Types

Our approach to types is to make our programs strongly static typed. We will add syntax to provide type information for procedures, and every program will be checked for proper type matching prior to evaluating the program. A static typed language is one in which type information can be determined at “compile time” rather than at runtime. In this way, any type errors will be found before the program is evaluated.

Our typed language will be based on the SET language. In the SET language, only the integer value 0 is considered false, and all other values – including ProcVal – are true when used in conditional expressions. The Java programming language requires that the test expression in an if statement evaluate to a boolean. We will follow this example by introducing primitive types, int and bool, corresponding to integer and boolean expressed values, respectively. To implement this, we will add a new BoolVal subclass of the Val class. With this addition, we can require that the test expression in an if expression be a boolean.

For procedures, we specify the types of each of the procedure’s formal parameters and the procedure’s return type. We use the notation

\[ [t_1, t_2, \cdots, t_n \Rightarrow t] \]

to represent the type of a procedure that has \( n \) formal parameters of types \( t_1, t_2, \cdots t_n \), respectively, and that returns a value of type \( t \).
Types (continued)

A type is one of the two primitive types or a procedure type. A type expression is a syntactic category in our language that describes a type:

\[
\begin{align*}
\text{<typeExp>:PrimTypeExp} & \quad ::= \quad \text{<primType>} \\
& \quad \quad \text{PrimTypeExp(PrimType primType)} \\
\text{<typeExp>:ProcTypeExp} & \quad ::= \quad \text{LBRACK <typeExps> RARROW <typeExp> RBRACK} \\
& \quad \quad \text{ProcTypeExp(TypeExps typeExps, TypeExp typeExp)} \\
\text{<primType>:IntPrimType} & \quad ::= \quad \text{INT} \\
& \quad \quad \text{IntPrimType()} \\
\text{<primType>:BoolPrimType} & \quad ::= \quad \text{BOOL} \\
& \quad \quad \text{BoolPrimType()} \\
\text{<typeExps>} & \quad ::= \quad \text{<typeExp> +COMMA} \\
& \quad \quad \text{TypeExps(List<TypeExp> typeExpList)}
\end{align*}
\]
Types (continued)

Here are some examples of type expressions:

1. int
2. bool
3. [ bool, int => int ]
4. [ [int=>bool], [int,int=>int] => [int,int=>bool] ]
5. [ => int ]

Type expressions 1 and 2 describe integer and boolean types, respectively. Type expression 3 describes a procedure type that takes two parameters of type boolean and integer, respectively, and that returns an integer. Type expression 4 describes a procedure type that takes two procedure parameters of type [int=>bool] and [int,int=>int], respectively, and that returns a procedure of type [int,int=>bool]. Type expression 5 describes a procedure type that takes no parameters and that returns an integer.

Our typed language will now require that every procedure expression be annotated with type expressions for each of the procedure’s formal parameters and a type expression for the procedure’s return value. Together, these annotations will completely define the procedure’s type, called (not surprisingly) the defined type of the procedure.
Types (continued)

The following examples show how to annotate a procedure expression in the SET language so that it will conform to the TYPE0 language rules (see slide 8 for the start of the discussion of the TYPE0 language):

% SET language, no type annotations
proc(t) +(t,5)
% the same procedure in the TYPE0 language with type annotations
proc(t:int):int +(t,5) % type [int=>int]

% SET language, no type annotations
proc(f,x) .f(x)
% the same procedure in the TYPE0 language with type annotations
proc(f:[int=>int], x:int):int .f(x) % type [[int=>int],int=>int]

We will shortly define grammar rules for the TYPE0 language that will parse expressions such as these.
Types (continued)

When used in an expression, a procedure application is considered to have a type equal to the return type of the procedure. For example, the type of the following procedure application is `int`:

```
.proc(x:int):int +(3,x) (4) % evaluates to 7
```

A *type error* is one of the following:

- an attempt to define a procedure whose declared return type does not match the type of the body of the procedure;
- an attempt to apply a non-procedure as a procedure;
- an attempt to apply a procedure to the wrong number of arguments;
- an attempt to apply a procedure to actual parameters whose types do not match the types of the corresponding formal parameters;
- an attempt to use a non-boolean as the test in an *if* statement;
- an attempt to have expressions of different types in the *then* and *else* parts of an *if* statement;
- an attempt to assign a value to a variable with a *set* statement where the type of the variable does not match the type of the RHS expression.

