语法分析

1 语法分析的任务

语法分析的主要任务是在词法分析结果的基础上，进一步得到解析树（parsing tree）。例如，下面两个算术表达式：

```
1 + x * y
```

```
(1 + x) * y
```

它们经过词法分析结果如下：

```
TOK_NAT(1) TOK_PLUS TOK_IDENT(x) TOK_MUL TOK_IDENT(y)
```

```
TOK_LEFT_PAREN TOK_NAT(1) TOK_PLUS TOK_IDENT(x)
TOK_RIGHT_PAREN TOK_MUL TOK_IDENT(y)
```

期望的语法分析结果如下：

```
      *                   *
     / \                 / \
   ( )   y             +   y
    |                 / \
    +                1   x
   / \
  1   x
```

2 上下文无关语法与解析树

下面是一套上下文无关语法（context-free grammar，CFG）的例子：

```
S -> S ; S          E -> ID          L -> E
S -> ID := E        E -> NAT         L -> L , E
S -> PRINT ( L )    E -> E + E
                    E -> ( E )
```

下面是这套上下文无关语法的一个派生（derivation）：

```
S -> S; S
  -> ID := E; S
  -> ID := E + E; S
  -> ID := ID + E; S
  -> ID := ID + NAT; S
  -> ID := ID + NAT; PRINT(L)
  -> ID := ID + NAT; PRINT(L, E)
  -> ID := ID + NAT; PRINT(E, E)
  -> ID := ID + NAT; PRINT(ID, E)
  -> ID := ID + NAT; PRINT(ID, ID)
```

在上面例子中，`S` 表示语句，`E` 表示表达式，`L` 表示表达式列表。在这个语言中，`PRINT` 指令可以带多个参数，表达式中只允许出现加法运算，多条语句只允许顺序执行，没有条件分支与循环。

一套上下文无关语法包含以下几个组成部分：

- 一个初始符号，例如：`S` ；
- 一个终结符（terminal symbols）集合，例如：`ID NAT , ; ( ) + :=` ，当词法分析器和语法分析器结合使用的时候，这个终结符集合一般就是词法分析中的标记集合；
- 一个非终结符（nonterminal symbols）集合，例如：`S E L` ；
- 一系列产生式（production），每个产生式的左边是一个非终结符，每个产生式的右边是一列（可以为空）终结符或非终结符。

将初始符号依据产生式不断展开最后得到一个终结符序列的过程称为派生（derivable）。

下面是这套上下文无关语法的一棵解析树（parsing tree）：

```
          E                          E
        / | \                        -> E * E
       E  *  E                       -> E + E * E
     / | \   |                       -> ID + E * E
    E  +  E  ID                      -> ID + ID * E
    |     |                          -> ID + ID * ID
   ID     ID
```

解析树具有下面性质：

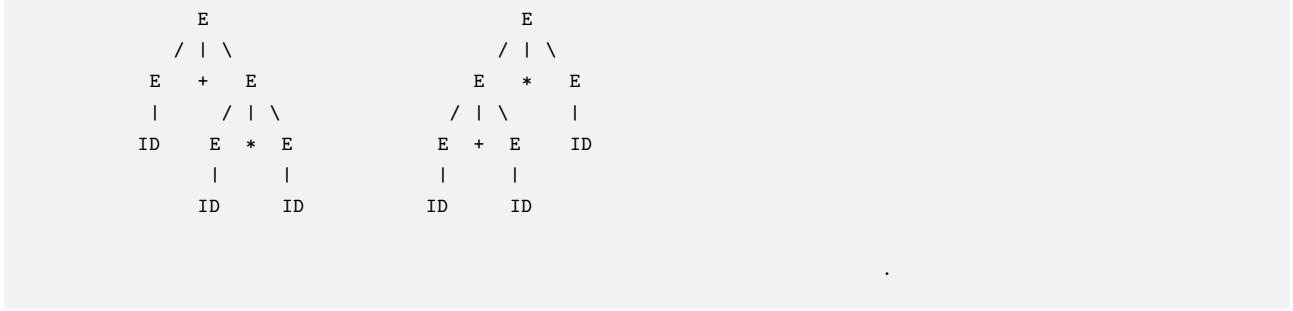
- 根节点为上下文无关语法的初始符号；
- 每个叶子节点是一个终结符，每个内部节点是一个非终结符；
- 每一个父节点和他的子节点构成一条上下文无关语法中的产生式；

