Section

Compila 23

Oblig 1

- material (also for oblig 2) 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 23* are
syntactically correct:

- scanner
- parser

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

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
      re: float;            // no subtyping, polymorphism...
      im: float
    }
end;

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

Another glance

```
proc swap (a: ref(int), b: ref(int))   // passed as 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 {";" DECL }]
"end"

DECL   —> VAR_DECL | PROC_DECL | REC_DECL

VAR_DECL   —> "var" NAME ":" TYPE [":=" EXP] |
"var" NAME ":=" EXP

PROC_DECL   —> "procedure" NAME
                              "(" [PARAMFIELD_DECL { "," PARAMFIELD_DECL }]
                              [":" TYPE]
                              "begin" [[DECL{";" DECL}] "in"] STMT_LIST "end"
REC_DECL   —> "struct" NAME "{" [PARAMFIELD_DECL
                                                  {";" PARAMFIELD_DECL }]
"}"
PARAMFIELD_DECL   —> NAME ":" TYPE

Grammar (2): expressions, statements, etc.

```
STMT_LIST          -> [STMT {";" STMT}]


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

REF_VAR            -> "ref" "(" VAR ")"    ;; change that, a bit unbal
DEREF_VAR          -> "deref" "(" VAR ")" | "deref" "(" DEREF_VAR ")"

VAR                -> NAME | EXP "." NAME

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

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

ARITH_OP           -> "+" | "-" | "*" | "/" | "^"
```

Grammar (3): statements and types

LITERAL	$\rightarrow$	FLOAT_LITERAL | INT_LITERAL
		| STRING_LITERAL | BOOL_LITERAL | "null"
BOOL_LITERAL	$\rightarrow$	"true" | "false"
STMT	$\rightarrow$	ASSIGN_STMT
		| IF_STMT
		| WHILE_STMT
		| RETURN_STMT
		| CALL_STMT
ASSIGN_STMT	$\rightarrow$	VAR ":=" EXP | DEREF_VAR ":=" EXP
IF_STMT	$\rightarrow$	"if" EXP "then" { STMT_LIST }
$\leftarrow$ NO		
		["else" { STMT_LIST }] "fi"
$\leftarrow$ NO		
WHILE_STMT	$\rightarrow$	"while" EXP "do" { STMT_LIST } "od" $\leftarrow$ NO
RETURN_STMT	$\rightarrow$	"return" [EXP]
CALL_STMT	$\rightarrow$	NAME "(" [EXP { "," EXP }] ")"
TYPE	$\rightarrow$	"float" | "int" | "string" | "bool" | NAME
		| "ref" "(" TYPE ")"

Section

Tools

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

User code

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

%%

Options/macros

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

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>    The following rules are applicable from the initial state
{
    "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); }
      …
}
```

Refers to names in the .cup file (next slides)

Lexical rules

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`
	`terminal`
	`...`
	`terminal`
	`terminal`
	
	`non terminal`
	`non terminal`
	`non terminal`
		`PROGRAM, CLASS;`
		`BEGIN, END;`
		
		`String    ID;`
		`String    STRING_LITERAL;`
		
		`Program        program;`
		`List<ClassDecl> decl_list;`
		`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

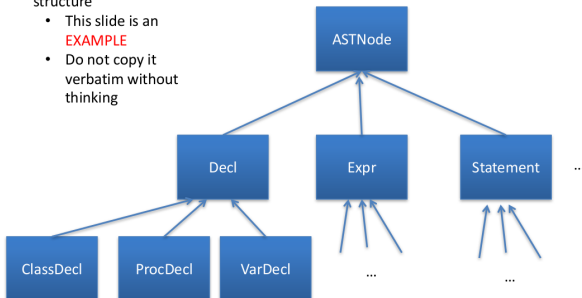- supplied configuration should take care of calling `jflex`, `cup`, and `javadoc` for you

