

**编译原理实验**

**课程报告**

**TOY语言编译器**

班 级 IS1701

姓 名 李强

学 号 U201714559

指导教师 刘铭

报告日期 2019/12/31

网络空间安全学院

**要 求**

1、报告需本人独立完成，内容真实。引用资料时，需进行标注说明，并列入参考文献中；如发现抄袭，成绩无效；

2、应说明实验的操作系统环境、采用的主要方法、设计的过程、设计的结果（主要源码文件功能、数据结构、函数说明）、遇到的问题、测试运行的情况；

3、按编译原理实验内容，应包含：语言的文法、语言的词法及语法分析、语义分析、中间代码生成、目标代码生成；

4、评分标准：格式规范美观，符合华中科技大学论文格式要求；采用的方法合适、设计合理；4个主要实验环节按任务书要求完成。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **格式**  **规范** | **词法**  **语法** | **语义**  **分析** | **中间**  **代码** | **目标**  **代码** | **总分** |
| **20** | **30** | **30** | **10** | **10** | **100** |
|  |  |  |  |  |  |

**目 录**

[1 选题背景 1](#_Toc28723276)

[1.1 任务 1](#_Toc28723277)

[1.2 目标 1](#_Toc28723278)

[1.3 语言定义 1](#_Toc28723279)

[1.4 主要技术 2](#_Toc28723280)

[2 实验一 词法分析和语法分析 3](#_Toc28723281)

[2.1 单词文法描述 3](#_Toc28723282)

[2.2 语法文法描述 3](#_Toc28723283)

[2.3 词法分析器设计 4](#_Toc28723284)

[2.4 语法分析器设计 4](#_Toc28723285)

[2.5 词法及语法分析器实现结果 5](#_Toc28723286)

[2.6 小结 5](#_Toc28723287)

[3 实验二 语义分析 9](#_Toc28723288)

[3.1 语义表示方法描述 9](#_Toc28723289)

[3.2 符号表结构 9](#_Toc28723290)

[3.3 错误类型定义 10](#_Toc28723291)

[3.4 语义分析实现技术 10](#_Toc28723292)

[3.5 语义分析结果 11](#_Toc28723293)

[3.6小结 11](#_Toc28723294)

[4 实验三 中间代码生成 12](#_Toc28723295)

[4.1 中间代码格式定义 12](#_Toc28723296)

[4.2 中间代码生成规则定义 12](#_Toc28723297)

[4.3 中间代码生成过程 13](#_Toc28723298)

[4.5 中间代码生成结果 13](#_Toc28723299)

[4.6 小结 13](#_Toc28723300)

[5 实验四 目标代码生成 15](#_Toc28723301)

[5.1 目标架构选择 15](#_Toc28723302)

[5.2 目标代码生成结果 15](#_Toc28723303)

[5.3 目标代码运行结果 15](#_Toc28723304)

[5.4 小结 16](#_Toc28723305)

[6 总结与展望 17](#_Toc28723306)

[6.1 全文总结 17](#_Toc28723307)

[6.2 展望 17](#_Toc28723308)

[参考文献 18](#_Toc28723309)

# 1 选题背景

1.1 任务

通过简单自定义语言编译器的完整实现，掌握编译原理理论知识，提高灵活运用理论知识以解决实际问题的能力；提高系统软件编写能力。

1.2 目标

课程目标是构造一个高级语言的子集的编译器，目标代码是汇编语言。通过设计自定义语言的文法，进行词法分析、语法分析和语义分析最终生成中间代码IR来实现自定义语言的前端。通过设计目标机器架构寄存器调度算法，指令调度方法等实现自定义语言的后端。最后通过链接器将汇编代码转换为机器码。

1.3 语言定义

本次实验中实现的语言为TOY。

TOY语言采用了C语言中的int，float和char数据类型，函数定义和函数原型声明，while循环和if流程控制等。采用Python语言中以换行表示一条语句结束的方法，结合C语言中以大括号开始的代码域，语言中不采用分号结尾。

