

CompilerC0

考虑到有些图片等内容显示不全，有pdf文档

github地址: <https://github.com/chen2511/CompilerC0>

CompilerC0

零、文件说明

0.1 统计信息

一、词法分析:

1.1 分析思路

1.1.1 正则表达式

1.1.2 NFA

1.1.3 DFA

1.2 数据结构

1.2.1 Token

1.2.2 TokenType

1.2.3 保留字表

1.2.4 状态

1.3 DFA编程思想

1.3.1、隐含状态

1.3.2、双重case

1.3.3、转换表

1.4 注意问题:

1.4.1、读入缓冲

1.4.2、保留字的识别

1.4.3、带符号数处理

分析:

处理方法

1.3.4、冲突解决

1.3.5、错误处理

二、语法分析

2.1 文法改造

2.1.1 二义性

2.1.2 左递归

2.1.3 回溯

2.1.4 计算FIRST和FOLLOW集

2.1.5 EBNF

2.2 编程思想

2.2.1 基本方法

2.2.2 重复和选择: 使用EBNF

可选结构

重复结构

2.3 数据结构

2.3.1 AST

2.3.2 回溯指针

2.4 关键函数

2.4.1 匹配终结符函数

2.4.2 非终结符过程

2.5 注意问题

2.5.1、公共因子

2.5.2、单元测试

2.5.3、错误处理

2.6 抽象语法树AST

- 2.6.1 设计抽象语法树
 - 2.6.2 数据结构
 - 2.6.3 编程验证
 - 2.6.4 输出到文件
- 三、语义分析
 - 3.1 符号表
 - 3.1.1 符号表结构
 - 3.1.2 相关操作
 - 3.2 语义分析程序
 - 3.2.1 符号表建立阶段
 - 3.2.2 语义检查
 - 可能出现的问题和优化：
- 四、中间代码生成
 - 4.1 表达式的翻译：
 - 4.2 四元式
- 五、目标代码生成
 - 5.0 理论基础
 - 5.1 寄存器分配思路
 - 1、一种较为简单的思路：（无寄存器分配）
 - 2、寄存器分配（FIFO）
 - 注意
 - 5.3 过程
 - 1、genasm:
 - 2、Func
 - 3、逐个分析
 - bug
 - 5.4 测试
 - Text03: 输入输出
 - Text04: 赋值语句：整形、字符，语句右边不考虑（由固定函数加载，单独测试）
 - Text05: 函数调用、返回语句
 - Text06: 算术表达式
 - Text07: 布尔表达式、条件循环
- 六、优化
 - 6.1 常量合并
 - 6.2 强度削弱
 - 6.3 删除公共子表达式
 - 6.4 寄存器分配
 - 6.4.1 基本思想：
 - 6.4.2 实现
- 七、错误处理
 - 7.1、错误识别
 - 7.2、跳读到可以正确分析的位置
 - 7.3 语法树错误屏蔽
- 八、bug修复
 - 1、asm生成阶段：scanf语句

零、文件说明

- global.h: 全局数据结构、全局变量
- scan.h:
- scan.cpp: 词法分析实现
- parser.h:
- parser.cpp: 语法分析：递归下降分析
- ast.h:
- ast.cpp: AST创建不同类型节点、输出AST

- symtab.h:
- symtab.cpp: 符号表基本操作：插入、查找、初始化
- semantic.h:
- semantic.cpp: 遍历AST，构建符号表、语义检查

0.1 统计信息

一、词法分析：

理论来自louden书和哈工大慕课、ppt和曾老师的课堂

文件：scan.h;scan.cpp

1.1 分析思路

理论：

文法》正则表达式》NFA》DFA》DFA最小化》编程实现

分为三种类型：保留字、特殊符号（运算符等等）、其他（数字、标识符、字符串、字符）

把保留字先看作标识符、再进行识别

所以，识别特殊符号、无符号数字、标识符、字符串、字符即可

1.1.1 正则表达式

数字：`[1-9][0-9]*|0`

标识符：`[_a-zA-Z][_a-zA-Z0-9]*`

运算符：

```
1 + - * /
2 && '||' !
3 < <= > >= != ==
4 = : , ; [ ] { } ( )
```

字符：`' [<加法运算符> | <乘法运算符> | <字母> | <数字>] '`

字符串：`" {十进制编码为32,33,35-126的ASCII字符} "`

1.1.2 NFA

1.1.3 DFA

比较简单。。。似乎不用什么操作了

1.2 数据结构

1.2.1 Token

```

1 typedef struct{
2     TokenType opType;
3     char * value;
4 }Token;

```

1.2.2 TokenType

枚举类型

```

1 typedef enum {
2     //关键字:
3     CHAR, CONST, ELSE, FALSE, FOR,           //0-4
4     IF, INT, MAIN, PRINTF, RETURN,           //5-9
5     SCANF, TRUE, VOID, WHILE,                //10-13
6     NUM, IDEN, LETTER, STRING,              //14-17:数字、标识符、字符、字
    符串
7     PLUS, MINU, MULT, DIV,                  //18-21:+ - * /
8     AND, OR, NOT,                          //22-24:&& || !
9     LSS, LEQ, GRE, GEQ, NEQ, EQL,           //25-30关系运算符: < <= > >=
    != ==
10    ASSIGN, COLON, COMMA, SEMICOLON,         //31-34: = : , ;
11    LBRACE, RBRACE, LBRACKET, RBRACKET,      //35-38:{ } [ ]
12    LPARENTHESES, RPARENTHESES,              //39-40:( )
13 }TokenType;

```

1.2.3 保留字表

```

1 const char* reservedwords[] = {
2     "case", "char", "const", "default", "else",
3     "false", "for", "if", "int", "main",
4     "printf", "return", "scanf", "switch", "true",
5     "void", "while"
6 };

```

1.2.4 状态

```

1 typedef enum {
2     STATE_START, STATE_NUM, STATE_ID, STATE_CHAR, STATE_STRING, STATE_DONE
3 }StateType;

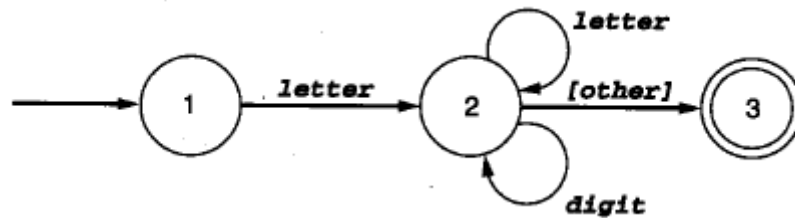
```

1.3 DFA编程思想

来自louden书的思路

1.3.1、隐含状态

注意 advance the input 表示读入一个符号



模拟这个DFA最早且最简单的方法是在下面的格式中编写代码：

```

{ starting in state 1 }
if the next character is a letter then
  advance the input;
  { now in state 2 }
  while the next character is a letter or a digit do
    advance the input; { stay in state 2 }
  end while;
  { go to state 3 without advancing the input }
  accept;
else
  { error or other cases }
end if;

```

这个代码使用代码中的位置（嵌套于测试中）来隐含状态，这与由注释所指出的一样。如果没有太多的状态（要求有许多嵌套层）且DFA中的循环较小，那么就合适了。类似这样的代码已被用来编写小型扫描程序了。但这个方法有两个缺点：首先它是特殊的，即必须用略微不同的方法处理各个DFA，而且规定一个用这种办法将每个DFA翻译为代码的算法较难。其次：当状态增多或更明确时，且当相异的状态与任意路径增多时，代码会变得非常复杂。现在来考虑一下在例2.9（图2-4）中接受注释的DFA，它可用以下的格式来实现：

```

{ state 1 }
if the next character is "/" then
  advance the input; { state 2 }
if the next character is "*" then
  advance the input; { state 3 }
  done := false;
  while not done do
    while the next input character is not "*" do
      advance the input;
    end while;
    advance the input; { state 4 }
  end while;
end if;

```

1.3.2、双重case

一个较之好得多的实现方法是：利用一个变量保持当前的状态，并将转换写成一个双层嵌套的case语句而不是一个循环。其中第1个case语句测试当前的状态，嵌套着的第2层测试输入字符及所给状态。例如，前一个标识符的DFA可翻译为程序清单2-1的代码模式。

程序清单2-1 利用状态变量和嵌套的case测试实现标识符DFA

```
state := 1; { start }
while state = 1 or 2 do
  case state of
    1: case input character of
        letter : advance the input;
            state := 2;
        else state := ... { error or other };
      end case;
    2: case input character of
        letter, digit: advance the input;
            state := 2; { actually unnecessary }
        else state := 3;
      end case;
  end case;
end while;
if state = 3 then accept else error ;
```

