TDDD55- Compilers and Interpreters

Lesson 1

November 1 2011

Kristian Stavåker (kristian.stavaker@liu.se)

Department of Computer and Information Science
Linköping University

PURPOSE OF LESSONS

The purpose of the lessons is to practice some theory, introduce the laboratory assignments, and prepare for the final examination.

Read the laboratory instructions, the course book, and the lecture notes.

All the laboratory instructions and material available in the ***course directory, ~TDDD55/lab/***. Most of the PDF's also available from the course homepage.

LABORATORY ASSIGNMENTS

In the laboratory exercises you should get some practical experience in compiler construction.

There are 4 separate assignments to complete in 4x2 laboratory hours. You will also (most likely) have to work during non-scheduled time.

LESSON SCHEDULE

November 1,	13.15 -15:	Formal languages and automata theory
November 11,	8.15 - 10:	Formal languages and automata theory, Flex
November 22,	15.15 - 17:	Bison and intermediate code generation
December 6,	15.15 - 17:	Exam preparation

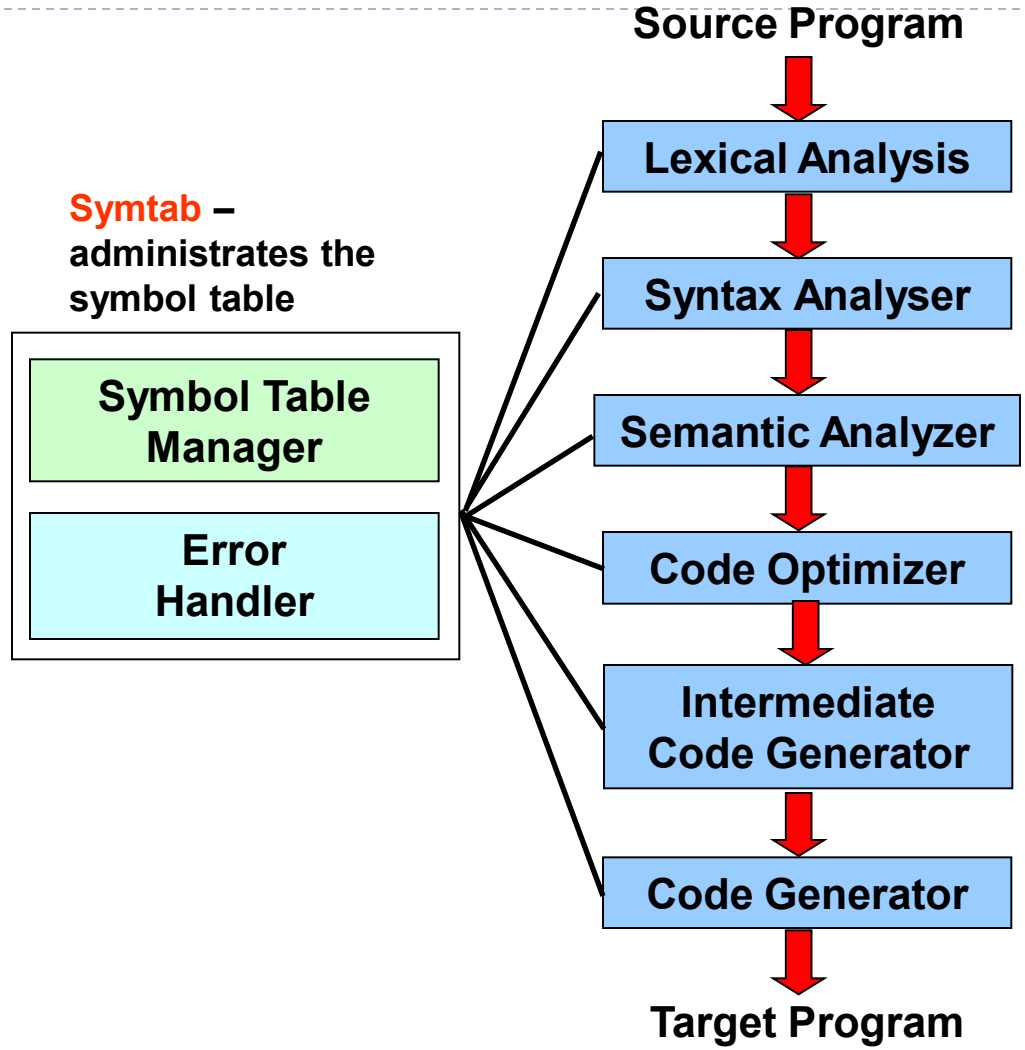
HANDING IN AND DEADLINE

- Demonstrate the working solutions to your lab assistant during scheduled time. Then send the modified files to the same assistant as well as answers to questions if any (put *TDDD55 <Name of the assignment>* in the topic field). One e-mail per group.

- Deadline for all the assignments is: **December 15, 2011**.

- Remember to register yourself in the webreg system, www.ida.liu.se/webreg (closed, e-mail your laboratory assistant)