G[program]:

program → ExtDefList//开始

ExtDefList → ε //变量定义或函数定义或函数声明

| Specifier DecList ENTERS ExtDefList // 外部变量

| Specifier DecList ExtDefList //外部变量

| Specifier FunDec CompSt ENTERS ExtDefList //函数声明

| Specifier FunDec ENTERS ExtDefList

Specifier → int | float | char//类型

DecList → Dec | Dec , DecList //解决一次性定义多个情况

Dec → VarDec | VarDec = Exp //初始化问题

VarDec → ID//变量名字对应到ID

FunDec → ID ( VarList ) //形参比较特殊并没有分配实际的空间因此需要特殊处理

VarList → ε//无参

| ParamDec

| ParamDec , VarList //参数变量定义，同样是递归定义

ParamDec → Specifier VarDec//参数只是临时或内部变量

CompSt → { ENTERS DefList StmList }//代码段

DefList → ε | Specifier DecList ENTERS DefList

StmList → ε | Stmt StmList

Stmt → Exp ENTERS

| CompSt ENTERS

| return Exp ENTERS

| if ( Exp ) ENTERS CompSt ENTERS

| if ( Exp ) ENTERS CompSt else CompSt ENTERS

| while ( Exp ) ENTERS CompSt ENTERS

| BREAK ENTERS

Exp → Exp =Exp | Exp += Exp | Exp -= Exp | Exp \*= Exp | Exp /= Exp //复合赋值运算

| ++ Exp | -- Exp | Exp -- | Exp ++ //自增自减运算

| ! Exp | Exp && Exp | Exp || Exp//逻辑运算

| Exp < Exp | Exp <= Exp | Exp == Exp | Exp != Exp | Exp > Exp | Exp >= Exp //比较运算

| Exp + Exp | Exp – Exp | Exp \* Exp | Exp / Exp | Exp % Exp //算术运算

| ( Exp ) //括号表达式

| - Exp

| ID ( Args )

| ID

| INT | FLOAT |CHAR //终结符

Args→ ε

| Exp , Args

| Exp

ENTERS→ ENTER

| ENTER ENTERS

1.4 主要技术

编译器所在环境为Linux。通过Flex和Bison编写词法分析器和语法分析器并生成抽象语法树，通过对抽象语法树进行递归遍历进行语义分析。因为TAC序列没有一个完整的标准，并且对于与或非等逻辑表达式没有注明，对于比较运算并没有转换，这不利于翻译到汇编代码，因此采用同样是静态单赋值三地址码形式的LLVM的IR[1]。在实现TOY编译器后端时，可以借助LLVM的框架将TOY前端整合到LLVM中，或是自写特定机器架构的后端或是直接使用LLVM已经实现的部分后端，构建完整的编译器。

# 2 实验一 词法分析和语法分析

2.1 单词文法描述

主要通过正则表达式借助Flex对源程序进行处理，除去语言中的保留字和标识符等信息还剩下自定义id和数字字符等常量。其中id的正则表达式如公式2-1所示，以字母下划线开头，字母数字和下划线作为内容。INT类型的正则表达式如公式2-2所示，由1或多个0-9的数字构成，char类型的正则表达式如公式2-3所示，由单引号作为开头和结尾，float类型的正则表达式如公式2-4所示，由两个数字之间通过句号连接。

因为注释在后续处理中没有实质性作用，可以通过正则表达式过滤掉。正则表达式2-5表示以“//”开始到换行结尾的注释，2-6表示以“/\*”开始以“\*/”结尾的注释。

同时需要对源程序中错误的token进行过滤和提示，方便下一步进行。公式2-7表示以数字开始的错误id，公式2-8表示缺少单引号的闭合符号，公式2-9表示其他错误。

(2-1)

(2-2)

(2-3)

(2-4)

(2-5)

(2-6)

(2-7)

(2-8)

(2-9)

2.2 语法文法描述

利用Bison工具根据Bison对应的规则编写自定义的文法，在进行词法分析的同时建立抽象语法树，因为自定义文法中一条文法最多包含4个语义成分，所以抽象语法树应以四叉树的形式保存。

Bison中文法的编写和自定义文法相似，不同的是需要对不符合文法的错误情况进行处理和恢复。在进行错误恢复时，主要的方法是同步到当前行尾，或者根据语法结构对出错的位置进行补充使其能够继续分析。从编写程序的顺序分析，外部只能定义变量，不能使用if等结构。函数中，括号要匹配，并且定义的多个变量之间要以逗号隔开，逗号位置不能错，也不能有多个逗号并列。一个以“｛”开始的代码块最后要以“｝”结尾。最主要的错误处理应是表达式的处理，分为两种情况，表达式内部出错则同步到表达式尾，否则同步到换行。

