SOOL: Simple Object-Oriented Language

Compiler Specification
CSCI 5525: Compiler Construction Tools

Developed by:
Shibani Basava
Timsy Bir

Date: 12/09/2005
# Table of Contents

1 Introduction

1.1 Formalism of the Syntactic Description

1.2 Formalism of the Semantically-Equivalent Symbols

1.3 Formalism of the Semantic Description

1.4 Formalism for Properties of Quantities

1.5 Formalism of the Output

2 Basic Symbols, Identifiers and Numbers

2.1 Identifiers

2.2 Numbers

2.3 Logical Values

3 Phrase Structures

3.1 Equivalencies

3.2 Properties of Quantities

4 Semantic Analysis

4.1 Types

4.2 Entities, Scopes and Computational Roles

4.3 Values and Types

4.4 SOOL Types

4.5 User-Defined Types

4.6 Function Types

4.7 Object Types

4.8 Definitions

4.9 Variable Definitions

4.10 Function and Method Definitions

4.11 Class Definitions

4.12 Expressions

4.12.1 Simple Expressions

4.12.2 Arithmetic Expressions

4.12.3 Boolean Expressions

4.12.4 Function and Method Calls

4.12.5 Class Instantiation

4.13 Statements

4.13.1 Assignment Statement

4.13.2 Conditional Statement

4.13.3 Iteration Statement

4.13.4 Nop Statement

4.13.5 Return Statement

4.14 Program and Main Block

5 Error Checking and Reporting

5.1 Name analysis errors

5.2 Type-checking errors

5.3 Implementation Decisions

6 SPIM Instructions

6.1 Expression Handling
1 Introduction
The following document details the specifications from which an analyzer can be generated for SOOL, a Simple Object-Oriented Language. SOOL is a theoretical language designed by Kim Bruce, Professor of CS at the Department of Computer Science in Williams College. We have structured and scoped the language specification in order to build an executable compiler that can be used for general purpose programming. We hope that the conventions and the design decisions we have undertaken will conform to Bruce’s description of the language given in his book, *Fundamentals of Object-Oriented Languages*.

From the specification module detailed in this write up, the Eli system can generate a functional SOOL compiler that reads programs written using SOOL syntax, parses it for verification, analyses the program for any name and type errors, and finally print the output of the program using SPIM machine instructions. The following sections will provide detailed descriptions and code write ups for the SOOL analyzer.

1.1 Formalism of the Syntactic Description
This document will cover the syntax description of SOOL by covering the phrase structures and the basic symbols and identifiers. We will be using the standard ELI rule notations to describe the abstract syntax structured in the SOOL compiler. The basic identifiers will be given using regular expressions or ELI’s canned descriptions as they appear in the gla file. The phrase structure of the language will be covered using context-free grammar for SOOL using ELI’s type-con file. More about this can be found in Sections 2 and 3.

1.2 Formalism of the Semantically-Equivalent Symbols
Certain SOOL symbols that appear in the context-free grammar can be abstracted to more general nodes. These symbols are considered equivalent to the abstract nodes, and will be covered as equivalent symbols in Section 3.1.

1.3 Formalism of the Semantic Description
We use an attribute grammar to describe the behavior of the program when a certain input has been parsed to a syntactically correct tree. Sections 4 will provide descriptions of the attribute computations. These attributes are also used for providing dependency order in the computations. We use ELI’s attribute grammar specification, LIDO, to give full descriptions of the attribute computations made in the SOOL compiler.

Along with the attribute grammar computations, we assign our symbols certain computational roles that allow them to conform to the normal standard for programming languages. These name analysis and type analysis computational roles also include scoping rules for binding variable identifiers to their definitions, provide error checking for applied occurrences of undefined variables, check for type-consistency with input statements, and describe operator roles for structures and routines.

---

1 Eli Canned Symbol Descriptions: [http://eli-project.sourceforge.net/elionline4.4/lex_2.html#SEC13](http://eli-project.sourceforge.net/elionline4.4/lex_2.html#SEC13)
1.4 Formalism for Properties of Quantities

Each entity in SOOL is represented internally as a definition table key and is given a list of properties that can be accessed using the definition key as the index. We use ELI’s type-pdl file to describe our properties. Section 3.2 covers the properties we are using for the SOOL compiler.

1.5 Formalism of the Output

We use a combination of SPIM tree transformation functions and PTG functions to output MIPS machine instructional code for the given input. Sections 6 cover the SPIM transformation functions that we use for the text segment of the program and the PTG node functions used for the data segment.

2 Basic Symbols, Identifiers and Numbers

The following are the terminal symbols that are identified as individual tokens by our lexical analyser. All input is broken into identifiable symbols, identifiers and numbers before their relationships are formed by the tree parser. The following sections describe the basic symbols that are identified in SOOL.

2.1 Identifiers

Identifiers are used as basic symbols to represent all identifiers and string values in the input.

\[
\text{id} : \text{C\_IDENTIFIER}
\]

C\_IDENTIFIER is a canned description for symbols in ELI. It is interpreted as following in regular expressions.

\[
\[$[a-zA-Z_] [a-zA-Z_0-9]* [\text{mkidn}]$
\]

As id is being used as an identifier for variables, types, functions and classes, it needs to appear in the string table only once. Therefore, we process the token as mkidn.2

2.2 Numbers

Numbers are used as basic symbols to represent all integer values in the input.

\[
\text{Number} : \text{C\_INTEGER} [\text{mkidn}]
\]

C\_INTEGER is a canned description for symbols in ELI. It is interpreted as following in regular expression.

2 [http://eli-project.sourceforge.net/elionline4.4/lex_1.html#SEC4](http://eli-project.sourceforge.net/elionline4.4/lex_1.html#SEC4)
As number is an integer value that will be used to display spim instructions we process the token as mkidn by putting the value in the string table.

### 2.3 Logical Values

Logical values are denoted by keywords of the language. For SOOL, the only logical values present in the language are *Booleans, Void, Nil* and *Nop* values.

**Booleans** =

```
Expr ::= 'true' / 'false' .
```

**Void** =

```
Void =
Expr ::= '('
```

**Nil** =

```
Nil =
Expr ::= 'nil'
```

**Nop** =

```
Stmt ::= 'nop'
```

### 3 Phrase Structures

The phrase structure of the SOOL syntax is given using a BNF notation. The following is the context-free syntax for SOOL.

```
Program   : 'PROGRAM' id ':' Block .
Block      : TypeDefs ConstDefs '{' Stmts '}' .
TypeDefs   : 'type' TypeDefList / .
TypeDefList : TypeDef TypeDefList / TypeDef .
TypeDef    : id ' =' Type ';' .
ConstDefs  : 'const' ConstDefList / .
ConstDefList : ConstDef ConstDefList / ConstDef .
ConstDef   : VariableDef / ClassDef / FunctionDef .
VariableDef : id ':' Type '=' Expr ';' .
ClassDef   : 'class' id ':' Type ClassBlock ';' /
            'class' id 'inherits' Inherits
            'modifies' Labels ':' Type ClassBlock .
FunctionDef : 'function' ProcIdDef Lambda .
Lambda     : ProcFormals ':' ReturnType 'is'
            FuncBlock .
ProcFormals : '()' Formals ')'.
FuncBlock   : Block .
ReturnType  : Type .
Inherits    : InheritsClassIdUse .
ClassBlock  : '{' DefList '}' .
DefList     : Def DefList / .
Def         : VariableDef / MethodDef .
MethodDef   : 'function' MethodIdUse ProcFormals ':'
            ReturnType 'is' FuncBlock .
Type        : FuncType / PrimitiveType .
PrimitiveType : id / ObjectType / TypeConst .
TypeConst   : 'INTEGER' / 'VOID' / 'COMMAND' /
'B0OLEAN' .
```
3.1 Equivalencies

Certain symbols in the context free grammar are considered equivalent to certain symbols in the abstract grammar. These are shown below using ELI’s map file notation.

```
MAPSYM
Type ::= PrimitiveType TypeConst ObjectType .
Expr ::= ExprFactor ExprTerm FuncCall MethodCall InstAccess ExprConsts Factor Term ClassInst .
Stmt ::= Assignment Conditional Iteration Return .
```

3.2 Properties of Quantities

We give the entities in our language certain properties that can be used throughout the syntax tree for computations. The following are the properties defined for the SOOL entities using the property definition language of ELI.

```
/*Primitive types*/
intType -> IsType = {1};
```
semantic analysis

The semantics of SOOL are determined using a number of type-checking rules and attribute computations related to different symbols. SOOL is a language primarily made up of Types, Definitions, Expressions and Statements. The following sections will describe the scoping rules associated with different symbols, discuss the different entities and types in the language and, give the type checking rules for Statements and Expressions.

4.1 Types

All numbers are of type integer. In the logical values, boolean logical values are of type Boolean, Void is of type void, Nil is of type nil (which for SOOL is the bottom of the type hierarchy), and Nop is of type command.
RULE NilExpr : Expr ::= 'nil'

COMPUTE
    Expr.Type = nilType;
    PrimaryContext(Expr,nilType);
END;

RULE VoidExpr : Expr ::= '()'

COMPUTE
    Expr.Type = voidType;
    PrimaryContext(Expr,voidType);
END;

RULE NopStatement : Stmt ::= 'nop'

COMPUTE
    Stmt.Type = commandType;
END;

4.2 Entities, Scopes and Computational Roles
The following are the kind of entities that exist in the SOOL compiler: variables, types, procedures, classes. They are represented using identifier symbols as shown below.

Identifiers =
RULE Variable_ID_DEF : VarIdDef ::= id END;
RULE Type_ID_DEF : TypeIdDef ::= id END;
RULE Type_ID_USE : TypeIdUse ::= id END;
RULE Variable_ID_USE : VarIdUse ::= id END;
RULE Procedure_ID_DEF : ProcIdDef ::= id END;
RULE Class_ID_DEF : ClassIdDef ::= id END;
RULE Class_ID_USE : ClassIdUse ::= id END;
RULE Procedure_ID_USE : ProcIdUse ::= id END;
RULE Method_ID_DEF : MethodIdDef ::= id END;
RULE Method_ID_USE : MethodIdUse ::= id END;
RULE Inherit_Class_ID_USE : InheritsClassIdUse ::= id END;

Each of the entity shown above inherits certain computational roles that allow the analyzer to assign properties to the identifiers and provide error reporting if the role is violated. These computational roles are shown below as symbols.

;/*-----------------------------------------------*/
/*Generic name analysis and type analysis roles*/
;/*-----------------------------------------------*/

/*Name analysis role for defining occurrences*/
CLASS SYMBOL IdDef_Name INHERITS IdDefScope, ChkUnique
    COMPUTE
        SYNT.Sym = TERM;
END;
/*Name analysis role for applied occurrences*/
CLASS SYMBOL IdUse_Name INHERITS IdUseEnv, ChkIdUse COMPUTE
   SYNT.Sym = TERM;
END;

/*Type analysis role for the defining occurrence of a user-defined type*/
CLASS SYMBOL UserTypeDef_Type INHERITS TypeDefDefId, ChkTypeDefDefId END;

/*Type analysis role for an applied occurrence of a user-defined type*/
CLASS SYMBOL UserTypeUse_Type INHERITS TypeDefUseId, ChkTypeDefUseId END;

/*Type analysis role for defining occurrences of other identifiers*/
CLASS SYMBOL IdDef_Type INHERITS TypedDefId END;

/*Type analysis role for applied occurrences of other identifiers*/
CLASS SYMBOL IdUse_Type INHERITS TypedUseId, ChkTypedUseId END;

CLASS SYMBOL TypeIdUse_Name INHERITS IdDefScope COMPUTE
   SYNT.Sym = TERM;
END;

/*========================================================================
* Identifier symbols
/*========================================================================

/*Symbols representing defining occurrences*/
TREE SYMBOL TypeIdDef INHERITS IdDef_Name, UserTypeDef_Type END;

TREE SYMBOL VarIdDef INHERITS IdDef_Name, IdDef_Type END;
TREE SYMBOL ProcIdDef INHERITS IdDefScope, IdDef_Type,
   ChkProcUnique COMPUTE
   SYNT.Sym = TERM;
END;

TREE SYMBOL ClassIdDef INHERITS IdDefScope, ChkClassUnique,
   IdDef_Type COMPUTE
   SYNT.Sym = TERM;
END;

TREE SYMBOL MethodIdDef INHERITS IdDefScope, IdDef_Type,
   ChkMethodUnique COMPUTE
   SYNT.Sym = TERM;
END;

/*Symbols representing applied occurrences*/
TREE SYMBOL TypeIdUse INHERITS TypeIdUse_Name,
   UserTypeUse_Type END;
The following are some general check rules that provide error reporting for violating definition rules.

