TDDD55 Lesson 3
The Bison Parser generator & Intermediate Code generation

John Tinnerholm

john.tinnerholm@liu.se

LINKÖPING UNIVERSITY

Agenda

- Hour I
  - Lab III, Parser Generators
  - Lab IV, Intermediate code generation
- Hour 2
  - Lab work

Lab 3 Parser Generators

Lab 3

- Your task:
  - Create a parser from a language specification
  - You will use GNU Bison, an LARL(1) parser generator
  - Write specifications for expressions, conditions and function definitions
  - **Make sure that both children of an operator node have the same type**

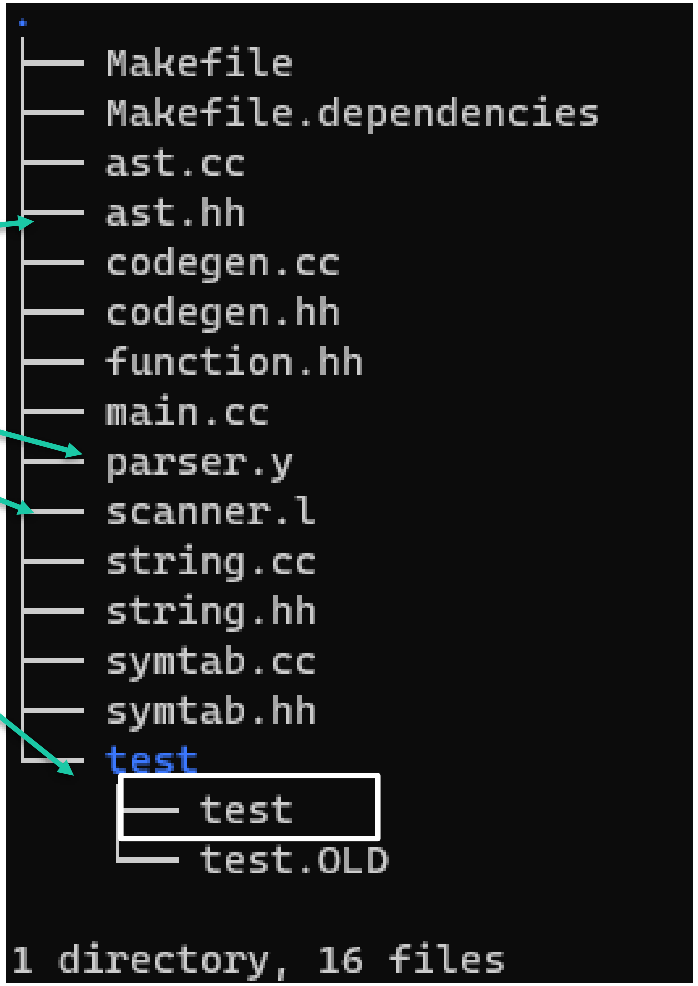
```
.
├── Makefile
├── Makefile.dependencies
├── ast.cc
├── ast.hh
├── codegen.cc
├── codegen.hh
├── function.hh
├── main.cc
├── parser.y
├── scanner.l
├── string.cc
├── string.hh
├── symtab.cc
├── symtab.hh
└── test
    ├── test
    └── test.OLD

1 directory, 16 files
```

Lab 3

Files for Lab 3

- The test file is used for testing your implementation
- Supply scanner.l from your lab-1 implementation (Introduce your rules, there is some existent setup code in the provided file)
- Note that simply copy paste will get you into trouble. Edit the scanner file appropriately



Lab 3

- The test file is used for testing your implementation
- Try your implementation by writing output to file and use the **diff tool with** trace-lab3.txt

```
diff -y <path-to-trace> <path-to-output-file>
```

- How to run with debugging information?

```
./compiler -d test/test
```

```
Starting parse
Entering state 0
Reading a token: Next token is token DECLARE ()
Shifting token DECLARE ()
Entering state 2
Reading a token: flex scanner jammed
```

```
├── lab3-4
│   ├── Makefile
│   ├── Makefile.dependencies
│   ├── ast.cc
│   ├── ast.hh
│   ├── codegen.cc
│   ├── codegen.hh
│   ├── function.hh
│   ├── main.cc
│   ├── parser.y
│   ├── scanner.l
│   ├── string.cc
│   ├── string.hh
│   ├── symtab.cc
│   ├── symtab.hh
│   └── test
│       ├── test
│       └── test.OLD
└── traces
    └── trace-lab3.txt
```

