

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

-----*-----



Môn học

Lí thuyết ngôn ngữ và phương pháp dịch

Tên đề tài

Tìm hiểu về Bison

Giảng viên hướng dẫn: *TS Phạm Đăng Hải*

Nhóm sinh viên thực hiện: *Phạm Minh Tâm, Lâm Xuân Thu*

Lớp: KSTN CNTT K60

Hà Nội, Ngày 21 tháng 11 năm 2018

Mục lục

Chương 1. Tổng quan	3
1.1 Bison là gì?.....	3
1.2 Ngôn ngữ và văn phạm phi ngữ cảnh	3
1.3 Cú pháp của Bison – đầu vào của Bison.....	4
1.4 Giá trị ngữ nghĩa	4
1.5 Semantic Actions	5
1.6 Đầu ra của Bison	5
1.7 Các giai đoạn của quá trình sử dụng Bison	6
Chương 2. Cấu trúc file văn phạm trong bison.....	7
2.1 Prologue	7
2.2 Bison declarations	8
2.2.1 Khai báo phiên bản:	8
2.2.2. Khai báo từ tổ.....	8
2.2.3 Khai báo kiểu dữ liệu	9
2.2.3 Khai báo ký tự không kết thúc:.....	9
2.2.4 Khai báo ký tự bắt đầu	10
2.2.5 Khai báo mức độ ưu tiên.....	10
2.3 Grammar Rules	11
2.4 Epilogue	12
2.5 Các ký tự đặc biệt trong chương trình Bison	13
Chương 3. Kết hợp giữa Flex và Bison	14
Chương 4. Demo chương trình đơn giản	15
4.1 Chương trình tính phép cộng	15
4.2 Văn phạm tính biểu thức đơn giản.....	17
Tài liệu tham khảo.....	19

Chương 1. Tổng quan

1.1 Bison là gì?

Bison là một bộ sinh chương trình dịch được viết bởi Robert Corbett và sau đó được phát triển bởi nhiều chuyên gia khác. Bison có thể dùng để phát triển phạm vi rộng lớn các trình dịch ngôn ngữ, từ những bộ tính toán đơn giản cho đến những ngôn ngữ lập trình phức tạp. Để làm việc với Bison, người lập trình viên cần thành thạo ngôn ngữ C/C++.

1.2 Ngôn ngữ và văn phạm phi ngữ cảnh

Để Bison thực hiện dịch một ngôn ngữ thì nó phải được mô tả bởi một văn phạm phi ngữ cảnh. Tức là ta phải đưa ra một hoặc nhiều nhóm cú pháp (*syntactic groupings*) và các luật để cấu trúc chúng. Ví dụ, trong ngôn ngữ C, một loại grouping là “expression”. Một luật để tạo một ‘expression’ có thể là: “Một expression được tạo bởi một dấu trừ và một expression khác”. Một luật khác là: “Một biểu thức có thể là một số nguyên”. Các luật có thể có tính chất đệ quy, nhưng bắt buộc phải có ít nhất một luật giúp thoát ra khỏi lặp đệ quy.

Hệ thống phổ biến nhất để biểu diễn các luật như trên để con người có thể đọc được là *Backus-Naur Form* hay “BNF”. Đầu vào của Bison bản chất là BNF.

Có nhiều lớp con quan trọng khác nhau của văn phạm phi ngữ cảnh. Mặc dù có thể làm việc với gần như tất cả văn phạm phi ngữ cảnh, Bison được sử dụng tối ưu đối với văn phạm LR(1).

1.3 Cú pháp của Bison – đầu vào của Bison

Để định nghĩa ngôn ngữ cho Bison, ta phải viết một file mô tả văn phạm theo cú pháp của Bison: *Bison grammar file*.

Một kí hiệu không kết thúc trong văn phạm được biểu diễn trong file đầu vào của Bison như một định danh, tương tự như một định danh trong C. Theo quy ước, nó được viết dưới dạng chữ thường, ví dụ như *expr*, *stm*, *declaration*.

