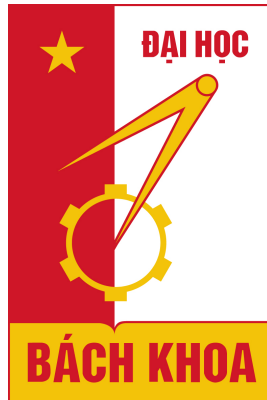


TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI



Báo cáo bài tập lớn môn học:
Lý thuyết ngôn ngữ và phương pháp dịch

TÌM HIỂU BỘ SINH
TRÌNH PHÂN TÍCH TỪ VỰNG FLEX

Sinh viên thực hiện:

Tạ Quang Tùng
MSSV: 20154280

Giáo viên hướng dẫn:

TS. Phạm Đăng Hải

Hà Nội, Ngày 25 tháng 10 năm 2018

Mục lục

1	Giới thiệu về bộ sinh trình phân tích từ vựng Flex	1
2	Cấu trúc của file chỉ dẫn trong Flex	1
2.1	Vùng định nghĩa - Definitions Section	1
2.2	Vùng các luật - Rules Section	2
2.3	Vùng mã nguồn bổ sung - User Code Section	3
3	Các điều kiện bắt đầu (start conditions)	3
4	Cách sử dụng FLex	4
4.1	Hoạt động của Flex	4
4.2	Ví dụ một file đầu vào Flex đơn giản	5
4.3	Sử dụng Flex bằng dòng lệnh	6
5	Sử dụng Flex để tạo bộ phân tích từ vựng cho ngôn ngữ PL/0 mở rộng	6
5.1	File chỉ dẫn đầu vào	6
5.2	Chạy thử nghiệm	9

1 Giới thiệu về bộ sinh trình phân tích từ vựng Flex

Flex là bộ sinh chương trình phân tích từ vựng, là một phần mềm miễn phí và mã nguồn mở thay thế cho Lex. [3] Nó thường xuyên được sử dụng đi kèm với bộ sinh chương trình phân tích cú pháp Yacc (hoặc mã nguồn mở thay thế là GNU Bison) để tạo nên 2 khối chức năng cơ bản của một chương trình dịch.

Flex được viết bằng ngôn ngữ C bởi Vern Paxson vào khoảng năm 1987. Flex nhận đầu vào là một file chỉ dẫn, thông thường có đuôi là `.l`. Từ đó Flex có thể sinh ra mã C có thể biên dịch và thực thi mà không cần thêm bất kỳ thư viện ngoài nào.

Mã C sinh ra bởi Flex sử dụng Automata hữu hạn đơn định (Deterministic Finite Automation - DFA) để thực hiện việc tách xâu đầu vào thành các từ tố tương ứng. Thuật toán thường có độ phức tạp thời gian tính là $O(n)$ với n là độ dài xâu đầu vào.

Flex chỉ có thể sinh ra mã code C hoặc C++.

2 Cấu trúc của file chỉ dẫn trong Flex

Mỗi file chỉ dẫn đầu vào cho Flex bao gồm 3 vùng, được phân tách nhau bởi các dòng chỉ chứa xâu `'%%':` [2]

```
|| definitions
|| %%
|| rules
|| %%
|| user code
```

Trong đó:

- **definitions:** Vùng chứa các định nghĩa.
- **rules:** Vùng chứa luật của các từ tố.
- **user code:** Vùng chứa mã C/C++ được thêm vào.

2.1 Vùng định nghĩa - Definitions Section

Vùng định nghĩa chứa các định nghĩa tên và các định nghĩa điều kiện bắt đầu. Định nghĩa tên có dạng:

```
|| name definition
```

'name' là các tên có cú pháp giống như các tên trong ngôn ngữ C. 'definition' là phần được lấy từ kí tự không phải kí tự trắng đầu tiên sau 'name' và đến hết dòng. 'name' không được thụt lề đầu dòng. Các định nghĩa có thể được tham chiếu đến bằng cách sử dụng '{name}' và nó sẽ được chuyển thành '(definition)'.

Ví dụ:

```
|| DIGIT [0-9]
```

Thì '{DIGIT}' sẽ được chuyển thành '([0-9])'.

Bất kì một dòng nào trong vùng này được viết thụt lề đầu dòng hoặc được bao xung quanh bởi `%{` và `%}` thì sẽ được sao chép nguyên văn vào file mã nguồn sinh ra.

