
COMPILER CONSTRUCTION

Seminar 01 - TDDDB44 2016

Martin Sjölund (martin.sjolund@liu.se)

Mahder Gebremedhin (mahder.gebremedhin@liu.se)

Department of Computer and Information Science

Linköping University

SEMINARS AND LABS

In the laboratory exercises, you shall complete a compiler for DIESEL – a small Pascal like language, giving you a practical experience of compiler construction

There are 7 separate parts of the compiler to complete in **11x2** laboratory hours and **4x2** seminar hours. You will also have to work during non-scheduled time.

PURPOSE OF SEMINARS

The purpose of the seminars is to introduce you to the lab

You need to read the introductions, the course book and the lecture notes!

You can buy the lab compendium as well as a small exercise compendium (suitable as a revision for the exam) in the student book store in Kårallen.

SEMINARS

Seminar 1 (Today): Lab 1 & 2

Seminar 2: Lab 3 & 4

Seminar 3: Lab 5, 6 & 7

Seminar 4: Exam prep

RELATING LABS TO THE COURSE

- Building a complete compiler
 - We use a language that is small enough to be manageable.
 - Scanner, Parser, Semantic Elaboration, Code Generation.
 - Experience in compilation and software engineering.

LAB EXERCISES

This approach (building a whole compiler) has several advantages and disadvantages:

Advantages

- Students gains deep knowledge
- Experience with complex code
- Provides a framework for the course
- Success instills confidence

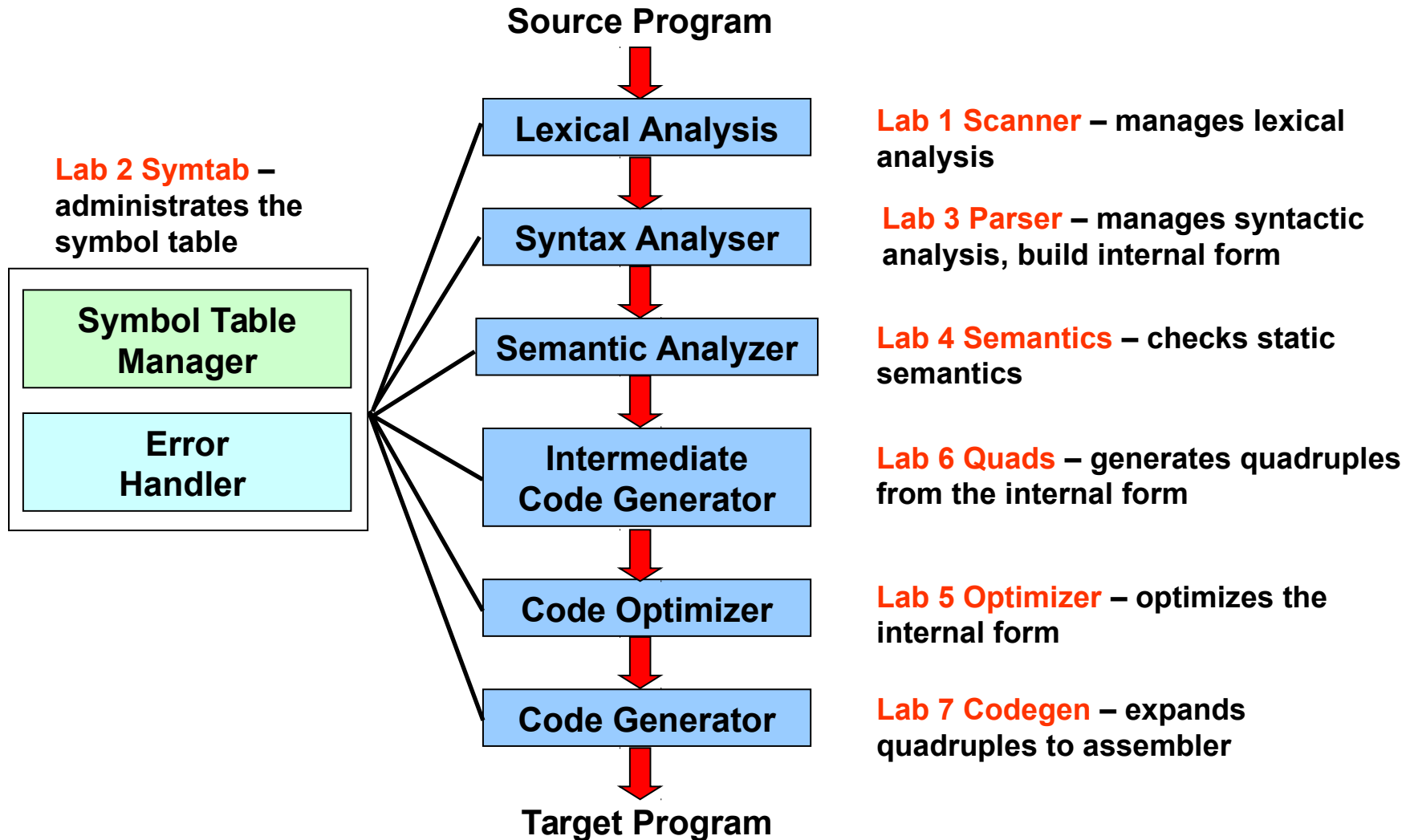
Disadvantages

- High ratio of programming to thought
- Cumulative nature magnifies early failures
- Many parts are simplified

LABS

Lab 0	Formal languages and grammars
Lab 1	Creating a scanner using " flex "
Lab 2	Symbol tables.
Lab 3	LR parsing and abstract syntax tree construction using " bison "
Lab 4	Semantic analysis (type checking)
Lab 5	Optimization
Lab 6	Intermediary code generation (quads)
Lab 7	Code generation (assembler) and memory management

PHASES OF A COMPILER



PHASES OF A COMPILER (cont'd)

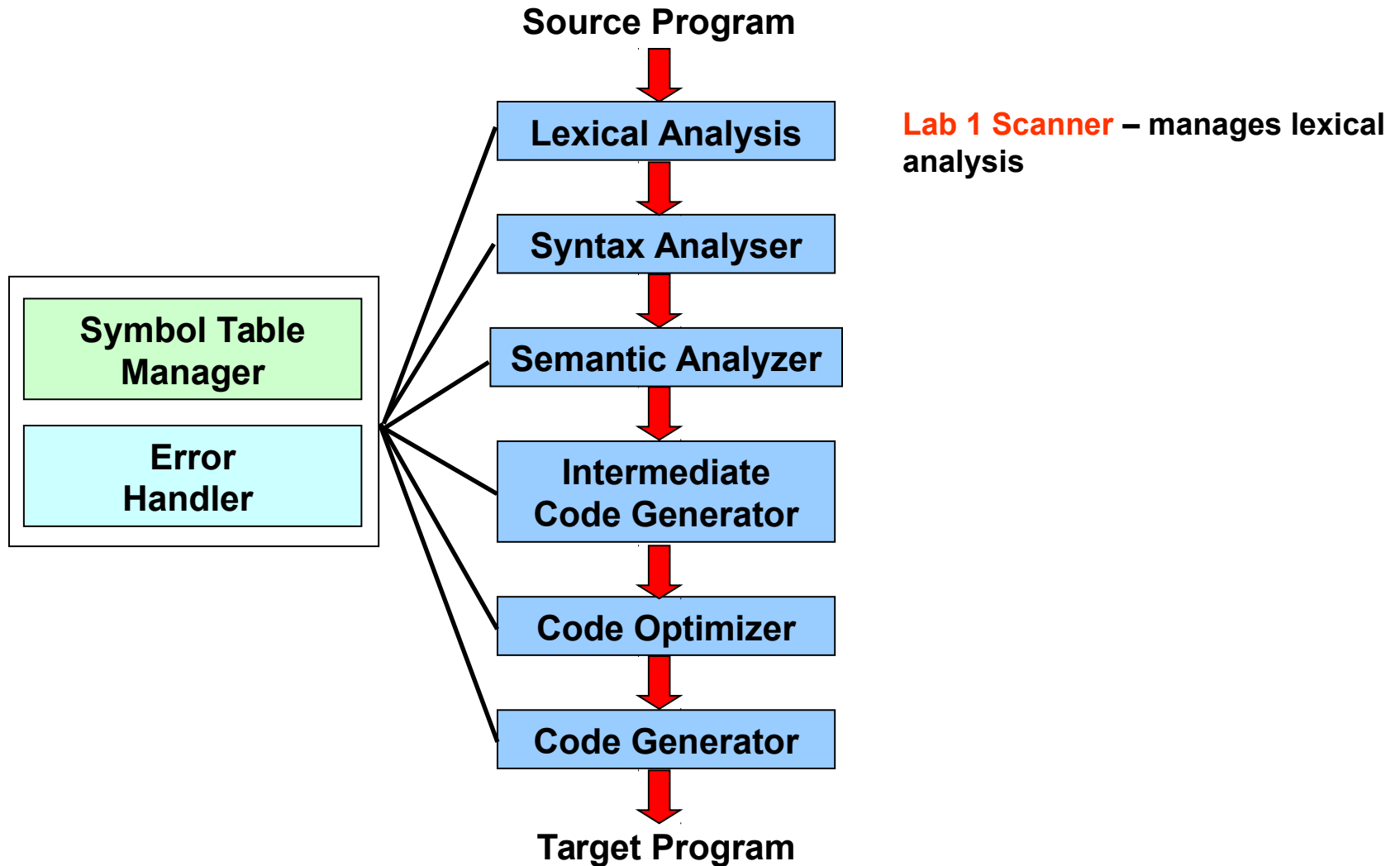
Let's consider this DIESEL program:

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

Declarations

Instructions block

PHASES OF A COMPILER (cont'd)



PHASES OF A COMPILER (SCANNER)

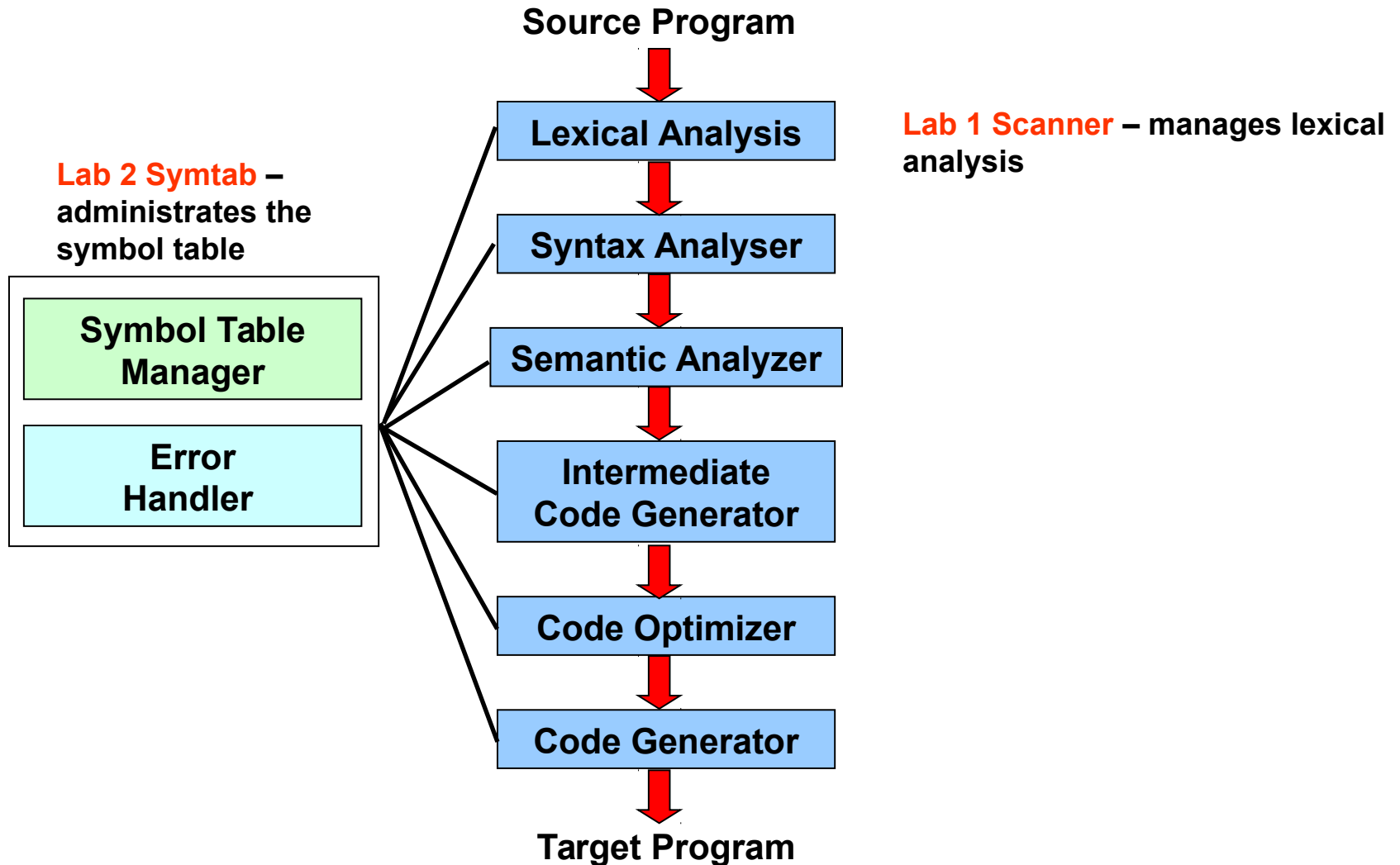
INPUT

```
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		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

PHASES OF A COMPILER (cont'd)



PHASES OF A COMPILER (SYMTAB)

INPUT