2.3 词法分析器设计

词法分析器通过Flex规则编写上述正则表达式以及特定保留字的正则表达式，但是由于部分正则表达式之间存在二义性，比如保留字同样也符合id的正则文法，因此需要根据Flex进行二义性处理，Flex通过匹配尽可能多的字符串和当两个模式都可以匹配时，匹配在程序中更早出现的模式，两种方式处理二义性。因此保留字的正则表达式应位于id的正则表达式之前。

2.4 语法分析器设计

和词法分析器相同，自定义的文法中对于表达式的除了并没有严格考虑二义性，因此需要利用Bison的规则对表达式不同运算符进行显示的优先级标识。这里参考百度百科C语言运算符的优先级定义[2]，主要的原则是单目运算符的优先级高于双目运算符，乘除模优先于加减。因为自增自减运算符可以位于变量的前或后，这里需要定义右结合性的token标识其优先级，在文法中通过%prec关键字标识对应得等价优先级。

为了更加灵活地进行错误处理，添加一个用于显示调用的错误处理函数myyyerror，并且设置255字节的全局字符数组myerror，当出现错误时将自生成的错误信息保存到myerror中，并调用myyerror进行处理输出。

由于需要在语法分析的过程中建立抽象表达式，因此需要定义token对应的语义值类型，文法中左部类型为抽象语法树的节点，因此定义为结点指针类型，int、float和char常量分别定义为对应的数据类型。同时由于抽象语法树中需要标明节点类型，在def.h头文件中定义枚举类型的各节点名，同时需要和对应的token作以区分，因为它们不严格属于同一类型。

2.5 词法及语法分析器实现结果

图2.1和2.2分别是对词法分析器测试的源代码和结果，分析发现，分号、错误id、不完整单字符和指数形式数均报错。

图2.3和2.4分别是对语法分析器测试的源代码和结果，分析发现，多余逗号、不恰当逗号、不合文法的表达式、缺少运算符和缺少闭合括号均报错。

图2.5和2.6分别是对抽象语法树测试的源代码和结果，分析发现，表达式、变量类型、while结构和函数调用均正常输出。

2.6 小结

本次实验主要根据任务书的提示改写mini-c的语法，进行词法分析和语法分析。实现了支持int、char和float类型，支持算术运算、比较运算、逻辑运算自增自减运算和复合赋值运算，支持if和while控制语句，支持函数声明和定义的语言TOY。

本次实验的难点在于Bison中语法分析器错误处理，Bison本身使用了error进行错误恢复，但是仅仅使用error不能完全覆盖所有错误，并且Bison本身也较为复杂，因此使用自定义处理程序，减少对于Bison特性的依赖有利于快速进行试验。另一难点在于词法分析和语法分析中对于单引号、单括号不闭合情况的处理。通过使用gcc编译器实践发现它通过一直寻找闭合单引号并且忽略后续错误的方式进行处理，因此TOY编译器中也采用这种方式，不过同时结合行尾进行同步。

本次实验中的一个较为繁琐的位置在Bison中规约规约冲突和规约移位冲突的解决。因为在文法中去掉了分号，并且合并了mini-c中的部分文法，Bison在进行分析时容易产生冲突，这里只能一步一步的根据提示修改文法避免。

因为后续实验中使用了c++的一部分内容，而Bison和Flex并不完全支持c++，对文法中的节点类型重新定义，防止其与token冲突，并且吃用extern c等方式将不兼容的部分合理处理。

尽管可以进行简单的词法分析和语法分析，但是一些细节并没有处理好，如果一个数字写作“01”，词法分析器也不会报错，因为对应的正则表达式中没有做特定限制。抽象语法树每个节点会有4个子节点，子节点数量相对较多，并且文法中只有少量文法会用到4各节点，比较浪费空间。