Kí hiệu kết thúc trong Bison được biểu diễn dưới dạng gọi là Token Type. Theo quy ước, các định danh này viết dưới dạng chữ in hoa để phân biệt với kí hiệu không kết thúc, ví dụ như: *INTEGER*, *IDENTIFIER*, *IF*, *RETURN*. Một kí hiệu kết thúc tương ứng với một từ khoá cụ thể trong ngôn ngữ nên được đặt tên sau khi từ khoá được chuyển về dạng chữ in hoa. Kí hiệu kết thúc *error* được dành riêng cho việc phục hồi lỗi.

Nếu token là một kí tự đơn (dấu ngoặc, dấu cộng, chấm phẩy, ...) thì kí hiệu kết thúc có thể được biểu diễn bằng đúng kí tự đó.

Các luật của văn phạm được biểu diễn dạng các biểu thức trong cú pháp của Bison. Ví dụ đây là luật Bison cho một mệnh đề *return* trong C:

stmt : *RETURN expr ';' ;*

1.4 Giá trị ngữ nghĩa

Một văn phạm thông thường chọn các token chỉ dựa trên phân loại của chúng, ví dụ: một luật đề cập đến “hằng số nguyên”, có nghĩa là bất kì hằng số số nguyên nào đều thoả mãn hợp lệ ở vị trí này. Giá trị chính xác của hằng số không liên quan đến việc input sẽ được dịch như thế nào, ví dụ: nếu “*x + 1*” thoả mãn ngữ pháp hợp lệ thì “*x + 4*” hay “*x + 1997*” cũng thoả mãn ngữ pháp hợp lệ.

Tuy nhiên giá trị chính xác cụ thể là rất quan trọng đối với việc input có ý nghĩa như thế nào khi nó được biên dịch. Một trình dịch sẽ ít hữu dụng nếu nó không thể phân biệt giữa 1, 4, và 1997 trong chương trình. Vì điều đó mà mỗi token trong một ngữ pháp Bison vừa có cả kiểu token và giá trị ngữ nghĩa.

Token type là một kí hiệu không kết thúc được định nghĩa trong văn phạm, như *INTEGER*, *IDENTIFIER* hay ‘;’. Nó giúp ta biết được token có xuất hiện hợp lệ và cách ta nhóm nó với các token khác. Các luật của văn phạm không biết gì khác về ngoài token type.

Giá trị ngữ nghĩa chứa toàn bộ thông tin còn lại của token, như giá trị của một số nguyên, hoặc tên của một định danh.

Ví dụ: một token được phân loại là *INTEGER*, và có giá trị ngữ nghĩa là 4. Một token khác cũng có kiểu token là *INTEGER* nhưng có giá trị là 1997.

1.5 Semantic Actions

Một chương trình hữu ích ngoài việc biên dịch input cần phải sản xuất ra output nào đó dựa trên input. Trong cú pháp của Bison, một luật có thể có một *action* được tạo bởi các câu lệnh C. Mỗi khi trình dịch nhận diện khớp với một luật, *action* sẽ được thực thi.

Hầu hết các action có mục đích là tính toán giá trị ngữ nghĩa cho cấu trúc tổng thể từ các giá trị ngữ nghĩa của các thành phần của nó. Ví dụ giả sử ta có một luật nói rằng một biểu thức có thể là tổng của hai biểu thức. Luật mô tả sẽ như sau:

$expr: expr '+' expr \{ \$\$ = \$1 + \$3; \};$
--

1.6 Đầu ra của Bison

Khi chạy Bison ta phải đưa đầu vào cho nó là một file Bison grammar. Đầu ra quan trọng nhất là một file source C, mà file C này thực thi một bộ phân tích cú pháp cho ngôn ngữ được miêu tả bởi grammar. Bộ phân tích được gọi là một *Bison parser*, và file này được gọi là *Bison parser implementation file*.