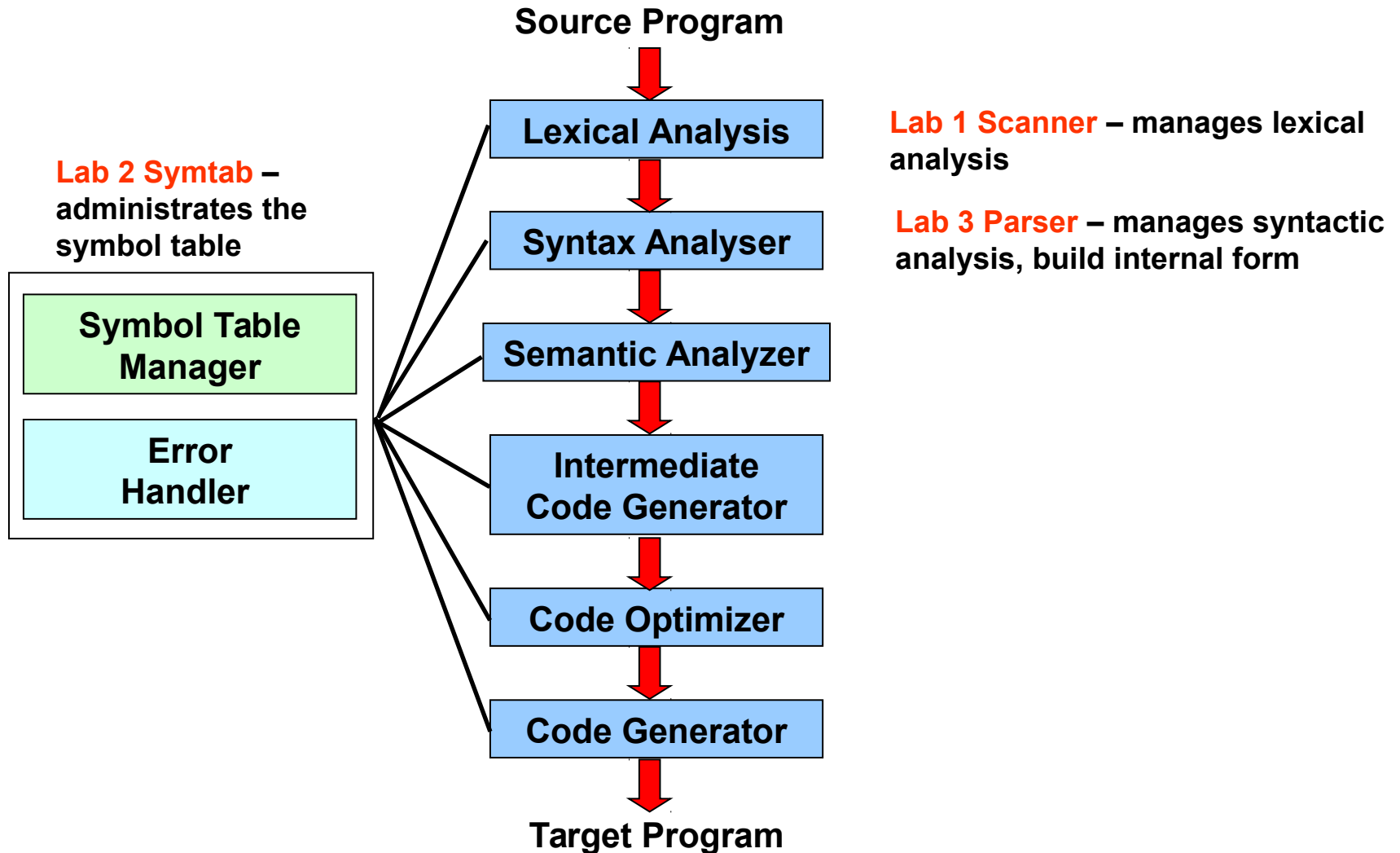
OUTPUT

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



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 (cont'd)



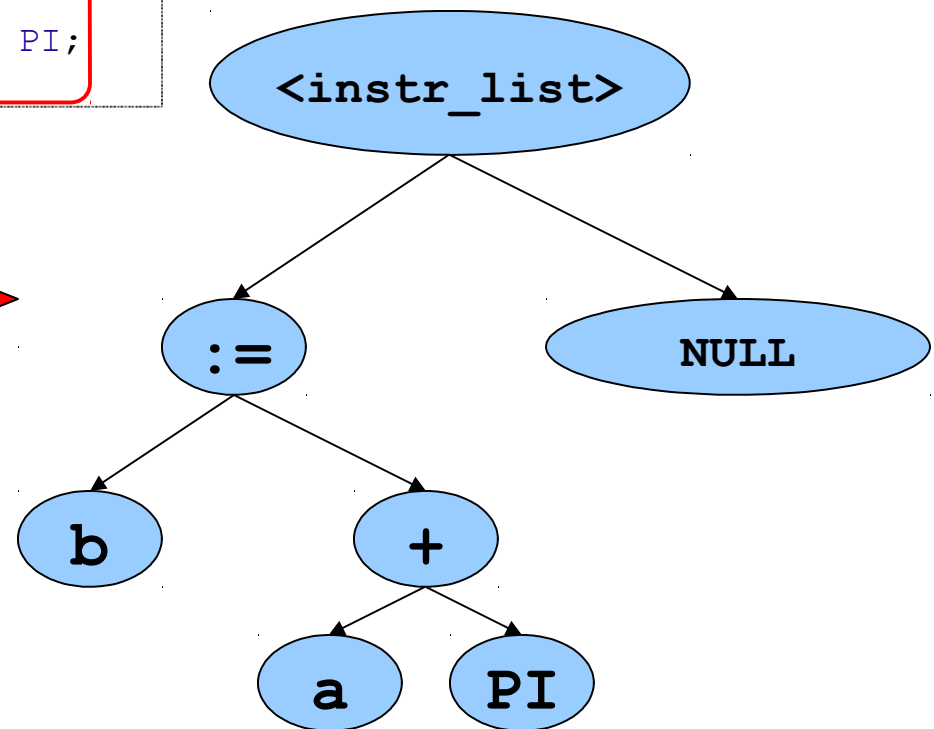
PHASES OF A COMPILER (PARSER)

INPUT

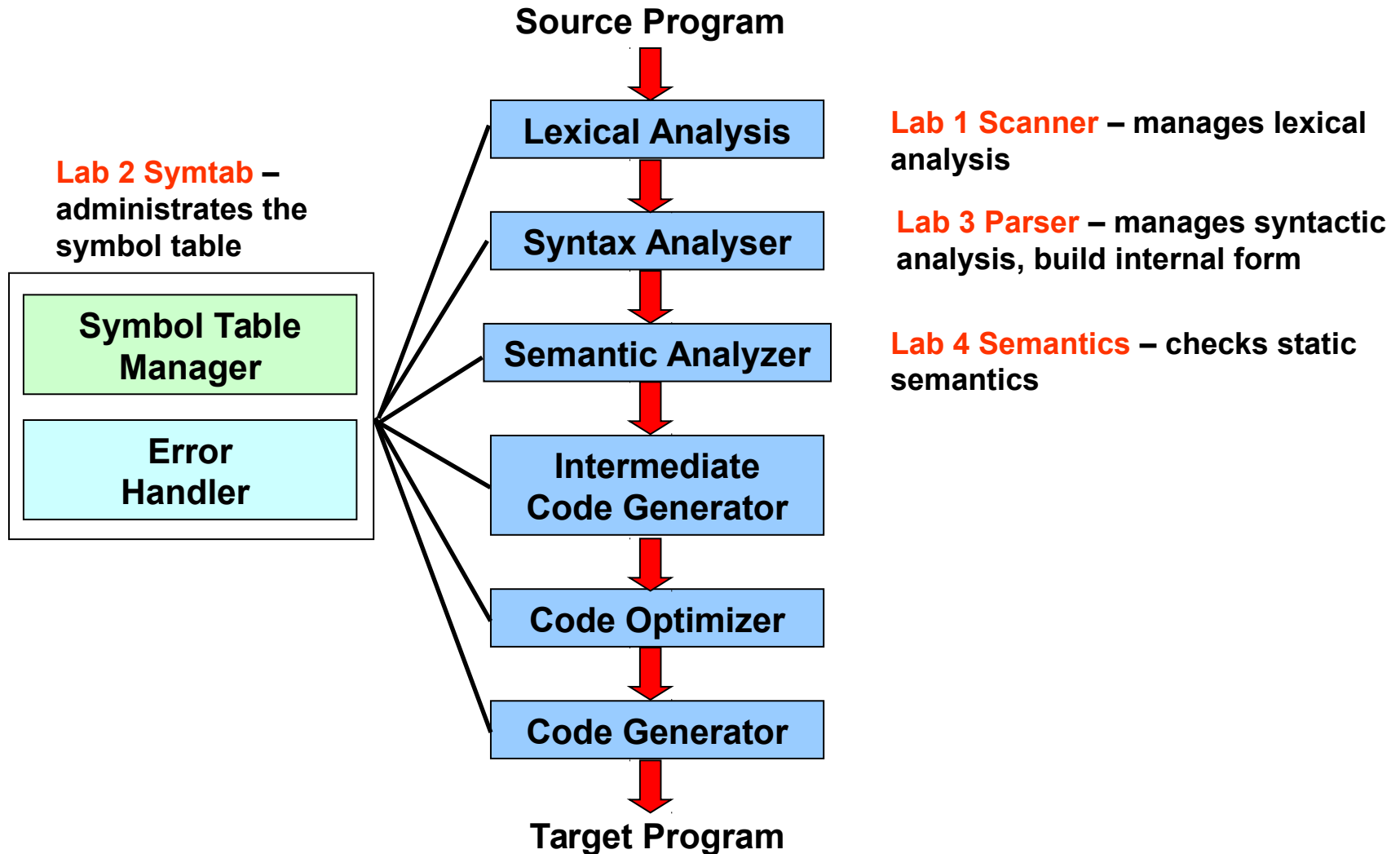
token	pool_p	val	type
T_PROGRAM			keyword
T_IDENT	EXAMPLE		identifier
T_SEMICOLON			separator
T_CONST			keyword
