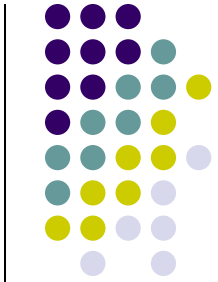
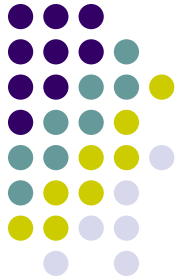
Type system implementation



- We extend our simple3 language to simple4 with the addition of a type system with three types:
  - int
  - float
  - string
- We also assume that int is a subtype of float and float is a subtype of string, that is, a compiler/interpreter is allowed to insert widening conversions and should flag errors for narrowing conversions.

Type system implementation

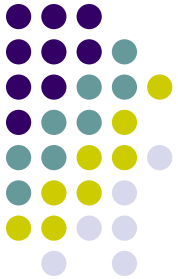


- We want to be able to write programs such as these:

```
int inc(int x) return x+1;
int y = inc(3);
put "the result is", y;
```

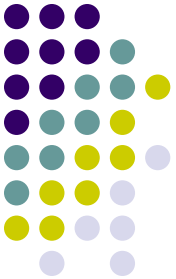
```
float pow(float b,int p) {
   if (p == 0)
      return 1.0;
   else
      return b*pow(b,p-1);
}

float v;
get v;
int p;
get p;
float result = pow(v,p);
put v,” to the power of “,p,” is “,result;
```

Type system implementation: Syntax



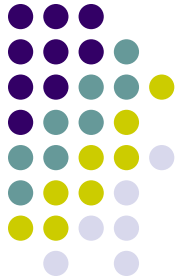
```
prog            :       stmt+;

stmt            :       dataType VAR '(' formalParamList? ')' stmt  // declare a function
                |       dataType VAR ('=' exp)? ';' // declare variable in current scope with optional initializer
                |       VAR '=' exp ';'                 // assign value to variable
                |       'get' (prompt ',')? VAR ';'     // prompt user for a value and assign it to variable
                |       'put' exp (',' exp)* ';'        // print out value(s) to terminal
                |       VAR '(' actualParamList? ')' ';'  // function call statement
                |       'return' exp? ';'
                |       'while' '(' exp ')' stmt
                |       'if' '(' exp ')' stmt ('else' stmt)?
                |       '{' stmt+ '}'                   // block statement (new local scope)
                ;

dataType        :       'int'
                |       'float'
                |       'string'
                ;

formalParamList
                :       dataType VAR (',' dataType VAR)*
                ;

actualParamList
                :       exp (',' exp)*
                ;
```

Type system implementation: Syntax



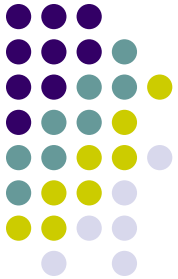
```
prompt      :       string;

exp         :       relexp;
relexp      :       addexp (('==' addexp) |('<=' addexp))*;
addexp      :       mulexp (('+' mulexp) | ('-' mulexp))*;
mulexp      :       atom (('*' atom) | ('/' atom))*;

atom        :       '(' exp ')'
            |       VAR '(' actualParamList? ')'  // function call within an expression
            |       VAR
            |       '-'? INT
            |       '-'? FLOAT
            |       string
            ;
```

Type system implementation: Semantics



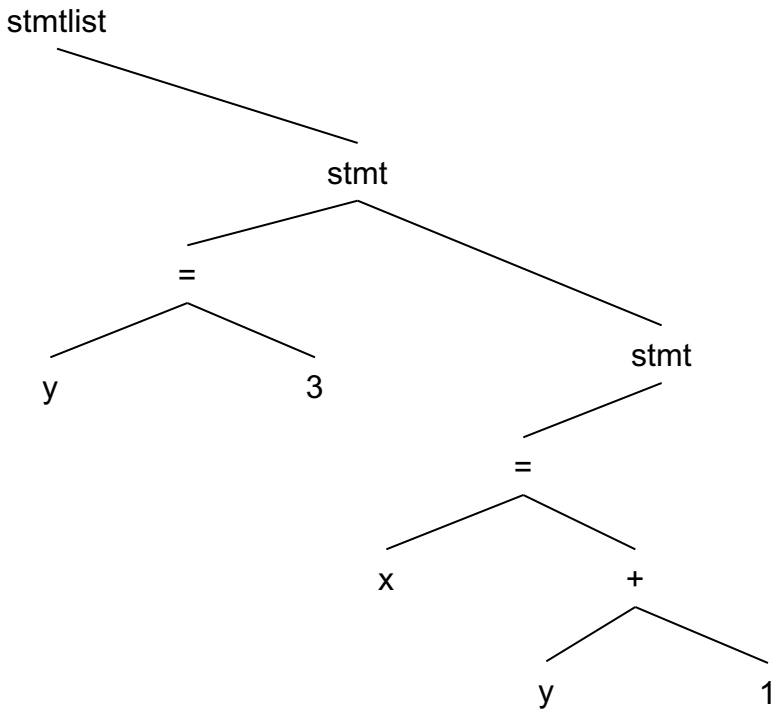
- At the semantic level we *annotate* all ASTs with type information
- We use *type propagation* to check that expressions/statements are properly typed.
  - Type propagation is the systematic tagging of an AST from leafs up with type information.