/* Name analysis role that ensures newly defined vars are unique */
/* Checks for unique identifiers */
CLASS SYMBOL ChkUnique INHERITS Count, TotalCnt COMPUTE
  IF (GT (THIS.TotalCnt, 1),
    message (ERROR, CatStrInd("identifier is multiply defined: ",
                              THIS.Sym), 0, COORDREF));
END;

/* Checks for unique procedures */
CLASS SYMBOL ChkProcUnique INHERITS Count, TotalCnt COMPUTE
  IF (GT (THIS.TotalCnt, 1),
    message (ERROR, CatStrInd("procedure is multiply defined: ",
                               THIS.Sym), 0, COORDREF));
END;

/* Checks for unique class names */
CLASS SYMBOL ChkClassUnique INHERITS Count, TotalCnt COMPUTE
  IF (GT (THIS.TotalCnt, 1),
    message (ERROR, CatStrInd("class is multiply defined: ",
                              THIS.Sym), 0, COORDREF));
END;

/* Checks for unique class usage */
CLASS SYMBOL ChkClassIdUse INHERITS TotalCnt COMPUTE
  IF (EQ (THIS.TotalCnt, 0),
Other than the identifiers, we also assign computational roles to determine scopes for functions, classes and blocks.

/*Root scope*/
TREE SYMBOL Program INHERITS RootScope END;

/*Range scopes*/
TREE SYMBOL Block INHERITS RangeScope COMPUTE
    INH.IND = NoKey;
END;
TREE SYMBOL CondStmts INHERITS RangeScope END;
TREE SYMBOL MethodDef INHERITS TypeDenotation, RangeScope END;
TREE SYMBOL Lambda INHERITS ProcedureDenotation,
    TypeDenotation, RangeScope END;
TREE SYMBOL ClassBlock INHERITS RangeScopeProp END;
TREE SYMBOL FuncTypeList INHERITS RangeScopeProp END;

4.3 Values and Types
For the scope of this project we have limited SOOL to only contain integer and boolean values. Valid integers are all non-negative numbers, while the keywords true and false are considered to be boolean representations. SOOL also consists of user defined types and object types.

4.4 SOOL Types
Besides integers and Booleans, SOOL contains Void, Command and Nil type as pre-defined types.

RULE IntegerType : Type ::= 'INTEGER'
    COMPUTE
        Type.Type = intType;
RULE VoidType  : Type ::= 'VOID'
    COMPUTE
        Type.Type = voidType;
    END;
RULE CommandType : Type ::= 'COMMAND'
    COMPUTE
        Type.Type = commandType;
    END;
RULE BooleanType : Type ::= 'BOOLEAN'
    COMPUTE
        Type.Type = booleanType;
    END;
TREE SYMBOL Type
    COMPUTE
        INH.ScopeInd = NoKey;
        SYNT.ScopeKey = NoKey;
    END;

4.5 User-Defined Types
SOOL also allows the user to define their own types by setting an identifier against a pre-defined type, or newly-defined type such as function type or an object type.

RULE UserDefType : TypeDef ::= TypeIdDef '=' Type ';'
    COMPUTE
        TypeIdDef.Type = Type.Type;
        Type.ScopeInd = TypeIdDef.Key;
    END;
RULE UserType  : Type ::= TypeIdUse
    COMPUTE
        Type.Type = TypeIdUse.Type;
        Type.ScopeKey = TypeIdUse.Key;
    END;

4.6 Function Types
Function Types are mainly used for setting up a defining occurrence of a method in an Object type definition. The arguments and return type specified in the function type is used to create a procedure denotation for the method key.

RULE FunctionTypeDecl : Type ::= FuncType
    COMPUTE
        Type.Type = FuncType.Type;
    END;
RULE FunctionTypeDef   : FuncType ::= ArgTypes '->'
    RETURN
        ReturnType
    COMPUTE
        ProcOperN(FuncType,ArgTypes,ReturnType.Type,FuncType.IND);
        ResetReturnType(FuncType.IND, ReturnType.Type)
        <- ReturnType.Type;
    END;
4.7 Object Types

SOOL Object Types are used as an interface definition for classes. No class can be defined unless it implements an interface. The defining occurrences of the class methods are stored in the object types while the classes implementing then define an applied occurrence while implementing them. In this manner a class would not be able to implement a method that hasn’t been defined in an object type. Also we have given Object types a Range Scope Prop so that when we call methods from a variable of the object type, it looks into the scope of the object type to determine if a method exists or not and what the method type is.

RULE ObjectTypeDef : Type ::= 'ObjectType' '{' FuncTypeList '}'
    COMPUTE
        Type.Type = objectType;
        FuncTypeList.ScopeKey = Type.ScopeInd;
    END;

RULE RuleArgList : ArgTypeList LISTOF ArgType END;
RULE RuleArgType : ArgType ::= Type
    COMPUTE
        ArgType.Type = Type.Type;
    END;
RULE RuleFuncTypeList : FuncTypeList LISTOF FuncTypeElm END;
RULE FunctionEleType : FuncTypeElm ::= MethodIdDef ':' FuncType
    COMPUTE
        FuncType.IND = MethodIdDef.Key;
        FuncTypeElm.IND = MethodIdDef.Key;
        FuncTypeElm.Type = FuncType.Type;
    END;
TREE SYMBOL FuncTypeList INHERITS RangeScopeProp END;
TREE SYMBOL Type
    COMPUTE
        INH.ScopeInd = NoKey;
        SYNT.ScopeKey = NoKey;
    END;
TREE SYMBOL FuncType INHERITS ProcedureDenotation,
    TypeDenotation
    COMPUTE
        INH.IND = NoKey;
    END;
TREE SYMBOL ArgType INHERITS ParameterListElem END;
TREE SYMBOL ArgTypes INHERITS ParameterListRoot END;
TREE SYMBOL FuncTypeElm INHERITS TypedDefinition END;
TREE SYMBOL FuncTypeElm INHERITS TypeDenotation END;
4.8 Definitions
SOOL includes the following types of definitions: variable definition, type definition, class definition and function definition.

```c
/*-----------------------------*/
/*Type Definitions*/
/*-----------------------------*/
RULE EmptyTypeDefs : TypeDefs ::= END;
RULE UserDefTypes  : TypeDefs ::= 'type' TypeDefList END;
RULE UserDefTypeList : TypeDefList LISTOF TypeDef END;

/*-----------------------------*/
/*Const Definitions*/
/*-----------------------------*/
RULE EmptyConstDefs : ConstDefs ::= END;
RULE RuleConstDefs : ConstDefs ::= 'const' ConstDefList END;
RULE RuleConstDefList : ConstDefList LISTOF ConstDef END;
RULE RuleConstVar       : ConstDef ::= VariableDef END;
RULE RuleConstFunc      : ConstDef ::= FunctionDef END;
RULE RuleConstClass     : ConstDef ::= ClassDef END;
```

4.9 Variable Definitions
Computation roles are assigned to allow ELI to recognize variable definitions and type definitions.

```c
TREE SYMBOL VariableDef INHERITS TypedDefinition END;
TREE SYMBOL TypeDef INHERITS TypeDenotation END;
RULE RuleVarDef         : VariableDef ::= VarIdDef ':' Type '=' Expr ';'
  COMPUTE
    VariableDef.Type = Type.Type;
    Expr.Required = Type.Type
    <- VariableDef.Type;
    ResetAssociatedType(VarIdDef.Key, Type.ScopeKey);
END;
```

TypeDefs have been defined under User-Defined Types in Section 4.5.

4.10 Function and Method Definitions
Similarly, procedures also are assigned computational roles to recognize them as procedure denotations. There are two kinds of procedures implemented for SOOL, simple routines that exist outside of any class and in the global scope and procedure definitions that exist only inside a class to determine the class’ behavior. For the purpose of distinguishing them in our compiler we have made simple routines as procedures that have their defining occurrence when they are implemented called Functions, while
methods are defined as a prototype in an Object type and is latter implemented inside a class definition.

```c
/*-------------------------------------------------------*/
/*Function definition*/
/*-------------------------------------------------------*/
ATTR Order: VOID;
ATTR Sym: int;

RULE RuleFuncDef : FunctionDef ::= 'function' ProcIdDef
Lambda
    COMPUTE
        /*lambda inherits type,proceduredentotation
        prociddef inherits typedefid */
        FunctionDef.IND = ProcIdDef.Key;
        FunctionDef.Type = Lambda.Type;
        Lambda.IND = ProcIdDef.Key;
        Lambda.Sym = ProcIdDef.Sym;
    END;
RULE LambdaRule : Lambda ::= ProcFormals ':'
ReturnType 'is' FuncBlock
    COMPUTE
        ProcOperN(Lambda,ProcFormals,ReturnType.Type,Lambda.I
ND);
        FuncBlock.IND = Lambda.IND;
        FuncBlock.Sym = Lambda.Sym;
    END;
RULE RuleProcFormals : ProcFormals ::= '(' Formals ')' END;
RULE RuleFuncBlock : FuncBlock ::= Block
    COMPUTE
        Block.IND = FuncBlock.IND;
        FuncBlock.IsReturn = Block.IsReturn;
        IF (EQ (FuncBlock.IsReturn, 0),
            message (ERROR,CatStrInd("procedure has
no return: ",
                FuncBlock.Sym), 0, COORDREF))
            <- FuncBlock.IsReturn;
    END;
RULE RuleReturnType : ReturnType ::= Type
    COMPUTE
        ReturnType.Type = Type.Type;
    END;
RULE FormalList : Formals LISTOF Formal END;
RULE FormalRule : Formal ::= VarIdDef ':' Type
    COMPUTE
        Formal.Type = Type.Type;
    END;
/*-------------------------------------------------------*/
/* Method Definition */
/*-------------------------------------------------------*/
```
4.11 Class Definitions

Classes are defined as a list of Variable definitions and Method Definitions in a Range Scope prop. Originally in SOOL a class is related to an Object Type using structural equivalence. However, for the scope of the project and to simplify method look up in the Object type we have changed the syntax to give the explicit type that a class implements. It allows us to pass the scope key that the object type uses from the type to the method identifiers.

In the subclass definitions, we limit SOOL to inherit from one class at a time, building a single inheritance hierarchy. This means, that we will be inheriting the scope of the super class along with defining the subclasses own scope.
4.12 Expressions

Expressions can evaluate to integers or Booleans.

