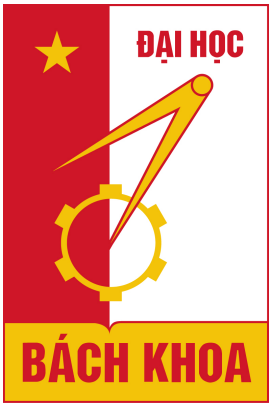
Trường đại học bách khoa hà nội




## Báo cáo bài tập lớn môn học:

# Lý thuyết ngôn ngữ và phương pháp dịch

---

# TÌM HIỂU BỘ SINH TRÌNH PHÂN TÍCH CÚ PHÁP BISON

---

*Sinh viên thực hiện:*
Tạ Quang Tùng
MSSV: 20154280

*Giáo viên hướng dẫn:*
TS. Phạm Đăng Hải

Hà Nội, Ngày 21 tháng 11 năm 2018

# Mục lục

# 1 Giới thiệu về bộ sinh trình phân tích cú pháp Bison

GNU Bison, hay thường được biết đến là Bison, là bộ sinh trình phân tích cú pháp parser, là một phần của dự án GNU. [4]

Bison đọc một file đặc tả của một ngôn ngữ phi ngữ cảnh (context-free language), cảnh báo về những nhập nhằng trong quá trình phân tích, và sinh ra một bộ phân tích cú pháp (Có thể bằng ngôn ngữ C, C++ hoặc Java).

Bison mặc định sinh ra LALR parser, nhưng cũng có thể tạo ra GLR parser.

Trong chế độ POSIX, Bison tương thích với Yacc, cùng với một vài những mở rộng. Đồng thời Flex, bộ sinh trình phân tích từ vựng, cũng thường được sử dụng cùng với Bison để cung cấp chuỗi các token đầu vào cho việc phân tích cú pháp.

Bison là một phần mềm miễn phí, mã nguồn mở được phát hành dưới giấy phép GPL.

# 2 Cấu trúc của file văn phạm trong Bison

## 2.1 Các thành phần của một file văn phạm

Một file mô tả văn phạm của Bison được chia thành 4 phần như sau: [5]

```
%{
    Prologue
%}

Bison declarations

%%
Grammar rules
%%

Epilogue
```

Trong đó, phần Prologue thường chứa các khai báo hàm, include của C.
Phần Bison Declarations chứa các khai báo được Bison sử dụng.
Phần Grammar Rules chứa các quy tắc ngữ pháp của ngôn ngữ đang cần biểu diễn và các lệnh thực hiện tương ứng.
Phần Epilogue thường chứa định nghĩa của các hàm C tương ứng với phần Prologue.

## 2.2 Phần Prologue

Phần Prologue chứa các định nghĩa macro và các khai báo của các hàm và biến được sử dụng trong các câu lệnh của phần Grammar Rules. [8]

Những đoạn code trong phần này sẽ được sao chép vào phần đầu file mã nguồn của chương trình dịch được sinh ra. Có thể sử dụng **#include "..."** để include các file header tương ứng.

Có thể có nhiều hơn một phần Prologue nằm đan xen với các phần Bison Declarations.

## 2.3 Phần Bison Declarations

Phần Bison Declarations định nghĩa các kí hiệu được sử dụng để xây dựng ngữ pháp chương trình và kiểu dữ liệu của các giá trị. [2]

Tất cả token (ngoại trừ các token có một kí tự như '+' hoặc '*') phải được khai báo trước khi có thể sử dụng. Các kí hiệu không kết thúc phải được khai báo nếu muốn định nghĩa kiểu của kí hiệu đó để sử dụng làm giá trị ngữ nghĩa.

Phần Bison Declarations cũng chỉ định kí hiệu không kết thúc bắt đầu của văn phạm. Mặc định là sử dụng kí hiệu bên trái của rule đầu tiên trong phần Grammar Rules.

