COMPILER CONSTRUCTION Lesson 1 – TDDB44

November, 6 2013

Kristian Stavåker (kristian.stavaker@liu.se)

Sergiu Rafiliu (sergiu.rafiliu@liu.se)

Department of Computer and Information Science

Linköping University

PURPOSE OF LESSONS

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

You can buy the laboratory compendium as well as a small compendium of exercises (suitable as a revision for the exam) in the **student book store in Kårallen**. Also the compendium from 2012 is fine (just minor formatting differences).

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

LABORATORY ASSIGNMENTS

In the laboratory assignments, 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. You will also (most likely) have to work during non-scheduled time.

HANDING IN AND DEADLINE

- 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.

- Deadline for all the assignments is: **December 20, 2013** (you will get 3 extra points on the final exam if you finish on time)

- Remember to register yourself in the webreg system, www.ida.liu.se/webreg

RELATING LABS TO THE COURSE

- Building a complete compiler
  - We use a language (Diesel) that is small enough to be manageable.
  - Scanning, Parsing, Semantic Elaboration, Code Generation, etc.
  - Experience in compiler construction and software engineering.
  - Compiler mostly written in C++.

LABORATORY EXERCISES

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

Advantages

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

Disadvantages

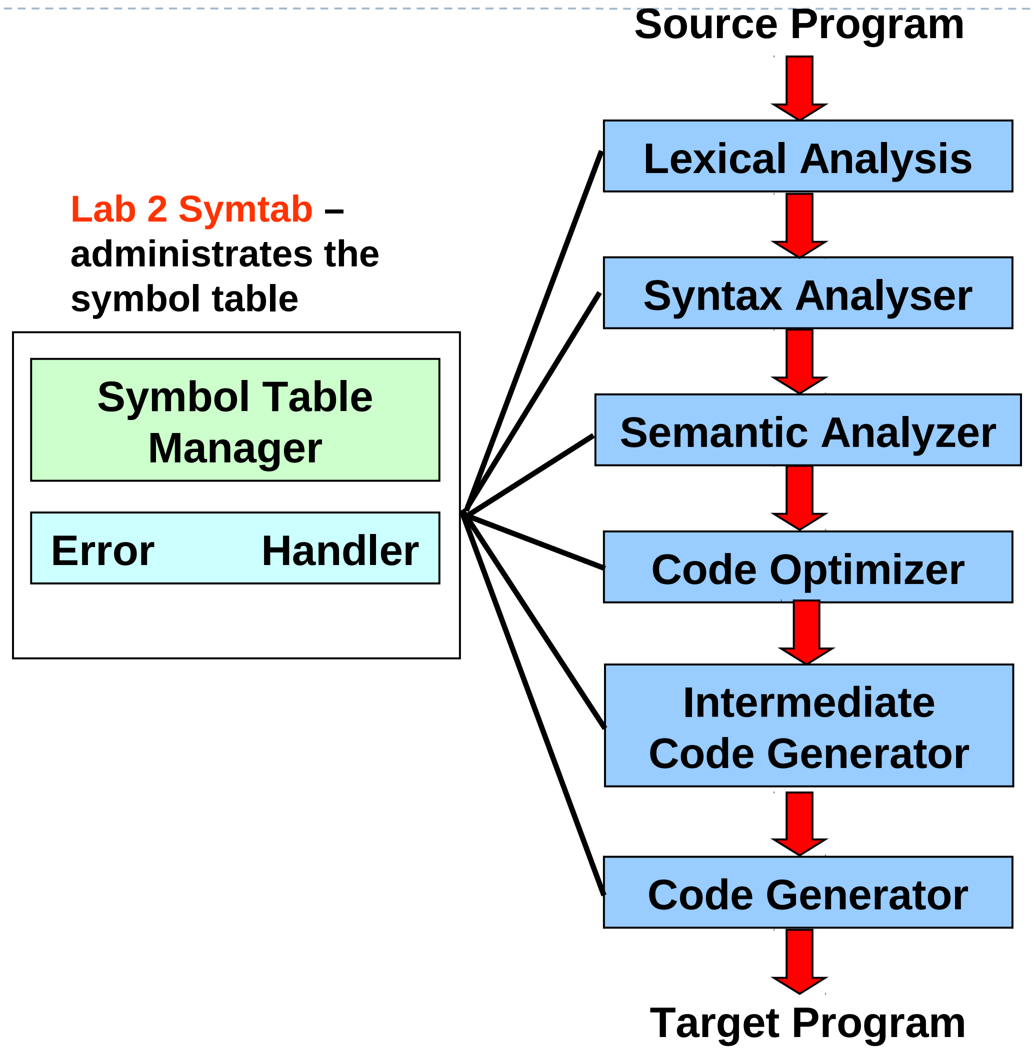
- High ratio of programming to theory
- Cumulative nature magnifies early failures
- Many parts are simplified

LABORATORY ASSIGNMENTS

(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 (quadruples)
Lab 7	Code generation (assembly) and memory management

PHASES OF A COMPILER

Lab 2 Symtab – administrates the symbol table



Lab 1 Scanner – manages lexical analysis

Lab 3 Parser – manages syntactic analysis, build internal form

Lab 4 Semantics – checks static semantics

Lab 5 Optimizer – optimizes the internal form

Lab 6 Quads – generates quadruples from the internal form

Lab 7 Codegen – expands quadruples into assembly

PHASES OF A COMPILER (continued)

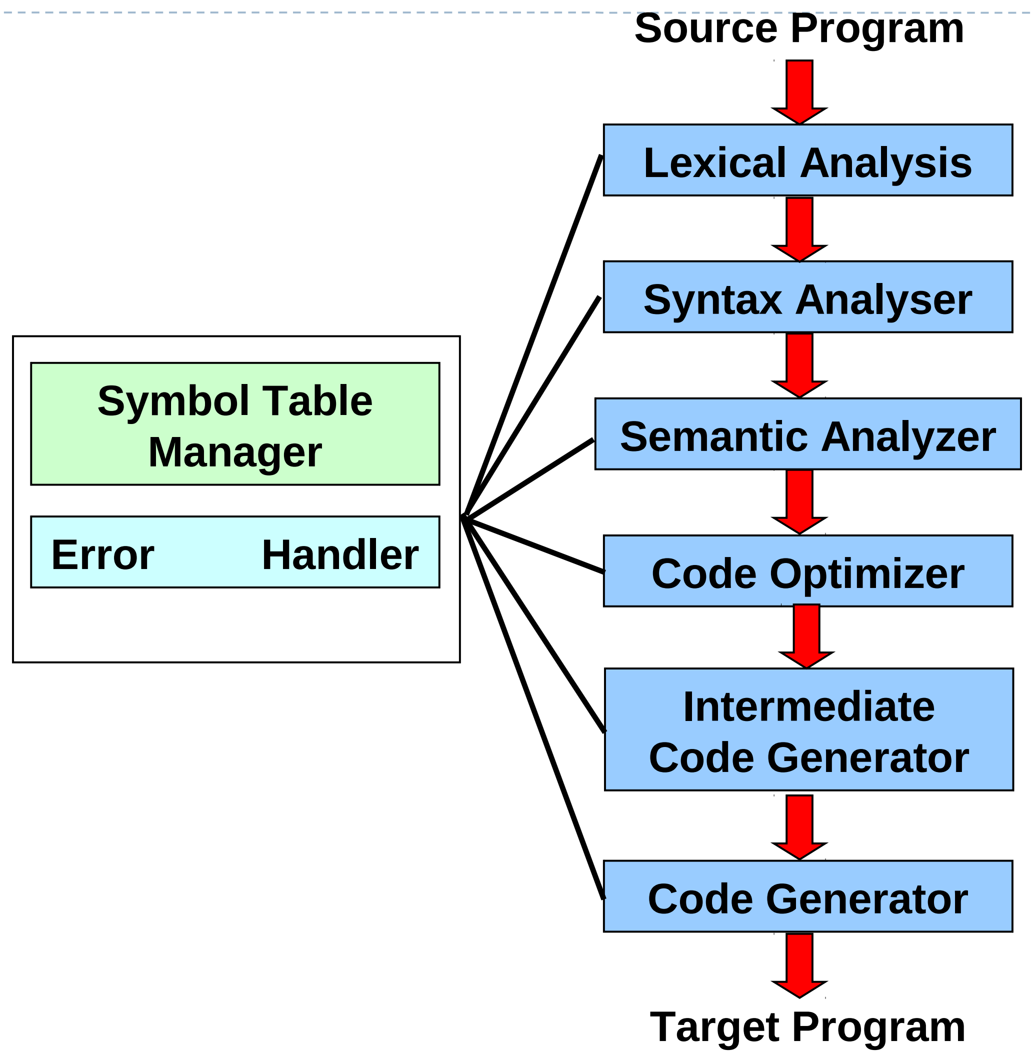
Let's consider this DIESEL program:

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

Declarations

Instruction block

PHASES OF A COMPILER



Lab 1 Scanner – manages lexical analysis

PHASES OF A COMPILER (SCANNER)

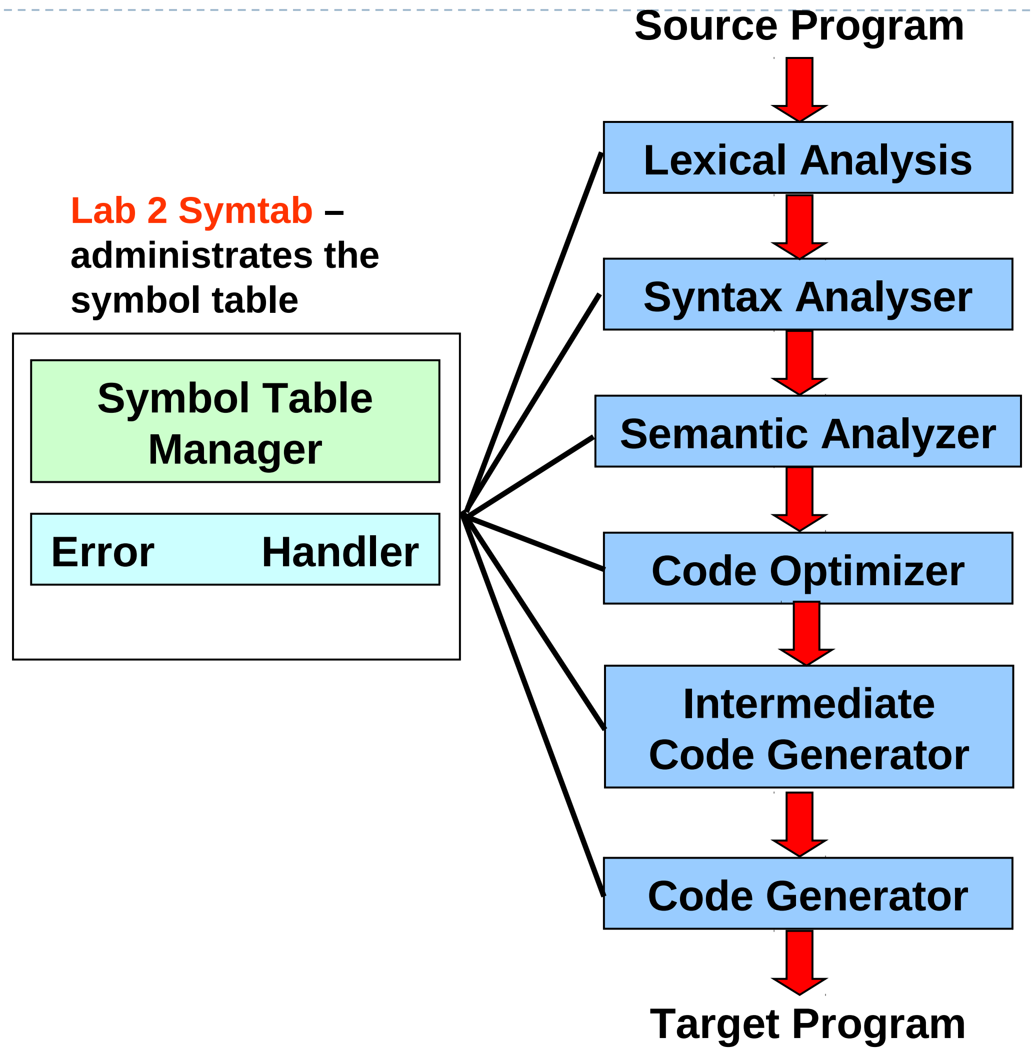
INPUT

```
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		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



Lab 1 Scanner – manages lexical analysis

Lab 2 Symtab – administrates the symbol table

PHASES OF A COMPILER (SYMTAB)

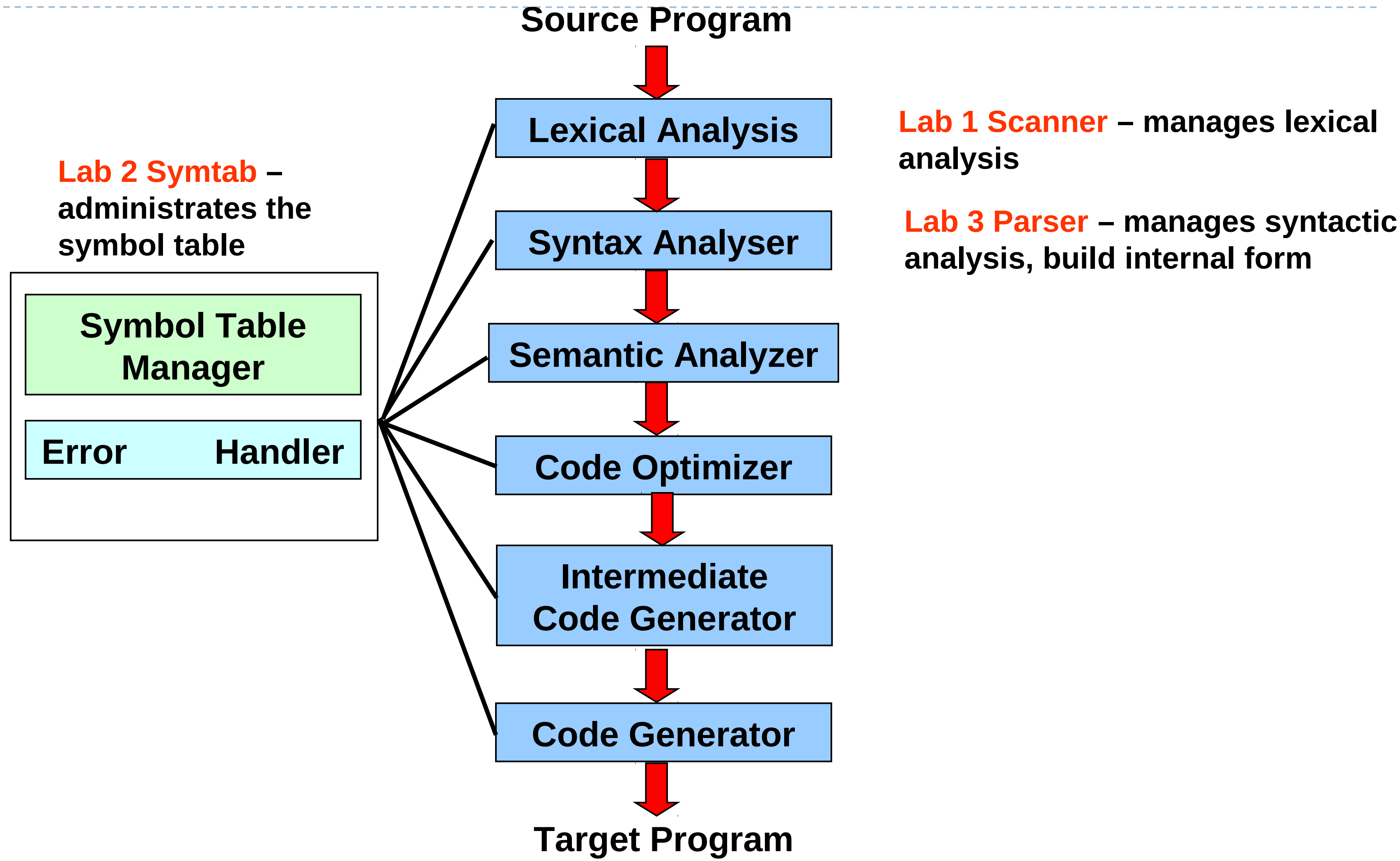
INPUT