/* Operators and expressions*/
/*TREE SYMBOL Expr INHERITS ExpressionSymbol END; */
TREE SYMBOL Expr INHERITS ExpressionSymbol
  COMPUTE
    SYNT.Type = NoKey;
    INH.Required = NoKey;
  END;

4.12.1 Simple Expressions

In SOOL, all integers, boolean keywords like ‘true’ and ‘false’, nil, void and an applied occurrence of an variable are considered as standalone expressions.

RULE NilExpr
  COMPUTE
    Expr ::= 'nil'
    Expr.Type = nilType;
4.12.2 Arithmetic Expressions

There are four arithmetic expressions, each of which take two integer operands and return an integer.

/*Addition expression*/
/* Type analysis: Required types for Expr[1], Expr[2], and Expr[3] are intType*/
RULE Addition : Expr ::= Expr '+' Expr
COMPUTE
  Expr[1].Type = intType;
  Expr[2].Required = intType;
  Expr[3].Required = intType;
END;

/*Subtraction expression*/
/* Type analysis: Required types for Expr[1], Expr[2], and Expr[3] are intType*/
RULE Subtraction : Expr ::= Expr '-' Expr
COMPUTE
  Expr[1].Type = intType;
  Expr[2].Required = intType;
  Expr[3].Required = intType;
END;

/*Multiplication expression*/
/* Type analysis: Required types for Expr[1], Expr[2], and Expr[3] are intType */
RULE Multiplication : Expr ::= Expr '*' Expr
    COMPUTE
        Expr[1].Type = intType;
        Expr[2].Required = intType;
        Expr[3].Required = intType;
    END;

/* Division expression */
/* Type analysis: Required types for Expr[1], Expr[2], and Expr[3] are intType */
RULE Division : Expr ::= Expr '/' Expr
    COMPUTE
        Expr[1].Type = intType;
        Expr[2].Required = intType;
        Expr[3].Required = intType;
    END;

4.12.3 Boolean Expressions

Boolean expressions evaluate to true or false. “True” and “false” are both Boolean expressions, and the less than operator takes two integer arguments and returns a Boolean.

/* Less than expression */
/* Type analysis: Required type for Expr[1] is booleanType 
   Required types for Expr[2] and Expr[3] are intType */
RULE LessThan : Expr ::= Expr '<' Expr
    COMPUTE
        Expr[1].Type = booleanType;
        Expr[2].Required = intType;
        Expr[3].Required = intType;
    END;

4.12.4 Function and Method Calls

As mentioned earlier, SOOL defines two kinds of routines, functions and methods. The syntax for calling each is different and performs different scope computations as well. For a function call, the compiler searches for the defining occurrence of the procedure in the current environment. On the other hand for a method call we provide the scope to look up method definition.

TREE SYMBOL Actuals INHERITS ArgumentListRoot END;
TREE SYMBOL Actual INHERITS ExpressionSymbol, ArgumentListElem END;

/* Function call */
RULE FuncCallExpr : Expr ::= ProcIdUse '(' Actuals ')' 
    COMPUTE
        CallContext(Expr,,Actuals);
Indication(ProcIdUse.Key);
IF(OR(BadIndication,BadOperator),
   message(ERROR, CatStrInd("not a procedure: ",
                ProcIdUse.Sym),
   0, COORDREF));
Expr.Type = GetReturnType(ProcIdUse.Key, NoKey);
END;

/*Method call - calls the method (Expr[2]) of an object
(Expr[3]) and produces a result (Expr[1])*/
/* Type analysis: Type of Expr[1] is the return value of
the method Required type of Expr[2] is a objectType */
RULE MethodCall  : Expr ::= VarIdUse '<='
MethodIdUse '(' Actuals ')' 
COMPUTE 
   MethodIdUse.ScopeKey =
   GetAssociatedType(VarIdUse.Key, NoKey);
   CallContext(Expr,,Actuals);
   Indication(MethodIdUse.Key);
   IF(OR(BadIndication,BadOperator),
      message(ERROR, CatStrInd("not a method:
         ",
         MethodIdUse.Sym),
      0,COORDREF))
<- MethodIdUse_SCOPEKEY;
Expr.Type = GetReturnType(MethodIdUse.Key,
NoKey)
<- MethodIdUse_SCOPEKEY;
END;

4.12.5 Class Instantiation
The instantiation of a new class is also considered an expression in SOOL.

/*Class instantiation*/
RULE ClassInstantiation : Expr ::= 'new' ClassIdUse
COMPUTE 
   PrimaryContext(Expr,ClassIdUse.Type);
   Expr.Type = ClassIdUse.Type;
END;

4.13 Statements
All statements in SOOL are of type command. Valid SOOL syntax contains assignment
statement, conditionals, iterations, return statement and nop statement.

TREE SYMBOL Stmt
COMPUTE 
   SYNT.IsReturn = 0;
END;
4.13.1 Assignment Statement
Assignment statement assigns the value of one expression to another. We check for type consistency by having the required type of the first expression be the same as the actual type of the expression we are assigning.

RULE Assignment : Stmt ::= Expr ':=' Expr
COMPUTE
   Expr[1].Required = Expr[2].Type;
   Stmt.Type = commandType;
END;

4.13.2 Conditional Statement
SOOL provides an if-else conditional statement as part of its syntax. We expect the expression for the conditional statement to evaluate to Boolean, while the statements under the then clause and the else clause to evaluate to type command.

RULE Conditional : Stmt ::= 'if' Expr 'then' '{' CondStmts '}' 'else' '{' CondStmts '}'
COMPUTE
   Expr.Required = booleanType;
   CondStmts[1].Required = commandType;
   CondStmts[2].Required = commandType;
   Stmt.Type = commandType;
   Stmt.IsReturn = ADD(CondStmts[1].IsReturn, CondStmts[2].IsReturn);
END;

RULE ConditionalStmts : CondStmts ::= Stmts
COMPUTE
   Stmts.Required = commandType;
   CondStmts.Type = commandType;
   CondStmts.IsReturn = Stmts.IsReturn;
END;

4.13.3 Iteration Statement
SOOL iterations are done using a while-do loop. The expression for the while statement is expected to evaluate to Boolean while the statements in the do clause are expected to evaluate to command.

RULE Iteration : Stmt ::= 'while' Expr 'do' '{' CondStmts '}'
COMPUTE
   Expr.Required = booleanType;
   CondStmts.Required = commandType;
   Stmt.Type = commandType;
   Stmt.IsReturn = CondStmts.IsReturn;
END;
4.13.4 Nop Statement
SOOL also provides a nop statement to signify no operation. It is similar in functionality to MIPS nop instruction.

RULE NopStatement : Stmt ::= 'nop'
COMPUTE
    Stmt.Type = commandType;
END;

4.13.5 Return Statement
Our syntax asks that every function should have a return statement, therefore we check for the proper return statement using attribute computations. We assign the attribute IsReturn for all Statements to equal zero except for the return statement. When we have a list of statements in a block we perform add operation on the statements in the list to determine that there is at least one return statement in the block for a function (the result should be greater or equal to 1). We do not worry about returning anything in the main block. Also, we make that the type of the expression we are returning is consistent with the return type of the function. We use properties to associate method and function keys to a return type and judge the expression type against that type.

RULE Return : Stmt ::= 'return' Expr
COMPUTE
    Expr.Required = GetReturnType(INCLUDING Block.IND, NoKey);
    Stmt.Type = commandType;
    Stmt.IsReturn = 1;
END;
RULE RuleStmts : Stmts LISTOF Stmt
COMPUTE
    Stmts.Type = commandType;
    Stmts.IsReturn = CONSTITUENTS Stmt.IsReturn WITH (int, ADD, IDENTICAL, ZERO);
END;

4.14 Program and Main Block
A program in SOOL starts with the name of the program and a main block. The main block can contain an arbitrary number of type definitions and constant definitions.

RULE ProgramRule : Program ::= 'PROGRAM' id ':'
Block $Target END;
RULE BlockRule : Block ::= TypeDefs ConstDefs '{'
Stmts '}'
COMPUTE
    Stmts.Required = commandType;
    Block.Type = commandType;
    Block.IsReturn = Stmts.IsReturn;
END;
5 **Error Checking and Reporting**
Before the actual execution of the code, our program is analyzed for all name based occurrences and type checks. Section 4 covers all the error reporting and related computational roles. The following sections provide the criteria and some the implementation decisions we made to cover the criteria.

5.1 **Name analysis errors**
This section covers the name analysis criteria for error checking and reporting:

- In a given environment, variables must be defined before they are used.
- Variables must not be defined more than once in the same scope.
- In a given environment, functions must be defined before they are used.
- Functions must not be defined more than once in the same scope.
- Any access of a method from a class must belong to the object type that the class implements.
- A class cannot inherit from a class that does not exist.
- A variable cannot be used outside of its scope.
- A class can only implement methods that exist in the object type that it is assigned to.
- A class cannot implement a method that hasn’t been defined in the object type that it is assigned to.
- All functions and methods must return a type.

5.2 **Type-checking errors**
This section covers the type analysis criteria for error checking and reporting:

- A variable must be initialized with a value of the correct type.
- All statements must be of type command.
- The RHS value in an assignment statement must have the same type as the LHS.
- The condition expression in if and while statements must be of type boolean.
- The result type of the < operation must be boolean.
- The result type of +, -, *, and / must be integer.
- The operand types of +, -, *, /, and < must be integer.
- The number of arguments and type of each argument in a function call must equal the number of formals and type of the each formal in the corresponding function definition.
- The number of arguments and type of each argument in a method call must equal the number of formals and type of the each formal in the corresponding method definition in object type.
- The type of the result returned by a function call must match the return type listed in the corresponding function definition.
• The type of the result returned by a method call must match the return type listed in the corresponding method definition.
• The type of the result returned by a function call must be acceptable in the context the function was called in. (For example, a function returning an integer cannot be placed on the RHS of an assignment statement that expects a boolean).
• The type of the result returned by a method call must be acceptable in the context the method was called in. (For example, a method returning an integer cannot be placed on the RHS of an assignment statement that expects a boolean).

5.3 Implementation Decisions
In order to meet the criteria given in Section 5.1 and 5.2, a number of implementation decisions where made for symbol computations. This section summarizes these decisions (they have been covered in previous sections).

• VarIdDef, ProcIdDef, MethodIdDef and ClassIdDef are identifiers representing unique defining occurrences.
• VarIdUse, ProcIdUse, MethodIdUse, InheritedClassIdUse and ClassIdUse are identifiers representing applied occurrences that are pre-defined.
• The Program has the top level scope.
• Each block has its own scope.
• Each function block has its own scope.
• Each object type has a RangeScopeProp with each MethodIdUse accessing the scope using ScopeKey. The MethodIdUse is IdUseScopeProp.
• Lambda function is defined as procedure denotation that is bounded to the formals and the return type.
• FunctionDefs have a RangeSequence where the FunctionBlock is binded with the formals and the Return type.
• MethodDefs are defined as procedure denotation and are bounded to the formal type list and the return type given to it in the object type.
• Each class definition explicitly implements an object type.
• The object type contains access to the scope which it defines.
• Each method in the class is a MethodIdUse and has to be defined in the object type before it can be implemented in the class. The implementation in the class is not a procedure denotation.
• Predefined types are INTEGER, VOID, BOOLEAN and COMMAND.

6 SPIM Instructions
Each program of SOOL will be analyzed and type-checked by the SOOL analyzer and will be transformed into SPIM instruction tree that can outputted using PTG functions to console. The following sections cover the transformation performed for Expressions, Statements and Definitions.

TREE SYMBOL Expr INHERITS ExpressionSymbol
COMPUTE
INH.lv = 0;
6.1 Expression Handling

All expressions in SOOL will be transformed as IntReg nodes for SPIM tree.

