南開大學
Nankai University

软件学院

《编译原理》期末实验报告

实现简单的 C 语言编译器

小组成员 ：杨万里 郁万祥 钟习明 沈永腾

专业：软件工程

2022 年 12 月 24 日

目录

1 实验目标

实现一个简单的 C 语言编译器，可以将 C 语言测试程序编译为目标代码——汇编程序，利用汇编器转换为二进制程序后运行无误。

1.1 基本要求

- 数据类型：int
- 语句：注释，声明，赋值，循环（while 和 for），判断（if），输入输出（printf）
- 算术运算：+，-，*，/，%
- 关系运算：==，>，<，>=，<=，!=
- 逻辑运算：&&（与）、||（或）、!（非）

1.2 完成功能

- 词法分析
- 语法分析
- 类型检查
- 代码优化
- 错误分析
- 汇编程序

1.3 实现的加分项

- 支持函数调用
- 支持一维数组与多维数组
- 支持一维指针

2 实验环境及工具

本实验在 Linux 的 Ubuntu 系统上，借助 Flex 和 Yacc 工具实现编译器。同时需要使用 GCC 编译工具将编译器生成的汇编代码转换为可执行程序。由于我们实现的汇编代码是 32bit AT&T 格式，因此在 64 位的设备上需要借助 Qemu 执行程序。

3 整体工作流程介绍

本次实验设计的编译器在词法分析、语法分析、语义分析、中间代码生成、代码优化与代码生成等步骤后，将高级语言（C 语言）翻译为汇编代码。

其中，词法分析阶段借助 Flex 工具，设计各个词法单元的正则表达式之后，转换出相应的识别程序。高级语言程序当中的字符流经过词法分析器之后，输出 token 序列，并且将相关的信息存储到符号表当中。

语法分析阶段借助 Yacc 工具，设计各个“分语言”的产生式，转换出识别程序，能够做到将输入的 token 序列，识别构建出相应的语法树。该步骤可以借助 AST 树分析语法结构，检查类型错误等语法错误。

生成中间代码之后，可以借助 CFG 图等结构优化程序流程，进行代码优化。

最后利用 GCC 的汇编器将汇编代码转化为对应平台的机器指令，形成可执行文件。

4 实验工具介绍

4.1 Flex

Flex（快速词法分析器生成器）是 lex 的免费开源软件替代品。它是生成词法分析器（也称为“扫描器”）的计算机程序。词法分析器是识别文本中的词汇模式的程序。该 flex 程序读取给定的输入文件，或它的标准输入，如果没有指定文件名，为词法分析器产生的描述。描述采用成对的正则表达式和 C 代码（称为 rules）形式。flex 生成一个 C 源文件作为输出，文字库默认情况下，它定义了一个例程 yylex()。可以编译该文件并将其与 Flex 运行时库链接以生成可执行文件。运行可执行文件时，它将分析其输入以查找正则表达式的出现。只要找到一个，它就会执行相应的 C 代码。经常连同 Yacc 或 GNU Bison 一起使用。

Flex 程序通常由三部分组成：

- 定义部分
- %%
- 规则部分
- %%
- 用户附加的 C 语言部分

4.2 Yacc

yacc 为利用计算机程序输入的结构提供了一个通用的工具。yacc 使用者准备了一种输入处理的规范，包括描述输入结构的规则，这些规则被识别调用的代码，一个低级别程序来执行基本的输入。yacc 接着会生成一个函数来控制输入过程。这个函数，被称为一个解析器，调用用户提供的低级别的输入程序（词法分析器）从输入流挑选出基本项（被称为 tokens）。这些 tokens 根据输入结构规则来组成，称为语法规则；当其中一个规则被识别，然后用户代码提供了这种规则，会执行相应的动作；这些动作拥有返回结果和使用其他动作的值的能力。

5 实现词法分析器

5.1 概述

词法分析器的作用是读取源程序生成词法单元，并过滤掉注释和空白，本项目的词法分析主要用到了 Flex。Flex 是一个生成词法分析器的工具，它可以利用正则表达式来生成匹配相应字符串的 C 语言代码，其语法格式基本同 Lex 相同。