请注意这个代码是如何直接反映DFA的：转换与对state变量新赋的状态相对应，并提前输入（除了由状态2到状态3的“非消耗”转换）。

1.3.3、转换表

状态 \ 输入	/	*	其他	接受
1	2			不
2		3		不
3	3	4	3	不
4	5	4	3	不
5				是

现在若给定了恰当的数据结构和表项，就可以在一个将会实现任何 DFA的格式中编写代码了。下面的代码图解假设了转换被保存在一个转换数组T中，而T由状态和输入字符索引；先行输入的转换（即：那些在表格中未被括号标出的）是由布尔数组 Advance给出，它们也由状态和输入字符索引；而由布尔数组Accept给出的接受状态则由状态索引。下面就是代码图解：

```
state := 1;
ch := next input character;
while notAccept[state] and not error (state) do
  newstate := T[state,ch];
  if Advance [state,ch] then ch := next input char;
  state := newstate;
end while;
if Accept [state] then accept;
```

类似于刚刚讨论过的算法方法被称作表驱动（table driven），这是因为它们利用表格来引导算法的过程。表驱动方法有若干优点：代码的长度缩短了，相同的代码可以解决许多不同的问题，代码也较易改变（维护）了。但也有一些缺点：表格会变得非常大，使得程序要求使用的空间也变得非常大。实际上，我们刚描述过的数组中的许多空间都是浪费了的。因此，尽管

1.4 注意问题:

1.4.1、读入缓冲

缓冲的意思是，不是直接通过指针操作，而是，先读进来一个数组的数据，在数组中操作；

然后有单缓冲和双缓冲的区别；而北航是多了一个backup数组；

louden书中的例子，也是有一个缓冲数组lineBuf

1.4.2、保留字的识别

先看做标识符，再识别保留字；

保留字表，字典序查询

在上面的讨论或图 2-8 中的 DFA 都未包括保留字。这是因为根据 DFA 的观点，而认为保留字与标识符相同，以后再在接受后的保留字表格中寻找标识符是最简单的。当然，最长子串原

TINY 对保留字的识别是通过首先将它们看作是标识符，之后再在保留字表中查找它们来

China-pub.com

下载

第2章 词法分析

57

完成的。这在扫描程序中很平常，但它却意味着扫描程序的效率须依赖于在保留字表中查找过程的效率。我们的扫描程序使用了一种非常简便的方法——线性搜索，即按顺序从开头到结尾搜索表格。这对于小型表格不成问题，例如 TINY 中的表格，它只有 8 个保留字，但对于真实语言而言，这却是不可接受的，因为它通常有 30~60 个保留字。这时就需要一个更快的查找，而这又要求使用更好的数据结构而不是线性列表。假若保留字列表是按字母表的顺序写出的，那么就可以使用二分搜索。另一种选择是使用杂凑表，此时我们希望利用一个冲突性很小的杂凑函数。由于保留字不会改变（至少不会很快地），所以可事先开发出这样一个杂凑函数，它们在表格中的位置对于编译器的每一步运行而言都是固定的。人们已经确定了各种语言的最小完善杂凑函数（minimal perfect hash function），也就是说能够区分出保留字且具有最小数值的函数，因此杂凑表可以不大于保留字的数目。例如，如果只有 8 个保留字，则最小完善杂凑函数总会生成一个 0~7 的值，且每个保留字也会生成不同的值（参见“注意与参考”一节）。

在处理保留字时，另一个选择是使用储存标识符的表格，即：符号表。在过程开始之前，将所有的保留字整个输入到该表中并且标上“保留”（因此不允许重新定义）。这样做的好处在于只要求一个查找表。但在 TINY 扫描程序中，直到扫描阶段之后才构造符号表，因此这个方法对于这种类型的设计并不合适。

保留字表；

如果比较大，就用二分查找

1.4.3、带符号数处理

+1+1

第二个 +1 和第一个 +1 是有区别的，词法分析阶段识别带符号数，最长字符串原则，会有问题

分析：

带符号整数出现于变量定义时，这个地方不会存在问题，语法分析时处理；

另一个是位于表达式中：

数学上的形式：其中要么带符号数要么在开头，要么有括号；其他情况数字都是无符号的

因为表达式开头有符号定义，可以忽略这个；

而对于带符号数在表达式内部，肯定实在括号内部，所以又可以递归回去，用开头的符号；

所以总结起来也是语法分析阶段处理

还有就是布尔表达式的地方，也是单独分析

处理方法

放到语法分析中处理带符号数

1.3.4、冲突解决

1.3.5、错误处理

二、语法分析

2.1 文法改造

这一部分 花了大量时间，需要验证文法的可行性；以及编程的可行性

2.1.1 二义性

- if else
- 表达式

解决办法：改造文法和EBNF

未发现其他二义性的地方

2.1.2 左递归

可以较为容易地判断出：没有直接和间接左递归。

2.1.3 回溯

通过改造文法（简单的改造，提取公共左因子；但为了直观、更方便处理，有的不改造了）或者 算法处理（if else 进一步判断，判断后回溯，感觉思想类似LL(n)了，事实上就没有消除回溯）

2.1.4 计算FIRST和FOLLOW集

可以用于选择、重复的case判断

2.1.5 EBNF

EBNF更适合递归下降分析法

例如：

```
40. <布尔表达式> ::= <布尔项> { '||' <布尔项> }
```

```
41. <布尔项> ::= <布因子>{ '&&' <布因子> }
```

```
42. <布因子> ::= false | true | ! <布因子> | '(' <布尔表达式> ')' | <条件因子> [<条件运算符> <条件因子>]
```

```
43. <条件因子> ::= <标识符>['[' <算术表达式> ']'] | <整数> | <字符> | <有返回值函数调用语句>
```

2.2 编程思想

2.2.1 基本方法

4.1.1 递归下降分析的基本方法

递归下降分析的概念极为简单：将一个非终结符 A 的文法规则看作将识别 A 的一个过程的定义。 A 的文法规则的右边指出这个过程的代码结构：一个选择中的终结符与非终结符序列与

相匹配的输入以及对其他过程的调用相对应，而选择与在代码中的替代情况（`case`语句和`if`语句）相对应。

例如，考虑前一章的表达式文法：

```
exp  $\rightarrow$  exp addop term | term
addop  $\rightarrow$  + | -
term  $\rightarrow$  term mulop factor | factor
mulop  $\rightarrow$  *
factor  $\rightarrow$  ( exp ) | number
```

及`factor` 的文法规则，识别`factor` 并用相同名称进行调用的递归下降程序过程可用伪代码编写如下：

```
procedure factor ;
begin
  case token of
    ( : match( ) ;
      exp ;
      match( ) ;
    number :
      match (number) ;
    else error ;
  end case ;
end factor ;
```

在这段伪代码中，假设有一个在输入中保存当前下一个记号的`token`变量（以便这个例子使用先行的一个符号）。另外还假设有一个`match`过程，它用它的参数匹配当前的下一个记号。如果成功则前移，如果失败就表明错误：

```
procedure match ( expectedToken ) ;
begin
  if token = expectedToken then
    getToken ;
  else
    error ;
  end if ;
```

case选择不同的，遇到终结符，就match；非终结符就调用函数

2.2.2 重复和选择：使用EBNF

这里面的判断条件是根据 first集的 或者 空语句的时候，就要follow集

总之就是一个要求：根据当前token，进入一个确定的分支，没有回溯

可选结构

可将它翻译成以下过程

```
procedure ifStmt ;
begin
  match (if) ;
  match ( ( ) ) ;
  exp ;
  match ( ) ;
  statement ;
  if token = else then
    match (else) ;
    statement ;
  end if ;
end ifStmt ;
```

在这个例子中，不能立即区分出文法规则右边的两个选择（它们都以记号 **if** 开始）。相反地，我们必须直到看到输入中的记号 **else** 时，才能决定是否识别可选的 **else** 部分。因此，**if** 语句的代码与EBNF

$$if_stmt \rightarrow if \ (\ exp \) \ statement \ [\ else \ statement \]$$

匹配的程度比与BNF的匹配程序要高，上面的EBNF的方括号被翻译成ifStmt的代码中的一个测试。实际上，EBNF表示法是为更紧密地映射递归下降分析程序的真实代码而设计的，如果使用的是递归下降程序，就应总是将文法翻译成EBNF。另外还需注意到即使这个文法有二义性（参见前一章），编写一个每当在输入中遇到 **else** 记号时就立即匹配它的分析程序也是很自然的。这与最近嵌套的消除二义性的规则精确对应。

重复结构

解决的办法是使用EBNF规则

$$exp \rightarrow term \{ \ addop \ term \ }$$

花括号表示可将重复部分翻译到一个循环的代码中，如下所示：

```
procedure exp ;
begin
  term ;
  while token = + or token = - do
```

```

    match (token);
    term;
end while;
end exp;

