

# **Formal Languages and Compilers Laboratory**

## **Lexical analysis: FLEX**

Daniele Cattaneo

Material based on slides by Alessandro Barenghi and Michele Scandale

# Lexical

*“Relating to words or vocabulary of a language as distinguished from its grammar and construction”*

Webster's Dictionary

# Words

Words are *simple constructs*:

- in a natural language we can just enumerate them
- enumeration is not possible with artificial languages (**too many words**)

## C identifiers rules

- a sequence of alphanumeric characters (plus underscore \_)
- cannot start with a digit

Technical words are simpler than natural words:

- structure is simple
- they follow specific rules
- they are usually a *regular language*

# Lexical analysis purpose

A lexical analysis must:

- *recognize* tokens in a stream of characters (e.g., identifiers, constants)
- possibly *decorate* tokens with additional info (e.g., the name of the identifier, line-wise location)

Such analysis is usually performed through a scanner:

- coding a scanner by hand is both tedious and error-prone
- there are scanner generators based on regular expression description (e.g., FLEX)

A scanner is just a big **Finite State Automaton**.

# f1ex: Fast Lexical Analyzer

For some applications, a scanner is enough:

- can be used to detect words and apply semantic actions (e.g., local transformations)

In a compiler, instead, the scanner *prepares the input* for the parser:

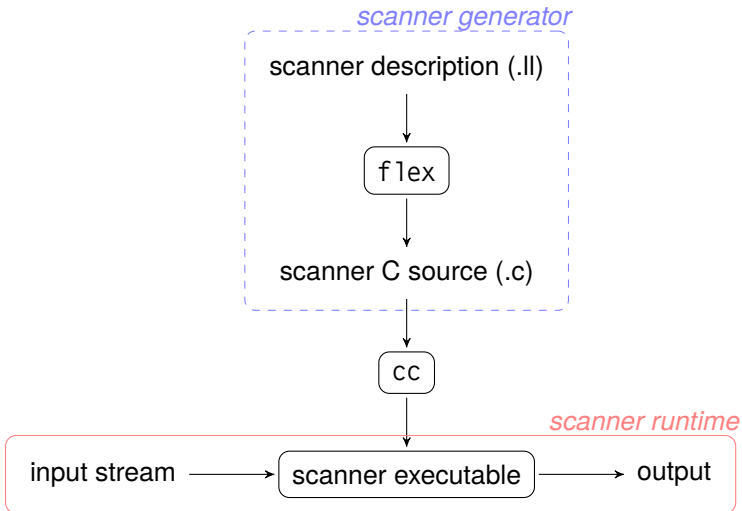
- detects the *tokens* of the language (e.g., identifiers, constants, keywords, punctuation)
- cleans the input (e.g. drops comments)
- adds information to the tokens (e.g. lexical value, location)

f1ex is a **lexical analyzer generator**:

- **Input**: a **specification file** of the scanner
- **Output**: a **C source code file** that **implements the scanner**

# flex

## Workflow



# flex

## File format

A flex file is structured in three sections separated by %%:

- **definitions**: declare useful REs
- **rules**: bind RE combinations to actions
- **user code**: C code (generally helper functions)

Definitions

%%

Rules

%%

User code

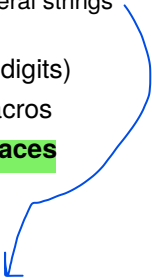
# flex

## File format: definitions

A definition associates a name to a set of *characters*:

- regular expressions can be used to define character sets
  - addition to the standard syntax: quotes used for literal strings (counts as 1 symbol for precedence purposes)
- usually employed to define *simple concepts* (e.g., digits)
- they perform a task similar to C's preprocessor macros
- they are recalled by putting their name in curly braces

LETTER	[a-zA-Z_]
DIGIT	[0-9]
HELLOWORLD	"*hello world*"
LETTERDIGIT	{LETTER}{DIGIT}





# flex

## File format: rules

A rule represents a full token to be recognized:

- the token is described by a regular expression
- exploits definitions to define aggregate concepts (e.g., numbers, identifiers)
- defines a **semantic action** to be made at each match

<i>// Regex</i>		<i>Semantic action</i>
{LETTER}({LETTER} {DIGIT})*		{ <b>return</b> 1; }
{DIGIT}+		{ <b>return</b> 2; }
[\t]+	ignores white spaces	{ /* do nothing */ }
"if"		{ <b>return</b> 3; }

# flex

## File format: rules

### Semantic actions:

- are executed every time the rule is matched
- can access matched textual data

### Global variables defined for semantic actions:

Variable	Type	Meaning
yytext	char*	matched text
yylen	int	matched text length

# flex

## File format: rules

Simple applications put the **business logic** directly inside semantic actions.

More complex applications that also use a separate **parser** (e.g. compilers) instead do the following:

- 1 assign a value to the recognized token (lexical value)
- 2 return the token type

# flex

## User code

User C code is copied to the generated scanner **as is**.

Useful stuff to have:

- the `main` function
- any other routine called by actions
- scanner-wrapping routines
- ...

```
/* Definitions */
```

```
%%
```

```
/* Rules */
```

```
%%
```

```
int main(int argc,  
         char *argv[])  
{  
    /* ... */  
}
```

# flex

## User code

Arbitrary code can also be put inside definitions and rules sections by **escaping** from flex through wrapping the code within `%{, %}` braces:

- the code is copied **as is** into the generated scanner
- generally used for header inclusions, globals, forward declarations of functions, ...

```
%{  
#include <limits.h>  
#include <stdio.h>  
  
int my_var = 0;  
%}
```

# flex

## The generated scanner

### Remember!

- flex **generates** a scanner
- It is not a scanner itself!

The generated scanner is a **C file** called **lex.yy.c**. It exports:

```
FILE *yyin = stdin;  
int yylex(void); --> function which is the scanner itself and reads  
                        from *yyin file
```

The `yylex()` function parses the file `yyin` **until**:

- A semantic action **returns**
  - The return value is the same as the one of the action
  - To continue parsing call `yylex()` again
- The file ends. Return value = 0 (zero)

# flex

## The generated scanner

Flex requires you to implement a single function:

```
int yywrap(void);
```

The *yywrap()* function is called **when the file ends**.

It gives the opportunity to **open another file** and continue scanning from there.

- Return 0 if the parsing should continue
- Return 1 if the parsing should stop

If you don't want this, you must put the following line in the scanner source:

```
%option noyywrap
```

# flex

## Scanner behavior

Some last important rules to remember:

### Longest matching rule

If more than one matching string is found, the rule that generates the **longest** one is selected

### First rule

If more than one string with the same length is matched, the rule listed **first** will be triggered (the first in the order we defined them in the definition file, that means that we should define the specific rules first)

### Default action

If no rules are found, the next character in input is considered matched implicitly and **printed to the output stream as is**  
when you don't match you print the text as is



# flex

## Case lowering tool

```
%{  
#include <ctype.h>  
%}  
%option noyywrap
```

In the definition part we don't have any definition, we only include a library and insert the noyywrap to avoid the yywrap function (see slide 15)

```
%%
```

```
[A-Z]      { printf("%c", tolower(yytext[0])); }
```

```
%%
```

code part, it doesn't do anything since it only calls yylex

```
int main(int argc, char **argv)  
{  
    return yylex();  
}
```

match each uppercase character,  
if don't match it prints non-matching character  
as they are (see default rule)