Type system implementation: Semantics

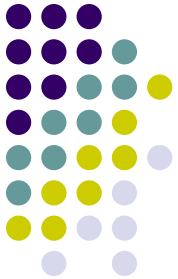


- Consider the simple example:

```
int y;
int x;
y = 3;
x = y + 1;
```

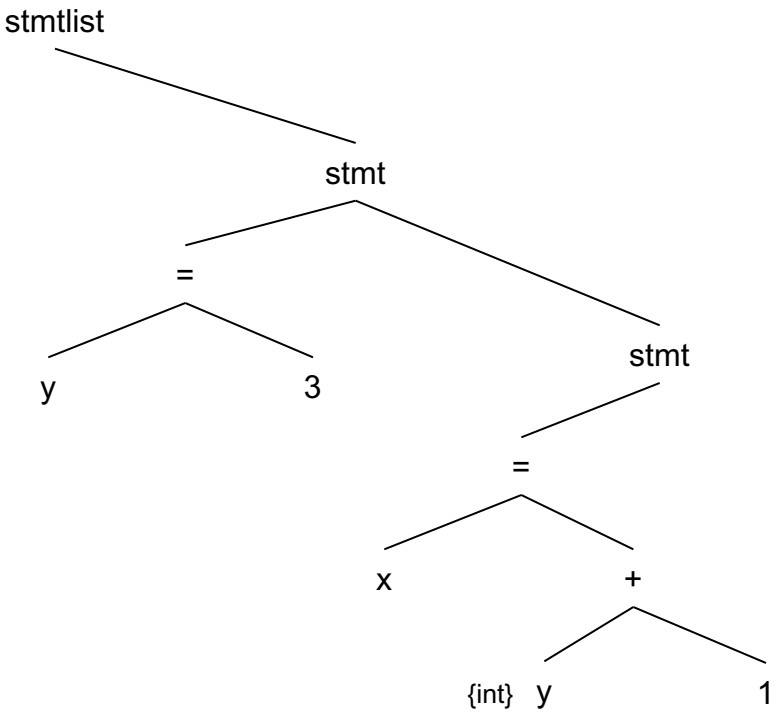


Type system implementation: Semantics

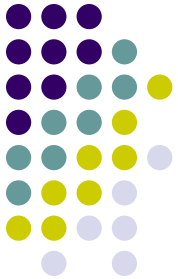


- Consider the simple example:

```
int y;
int x;
y = 3;
x = y + 1;
```

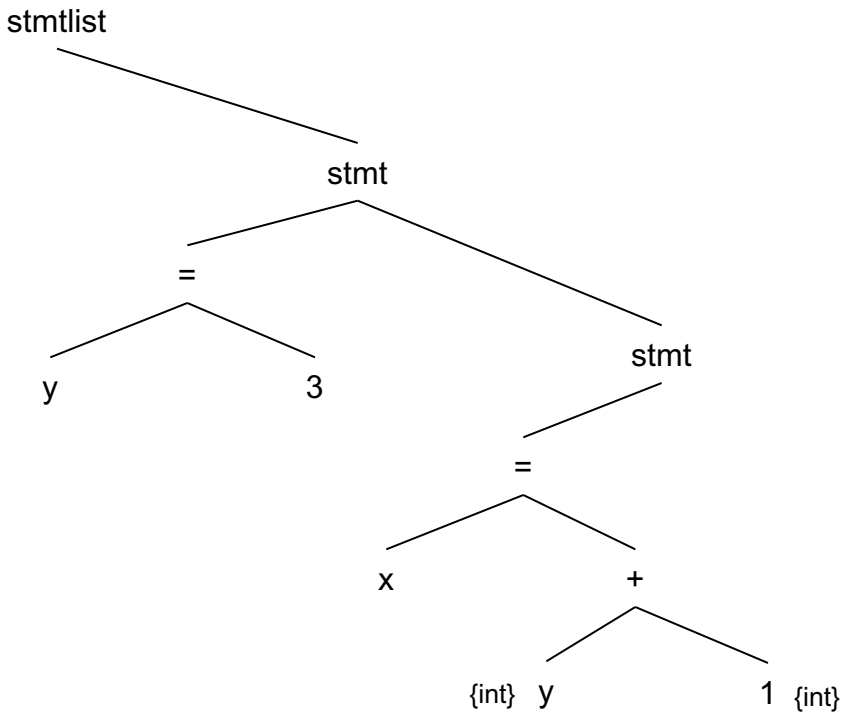


Type system implementation: Semantics

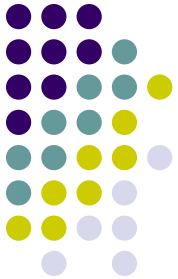


- Consider the simple example:

```
int y;
int x;
y = 3;
x = y + 1;
```

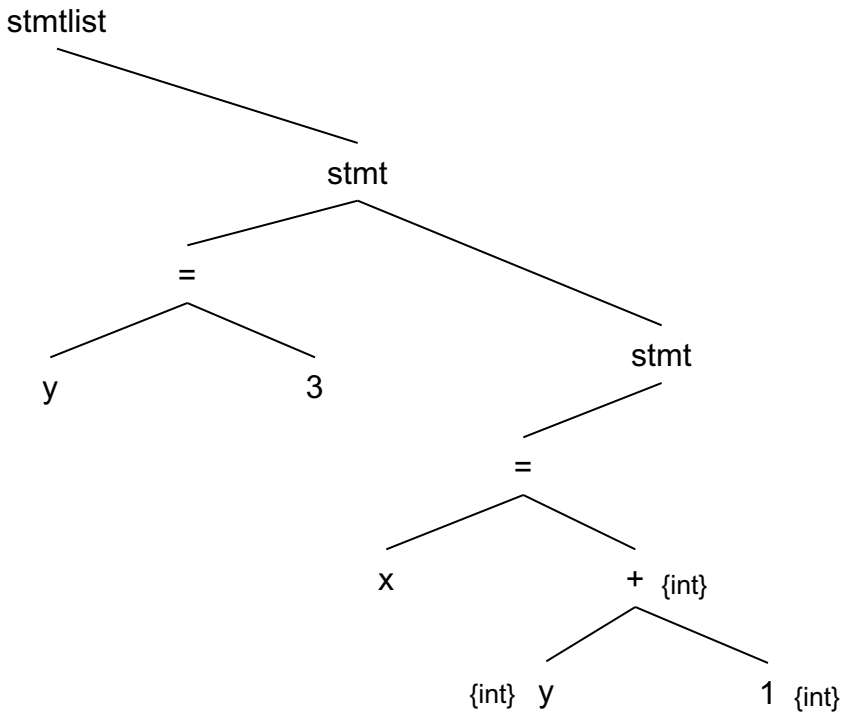


Type system implementation: Semantics

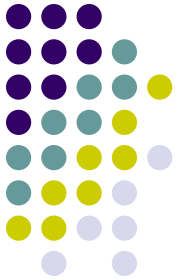


- Consider the simple example:

```
int y;
int x;
y = 3;
x = y + 1;
```

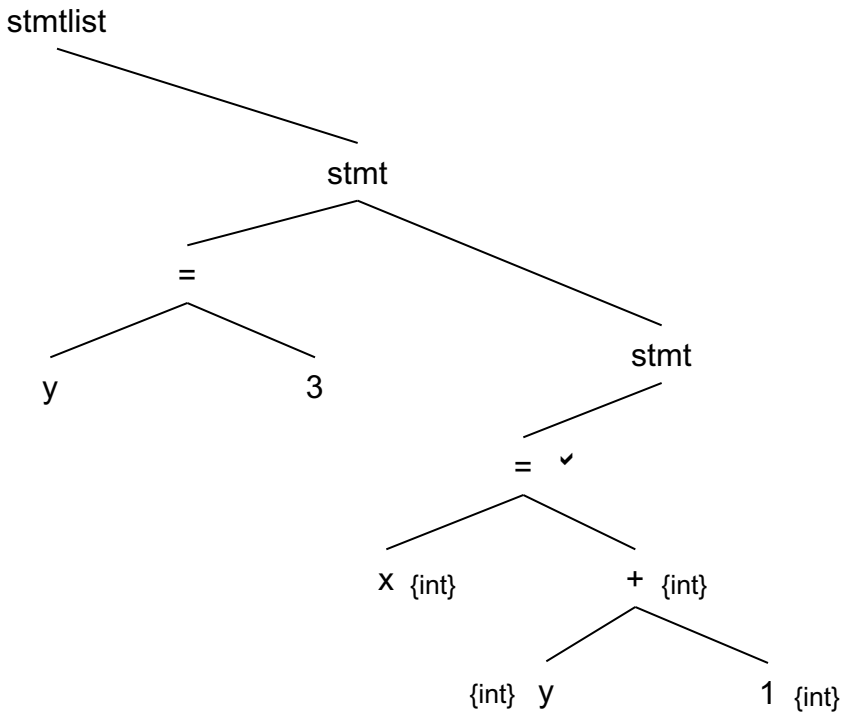


Type system implementation: Semantics

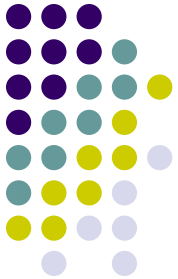


- Consider the simple example:

```
int y;
int x;
y = 3;
x = y + 1;
```

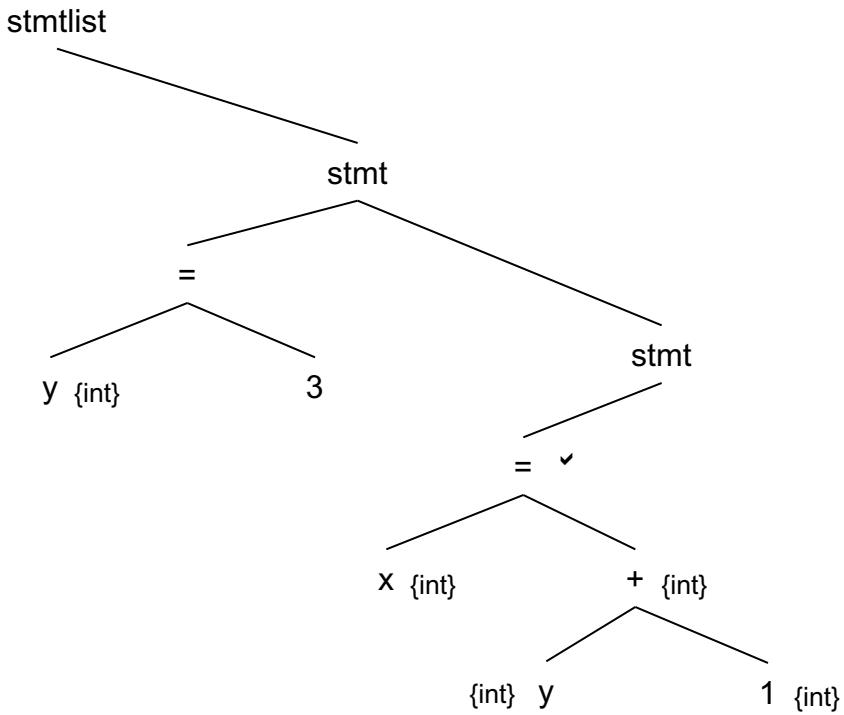


