语法分析（结尾部分）

1 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
```

2 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

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）

```
#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。