T_IDENT	PI		identifier
T_EQ			operator
T_REALCONST		3.14159	constant
T_SEMICOLON			separator
T_VAR			keyword
T_IDENT	A		identifier
T_COLON			separator
T_IDENT	REAL		identifier
T_SEMICOLON			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 (cont'd)

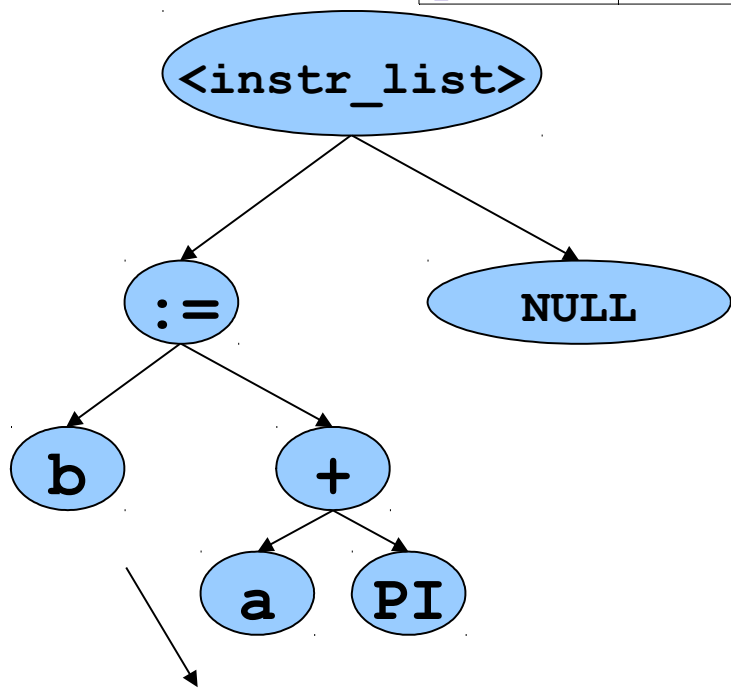


PHASES OF A COMPILER (SEMANTICS)

INPUT

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

OUTPUT