The Bison Parser generator

- The Bison parser generator is developed by the GNU project.
- Bison generates parsers to parse a supplied language
  - In order for Bison to parse a language, it must be described by a context-free grammar.
  - Bison is optimized for LR(1)[1]
- You need not to specify formal grammar as a part of the lab

[1]The grammar of your language is described in the instructions

Structure of Bison

%{

/ C declarations */*

%}

/ Bison declarations */*

%%

/ Grammar rules */*

%%

/ Additional C code */*

```
%{
#include <stdlib.h>
#include <iostream>
#include "string.hh"
#include "ast.hh"
#include "symtab.hh"

extern char                   *yytext;
extern int                     yylineno, errorCount, warningCount;
extern FunctionInformation    *currentFunction;

extern int yylex(void);
extern void yyerror(char *);
extern char CheckCompatibleTypes(Expression **, Expression **);
extern char CheckAssignmentTypes(LeftValue **, Expression **);
extern char CheckFunctionParameters(FunctionInformation *,
                                    VariableInformation *,
                                    ExpressionList      *);
char CheckReturnType(Expression **, TypeInformation *);
extern ostream& error(void);
extern ostream& warning(void);

#define YYDEBUG 1
%}
```

```
/*
 * We have multiple semantic types. The first couple of rules return
 * various kinds of symbol table information. The rules for the
 * program statements return nodes in the abstract syntax tree.
 *
 * The %union declaration declares all the kinds of data that
 * can be return. %type declarations later on will specify which
 * rules return what.
 */

%union
{
    ASTNode                 *ast;
    Expression              *expression;
    ExpressionList          *expressionList;
    Statement               *statement;
    StatementList           *statementList;
    Condition               *condition;
    ArrayReference          *aref;
    FunctionCall            *call;
    LeftValue               *lvalue;
    ElseIfList              *elseIfList;

    VariableInformation     *variable;
    TypeInformation         *type;
    FunctionInformation     *function;

  ::string                  *id;
    int                      integer;
    double                   real;
    void                    *null;
}

%type <expression>      expression
%type <expressionList>  expressions expressionz
%type <statement>       ifstmt whilestmt returnstmt callstmt assignstmt
```

```
%start program

%%


/*
 * A program is simply a list of variables, functions and
 * a code block. Very similar to a function really.
 */

program     :   variables functions block ';'
            {
                if (errorCount == 0)
                {
                    currentFunction->SetBody($3);
                    /* currentFunction->GenerateCode(); */
                    cout << currentFunction;
                }
            }
            ;

/*
 * We use this rule for all variable declarations.
 * Although parameters look almost the same, they
 * behave differently, so it's practical to have
 * separate rules for them.
 */

variables   :   DECLARE declarations
            |   error declarations
            |   /* Empty */
            ;
```

```
%%

int errorCount = 0;
int warningCount = 0;


/* --- Your code here ---
 *
 * Insert utility functions that you think you need here.
 */

/* It is reasonable to believe that you will need a function
 * that checks that two expressions are of compatible types,
 * and if possible makes a type conversion.
 * For your convenience a skeleton for such a function is
 * provided below. It will be very similar to CheckAssignmentTypes.
 */

/*
 * CheckCompatibleTypes checks that the expressions indirectly pointed
 * to by left and right are compatible. If type conversion is
 * necessary, the pointers left and right point to will be modified to
 * point to the node representing type conversion. That's why you have
 * to pass a pointer to pointer to Expression in these arguments.
 */

char CheckCompatibleTypes(Expression **left, Expression **right)
{
    return 0;
}
```