```
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



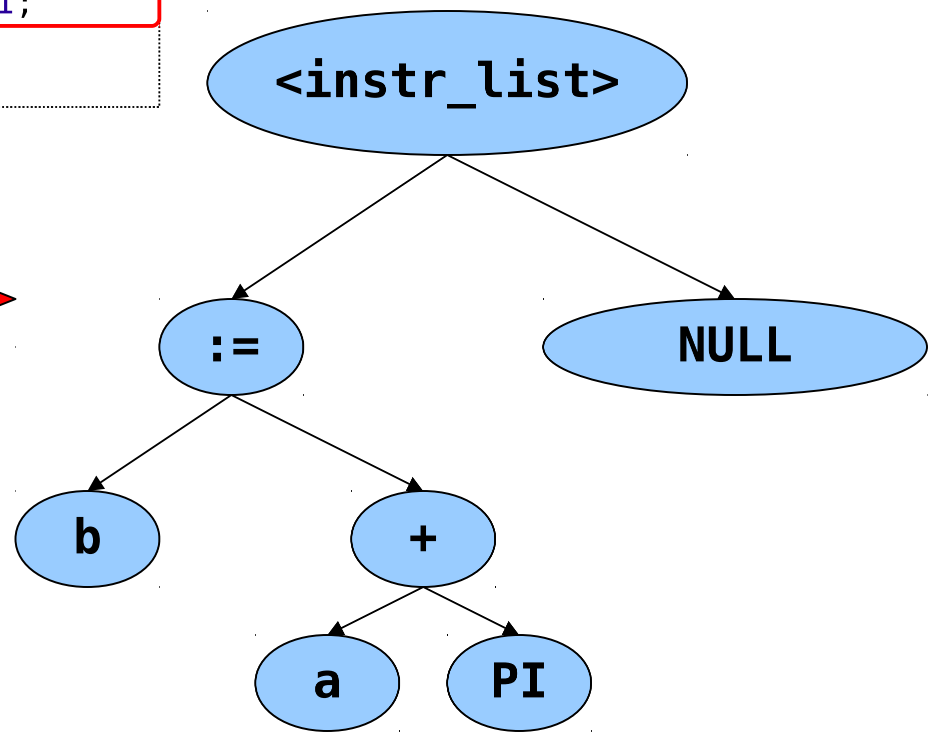
PHASES OF A COMPILER (PARSER)

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

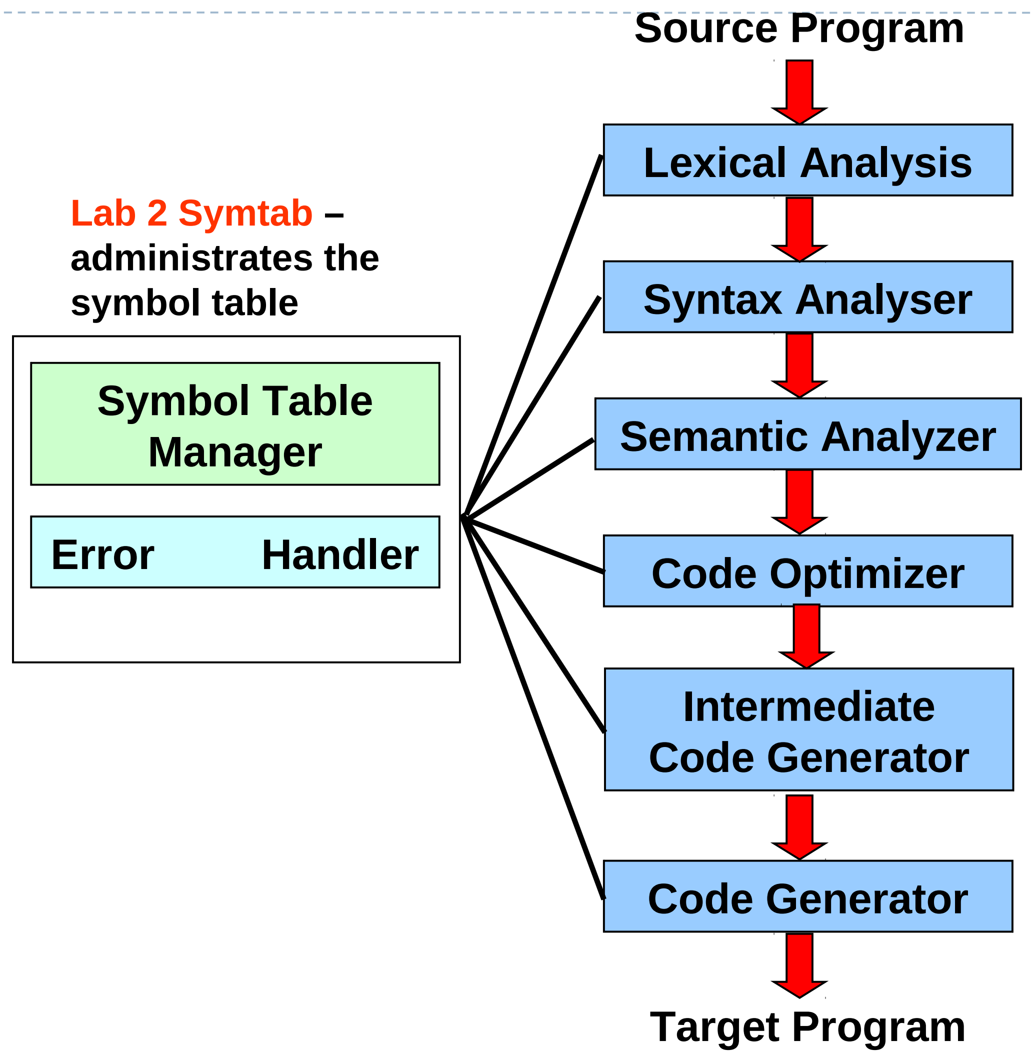
```
program example;
const
   PI = 3.14159;
var
   a : real;
   b : real;
begin
   b := a + PI;
end.
```

OUTPUT



PHASES OF A COMPILER

Lab 2 Symtab – administrates the symbol table



Lab 1 Scanner – manages lexical analysis

Lab 3 Parser – manages syntactic analysis, build internal form

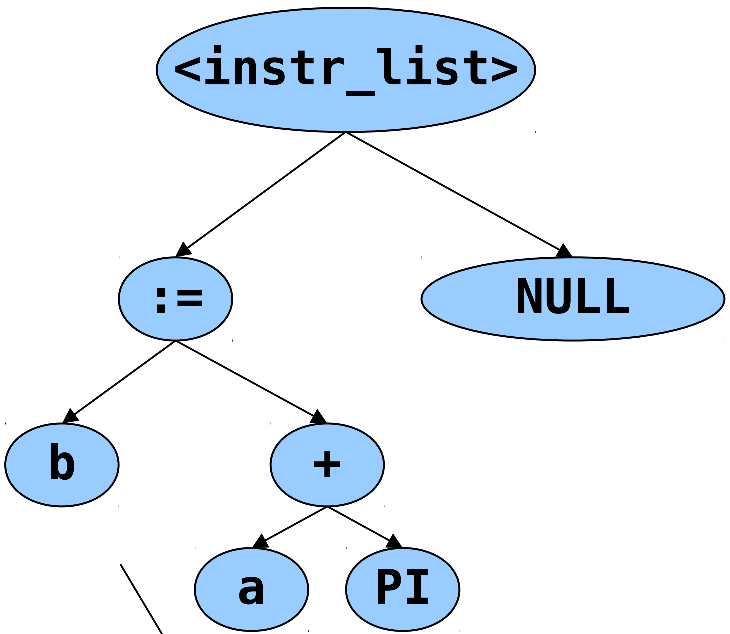
Lab 4 Semantics – checks static semantics