Công việc của Bison parser là nhóm các token lại theo các luật của văn phạm, ví dụ nhóm các identifier và các toán tử để tạo thành một biểu thức. Đồng thời nó thực thi action tương ứng với luật được sử dụng.

Các token được lấy từ một hàm *lexical analyzer* mà ta phải định nghĩa theo cách nào đó (như viết bằng code C). Bộ Bison parser gọi đến *lexical analyzer* mỗi khi nó cần một token mới.

1.7 Các giai đoạn của quá trình sử dụng Bison

Có bốn phần chính:

1. Chỉ định văn phạm dưới dạng có thể đọc được bởi Bison (Bison grammar files). Với mỗi luật của văn phạm, miêu tả action sẽ được thực thi khi bộ phân tích nhận dạng được một thực thể của luật đó. Action được viết bởi các câu lệnh C.
2. Viết một bộ nhận diện từ tổ để đọc đầu vào và trả về các token cho bộ phân tích. Bộ nhận diện từ tổ có thể được viết thủ công bằng code C, hoặc có thể được sinh bằng cách sử dụng Lex.
3. Viết một hàm điều khiển gọi Bison parser.
4. Viết các thủ tục báo lỗi.

Chương 2. Cấu trúc file văn phạm trong bison

Chương trình bison có dạng:

```
%{  
Prologue  
%}  
Bison declarations  
%%  
Grammar Rules  
%%  
Epilogue
```

Trong đó phần:

Prologue: chứa các khai báo hàm, include các thư viện C của chương trình

Bison declarations: chứa các khai báo các từ tổ, kiểu được sử dụng trong bison

Grammar Rules: định nghĩa các luật trong văn phạm cần biểu diễn và các hàm thực thi tương ứng

Epilogue: Chứa định nghĩa các hàm C

2.1 Prologue

Chứa các khai báo hàm và include thư viện C cần thiết khi sử dụng trong chương trình.

Ví dụ:

```
%{
```

```
#include "stdio.h"
int yyerror(char *s);
int yylex(void);
%}
```

Đoạn trên sử dụng thư viện `stdio.h` của c và khai báo header hai hàm sẽ được định nghĩa ở phần sau của chương trình là `yyerror` và `yylex`.

2.2 Bison declarations

Định nghĩa các từ tổ, kiểu dữ liệu ứng với giá trị của các từ tổ đó, các ký tự không kết thúc và ký tự bắt đầu trong ngôn ngữ. Tất cả các từ tổ phải được khai báo trước khi có thể đưa vào sử dụng.

Các khai báo trong phần declarations bao gồm:

2.2.1 Khai báo phiên bản:

Khai báo phiên bản bison nhỏ nhất để biên dịch. Nếu số phiên bản không thỏa mãn, chương trình sẽ thông báo lỗi:

```
%require "version"
```

Có thể bỏ qua phần khai báo phiên bản này trong chương trình.

2.2.2. Khai báo từ tổ

Để chỉ định tên các từ tổ sẽ dùng trong ngôn ngữ.

Cú pháp:

```
%token <type> token_name
```

Với: `token_name` là tên từ tổ

`type` là kiểu dữ liệu ứng với giá trị của các từ tổ đó, có thể có hoặc không

Ví dụ:

```
%token <int> NUMBER
%token PLUS
```


Trong đoạn trên NUMBER là một từ tổ với giá trị của từ tổ là một số nguyên.

PLUS là một từ tổ không có giá trị nên ta bỏ phần khai báo giá trị đi.

Khi dịch bison sẽ chuyển tên các từ tổ token_name ứng với một giá trị nào đó trong file đầu ra. Ta có thể chỉ định giá trị tương ứng với từ tổ đó bằng cú pháp :

```
%token PLUS num_ber
```

Ví dụ:

```
%token PLUS 500
```

Nhưng tốt hơn là để bison tự lựa chọn, chương trình sẽ chọn những số lớn hơn 255 để không xung đột với giá trị của các ký tự trong bảng mã ASCII.

