

INF5110: Mandatory Exercise 2



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 Compila16 are *semantically* valid
 - I.e. are they type-correct? (*static* semantics)
- Generate byte-code for Compila16(-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!

Learning outcomes

- Understand how type checking can be done
- 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 allowed to make *any* changes you want to your previous delivery

The Compila16 language at a(nother) glance

```
program MyProgram begin
```

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

Real and Imag are of the (built-in) float type

```
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;
```

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

```
  return retval;  
end;
```

```
proc Main()  
begin
```

```
  var c1 : Complex;  
  var c2 : Complex;  
  var result : Complex;  
  ...  
  result := Add ( c1, c2 );
```


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.

```
  ...  
  return;  
end; end;
```

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);
```

```
}
```

```
}
```

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



Code generation



The lecture about code generation

- is not until April 20th,
- So, if this looks 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

Code generation - limitations

- The interpreter and bytecode library are somewhat limited
 - Cannot express full Compila16
 - No block structures (only global and local variables)
 - No reference parameters
 -
- You delivery should support generating correct bytecode for the Compila16 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
```

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.getBytescode();  
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



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

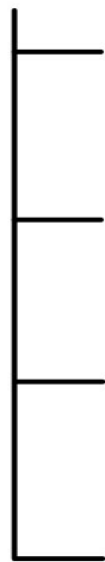
42 test programs. Use these to verify your type checking implementation (and hand in a printout of the results with your delivery)

NOTE: the error mentioned earlier is in this folder

Provided source code (the src folder)



src



compiler

Updated compiler class



test

Code for performing the tests



bytecode

Classes for constructing bytecode



runtime

Runtime environment for executing the bytecode

DEADLINE

- May 8, 2015 @ 23:59
- Don't miss the deadline!
 - Extensions are only possible if you have an agreement with the student administration (studadm)
 - *This time we must be stricter*, 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 Compila16
 - Run the supplied tests
- Code generator for (a subset of) Compila16
 - Test with RunMe.cmp
- Report
 - Front page with your name(s) and UiO user name(s)
 - Work alone or in pairs. Groups of three can be allowed after an application.
 - Discussion of your solution, choices you've made and assumptions that you depend on
 - Printout of test run
 - Printout of bytecode from RunMe.cmp
- The code you supply must build with “ant”
 - Test your delivery on a UiO computer