Lexical and Syntax Analysis

(of Programming Languages)

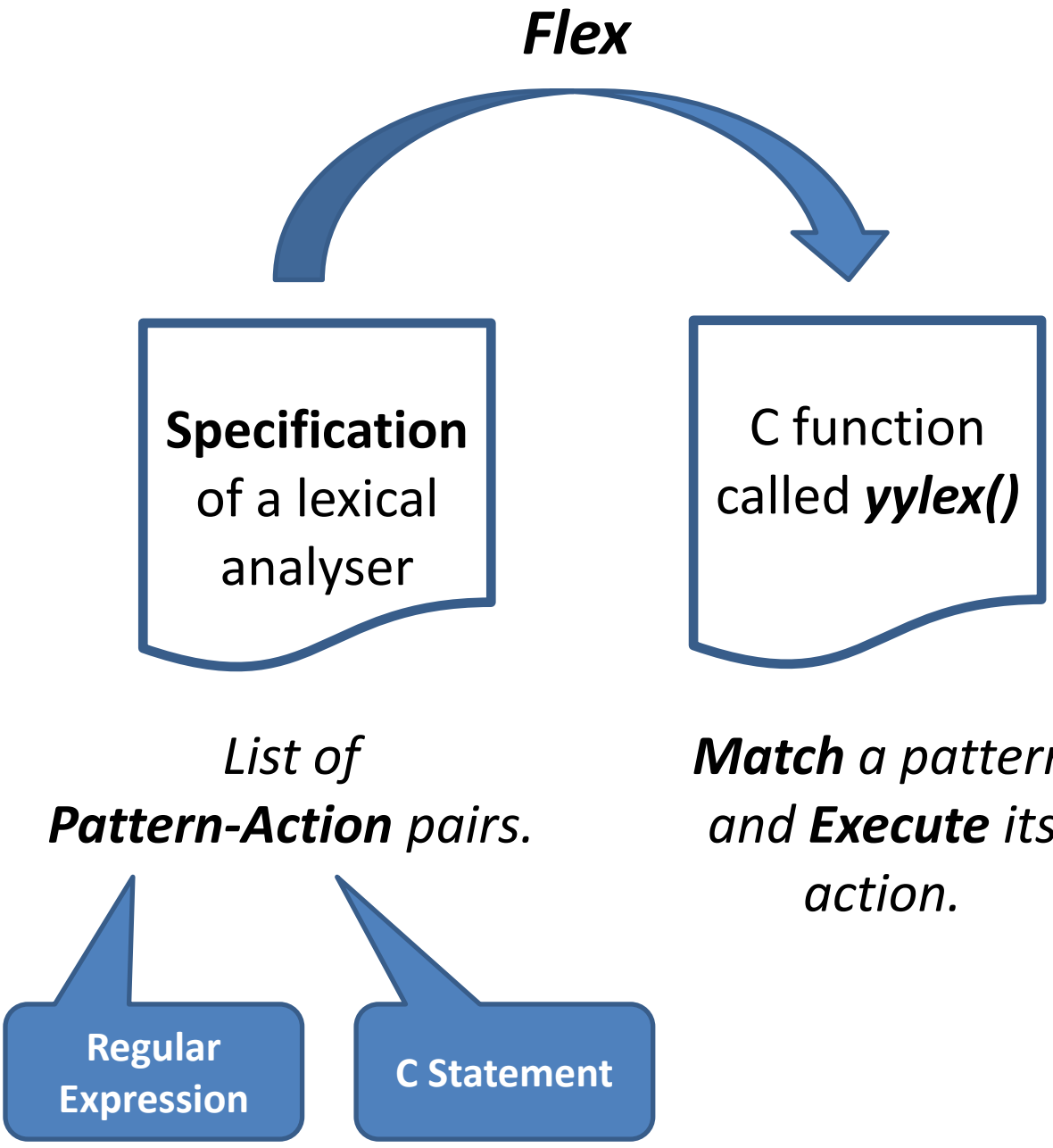
Flex, a Lexical Analyser Generator

Lexical and Syntax Analysis

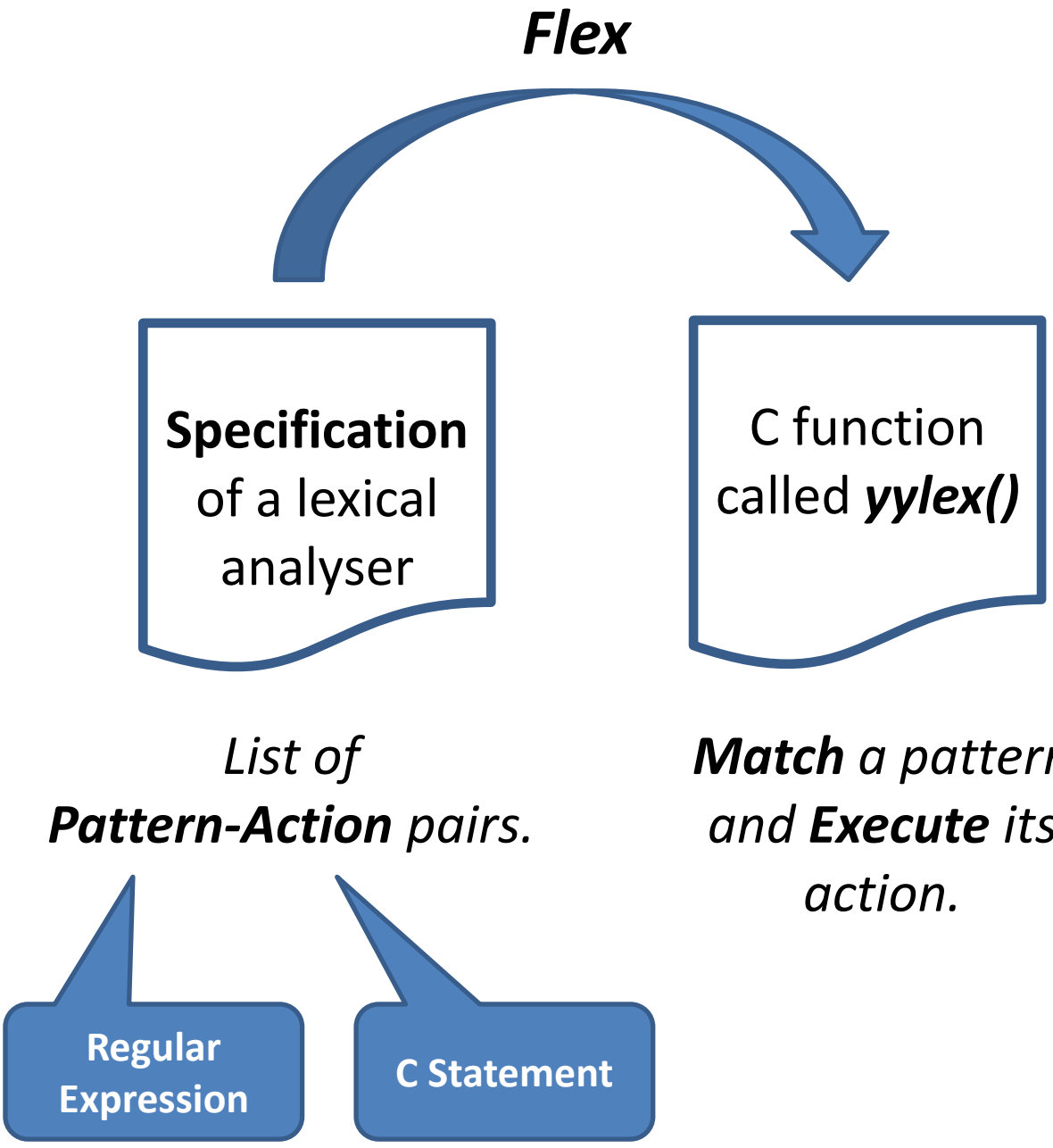
(of Programming Languages)

Flex, a Lexical Analyser Generator

Flex: a **f**ast **lex**ical analyser generator



Flex: a **f**ast **lex**ical analyser generator



Input to ***Flex***

The structure of a ***Flex*** (*.lex*) file is as follows.

/* **Declarations** */

%%

/* **Rules** (pattern-action pairs) */

%%

/* **C Code** (including *main* function) */

Any text enclosed in /* and */ is treated as a **comment**.

Input to ***Flex***

The structure of a ***Flex*** (*.lex*) file is as follows.

/* **Declarations** */

%%

/* **Rules** (pattern-action pairs) */

%%

/* **C Code** (including *main* function) */

Any text enclosed in /* and */ is treated as a **comment**.

What is a **rule**?

A **rule** is a pattern-action pair, written

pattern *action*

The **pattern** is (like) a regular expression. The **action** is a C statement, or a block of C statements in the form *{⋯}*.

What is a **rule**?