In the above, we consider primitive procedures such as `+`, `add1`, and `zero?` as procedures for the purposes of type checking. These primitive procedures will have pre-defined procedure types.
Types (continued)

Our static type checking implementation will also detect errors where an identifier is referred to in a particular expression without appearing on the LHS of an enclosing `let` or `letrec` or in a top-level `define` – *i.e.*, if the identifier occurs free in the expression. It will also detect errors where the same identifier appears twice on the LHS of a `let` or `letrec` or if the same identifier appears twice in the formal parameter list of a `proc`.

A variable is in *declaration position* if it occurs on the LHS of a `define`, `let`, or `letrec` definition, or if it appears as a formal parameter in a procedure definition. These occurrences determine the *defined type* of the variable implicitly, according to the following rules:

- The defined type of a variable appearing in declaration position in a `define`, `let`, or `letrec` is the type of its RHS expression.
- The defined type of a variable appearing in a formal parameter list is its declared type in the procedure definition.

When a variable appears in an expression, the type of the variable is its defined type using static scope rules.

Notice that static scope rules make this determination possible. With dynamic scope rules, the type of a variable appearing free in a procedure body may differ from one procedure invocation to another, making static type checking impossible.
Types (continued)

Numeric literals all have type `int`. We predefine the identifiers `true` and `false` to have type `bool`, with the obvious semantics in the test expression of `if...` expressions. The type of a `let` or `letrec` expression is the type of the expression’s body.

Since the type of a variable is determined by static scope rules, the only “interesting” types are those of procedures. We need to add syntax rules that will permit us to specify the return type of a procedure and the types of its formal parameters.
We define a language named TYPE0 that is based on the SET language and that implements type syntax. We first require that a procedure definition – in the `<proc>` grammar rule – specify the types of each of its formal parameters and the return type of the procedure. Each of these specifications will be given by a type expression, described earlier, in the following grammar rules related to a `<proc>`. Notice that these grammar rules are similar to the original grammar rules for the SET language, with the exception that they include type specifications.

```
<proc> ::= PROC LPAREN <formals> RPAREN COLON <typeExp> <exp> 
Proc(Formals formals, TypeExp typeExp, Exp exp)
<formals> **= <VAR> COLON <typeExp> +COMMA 
Formals(List<Token> varList, List<TypeExp> typeExpList)
```

These changes do not affect the behavior of the eval methods that define the semantics of the SET language. Thus the resulting TYPE0 language has the same semantics as the SET language, except for the presence of boolean values: the type information in a program is simply ignored.
Language TYPE0 (continued)

A procedure definition in our TYPE0 language will now look like this, for example:

```plaintext
define add2 = proc(x:int):int add1(add1(x))
```

This says that the *declared type* of the formal parameter `x` is `int` and that the *declared return type* of `add2` is `int`, so that the *defined type* of `add2` is `[int=>int]`.

Since we have made changes to our syntax, our expression parse trees will have additional structure, but because our `eval` methods are not affected by the type information, `eval` is entirely unchanged!

Notice, however, that we have not implemented any of our type rules, so no type checking is taking place.
Language TYPE0 (continued)

What is the type of a \{ ... \} expression? Since the purpose of this statement is normally to perform side-effects and to return only the value of the last sub-expression, we assume that the type of ‘\{ ... \}’ is the type of the last sub-expression. The types of the previous sub-expressions are ignored except that they must not have type errors.

Thus the expression

```plaintext
let
    x = 1
    double = proc(t:int):int set t = *(2,t)
in
    { if zero?(x) then 0 else set x = add1(x)
      ; .double(x)
      ; proc():int x
    }
```

has type \([=>int]\).
Our TYPE0 language has two new expressions: a TrueExp and a FalseExp, with obvious concrete and abstract syntax:

\[
\begin{align*}
<\text{exp}> : \text{TrueExp} & ::= \text{TRUE} \\
& \quad \text{TrueExp()} \\
<\text{exp}> : \text{FalseExp} & ::= \text{FALSE} \\
& \quad \text{FalseExp()}
\end{align*}
\]

What should the eval method return for these classes? So far, we have used an IntVal of zero to be false and everything else to be true. Once we have syntax for true and false expressions, we should introduce corresponding semantics. We therefore create a new Val type, BoolVal, that will represent the values of these expressions. The code for the BoolVal class is in the val file.

