

**本科实验报告**

|  |  |
| --- | --- |
| 课程名称： | 编译原理 |
| 姓 名： | 黄梓淇、官泽隆、尹幽潭 |
| 学 院： | 计算机科学与技术 |
| 系： | 计算机科学与技术 |
| 专 业： | 计算机科学与技术 |
| 学 号： | 3180106025、3180103008、3180105171 |
| 指导教师： | 冯雁 |

## 序：

### 0.1 实验环境

Windows环境下的编译和运行

（1）Visual Studio 2017

（2）YACC、LEX开发环境（Lex和Yacc的集成开发包）

（3）LLVM开发环境

（4）GraphViz软件及python包

### 0.2 分工明细

|  |  |
| --- | --- |
| 姓名 | 分工 |
| 官泽隆 | lex词法分析、解析生成AST、中间代码生成  提供I/O库和JIT 环境 |
| 尹幽潭 | yacc语法分析、语法生成树生成、语义分析+错误检测 |
| 黄梓淇 | 词法语法设计修订、可视化语法树代码、批处理代码、合并语义分析与中间代码生成的代码、测试、报告撰写、PPT撰写、展示 |

## 词法分析

### 正规表达式

1. 定义

|  |  |
| --- | --- |
| letter | [\_a-zA-Z](注意：包括下划线) |
| digit | [0-9] |
| Hex | [a-fA-F0-9] |

1. Token

|  |  |
| --- | --- |
| Token | 识别 |
| NUMBER  （包含十进制表示的number  以及十六进制的number）  注意：本编译器仅支持int类型计算因此NUMBER的值是int类型 | [1-9]{digit}\* |
| 0[0-7]\* |
| 0[xX]{Hex}+ |
| CONST | “const” |
| INT | “int” |
| VOID | “void” |
| IF | “if” |
| ELSE | “else” |
| WHILE | “while” |
| BREAK | “break” |
| CONTINUE | “continue” |
| RETURN | “return” |
| IDENT | {letter}({letter}|{digit})\* |
| ‘+’ | “+” |
| ‘-’ | “-” |
| ‘\*’ | “\*” |
| ‘/’ | “/” |
| ‘%’ | “%” |
| LE\_OP | “<=” |
| GE\_OP | “>=” |
| ‘<’ | “<” |
| ‘>’ | “>” |
| EQ\_OP | “==” |
| NE\_OP | “!=” |
| ‘!’ | “!” |
| ‘=’ | “=” |
| AND\_OP | “&&” |
| OR\_OP | “||” |
| ‘;’ | “;” |
| ‘,’ | “,” |
| ‘[’ | “[“ |
| ‘]’ | “]” |
| ‘(’ | “(“ |
| ‘)’ | “)” |
| ‘{’ | “{“ |
| ‘}’ | “}” |

共34个token

1. 辅助：

词法需处理注释并需跳过空格和错误单词：

|  |  |
| --- | --- |
| “//”[^\n]\* | just consume |
| “/\*” | comment(); |
| [ \t\n\v\f] | just consume（注意：包括空格） |
| . | error();错误处理 |

注释处理代码：

**void comment(void){**

**char c, prev = 0;**

**while ((c = input()) != 0){ /\* (EOF maps to 0) \*/**

**if (c == '/' && prev == '\*')**

**return;**

**prev = c;**

**}**

**error("unterminated comment");**

**}**

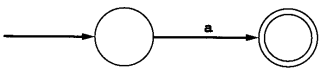
### 实现原理和方法

*——实现原理参考教材总结而得*

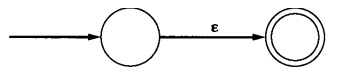
* + 1. **从正则表达式到NFA**

1. 基本正则表达式

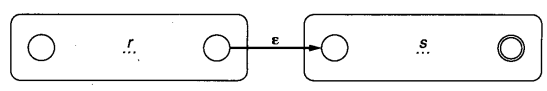
基本正则表达式格式a、ε或⊙，其中a表示字母表中单个字符的匹配，ε表示空串的匹配，而⊙则表示根本不是串的匹配。与正则表达式a等同的NFA（即在其语言中准确接受）的是：



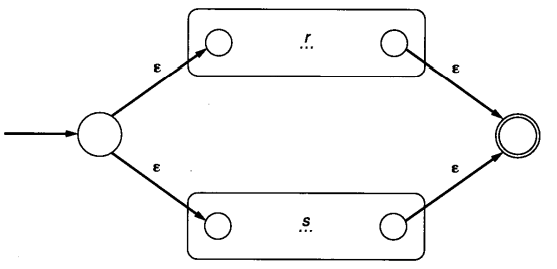
类似地，与ε等同的NFA是：



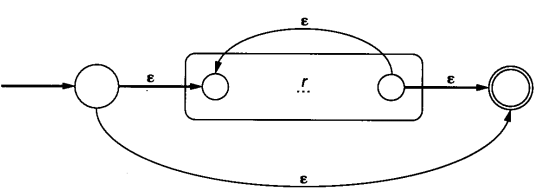
1. 并置



1. 在各选项中选择



1. 重复



* + 1. **从NFA到DFA**

1. 子集构造

首先计算M初始状态的大三ε-闭包，它就变成~M的初始状态。对于这个集合以及随后的每个集合，计算a字符之上的转换如下所示：假设有状态的S集和字母表中的字符a，计算集合S´a = {t | 对于S中的一些s，在a上有从s 到t 的转换}。接着计算~S’a，它是S´a 的闭包。这就定义了子集构造中的一个新状态和一个新的转换 ，继续这个过程直到不再产生新的状态或转换。当接受这些构造的状态时，按照包含了M的接受状态的方式作出记号。这就是DFA的~M。

1. 利用子集构造模拟NFA

模拟NFA的一种方法是使用子集构造，但并非是构造与DFA相关的所有状态，而是在由下一个输入字符指出的每个点上只构造一个状态。因此，这样只构造了在给出的输入串上被取用的D FA的路径中真正发生的状态集合。

1. DFA中的状态数最小化

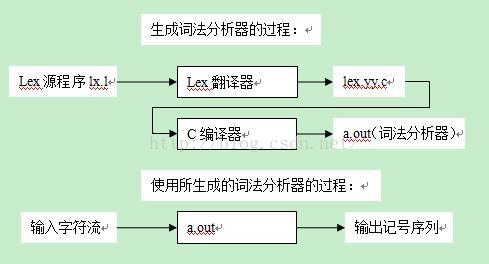
以最乐观的假设开始：创建两个集合，其中之一包含了所有的接受状态，而另一个则由所有的非接受状态组成。假设这样来划分原始 D FA的状态，还要考虑字母表中每个a上的转换。如果所有的接受状态在a上都有到接受状态的 转换，那么这样就定义了一个由新接受状态（所有旧接受状态的集合）到其自身的a-转换。类似地，如果所有的接受状态在a上都有到非接受状态的转换，那么这也定义了由新接受状态到新的非接受状态（所有旧的非接受状态的集合）的a-转换。另一方面，如果接受状态s和t在a上有转换且位于不同的集合，则这组状态不能定义任何a-转换，此时就称作a区分了状态s和t。在这种情况下必须根据考虑中状态集合（即所有接受状态的集合）的a-转换的位置而将它们分隔开。当然状态的每个其他集合都有类似的语句，而且一旦要考虑字母表中的所有字符时，就必须移到它们的位置之上。当然如果还要分隔别的集合，就得返回到开头并重复 这一过程。我们继续将原始D FA的各部分状态集中到集合里，并一直持续到所有集合只有一个元素（在这种情况下，就显示原始DFA为最小）或一直是到再没有集合可以分隔了。

* + 1. **基于Lex的词法分析**

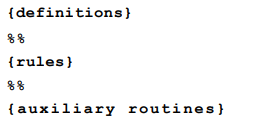
在本实验中，我们采用工具Lex来辅助进行词法分析，相关代码在cc.l文件内存放。

1. Lex基本原理

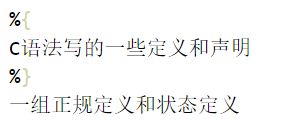
Lex的基本工作原理为：由正规式生成NFA，将NFA变换成DFA，DFA经化简后，模拟生成词法分析器。也即上述我们描述的词法分析原理。具体流程见下图：



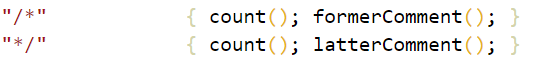
1. lex源程序的格式：



第一部分是定义段，包含2个部分：以C语法写的一些定义和声明和一组正规定义和状态定义：

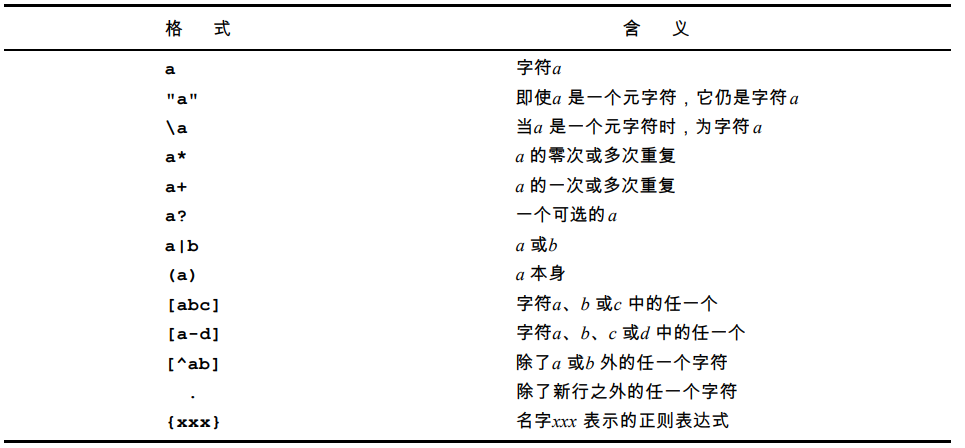


第二部分是词法规则段，词法规则段列出的是词法分析器需要匹配的正规式，以及匹配该正规式后需要进行的相关动作，例如：



第三部分是辅助函数段，辅助函数段用C语言语法来写，辅助函数一般是在词法规则段中用到的函数。这一部分一般会被直接拷贝到lex.yy.c中。

1. Lex中的元字符约定



1. 我们的Lex代码：

由于代码很长，请自行打开code目录下的cc.l

## 第二章、语法分析

*——实现原理参考教材总结而得，文法参考编译原理大赛的Sysy语言修改得到*

### 2.1 上下文无关文法

|  |  |
| --- | --- |
| 类型 | 文法 |
| 编译单元 | CompUnit🡪 CompUnit Decl | Decl  | CompUnit FuncDef | FuncDef |
| 声明 | Decl🡪ConstDecl | VarDecl |
| 常量声明 | ConstDecl🡪 **CONST** BType ConstDef ConstDef\_list ‘**;’**  ConstDef\_list 🡪 ConstDef\_ list ‘**,’** ConstDef | ε |
| 基本类型 | BType🡪**INT** |
| 常数定义 | ConstDef🡪**IDENT** ‘**=’** ConstInitVal  | **IDENT** ‘**[’** ConstExp ‘**]’ ‘=’** ConstInitVal |
| 常量初值 | ConstInitVal🡪ConstExp  |‘**{’ ‘}’**  |‘**{’** ConstExp ConstExp\_ list **‘}’**  ConstExp\_ list 🡪 ConstExp\_ list ‘**,’** ConstExp | ε |
| 变量声明 | VarDecl🡪BType VarDef VarDef\_ list ‘**;’**  VarDef\_ list 🡪 VarDef\_ list **‘,’** VarDef | ε |
| 变量定义 | VarDef🡪 **IDENT**  | **IDENT** ‘**[’** ConstExp **‘]’**  | **IDENT** **‘=’** InitVal  | **IDENT** **‘[’** ConstExp ‘**]’ ‘=’** InitVal |
| 变量初值 | InitVal🡪Exp  | **‘{’ ‘}’**  | **‘{’**Exp Exp\_ list **‘}’**  Exp\_ list 🡪 Exp\_ list **‘,’** Exp | ε |
| 函数定义 | FuncDef🡪BType **IDENT** **‘(’ ‘)’** Block  | BType **IDENT** **‘(’** FuncFParams **‘)’** Block |
| 函数形参表 | FuncFParams🡪 FuncFParam FuncFParam\_ list  FuncFParam\_ list 🡪 FuncFParam\_ list ‘**,’** FuncFParam | ε |
| 函数形参 | FuncFParam🡪BType **IDENT**  | BType **IDENT** **‘[’ ‘]’** |
| 语句块 | Block🡪**‘{’** BlockItem\_list **‘}’**  BlockItem\_list 🡪 BlockItem\_list BlockItem | ε |
| 语句块项 | BlockItem🡪Decl | Stmt |
| 语句 | Stmt🡪LVal **‘=’** Exp **‘;’** | Exp **‘;’** | **‘;’** | Block  | **IF** **‘(’** Cond **‘)’** Stmt  | **IF ‘(’** Cond **‘)’** Stmt **ELSE** Stmt  | **WHILE ‘(’** Cond **‘)’** Stmt  | **BREAK ‘;’** | **CONTINUE ‘;’**  | **RETURN** Exp **‘;’** |
| 表达式 | EXP🡪AddExp |
| 条件表达式 | Cond🡪LOrExp |
| 左值表达式 | LVal🡪 **IDENT** | **IDENT** **‘[’** Exp **‘]’** |
| 基本表达式 | PrimaryExp🡪**‘(’** Exp **‘)’** | LVal| **NUMBER** |
| 一元表达式 | UnaryExp🡪PrimaryExp  | **IDENT** **‘(’ ‘)’**  | **IDENT** **‘(’** FuncRParams **‘)’**  | UnaryOp UnaryExp |
| 单目运算符 | UnaryOp🡪**‘+’**|**‘-’**|‘!’  (！仅在条件表达式中出现，需要在yacc中解决) |
| 函数实参表 | FuncRParams🡪Exp Exp\_list  Exp\_list 🡪 Exp\_list **‘,’** Exp | ε |
| 加减表达式 | AddExp🡪 MulExp  | AddExp **‘+’** MulExp  | AddExp **‘-’** MulExp |
| 乘除模表达式 | MulExp🡪UnaryExp  | MulExp **‘\*’** UnaryExp  | MulExp **‘/’** UnaryExp  | MulExp **‘%’** UnaryExp |
| 关系表达式 | RelExp🡪AddExp  | RelExp **‘<’** AddExp  | RelExp **‘>’** AddExp  | RelExp **LE\_OP** AddExp  | RelExp **GE\_OP** AddExp |
| 相等性表达式 | EqExp🡪RelExp  | EqExp **EQ\_OP** RelExp  | EqExp **NE\_OP** RelExp |
| 逻辑与表达式 | LAndExp🡪EqExp | LAndExp **AND\_OP** EqExp |
| 逻辑或表达式 | LOrExp🡪LAndExp | LOrExp **OR\_OP** LAndExp |
| 表达式 | ConstExp🡪AddExp  (使用的Ident必须是常量，需要在yacc中解决) |