单词的描述称为模式 (Lexical Pattern)，模式一般用正规表达式进行精确描述。Flex 通过读取一个有规定格式的文本文件，输出一个 C 语言源程序。

5.2 设计思想

5.2.1 单词类型及其正规式

- 基本符

单词的值	单词类型	正规式 r
begin	beginsym	begin
call	callsym	call
const	constsym	const
do	dosym	do
end	endsym	end
if	ifsym	if
odd	oddsym	odd
procedure	proceduresym	procedure
read	readsym	read
then	thensym	then
var	varsym	var
while	whilesym	while
write	writesym	write

- 标识符

单词的值	单词类型	正规式 r
标识符	ident	[_a-zA-Z][_a-zA-Z0-9]*

- 常数

单词的值	单词类型	正规式 r
常数	number	0|[1-9][0-9]*

- 界符

单词的值	单词类型	正规式 r
(	lparen	(
)	rparen	)
,	comma	,
;	semicolon	;
.	period	.

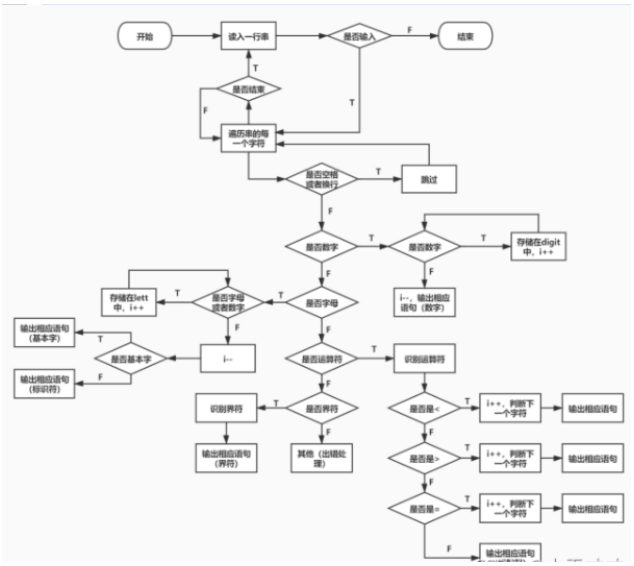
- 运算符

单词的值	单词类型	正规式 r
+	plus	+
-	minus	-
*	mul	*
/	div	/
%	mod	%
=	assign	=
<	les	<
<=	leseq	<=
>	gra	>
>=	graeq	>=
!=	neq	!=
==	eq	==
+=	plusassign	+=
-=	minusassign	-=
*=	mulassign	*=
/=	divassign	/=
&&	and	&&
||	or	||
!	not	!

5.3 实现流程

词法分析程序的输入是源程序，输出是一个个单词符号的二元组，它的任务是从左至右逐个字符地对源程序进行扫描，产生一个个的单词符号，把作为字符串的源程序改造成为单词符号串的中间程序。

首先逐行扫描源程序，然后遍历串的每一个字符，判断字符是不是空格、数学、字母、运算符或者界符，再进行下一步的判断，如果不符合这几种情况，将会归入出错处理。背后的算法原理如下图所示。



5.4 实现细节

5.4.1 声明起始状态

在定义部分，我们可以声明一些起始状态，用来限制特定规则的作用范围。用它可以很方便地做一些事情，我们用识别注释段作为一个例子，因为在注释段中，同样会包含数字字母标识符等等元素，但我们不应将其作为正常的元素来识别，这时候通过声明额外的起始状态以及规则会很有帮助。

在这之中，声明部分的%x 声明了一个新的起始状态，而在之后的规则使用中加入 < 状态名 > 的表明该规则只在当前状态下生效。而状态的切换可以看出通过在之后附加的语法块中通过定义好的宏 BEGIN 来切换，注意初始状态默认为 INITIAL，因此在结束该状态时我们实际写的是切换回初始状态。还有额外的一点说明%x 声明的为独占的起始状态，当处在该状态时只有规则表明为该状态的才会生效，而%s 可以声明共享的起始状态，当处在共享的起始状态时，没有任何状态修饰的规则也会生效。