Các khai báo trong phần Bison Declarations bao gồm:

- **Khai báo require**: Có thể chỉ định version nhỏ nhất của Bison trong khi xử lý file ngữ pháp. Nếu như yêu cầu về version không đủ, Bison dừng việc xử lý và báo lỗi.

```
%require ''version''
```

- **Khai báo token**: Để khai báo tên của token (kí tự kết thúc), ta sử dụng: [12]

```
%token name
```

  Bison sẽ chuyển khai báo token thành một hằng số trong file mã nguồn đầu ra (file header) để mà flex có thể sử dụng **name** để làm giá trị token cần trả về của hàm **yylex()**

  Có thể chỉ định giá trị số của token bằng cú pháp:

```
%token NUM 300
```

  Trong đó 300 là giá trị số của token **NUM**

  Nhưng tốt hơn cả là để Bison lựa chọn giá trị số cho tất cả các token. Bison sẽ tự động chọn các số mà không xung đột với giá trị của các kí tự ASCII và các token trước đó. (Giá trị Bison chọn luôn lớn hơn 255).

- **Khai báo union và kiểu dữ liệu**: Trong hầu hết các chương trình sẽ cần nhiều hơn một kiểu dữ liệu cho các token khác nhau. [7] Như giữa token đại diện cho số và token đại diện cho xâu.

  Mặc định Bison chỉ có một kiểu giá trị ngữ nghĩa là int. Để sử dụng nhiều hơn một kiểu dữ liệu cho các giá trị ngữ nghĩa, Bison yêu cầu trong file ngữ pháp phải có:

  - Chỉ định toàn bộ tập các kiểu dữ liệu sẽ được sử dụng (có thể sử dụng **%union**)
  - Trong các kí hiệu mà giá trị ngữ nghĩa được sử dụng (kết thúc hoặc không kết thúc), Chọn một trong các kiểu được chỉ định ở trên. (Sử dụng khai báo **%token** bổ sung kiểu cho các kí hiệu kết thúc, sử dụng **%type** cho các kí hiệu không kết thúc)

  Để khai báo union, ta dùng cú pháp:

```
%union {
    int num;
    char *name;
}
```

  Để khai báo token có chứa kiểu, ta dùng cú pháp:

```
%token <num> NUMBER
```

  Trong đó token **NUMBER** có kiểu là kiểu của biến **num** trong **union**.

  Để khai báo kí hiệu không kết thúc có chứa kiểu, ta dùng:

```
%type <num> NONTERMINAL
```

Trong đó kí hiệu không kết thúc **NONTERMINAL** có kiểu là kiểu của biến **num** trong **union**. [13]

- **Khai báo kí hiệu bắt đầu văn phạm**: [11] Bison mặc định rằng kí hiệu bắt đầu văn phạm là kí tự không kết thúc đầu tiên bên trái trong Phần Grammar Rules. Có thể thay đổi kí hiệu này bằng:

```
%start symbol
```

Ngoài các phần kể trên, Bison còn hỗ trợ độ ưu tiên các toán tử và thứ tự thực hiện của chúng (từ trái qua phải hoặc ngược lại). Tuy nhiên những điều đó có thể được biểu diễn bằng cách sử dụng văn phạm phù hợp, dựa trên các kĩ thuật khử văn phạm nhập nhằng. Vì vậy trong báo cáo này sẽ không đề cập các phần đó.

## 2.4 Phần Grammar Rules

Phần khai báo ngữ pháp của Bison bao gồm các grammar rule có dạng tổng quát như sau: [10]

```
result: components...;
```

Trong đó **result** là kí hiệu không kết thúc mà rule này mô tả, và **components** là một chuỗi các kí hiệu kết thúc và không kết thúc tạo nên một sản xuất.

Ví dụ:

```
exp: exp '+' exp;
```

Mô tả một sản xuất $exp \to exp + exp$. Các kí tự dấu trắng không ảnh hưởng đến định nghĩa các rule.

Giữa các **component** có thể chứa các **action** dùng để định nghĩa ngữ nghĩa của rule đó. Một **action** có dạng:

```
{C statements}
```

**C statements** là một tập các câu lệnh của C được bao bọc trong cặp ngoặc nhọn, các lệnh này là các lệnh giống như được sử dụng bên trong định nghĩa hàm C. Bison không kiểm tra tính đúng đắn của các lệnh C này, chỉ copy nguyên vẹn vào file mã nguồn đầu ra. Các lệnh này sẽ được chạy khi mà đã xác định cây cú pháp và tiến hành thực hiện phân tích ngữ nghĩa.

Nhiều các **rule** cho cùng một **result** có thể được định nghĩa bằng cách kết nối chúng bằng kí tự '|':

```
result   :   rule1-components...
         |   rule2-components...
         ...
         ;
```

Ví dụ:

```
E    :   E '+' T      { printf(``E\n'');
     |   T            { printf(``T\n'');
     ;
```

Mô tả một sản xuất $E \to E + T|T$.