AST data structure

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

Overview over the directory + first steps

- see the Readme at/from the `github.uio.no`

Compila 23

Tools

Official

```
/uio/kant/ifi-ansatt-u00/msteffen/cor/teaching/compila:
total used in directory 64 available 49217216
drwxrwxr-x. 12 msteffen ifi 2048 Feb  3 11:31 .
drwxrwxr-x. 11 msteffen ifi 2048 Jan 24 14:18 ..
drwxrwxr-x.  8 msteffen ifi 2048 Feb  3 11:31 .git
-rw-rw-r--.  1 msteffen ifi   77 Mar 23  2019 .gitignore
-rw-rw-r--.  1 msteffen ifi 5506 Jan 17 07:03 Readme.org
drwxrwxr-x.  3 msteffen ifi 2048 Feb 12  2019 build
-rwxrwxr-x.  1 msteffen ifi 3231 Feb 12  2019 build.xml
drwxrwxr-x.  5 msteffen ifi 2048 Feb 18  2019 doc
drwxrwxr-x.  2 msteffen ifi 2048 Jan  9  2017 lib
drwxrwxr-x.  4 msteffen ifi 2048 Feb  3 11:29 material
drwxrwxr-x.  5 msteffen ifi 2048 Feb  3 11:15 oblig2patch
drwxrwxr-x.  5 msteffen ifi 2048 Feb  3 11:28 previoussemesters
drwxrwxr-x. 12 msteffen ifi 2048 Feb  3 11:31 src
drwxrwxr-x.  3 msteffen ifi 2048 Feb 12  2019 src-gen
drwxrwxr-x.  3 msteffen ifi 2048 Feb 12  2019 tmp

/uio/kant/ifi-ansatt-u00/msteffen/cor/teaching/compila/lib:
total used in directory 280 available 49217216
drwxrwxr-x.  2 msteffen ifi   2048 Jan  9  2017 .
drwxrwxr-x. 12 msteffen ifi   2048 Feb  3 11:31 ..
-rwxrwxr-x.  1 msteffen ifi 179102 Jan  9  2017 JFlex.jar
-rwxrwxr-x.  1 msteffen ifi  96121 Jan  9  2017 java-cup-11a.jar

/uio/kant/ifi-ansatt-u00/msteffen/cor/teaching/compila/src:
total used in directory 48 available 49217216
drwxrwxr-x. 12 msteffen ifi 2048 Feb  3 11:31 .
drwxrwxr-x. 12 msteffen ifi 2048 Feb  3 11:31 ..
drwxr-xr-x.  4 msteffen ifi 2048 Mar 15  2017 bytecode
drwxrwxr-x.  2 msteffen ifi 2048 Feb  3 11:26 compiler
drwxrwxr-x.  6 msteffen ifi 2048 Feb 13  2019 doc
drwxrwxr-x.  2 msteffen ifi 2048 Feb 12  2019 grammars
drwxrwxr-x.  2 msteffen ifi 2048 Jan 16 09:13 org
drwxr-xr-x.  2 msteffen ifi 2048 Mar 16  2017 runtime
drwxrwxr-x.  2 msteffen ifi 2048 Feb 18  2019 src-gen
drwxrwxr-x.  2 msteffen ifi 2048 Feb 12  2019 syntaxtree
drwxrwxr-x.  2 msteffen ifi 2048 Feb  3 11:22 test
drwxrwxr-x.  8 msteffen ifi 2048 Feb  3 10:48 tests

/uio/kant/ifi-ansatt-u00/msteffen/cor/teaching/compila/src/compiler:
total used in directory 12 available 49217216
drwxrwxr-x.  2 msteffen ifi 2048 Feb  3 11:26 .
drwxrwxr-x. 12 msteffen ifi 2048 Feb  3 11:31 ..
-rwxrwxr-x.  1 msteffen ifi  981 Feb  3 11:26 Compiler.java

/uio/kant/ifi-ansatt-u00/msteffen/cor/teaching/compila/src/grammars:
total used in directory 16 available 49217216
drwxrwxr-x.  2 msteffen ifi 2048 Feb 12  2019 .
drwxrwxr-x. 12 msteffen ifi 2048 Feb  3 11:31 ..
-rwxrwxr-x.  1 msteffen ifi 1305 Feb 12  2019 Compila.cup
```
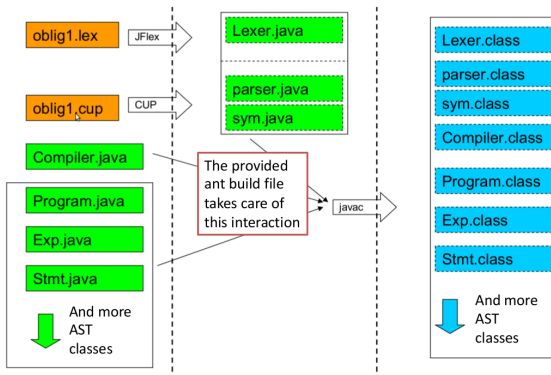
Building: putting it together

Section

Official

Deadline

Deadline

Friday 17. 03. 2023, 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 (linux) platform

Ask

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

Hand-in procedure

- as the previous years since some time, we use *git*
- https://github.uio.no resp.
  https://github.uio.no/compilerconstruction-inf5110/compila23-<xx>
- you need
  - a login
  - send me emails that you want to do oblig (+ potential partner) ⇒ I tell you group number
- see also the handout and Readme