5.4.2 行号使用

如果你有需要了解当前处理到文件的第几行，通过添加%option yylineno，Flex 会定义全局变量 yylineno 来记录行号，遇到换行符后自动更新，但要注意 Flex 并不会帮你做初始化，需要自行初始化。

5.4.3 符号表

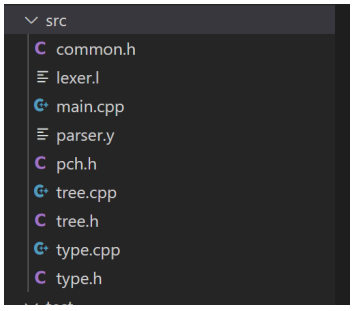
对于标识符 (ID)，相同的标识符可能在相同作用域而指向相同的内存，也可能因为重新声明或在不同作用域而指向不同内存。我们希望词法程序可以对这些情况做区分。

我们定义的编译器中一定是会有一些关键字的，我们可以对每个关键字进行声明，在规则中单独找出它们，另一种思路是将所有的关键字都视作普通的符号写入符号表，通过在符号表中提前定义好相关的关键字，可以减少定义与声明的内容。

6 实现语法分析器

6.1 代码结构

运行编译命令 make 之前的代码结构如下图所示：



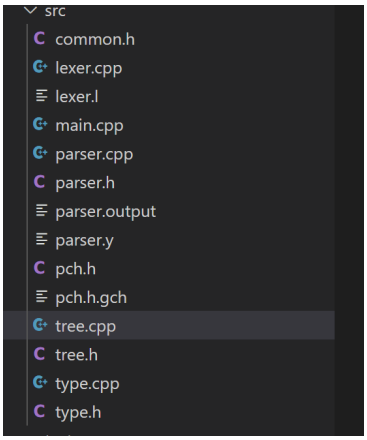
其中：

parser.y 为借助 yacc 工具进行编写的语法分析器。

tree.h 是对抽象语法树结构的定义。

tree.cpp 是对抽象语法树具体的实现以及包括各种功能函数的实现。

运行编译命令 make 以后的代码结构如下图所示：



借助于 yacc 工具，提前编写好的 paser.y 文件自动编译生成用于语法分析以及输出语法分析树的工具：parser.y 和 parser.output。

6.2 实现原理

6.2.1 Yacc 的深入介绍

分析程序生成器 (parser generator) 是一个指定某个格式中的一种语言的语法作为它的输入，并为该种语言产生分析过程以作为它的输出的程序。合并 LALR(1) 分析算法是一种常用的分析生成器，它被称作 Yacc(yet another compiler–compiler)。

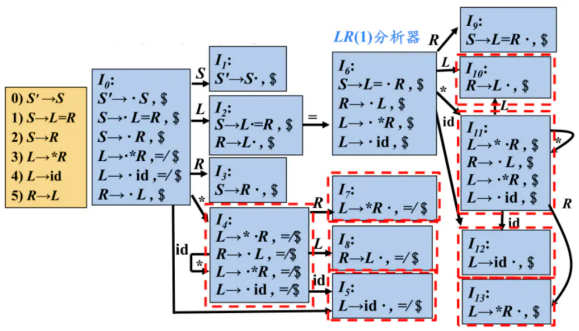
作为 Yacc 对说明文件中的%token NUMBER 声明的对应。Yacc 坚持定义所有的符号记号本身，而不是从别的地方引入一个定义。但是却有可能通过在记号声明中的记号名之后书写一个值来指定将赋给记号的数字值。yacc 的输入是巴科斯范式 (BNF) 表达的语法规测以及语法规约的处理代码，Ycc 输出的是基于表驱动的编译器，包，含输入的语法规约的处理代码部分。

ycc 是开发编译器的一个有用的工具，采用 LALR(1) 语法分析方法。由于所产生的解析器需要词法分析器配合，因此 Yacc 经常和词法分析器的产生器，一般就是 Lex 一联合使用。