### 2.2实现原理和方法

* + 1. **LALR(1)分析**

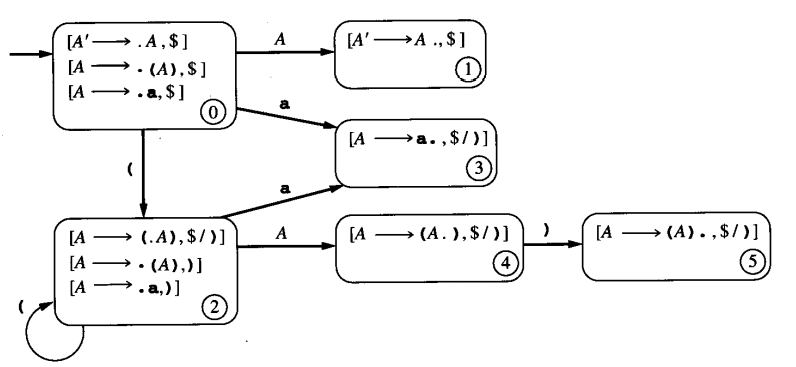
1. LALR(1)分析的第1个原则

LR(1)项目的DFA的状态核心是LR(0)项目的DFA的一个状态。

1. LALR(1)分析的第2个原则

若有具有相同核心的LR(1)项目的DFA的两个状态s1 和s2，假设在符号X上有一个从s1 到状态t1 的转换，那么在X上就还有一个从状态s2 到一个状态t2 的转换，且状态t1 和t2 具有相同的核心。

1. 这两个原则允许我们构造LALR(1)项目的DFA(DFA of LALR(1) items)，它是通过识别具有相同核心的所有状态以及为每个LR(0)项目构造出先行符号的并，而从LR(1)项目的DFA构造出来。因此，这个D FA的每个LALR(1)项目都将一个LR(0)项目作为它的第1个成分，并将一个先行记号的集合作为它的第2个成分。示例：（教材上例5.17的LALR(1)项目集合的DFA）



* + 1. **基于YACC的语法分析**

1. YACC原理

YACC包含三个部分：

• 总控程序，也可以称为驱动程序。

对所有的LR分析器总控程序都是相同的。

• 分析表或分析函数。

不同的文法分析表将不同，同一个文法采用的LR分析器不同时，分析表也不同，分析表又可分为动作(ACTION)表和状态转换(GOTO)表两个部分，它们都可用二维数组表示。

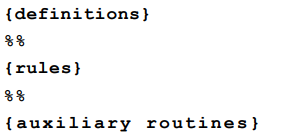
• 分析栈，包括文法符号栈和相应的状态栈。

它们均是先进后出栈。 分析器的动作由栈顶状态和当前输入符号所决定(LR(0)分析器不需要向前查看输入符号)。

YACC的分析器工作过程如下:

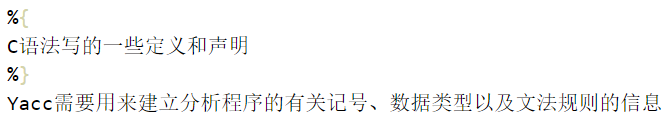
其中SP为栈指针，S[i]为状态栈，X[i]为文法符号栈。状态转换表内容按关系GOTO[Si，X] = Sj确定，该关系式是指当栈顶状态为Si遇到当前文法符号为X时应转向状态Sj。X为终结符或非终结符。ACTION[Si，a]规定了栈顶状态为Si是遇到输入符号a应执行的动作。

1. YACC源代码格式

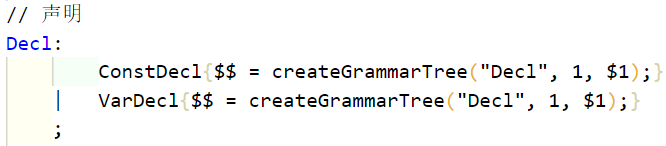


YACC的总体格式于Lex大同小异。

第一部分是定义段，包含2个部分：以C语法写的一些定义和声明和Yacc需要用来建立分析程序的有关记号、数据类型以及文法规则的信息：



第二部分是规则段，语法规则段列出的是语法分析器需要的文法规则，以及进行改文法规约后需要进行的相关动作，例如：



第三部分是辅助函数段，辅助函数段用C语言语法来写，辅助函数一般是在语法规则段中用到的函数或yyerror这类自动调用的函数。这一部分一般会被直接拷贝到y.tab.c中。

1. Yacc关于if-else的解决

Yacc分析器遇到shift/reduce冲突后，会自动优先shift而不是reduce，因此这样就可以很轻松的解决if-else的else悬挂问题。

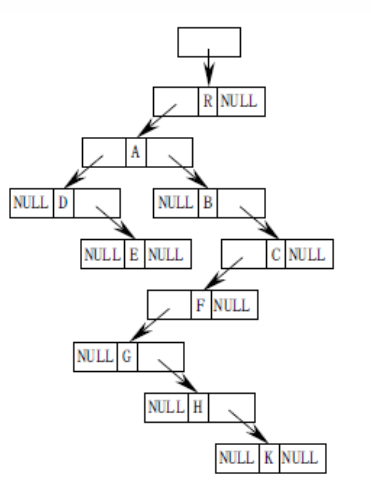
1. 语法分析树数据结构

· 数据结构定义：

|  |
| --- |
| struct grammarTree  {  string name;  string content;  int lineno;  int id;  struct grammarTree \*left; // 1st child  struct grammarTree \*right; // next sibling  // prep for tailor; classified on Nb\_opr  using Type\_t = enum { BinExpr, List, Garbage, NA };  Type\_t type() const { //返回节点的类型，如BinExpr}  bool orphan() { return right == nullptr; }  int nb\_child() {//返回这个节点的孩子数目}  grammarTree \*fold\_lchain() {//用于裁剪节点，删除本节点并将本节点左节点返回，  同时本节点的右兄弟接入左节点的右兄弟中}  grammarTree \*fold\_rchain() {//用于裁剪节点，删除本节点并将本节点右节点返回}  void tailor(); // to be called from root用于裁剪语法树节点  grammarTree \*tailor\_inner();  ~grammarTree() {// cascading deletion级联的删除节点}  }; |

· 数据结构分析：

(1) 语法树数据结构采用左孩子右兄弟的多叉树转二叉树的定义方式，类似下图所示：



(2) 参数说明：

- name是语法树节点的名称，如IDENT、NUMBER，与Token名称一致

- content是语法树节点的内容，针对IDENT、NUMBER，记录其具体值

- lineno记录其出现行号，方便后续报错

- id是赋予每一个语法树节点的唯一标识符

- left指向左孩子

- right指向右兄弟

(3) 内嵌函数：

- Type\_t type()返回节点的类型，如BinExpr

- bool orphan() 判断有没有右兄弟

- int nb\_child()返回这个节点的孩子数目

- grammarTree \*fold\_lchain() 用于裁剪节点，删除本节点并将本节点左节点返回，同时本节点的右兄弟接入左节点的右兄弟中

- grammarTree \*fold\_rchain()用于裁剪节点，删除本节点并将本节点右节点返回

void tailor()用于裁剪语法树节点

grammarTree \*tailor\_inner();被tailor调用

~grammarTree()用于级联的释放空间(删除树)

1. 我们的YACC代码

由于代码很长，请自行打开code目录下的cc.y

## 第三章、语义分析

### 3.1 实现的错误类型检测（包含词法语法语义分析阶段）

**3.1.1 语言规则**

我们对我们实现的编辑器/定义的语言做出以下假设：

**文件假设：**

假设1：可以在命令行一次性读入多个文件编译，没有实质意义内容的空白文件将会直接跳过。

**词法假设：**

假设2：仅支持八进制、十六进制和十进制int类型数据。

假设3：不匹配错误格式的变量名。

文法假设：

假设4：仅支持我们所定义的token和rule。

**语义假设**：

**定义假设**

假设5：只支持int数据类型

假设6：const型变量定义时必须初始化，且初始化形式只能是 const int a=NUMBER。

假设7：const数组定义时必须初始化，且初始化形式只能是const int a[NUMBER] = {NUMBER, NUMBER, … , NUMBER},初始化大小与定义的大小一致。

假设8：非const变量定义时可以不初始化，如果初始化，仅适用const的规则或适用非数组型变量。

假设9：非const数组定义时可以不初始化，如果初始化，仅适用const的规则。

**函数假设**

假设10：必须返回且只能返回int类型

假设11：参数不能是const

假设12：没有声明且不能声明，首次出现即定义，不能在函数体中（嵌套）定义

假设13：参数不能是其他函数

除了以上需要特别说明的以外，相同符号的使用或规则与 c 语言的使用或规则并无区别，可以视为c的子集。当然，很可能会出现其他能够在c里面出现的书写，但是在我们的编辑器里不支持的情形，也可能会出现我们未描述到的假设或规则，比如数组索引不能使用比较表达式。一般情况下，没有特别说明不可以，就是可以这样做，或者产生不那么精细的报错。

**3.1.2 错误类型**

依据上述假设，我们可以定义以下一些错误类型：

**1.**未定义的数据类型：在词法中实现了对一些c++中常用，但我们的语言中没有的数据类型的关键字的匹配，可以报错未定义的数据类型。

**2.**错误的数据类型：在词法中实现了对一些c++中常用，但我们的语言中没有的数据类型的匹配，可以报错不支持的数据类型；还实现了对错误的整型数字的匹配，报错错误的数据格式。

**3.**注释错误：可以检测c语言风格的注释不匹配。

**4.**未能匹配的符号：报错未知符号。

**5.**错误定义的const/var变量：针对每种错误有细微不同的说明，具体定义错在哪里。

**6.**符号重复定义/未定义：针对不同情形有略微不同的说明，指出具体什么符号。

**7.**符号使用错误：int变量作为函数或数组使用。

**8.**表达式错误：对于我们的语言而言，表达式错误主要有三种：

①数组类型变量加入计算

②比较表达式两边类型不匹配

③赋值给数字或函数返回

**9.**函数返回错误：没有返回或者返回数组。

**10.**函数调用错误：参数个数不匹配或参数类型不匹配。

**11.**其他未说明的错误：如果产生错误，则提示 syntax error。

一些错误将在运行的时候检测或是提出，他们不会引起语义错误。例如：作为数组索引的表达式其实非法访问数组等，我们只检查表达式的类型错误。当然，可能还会出现我们所预料之外的其他语义错误，导致语义分析的意外结束，进而导致我们编辑器运行的错误。

### 3.2 实现原理

**3.2.1 数据结构**

在语义分析中，我们实现一个符号表来记录程序中的符号。从最基础的开始，显然，变量应该有属于它自己的数据结构 Data\_，数据结构定义：

|  |
| --- |
| struct Data\_{  myDataType data\_type;  bool is\_r\_value;  union{  myBasicType basic; // 基本类型  struct{  myData \*elem;  int size;  } array; // 数组类型  };  int value;  }; |

is\_r\_value记录变量是否是右值。数组通过 array->array.elem形式的链表保存。

其中，myDataType代表变量的结构类型，即，是数组还是单个变量：

|  |
| --- |
| typedef enum DataType\_  {  BASIC,  ARRAY,  } myDataType; |

myBasicType代表变量的数据类型，我们只用一种数据类型int：

|  |
| --- |
| typedef enum BasicType\_  {  INT,  } myBasicType; |

除了变量，函数当然也应该有属于自己的数据结构 Func\_：

|  |
| --- |
| struct Func\_  {  myData \*ret\_type;  int param\_num;  myParam \*param\_list; // 函数参数列表  }; |

其中，根据我们设定的语言规则，函数返回类型只能是int且必须返回，用一个变量的数据结构记录函数的返回类型，myParam\*定义函数的参数，param\_num定义函数的参数个数。myParam的具体定义如下所示：

|  |
| --- |
| struct Param\_  {  myData \*type; // 参数的类型  myParam \*next; // 下一个参数  string para\_name;  }; |

使用myData（即Data\_）定义每个参数，next指针用于构建参数的名字，para\_name的目的是为了给错误信息输出提供方便。而且由于我们的语言没有声明，实参和形参的名字不能不同，所以定义一个名字是很有必要的。Data\_没有单独定义名字主要出于以下的考虑：

①我们的符号表主要功能是创建符号和查表，不对符号数据进行维护，查表主要依赖于下文提出的更高层次的符号数据结构体的名。

②变量的名字可以很容易地从叶子节点得到，而且一般使用的地方能容易地访问到叶子节点。

③为数字建立myData\*变量的时候，也不太好决定用什么名字。

有了记录函数和变量的结构，我们就可以定义出我们的符号结构体：

|  |
| --- |
| typedef enum SymbolType\_  {  VAR,  FUNC,  CONST  } mySymbolType;  struct Symbol\_  {  string name;  mySymbolType symbol\_type;  union  {  myData \*type;  myFunc \*func;  };  }; |

根据语言规则，我们将符号设置为VAR，FUNC和CONST三种类型。一个符号可能是函数也可能是变量。在符号的基础上，我们构建符号表的数据结构，使用哈希散列表，这里直接自己手搭一个散列表：申请一个大数组，计算一个散列函数的值，然后根据该值将对应的符号放到数组相应下标的位置即可。对于符号表来说，最简单的方法就是使用hash函数将符号名中的所有字符相加，然后对符号表的大小取模即可。这里我们使用了github上找到的一个散列函数：

|  |
| --- |
| unsigned int calHash(string name)  {  unsigned int val = 0, tmp;  for (int i = 0; i < name.length(); i++)  {  val = (val << 2) + name.at(i);  if (tmp = val & ~(HASH\_SIZE - 1))  val = (val ^ (tmp >> 12)) & (HASH\_SIZE - 1);  }  return val;  } |

我们首先将单个符号连成链表：

|  |
| --- |
| struct SymbolList\_  {  mySymbol \*symbol;  mySymbolList \*next;  }; |