# flex

Not a compiler yet but an interpreter: it interprets the input language

## “Simple” calculator (only + and -)

```
%{  
#include <stdlib.h>  
#include <stdio.h>  
  
#define END 0  
#define NUM 1  
#define PLUS 2  
#define MINUS 3  
%}  
  
%option noyywrap  
  
%%  
[ \t]+ --> silently discard spaces  
[0-9]+ { return NUM; }  
"+" { return PLUS; }  
"- " { return MINUS; }  
"\n" { return END; }  
  
%%  
  
the function  
that implements  
the actual parser  
int main(int argc, char *argv[])  
{  
    checks if the first  
    element is not a  
    number, is so we have  
    an error  
    if (yylex() != NUM)  
        return 1;  
    int accum = atoi(yytext);  
  
    int op_input = yylex();  
    while (op_input != END) {  
        if (yylex() != NUM)  
            return 1; --> error if not a number  
        int right_side=atoi(yytext);  
  
        otherwise checks for the operator  
        if (op_input == PLUS)  
            accum += right_side;  
        else if (op_input == MINUS)  
            accum -= right_side;  
        else  
            return 1;  
  
        op_input = yylex();  
    }  
    printf("%d\n", accum);  
    return 0;  
}
```

# flex

## How it works

The generated parser implements a **non-deterministic finite state automaton**

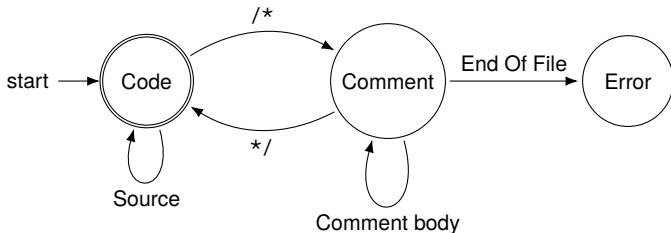
- The automaton tries to match all possible tokens at the same time
- As soon as one is recognized:
  - 1 The semantic action is executed
  - 2 The stream skips past the end of the token
  - 3 The automaton reboots

Actually, the NFA is translated into a deterministic automaton using a modified version of the Berry-Sethi algorithm

# flex

## Multiple scanners

Sometimes is useful to have more than one scanner together (e.g., a code scanner and a comment scanner).



Put together the two scanners, and move from one to another when special characters are recognized

# flex

## Multiple scanners

In order to support multiple scanners:

- rules can be marked with the name of the associated scanner (**start condition**)
- special actions to switch between scanners

A start condition S:

- is used to mark rules with as a prefix <S>RULE
- marks rules as active when the scanner is running the S scanner

Moreover:

- the \* start condition matches every start condition
- the initial start condition is INITIAL
- start conditions are stored as integers
- the current start condition is stored in the YY\_START variable

# flex

## Multiple scanners

Start conditions can be:

**exclusive** declared with %x S; disables unmarked rules when the scanner is in the S start condition

**inclusive** declared with %s S; unmarked rules active when scanner is in the S start condition

The INITIAL condition is **inclusive**.

Here is a table with relevant special actions:

Action	Meaning
BEGIN(S)	place scanner in start condition S
ECHO	copies yytext to output

# flex

## Multiple scanners: example

If rules are marked they are active only when we get into that start condition, the ones that are not marked are always marked are always active unless we got into an exclusive start condition


Let's implement a C block comment eater:

**%x** COMMENT --> defining exclusive start condition: when we get into COMMENT start condition, the rules not marked as COMMENT rules are not active  
**%option** noyywrap

**%%**

initial state: `"/"`

**COMMENT** start condition: `<COMMENT>[*]*` *// eat all characters except `"*"`*  
`<COMMENT>"*" + [* /]*` *// eat string of `"*"` not followed by `"/"`*  
`<COMMENT>"*" + "/"` *{ BEGIN(INITIAL); } --> go back to the initial state*  
**%%**



```
int main(int argc , char* argv[])
{
    return yylex();
}
```

# Homework

For normal people:

- Modify the simple calculator shown before to generate syntax errors when invalid characters are inserted

For masochists:

- Modify the block comment eater to handle **nested comments**  
`/* ... /* ... */ still in comment */`
- **Exam term 2017-07-19:** Write a flex program that:
  - ① Replaces all the uppercase letters enclosed in square brackets with their corresponding lowercase ones
  - ② Replaces all the letters enclosed in curly braces with their corresponding ASCII code, printed as a decimal number
  - ③ Leaves all the remaining characters untouched

You may assume that no brackets nesting of any kind appears in the input. Any character which is not a letter is printed in output without change.