INF5110: Mandatory Exercise 1

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 goal

Determine if programs written in the language Compila16 are *syntactically* valid.

- Write a scanner
- And a parser
- Compila16 is described in detail in a separate document available on the course page.

Learning outcomes

- Using tools for scanner and parser generation
  - JFlex and CUP

- Variants of a grammar for the same language
  - Transforming from one form (extended BNF) to another (BNF for the tools we will be using).
  - Controlling precedence and associativity

- Defining ASTs as node classes in Java
  - Using the parsing tools to build such trees
  - Pretty-printing ASTs.

The Compila16 language at a glance

```
program MyProgram
begin
```

Programs are written enclosed in program NAME begin … end

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

The language supports very simple "classes", but no real OO (inheritance, polymorphism, etc)

```
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 ;
```

Procedures are declared within programs (but not within classes). They perform calculations and create new objects.

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

end ;
```

Execution starts in the Main method.

```
PROGRAM              -> "program" NAME "begin" { DECL ";" }  "end" ";"

DECL                 -> VAR_DECL | PROC_DECL | CLASS_DECL

VAR_DECL             -> "var" NAME ":" TYPE

PROC_DECL            -> "proc" NAME "(" [ PARAM_DECL { "," PARAM_DECL } ] ")"
                        [ ":" TYPE ]  "begin" { DECL ";" } { STMT ";" } "end"

CLASS_DECL           -> "class" NAME "begin" { VAR_DECL ";" } "end"

PARAM_DECL           -> [ "ref" ] NAME ":" TYPE

EXP                  -> EXP LOG_OP EXP | "not" EXP  | EXP REL_OP EXP   | EXP ARIT_OP EXP   | "(" EXP ")"
                        | LITERAL  | CALL_STMT  | "new" NAME  | VAR

VAR                  -> NAME | EXP "." NAME

LOG_OP               -> "&&" | "||"

REL_OP               -> "<" | "<=" | ">" | ">=" | "=" | "<>"

ARIT_OP              -> "+" | "-" | "*" | "/" | "#"

LITERAL              -> FLOAT_LITERAL | INT_LITERAL | STRING_LITERAL | "true" | "false" | "null"

STMT                 -> ASSIGN_STMT | IF_STMT  | WHILE_STMT  | RETURN_STMT | CALL_STMT

ASSIGN_STMT          -> VAR ":=" EXP

IF_STMT              -> "if" EXP "then" "begin" { STMT ";" } "end"   [ "else" "begin" { STMT ";" } "end" ]

WHILE_STMT           -> "while" EXP "do" "begin" { STMT ";" } "end"

RETURN_STMT          -> "return" [ EXP ]

CALL_STMT            -> NAME "(" [ ACTUAL_PARAM { "," ACTUAL_PARAM } ] ")"

ACTUAL_PARAM         -> "ref" VAR | EXP

TYPE                 -> "float" | "int" | "string" | "bool" | NAME
```

Compila16 grammar

"terminal"
NON-TERMINAL
[optional]
{ repetition }
Alternative1 | Alternative2

Tool: JFlex

- A tool to easily (YMMV) generate *scanners*
  - Input: lexical specification
  - Output: scanner program written in Java

- The lexical specification is written in a .lex file
  - Consists of three separate parts
    - User code
    - Options and macros
    - Lexical rules

User code

```
package oblig1parser;
import java_cup.runtime.*;
```
Copied to the generated class, before the class definition

`%%`

Options/ macros

```
%class Lexer
%unicode
%cup
```
Options (class name, unicode support, CUP integration)

Defined in package java_cup.runtime.

