Types

Our approach to types is to create a language that is *strongly static*
syntax to provide type information for values, and every program
proper type matching prior to evaluating the program.

A *static typed* language is one in which type information can be
"compile time" rather than at runtime. In this way, any type errors a
the program is evaluated. Saying that a language is *strongly typ*
there are no type "holes" in the system: the declared type of a sym
or procedure parameter) dictates the types of the values that can b
symbol. Everything is type checked, without exception, and all ex
conform to type rules.

Our typed language is based on the `SET` language. In the `SET` lan
integer value `0` is considered false, and all other values – includin
are true when used in conditional expressions. The Java programm
which is staticaly typed, requires that the test expression in an `if`
uates to a boolean. We follow this example in our typed language
primitive types, `int` and `bool`, corresponding to integer and boo
values, respectively. To implement this, we add a new `BoolVal`
`Val` class. With this addition, we can require that the test expre
expression be an instance of `BoolVal`.

Types

Observe that the `int` type in our (as yet unnamed) language is *not* t
`int` type in Java, even though they are closely related and have t
There should be no confusion about `bool`, since in Java the corres
`boolean`.

Our objective is to associate a type with every instance of the c
`IntVal` has type `int`, and a `BoolVal` has type `bool`. The c
object is a `ProcVal`, so we need a way to give a type to a `ProcVa`

When we define a procedure, we *declare* the types of each of the proc
parameters and the procedure's return type. We show how to do thi
we know these declared types, we can define the type of the procedu

We use the notation

$$[\, t_1 \,,\, t_2 \,,\, \cdots \,,\, t_n \;=>\; t \,]$$

to represent the type of a procedure that has n formal parameters of
t_n, respectively, and that returns a value of type t.

To simplify things, we do not permit the creation of new (named)
type is either an integer, boolean, or procedure.

Types (continued)

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

```
<typeExp>:PrimTypeExp   ::= <primType>
                         PrimTypeExp(PrimType primType)
<typeExp>:ProcTypeExp   ::= LBRACK <typeExps> RARROW <t
                         ProcTypeExp(TypeExps typeExps,
<primType>:IntPrimType  ::= INT
                         IntPrimType()
<primType>:BoolPrimType::= BOOL
                         BoolPrimType()
<typeExps>              **= <typeExp> +COMMA
                         TypeExps(List<TypeExp> typeExp
```

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 primitive `int` and `bool` tively. Type expression 3 describes a procedure type that takes t of type `bool` and `int`, respectively, and that returns an `int`. sion 4 describes a procedure type that takes two procedure para `[int=>bool]` and `[int,int=>int]`, respectively, and that r dure of type `[int,int=>bool]`. Type expression 5 describes a that takes no parameters and that returns an `int`. Observe that **any t that starts with a '`[`' must be a procedure type**.

Our typed language now requires that every procedure expression be type expressions for each of the procedure's formal parameters – its *mal parameter types* – and a type expression for the procedure's re *(declared) return type*. Together, these annotations completely de dure's type, called (not surprisingly) the *defined type* of the procedu

Types (continued)

The following examples show how to modify a procedure express language by adding type annotations so that it conforms to the TY rules (see Slide 9 for the start of the discussion of the `TYPE0` langu

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

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

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

```
.proc(t:int):int +(t,5) (4) % evaluates to 9, an int
```

We will shortly define grammar rules for the `TYPE0` language tha sions such as these.

Types (continued)

Every expression has a type. The type of a variable (`VarExp`) i[s] is bound to the variable in the current type environment. All have a type binding.

A *type error* is one of the following:

- an attempt to define a procedure whose declared return type doe[s] 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 actual p
- an attempt to apply a procedure to actual parameters whose type the declared types of the corresponding formal parameters;
- an attempt to use a non-boolean as the test in an `if` expression;
- an attempt to have expressions of different types in the `then` a of an `if` expression;
- an attempt to assign a value to a LHS variable in a `set` expres type of the variable does not match the type of the RHS expressi

In the above, we consider primitive procedures such as +, `add1`, procedures for the purposes of type checking. These primitive p[r] pre-defined procedure types.

