语法实验

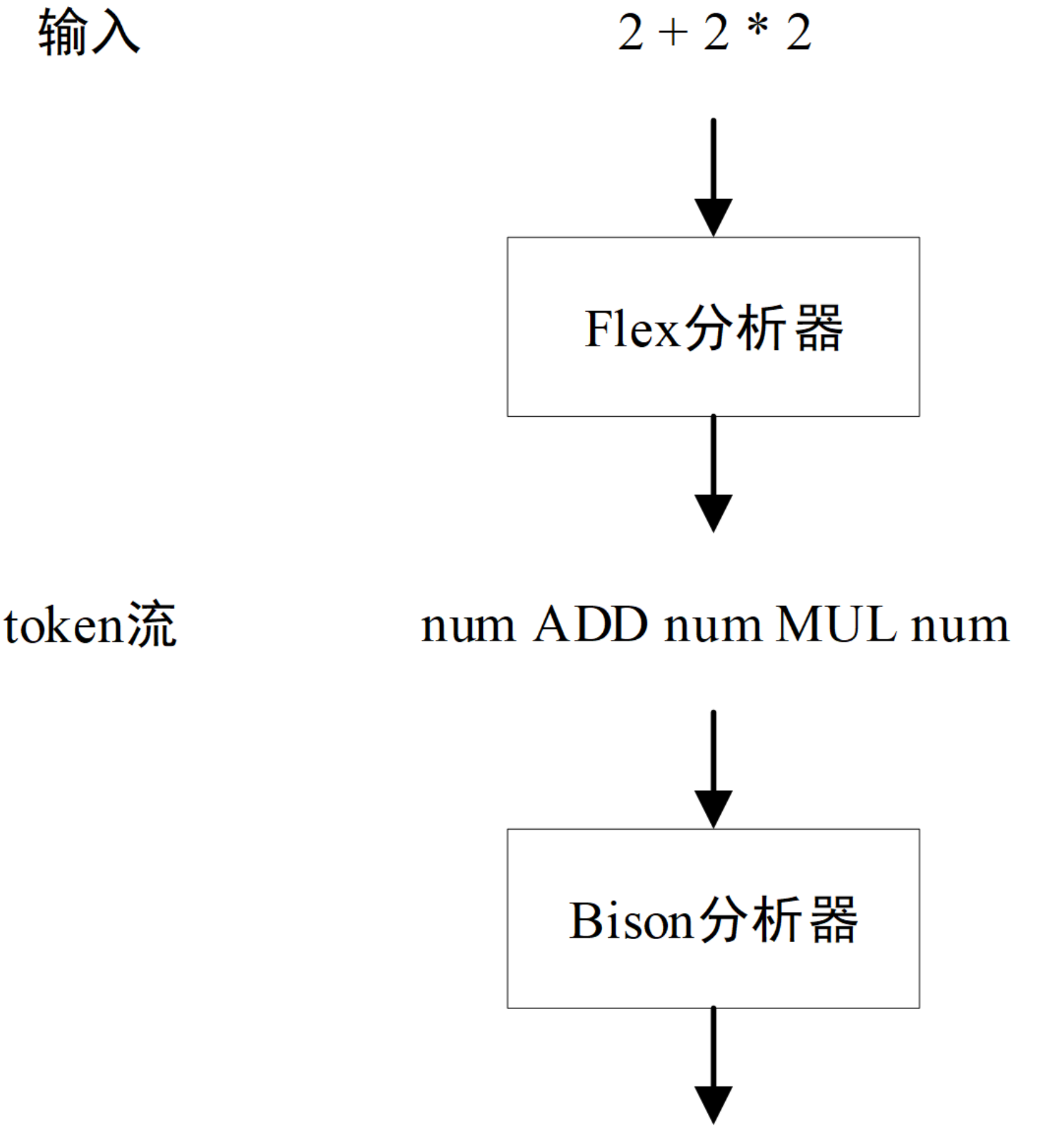
Bison

简介

Bison 是一款 LALR 文法解析器生成器，可转换为可编译的 C 代码，减轻手动设计解析器的工作。它重新实现了早期 Unix 上的 Yacc 工具，文件扩展名为 .y（Yacc 意为 Yet Another Compiler Compiler）。

