The Abstract Syntax Tree
Differences Between Tolmach's and Porter's Representations

Harry Porter
January 21, 2006

Introduction

This document discusses the Abstract Syntax Tree (AST) representation used by Andrew Tolmach and that used by Harry Porter.

Class Names

There is a rough correspondence between class names used in by Tolmach and by Porter. Some of the classes in Tolmach (e.g., Decs) have no analog in Porter. Some of the classes in Porter (e.g., NamedType) have no analog in Tolmach.

In the following lists, indentation shows the subclass relationship.

<table>
<thead>
<tr>
<th>Tolmach</th>
<th>Porter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>Node</td>
</tr>
<tr>
<td>Program</td>
<td>Body</td>
</tr>
<tr>
<td>Decs</td>
<td></td>
</tr>
<tr>
<td>VarDecs</td>
<td></td>
</tr>
<tr>
<td>TypeDecs</td>
<td></td>
</tr>
<tr>
<td>ProcDecs</td>
<td></td>
</tr>
<tr>
<td>Dec</td>
<td>VarDecl</td>
</tr>
<tr>
<td>VarDec</td>
<td></td>
</tr>
<tr>
<td>TypeDec</td>
<td>TypeDecl</td>
</tr>
<tr>
<td>ProcDec</td>
<td>ProcDecl</td>
</tr>
<tr>
<td>FormalParam</td>
<td>Formal</td>
</tr>
<tr>
<td>ConstDec</td>
<td></td>
</tr>
</tbody>
</table>
Type

  ArrayType
  RecordType
  BuiltinType

Component

  St
    AssignSt
    CallSt
    ReadSt
    WriteSt
    IfSt
    WhileSt
    LoopSt
    ForSt
    ExitSt
    ReturnSt
    SequenceSt

Exp

  BinOpExp
  UnOpExp
  LValueExp
  CallExp
  ArrayExp
  RecordExp
  IntLitExp
  RealLitExp
  StringLitExp

  Argument
  ArrayValue
  FieldInit
  LValue

  VarLvalue
  ArrayDerefLvalue
  RecordDerefLvalue
## Differences in Field Names

Many of the fields have different names in Tolmach and in Porter. The following chart gives a rough correspondence. For Porter, the type of the field is also given. Fields marked *** contain semantic information that was added to the AST during type-checking. The class **IntToReal** is also added during type-checking.

The other fields capture the same syntactic information as in Tolmach’s AST (except for minor syntactic differences in the languages).

<table>
<thead>
<tr>
<th><strong>Tolmach</strong></th>
<th><strong>Porter</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>Node</td>
</tr>
<tr>
<td>line</td>
<td>lineNumber: int</td>
</tr>
<tr>
<td>newline (const)</td>
<td></td>
</tr>
<tr>
<td>Program</td>
<td></td>
</tr>
<tr>
<td>body</td>
<td></td>
</tr>
<tr>
<td>Body</td>
<td></td>
</tr>
<tr>
<td>decsList</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>statement</td>
<td></td>
</tr>
<tr>
<td>Dec</td>
<td>VarDecl</td>
</tr>
<tr>
<td>VarDecs</td>
<td></td>
</tr>
<tr>
<td>vardeclist</td>
<td></td>
</tr>
<tr>
<td>TypeDecs</td>
<td></td>
</tr>
<tr>
<td>typedeclist</td>
<td></td>
</tr>
<tr>
<td>ProcDecs</td>
<td></td>
</tr>
<tr>
<td>procdeclist</td>
<td></td>
</tr>
<tr>
<td>Dec</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>VarDec</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>TypeDec</td>
<td>VarDecl</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Notes

- **IntToReal** is added during type-checking.
- The other fields capture the same syntactic information as in Tolmach’s AST (except for minor syntactic differences in the languages).
ProcDec

ProcDecl
id: String
retype: TypeName
formals: Formal
body: Body
next: ProcDecl
lexLevel: int ***

FormalParam
Formal
id: String
type: TypeName
next: Formal
lexLevel: int ***

TypeName
id: String
myDef: CompoundType ***

ConstDec
type

Type
CompoundType
ArrayType
ArrayType
elementType: Type
RecordType
RecordType
fieldDecls: FieldDecl