/*Nil*/
RULE NilExpr : Expr ::= 'nil'
COMPUTE
    Expr.spim = NoItems();
END;

/*Void - empty argument or formal set*/
/*COMMENT: Do we need this? Or can we define a rule
for an empty argumentList or an empty formalList?*/
RULE VoidExpr : Expr ::= '()'  
COMPUTE
    Expr.spim = NoItems();
END;

/*Variable use*/
/*Spim transformation: We need to either get the value or
the address, depending on context. */
ATTR VarSpim: NODEPTR;
RULE VariableExpr : Expr ::= VarIdUse
COMPUTE
    /*If an address is required, then get the address,
    otherwise get the value*/
    .VarSpim = based_memory(symbolic_address(VarIdUse.Key),0);
    Expr.spim = IF(EQ(Expr.lv,1), .VarSpim,
                    lw_iLoad(.VarSpim));
END;

RULE NumberExpr : Expr ::= Number
COMPUTE
    Expr.spim = li_iRegi(Number);
    Expr.data = PTGStringConvert(StringTable(Number));
END;

ATTR boolIndex : int;
RULE TrueExpr : Expr ::= 'true'
COMPUTE
    .boolIndex = stostr("1", 1);
    Expr.spim = li_iRegi(.boolIndex);
    Expr.data = PTGStringConvert(StringTable(.boolIndex));
END;
RULE FalseExpr : Expr ::= 'false'
COMPUTE
  .boolIndex = stod(str("0", 1));
  Expr.spim = li_iRegi(.boolIndex);
  Expr.data = PTGStringConvert(StringTable(.boolIndex));
END;

/*Function call*/
RULE FuncCallExpr : Expr ::= ProcIdUse '()' Actuals ')
COMPUTE
  Expr.spim = NoItems(); /*CHANGE ME!!!!!!*/
END;

/*Class instantiation*/
RULE ClassInstantiation : Expr ::= 'new' ClassIdUse
COMPUTE
  Expr.spim = NoItems(); /*CHANGE ME!!!!!!!*/
END;

/*Method call - calls the method (Expr[2]) of an object (Expr[3]) and produces a result (Expr[1])*/
RULE MethodCall : Expr ::= VarIdUse '<=' MethodIdUse '()' Actuals ')
COMPUTE
  Expr.spim = NoItems(); /*CHANGE ME!!!!!!*/
END;

/*Instance variable access - gets the instance variable (Expr[3]) of an object (Expr[2]) and produces a result (Expr[1])*/
RULE InstanceAccess : Expr ::= Expr '.' Expr
COMPUTE
  Expr[1].spim = NoItems(); /*CHANGE ME!!!!!!!*/
END;

/*Addition expression*/
RULE Addition : Expr ::= Expr '+' Expr
COMPUTE
  Expr[1].spim = add_iRegrr(Expr[2].spim,
  Expr[3].spim);
END;

/*Subtraction expression*/
RULE Subtraction : Expr ::= Expr '-' Expr
COMPUTE
  Expr[1].spim = sub_iRegrr(Expr[2].spim,
  Expr[3].spim);
END;

/*Multiplication expression*/
RULE Multiplication : Expr ::= Expr '*' Expr
COMPUTE
  Expr[1].spim = mul_iRegrr(Expr[2].spim,
  Expr[3].spim);
END;
/*Division expression*/
RULE Division : Expr ::= Expr '/' Expr
    COMPUTE
        Expr[1].spim = div_iRegrr(Expr[2].spim, Expr[3].spim);
    END;

/*Less than expression*/
RULE LessThan : Expr ::= Expr '<' Expr
    COMPUTE
        Expr[1].spim = slt_iRegrr(Expr[2].spim,Expr[3].spim);
    END;

6.2 Statement Handling
All statements in SOOL will be transformed as Item nodes for SPIM tree.

/*List of statements*/
RULE RuleStmts : Stmts LISTOF Stmt
    COMPUTE
        Stmts.spim = CONSTITUENTS Stmt.spim
            WITH (NODEPTR, AddOne, IDENTICAL, NoItems);
    END;

/*No-operation statement*/
RULE NopStatement : Stmt ::= 'nop'
    COMPUTE
        /*Nop doesn't do anything, we don't need any
         machine instructions*/
        Stmt.spim = NoItems();
    END;

/*Assignment statement*/
/* Expr[1] is guaranteed to be an id because of the .con
 file.
 Type analysis: Required type for Expr[1] is the type
 of Expr[2]
 Spim transformation: Store value of Expr[2] in address
 of Expr[1]
 We must allow booleans to be stored in integer
 registers.
 */
RULE Assignment : Stmt ::= Expr '==' Expr
    COMPUTE
        Expr[1].lv = 1; /*The LHS Expression must be an
            address, not a value*/
            /*The RHS Expression must be a value,
                which it is because of the default
                symbol computation*/
        Stmt.spim = sw_iStore(Expr[2].spim,Expr[1].spim);
    END;
/*If statement*/
RULE Conditional : Stmt ::= 'if' Expr 'then' '{' CondStmts '}' 'else' '{' CondStmts '}'
COMPUTE
Stmt.spim = AddThree(
do_branch(beqz_Branchr(Expr.spim), ._elseLbl),
AddOne(CondStmts[1].spim, j_Jump(.endLbl)),
AddOne(define_label(.elseLbl), CondStmts[2].spim),
define_label(.endLbl));
._elseLbl = NewKey();
._endLbl = NewKey();
END;

ATTR _continueLbl: DefTableKey;

/*While statement*/
RULE Iteration : Stmt ::= 'while' Expr 'do' '{' CondStmts '}'
COMPUTE
Stmt.spim = AddOne(AddTwo(
define_label(_continueLbl),
do_branch(beqz_Branchr(Expr.spim), .endLbl),
AddOne(CondStmts.spim, j_Jump(_continueLbl))),
define_label(_endLbl));
._continueLbl = NewKey();
._endLbl = NewKey();
END;

/*The conditional statements in if and while clauses*/
RULE ConditionalStmts : CondStmts ::= Stmts
COMPUTE
CondStmts.spim = Stmts.spim;
END;

RULE Return : Stmt ::= 'return' Expr
COMPUTE
Stmt.spim = Expr.spim; /* CHANGE ME!!! */
END;

6.3 Definition Handling
The following cover the tree transformations for variable and function definitions in SPIM.

 retreated to next page
7 Test Cases

The following code segments are written in the SOOL source language and are input to our compiler. They were run using Noosa and designed to test various name and type analysis computations. Descriptions of the output are provided with each code segment. The following code has been separated into two segments, one for all reporting of error test cases, the other for SPIM instructions.
7.1 Error Testing

Test01 – Produces error because \( v \) is multiply defined.

```
PROGRAM test01:
    const
    v: INTEGER = 0;
    v: BOOLEAN = true;
    { 
        nop 
    }
```

Test02 – Produces errors because \( v \) is multiply defined and because \( w \) is undefined.

```
PROGRAM test02:
    const
    v: INTEGER = 0;
    v: BOOLEAN = true;
    { 
        return w 
    }
```

Test03 – Produces error because \( w \) is initialized with a boolean.

```
PROGRAM test03:
    const
    v: INTEGER = true;
    w: BOOLEAN = v;
    { 
        return w 
    }
```

Test04 – Does not produce an error.

```
PROGRAM test04:
    const
    v: INTEGER = 0;
    { return v }
```

Test05 – Produces error because a boolean is assigned to an integer variable.

```
PROGRAM test05:
    const
    v: INTEGER = 0;
    w: BOOLEAN = true;
    { v := w }
```

Test06 – Does not produce an error.

```
PROGRAM test06:
    const
    v: INTEGER = 0;
    w: INTEGER = 1;
    { v := w }
```

Test08 – Produces an error because the while expression is not a boolean.

```
PROGRAM test08:
    const
    i: INTEGER = 0;
    { while (i) do { i := (i + 1) } }
```
**Test10** – Does not produce an error.

```pascal
PROGRAM test10:
const
i: INTEGER = 0;
{ while (i < 2) do { i := (i + 1) } }
```

**Test11** – Produces an error because the if expression is not a boolean.

```pascal
PROGRAM test11:
const
i: INTEGER = 0;
{ if (i) then { i := (i - 1) } else { i := (i + 1) } }
```

**Test13** – Does not produce an error.

```pascal
PROGRAM test13:
const
i: INTEGER = 0;
{ if (i < 1) then { i := (i - 1) } else { i := (i + 1) } }
```

**Test15** – Does not produce an error.

```pascal
PROGRAM test15:
const
i: INTEGER = 0;
{ nop; nop; i := (i + 2); nop }
```

**Test16** – Produces an error because function $p$ is multiply defined.

```pascal
PROGRAM test16:
const
function p (i: INTEGER): INTEGER is { return 1 }
function p (i: BOOLEAN): BOOLEAN is { return true } { nop }
```

**Test17** – Produces an error because $p$ is not defined.

```pascal
PROGRAM test17:
const
int: INTEGER = 0;
{ int := p(1 < 3) }
```

**Test18** – Produces an error because $p$ is called with too many arguments.

```pascal
PROGRAM test18:
const
function p (i: INTEGER): INTEGER is { return 1 }
int: INTEGER = 0;
{ int := p(1,3) }
```

**Test19** – Produces an error because $p$ is called with an argument of an incorrect type.

```pascal
PROGRAM test19:
const
function p (i: INTEGER): INTEGER is { return 1 }
int: INTEGER = 0;
{ int := p(true) }
```

**Test20** – Does not produce an error.

```pascal
PROGRAM test20:
const
function p (i: INTEGER): INTEGER is { return 1 }
```
Test21 – Produces an error because $b$ is used incorrectly in $p$.

PROGRAM test21:
const function p (b: BOOLEAN): INTEGER is { return (b+b)}
int: INTEGER = 0; { int := p(true) }

Test22 – Does not produce an error.

PROGRAM test22:
const
function p (i: INTEGER): INTEGER is { return (i+i) }
int: INTEGER = 0;
{ int := p(1) }

Test23 – Produces an error because $p$ expects a return value but has none.

PROGRAM test23:
const
function p (i: INTEGER): INTEGER is { nop }
int: INTEGER = 0;
{ int := p(1) }

Test24 – Produces an error because the return value in $p$ is the incorrect type.

PROGRAM test24:
const
function p (i: INTEGER): INTEGER is { return true }
int: INTEGER = 0;
{ int := p(1) }

Test25 – Does not produce an error.

PROGRAM test25:
const function p (i: INTEGER): INTEGER is { return i }
int: INTEGER = 0; { int := p(1) }

Test26 – Does not produce an error.

PROGRAM test26:
const
function p (i: INTEGER): INTEGER is { return i }
b: BOOLEAN = true;
int: INTEGER = 0;
{ int := p(1); return b }

Test27 – Produces an error because the class implements an object that does not exist.