Type system implementation: Semantics

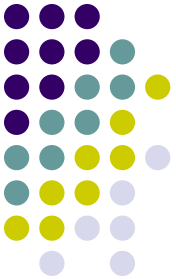


- Consider the simple example:

```
int y;
int x;
y = 3;
x = y + 1;
```

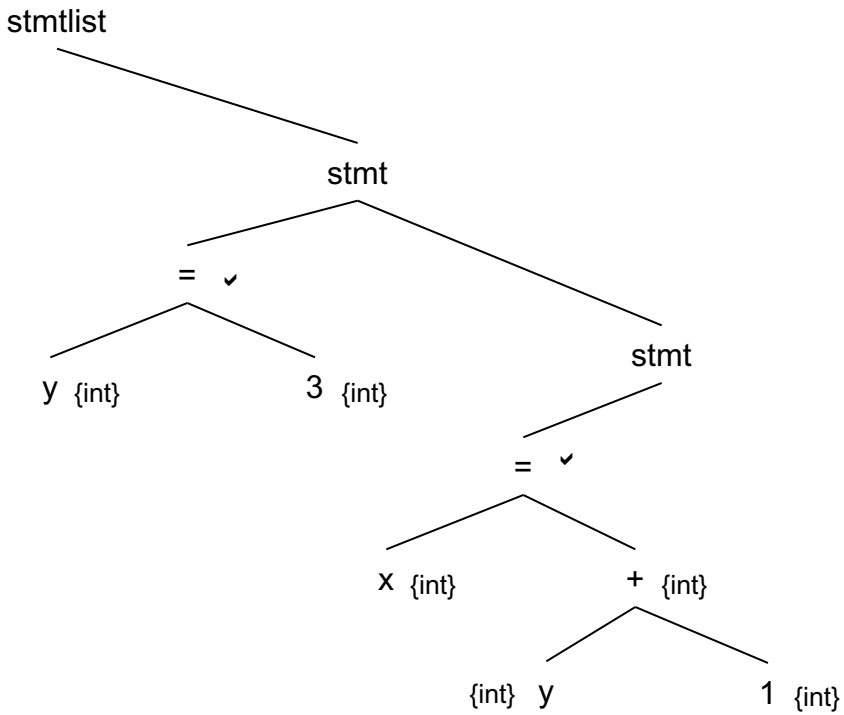


Type system implementation: Semantics

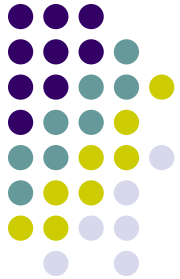


- Consider the simple example:

```
int y;
int x;
y = 3;
x = y + 1;
```

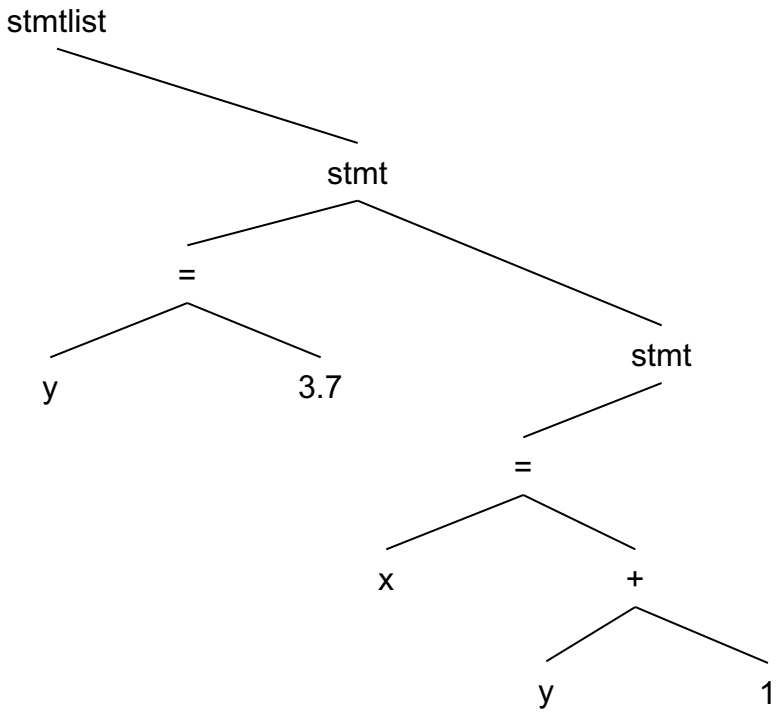


Type system implementation: Semantics

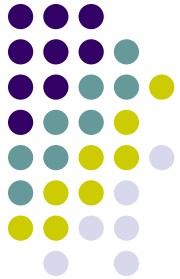


- Consider this example which has a typecheck error:

```
float y;
int x;
y = 3.7;
x = y + 1;
```

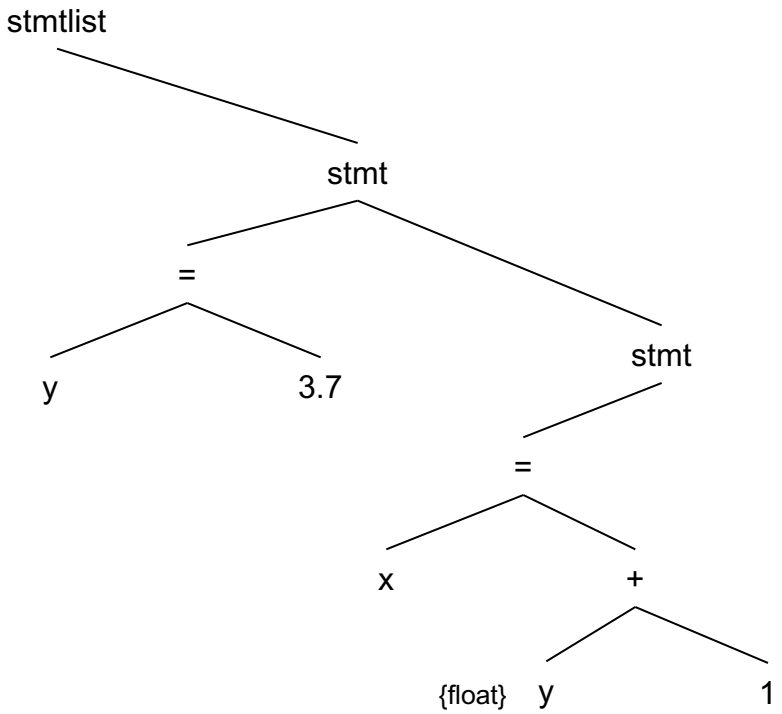


Type system implementation: Semantics

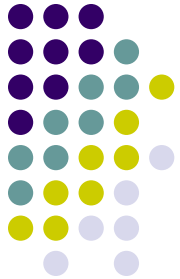


- Consider this example which has a typecheck error:

```
float y;
int x;
y = 3.7;
x = y + 1;
```

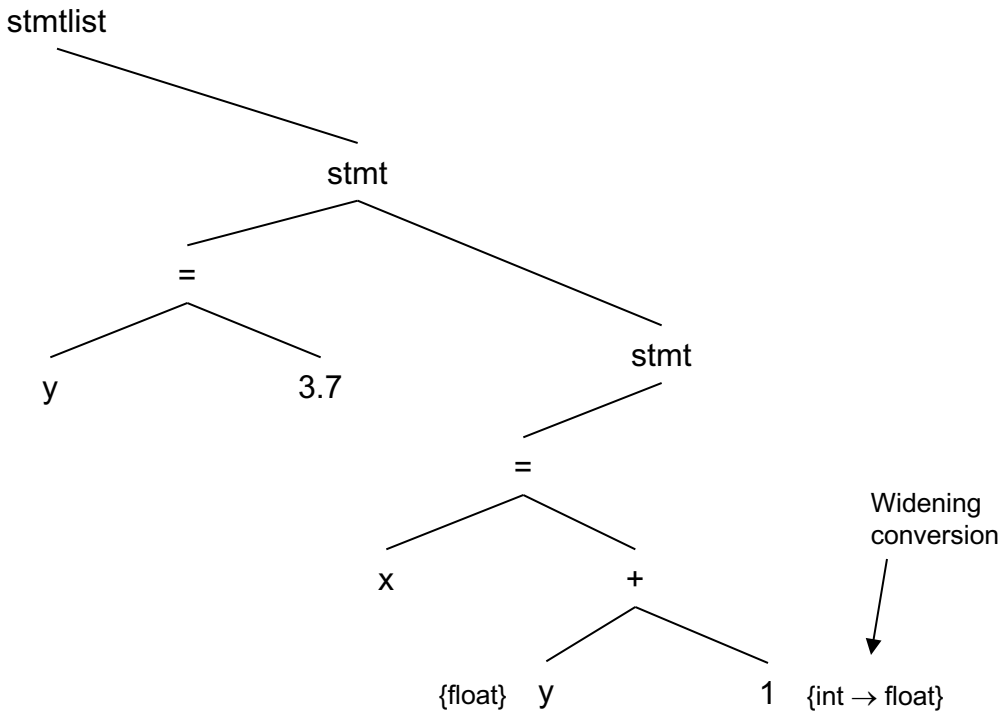


Type system implementation: Semantics

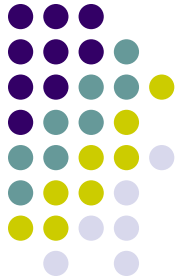


- Consider this example which has a typecheck error:

```
float y;
int x;
y = 3.7;
x = y + 1;
```