```

相似地，*term*的EBNF规则：

$$term \rightarrow factor \{ mulop factor \}$$

就变成代码

```

procedure term;
begin
    factor;
    while token = * do
        match (token);
        factor;
    end while;
end term;

```

在这里当分隔过程时，删去了非终结符 *addop* 和 *mulop*，这是因为它们仅有匹配算符功能：

$$addop \rightarrow + \mid -$$

$$mulop \rightarrow *$$

这里所做的是 *exp* 和 *term* 中的匹配。

这个代码有一个问题：由花括号（和原始的BNF中的显式）表示的左结合是否仍然保留。例如，假设要为本书中简单整型算术的文法编写一个递归下降程序计算器，就可通过在循环中轮转来完成运算，从而就保证了该运算是左结合（现在假设分析过程是返回一个整型结果的函数）：

```

function exp : integer;
var temp : integer;
begin
    temp := term;
    while token = + or token = - do
        case token of
            + : match (+);
                temp := temp + term;
            - : match (-);
                temp := temp - term;
        end case;
    end while;
    return temp;
end exp;

```

2.3 数据结构

2.3.1 AST

未打算使用抽象语法树，因为发现可以不使用抽象语法树，直接转IR（三地址码等），可能建立起来还会多此一举；而且考虑到未发现统一的语法树结构定义，而且文法不同，也要做出改变。

但可能不直观，不知道分析结果。所以调试代码的时候，只能写一部分代码，然后进行针对性的单元测试。

2.3.2 回溯指针

```
1 | static int flashBackIndex;
```

2.4 关键函数

2.4.1 匹配终结符函数

```
1 | // 匹配 期待的 token; 否则报错
2 | // 读取下一个token
3 | static void match(TokenType expectToken)
```

2.4.2 非终结符过程

```
1 | //为每一个非终结符创建一个函数
2 | void program();
3 |
4 | void constDeclaration();
5 | void constDefine();
6 | void varDeclaration();
7 | void varDefine();
8 | void typeID();
9 | void functionDefinitionWithReturn();
10 | void DeclarationHead();
11 | void functionDefinitionWithoutReturn();
12 | void paraTable();
13 | void complexSentence();
14 | void mainFunction();
15 |
16 | void signedNum();
17 |
18 | void statementSequence();
19 | void statement();
20 | void assignStatement();
21 | void ifStatement();
22 | void loopStatement();
23 | void callWithReturn();
24 | void callWithoutReturn();
25 | void valueParaTable();
26 | void readStatement();
27 | void writeStatement();
28 | void returnStatement();
29 |
30 | void exp();
31 | void term();
32 | void factor();
33 | void boolExp();
34 | void boolTerm();
35 | void boolFactor();
```

2.5 注意问题

2.5.1、公共因子

```
1 13.<程序> ::= [<常量说明>] [<变量说明>] {<有返回值函数定义>|<无返回值函数定义>}<主函数>
```

这一部分为了 文法上的简便，没有进行公共因子的提取；

选择类似LL(n)的方法，一次性往后看好多

分三段，设flag

2.5.2、单元测试

- 常、变量说明

类型测试、多条语句测试、是否函数定义测试

- 函数定义（语句列为空）

函数定义选择测试、结构测试、参数表、复合语句测试（实际上空语句）

- 语句列、语句
- 表达式：算术、布尔

2.5.3、错误处理

```
source file open successfully!
error in varDefine() :in line 6 ,tokenType 14 value: 0 ;array size is 0
parser error in line 14 : expect Token 31, but 32 value: ,
error in signedNum() :in line 14 ,tokenType 32 value: ,
parser error in line 14 : expect Token 31, but 33 value: ;
error in signedNum() :in line 14 ,tokenType 33 value: ;
parser error in line 15 : expect Token 31, but 33 value: ;
error in signedNum() :in line 15 ,tokenType 33 value: ;
parser error in line 16 : expect Token 31, but 32 value: ,
parser error in line 16 : expect Token 16, but 32 value: ,
parser error in line 16 : expect Token 31, but 33 value: ;
parser error in line 16 : expect Token 16, but 33 value: ;
parser error in line 17 : expect Token 31, but 33 value: ;
parser error in line 17 : expect Token 16, but 33 value: ;
```