The file to the right is for the file parser.y

Implementing a simple calculator using Bison

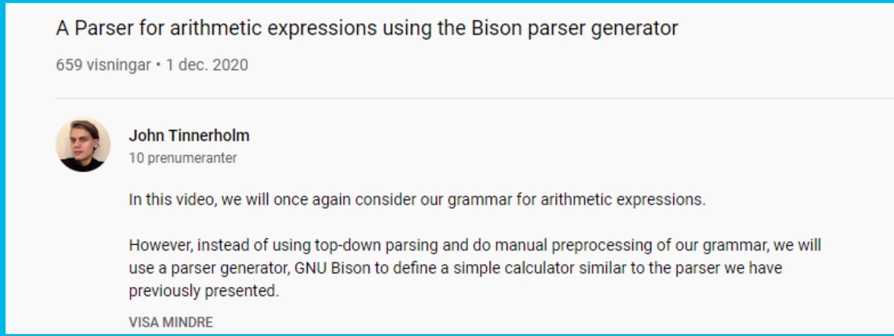
The Calculator grammar revisisted

- We will once again consider the grammar for arithmetic expressions.
- LR(K) vs LL(K)
- Let’s see how we can implement our calculator using Bison! (And flex..)

```
<exp> :=   <exp> + <term>
        |   <exp> - <term>
        |   <term>
<term> := <term> * <factor>
        |   <term> / <factor>
        |   <factor>
<factor>:= <num>
        |   (<exp>)
<num>:= [0-9]+
```

LR(K): Left to right scan, Rightmost derivation

Demonstration



Available on youtube as an extra resource

Resulting Bison-based Calculator

```
<exp> :=     <exp> + <term>
        |    <exp> - <term>
        |    <term>
<term> :=    <term> * <factor>
        |    <term> / <factor>
        |    <factor>
<factor>:=   <num>
        |    (<exp>)
<num>:= [0-9]+
```

Original grammar

```
/*C-Declarations*/
%{
#include <stdio.h>
#include <ctype.h>
/* Kill a warning */
int yylex();
#define TRUE 1

void yyerror (char const *s) {
  fprintf (stderr, "%s\n", s);
}

%}

/*Bison declarations*/
%token DIGIT
```

```
/*Bison rules*/
%%
line      : expr '\n'      { printf("> %d \n",$1); YYACCEPT;}
          | 'A' {YYABORT;} /* allows printing of the result */
expr      : expr '+' term  { $$ = $1 + $3;}
          | expr '-' term  { $$ = $1 - $3;}
          | term           { $$ = $1;}
          ;
term      : term '*' factor { $$ = $1 * $3; }
          | term '/' factor { $$ = $1 / $3; }
          | factor          { $$ = $1;}
          ;
factor    : num            { $$ = $1; }
          | '(' expr ')'   { $$ = $2; }
          ;
num       : DIGIT  { $$ = $1; }
```

```
/* Auxiliary C-Functions */
%%
int main() {
  while (TRUE) {
    int res = yyparse();
    if (res != 0) {
       return res;
    }
  }
}
int yylex(void) {
  int c;
  c = getchar();
  if (isdigit(c)) {
    yylval = c - '0';
    return DIGIT;
  }
  return c;
}
```

Inspiration from the desk calculator, see p 289 **Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. Compilers: Pinciples, Techniques, and Tools.**

Lab-3/4: The Language
A brief overview

The programming language for Lab 3 and Lab 4

- The language you are to compile is in some ways similar to Pascal but have syntax from C and Ada.
- A program consists of three sections.
  - The first section, declarations, holds declarations of all global variables.
  - The next section, functions, holds all functions defined in the program.
  - The final section, body, is a code block representing the main program body.

<program> ::= <variables> <functions> <block>;

//Both <variables>, <functions> and <block> might be **ε**

Function definitions

Function definitions start out with the keyword function followed by the function's name, parameters and return type. Next comes a block of local variable declarations and then local function declarations. The function is concluded with a code block for the function body.

Function that are declared within another function have access to the local variables and parameters of the surrounding function. **The language has a static scope**.

//Observe not exactly as in the lab.