图2.1 词法分析源程序

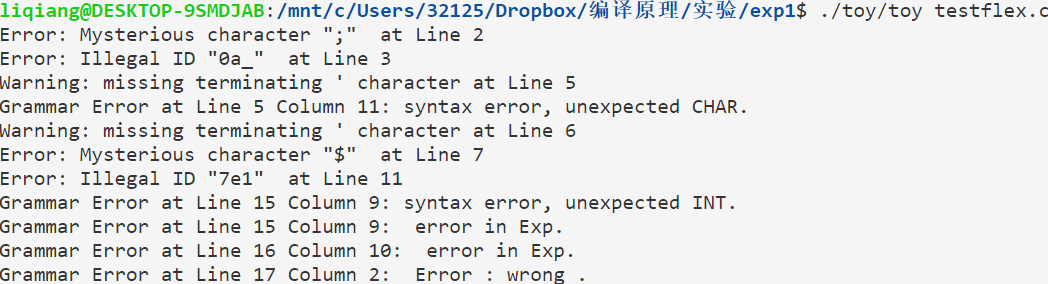


图2.2 词法分析结果

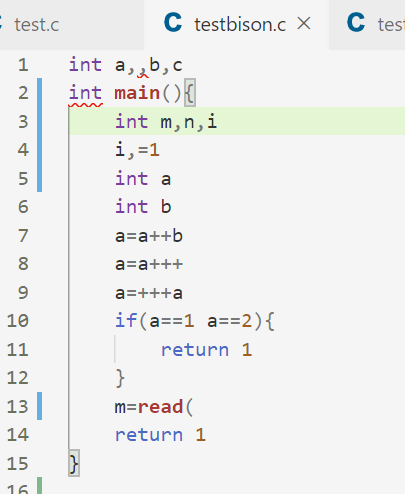


图2.3 语法分析源程序

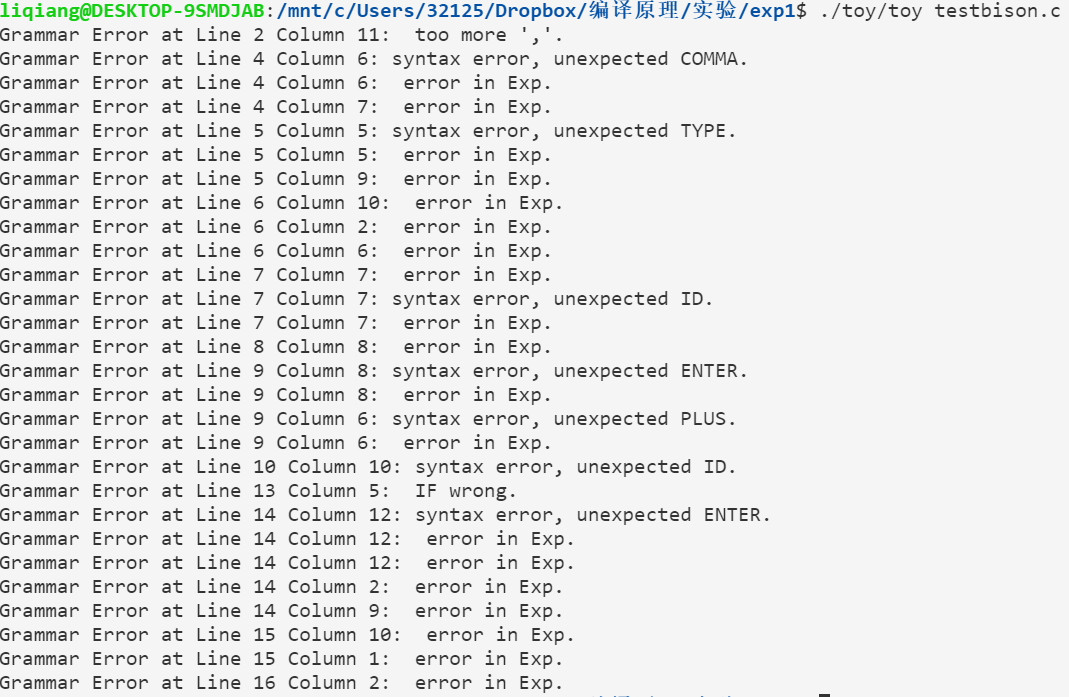


图2.4语法分析结果



图2.5 AST源程序



图2.6 AST结果

# 3 实验二 语义分析

3.1 语义表示方法描述

因为语义信息已经保存在了抽象语法树中，可以通过遍历抽象语法树对特定的文法计算语义进行静态语义分析。语义计算主要考虑继承属性和综合属性，对应的语义主要是表达式的类型，当前位置的结构等。

3.2 符号表结构

通过数组构造顺序表，表中元素为自定义结构。

对于变量和函数，名字保存在type\_id中，level表示其位于以大括号为作用域的层数，外部变量为0层，type为枚举类型的类型号，pos为行号，is\_func为函数标识，1表示函数函数声明，2表示函数定义，0表示非函数，id表示全局唯一标识符，father表示其所在位置对应的函数id，level\_father前期表示层的上一层实际无用，tempid无用。

对于作用域开始符“｛”和结束符“｝”，直接type类型分别是作用域开始OPEN\_和结束CLOSE\_，其他信息无用。

struct node {

enum node\_kind kind; //结点类型

union {

char type\_id[33]; //由标识符生成的叶结点

int type\_int; //由整常数生成的叶结点

float type\_float; //由浮点常数生成的叶结点

char type\_char;

