COMPILER CONSTRUCTION
Seminar 01 – TDDD44

Martin Sjölund (martin.sjolund@liu.se)
Adrian Horga (adrian.horga@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.
<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 4th</td>
<td>15-17</td>
<td>Lab 1 &amp; 2</td>
</tr>
<tr>
<td>November 18th</td>
<td>15-17</td>
<td>Lab 3 &amp; 4</td>
</tr>
<tr>
<td>December 2nd</td>
<td>15-17</td>
<td>Lab 5, 6 &amp; 7</td>
</tr>
<tr>
<td>December 16th</td>
<td>15-17</td>
<td>Exam prep</td>
</tr>
</tbody>
</table>
RELATING LABS TO THE COURSE

• Building a complete compiler
  – We use a language that is small enough to be manageable.
  – 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 intills confidence

**Disadvantages**
- High ratio of programming to though
- Cumulative nature magnifies early failures
- Many parts are simplified
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

Lab 1 Scanner – manages lexical analysis
Lab 2 Symbol Table Manager – administrates the symbol table
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 to assembler

Source Program

Lexical Analysis
Syntax Analyser
Semantic Analyzer
Intermediate Code Generator
Code Optimizer
Code Generator
Target Program

Symbol Table Manager
Error Handler
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)

Lab 1 Scanner – manages lexical analysis

Source Program

Lexical Analysis

Syntax Analyser

Semantic Analyzer

Intermediate Code Generator

Code Optimizer

Code Generator

Target Program

Symbol Table Manager

Error Handler
PHASES OF A COMPILER (SCANNER)

INPUT

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

OUTPUT

<table>
<thead>
<tr>
<th>token</th>
<th>pool_p</th>
<th>val</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_PROGRAM</td>
<td></td>
<td></td>
<td>keyword</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>EXAMPLE</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_SEMICOLON</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_CONST</td>
<td></td>
<td></td>
<td>keyword</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>PI</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_EQ</td>
<td></td>
<td></td>
<td>operator</td>
</tr>
<tr>
<td>T_REALCONST</td>
<td></td>
<td>3.14159</td>
<td>constant</td>
</tr>
<tr>
<td>T_SEMICOLON</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_VAR</td>
<td></td>
<td></td>
<td>keyword</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>A</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_COLON</td>
<td>REAL</td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>B</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_COLON</td>
<td>REAL</td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_IDENT</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_SEMICOLON</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_BEGIN</td>
<td></td>
<td></td>
<td>keyword</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>B</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_ASSIGNMENT</td>
<td></td>
<td></td>
<td>operator</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>A</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_ADD</td>
<td></td>
<td></td>
<td>operator</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>PI</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_SEMICOLON</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_END</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_DOT</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
</tbody>
</table>
PHASES OF A COMPILER (cont'd)

Lab 1 Scanner – manages lexical analysis

Lab 2 Symtab – administrates the symbol table

Symbol Table Manager

Error Handler

Source Program

Lexical Analysis

Syntax Analyser

Semantic Analyzer

Intermediate Code Generator

Code Optimizer

Code Generator

Target Program

Lab 1 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

```
<table>
<thead>
<tr>
<th>token</th>
<th>pool_p</th>
<th>val</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_IDENT</td>
<td>VOID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_IDENT</td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_IDENT</td>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_IDENT</td>
<td>EXAMPLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_IDENT</td>
<td>PI</td>
<td>3.14159</td>
<td>REAL</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>A</td>
<td></td>
<td>REAL</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>B</td>
<td></td>
<td>REAL</td>
</tr>
</tbody>
</table>
```
PHASES OF A COMPILER (cont'd)

Lab 1 Scanner – manages lexical analysis
Lab 2 Syntab – administrates the symbol table
Lab 3 Parser – manages syntactic analysis, build internal form

Source Program

Lexical Analysis

Syntax Analyser

Semantic Analyzer

Intermediate Code Generator

Code Optimizer

Code Generator

Target Program

Symbol Table Manager

Error Handler
PHASES OF A COMPILER (PARSER)

**INPUT**

<table>
<thead>
<tr>
<th>token</th>
<th>pool_p</th>
<th>val</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_PROGRAM</td>
<td></td>
<td></td>
<td>keyword</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>EXAMPLE</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_SEMICOLON</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_CONST</td>
<td></td>
<td></td>
<td>keyword</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>PI</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_EQ</td>
<td></td>
<td></td>
<td>operator</td>
</tr>
<tr>
<td>T_REALCONST</td>
<td>3.14159</td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>T_SEMICOLON</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_VAR</td>
<td></td>
<td></td>
<td>keyword</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>A</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_COLON</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>REAL</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_SEMICOLON</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>B</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_COLON</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>REAL</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_SEMICOLON</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_BEGIN</td>
<td></td>
<td></td>
<td>keyword</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>B</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_ASSIGNMENT</td>
<td></td>
<td></td>
<td>operator</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>A</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_ADD</td>
<td></td>
<td></td>
<td>operator</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>PI</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_SEMICOLON</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_END</td>
<td></td>
<td></td>
<td>keyword</td>
</tr>
<tr>
<td>T_DOT</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
</tbody>
</table>