Types (continued)

Our static type checking implementation also detects errors where referred to in a particular expression without appearing on the LHS `let` or `letrec` or in a top-level `define` – *i.e.*, if the identifier oc expresion. It also detects errors where the same identifier appears tw of a `let` or `letrec` or if the same identifier appears twice in the fo list of a `proc`.

A variable is in *declaration position* if it occurs on the LHS of a `de` `letrec` definition, or if it appears as a formal parameter in a proce These occurrences determine the *defined type* of the variable accor lowing rules:

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

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

Types (continued)

Notice that static scope rules make it possible to determine the typ
that appear free in a procedure body – in other words, variables tha
as procedure formal parameters.

Numeric literals all have type `int`. We predefine the identifiers `tr`
to have type `bool`, with the obvious semantics in test expressions
pressions.

The type of a `let` or `letrec` expression is the type of the expressi

Observe that the expressed value of the `testExp` in an `IfExp` ca
an `IntVal`, so an expression like

```
if 0 then 3 else 4
```

that we have seen in Language SET won't work in our typed langua

Since the type of a variable is determined by static scope rules, t
esting" types are those of procedures. We need to add syntax rule
definitions that permit us to declare the types of its formal paramete
type.

Language TYPE0

We define a language named `TYPE0` that is based on the `SET` lan
implements type syntax. We first require that a procedure defi
`<proc>` grammar rule – declare the types of each of its formal p
the return type of the procedure. Each of these declarations are g
expression, described earlier, in the following grammar rules relate
Notice that these grammar rules are similar to the original gramm
`SET` language, with the exception that they include type declaration

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

These changes do *not* affect the behavior of the `eval` methods tha
mantics of the `SET` language. Thus the resulting `TYPE0` languag
semantics as the `SET` language, except for the presence of `bool` v
information in a program is simply ignored.

Language TYPE0 (continued)

A procedure definition in our `TYPE0` language now looks like this,

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

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

Since we have made changes to our syntax, our expression parse ditional structure, but because our `eval` methods are not affected formation, the `eval` methods in the `code` file for the `SET` langua unchanged in the `code` file for `TYPE0`.

Notice, however, that we have not implemented any of our type ru checking is taking place.

Language TYPE0 (continued)

Our `TYPE0` language has two new expressions: a `TrueExp` and a `FalseExp`, crete and abstract syntax:

```
<exp>:TrueExp   ::= TRUE
                    TrueExp()
<exp>:FalseExp  ::= FALSE
                    FalseExp()
```

What should the `eval` method return for these classes? In Language SET, we us zero to be false and everything else to be true. Once we have syntax for `true` ar sions in Language TYPE0, we introduce corresponding semantics using the `Va` that represent the values of these expressions. The `BoolVal` class is simply a J for Java `boolean` values. The code for the `BoolVal` class is in the `val` file.

Once we have the `BoolVal` class, we can define the `eval` behavior for th `FalseExp` classes. Here's the code for `TrueExp` – the code for `FalseExp` cal:

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

Obviously the `isTrue` Java method applied to a `BoolVal` object constructed w turn true (in Java), and similarly the `isTrue` Java method applied to a `BoolVal` with `false` must return false. The `isTrue` method applied to *any* other value (or a `ProcVal`) must throw an exception.

Language TYPE0 (continued)

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

```
<?
<=?
>?
>=?
=?
<>?
```

The purpose of the `?` at the end of these is to remind you that they are testing s they return boolean values (they are *predicates*). We also change the value returr primitive to a `BoolVal`.

The following expressions evaluate as indicated:

```
<?(3,3)                    % => false (actually, a false BoolVal
<=?(3,3)                   % => true
<>?(x,y)                   % => depends on the current bindings
=?(proc(t) t, proc(u) u)   % => exception -- a proc is not an Ir
if 1 then 2 else 3         % => exception -- 1 is not a boolean
zero?(1)                   % => false
zero?(0)                   % => true
zero?(proc(t) t)           % => exception -- a proc is not an Ir
```

Language TYPE1

Now that we have built the syntax for type expressions and incorporated it into begin to implement type checking. Our strategy is to determine the type of any e type checks along the way.