};

struct node \*ptr[4]; //子树指针，由kind确定有多少棵子树

int level; //层号

int type; //结点对应值的类型

int pos; //语法单位所在位置行号

int is\_func;

int id; //唯一标识符

int father;

int level\_father;

int tempid;

};

3.3 错误类型定义

为方便观察报错，所有错误均以Error为开始输出，可以通过grep工具过滤出所有语义错误。

唯一性检查，当出现重复定义时输出redefination id at line形式错误。

控制流检查，当break出现在非循环位置输出token break wrong at形式错误。

上下文相关性检查，当id未声明而使用时输出id must declare before形式或func undeclare形式错误。

类型检查，当函数调用参数类型不匹配时输出type error at line形式错误。

赋值语句的左值检查，当出现非左值时输出id not other must before 形式错误。

3.4 语义分析实现技术

分析任务书中的多表方案可以发现每进入一个作用域都需要生成一个全新的符号表，在进行特定类型的分析时还需要去检测其他符号表，当作用域有多层嵌套时就会有一个作用域的树形结构图，这加大了语义分析的难度，因此本次实验中仅采用通过数组保存的顺序表。顺序表的好处在于不需要进行符号表管理，但是他需要进行作用域检查，并且顺序和逆序遍历符号表效率相对较低，对于表达式类型的综合属性需要向后计算可以保存当前节点先序遍历对应的抽象语法树进行特定的语义计算。综合来讲，顺序表对于快速实现编译器，弊大于利。

针对语义分析的程序主要保存在SemanticAnalysis.cpp中，包含displaytable、unique\_check、declare\_check、type\_check、break\_check、args\_check和semanticanalysis等几个函数。

Semanticanalysis函数为主程序，对抽象语法树进行递归遍历，并且在遍历构成中将id或作用域信息保存在全局遍历little中，每构造完一个little通过调用对应的分析函数进行分析是否有错误，通过检测后继续递归分析。

displaytable为符号表的输出函数，在主程序中每一个作用域的结束的位置也就是节点COMP\_STM末尾都会调用此函数。

unique\_check函数为变量id和函数id检测的函数。重点在于变量id与作用域的关系，逆序遍历符号表，初始化遍历close为0，当遇到作用域结尾标识也就是“｝”时close自增，当遇到开始标志时自减，因此当close为0时，符号表的内容可见，然后检查是否重复即可。

declare\_check检测函数调用时是否已经对函数定义或声明，顺序遍历符号表即可实现。

type\_check用于计算当前表达式的类型。这里同样递归计算，最终只取最大值。

break\_check原意是检测break位置，但由于只需要在while结构中通过全局变量whiles是否大于0即可。whiles表示进入while而未出while的数量。