然后使用哈希表，结构体hashSet\_就是我们的符号表。

|  |
| --- |
| #define HASH\_SIZE 16384  typedef struct hashSet\_ \*myHashSet;  typedef struct Bucket\_  {  mySymbolList \*symbol\_list;  } myBucket;  struct hashSet\_  {  int size;  myBucket \*buckets;  }; |

如上就是我们使用的符号表的数据结构。在语义分析中，我们的想法是尽可能减少其他非必要复杂结构的使用。

**3.2.2 分析过程**

我们按照错误类型来展开说明语义分析具体是怎么实现的。

**3.2.2.1** 总体思路

我们采用递归下降的方式分析可能存在的语义错误。对于文法分析中建立的语法树，我们在语义分析中顺着每条规则进行分析，从根节点直到叶子结点。如果存在推导错误，给出报错

**3.2.2.2** 作用域

按照c语言的标准建立作用域，符号表中，每个符号的名字由它所在的作用域+它在程序中实际的名字构成。维护一个作用域栈，栈的成员就是符号，在文法的分析block的地方，新建一个作用域并入栈，block分析完后，栈最顶上的作用域出栈。函数本身作为一个作用域，建立的作用域名字就是函数的名字，符号类型是FUNC；while语句的作用域将会涉及到break和continue和检查，以VAR为类型以示区分；其他作用域以CONST为类型。分析开始时加入一个global作用域。

**3.2.2.2** 定义时错误

错误定义：

在出现定义推导式时，置分析定义表达式为真，开始分析，在递归中碰到具体数据的时候开始检查，分析结束后置表达式标志为假。可以有很不同的检查标志，分别标志要检查数组定义的空间，初始化的格式等等。

重复定义：

对于函数，在符号表中寻找有无一样的名字，有就是重复定义了。（函数作为作用域，其符号名字就是程序中的名字，不会加作用域前缀）。对于变量，需要逆序遍历作用域栈，每次给当前要检查的名字加上作用域名字前缀，然后在符号表中寻找有无一样的名字，有就是重复定义了。通过插入符号表的函数，就可以检查符号表里有无重复。

没有定义：

对于函数，同样，在符号表中寻找有无一样的名字，没有就是没有定义。对于变量，首先在作用域下检查有无该变量，如果遍历到作用域是函数，还要检查函数参数中是否已经定义过该变量。

**3.2.2.3** 表达式与函数

在出现表达式推导时，置分析表达式标志为真，开始分析，在递归中碰到具体数据时开始检查，分析结束后置表达式标志为假。

对于函数返回，维护一个myData类型的变量，在函数定义推导式处，置检查函数返回标志，分析结束后（递归，函数结尾最后总会回到这里），检查有没有调用checkFuncRet函数即可知道函数有没有返回值，checkFuncRet函数检查维护的myData和符号表里记录的定义能否匹配。

对于函数调用，维护一个成员为myParam的容器，在函数调用推导式处，置检查函数调用标志，递归结束后，用checkFuncCall函数检查维护的容器和符号表里的函数定义是否匹配。

**3.2.2.4** 控制语句：只要当前作用域栈中有对应while类型的，就没有错误；否则就有错误。

**3.2.2.5** 关键函数及其作用

checkArray用于检查定义的数组有无错误

|  |
| --- |
| void checkArray(mySymbol \*symb, int lineno) |

checkRepeatFuncDef检查函数有无重复定义

|  |
| --- |
| bool checkRepeatFuncDef(mySymbol \*symb, int lineno) |

checkRepeatVarDef检查变量有无重复定义

|  |
| --- |
| void checkRepeatVarDef(mySymbol \*symb, int lineno) |

检查函数返回

|  |
| --- |
| void checkFuncRet(int lineno) |

检查函数调用

|  |
| --- |
| void checkFuncCall(string func\_name, int lineno) |

检查没有定义

|  |
| --- |
| bool checkNotDef(string name, int lineno, string form) |

在检查有无定义时，根据需要获取数据到维护的数据结构中。按照检查的数据是变量、数组、函数3种，和当前正在检查的是表达式、函数返回还是函数调用3种交叉，有3\*3=9种情形，每种都要按照实际设计，并处理特殊情况。在叶子结点，即分析变量名字和数字的地方，也分别要按当前正在检查的是什么，分情况处理。由于检查的类型可能嵌套，比如函数调用或参与表达式计算等等，所以会产生很复杂的控制，需要十分谨慎小心。

## 第四章、优化考虑（每个阶段的优化考虑）

### 4.1 词法语法分析的优化考虑

1. **词法分析加入代码检测不支持的C关键字**

由于我们的语法是类C语言，因此很多C语言相关关键字会被识别为IDENT导致语法分析报错，二作为类C语言很可能会不小心输入不支持的C关键字，因此专门加入了代码对于不支持的C关键字进行报错。

1. **语法分析加入语法错误检测报错**

原本我们初版语法分析直接采用YACC自带的报错函数，因此只会报错Syntax Error。这样的报错指向性不强，因此我们优化后能够将报错具体到行号，并且对于一些常见的错误类型能够较为精准的报出对应的错误。

1. **语法分析进行生成树的剪枝**

由于我们将优先级和结合性做进了文法当中，导致我们的语法分析树十分冗余不易理解，因此加入了裁剪生成树的代码，将一些关于递归、优先级和结合性的推到节点裁剪掉，获得一棵较为简洁易懂的语法生成树以供后续AST生成和IR生成使用。

1. **撰写了Python工程，借助工具GraphViz实现语法生成树的可视化**

在语法分析程序中我们输出指定格式的包含树结构的文件，然后使用python工程viewTree中main.py函数来实现生成树可视化，命令行执行格式为：

usage: main.py [-h] [-path PATH] [-path PATH] ... [-path PATH]

PATH即为需可视化文件的相对路径，可同时进行多个文件的可视化。

### 语义分析的优化考虑

在使用 lex 和 yacc 完成了词法与文法分析以后，我们小组决定一名同学写中间代码，另一名同学写语义分析和 AST 生成。由于误会，写中间代码的同学误以为文法分析时建立的树就是 AST。其实该树是基于每条文法推导规则建立的，每条规则的左侧为树的父节点，右侧为子节点，“忠实”地记录了推导的详细过程。写中间代码的同学通过剪枝得到了可供后续开发的 AST，而写语义分析的同学也定义了自己的符号表。所幸误会发现得及时，为了降低开发的困难和提升开发速度，虽然会使程序风格的整体性和一致性遭到破坏，我们还是选择按照各自的思路继续深入。

在此背景下，语义分析中的符号表仅供 debug 使用，它只记录每个符号被定义时的初始状态，采取递归下降遍历语法树的方案，分析可能存在的语义错误。这样就不用在整个语义分析过程中精心维护符号表，只用创建该表，然后读取该表的条目即可，涉及到代码段中的符号使用和变化，结合 C++ 的 STL 模板，以语句为单位进行解析，创建临时的容器存储语句中的符号，分析完一条语句就刷新容器。这样除了在创建符号和遍历符号表的时候，可以避免使用链表，大大降低了开发的难度和出错的可能，同时也提高了语义分析的速度。而对于开发中间代码及后续过程的同学而言，他所使用的剪枝方法，极大地压缩了臃肿的语法树，也可以看作是语义分析的一种优化。

除了在技术方面选择容器而非链表，以尽可能减小符号表的规模之外，我们还根据我们语言的简单特性，对表达式的语义分析采用了简化的处理方案。由于我们做出了以下的假设：

①函数的返回类型只能是int；

②函数的参数不能是const；

③只支持int数据类型；

④不支持数组的指针表达部分；

⑤没有其他的数据类型或数据结构。

我们很容易得出以下结论：

\*除了等于和不等于表达式，只要表达式里出现代表数组的变量名，就是一个错误的表达式。

而需要处理\*中唯一的例外：合法地函数调用数组是表达式的一部分。

基于此结论检查表达式的语义错误的效率将会大大提高，无需再为表达式维护数据结构，只需要关注代码段中有没有出现数组变量名即可。当然，随之而来的缺点就是需要为等于和不等于表达式的类型检查设计单独的机制。

除了上述时间和空间方面的考虑，在语义分析检测错误环节，我们并没有设计额外的代码优化内容。一种可能的想法是在语义分析阶段就计算出一些表达式的值，这样的确可以更精细地分析错误，压缩符号表或 AST 的空间，提升效率，但是开发也更为困难。由于产生了前述的误会，我们是在比较“臃肿”的树上直接进行的语义分析，递归下降的时候有很多冗余的子节点，在语义分析中动态计算符号值还要求额外分析运算符的优先级和结合性，考虑到链表在函数之间的传递和其他多出来的内容，感觉得不偿失，还不如就直接剪枝后继续开发，不在语义阶段提供优化。

### 4.3 中间代码生成的优化考虑

借助LLVM库函数：

static std::unique\_ptr<legacy::FunctionPassManager> TheFPM;

Function \*TheFunction;

TheFPM->run(\*TheFunction);

以函数为单位进行中间代码优化。

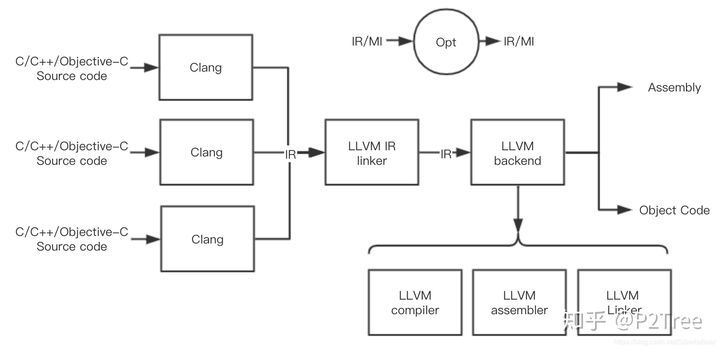
## 第五章、代码生成 （所有语句的代码生成的处理）

### 5.1 中间代码生成：生成LLVM的IR

1. **LLVM简介**

LLVM最初设计时，目标是做优化方面的研究，所以只是想搭建一套虚拟机，当时的全称叫Low Level Virtual machine，后来因为要变成编译器，官方就放弃了这个称呼，但LLVM的简称被保留下来。因为LLVM只是一个编译器框架，所以还需要一个前端来支撑整个系统，由此Clang被研发了出来。而我们的编译器后端采用了LLVM，因此我们需要生成LLVM IR作为中间代码链接我们的词法语法分析到LLVM后端。

LLVM的简单架构如下图：



1. **LLVM中间代码格式说明(仅简单介绍我们使用的部分)**

**1.** 注释

在LLVM IR中，注释以;开头，并一直延伸到行尾。

**2.** 在LLVM IR中，开头一定有：target datalayout，它注明了目标汇编代码的数据分布。这里我们是在代码中使用LLVM提供的函数自动获取的。

**3.** main函数

define i32 @main(){}是定义主函数的方式，如最简单翻译：

|  |  |
| --- | --- |
| int main() {  return 0;  } | define i32 @main() {  ret i32 0  } |

**4.** LLVM IR中数据表示及赋值

· 存储在数据区中的全局变量：

*@global\_variable = global i32 0*

(定义了一个i32类型的全局变量@global\_variable，并且将其初始化为0)

· 只读的全局变量，即常量：

*@global\_constant = constant i32 0*

(定义了一个i32类型的全局常量@global\_ constant，并且将其初始化为0)

· 虚拟寄存器

对于寄存器而言，我们只需要像普通的赋值语句一样操作，但需要注意名字必须以%开头。如：

*%local\_variable = add i32 1, 2*

此时，%local\_variable这个变量就代表一个寄存器，它此时的值就是1和2相加的结果。

如果我们把所有没有被保留的寄存器都用光了，那么LLVM IR会帮我们把这些被保留的寄存器放在栈上，然后继续使用这些被保留寄存器。当函数退出时，会帮我们自动从栈上获取到相应的值放回寄存器内。由此简化了操作。

· 栈上的变量

在需要操作地址以及需要可变变量（之后会提到为什么）时，我们就需要使用栈。LLVM IR对栈的使用十分简单，直接使用alloca指令即可。如：

*%local\_variable = alloca i32*

表示声明一个在栈上的变量。

· 全局变量和栈上变量的获取与赋值操作

如果我们要将值存储到全局变量或栈上变量，需要store命令：

*store i32 1, i32\* @global\_variable*

这个代表将i32类型的值1赋给i32\*类型的全局变量@global\_variable所指的内存区域中，栈上变量赋值也类似。

如果我们要获取@global\_variable的值，就需要

*%1 = load i32, i32\* @global\_variable*

这个指令的意思是，把一个i32\*类型的指针@global\_variable的i32类型的值赋给虚拟寄存器%1

**5.** LLVM IR中的类型

这里我们只用到了整型(整型是指i1, i8, i16, i32, i64这类的数据类型)也就是int类型。

**6.** LLVM IR中的控制语句

· Label：标签与汇编语言的标签一致，也是以:结尾作标记

· 比较指令

LLVM IR提供的比较指令为icmp。其接受三个参数：比较方案以及两个比较参数，如：

*%comparison\_result = icmp uge i32 %a, %b*

uge是比较方案，%a和%b就是用来比较的两个数，而icmp则返回一个i1类型的值，也就是C++中的bool值，用来表示结果是否为真。

uge可以是：eq与ne，分别代表相等或不相等；无符号的比较ugt, uge, ult, ule，分别代表大于、大于等于、小于、小于等于；有符号的比较sgt, sge, slt, sle，功能类比无符号的比较符。

· 条件跳转：

LLVM IR提供的条件跳转指令是br，其接受三个参数，第一个参数是i1类型的值，用于作判断；第二和第三个参数分别是值为true和false时需要跳转到的标签。如：

*br i1 %comparison\_result, label %A, label %B*

表示当%comparison\_result的值(上面的比较结果)，为true去label A，为false去label B

· 无条件跳转

在LLVM IR中，我们同样可以使用br进行条件跳转。如，如果要直接跳转到start标签处，则可以：

*br label %start*

· 接着借助Label、比较、条件跳转和无条件跳转我们就可以构造出while和if-else语句出来。

**7.** LLVM IR中的函数

· 函数定义

一个函数定义最基本的框架，就是返回值(i32)+函数名(@foo)+参数列表((i32 %a, i64 %b))+函数体({ ret i32 0 }),如：

|  |
| --- |
| define i32 @foo(i32 %a, i64 %b) {  ret i32 0  } |

· 函数声明

函数声明就是使用declare关键词替换函数定义的define，如：

*declare i32 @foo(i32 %a, i64 %b)*

· 函数调用

在LLVM IR中，函数的调用与高级语言几乎没有什么区别。使用call指令可以像高级语言那样直接调用函数。如：

*%1 = call i32 @foo(i32 1, i64 2)*

1. **借助LLVM生成中间代码**

**1.** 撰写生成AST代码

这里我们选择自己撰写代码生成便于我们撰写codeGen的AST，同时这减少了中间代码生成和语义分析的耦合度，中间代码将基于语法分析树提取AST，而语义分析将直接使用语法生成树。

