# 基础知识

在本次实验中，我们将用到 `Flex` 和 `Bison` 两个工具以及 `Cminus-f` 语言。这里对其进行简单介绍。

## Cminus-f词法

`Cminus` 是C语言的一个子集，该语言的语法在《编译原理与实践》第九章附录中有详细的介绍。而 `Cminus-f` 则是在 `Cminus` 上追加了浮点操作。

1. 关键字

```
else if int return void while float
```

2. 专用符号

```
+ - * / < <= > >= == != = ; , ( ) [ ] { } /* */
```

3. 标识符ID和整数NUM，通过下列正则表达式定义:

```
letter = a|...|z|A|...|Z
digit = 0|...|9
ID = letter+
INTEGER = digit+
FLOAT = (digit+. | digit*.digit+)
```

4. 注释用 `/*...*/` 表示，可以超过一行。注释不能嵌套。

```
/*...*/
```

## Cminus-f语法

本小节将给出Cminus-f的语法，该语法在Cminus语言（《编译原理与实践》第九章附录）的基础上增加了float类型。

我们将 Cminus-f 的所有规则分为五类。

1. 字面量、关键字、运算符与标识符
   - `type-specifier`
   - `relop`
   - `addop`
   - `mulop`
2. 声明
   - `declaration-list`
   - `declaration`
   - `var-declaration`
   - `fun-declaration`
   - `local-declarations`
3. 语句

- `compound-stmt`
- `statement-list`
- `statement`
- `expression-stmt`
- `iteration-stmt`
- `selection-stmt`
- `return-stmt`

4. 表达式
   - `expression`
   - `simple-expression`
   - `var`
   - `additive-expression`
   - `term`
   - `factor`
   - `integer`
   - `float`
   - `call`
5. 其他
   - `params`
   - `param-list`
   - `param`
   - `args`
   - `arg-list`

起始符号是 `program`。文法中用到的 token 均以下划线和粗体标出。

1. \program → declaration-list\
2. \declaration-list → declaration-list declaration | declaration\
3. \declaration → var-declaration | fun-declaration\
4. \var-declaration → type-specifier **ID** **;** | type-specifier **ID** **[** **INTEGER** **]** **;**\
5. \type-specifier → **int** | **float** | **void**\
6. \fun-declaration → type-specifier **ID** **(** params **)** compound-stmt\
7. \params → param-list | **void**\
8. \param-list → param-list **,** param | param\
9. \param → type-specifier **ID** | type-specifier **ID** **[** **]**\
10. \compound-stmt → **{** local-declarations statement-list **}**\
11. \local-declarations → local-declarations var-declaration | empty\
12. \statement-list → statement-list statement | empty\
13. \
    statement → expression-stmt
    | compound-stmt
    | selection-stmt \
    | iteration-stmt
    | return-stmt
14. \expression-stmt → expression **;** | **;**\
15. \
    selection-stmt → **if** **(** expression **)** statement
    | **if** **(** expression **)** statement **else** statement\
16. \iteration-stmt → **while** **(** expression **)** statement\
17. \return-stmt → **return** **;** | **return** expression **;**\

18. \expression → var = expression | simple-expression\
19. \var → **ID** | **ID** [ expression ]\
20. \simple-expression → additive-expression relop additive-expression | additive-expression\
21. \relop → <= | < | > | >= | == | !=\
22. \additive-expression → additive-expression addop term | term\
23. \addop → + | -\
24. \term → term mulop factor | factor\
25. \mulop → * | /\
26. \factor → ( expression ) | var | call | integer | float\
27. \integer → **INTEGER**\
28. \float → **FLOATPOINT**\
29. \call → **ID** ( args)\
30. \args → arg-list | empty\
31. \arg-list → arg-list , expression | expression\

# Flex用法简介

FLEX是一个生成词法分析器的工具。利用FLEX，我们只需提供词法的正则表达式，就可自动生成对应的C代码。整个流程如下图：

首先，FLEX从输入文件*.lex或者stdio读取词法扫描器的规范，从而生成C代码源文件lex.yy.c。然后，编译lex.yy.c并与-lfl库链接，以生成可执行的a.out。最后，a.out分析其输入流，将其转换为一系列token。