<function> := function <name> (<parameters>) : <type>
<variables> <functions><block>

Task in parser.y

```
/* --- Your code here ---
 *
 * Write the function production. Take care to enter and exit
 * scope correctly. You'll need to understand how shift-reduce
 * parsing works and when actions are run to do this.
 *
 * Solutions that rely on shift-time actions will not be
 * acceptable. You should be able to solve the problem
 * using actions at reduce time only.
 *
 * In lab 4 you also need to generate code for functions after parsing
 * them. Just calling GeneratCode in the function should do the trick.
 */
```


A program and a function is essentially the same thing.
While the syntax is different there is a similarity in the semantics.

Declarations & Declaration blocks

- Declarations appear
  - At the beginning of a program
  - At the beginning of a function
- A declaration block starts with the keyword declare, followed by one or more declarations. The declaration block is terminated by the start of anything that does not look like a declaration.

<variables> := DECLARE <declarations>
| ε
<declarations> := <declarations>
| <declaration>
<declaration> := <name> : <type> ;

Code-blocks & Statements

- Code blocks are defined using the keyword **begin** and ended with the keyword **end** followed by a **;**
- Code blocks contain a list of statements

- Five statements
  - If-statments
  - Function calls
  - Assignments
  - Return statements
  - While statements

```
//Observe not exactly as in the lab (parser.y)
<block> := begin <statements> <end>
<statements> := <statements> <statement>
<statement> := <assign>
             |   <if>
             |   <while>
             |   <call>
             |   <return>
```

Statements

<if-statment> := if <condition> then <block> <elseifpart> <elsepart>

<elseifpart> := elseif <condition> then <block>

| **ε**

<elsepart> := else <block> if

| if

<while> := while <condition> do <block> while

<return> := return <expression>

<call> := <name> (<expressions>)

<assign> := <lvalue> assign <expression>

Lab 4
Intermediate Code Generation

Intermediate code

- Intermediate code, sometimes also refereed to as intermediate representations
  - Platform independent
  - Easier to work with
  - Optimizations, such as estimating optimal register allocation and constant folding. Also different optimizations are suitable on different levels of the IR.
- Examples
- Postfix or reverse polish notation
  - HP calculators!
  - 4 4 + = 8
- Triples
- **Quadruples**
  - This is what we will look at in the lab

Demonstration

Quadruples

- Quadruples is a low level intermediate represenatation consisting of four parts.
  - Operator
  - Operands 1 & 2
  - Result

Operator	Operand 1	Operand 2	Result

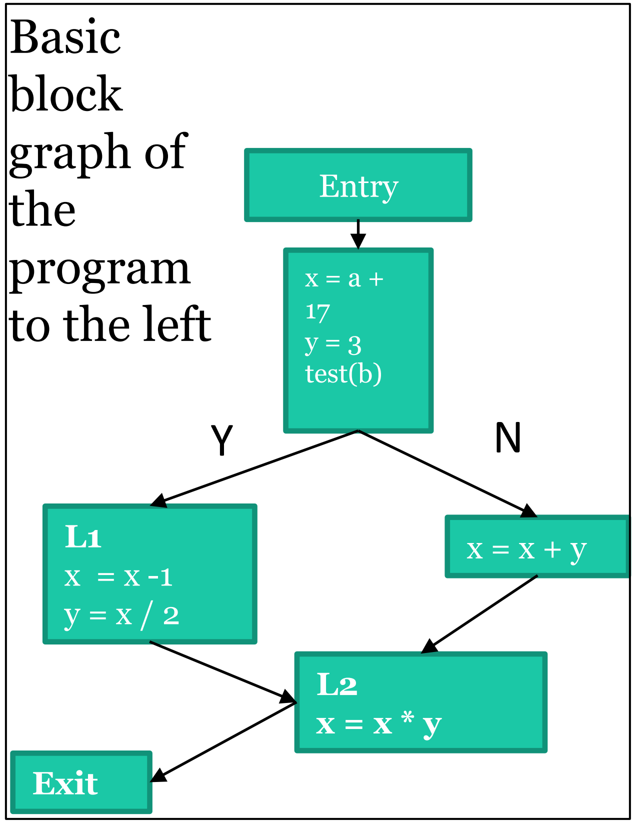
(A + B) * (C + D) - E

Operator	Operand 1	Operand 2	Result
+	A	B	TEMP 1
+	C	D	TEMP 2
*	TEMP 1	TEMP 2	TEMP 3
-	TEMP 3	E	TEMP 4