3 歧义与歧义的消除

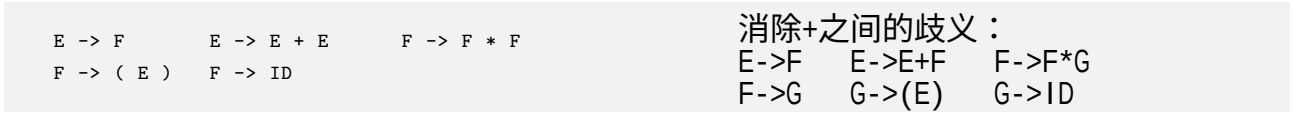
下面上下文无关语法有歧义：

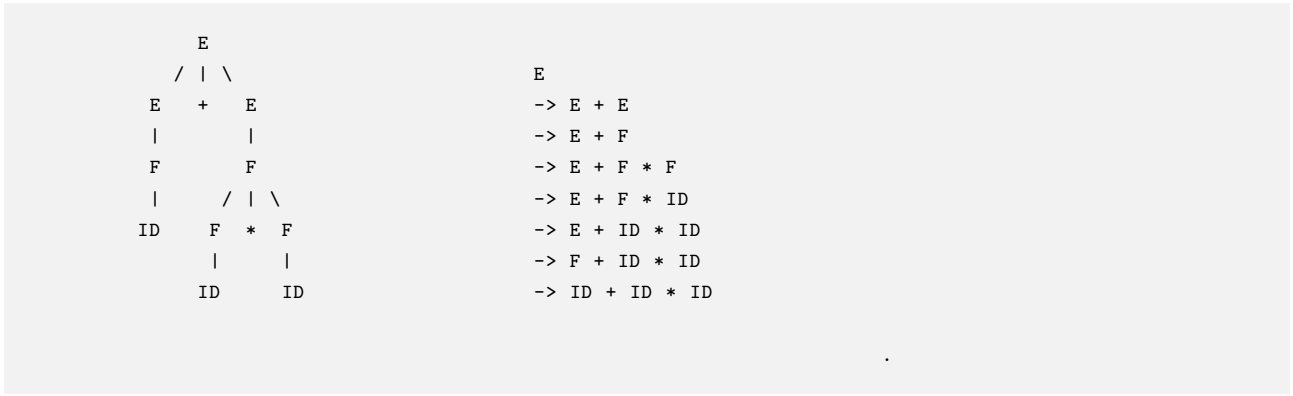
```
E -> ID       E -> E + E      E -> E * E      E -> ( E )
```