PHASES OF A COMPILER (SEMANTICS)

INPUT

OUTPUT

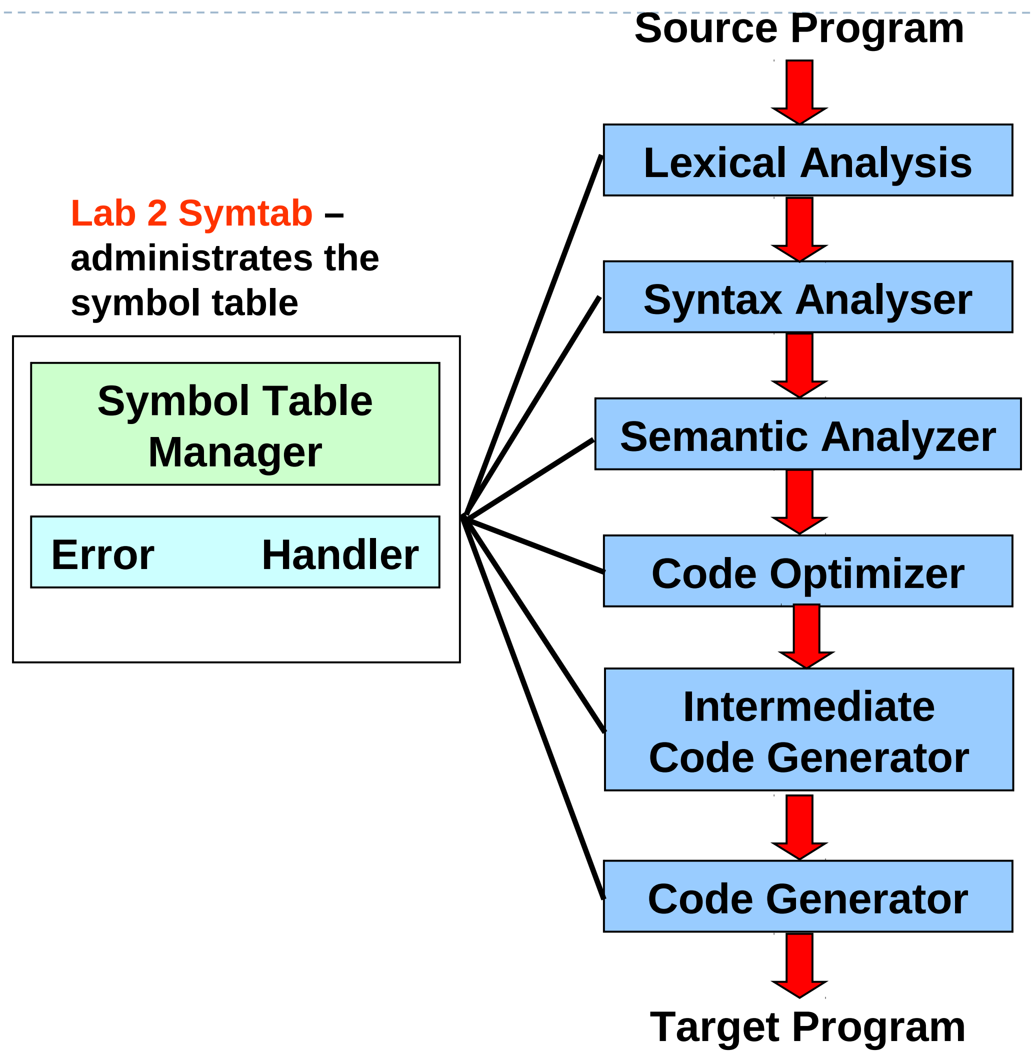
token	pool_p	val	type
T_IDENT	VOID		
T_IDENT	INTEGER		
T_IDENT	REAL		
T_IDENT	EXAMPLE		
T_IDENT	PI	3.14159	REAL
T_IDENT	A		REAL
T_IDENT	B		REAL



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

YES

PHASES OF A COMPILER



Lab 2 Symtab – administrates the symbol table

Lab 1 Scanner – manages lexical analysis

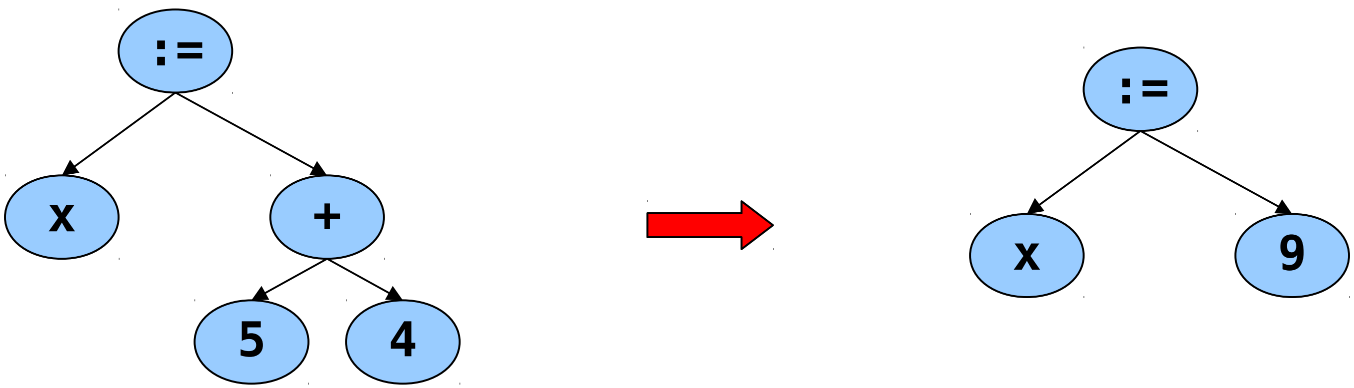
Lab 3 Parser – manages syntactic analysis, build internal form

Lab 4 Semantics – checks static semantics

Lab 5 Optimizer – optimizes the internal form

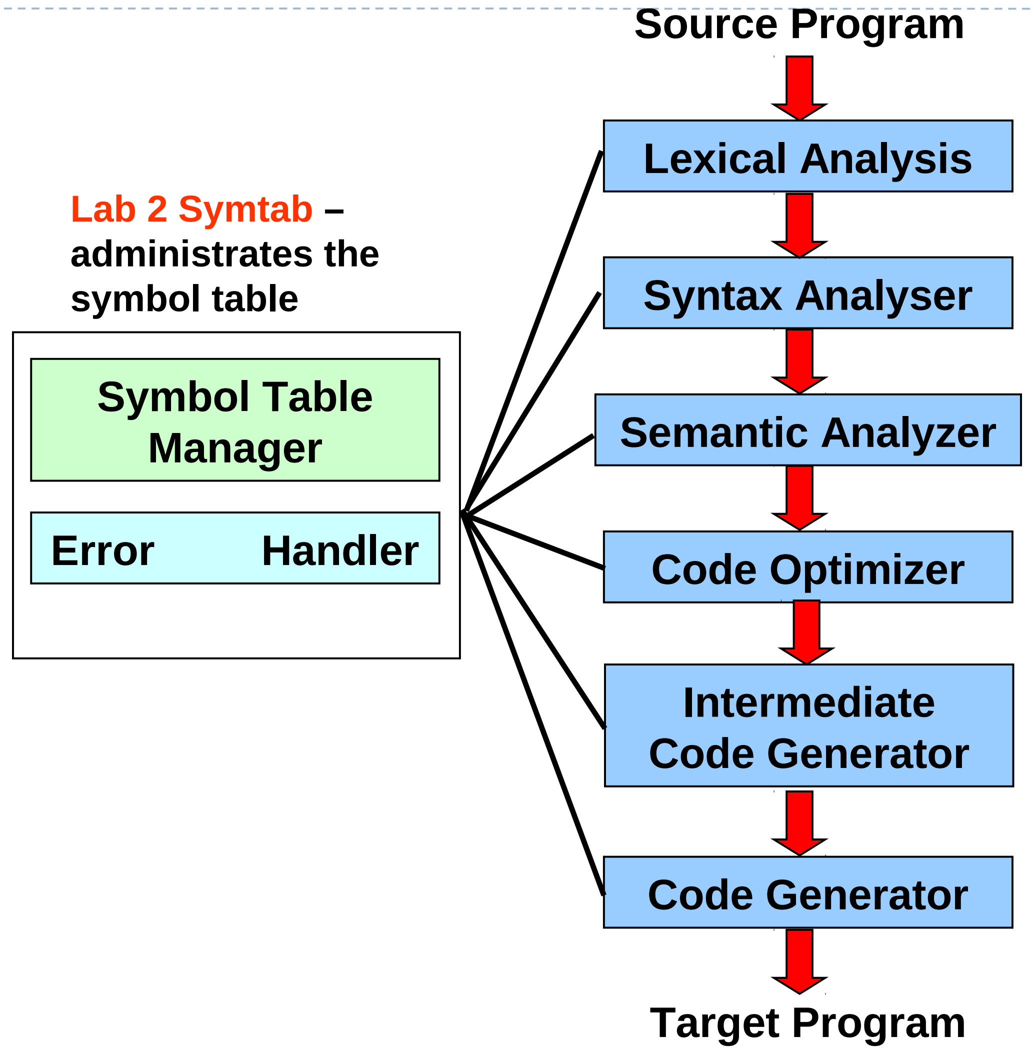
PHASES OF A COMPILER (OPTIMIZER)