A **rule** is a pattern-action pair, written

pattern *action*

The **pattern** is (like) a regular expression. The **action** is a C statement, or a block of C statements in the form *{⋯}*.

Example 1

Replace all tom's with jerry's and vice-versa.

tomandjerry.lex

```
/* No declarations */

%%

tom             printf("jerry");
jerry           printf("tom");

%%

/* No main function */
```

Example 1

Replace all tom's with jerry's and vice-versa.

tomandjerry.lex

```
/* No declarations */

%%

tom             printf("jerry");
jerry           printf("tom");

%%

/* No main function */
```

Output of *Flex*

Flex generates a C function

```
int yylex() {
   ...
}
```

When *yylex()* is called:

- a pattern that matches a **prefix** of the input is **chosen**;
- the matching prefix is **consumed**.
- the **action** corresponding to the chosen pattern is **executed**;
- if no pattern chosen, a single character is consumed and echoed to output.
- repeats until all input consumed or an action executes a ***return*** statement.

Output of *Flex*

Flex generates a C function

```
int yylex() {
   ...
}
```

When *yylex()* is called:

- a pattern that matches a **prefix** of the input is **chosen**;
- the matching prefix is **consumed**.
- the **action** corresponding to the chosen pattern is **executed**;
- if no pattern chosen, a single character is consumed and echoed to output.
- repeats until all input consumed or an action executes a ***return*** statement.

Example 1, revisited

Replace all tom's with jerry's and vice-versa.

tomandjerry.lex

```
/* No declarations */

%%

tom             printf("jerry");
jerry           printf("tom");

%%

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

Example 1, revisited

Replace all tom's with jerry's and vice-versa.

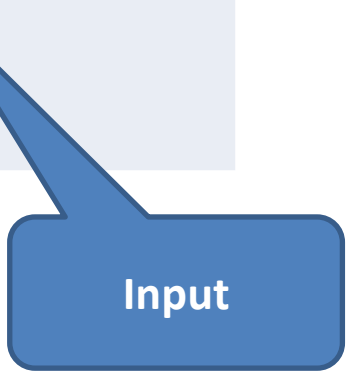
tomandjerry.lex

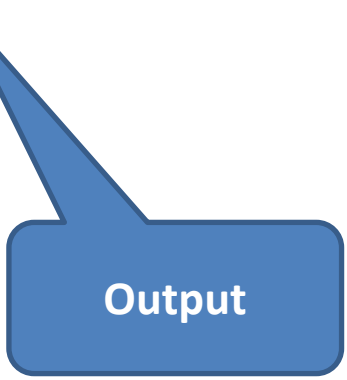
```
/* No declarations */

%%

tom             printf("jerry");
jerry           printf("tom");

%%

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

Running Example 1

At a command prompt '>':

```
> flex -o tomandjerry.c  tomandjerry.lex

> gcc -o tomandjerry  tomandjerry.c -lfl

> tomandjerry

jerry should be scared of tom.

tom should be scared of jerry.
```

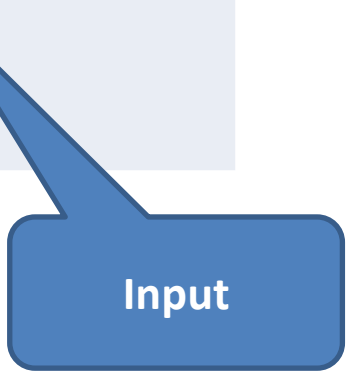


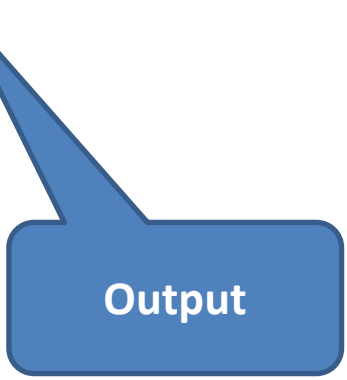


Running Example 1

At a command prompt '>':

```
> flex -o tomandjerry.c  tomandjerry.lex

> gcc -o tomandjerry  tomandjerry.c -lfl

> tomandjerry

jerry should be scared of tom.

tom should be scared of jerry.
```





Maximal munch!

Many patterns may match a prefix of the input. Which one does ***Flex*** choose?

- The one that matches the **longest** string.
- If different patterns match strings of the same length then the **first** pattern in the file is preferred.

Maximal munch!

Many patterns may match a prefix of the input. Which one does ***Flex*** choose?

- The one that matches the **longest** string.
- If different patterns match strings of the same length then the **first** pattern in the file is preferred.