同一标记串，有两种解析树：



修正上下文无关语法，消除加号与乘号优先级有关的歧义



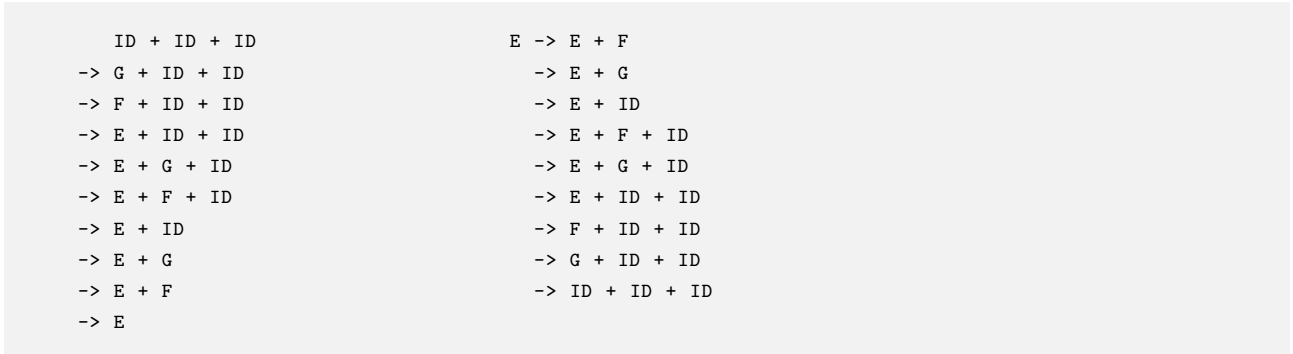


4 派生与规约

最左派生与最右派生：

- 一个派生中，如果每次都展开最左侧的非终结符，那么这个派生就称为一个最左派生（left-most derivation）；
- 一个派生中，如果每次都展开最右侧的非终结符，那么这个派生就称为一个最右派生（right-most derivation）；
- 同一棵解析树能够唯一确定一种最左派生，同一棵解析树能够唯一确定一种最右派生；
- 如果一串标记串没有歧义，那么只有唯一的最左派生可以生成这一标记串，也只有唯一的最右派生可以生成这一标记串；

规约是派生反向过程：



请大家注意，上图中的派生是最右派生，但是他对应的规约却是每次尽可能地进行左侧规约。这是因为派生与规约的方向是相反的。

5 移入规约分析

下面将要介绍的移入规约分析，是一个计算生成最左规约（或者说最右派生）的过程。移入规约分析的主要思想是：从左向右扫描标记串，从左向右进行规约。

- 共有两类操作：移入、规约；
- 扫描线的右侧全部都是终结符；
- 规约操作都发生在扫描线的左侧紧贴扫描线的区域内。

例子

```
   | ID + ID + ID
-> ID | + ID + ID
-> G | + ID + ID
-> F | + ID + ID
-> E | + ID + ID
-> E + | ID + ID
-> E + ID | + ID
-> E + G | + ID
-> E + F | + ID
-> E | + ID
-> E + | ID
-> E + ID |
-> E + G |
-> E + F |
-> E |
```

移入与规约的选择问题：

- 既能移入，又能规约的情形，应当选择移入还是规约？
- 有多种规约方案的情形，应当如何选择？

例子：

```
   | ID + ID + ID
-> ID | + ID + ID
-> G | + ID + ID          E + E | + ID
-> F | + ID + ID
-> E | + ID + ID          E | + ID
-> E + | ID + ID
-> E + ID | + ID          E + F + | ID
-> E + G | + ID
-> E + F | + ID
```

6 已移入部分的结构

已移入部分的结构

- 已移入规约的部分可以分为 n 段；
- 每段对应一条产生式；

- 第 i 段拼接上第 $i+1$ 条产生式的左侧非终结符是第 i 条产生式右侧符号串的一个前缀。

例子

- `F * (E) |` 中已移入规约的部分可以分为：`F *` `( E )`
- 分别对应产生式：`F -> F * . G` `G -> ( E ) .`
- `F * (E + | ID )` 中已移入规约的部分可以分为：`F *` `(` `E +`
- 分别对应产生式：`F -> F * . G` `G -> ( . E )` `E -> E + . F`

移入与规约对应的扫描线左侧结构变化：
原操作：移入

```
| ID + ID + ID     ->     ID | + ID + ID
```

带结构的操作：

```
(START -> . E)     ->     (START -> . E)
                          (E -> . E + F)
                          (E -> . E + F)
                          (E -> . F)
                          (F -> . G)
                          (G -> ID .)
                                                    .
```