Ví dụ:

```
||      int number , count ;
||      %{
||      #include <math.h>
||      %}
```

Thì đoạn code sau sẽ được sao chép nguyên văn vào file đầu ra:

```
||      int number , count ;
||      #include <math.h>
```

2.2 Vùng các luật - Rules Section

Là vùng bao gồm một chuỗi các luật có dạng:

```
|| pattern action
```

Trong đó, **pattern** không được phép thụt lề đầu dòng. Và phần **action** phải được bắt đầu trên cùng một dòng. Các **action** là các lệnh C/C++ đơn hoặc lệnh C/C++ khối nếu sử dụng `{}`. Khi sử dụng lệnh khối ta có thể viết trên nhiều dòng.

Ở trong vùng các luật, bất kì một dòng nào mà có thụt lề đầu dòng hoặc các dòng được bao bởi `%{ %}` nằm trước luật đầu tiên có thể được sử dụng để khai báo những biến địa phương cho hàm phân tích từ tổ. Và các đoạn code này sẽ được thực thi mỗi khi hàm phân tích từ tổ được gọi.

Các **pattern** là các biểu thức chính quy phiên bản mở rộng, bao gồm:

- **x**
Nhận diện được kí tự 'x'.
- **.**
Bất kì cú kí tự nào ngoại trừ dấu xuống dòng.
- **[xyz]**
Là một "Lớp các kí tự". Nhận diện được một trong các kí tự: 'x', 'y', 'z'.
- **[abj-oZ]**
Là một "Lớp các kí tự". Nhận diện được kí tự 'a', 'b', bất kì kí tự nào trong khoảng từ 'j' đến 'o', hoặc kí tự 'Z'.
- **[^A-Z\n]**
Là một "Lớp phủ định các kí tự". Nhận diện được bất kì kí tự nào không nằm trong khoảng được chỉ định. Ở đây, khoảng đó là các kí tự từ 'A' tới 'Z' và kí tự xuống dòng.
- **r***
Không hay nhiều **r**. Trong đó **r** là một biểu thức chính quy bất kì.
- **r+**
Một hay nhiều **r**. Trong đó **r** là một biểu thức chính quy bất kì.

- **r?**
Không hay một **r**. Trong đó **r** là một biểu thức chính quy bất kì.
- **r{2, 5}**
r được xuất hiện liên tiếp từ 2 đến 5 lần. Trong đó **r** là một biểu thức chính quy bất kì.
- **r{3..}**
r được xuất hiện từ 3 lần trở lên. Trong đó **r** là một biểu thức chính quy bất kì.
- **r{4}**
r được xuất hiện chính xác 4 lần. Trong đó **r** là một biểu thức chính quy bất kì.
- **{name}**
Thay thế định nghĩa 'name' thành 'definition' đã được chỉ định ở vùng định nghĩa.
- **rs**
Phép nối biểu thức chính quy.
- **r|s**
Phép hợp biểu thức chính quy.
- **^r**
Nhận diện các xâu từ **r** nhưng phải bắt đầu dòng.
- **r\$**
Nhận diện các xâu từ **r** nhưng phải kết thúc dòng.
- **(?r-s:pattern)**
Áp dụng tùy chọn **r**, loại bỏ tùy chọn **s** lên biểu thức chính quy **pattern**. Ví dụ tùy chọn **i** để không phân biệt hoa và thường (case-insensitive).

2.3 Vùng mã nguồn bổ sung - User Code Section

Các dòng code trong vùng này đơn giản được sao chép vào file mã nguồn đầu ra. Vùng này có thể không có, có thể bỏ '%%' cuối cùng nếu vùng này trống rỗng.

3 Các điều kiện bắt đầu (start conditions)

Flex cung cấp cơ chế cho việc kích hoạt các luật dưới những điều kiện cụ thể. Bất kì một luật nào được bắt đầu bằng '<sc>' sẽ chỉ được kích hoạt khi trình phân tích từ tổ (scanner) ở trong một "điều kiện bắt đầu" cụ thể, ở đây là **sc**. Điều kiện bắt đầu có thể hiểu đơn giản là những trạng thái mà scanner có thể chuyển tới, những trạng thái này quyết định những luật (rules) nào được kích hoạt tại thời điểm đó. Ví dụ:

```
|| <STRING>[a-zA-Z]*           do_this();
|| <INITIAL, STRING, QUOTE>"abc" do_that();
```

Trong đó **STRING**, **INITIAL**, **QUOTE** là các điều kiện bắt đầu.