**OUTPUT**

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

**EXAMPLE**

```
INPUT

<table>
<thead>
<tr>
<th>token</th>
<th>pool_p</th>
<th>val</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_PROGRAM</td>
<td></td>
<td></td>
<td>keyword</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>EXAMPLE</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_SEMICOLON</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_CONST</td>
<td></td>
<td></td>
<td>keyword</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>PI</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_EQ</td>
<td></td>
<td></td>
<td>operator</td>
</tr>
<tr>
<td>T_REALCONST</td>
<td>3.14159</td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>T_SEMICOLON</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_VAR</td>
<td></td>
<td></td>
<td>keyword</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>A</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_COLON</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>REAL</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_SEMICOLON</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>B</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_COLON</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>REAL</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_SEMICOLON</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_BEGIN</td>
<td></td>
<td></td>
<td>keyword</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>B</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_ASSIGNMENT</td>
<td></td>
<td></td>
<td>operator</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>A</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_ADD</td>
<td></td>
<td></td>
<td>operator</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>PI</td>
<td></td>
<td>identifier</td>
</tr>
<tr>
<td>T_SEMICOLON</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
<tr>
<td>T_END</td>
<td></td>
<td></td>
<td>keyword</td>
</tr>
<tr>
<td>T_DOT</td>
<td></td>
<td></td>
<td>separator</td>
</tr>
</tbody>
</table>
```

**TDDB44 Compiler Construction 2014 - Tutorial 1**
PHASES OF A COMPILER (cont'd)

Lab 1 Scanner – manages lexical analysis

Lab 3 Parser – manages syntactic analysis, build internal form

Lab 4 Semantics – checks static semantics

Symbol Table Manager

Error Handler

Source Program

Lexical Analysis

Syntax Analyser

Semantic Analyzer

Intermediate Code Generator

Code Optimizer

Code Generator

Target Program
PHASES OF A COMPILER (SEMANTICS)

INPUT

\[
\text{<instr\_list>}
\]

\[
\begin{align*}
\text{:=} & \quad \text{NULL} \\
\text{b} & \quad + \\
\text{a} & \quad \text{PI}
\end{align*}
\]

\[
\text{type(a) == type(b) == type(PI) ?}
\]

OUTPUT

<table>
<thead>
<tr>
<th>token</th>
<th>pool_p</th>
<th>val</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_IDENT</td>
<td>VOID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_IDENT</td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_IDENT</td>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_IDENT</td>
<td>EXAMPLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_IDENT</td>
<td>PI</td>
<td>3.14159</td>
<td>REAL</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>A</td>
<td></td>
<td>REAL</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>B</td>
<td></td>
<td>REAL</td>
</tr>
</tbody>
</table>

YES
PHASES OF A COMPILER (cont'd)

Lab 1 Scanner – manages lexical analysis
Lab 2 Symtab – administrates the symbol table
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

\[ x + 5 + 4 \]

OUTPUT

\[ x + 9 \]
PHASES OF A COMPILER (cont'd)

Lab 1 Scanner – manages lexical analysis
Lab 2 Symtab – administrates the symbol table
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

Source Program
Lexical Analysis
Syntax Analyser
Semantic Analyzer
Intermediate Code Generator
Code Optimizer
Code Generator
Target Program
PHASES OF A COMPILER (QUADS)

INPUT

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

OUTPUT

```
q_rplus    A    PI   $1
q_rassign  $1   -    B
q_labl     4    -    -
```
PHASES OF A COMPILER (cont'd)

Lab 1 Scanner – manages lexical analysis

Lab 2 Symtab – administrates the symbol table

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 to assembler

Source Program

Lexical Analysis

Syntax Analyser

Semantic Analyzer

Intermediate Code Generator

Code Optimizer

Code Generator

Target Program
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

~TDB44

/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
• 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.
HANDING IN LABS 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, 2014 (you will get 3 extra points on the final exam if you finish on time!)
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
**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 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.
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.
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…

BUILDING PARSERS WITH JAVA

Steven John Metsker

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

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

lex.l → **Lex Compiler** → lex.yy.c
**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

lex.yy.c  →  C Compiler  →  a.out
```

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

input stream  →  sequence of tokens  →  a.out
```
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:

  \[
  \text{name definition}
  \]

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

- Example:

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

  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); }
```
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 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'
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 `%%%`
**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

```c
/* includes and defines should be stated in this section */

do|does|did|done|has |[a-zA-Z]+ { printf ("\%s: is a verb\n", yytext); }  
|\n .|

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

Mary has a little lamb
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
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...)

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.)
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

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

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

```c
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
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
  (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.