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

```
IntType
BoolType
ProcType
```

These correspond to the three classes that extend the `Val` class. Observe that `Va` to evaluate expressions, whereas `Type` objects are used for type checking. They poses but are used in different ways.

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

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

An expression is of type `IntType` if it evaluates to an integer. Similarly for a `Bo`

When we evaluate an expression, we use an environment to determine its expresse are doing type checking, we don't care what the expressed value actually is: only

Language TYPE1 (continued)

Our approach is to evaluate the type (`Type`) of an expression, doing sub-expressions along the way. If there are no type errors, we can th value (`Val`) of the expression and return this value as we have done

First, when we encounter a variable in an expression, we want to det of that variable. If the variable is a formal parameter in the body of type is determined by the type declaration of the formal parameter. variable, its type can be determined using static scope rules. This s have a way of binding the variable to its type during type checking similar to binding the variable to its value during expression evaluati this by creating a *type environment* that is almost identical to a val that we have been using so far, except that our type environment bi types rather than to values.

The appropriate classes are shown on the next slide.

Language TYPE1 (continued)

```
abstract class TypeEnv
    public static TypeEnv emptyTypeEnv() // returns an empty e
    public abstract Type applyTypeEnv(String sym)
    public abstract TypeEnv extendTypeEnv(TypeBindings typeBi
    public abstract TypeEnv extendTypeEnv(List<String> idList,

class TypeEnvNull extends TypeEnv  // empty type environment c

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` f

Since most of our type checking rules relate to type equality, we must define wha types to be equal. There are only three basic types, and for each of them we define method. This `void` method silently returns if the types are the same, otherwise `typeMismatch` method in the `Type` class; this method throws a runtime exce propriate message.

The two primitive types are simple. For the `BoolType` class, the `checkEqua` fined as follows:

```
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 othe `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: objects are equal if they both have the same number of formal parameter types parameter types are pairwise equal (in the proper order), and if they both have the

Language TYPE1 (continued)

Here is the definition of `checkEquals` in the `ProcType` class:

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

// check to see if the type of the ProcType object t is the sa
// 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: fi sizes; then iterate through both of the lists, checking pairwise for type equality.

```
public static void checkEqualTypes(List<Type> t1List, List<Typ
    if (t1List.size() != t2List.size())
        throw new PLCCException("Type error", "argument number
    Iterator<Type> t1i = t1List.iterator();
    Iterator<Type> t2i = t2List.iterator();
    while (t1i.hasNext()) {
        Type t1 = t1i.next();
        Type t2 = t2i.next();
        t1.checkEquals(t2);
    }
}
```

Language TYPE1 (continued)

At this point we can also add primitives operations for `and`, `or`, and `not`.

```
<prim>AndPrim ::= AND
<prim>OrPrim  ::= OR
<prim>NotPrim ::= NOT
```

Here is the `apply` code for an `AndPrim` object:

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

It's easy to see how the `apply` code should be implemented for the other two
`not`.

Language TYPE1 (continued)

Although primitives are not procedures (you can't bind a variable
example), they do have specific type behaviors that can be describ
procedure types. The + primitive behaves like `[int,int=>int]`
primitive behaves like `[int=>int]`. Each of the primitives are as
`ProcType` that is used for type checking. We define these types in
as static instance variables.

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

```
public static ProcType ii_i = compile("ii>i"); // [i
public static ProcType i_i  = compile("i>i");  // [i
public static ProcType ii_b = compile("ii>b"); // [i
public static ProcType bb_b = compile("bb>b"); // [b
```

Language TYPE1 (continued)

The `definedType` method in the `Prim` class asks the primitive procedure type is, and the object returns the appropriate type from a compiled in the `Type` class. For example, in the `AddPrim` class, w

```
public ProcType definedType() {
    return Type.ii_i;
}
```

When declaring the formal parameter types of procedures and proce `letrec` expressions, we ensure that there are no duplicate variabl the `:init` hook during parsing.

Language TYPE1 (continued)

While the type environment classes are similar to those for expre ments, it is unnecessary to use references, since we never modify variable. Therefore type environments are similar to the environme tion in language `V6`. The essential changes are to replace `Var` wi with `TypeEnv`, and `Binding` with `TypeBinding`.