发现：

要是少了一些token，继续往下匹配，还能成功继续；（token不动，分析继续）

但是有时候，比如int写成in，有时候死活下不去

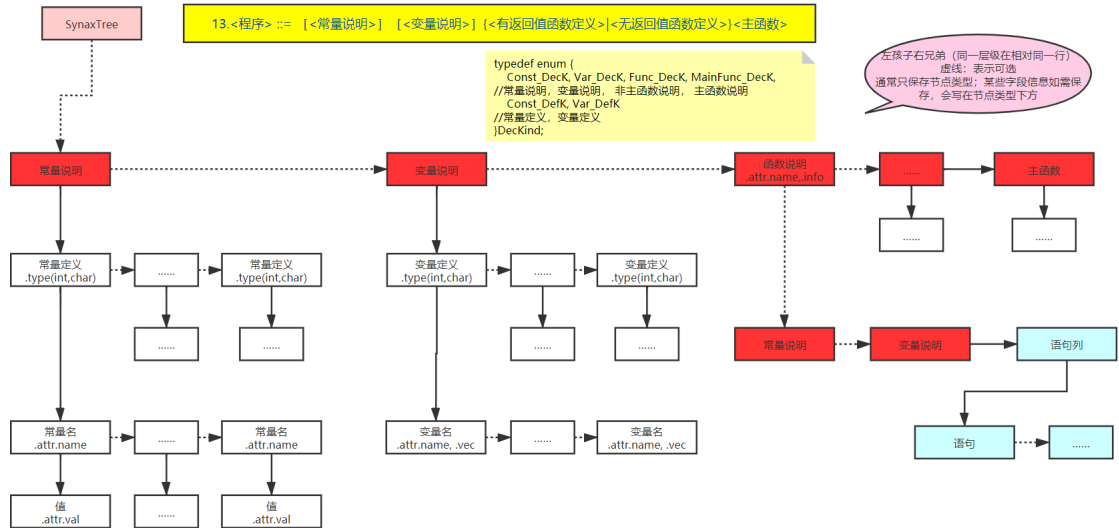
2.6 抽象语法树AST

因为一开始没有这个打算，这是临时加的，所以放到最后了。

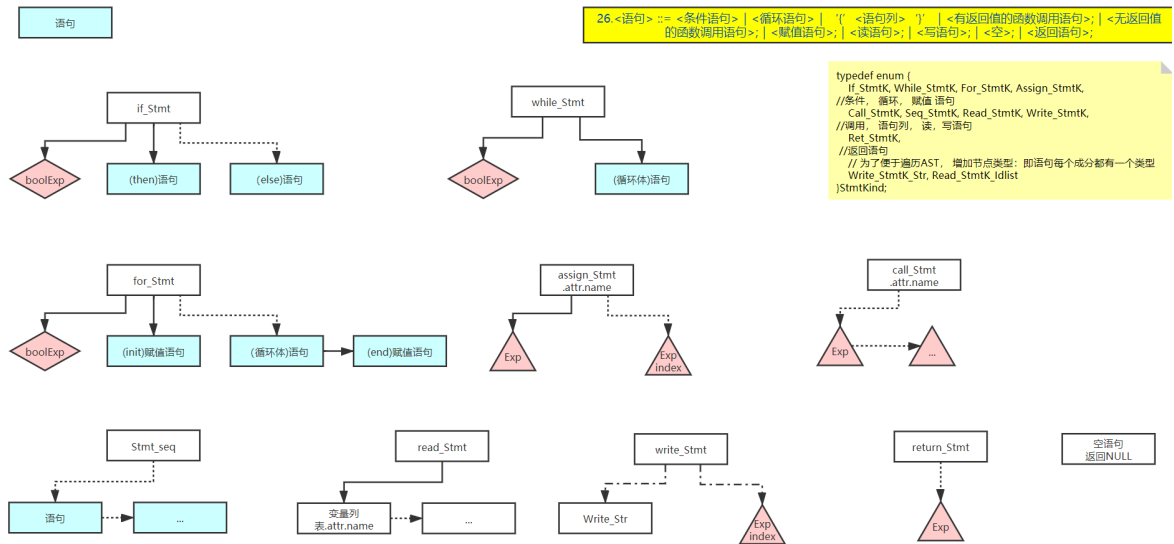
2.6.1 设计抽象语法树

先根据 语法定义手动画出预期抽象语法树；

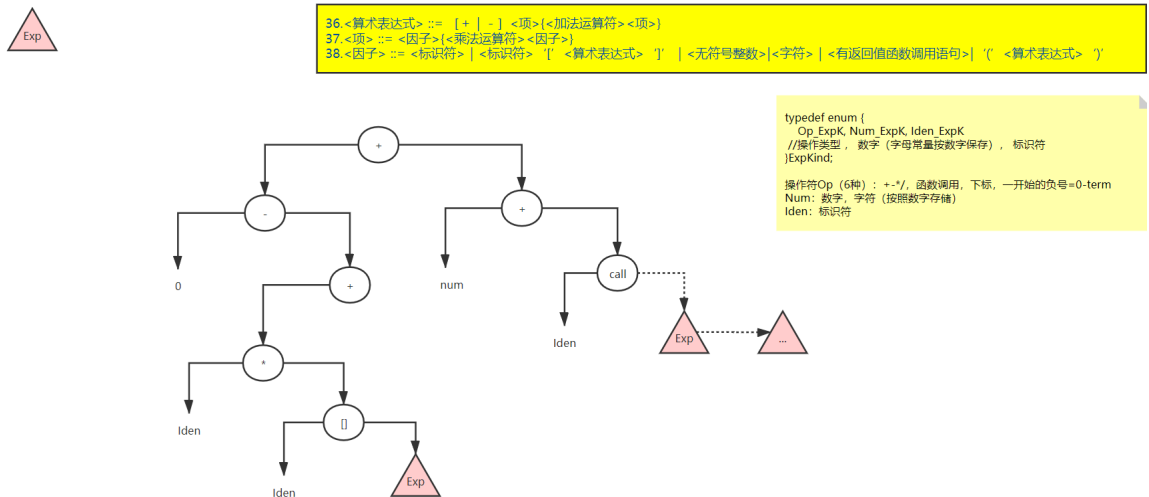
高清图可查看svg



语句:



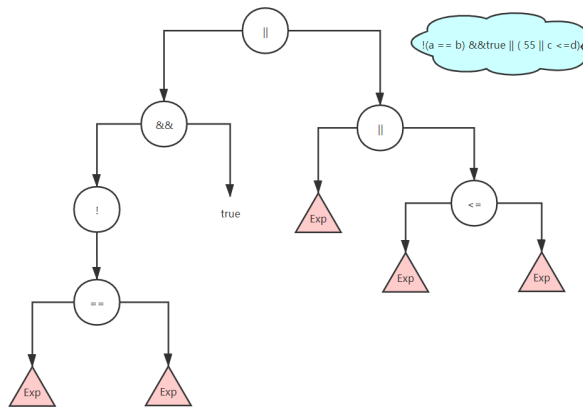
表达式



布尔表达式



```
39 <布尔表达式> ::= <布尔项> | '!' <布尔项>
40 <布尔项> ::= <因子> | '&&' <因子>
41 <因子> ::= false | true | ! <因子> | ( ' <布尔表达式> ' ) | <条件因子> | <条件运算符> <条件因子>
42 <条件因子> ::= <标识符> [ '!' <算术表达式> '!' ] | <无符号整数> | <字符> | <有返回值的函数调用语句>
```



```
typedef enum {
    Op_BoolEK,
    //布尔表达式操作: 与或非;
    Const_BoolEK, ConOp_BoolEK
    //布尔常量, 条件运算符
    //条件因子 用的是表达式节点, 与表达式构造方法相同, 但是在另一
    //过程里实现 (语法定义中不是表达式, 但实际是)
} BoolExpKind;
```

2.6.2 数据结构

然后结合图和文法, 设计数据结构;

大体说明: 总的有四种类型: Deck, StmtK, ExpK, BoolExpK (NodeKind nodekind; 字段保存)

具体类型: union确定

```
1  #define MAX_TREENODE_CHILD_NUM 3 // 最大孩子节点数
2  #define MAX_PARAMETER_NUM 8 // 最大参数个数
3
4  typedef enum {
5      Deck, StmtK, ExpK, BoolExpK //声明, 语句, 表达式,
6      //布尔表达式
7  } NodeKind;
8
9  typedef enum {
10     Const_Deck, Var_Deck, Func_Deck, MainFunc_Deck, //常量说明, 变量说
11     //明, 非主函数说明, 主函数说明
12     Const_DefK, Var_DefK //常量定义, 变量定
13     //义
14 } DeckKind;
15
16 typedef enum {
17     If_StmtK, while_StmtK, For_StmtK, Assign_StmtK, //条件, 循环, 赋
18     //值 语句
19     Call_StmtK, Seq_StmtK, Read_StmtK, Write_StmtK, //调用, 语句列,
20     //读, 写语句
21     Ret_StmtK, //返回语句
22     // 为了便于遍历AST, 增加节点类型: 即语句每个成分都有一个类型
23     Write_StmtK_Str,
24     Read_StmtK_Idlist
25 } StmtKind;
26
27 typedef enum {
28     Op_ExpK, Num_ExpK, Iden_ExpK //操作类型, 数字
29     // (字母常量按数字保存), 标识符
30 } ExpKind;
31
32 typedef enum {
33     Op_BoolEK, //布尔表达式操作:
34     //与或非;
```



```

28     Const_BoolEK, ConOp_BoolEK                                //布尔常量， 条件
运算符
29     //ConFac_BoolEK                                           // 条件因子，
其实就是简化了一点的表达式
30     // 条件因子 用的是表达式节点，与表达式构造方法相同，但是在另一个过程里实现（语法定义
中不是表达式，但实际是）
31 }BoolExpKind;
32
33 // 类型信息： 可以是定义时 保存的类型信息；也可以用于 检验表达式中类型是否匹配
34 typedef enum {
35     T_VOID, T_INTEGER, T_CHAR
36 }Type;
37
38 // 函数信息：返回类型和参数表；也可以链接到符号表中
39 typedef struct {
40     Type rettype;
41     struct {
42         Type ptype;
43         char* pname;
44     }paratable[MAX_PARAMETER_NUM];
45     int paranum;
46 }FuncInfo;
47
48 // AST 的节点：左孩子右兄弟的树形结构；但表达式部分 又是二叉树结构
49 typedef struct TreeNode {
50     struct TreeNode* child[MAX_TREENODE_CHILD_NUM];    // 左孩子，最多三个，通
常只有一个，特定语句有多个
51     struct TreeNode* sibling;                            // 右兄弟
52     int lineno;                                         // 错误报告行号
53
54     NodeKind nodekind;                                // 节点类型
55     union {
56         DeckKind dec;
57         StmtKind stmt;
58         ExpKind exp;
59         BoolExpKind bexp;
60     }kind;                                             // 节点具体类型
61
62     union {
63         TokenType op;                                // 操作类型：通常是表达
式中
64         int val;                                     // NUM的值
65         char cval;                                  // Char 型 值
66         char* name;                                // Id 的值，也可以是函
数名，Str的值
67         bool bval;                                  // bool 常量
68         char* str;                                  // String 类型
69     }attr;                                           // 节点属性
70     int vec;                                         // 变量定义阶段设置：数组长
度，不是数组就是-1；
71     Type type;                                       // 常、变量定义，类型说明
和 表达式类型检查
72
73     FuncInfo* pfuncinfo;                             // 函数信息：返回类型和参数
表；也可以链接到符号表中
74
75     FuncInfo* pfuncinfo;                             // 函数定义阶段设置：函数信
息；或者是函数调用阶段的参数表
76 }TreeNode;

```

之后再验证数据结构

2.6.3 编程验证

```
1  #pragma once
2  #ifndef AST_H
3  #define AST_H
4
5  #include "global.h"
6
7  // 创建声明节点：包括常量、变量、函数等等说明定义，传入具体声明类型，返回节点指针
8  TreeNode* newDecNode(DecKind kind);
9
10 // 创建语句节点：传入具体声明类型，返回节点指针
11 TreeNode* newStmtNode(StmtKind kind);
12
13 // 创建表达式节点：传入具体声明类型，返回节点指针
14 TreeNode* newExpNode(ExpKind kind);
15
16 // 创建布尔表达式节点：传入具体声明类型，返回节点指针
17 TreeNode* newBoolExpNode(BoolExpKind kind);
18
19 // 定义函数时，保存函数信息：返回类型和参数表
20 // FuncInfo* newFuncInfo(Type t);
21
22 // 拷贝token中的字符串到动态分配的空间
23 char* copyString(char* s);
24
25 // 输出抽象语法树到文件
26 void printAST(TreeNode* tree);
27
28 #endif // !AST_H
```