PHASES OF A COMPILER



Scanner – manages lexical analysis

Parser – manages syntactic analysis, build internal form

Semantics – checks static semantics

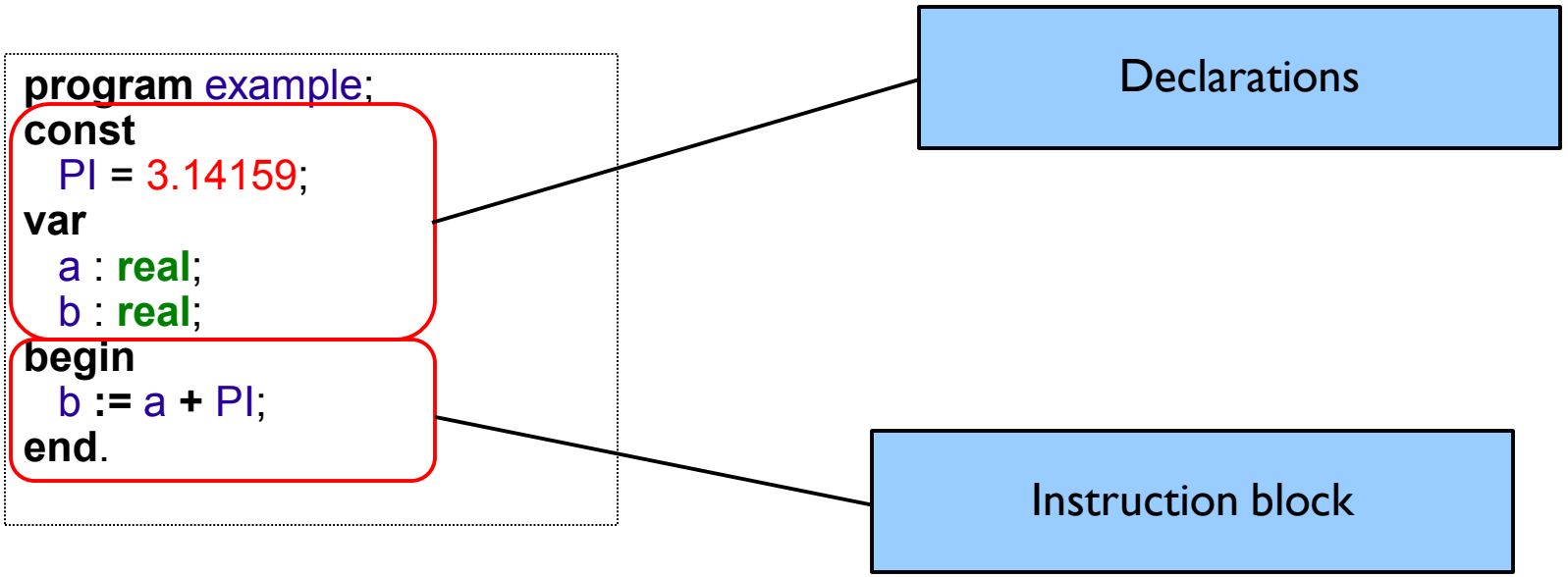
Optimizer – optimizes the internal form

Quads – generates quadruples from the internal form

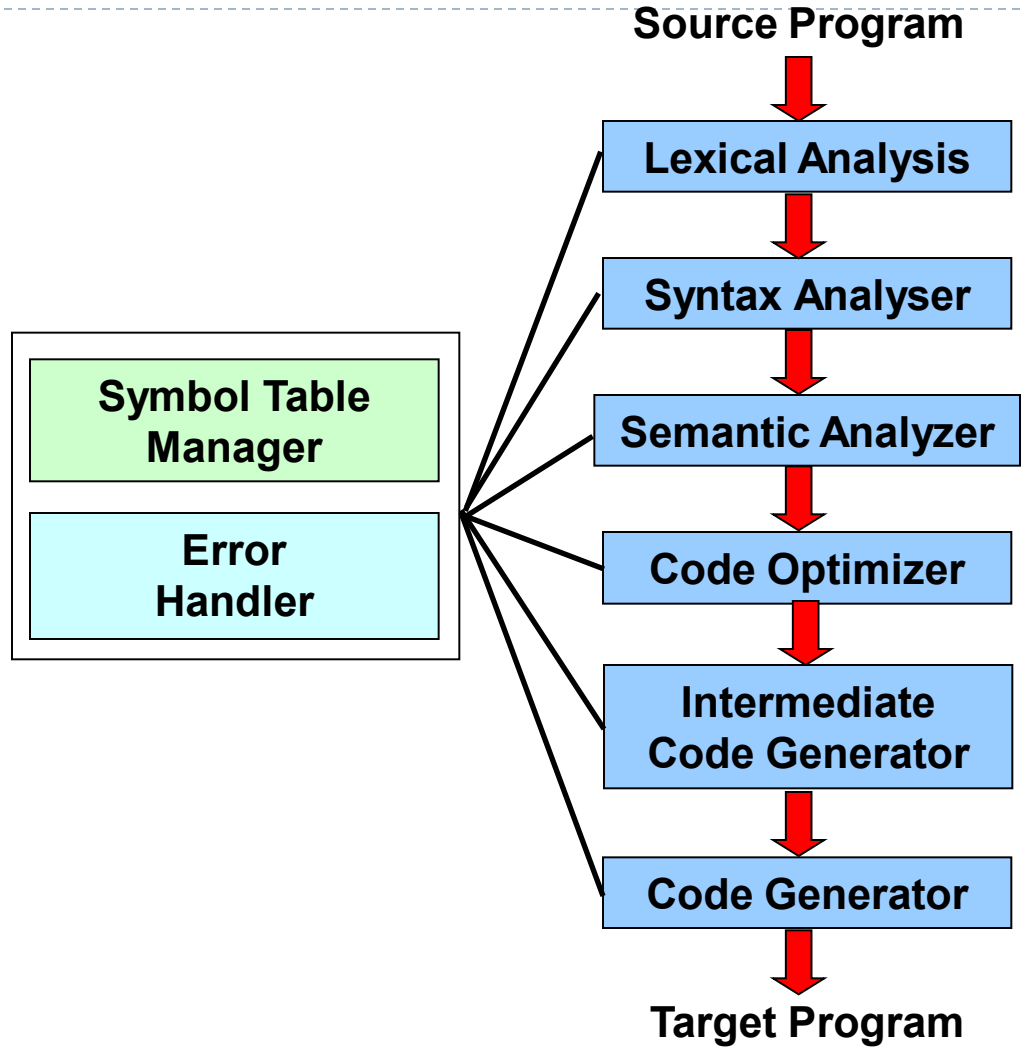
Codegen – expands quadruples into assembly

PHASES OF A COMPILER

Let's consider this program:



PHASES OF A COMPILER

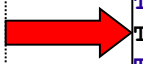


Scanner – manages lexical analysis

PHASES OF A COMPILER (SCANNER)