What does “*yy*” mean?

The string “*yy*” is a prefix used by ***Flex*** to avoid **name clashes** with user-defined C code.

You can change the prefix by, for example, writing:

```
%option  prefix="tomandjerry_"
```

in the declarations section.

What does “*yy*” mean?

The string “*yy*” is a prefix used by ***Flex*** to avoid **name clashes** with user-defined C code.

You can change the prefix by, for example, writing:

```
%option  prefix="tomandjerry_"
```

in the declarations section.

Example 1, revisited

Replace all tom's with jerry's and vice-versa.

tomandjerry.lex

```
%option  prefix="tomandjerry_"

%%

tom             printf("jerry");
jerry           printf("tom");

%%

void main() {
   tomandjerry_lex();
}
```

Example 1, revisited

Replace all tom's with jerry's and vice-versa.

tomandjerry.lex

```
%option  prefix="tomandjerry_"

%%

tom             printf("jerry");
jerry           printf("tom");

%%

void main() {
   tomandjerry_lex();
}
```

What is a **pattern**? (Base cases)

Pattern	Meaning
x	Match the character ‘x’.
.	Match any character **except** a newline character (‘\n’).
[xyz]	Match either an ‘x’, ‘y’ or ‘z’.
[ad-f]	Match an ‘a’, ‘d’, ‘e’, or ‘f’.
[^A-Z]	Match any character **not** in the range ‘A’ to ‘Z’.
[a-z]{-}[aeiuo]	Lower case consonants.
<<EOF>>	Matches end-of-file.

What is a **pattern**? (Base cases)

Pattern	Meaning
x	Match the character ‘x’.
.	Match any character **except** a newline character (‘\n’).
[xyz]	Match either an ‘x’, ‘y’ or ‘z’.
[ad-f]	Match an ‘a’, ‘d’, ‘e’, or ‘f’.
[^A-Z]	Match any character **not** in the range ‘A’ to ‘Z’.
[a-z]{-}[aeiuo]	Lower case consonants.
<<EOF>>	Matches end-of-file.

What is a **pattern**? (Inductive cases)

If p, p_1, p_2 are patterns then:

Pattern	Meaning
p_1p_2	Match a p_1 **followed by** an p_2.
$p_1\|p_2$	Match a p_1 **or** an p_2.
$p*$	Match zero or more p's.
$p+$	Match one or more p's.
$p?$	Match zero or one p's.
$p\{2,4\}$	At least 2 p's and at most 4.
$p\{4\}$	Exactly 4 p's.
(p)	Match a p, used to override precedence.
^p	Match a p at beginning of a line
$p$$	Match a p at end of a line

What is a **pattern**? (Inductive cases)

If p, p_1, p_2 are patterns then:

Pattern	Meaning
p_1p_2	Match a p_1 **followed by** an p_2.
$p_1\|p_2$	Match a p_1 **or** an p_2.
$p*$	Match zero or more p's.
$p+$	Match one or more p's.
$p?$	Match zero or one p's.
$p\{2,4\}$	At least 2 p's and at most 4.
$p\{4\}$	Exactly 4 p's.
(p)	Match a p, used to override precedence.
^p	Match a p at beginning of a line
$p$$	Match a p at end of a line

Escaping

Reserved symbols include:

*. $ ^ [ ] - ? * + | ( ) / { } < >*

All other symbols are treated **literally**. Reserved symbols can be matched by enclosing them in double quotes. For example:

Pattern	Meaning
"[xy]"	Match '[' then 'x' then 'y' then ']'.
*"+"**	Match zero or more '+' symbols.
"\""	Match a double-quote symbol.

Escaping

Reserved symbols include:

*. $ ^ [ ] - ? * + | ( ) / { } < >*

All other symbols are treated **literally**. Reserved symbols can be matched by enclosing them in double quotes. For example:

Pattern	Meaning
"[xy]"	Match '[' then 'x' then 'y' then ']'.
*"+"**	Match zero or more '+' symbols.
"\""	Match a double-quote symbol.

What is a **declaration**?

A declaration may be:

- a **C declaration**, enclosed in *%{* and *%}*, visible to the action part of a rule.
- a **regular definition** of the form

```
name        pattern
```

introducing a new pattern *{name}* equivalent to *pattern.*

What is a **declaration**?