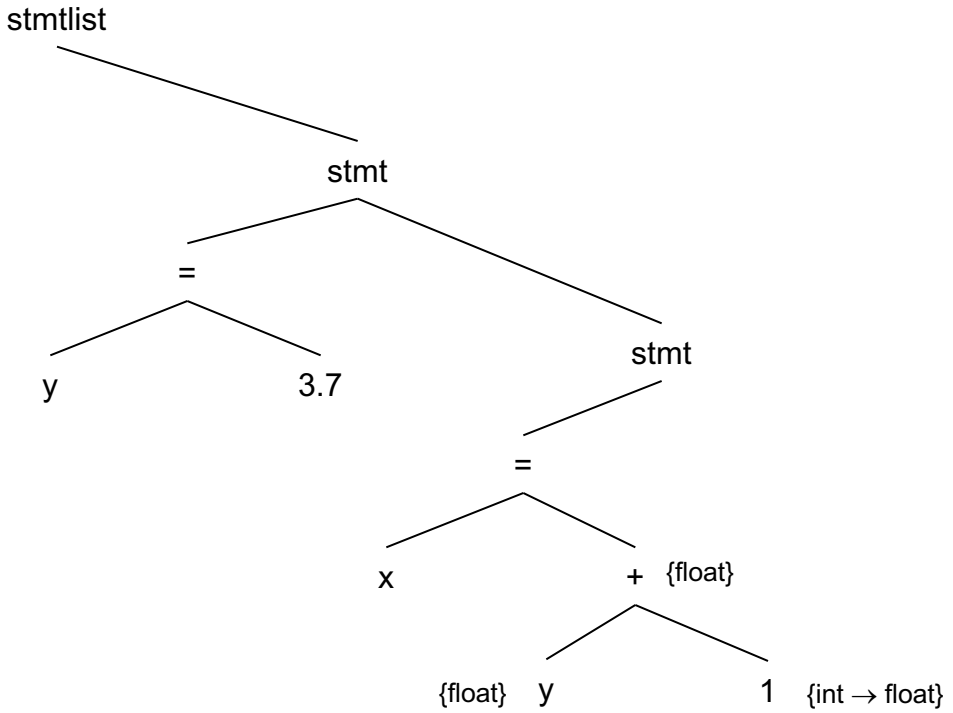


Type system implementation: Semantics

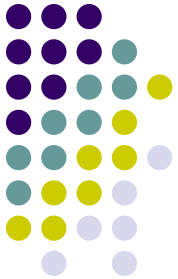


- Consider this example which has a typecheck error:

```
float y;
int x;
y = 3.7;
x = y + 1;
```

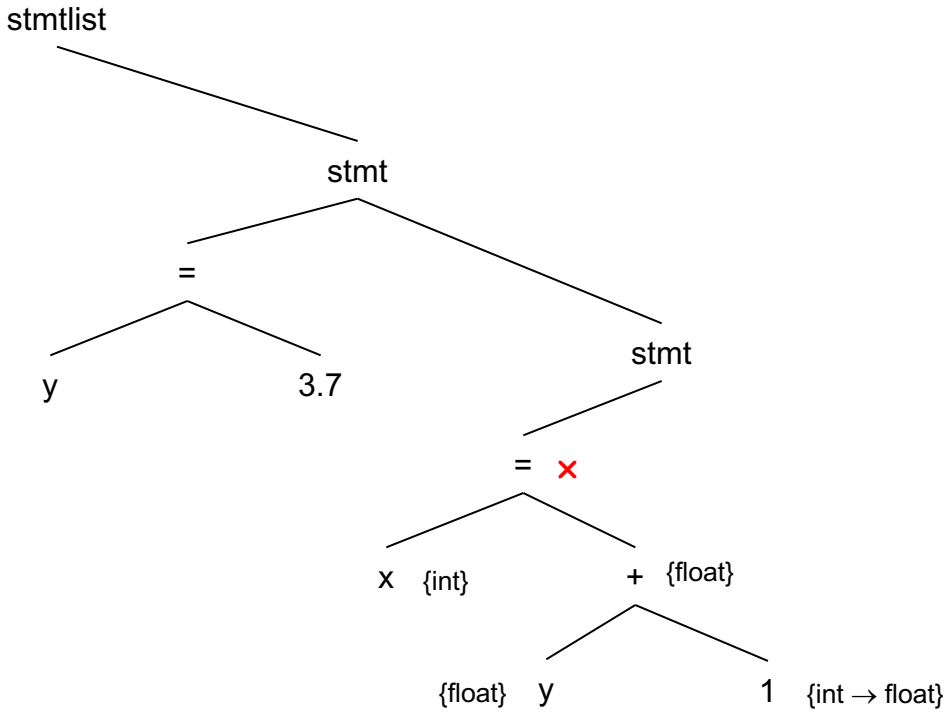


Type system implementation: Semantics

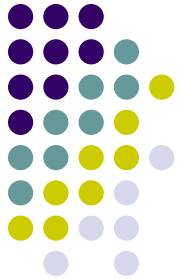


- Consider this example which has a typecheck error:

```
float y;
int x;
y = 3.7;
x = y + 1;
```

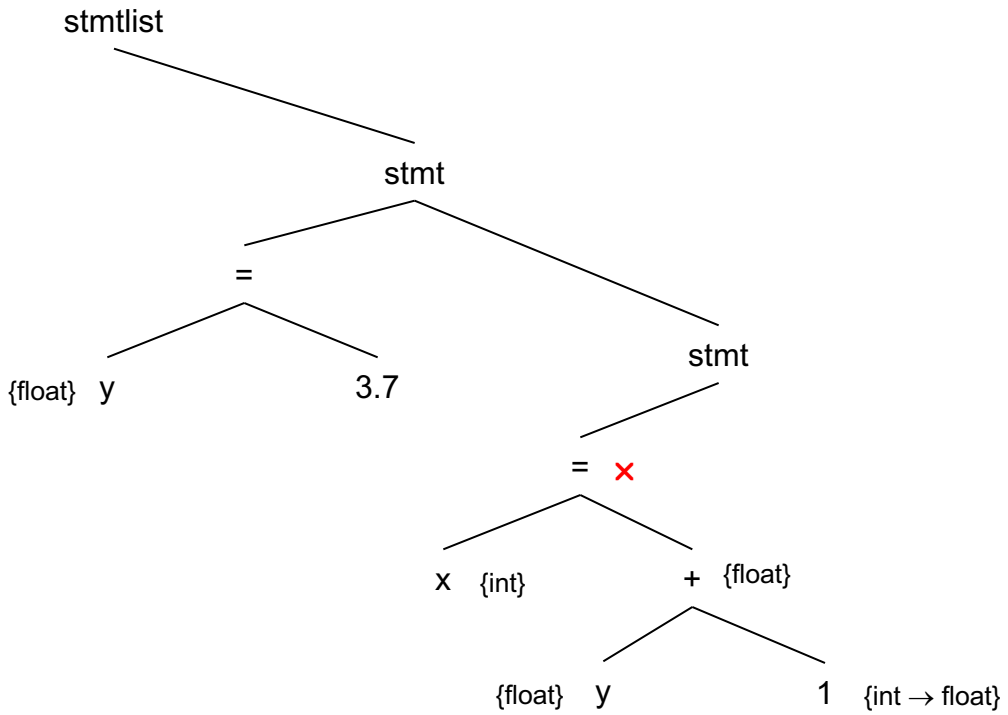


Type system implementation: Semantics

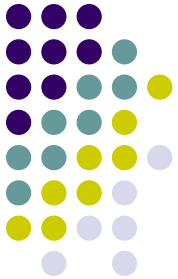


- Consider this example which has a typecheck error:

```
float y;
int x;
y = 3.7;
x = y + 1;
```