BuiltInType
Component
FieldDecl
id: String
typeName: TypeName
next: FieldDecl

St
Stmt
next: Stmt

AssignSt
AssignStmt
lhs: LValue
rhs: Expr

callSt
CallStmt
procName
id: String
args: Argument
myDef: ProcDecl ***

ReadSt
ReadStmt
targets: ReadArg

WriteSt
WriteStmt
exps: Arguments
IfSt
  test
  ifTrue
  ifFalse
WhileSt
  test
  body
LoopSt
  body
ForSt
  loopVar: String
  start
  stop
  step
  body
ExitSt
ReturnSt
  return
SequenceSt
  statements
Exp
BinaryOpExp
  binOp
  left
  right
  binOpName (const)
UnOpExp
  unOp
  operand
  unOpName (const)
LvalueExp
  lval
CallExp
  procName
  args
IfStmt
  expr: Expr
  thenStmts: Stmt
  elseStmts: Stmt
WhileStmt
  expr: Expr
  stmts: Stmt
LoopStmt
  stmts: Stmt
ForStmt
  lValue: LValue  (note difference in type)
  expr1: Expr
  expr2: Expr
  stmts: Stmt
ExitStmt
  myLoop: Stmt ***
ReturnStmt
  expr: Expr
  myProc: ProcDecl
Expr
BinaryOp
  op: int
  expr1: Expr
  expr2: Expr
  mode: int ***
UnaryOp
  op: int
  expr: Expr
  mode: int ***
IntToReal ***
  expr: Expr
ValueOf
  lValue: LValue
FunctionCall
  id: String
  args: Argument
  myDef: ProcDecl ***
Argument
  expr: Expr
  mode: int ***
  next: Argument
ArrayExp
  typeName
  initializers

ArrayInit
  count
  value

RecordExp
  typeName
  initializers

RecordInit
  name
  value

IntLitExp
  lit

RealLitExp
  lit

StringLitExp
  lit

VarLvalue
  name

ArrayDerefLvalue
  array
  index

RecordDerefLvalue
  record
  name

ArrayConstructor
  id: String
  values: ArrayValues
  myDef: TypeDecl ***

ArrayValue
  countExpr: Expr
  valueExpr: Expr
  next: ArrayValue

RecordConstructor
  id: String
  values: FieldInit
  myDef: TypeDecl ***

FieldInit
  id: String
  expr: Expr
  myFieldDecl: FieldDecl ***
  next: FieldInit

IntegerConst
  intValue: int

RealConst
  rValue: double

StringConst
  sValue: String

BooleanConst
  iValue: int

NilConst

LVvalue

Variable
  id: String
  myDef: Node (either VarDecl or Formal) ***
  currentLevel: int ***

ArrayDeref
  lValue: LVvalue
  expr: Expr

RecordDeref
  lValue: LVvalue
  id: String
  myFieldDecl: FieldDecl ***
Lists vs. Arrays

In PCAT, there are a number of syntactic constructs that repeat. For example, following the ELSE keyword, there can be zero or more statements. As another example, there can be zero or more formal parameters in a procedure declaration.

Tolmach represents sequences with arrays. For example:

```java
class ProcDec extends Dec {
    FormalParam[] formals;
    ...
}
```

Porter represents sequences with linked lists. For example:

```java
class ProcDecl extends Node {
    Formal formals;
    ...
}
class Formal extends Node {
    Formal next;
}
```

Tolmach might use code like this to go through a list of formals:

```java
for (int i = 0; i < formals.length; i++) {
    ... formals[i] ...
}
```

Porter might use code like this:

```java
for (Formal f = formals; f = f.next; f) {
    ... f ...
}
```
Tolmach uses arrays for the following fields:

- **Body** . decsList
- **VarDecs** . vardeclist
- **TypeDecs** . typedeclist
- **ProcDecs** . procdeclist,
- **ProcDec** . formals
- **RecordType** . components
- **CallStmt** . args
- **ReadSt** . targets,
- **WriteSt** . exps
- **SequenceSt** . statements
- **CallExp** . args
- **ArrayExp** . initializers,
- **RecordExp** . initializers

Porter uses linked lists for sequences of...

- **VarDecl** (see comments on declaration grouping)
- **TypeDecl**
- **ProcDecl**
- **Formal** (like **ProcDecl.formals**)
- **FieldDecl** (like (**RecordType.components**))
- **Stmt** (see comments on statement sequences)
- **ReadArg** (like **WriteSt.exps, CallSt.args, and CallExp.args**)
- **Argument** (like **RecordExp inicializers**)
- **ArrayValue** (like **ArrayExp inicializers**)
- **FieldInit** (like **RecordExp inicializers**)

**Statement Sequencing**

In a number of places in the PCAT syntax, there can be zero-or-more statements. For example, a “while” loop can contain zero-or-more statements in its body and an “if” statement can have zero or more statements in its “then” part or its “else” part.