简答的说，Flex根据用户定义的正则表达式对输入的字符串进行分析，生成token stream。在我们的编译原理实验中，token stream将被用于后续的语法树生成等后续工作。一个简单的示意如下：

```
"int main() {int a; a = 1;}"   #main.c文件，现在是一个文本
-> flex分析
-> "}" ";" "1" "=" "a" ";" "a" "int" "{" ")" "(" "main" "int"  # 生成的token stream
#   >>>>>>>>>>>>>>>>>>token stream>>>>>>>>>>>>>>>>>>>>>>>>>>
```

我们以一个简单的单词数量统计的程序wc.l为详细介绍下Flex的功能和用法（请仔细看程序中的注释内容）：

```
%option noyywrap
%{
//在%{和%}中的代码会被原样照抄到生成的lex.yy.c文件的开头，也就是在%{和}%中，你应该按C语言写代码，在这里可以完成变量声明与定义、相关库的导入和函数定义
#include <string.h>
int chars = 0;
int words = 0;
%}

%%
 /* 注意这里的%%开头*/
 /* %%开头和%%结尾之间的内容就是使用flex进行解析的部分 */
 /* 你可以按照我这种方式在这个部分写注释，注意注释最开头的空格，这是必须的 */
```

```
 /* 你可以在这里使用你熟悉的正则表达式来编写模式， 你可以用C代码来指定模式匹配时对应的动作 */
 /* 在%%和%%之间，你应该按照如下的方式写模式和动作 */
 /* 模式 动作 */
 /* 其中模式就是正则表达式，动作为模式匹配执行成功后执行相应的动作，这里的动作就是相应的代码 */
 /* 你可以仔细研究下后面的例子 */
 /* [a-zA-Z]+ 为正则表达式，用于匹配大小写字母 */
 /* {chars += strlen(yytext);words++;} 则为匹配到大小写字母后，执行的动作（代码），这里是完成一个字符累加操作 */
 /* 这里yytext的类型为 char*， 是一个指向匹配到字符串的指针 */
 /* yytext是flex自动生成的，在%%和%%之中无需额外定义或者声明 */

 /* 一条 模式 + 动作 */
[a-zA-Z]+ { chars += strlen(yytext);words++;}

 /* 另一条 模式 + 动作； . 匹配任意字符，这里匹配非大小写字母的其他字符。这里思考一个问题，A既可以被[a-zA-Z]+匹配，也可以被.匹配，在这个程序中为什么A优先被[a-zA-Z]+匹配？如果你感兴趣可以去看另一个文档 */
. {}
 /* 对其他所有字符，不做处理，继续执行 */
 /* 注意这里的%%结尾 */
%%

// flex部分结束，这里可以正常写C代码了
int main(int argc, char **argv){
    // yylex()是flex提供的词法分析例程，调用yylex()即开始执行Flex的词法分析，同样的yylex()也是flex自行生成的，无需额外定义和生成，默认输入读取stdin
    // 如果不清楚什么是stdin，可以自己百度查一下
    yylex();
    // 输出 words和chars，这些变量在匹配过程中，被执行相应的动作
    printf("look, I find %d words of %d chars\n", words, chars);
    return 0;
}
```

使用Flex生成lex.yy.c

```
$ flex wc.l
$ gcc lex.yy.c
$ ./a.out
hello world
^D
look, I find 2 words of 10 chars
```

*注: 在以stdin为输入时，需要按下ctrl+D以退出*

至此，你已经成功使用Flex完成了一个简单的分析器!

为了对实验有较好的体验，我建议你好好阅读以下两个关于flex文档:

- Flex matching
- Flex regular expressions

# Bison用法简介

Bison 是一款解析器生成器（parser generator），它可以将 LALR 文法转换成可编译的 C 代码，从而大大减轻程序员手动设计解析器的负担。Bison 是 GNU 对早期 Unix 的 Yacc 工具的一个重新实现，所以文件扩展名为 `.y`。（Yacc 的意思是 Yet Another Compiler Compiler。）