```
%{
   private Symbol symbol(int type) {
     return new Symbol(type, yyline, yycolumn);
   }

%}
LineTerminator = \r|\n|\r\n
```
Inserted into generated class

Variables holding current line/column

Macros, defined as regular expressions

`%%`

Lexical rules

```
<YYINITIAL>
{
     "program"           { return symbol(sym.PROGRAM); }
     "class"             { return symbol(sym.CLASS); }
     "begin"             { return symbol(sym.BEGIN); }
     "end"               { return symbol(sym.END); }
     "var"               { return symbol(sym.VAR); }
      …
}
```
The following rules are applicable from the initial state

Lexical rules

Tool: CUP – *Construction of Useful Parsers* - for Java

- A tool to easily (YMMV) generate *parsers*
  - Reads tokens from the scanner using **next_token**()
    - The **%cup** option (prev. slide) makes this work
  - Input: Grammar defined as BNF with *action code*

Assign names to parts of production so we can reuse them in action code

```
var_decl   ::= VAR ID:name COLON ID:type SEMI
  {: RESULT = new VarDecl(name, type); :};
```

Build AST with user defined node classes (java code)

  - Output: a parser program written in Java

LALR Parser Generator for Java™

Scott Hudson, GVU Center, Georgia Tech

Package/ imports

```
package oblig1parser;
import java_cup.runtime.*;
import syntaxtree.*;
```

Package name for generated code and imports of packages we need

The syntaxtree package contains our own AST classes

User code

```
parser code {: :};
```

Code between {: and :} is inserted directly into the generated class (parser.java)

Symbol list

```
terminal                PROGRAM, CLASS;
terminal                BEGIN, END;
…
terminal        String      ID;
terminal        String      STRING_LITERAL;

non terminal    Program         program;
non terminal    List<ClassDecl>  decl_list;
non terminal    ClassDecl       class_decl, decl;
```

Terminals and non-terminals are defined here. They can also be given a Java type for the "value" that they carry, e.g. a node in the AST

Precedence

```
precedence left      AND;
```

Precedence declarations are listed in ascending order

Grammar

```
program      := PROGRAM BEGIN decl_list:dl END SEMI {: RESULT = new Program(dl); :} ;

decl_list    ::= decl:d
        {: List<ClassDecl> l = new LinkedList<ClassDecl>(); l.add(d); RESULT = l; :} ;

decl         ::= class_decl:sd {: RESULT = sd; :}   ;