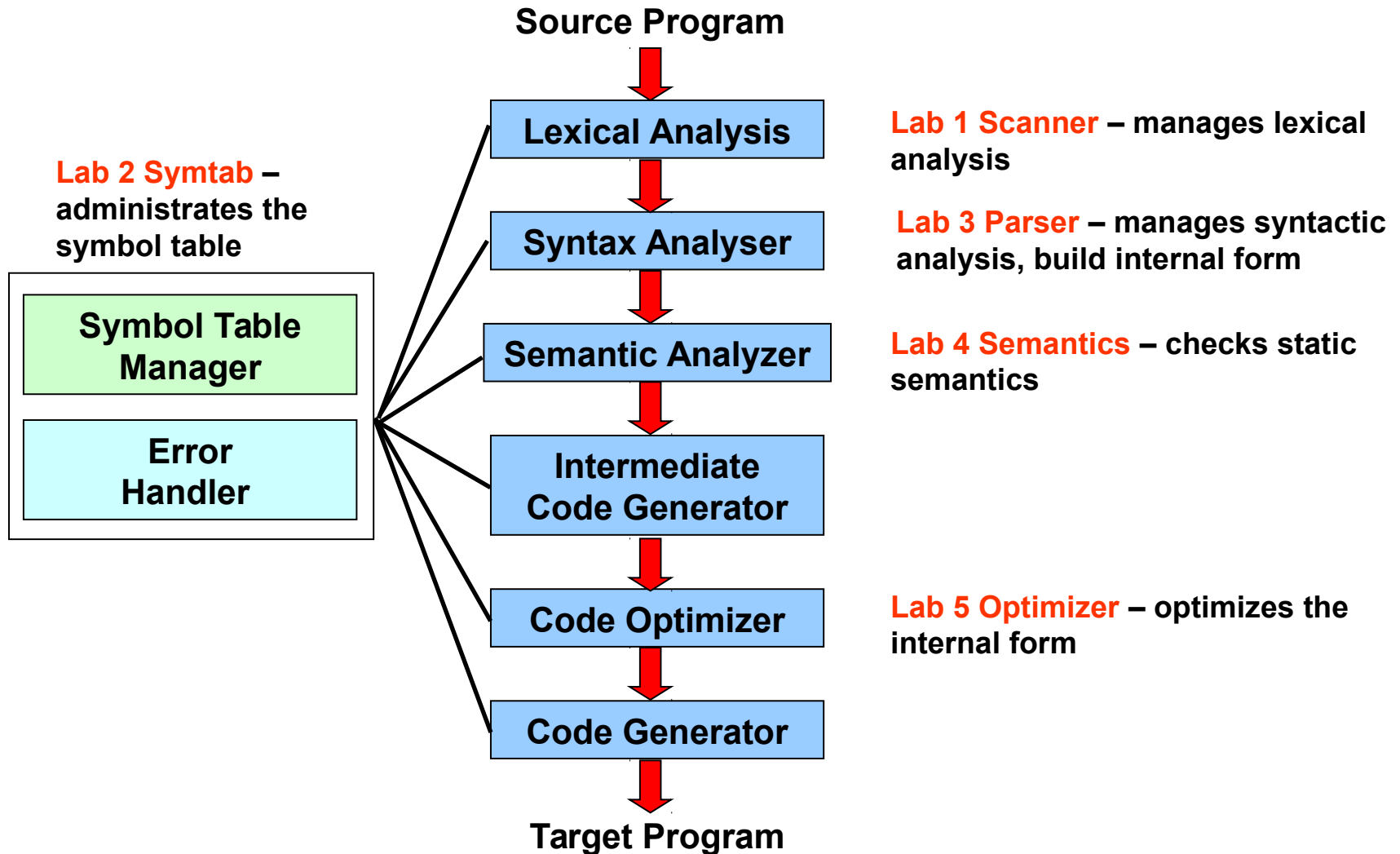


type(a) == type(b) == type(PI) ?



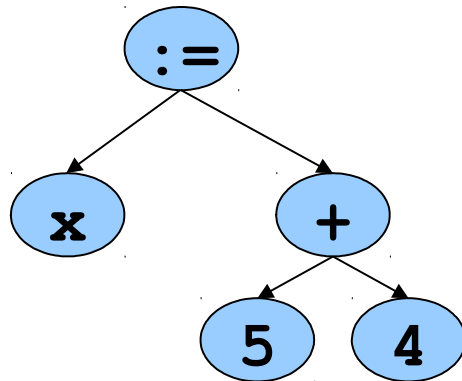
YES

PHASES OF A COMPILER (cont'd)

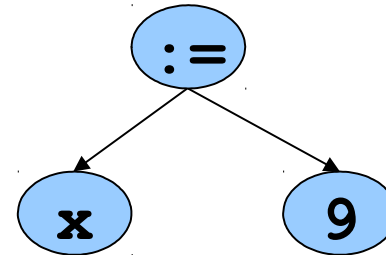


PHASES OF A COMPILER (OPTIMIZER)

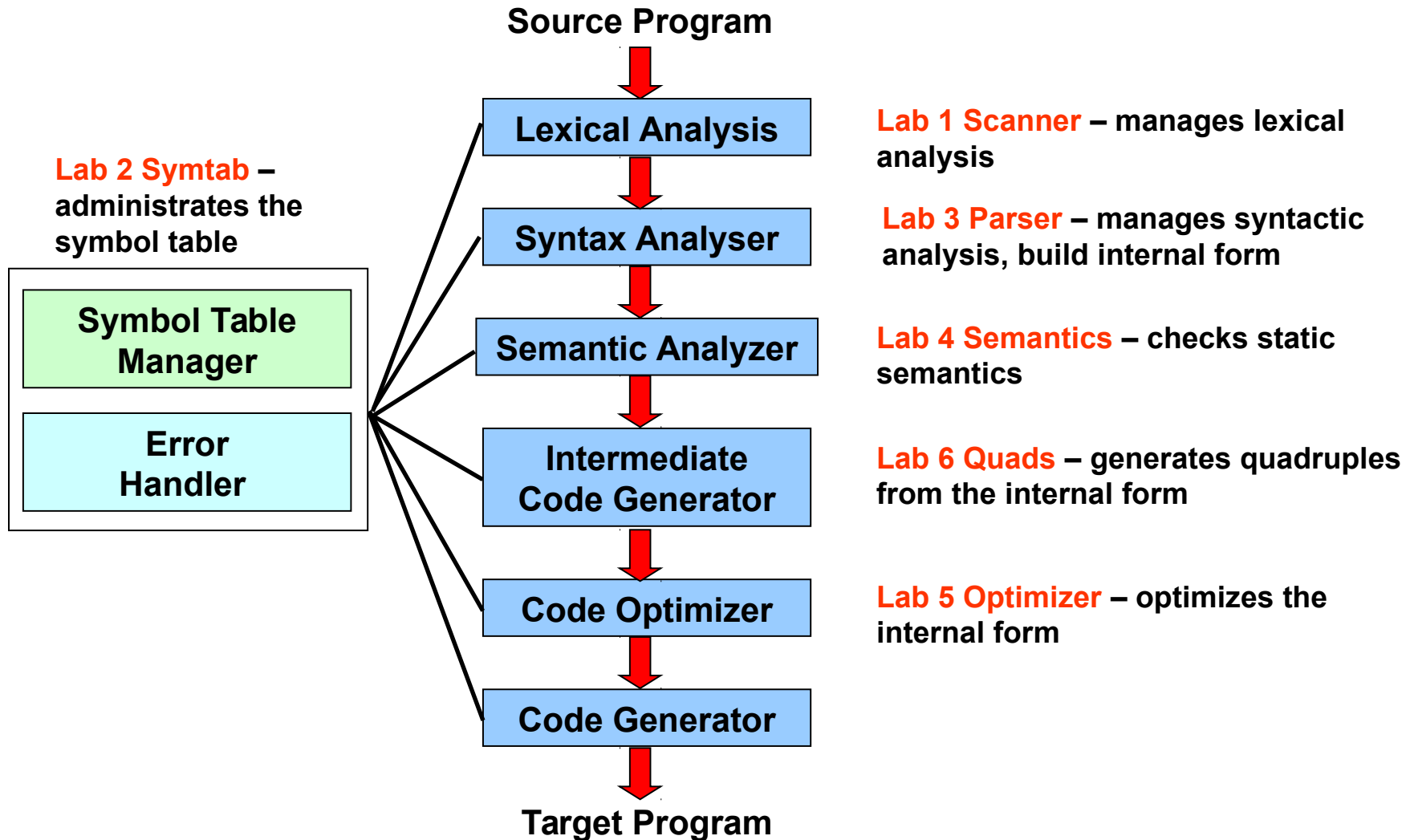
INPUT



OUTPUT



PHASES OF A COMPILER (cont'd)



PHASES OF A COMPILER (QUADS)

INPUT

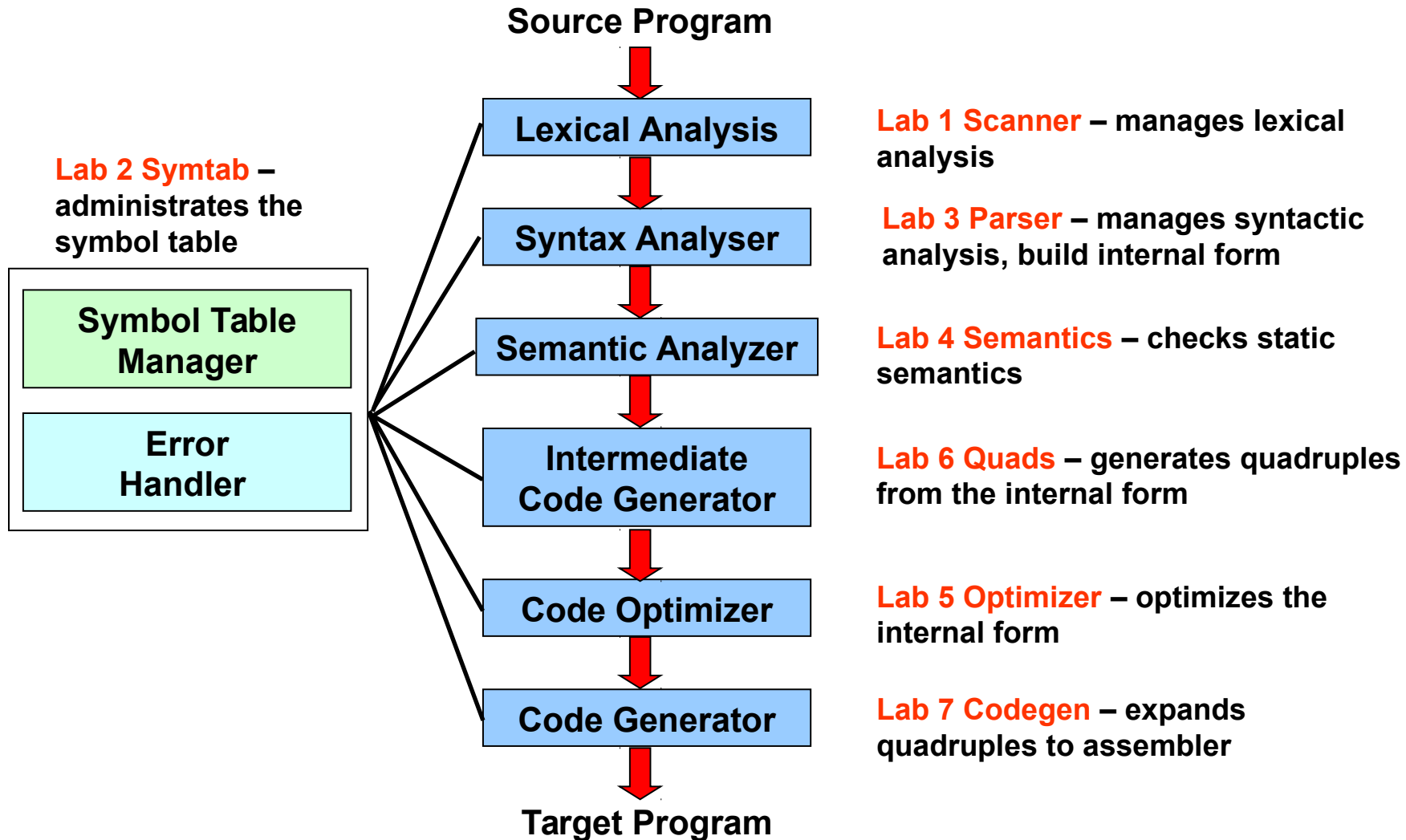
OUTPUT