原操作：规约

```
ID | + ID + ID    ->    G | + ID + ID
```

带结构的操作：

```
(START -> . E)    ->    (START -> . E)
(E -> . E + F)          (E -> . E + F)
(E -> . E + F)          (E -> . E + F)
(E -> . F)              (E -> . F)
(F -> . G)              (F -> G .)
(G -> ID .)
```

我的目的是要把G这个东西弄出来，即看看他是不是|左边的可行结构
所以他必须是用产生式转化这转化着和开头段相同
从START开始，到.E，没有消耗掉G，
从E继续走，有三种走法，前两种的开头还是E，又回去了
第三种走到了.F，接下来就看F继续走
F可以走到.G，此时消耗一个G，变成G.，则完成了匹配，说明可行

原操作：规约

```
G | + ID + ID      ->     F | + ID + ID
```

带结构的操作：

```
(START -> . E)     ->     (START -> . E)
(E -> . E + F)            (E -> . E + F)
(E -> . E + F)            (E -> . E + F)
(E -> . F)                (E -> F .)
(F -> G .)
                                                    .
```

原操作：规约

```
F | + ID + ID      ->     E | + ID + ID
```

增加结构：

```
(START -> . E)     ->     (START -> . E)
(E -> . E + F)            (E -> . E + F)
(E -> . E + F)            (E -> E . + F)
(E -> F .)
                                                    .
```

原操作：移入

```
E | + ID + ID      ->     E + | ID + ID
```

增加结构：

```
(START -> . E)     ->     (START -> . E)      这里我们来看E+可行吗？
(E -> . E + F)            (E -> . E + F)      同样先转换为.E，没有消耗
(E -> E . + F)            (E -> E + . F)      然后E只能转换为.E+F,可以消耗E和+，变成E+.F，说明可以消耗全
                                              说明是可行结构
                                                    .
```

原操作：移入

```
E + | ID + ID      ->     E + ID | + ID
```

增加结构：

```
(START -> . E)     ->     (START -> . E)
(E -> . E + F)            (E -> . E + F)
(E -> E + . F)            (E -> E + . F)
                          (F -> .G)
                          (G -> ID .)
                                                    .
```

7 已移入部分的结构判定

以扫描线左侧的最右段为状态，构建 NFA。新增一段带来的变化对应一条 ϵ 边，其他加入符号带来的变化对应一条普通边。该 NFA 中的所有状态都是终止状态，要判断一串符号串是否是可行的扫描线左侧结构，只需判断这个符号串能否被这个 NFA 接受。

ϵ: `START -> . E` 变为 `E -> . F`

ϵ: `START -> . E` 变为 `E -> . E + F`

E: `START -> . E` 变为 `START -> E .`

ϵ: `E -> . F` 变为 `F -> . F * G`

ϵ: `E -> . F` 变为 `F -> . G`

F: `E -> . F` 变为 `E -> F .`

消耗一个符号 = NFA中的一个非\epsilon边
没消耗成功则为空边

ϵ: `E -> . E + F` 变为 `E -> . F`

ϵ: `E -> . E + F` 变为 `E -> . E + F`

E: `E -> . E + F` 变为 `E -> E . + F`

+: `E -> E . + F` 变为 `E -> E + . F`

ϵ: `E -> E + . F` 变为 `F -> . F * G`

ϵ: `E -> E + . F` 变为 `F -> . G`

...

判断 `E + F + | ...` 是否是可行的扫描线左侧结构：

```
E + F + | ...

START -> . E
    E -> . F
    E -> . E + F
    F -> . G
    F -> . F * G
    G -> . ( E )
    G -> . ID                                   .
```

```
E + F + | ...

START -> E .
    E -> E . + F




                                                .
```

```
E + F + | ...

    E -> E + . F
    F -> . G
    F -> . F * G
    G -> . ID
    G -> . ( E )

                                                .
```

```
E + F + | ...

    E -> E + F .
    F -> F . * G




                                                .
```

```
E + F + | ...

    ( No possible state )
```

.