Flex 和 Bison 是 Linux 下生成词法分析器和语法分析器的工具，用于处理结构化输入，协同工作解析复杂文件。Flex 将文本文件拆分为有意义的词法记号（token），而 Bison 根据语法规则生成抽象语法树（AST），Bison 在协同工作中担任主导角色，而 Flex 辅助生成 yylex 函数。

以计算器程序（该程序即为下文的一个复杂的 Bison 程序）为例，用户在界面输入 2 + 2 * 2，Flex 将输入识别为 token 流，其中 2 被识别为 number，+ 被识别为 ADD，* 被识别为 MUL。接下来，Bison 负责根据语法规则将这些 token 组织成 AST 树，流程如下图所示：



使用方法

Bison 源程序后缀一般为 .y，其结构可分为定义部分、语法规则部分、C代码部分。

定义部分可以分为以下两个部分：

1. 包括 C 语言代码、头文件引用、宏定义、全局变量定义和函数声明等内容，位于 `%{` 和 `%}` 之间。
2. 终结符和非终结符声明：用于定义语法中使用的终结符（也称为记号）和非终结符，常见声明包括 `%token`、`%union`、`%start`、`%type`、`%left`、`%right`、`%nonassoc` 等。
    - `%token` 定义终结符。定义形式：`%token TOKEN1 TOKEN2`。一行可定义多个终结符，空格分隔。一般约定终结符都是大写，非终结符的名字是小写。
    - `%type` 定义非终结符。
    - `%left`、`%right`、`%nonassoc` 定义终结符的结合性和优先级关系。定义形式与 `%token` 类似。先定义的优先级低，最后定义的优先级最高，同时定义的优先级相同。`%left` 表示左结 合（如“+”、“-”、“*”、“/”）；`%right` 表示右结合（例如“=”）；`%nonassoc` 表示不可结合（即它定义的终结符不能连续出现。例如“-”负号。如下定义中，优先级关系为：AA = BB < CC < DD；表示结合性为：AA、BB 左结合，DD 右结合，CC 不可结合。

        ```
        %left AA BB
        %nonassoc CC
        %right DD
        ```

    - `%union` 定义了语法符号的语义值类型的集合。在 Bison 中，每个符号，包括记号和非终结符，都有一个不同数据类型的语义值，并且这些值通过 yylval 变量在移进和归约操作中传递。默认情况下，YYSTYPE（宏定义）为 yylval 的类型，通常为 int。但通过使用 `%union`，你可以重新定义符号的类型。使用 union 是因为不同节点可能需要不同类型的语义值，比如下面的定义，希望 ADDOP 的 值是 char 类型，而 NUMBER 应该是 double 类型。

        ```
        %token <num> NUMBER
        %token <op> ADDOP MULOP LPAREN RPAREN
        %union {
         char op;
         double num;
        }

        # 注意：一旦 %union 被定义，需要指定 Bison 每种符号使用的值类型，值类型通过放在尖括号中的 union 类型对应的成员名称确定，如 %token <num>。
        ```

3. 使用 `%start` 指定文法的开始符号，表示最终需要规约成的符号，例如 `%start program`。如果不使用 `%start` 定义文法开始符 号，则默认在第二部分规则段中定义的第一条生产式规则的左部非终结符为开始符号。

语法规则部分：

- 语法规则部分由归约规则和动作组成。规则基本按照巴科斯范式（BNF）描述。规则中目标或非终端符放在左边，后跟一个冒号：然后是 产生式的右边，之后是对应的动作（用 `{}` 包含）。**Bison 的语法树是按自下而上的归约方式进行构建的。**如下所示：