argc\_check用于判断当前函数调用参数是否正确，这里只需要按照实际的顺序逐个比对即可。

3.5 语义分析结果

图3.1中结果一次表示重复定义，未声明，类型错误，break错误和类型冲突等语义错误。

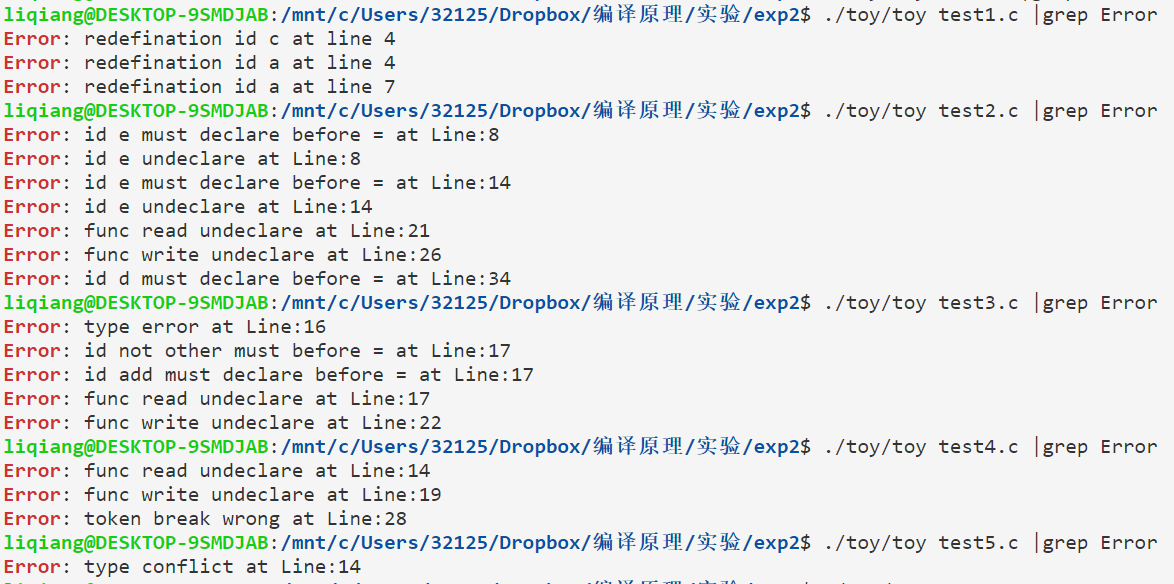


图3.1 语义分析检测

3.6小结

本次实验通过遍历建立好的抽象语法树对日常语义错误进行检测，通过语义分析后的源代码可以理解为没有词法语法和逻辑错误，可以进行后续的中间代码生成等工作。

本次实验的难点在于作用域的确认，即便是通过抽象语法树建立了符号表，但是如何判断作用域仍是难题，因此学习课本中开闭作用域的思想，通过简单判断确定变量是否可见。同样由于采用顺序表，当处理较大型程序时，效率会明显下降。

# 4 实验三 中间代码生成

4.1 中间代码格式定义

中间代码格式参照LLVM的IR。由于没有复杂的特性比如内联、重载等，不需要使用LLVM的IR的特性成分。

数据类型中float表示32位浮点数，i32表示32位整型，i8表示8位整型可用来保存字符，i1表示1位整型用来表示bool数。

函数定义采用define i32 @fibo(i32){ }形式，分别是define关键字，返回值，函数名和参数。

数据定义处采用%2= alloca i32，align 4 形式，%2表示SSA的数字，alloca表示申请空间，i32表示数据类型，align 4表示对齐4字节。

引用数据采用%3=load i32，i32\* %2，align4，load关键字表示读入。对应的store关键字表示存储。

跳转关键字使用br。它有两种形式，一种是无条件跳转br label %mylabel；另一种是根据bool值跳转br i1 %4，label %label1 ，label %label2。

比较运算通过icmp关键字实现。%6=icmp eq i32 %5，2，指令中eq表示相等，2表示常量2。

加减乘除模分别通过关键字add，sub，mul，udiv和urem实现。

函数调用通过关键字call实现。%13=call i32 @fibo(i32 %12)，%12需要事先load。

返回值通过ret实现。ret i32 %13。

4.2 中间代码生成规则定义

全局变量和函数均以@开头，局部变量以%开头。

函数内部局部变量的引用数字从0递增。

对于if结构，先计算条件，根据结果跳转到对应的label。If结构label构成为hard+数字+begin，hard+数字+end，分别表示条件成立和不成立的情况。

对于while结构，先计算条件，根据结果跳转到对应的label。While结构label构成为hard+数字+before，hard+数字+begin，hard+数字+end，分别表示while入口，循环体和出口。

对于表达式计算，引用的变量需要先装载，计算完成后，需要保存到对应变量中。

对于逻辑表达式，逻辑表达式一定出现在if或while结构条件判断处，根据label和逻辑语义进行跳转。

4.3 中间代码生成过程

理论上双向或单向链表更容易保存和修改中间代码。但是本次实验中采用不保存到特定数据结构的形式，因为引用链表就意味着需要进行内存管理，不当的内存管理在处理特定程序时可能会产生漏洞。利用递归遍历抽象语法树中天然的栈结构保存信息，同时学习语法分析中LL(1)的思想，先判断后续节点内容，然后根据特定的内容生成对应的中间代码。

