INF5110: Mandatory Exercise 2

Eyvind W. Axelsen
eyvinda@ifi.uio.no
@eyvindwa
http://eyvinda.at.ifi.uio.no

Slides are partly based on material from previous years, made by Henning Berg, Fredrik Sørensen, and others.

Main goals

- Determine if programs written in the language Compila17 are *semantically* valid
  - I.e. are they type-correct? (*static* semantics)
  - Oblig 1: *syntactically* valid
- Generate byte-code for Compila17(-ish) programs
  - Write a code generator

Last time

- You made
  - a Lexer
  - a Parser
  - an Abstract Syntax Tree
- This time we expand on this
  - Use your previous delivery!
  - Work in the groups you already have

Learning outcomes

- Understand how type checking can be done in practice, implement a simple variant
- Understand what bytecode is, and how it can be generated from source code
- Extend an existing compiler code base with new functionality

Semantic analysis/Type checking

- A parser cannot check all the properties of the language specification
  - Context-free grammars are not powerful enough
- Thus, we shall extend our compiler with a type checker
  - Use the AST classes you defined last time
  - Add type-checking code
  - You are allow to make *any* changes you want to your previous delivery

Example: Java/C#/etc:

```
public class C {


 int m(int i) {


       m(i);
   }
}
```

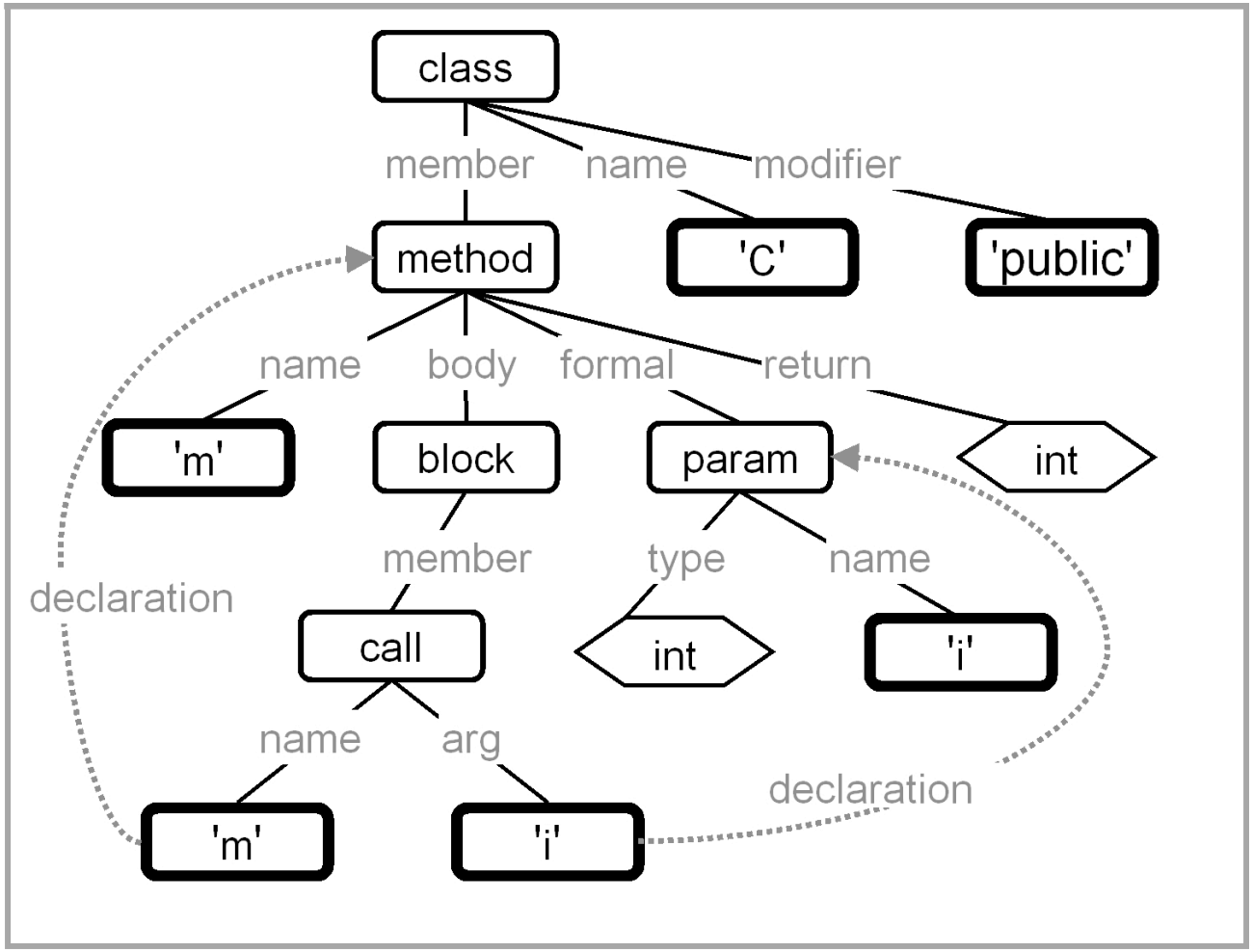


Image from JTransformer website

The Compila17 language at a(nother) glance

```
program MyProgram begin

 class Complex begin
   var Real : float;
   var Imag : float;
 end;


 proc Add(a : Complex, b : Complex) : Complex
 begin
   var retval : Complex;
   retval := new Complex;
   retval.Real := a.Real + b.Real;
   retval.Imag := a.Imag + b.Imag;

   return retval;
 end;

 proc Main()
 begin
   var c1 : Complex;
   var c2 : Complex;
   var result : Complex;
   …
   result := Add ( c1, c2 );
   …
   return;
end; end;
```

Real and Imag are of the (built-in) float type.
Complex defines a new (user-defined) type.

Check that the + operator is compatible with its operands' types, and that the assignment is legal.

Check that the actual parameters to Add(…) are of the correct type, according to the formal parameters, and that the assignment to result is legal.

Type checking – example

```
class IfStatement extends Statement {
   …
   public void typeCheck() {
      String condType = condition.getType();
      if(condType != “bool”) {
         throw new TypeException(“condition in if-
               statement must be of type bool”);
      }
}
```

Implement such a method in e.g. the various Expression classes

Type checking – example

```
class Assignment extends Statement {
   …