Một **rule** là trống rỗng nếu như bên phải của nó (các **component**) là trống rỗng. [3] Nó giống như kí hiệu $\varepsilon$ thể hiện một xâu rỗng.

Ví dụ:

```
E    :    TE1
E1   :    '+' T E1
     |
     ;
```

Mô tả hai sản xuất là $E \to TE_1$ và $E_1 \to +TE_1|\varepsilon$

Tuy nhiên việc không sử dụng kí tự gì khiến việc xác định rule trống rỗng khó khăn, ta có thể sử dụng **%empty**:

```
E    :    TE1
E1   :    '+' T E1
     |    %empty
     ;
```

Tuy nhiên **%empty** là một mở rộng của Bison và không tồn tại trên Yacc.

Trong Bison ta nên khai báo các rule (hay các sản xuất) dưới dạng đệ quy trái thay vì đệ quy phải để tránh sử dụng stack trong quá trình hoạt động của parser. [9]

### 2.5 Phần Epilogue

Những đoạn code trong phần Epilogue sẽ được sao chép nguyên vẹn vào cuối của file mã nguồn parser đầu ra, giống như là code trong phần Prologue được sao chép vào phần đầu của file đầu ra. Nếu như phần Epilogue trống rỗng, ta có thể bỏ đi %% kế thúc phần Grammar Rules.

## 3 Các hàm C tương tác với Bison

[6] Chương trình được Bison sinh ra là một chương trình C và hàm thực sự thực hiện quá trình phân tích cú pháp là **yyparse**.

### 3.1 Hàm phân tích cú pháp yyparse

Để quá trình phân tích cú pháp được thực thi, ta gọi hàm **yyparse**. Hàm này sẽ đọc các token, thực thi các **action** trong phần Grammar Rules tương ứng, rồi cuối cùng trả về một giá trị khi gặp kí tự báo kết thúc file hoặc khi không thể phục hồi từ lỗi. Ta cũng có thể chỉ định cho hàm **yyparse** trả về ngay lập tức mà không đọc thêm từ tố nào nữa.

```
int yyparse(void);
```

Hàm trả về 0 nếu như quá trình phân tích cú pháp thành công (hàm trả về do gặp kí tự kết thúc file).
Hàm trả về 1 nếu như gặp lỗi do đầu vào không hợp lệ (Ví dụ: Chuỗi token đầu vào không thỏa mãn cú pháp đã được chỉ định) hoặc do hàm **YYABORT** được gọi.
Hàm trả về 2 nếu như trình phân tích sử dụng hết bộ nhớ cho phép.

Trong một **action** bất kì của phần Grammar Rules, ta có thể chỉ định cho hàm **yyparse** trả về ngay lập tức bằng cách sử dụng 2 macro sau:

- **YYACCEPT**: Trả về ngay lập tức với giá trị 0 (thành công)
- **YYABORT**: Trả về ngay lập tức với giá trị 1 (lỗi)

### 3.2 Hàm phân tích từ tố yylex

Hàm **yylex** là hàm nhận diện các từ tố từ một xâu đầu vào và trả về các giá trị từ tố đó cho parser sử dụng. Bison không tạo hàm này một cách từ động mà ta cần phải tạo nó để hàm **yyparse** có thể gọi nó.

Trong một chương trình đơn giản, ta có thể tạo hàm **yylex** ngay trong phần Epilogue. Nhưng nếu **yylex** được định nghĩa ở một file riêng (ví dụ như dùng Flex để sinh) thì ta cần phải cho phép file đó có thể đọc được định nghĩa của các token (do sử dụng **%token**). Để làm điều đó, ta sử dụng tùy chọn '-d' khi chạy Bison, để cho Bison sẽ ghi những định nghĩa token và các định nghĩa macro cần thiết ra một file header riêng biệt, thông thường là **name.tab.h**.

Quy ước của hàm **yylex**:

```
int yylex(void);
```

Hàm phải trả về một giá trị số không âm cho các token đã được nhận diện; giá trị 0 hoặc giá trị âm báo hiệu kết thúc file.

Quy ước này được thiết kế để đầu ra của chương trình sinh bởi lex/flex có thể được sử dụng làm đầu vào cho Bison.