Tolmach uses a special kind of statement called **SequenceSt**, which contains a single field. This field, called **statements**, is an array of **St** nodes.
Tolmach

```java
public abstract static class St extends Node {
    ... (no fields)...
}
public static class WhileSt extends St {
    ...
        St body;
    ...
}
public static class SequenceSt extends St {
    ...
        St[] statements;
    ...
}
```

Porter has no class corresponding to `SequenceSt`. Instead, every statement node has a `next` field. Thus, statements are always part of linked lists.

Porter

```java
abstract static class Stmt extends Node {
    Stmt next;
}
static class WhileStmt extends Stmt {
    ...
        Stmt stmts;
    ...
}
```

In Porter’s AST, whenever a field points to a linked list of nodes, the name of the field is pluralized. In this example, the field `stmts` ends with an “s” indicating that it points to a linked list of Stmt nodes.

Porter’s “Checker” Class

Porter’s type checker is written as a separate class, called `Checker`. There is one instance of this class and this class contains a lot of routines, with names such as:

```java
    checkIfStmt
    checkBinaryOp
    etc...
```

Tolmach has a method called `check` in each of the AST classes.
Porter’s AST nodes are entirely data; the AST classes contain no methods.

In the `main` method, the code creates a `Checker` object and then invokes the `checkAst` method on it. The `main` method looks roughly like this:

```java
Ast.Body ast;
Parser parser;
Checker checker;
...
// Parse the source and return the AST.
parser = new Parser (args);
ast = parser.parseProgram ();

// Check the AST.
checker = new Checker ();
checker.checkAst (ast);
```

**Types**

In Porter, the following basic, predefined types were used during type checking:

- `integer`
- `real`
- `boolean`
- `_string`
- `_nilType`

(_`string` is the name of the type of string constants. Since string constants can only be used in `Write` statements and since variables can never contain string values, there is no “string” keyword in the PCAT language. The underscore in the type’s name is completely internal to the compiler. Likewise, `_nilType` is the type of the “nil” constant and does not correspond to a PCAT keyword.)

Each of the basic types is represented with a `TypeName` node, with the `id` field equal to one of the above strings. The `TypeName` nodes are also used for user-defined types, as in:

```java
type MyType is array of ...;
```

Each `TypeName` node has an `id` field and a `myDef` field. The `myDef` field will point to either an `ArrayType` node or a `RecordType` node. For the `TypeName` nodes for the basic types, the `myDef` field will be null.

In both Tolmach and Porter, each type has a unique name. A type can be represented by its name (a `String`) and type equality was checked in CS-321 by simply comparing strings. Recall that PCAT uses “name equality” and not “structural equality.”
Everything concerning types should be unimportant during code generation, since all type information will be ignored. A new field called mode exists in those nodes where it will be needed during code generation. The mode field was filled in during type checking, and any decisions to be made during code generation will make use of a node’s mode.

The Abstract Class “Dec”

In PCAT, each “body” will have zero-or-more variable declarations, zero-or-more type declarations, and zero-or-more procedure declarations. Each declaration has name (i.e., an identifier) and some other information (the definition).

In each declaration, Tolmach has a field called name. Porter calls this id, but it contains the same information, namely the string name being defined.

Tolmach uses the classes VarDec, TypeDec, and ProcDec to represent declarations.

Tolmach has an abstract superclass of VarDec, TypeDec, and ProcDec called Dec, which factors out the String field called name. In other words, all declaration nodes have a field called name, and this is defined in the abstract superclass Dec, instead of being defined once in each of the classes VarDec, TypeDec, and ProcDec.

Porter also has three classes, which have similar names: VarDecl, TypeDecl, and ProcDecl. However, Porter does not have a class corresponding to Tolmach’s Dec. Instead, each of the “decl” classes contains the String field directly. In the case of Porter, the String field is called id, but it contains the same information as Tolmach’s name field.