Once we have a BoolVal, we can define the eval behavior for the TrueExp and FalseExp classes. Here’s the code for TrueExp – the code for FalseExp is nearly identical:

```java
public Val eval(Env env) {
    return new BoolVal(true);
}
```

Obviously the isTrue method applied to a BoolVal object constructed with true should return true, and similarly a BoolVal object constructed with false should return false. The isTrue method applied to any other value (either an IntVal or a ProcVal) should throw an exception.
Language TYPE0 (continued)

We can add a number of new primitives applied to integer arguments that return BoolVal values, such as the following, with obvious meanings:

```plaintext
```

The purpose of the `?` at the end of these is to remind you that they are testing something and that their expressions each return a boolean value. We will also change the value returned by the `zero?` primitive to a `BoolVal`.

The following expressions evaluate as indicated:

```plaintext
<?(3,3) => false (actually, a false BoolVal)
<=?(3,3) => true
<>?(x,y) => depends on the current bindings of x and y
=?(proc(t) t, proc(u) u) => exception -- can’t apply to procedures
if 1 then 2 else 3 => exception -- 1 is not a boolean
```
Now that we have built the syntax for type expressions and incorporated it into our grammar, we begin to implement type checking. Our strategy will be to determine the type of any expression, making type checks along the way.

Our first step will be to define `Type` as a semantic entity in our implementation. We will implement `Type` as an abstract class with three subclasses:

- `IntType`
- `BoolType`
- `ProcType`

These correspond to the three classes that extend the `Val` class. However, `Val` objects are used to evaluate expressions, while `Type` objects are used for type checking. They serve similar purposes but are used in different ways.

Neither `BoolType` nor `IntType` has any instance variables. A `ProcType` object has two instance variables, as described here:

```java
ProcType(List<Type> paramTypeList, Type returnType)
```

An expression will be of type `IntType` if it evaluates to an integer. Similarly for a `BoolType`. When we evaluate an expression, we want to know the actual value of that expression in its proper environment. But when we are doing type checking, we don’t care what the expressed value actually is: only its type matters.
Our approach will be to evaluate the type (Type) of an expression, doing type checks of sub-expressions along the way. If there are no type errors, we can evaluate the value (Val) of the expression and return this value as we have done before.

First, when we encounter a variable in an expression, we want to determine the type of that variable. If the variable is a formal parameter in the body of a procedure, its type will be determined by the type declaration of the formal parameter. For any other variable, its type can be determined using static scope rules. This suggests that we have a way of binding the variable to its type during type checking in a way that is similar to binding the variable to its value during expression evaluation. We achieve this by creating a *type environment* that is almost identical to a value environment that we have been using so far, except that our type environment will return the type bound to a variable rather than its value.

The appropriate classes are shown on the next slide.
abstract class TypeEnv
  public static TypeEnv emptyTypeEnv() // returns an empty environment
  public abstract Type applyTypeEnv(String sym)
  public abstract TypeEnv extendTypeEnv(TypeBindings typeBindings)
  public abstract TypeEnv extendTypeEnv(List<String> idList, List<Type> typeList)

class TypeEnvNull extends TypeEnv  // empty type environment class

class TypeEnvNode extends TypeEnv
  public TypeBindings typeBindings // local bindings
  public TypeEnv typeEnv  // next environment

class TypeBindings
  public List<TypeBinding> typeBindingList

class TypeBinding
  public String sym
  public Type type

abstract class Type

class IntType extends Type
class BoolType extends Type
class ProcType extends Type
Language TYPE1 (continued)

The definitions of the classes relating to type environments is given in the tenv file.

Since most of our type checking rules relate to type equality, we must define what it means for two types to be equal. There are only three basic types, and for each of them we define a checkEquals method. This void method will silently return if the types are the same, otherwise it will call the static typeMismatch method in the Type class; this method throws a runtime exception with an appropriate message.

The two primitive types are simple. For the BoolType class, the checkEquals method is defined as follows:

```java
public void checkEquals(Type t) {
    t.checkBoolType(this);
}

public void checkBoolType(BoolType t) {
    // if we get here, this must be a BoolType
}
```

The default checkBoolType method in the Type class – defined for all other types except for BoolType – throws a type mismatch exception.

A similar definition works for the IntType.

We are left to define checkEquals for the ProcType class. This is easy: two ProcType objects are equal if they both have the same number of formal parameter types and these formal parameter types are pairwise equal (in the proper order), and if they both have the same return type.
Here is the definition of checkEquals in the ProcType class:

```java
public void checkEquals(Type t) {
    t.checkProcType(this);
}
```

// check to see if the type of the ProcType object t is the same
// as this ProcType object
public void checkProcType(ProcType t) {
    // first check the return types
    this{returnType}.checkEquals(t{returnType});
    // then check the types of the formal parameters
    checkEqualTypes(this{paramTypeList}, t{paramTypeList});
}
```

The checkEqualTypes static method in the Type class is straight-forward: first check for equal sizes; then iterate through both of the lists, checking pairwise for type equality.