   public void typeCheck() {
      String varType = var.getType();
      String expType = exp.getType();
      if(varType != expType &&
            !isAssignmentCompatible(varType, expType)) {
         throw new TypeException("cannot assign "
            + vartype + " from " + expType);
      }
}
```

Implement such a method in e.g. the various Expression classes

Check supported type conversions, e.g. float to int

Code generation

- The lecture about code generation has not been held yet
  - So, if this looks a bit difficult now, don't worry!
- Byte code API and operations are described in the document "Interpreter and bytecode for INF5110"
  - Available on the course page
- Add bytecode generation methods to your AST classes
  - E.g. AstNode.generateCode(…)
  - Again, any changes you want to make to the structure is OK

Code generation - limitations

- The interpreter and bytecode library are somewhat limited
  - Cannot express full Compila17
  - No block structures (only global and local variables)
  - No reference types
- You delivery should support generating correct bytecode for the Compila17 source code file RunMe.cmp
  - Available from the material on the course webpage

Code generation – creating a procedure

```
CodeFile codeFile = new CodeFile();
// add the procedure by name first
codeFile.addProcedure(”Main”);

// then define it
CodeProcedure main = new
        CodeProcedure( ”Main”, VoidType.TYPE,
        codeFile );
main.addInstruction( new RETURN() );

// then update it in the code file
codeFile.updateProcedure( main );
```

Code generation - assignment

```
//1: proc add(a: int, b : int ) : int {
//2: var res : int;
//3: res := a + b; // only bytecode for this line
//4: return res;
//5: }

// push a onto the stack
proc.addInstruction(new LOADLOCAL(proc.variableNumber("a")));
// push b onto the stack
proc.addInstruction(new LOADLOCAL(proc.variableNumber("b")));
// perform addition with arguments on the stack
proc.addInstruction(new ADD());
// pop result from stack, and store it in variable res
proc.addInstruction(new
        STORELOCAL(proc.variableNumber("res")));
```

Code generation – writing to file

```
String filename = “myfile.bin”;
byte[] bytecode = codeFile.getBytecode();
DataOutputStream stream = new
        DataOutputStream(
                new FileOutputStream (filename));
stream.write(bytecode);
stream.close();
```

Testing

- 42 supplied tests in test folder, for testing the type checker
- Run tests with “ant test”
- Tests ending with “fail” are supposed to fail (i.e., they contain an erroneous program)
  - Compiler returns error code 2 for semantic failure
- 32 of the 42 tests must pass for the delivery to be successful