前序遍历抽象语法树和语义分析相同，不同的位置主要是输出字符串的构造，需要构造特定的IR代码。仅从输出代码来讲，实现相对容易。

4.5 中间代码生成结果

图4.1左侧右侧分别是源程序和对应的IR。可以发现IR冗长。

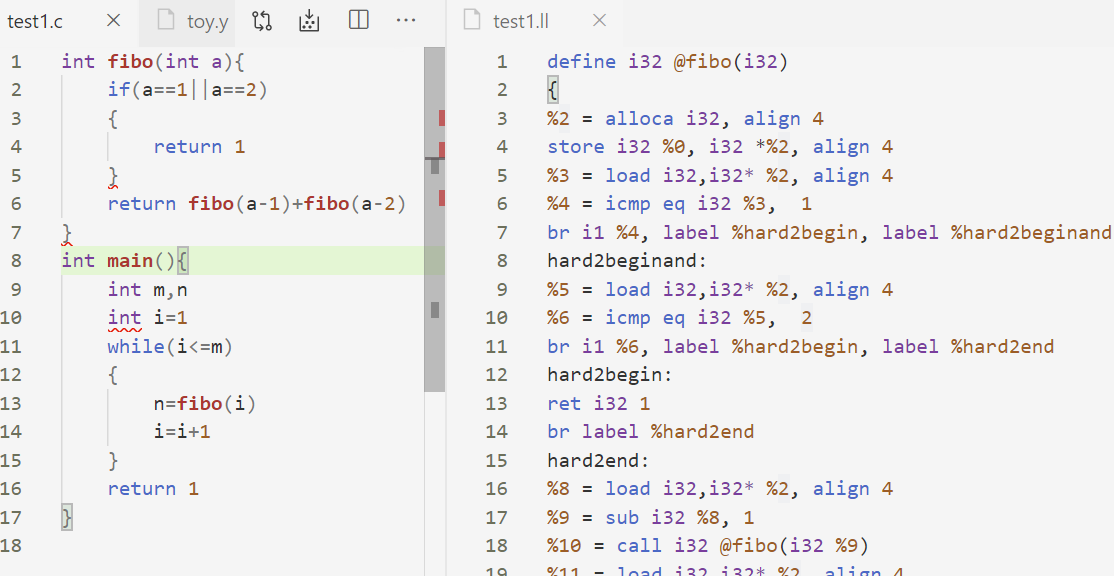


图4.1 IR测试

4.6 小结

本次实验为了快速实现编译器，对原来的文法进行了精简。自增自减和复合赋值运算因为可以通过加减乘除替换，在后面不再实现。IF\_THEN\_ELSE结构也可使用IF\_THEN代替，也不再实现。尽管中间代码生成了float的类型，但是由于输出float数字太过复杂，float类型只可以用作计算而无法输出。

本次实验一个难点在于SSA中IR的临时变量数值的引用。通过累计变量counter，并在函数结尾清零，实现临时变量的定义。并且不论是运算符还是函数调用这之前的id一定会先用load关键字加载，因此可以根据运算符的语言直接引用上一个counter对应的临时变量。

另一难点在于基本块的划分。每个函数的开始可以当做隐式的入口，隐式的label设置同样引用counter，对应if和while以及break和return关键字，其实都可以翻译到特定的基本块，通过label跳转即可实现。

另一难点在于函数参数的引用，在将参数直接作为变量进行生成时，发现lli工具会解析错误，分析正确的IR可以发现LLVM将参数先保存到新的变量中，并且参数都是以指针的形式存在，因此可以为函数中的参数优先更新一次变量符号表和SSA中的临时变量对应关系，这样就可以正常使用。这应该还函数调用传值有关。

另一难点在于任务书中不同作用域同名变量的冲突问题，这个问题通过语义分析中全局唯一变量id作以区分。

生成IR过程中最难的问题应是if结构或while结构中对逻辑表达式的翻译。因为与或表达式的不同其对应表达式计算结果的真假对应跳转的label也会不同，当表达式a||b中a成立时会忽略b，而a&&b中a不成立时会忽略b，两者嵌套则更加复杂。因此在与或节点中需要先生对应的label，hard+数字+beginand和hard+数字+endand等，然后进入子节点分析。而对于if和while节点处，需要先向后读一节点，判断是否为逻辑表达式，如果是则具体分析，如果不是，则直接计算。

本次实验最大的遗憾在于没有通过LLVM的IR的系统调用实现输入输出函数的转换，这导致生成的IR不够完美，并且后续通过putchar等才能实现输入和输出。因为过于急躁没有事先分析生成IR过程中可能会遇到的问题，导致代码冗长不够精简。

尽管生成了IR，但从逻辑上特定源程序对应的IR代码是有问题的，因为自定义的label系统中数字是引用的counter，这就导致多个if嵌套时，如果if条件中全部是常量，这些if结构对应的label可能会相同，对于特定的程序会执行错误。