We are now prepared to write the code to type-check expressions. tialize a top-level type environment, which we call `initTypeEnv` in the `Program` class, where we also define the top-level expressio

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

In the `Eval` class's `$run` method, we type-check by applying the the `exp` object – the "program" that we are to evaluate. The actu of this is irrelevant, since its purpose is simply to carry out type ch expression evaluation. If the type checking succeeds, the `$run` m the string representation of the value of the expression.

```
public void $run() {
    exp.evalType(Program.tenv); // type check first
    Val val = exp.eval(Program.env);
    System.out.println(val);
}
```

Language TYPE1 (continued)

The type of a `LitExp` is easy:

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

So are the types of a `TrueExp` and a `FalseExp`. For both the `TrueExp` and `Fa`
we have

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

The type of a `VarExp` looks up the type of the variable in the type environme
result:

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

Language TYPE1 (continued)

The type of a `LetExp` involves checking the LHS variable list for d
during parsing), determining the types of the RHS expressions us
type environment, extending the current type environment by bindin
to the types of the RHS expressions, and evaluating the type of the `l`
this extended environment.

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

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

```
public TypeEnv addTypeBindings(TypeEnv tenv) {
    List<String> idList = new ArrayList<String>(); /
    for(Token t : varList) // LHS tokens
        idList.add(t.toString());
    Rands rands = new Rands(expList);  // RHS expres
    List<Type> typeList = rands.evalTypeRands(tenv);
    TypeBindings typeBindings = new TypeBindings(idL
    return tenv.extendTypeEnv(typeBindings);
}
```

Language TYPE1 (continued)

The `Rands` object must define the `evalTypeRands` method, which is straigh
that this is where static scope rules come into play: in a `let` expression, the
expressions are evaluated in the enclosing type environment, `tenv`.

```
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
meaning to the type expressions used to declare the types of a proc
parameters and return type. Specifically, we need to take a type expre
the `<typeExp>` grammar rules and evaluate its `Type`: in other wo
tics of a type expression is its `Type`. This is parallel to taking a va
give by `<exp>` grammar rules and evaluating its `Val`. Both of th
are examples of semantics.

For each type expression, we implement a `toType` method that ret
sponding instance of `Type`. One thing to observe is that a type expre
involve any variables, so there is no need to keep track of a type env

Language TYPE1 (continued)

For example, a type expression of the form `[int=>bool]` would evaluate to a P type expression of the form `int` would evaluate to a `IntType`.

Here's the definition of `toType` for the `PrimTypeExp` class (representing ei `bool`) in the code file:

```
PrimTypeExp
%%%
    public Type toType() {
        return primType.toType(); // intType or boo
    }
%%%
```

For the `BoolPrimType` class, which corresponds to grammar rules for the t tokens, we simply return `boolType`.

```
BoolPrimType
%%%
    public Type toType() {
        return Type.boolType; // a singleton in the
    }
%%%
```

Simlar remarks apply to `IntPrimType`.

Language TYPE1 (continued)

For a `ProcTypeExp`, we

0. determine the types (using the `toType` method) of the (formal p expressions that appear to the left of the '=>' token and inse `List<Type>` object,
1. determine the the type of the (return) type expresion that appear the '=>' token, and then
2. construct the appropriate `ProcType` object.

```
ProcTypeExp
%%%
    public Type toType() {
        List<Type> paramTypeList = typeExps.toTypes
        Type returnType = typeExp.toType();
        return new ProcType(paramTypeList, returnTy
    }
%%%
```

The `toTypes()` method in the `TypeExps` class is straight-forwar its `typeExpList` and calling `toType` on each of its type express

Language TYPE1 (continued)

We are now in a position to determine the *defined type* of a `proc` (defined type, we mean the type of the procedure (a `ProcType`) as the declared types of each of the formal parameters and the declar For example,

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

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

However, the type of the procedure *body* is dependent on the type (which the procedure is evaluated, since the procedure body may hav are free with respect to the procedure's formal parameters but tha types defined in an enclosing environment. One of our type checks i the procedure's declared return type is the same as the type of the pı

Language TYPE1 (continued)

The `definedType` method in the `Proc` class evaluates the procedure's defined be a `ProcType`: it gets the declared types of the formal parameters from the `Fo` the declared return type from the procedure's `typeExp`. The defined type is deter by the declared types of the formal parameters and the procedure's declared retur

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

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