```java
public static void checkEqualTypes(List<Type> t1List, List<Type> t2List) {
    if (t1List.size() != t2List.size())
        throw new RuntimeException("argument number mismatch");
    Iterator<Type> t1i = t1List.iterator();
    Iterator<Type> t2i = t2List.iterator();
    while (t1i.hasNext()) {
        Type t1 = t1i.next();
        Type t2 = t2i.next();
        t1.checkEquals(t2);
    }
}
```
At this point we can also add primitives operations for and, or, and not.

\[
\begin{align*}
\text{<prim>AndPrim} & \ ::= \ \text{AND} \\
\text{<prim>OrPrim} & \ ::= \ \text{OR} \\
\text{<prim>NotPrim} & \ ::= \ \text{NOT}
\end{align*}
\]

Here is the apply code for an AndPrim object:

```java
public Val apply(Val [] va) {
    if (va.length != 2)
        throw new RuntimeException("two arguments expected");
    boolean b0 = ((BoolVal)va[0]).val;
    boolean b1 = ((BoolVal)va[1]).val;
    return new BoolVal(b0 && b1);
}
```

It’s easy to see how the apply code should be implemented for the other two primitives.
Although primitives are not procedures (you can’t bind a variable to `add1`, for example), they do have specific type behaviors that can be described in terms of procedure types. The `+` primitive behaves like a `[int, int=>int]`, and the `add1` primitive behaves like a `[int=>int]`. Each of the primitives will then have a `ProcType` that can be used for type checking. We define these types in the `Type` class as static instance variables.

Rather than building each of these types by hand with constructors, we define and use a special static method `compile` in the `Type` class that takes a compact string representation of a procedure type such as `[int, int=>int]` (in this case, the string is "ii>i") and returns a `ProcType` object with the proper formal parameter types and result type. You can see how the `compile` method is used in the following example definitions and their corresponding interpretations:

```java
public static Type ii_i = compile("ii>i"); // [int, int=>int]
public static Type i_i = compile("i>i"); // [int=>int]
public static Type ii_b = compile("ii>b"); // [int, int=>bool]
public static Type bb_b = compile("bb>b"); // [bool, bool=>bool]
```

The `evalType` method for the `PrimappExp` class will ask the primitive object what its type is, and the object will return the appropriate type from among the ones compiled in the `Type` class. For example, in the `AddPrim` class, we define:

```java
public Type toType() {
    return Type.ii_i;
}
```
When processing `let` and `letrec` expressions and defining procedures, we want to ensure that there are no duplicate variable names. This amounts to checking the `varList` instance variables in the `Formals`, `LetDecls`, and `LetrecDecls` classes. This must be done after parsing, since the parser only uses the grammar rules and these rules do not have any way to check for duplicates. We will check for duplicates during type checking and before expression evaluation.

One way to check for duplicates is to use a Java `Map` object: for each variable in the list of variables, if it’s already in the map, return an error, otherwise add it to the map. Here is the code:

```java
public void checkDuplicates(List<String> varList) {
    Map<String,String> map = new HashMap<String,String>();
    for (String id : varList) {
        String s = map.get(id);
        if (s != null)
            throw new RuntimeException("duplicate identifier: " + id);
        map.put(id, id);
    }
}
```
While the type environment classes are similar to those for expression environments, it is unnecessary to use references, since we will never be modifying the type of a variable. Therefore type environments will be similar to the environment implementation in language V6. The essential changes are to replace Var with Type, Env with TypeEnv, and Binding with TypeBinding.

We are now prepared to write the code to type-check expressions. We need to initialize a top-level type environment, which we will call initTypeEnv. This will happen in the Program class, where we also define the top-level expression environment.

```java
static TypeEnv tenv = TypeEnv.initTypeEnv();
```

In the Eval class’s toString method, we type-check by applying the evalType on the exp object – the “program” that we are to evaluate. The actual return value of this is irrelevant, since its purpose is simply to carry out type checking prior to expression evaluation. If the type checking succeeds, the toString method returns the string representation of the value of the expression.