Các token cũng có thể được thể hiện thông qua một chuỗi ký tự:

Ví dụ:

```
%token arrow "=>"
```

2.2.3 Khai báo kiểu dữ liệu

Khai báo tất cả các kiểu dữ liệu ứng với giá trị của các từ tổ và các ký tự không kết thúc sẽ sử dụng.

```
%union{  
    int num;  
    string*ident;  
}
```

Định nghĩa kiểu dữ liệu num thay cho int, ident thay cho string*.

2.2.3 Khai báo ký tự không kết thúc:

Cú pháp:

```
%type <type> nonterminal
```

Với <type> là kiểu dữ liệu ứng với giá trị của ký tự không kết thúc nonterminal.

Kiểu dữ liệu type đã được định nghĩa sẵn trong phần khai báo %union.

Ví dụ:

```
%type <num> EXPR
```

Kiểu dữ liệu ứng với giá trị của ký tự không kết thúc EXPR là num.

2.2.4 Khai báo ký tự bắt đầu

Bison mặc định rằng ký hiệu bắt đầu văn phạm là ký tự không kết thúc đầu tiên bên trái trong phần grammar. Ta có thể khai báo ký hiệu bắt đầu bằng cú pháp

```
%start startsymbol
```

2.2.5 Khai báo mức độ ưu tiên

Cú pháp cho khai báo mức độ ưu tiên như sau:

```
%left <type> symbols ...
```

Trong đó:

%left là phần khai báo cho biết thứ tự kết hợp của toán tử là từ trái sang phải

<type> kiểu dữ liệu ứng với giá trị của từ tố

Symbols là các ký hiệu toán tử

Ngoài các khai báo với %left, chúng ta còn có các khai báo khác đó là %right, %precedence, %noassoc với các ý nghĩa khác nhau. Các chỉ thị này cho chúng ta biết được sự kết hợp và mức độ ưu tiên của các ký hiệu symbols:

- Sự kết hợp của một toán tử: Khi chúng ta có input dưới dạng “x op y op z” với op là một toán tử nào đó, khi đó bộ phân tích cần biết rõ là nên thực hiện x op y trước hay y op z trước. Khi đó sử dụng chỉ thị %left sẽ quy định là sử dụng kết hợp từ trái sang phải (thực hiện x op y trước), ngược lại %right là sử dụng kết hợp từ phải sang trái. Ngoài ra %noassoc qui định là không có sự kết hợp nào được phép, điều đó nghĩa là trong một input sẽ chỉ tồn tại nhiều nhất là một toán tử, hay trong trường hợp này câu “x op y op z” sẽ được thông báo là lỗi cú pháp.
- Thứ tự ưu tiên được quy định như sau: Những kí hiệu được liệt kê trong cùng một khai báo thứ tự ưu tiên sẽ có độ ưu tiên như nhau và được phân chia thứ

tự thực hiện dựa trên tính kết hợp đã được định nghĩa. Còn đối với các token nằm ở các khai báo thứ tự ưu tiên khác nhau thì khai báo nào được chỉ ra sau thì tương ứng các token tương ứng sẽ có độ ưu tiên cao hơn so với các token ở khai báo trước đó.

Ví dụ:

```
%left "+" "-"  
%left "*" "/"
```

Trong ví dụ này do toán tử "*" và "/" được khai báo sau toán tử "+" và "-" nên có độ ưu tiên cao hơn. Tất cả các toán tử +, -, *, / trên đều là toán tử thực hiện từ trái sang phải.