INPUT OUTPUT



PHASES OF A COMPILER

Lab 2 Symtab – administrates the symbol table



Lab 1 Scanner – manages lexical analysis

Lab 3 Parser – manages syntactic analysis, build internal form

Lab 4 Semantics – checks static semantics

Lab 5 Optimizer – optimizes the internal form

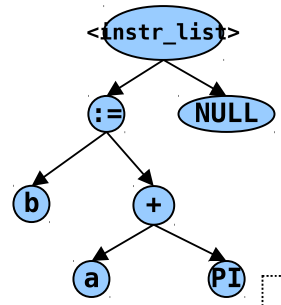
Lab 6 Quads – generates quadruples from the internal form

PHASES OF A COMPILER (QUADRUPLES)

INPUT

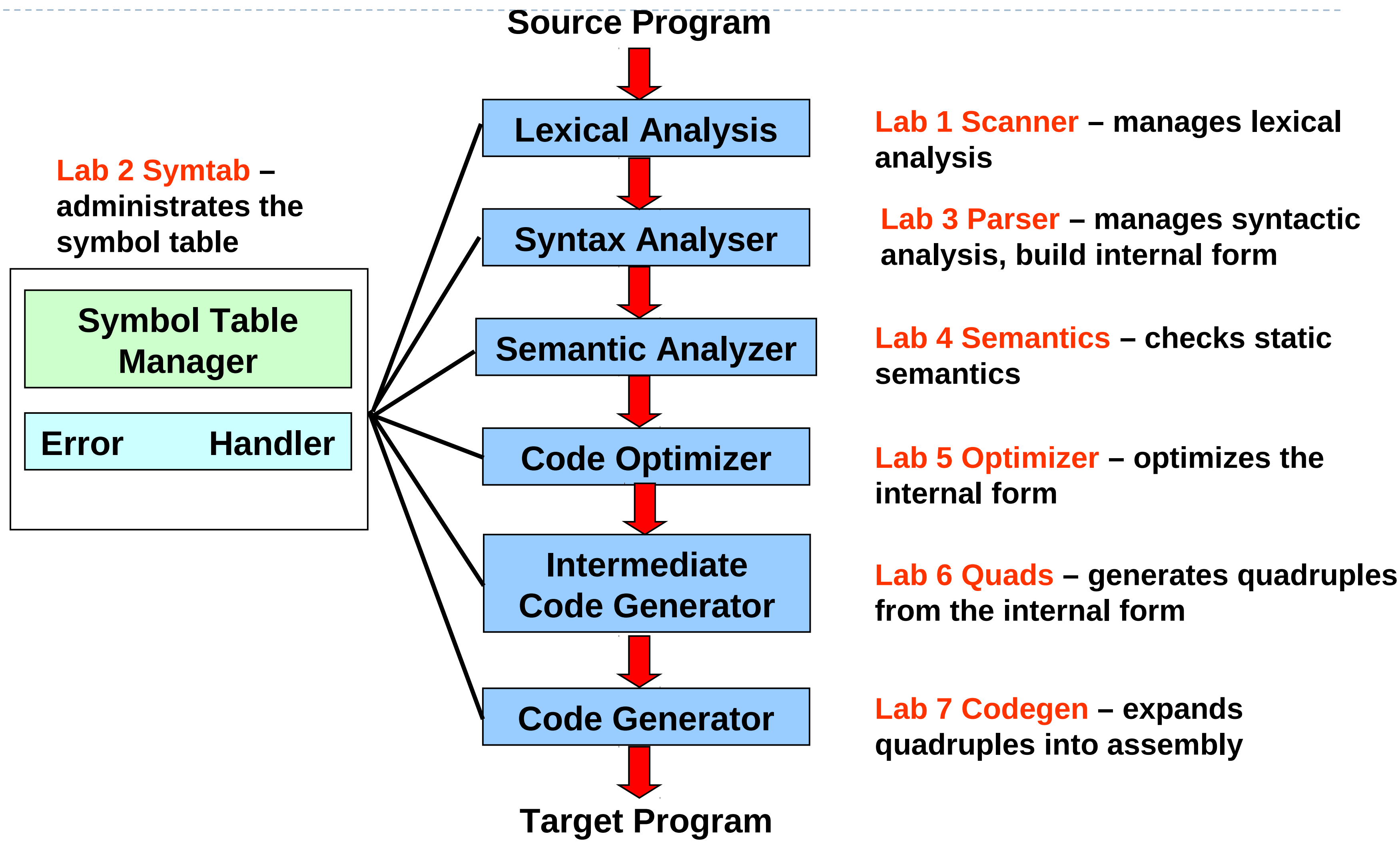
OUTPUT

```
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    -    -
```

PHASES OF A COMPILER



PHASES OF A COMPILER (CODEGEN)

INPUT

OUTPUT

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

```
#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 SKELETON

- `~TDDB44`
  - `/src`
    - `/scan`
      - Contains all the necessary files to complete the first lab
    - `/symtab`
      - Contains all the necessary files to complete the second lab
    - `/remaining`
      - Contains all the necessary files to complete the rest of the labs
    - `/testpgm`
      - Diesel programs for testing the implementation

INSTALLATION

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

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

  - Install **g++** on your account, if you don't have it:

```
module initadd prog/gcc
module add prog/gcc
```

  - More information in the Laboratory Compendium

HOW TO COMPILE

- To compile:
  - Execute **`make`** in the proper source directory
- To run:
  - Call the **`diesel`** script with the proper flags
  - The Laboratory Compendium specifies, for each lab, what test programs to run, and what flags to use.
  - (diesel script only used from assignment 3)

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 mapping the from regular expressions to Finite State Machine is straightforward, so why not 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 of the well-known **`lex`** program

HOW IT WORKS

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

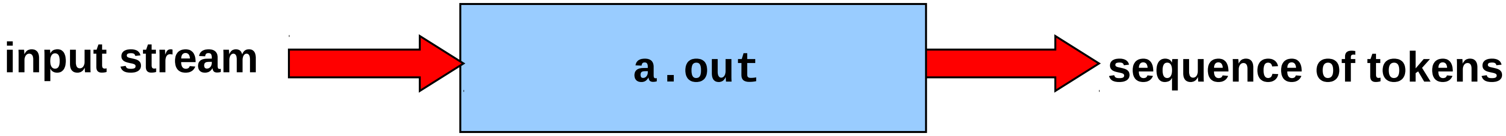
```
>> flex lex.l
```

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}**, witch 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 *translation 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

x	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 USEFULL 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.

If the lex program is to be used on its own, this section will contain a main program. If you leave this section empty you will get the default main.

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`

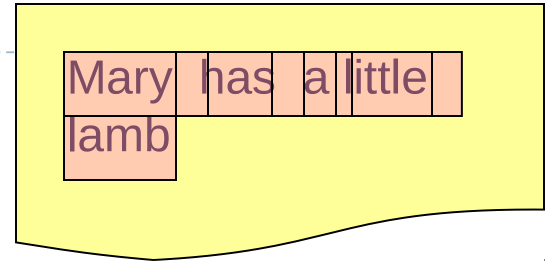
yyleng The length of the string `yytext`

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

*.|