### 3.3 Hàm báo lỗi yyerror

Khi parser sinh bởi Bison phát hiện một lỗi cú pháp bởi đọc một token mà không có sản xuất nào thỏa mãn. Ngoài ra, một **action** trong Grammar Rules sử dụng macro **YYERROR** để báo lỗi.

Khi parser sinh bởi Bison muốn báo lỗi, nó sử dụng hàm **yyerror** mà ta bắt buộc phải khai báo và định nghĩa. Hàm **yyerror** có một tham số đầu vào là một xâu, thông thường là xâu "syntax error".

Parser cũng có thể phát hiện một loại lỗi khác: cạn kiệt bộ nhớ. Nó có thể 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. Xâu được truyền vào hàm **yyerror** là: "memory exhausted".

Hàm **yyerror** được định nghĩa đơn giản như sau:

```
void yyerror(const char *s) {
    fprintf(stderr, ``%s\n'', s);
}
```

Tuy nhiên ta cần khai báo hàm này trong phần Prologue để tránh trình dịch báo lỗi.

Sau khi hàm **yyerror** trả về, hàm **yyparse** sẽ cố gắng khôi phục lỗi nếu ta đã viết những Grammar Rule phù hợp.

## 4 Sử dụng Bison

### 4.1 Các lệnh biên dịch chương trình

Nếu ta có file grammar đầu vào là **example.y**, lệnh để sinh mã nguồn C từ file này là:

```
$ bison -d -o example.yacc.c example.y
```

Lệnh này sẽ tạo ra 2 file **example.yacc.c** và **example.yacc.h**. Tùy chọn **-d** nghĩa là sẽ tạo ra file header cùng với file mã nguồn. Tùy chọn **-o <filename>** chỉ định tên file sẽ được sinh ra.

Nếu **-o example.yacc.cxx** hoặc **-o example.yacc.cpp** thì file header tương ứng là **example.yacc.hxx** hoặc **example.yacc.hpp**

File **example.yacc.h** sẽ thường được include vào mã lex và mã nguồn chứa hàm main tương ứng.

## 4.2 Parser sinh bởi Bison đơn giản

Ở đây ta dùng Bison để sinh chương trình tính toán biểu thức đơn giản.

File ngữ pháp bison: **calc.y**

```
%{
    #include <stdio.h>
    #include <stdlib.h>

    extern int yylex();
    extern void yyerror(const char *s);
    extern void set_number(int n);
%}

%union {
    int num;
}
%token <num> NUMBER
%type <num> F U T E
%start E

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

File chỉ thị của flex: **calc.l**

```
%{
    #include "calc.yacc.h"
    #include <stdio.h>
%}

%%
[0-9]+      { yylval.num = atoi(yytext); return NUMBER; }
"+"         { return '+'; }
"-"         { return '-'; }
```

```
"*"          { return '*'; }
"/"          { return '/'; }
"%"          { return '%'; }
"("          { return '('; }
")"          { return ')'; }
[ \t\b\r\n]      ;
.    {
    printf("Khong nhan dien duoc ki tu: ");
    ECHO;
    printf("\n");
}
%%

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

File chứa hàm main: **main.c**

```
#include "calc.yacc.h"
#include <stdio.h>

void yyerror(const char *s) {
    printf("%s\n", s);
}

static int g_number;
void set_number(int n) {
    g_number = n;
}

int main() {
    int res = yyparse();
    printf("%d\n", g_number);
    return res;
}
```

Để biên dịch chương trình:

```
bison -d -o calc.yacc.c calc.y
flex -o calc.lex.c calc.l
gcc main.c calc.yacc.c calc.lex.c -o main
```