INPUT

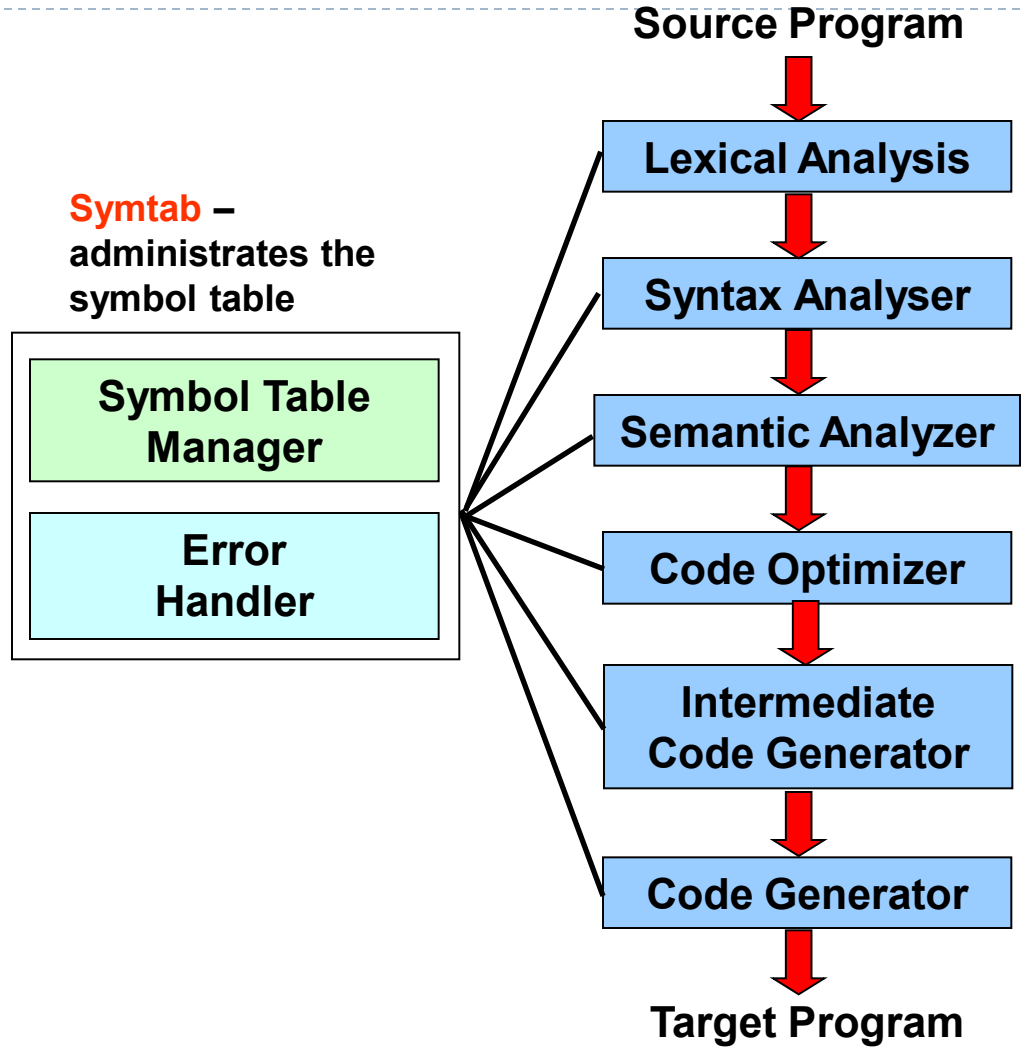
```
program example;
const
   PI = 3.14159;
var
   a : real;
   b : real;
begin
   b := a + PI;
end.
```



OUTPUT

token	pool_p	val	type
T_PROGRAM			keyword
T_IDENT	EXAMPLE		identifier
T_SEMICOLON			separator
T_CONST			keyword
T_IDENT	PI		identifier
T_EQ			operator
T_REALCONST		314159	constant
T_SEMICOLON			separator
T_VAR			keyword
T_IDENT	A		identifier
T_COLON			separator
T_IDENT	REAL		identifier
T_SEMICOLON			separator
T_IDENT	B		identifier
T_COLON			separator
T_IDENT	REAL		identifier
T_SEMICOLON			separator
T_BEGIN			keyword
T_IDENT	B		identifier
T_ASSIGNMENT			operator
T_IDENT	A		identifier
T_ADD			operator
T_IDENT	PI		identifier
T_SEMICOLON			separator
T_END			keyword
T_DOT			separator

PHASES OF A COMPILER

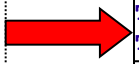


Scanner – manages lexical analysis

PHASES OF A COMPILER (SYMTAB)

INPUT

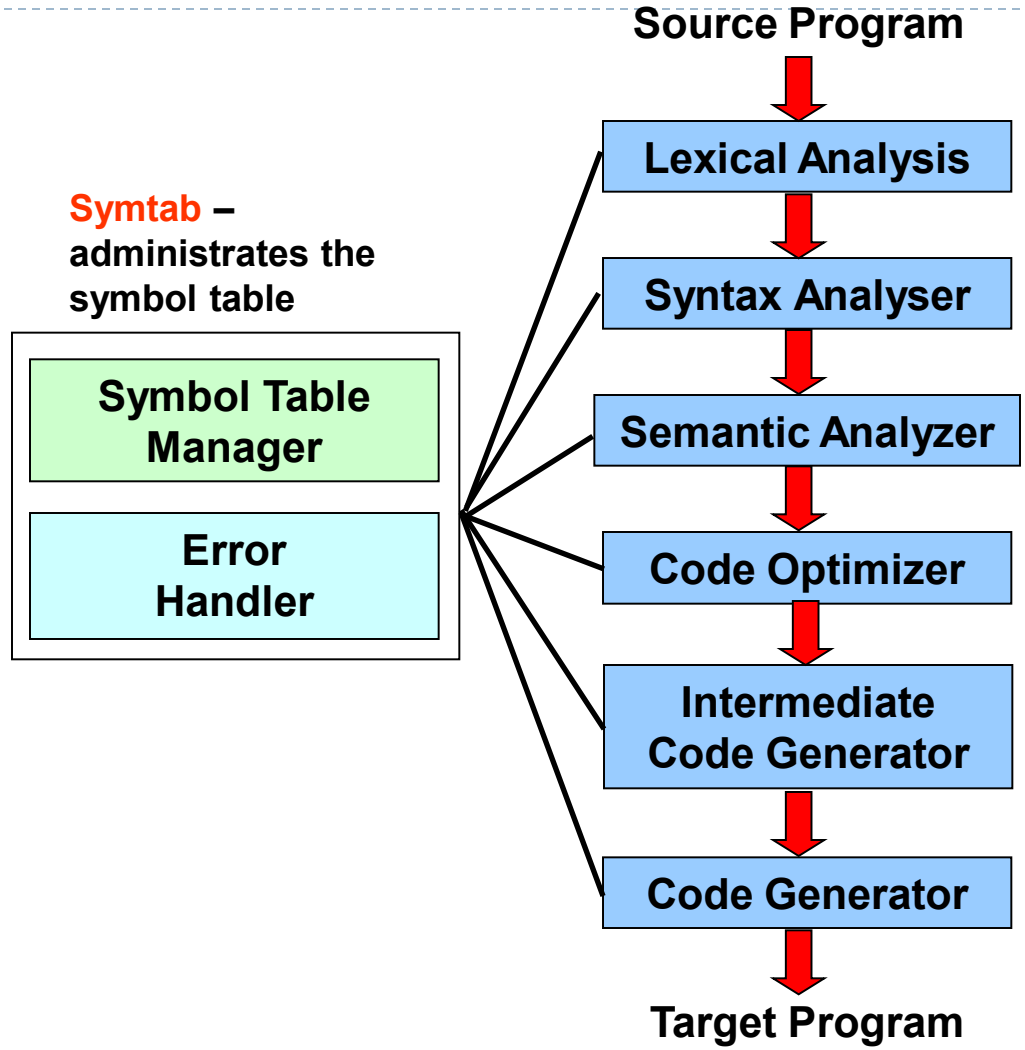
```
program example;
const
   PI = 3.14159;
var
   a : real;
   b : real;
begin
   b := a + PI;
end.
```