· proc\_CompUnit：解析开始符号CompUnit对应节点，并调用解析其生成的节点，是执行AST生成的入口：

|  |
| --- |
| std::vector<  std::variant<std::unique\_ptr<FunctionAST>, std::unique\_ptr<ExprAST>>>  proc\_CompUnit(const grammarTree \*r); |

· get\_FuncDef\_AST：解析FuncDef对应节点，并调用解析其生成的节点：

|  |
| --- |
| std::unique\_ptr<FunctionAST> get\_FuncDef\_AST(const grammarTree \*r); |

· get\_Decl\_AST：解析Decl对应节点，并调用解析其生成的节点：

|  |
| --- |
| std::unique\_ptr<VarDefAST> get\_Decl\_AST(const grammarTree \*r,  bool isGlbl = false); |

· get\_Stmt\_AST：解析Stmt对应节点，并调用解析其生成的节点：

|  |
| --- |
| std::unique\_ptr<ExprAST> get\_Stmt\_AST(const grammarTree \*r); |

· get\_Block\_AST：解析Block对应节点，并调用解析其生成的节点：

|  |
| --- |
| std::unique\_ptr<BlockAST> get\_Block\_AST(const grammarTree \*r); |

· get\_Exp\_AST：解析Exp对应节点，并调用解析其生成的节点：

|  |
| --- |
| std::unique\_ptr<ExprAST> get\_Exp\_AST(const grammarTree \*r); |

· get\_LVal\_AST：解析LVal对应节点，并调用解析其生成的节点：

|  |
| --- |
| std::unique\_ptr<VariableExprAST> get\_LVal\_AST(const grammarTree \*r); |

· get\_BinExpr\_AST：解析BinExpr对应节点，并调用解析其生成的节点：

|  |
| --- |
| std::unique\_ptr<BinaryExprAST>  get\_BinExpr\_AST(const grammarTree \*r,  std::stack<std::unique\_ptr<ExprAST>> &out); |

**2.** 基于AST撰写每个AST结构对应的codegen()

· BlockAST::codegen()：调用Block的孩子节点的codegen()

|  |
| --- |
| Value \*BlockAST::codegen() |

· NumberExprAST::codegen()：调用llvm Build创建一个int数值

|  |
| --- |
| Value \*NumberExprAST::codegen() |

· VariableExprAST::codegen()：调用llvm Build创建对应变量

|  |
| --- |
| Value \*VariableExprAST::codegen() |

· UnaryExprAST::codegen()：层层向下解析AST，并调用llvm Build创建单目运算代码

|  |
| --- |
| Value \*UnaryExprAST::codegen() |

· BinaryExprAST::codegen()：层层向下解析AST，并调用llvm Build创建双目运算代码

|  |
| --- |
| Value \*BinaryExprAST::codegen() |

· VarAssignAST::codegen()：调用llvm Build创建赋值代码

|  |
| --- |
| Value \*VarAssignAST::codegen() |

· ReturnAST::codegen()：调用llvm Build创建return代码

|  |
| --- |
| Value \*ReturnAST::codegen() |

· CallExprAST::codegen()：调用llvm Build创建函数调用代码

|  |
| --- |
| Value \*CallExprAST::codegen() |

· IfAST::codegen()：调用llvm Build创建if控制语句代码

|  |
| --- |
| Value \*BlockAST::codegen() |

·WhileAST::codegen()：调用llvm Build创建while控制语句代码

|  |
| --- |
| Value \*WhileAST::codegen() |

·GotoAST::codegen()：调用llvm Build创建goto控制语句代码

|  |
| --- |
| Value \*GotoAST::codegen() |

·VarDefAST::codegen()：调用llvm Build创建变量声明代码

|  |
| --- |
| Value \*VarDefAST::codegen() |

·GlblVarDefAST::codegen()：调用llvm Build创建全局变量声明代码

|  |
| --- |
| Value \*GlblVarDefAST::codegen() |

### 5.2 目标代码生成：基于LLVM

在中间代码生成中，我们生成了LLVM IR的可视化文件.ll文件，这里以main.ll文件为例演示如何生成汇编代码：

|  |
| --- |
| llvm-as main.ll -o main.bc  llc main.bc -o main.s |

其中生成的main.bc是LLVM IR的字节码格式，main.s是生成的汇编代码。

同时我们也可以直接运行中间代码：

|  |
| --- |
| lli --dlopen="x64/Debug/libsysy.dll" .\main.ll |

这里我们必须链接我们的输入输出用的动态库libsysy.dll。

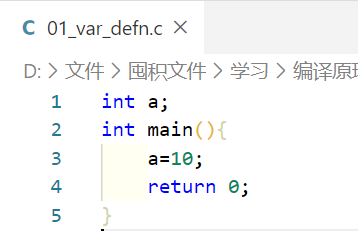
## 第六章、测试案例 （每个语句成分的测试案例，至少两个复杂语句组合后的测试案例）

### 6.1语法分析功能性测试

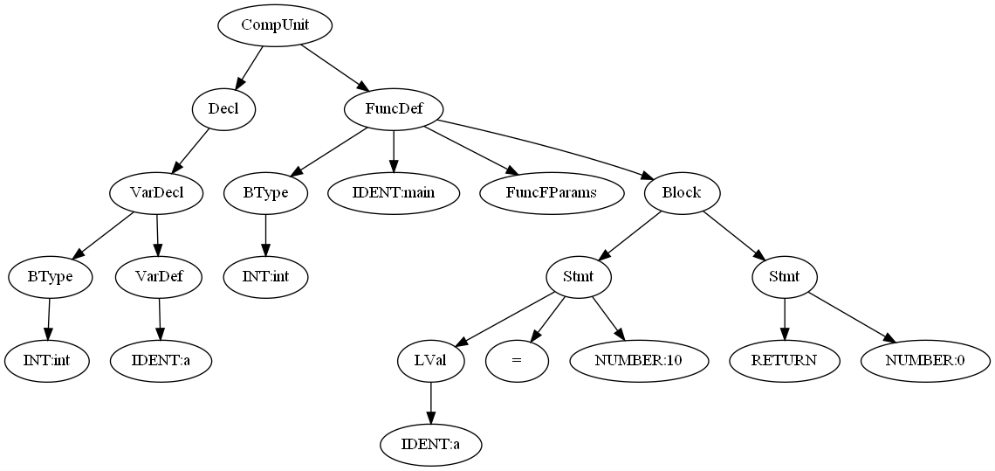
语法分析测试所有的测试样例在文件./code/functional\_test /文件夹内，而输出的可视化语法树均生成在文件夹./code/viewTree/内，测试时在Window环境下将VS生成的exe文件拷入code目录，然后在code目录下命令行运行 .\test\_syntax.bat。运行过程中有些文件需要输入，这些输入在./code/functional\_test/input.txt内提供。此处仅展示几个较简单的测试案例（因为报告篇幅原因，语法树可能放不下）