```
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	-	-

PHASES OF A COMPILER (cont'd)



PHASES OF A COMPILER (CODEGEN)

INPUT

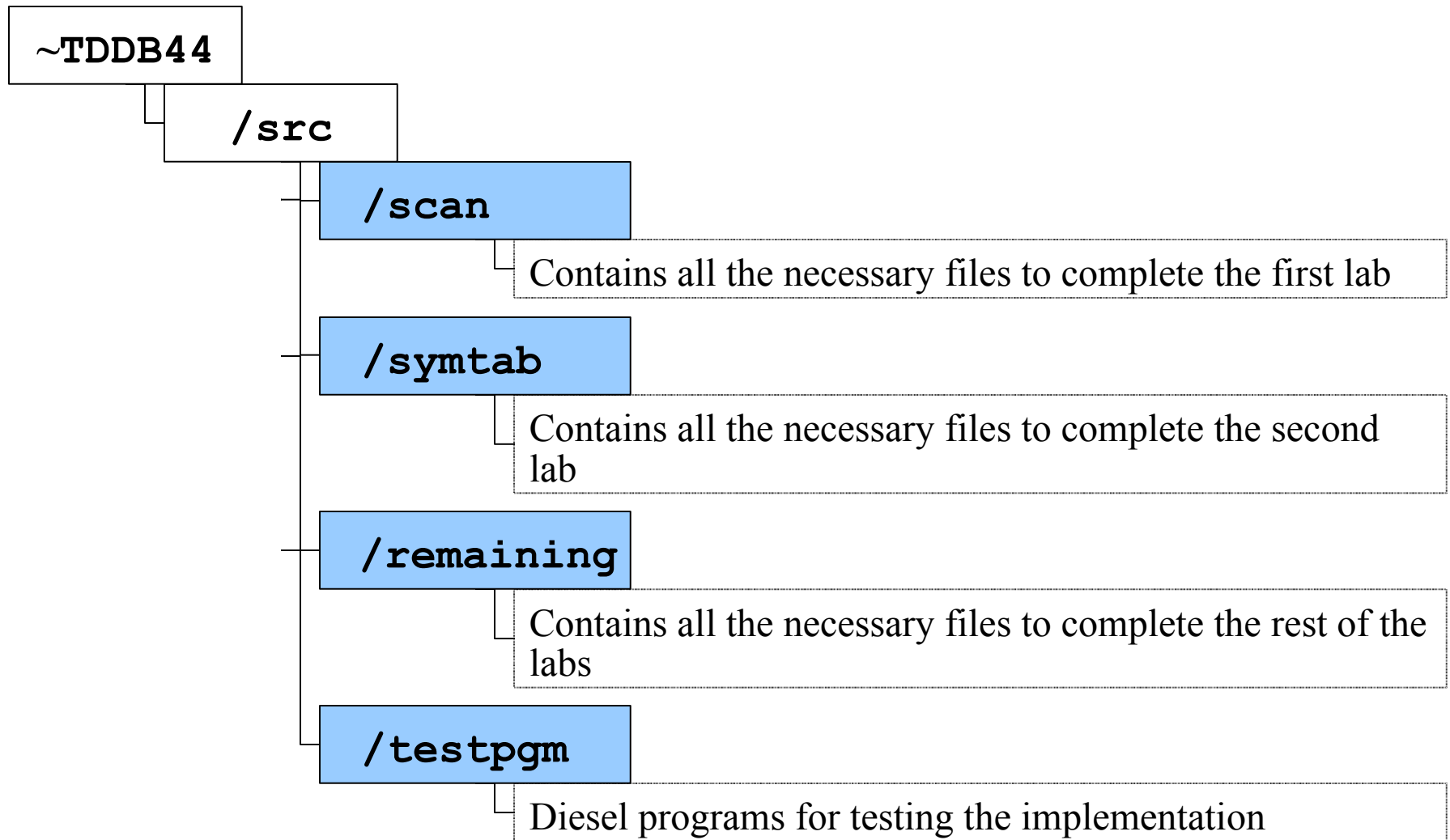
```
program example;  
const  
    PI = 3.14159;  
var  
    a : real;  
    b : real;  
begin  
    b := a + PI;  
end.
```



OUTPUT

```
L3:                                # EXAMPLE  
    push    rbp  
    mov     rcx, rsp  
    push    rcx  
    mov     rbp, rcx  
    sub     rsp, 24  
    mov     rcx, [rbp-8]  
    fld     qword ptr [rcx-16]  
    mov     rcx, 4614256650576692846  
    sub     rsp, 8  
    mov     [rsp], rcx  
    fld     qword ptr [rsp]  
    add     rsp, 8  
    faddp  
    mov     rcx, [rbp-8]  
    fstp    qword ptr [rcx-32]  
    mov     rcx, [rbp-8]  
    mov     rax, [rcx-32]  
    mov     rcx, [rbp-8]  
    mov     [rcx-24], rax  
  
L4:  
    leave  
    ret
```

LAB 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 TDDB44  
cp -r ~TDDB44/src TDDB44
```

- More information in the Lab Compendium

HOW TO COMPILE

- To compile:
 - Execute **make** in the proper source directory
- To run:
 - Call the **diesel** script with the proper flags
 - The Lab Compendium specifies, for each lab, what test programs to run, and what flags to use.
- To test:
 - Execute for example **make lab3** in the proper source directory
 - Running the test target checks that your output matches that of the `/trace` subdirectory

HANDING IN LABS

- Demonstrate the working solutions to your lab assistant during scheduled time. Then send the modified files to the same assistant (put *TDDB44 <Name of the assignment>* in the topic field). One e-mail per group.
- You should get a webreg notification that the source code was received, when it is approved, and if the code needs to be revised
- You should get a webreg notification within 24 hours of an approved demonstration of the lab

DEADLINE

- Deadline for all the assignments is the end of HT2 study period (you will get 3 extra points on the final exam if you finish on time!)
- Note: Check with your lab assistant for handing in solutions after the last scheduled lab

DIESEL EXAMPLE