```
Formals
%%%
    public List<Type> formalTypeList() {
        List<Type> typeList = new ArrayList<Type>(typeExpList.
        for (TypeExp texp : typeExpList)
            typeList.add(texp.toType());
        return typeList;
    }
%%%
```

Language TYPE1 (continued)

We are now ready to define `evalType` in the `Proc` class.

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

Language TYPE1 (continued)

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

```
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; // same as falseExpType if w
}
```

Language TYPE1 (continued)

What is the type of a `{ ... }` expression? Since the purpose of
is normally to perform side-effects and to return only the value c
expression, we assume that the type of '`{ ... }`' is the type c
expression. The types of the previous sub-expressions are ignored e
must not have type errors.

Thus the expression

```
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 ]`.

Language TYPE1 (continued)

The following method implements type checking in the `SeqExp` cl

```
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
it matches the type of the RHS expression. The following method i
`SetExp` class:

```
public Type evalType(TypeEnv tenv) {
    Type varType = tenv.applyTypeEnv(var);
    Type expType = exp.evalType(tenv);
    Type.checkEquals(varType, expType);
    return varType;
}
```

Language TYPE1 (continued)

A `letrec` expression is treated similar to a `let` expression, excep
of the RHS procedures must be evaluated in an environment that ha
for the LHS variables. Since the defined type of a RHS procedure car
solely by declared types of its parameters and return type, the typ
variables can be determined independent of any environment. So w
LHS variables to extend the type environment and then evalute the ty
procedure *bodies* using this extended environment, for the purpose o
with their declared return types. The `addLetrecTypeBindings`
`LetDecls` class does all the work.

```
public TypeEnv addLetrecTypeBindings(TypeEnv tenv)
    tenv = tenv.extendTypeEnv(new TypeBindings(varL
    Iterator<Token> varIter = varList.iterator();
    Iterator<Exp> expIter = expList.iterator();
    while (varIter.hasNext()) {
        String str = varIter.next().toString();
        Type typ = expIter.next().evalType(tenv);
        tenv.add(new TypeBinding(str, typ));
    }
    return tenv;
}
```

Language TYPE1 (continued)

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

```
LetrecExp
%%%
    public Type evalType(TypeEnv tenv) {
        TypeEnv ntenv = letDecls.addLetrecTypeBindi
        return exp.evalType(ntenv);
    }
%%%
```

Language TYPE1 (continued)

Two to go: AppExp and PrimappExp. Both of these are similar i
to check if the formal parameter types of the operator agree in nu
with the actual parameter (operand) types.

For an AppExp, we must ensure that the thing we are applying is ac
dure – it must evaluate to a ProcType. We then check that the ac
expression types match the declared types of the formal paramete
ProcType object already knows (and has checked the validity of)
the type of an application is simply the return type of the procedure.

```
AppExp
%%%
    public Type evalType(TypeEnv tenv) {
        Type tt = exp.evalType(tenv);
        ProcType pt = tt.procType(); // make sure tt
        List<Type> argTypeList = rands.evalTypeRands
        Type.checkEqualTypes(pt.paramTypeList, argTy
        return pt.returnType;
    }
%%%
```

Language TYPE1 (continued)

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

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

Notice that the type of a primitive is defined statically. T
definedType method just retrieves the type, which must be a Pr

Language TYPE1 (continued)

We are left to handle top-level `defines`. While this seems to be st
we are faced with problems of unbound variables. For example, supp
to make the follwing definitions:

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

When the procedure body of `odd?` is first encountered, the vari
unbound, so the type of the body of the procedure cannot be evaluate
to type-check `odd?` fails with a type error.

We can remedy this situation in two ways:

- forbid top-level `defines` altogether; or
- create a mechanism to deal with unbound variables that m `defines` work.

Language TYPE1 (continued)

We take the second approach, with some restrictions.

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

Pascal provides such a mechanism with *forward* declarations, as do
procedure declarations. Java does not suffer from this problem, in p
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>
                      Declare(Token var, TypeExp typeE
```