A declaration may be:

- a **C declaration**, enclosed in *%{* and *%}*, visible to the action part of a rule.
- a **regular definition** of the form

```
name        pattern
```

introducing a new pattern *{name}* equivalent to *pattern.*

Example 2

```
%{
   int chars = 0;
   int lines = 0;
%}

%%

.          { chars++; }
\n         { lines++; chars++; }

%%

void main() {
   yylex();
   printf("%i %i\n", chars, lines);
}
```

Example 2

```
%{
   int chars = 0;
   int lines = 0;
%}

%%

.          { chars++; }
\n         { lines++; chars++; }

%%

void main() {
   yylex();
   printf("%i %i\n", chars, lines);
}
```

Example 3

```
SPACE           [ \t\r\n]
WORD            [^ \t\r\n]+

%{
   int words = 0;
%}

%%

{SPACE}
{WORD}          { words++; }

%%

void main() {
   yylex();
   printf("%i\n", words);
}
```

Example 3

```
SPACE           [ \t\r\n]
WORD            [^ \t\r\n]+

%{
   int words = 0;
%}

%%

{SPACE}
{WORD}          { words++; }

%%

void main() {
   yylex();
   printf("%i\n", words);
}
```

yytext and ***yyleng***

The string matching a pattern is available to the action of a rule via the *yytext* variable, and its length via *yyleng*.

```
char* yytext;
int yyleng;
```

Global variables

Warning: the memory pointed to by *yytext* is destroyed upon completion of the action.

yytext and ***yyleng***

The string matching a pattern is available to the action of a rule via the *yytext* variable, and its length via *yyleng*.

```
char* yytext;
int yyleng;
```

Global variables

Warning: the memory pointed to by *yytext* is destroyed upon completion of the action.

Example 4

inc.lex

```
DIGIT          [0-9]

%%

{DIGIT}+       {
                   int i = atoi(yytext);
                   printf("%i", i+1);
               }

%%

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

Example 4

inc.lex

```
DIGIT          [0-9]

%%

{DIGIT}+       {
                   int i = atoi(yytext);
                   printf("%i", i+1);
               }

%%

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

Tokenising using Flex

The idea is that *yylex()* returns **the next token**. This is achieved by using a ***return*** statement in the action part of a rule.

Some tokens have a **component value**, e.g. *NUM*, which by convention is returned via the global variable *yylval*.

```
int yylval;
```

Global variable

Tokenising using Flex

The idea is that *yylex()* returns **the next token**. This is achieved by using a ***return*** statement in the action part of a rule.

Some tokens have a **component value**, e.g. *NUM*, which by convention is returned via the global variable *yylval*.

```
int yylval;
```

Global variable

Example 5

nums.lex

```
%{
   typedef enum { END, NUM }  Token;
%}

%%

[^0-9]          /* Ignore */
[0-9]+          {
                   yylval = atoi(yytext);
                   return NUM;
                }
<<EOF>>         { return END; }

%%

void main() {
   while (yylex() != END)
      printf("NUM(%i)\n", yylval);
}
```

Example 5

nums.lex

```
%{
   typedef enum { END, NUM }  Token;
%}

%%

[^0-9]          /* Ignore */
[0-9]+          {
                   yylval = atoi(yytext);
                   return NUM;
                }
<<EOF>>         { return END; }

%%

void main() {
   while (yylex() != END)
      printf("NUM(%i)\n", yylval);
}
```

The type of ***yylval***

By default *yylval* is of type ***int***, but it can be overridden by the user. For example:

```
union {
    int number;
    char* string;
} yylval;
```

Now *yylval* is of union type and can either hold a number or a string.

NOTE: When interfacing ***Flex*** and ***Bison***, the type of *yylval* is defined in the ***Bison*** file using the *%union* option.

The type of ***yylval***

By default *yylval* is of type ***int***, but it can be overridden by the user. For example:

```
union {
    int number;
    char* string;
} yylval;
```

Now *yylval* is of union type and can either hold a number or a string.

NOTE: When interfacing ***Flex*** and ***Bison***, the type of *yylval* is defined in the ***Bison*** file using the *%union* option.

Start conditions and states

If *p* is a pattern then so is *<s>p* where *s* is a state. Such a pattern is only **active** when the scanner is in state *s*.

Initially, the scanner is in state *INITIAL*. The scanner **moves** to a state *s* upon execution of a *BEGIN(s)* statement.

Start conditions and states

If *p* is a pattern then so is *<s>p* where *s* is a state. Such a pattern is only **active** when the scanner is in state *s*.

Initially, the scanner is in state *INITIAL*. The scanner **moves** to a state *s* upon execution of a *BEGIN(s)* statement.