```java
public String toString() {
    exp.evalType(Program.tenv);
    return exp.eval(Program.env).toString();
}
```
Types (continued)

The type of a LitExp is easy:

```java
public Type evalType(TypeEnv tenv) {
    return Type.intType; // a singleton in the Type class
}
```

So are the types of a TrueExp and a FalseExp. For both the TrueExp and FalseExp classes, we have

```java
public Type evalType(TypeEnv tenv) {
    return Type.boolType;
}
```

The type of a VarExp looks up the type of the variable in the type environment and returns the result:

```java
public Type evalType(TypeEnv tenv) {
    return tenv.applyTypeEnv(var);
}
```
Types (continued)

The type of a `LetExp` involves checking the LHS variable list for duplicates, determining the types of the RHS expressions using the current type environment, extending the current type environment by binding the variables to the types of the RHS expressions, and evaluating the type of the `let` body using this extended environment.

```java
public Type evalType(TypeEnv tenv) {
    TypeEnv ntenv = letDecls.addTypeBindings(tenv); // extended tenv
    return exp.evalType(ntenv);
}
```

Most of the work is done in the `addTypeBindings` method in the `LetDecls` class, which we show here.

```java
public TypeEnv addTypeBindings(TypeEnv tenv) {
    List<String> stringVarList = new ArrayList<String>(); // LHS vars
    for (Token t : varList) // get the variable symbols
        stringVarList.add(t.toString());
    TypeEnv.checkDuplicates(stringVarList);
    TypeEnv.checkDuplicates(expList); // RHS expressions
    List<Type> typeList = rands.evalTypeRands(tenv); // RHS types
    TypeBindings typeBindings = new TypeBindings(stringVarList, typeList);
    return tenv.extendTypeEnv(typeBindings);
}
```
The `Rands` object must define the `evalTypeRands` method, which is straight-forward. Notice that this is where static scope rules come into play: in a `Let` expression, the types of the RHS expressions are evaluated in the enclosing type environment, `tenv`.

```java
public List<Type> evalTypeRands(TypeEnv tenv) {
    List<Type> typeList = new ArrayList<Type>();
    for (Exp e : expList)
        typeList.add(e.evalType(tenv));
    return typeList;
}
```
Language TYPE1 (continued)

In order to determine the defined type of a procedure, we need to give semantic meaning to the type expressions used to declare the types of a procedure’s formal parameters and return type. Specifically, we need to take a type expression given by the \(<\text{typeExp}>\) grammar rules and evaluate its \text{Type}: in other words, the semantics of a type expression is its \text{Type}. This is parallel to taking a value expression give by \(<\text{exp}>\) grammar rules and evaluating its \text{Val}. Both of these evaluations are examples of semantics.

For each type expression, we will implement a \text{toType} method that returns the corresponding instance of \text{Type}. One thing to observe is that a type expression does not involve any variables, so there is no need to keep track of a type environment.
Language TYPE1 (continued)

For example, a type expression of the form \([\text{int}=>\text{bool}]\) would evaluate to a \text{ProcType}, and a type expression of the form \text{int} would evaluate to a \text{IntType}.

Here’s the definition of \text{toType} for the \text{PrimTypeExp} class (representing either an \text{int} or a \text{bool}) in the code file:

\begin{verbatim}
PrimTypeExp
  {public Type toType() {
    return primType.toType(); // intType or boolType
  }}
\end{verbatim}

For the \text{BoolPrimType} class, which corresponds to grammar rules for the \text{true} and \text{false} tokens, we simply return \text{boolType}.

\begin{verbatim}
BoolPrimType
  {public Type toType() {
    return Type.boolType; // a singleton in the Type class
  }}
\end{verbatim}

Similar remarks apply to \text{IntPrimType}.
Language TYPE1 (continued)

For a `ProcTypeExp`, we

1. determine the types (using the `toType` method) of the (formal parameter) type expressions that appear to the left of the `=>` token and insert them into a `List<Type>` object,

2. determine the type of the (return) type expression that appears to the right of the `=>` token, and then

3. construct the appropriate `ProcType` object.