OUTPUT

token	pool_p	val	type
T_IDENT	VOID		
T_IDENT	INTEGER		
T_IDENT	REAL		
T_IDENT	EXAMPLE		
T_IDENT	PI	3.14159	REAL
T_IDENT	A		REAL
T_IDENT	B		REAL

PHASES OF A COMPILER



Scanner – manages lexical analysis

Parser – manages syntactic analysis, build internal form

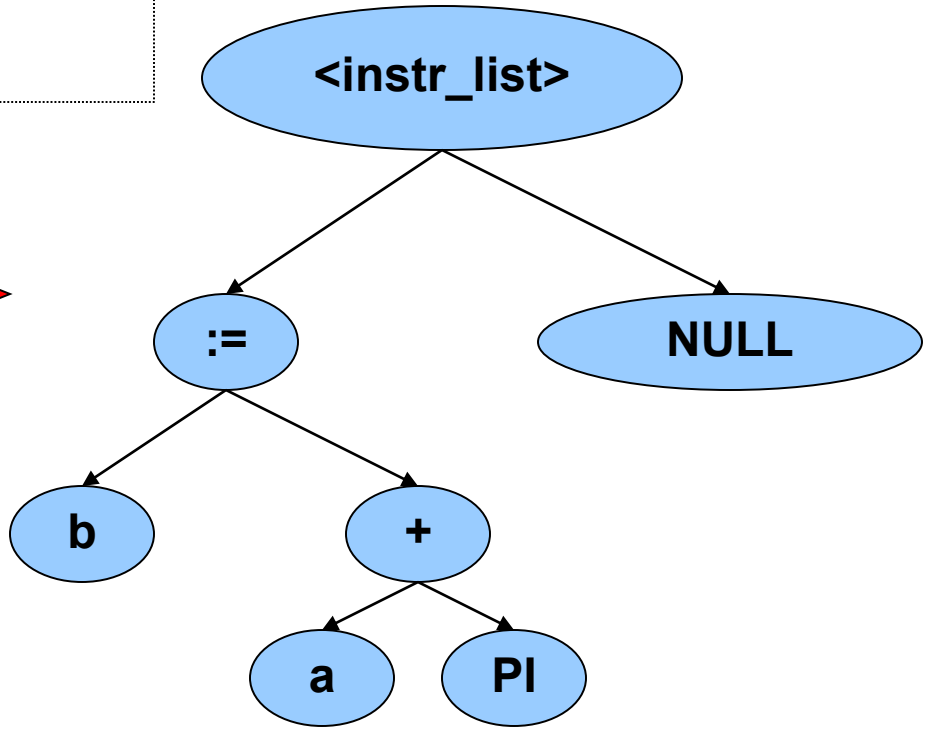
PHASES OF A COMPILER (PARSER)

INPUT

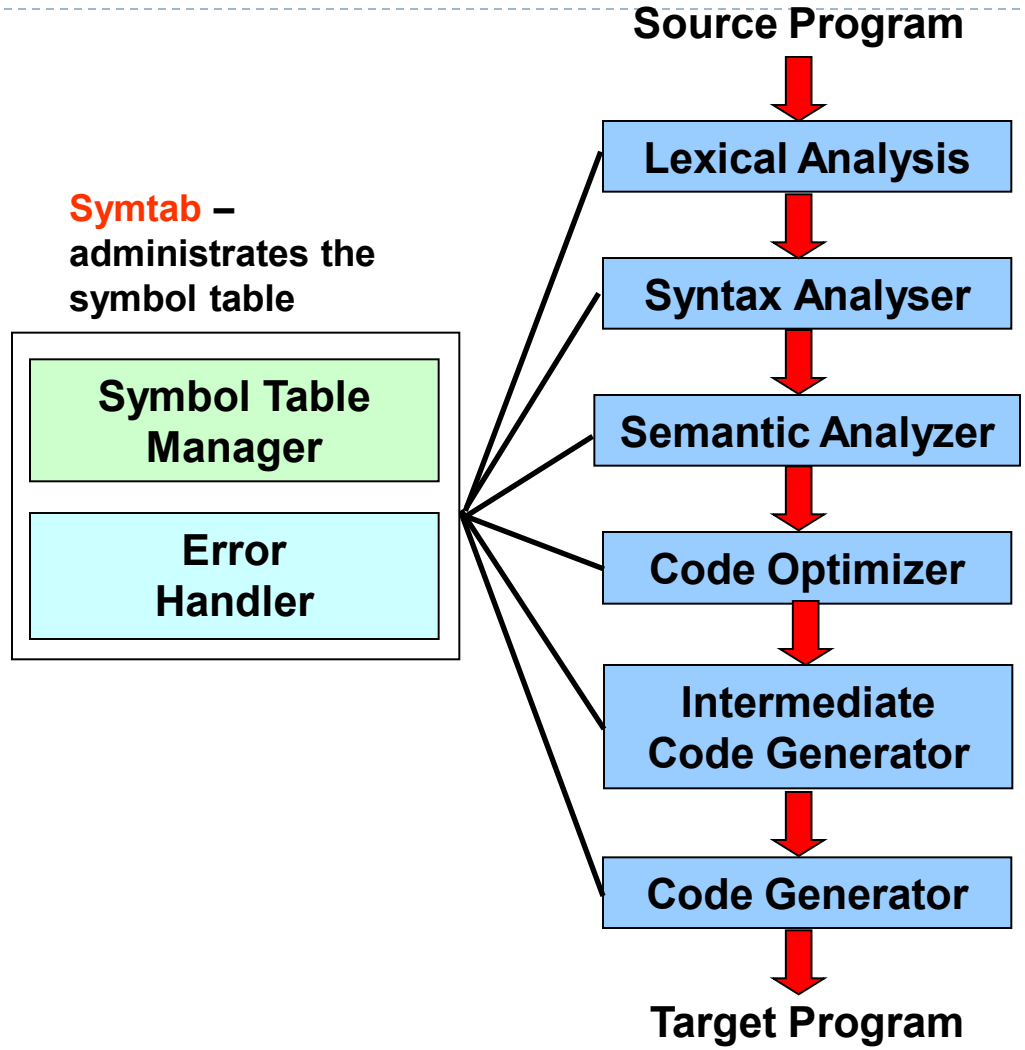
token	pool_p	val	type
T_PROGRAM			keyword
T_IDENT	EXAMPLE		identifier
T_SEMICOLON			separator
T_CONST			keyword
T_IDENT	PI		identifier
T_EQ			operator
T_REALCONST		3.14159	constant
T_SEMICOLON			separator
T_VAR			keyword
T_IDENT	A		identifier
T_COLON			separator
T_IDENT	REAL		identifier
T_SEMICOLON			separator
T_IDENT	B		identifier
T_COLON			separator
T_IDENT	REAL		identifier
T_SEMICOLON			separator
T_BEGIN			keyword
T_IDENT	B		identifier
T_ASSIGNMENT			operator
T_IDENT	A		identifier
T_ADD			operator
T_IDENT	PI		identifier
T_SEMICOLON			separator
T_END			keyword
T_DOT			separator

```
program example;
const
  PI = 3.14159;
var
  a : real;
  b : real;
begin
  b := a + PI;
end.
```

