Chapter 1

Oblig 1

Course “Compiler Construction”

Martin Steffen

Spring 2018

Section

Compila 18

Chapter 1 “Oblig 1”
Course “Compiler Construction”
Martin Steffen
Spring 2018

Oblig 1

- material based on previous years, including contributions from Eyvind W. Axelsen, Henning Berg, Fredrik Sørensen, and others
- see also the course web-page, containing links to "resources"

Goal (of oblig 1)

Parsing

Determine if programs written in *Compila 18* are syntactically correct:

- scanner
- parser

- first part of a compiler, oblig 2 will add to it
- language spec provided separatly

Learning outcomes

- using tools for parser/scanner generation
  - JFlex
  - CUP
- variants of a grammar for the same languages
  - transforming one form (EBNF) to another (compatible with the used tools)
  - controlling precedence and associativity
- designing and implementing an AST data structure
  - using the parsing tools to build such trees
  - pretty-printing such trees

Compila language at a glance

```
program MyProgram
begin
    struct complex {      // record data type, but
      var   re: float;    // no subtyping, polymorphism...
      var   im: float;
    }
end;

proc add (a: complex, b: complex) : complex
begin
    var retval : complex;
    retval :=  new complex;
    retval.re := a.re + b.re;
    retval.im := a.im + b.im;
    return retval;
end;
proc main()               // exectution start here
begin
   var c1: complex;
   var c2: complex:
   var result := add (c1,c2);
   ...
   return;
end;
end;
```

Another glance

```
proc swap (a: ref(int), b: ref(int))  // passed a reference
begin
    var tmp: int;
    tmp := deref(a);                  // dereferencing
    deref(a) := deref(b);             // deref can be used both
    deref(b) := tmp;                  // left and right of
                                      //  an assignment.
end;
```

Grammar (1): declarations

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

DECL             -> VAR_DECL | PROC_DECL | REC_DECL

VAR_DECL         -> "var" NAME ":" TYPE

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

REC_DECL       -> "struct" NAME "{" { VAR_DECL ";" } "}"

PARAM_DECL       -> NAME ":" TYPE
```

Grammar (2): declarations

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

REF_VAR      -> "ref" "(" VAR ")"
DEREF_VAR    -> "deref" "(" VAR ")" | "deref" "(" DEREF_VAR ")"

VAR          -> NAME | EXP "." NAME

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

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

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

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

Grammar (3): statements and types

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

ASSIGN_STMT     -> VAR ":=" EXP | DEREF_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 "(" [ EXP { "," EXP } ] ")"

TYPE            -> "float" | "int" | "string" | "bool" | NAME
                 | "ref" "(" TYPE ")"
```