```
program circle;
const
    PI = 3.14159;
var
    o : real;
    r : real;
procedure init;
begin
    r := 17;
end;
function circumference(radius : real) : real;
    function diameter(radius : real) : real;
    begin
        return 2 * radius;
    end;
begin
    return diameter(radius) * PI;
end;
begin
    init();
    o := circumference(r);
end.
```

LAB 1

THE SCANNER

SCANNING

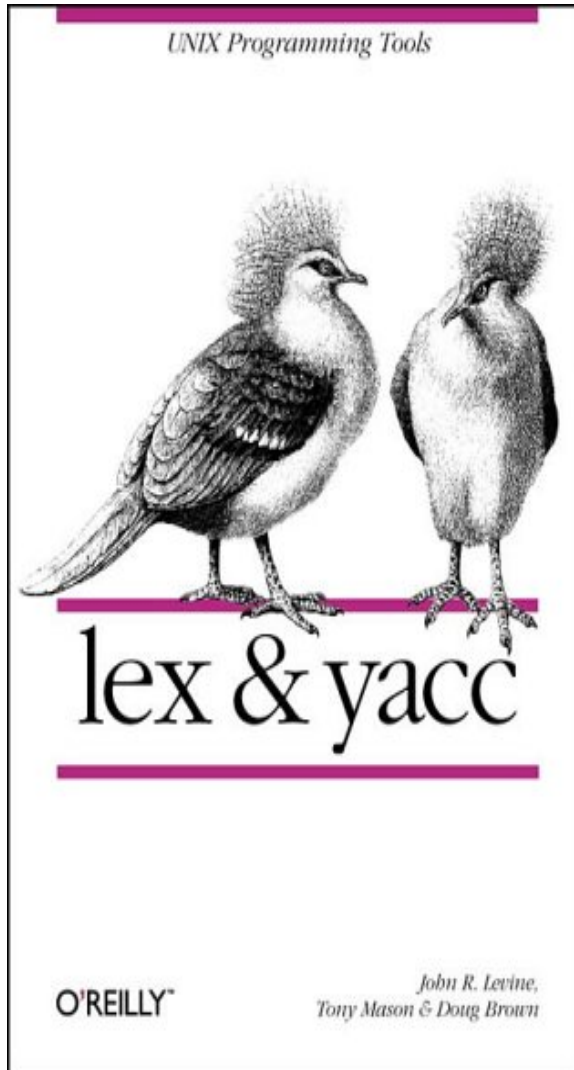
Scanners are programs that recognize lexical patterns in text

- Its **input** is text written in some language
- Its **output** is a sequence of tokens from that text. The tokens are chosen according with the language
- Building a scanner manually is hard
- We know that the mapping from regular expressions to FSM is straightforward, so why not we **automate the process**?
- Then we just have to type in regular expressions and get the code to implement a scanner back

SCANNER GENERATORS

- Automate is exactly what **flex** does!
- **flex** is a fast lexical analyzer generator, a tool for generating programs that perform pattern matching on text
- **flex** is a free implementation (started 1987) of the well-known **lex** program (~1977)

MORE ON LEX/BISON



If you'll use flex/bison in the future...

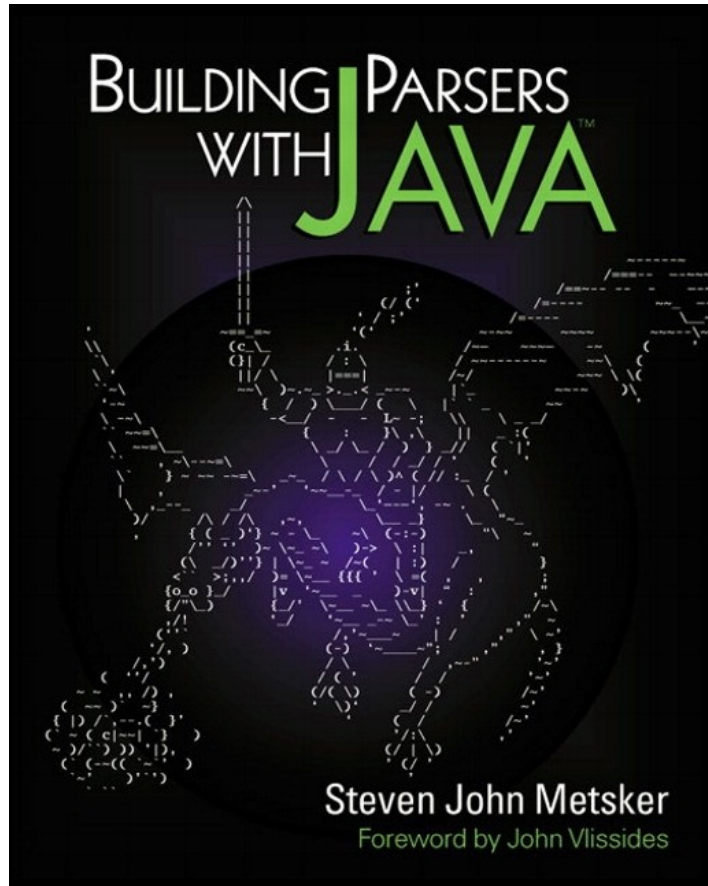
Lex & Yacc, 2nd ed

By, John R Levine, Tony Mason
& Doug Brown

O'Reilly & Associates

ISBN: 1565920007

MORE REFERENCES

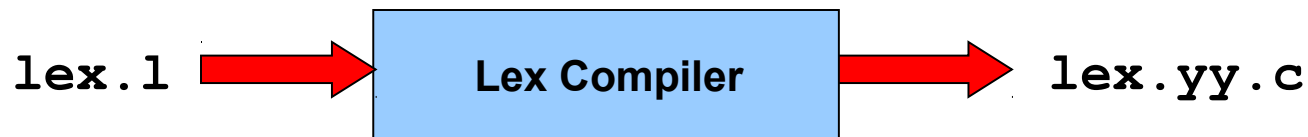


For those who would like to learn more about parsers by using Java...

HOW IT WORKS

flex generates at output a **C** source file **lex.yy.c** which defines a routine **yylex()**

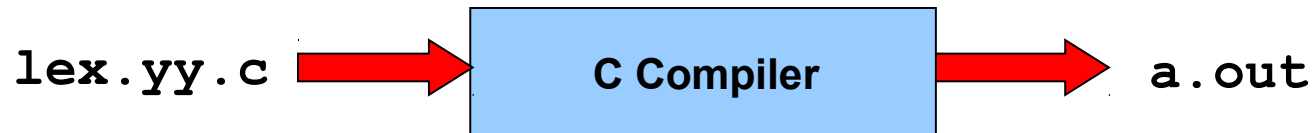
```
>> flex lex.1
```



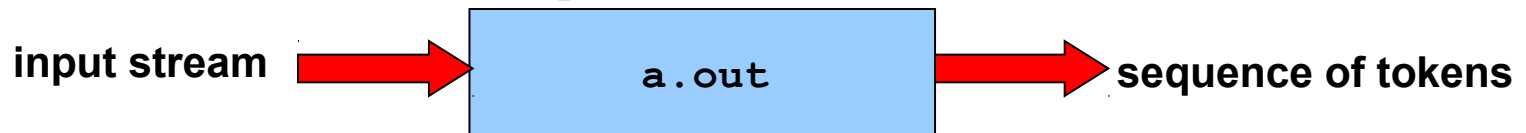
HOW IT WORKS

`lex.yy.c` is compiled and linked with the `-lfl` library to produce an executable, which is the scanner

```
>> g++ lex.yy.c -lfl
```



```
>> a.out < input.txt
```



FLEX SPECIFICATIONS

Lex programs are divided into three components

```
/* Definitions - name definitions
 *               - variables defined
 *               - include files specified
 *               - etc
 */

%%

/* Translation rules - pattern actions {C/C++statements} */

%%

/* User code - supports routines for the above C/C++
 *               statements
 */
```

NAME DEFINITIONS

- Name definition are intended to simplify the scanner specification and have the form:

```
name      definition
```

- Subsequently the definition can be referred to by {name}, which then will expand to the definition.
- Example:

```
DIGIT      [0-9]  
{DIGIT}+"."{DIGIT}*
```

is identical/will be expanded to:

```
([0-9])+"."([0-9])*
```

PATTERN ACTIONS

- The *transformation rules* section of the **lex/flex** input, contains a series of rules of the form:

pattern	action
----------------	---------------

- Example:

[0-9]*	{ printf ("%s is a number", yytext); }
---------------	--

SIMPLE PATTERNS

Match only one specific character

- ✗ The character '**x**'
- Any character except newline

CHARACTER CLASS PATTERNS

Match any character within the class

[xyz] The pattern matches either 'x', 'y', or 'z'

[abj-o] This pattern spans over a range of characters and matches 'a', 'b', or any letter ranging from 'j' to 'o'

NEGATED PATTERNS

Match any character not in the class

[^z] This pattern matches any character
EXCEPT 'z'

[^A-Z] This pattern matches any character
EXCEPT an uppercase letter

[^A-Z\n] This pattern matches any character
EXCEPT an uppercase letter or a
newline

SOME USEFUL PATTERNS

- r*** Zero or more 'r', 'r' is any regular expr.
- \\0** NULL character (ASCII code 0)
- \\123** Character with octal value 123
- \\x2a** Character with hexadecimal value 2a
- p|s** Either 'p' or 's'
- p/s** 'p' but only if it is followed by an 's', which is not part of the matched text
- ^p** 'p' at the beginning of a line
- p\$** 'p' at the end of a line, equivalent to 'p/\\n'

FLEX USER CODE

Finally, the user code section is simply copied to `lex.yy.c` verbatim

It is used for companion routines which call, or are called by the scanner

The presence of this user code is optional, if you don't have it there's no need for the second `%%`

FLEX PROGRAM VARIABLES

- yytext** Whenever the scanner matches a token, the text of the token is stored in the null terminated string **yytext**
- yylen** The length of the string **yytext**
- yylex()** The scanner created by the Lex has the entry point **yylex()**, which can be called to start or resume scanning. If **lex** action returns a value to a program, the next call to **yylex()** will continue from the point of that return

A SIMPLE FLEX PROGRAM

Recognition of verbs

*. |

Mary has a
little lamb

```
%{
/* includes and defines should be stated in this section */
}%

%%

[\\t]+          /* ignore white space */

do|does|did|done|has { printf ("%s: is a verb\\n", yytext); }
[a-zA-Z]+       { printf ("%s: is not a verb\\n",yytext); }
.|\n           { ECHO; /* normal default anyway */ }

%%

main()          { yylex(); }
```

A SIMPLE FLEX PROGRAM

A scanner that counts the number of characters and lines in its input

```
int num_lines = 0, num_chars = 0; /* Variables */

%%

\n { ++num_lines; ++num_chars; } /* Take care of newline */
.  { ++num_chars; }             /* Take care of everything else */

%%

main() { yylex();
        printf("lines: %d, chars: %d\n", num_lines, num_chars );
    }
```

The output is the result

A PASCAL SCANNER

```
%{  
    #include <math.h>  
%}  
  
DIGIT      [0-9]  
ID          [a-z][a-z0-9]*  
  
%%  
  
{DIGIT}+   { printf("An integer: %s (%d)\n", yytext, atoi( yytext ));  
            }  
  
{DIGIT}+"."{DIGIT}*  
            { printf("A float: %s (%g)\n", yytext, atof( yytext )); }  
  
if|then|begin|end|procedure|function  
    { printf("A keyword: %s\n", yytext); }  
  