Inclusive states

An **inclusive state** *S* can be declared as follows.

```
%s S
```

When the scanner is in state *S* any rule with start condition *S* **or** no start condition is active.

Inclusive states

An **inclusive state** *S* can be declared as follows.

```
%s S
```

When the scanner is in state *S* any rule with start condition *S* **or** no start condition is active.

Exclusive states

An **exclusive state** *S* can be declared as follows.

```
%x S
```

When the scanner is in state *S* **only** rules with the start condition *S* are active.

Exclusive states

An **exclusive state** *S* can be declared as follows.

```
%x S
```

When the scanner is in state *S* **only** rules with the start condition *S* are active.

Example 6

capitalise.lex

```
%s   NS          /* New Sentence */

%%

"."              {  putchar('.');
                    BEGIN(NS);
                 }
<NS>[a-zA-Z]     { putchar(toupper(*yytext));
                   BEGIN(INITIAL);
                 }

%%

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

Example 6

capitalise.lex

```
%s   NS          /* New Sentence */

%%

"."              {  putchar('.');
                    BEGIN(NS);
                 }
<NS>[a-zA-Z]     { putchar(toupper(*yytext));
                   BEGIN(INITIAL);
                 }

%%

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

Other aspects of ***Flex***

- In an action, executing

  ***yyless**(n)*

  puts *n* characters back into the input stream.

- The global variables

*FILE** ***yyin**;*	(Default: ***stdin**)*
*FILE** ***yyout**;*	(Default: ***stdout**)*

  define the input and output streams used by ***Flex***.

Other aspects of ***Flex***

- In an action, executing

  ***yyless**(n)*

  puts *n* characters back into the input stream.

- The global variables

*FILE** ***yyin**;*	(Default: ***stdin**)*
*FILE** ***yyout**;*	(Default: ***stdout**)*

  define the input and output streams used by ***Flex***.

Exercise 1

Consider the following output of the Unix command *ls -l*.

```
-rw-r--r-- 1 mfn staff 12 2011-03-06 17:43 file1.txt
-rw-r--r-- 1 mfn staff 13 2011-03-06 17:43 file2.txt
-rw-r--r-- 1 mfn staff  9 2011-03-06 17:43 file3.txt
```

Write a ***Flex*** specification that takes the output of *ls -l* and produces the sum of the sizes of all the listed files.

Exercise 1

Consider the following output of the Unix command *ls -l*.

```
-rw-r--r-- 1 mfn staff 12 2011-03-06 17:43 file1.txt
-rw-r--r-- 1 mfn staff 13 2011-03-06 17:43 file2.txt
-rw-r--r-- 1 mfn staff  9 2011-03-06 17:43 file3.txt
```

Write a ***Flex*** specification that takes the output of *ls -l* and produces the sum of the sizes of all the listed files.

Exercise 2

Recall the source language of the expression simplifier.

$$\underline{v} = [a\text{-}z]^+$$

$$\underline{n} = [0\text{-}9]^+$$

$$\begin{array}{lcl} \underline{e} & \rightarrow & \underline{v} \\ & | & \underline{n} \\ & | & \underline{e} + \underline{e} \\ & | & \underline{e} * \underline{e} \\ & | & (\underline{e}) \end{array}$$

Write a lexical analyser for the expression language.

Exercise 2

Recall the source language of the expression simplifier.

$$\underline{v} = [a\text{-}z]^+$$

$$\underline{n} = [0\text{-}9]^+$$

$$\begin{array}{lcl} \underline{e} & \rightarrow & \underline{v} \\ & | & \underline{n} \\ & | & \underline{e} + \underline{e} \\ & | & \underline{e} * \underline{e} \\ & | & (\underline{e}) \end{array}$$

Write a lexical analyser for the expression language.

Variants of ***Flex***

There are ***Flex*** variants available for many languages:

Language	Tool
C++	Flex++
Java	JLex
Haskell	Alex
Python	PLY
Pascal	TP Lex
*	ANTLR

Variants of ***Flex***

There are ***Flex*** variants available for many languages:

Language	Tool
C++	Flex++
Java	JLex
Haskell	Alex
Python	PLY
Pascal	TP Lex
*	ANTLR

Summary

- ***Flex*** converts a list of **pattern-action** pairs into C function called *yylex()*.
- Patterns are similar to **regular expressions**.
- The idea is that *yylex()* identifies and returns the **next token** in the input.
- Gives a declarative (**high level**) way to define lexical analysers.