    ```
    %%
     program: program expr '\n' { printf("%d\n", $2); }
     ;
     expr: expr '+' expr { $$ = $1 + $3; }
     | expr '-' expr { $$ = $1 - $3}
     ;
    %%
    /* 动作中“$1”表示右边的第一个标记的值，“$2”表示右边的第二个标记的值，依
    次类推。“$$”表示归约后的值。以“expr: expr '+' expr { $$ = $1 + $3; }”，说明“$$”表示从左
    向右第一个 expr，即规约的结果，“$1”表示从左向右第二个 expr，“$2”表示“+”加号，“$3”
    表示从左向右第三个 expr。“|”符号表示其他的规约规则。 */
    ```

C 代码部分：

- C 代码部分为 C 代码，会被原样复制到 c 文件中，这里一般自定义一些函数。主要包括调用 Bison 的语法分析程序 `yyparse()`。其中 `yyparse` 函数由 Bison 根据语法规则自动生成，用于语法分析。

实例分析

我们接下来展示一个复杂的 Bison 程序，该程序同时使用 Flex 和 Bison，使用 Flex 生产的 yylex 函数进行字符串分析，Bison 生成的 `yyparse` 进行语法树构建。共涉及 2 个文件，`calc.y` 和 `calc.l`。其功能是实现一个数值计算器。

```
/* 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; // 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);
}

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

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

%{
/* 引入 calc.y 定义的 token */
/* calc.tab.h 文件由 Bison 生成 */
```

```
/* 当我们在.y 文件中使用 %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，供 Flex 使用。就如 .l 文件中的\( { return LPAREN; }，
   其中 LPAREN 定义来自 calc.tab.h，由对应的 .y 文件生成 */
#include "calc.tab.h"
%}


/* 规则部分 yylval 同样来自 calc.tab.h 文件，其类型为 YYSTYPE，用于 token 的相关属性，比如 NUMBER 对应的实际数值 */
/* 在这个例子中，YYSTYPE 定义如下

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

其同样由 .y 文件根据 %union 生成，在文件中我们的 %union 定义如下

%union {
    char   op;
    double num;
}
*/

%%
\( { return LPAREN; }
\) { 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; }
%%
```