2.3 Grammar Rules

Định nghĩa các luật ngữ pháp của ngôn ngữ cần biên dịch, có dạng tổng quát như sau

```
result: components { action };
```

Trong đó: result là một ký tự không kết thúc mà luật này mô tả

components gồm tập hợp các ký tự kết thúc và không kết thúc

action là tập các câu lệnh c được thực thi khi các luật thỏa mãn, phần này có thể có hoặc không.

Ví dụ với cú pháp công 2 số nguyên

```
EXP: NUMBER PLUS NUMBER{ $$ = $1 + $3; }
```

Ở ví dụ trên ta thấy EXP được định nghĩa là một chuỗi gồm 3 token là NUMBER, PLUS và NUMBER, kết quả trả về của EXP chính là tổng giá trị của 2 token NUMBER

Một ký tự không kết thúc result có nhiều luật có thể được định nghĩa bằng cách kết nối các luật lại với nhau bởi ký tự "|":

```
Result : components-1 { action };
```

		components-2	...
		components-3	...
		
		;	

Ví dụ:

A	:	Aa b
		b;

Định nghĩa một cấu trúc ngữ pháp dạng $E \rightarrow E+T \mid T$.

Một luật cũng có thể là trống rỗng nếu như ngữ pháp có một sản xuất epsilon.

Ví dụ:

A	:	bB;
B	:	aB
		;

Định nghĩa văn phạm dạng $A \rightarrow bB$; $B \rightarrow aB \mid \epsilon$

Ta có thể sử dụng **%empty** để định nghĩa một luật rỗng:

A	:	bB;
B	:	aB
		%empty;

2.4 Epilogue

Phần Epilogue được sao chép nguyên vẹn vào cuối file mã nguồn khi tiến hành dịch bison. Phần này sẽ định nghĩa các hàm đã được khai báo ở phần prologue.

Nếu phần epilogue trống rỗng ta có thể bỏ đi.

2.5 Các ký tự đặc biệt trong chương trình Bison

Các ký tự đặc biệt dùng trong bison để định nghĩa các từ khóa và các và các cấu trúc trong bison.

%	Tất cả các khai báo ở trong phần declaration đều được bắt đầu bằng % như <i>%token</i> , <i>%start</i>
\$	Tham chiếu tới một giá trị của từ tổ trong văn phạm được khai báo. Ví dụ: <i>\$3</i> tham chiếu tới giá trị của từ tổ thứ 3 trong câu văn phạm
@	Chỉ định một tham chiếu vị trí
'	Chỉ định các ký tự như là một từ tổ trong văn phạm
{ }	Khối mã C trong chương trình
	Định nghĩa luật với nhiều sản xuất

Chương 3. Kết hợp giữa Flex và Bison

Kết quả Flex được sử dụng như dữ liệu đầu vào của chương trình Bison.

Flex sẽ đọc file input của chương trình, và trả về chuỗi các từ tổ tương ứng. Bison sử dụng tập các từ tổ đó để kiểm tra cú pháp.

Flex sẽ chạy hàm `yylex` để nhận diện các từ tổ và trả về giá trị tương ứng.

Bison chạy hàm `yyparse` để đọc các từ tổ và kiểm tra ngữ pháp của chuỗi từ tổ nhận vào. Nếu tồn tại các action tương ứng ta đã định nghĩa trong phần grammar rule, chương trình sẽ thực thi các action này. Ta cũng có thể chỉ định cho hàm `yyparse` trả về ngay lập tức mà không đọc thêm từ tổ nào nữa:

```
int parse(void);
```

Hàm trả về 0 nếu quá trình phân tích cú pháp thành công.

1 nếu gặp lỗi do đầu vào không hợp lệ

2 nếu quá trình phân tích bị tràn bộ nhớ

Khi xuất hiện một lỗi cú pháp do đọc một token mà không có sản xuất nào thỏa mãn hoặc có một action sử dụng macro `yyerror` để báo lỗi hàm `yyerror` được chạy. Hàm `yyerror` là một hàm với đầu vào là một xâu thông báo lỗi.

```
void yyerror(const char *s);
```

Khi thực hiện chương trình có thể sinh ra một loại lỗi khác là tràn bộ nhớ xảy ra khi mà chuỗi token đầu vào chứa một cấu trúc lồng nhau rất sâu. Nhưng lỗi này thường không xảy ra do parser tạo bởi bison sẽ tự động mở rộng kích thước stack khi nó chạm tới ngưỡng.