OUTPUT



PHASES OF A COMPILER



Scanner – manages lexical analysis

Parser – manages syntactic analysis, build internal form

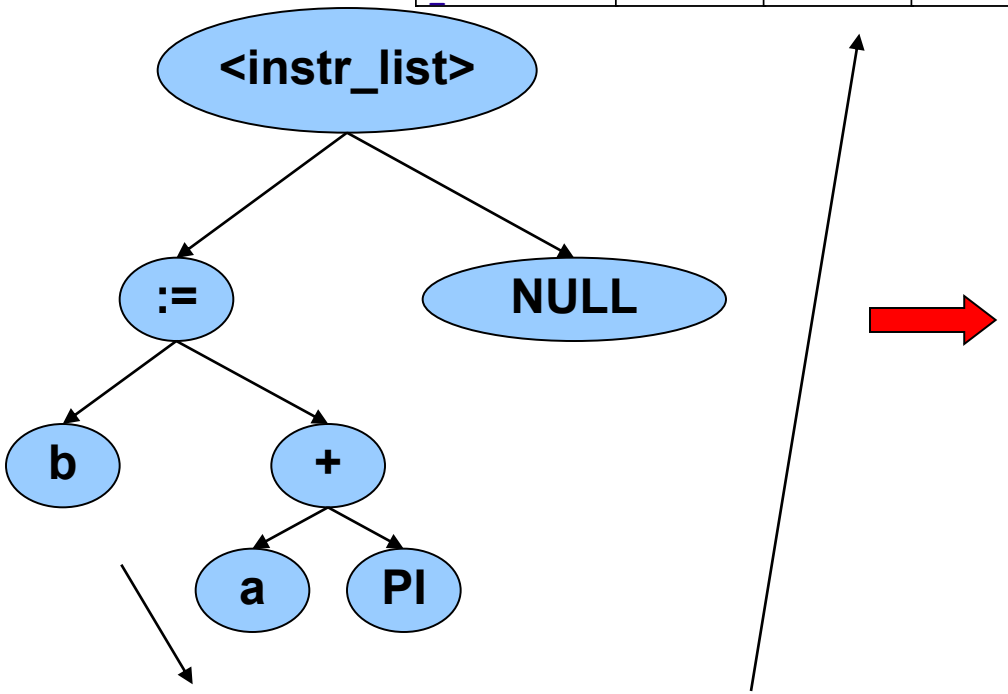
Semantics – checks static semantics

PHASES OF A COMPILER (SEMANTICS)

INPUT

OUTPUT

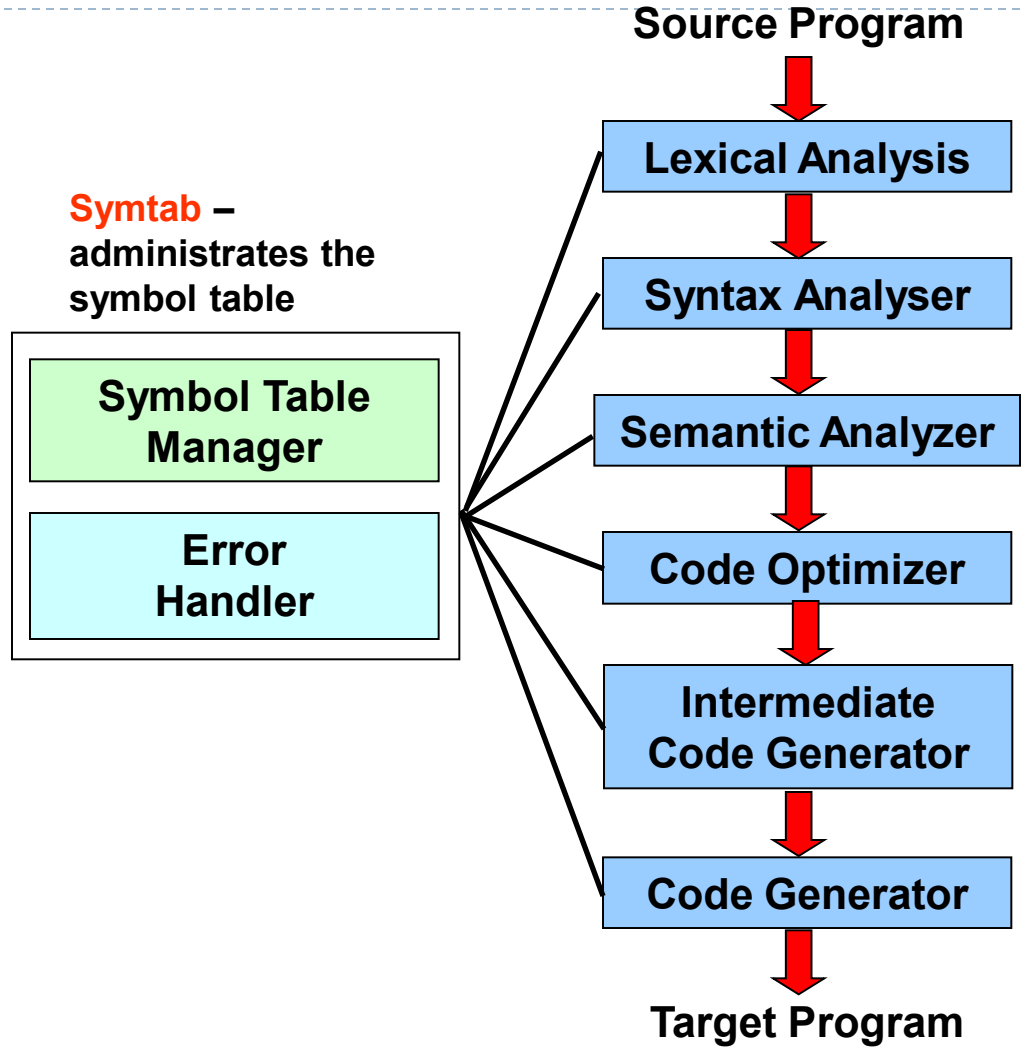
token	pool_p	val	type
T_IDENT	VOID		
T_IDENT	INTEGER		
T_IDENT	REAL		
T_IDENT	EXAMPLE		
T_IDENT	PI	3.14159	REAL
T_IDENT	A		REAL
T_IDENT	B		REAL