Khi đó với file đầu vào: **input**

```
(-3 - 2) * (20 + 80) + 50
```

Ta được đầu ra:

```
$ ./main < input
-450
```

## 4.3 Biểu diễn ngữ pháp của PL0 trong Bison

```
%{
    extern int yylex();
    extern void yyerror(const char *s);
%}

%token PROGRAM
%token IDENT
%token CONST
%token VAR
%token NUMBER
%token PROCEDURE
%token TOKEN_BEGIN
%token END
%token ASSIGN
%token CALL
%token IF
%token THEN
%token ELSE
%token WHILE
%token DO
%token FOR
%token TO
%token ODD

%token EQ
%token GT
%token GE
%token LT
%token LE
%token NE

%start program
%nonassoc THEN
%nonassoc ELSE

%%
program     :   PROGRAM IDENT ';' block '.' ;

block       :   const_decl_opt var_decl_opt procedure_decl_seq begin_block ;

const_decl_opt  :   %empty
                |   CONST const_list ';'
                ;
const_list  :   const_list ',' IDENT EQ NUMBER
            |   IDENT EQ NUMBER
            ;
var_decl_opt    :   %empty
                |   VAR var_list ';'
                ;
var_list    :   var_list ',' IDENT array_decl_opt
            |   IDENT array_decl_opt
            ;
array_decl_opt  :   %empty
                |   '[' NUMBER ']'
                ;
procedure_decl_seq  :   procedure_decl_seq procedure_decl
```

```
                    |   %empty
                    ;
procedure_decl  :   PROCEDURE IDENT procedure_params_opt ';' block ';' ;

procedure_params_opt    :   %empty
                        |   '(' param_list ')'
                        ;
param_list      :   param_list ';' var_opt IDENT
                |   var_opt IDENT
                ;
var_opt     :   %empty
            |   VAR
            ;

begin_block     :   TOKEN_BEGIN stmt_list END ;

stmt_list   : stmt_list ';' stmt
            | stmt
            ;
stmt    :   assign_stmt
        |   call_stmt
        |   begin_block
        |   if_stmt
        |   while_stmt
        |   for_stmt
        |   %empty
        ;
assign_stmt     :   IDENT array_subscript_opt ASSIGN expr ;

array_subscript_opt     :   %empty
                        |   '[' expr ']'
                        ;
call_stmt   :   CALL IDENT call_subscript_opt ;

call_subscript_opt  :   %empty
                    |   '(' expr_list ')'
                    ;

expr_list   :   expr_list ',' expr
            |   expr
            ;
if_stmt     :   IF cond THEN stmt
            |   IF cond THEN stmt ELSE stmt
            ;
while_stmt  :   WHILE cond DO stmt ;

for_stmt    :   FOR IDENT ASSIGN expr TO expr DO stmt ;

expr    :   expr '+' term
        |   expr '-' term
        |   term
        |   '-' term
        |   '+' term
        ;
term    :   term '*' factor
        |   term '/' factor
```

```
        |   term '%' factor
        |   factor
        ;
factor  :   IDENT array_subscript_opt
        |   NUMBER
        |   '(' expr ')'
        ;
cond    :   ODD expr
        |   expr comparison_operator expr
        ;
comparison_operator     :   EQ
                        |   GT
                        |   GE
                        |   LT
                        |   LE
                        |   NE
                        ;
%%
```

Trong đó

```
% nonassoc THEN
% nonassoc ELSE
```

được sử dụng để giải quyết vấn đề nhập nhằng của văn phạm do "Dangling else" [1].

# Tài liệu tham khảo


[1] Dangling else - Wikipedia. https://en.wikipedia.org/wiki/Dangling_else.

[2] Declarations (Bison 3.2.1). https://www.gnu.org/software/bison/manual/html_node/Declarations.html#Declarations.

[3] Empty Rules (Bison 3.2.1). https://www.gnu.org/software/bison/manual/html_node/Empty-Rules.html#Empty-Rules.

[4] GNU Bison - Wikipedia. https://en.wikipedia.org/wiki/GNU_Bison.

[5] Grammar Outline (Bison 3.2.1). https://www.gnu.org/software/bison/manual/html_node/Grammar-Outline.html#Grammar-Outline.

[6] Interface (Bison 3.2.1). https://www.gnu.org/software/bison/manual/html_node/Interface.html#Interface.

[7] Multiple Types (Bison 3.2.1). https://www.gnu.org/software/bison/manual/html_node/Multiple-Types.html#Multiple-Types.

[8] Prologue (Bison 3.2.1). https://www.gnu.org/software/bison/manual/html_node/Prologue.html#Prologue.

[9] Recursion (Bison 3.2.1). https://www.gnu.org/software/bison/manual/html_node/recursion.html#recursion.

[10] Rules Syntax (Bison 3.2.1).

[11] Start Decl (Bison 3.2.1). https://www.gnu.org/software/bison/manual/html_node/Start-Decl.html#Start-Decl.

[12] Token Decl (Bison 3.2.1). https://www.gnu.org/software/bison/manual/html_node/Token-Decl.html#Token-Decl.

[13] Type Decl (Bison 3.2.1). https://www.gnu.org/software/bison/manual/html_node/Type-Decl.html#Type-Decl.