表达式策略：计算时只有三种节点：操作（+*/数组、call、取负（改为0-term））、数字（字符也当作数字运算）、标识符

如果是赋值会进行截断处理；比较时都是整形

终于明白为什么参数个数一致即可；int转char截断即可；char转int扩展；所以只需要记录个数

2.6.4 输出到文件

主要工作是遍历AST

主要思想是：（深度优先思想）先遍历所有孩子节点，再遍历兄弟节点。

重大bug：不要用return，不然兄弟节点会被跳过；break即可

例子：

```
1  const int gc_a = 22;
2  const int gc_b = -22, gc_c = 0;
3  const char gc_d = 'z', gc_e = 'x';
4
5  int g_a, g_b;
```

```

6 char g_c[0], g_d[5];
7 int g_e;
8
9 void f_a(){
10     a1 = -5 * func() * ( a2 + '5') + a3[3] + 599;
11 }
12
13 int f_b(int a, char b){
14     const int a = 1, b = 2;
15     const int c = 2;
16     const char d = '2', e = '3';
17     const char f = '4';
18
19
20     int g,h;
21     int i;
22     char j;
23     char k,l;
24
25     return (a + b);
26
27 }
28
29 void f_c(int c, int d){
30     int c;
31
32     ;
33 }
34
35 void main()
36 {
37     const int a = 1, b = 2;
38     const int c = 2;
39     const char d = '2', e = '3';
40     const char f = '4';
41
42
43     int g,h;
44     int i;
45     char j[30];
46     char k,l;
47
48
49     k = a_2 * 55 * ( -1 + k[2] + g_a + j - 'g' ) + f_b(dddd, dddd2) + f_b()
    * f_c() * f_undefine();
50
51     f_un();
52     f_c(g, h);
53     f_c(g);
54
55     scanf(a);
56     scanf(g, h);
57     scanf(k);
58
59
60     if( ( true || 8 )&& ! false && true || 88 <= '8' && f_b() != 22){
61         printf("aaaaaa");
62     }

```

```

63
64     while(true){
65         ;
66     }
67
68     for(a=1; a == eee; b=3)
69         ;
70
71     printf("aaaa11",a);
72     printf("aaaa12221%c");
73     printf(acc);
74
75     if(false)
76     if(____true){
77
78     }
79     else{
80
81     }
82
83
84     return ;
85 }

```

```

1
2  ----- Const Declaration -----
3
4  Type: 1 (VOID,INT,CHAR) list:
5  ID name:      gc_a      value:  22
6  Type: 1 (VOID,INT,CHAR) list:
7  ID name:      gc_b      value: -22
8  ID name:      gc_c      value:  0
9  Type: 2 (VOID,INT,CHAR) list:
10 ID name:      gc_d      value:  z
11 ID name:      gc_e      value:  x
12
13 ----- Var Declaration -----
14 Type: 1 (VOID,INT,CHAR) list:
15 ID name: g_a
16 ID name: g_b
17
18 Type: 2 (VOID,INT,CHAR) list:
19 ID name: g_c
20 ID name: g_d
21
22 Type: 1 (VOID,INT,CHAR) list:
23 ID name: g_e
24
25
26 >>>>>>>>>> Function Declaration >>>>>>>>>>
27   Function Name: f_a
28   Function Info: Return Type(VOID,INT,CHAR):0
29   ParaTable(VOID,INT,CHAR):
30
31 Stmt Sequence>>>

```

```

32 Assign to: a1
33   0 - 5 * func () * a2 + 53 + a3 [] 3 + 599
34 <<<Stmt Sequence End
35 
36 
37 >>>>>>>>>> Function Declaration >>>>>>>>>>>>>>
38   Function Name: f_b
39   Function Info: Return Type(VOID,INT,CHAR):1
40                 ParaTable(VOID,INT,CHAR): 1 a    1 a
41 
42 ----- Const Declaration -----
43 
44 Type: 1 (VOID,INT,CHAR) list:
45 ID name:      a          value: 1
46 ID name:      b          value: 2
47 Type: 1 (VOID,INT,CHAR) list:
48 ID name:      c          value: 2
49 Type: 2 (VOID,INT,CHAR) list:
50 ID name:      d          value: 2
51 ID name:      e          value: 3
52 Type: 2 (VOID,INT,CHAR) list:
53 ID name:      f          value: 4
54 
55 ----- Var Declaration -----
56 Type: 1 (VOID,INT,CHAR) list:
57 ID name: g
58 ID name: h
59 
60 Type: 1 (VOID,INT,CHAR) list:
61 ID name: i
62 
63 Type: 2 (VOID,INT,CHAR) list:
64 ID name: j
65 
66 Type: 2 (VOID,INT,CHAR) list:
67 ID name: k
68 ID name: l
69 
70 Stmt Sequence>>>
71 Ret Stmt:
72   a + b
73 <<<Stmt Sequence End
74 
75 >>>>>>>>>> Function Declaration >>>>>>>>>>>>>>
76   Function Name: f_c
77   Function Info: Return Type(VOID,INT,CHAR):0
78                 ParaTable(VOID,INT,CHAR): 1 c    1 c
79 
80 ----- Var Declaration -----
81 Type: 1 (VOID,INT,CHAR) list:
82 ID name: c
83 
84 Stmt Sequence>>>
85 <<<Stmt Sequence End

```

```

90
91 *****
92 ***** Main Function Declaration *****
93 *****
94
95 ----- Const Declaration -----
96
97 Type: 1 (VOID,INT,CHAR) list:
98 ID name:      a      value: 1
99 ID name:      b      value: 2
100 Type: 1 (VOID,INT,CHAR) list:
101 ID name:      c      value: 2
102 Type: 2 (VOID,INT,CHAR) list:
103 ID name:      d      value: 2
104 ID name:      e      value: 3
105 Type: 2 (VOID,INT,CHAR) list:
106 ID name:      f      value: 4
107
108 ----- Var Declaration -----
109 Type: 1 (VOID,INT,CHAR) list:
110 ID name: g
111 ID name: h
112
113 Type: 1 (VOID,INT,CHAR) list:
114 ID name: i
115
116 Type: 2 (VOID,INT,CHAR) list:
117 ID name: j
118
119 Type: 2 (VOID,INT,CHAR) list:
120 ID name: k
121 ID name: l
122
123
124 Stmt Sequence>>>
125
126 Assign to: k
127   a_2 * 55 * 0 - 1 + k [] 2 + g_a + j - 103 + f_b ()
   dddd + f_b () * f_c () * f_undefine ()
128 Call Stmt: Function: f_un
129 Paras:
130
131 Call Stmt: Function: f_c
132 Paras: g h
133
134 Call Stmt: Function: f_c
135 Paras: g
136
137 Read Stmt:
138 Read ID: a
139
140 Read Stmt:
141 Read ID: g
142 Read ID: h
143
144 Read Stmt:
145 Read ID: k
146

```

```

147 If :
148   || && && || bool(1) 8 || bool(0) bool(1) && <= 88 56 != f_b
    () 22
149 Stmt Sequence>>>
150
151 Write Stmt:
152 Write Str: aaaaaa
153
154 <<<Stmt Sequence End
155
156 While Stmt:
157   bool(1)
158   Stmt Sequence>>>
159
160 <<<Stmt Sequence End
161
162 For Stmt:
163
164 Assign to: a
165   1 == a eee
166 Assign to: b
167   3
168 Write Stmt:
169 Write Str: aaaa11
170   a
171 Write Stmt:
172 Write Str: aaaa12221%c
173
174 Write Stmt:
175   acc
176 If :
177   bool(0)
178 If :
179   ____true
180 Stmt Sequence>>>
181
182 <<<Stmt Sequence End
183
184 Stmt Sequence>>>
185 <<<Stmt Sequence End
186 Ret Stmt:
187
188 <<<Stmt Sequence End

```

三、语义分析

一遍的编译器中，词法分析、语法分析、语义分析是同时进行的

我的顺序：在生成抽象语法树之后，再遍历抽象语法树，构建符号表，进行语义分析（事实上技术上的难度没什么差别，都很简单，做的事情基本一样）

语义分析包括构造符号表、记录声明中建立的名字的含义、在表达式和语句中进行类型推断和类型检查以及在语言的类型规则作用域内判断它们的正确性。

语义分析阶段的一个主要工作是符号表的管理，其作用是将标识符映射到它们的类型和储存位置。

3.1 符号表

3.1.1 符号表结构

符号表的结构选择：分级hash链表；

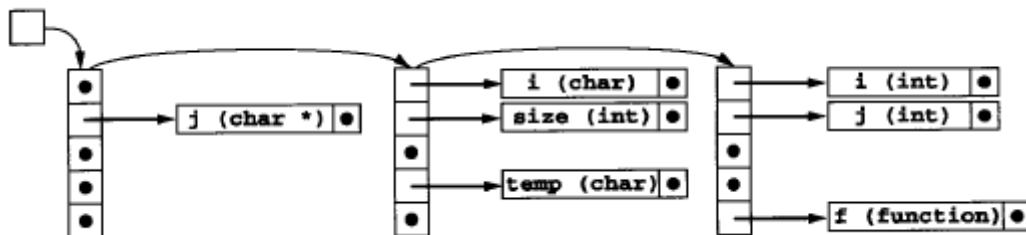
分级表示不同作用域；当有同名标识符时，选择作用域最近的；以链表的方式解决冲突问题

```
1 #define SYMBOL_TABLE_SIZE 211
2
3 typedef enum IDType {
4     Const_ID, Var_ID, Para_ID, Func_ID
5 } IDType;
6
7 // 符号表中的每一项
8 typedef struct Symbol {
9     char* name;           // 标识符名字
10    IDType type;           // ID类型: const, var, para, function
11    Type valueType;        // ID的类型值: 常变量的类型、参数类型、函数返回类型
12    int value;             // 常量定义值           只有常数定义才会传入
13    int address;           // 内存地址
14    int vec;               // 数组大小, 不是数组为-1;           只有定义数组时, 才会传入
15    FuncInfo* pfinfo;      // 函数信息, AST中已有, 拷贝即可;   只有函数定义是, 才会传进, 否则NULL
16    struct Symbol* next;   // 有相同hash值时, 下一条
17 } Symbol, * SymbolList;
18
19 typedef struct SymTab {
20     SymTab* next;         // 多张表: 指向下一张表
21     char* fname;
22     SymbolList hashTable[SYMBOL_TABLE_SIZE];
23 } SymTab;
```