```java
ProcTypeExp

public Type toType() {
    List<Type> paramTypeList = new ArrayList<Type>();
    for (TypeExp e : typeExps.typeExpList)
        paramTypeList.add(e.toType());
    Type returnType = typeExp.toType();
    return new ProcType(paramTypeList, returnType);
}
```
Language TYPE1 (continued)

We are now in a position to determine the *defined type* of a proc expression. By defined type, we mean the type of the procedure (a ProcType) as determined by the declared types of each of the formal parameters and the declared return type. For example,

```plaintext
proc(x:int):bool ...
```

has a formal parameter of declared type int and a declared return type of bool, so the defined type of this procedure is [int=>bool] *The defined type of a procedure is not dependent on the type environment in which the procedure expression appears, nor on the names of the formal parameters.*

However, the type of the procedure *body* is dependent on the type environment in which the procedure is evaluated, since the procedure body may have variables that are free with respect to the procedure’s formal parameters but that are bound to types defined in an enclosing environment. One of our type checks is to ensure that the procedure’s declared return type is the same as the type of the procedure body.
The `definedType` method in the `Proc` class evaluates the procedure’s defined type, which must be a `ProcType`: it gets the declared types of the formal parameters from the `Formals` object and the declared return type from the procedure’s `typeExp`. The defined type is determined exclusively by the declared types of the formal parameters and the procedure’s declared return type.

```java
Proc
/**
   * public ProcType definedType() {
   *     List<Type> declaredTypeList = formals.formalTypeList();
   *     Type declaredReturnType = typeExp.toType();
   *     return new ProcType(declaredTypeList, declaredReturnType);
   * }
/**
```

The `Formals` class defines the `formalTypeList` method:

```java
Formals
/**
   * public List<Type> formalTypeList() {
   *     List<String> stringVarList = new ArrayList<String>();
   *     for (Token t : varList) {
   *         stringVarList.add(t.toString());
   *     }
   *     TypeEnv.checkDuplicates(stringVarList);
   *     List<Type> typeList = new ArrayList<Type>();
   *     for (TypeExp te : typeExpList)
   *         typeList.add(te.toType());
   *     return typeList;
   * }
/**
```
Language TYPE1 (continued)

We are now ready to define `evalType` on a `Proc`.

```java
public Type evalType(TypeEnv tenv) {
    // retrieve the declared return type of the proc
    Type declaredReturnType = typeExp.toType();
    // bind the formal parameters to their declared types
    TypeBindings typeBindings = formals.declaredTypeBindings();
    // extend the tenv by the formal param type bindings
    TypeEnv ntenv = tenv.extendTypeEnv(typeBindings);
    // evaluate the type of the body using this extended type environment
    Type expType = exp.evalType(ntenv);
    // and check that the declared return type matches the body type
    Type.checkEquals(declaredReturnType, expType);
    // finally build the ProcType
    List<Type> formalTypeList = formals.formalTypeList();
    return new ProcType(formalTypeList, declaredReturnType);
}
```

Type checking for an `if` expression involves making sure that the test expression has type `bool` and that the true part and the false part have the same types. Thus, for a `IfExp`, we have:

```java
public Type evalType(TypeEnv tenv) {
    Type testType = testExp.evalType(tenv);
    testType.checkBoolType(Type.boolType);
    Type trueExpType = trueExp.evalType(tenv);
    Type falseExpType = falseExp.evalType(tenv);
    Type.checkEquals(trueExpType, falseExpType);
    return trueExpType;
}
```
As we described earlier, a sequence expression has type equal to the type of the last expression in the sequence. The following method is defined in the `SeqExp` class:

```java
public Type evalType(TypeEnv tenv) {
    Type t = exp.evalType(tenv);
    for (Exp e : seqExps.expList)
        t = e.evalType(tenv);
    return t;
}
```

In a set expression, we look up the type of the LHS variable and make sure that it matches the type of the RHS expression. The following method is defined in the `SetExp` class:

```java
public Type evalType(TypeEnv tenv) {
    Type tv = tenv.applyTypeEnv(var);
    Type te = exp.evalType(tenv);
    Type.checkEquals(tv, te);
    return tv;
}
```
A letrec expression will be treated similar to a let expression, except that the types of the RHS procedures must be evaluated in an environment that has type bindings for the LHS variables. Since the defined type of a RHS procedure can be determined solely by declared types of its parameters and return type, the types of the LHS variables can be determined independent of any environment. So we can use these LHS variables to extend the type environment and then evaluate the types of the RHS procedure bodies using this extended environment, for the purpose of type checking with their declared return types. The addTypeBindings method in the LetrecDecls class does all the work.