Section

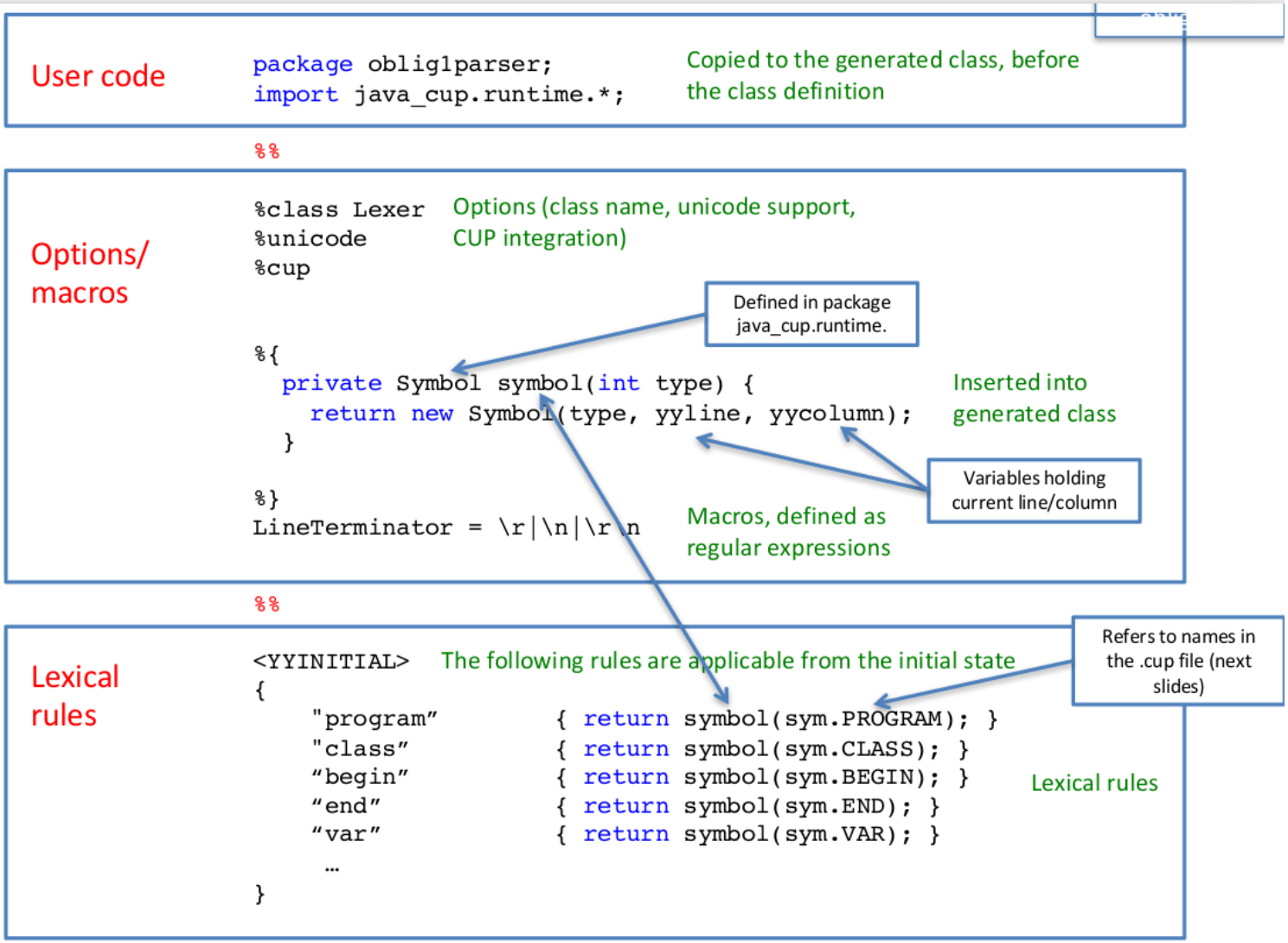
Tools

Chapter 1 “Oblig 1”
Course “Compiler Construction”
Martin Steffen
Spring 2018

Tools: JFlex

- scanner generator (or lexer generator) tool
  - input: lexical specification
  - output: scanner program in Java
- lexical spec written as `.lex` file
- consists of 3 parts
  - user code
  - options and macros
  - lexical rules

Sample lex code



CUP: Construction of useful parsers (for Java)

- a tool to easily (ymmv) generate *parsers*
- reads tokes from the scanner using `next_token()`
- the `%cup` option (previous slide) makes that work

Input

grammar in BNF with action code

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

- output: parser program (in Java)