type(a) == type(b) == type(PI) ?

YES

PHASES OF A COMPILER



Scanner – manages lexical analysis

Parser – manages syntactic analysis, build internal form

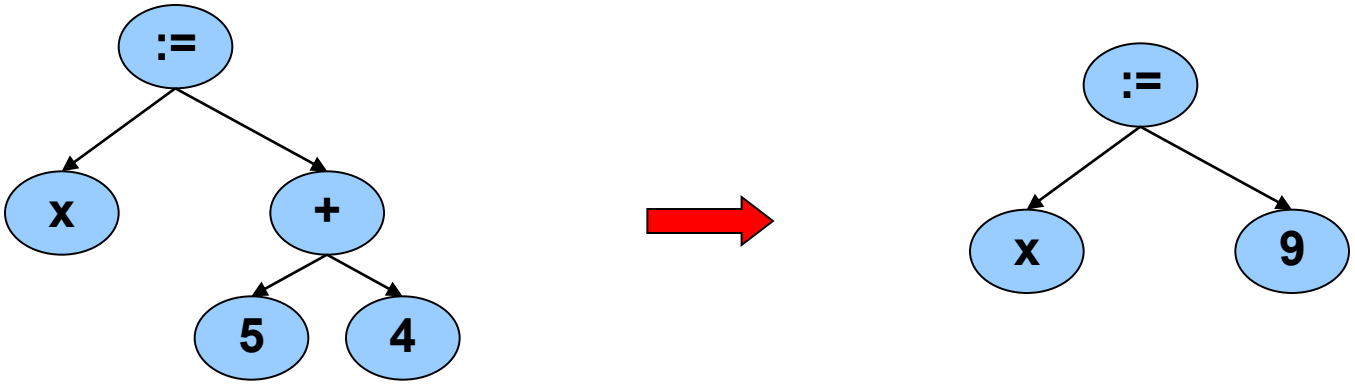
Semantics – checks static semantics

Optimizer – optimizes the internal form

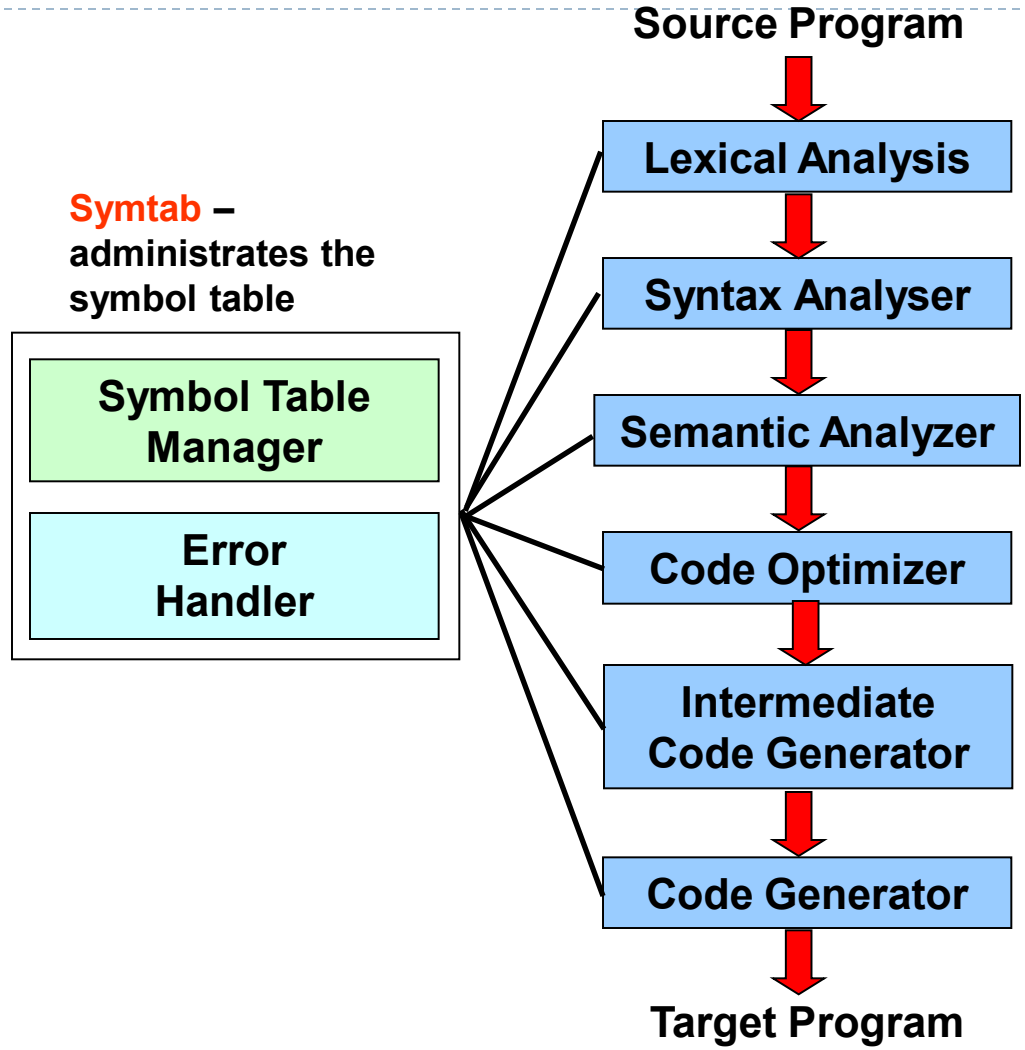
PHASES OF A COMPILER (OPTIMIZER)