```java
public TypeEnv addTypeBindings(TypeEnv tenv) {
    // first check for duplicate LHS identifiers
    List<String> stringVarList = new ArrayList<String>();
    for (Token t : varList)
        stringVarList.add(t.toString());
    TypeEnv.checkDuplicates(stringVarList);
    // next get the defined procedure types in the letrec,
    // ignoring the procedure bodies for now
    List<Type> definedTypeList = new ArrayList<Type>();
    for (Proc p : procList)
        definedTypeList.add(p.definedType()); // just the defined type
    // extend tenv with bindings of the LHS vars to the RHS defined types
    TypeBindings typeBindings =
        new TypeBindings(stringVarList, definedTypeList);
    TypeEnv ntenv = tenv.extendTypeEnv(typeBindings);
    // finally check that the proc bodies are OK in the ntenv environment
    for (Proc p : procList)
        p.evalType(ntenv);
    return ntenv;
}
```
Language TYPE1 (continued)

Once we have type checked the RHS entries and built the extended type environment with bindings for the LHS variables, we can evaluate the type of the letrec body using this extended environment.

```java
LetrecExp
%%{
    public Type evalType(TypeEnv tenv) {
        TypeEnv ntenv = letrecDecls.addTypeBindings(tenv);
        return exp.evalType(ntenv);
    }
} %}
```
Types (continued)

Two to go: AppExp and PrimappExp. Both of these are similar in that we need to check if the formal parameter types of the operator agree in number and type with the actual parameter (operand) types. For an AppExp, we also must ensure that the thing we are applying is actually a procedure – it must evaluate to a ProcType.

Because a ProcType object already knows (and has checked the validity of) its return type, the type of an application is simply the return type of the procedure.

AppExp

```java
public Type evalType(TypeEnv tenv) {
    Type tt = exp.evalType(tenv);
    tt.checkProcType(); // make sure tt is a ProcType
    ProcType pt = (ProcType)tt;
    List<Type> argTypeList = rands.evalTypeRands(tenv);
    Type.checkEqualTypes(pt.paramTypeList, argTypeList);
    return pt.returnType;
}
```

```java
```
Types (continued)

For PrimappExp, we do the same thing, except that we simply ask the primitive to identify its type using its definedType method:

```java
public Type evalType(TypeEnv tenv) {
    ProcType pt = prim.definedType();
    List<Type> argTypeList = rands.evalTypeRands(tenv);
    Type.checkEqualTypes(pt.paramTypeList, argTypeList);
    return pt.returnType;
}
```

Notice that the type of a primitive is defined statically. The primitive’s definedType method just retrieves the type, which must be a ProcType.
Types (continued)

We are left to handle top-level defines. While this seems to be straight-forward, we are faced with problems of unbound variables. For example, suppose we attempt to make the following definitions:

```plaintext
define odd? =
    proc(x:int):bool if zero?(x) then false else .even?(sub1(x))

define even? =
    proc(x:int):bool if zero?(x) then true else .odd?(sub1(x))
```

When the procedure body of `odd?` is first encountered, the variable `even?` is unbound, so the type of the body of the procedure cannot be evaluated. Any attempt to type-check `odd?` will fail with a type error.

We can remedy this situation in two ways:

- forbid top-level `define` s altogether; or
- create a mechanism to deal with unbound variables that will make top-level `define` s work.
Types (continued)

We will take the second approach, with some restrictions.

First, we create a new top-level operator declare that has a syntax similar to how we declare the type of a formal parameter in a procedure. This will allow us to bind a type to an identifier without actually creating a value binding for the identifier.

Pascal provides such a mechanism with forward declarations, as does C/C++ with procedure declarations. Java does not suffer from this problem, in part because the order of declaration of fields and methods is not significant.

A declaration has the following concrete and abstract syntax:

```
<program>:Declare ::= DECLARE <VAR> COLON <typeExp>
```

```python
Declare(Token var, TypeExp typeExp)
```
Types (continued)

The rules for top-level defines and declares are:

1. A top-level identifier definition serves as a declaration for the purpose of determining the type of the identifier.

2. It is an error for a top-level identifier to be declared (hence defined) more than once.

3. If an identifier declaration precedes its definition, it is an error if the declared and defined types are not identical.

4. It is an error for a declaration not to have a subsequent definition, although this error need not be checked.

Although you cannot define a top-level variable more than once, you can redefine a top-level variable using set if the variable has been previously defined. The type rule for variable assignment still requires that the RHS type of the set must agree with the declared type of the variable.
Types (continued)

Examples of top-level declarations/definitions:

1. define x = 3 % type int
   define x = 4 % error – multiple definitions for x

2. define x = 3 % type int
   set x = 4 % ok – types agree
   set x = true % error – type of RHS must be int

3. define fact = proc(x:int):int
   if zero?(x) then 1 else *(x,.fact(sub1(x)))
   % error – fact on the RHS has no type binding!

4. declare fact : [int=>int]
   define fact = proc(x:int):int
     if zero?(x) then 1 else *(x,.fact(sub1(x))) %ok

5. declare f : [=>$bool]
   define y = 2
   define f = proc():int y
   % error – defined type does not match declared type

6. declare f : [=>$bool]
   define y = 2
   define f = proc():bool y
   % error – body type does not match return type
Types (continued)

As a final example, consider