此处有些设计不好，枚举类型和成员名字容易混淆

一张全局表，每个函数对应一张符号子表

形式如下：



3.1.2 相关操作


```

1 // 初始化一张 单表；每个函数一张表、全局一张表
2 SymTab* initSimpleSymTable(char* name);
3
4 // 返回是否插入成功，如空间不足、重复定义； 后面并没有用到返回值，内部直接错误处理
5 bool insert_SymTab(bool isGlobal, char* name, IDType type, Type valuetype,
6 int value, int vec = -1, FuncInfo* p = NULL);
7 // value字段，只有常量定义才会赋初值
8 // adress字段由文件内部静态变量控制，一旦重新函数定义，指针变0，重新计算相对地址
9 // 数组标志默认-1；只有数组(变量定义)定义时才会是 其他值
10 // 函数信息默认为空，只有函数定义时，才会赋值
11 // 返回符号节点指针、未定义返回空指针；
12 Symbol* lookup_SymTab(char* name);
13 // 遵循原则：先查子表，再查全局表（作用域最近的）

```

全局常、变量定义，函数定义将会放到全局表中；参数、局部变量放在函数表中

根据 `isGlobal` 标志 确定插入全局表还是 函数表

```

1 static int g_adress = 0;
2 static int f_adress = 0;

```

还设置了两个静态变量来保存全局表和函数表的标识符地址

关于同名函数：不允许

3.2 语义分析程序

3.2.1 符号表建立阶段

要将声明的标识符插入到符号表，包括四种类型：常量、变量、参数、函数定义

插入的位置分为：全局表还是函数表

常量变量：建立isGlobal标记，只有最开始声明时，会是true；一旦开始函数定义，变成false，只能进入函数表

参数：统一false，进入函数表

函数定义：进入全局表，true（main函数也进去，可以递归main函数），函数不允许同名

可能报的错误：（输出显示）

内存不足

变量、函数重复定义

```

source file open successfully!
error in varDefine() :in line 6 ,tokenType 14 value: 0 ;array size is 0
Error in lineno 14 : a had been defined
Error in lineno 14 : b had been defined
Error in lineno 30 : c had been defined

```

3.2.2 语义检查

遵循原则：先查子表，再查全局表（作用域最近的）

可能出现的问题和优化:

表达式中:

表达式中有几种节点: 常量、标识符 (函数调用看作Op)、Op (+-*/、Call、[])

- 标识符: 未定义就使用, 是数组不进行下标运算
- CallOp: 函数未定义使用, 不是有返回值调用, 参数个是否匹配
- 数组Op: 不是数组, 进行下标运算;

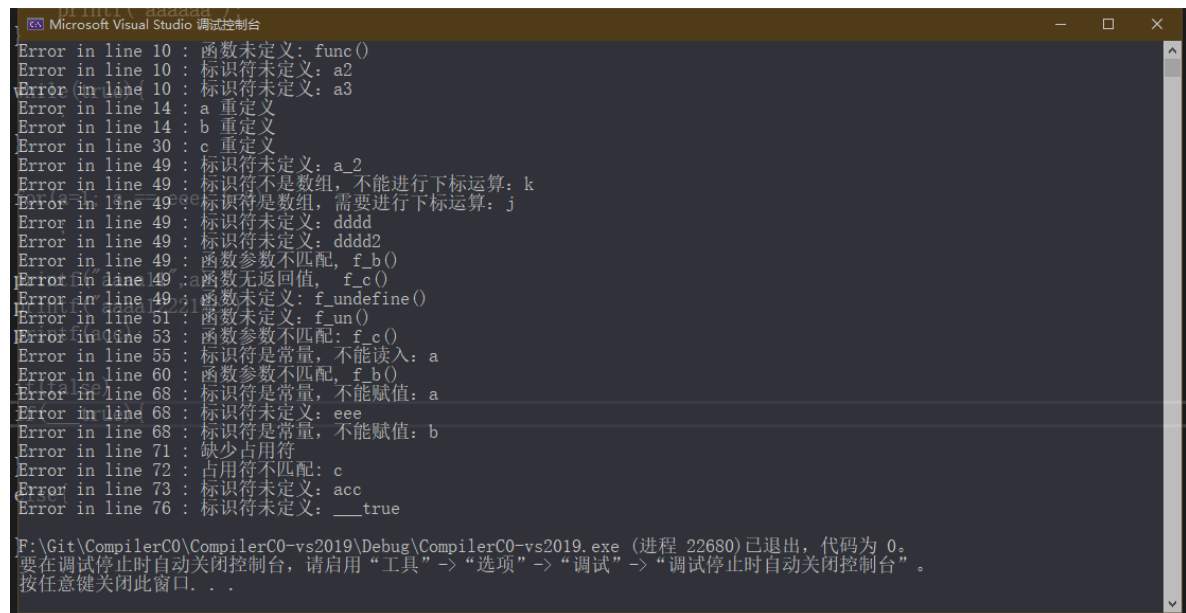
运算符两侧操作数是否合法, 比如int和bool类型相加, 由于只有char和int, 暂不考虑

语句中:

- 赋值语句: 检查是否变量 (常量不能赋值)、不是数组进行下标运算、是数组不进行下标运算、标识符未定义就使用
- 函数调用语句: 标识符未定义、是否是函数、参数是否匹配、
- 读语句: 标识符未定义、不能是常量
- 写语句: 占用符与表达式不匹配 (是否有)

- 赋值语句左侧类型确定 (优化AST: 保存在语句节点Type字段)
- 函数调用语句: 参数类型优化 (保存在参数表达式节点Type字段)
- Write语句中: 占用符类型检查 (优化AST: 保存在语句节点Type字段)

表达式都看作是int型运算, 只有语句中需要时根据左值类型进行转换



```
Microsoft Visual Studio 调试控制台
Error in line 10 : 函数未定义: func()
Error in line 10 : 标识符未定义: a2
Error in line 10 : 标识符未定义: a3
Error in line 14 : a 重定义
Error in line 14 : b 重定义
Error in line 30 : c 重定义
Error in line 49 : 标识符未定义: a_2
Error in line 49 : 标识符不是数组, 不能进行下标运算: k
Error in line 49 : 标识符是数组, 需要进行下标运算: j
Error in line 49 : 标识符未定义: dddd
Error in line 49 : 标识符未定义: dddd2
Error in line 49 : 函数参数不匹配, f_b()
Error in line 49 : 函数无返回值, f_c()
Error in line 49 : 函数未定义: f_undefine()
Error in line 51 : 函数未定义: f_un()
Error in line 53 : 函数参数不匹配: f_c()
Error in line 55 : 标识符是常量, 不能读入: a
Error in line 60 : 函数参数不匹配, f_b()
Error in line 68 : 标识符是常量, 不能赋值: a
Error in line 68 : 标识符未定义: eee
Error in line 68 : 标识符是常量, 不能赋值: b
Error in line 71 : 缺少占用符
Error in line 72 : 占用符不匹配: c
Error in line 73 : 标识符未定义: acc
Error in line 76 : 标识符未定义: __true

F:\Git\CompilerC0\CompilerC0-vs2019\Debug\CompilerC0-vs2019.exe (进程 22680) 已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口...
```

以上全部测试完成

四、中间代码生成

一些想法: 对于生成AST:

麻烦：增加了工作量。原本可以在语法分析的同一位置进行语义分析（建立符号表、进行语义检查），然后可以在同位置生成中间代码；生成AST的话：需要先设计、建立AST，然后验证，还有输出验证；之后是语义分析，这个方面没什么差别，就是原本是一遍分析了，现在是需要两遍；最后再遍历AST生成中间代码。并且由于考虑不周（事先对知识缺失、不了解，下一阶段的输出不确定），以及AST结构不相对统一；导致后面遍历的时候（输出AST、语义分析、生成中间代码）都需要写遍历的方法（但后面差别不大；主要是，要是早知道，就可以统一写一个方法了）。

好处：利于优化（DAG什么的，未验证），结构性强，不会把代码全都堆在语法分析内部了。

分为三个层次：声明、语句、表达式

声明：较为简单，不多说；

语句：主要是个别语句、单独处理即可；没有大问题

表达式：布尔表达式要考虑较多，还没有细致想法。算术表达式基本思想：运算全都是整形，只有赋值、函数调用、写、返回语句用到它的值的时候进行处理

这样会导致两个问题：有int和char，就需要有扩展和截断两个操作。表达式中统一扩展，包括标识符和常量；语句中才有截断。这样就要确定左值类型；（语义分析时优化）

4.1 表达式的翻译：

```
1  #define MAX_QUADVAR_NUM 1000
2
3  typedef struct {
4      char op[15];
5      char* var1;
6      char* var2;
7      char* var3;
8  }Quadvar;
9
10 extern Quadvar quadvarlist[MAX_QUADVAR_NUM];
11 extern int NXQ;
```

4.2 四元式

op	var1	var2	var3
+,-,*/	id/num	id/num	result
callret	id (函数名)		id(ret)
getarray	id (数组名)	id/num (index)	id
jop	id	id	label
j			label
jnz	id/num		label
const	int/char	val	name
int/char			name
intarray	size		name
chararray	size		name
Func	int/char/void		name
para	int/char		name
Main			
setarray	id/num	index	name
assign	id/num		name
lab			label
call	name		
vpara			id/num
scanf			name
print	str_index		
print		id/num	int/char
ret			id/num
ret			
endf			

五、目标代码生成

5.0 理论基础

https://firmianay.gitbooks.io/ctf-all-in-one/doc/1.5.2_assembly.html#36-mips%E6%B1%87%E7%BC%96%E5%9F%BA%E7%A1%80

<https://www.cnblogs.com/thoupin/p/4018455.html>

两种格式用于寻址:

- 使用寄存器号码, 例如 `$ 0` 到 `$ 31`
- 使用别名, 例如 `$ t1`, `$ sp`
- 特殊寄存器 `Lo` 和 `Hi` 用于存储乘法和除法的结果
- 不能直接寻址; 使用特殊指令 `mfhi` (“从 `Hi` 移动”) 和 `mflo` (“从 `Lo` 移动”) 访问的内容

寄存器	别名	用途
<code>\$0</code>	<code>\$zero</code>	常量0(constant value 0)
<code>\$1</code>	<code>\$at</code>	保留给汇编器(Reserved for assembler)
<code>\$2-\$3</code>	<code>\$v0-\$v1</code>	函数调用返回值(values for results and expression evaluation)
<code>\$4-\$7</code>	<code>\$a0-\$a3</code>	函数调用参数(arguments)
<code>\$8-\$15</code>	<code>\$t0-\$t7</code>	暂时的(或随便用的)
<code>\$16-\$23</code>	<code>\$s0-\$s7</code>	保存的(或如果用, 需要SAVE/RESTORE的)(saved)
<code>\$24-\$25</code>	<code>\$t8-\$t9</code>	暂时的(或随便用的)
<code>\$26~\$27</code>	<code>\$k0~\$k1</code>	保留供中断/陷阱处理程序使用
<code>\$28</code>	<code>\$gp</code>	全局指针(Global Pointer)
<code>\$29</code>	<code>\$sp</code>	堆栈指针(Stack Pointer)
<code>\$30</code>	<code>\$fp</code>	帧指针(Frame Pointer)
<code>\$31</code>	<code>\$ra</code>	返回地址(return address)

东: 先处理 `fp`, `fp` 到 `sp`, `fp=sp`, 再处理 `ra`, `jal`; 进入函数: 保存参数到内存, 之后就是函数体;
返回阶段: 处理返回值 `v1` 寄存器, `ra` 到 `t0`, 再恢复 `ra`, `sp`, `fp`, `jr t0`

杨:

MIPS编程入门 (妈妈说标题要高大上, 才会有人看>_<!) <https://www.cnblogs.com/thoupin/p/4018455.html>

复习：MIPS程序和数据的存储器分配

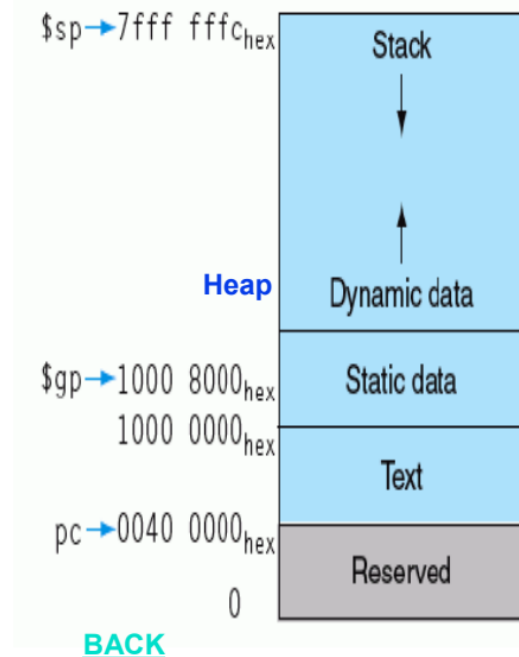
- ◆ 每个MIPS程序都按如下规定进行存储器分配
- ◆ 每个可执行文件都按如下规定给出代码和数据的地址

栈区位于堆栈高端，堆区位于堆栈低端

静态数据区存放全局变量（也称静态变量），指所有过程之外声明的变量和用Static声明的变量；从固定的0x1000 0000处开始向高地址存放

全局指针\$gp总是0x1000 8000，其16位偏移量的访问范围为0x1000 0000~0x1000 ffff，可遍及整个静态数据区的访问

程序代码从固定的0x0040 0000处开始存放故PC的初始值为0x0040 0000



5.1 寄存器分配思路

1、一种较为简单的思路：（无寄存器分配）

首先处理全局变量：.data段

之后就是处理函数：.text段，处理 $\$fp, \$sp, \$ra$ ，局部变量的分配，临时变量分配（全分配，出现位置var3）；之后开始函数体，主要是一些语句：流程控制（选择、循环），赋值，函数调用（无返回值），读写（syscall）；

返回阶段：寄存器还原、返回值

事实上，这里用到的寄存器，除特殊指针外，只有t1, t0；计算出中间变量，立即送入内存（事实上这里并不高效，因为接下来可能会使用，又需要取出，完全是多余的；而且临时变量全部分配空间，内存会浪费）；但这有一个好处，就是处理简单，寄存器中的数据是不需要保存的。

（下一步，优化阶段，计划采用一定的寄存器分配策略，优化时空）

再一步细化就是表达式的问题，针对算术表达式：+、*、/，数组下标，较为简单，取操作数即可，计算完将结果送给内存。函数调用（有返回值）处理完参数之后，直接调用即可，无需考虑寄存器的问题。

（但，若后续使用一定的寄存器分配策略就需要保存寄存器状态）。

2、寄存器分配（FIFO）

```

1 typedef struct {
2     int regindex;
3     char* varname;
4 }RegInfo;
5 // 寄存器与变量的映射队列
6 std::list<RegInfo> regInfoList;
7 // 剩余可用reg数量:t0-t9, 每次函数调用前, 应保存寄存器数据, 此时重新赋值10
8 static int emptyRegNum = 10;

```

用到三种数据结构：剩余可用reg数量、映射队列、符号表

每次需要用到寄存器时，都调用以下过程；会返回可用的寄存器序号。外界不在乎内部如何实现

```

1 int getRegIndex(char* varname)

```

由于符号表结构（数组只是一项，不能反映每一个元素状态）决定：

针对全局、局部、临时变量，寄存器中存储的是值

对于数组：寄存器中存储的是基地址（取值和赋值有临时变量，不影响）

具体实现思路：

有三种情况：

- 1、变量已经在寄存器中：只需要查找映射队列，返回所在的寄存器序号，再更新映射队列即可（放到末尾）
- 2、变量不在寄存器，有空闲的寄存器：符号表isreg标志置为true，空闲寄存器数量-1，映射队列插入新元素
- 3、变量不在寄存器，无空闲的寄存器：首先（FIFO）将一个寄存器释放，送入内存，isreg标志位改变，映射队列pop；新元素isreg为true，映射队列pushback。