{ID}       { printf("An identifier: %s\n", yytext); }
```


A PASCAL SCANNER

```
"+" | "-" | "*" | "/"      { printf("An operator: %s\n", yytext); }

"{" [^\{ $\; $} }\n] *"}" /* eat up one-line comments */

[ \t\n ]+                  /* eat up whitespace */

.                            { printf("Unknown character: %s\n", yytext ); }

%%

main(argc, argv) {
    ++argv, --argc; /* skip over program name */
    if ( argc > 0 ) yyin = fopen( argv[0], "r" );
    else yyin = stdin;
    yylex();
}
```

FILES OF INTEREST

- Files you will need to modify:
 - `scanner.l` : is the `flex` input file, which you're going to complete. This is the only file you will need to edit in this lab.
- Other files of interest
 - `scanner.hh` : is a temporary include file used for scanner testing.
 - `scantest.cc` : is an interactive test program for your scanner.
 - `symtab.hh` : contains symbol table information, including string pool methods.
 - `symbol.cc` : contains symbol implementations (will be edited in lab 2).
 - `symtab.cc` : contains the symbol table implementation.
 - `error.hh` and `error.cc` contain debug and error message routines.

LAB2

THE SYMBOL

TABLE

SYMBOL TABLES

A Symbol table contains all the information that must be passed between different phases of a compiler/interpreter

A **symbol** (or token) has at least the following **attributes**:

- Symbol **Name**
- Symbol **Type** (int, real, char,)
- Symbol **Class** (static, automatic, cons...)

SYMBOL TABLES

In a compiler we also need:

- **Address** (where is the information stored?)
- Other information due to used data structures

Symbol tables are typically implemented using hashing schemes because good efficiency for the lookup is needed

SYMBOL TABLES

The symbol table primarily helps ...

... in checking the program's semantic correctness
(type checking, etc.)

... in generating code (keep track of memory
requirements for various variables, etc.)

SIMPLE SYMBOL TABLES

We classify symbol tables as:

- Simple
- Scoped

Simple symbol tables have...

... only one scope

... only “global” variables

Simple symbol tables may be found in BASIC and FORTRAN compilers

SCOPED SYMBOL TABLES

Complication in simple tables involves languages that permit multiple scopes

C permits at the simplest level two scopes: global and local (it is also possible to have nested scopes in C)

WHY SCOPES?

The importance of considering the scopes are shown in these two C programs

```
int a=10; //global variable

main() {
    changeA();
    printf("Value of a=%d\n,a);
}

void changeA(){
    int a; //local variable
    a=5;
}
```

```
int a=10; //global variable

main() {
    changeA();
    printf("Value of a=%d\n,a);
}

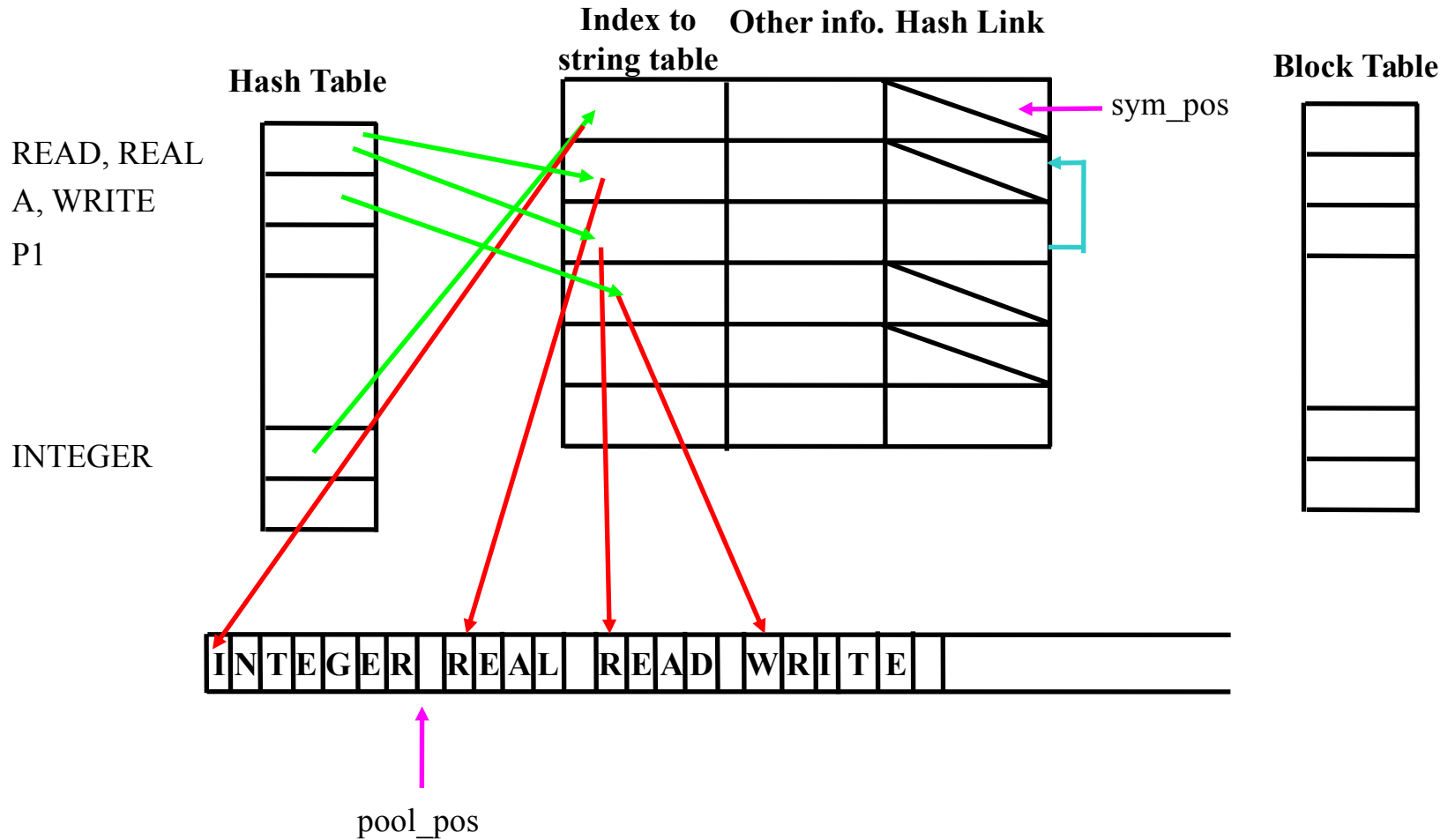
void changeA(){
    a=5;
}
```

SCOPED SYMBOL TABLES

Operations that must be supported by the symbol table in order to handle scoping:

- **Lookup in any scope** – search the most recently created scope first
- **Enter a new symbol** in the symbol table
- **Modify** information about a symbol in a “visible” scope
- **Create** a new scope
- **Delete** the most recently scope

HOW IT WORKS



A SMALL PROGRAM

```
program prog;  
  var  
    a : integer;  
    b : integer;  
    c : integer;  
  
  procedure p1;  
    var  
      a : integer;  
    begin  
      c := b + a;  
    end;  
  
  begin  
    c := b + a;  
  end.
```

YOUR TASK

- Implement the methods *open_scope()* and *close_scope()*, called when entering and leaving an environment.
- Implement the method *lookup_symbol()*, it should search for a symbol in open environments.
- Implement the method *install_symbol()*, it should install a symbol in the symbol table.
- Implement the method *enter_procedure()*.

FILES OF INTEREST

- Files you will need to modify
 - **syntab.cc** : contains the symbol table implementation.
 - **scanner.l** : minor changes.

- Other files of interest

(Other than the Makefile, use the same files you were given in the first lab.)

- **syntab.hh** : contains all definitions concerning symbols and the symbol table.
- **symbol.cc** : contains the symbol class implementations.
- **error.hh** and **error.cc** : contain debug and error message routine
- **syntabtest.cc** : used for testing. Edit this file to simulate various calls to the symbol table.
- **Makefile** : not the same as in the first lab!

DEBUGGING

- All symbols can be sent directly to cout. The entire symbol table can be printed using the *print()* method with various arguments.