Note that any object representing (say) a variable declaration object will look the same, whether it is represented in Tolmach’s AST or in Porter’s AST. In particular, the object will contain a String field regardless of whether the field was defined in the abstract class or was defined directly in the VarDecl class.
**Declaration Grouping**

Consider these PCAT programs:

**Program #1:**
\[
\begin{align*}
\text{var} & \ x, \ y, \ z : \text{int} := 0; \\
\text{var} & \ a, \ b, \ c : \text{int} := 2;
\end{align*}
\]

**Program #2:**
\[
\begin{align*}
\text{var} & \ x, \ y, \ z : \text{int} := 0; \\
& \ a, \ b, \ c : \text{int} := 2;
\end{align*}
\]

They are semantically identical. Tolmach’s representation will capture the difference, but Porter will represent both programs with the same AST. Porter will also represent these programs the same as the following:

**Program #3:**
\[
\begin{align*}
\text{var} & \ x : \text{int} := 0; \\
& \ y : \text{int} := x; \\
& \ z : \text{int} := x; \\
& \ a : \text{int} := 2; \\
& \ b : \text{int} := a; \\
& \ c : \text{int} := a;
\end{align*}
\]

A similar effect happens with type definitions and procedure definitions. In the case of recursively defined types and recursively defined procedures, there is also a semantic difference in the PCAT language itself, as discussed in the PCAT Delta document, but this doesn’t concern us here.

Tolmach’s **Body** node contains a pointer to an array of **Decs**. There are three subclasses of **Decs**, called **VarDecs**, **TypeDecs**, and **ProcDecs**. Each of these classes contains an array. For example, **VarDecs** contains an array of **VarDec** nodes. Thus, Tolmach has an array of arrays of **VarDec** nodes. In program #1, the top-level array points to two sub-arrays and the each sub-array has 3 elements. In the program #2, the top level array points to a single sub-array, which has 6 elements.

Porter’s **Body** node contains pointers to 3 linked lists: one for all **VarDecl** nodes, one for all **TypeDecl** nodes, and one for all **ProcDecl** nodes. Thus, Porter will have one linked lists of **VarDecl** nodes and there will be 6 elements in the list.

Thus, Tolmach retains the original “grouping” information, while Porter looses this information during parsing.
Alter the AST During Type-Checking

Porter alters the AST during type checking, while it appears that Tolmach’s AST is not modified during type-checking.

The nature of these modifications is purely additive: new fields are filled in, but for the most part, the old fields remain unchanged. Thus, the AST is only augmented with new information about the program. Information from the parse is not lost or changed.

For example, when the type checker determines the type of an addition operator (either INTEGER or REAL), this information is stored in the BinaryOp node, in a field called mode. This information will come in handy later, during code generation.

Here are the fields that are filled in during type-checking:

<table>
<thead>
<tr>
<th>Class</th>
<th>New field</th>
<th>Type of field</th>
</tr>
</thead>
<tbody>
<tr>
<td>TypeName</td>
<td>myDef</td>
<td>CompoundType</td>
</tr>
<tr>
<td>ExitStmt</td>
<td>myLoop</td>
<td>Stmt (ForStmt, LoopStmt, or WhileStmt)</td>
</tr>
<tr>
<td>ReturnStmt</td>
<td>myProc</td>
<td>ProcDecl</td>
</tr>
<tr>
<td>CallStmt</td>
<td>myDef</td>
<td>ProcDecl</td>
</tr>
<tr>
<td>FunctionCall</td>
<td>myDef</td>
<td>ProcDecl</td>
</tr>
<tr>
<td>ReadArg</td>
<td>mode</td>
<td>int</td>
</tr>
<tr>
<td>Argument</td>
<td>mode</td>
<td>int</td>
</tr>
<tr>
<td>ArrayConstructor</td>
<td>myDef</td>
<td>TypeDecl (a NamedType)</td>
</tr>
<tr>
<td>RecordConstructor</td>
<td>myDef</td>
<td>TypeDecl (a NamedType)</td>
</tr>
<tr>
<td>FieldInit</td>
<td>myFieldDecl</td>
<td>FieldDecl</td>
</tr>
<tr>
<td>Variable</td>
<td>myDef</td>
<td>Node (VarDecl or Formal)</td>
</tr>
<tr>
<td>RecordDeref</td>
<td>myFieldDecl</td>
<td>FieldDecl</td>
</tr>
<tr>
<td>BinaryOp</td>
<td>mode</td>
<td>int (INTEGER_MODE or REAL_MODE; never 0)</td>
</tr>
<tr>
<td>UnaryOp</td>
<td>mode</td>
<td>int (INTEGER_MODE or REAL_MODE; never 0)</td>
</tr>
</tbody>
</table>

The “IntToReal” Node

In Porter, there is a node called IntToReal, which inserted into the AST during type-checking in exactly those places where a data conversion will be necessary.

```java
static class IntToReal extends Expr {
    Expr expr;
}
```
The **IntToReal** class contains a single field. This node simply serves as a placeholder within an expression to indicate to the code generator where code must be insert to convert an integer value into a floating point value.

