**小组实验作业：(三个人1组，代码如果有抄袭，分数全为0，如果任何贡献都没有，为0)**

**提交截止时间：夏学期6月6日22:00**

**用lex写出一个tiger语言或者类C或者类PASCAL某个语言的词法分析器，用YACC的分析方法完成语法分析，并生成语法树和中间代码（如果生成目标代码：可加分）。 (30分)**

一、实验目的：掌握词法分析、语法分析、语义分析和代码生成方法。

二、实验环境：

Windows系统环境或者Linux环境。

1、Linux环境下的编译和运行

（1）Linux 2.6以上版本

（2）GCC3.4以上版本

（3）Bison 2.2以上版本

（4）Flex 2.5.33以上版本

发行版可以采用Ubantu, Gentoo, Fedora Core等。

2、Windows环境下的编译和运行

（1）Visual Studio 6.0

（2）Masm 6.0以上版本

（3）ParseGenerator 4.0 （Lex和Yacc的集成开发包）

三、实验要求：

1、实验内容

编译器包含词法分析、语法分析、语义分析、代码生成、代码优化、运行环境等阶段和环节。报告中要包括词法分析部分的正规表达式描述、数据结构、原理以及实现，语法分析部分的文法描述、数据结构、原理以及实现，语义分析的方法描述，中间代码的格式、数据结构描述以及中间代码生成的实现及部分优化（目标代码的生成）。

要明确组员分工情况。

2、如果生成语法树，要求结果能用可视的方法表示出来（图或者采用数据结构中学过的方法）

3、提交压缩包内的文件要求，压缩文件名为组号，如group1.zip

包括文件：

原代码文件：包括 lex文件，yacc文件以及其他程序等等

可执行文件

实验报告：文件名为report\_组号

4、实验报告建议格式 （不限于该格式）

报告封面 （见后一页）

序言 （包括对整个编译器的描述，对所提交的各个文件的说明，组员的分工）

1. 词法分析（正规表达式，实现原理和方法）
2. 语法分析 （上下文无关文法，实现原理和方法）
3. 语义分析 （实现方法）
4. 优化考虑 （每个阶段的优化考虑）
5. 代码生成 （所有语句的代码生成的处理）
6. 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）



**本科实验报告**

|  |  |
| --- | --- |
| 课程名称： | 编译原理 |
| 姓 名： | 范源颢（3180103574）  宋天泽（3180105221）  钱隆 （3180106068） |
| 学 院： | 计算机科学与技术学院 |
| 系： | 计算机科学与技术 |
| 专 业： | 计算机科学与技术 |
| 学 号： | 3180103574、3180105221 3180106068 |
| 指导教师： | 冯雁 陈纯 |

目录

[序言 4](#_Toc73730401)

[0.1 功能描述 4](#_Toc73730402)

[0.2 文件说明 4](#_Toc73730403)

[0.3运行环境 4](#_Toc73730404)

[0.4 分工 4](#_Toc73730405)

[第一章 词法分析 4](#_Toc73730406)

[1.1 约定 4](#_Toc73730407)

[1.2 正规表达式 4](#_Toc73730408)

[1.3 lex实现原理和方法 4](#_Toc73730409)

[第二章 语法分析 4](#_Toc73730410)

[2.1 约定 5](#_Toc73730411)

[2.2 上下文无关文法 5](#_Toc73730412)

[2.3 语法树结点定义 5](#_Toc73730413)

[2.4 Yacc实现原理和方法 5](#_Toc73730414)

[2.5 简易测试 5](#_Toc73730415)

[第三章 语义分析 5](#_Toc73730416)

[3.1 符号表设计 5](#_Toc73730417)

[3.2 语义分析（生成符号表和作用域） 5](#_Toc73730418)

[3.3 类型检查 5](#_Toc73730419)

[3.4 测试 5](#_Toc73730420)

[第四章 优化考虑 5](#_Toc73730421)

[4.1 词法设计优化 6](#_Toc73730422)

[4.2 语法设计优化 6](#_Toc73730423)

[4.3 llvm优化 6](#_Toc73730424)

[第五章 代码生成 6](#_Toc73730425)

[5.1 llvm模块和函数 6](#_Toc73730426)

[5.2 环境维护 6](#_Toc73730427)

[5.3 llvm 变量生成 6](#_Toc73730428)

[5.4 llvm中间代码 6](#_Toc73730429)

[5.5 llvm 目标代码 6](#_Toc73730430)

[第六章 测试案例 6](#_Toc73730431)

[6.1 构建工程 7](#_Toc73730432)