8 基于 follow 集合的判定法

定义

- `X` 是 Follow(`Y`) 的元素当且仅当 `... Y X ...` 是可能被完全规约的。
- `X` 是 First(`Y`) 的元素当且仅当 `X ...` 是可能被规约为 `Y` 的。
- 上述计算只考虑 `X` 是终结符的情形。

First 集合的计算

- 对任意终结符 `X` ，`X` 都是 First(`X`) 的元素。
- 对任意产生式 `Y -> Z ...` ，First(`Z`) 的元素都是 First(`Y`) 的元素。

Follow 集合的计算

- 对任意产生式 `U -> ... Y Z ...` ，First(`Z`) 是 Follow(`Y`) 的子集。
- 对任意产生式 `Z -> ... Y` ，Follow(`Z`) 是 Follow(`Y`) 的子集。

9 移入规约分析完整过程实例

`ID * (ID + ID)` 的移入规约分析：

- 初始：`| ID * ( ID + ID )`
- 移入：`ID | * ( ID + ID )`
- 规约：`G | * ( ID + ID )` ，因为 `ID * | ...` 不可行
- 规约：`F | * ( ID + ID )` ，因为 `G * | ...` 不可行
- 移入：`F * | ( ID + ID )` ，因为 `*` 不是 Follow(`E`) 中的元素
- 移入：`F * ( | ID + ID )`
- 移入：`F * ( ID | + ID )`
- 规约：`F * ( G | + ID )` ，因为 `F * ( ID + | ...` 不可行
- 规约：`F * ( F | + ID )` ，因为 `F * ( G + | ...` 不可行
- 规约：`F * ( E | + ID )` ，因为 `F * ( F + | ...` 不可行
- 移入：`F * ( E + | ID )`

- 移入：`F * ( E + ID | )`
- 规约：`F * ( E + G | )`，因为 `F * ( E + ID ) | ...` 不可行
- 规约：`F * ( E + F | )`，因为 `F * ( E + G ) | ...` 不可行
- 规约：`F * ( E | )`，因为另外两种方案扫描线左侧的结构都不可行
- 移入：`F * ( E ) |`
- 规约：`F * G |`
- 规约：`F |`
- 规约：`E |`，亦可看做 `E | EOF`
- 移入：`E EOF |`
- 规约：`START |`
- 语法分析结束

10 冲突

移入/规约冲突

- 如果在同一时刻既能进行移入操作，又能进行规约操作，这就称为一个移入/规约冲突（shift/reduce conflict）。

规约/规约冲突

- 如果在同一时刻能进行两种不同的规约操作，这就称为一个规约/规约冲突（reduce/reduce conflict）。

歧义与冲突的关系

- 给定一套上下文无关语法，如果移入规约分析中，一定不会出现移入/规约冲突，也不会出现规约/规约冲突，那么任何一串标记串都不会有歧义；
- 反之不一定。

```
A -> B X Y      A -> C X Z
B -> U V W      C -> U V W
```

比如，UVWXY在规约的时候，会产生冲突，但是本身是没有歧义的

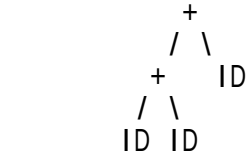
这是由于使用的是LALR(1)在做分析，1指的是只向扫描线右侧看一个字符判断follow集。
也可以LALR(2)，即看"XY"能否在W后面，这样就不会有冲突，不过在课程当中设计的语法以及分析都默认LALR(1)

对于有冲突/歧义的语法，可以增加优先级和结合性来消除

11 Bison 语法分析器

- 输入（.y 文件）：基于上下文无关语法与优先级、结合性的语法分析规则
- 输出（.c 文件）：用 C 语言实现的基于移入规约分析算法的语法分析器
- C 程序中不构造解析树
- 解析树的构造只与语法分析的过程相对应
- 语法分析中直接构造抽象语法树