PROGRAM id :
type
t = ObjectType {
  m: INTEGER -> INTEGER ;
  n: INTEGER -> VOID
};
const
class fd : K {
  r: INTEGER = 0;
  function m (i: INTEGER): INTEGER is {
    return r
  }
}
function n (i: INTEGER): VOID is {
    return ()
}

nop

Test28 – Produces an error because the class implements an object that it is not structurally equal to.
PROGRAM id :
    type
t = ObjectType {
    m: INTEGER -> INTEGER
};
const
class fd : t {
    r: INTEGER = 0;
    function m (i: INTEGER): INTEGER is {
        return r
    }
    function n (i: INTEGER): VOID is {
        return ()
    }
}

Test29 – Does not produce an error
PROGRAM id :
    type
t = ObjectType {
    m: INTEGER -> INTEGER ;
    n: INTEGER -> VOID
};
const
class fd : t {
    x: INTEGER = 0;
    function m (i: INTEGER): INTEGER is {
        return x
    }
    function n (i: INTEGER): VOID is {
        return ()
    }
}

Test35 – Produces an error because a method is called that hasn’t been defined in the class.
PROGRAM id :
    type
t = ObjectType {
    m: INTEGER -> INTEGER ;
n: INTEGER -> VOID
};
const
class fd : t {
    r: INTEGER = 0;
    function m (i: INTEGER): INTEGER is {
        return r
    }
    function n (i: INTEGER): VOID is {
        return ()
    }
};
v: t = new fd;
{
    return v<=k(2)
}

7.2 Machine Instruction Generation

Test: Testing for assignment statement
PROGRAM test02:
    const
    v: INTEGER = 0;
    {
        v := 1
    }

Output:
<data>
\text{L1: .word 0}
</data>
\begin{verbatim}
.main:
    .globl main
    li $4,0
    sw $4,L1($0)
    li $4,1
    sw $4,L1($0)
</verbatim>

Test: Testing for iteration statements
PROGRAM prog5:
    const
    i1: INTEGER = 0;
    b1: BOOLEAN = true;
    {
        i1:= 45;
        while (0 < 2) do
            {
                i1 := i1+1
            }
        i1 := 3
    }

Output:
<data>
\text{L1: .word 0}
</data>
L2: .word 1

.text
main:
.globl main
li $4,0
sw $4,L1($0)
li $4,1
sw $4,L2($0)
li $4,45
sw $4,L1($0)
L3:
li $4,2
li $5,0
slt $4,$5,$4
beqz $4,L4
li $4,1
lw $5,L4($0)
add $4,$5,$4
sw $4,L1($0)
j L3
L4:
li $4,3
sw $4,L1($0)

Test: Testing for conditional statements

PROGRAM id :
const
i: INTEGER = 0;
{
    if (i < 1) then { i := (i - 1) } else { i := (i + 1) }
}

Output:

.L1: .word 0

.text
main:
.globl main
li $4,0
sw $4,L1($0)
li $4,1
lw $5,L1($0)
slt $4,$5,$4
beqz $4,L2
li $4,1
lw $5,L1($0)
sub $4,$5,$4
sw $4,L1($0)
j L3
L2:
li $4,1
lw $5,L1($0)
add $4,$5,$4
sw $4,L1($0)
L3:
Test: Testing for expression computations (add and subtract)

PROGRAM prog5:
const
  i1: INTEGER = 0;
i2: INTEGER = 22;
{
  i1 := 24;
i2 := ((i1 + 4) - 35)
}

Output:
.data
L1: .word 0
L2: .word 22
.text
main:
.globl main
li $4,0
sw $4,L1($0)
i $4,22
sw $4,L2($0)
i $4,24
sw $4,L1($0)
i $4,4
lw $5,L1($0)
add $4,$5,$4
li $5,35
sub $4,$4,$5
sw $4,L2($0)

Test: Testing for expression computations (mult and div)

PROGRAM prog5:
const
  i1: INTEGER = 0;
i2: INTEGER = 22;
{
  i1 := 24;
i2 := ((i1 / 4) * 35)
}

Output:
.data
L1: .word 0
L2: .word 22
.text
main:
.globl main
li $4,0
sw $4,L1($0)
i $4,22
sw $4,L2($0)
i $4,24
sw $4,L1($0)
i $4,4
lw $5,L1($0)
add $4,$5,$4
li $5,35
sub $4,$4,$5
sw $4,L2($0)
Appendix

SOOL.specs =

SOOL.con
SOOL.gla
SOOL.lido
SOOL.map
SOOL.pdl
SOOLChkRules.lido
SOOL.oil
SOOL.ctl
SOOLspim.lido
SOOL.ptg

$/Name/CScope.gnrc :inst
$/Name/CRangeSeq.gnrc :inst
$/Name/ScopeProp.gnrc : inst
$/Type/Typing.gnrc :inst
$/Tech/Strings.specs
$/Prop/OccCnt.gnrc:inst
$/Type/Expression.fw
$/Name/CInh.gnrc :inst

Spim%Spim.specs

SOOL.con =

Program : 'PROGRAM' id ':' Block .
Block : TypeDefs ConstDefs '{' Stmts '}' .
TypeDefs : 'type' TypeDefList / .
TypeDefList :TypeDef TypeDefList /TypeDef .
TypeDef : id '=' Type ';' .
ConstDefs : 'const' ConstDefList / .
ConstDefList :ConstDef ConstDefList / ConstDef .
ConstDef : VariableDef / ClassDef / FunctionDef .
VariableDef : id ':' Type '=' Expr ';' .
Type : FuncType / PrimitiveType .
PrimitiveType : id / ObjectType / TypeConst .
TypeConst : 'INTEGER' / 'VOID' / 'COMMAND' / 'BOOLEAN' .
FuncType : ArgTypeList '->' PrimitiveType .
ArgTypeList : PrimitiveType // ']' .
ObjectType : 'ObjectType' '{' FuncTypeList '}' .
FuncTypeList : [FuncTypeElm // ';' ] .
FuncTypeElm : id ':' FuncType .
Expr   : Factor / Factor '<' Factor .
Factor   : Term / Term '+' Factor / Term '-' Factor .
Term   : ExprConists / ExprFactor '*' Term / ExprConists '/'
        Term .
ExprConists : 'nil' / '()' / ExprFactor .
ExprFactor : ExprTerm / MethodCall / FuncCall / InstAccess /
            ClassInst .
ExprTerm   : id / Number / {' Expr '} .
FuncCall   : id {' Actuals '} .
MethodCall : id '<=' id {' Actuals '} .
Actuals   : Actual // ',' .
InstAccess   : id '.' id .
ClassInst : 'new' id .
Formals   : Formal // ' ', .
Formal   : id ':' Type .
Labels   : [Label // ' ',] .
Label   : id .
Stmts   : [Stmt // ';' ] .
Stmt   : 'nop' / Assignment / Conditional / Iteration /
        Return .
Assignment : id ':=' Expr .
Conditional : 'if' Expr 'then' '{' Stmts '} 'else' '{' Stmts '}'
        .
Iteration : 'while' Expr 'do' '{' Stmts '}'
Return : 'return' Expr .

SOOL.gla =

id      : C_IDENITIFIER
Number   : C_INTEGER [mkidn]

SOOL.map =

MAPSYM

Type ::= PrimitiveType TypeConst ObjectType .
Expr ::= ExprFactor ExprTerm FuncCall MethodCall InstAccess
        ExprConists Factor Term ClassInst .
Stmt ::= Assignment Conditional Iteration Return .

SOOL.pdl =

/*Primitive types*/
intType -> IsType = {1};
voidType -> IsType = {1};
commandType -> IsType = {1};
booleanType -> IsType = {1};
nilType -> IsType = {1};
objectType -> IsType = {1};
refType -> IsType = {1};
classType -> IsType = {1};

intKey -> Defer = {intType};
booleanKey -> Defer = {booleanType};
voidKey -> Defer = {voidType};
commandKey -> Defer = {commandType};
refKey -> Defer = {refType};
nilKey -> Defer = {nilType};
nopKey -> TypeOf = {commandType};

ReturnType: DefTableKey;
AssociatedType: DefTableKey;

mainKey;

**SOOL.ptg**

Directives:
$ "\n"

Directivei:
$ " " $ "\n"

Block:
"\t.data\n" $ "\n\t.text\n"

WordDef:
$ ":t.word " $

List:
$ "\n" $

StringConvert:
$ string

IntConvert:
$ int

**SOOL.oil**

CLASS Subclass(super) BEGIN
    COERCION widen(Subclass):super;
    COERCION upNil(nilType):Subclass;
END;

**SOOL.ctl**

ORDER : PARTITION EARLY ;

**SOOL.ChkRules.lido**

/*Name analysis role that ensures newly defined vars are unique*/
/* Checks for unique identifiers */
CLASS SYMBOL ChkUnique INHERITS Count, TotalCnt COMPUTE
    IF (GT (THIS.TotalCnt, 1),
message (ERROR, CatStrInd ('identifier is multiply defined: ',
    THIS.Sym), 0, COORDREF));
END;

/* Checks for unique procedures */
CLASS SYMBOL ChkProcUnique INHERITS Count, TotalCnt COMPUTE
IF (GT (THIS.TotalCnt, 1),
    message (ERROR, CatStrInd ('procedure is multiply defined: ',
        THIS.Sym), 0, COORDREF));
END;

/* Checks for unique class names */
CLASS SYMBOL ChkClassUnique INHERITS Count, TotalCnt COMPUTE
IF (GT (THIS.TotalCnt, 1),
    message (ERROR, CatStrInd ('class is multiply defined: ',
        THIS.Sym), 0, COORDREF));
END;

/* Checks for unique class usage */
CLASS SYMBOL ChkClassIdUse INHERITS TotalCnt COMPUTE
IF (EQ (THIS.TotalCnt, 0),
    message (ERROR, CatStrInd ('class is not defined: ',
        THIS.Sym), 0, COORDREF));
END;

/* Checks for unique class usage */
CLASS SYMBOL ChkMethodIdUse INHERITS TotalCnt COMPUTE
IF (EQ (THIS.TotalCnt, 0),
    message (ERROR, CatStrInd ('method is not defined: ',
        THIS.Sym), 0, COORDREF));
END;
CLASS SYMBOL ChkMethodUnique INHERITS Count, TotalCnt COMPUTE
IF (GT (THIS.TotalCnt, 1),
    message (ERROR, CatStrInd ('method is multiply defined: ',
        THIS.Sym), 0, COORDREF));
END;

SOOL.lido =

/*----------------------------------------*/
/*Attributes*/
/*----------------------------------------*/
ATTR Type: DefTableKey;
ATTR Required: DefTableKey;
ATTR IsReturn: int;
ATTR IND: DefTableKey;
ATTR ScopeInd: DefTableKey;
ATTR ScopeKey: DefTableKey;

/*----------------------------------------*/
/*Program*/
/*----------------------------------------*/
RULE ProgramRule : Program ::= 'PROGRAM' id ':' Block
END;
RULE BlockRule : Block ::= TypeDefs ConstDefs '{' Stmts '}'

COMPUTE
  Stmts.Required = commandType;
  Block.Type = commandType;
  Block.IsReturn = Stmts.IsReturn;
END;

/*----------------------------------------------------------*/
/*Type Definitions*/
/*----------------------------------------------------------*/
RULE EmptyTypeDefs : TypeDefs ::= END;
RULE UserDefTypes : TypeDefs ::= 'type' TypeDefList END;
RULE UserDefTypeList : TypeDefList LISTOF TypeDef END;
RULE UserDefType : TypeDef ::= TypeIdDef '=' Type ';'
COMPUTE
  TypeIdDef.Type = Type.Type;
  Type.ScopeInd = TypeIdDef.Key;
END;

/*----------------------------------------------------------*/
/*Const Definitions*/
/*----------------------------------------------------------*/
RULE EmptyConstDefs : ConstDefs ::= END;
RULE RuleConstDefs : ConstDefs ::= 'const' ConstDefList END;
RULE RuleConstDefList : ConstDefList LISTOF ConstDef END;
RULE RuleConstVar : ConstDef ::= VariableDef END;
RULE RuleConstFunc : ConstDef ::= FunctionDef END;
RULE RuleConstClass : ConstDef ::= ClassDef END;

/*----------------------------------------------------------*/
/*Variable definition*/
/*----------------------------------------------------------*/
RULE RuleVarDef : VariableDef ::= VarIdDef ':' Type '=' Expr ';'
COMPUTE
  VariableDef.Type = Type.Type;
  Expr.Required = Type.Type
  <- VariableDef.Type;
  ResetAssociatedType(VarIdDef.Key, Type.ScopeKey);
END;