# 5 实验四 目标代码生成

5.1 目标架构选择

生成的IR代码可以看做是类型JAVA中的中间代码，但是此IR更加底层，此IR同样可以通过llvm的工具直接执行，但效率相对低。

选择X86平台指令集。因为LLVM已经在其Target对应平台中实现了X86对应的指令集，寄存器以及相应的调度算法，迫于时间限制，采用LLVM的方法。使用llc工具将生成的IR代码直接转换到对应的X86架构汇编代码。最后借助clang将汇编代码转换到二进制代码即可执行。

5.2 目标代码生成结果

如图5.1左侧和右侧分别是源程序和对应的汇编代码。



图5.1 汇编代码

5.3 目标代码运行结果

如图5.2是程序test1的执行结果，它的输出是斐波那契数列的前5项。

如图5.3时程序test2的执行结果，它的输出是连续累乘的前5项。

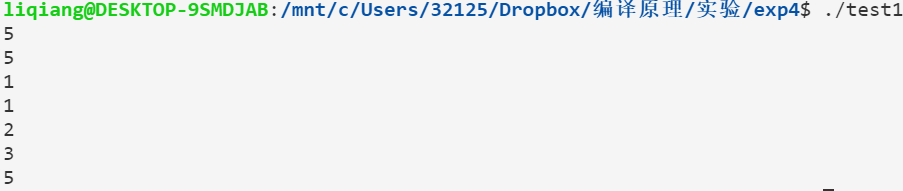


图5.2 test1测试

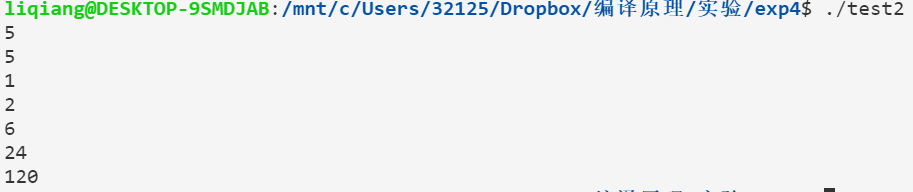


图5.2 test2测试

5.4 小结

本次实验直接采用了LLVM的X86架构生成汇编代码。实际上LLVM的IR已经很接近汇编了，如果再次编写程序生成汇编代码不会有很大的收获，因此本次实验采用的思想是将TOY整合到LLVM中，使用TOY的前端借助LLVM的后端实现整个编译器。遗憾的是自己编写特定后端需要编写InstrFormats.td、RegisterInfo.td等特定架构的寄存器描述文件，指令集描述文件，指令集调度文件等诸多文件。在相对紧张的时间条件下只能先采用LLVM已经实现好的后端。

# 6 总结与展望

6.1 全文总结

本次实验根据编译原理中文法的相关理论知识设计了TOY语言的词法分析器语法分析器和语义分析器。然后学习了LLVM的IR语法，并编写了中间代码生成程序，将TOY语言的源程序生成到了IR。最后学习了部分LLVM后端架构及实现方法，利用LLVM后端完成了整个TOY编译器。

Flex和Bison工具帮助我实现了TOY语言的词法和语法分析器，这是对课堂中文法相关内容的实践，如果没有Bison工具，我可能需要对课堂中LR分析部分做一个实践，但是重复做轮子对于实现一个项目没有很多好处，TOY语言的后端实现同样适用LLVM工具，把LLVM作为轮子辅助实现整个编译器。

由于时间太过仓促，代码中可能留有少量漏洞，在编写TOY代码时程序末尾需要多一行。生成IR代码的方式也不尽合理，最终机器码的生成更加简陋，因此TOY语言仍旧需要完善。

6.2 展望

尽管本次实验中实现了TOY语言对应的编译器，但是IR代码并没有使用TAG图作出一定的优化，后端相对依赖于LLVM的部分功能。因此如何根据IR代码做优化，需要深入学习研究。最终汇编代码和机器指令的生成同样需要磨练。同样值得思考的是为什么会有LLVM这样编译器的轮子出现，如果由我参与设计，我应当如何做。

参考文献

[1] llvm-admin team.LLVM Language Reference Manual[EB/OL].http://llvm.org/docs/LangRef.html #instruction-reference,2019-12-31.

[2] 百度百科.C语言运算符[EB/OL].https://baike.baidu.com/item/C语言运算符/1539281,2019-12-18.

[3] 陆军.flex与bison(中文版)[M].东南大学出版社:江苏南京,2011:25-212.

[4] llvm.The LLVM Compiler Infrastructure[EB/OL].https://github.com/llvm/llvm-roject/tree/master/llvm/lib/Target/X86,2019-12-31.