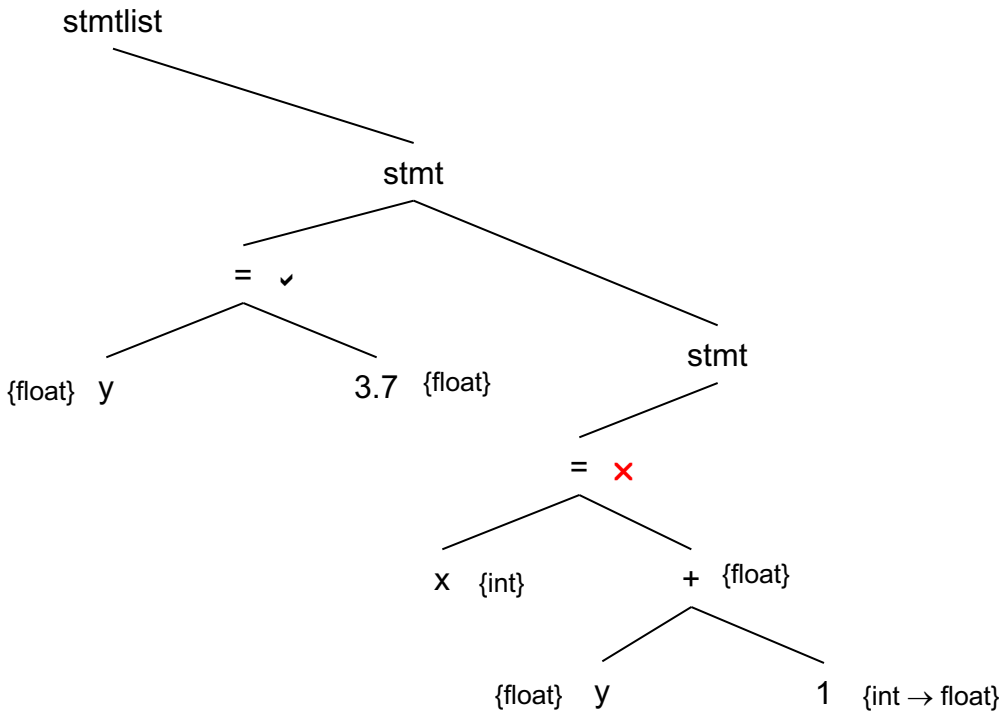


Type system implementation: Semantics

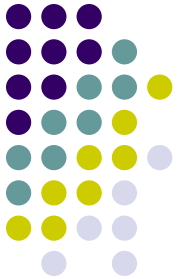


- Consider this example which has a typecheck error:

```
float y;
int x;
y = 3.7;
x = y + 1;
```

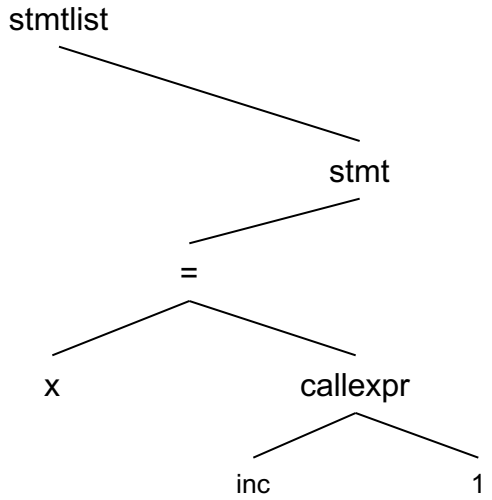


Type system implementation: Semantics

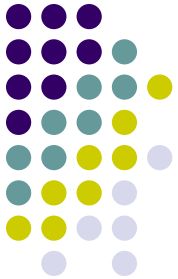


- Here is an example with a function call:

```
int inc(int i) return i+1;
int x;
x = inc(1);
```

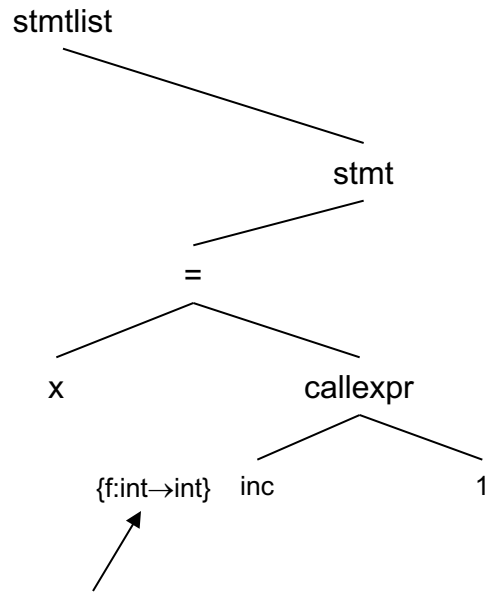


Type system implementation: Semantics



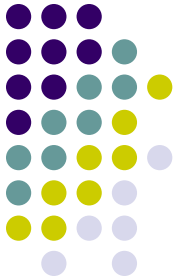
- Here is an example with a function call:

```
int inc(int i) return i+1;
int x;
x = inc(1);
```



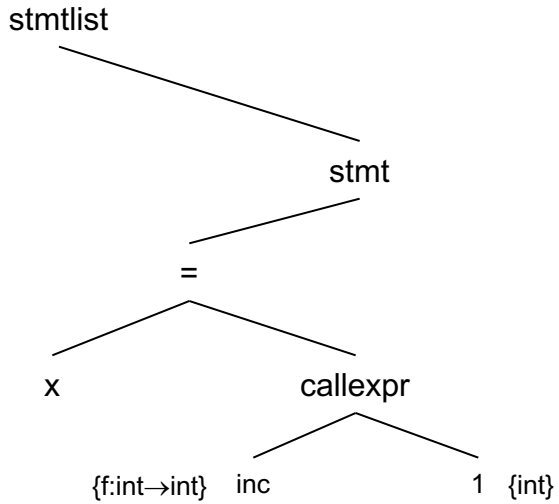
We have to track function symbols, both for their formal parameter types and return types.

Type system implementation: Semantics

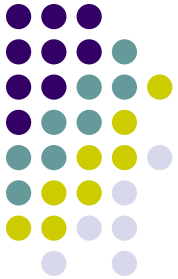


- Here is an example with a function call:

```
int inc(int i) return i+1;
int x;
x = inc(1);
```

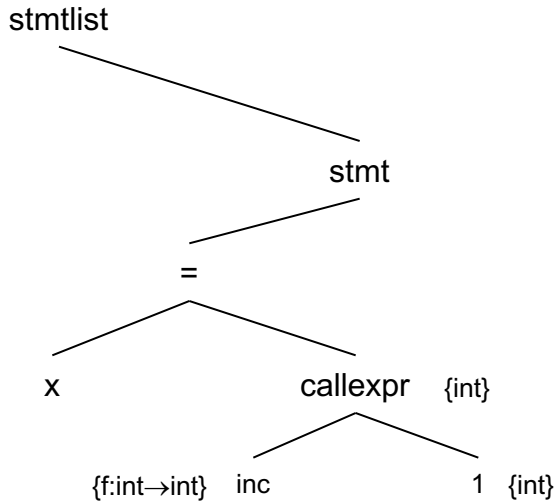


Type system implementation: Semantics

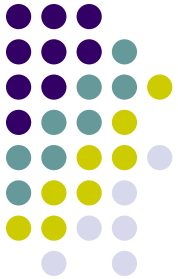


- Here is an example with a function call:

```
int inc(int i) return i+1;
int x;
x = inc(1);
```

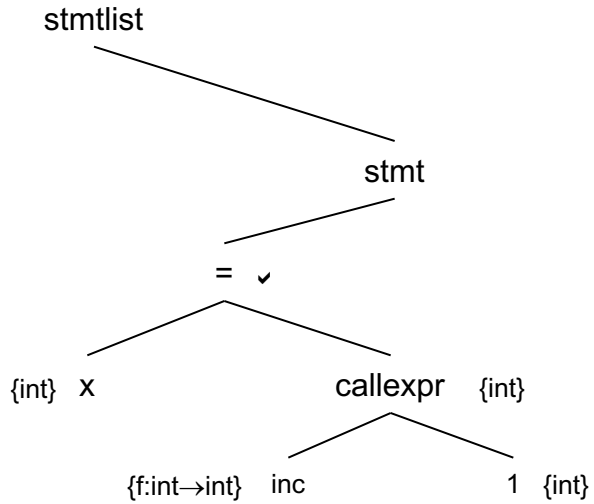


Type system implementation: Semantics

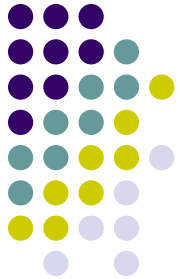


- Here is an example with a function call:

```
int inc(int i) return i+1;
int x;
x = inc(1);
```