```
%{
/* 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 printed output is the result

A SIMPLE FLEX PROGRAM

'\n' A newline increments the line count and the character count

```
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 );
        }
```

'.' Any character other than the newline only increment the character count

ANOTHER 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); }
```

ANOTHER 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.

LAB 2
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, constant, ...)

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 (keeping 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

```
main(){
   int a=10; //global variable
   changeA();
   printf(”Value of a=%d\n,a);
}

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

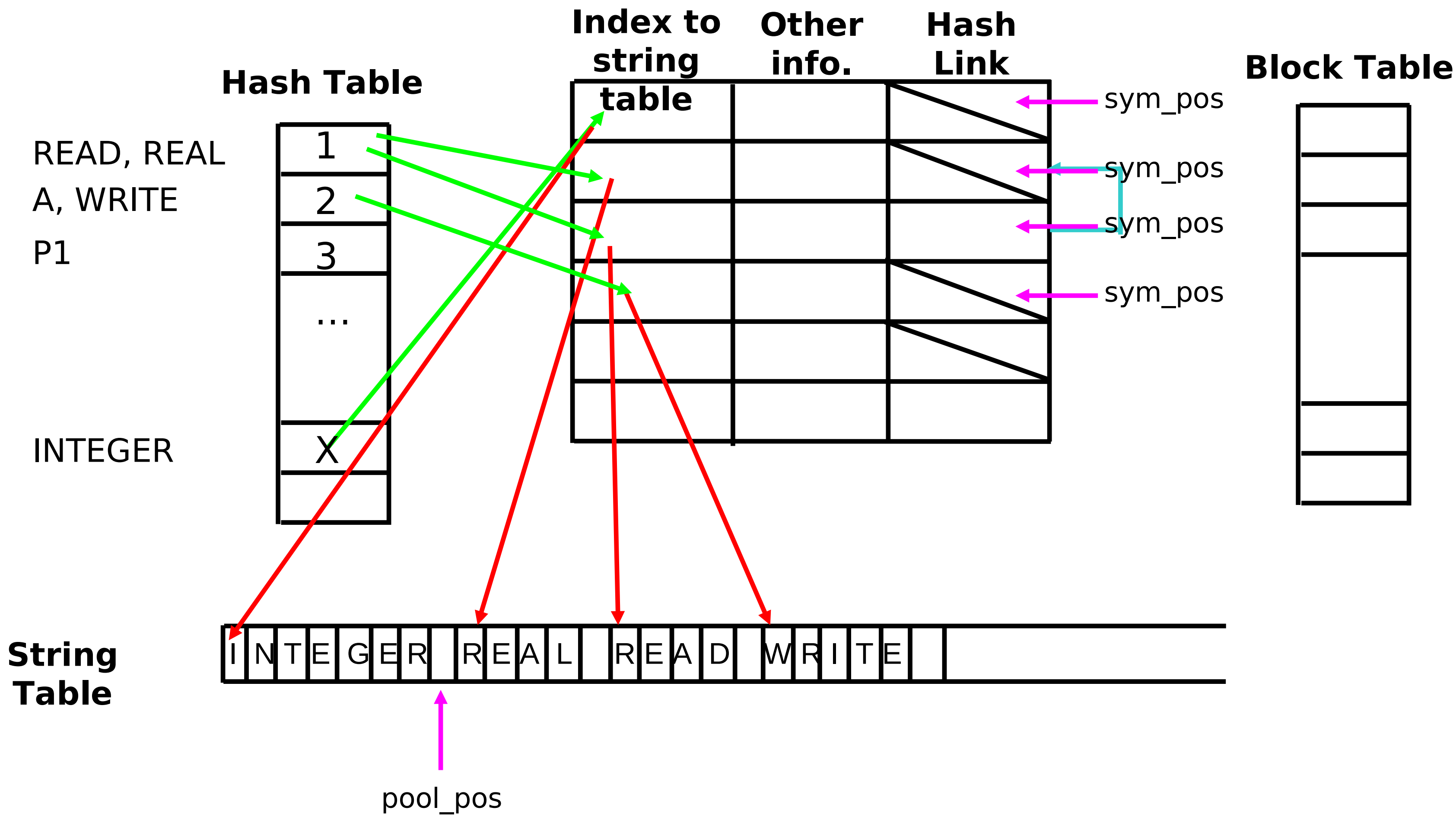
```
int a=10; //global variable

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 created scope

HOW IT WORKS



Index to string table
Other info.
Hash Link
Hash Table
Block Table
READ, REAL
A, WRITE
P1
INTEGER
1
2
3
...
X
sym_pos
sym_pos
sym_pos
sym_pos
String Table
I N T E G E R R E A L R E A D W R I T E
pool_pos

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()*.

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.
```

FILES OF INTEREST

- Files you will need to modify

(First of all you need to set the constant TEST_SCANNER in symtab.hh to 0)

  - **symtab.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.)

  - **symtab.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
  - **symtabtest.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.

LAB 3
THE PARSER

SYNTAX ANALYSIS

- The parser accepts tokens from the scanner and verifies the *syntactic correctness* of the program.
- Along the way, it also derives information about the program and builds a fundamental data structure known as **parse tree** or **abstract syntax tree** (**ast**).
- The parse tree is an internal representation of the program and augments the symbol table.

PURPOSE

- To verify the syntactic correctness of the input token stream, reporting any errors and to produce a parse tree and certain table for use by later phases.
  - Syntactic correctness is judged by verification against a formal grammar which specifies the language to be recognized.
  - Error messages are important and should be as meaningful as possible.
  - Parse tree and tables will vary depending on compiler.

METHOD

Match token stream using manually or automatically generated parser.

PARSING STRATEGIES

Two categories of parsers:

- Top-down parsers
- Bottom-up parsers

Within each of these broad categories are a number of sub strategies depending on whether leftmost or rightmost derivations are used.

TOP-DOWN PARSING

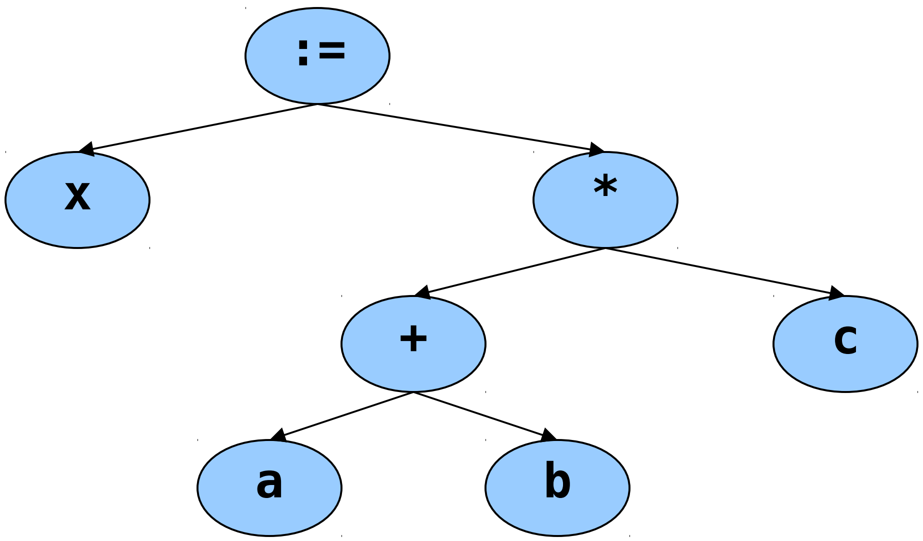
Start with a goal symbol and recognize it in terms of its constituent symbols.