Basic blocks

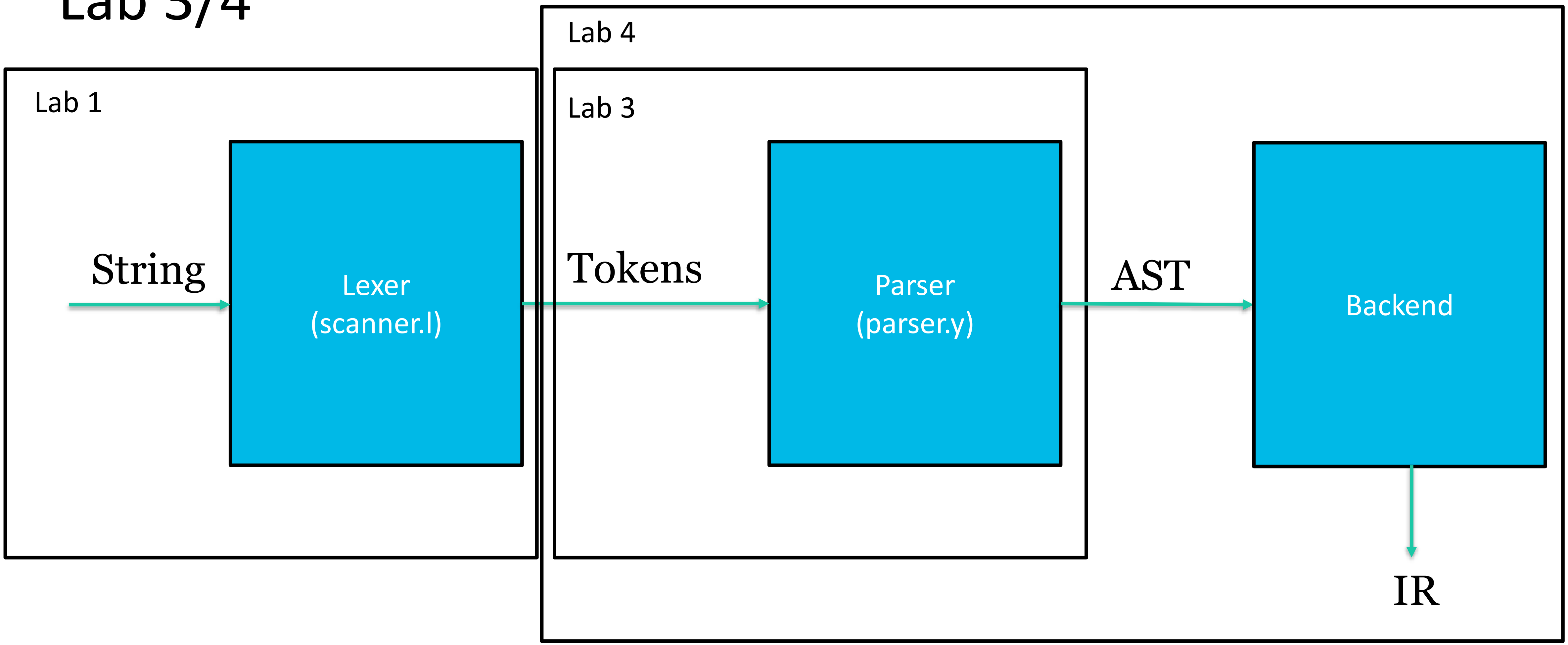
- A basic block
  - Code without control flow
- One entry and one exit
- *Some languages consider function calls to terminate basic blocks*
- *The basic block forms the vertices of the control flow graph*
- Basis of many optimization algortihms
  - More on this in the lectures

```
1.    x = a + 17;
2.    y = 3;
3.    if (b) goto L1;
4.    x = x + y;
5.    goto L2;
L1:
1.    x = x - 1;
2.    y = x / 2;
L2:
1.    x = x * y;
```