**6.1.1 样例1：**

输入代码：01\_var\_defn.c--main函数+声明+赋值

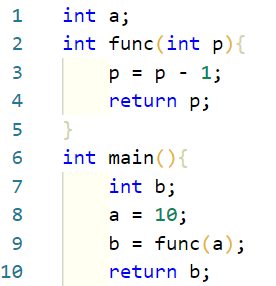


输出语法树：

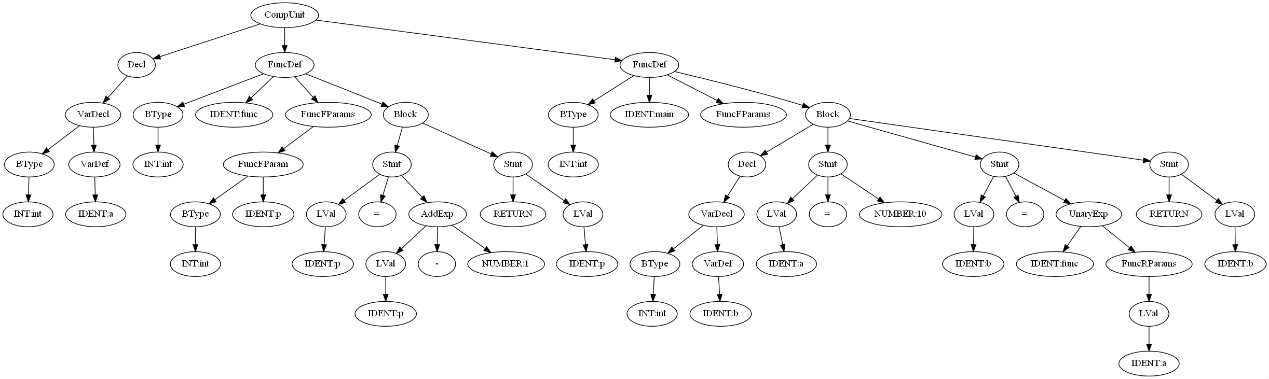


**6.1.2样例2：**

输入代码：04\_func\_defn.c--函数定义+函数调用+声明+运算

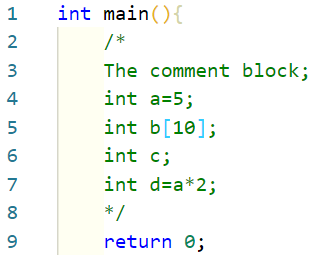


输出语法树：

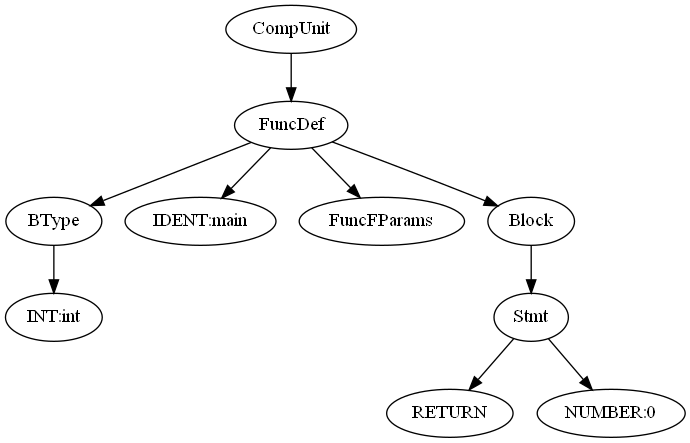


**6.1.3样例3：**

输入代码：08\_comment2.c--注释

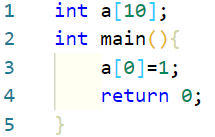


输出语法树：

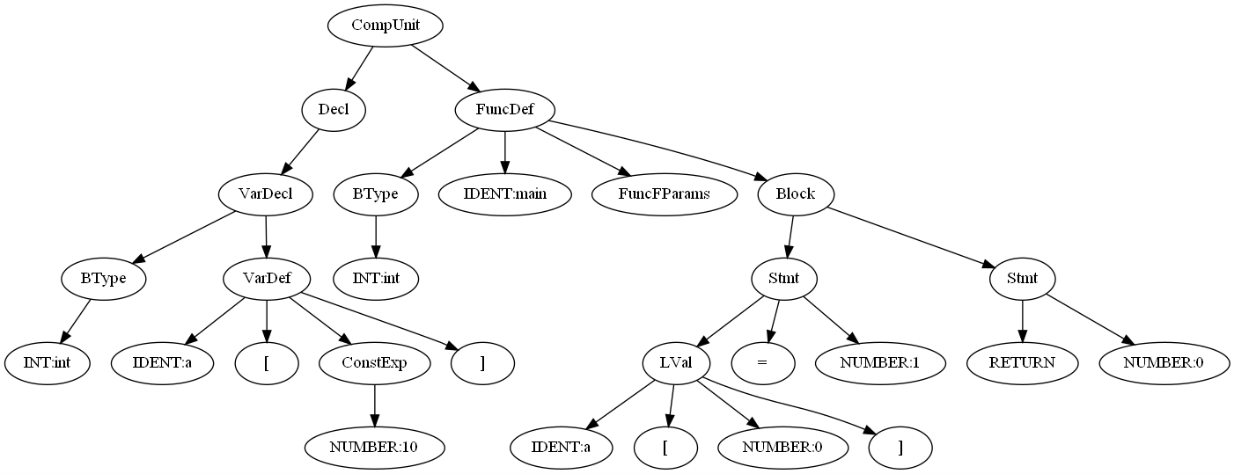


**6.1.4样例4：**

输入代码：08\_arr\_assign.c--数组声明及赋值

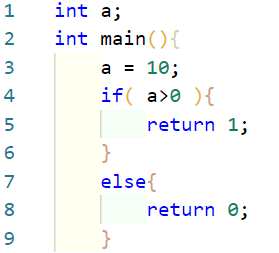


输出语法树：

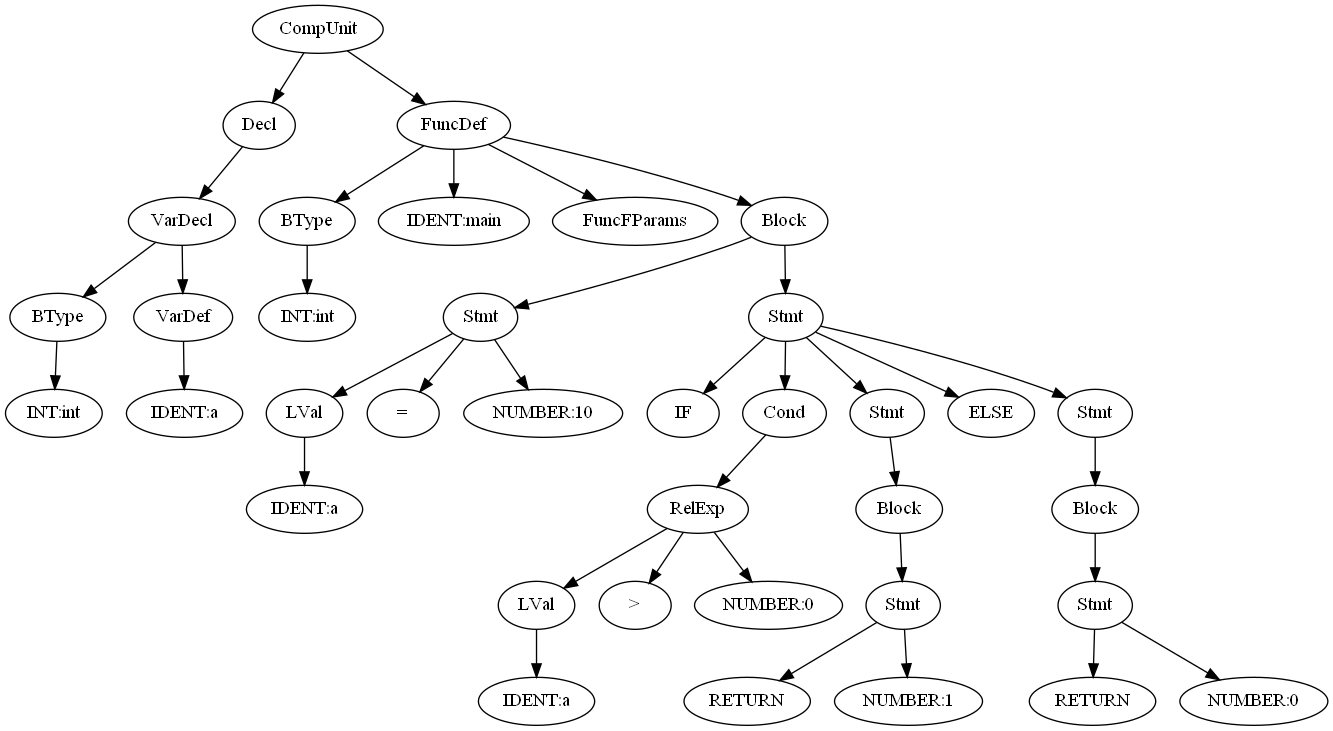


**6.1.5样例5：**

输入代码：10\_if\_else.c--if-else

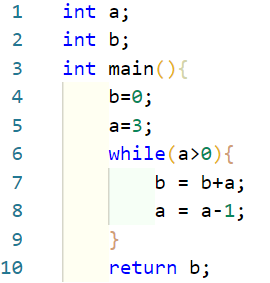


输出语法树：

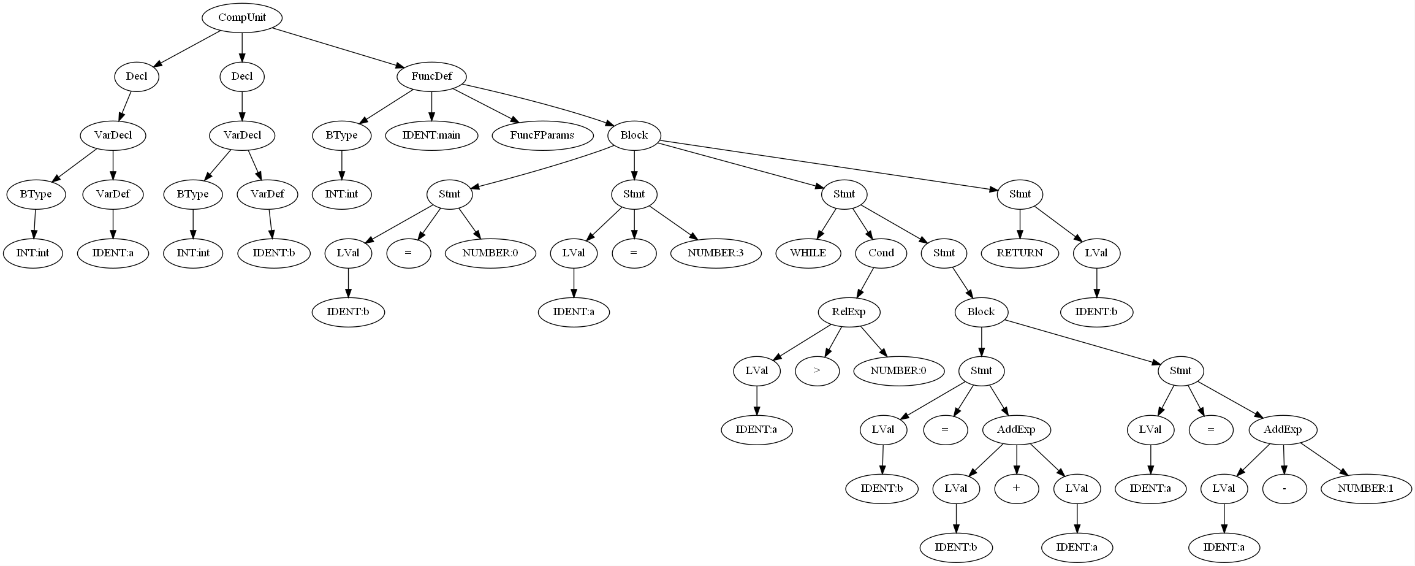


**6.1.6样例6：**

输入代码：11\_while.c--while



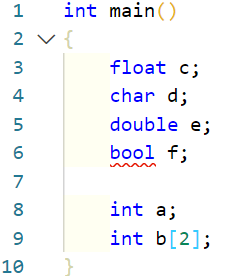
输出语法树：



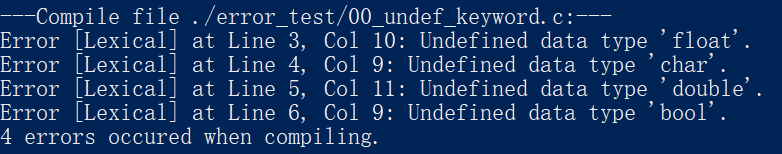
### 6.2 错误检测测试（即语义分析测试+语法词法纠错测试）

**6.2.1 样例1：**

输入错误代码：出现我们语言不支持的C关键字

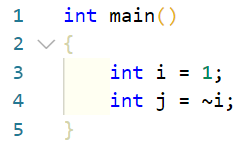


输出错误：

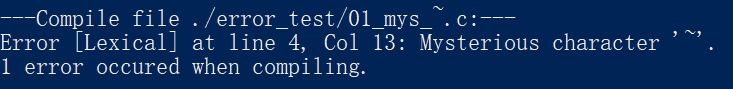


**6.2.2样例2：**

输入错误代码：未知的符号

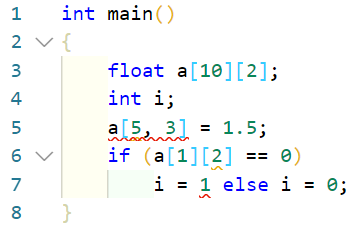


输出错误：

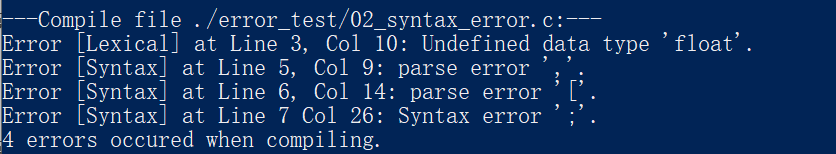


**6.2.3样例3：**

输入错误代码：语法错误混合未定义关键字错误

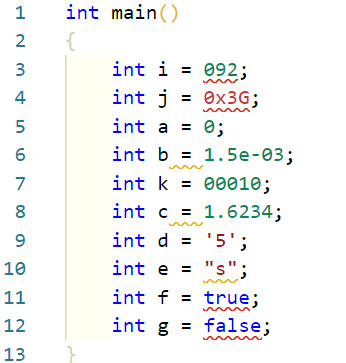


输出错误：

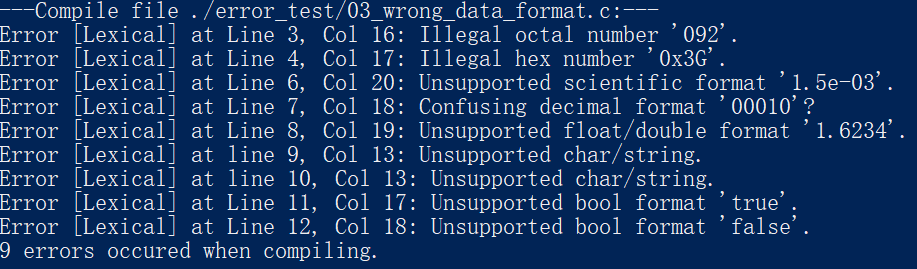


**6.2.4样例4：**

输入错误代码：错误的数据表示形式及不支持的数据常量

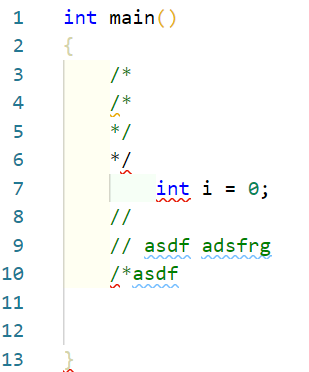


输出错误：

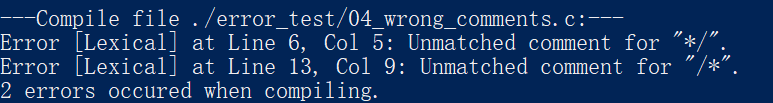


**6.2.5样例5：**

输入错误代码：错误注释

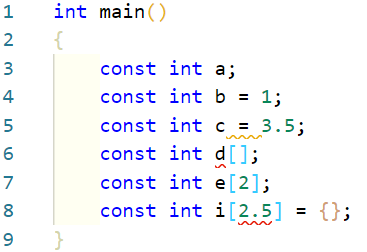


输出错误：

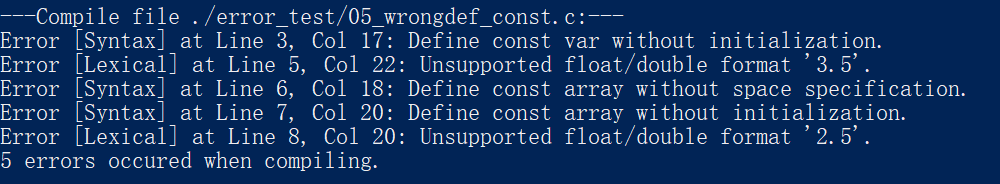


**6.2.6样例6：**

输入错误代码：错误的常量定义（我们要求常量必须在定义时初始化）+不支持的数据常量+错误的常量数组定义（我们要求常量数组必须在定义时初始化，并且要求必须指明数组长度，而且长度必须为整数）