Example: recognize a procedure in terms of its sub-components (header, declarations, and body).

The parse tree is then built from the top (root) and down (leaves), hence the name.

TOP-DOWN PARSING (cont'd)

```
X := ( a + b ) * c;
```



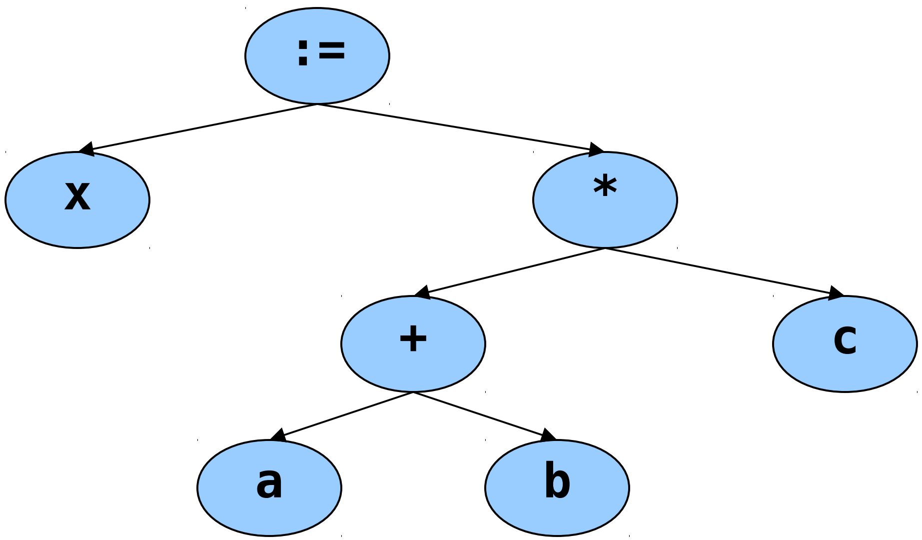
BOTTOM-UP PARSING

Recognize the components of a program and then combine them to form more complex constructs until a whole program is recognized.

The parse tree is then built from the bottom and up, hence the name.

BOTTOM-UP PARSING (cont'd)

```
X := ( a + b ) * c;
```



PARSING TECHNIQUES

A number of different parsing techniques are commonly used for syntax analysis, including:

- Recursive-descent parsing
- LR parsing
- Operator precedence parsing
- Many more …

LR PARSING

A specific bottom-up technique

- LR stands for `Left->right` scan, `Rightmost` derivation.
- Probably the most common & popular parsing technique.
- **`yacc`**, **`bison`**, and many other parser generation tools utilize LR parsing.
- Great for machines, not so great for humans …

+ AND – LR

- **Advantages of LR:**
  - Accept a wide range of grammars/languages
  - Well suited for automatic parser generation
  - Very fast
  - Generally easy to maintain

- **Disadvantages of LR:**
  - Error handling can be tricky
  - Difficult to use manually

bison AND yacc USAGE

bison is a general-purpose parser generator that converts a grammar description of an LALR(1) context-free grammar into a **C** program to parse that grammar

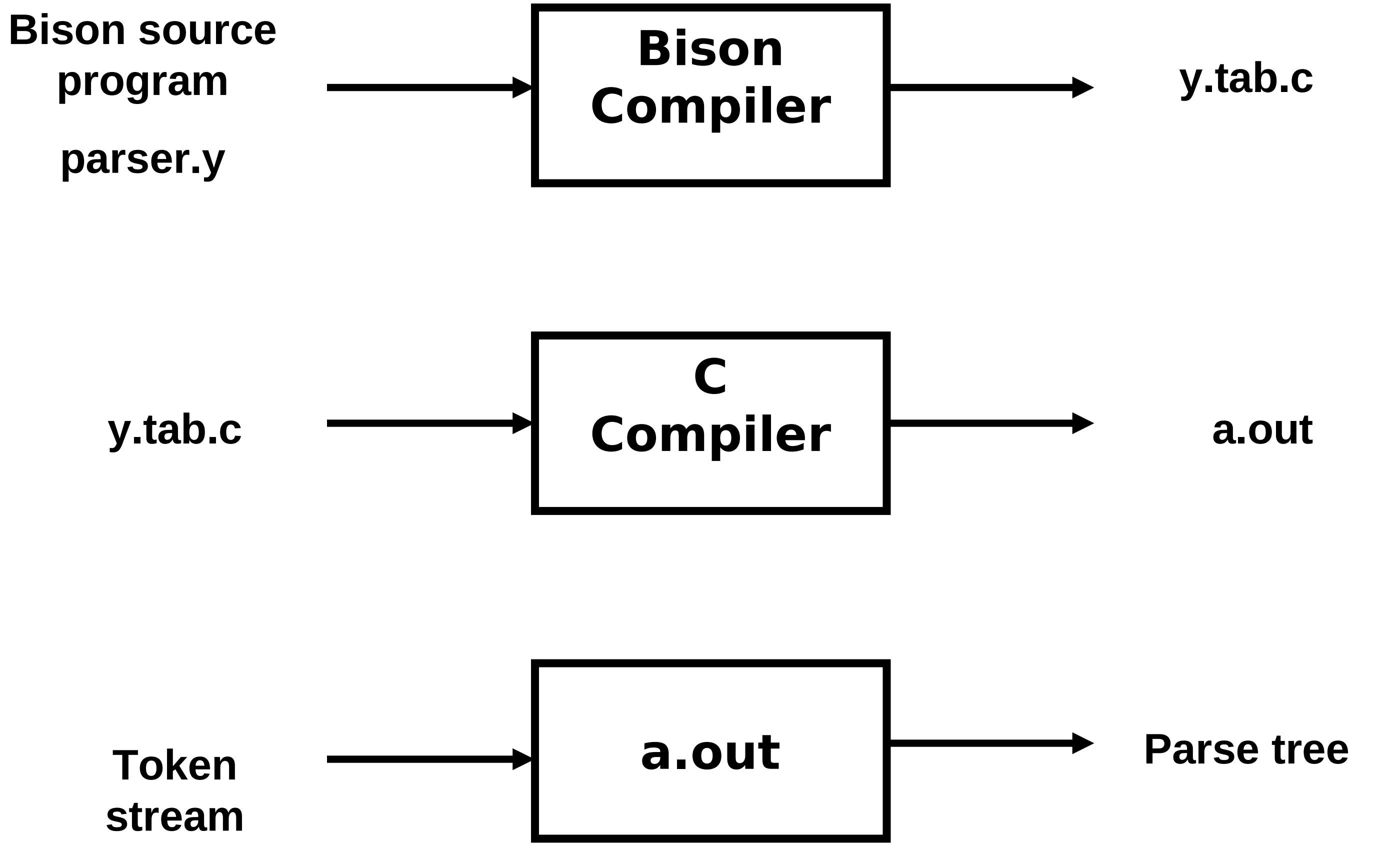
bison AND yacc USAGE

One of many parser generator packages

Yet **A**nother **C**ompiler **C**ompiler

- Really a poor name, is more of a parser compiler
- Can specify actions to be performed when each construct is recognized and thereby make a full fledged compiler but its the user of **bison** that specify the rest of the compilation process…
- Designed to work with **flex** or other automatically or hand generated “lexers”

`bison` USAGE


Bison source program
parser.y
Bison Compiler
y.tab.c
y.tab.c
C Compiler
a.out
Token stream
a.out
Parse tree

`bison` SPECIFICATION

A **`bison`** specification is composed of 4 parts

```
%{
        /* C declarations */
%}
        /* Bison declarations */

%%

        /* Grammar rules */

%%

        /* Additional C code */
```

Looks like **`flex`** specification, doesn't it?