INITIAL là điều kiện bắt đầu (hay trạng thái) ban đầu của scanner. Luật **[a-zA-Z]*** chỉ hoạt động khi điều kiện bắt đầu là **STRING**. Luật **"abc"** chỉ hoạt động khi điều kiện bắt đầu là một

trong ba khả năng: **STRING**, **INITIAL** và **QUOTE**. Nếu ta không chỉ định điều kiện bắt đầu thì mặc định sẽ là **INITIAL**

```
|| ABCD[0-9] do_something();
```

Tương đương với:

```
|| <INITIAL>ABCD[0-9] do_something();
```

Để sử dụng được các điều kiện bắt đầu, trước tiên ta phải khai báo chúng, có hai cách khai báo:

```
|| %s EXAMPLE1  
|| %x EXAMPLE2
```

Các khai báo thứ nhất: '**%s**' được gọi là kiểu khai báo 'inclusive'.

Các khai báo thứ hai: '**%x**' được gọi là kiểu khai báo 'exclusive'.

Điểm khác nhau giữa chúng là: Những luật không bắt đầu bởi điều kiện bắt đầu sẽ mặc định bao gồm **INITIAL** và các điều kiện bắt đầu 'inclusive', ví dụ:

```
|| "xyz" something();
```

Tương đương với:

```
|| <INITIAL, EXAMPLE1>"xyz" something();
```

Hay những điều kiện bắt đầu 'exclusive' sẽ không được thêm vào danh sách mặc định.

Ta có thể chuyển đổi điều kiện bắt đầu bằng cách gọi hàm:

```
|| BEGIN(some_start_condition)
```

Trong các **action**.

Trong báo cáo này sẽ sử dụng các điều kiện bắt đầu để loại bỏ các comment (một dòng và nhiều dòng) trong ngôn ngữ PL/0 mở rộng. Ví dụ cho comment một dòng:

```
|| %x COMMENT  
|| %%  
|| "//" BEGIN(COMMENT);  
|| <COMMENT>"\n" BEGIN(INITIAL);  
|| <COMMENT>. ;  
|| %%  
|| ...
```

4 Cách sử dụng FLex

4.1 Hoạt động của Flex

Flex sẽ phân tích file chỉ dẫn đầu vào rồi tạo một DFA tương ứng. DFA này được mã hóa trong mã nguồn C/C++. DFA này sẽ ưu tiên những **pattern** có độ dài càng lớn.

Chương trình sinh ra bởi Flex sẽ mặc định đọc dữ liệu từ **stdin**. Ta có thể sử dụng hàm **void yyrestart(FILE *file)** để thiết lập con trỏ file mà chương sẽ sử dụng để đọc file đầu vào.

Chương trình sẽ bắt đầu quá trình nhận diện xâu khi hàm **int yylex()**; được gọi. Khi đó, nếu một xâu được nhận diện bởi một luật nào đó, mã nguồn C/C++ tương ứng với **pattern** được thực thi. Nếu đoạn mã đấy không có **return**, chương trình sẽ tiếp tục phân tích từ tổ tiếp theo [4]. Nếu có **return**, giá trị từ tổ (hay giá trị **return**) được trả về qua giá trị trả về của hàm **yylex()**.

Xâu được nhận diện sẽ nằm trong biến toàn cục **yytext**, độ dài của xâu sẽ nằm trong biến toàn cục **yylen**.

Nếu không một **pattern** nào nhận diện được xâu đầu vào, **luật mặc định** sẽ được thực thi: Ký tự tiếp theo trong file đầu vào sẽ được coi như là nhận diện được, và sẽ được in ra màn hình.

Kích thước giới hạn của **yytext** được định nghĩa bởi hằng **YYLMAX**, là một số khá lớn. Ta có thể thay đổi nó đơn giản bởi định nghĩa lại: **#define YYLMAX <number>**.

Hơn nữa, để có thể biên dịch được chương trình, ta cần phải định nghĩa hàm **yywrap**. Hàm **int yywrap()** được gọi khi đầu vào đã được đọc hết. Nếu hàm trả về giá trị 1, quá trình phân tích kết thúc, trả về 0 nếu như việc xử lý vẫn còn tiếp tục. Ta có thể thay đổi biến toàn cục **yyin** trỏ đến một file khác (hoặc sử dụng **yyrestart**) và chương trình sẽ tiếp tục phân tích từ file mới đó. [1]

Hoặc đơn giản ta có thể định nghĩa:

```
int yywrap() {  
    return 1;  
}
```

Trong vùng **User Code Section** của file chỉ thị.