## 识别一个简单的语言

下面我们以一个简单的语言为例，介绍 Bison 的用法。

每个 Bison 文件由 `%%` 分成三部分。

```
%{
#include <stdio.h>
/* 这里是序曲 */
/* 这部分代码会被原样拷贝到生成的 .c 文件的开头 */
int yylex(void);
void yyerror(const char *s);
%}

/* 这些地方可以输入一些 bison 指令 */
/* 比如用 %start 指令指定起始符号，用 %token 定义一个 token */
%start reimu
%token REIMU

%%
/* 从这里开始，下面是解析规则 */
reimu : marisa { /* 这里写与该规则对应的处理代码 */ puts("rule1"); }
      | REIMU  { /* 这里写与该规则对应的处理代码 */ puts("rule2"); }
      ; /* 规则最后不要忘了用分号结束哦～ */

/* 这种写法表示 ϵ —— 空输入 */
marisa : { puts("Hello!"); }

%%
/* 这里是尾声 */
/* 这部分代码会被原样拷贝到生成的 .c 文件的末尾 */

int yylex(void)
{
    int c = getchar(); // 从 stdin 获取下一个字符
    switch (c) {
    case EOF: return YYEOF;
    case 'R': return REIMU;
    default:  return YYUNDEF;     // 报告 token 未定义，迫使 bison 报错。
    // 由于 bison 不同版本有不同的定义。如果这里 YYUNDEF 未定义，请尝试 YYUNDEFTOK 或使用一个随意的整数，如 114514 或 19260817。
    }
}

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

int main(void)
{
```

```
    yyparse(); // 启动解析
    return 0;
}
```

另外有一些值得注意的点：

1. Bison 传统上将 token 用大写单词表示，将 symbol 用小写字母表示。
2. Bison 能且只能生成解析器源代码（一个 `.c` 文件），并且入口是 `yyparse`，所以为了让程序能跑起来，你需要手动提供 `main` 函数（但不一定要在 `.y` 文件中——你懂“链接”是什么，对吧？）。
3. Bison 不能检测你的 action code 是否正确——它只能检测文法的部分错误，其他代码都是原样粘贴到 `.c` 文件中。
4. Bison 需要你提供一个 `yylex` 来获取下一个 token。
5. Bison 需要你提供一个 `yyerror` 来提供合适的报错机制。

顺便提一嘴，上面这个 `.y` 是可以工作的——尽管它只能接受两个字符串。把上面这段代码保存为 `reimu.y`，执行如下命令来构建这个程序：

```
$ bison reimu.y
$ gcc reimu.tab.c
$ ./a.out
R<-- 不要回车在这里按 Ctrl-D
rule2
$ ./a.out
<-- 不要回车在这里按 Ctrl-D
Hello!
rule1
$ ./a.out
blablabla <-- 回车或者 Ctrl-D
Hello!
rule1     <-- 匹配到了 rule1
syntax error <-- 发现了错误
```

于是我们验证了上述代码的确识别了该文法定义的语言 `{ "", "R" }`。

## Bison 和 Flex 联动

聪明的你应该发现了，我们这里手写了一个 `yylex` 函数作为词法分析器。而在上文中我们正好使用 flex 自动生成了一个词法分析器。如何让这两者协同工作呢？特别是，我们需要在这两者之间共享 token 定义和一些数据，难道要手动维护吗？哈哈，当然不用！下面我们用一个四则运算计算器来简单介绍如何让 bison 和 flex 协同工作——重点是如何维护解析器状态、`YYSTYPE` 和头文件的生成。

首先，我们必须明白，整个工作流程中，bison 是占据主导地位的，而 flex 仅仅是一个辅助工具，仅用来生成 `yylex` 函数。因此，最好先写 `.y` 文件。

```
/* calc.y */
%{
#include <stdio.h>
    int yylex(void);
    void yyerror(const char *s);
%}

%token RET
```

```
%token <num> NUMBER
%token <op> ADDOP MULOP LPAREN RPAREN
%type <num> top line expr term factor

%start top

%union {
    char   op;
    double num;
}

%%

top
: top line {}
| {}

line
: expr RET
{
    printf(" = %f\n", $1);
}

expr
: term
{
    $$ = $1;
}
| expr ADDOP term
{
    switch ($2) {
    case '+': $$ = $1 + $3; break;
    case '-': $$ = $1 - $3; break;
    }
}

term
: factor
{
    $$ = $1;
}
| term MULOP factor
{
    switch ($2) {
    case '*': $$ = $1 * $3; break;
    case '/': $$ = $1 / $3; break; // 想想看，这里会出什么问题？
    }
}

factor
: LPAREN expr RPAREN
{
    $$ = $2;
}
| NUMBER
```

```
{
    $$ = $1;
}

%%

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

