Formal Languages and Compilers

Lexical analysis: FLEX

Alessandro Barenghi & Michele Scandale

January 8, 2020

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 technical languages (too many words)

C identifiers rules

- a sequence of non-digit characters (including underscore _, the lower and upper case Latin letters) and digits
- 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.

`flex`: Fast Lexical Analyzer

For some applications, a scanner is enough:

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

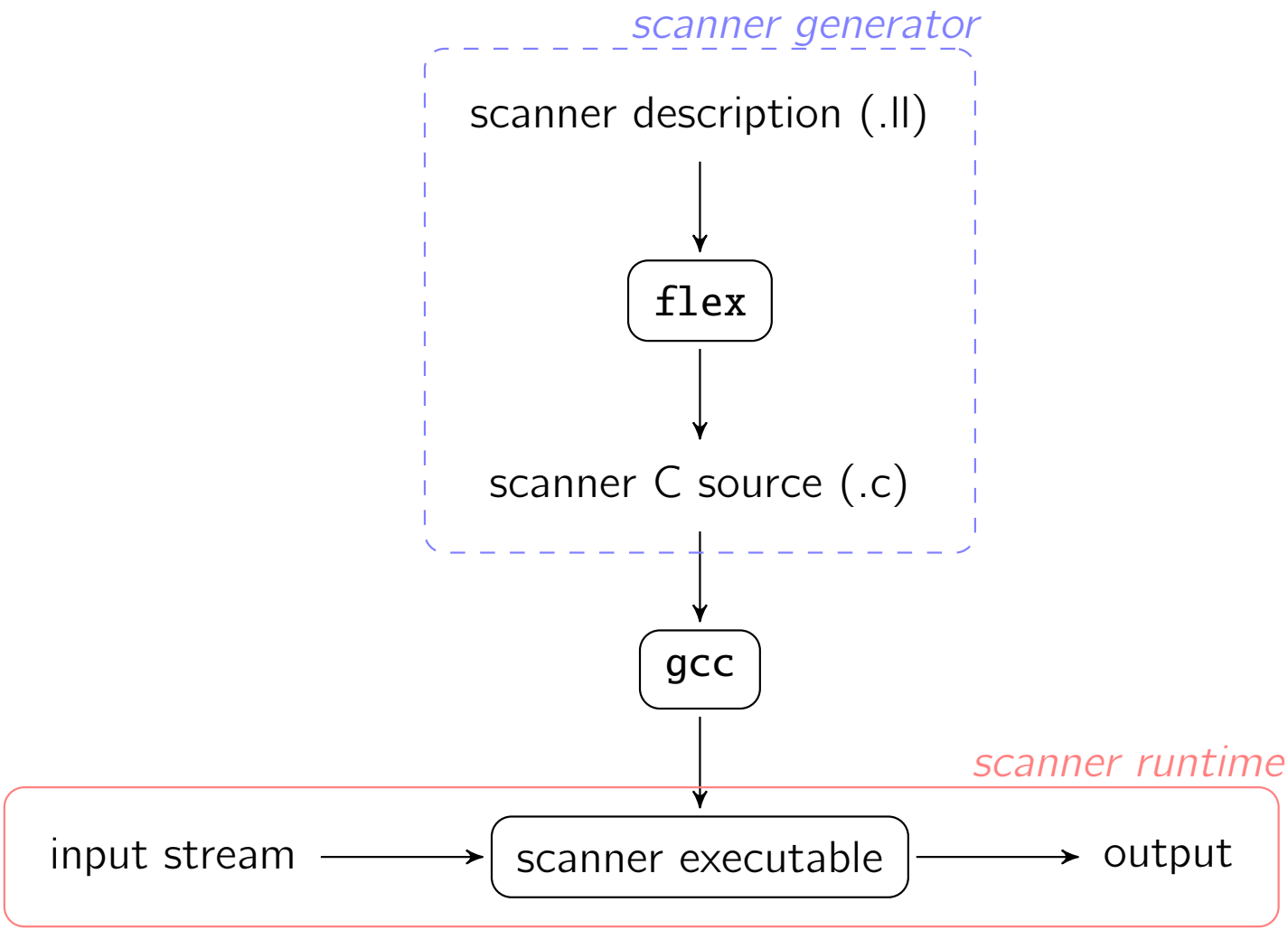
The task of a compiler cannot be accomplished only by a scanner, thus a scanner *prepares the input* for the parser:

- 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)

Main function: `int yylex()`.

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
- usually employed to define *simple concepts* (e.g., digits)
- they perform a task similar to C's preprocessor macros

```
// lower and upper case letters
LETTER          [a-zA-Z]
// numerical digits
DIGIT           [0-9]
```

flex

File format: rules

A rule represents a full token to be recognized:

- uses common regular expressions
- exploits definitions to define aggregate concepts (e.g., numbers, identifiers)
- defines an action to be made at each match

```
// C identifiers
{LETTER}({LETTER}|{DIGIT})*      { return ID; }
// numerical digits
{DIGIT}+                         { return NUMBER; }
// 'if' keyword
"if"                             { return IF; }
```

flex

Regular Expressions

Basic regular expressions:

Syntax	Matches
`x`	the `x` character
`.`	any character except newline
`[xyz]`	`x` or `y` or `z`
`[a-z]`	any character between `a` and `z`
`[^a-z]`	any character except those between `a` and `z`
`{X}`	expansion of `X` definition
`"hello"`	the `hello` string

flex

Regular Expressions

Composition rules:

Syntax	Matches
`R`	the `R` regular expression
`RS`	concatenation of `R` and `S`
`R|S`	either `R` or `S`
`R*`	zero or more occurrences of `R`
`R+`	one or more occurrences of `R`
`R?`	zero or one occurrence of `R`
`R{m,n}`	a number or `R` occurrences ranging from `n` to `m`
`R{n,}`	`n` or more occurrences of `n`
`R{n}`	exactly `n` occurrences of `R`

flex

Regular Expressions

Regular expression utilities:

Syntax	Matches
`(R)`	override precedence
`^R`	`R` at beginning of a line
`R$`	`R` at the end of a line

Note that most of the UNIX tools handling regular expressions (e.g., **`grep`**) accept *the same* syntax.

flex

File format: rules

Actions:

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

Simple scanners execute directly the semantic action.
Complex scanners (e.g. programming language tokenizer):

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

Here are some useful variables you may need to access during the lexing actions:

Variable	Type	Meaning
`yytext`	`char*`	matched text
`yyleng`	`int`	matched text length

flex

User code

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

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

Arbitrary code can 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

Example

Lets implement a case lowering tool:

```
%{
#include <ctype.h>
%}
%option noyywrap
UPPER [A-Z]
%%
{UPPER}       { printf("%c", tolower(yytext[0])); }
%%
int main(int argc, char **argv) {
  return yylex();
}
```

flex

Example

The generated tables describe a *finite state automaton*. Here's the source:

```
/* States */
static yyconst flex_int16_t yy_def[7] =
  { 0, 6, 1, 6, 6, 6, 0 };
/* Accepting states */
static yyconst flex_int16_t yy_accept[7] =
  { 0, 0, 0, 3, 2, 1, 0 };
/* Starting state */
static int yy_start = 0;

/* Transitions */
static yyconst flex_int16_t yy_nxt[7] =
  { 0, 4, 5, 6, 3, 6, 6 };
```

flex

Scanner behavior

The scanner applies the following rules:

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

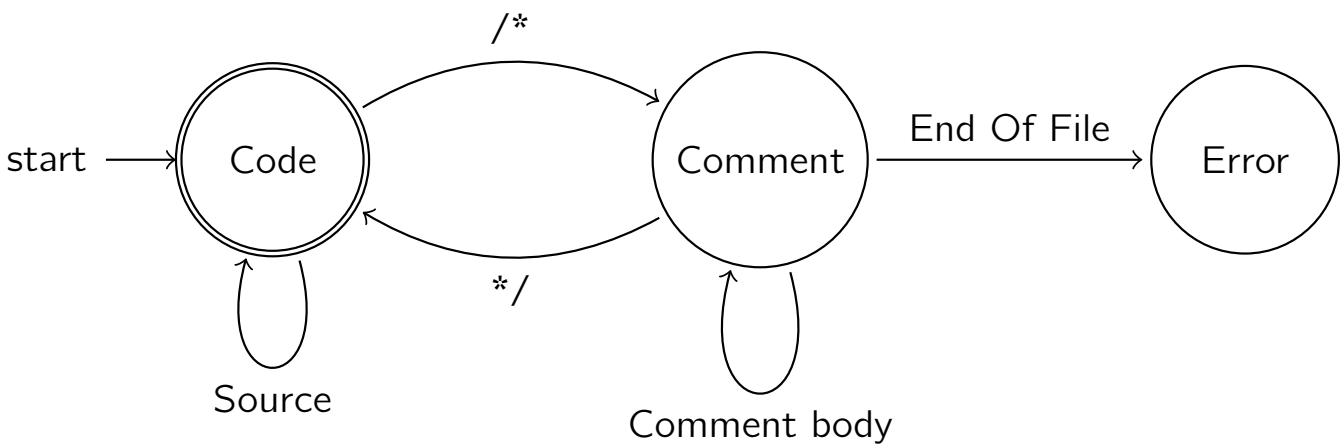
first rule if more than one string with the same length is matched, the rule listed first is will be triggered

default action if no rules are found, the next character in input is considered matched implicitly and copied to the output stream as is

flex

Multiple scanners

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



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

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

Let's implement a C99-style comment eater:

```
%x COMMENT
%option noyywrap

%{
  #define MAX_DEPTH 10

  int nest = 0;
  int caller[MAX_DEPTH];
%}

%%

<INITIAL>[^/]*       { ECHO; }
<INITIAL>"/"+[^*/]*  { ECHO; }
<INITIAL>"/*"        {
                       caller[nest++] = YY_START;
                       BEGIN(COMMENT);
                     }
```

flex

Multiple scanners: example

```
<COMMENT>[^/*]*
<COMMENT>"/"+[^*/]*
<COMMENT>"/*"         {
                        caller[nest++] = YY_START;
                        BEGIN(COMMENT);
                      }
<COMMENT>"*"+[^*/]*
<COMMENT>"*"+"/"      { BEGIN ( caller [-- nest ]); }

%%

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

Clean Regular Expressions

Regular expression can describe **`simple`** concepts:

- complex structures are typically described by cryptic regular expression

Even with simple concepts is better to keep the regular expression as clean as possible:

- they becomes unreadable very quickly

Exploit tool features to simplify regular expressions (e.g. definitions).