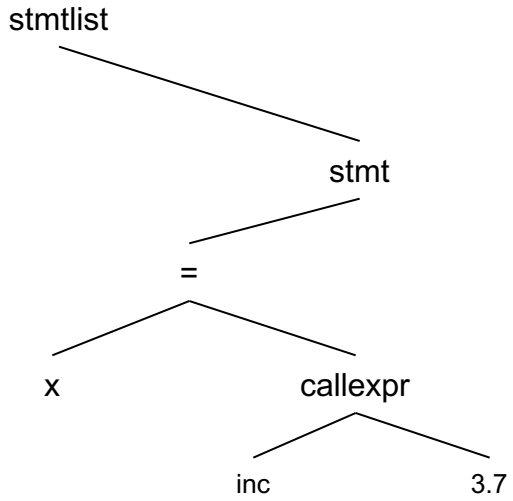


Type system implementation: Semantics

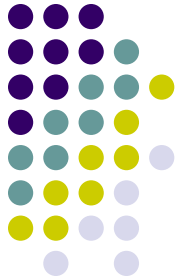


- Here is an example with a function call and a type error:

```
int inc(int i) return i+1;
int x;
x = inc(3.7);
```

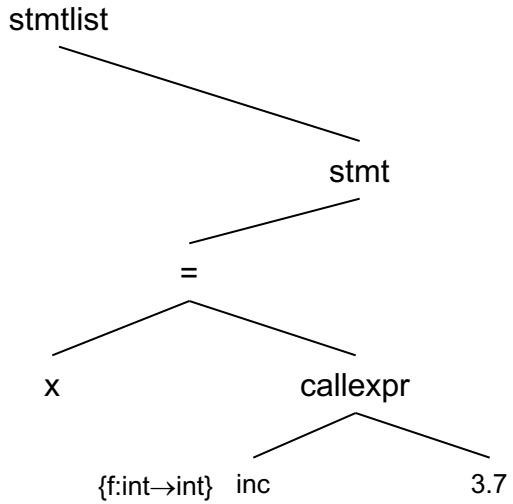


Type system implementation: Semantics

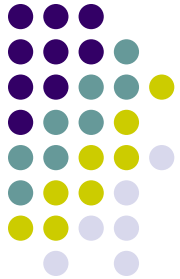


- Here is an example with a function call and a type error:

```
int inc(int i) return i+1;
int x;
x = inc(3.7);
```

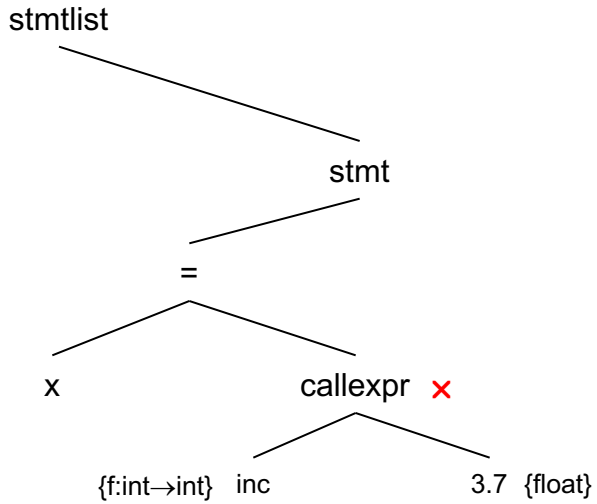


Type system implementation: Semantics

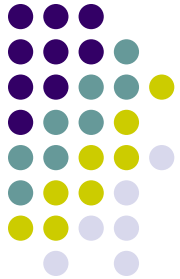


- Here is an example with a function call and a type error:

```
int inc(int i) return i+1;
int x;
x = inc(3.7);
```