INPUT

OUTPUT



PHASES OF A COMPILER



Scanner – manages lexical analysis

Parser – manages syntactic analysis, build internal form

Semantics – checks static semantics

Optimizer – optimizes the internal form

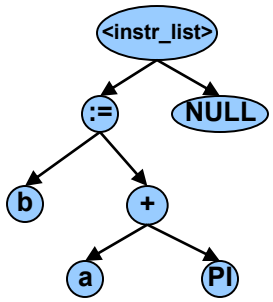
Quads – generates quadruples from the internal form

PHASES OF A COMPILER (QUADRUPLES)

INPUT

OUTPUT

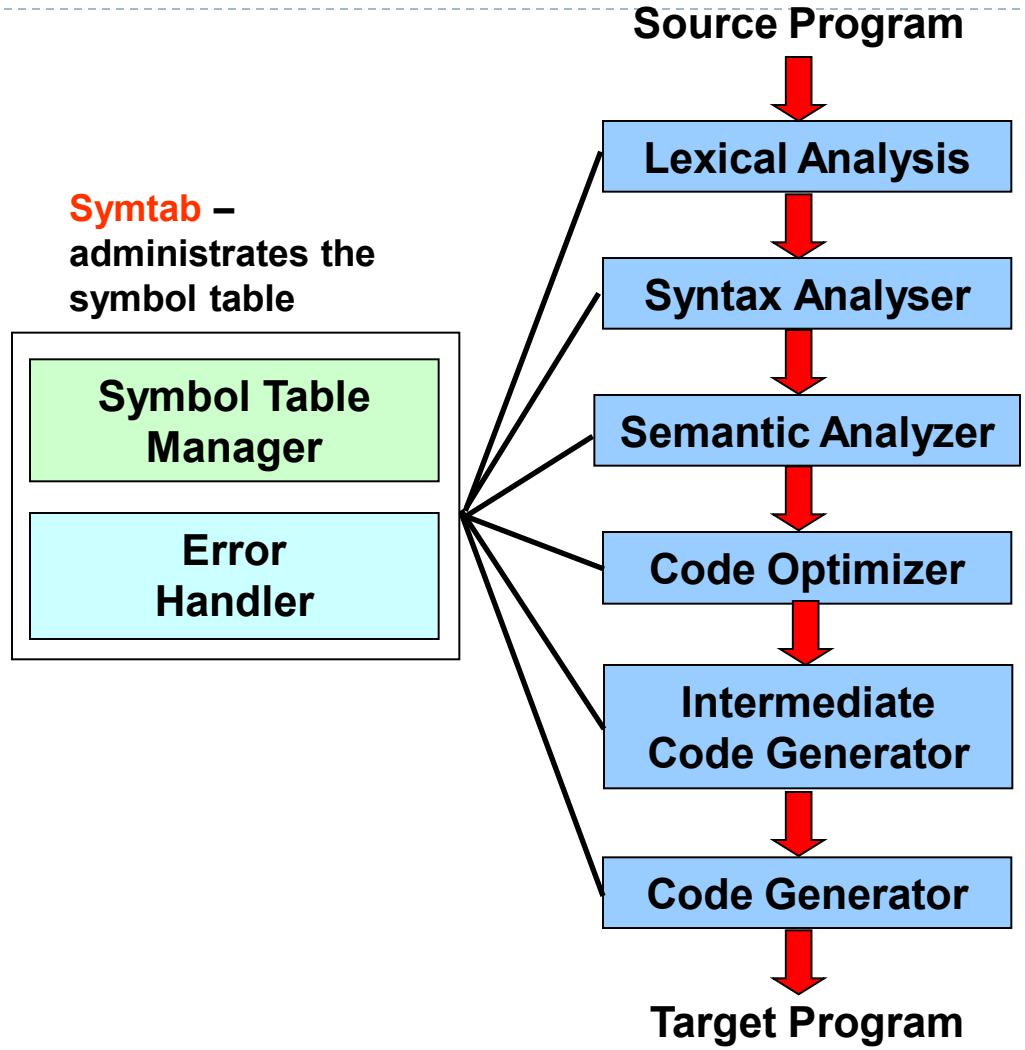
```
program example;
const
   PI = 3.14159;
var
   a : real;
   b : real;
begin
   b := a + PI;
end.
```




```
q_rplus   A   PI   $1
q_rassign $1  -    B
q_labl    4   -    -
```

Several steps

PHASES OF A COMPILER



Scanner – manages lexical analysis

Parser – manages syntactic analysis, build internal form

Semantics – checks static semantics

Optimizer – optimizes the internal form

Quads – generates quadruples from the internal form

Codegen – expands quadruples into assembly

PHASES OF A COMPILER (CODEGEN)

INPUT

```
program example;
const
   PI = 3.14159;
var
   a : real;
   b : real;
begin
   b := a + PI;
end.
```

OUTPUT

```
#include "diesel_glue.s"
L3:          ! EXAMPLE
      set     -104,%l0
      save     %sp,%l0,%sp
      st      %g1,[%fp+64]
      mov      %fp,%g1
      ld      [%g1-4],%f0
      set     1078530000,[%sp+64]
      ld      [%sp+64],%f1
      fadds    %f0,%f1,%f2
      st      %f2,[%g1-12]
      ld      [%g1-12],%o0
      st      %o0,[%g1-8]
L4:
      ld      [%fp+64],%g1
      ret
      restore
```

Several steps

LABORATORY ASSIGNMENTS

Lab 1 Attribute Grammars and Top-Down Parsing
Lab 2 Scanner Specification
Lab 3 Parser Generators
Lab 4 Intermediate Code Generation

1. Attribute Grammars and Top-Down Parsing

- Some grammar rules are given
- Your task:
  - Rewrite the grammar (elimate left recursion, etc.)
  - Add attributes to the grammar
  - Implement your attribute grammar in a C++ class named **Parser**. The **Parser** class should contain a method named **Parse** that returns the value of a single statement in the language.

2. Scanner Specification

- Finish a scanner specification given in a *scanner.l* flex file, by adding rules for C and C++ style comments, identifiers, integers, and reals.
- More on flex in lesson 2.