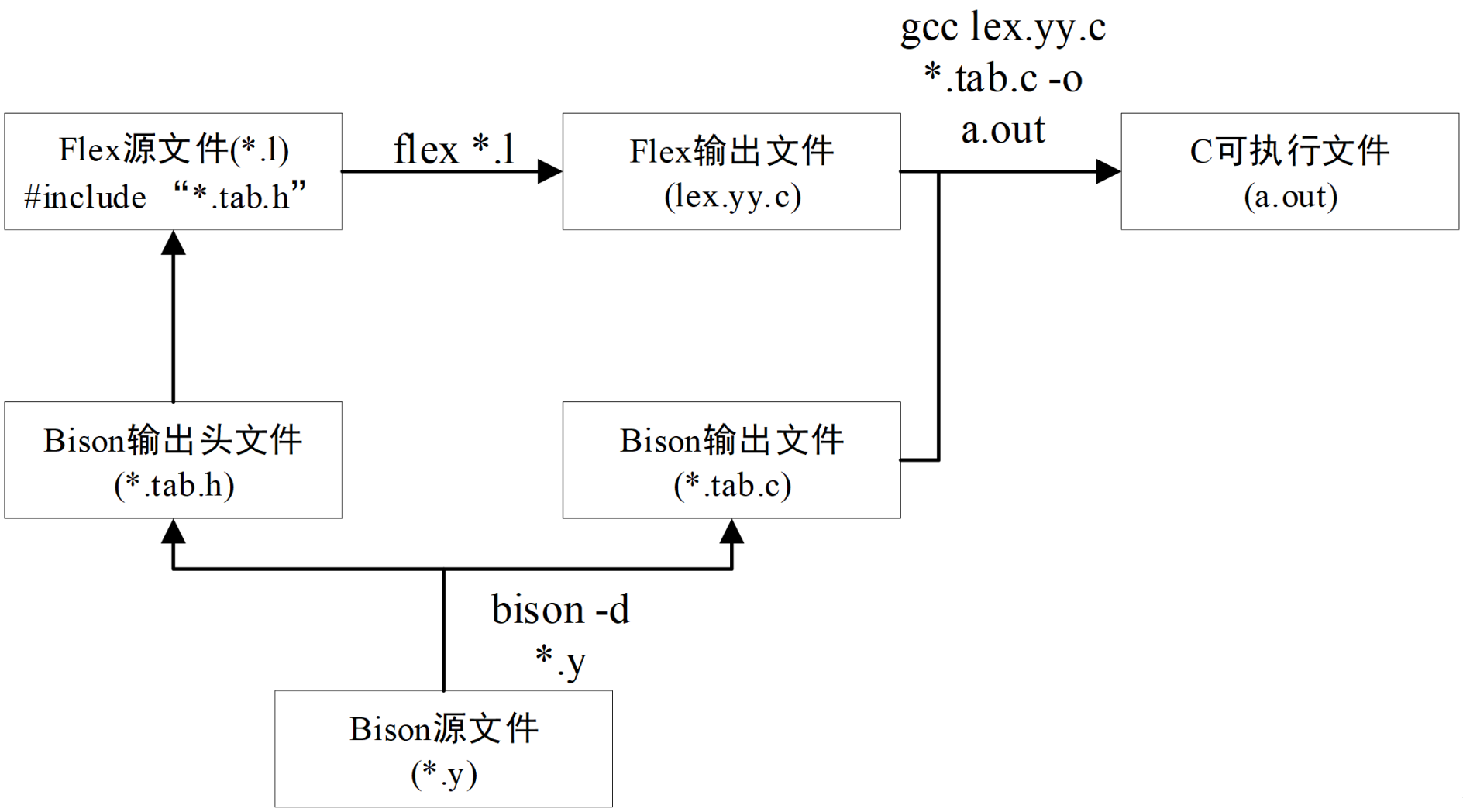
使用如下命令构建并测试程序：

```
# 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。
$ bison -d calc.y
# 生成 lex.yy.c
$ flex calc.l

$ ls calc.tab.c calc.tab.h lex.yy.c
calc.tab.c  calc.tab.h  lex.yy.c

# 编译
$ gcc lex.yy.c calc.tab.c -o calc
$ ./calc
1+1
 = 2.000000
2*(1+1)
 = 4.000000
2*1+1
 = 3.000000
```

其大致工作流程如下:



graphviz

该工具仅供参考，可以自己选择其他绘图工具。

- 定义有向图：

  在大括号内部编写结点之间的连边关系。

```
digraph " "{
    ...
}
```

- 设置结点形状：

```
node [shape = record,height=.1]
```

- 定义节点：

  中括号外为结点的名称（不会显示在图片中）。

  中括号中的双引号内部为结点每个部分的名称。尖括号中为该部分的标号，后续的空格为该部分的名称（显示在图片中的）

```
NAME_NODE[label = "<f0> NAME0|<f1> NAME1"];
```

- 结点连边：

  连边可以从一个结点的某个部分联想另一个结点。

  下面是一个从node0结点的f0部分连向node1结点的示例：

```
"node0":f0->"node1";
```

- 在命令行中使用如下指令，将Tree.dot文件转化为Tree.png文件：

```
dot -Tpng -o Tree.png Tree.dot
```

实验要求

在词法实验的基础上，借助bison工具实现一个语法分析器，要求参考SysY语言文法编写适当的语义动作，能够按照规约顺序输出需要用到的规约规则，同时绘制SysY代码的语法树。

输入： SysY源程序
输出： 规约规则与对应的语法树

特别提示

- 注意if-else语句中的移进-规约冲突
- 注意四则运算、条件语句等各类优先级问题
- 改写语法

  大部分语法可以直接照抄文档，但是由于bison不支持正则表达式，在编写规则的时候需要适当的修改语法，使其能够被识别。

  通过定义多条类似的语法，实现正则表达式中“出现一次或多次”、“至多出现一次”等功能。

- 错误恢复（可选）

  错误恢复旨在尽可能多地找出程序中的错误语法，以提高编程效率。

  在bison中，可以使用自带的终结符$error$通配错误的语法，而不中止编译过程。当匹配到$error$终结符时，会自动调用yyerror函数。