```
declare f : [=>int]
declare y : int
define f = proc():int y
  .f() % error - no binding for y
```

The definition for \( f \) is type valid, but \( y \) has no defined value binding, so the procedure application \( .f() \) results in a run-time error.

If we subsequently give a value definition for \( y \), the procedure application will succeed, as the following shows:

```
declare f : [=>int]
declare y : int
define f = proc():int y
define y = 3
  .f() % ok - returns 3
```
Types (continued)

We implement top-level declarations as follows:

```java
Declare

// calling toString may trigger a modification
// of the initial type environment
public String toString() {
    TypeEnv tenv = Program.tenv; // top-level type environment
    Type tv; // variable’s declared type
    String sym = var.toString(); // the LHS symbol
    try {
        // look up the variable in the initial type environment
        tv = tenv.applyTypeEnv(sym);
    } catch (RuntimeException e) {
        // no type binding -- must be a new type declaration
        // that we add to the top-level type environment
        tv = typeExp.toType();
        tenv.add(new TypeBinding(sym, tv)); // type binding
        return sym + ":" + tv;
    }
    throw new RuntimeException(sym + ": duplicate variable declaration");
}
```

If a type binding (either through a declaration of definition) already exists, it is flagged as a duplicate declaration. Otherwise a top-level binding is made between the identifier and the type represented by its type expression. The `toString` return value for a top-level declaration/definition is the identifier being declared/defined along with its declared type.
We expand on top-level definitions as follows. Notice that this code is in two parts: the first part is where the variable hasn’t been declared/defined before:

```java
public String toString() {
    Env env = Program.env; // top-level environment
    TypeEnv tenv = Program.tenv; // top-level type environment
    Type rhst; // RHS expression type
    Val rhsv; // RHS expression value
    Type lhst; // LHS variable declared type
    Val lhsv; // LHS variable’s current value
    String sym = var.toString(); // the LHS symbol
    try {
        lhst = tenv.applyTypeEnv(sym);
    } catch (RuntimeException e) {
        rhst = exp.evalType(tenv);
        tenv.add(new TypeBinding(sym, rhst)); // type binding
        rhsv = exp.eval(env);
        env.add(new Binding(sym, new ValRef(rhsv))); // value binding
        return sym + "":" + rhst; // return *something*
    }
    ...
}
```
The second part is where the variable has a declaration but possibly not a definition:

Define

```
    ...

    // the variable has a declared type, lhst -- see if it needs a def’n
    try {
        // look up the value of var in the initial environment
        lhsv = env.applyEnv(sym);
    }
    catch (RuntimeException e) {
        // the variable has a declared type, but no value bound to it
        // so we want to add the value binding to the top-level env.
        // first check the defined type of the RHS
        rhst = exp.evalType(tenv);
        // the declared and defined types must be the same
        Type.checkEquals(lhst, rhst);
        // get the RHS expression value
        rhsv = exp.eval(env);
        // and bind it to the variable
        env.add(new Binding(sym, new ValRef(rhsv)));
        return sym + "::" + lhst;
    }

    // the variable has a value, so must be a duplicate
    throw new RuntimeException(sym + ":: duplicate variable definition");
```

```
Types (continued)

The code on the previous slide implements a top-level definition, binding a value to an identifier in the initial environment. According to our type rules, if a value binding already exists, it is flagged as an error. If a declared type binding exists without a value binding, the expression type is checked for conformance with the declared type. If a declared type binding does not exist, the top-level type environment is extended to include the type binding. In case there are no errors, the top-level value environment is extended to include the value binding.

Armed with both declare and define, we can now implement our odd? and even? procedures:

```
declare odd? : [int=>bool]
define even? =
    proc(t:int):bool if zero?(t) then true else .odd?(sub1(t))
define odd? =
    proc(t:int):bool if zero?(t) then false else .even?(sub1(t))
```