解析树是一步一步地，E-F-G-...，
抽象语法树是



定义 While 语言的抽象语法树

lang.h

```
enum BinOpType {
  T_PLUS, T_MINUS, T_MUL, T_DIV, T_MOD,
  T_LT, T_GT, T_LE, T_GE, T_EQ, T_NE,
  T_AND, T_OR
};

enum UnOpType {
  T_UMINUS, T_NOT
};

enum ExprType {
  T_CONST, T_VAR,
  T_BINOP, T_UNOP,
  T_DEREF,
  T_MALLOC, T_RI, T_RC
};
```

```
struct expr {
  enum ExprType t;
  union {
    struct {unsigned int value; } CONST;
    struct {char * name; } VAR;
    struct {enum BinOpType op;
            struct expr * left;
            struct expr * right; } BINOP;
    struct {enum UnOpType op; struct expr * arg; } UNOP;
    struct {struct expr * arg; } DEREF;
    struct {struct expr * arg; } MALLOC;
    struct {void * none; } RI;
    struct {void * none; } RC;
  } d;
};
```

```
enum CmdType {
  T_DECL, T_ASGN, T_SEQ, T_IF, T_WHILE, T_WI, T_WC
};

struct cmd {
  enum CmdType t;
  union {
    struct {char * name; } DECL;
    struct {struct expr * left; struct expr * right; } ASGN;
    struct {struct cmd * left; struct cmd * right; } SEQ;
    struct {struct expr * cond;
            struct cmd * left; struct cmd * right; } IF;
    struct {struct expr * cond; struct cmd * body; } WHILE;
    struct {struct expr * arg; } WI;
    struct {struct expr * arg; } WC;
  } d;
};
```

构造抽象语法树的辅助函数 lang.h

```
struct expr * TConst(unsigned int value);
struct expr * TVar(char * name);
struct expr * TBinOp(enum BinOpType op,
                     struct expr * left,
                     struct expr * right);
struct expr * TUnOp(enum UnOpType op, struct expr * arg);
struct expr * TDeref(struct expr * arg);
struct expr * TMalloc(struct expr * arg);
struct expr * TReadInt();
struct expr * TReadChar();
```

```
struct cmd * TDecl(char * name);
struct cmd * TAsgn(struct expr * left, struct expr * right);
struct cmd * TSeq(struct cmd * left, struct cmd * right);
struct cmd * TIf(struct expr * cond,
                 struct cmd * left, struct cmd * right);
struct cmd * TWhile(struct expr * cond, struct cmd * body);
struct cmd * TWriteInt(struct expr * arg);
struct cmd * TWriteChar(struct expr * arg);
```

输出调试函数 lang.h

```
void print_binop(enum BinOpType op);
void print_unop(enum UnOpType op);
void print_expr(struct expr * e);
void print_cmd(struct cmd * c);
```

文件头 lang.y

```
%{
  #include <stdio.h>
  #include "lang.h"
  #include "lexer.h"
  int yyerror(char * str);
  int yylex();
  struct cmd * root;
%}
```

- 描述语法分析器所需的头文件、全局变量以及函数；
- 其中 `yylex()` 是 Flex 词法分析器提供的函数；
- `yyerror(str)` 是处理语法错误的必要设定；
- `root` 是语法分析最后生成的语法树根节点。

语法分析结果在 C 中的存储 lang.y

```
%union {
  unsigned int n;
  char * i;
  struct expr * e;
  struct cmd * c;
  void * none;
}
```

- 描述多种语法结构产生对应的语法分析结果；
- `n` 表示自然数常数的词法语法分析结果；
- `i` 表示变量的词法语法分析结果；
- `e` 表示表达式的语法分析结果；
- `c` 表示程序语句的语法分析结果；