Provided source code

You are given a patch folder, that replaces certain files in your existing oblig 1 directory structure. Create a backup before you replace your existing files!

code-examples

Three example programs, including RunMe.cmp, that you're going to compile

src

Revised source code, see next slide

compila-code

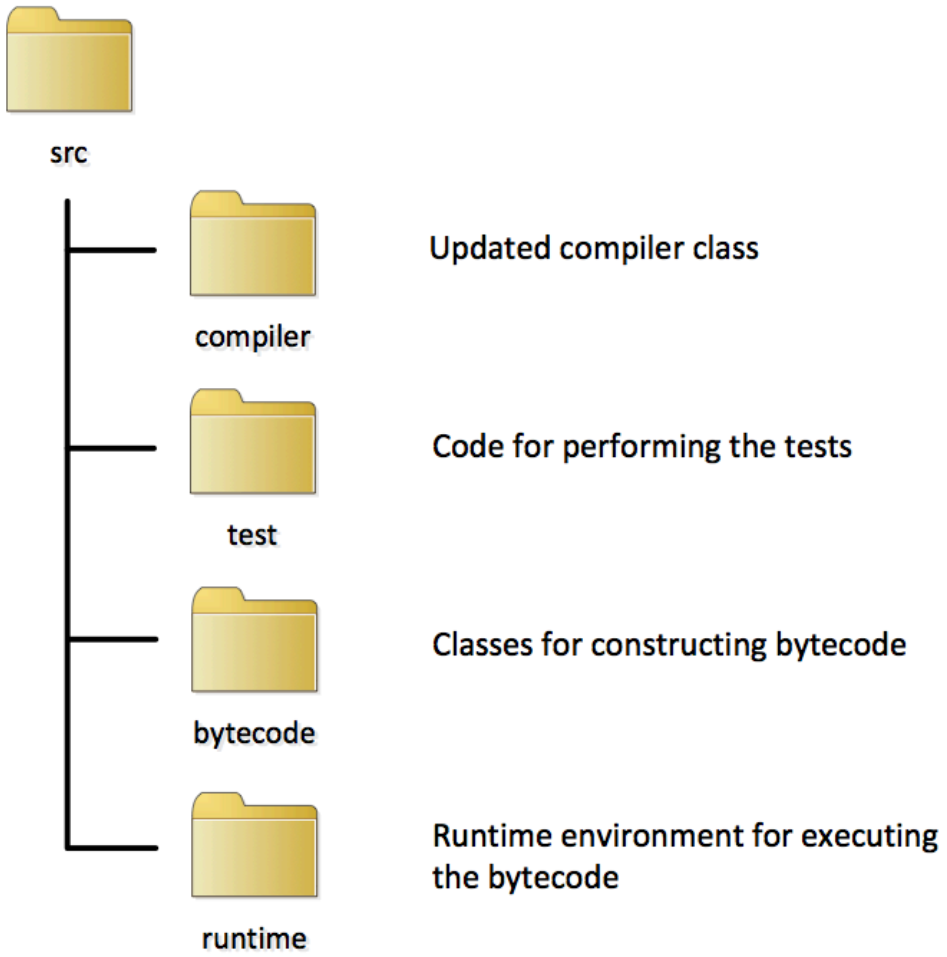
Revised version of Compila.cmp (not really needed for this exercise)

tests

42 test programs. Use these to verify your type checking implementation (and hand in a printout of the results with your delivery)

See also the patch.README file at the root directory of the patch code

Provided source code (the src folder)



DEADLINE

- May 7th, 2017 @ 23:59
- Don’t miss the deadline!
  - Extensions are only possible if you have an agreement with the student administration (studadm)
  - We must be a bit strict, because of deadlines for exam lists etc
  - Contact them if you are sick, etc.
- Even if you are not 100% finished, deliver what you have before the deadline

Deliverables

- Working type checker for Compila17
  - Run the supplied tests
- Working code generator for (a subset of) Compila17
  - Test with RunMe.cmp
- Report
  - Front page with your name(s) and UiO user name(s)
    - Work in the groups from oblig 1
  - Discussion of your solution, choices you've made and assumptions that you depend on
  - Printout of test run
  - Printout of bytecode from RunMe.cmp (use `ant list-runme`)
- The code you supply must build with "ant"
  - Test your delivery on a UiO computer
- Deliver through Devilry
  - Feel free to send questions at any time to eyvinda@ifi.uio.no
  - Read the exercise description thoroughly!