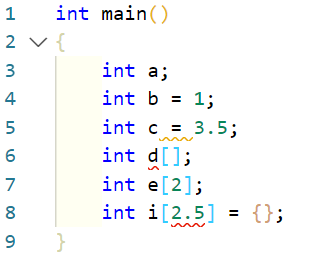


输出错误：

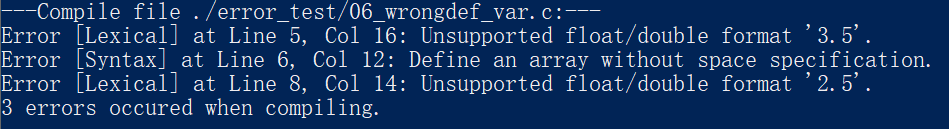


**6.2.7样例7：**

输入错误代码：不支持的数据常量+错误的整型数组定义（我们要求整型数组必须指明数组长度，而且长度必须为整数）

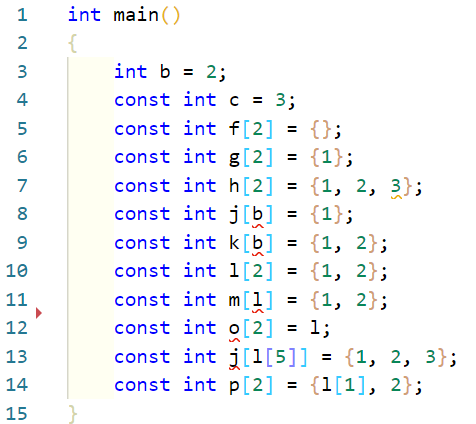


输出错误：

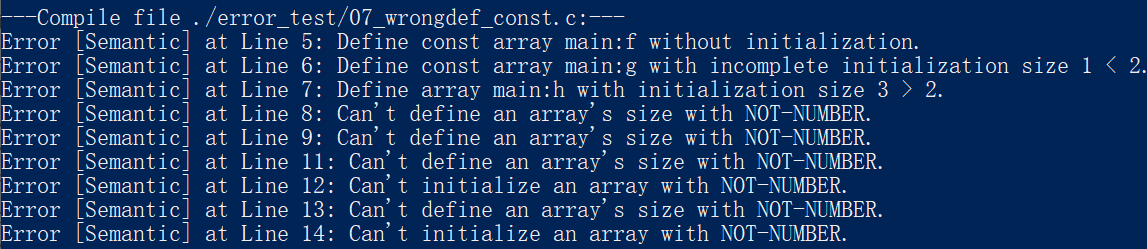


**6.2.8样例8：**

输入错误代码：未初始化完全或多初始化的常量数组(我们要求常量数组必须按长度初始化)+数组长度出现非数字

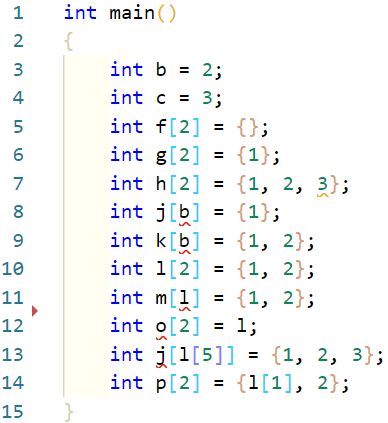


输出错误：

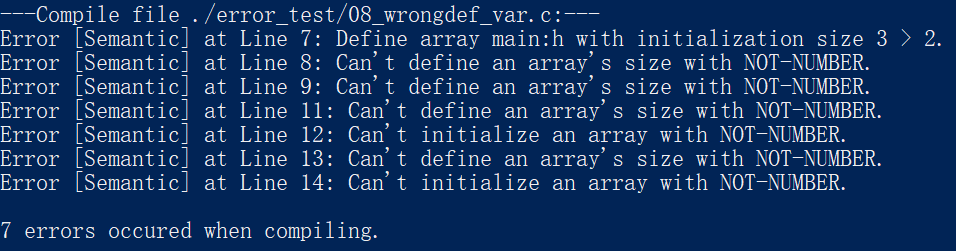


**6.2.9样例9：**

输入错误代码：未初始化完全或多初始化的整型数组(我们要求整型数组初始化可以可变长度，但是不能超出数组长度限制)+数组长度出现非数字

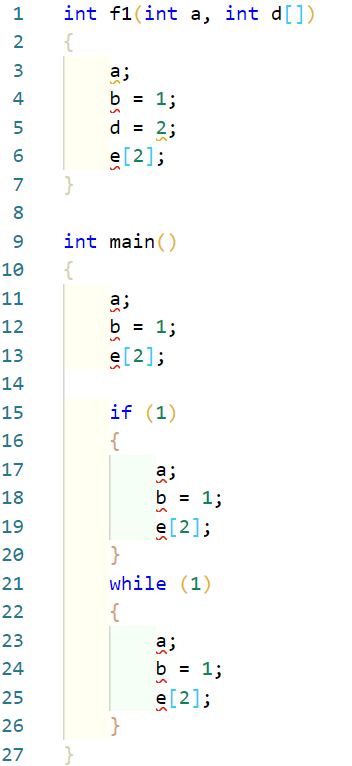


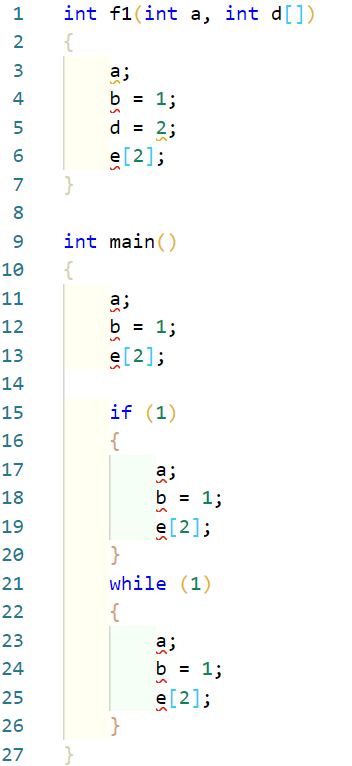
输出错误：



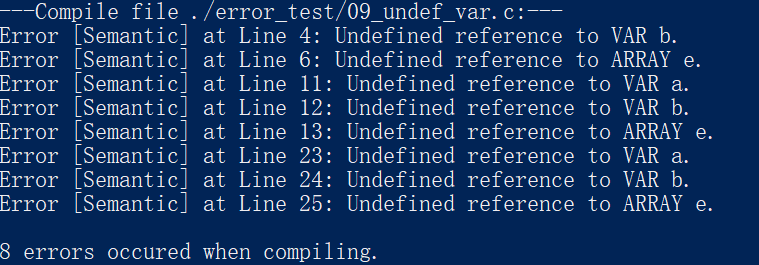
**6.2.10样例10：**

输入错误代码：未声明/定义的数组、变量引用



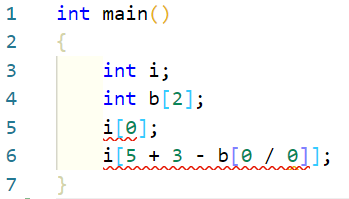


输出错误：

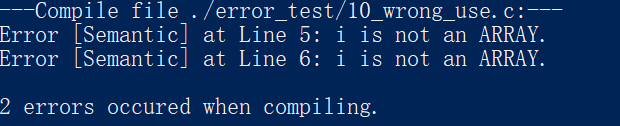


**6.2.11样例11：**

输入错误代码：将非数组当作数组使用

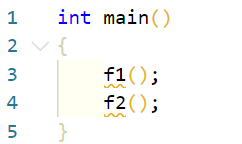


输出错误：

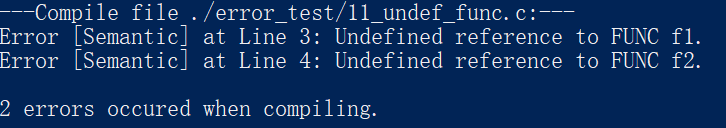


**6.2.12样例12：**

输入错误代码：未声明/定义的函数引用

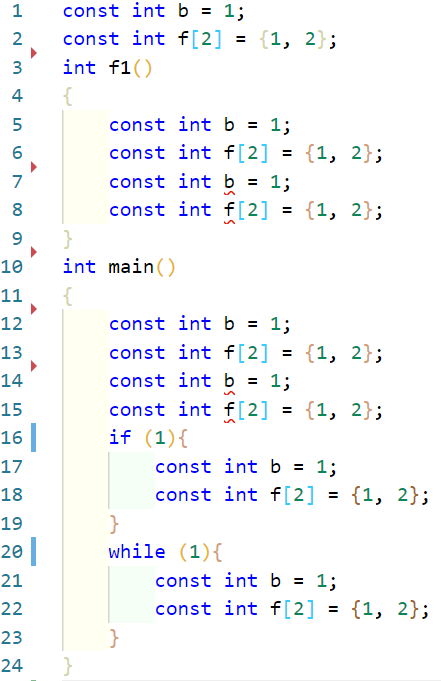


输出错误：

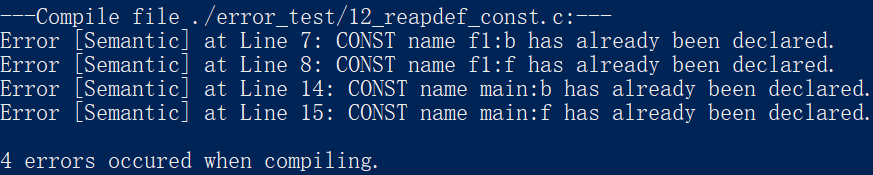


**6.2.13样例13：**

输入错误代码：常量的重复定义错误

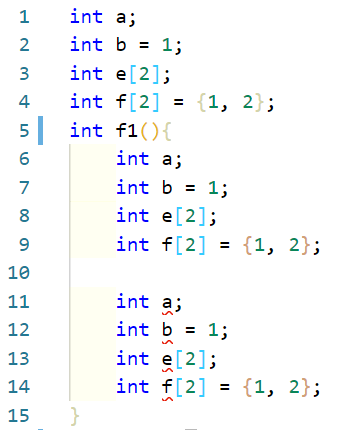


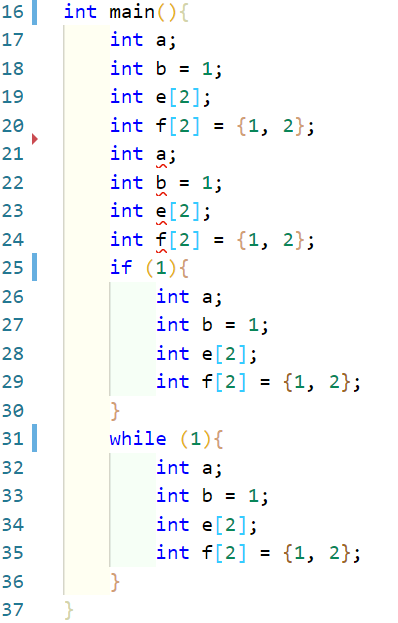
输出错误：



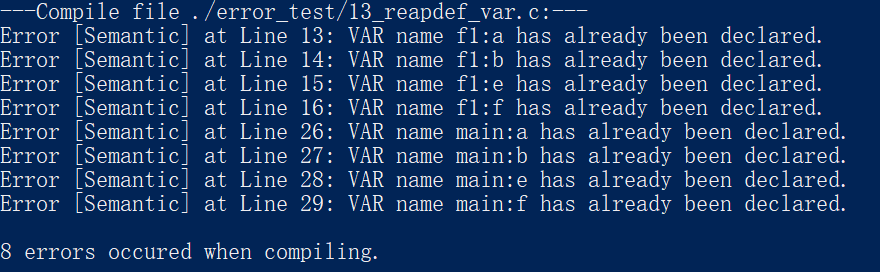
**6.2.14样例14：**

输入错误代码：整型变量的重复定义错误



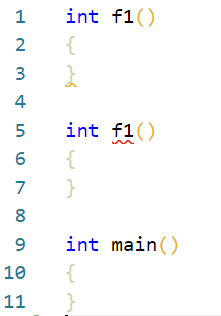