4.2 Ví dụ một file đầu vào Flex đơn giản

```
int num_lines = 0, num_chars = 0;  
%%  
\n    { ++num_lines; ++num_chars; return 1; }  
.    { ++num_chars; return 2; }  
%%  
  
int yywrap() {  
    return 1;  
}  
  
int main() {  
    while (yylex() != 0) {}  
    printf("So dòng: %d, so ki tu: %d\n", num_lines, num_chars);  
}
```

Bắt đầu file là việc viết một dòng mã C để khai báo 2 biến **num_lines** và **num_chars** gán giá trị mặc định bằng 0.

Vùng định nghĩa các luật (vùng thứ 2) bao gồm 2 luật:

- **\n**

Là luật nhận diện ký tự xuống dòng. Phần **action** thực hiện tăng giá trị 2 biến, rồi trả về 1. Giá trị trả về này sẽ là giá trị trả về của hàm **yylex()**.

- **.**

Là luật nhận diện ký tự bất kì. Phần **action** thực hiện tăng giá trị của **num_chars**, rồi trả về 2.

Vùng mã nguồn bổ sung bao gồm định nghĩa 2 hàm:

- **int yywrap()**

Chỉ đơn giản trả về 1, chỉ định rằng sau khi kết thúc đọc dữ liệu đầu vào file (hoặc từ bàn phím) sẽ kết thúc luôn việc phân tích từ tổ (hay sẽ không có dữ liệu tiếp theo nào nữa).

- **int main()**

Gọi hàm **yylex()** liên tục để phân tích từng từ tổ cho đến khi không thể phân tích thêm (giá trị trả về là 0). Và thực hiện in ra số dòng và số kí tự.

4.3 Sử dụng Flex bằng dòng lệnh

Một file chỉ thị đầu vào cho Flex có đuôi thông thường là **.l**, ví dụ **scanner.l**. Để biên dịch file này sang file C/C++, ta sử dụng:

```
$ flex scanner.l
```

Mặc định lệnh trên sẽ sinh ra một file **lex.yy.c**. Ta có thể dịch file này bằng một trình dịch C thông thường, ví dụ **gcc**:

```
$ gcc lex.yy.c -o main
```

Ta có thể chỉ định tên file xuất ra bởi flex bằng cách sử dụng tùy chọn **-o <filename>**:

```
$ flex -o scanner.lex.cpp scanner.l
```

5 Sử dụng Flex để tạo bộ phân tích từ vựng cho ngôn ngữ PL/0 mở rộng

5.1 File chỉ dẫn đầu vào

scanner.l

```
%{  
#include <string.h>  
#include <ctype.h>  
  
enum Token {  
    TOKEN_IDENT = 1,  
    TOKEN_NUMBER,  
  
    TOKEN_BEGIN, TOKEN_CALL, TOKEN_CONST, TOKEN_DO,  
    TOKEN_ELSE, TOKEN_END,  
    TOKEN_FOR, TOKEN_IF, TOKEN_ODD,  
    TOKEN_PROCEDURE, TOKEN_FUNCTION,  
    TOKEN_PROGRAM, TOKEN_THEN, TOKEN_TO,  
    TOKEN_VAR, TOKEN_WHILE,  
  
    TOKEN_PLUS, TOKEN_MINUS, TOKEN_TIMES, TOKEN_DIVIDE, TOKEN_REMAINDER,  
  
    TOKEN_EQ, TOKEN_NE, TOKEN_LT, TOKEN_GT, TOKEN_LE, TOKEN_GE,  
  
    TOKEN_LPARENT, TOKEN_RPARENT,  
    TOKEN_LBRACKET, TOKEN_RBRACKET,  
  
    TOKEN_PERIOD, TOKEN_COMMA, TOKEN_SEMICOLON, TOKEN_ASSIGN,  
};
```



```

#define IDENT_NAME_SIZE 10
#define MAX_DIGIT_COUNT 9

%}

DIGIT      [0-9]
ALPHA      [A-Za-z]
ALPHADIGIT [A-Za-z0-9]
WHITESPACE [ \n\t\r]

%x COMMENT_SINGLE
%x COMMENT_MULTIPLE

%%

<INITIAL>"//"      BEGIN(COMMENT_SINGLE);
<INITIAL>"/*"      BEGIN(COMMENT_MULTIPLE);
<COMMENT_SINGLE,COMMENT_MULTIPLE>.\n    ;
<COMMENT_SINGLE>"\n"  BEGIN(INITIAL);
<COMMENT_MULTIPLE>"\n" ;
<COMMENT_MULTIPLE>"*/" BEGIN(INITIAL);

(?i:begin)      return  TOKEN_BEGIN;
(?i:call)       return  TOKEN_CALL;
(?i:const)      return  TOKEN_CONST;
(?i:do)         return  TOKEN_DO;
(?i:else)       return  TOKEN_ELSE;
(?i:end)        return  TOKEN_END;

(?i:for)        return  TOKEN_FOR;
(?i:if)         return  TOKEN_IF;
(?i:odd)        return  TOKEN_ODD;
(?i:procedure)  return  TOKEN_PROCEDURE;
(?i:function)   return  TOKEN_FUNCTION;
(?i:program)    return  TOKEN_PROGRAM;
(?i:then)       return  TOKEN_THEN;
(?i:to)         return  TOKEN_TO;
(?i:var)        return  TOKEN_VAR;
(?i:while)      return  TOKEN_WHILE;

"+"           return  TOKEN_PLUS;
"_"           return  TOKEN_MINUS;
"*"           return  TOKEN_TIMES;
"/"           return  TOKEN_DIVIDE;
"%"           return  TOKEN_REMAINDER;

"="           return  TOKEN_EQ;
"<>"          return  TOKEN_NE;
"<"           return  TOKEN_LT;
">"           return  TOKEN_GT;
"<="          return  TOKEN_LE;
">="          return  TOKEN_GE;

"("           return  TOKEN_LPARENT;
")"           return  TOKEN_RPARENT;
"["           return  TOKEN_LBRACKET;