/*----------------------------------------------------------*/
/*Function definition*/
/*----------------------------------------------------------*/
ATTR Order: VOID;
ATTR Sym: int;

RULE RuleFuncDef : FunctionDef ::= 'function' ProcIdDef Lambda
COMPUTE
  /*lambda inherits type,proceduredentotation
   prociddef inherits typedefid */
  FunctionDef.IND = ProcIdDef.Key;
  FunctionDef.Type = Lambda.Type;
Lambda.IND = ProcIdDef.Key;
Lambda.Sym = ProcIdDef.Sym;
END;
RULE LambdaRule : Lambda ::= ProcFormals ':' ReturnType
'is' FuncBlock
COMPUTE
ProcOperN(Lambda,ProcFormals,ReturnType.Type,Lambda.IND);
ResetReturnType(Lambda.IND, Return Type.Type);
FuncBlock.IND = Lambda.IND;
FuncBlock.Sym = Lambda.Sym;
END;
RULE RuleProcFormals : ProcFormals ::= '(' Formals ')' END;
RULE RuleFuncBlock : FuncBlock ::= Block
COMPUTE
Block.IND = FuncBlock.IND;
FuncBlock.IsReturn = Block.IsReturn;
IF (EQ (FuncBlock.IsReturn, 0),
message (ERROR,CatStrInd("procedure has no return: ",
FuncBlock.Sym), 0, COORDREF))
<- FuncBlock.IsReturn;
END;
RULE RuleReturnType : ReturnType ::= Type
COMPUTE
ReturnType.Type = Type.Type;
END;
RULE FormalList : Formals LISTOF Formal END;
RULE FormalRule : Formal ::= VarIdDef ':' Type
COMPUTE
Formal.Type = Type.Type;
END;

/*----------------------------------------------------------*/
/*Class and subclass definitions*/
/*----------------------------------------------------------*/
RULE RuleClassDef : ClassDef ::= 'class' ClassIdDef ':'
Type ClassBlock ';'
COMPUTE
ClassDef.Type = Type.Type;
ClassBlock.ScopeKey = ClassIdDef.Key;
ClassDef.ScopeInd = Type.ScopeKey;
IF(EQ(Type.ScopeKey, NoKey), message(ERROR,CatStrInd("Object type not reachable for class: ",
ClassIdDef.Sym), 0, COORDREF));
InstClass1(ClassDef, ClassIdDef.Key, objectType);
END;
RULE RuleSubclassDef : ClassDef ::= 'class' ClassIdDef '
'inherits'
'coordinates' Labels ':' Type
ClassBlock
COMPUTE
ClassDef.Type = Type.Type;
ClassBlock.ScopeKey = ClassIdDef.Key;
ClassDef.ScopeInd = Type.ScopeKey;
IF(EQ(Type.ScopeKey, NoKey), message(ERROR, 
    CatStrInd("Object type not reachable for
    class: ", 
        ClassIdDef.Sym), 0, COORDREF));
END;
RULE InheritedClassId  : Inherits ::= InheritedClassIdUse END;
RULE RuleClassBlock     : ClassBlock ::= '{' DefList '}' END;
RULE RuleDefList  : DefList LISTOF Def END;
RULE Definitions : Def ::= VariableDef END;
RULE MethodDefinition  : Def ::= MethodDef END;
RULE LabelList  : Labels LISTOF LabelElm END;
RULE LabelRule : LabelElm ::= ProcIdUse END;
RULE MethodDefs  : MethodDef ::= 'function' MethodIdUse ProcFormals ':' ReturnType 'is' FuncBlock
    COMPUTE
        FuncBlock.IND = MethodIdUse.Key;
        FuncBlock.Sym = MethodIdUse.Sym;
        MethodIdUse.ScopeKey = INCLUDING ClassDef.ScopeInd;
    IF(NOT(EQ(ReturnType.Type,GetReturnType(MethodIdUse.Key,NoKey))),
        message(ERROR, CatStrInd("return type does not match
    with the object type definition: ", 
        MethodIdUse.Sym), 0, COORDREF));
END;

/*----------------------------------------------------------*/
/*Types*/
/*----------------------------------------------------------*/
RULE IntegerType : Type ::= 'INTEGER'
    COMPUTE
        Type.Type = intType;
    END;
RULE VoidType  : Type ::= 'VOID'
    COMPUTE
        Type.Type = voidType;
    END;
RULE CommandType : Type ::= 'COMMAND'
    COMPUTE
        Type.Type = commandType;
    END;
RULE BooleanType : Type ::= 'BOOLEAN'
    COMPUTE
        Type.Type = booleanType;
    END;
RULE UserType  : Type ::= TypeIdUse
    COMPUTE
        Type.Type = TypeIdUse.Type;
        Type.ScopeKey = TypeIdUse.Key;
    END;
RULE FunctionTypeDecl : Type ::= FuncType 
    COMPUTE
        Type.Type = FuncType.Type;
    END;
RULE FunctionTypeDef   : FuncType ::= ArgTypes '->' ReturnType
COMPUTE
ProcOperN(FuncType,ArgTypes,ReturnType.Type,FuncType.IND);
    ResetReturnType(FuncType.IND, ReturnType.Type)
    <- ReturnType.Type;
END;
RULE RuleArgs   : ArgTypes ::= ArgTypeList END;
RULE RuleArgList        : ArgTypeList LISTOF ArgType END;
RULE RuleArgType  : ArgType ::= Type
COMPUTE
    ArgType.Type = Type.Type;
END;
RULE ObjectTypeDef : Type ::= 'ObjectType' '{' FuncTypeList
COMPUTE
    Type.Type = objectType;
    FuncTypeList.ScopeKey = Type.ScopeInd;
END;
RULE RuleFuncTypeList : FuncTypeList LISTOF FuncTypeElm END;
RULE FunctionEleType : FuncTypeElm ::= MethodIdDef ':'
FuncType
COMPUTE
    FuncType.IND = MethodIdDef.Key;
    FuncTypeElm.IND = MethodIdDef.Key;
    FuncTypeElm.Type = FuncType.Type;
END;

/*----------------------------------------------------------*/
/*Expressions*/
/*----------------------------------------------------------*/
RULE NilExpr  : Expr ::= 'nil'
COMPUTE
    Expr.Type = nilType;
    PrimaryContext(Expr,nilType);
END;
RULE VoidExpr  : Expr ::= '()'  
COMPUTE
    Expr.Type = voidType;
    PrimaryContext(Expr,voidType);
END;
RULE VariableExpr : Expr ::= VarIdUse
COMPUTE
    PrimaryContext(Expr, VarIdUse.Type);
    Expr.Type = VarIdUse.Type;
END;
RULE NumberExpr  : Expr ::= Number
COMPUTE
    Expr.Type = intType;
    PrimaryContext(Expr,intType);
END;
RULE TrueExpr  : Expr ::= 'true'
COMPUTE
END;
Expr.Type = booleanType;
PrimaryContext(Expr,booleanType);
END;
RULE FalseExpr : Expr ::= 'false'
COMPUTE
  Expr.Type = booleanType;
  PrimaryContext(Expr,booleanType);
END;

/*Function call*/
RULE FuncCallExpr : Expr ::= ProcIdUse '(' Actuals ')' 
COMPUTE
  CallContext(Expr,,Actuals);
  Indication(ProcIdUse.Key);
  IF(OR(BadIndication,BadOperator),
      message(ERROR, CatStrInd("not a procedure: ",
        ProcIdUse.Sym),
      0, COORDREF));
  Expr.Type = GetReturnType(ProcIdUse.Key, NoKey);
END;

/*Class instantiation*/
RULE ClassInstantiation : Expr ::= 'new' ClassIdUse 
COMPUTE
  PrimaryContext(Expr,ClassIdUse.Type);
  Expr.Type = ClassIdUse.Type;
END;

/*Method call - calls the method (Expr[2]) of an object (Expr[3])
and produces a result (Expr[1])*/
/* Type analysis: Type of Expr[1] is the return value of the method
   Required type of Expr[2] is a objectType */
RULE MethodCall : Expr ::= VarIdUse '<=' MethodIdUse '(' Actuals ')'
COMPUTE
  MethodIdUse.ScopeKey =
  GetAssociatedType(VarIdUse.Key, NoKey);
  CallContext(Expr,,Actuals);
  Indication(MethodIdUse.Key);
  IF(OR(BadIndication,BadOperator),
      message(ERROR, CatStrInd("not a method: ",
        MethodIdUse.Sym),
      0, COORDREF));
  <- MethodIdUse.ScopeKey;
  Expr.Type = GetReturnType(MethodIdUse.Key, NoKey)
  <- MethodIdUse.ScopeKey;
END;

/*Instance variable access - gets the instance variable (Expr[3])
of an object (Expr[2]) and produces a result (Expr[1])*/
/* Type analysis: Type of Expr[1] is the type of the variable (Expr[3]) Required type of Expr[2] is a objectType No required type for Expr[3] */
RULE InstanceAccess : Expr ::= Expr .' Expr
COMPUTE
    Expr[1].Type = Expr[3].Type;
    Expr[2].Required = objectType;
END;

/*Addition expression*/
/* Type analysis: Required types for Expr[1],Expr[2], and Expr[3] are intType*/
RULE Addition : Expr ::= Expr ' + ' Expr
COMPUTE
    Expr[1].Type = intType;
    Expr[2].Required = intType;
    Expr[3].Required = intType;
END;

/*Subtraction expression*/
/* Type analysis: Required types for Expr[1],Expr[2], and Expr[3] are intType*/
RULE Subtraction : Expr ::= Expr ' - ' Expr
COMPUTE
    Expr[1].Type = intType;
    Expr[2].Required = intType;
    Expr[3].Required = intType;
END;

/*Multiplication expression*/
/* Type analysis: Required types for Expr[1],Expr[2], and Expr[3] are intType*/
RULE Multiplication : Expr ::= Expr ' * ' Expr
COMPUTE
    Expr[1].Type = intType;
    Expr[2].Required = intType;
    Expr[3].Required = intType;
END;

/*Division expression*/
/* Type analysis: Required types for Expr[1],Expr[2], and Expr[3] are intType*/
RULE Division : Expr ::= Expr ' / ' Expr
COMPUTE
    Expr[1].Type = intType;
    Expr[2].Required = intType;
    Expr[3].Required = intType;
END;

/*Less than expression*/
/* Type analysis: Required type for Expr[1] is booleanType Required types for Expr[2] and Expr[3] are intType*/
RULE LessThan : Expr ::= Expr ' < ' Expr
COMPUTE
    Expr[1].Type = booleanType;
Expr[2].Required = intType;
Expr[3].Required = intType;
END;

/*Actuals list*/
/* Type analysis: this symbol must inherit ArgumentsListRoot*/
RULE RuleActuals : Actuals LISTOF Actual END;

/*Actual*/
/* Type analysis: The actual's type is the type of the expression. This symbol must inherit from ArgumentListElem*/
RULE ActualRule : Actual ::= Expr
    COMPUTE
        TransferContext(Actual,Expr);
    END;

/*----------------------------------------------------------*/
/*Statements*/
/*----------------------------------------------------------*/

RULE RuleStmts : Stmts LISTOF Stmt
    COMPUTE
        Stmts.Type = commandType;
        Stmts.IsReturn = CONSTITUENTS Stmt.IsReturn WITH (int, ADD, IDENTICAL, ZERO);
    END;
RULE NopStatement : Stmt ::= 'nop'
    COMPUTE
        Stmt.Type = commandType;
        Stmt.IsReturn = 0;
    END;