Sample CUP code

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;
```

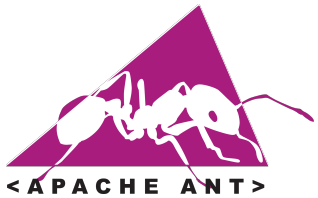
Precedence declarations are listed in ascending order, last = highest

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.

Build tool: ant



- Java-based build tool (think “make”)
- config in `build.xml`
- can contain different targets

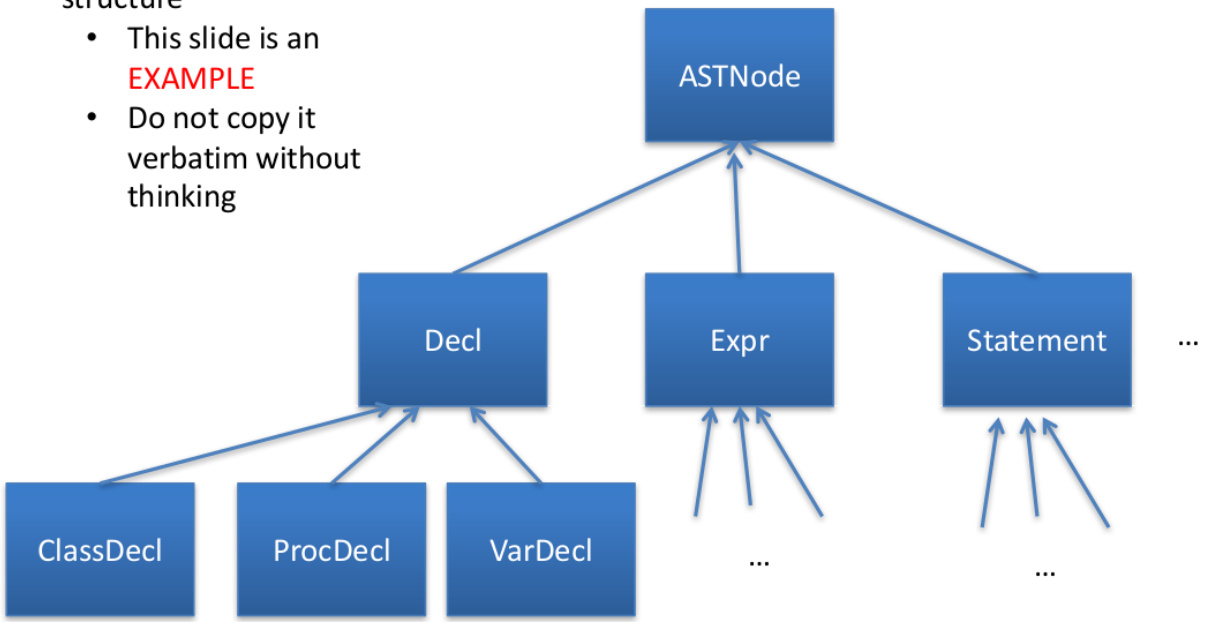
typical general targets

- test
- clean
- build
- run

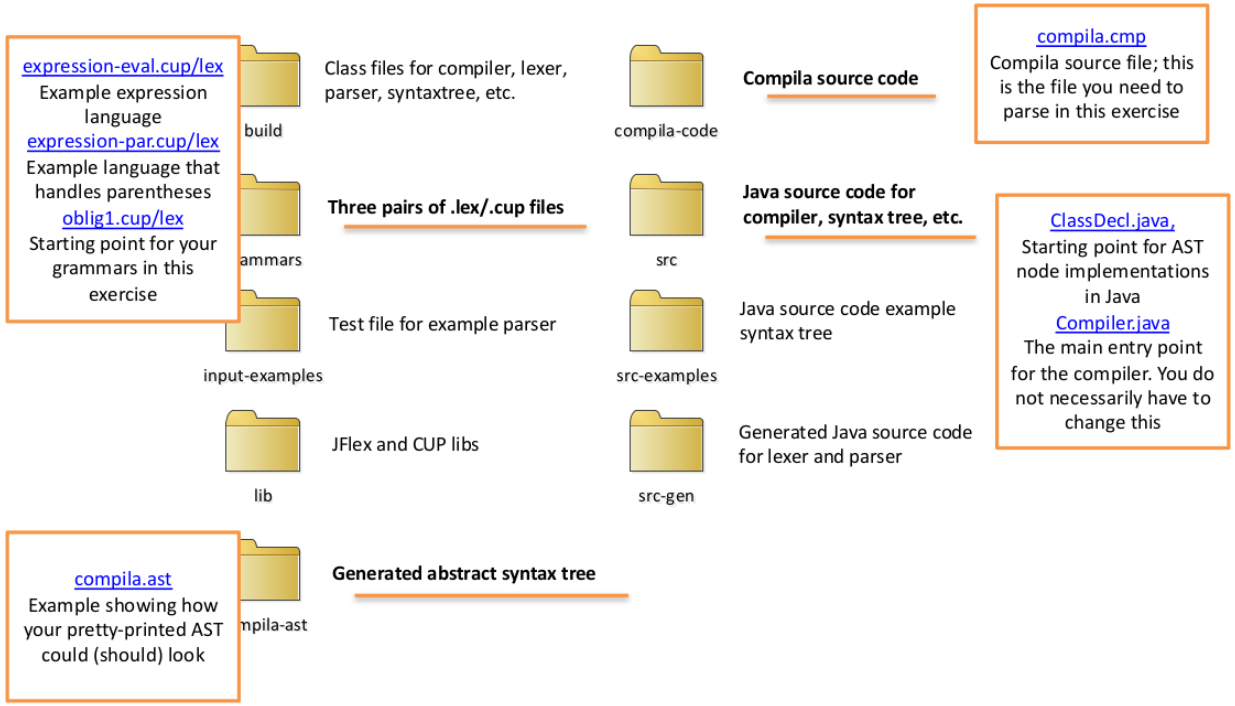
- supplied configuration should take care of calling `jflex`, `cup`, and `javadoc` for you

AST data structure

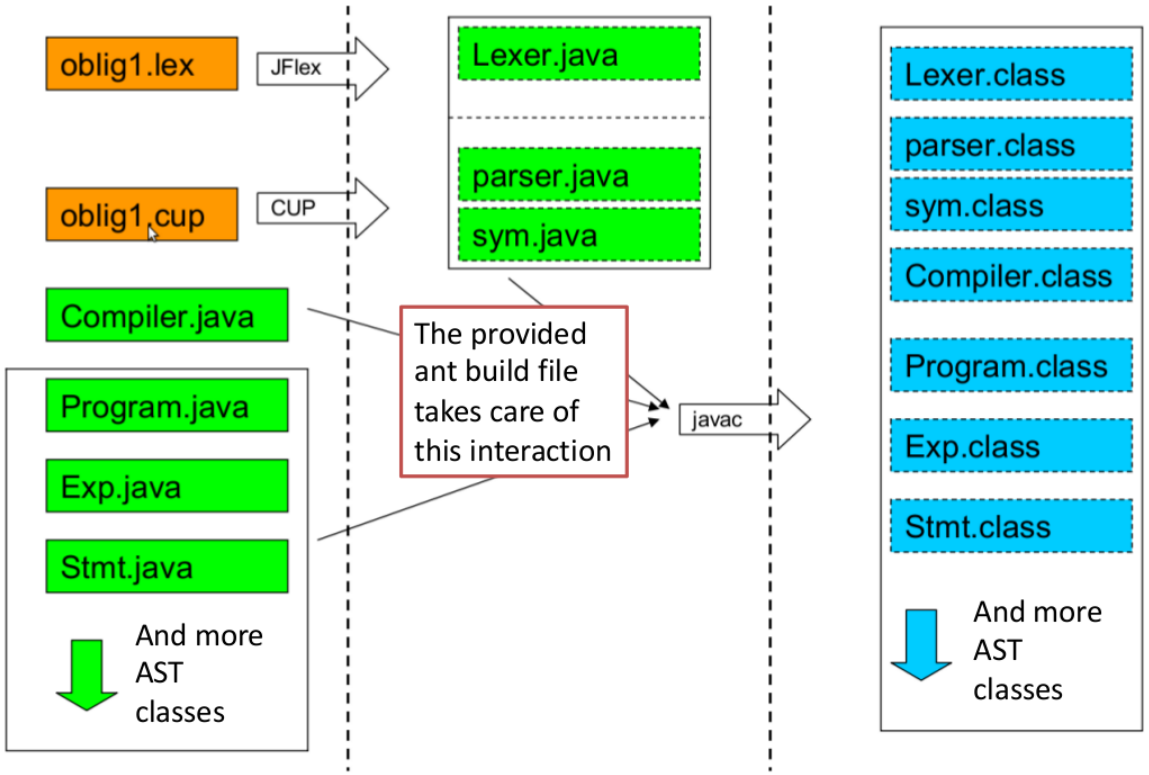
- Make a reasonable structure
  - This slide is an EXAMPLE
  - Do not copy it verbatim without thinking



Provides source code



Building: putting it together



Section

Official

Chapter 1 “Oblig 1”
Course “Compiler Construction”
Martin Steffen
Spring 2018

Deadline

Deadline

Friday 23. 03. 2018, 23:59

- don't miss the deadline
- for extensions, administration needs to agree (`studadm`), contact them if sick etc
- even if not 100% finished
  - deliver what you have
  - contact early when problems arise

Deliverables

- see also the “handout”

Deliverables (1)

- working parser
  - parse the supplied sample programs
  - printout the resulting AST
- two grammars (two `.cup`-files)
  - one unambiguious
  - one ambiguous, where ambibuities resolved through precedence declations in *CUP*, e.g.

```
precendence left AND;
```

Deliverables

Deliverables (2)

- report (with name(s) and UiO user name(s)
- discussion of the solution (see handout for questions)
- in particular: comparison of the two grammars
- "Readme"

- the code must *build* (with ant) and run
- test it on the UiO RHEL platform

Ask

If problems, ask in time (NOT Friday at the deadline)

Hand-in procedure

- this year we try *git*
- `https://github.uio.no` resp. `https://github.uio.no/msteffen/compila`
- you need
  - a login
  - send me emails that you want to do oblig (+ potential partner) ⇒ I tell you group number
  - create a project `compila<n>` (n = group number)
  - add collaborator + (at some point me)
- see also the handout
- code ready *tomorrow*