This is the only case where the AST is changed from what was produced by the parser. The AST still has pretty much the same shape, however, since all the type-checker does it to insert a node. In all other cases, the type-checker fills in new fields, rather than altering existing fields.

**Misc. Differences**

Tolmach uses a **Program** node; for Porter the entire program is a **Body**. Porter does not have anything corresponding to Tolmach’s **Program** class.

**TRUE, FALSE, and NIL**

The constants **TRUE**, **FALSE**, and **NIL** are represented as **VarLvalue** nodes in Tolmach. There is a **ConstDec** node which is used to define names like **TRUE**, **FALSE**, and **NIL**, so they can be looked up in the **Env**.

In Porter, these constants are recognized during parsing and treated specially. Instead of creating a **Variable** node, the constants **true**, **false**, and **nil** are represented with **BooleanConst** and **NilConst** nodes. The **BooleanConst** node has a single field that tells whether it is true (1) or false (0).

```java
static class BooleanConst extends Expr {
    int iValue;
}
static class NilConst extends Expr {
}
```

**The “myDef” Field**

In Porter’s type-checker, additional information is learned about the PCAT program and some of this information is added to the AST. This information will make generating IR code significantly easier.

Consider a variable in a PCAT program, such as `x`. The variable will be declared and, in fact, there may be several definitions of variable `x`, each in a different procedure. Each declaration defines a different variable which will be stored in a different memory location.
Now consider a use of variable $x$. Which variable does it refer to? The question is really which “declaration” of $x$ does some particular “use” refer to? Each “use” of a variable is represented with a Variable node in Porter (and a VarLvalue in Tolmach). Each “declaration” of a variable is represented with a VarDecl or Formal node in Porter (and a VarDecl or FormalParam node in Tolmach).

During code generation, we will need to store information about variable $x$. For example, we will need to decide how many bytes we’ll use for it and where in memory to put it. We will store this information directly in the VarDecl (or Formal) node in the AST. (Later, we’ll add more fields to these nodes to hold this information.)

When we are generating code, we will encounter “uses” of $x$ from time to time. When we do, we’ll have a Variable node in hand, although we’ll need the information from the VarDecl or Formal node. How do we get from the Variable node to the corresponding VarDecl or Formal node?

Fortunately, during type-checking we saved this link at the time we checked whether each variable was properly declared and used. This is the myDef field in the Variable node, which was filled in during type-checking. The myDef field in a Variable node points to either a VarDecl or a Formal node. So, during code generation, we can simply follow the myDef link to find out all we need to generate code for a “use” of $x$.

Other Information Added During Type-Checking

In PCAT, a procedure is invoked in either a call statement or a function call in an expression. In Porter, these are represented with CallStmt and FunctionCall nodes. During type-checking in Porter, a link was saved between the call and the procedure in question. The link is stored in the myDef field in the CallStmt node and in the FunctionCall node. The myDef field will point to a ProcDecl node.

During code-generation, we will save information about the procedure (such as where in memory its assembly code is stored) in the ProcDecl node. When we generate code for the CallStmt or the FunctionCall, we’ll be able to get to this information by following the myDef field.

In Porter, there are also other links stored in the AST during type-checking.

In the ArrayConstructor node, there is a myDef field, but this will not be used during code generation.

In the RecordConstructor node, there is a myDef field, which will come in handy during code generation.

In VarDecl and Formal there is a field named lexLevel. This is an integer telling the lexical level at which the variable was declared.

In the Variable node there is a field named currentLevel. This is an integer telling the lexical level at which the variable was used.
In the **ProcDecl** node there is a field named **lexLevel**. This is an integer telling the lexical level at which the procedure was declared / defined.

In the **ExitStmt** node there is a field named **myLoop**. This points to either a **WhileStmt**, **LoopStmt**, or **ForStmt**. This field associates the EXIT statement with the loop it exits from.

In the **BinaryOp** and **UnaryOp** nodes there is a field named **mode**. This is an integer field which is significant for some operators. For example, the addition operator (+) and the unary minus operator (-) can be applied to either integers or reals. The value of this field will be either **INTEGER_MODE** or **REAL_MODE**.