class_decl   ::= CLASS ID:name BEGIN END
                    {: RESULT = new ClassDecl(name); :}  ;
```

AST is built during parsing. The left hand side of each production is implicitly labeled RESULT.

AST

- Make a reasonable structure
  - This slide is an EXAMPLE
  - Do not copy it verbatim without thinking
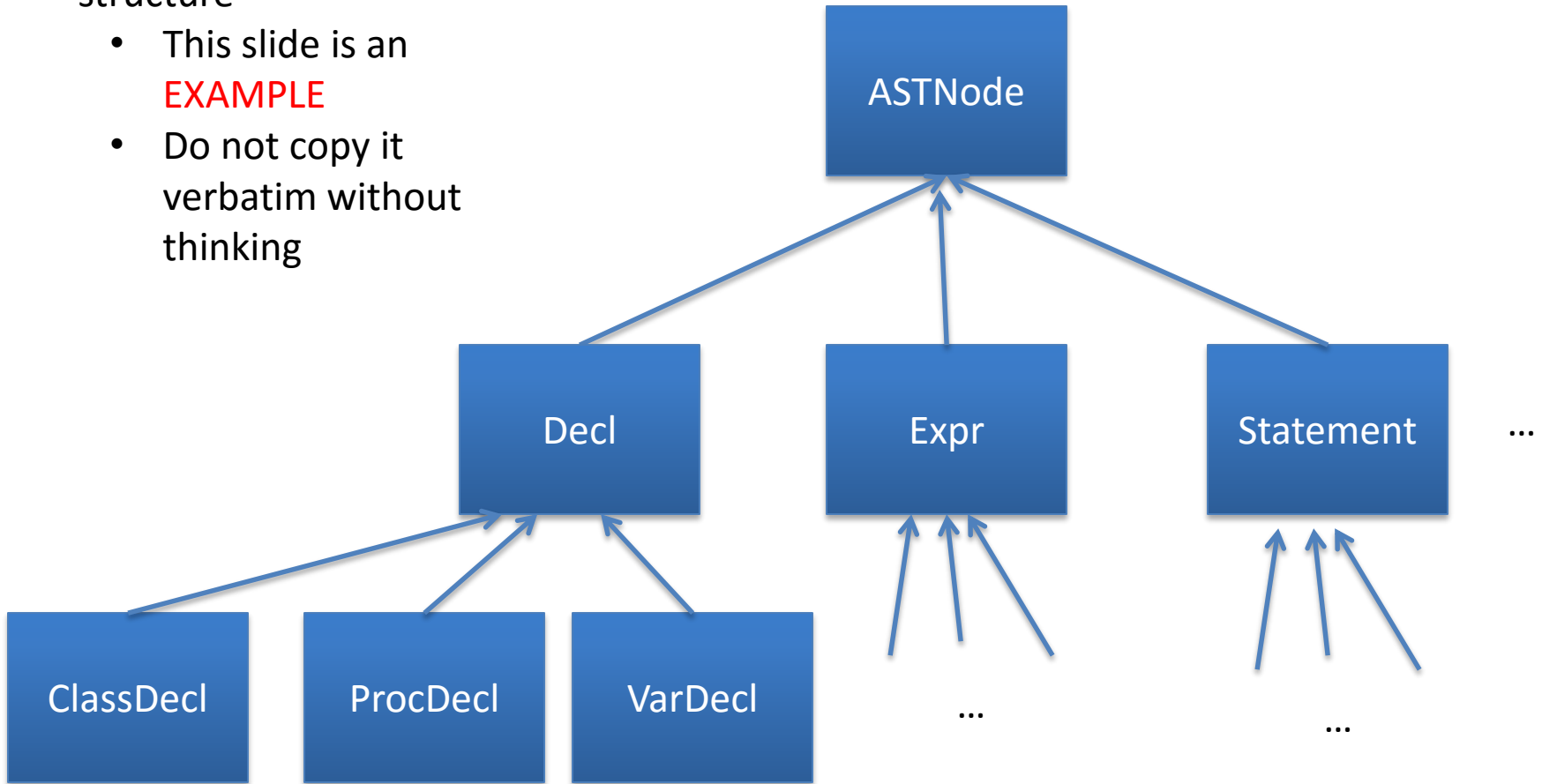
Tool:

- A Java-based build tool
  - Configuration in build.xml
    - Can contain different targets, for instance test, clean, build, run, etc
  - The supplied configuration takes care of calling jflex, cup and javac for you.
    - Note that ant might continue even if jflex or cup encounter errors!

Provided source code

expression-eval.cup/lex
Example expression language
expression-par.cup/lex
Example language that handles parentheses
oblig1.cup/lex
Starting point for your grammars in this exercise

Class files for compiler, lexer, parser, syntaxtree, etc.

build

Three pairs of .lex/.cup files

grammars

Test file for example parser

input-examples

JFlex and CUP libs

lib

Compila source code

compila-code

Java source code for compiler, syntax tree, etc.

src

Java source code example syntax tree

src-examples

Generated Java source code for lexer and parser

src-gen

compila.ast