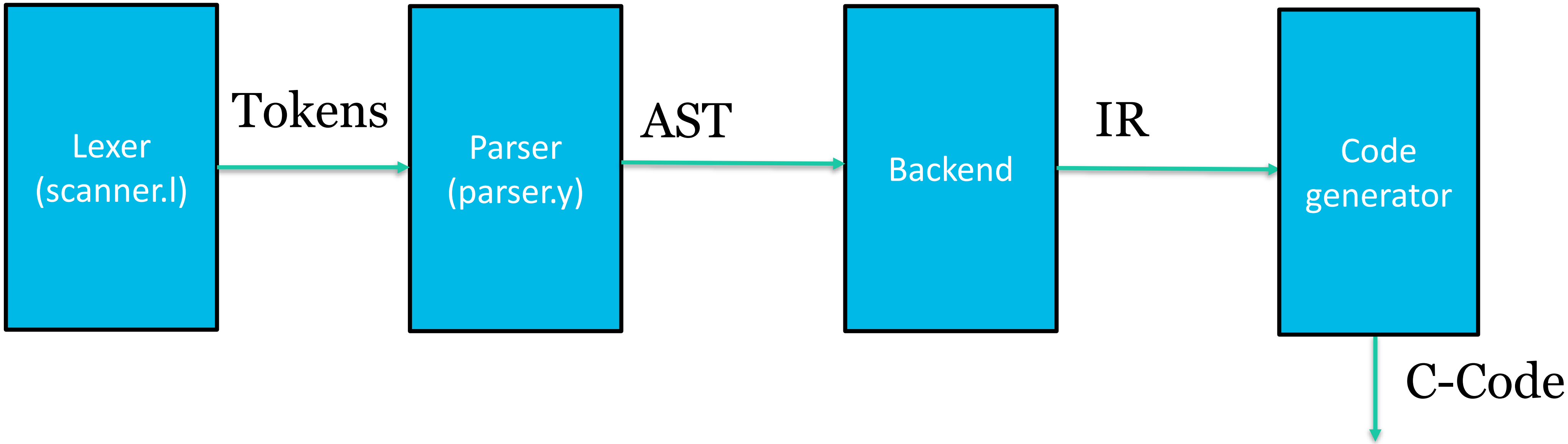


In the lab this structure is given implicitly by the quadruples

Lab 3/4


Lab 4
Lab 1
Lab 3
String
Lexer
(scanner.l)
Tokens
Parser
(parser.y)
AST
Backend
IR

Lab 3/4


Lexer
(scanner.l)
Tokens
Parser
(parser.y)
AST
Backend
IR
Code
generator
C-Code


GCC

Lab 4 Intermediate code generation

- The purpose of this exercise is to learn about how parse trees can be translated into intermediary code.
- Write methods for
  - If statements (including the elseif and else branches)
  - Array references
  - All binary operators
    - Use BinaryGenerateCode()
- You will work in codegen.cc

When completed, you should have a program that is capable of generating intermediate code for the small programming language used in exercises two, three and four.

Lab 4 Computing absolute and relative adress of arrays

- The absolute address is computed as follows: absolute_adress = base_adress + **array_type_size** * index

- Say that we want to access the first element:
  - 0 + 0x0 * 0 = 0x0 = 0
- Element two:
  - 0 + 0x20 * 2 = 0x40 = 64

Memory adress	0X0 D0	0X20 D32	0X40 D64	0X60 D128
Index	0	1	2	3
Element	1	2	3	4

- **Note that the size of the integer and real in our language is 32 bits**
- **But in pratice it will be 64.**
- **Use the sizeof operator**

If we would have had 1 as our start index the formula is a bit different

Lab 4 Theory question

- **Theory exercise**
  - Demonstrate how badly generated code could be optimised. Do so by suggesting a **concrete algorithm with a concrete** example of algorithm would transform the presented code
  - Of course the algortihm could be implemented in the code generator as well, however, that is optional
    - Please do as an extra exercise☺

One more thing…

- I am looking for thesis workers
  - Interested in Compiler Construction
    - Work on a visualization tool to showcase complex systems and battle climate change
    - Visual GUI testing for equation oriented languages
    - Language Server for equation oriented languages
    - FMI/FMU support for equation oriented languages
- **Work with Compilers/Interpreters + Julia + interface design = Awesome!**
- **Possible result is a research report**

Thank you!