[6.2 编译器运行测试 7](#_Toc73730433)

[6.3 各语句测试 7](#_Toc73730434)

[6.4 复合语句测试 7](#_Toc73730435)

# 序言

（包括对整个编译器的描述，对所提交的各个文件的说明，组员的分工）

## 功能描述

本报告是编译原理期末大作业小组实验的实验报告，此次编译原理期末大作业的目标是实现一个编译器，可使用的工具包括lex、Yacc、llvm等等，具体的实现形式可以是一个类C、类Pascal或其他程序语言的词法、语法分析器，需要生成语法树和中间代码，语法树需要可视化展示，但具体展示形式不限，中间代码的形式并无规定，从中间代码到目标机器的目标代码的转化为选做模块。

这次编译原理实验，我们完成了一个使用C++语言，利用lex、Yacc、llvm工具的类C（mytinyC）语言的编译器，这种类C语言在文法上基本参照C语言文法，在功能上去除了指针、结构体、联合体等C语言的特性，此外对于函数和语句的定义有比起C语言更加严格的限制，对于这种类C语言，我们设计的编译器可以有效的实现词法分析、语法分析、语义分析、代码生成以及代码优化，最终可以展示语法树、llvm中间代码和目标机器的目标代码以及可执行文件。

## 文件说明

ASTNodes.cpp 实验中编写，用于实现构建和展示语法树功能的C++语言代码文件；

ASTNodes.hpp 实验中编写，用于实现构建和展示语法树功能的C++语言头文件；

compile.sh 实验中编写，用于在Ubuntu 16.04+系统中运行完整运行编译器的批处理脚本文件；

lexer.l  实验中编写，用于对类C语言进行词法分析的lex文件

parser.y 实验中编写，用于对类C语言进行语法分析的yacc文件

lex.yy.c flex根据lexer.l生成的，可用于实现词法分析的C源文件

parser.tab.c bison根据parser.y生成的，可用于实现语法分析的C源文件

parser.tab.h bison根据parser.y生成的，可用于实现语法分析的C头文件

our\_syntax.md 实验中编写，用于提示我们设计的类C语言语法规则的Markdown文件，不参与编译器工程的构建。

c\_minus\_syntax.md 上交时删除

other file may also be included, and the file-organizing may be modified before handing-in.

## 0.3运行环境

使用cmake构建，依赖包括：

* Linux 16.04+
* gcc5.4.0+， g++ 5.4.0+
* flex 2.6.0+
* bison 2.6.0+, (推荐3.7.6+)
* llvm 3.8.0+
* clang 3.8.0+

(其中‘+’表示当前版本及以上)

## 0.4 分工

范源颢：词法分析 语法分析

宋天泽：语法分析（语法树构建） 语义分析

钱隆：中间代码和目标代生成

# 词法分析

## 1.1 约定

我们实现的类C语言mytinyC对于C语言的语法做了一些简化，因此我们有必要约定简化的内容，如下所述：

### 关键字

mytinyC关键字如下表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| int | float | char | double | bool |
| void | if | else | for | while |
| return | extern |  |  |  |

其中bool用于声明mytinyC中的布尔类型的变量，bool类型仅包含true和false两种，分别可以被声明为true/True /TRUE和false/False/FALSE，运算规则和C++一致。其余的关键字和标准C语言中的关键字意义相同。

于是，我们的mytinyC支持的基本类型包括：整形int、浮点数float、字符char、布尔类型bool、双精度浮点数double。在函数的返回类型方面，我们支持虚返回void。我们的mytinyC语法支持的控制语句包括if-else分支语句， while/for循环语句，函数返回语句return。

注意比起标准C语言，我们在关键字便不再支持结构体（struct）、联合体（union）； unsigned、signed等复杂类型不再支持，我们不支持使用typedef自定义类型，也不支持使用const、static约束变量和函数的作用域。在流程控制方面，我们没有实现switch case 语法，也不支持使用break关键字跳出循环。在底层构建方面，我们也不支持使用sizeof语法查看数据类型的长度。

我们保留了extern关键字，这是为了在使用llvm的过程中，利用extern可以直接llvm中实现的函数，例如，假如在mytinyC文件中按照如下语句声明了输出函数printf：

extern int printf(char[] str)

我们就可以在测试在mytinyC文件中调用printf进行输出。

### 运算符

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| = | += | -= | \*= | /= |
| %= | |= | &= | ^= | >>= |
| <<= | || | && | | | & |
| ^(位抑或) | > | < | >= | <= |
| == | != | << | >> | + |
| - | \* | / | % | ! |
| ~（位取反） | ++（仅后缀） | --（仅后缀） | ( | ) |
| { | } | [ | ] | , |
| ; | ‘’ |  |  |  |