Chương 4. Demo chương trình đơn giản

4.1 Chương trình tính phép cộng

Chương trình với đầu vào là một chuỗi có hai số và dấu "+" ở giữa.

Đầu ra sẽ là kết quả của phép tính.

File flex :

add.l
<pre>%{ #include "add.bison.h" %} DIGIT [0-9] %% {DIGIT}+ {yylval.num = atoi(yytext); return NUMBER;} "+" {return PLUS;} [\n\t\r] ; . {ECHO; printf("Loi!!!\n"); exit(1);} %% int yywrap(){ return 1; }</pre>

File bison:

add.y
<pre>%{</pre>

```

#include<stdio.h>
extern int yylex();
extern void yyerror(char *c);

%}
%union {int num;}
%start EXPR
%token <num> NUMBER
%token PLUS
%type <num> EXPR
%%

EXPR : NUMBER PLUS NUMBER {$$ = $1 + $3; printf("%d\n",$$);
}
;

%%

```

File main.c

```

main.c
#include "add.bison.h"
#include <stdio.h>

void yyerror(char *s){
    printf("%s\n", s);
}
int main(){
    return yyparse();
}

```

Tiến hành dịch chương trình theo các bước sau:

- Sử dụng bison để tạo ra file header và file mã nguồn c bằng câu lệnh:

```
$ bison -d -o add.bison.c add.y
```


Tùy chọn -d sẽ tạo ra file header cùng với file mã nguồn C với tên tương ứng là "add.bison.h". File "add.bison.h" này sẽ được include vào file flex "add.l" và file mã nguồn C chứa hàm main "main.c".

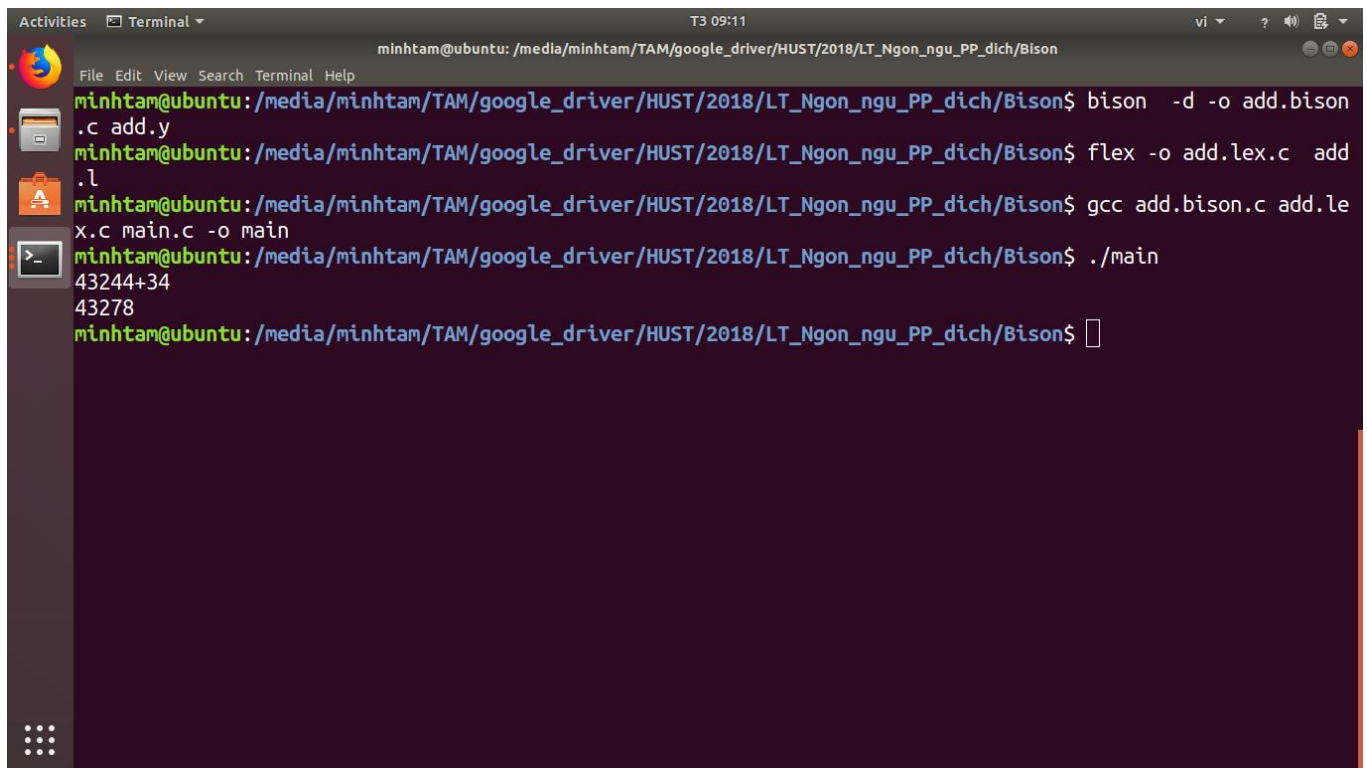
- Sử dụng flex để tạo file mã nguồn lex bằng lệnh:

```
$ flex -o add.lex.c add.l
```