```

```

"]"                return  TOKEN_RBRACKET;

"."                return  TOKEN_PERIOD;
","                return  TOKEN_COMMA;
";"                return  TOKEN_SEMICOLON;
":="               return  TOKEN_ASSIGN;

{ALPHA}{ALPHADIGIT}*  return  TOKEN_IDENT;
{DIGIT}+              return  TOKEN_NUMBER;
{WHITESPACE}          ;

. {
    printf("Khong nhan dien duoc ki tu: %s\n", yytext);
    exit(-1);
}

%%

int yywrap() {
    return 1;
}

int main(int argc, char **argv) {
    if (argc <= 1) {
        printf("Thieu ten file\n");
        return -1;
    }

    FILE *file = fopen(argv[1], "rb");
    if (file == NULL) {
        printf("Khong the doc file %s\n", argv[1]);
        return -1;
    }
    yyrestart(file);

    int token;
    while (token = yylex()) {
        if (token == TOKEN_IDENT) {
            char str[256];
            // Lay xau con cua xau yytext
            strncpy(str, yytext, IDENT_NAME_SIZE);
            str[IDENT_NAME_SIZE] = '\0';
            printf("Token: %s\n", str);
        }
        else if (token == TOKEN_NUMBER) {
            int i = 0;
            for (; i < yyleng; i++)
                if (yytext[i] != '0')
                    break;
            if (yyleng - i > MAX_DIGIT_COUNT) {
                printf("So vuot qua 9 chu so!!\n");
                exit(-1);
            }
            int num = atoi(yytext + i);
            printf("Number: %d\n", num);
        }
    }
}

```

```

        else {
            for (int i = 0; i < yyleng; i++)
                yytext[i] = tolower(yytext[i]);
            printf("%s\n", yytext);
        }
    }

    fclose(file);
    return 0;
}

```

5.2 Chạy thử nghiệm

Biên dịch:

```

$ flex scanner.l
$ gcc lex.yy.c -o main

```

File **input** đầu vào:

```

tungtaquangfjalskdfj kstn PR0gram
// hhhhhhhhhhhhhhhhhhh
/*
ta quang tung mssv: 20154280
wwwwwwwwwwwwwww
*/
begin
(([]):= <> >= =
1231241
1234567890

```

Kết quả chạy:

```

$ ./main input
Token: tungtaquan
Token: kstn
program
begin
(
(
)
[
]
)
:=
<>
>=
=

```

Number: 1231241

So vuot qua 9 chu so!!

Tài liệu tham khảo

- [1] The Flex Manual Page, Mar 2006. [Online; accessed 25. Oct. 2018].
- [2] Lexical Analysis With Flex, for Flex 2.6.2: Top, Oct 2016.
- [3] Flex (lexical analyser generator) - Wikipedia, Oct 2018.
- [4] IBM Knowledge Center - Using `yylex()`, Oct 2018.