基本上涵盖了所有的C语言操作符。

mytinyC约定++和—只有作为后缀表达式出现，同时，我们没有指针运算，因此我们将\*仅理解为乘法，&理解为位与。

### 标识符和字面量

我们对于标识符的约定和C语言基本相同，用户自定义标识符需满足：字母（大小写均可）或者下划线开头，之后包含若干字母、数字和下划线的组合，推荐总长度在20字符以下。mytinyC不得将true/True/TRUE/false/False/False作为标识符。

我们对于字面量的约定和C语言也基本相同，开头或有字符-（负号），之后仅有数字组成的字符串我们认为是整形数字；开头或有负号，之后含有若干数字，在此之后包含小数点.，小数点之后若有字符则全为数字的字符串我们认为是浮点数（注意若小数点之后没有字符，也认为是浮点数，比如4.0和4.均为浮点数，而4为整形数）；‘’中包含的单个字符是C语言的char类型；三种不同大小写风格的true和三种不同大小写风格的false是布尔类型的字面量。

### 注释

我们实现了C语言风格的注释，包括//开头的单行注释，以及/\*和\*/之间的多行注释。

## 1.2 正规表达式

对于C语言关键字、操作符以及字面量false、true，我们都可以使用穷举的方式列出他们的正规表达式，例如，true的正规表达式可以写成：

true|True|TRUE

标识符和整形数、浮点数的正规表达式亦不难写出，我们给出整形、浮点型和标识符的正规表达式，如下：

int  [\-]?[0-9]+

float [\-]?[0-9]+\.[0-9]\*

identifier [a-zA-Z\_][a-zA-Z0-9\_]\*

注意，我们没有使用正规表达式的思路进行注释的词法分析，因为我们知道多行注释的正规表达式将比较复杂，为此，我们将使用下文中提到的lex的另一特性实现C语言的注释。

## 1.3 lex实现原理和方法

lex主要是通过有限状态机的思路实现词法分析的，我们不在赘述有限状态机的知识，而将注意力集中在如何通过书写lexer.l实现代码的分析。

在.l文件的第一部分（第一个%%之前）我们首先通过正规表达式定义一些语词：

 /\*空白符\*/

macro  #.\*

whitespace [ \t\r\n]+

linecomment \/\/.\*\n

 /\*字面量\*/

int  [\-]?[0-9]+

float [\-]?[0-9]+\.[0-9]\*

identifier [a-zA-Z\_][a-zA-Z0-9\_]\*

char  \'.\'

true  true|True|TRUE

false false|False|FALSE

之后，我们在第二部分定义lex在遇到这些语词的过程中需要执行的操作，若分析的结论是关键字和运算符，直接返回token即可：

"if"        return IF;

"else"      return ELSE;

"return"    return RETURN;

"for"       return FOR;

"while"     return WHILE;

"||"    return LOR;

"&&"    return LAND;

若是类型名称，我们需要保存内容，并且通过token表示其类型，这样将有助于日后的类型检查。

"int"  {

    strcpy(yylval.type\_id, yytext);

    return INT\_TYPE;

}

若是字面量，则我们需要保存字面量的值，并且通过token表示其类型。

{int} {

    yylval.type\_int = atoi(yytext);

    return INT;

}

{float} {

    yylval.type\_float = atof(yytext);

    return FLOAT;

}

{true} {

    yylval.type\_bool = 1;

    return TRUE;

}

{identifier} {

    strcpy(yylval.type\_id, yytext);

    return ID;

}

最后我们讲述注释的处理，如上所述，一旦匹配到//，则此行的内容均视为注释。（其后若出现/\*，标准C也不认为是多行注释的开始，下一行除非再有注释符号，否则即使上一行有未闭合的/\*，下一行的语句不会被编译器作为注释跳过）因此，我们可以直接用正则表达式匹配单行注释。

多行注释我们直接利用lex作为有限自动机的机制进行实现，lex在.l文件的第一部分支持直接定义状态，我们可以使用语句：

%x C\_COMMENT

定义一个名为C\_COMMENT的状态，

在.l文件的第二部分，我们定义一旦遇到字符/\*，使用lex定义的宏BEGIN进入C\_COMMENT状态：

"/\*" BEGIN(C\_COMMENT);