```
/* calc.l */
%option noyywrap

%{
/* 引入 calc.y 定义的 token */
#include "calc.tab.h"
%}

%%

\( { return LPAREN; }
\) { return RPAREN; }
"+"|"-" { yylval.op = yytext[0]; return ADDOP; }
"*"|"/" { yylval.op = yytext[0]; return MULOP; }
[0-9]+|[0-9]+\.[0-9]*|[0-9]*\.[0-9]+ { yylval.num = atof(yytext); return NUMBER; }
" "|\t {  }
\r\n|\n|\r { return RET; }

%%
```

最后，我们补充一个 `driver.c` 来提供 `main` 函数。

```
int yyparse();

int main()
{
    yyparse();
    return 0;
}
```

使用如下命令构建并测试程序：

```
$ bison -d calc.y
   (生成 calc.tab.c 和 calc.tab.h。如果不给出 -d 参数，则不会生成 .h 文件。)
$ flex calc.l
   (生成 lex.yy.c)
$ gcc lex.yy.c calc.tab.c driver.c -o calc
$ ./calc
1+1
 = 2.000000
2*(1+1)
 = 4.000000
2*1+1
 = 3.000000
```

如果你复制粘贴了上述程序，可能会觉得很神奇，并且有些地方看不懂。下面就详细讲解上面新出现的各种构造。

- `YYSTYPE`：在 bison 解析过程中，每个 symbol 最终都对应到一个语义值上。或者说，在 parse tree 上，每个节点都对应一个语义值，这个值的类型是 `YYSTYPE`。`YYSTYPE` 的具体内容是由 `%union` 构造指出的。上面的例子中，

  ```
  %union {
    char   op;
    double num;
  }
  ```

  会生成类似这样的代码

  ```
  typedef union YYSTYPE {
    char op;
    double num;
  } YYSTYPE;
  ```

  为什么使用 `union` 呢？因为不同节点可能需要不同类型的语义值。比如，上面的例子中，我们希望 `ADDOP` 的值是 `char` 类型，而 `NUMBER` 应该是 `double` 类型的。

- `$$` 和 `$1`，`$2`，`$3`，...：现在我们来看如何从已有的值推出当前节点归约后应有的值。以加法为例：

  ```
  term : term ADDOP factor
       {
          switch $2 {
          case '+': $$ = $1 + $3; break;
          case '-': $$ = $1 - $3; break;
          }
       }
  ```

  其实很好理解。当前节点使用 `$$` 代表，而已解析的节点则是从左到右依次编号，称作 `$1`，`$2`，`$3`...

- `%type <>` 和 `%token <>`：注意，我们上面可没有写 `$1.num` 或者 `$2.op` 哦！那么 bison 是怎么知道应该用 `union` 的哪部分值的呢？其秘诀就在文件一开始的 `%type` 和 `%token` 上。

  例如，`term` 应该使用 `num` 部分，那么我们就写

```
%type <num> term
```

这样，以后用 `$` 去取某个值的时候，bison 就能自动生成类似 `stack[i].num` 这样的代码了。

`%token<>` 见下一条。

- `%token`：当我们用 `%token` 声明一个 token 时，这个 token 就会导出到 `.h` 中，可以在 C 代码中直接使用（注意 token 名千万不要和别的东西冲突！），供 flex 使用。`%token <op> ADDOP` 与之类似，但顺便也将 `ADDOP` 传递给 `%type`，这样一行代码相当于两行代码，岂不是很赚。
- `yylval`：这时候我们可以打开 `.h` 文件，看看里面有什么。除了 token 定义，最末尾还有一个 `extern YYSTYPE yylval;`。这个变量我们上面已经使用了，通过这个变量，我们就可以在 lexer **里面**设置某个 token 的值。

呼……说了这么多，现在回头看看上面的代码，应该可以完全看懂了吧！这时候你可能才意识到为什么 flex 生成的分析器入口是 `yylex`，因为这个函数就是 bison 专门让程序员自己填的，作为一种扩展机制。另外，bison（或者说 yacc）生成的变量和函数名通常都带有 `yy` 前缀，希望在这里说还不太晚……

最后还得提一下，尽管上面所讲已经足够应付很大一部分解析需求了，但是 bison 还有一些高级功能，比如自动处理运算符的优先级和结合性（于是我们就不需要手动把 `expr` 拆成 `factor`, `term` 了）。这部分功能，就留给同学们自己去探索吧！