Similar function, tools, look and feel

C DECLARATIONS

- Contains macro definitions and declarations of functions and variables that are used in the actions in the grammar rules
- Copied to the beginning of the parser file so that they precede the definition of **`yyparse`**
- Use **`#include`** to get the declarations from a header file. If **C** declarations isn’t needed, then the **`%{`** and **`%}`** delimiters that bracket this section can be omitted

`bison` DECLARATIONS

- Contains declarations that define *terminal* and *non-terminal* symbols, and specify precedence

GRAMMAR RULES

- Contains one or more **bison** *grammar rules*, and nothing else.

- There must always be at least one grammar rule, and the first **%%** (which precedes the grammar rules) may never be omitted even if it is the first thing in the file.

ADITIONAL C CODE

- Copied verbatim to the end of the parser file, just as the **C** declarations section is copied to the beginning.
- This is the most convenient place to put anything that should be in the parser file but isn’t needed before the definition of **`yyparse`**.
- The definitions of **`yylex`** and **`yyerror`** often go here.

bison EXAMPLE

```
%{
#include <ctype.h> /* standard C declarations here */
// extern int yylex();
}%
%token DIGIT /* bison declarations */
%%
/* Grammar rules */
line : expr '\n'        {  printf { "%d\n", $1 };  }    ;
expr : expr '+' term    {  $$ = $1 + $3;  }
        | term                                          ;
term : term '*' factor      {  $$ = $1 * $3;  }
        | factor                                        ;
```

bison EXAMPLE

```
factor : '(' expr ')'    {  $$ = $2;  }
       | DIGIT                                  ;
%%
/* Additional C code */

void yylex () {
   /* A really simple lexical analyzer */
   int c;
   c = getchar ();
   if ( isdigit (c) ) {
      yylval = c - '0' ;
      return DIGIT;
   }
   return c;
}
```

Note: **bison** uses **yylex**, **yylval**, etc - designed to be used with **flex**

bison EXAMPLE

```
line ::= expr \n

expr ::= term
     ::= expr + term
     ::= expr + term + term
     ::= term +    ...   + term + term + term


term ::= factor
     ::= term * factor
     ::= term * factor * factor
     ::= factor * ... * factor * factor * factor


 factor ::= DIGIT
        ::= ( expr )
        ::= ( term + term + ... + term )
        ::= ( factor * ... factor + term + ... term )
        ::= ...


DIGIT ::= [0-9]
```

bison EXAMPLE

```
|'(' '1' '*' '3' '+' '2' ')' '*' '5' '\n'
```

```
line ::= |expr '\n'
```

bison EXAMPLE

```
'(' | '1' '*' '3' '+' '2' ')' '*' '5' '\n'
```

```
  line ::=  expr '\n'
  expr ::=  term
  term ::=  factor
factor ::=  '(' | expr ')'
```

`bison` EXAMPLE

```
'(' '1'|'*' '3' '+' '2' ')' '*' '5' '\n'
```

```
  line ::=  expr '\n'
  expr ::=  term
  term ::=  factor
factor ::=  '(' expr ')'
  expr ::=  term
  term ::=  factor
factor ::=  DIGIT|
```

bison EXAMPLE

```
'(' '1' '*'|'3' '+' '2' ')' '*' '5' '\n'
```

```
  line ::=  expr '\n'
  expr ::=  term
  term ::=  factor
factor ::=  '(' expr ')'
  expr ::=  term
  term ::=  term '*'|factor
```

bison EXAMPLE

```
'(' '1' '*' '3'|'+' '2' ')' '*' '5' '\n'
```

```
  line ::=  expr '\n'
  expr ::=  term
  term ::=  factor
factor ::=  '(' expr ')'
  expr ::=  term
  term ::=  term '*' factor
factor ::=  DIGIT|
```

`bison` EXAMPLE

```
'(' '1' '*' '3' '+'|'2' ')' '*' '5' '\n'
```

```
  line ::=  expr '\n'
  expr ::=  term
  term ::=  factor
factor ::=  '(' expr ')'
  expr ::=  expr '+'|term
  term ::=  term '*' factor
```

bison EXAMPLE

```
'(' '1' '*' '3' '+' '2'|')' '*' '5' '\n'
```

```
  line ::=  expr '\n'
  expr ::=  term
  term ::=  factor
factor ::=  '(' expr ')'
  expr ::=  expr '+' term
  term ::=  factor
factor ::=  DIGIT|
```

bison EXAMPLE

```
'(' '1' '*' '3' '+' '2' ')'|'*' '5' '\n'
```

```
  line ::=  expr '\n'
  expr ::=  term
  term ::=  factor
factor ::=  '(' expr ')'|
  expr ::=  expr '+' term
  term ::=  factor
```

`bison` EXAMPLE

```
'(' '1' '*' '3' '+' '2' ')' '*'|'5' '\n'
```

```
  line ::=  expr '\n'
  expr ::=  term
  term ::=  term '*'|factor
factor ::=  '(' expr ')'
```

bison EXAMPLE

```
'(' '1' '*' '3' '+' '2' ')' '*' '5'|'\n'
```

```
  line ::=  expr '\n'
  expr ::=  term
  term ::=  term '*' factor
factor ::=  DIGIT|
```

bison EXAMPLE

```
'(' '1' '*' '3' '+' '2' ')' '*' '5' '\n'|
```

```
line ::=  expr '\n'|
expr ::=  term
term ::=  term '*' factor
```

USING `bison` WITH `flex`

`bison` and **`flex`** are obviously designed to work together

- **`bison`** produces a driver program called **`yylex()`** (actually its included in the **`lex`** library **`-ll`**)
  - **`#include “lex.yy.c”`** in the third part of **`bison`** specification
  - this gives the program **`yylex`** access to **`bisons`**’ token names

USING BISON WITH FLEX

- Thus do the following:
  - `% flex scanner.l`
  - `% bison parser.y`
  - `% cc y.tab.c -ly -ll`
- This will produce an **`a.out`** which is a parser with an integrated scanner included

ERROR HANDLING IN `bison`

Error handling in **`bison`** is provided by error productions

An error production has the general form

- **`non-terminal: error synchronizing-set`**
  - **`non-terminal`** **`where did it occur`**
  - **`error`** **`a keyword`**
  - **`synchronizing-set`** **`possible empty subset of tokens`**

When an error occurs, **`bison`** pops symbols off the stack until it finds a state for which there exists an error production which may be applied

FILES TO BE CHANGED

- **`parser.y`** is the input file to **`bison`**. This is the file you will edit most.
- **`scanner.l`** need a small, but important change. The file **`scanner.hh`** is no longer needed since there is a file **`parser.hh`**, which will contain (among other things) the same declarations. **`parser.hh`** will be generated automatically by **`bison`**. Add (in this order):
  - **`#include "ast.h"`**
  - **`#include "parser.hh"`**
  - and comment out
    - **`#include "scanner.hh"`**
  - at the top of **`scanner.l`** to reflect this.

OTHER FILES OF INTEREST

- **error.h, error.cc, symtab.hh, symbol.cc, symtab.cc** Use your completed versions from the earlier labs.
- **ast.hh** contains the definitions for the AST nodes. You'll be reading this file a lot.
- **ast.cc** contains the implementations of the AST nodes.
- **semantic.hh** and **semantic.cc** contain type checking code.
- **optimize.hh** and **optimize.cc** contain optimization code.
- **quads.hh** and **quads.cc** contain quad generation code.
- **codegen.hh** and **codegen.cc** contain assembler generation code.

OTHER FILES OF INTEREST

- **`main.cc`** this is the compiler wrapper, parsing flags and the like.
- **`Makefile`** this is not the same as the last labs. It generates a file called **`compiler`** which will take various arguments (see **`main.cc`** for information). It also takes source files as arguments, so you can start using **`diesel`** files to test your compiler-in-the-making.
- **`diesel`** this is a shell script which works as a wrapper around the binary compiler file, handling flags, linking, and such things. Use it when you want to compile a **`diesel`** file. At the top of this file is a list of all flags you can send to the compiler, for debugging, printouts, symbolic compilation and the like.