处于C\_COMMENT状态中时，无论遇到何种字符均进行空操作，

<C\_COMMENT>.  {}

注意lex使用尖括号标识遇到字符串时有限状态机的状态，如无标识，则推定此时是有限状态机处于lex内置的初始状态INITIAL中。

最后，如果处于C\_COMMENT状态中遇到\*/，则有限状态机应当返回初始状态INITIAL：

<C\_COMMENT>"\*/" BEGIN(INITIAL);

注意书写时，<C\_COMMENT>"\*/"语句应该先于<C\_COMMENT>.，因为lex优先匹配文件中先行的规则，如若带有.的规则先行，则之后的规则便失去意义，因为之后规则中的字符都会被.优先匹配掉。

# 语法分析

（上下文无关文法， 实现的原理和方法）

## 2.1 约定

我们在设计mytinyC的语法的时候亦有一些语法上的限制：

1. 我们在大括号中语句需要严格满足所有局部变量的定义先于其他语句；我们不支持for语句的小括号内第一个分句直接定义域内变量

2. 我们支持多维数组和变量在定义时的初始化。

3. 同类型变量的声明不支持使用逗号分隔，均需要使用分号指明类型和名称，例如：

int a,b,c=1;和int a,b,c;均视为不合法，需要修改为：int a; int b; int c;。

4. 函数调用数组作为参数时，需要指明数组维数和长度，例如，

5.待补充

## 2.2 上下文无关文法

粘贴语法＋讲解注意解释shift-reduce conflict

## 2.3 语法树结点定义

我们使用llvm

粘贴相关代码/.drawio贴图

## 2.4 Yacc实现原理和方法

粘贴相关代码（设计例子即可）

## 2.5 测试

### 简易测试和冲突处理

粘贴感觉加上bison parser.y之后的贴图就可以了。

注意一个粗暴的处理冲突的方法，将LALR算法改为LR的方法

## 

### 打印语法树的效果

粘贴结果

# 语义分析

（实现方法）

## 3.1 符号表设计

### 数据结构设计

### 符号表操作

如何访问数据结构。

### 作用域

## 3.2 语义分析（生成符号表和作用域）

似乎可以通过一个遍历方法把符号表得出

## 3.3 类型检查

计算属性。

## 3.4 测试

# 优化考虑

（每个阶段的优化考虑）

## 4.1 词法设计优化

对注释的处理

## 4.2 语法设计优化

1.使用%left来实现结合性，而不是写死在语法规则里面

2. 使用左递归

## 4.3 llvm优化

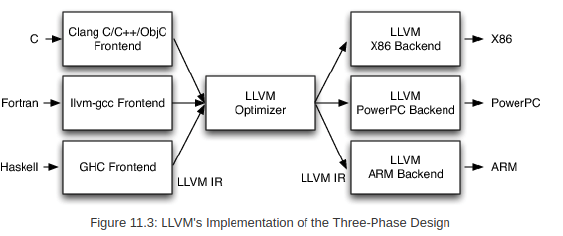
llvm自己的常量运算优化；

llvm中的控制流优化。

# 代码生成

（所有语句的代码生成的处理）

## 5.1 llvm模块和函数



调用<llvm/IR/……>中的文件内容，我们可以使用llvm现成的模块，有利于中间代码的生成。

### 5.1.1 Module

### 5.1.2 Function

### 5.1.3 BasicBlock

### 5.1.4 IRBuilder

### 5.1.5 Value

### 5.1.6 Name and Identity

## 5.2 环境维护

### 5.2.1代码生成的环境种类

### 5.2.2标签信息的维护

### 5.2.3 ID信息的维护

### 5.2.4函数信息的维护

### 5.2.5控制流信息的维护

## 5.3 llvm 变量生成

### 5.3.1 类型生成

### 5.3.2 类型转换？

### 5.3.3常量信息

## 5.4 llvm中间代码生成

### 5.4.1表达式生成

常规的和短路的

### 5.4.2语句生成

#### 5.4.2.1 变量声明语句

#### 5.4.2.2控制流语句（if/for/while）

### 5.4.3 函数生成

#### 5.4.3.1 用户定义函数

#### 5.4.3.2 系统函数

### 5.4.4 效果测试：

表达式+变量

控制流

函数定义和调用；

## 5.5 llvm 目标代码

### 5.5.1 .asm结构

命令

### 5.5.2 .obj结构

命令

### 5.5.3 .out, Linux可执行文件

命令

# 测试案例

（每个语句成分的测试案例， 至少两个复合语句组合后的测试案例）

## 6.1 构建工程

shell+makefile脚本

## 6.2 编译器运行测试

## 6.3 各语句测试

表达式+变量

控制流

函数定义和调用；

## 6.4 复合语句测试

表达式+变量

控制流

函数定义和调用；

22组合