注意

1、类型转换：

赋值语句：赋值前类型转换

函数调用：在函数体内转换

输入语句：查符号表确定类型，系统调用控制

返回语句：返回前类型转换

2、返回值问题

return语句：只是赋值给\$v0, j（这里要检查，是否有返回值和返回数据类型转换）

之后的函数返回的栈的处理都是放在末尾处理；

（这里允许函数定义时有返回值，但没写return语句，这样返回值不确定）

5.3 过程

1、genasm:

updateST:

.data:

.text:

2、Func

保存返回类型

函数体：栈帧变化、参数类型转换、变量定义（分配空间）

3、逐个分析

先处理表达式：+、*、/、vpara, callret, getarray

函数及调用：包括函数栈帧变化，变量分配空间，参数处理

语句：赋值，读，写，返回语句

最后是布尔表达式：jop, j, jnz, lab

bug

1、语法树中char的类型改为unsigned char

5.4 测试

按照几大类测试：算术表达式、函数调用、语句、布尔表达式

Text03：输入输出

输出的地方检查出bug，四元式生成不对，类型不匹配

.data：分配方式错误

Text04：赋值语句：整形、字符，语句右边不考虑（由固定函数加载，单独测试）

全局变量：pass

局部变量：pass

全局数组：pass

局部数组：pass

下标运算：有立即数，变量（这里的变量包含表达式，因为运算结果就是临时变量）

pass

Text05：函数调用、返回语句

call：bug：若函数内没有变量只有临时变量，初始地址错误

callret：pass

ret: 有返回值时要检查类型: 有bug, 函数类型判断错误

实参类型检查: pass, 由被调用函数实现

返回语句这里会有最后一次查错: 返回类型检查, 有无返回值。(有返回值未写return, 返回值不确定)

只有void才能写return;

char会对之后的表达式进行类型转换

然后跳转

Text06: 算术表达式

六种操作。pleace字段保存立即数或者变量名(全局、临时、局部)

bug: 前面没发现, 函数类型错误

Text07: 布尔表达式、条件循环

没什么bug

六、优化

6.1 常量合并

常量计算改为赋值

6.2 强度削弱

乘法和除法改为移位

6.3 删除公共子表达式

检查当前基本块中(+, *, /, assign), 是否有相同操作, 操作数; 若相同, 则删除重复计算、替换中间变量;

注意检查操作数是否被改变

6.4 寄存器分配

6.4.1 基本思想:

FIFO原则

6.4.2 实现

寄存器分配器, 返回一个可用的寄存器

```

1  /*
2  用途：管理寄存器堆。当需要加载操作数时（立即数、变量值、数组基地址），查看是否已在寄存器，
   或加载到哪个寄存器。
3  输入：变量名
4  输出：该变量可以存入的寄存器index，或者是已经存在的寄存器序号，isInReg, isGlobal
5  首先查找符号表中该变量是否在寄存器中，
6      如果在：返回序号，维护寄存器状态列表（对于全局、局部、临时变量都表示值是否在寄存器，数
   组表示基地址是否在）
7      不在：查看是否有空寄存器，若无，FIFO原则维护寄存器堆
8
9      此外还需维护两个全局变量：表示是否在寄存器中（用于该函数体外），是否全局变量（.data段
   全局变量可以通过变量名直接获取、存储）
10 */
11 int getRegIndex(char* varname)

```

之后再调入寄存器

```

1  // 从内存读入寄存器
2  void mem2reg(char* varname, int reg)

```

七、错误处理

7.1、错误识别

```

1  // 匹配 期待的 token；否则报错
2  // 读取下一个token
3  static bool match(TokenType expectToken) {
4      if (expectToken == g_token.opType) {    // 与预期相同，跳过token， true
5          getNextToken();
6          return true;
7      }
8      else {                                    // 提示错误，但在这里不跳读
9          g_errorNum++;
10         printf("match error in line %d :\t\t\texpect Token %d, but %d value:
   %s \n", g_lineNumber, expectToken, g_token.opType, g_token.value);
11         return false;
12     }
13 }

```

7.2、跳读到可以正确分析的位置

直到识别到某些token才停止；

```

1  typedef enum {
2      LACK_SEMI_CST,                // 常量定义没有分号，跳出当前，直到再次遇到常量定
   义、变量定义、语句
3      LACK_TYPE_CST,                // 没有类型
4      LACK_ID_CST,                  // 没有标识符
5      LACK_ASSIGN_CST,              // 没有赋值符号
6

```

```

7      LACK_XXX_VARDEF,          // 变量定义出错，直接抛弃当前语句
8
9      LACK_TYPE_FUN,           // 函数类型未说明
10     LACK_IDEN_FUN,            // 函数名未说明    ，
11     LACK_KUOHAO_FUN,          // 函数括号丢失    ， 跳过当前函数
12
13     SENTENCE_ERROR
14 }ErrorType;

```

大致思想：

常量定义、变量定义、函数定义：跳过当前定义

语句：跳过当前语句

```

1  switch (e)
2  {
3      case ErrorType::LACK_SEMI_CST:
4      case ErrorType::LACK_TYPE_CST:
5      case ErrorType::LACK_ID_CST:
6      case ErrorType::LACK_ASSIGN_CST: {
7          while (g_token.opType != TokenType::CONST && g_token.opType !=
TokenType::INT && g_token.opType != TokenType::CHAR
8              && g_token.opType != TokenType::VOID && g_token.opType !=
TokenType::IF
9              && g_token.opType != TokenType::WHILE && g_token.opType !=
TokenType::FOR
10             && g_token.opType != TokenType::IDEN && g_token.opType !=
TokenType::RETURN
11             && g_token.opType != TokenType::SCANF && g_token.opType !=
TokenType::PRINTF
12             && g_token.opType != TokenType::LBRACE && g_token.opType !=
TokenType::RBRACE
13         )
14         {
15             if (g_token.opType == TokenType::END)
16             {
17                 exit(0);
18             }
19             getNextToken();
20         }
21         break;
22     }
23     // 以上都是常量定义阶段的错误
24     // 变量定义错误:
25     case ErrorType::LACK_XXX_VARDEF: {
26         while (g_token.opType != TokenType::INT && g_token.opType !=
TokenType::CHAR
27             && g_token.opType != TokenType::VOID && g_token.opType !=
TokenType::IF
28             && g_token.opType != TokenType::WHILE && g_token.opType !=
TokenType::FOR
29             && g_token.opType != TokenType::IDEN && g_token.opType !=
TokenType::RETURN
30             && g_token.opType != TokenType::SCANF && g_token.opType !=
TokenType::PRINTF
31             && g_token.opType != TokenType::LBRACE && g_token.opType !=
TokenType::RBRACE

```

```

32     )
33     {
34         if (g_token.opType == TokenType::END)
35         {
36             exit(0);
37         }
38         getNextToken();
39     }
40     break;
41 }
42 // 函数定义错误
43 case ErrorType::LACK_TYPE_FUN:
44 case ErrorType::LACK_IDEN_FUN:
45 case ErrorType::LACK_KUOHAO_FUN: {
46
47     while (g_token.opType != TokenType::INT && g_token.opType !=
TokenType::CHAR
48         && g_token.opType != TokenType::VOID
49     )
50     {
51         if (g_token.opType == TokenType::END)
52         {
53             exit(0);
54         }
55         getNextToken();
56     }
57
58
59     break;
60 }
61 // 语句错误
62 case ErrorType::SENTENCE_ERROR: {
63     while (g_token.opType != TokenType::IF
64         && g_token.opType != TokenType::WHILE && g_token.opType !=
TokenType::FOR
65         && g_token.opType != TokenType::IDEN && g_token.opType !=
TokenType::RETURN
66         && g_token.opType != TokenType::SCANF && g_token.opType !=
TokenType::PRINTF
67         /*&& g_token.opType != TokenType::LBACE */&& g_token.opType !=
TokenType::RBRACE
68     )
69     {
70         if (g_token.opType == TokenType::END)
71         {
72             exit(0);
73         }
74         getNextToken();
75     }
76     break;
77 }

```

7.3 语法树错误屏蔽

语法树标识错误，语义分析时跳过。

```
1 // 错误处理
2 bool error;
```

八、bug修复

1、asm生成阶段：scanf语句

bug：多次读取同名变量时，由于scanf是直接写入内存，但或许之前使用过这个变量，并且已经调入寄存器；查表的时候会显示在

寄存器堆中，这样就会读取就值；

发现于 Test11.c0；多次输入x，发现后面使用的x的值始终不变

(old, 会带来新问题，可用寄存器数目不断减少)处理：从寄存器堆中移除此变量，且不用写回内存；

故只需更新符号表状态、寄存器映射队列、可用寄存器数目

(new)处理：如果在寄存器，则更新寄存器数据即可，其他不变