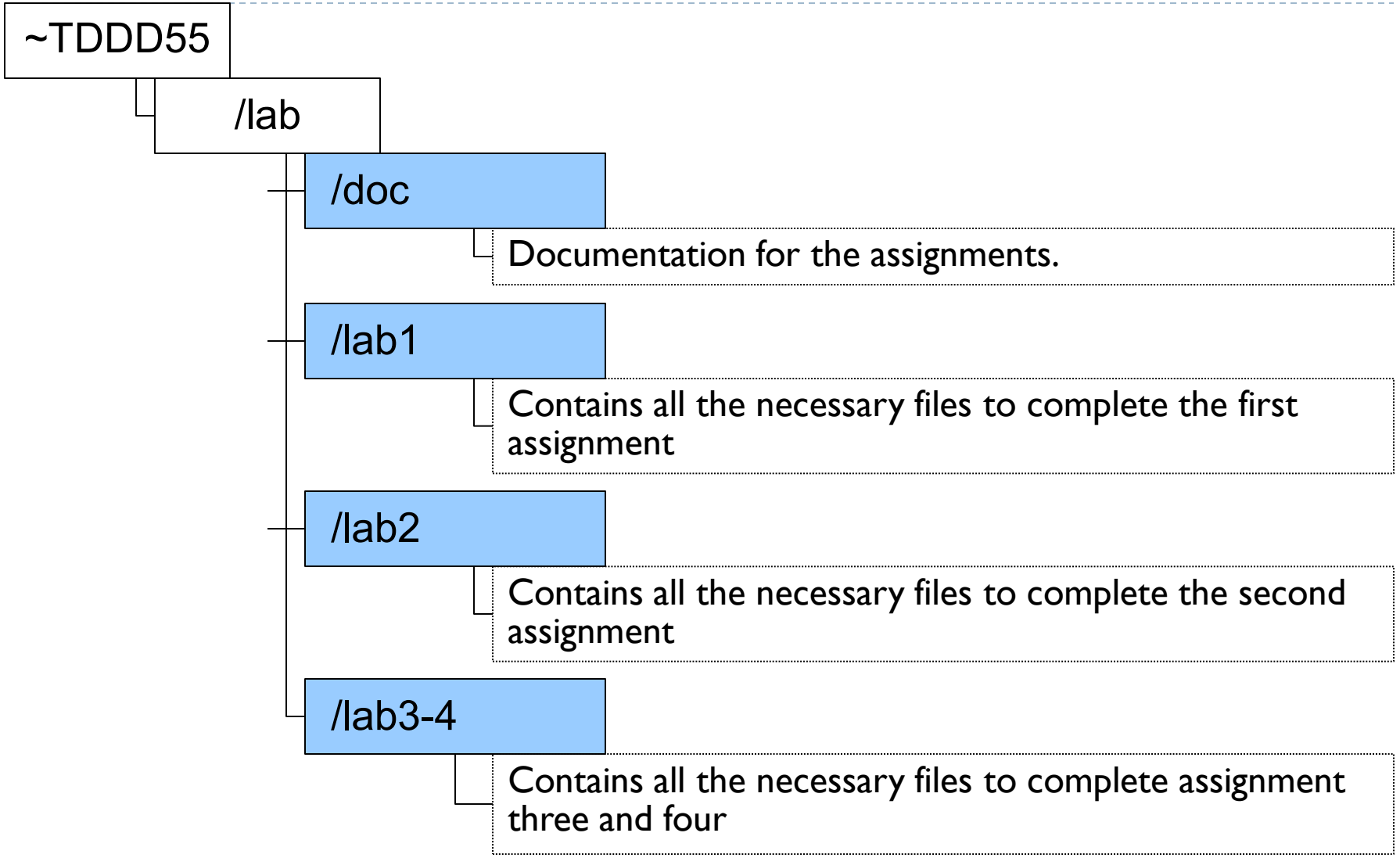
3. Parser Generators

- Finish a parser specification given in a *parser.y* bison file, by adding rules for expressions, conditions and function definitions, ...You also need to augment the grammar with error productions.
- More on bison in lesson 3.

4. Intermediate Code Generation

- The purpose of this assignment to learn about how parse trees can be translated into intermediate code.
- You are to finish a generator for intermediate code by adding rules for some language statements.
- More in lesson 3

LABORATORY SKELETON



INSTALLATION

- Take the following steps in order to install the lab skeleton on your system:
  - Copy the source files from the course directory onto your local account:

```
mkdir TDDD55
cp -r ~TDDD55/lab TDDD55
```

  - You might also have to load some modules (more information in the laboratory instructions).

HINTS LABORATORY ASSIGNMENT 1

REWRITING THE GRAMMAR

- Use one non-terminal for each precedence level.

  E ::= E + E | E – E | T

  T ::= T * T | T / T | …

- (Left) Associativity:

  E ::= E + E | E – E | T => E ::= E + T | E – T | T

- See for instance:

http://www.lix.polytechnique.fr/~catuscia/teaching/cg428/02Spring/lecture_notes/L03.html

REWRITING THE GRAMMAR (2)

- Transform the grammar to right recursive form:

  A ::= A α | β (where β may not be preceded by A)

  is rewritten to

  A ::= β A'
  A' ::= α A' | ε

- See *Lecture 5 Syntax Analysis, Parsing*

IMPLEMENTATION

- You have been give a main function in *main.cc*.

```
int main(void) {
        Parser parser; double val;

        while (1) {

                try    {
                   cout << "Expression: " << flush;
                   val = parser.Parse();
                   cout << "Result:     " << val << '\n' << flush;
                }
                catch (ScannerError& e) {
                     cerr << e << '\n' << flush;
                     parser.Recover();
                }
                catch (ParserError) { parser.Recover(); }

                 catch (ParserEndOfFile)  { cerr << "End of file\n" << flush; exit(0); }
                }
        }
}
```

IMPLEMENTATION (2)

- You have also been given files *lab1.cc* and *lab1.hh* for implementing your *Parser* class.
- In the function *Parse*, start the parsing.

```
double Parser::Parse(void) {
        Trace x("Parse");
        double val;
        val= 0;
        crt_token = the_scanner.Scan();
        switch (crt.token.type)
        {
                case kIdentifier:
                case kNumber:
                case kLeftParen:
                case kMinus:
                        val = pExpression();
                        if (crt_token.type != kEndOfLine) throw ParserError();
                        return val;
                default: throw ParserError();
        }
        return val;
}
```

IMPLEMENTATION (3)

- Add one function for each non-terminal in the grammar to your *Parser* class.
- See Lecture 5 *Syntax Analysis, Parsing*

```
double Parser::pExpression(void) {
      switch (crt_token.type) {
      ... ...
      }
}
```

IMPLEMENTATION (3)

- You don't need to change anything in *lex.cc* and *lex.hh*.
- Also implement some simple error recovery in your *Parser* class.