终结符与非终结符 lang.y

```
%token <n> TM_NAT
%token <i> TM_IDENT
%token <none> TM_LEFT_BRACE TM_RIGHT_BRACE
%token <none> TM_LEFT_PAREN TM_RIGHT_PAREN
%token <none> TM_MALLOC TM_RI TM_RC TM_WI TM_WC
%token <none> TM_VAR TM_IF TM_THEN TM_ELSE TM_WHILE TM_DO
%token <none> TM_SEMICOL TM_ASGNOP TM_OR TM_AND TM_NOT
%token <none> TM_LT TM_LE TM_GT TM_GE TM_EQ TM_NE
%token <none> TM_PLUS TM_MINUS TM_MUL TM_DIV TM_MOD
%type <c> NT_WHOLE NT_CMD
%type <e> NT_EXPR
```

- `%token` 表示终结符，`%type` 表示非终结符；
- 尖括号内表示语义值的存储方式。

优先级与结合性 lang.y

```
%nonassoc TM_ASGNOP
%left TM_OR
%left TM_AND
%left TM_LT TM_LE TM_GT TM_GE TM_EQ TM_NE
%left TM_PLUS TM_MINUS
%left TM_MUL TM_DIV TM_MOD
%left TM_NOT
%left TM_LEFT_PAREN TM_RIGHT_PAREN
%right TM_SEMICOL
```

- 越先出现优先级越低；
- 同一行声明内的优先级相同。

上下文无关语法 lang.y

```
%%

NT_WHOLE:
  NT_CMD {
    $$ = ($1);
    root = $$;
  }
;
...
```

- 所有词法分析规则都放在一组 `%%` 中；

- 第一条语法分析规则描述初始符号对应的产生式
- 用 `$$` 表示产生式左侧符号的语义值；
- 用 `$1` 、 `$2` 等表示产生式右侧符号的语义值；

上下文无关语法（续）

```
NT_EXPR:
  TM_NAT {
    $$ = (TConst($1));
  }
| TM_LEFT_PAREN NT_EXPR TM_RIGHT_PAREN {
    $$ = ($2);
  }
| TM_MINUS NT_EXPR {
    $$ = (TUnOp(T_UMINUS,$2));
  }
| NT_EXPR TM_PLUS NT_EXPR {
    $$ = (TBinOp(T_PLUS,$1,$3));
  }
| NT_EXPR TM_MINUS NT_EXPR {
    $$ = (TBinOp(T_MINUS,$1,$3));
  }
...
```

Flex 与 Bison 的协同使用

```
%option noyywrap yylineno
%option outfile="lexer.c" header-file="lexer.h"
%{
#include "lang.h"
#include "parser.h"
%}

%%
0|[1-9][0-9]* {
    yylval.n = build_nat(yytext, yyleng);
    return TM_NAT;
}
"var" {
    return TM_VAR;
}
...
```

main.c （只打印结果）

```c
#include <stdio.h>
#include "lang.h"
#include "lexer.h"
#include "parser.h"

extern struct cmd * root;
void yyparse();

int main(int argc, char **argv) {
    yyin = stdin;
    yyparse();
    fclose(stdin);
    print_cmd(root);
}
```

编译 Makefile

```
lexer.c: lang.l
        flex lang.l
parser.c: lang.y
        bison -o parser.c -d -v lang.y
lang.o: lang.c lang.h
        gcc -c lang.c
parser.o: parser.c parser.h lexer.h lang.h
        gcc -c parser.c
lexer.o: lexer.c lexer.h parser.h lang.h
        gcc -c lexer.c
main.o: main.c lexer.h parser.h lang.h
        gcc -c main.c
main: lang.o parser.o lexer.o main.o
        gcc lang.o parser.o lexer.o main.o -o main
%.c: %.y
%.c: %.l
```

语法分析之后

- 可以基于 AST 进行进一步的合法性检查；
- 例如：判定 while 语言程序是否有变量重名，如果有则判定为非法程序；
- 例如：判定 while 语言程序是否所有使用过的变量名都有实现声明，如果没有则判定为非法程序；
- 等等

Bison 生成语法分析器的调试

- 编译时 `bison -v` 指令会将移入规约分析中的 DFA 信息输出到 parser.output 文件中。