/*Assignment statement*/
/* Type analysis: Required type for Expr[1] is the type of Expr[2] */
RULE Assignment : Stmt ::= Expr ':=' Expr
    COMPUTE
        Expr[1].Required = Expr[2].Type;
        Stmt.Type = commandType;
        Stmt.IsReturn = 0;
    END;

/*If statement*/
/* Type analysis: Required type for Expr is booleanType */
RULE Conditional : Stmt ::= 'if' Expr 'then' '{' CondStmts '}' 'else' '{' CondStmts '}'
    COMPUTE
        Expr.Required = booleanType;
        CondStmts[1].Required = commandType;
        CondStmts[2].Required = commandType;
        Stmt.Type = commandType;
Stmt.IsReturn = ADD(CondStmts[1].IsReturn,
CondStmts[2].IsReturn);
END;

RULE ConditionalStmts : CondStmts ::= Stmts
COMPUTE
  Stmts.Required = commandType;
  CondStmts.Type = commandType;
  CondStmts.IsReturn = Stmts.IsReturn;
END;

/*While statement*/
/*   Type analysis: Required type for Expr is booleanType */
RULE Iteration : Stmt ::= 'while' Expr 'do' '{'
CondStmts '}'
COMPUTE
  Expr.Required = booleanType;
  CondStmts.Required = commandType;
  Stmt.Type = commandType;
  Stmt.IsReturn = CondStmts.IsReturn;
END;

RULE Return : Stmt ::= 'return' Expr
COMPUTE
  Expr.Required = GetReturnType(INCLUDING Block.IND,
NoKey);
  Stmt.Type = commandType;
  Stmt.IsReturn = 1;
END;

/*----------------------------------------------------------*/
/*Identifiers*/
/*----------------------------------------------------------*/
RULE Variable_ID_DEF : VarIdDef ::=  id END;
RULE Type_ID_DEF : TypeIdDef ::= id END;
RULE Type_ID_USE : TypeIdUse ::= id END;
RULE Variable_ID_USE : VarIdUse ::= id END;
RULE Procedure_ID_DEF : ProcIdDef ::= id END;
RULE Class_ID_DEF : ClassIdDef ::= id END;
RULE Class_ID_USE : ClassIdUse ::= id END;
RULE Procedure_ID_USE : ProcIdUse ::= id END;
RULE Method_ID_DEF : MethodIdDef ::= id END;
RULE Method_ID_USE : MethodIdUse ::= id END;
RULE Inherit_Class_ID_USE : InheritsClassIdUse ::= id END;
/*----------------------------------------------------------*/
/*Generic name analysis and type analysis roles*/
/*----------------------------------------------------------*/

/*Name analysis role for defining occurrences*/
CLASS SYMBOL IdDef_Name INHERITS IdDefScope, ChkUnique COMPUTE
  SYNT.Sym = TERM;
END;

/*Name analysis role for applied occurrences*/
CLASS SYMBOL IdUse_Name INHERITS IdUseEnv, ChkIdUse COMPUTE
SYNT.Sym = TERM;
END;

/*Type analysis role for the defining occurrence of a user-defined type*/
CLASS SYMBOL UserTypeDef_Type INHERITS TypeDefDefId,
ChkTypeDefDefId END;

/*Type analysis role for an applied occurrence of a user-defined type*/
CLASS SYMBOL UserTypeUse_Type INHERITS TypeDefUseId,
ChkTypeDefUseId END;

/*Type analysis role for defining occurrences of other identifiers*/
CLASS SYMBOL IdDef_Type INHERITS TypedDefId END;

/*Type analysis role for applied occurrences of other identifiers*/
CLASS SYMBOL IdUse_Type INHERITS TypedUseId, ChkTypedUseId END;

CLASS SYMBOL TypeIdUse_Name INHERITS IdDefScope COMPUTE
SYNT.Sym = TERM;
END;

/*---------------------------------------------*/
/*Identifier symbols*/
/*---------------------------------------------*/
/*Type definitions*/

TREE SYMBOL VariableDef INHERITS TypedDefinition END;
TREE SYMBOL Formal INHERITS TypedDefinition END;
TREE SYMBOL TypeDef INHERITS TypeDenotation END;
TREE SYMBOL FuncTypeElm INHERITS TypeDenotation END;
TREE SYMBOL FunctionDef INHERITS TypeDenotation END;
TREE SYMBOL MethodDef INHERITS TypeDenotation, RangeScope END;

/*Symbols representing defining occurrences*/
TREE SYMBOL TypeIdDef INHERITS IdDef_Name, UserTypeDef_Type END;
TREE SYMBOL VarIdDef INHERITS IdDef_Name, IdDef_Type END;
TREE SYMBOL ProcIdDef INHERITS IdDefScope, IdDef_Type,
ChkProcUnique COMPUTE
SYNT.Sym = TERM;
END;
TREE SYMBOL ClassIdDef INHERITS IdDefScope, ChkClassUnique,
IdDef_Type COMPUTE
SYNT.Sym = TERM;
END;
TREE SYMBOL MethodIdDef INHERITS IdDefScope, IdDef_Type,
ChkMethodUnique COMPUTE
SYNT.Sym = TERM;
END;

/*Symbols representing applied occurrences*/
TREE SYMBOL TypeIdUse INHERITS TypeIdUse_Name, UserTypeUse_Type END;
TREE SYMBOL VarIdUse INHERITS IdUse_Name, IdUse_Type END;
TREE SYMBOL ProcIdUse INHERITS IdUseEnv, IdUse_Type,
Operator Symbol COMPUTE
    SYNT.Sym = TERM;
END;

TREE SYMBOL ClassIdUse INHERITS IdUseEnv, ChkClassIdUse,
IdUse_Type COMPUTE
    SYNT.Sym = TERM;
END;
TREE SYMBOL InheritsClassIdUse INHERITS IdUseEnv, ChkClassIdUse
COMPUTE
    SYNT.Sym = TERM;
END;
TREE SYMBOL MethodIdUse INHERITS IdUseScopeProp, IdUse_Type,
OperatorSymbol, ChkMethodIdUse COMPUTE
    SYNT.Sym = TERM;
END;

/*----------------------------------------------------------*/
/*Scopes*/
/*----------------------------------------------------------*/

/*Root scope*/
TREE SYMBOL Program INHERITS RootScope END;

/*Range scopes*/
TREE SYMBOL Block INHERITS RangeScope
    COMPUTE
        INH.IND = NoKey;
    END;
TREE SYMBOL ClassBlock INHERITS RangeScopeProp END;
TREE SYMBOL CondStmts INHERITS RangeScope END;

/*Operators and expressions*/
/*TREE SYMBOL Expr INHERITS ExpressionSymbol END; */
TREE SYMBOL Expr INHERITS ExpressionSymbol
    COMPUTE
        SYNT.Type = NoKey;
        INH.Required = NoKey;
    END;

/*Procedures*/
TREE SYMBOL FunctionDef INHERITS TypedDefinition END;
TREE SYMBOL ClassDef INHERITS TypedDefinition, TypeDenotation
    END;
TREE SYMBOL Lambda INHERITS ProcedureDenotation, TypeDenotation,
    RangeScope END;
TREE SYMBOL ProcFormals INHERITS ParameterListRoot END;
TREE SYMBOL Formal INHERITS ParameterListElem END;
TREE SYMBOL Actuals INHERITS ArgumentListRoot END;
TREE SYMBOL Actual INHERITS ExpressionSymbol, ArgumentListElem
    END;
TREE SYMBOL FuncTypeList INHERITS RangeScopeProp END;
TREE SYMBOL Type
   COMPUTE
      INH.ScopeInd = NoKey;
      SYNT.ScopeKey = NoKey;
   END;
TREE SYMBOL FuncType INHERITS ProcedureDenotation, TypeDenotation
   COMPUTE
      INH.IND = NoKey;
   END;
TREE SYMBOL ArgType INHERITS ParameterListElem END;
TREE SYMBOL ArgTypes INHERITS ParameterListRoot END;
TREE SYMBOL FuncTypeElm INHERITS TypedDefinition END;

SOOL.spim.lido =

/*----------------------------------------------------------*/
/*Attributes*/
/*----------------------------------------------------------*/
ATTR spim: NODEPTR;      /*SPIM tree (or subtree) root*/
ATTR mlbl: DefTableKey;  /*Label for SPIM tree*/
ATTR lv: int;            /*Address of id required (as opposed to
value)/
/* ATTR value: int;         /*Actual values for expressions*/
ATTR data: PTGNode;      /*Data segment of the program*/
/*----------------------------------------------------------*/
/*Program*/
/*----------------------------------------------------------*/
ATTR progchain: VOID;
RULE SpimTarget  : SOOLTarget ::= Program $Target
   COMPUTE
      Target.GENTREE = Program.spim;
      SOOLTarget.progchain = PTGOut(Program.data);
      PTGOut(PTGMain(Target.Code)) <- SOOLTarget.progchain;
   END;
RULE ProgramRule        : Program ::= 'PROGRAM' id ':' Block
   COMPUTE
      Program.spim = Spim(define_routine(mainKey,
          Block.spim));
      Program.data = Block.data;
   END;
ATTR mainLbl: DefTableKey;
RULE BlockRule  : Block ::= TypeDefs ConstDefs '{' Stmts '}'
   COMPUTE
      Block.spim = AddTwo(TypeDefs.spim,
          ConstDefs.spim,
          Stmts.spim);
      Block.data = PTGBlock(ConstDefs.data);
   END;

/*----------------------------------------------------------*/
/* Type Definitions */
/*----------------------------------------------------------*/
RULE EmptyTypeDefs : TypeDefs ::= 
    COMPUTE
        TypeDefs.spim = NoItems();
    END;
RULE UserDefTypes : TypeDefs ::= 'type' TypeDefList 
    COMPUTE
        TypeDefs.spim = TypeDefList.spim;
    END;
RULE UserDefTypeList : TypeDefList LISTOF TypeDef 
    COMPUTE
        TypeDefList.spim = CONSTITUENTS TypeDef.spim
            WITH (NODEPTR, AddOne, IDENTICAL,
                NoItems);
    END;
RULE UserDefType : TypeDef ::= TypeIdDef '=' Type ';'
    COMPUTE
        TypeDef.spim = NoItems(); /*CHANGE ME!!!!!!*/
    END;
/*----------------------------------------------------------*/
/* Const Definitions */
/*----------------------------------------------------------*/
RULE EmptyConstDefs : ConstDefs ::= 
    COMPUTE
        ConstDefs.spim = NoItems();
        ConstDefs.data = PTGNULL;
    END;
RULE RuleConstDefs : ConstDefs ::= 'const' ConstDefList 
    COMPUTE
        ConstDefs.spim = ConstDefList.spim;
        ConstDefs.data = ConstDefList.data;
    END;
RULE RuleConstDefList : ConstDefList LISTOF ConstDef 
    COMPUTE
        ConstDefList.spim = CONSTITUENTS ConstDef.spim
            WITH (NODEPTR, AddOne, IDENTICAL,
                NoItems);
        ConstDefList.data = CONSTITUENTS ConstDef.data
            WITH (PTGNode, PTGList, IDENTICAL,
                PTGNull);
    END;
RULE RuleConstVar : ConstDef ::= VariableDef 
    COMPUTE
        ConstDef.spim = VariableDef.spim;
        ConstDef.data = VariableDef.data;
    END;
RULE RuleConstFunc : ConstDef ::= FunctionDef 
    END;
COMPUTE
    ConstDef.spim = FunctionDef.spim;
    ConstDef.data = PTGNULL; /*CHANGE ME!!!!!*/
END;