输出错误：

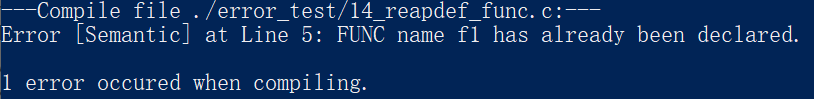


**6.2.15样例15：**

输入错误代码：函数名的重复定义错误

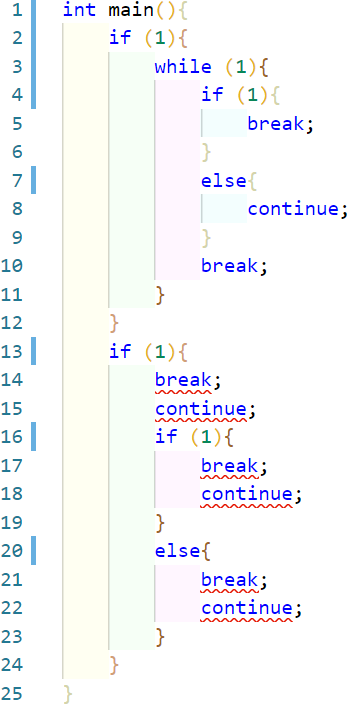


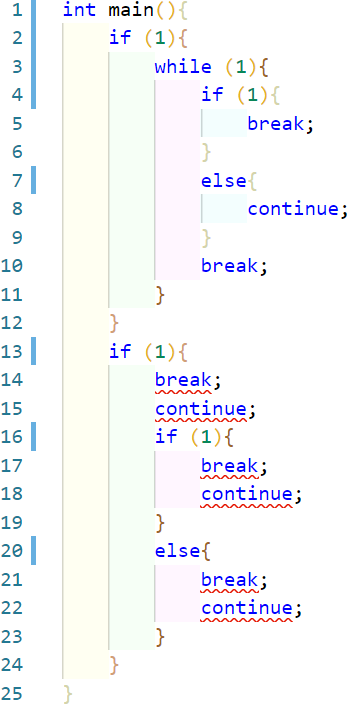
输出错误：



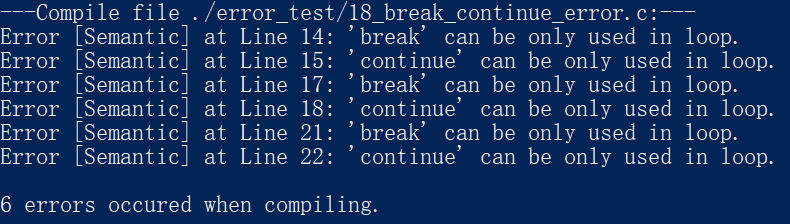
**6.2.16样例16：**

输入错误代码：break、continue的错误使用（我们规定break、continue只能再循环题中使用）



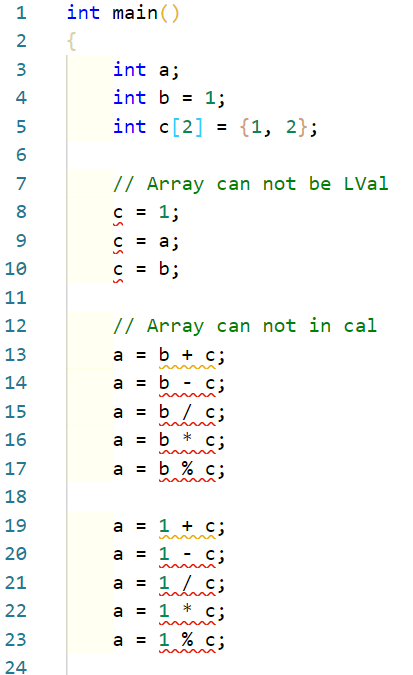


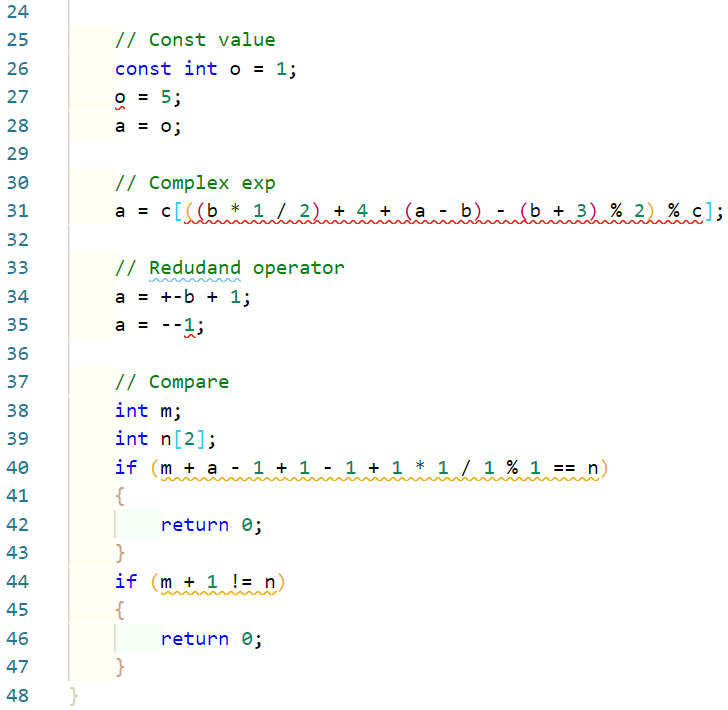
输出错误：



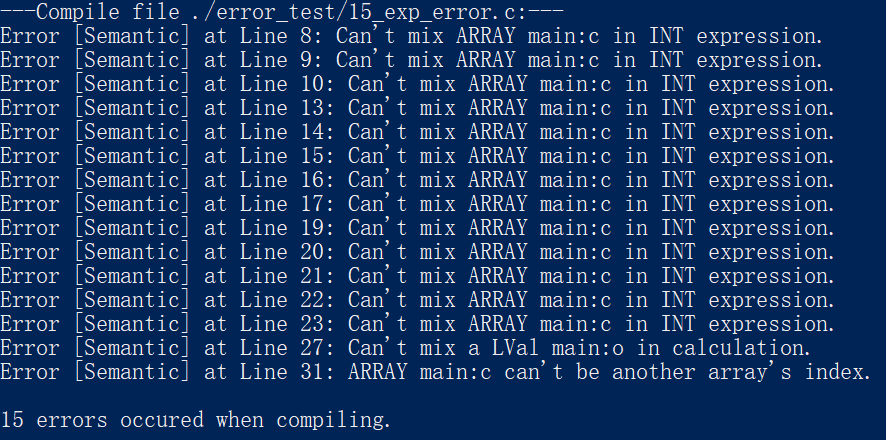
**6.2.17样例17：**

输入错误代码：表达式的错误使用



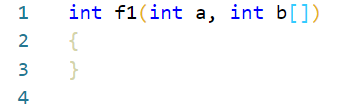


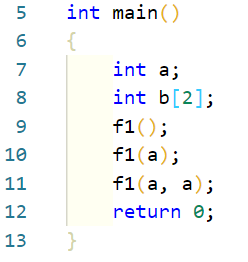
输出错误：



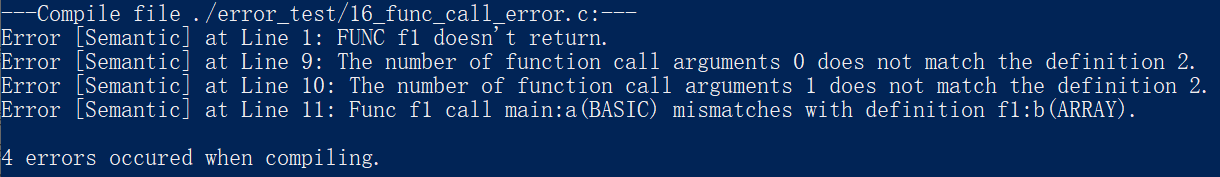
**6.2.18样例18：**

输入错误代码：函数调用格式有误



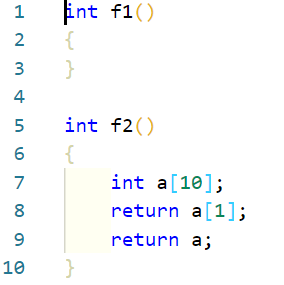


输出错误：

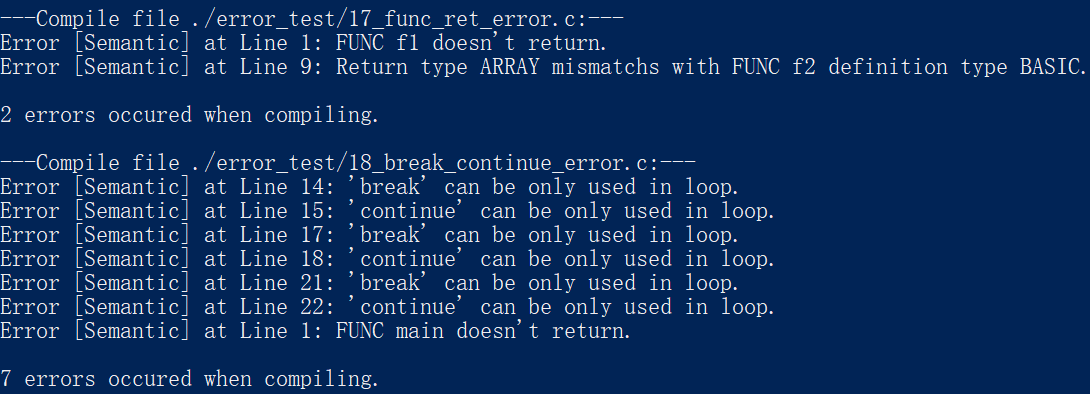


**6.2.19样例19：**

输入错误代码：函数返回错误



输出错误：



**6.2.20样例20：**

输入错误代码：空文件



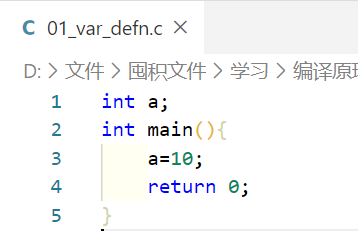
输出错误：



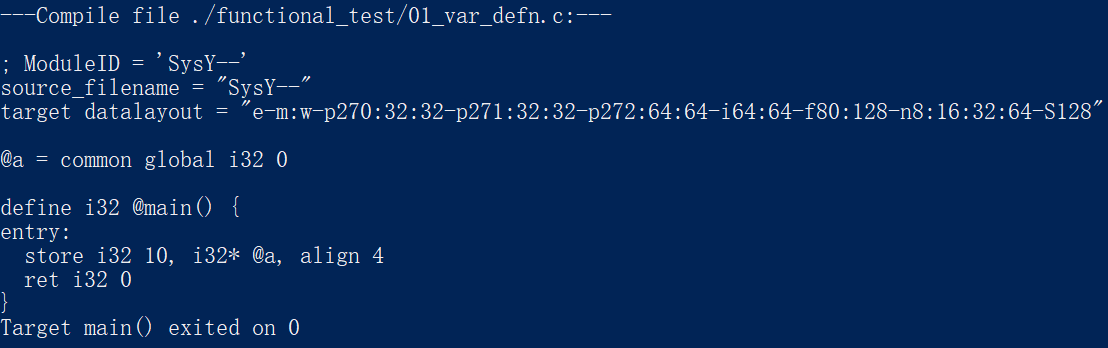
### 6.3 中间代码及目标代码生成测试

**6.3.1 样例1：**

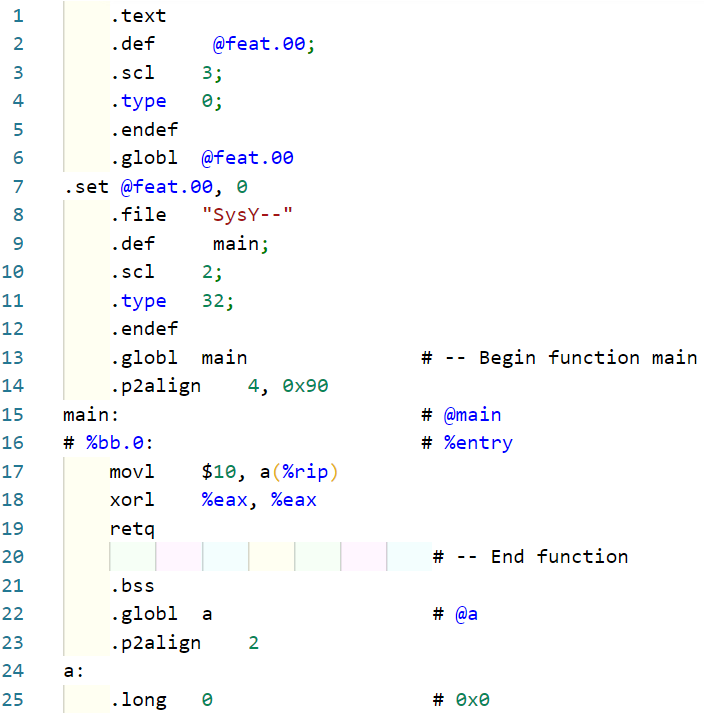
输入类C代码： 01\_var\_defn.c--main函数+声明+赋值

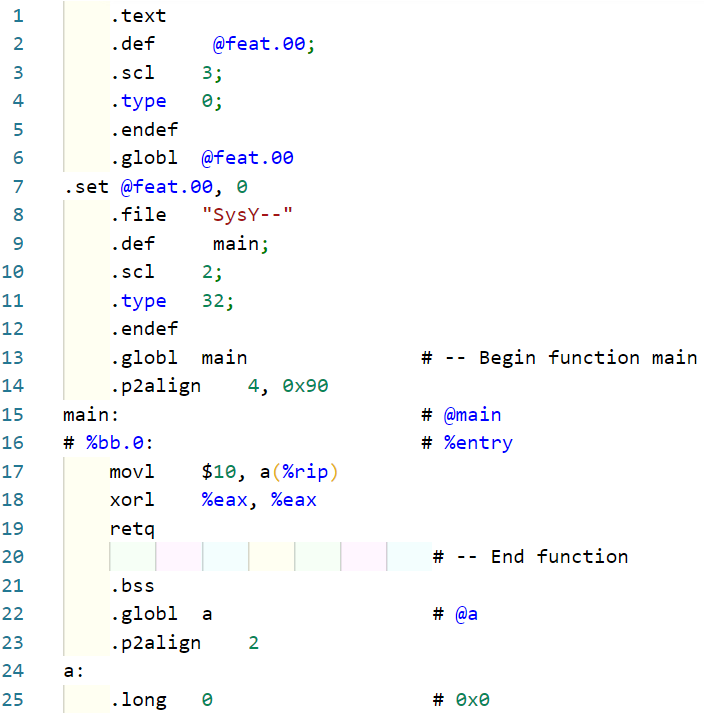


输出中间代码：



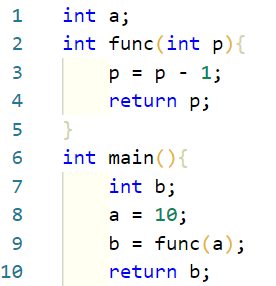
输出目标代码：



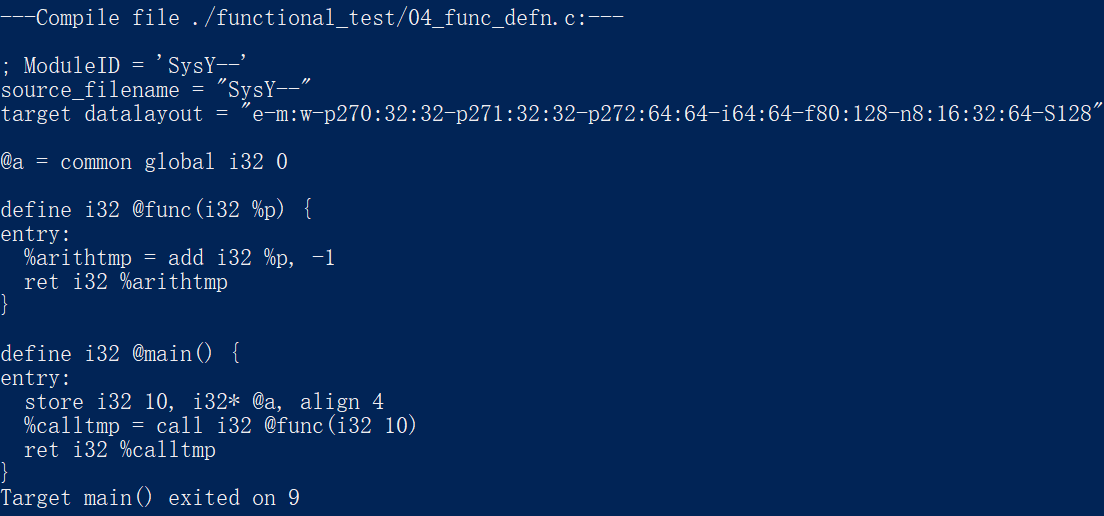


**6.3.2样例2：**

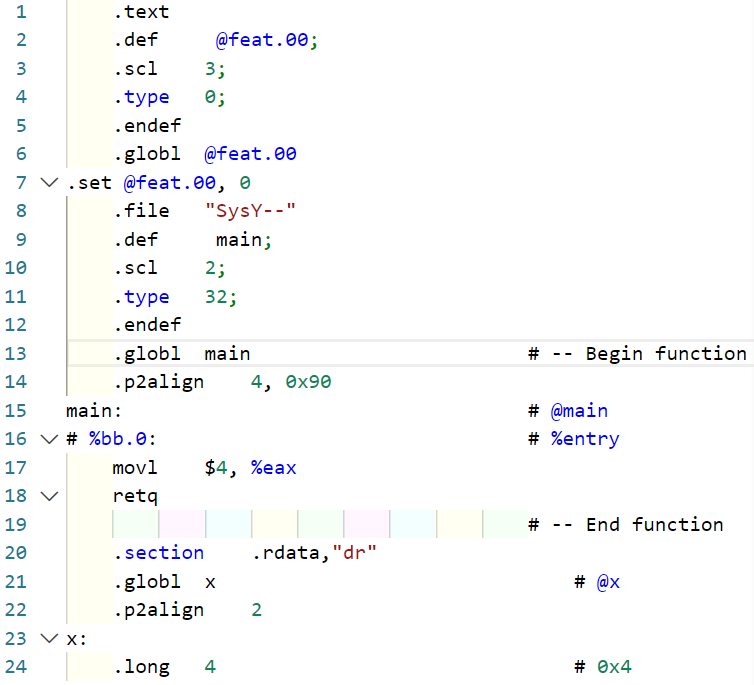
输入类C代码：04\_func\_defn.c--函数定义+函数调用+声明+运算