Language TYPE1 (continued)

The rules for top-level `define`s and `declare`s are:

0. A top-level identifier definition serves as a declaration for the pu
   mining the type of the identifier.
1. It is an error for a top-level identifier to be declared (hence defi
   once.
2. If an identifier declaration precedes its definition, it is an error
   and defined types are not identical.
3. It is an error for a declaration not to have a subsequent definitio
   error need not be checked.

Although you cannot `define` a top-level variable more than once, y
a top-level variable using `set` if the variable has been previously c
type rule for variable assignment still requires that the RHS type o
agree with the declared type of the variable.

Language TYPE1 (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))) % o
```

5.
```
declare f : [=>bool]
define y = 2
define f = proc():int y
  % error - defined type of f does not match decl
```

6.
```
declare f : [=>bool]
define y = 2
define f = proc():bool y
  % error - body type y does not match declared r
```

Language TYPE1 (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 bindin
dure application `.f()` results in a run-time error.

If we subsequently give a value definition for `y`, the procedure applic
as the following shows:

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

Language TYPE1 (continued)

We implement top-level declarations as follows:

```
Declare
%%%
    // calling $run may trigger a modificaiton
    // of the initial type environment
    public void $run() {
        TypeEnv tenv = Program.tenv; // top-level type environ
        Type tv; // variable's declared type
        String sym = var.toString(); // the LHS symbol
        try {
            // look up the variable in the initial type enviro
            tv = tenv.applyTypeEnv(sym);
        } catch (PLCCException e) {
            // no type binding -- must be a new type declarati
            // that we add to the top-level type environment
            tv = typeExp.toType();
            tenv.add(new TypeBinding(sym, tv)); // type bindin
            System.out.println(sym + ":" + tv);
            return;
        }
        throw new PLCCException(sym + ": duplicate variable de
    }
%%%
```

If a type binding (either through a declaration of definition) already exists, it is flag
declaration. Otherwise a top-level binding is made between the identifier and the ty
its type expression. The `$run` behavior of a variable declaration/definition is to d
being declared/defined along with its declared type.

Language TYPE1 (continued)

We expand on top-level definitions as follows. Notice that this code is in two par
where the variable hasn't been declared/defined before:

```
Define
%%%
    // calling $run may trigger a modification
    // of the initial type and value environments
    public void $run() {
        Env env = Program.env;       // top-level environment
        TypeEnv tenv = Program.tenv; // top-level type environ
        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 {
            // look up the LHS variable in the initial type en
            lhst = tenv.applyTypeEnv(sym);
        } catch (RuntimeException e) {
            // no type binding -- must be a new variable defir
            rhst = exp.evalType(tenv);
            tenv.add(new TypeBinding(sym, rhst));        // ty
            rhsv = exp.eval(env);
            env.add(new Binding(sym, new ValRef(rhsv))); // va
            System.out.println(sym + ":" + rhst);
            return
        }
        ...
%%%
```

Language TYPE1 (continued)

The second part is where the variable has a declaration but possibly not a definitio

```
Define
%%%
    ...
        // the variable has a declared type, lhst -- see if it
        try {
            // look up the value of var in the initial enviror
            lhsv = env.applyEnv(sym);
        } catch (RuntimeException e) {
            // the variable has a declared type, but no value
            // so we want to add the value binding to the top-
            // 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)));
            System.out.println(sym + ":" + lhst);
            return;
        }
        // the variable has a value, so must be a duplicate
        throw new RuntimeException(sym + ": duplicate variable
    }
%%%
```

Language TYPE1 (continued)

The code on the previous slide implements a top-level definition, bir
an identifier in the initial environment. According to our type rules,
ing already exists, it is flagged as an error. If a declared type binding
a value binding, the expression type is checked for conformance wi
type. If a declared type binding does not exist, the top-level type
extended to include the type binding. In case there are no errors, the
environment is extended to include the value binding.

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

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