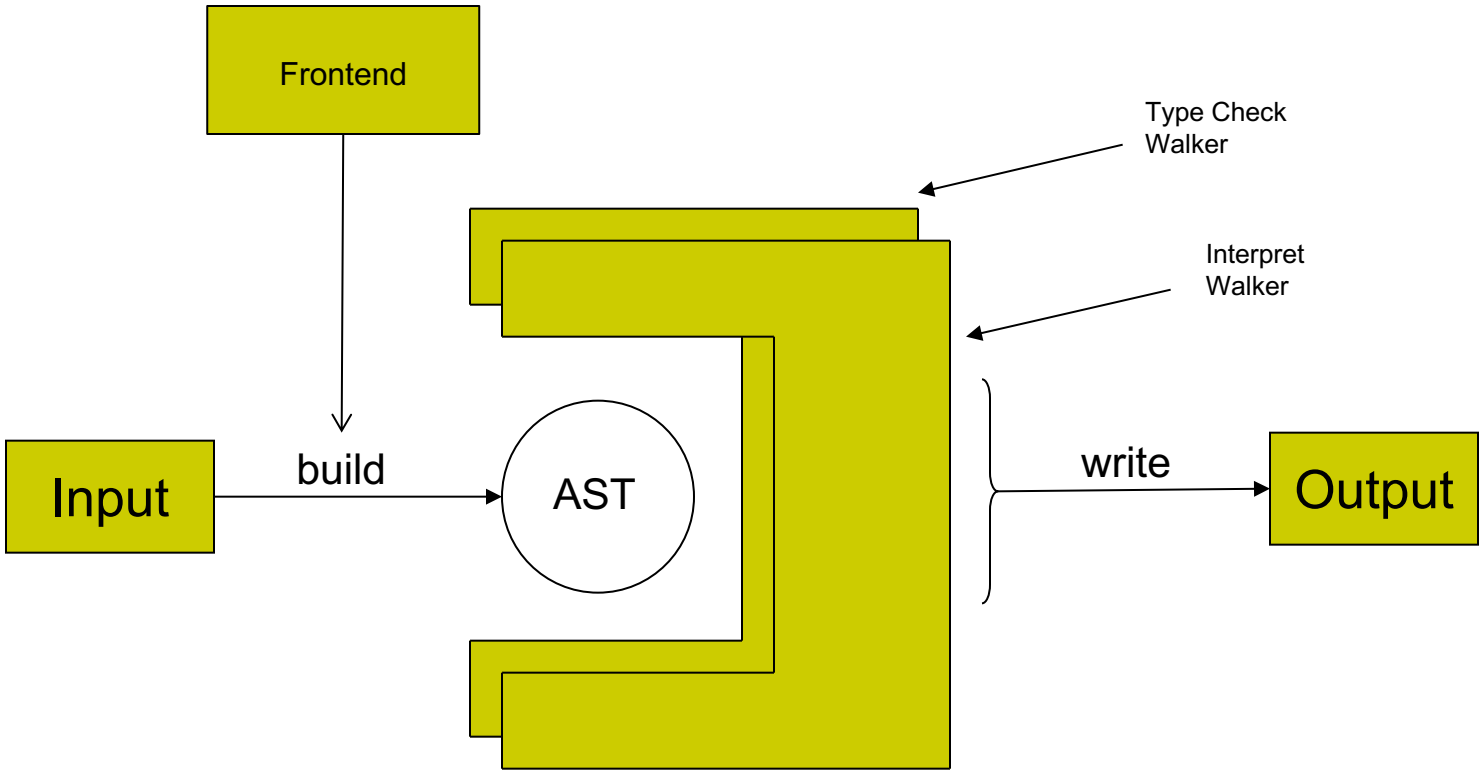


Type system implementation

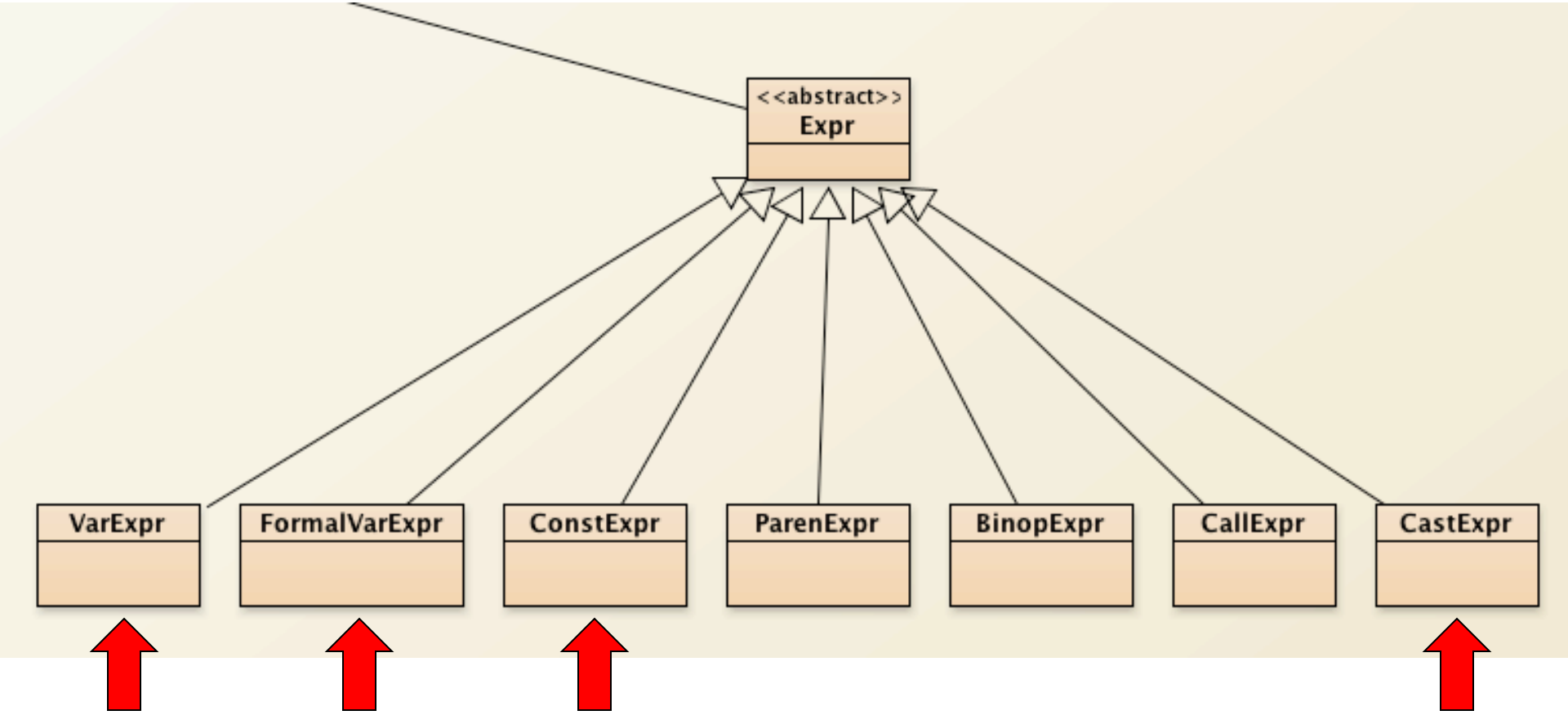


- To implement the type system we introduce the following type tags in the interpreter / symbol table
  - int
  - float
  - string
  - function
- We implement a *static type checker* as a separate walker in the interpreter

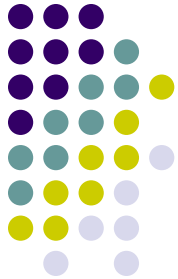
Architecture of our Interpreter



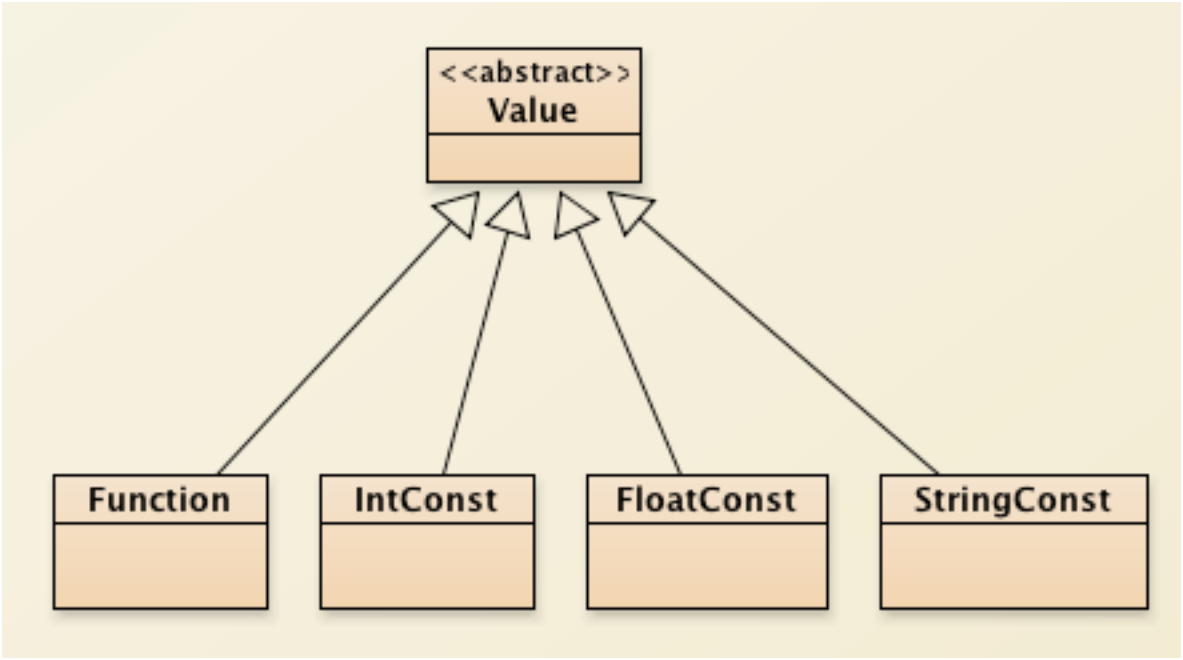
Extended AST



Extended AST



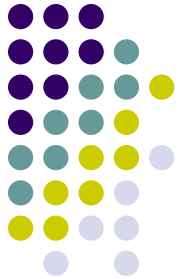
- In our previous implementations we only had function values and integer scalars
- We now have additional value types



The Value Base Class

```
abstract class Value {
    public static final int NOTYPE = -1;
    public static final int INTEGER = 0;
    public static final int FLOAT = 1;
    public static final int STRING = 2;
    public static final int FUNCTION = 3;

    // Type Promotion Table
    // This table implements the following type hierarchy:
    //    int < float < string
    // Note: functions are not allowed to appear
    // in the context of any operations.
    private static int[][] typeArray = {
    // INTEGER   FLOAT    STRING    FUNCTION
    //-----------------------------------------
    {  INTEGER,  FLOAT,   STRING,   NOTYPE },   // INTEGER
    {  FLOAT,    FLOAT,   STRING,   NOTYPE },   // FLOAT
    {  STRING,   STRING,  STRING,   NOTYPE },   // STRING
    {  NOTYPE,   NOTYPE,  NOTYPE,   NOTYPE }    // FUNCTION
    };

    public static int getResultType(int lt,int rt) {
        if (lt == NOTYPE || rt == NOTYPE)
            return NOTYPE;
        else
            return typeArray[lt][rt];
    }

    // every derived class needs to implement the following behavior
    public abstract int getType();
    public abstract String toString();
}
```

Value Classes

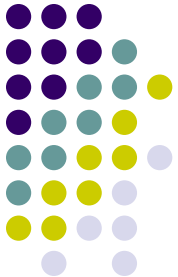


```
class IntConst extends Value {

    private Integer value;

    public IntConst(String value) {
        this.value=new Integer(Integer.parseInt(value));
    }

    public IntConst(Integer value) {
        this.value=value;
    }

    public Integer getValue() {
        return value;
    }

    public int getType() {
        return Value.INTEGER;
    }

    public String toString() {
        return value.toString();
    }
}
```

```
class StringConst extends Value {

    private String value;

    public StringConst(String value) {
        this.value=value;
    }

    public String getValue() {
        return value;
    }

    public int getType() {
        return Value.STRING;
    }

    public String toString() {
        return value;
    }
}
```

Symbol Ta

```
public class SymbolTableScope {
    // scope stack is built as a linked list
    private SymbolTableScope parentScope = null;

    // function value, if this is the scope of a function call
    // otherwise null
    Function function = null;

    // symbols are kept in a hashmap indexed by their name
    // their initialization value depends on their kind:
    //    integer,float, string constants, function values
    //
    private HashMap<String,Value> value = new HashMap<String,Value>();

    public SymbolTableScope(SymbolTableScope parentScope) {
                this.parentScope = parentScope;
    }

    public SymbolTableScope getParentScope() {
                return parentScope;
    }

    public void setParentScope(SymbolTableScope parentScope) {
                this.parentScope = parentScope;
    }

    public void enterSymbol(String name, Value value) {
                this.value.put(name,value);
    }

    public Value lookupSymbol(String name) {
                return value.get(name);
    }

    public void setFunctionValue(Function value) {
                function = value;
    }

    public Function getFunctionValue() {
                return function;
    }
}
```