- Dùng gcc để biên dịch file thành file thực thi bằng câu lệnh:

```
$ gcc add.bison.c add.lex.c main.c -o main
```

Chương trình thực thi như hình



```
minhtam@ubuntu: /media/minhtam/TAM/google_driver/HUST/2018/LT_Ngon_ngu_PP_dich/Bison$ bison -d -o add.bison.c add.y
minhtam@ubuntu: /media/minhtam/TAM/google_driver/HUST/2018/LT_Ngon_ngu_PP_dich/Bison$ flex -o add.lex.c add.l
minhtam@ubuntu: /media/minhtam/TAM/google_driver/HUST/2018/LT_Ngon_ngu_PP_dich/Bison$ gcc add.bison.c add.lex.c main.c -o main
minhtam@ubuntu: /media/minhtam/TAM/google_driver/HUST/2018/LT_Ngon_ngu_PP_dich/Bison$ ./main
43244+34
43278
minhtam@ubuntu: /media/minhtam/TAM/google_driver/HUST/2018/LT_Ngon_ngu_PP_dich/Bison$
```

4.2 Văn phạm tính biểu thức đơn giản

```
%{
```

```

#include <stdio.h>
#include <stdlib.h>
int yylex();
void yyerror(const char *s);
%}
%union {
int num;
}
%token <num> NUMBER
%type <num> E T F
%start E
%%

F    : NUMBER { $$ = $1; }
    | '(' E ')' { $$ = $2; }
;
T    : U { $$ = $1; }
    | T '*' U { $$ = $1 * $3; }
;
E    : T { $$ = $1; }
    | E '+' T { $$ = $1 + $3; }
;
%%

```

Văn phạm trên biểu diễn cấu trúc ngữ pháp

```

E -> E+T
E -> T
T -> F
F -> (E)
F -> NUMBER

```

Ngữ pháp trên dùng để đoán nhận các biểu thức toán học chứa dấu +,*,(,) ví dụ như:
(234+43)*96.

Tài liệu tham khảo

[1] <https://www.gnu.org/software/bison/manual/bison.html>

[2] flex & bison, John R. Levine, Tony Mason, Doug Brown , O'Reilly & Associates