输出中间代码：

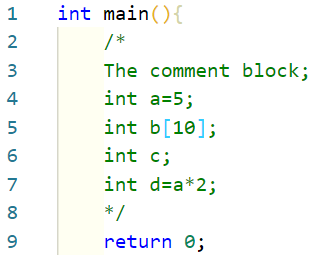


输出目标代码：

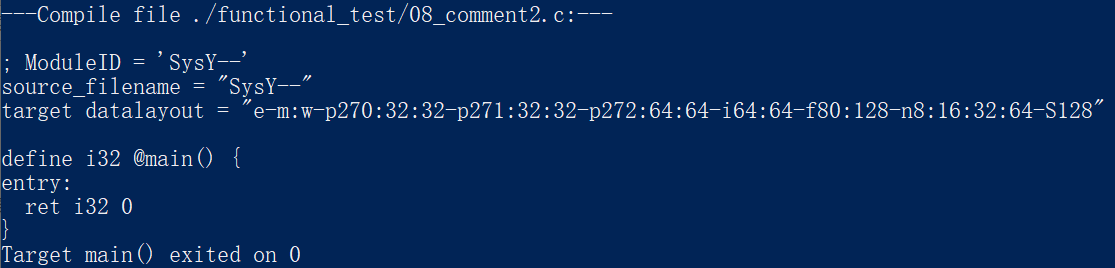


**6.3.3样例3：**

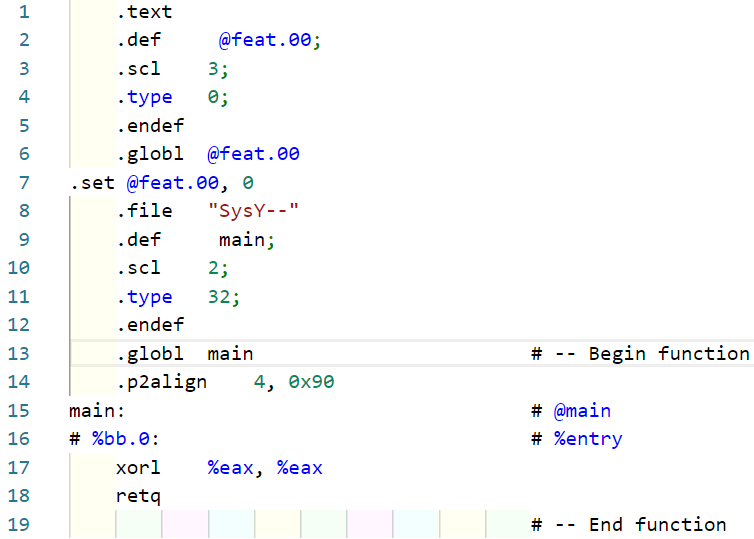
输入类C代码：08\_comment2.c--注释



输出中间代码：

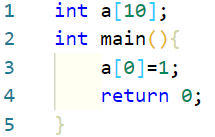


输出目标代码：

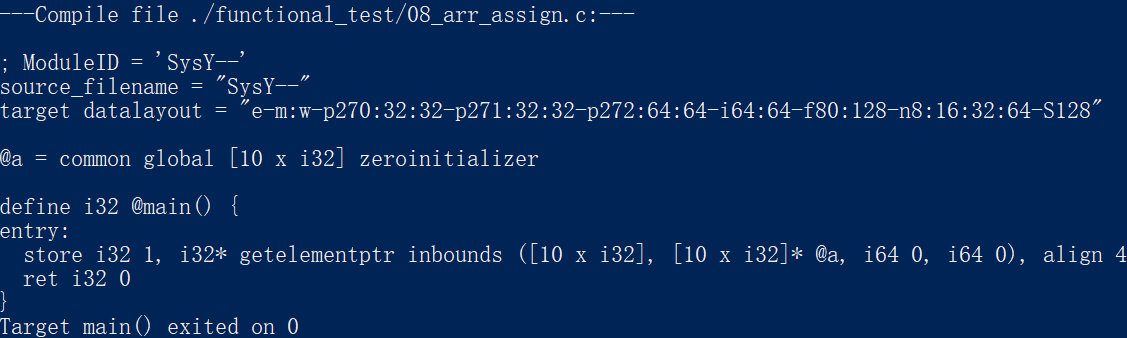


**6.3.4样例4：**

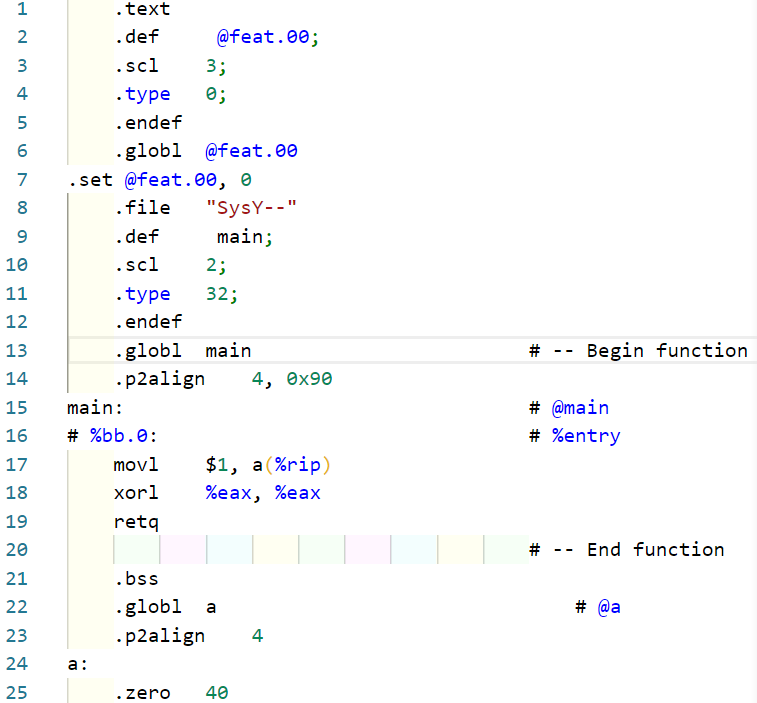
输入类C代码：08\_arr\_assign.c--数组声明及赋值

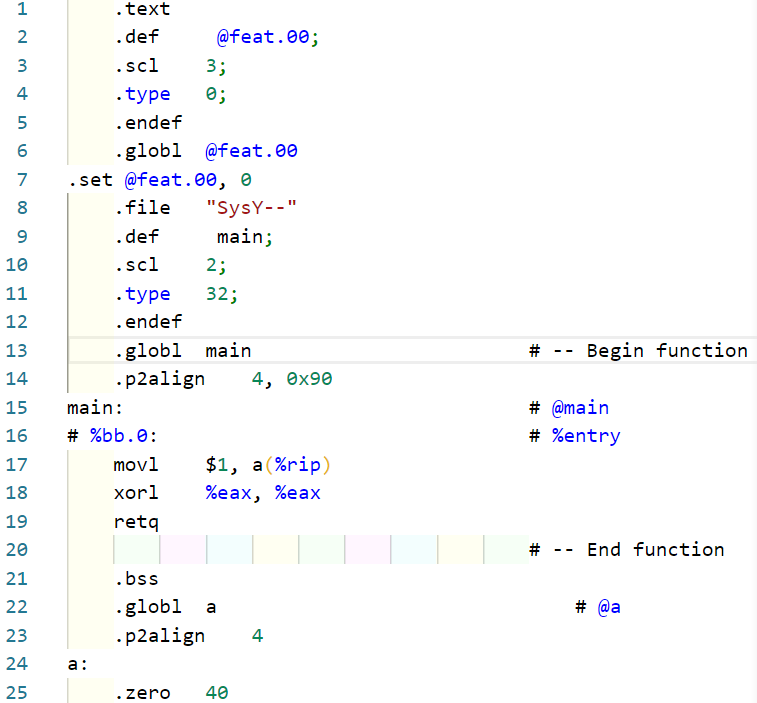


输出中间代码：



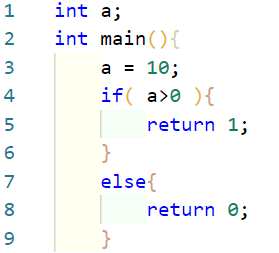
输出目标代码：



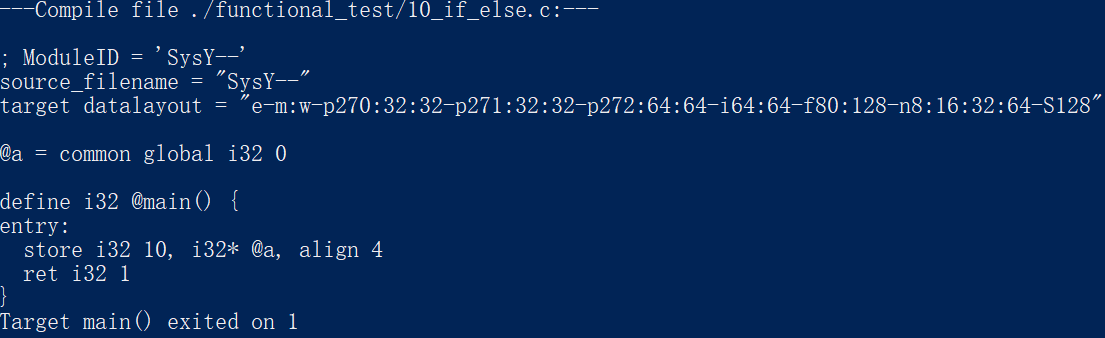


**6.3.5样例5：**

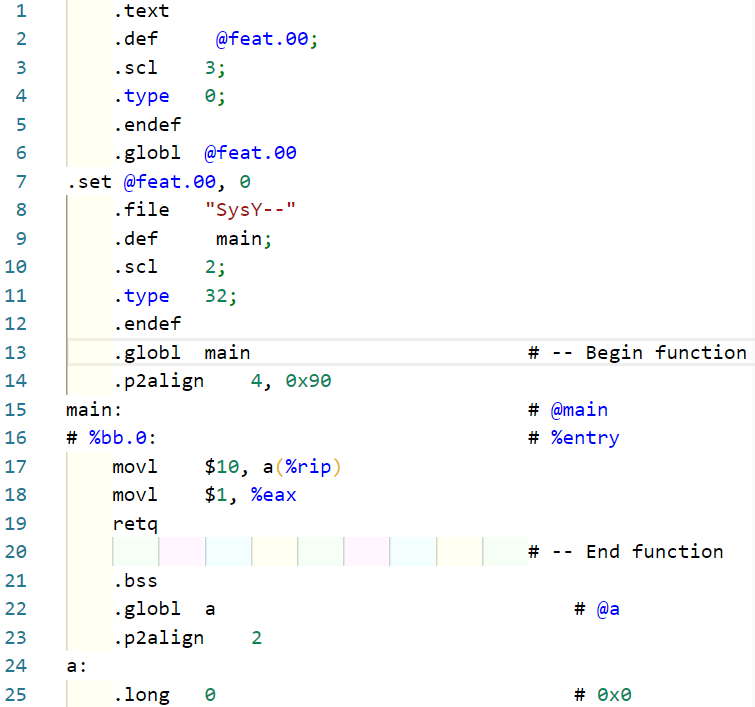
输入类C代码：10\_if\_else.c--if-else

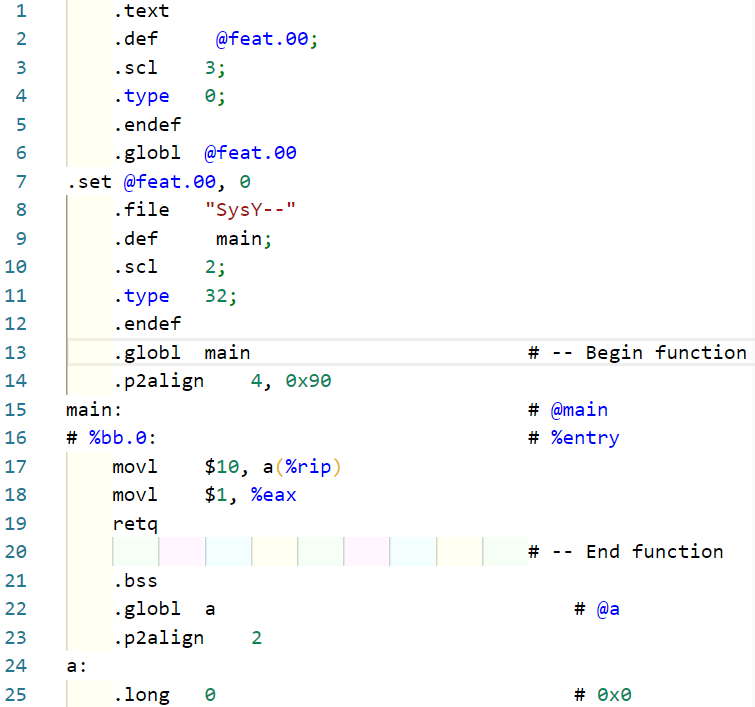


输出中间代码：



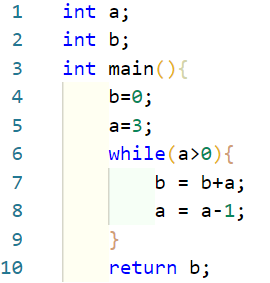
输出目标代码：



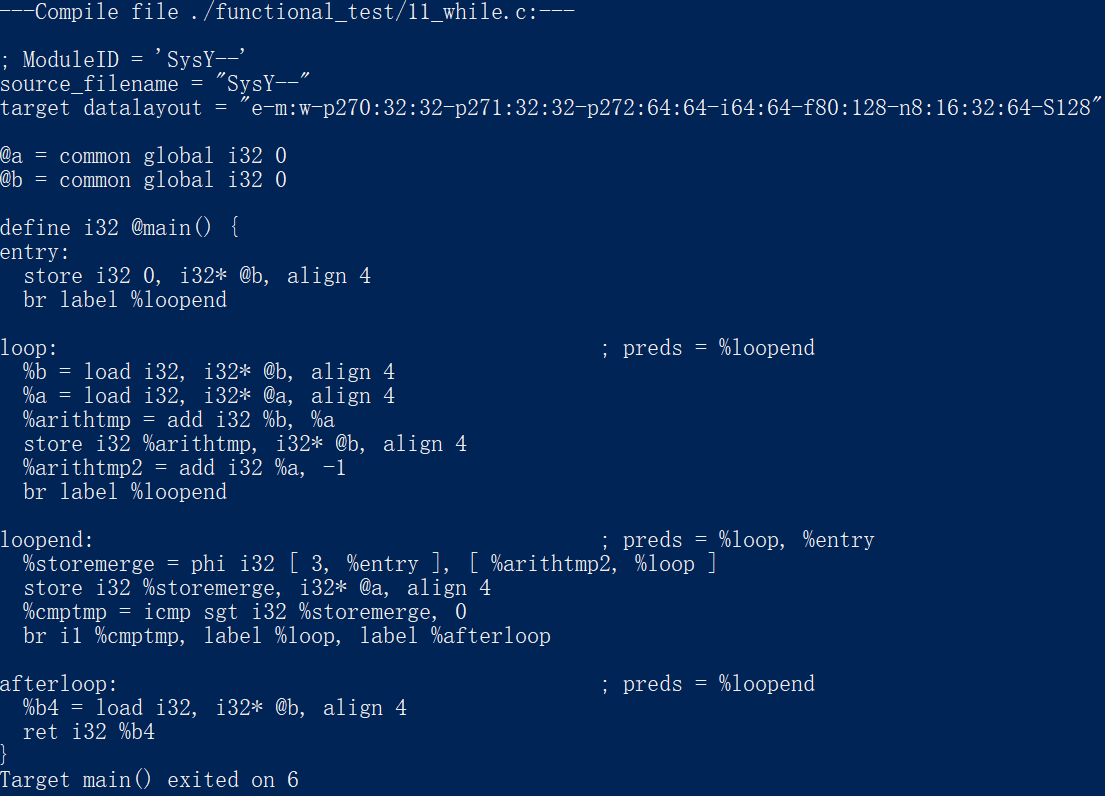


**6.3.6样例6：**

输入类C代码：11\_while.c--while



输出中间代码：



输出目标代码：

