar defined above uses the two terminal values shown below in our gla file:

SOOL.gla

id : C_IDENTIFIER
Number : C_INTEGER

Structured output:

We will need to transform the shape of the abstract syntax tree from one representing the SOOL program to one representing the SPIM program. The compiler will produce structured output (the SPIM assembly instructions) using the SPIM specification.