RULE RuleConstClass : ConstDef ::= ClassDef
COMPUTE
    ConstDef.spim = ClassDef.spim;
    ConstDef.data = PTGNULL; /*CHANGE ME!!!!!*/
END;

/*----------------------------------------------------------*/
/*Variable definition*/
/*----------------------------------------------------------*/
RULE RuleVarDef : VariableDef ::= VarIdDef ':=' Type '=' Expr ';
COMPUTE
    VariableDef.spim = sw_iStore(Expr.spim,
        based_memory(symbolic_address(VarIdDef.Key),0));
    VariableDef.data = PTGWordDef(VarIdDef.data,Expr.data);
END;

/*----------------------------------------------------------*/
/*Function definition*/
/*----------------------------------------------------------*/
ATTR continueLbl:DefTableKey;
RULE RuleFuncDef : FunctionDef ::= 'function' ProcIdDef Lambda
COMPUTE
    /* FunctionDef.spim = AddTwo(
        j_jump(.continueLbl),
        AddOne(define_label(ProcIdDef.Key),
            Lambda.spim),
        define_label(.continueLbl)
    );
    .continueLbl = NewKey(); */
    FunctionDef.spim = NoItems();
END;
RULE LambdaRule : Lambda ::= ProcFormals ':' ReturnType 'is'
FuncBlock
COMPUTE
    Lambda.spim = AddOne(ProcFormals.spim,
        FuncBlock.spim);
END;
RULE RuleProcFormals : ProcFormals ::= '( ' Formals ' )'
COMPUTE
    ProcFormals.spim = Formals.spim;
END;
RULE RuleFuncBlock : FuncBlock ::= Block
COMPUTE
    FuncBlock.spim = Block.spim;
END;
RULE FormalList : Formals LISTOF Formal
COMPUTE
    Formals.spim = CONSTITUENTS Formal.spim
WITH (NODEPTR, AddOne, IDENTICAL,
NoItems);
END;
RULE FormalRule : Formal ::= VarIdDef ':' Type
COMPUTE
  Formal.spim = NoItems(); /* Change Me!!!!*/
END;

/*----------------------------------------------------------*/
/*Class and subclass definitions*/
/*----------------------------------------------------------*/
RULE RuleClassDef : ClassDef ::= 'class' ClassIdDef ':'
  Type ClassBlock ';
  COMPUTE
    ClassDef.spim = NoItems(); /*CHANGE ME!!!!!!!*/
END;
RULE RuleSubclassDef : ClassDef ::= 'class' ClassIdDef
  'inherits' Inherits 'modifies' Labels ':' Type ClassBlock
  COMPUTE
    ClassDef.spim = NoItems(); /*CHANGE ME!!!!!!!*/
END;

/*----------------------------------------------------------*/
/*Expressions*/
/*----------------------------------------------------------*/
/*Nil*/
RULE NilExpr : Expr ::= 'nil'
  COMPUTE
    Expr.spim = NoItems();
END;

/*Void - empty argument or formal set*/
/*COMMENT: Do we need this? Or can we define a rule
  for an empty argumentList or an empty formalList?*/
RULE VoidExpr : Expr ::= '('
  COMPUTE
    Expr.spim = NoItems();
END;

/*Variable use*/
/*Spim transformation: We need to either get the value or the
  address, depending on context. */
ATTR VarSpim: NODEPTR;
RULE VariableExpr : Expr ::= VarIdUse
  COMPUTE
    /*If an address is required, then get the address,
      otherwise get the value*/
    .VarSpim = based_memory(symbolic_address(VarIdUse.Key),0);
    Expr.spim = IF(EQ(Expr.lv,1), .VarSpim, lw_iLoad(.VarSpim));
END;
RULE NumberExpr : Expr ::= Number
COMPUTE
    Expr.spim = li_iRegi(Number);
    Expr.data = PTGStringConvert(StringTable(Number));
END;

RULE TrueExpr : Expr ::= 'true'
COMPUTE
    Expr.spim = li_iRegi(IdnNumb(0,1));
    Expr.data = PTGIntConvert(1);
END;

RULE FalseExpr : Expr ::= 'false'
COMPUTE
    Expr.spim = li_iRegi(IdnNumb(0,0));
    Expr.data = PTGIntConvert(0);
END;

/*Function call*/
RULE FuncCallExpr : Expr ::= ProcIdUse '(' Actuals ')'?
COMPUTE
    Expr.spim = NoItems(); /*CHANGE ME!!!!!!*/
    /* Expr.spim =
    sool_func_call(ProcIdUse.Sym,Actuals.spim);  */
    /* We need the value from register 2 stored here */
    /*Expr.spim = add_iRegri(,0); */
END;

/*Call*/
RULE Call : IntReg ::= Label Args
COMPUTE
    IntReg.Instr =
    PTGCall(StringTable(GenSym(Label)),IntReg.reg);
END;

RULE arglist: Args LISTOF Arg END;

/*Store the value in the int register onto the stack*/
RULE arg : Arg ::= IntReg
COMPUTE
    sw_iStore(IntReg.reg,based_memory(30,cnt));
END;

/*Class instantiation*/
RULE ClassInstantiation : Expr ::= 'new' ClassIdUse
COMPUTE
    Expr.spim = NoItems(); /*CHANGE ME!!!!!!*/
END;

/*Method call - calls the method (Expr[2]) of an object (Expr[3])
and produces a result (Expr[1])*/
/*Type analysis: Type of Expr[1] is the return value of the method*/
Required type of Expr[2] is a objectType */

RULE MethodCall : Expr ::= VarIdUse '==' MethodIdUse '(' Actuals ')' :
    COMPUTE
        Expr.spim = NoItems(); /*CHANGE ME!!!!!!!*/
    END;

/*Instance variable access - gets the instance variable (Expr[3])
of an object (Expr[2]) and produces a result (Expr[1]) */
/* Type analysis: Type of Expr[1] is the type of the variable
(Expr[3])
Required type of Expr[2] is a objectType
No required type for Expr[3] */
RULE InstanceAccess : Expr ::= Expr '.' Expr
    COMPUTE
        Expr[1].spim = NoItems(); /*CHANGE ME!!!!!!!*/
    END;

/*Addition expression*/
/* Type analysis: Required types for Expr[1],Expr[2],and
Expr[3] are intType*/
RULE Addition : Expr ::= Expr '+' Expr
    COMPUTE
        Expr[1].spim = add_iRegrr(Expr[2].spim, Expr[3].spim);
    END;

/*Subtraction expression*/
/* Type analysis: Required types for Expr[1],Expr[2],and
Expr[3] are intType*/
RULE Subtraction : Expr ::= Expr '-' Expr
    COMPUTE
        Expr[1].spim = sub_iRegrr(Expr[2].spim, Expr[3].spim);
    END;

/*Multiplication expression*/
/* Type analysis: Required types for Expr[1],Expr[2],and
Expr[3] are intType*/
RULE Multiplication : Expr ::= Expr '//' Expr
    COMPUTE
        Expr[1].spim = mul_iRegrr(Expr[2].spim, Expr[3].spim);
    END;

/*Division expression*/
/* Type analysis: Required types for Expr[1],Expr[2],and
Expr[3] are intType*/
RULE Division : Expr ::= Expr '/' Expr
    COMPUTE
        Expr[1].spim = div_iRegrr(Expr[2].spim, Expr[3].spim);
    END;

/*Less than expression*/
/* Type analysis: Required type for Expr[1] is booleanType
Required types for Expr[2] and Expr[3] are
intType*/
RULE LessThan : Expr ::= Expr '<' Expr
    COMPUTE
        Expr[1].spim = slt_iRegrr(Expr[2].spim,Expr[3].spim);
    END;

/*Actuals list*/
/* Type analysis: this symbol must inherit ArgumentsListRoot*/
RULE RuleActuals : Actuals LISTOF Actual
    COMPUTE
        Actuals.spim = CONSTITUENTS Actual.spim WITH
            (NODEPTR, AddOne, IDENTICAL, NoItems);
    END;

/*Actual*/
/* Type analysis: The actual's type is the type of the
expression. This symbol must inherit from ArgumentListElem*/
RULE ActualRule : Actual ::= Expr
    COMPUTE
        Actual.spim = Expr.spim;
    END;

/*-----------------------------*/
/*Statements*/
/*-----------------------------*/

/*List of statements*/
RULE RuleStmts : Stmts LISTOF Stmt
    COMPUTE
        Stmts.spim = CONSTITUENTS Stmt.spim
            WITH (NODEPTR, AddOne, IDENTICAL, NoItems);
    END;

/*No-operation statement*/
RULE NopStatement : Stmt ::= 'nop'
    COMPUTE
        /*Nop doesn't do anything, we don't need any machine
    instructions*/
        Stmt.spim = NoItems();
    END;

/*Assignment statement*/
/* Expr[1] is guaranteed to be an id because of the .con file.
Type analysis: Required type for Expr[1] is the type of
Expr[2]
Spim transformation: Store value of Expr[2] in address of
Expr[1]
We must allow booleans to be stored in integer registers.*/
RULE Assignment : Stmt ::= Expr '==' Expr
    COMPUTE
        Expr[1].lv = 1; /*The LHS Expression must be an address,
not a value*/
/*The RHS Expression must be a value, which it is because of the default symbol computation*/
Stmt.spim = sw_iStore(Expr[2].spim,Expr[1].spim);
END;

ATTR _elseLbl: DefTableKey;
ATTR _endLbl: DefTableKey;

/*If statement*/
/* Type analysis: Required type for Expr is booleanType */
RULE Conditional : Stmt ::= 'if' Expr 'then' '{' CondStmts '}'
  'else' '{' CondStmts '}'
COMPUTE
  Stmt.spim = AddThree(do_branch(beqz_Branchr(Expr.spim), ._elseLbl),
    AddOne(CondStmts[1].spim, j_Jump(_endLbl)),
    AddOne(define_label(_elseLbl),
      define_label(_endLbl));
  ._elseLbl = NewKey();
  ._endLbl = NewKey();
END;

ATTR _continueLbl: DefTableKey;

/*While statement*/
/* Type analysis: Required type for Expr is booleanType */
RULE Iteration : Stmt ::= 'while' Expr 'do' '{'
  CondStmts '}'
COMPUTE
  Stmt.spim = AddOne(AddTwo( define_label(_continueLbl),
    do_branch(beqz_Branchr(Expr.spim), ._endLbl),
    j_Jump(_continueLbl)),
    define_label(_endLbl));
  ._continueLbl = NewKey();
  ._endLbl = NewKey();
END;

/*The conditional statements in if and while clauses*/
RULE ConditionalStmts : CondStmts ::= Stmts
COMPUTE
  CondStmts.spim = Stmts.spim;
END;

RULE Return : Stmt ::= 'return' Expr
COMPUTE
  Stmt.spim = Expr.spim;
END;

/*----------------------------------------------------------*/
/*Identifiers*/
/*-----------------------------------------------*/
RULE Variable_ID_DEF : VarIdDef ::= id
  COMPUTE
    VarIdDef.data =
    PTGStringConvert(StringTable(GenSym(VarIdDef.Key)));
    /* PTGStringConvert(StringTable(id)) */
END;

/*-----------------------------------------------*/

TREE SYMBOL Expr INHERITS ExpressionSymbol
  COMPUTE
    INH.lv = 0;
    SYNT.data = PTGNULL;
END;

TREE SYMBOL VarIdUse INHERITS IdUse_Name, IdUse_Type
  COMPUTE
    SYNT.lv = 0;
END;

References

ELI http://eli-project.sourceforge.net/