In the **Argument** node there is a field named **mode**. Linked lists of **Argument** nodes are pointed to by **CallStmt** and **FunctionCall** nodes to represent the list of arguments in a procedure invocation. For arguments to a procedure invocation, the **mode** field is not used and will be set to zero. Each **WriteStmt** node will also point to a linked list of **Argument** nodes, representing the list of expressions to be printed. For these, the **mode** will be either **INTEGER_MODE**, **REAL_MODE**, **BOOLEAN_MODE**, or **STRING_MODE**. During code generation, we will have to generate different code to print integers than the code to print reals. Strings and booleans will also be printed with different code. The **mode** field will help us out in knowing what type of arguments we have.

In the **RecordDeref** node there is a field named **myFieldDecl**. During type-checking, this field is set to point to the corresponding **FieldDecl** node.

In the **FieldInit** node there is a field named **myFieldDecl**. During type-checking, this field is set to point to the corresponding **FieldDecl** node.

### Dealing with Errors

Tolmach’s code throws **CheckError** when the compiler detects a semantic error. Porter’s code calls a method named **semanticError** and then resumes checking.

Porter provides an exception called **LogicError**, which should be thrown if the compiler encounters an unexpected internal program logic error, indicating a bug. **LogicError** is a subclass of an exception called **FatalError**. Here is an example of its use:

```java
if (...) 
    ...
} else {
    throw new LogicError("Unknown class within checkExpr");
}
```
Walking the Tree

The type-checker walks the AST looking for semantic errors. The code generator will also need to walk the AST generating intermediate code (“IR” code). The code to walk the tree works differently in Tolmach and Porter.

In Tolmach’s approach, each Node defines a method named check. Thus, there are many methods with the same name, one in each of the AST classes. To walk the tree, the main code will invoke the check method on the root node. This method will (recursively) invoke the check method on its children. Which particular check method will get executed will depend on the class of the node it is invoked on.

For example, here is the check method in class WriteSt:

```java
void check(String expectedReturnType, int level, Env env) throws CheckError {
    for (int i = 0; i < exps.length; i++) {
        String t = exps[i].check(env);
        if (...) throw new CheckError(line,"...");
    }
}
```

In Porter, there are many “check” methods, each with a slightly different name. For example, there are methods called checkWriteStmt, checkExpr, checkBody, etc. All these methods are members of the Checker class, not the AST classes.

For example, here is the checkWriteStmt method:

```java
void checkWriteStmt (Ast.WriteStmt t) throws FatalError {
    Ast.Argument arg = t.args;
    while (arg != null) {
        Ast.Type argType = checkExpr (arg.expr);
        if (...) {
            semanticError (arg, "...");
        }
        arg = arg.next;
    }
}
```

In Tolmach, a sequence of statements is represented with an array of St objects, in the SequenceSt node. Here is the check method for SequenceSt:
void check(String expectedReturnType, int level, Env env)
throws CheckError {
    for (int i = 0; i < statements.length; i++)
        statements[i].check(expectedReturnType, level, env);
}

In Porter, there is nothing corresponding to SequenceSt; instead each statement node contains a next pointer. Here is Porter’s checkStmts method:

    void checkStmts (Ast.Stmt stmt)
    throws FatalError
    {
        while (stmt != null) {
            if (stmt instanceof Ast.AssignStmt) {
                checkAssignStmt ((Ast.AssignStmt) stmt);
            } else if (stmt instanceof Ast.CallStmt) {
                checkCallStmt ((Ast.CallStmt) stmt);
            } else if (stmt instanceof Ast.WriteStmt) {
                checkWriteStmt ((Ast.WriteStmt) stmt);
            ...
            } else {
                throw new LogicError ("Unknown class in checkStmts");
            }
            stmt = stmt.next;
        }
    }

In both Tolmach and Porter, a sequence of statements can contain any sort of statement and, during type-checking, each individual statement node in a sequence must somehow be examined to determine what sort of statement it represents and which method to use to type-check the statement.

In Porter, the testing is done directly by the programmer in the code. In Tolmach, the test is done by the Java runtime system. When the check method is invoked in Tolmach, object-oriented method dispatching occurs and the runtime system will use the class of the node to determine which method to invoke.

The object-oriented approach used by Tolmach is faster. The explicit dispatch used by Porter may be easier for some people to follow.

It is difficult to say for sure which design decision is superior. Even with extensive timing studies, it may still boil down to personal preference. These are the sorts of design decisions which make engineering so interesting, since making the best choice requires analysis, experience, intuition, and a even a sense of aesthetics.

Of course the first step is learning which design choices exist.