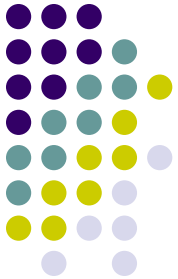


The Reader

```
stmt returns [Stmt ast]
        // function declarations can have a void return type...
    :   'void' VAR '(' ')' s=stmt
          { $ast = new FuncDeclStmt($VAR.text,new Function(Value.NOTYPE,new ArgList(),$s.ast)); }
    |   dt=dataType VAR '(' ')' s=stmt
          { $ast = new FuncDeclStmt($VAR.text,new Function($dt.type,new ArgList(),$s.ast)); }
    |   'void' VAR '(' l=formalParamList ')' s=stmt
          { $ast = new FuncDeclStmt($VAR.text,new Function(Value.NOTYPE,$l.ast,$s.ast)); }
    |   dt=dataType VAR '(' l=formalParamList ')' s=stmt
          { $ast = new FuncDeclStmt($VAR.text,new Function($dt.type,$l.ast,$s.ast)); }
    |   dt=dataType VAR '=' exp ';'
          { $ast = new VarDeclStmt($dt.type,$VAR.text,$exp.ast); }
    |   dt=dataType VAR ';'
          { $ast = new VarDeclStmt($dt.type,$VAR.text,new ConstExpr(new IntConst("0"))); }
    |   VAR '=' exp ';'
          { $ast = new AssignStmt($VAR.text,$exp.ast); }
    |   'get' prompt ',' VAR ';'
          { $ast = new GetStmt($prompt.text,$VAR.text); }
    |   'get'  VAR ';'
          { $ast = new GetStmt("",$VAR.text); }
    |   'put' argList ';'
          { $ast = new PutStmt($argList.ast); }
    |   VAR '(' l=actualParamList ')' ';'
          { $ast = new CallStmt($VAR.text,$l.ast);}

…
```

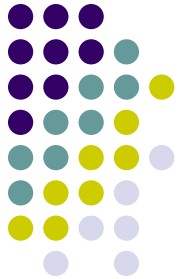
```
dataType returns [int type]
    :   'int'       {$type=Value.INTEGER;}
    |   'float'     {$type=Value.FLOAT;}
    |   'string'    {$type=Value.STRING;}
    ;
```

The Reader



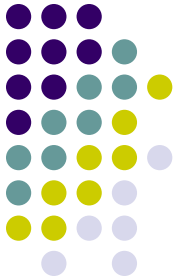
```
atom returns [Expr ast]
    :   '(' exp ')'                          { $ast = new ParenExpr($exp.ast); }
    |   VAR '(' l=actualParamList ')'  { $ast = new CallExpr($VAR.text,$l.ast);}
    |   VAR '(' ')'                           { $ast = new CallExpr($VAR.text);}
    |   VAR                                    { $ast = new VarExpr($VAR.text); }
    |   '-' INT                                { $ast = new ConstExpr(new IntConst('-'+$INT.text)); }
    |   INT                                   { $ast = new ConstExpr(new IntConst($INT.text)); }
    |   '-' FLOAT                           { $ast = new ConstExpr(new FloatConst('-'+$FLOAT.text)); }
    |   FLOAT                                { $ast = new ConstExpr(new FloatConst($FLOAT.text)); }
    |   string                                { $ast = new ConstExpr(new StringConst($string.text)); }
    ;
```

The Type Check Walker



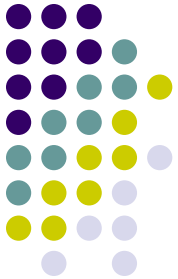
- The type check walker looks like an interpreter…
- …but it computes types instead of values.

The Type Check Visitor



```
// the dispatcher for the type check visitor  - returns a type tag
    public int dispatch(AST ast) {
    if      (ast.getClass() == AssignStmt.class) return interp((AssignStmt)ast);
    else if (ast.getClass() == BlockStmt.class) return interp((BlockStmt)ast);
    else if (ast.getClass() == GetStmt.class) return interp((GetStmt)ast);
    else if (ast.getClass() == IfStmt.class) return interp((IfStmt)ast);
    else if (ast.getClass() == PutStmt.class) return interp((PutStmt)ast);
    else if (ast.getClass() == WhileStmt.class) return interp((WhileStmt)ast);
    else if (ast.getClass() == StmtList.class) return interp((StmtList)ast);
    else if (ast.getClass() == BinopExpr.class) return interp((BinopExpr)ast);
    else if (ast.getClass() == ConstExpr.class) return interp((ConstExpr)ast);
    else if (ast.getClass() == ParenExpr.class) return interp((ParenExpr)ast);
    else if (ast.getClass() == VarExpr.class) return interp((VarExpr)ast);
    else if (ast.getClass() == FuncDeclStmt.class) return interp((FuncDeclStmt)ast);
    else if (ast.getClass() == VarDeclStmt.class) return interp((VarDeclStmt)ast);
    else if (ast.getClass() == CallStmt.class) return interp((CallStmt)ast);
    else if (ast.getClass() == CallExpr.class) return interp((CallExpr)ast);
    else if (ast.getClass() == ReturnStmt.class) return interp((ReturnStmt)ast);
    else {
                    System.err.println("Error (InterpVisitor): unknown class type");
       System.exit(1);
       return Value.NOTYPE;
    }
    }
```

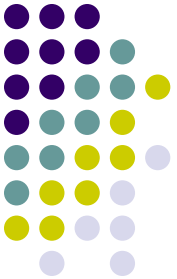
The Type Check Visitor



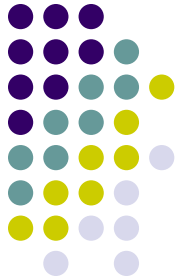
```
// assignment statements
   private int interp(AssignStmt ast) {

   // typecheck the expression
   int exprType = this.dispatch(ast.getAST(0));
   // get the type of the variable
   int varType = Interpret.symbolTable.lookupSymbol(ast.lhsVar()).getType();

   // types compatible?
   int resultType = Value.getResultType(varType,exprType);

   // check for type errors
   if (resultType == Value.NOTYPE ||
       resultType != varType) {  // second condition means: assigning supertype to subtype
       System.err.println("Error (assignmentstmt): expression type "+resultType+" cannot be assigned to variable of type "+varType);
       System.exit(1);
       return Value.NOTYPE;
   }

   // check if we have to insert a type promotion
   if (resultType != exprType) {
       AST newAst = new CastExpr(exprType,resultType,(Expr)ast.getAST(0));
       ast.putAST(0,newAst);
   }

   // statements do not have types
   return Value.NOTYPE;
   }
```

The

```
// binop expressions
   private int interp(BinopExpr ast) {

   // typecheck left child
   int leftType = this.dispatch(ast.getAST(0));

   // typecheck right child
   int rightType = this.dispatch(ast.getAST(1));

   // see if the expression is well typed
   int resultType = Value.getResultType(leftType,rightType);

   // check for type errors
   // NOTE: add on type string is string concatenation
   if (resultType == Value.NOTYPE) {
      System.err.println("Error (binopexpr): binop expression with types "+leftType+" and  "+rightType+" is ill-typed");
      System.exit(1);
      return Value.NOTYPE;
   }

   // check if we have to insert a type promotion
   if (resultType != leftType) {
      AST newAst = new CastExpr(leftType,resultType,(Expr)ast.getAST(0));
      ast.putAST(0,newAst);
   }

   // check if we have to insert a type promotion
   if (resultType != rightType) {
      AST newAst = new CastExpr(leftType,resultType,(Expr)ast.getAST(1));
      ast.putAST(1,newAst);
   }

   // the result type is correct except for the relational operators which
   // always construct an integer return value.
   if (ast.getOp() == BinopExpr.EQ || ast.getOp() == BinopExpr.LESSEQ) {
      return Value.INTEGER;
   }
   else {
      return resultType;
   }
   }
```



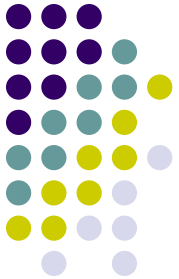
The Type Check Visitor



```
// while statements
   private Integer interp(WhileStmt ast) {
   // typecheck the expression -- has to be an integer
   int exprType = this.dispatch(ast.getAST(0));

   if (exprType != Value.INTEGER) {
       System.err.println("Error: expression of a while-stmt has to be of type integer.");
       System.exit(1);
       return Value.NOTYPE;
   }

   // type check the body of the loop
   this.dispatch(ast.getAST(1));

   // statements do not have types
   return Value.NOTYPE;
   }
```

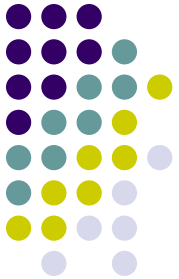
Note: we do not execute the loop, we simply compute all the types.

Code



- SIMPLE4INTERPRETER.zip

Assignment



- Assignment #8 – see website