Example showing how your pretty-printed AST could (should) look

compila-ast

Generated abstract syntax tree

compila.cmp
Compila source file; this is the file you need to parse in this exercise

ClassDecl.java,
Starting point for AST node implementations in Java
Compiler.java
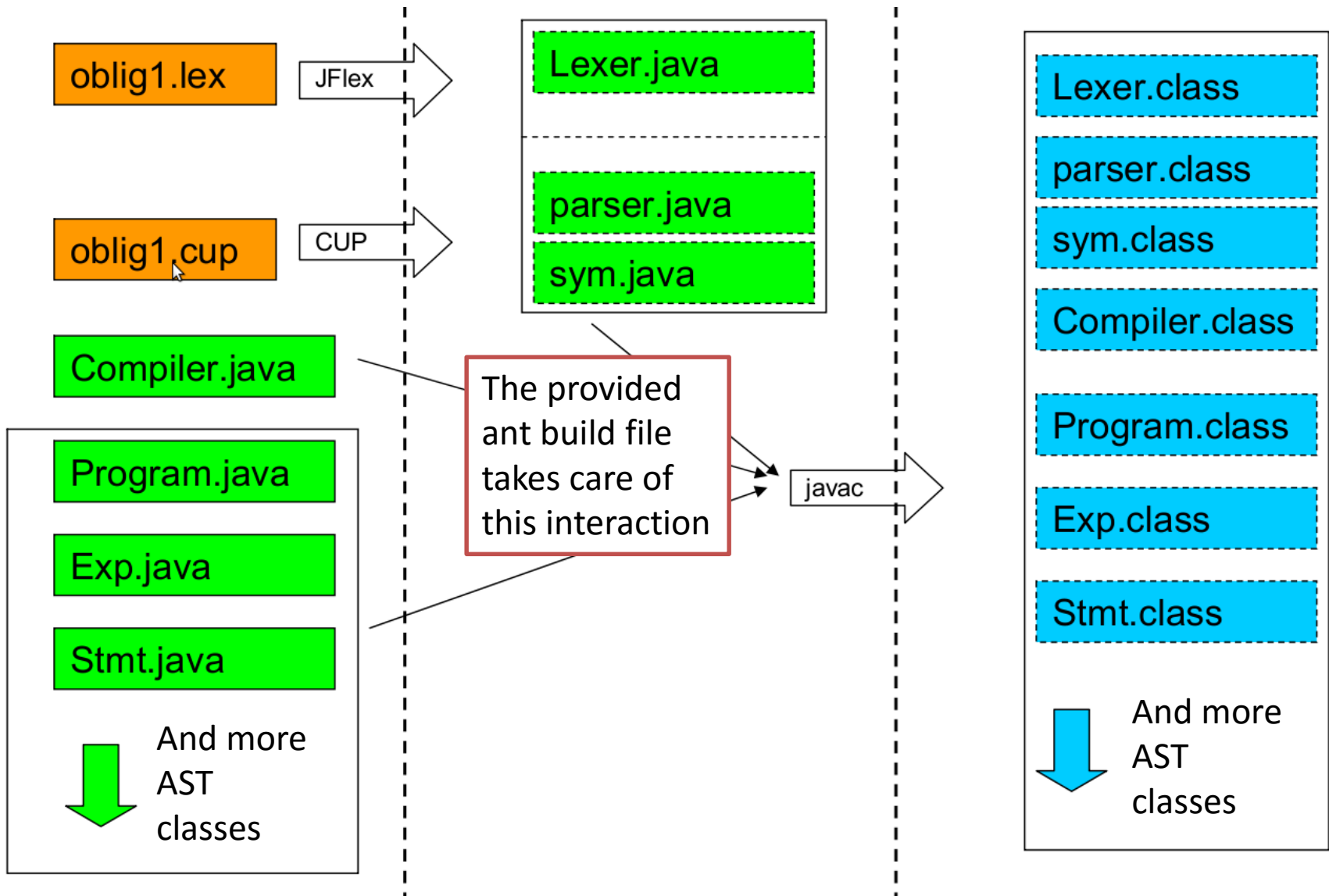The main entry point for the compiler. You do not necessarily have to change this

Putting it all together

DEADLINE

- March 20th, 2015 @ 23:59
- Don't miss the deadline!
  - Extensions are only possible if you have an agreement with the student administration (studadm)
  - Contact them if you are sick, etc.
- Even if you are not 100% finished, deliver what you have before the deadline

Deliverables

- Working parser for Compila16
  - Parse the supplied example program
  - Printout of the resulting AST
- Two grammars
  - One ambiguous, with ambiguities resolved through precedence declarations
  - One inherently unambiguous grammar
- 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
  - A comparison of the two grammars
- The code you supply must build with "ant"
  - Test your delivery on a UiO computer
- Deliver a zipped folder by email to eyvinda@ifi.uio.no
  - Feel free to send questions at any time!
  - Read the exercise description thoroughly!