6.2.2 LALR（1）语法分析方法

（1）LALR 分析法的提出：

LR(1) 分析法实际上是根据展望符集合的不同将原始的 LR(0) 项目进行分裂，分裂成不同的 LR(1) 项目。这就使得 LR(1) 的状态数较 LR(0) 的状态数多了很多。比如说 C 语言的语法在构造其 LR(0) 分析表的时候通常只有几百个状态，而构造其 LR(1) 语法的时候会有几千个状态。为了使 LR(1) 实用化，必须要想办法去减少其状态数。



如上图有 4 对同心项目集，I4 与 I11,I7 与 I13，I8 与 I10,I5 与 I12，把没有状态冲突的项目进行合并可以大大减少自动机的状态数，空间上会更节省。这就是 LALR 分析法的基本思想。

（2）LALR(lookahead-LR) 分析的基本思想：

- 寻找具有相同核心的 LR(1) 项集，并将这些项集合并为一个项集。所谓项集的核心就是其第一分量的集合。
- 然后根据合并后得到的项集族构造语法分析表。
- 如果分析表中没有语法分析动作冲突，给定的文法就称为 LALR(1) 文法，就可以根据该分析表进行语法分析。

（3）LALR（1）的特点：

- 形式上与 LR（1）相同
- 大小上与 LR（0）/SLR 相同
- 分析能力介于 SLR 和 LR（1）二者之间

6.2.3 抽象语法树 AST

抽象语法树 (abstract syntax tree,AST) 是源代码的抽象语法结构的树状表示，树上的每个节点都表示源代码中的一种结构，这所以说是抽象的，是因为抽象语法树并不会表示出真实语法出现的每一个细节，比如说，嵌套括号被隐含在树的结构中，并没有以节点的形式呈现。抽象语法树并不依赖于源语言的语法也就是说语法分析阶段所采用的上下文无文文法，因为在写文法时，经常会对文法进行等价的转换（消除左递归，回溯，二义性等），这样会给文法分析引入一些多余的成分，对后续阶段造成不利影响，甚至会使合个阶段变得混乱。因些，很多编译器经常要独立地构造语法分析树，为前端，后端建立一个清晰的接口。

6.3 实现细节

6.3.1 数据结构：树节点

我们唯一需要面对的数据结构就是语法树。需要注意的是，我们随着语法分析的进行，构建的是一棵抽象语法树（AST），因为我们并不需要保留产生式推导过程中的每一层，这对应着具体语法树 (CST)。我们要设计的就是语法树的结点。结点分为许多类，除了一些共用属性外，不同类结点有着各自的属性、各自的子树结构、各自的函数实现。我们可以简单用 struct 去涵盖所有需要的内容，也可以设计复杂的继承结构。直观上，结点的类型被分为词法分析得到的叶节点、表达式、基本语句（不嵌套地有子语句的语句）、复合语句（比如 for,if-e1se,], 函数），以及象征完成程序的根节点。

```
#ifndef TREE_H
#define TREE_H

#include "pch.h"
#include "type.h"

enum NodeType
{
```

```
    NODE_OP,
    NODE_BOOL,
    NODE_CONST,
    NODE_VAR,
    NODE_TYPE,

    NODE_PROG,
    NODE_STMT,
    NODE_EXPR,
    NODE_VARLIST,
    NODE_PARAM,
};

enum OperatorType
{
    OP_EQ,       // ==
    OP_NEQ,      // !=
    OP_GRAEQ,    // >=
    OP_LESEQ,    // <=
    OP_ADDASSIGN,  // +=
    OP_SUBASSIGN,  // -=
            // ......
}

// ......
```

6.3.2 算法：词法分析，语法分析与树的构建

词法分析得到的，实质是语法树的叶子结点，语法树所有内部节点均由语法分析创建。在自底向上构建语法树时（与预测分析法相对），我们使用孩子结点构造父节点。在 yacc 每次确定一个产生式发生 reduce 时，我们会 new 出父节点、根据子结点正确设置父节点的属性、记录继承关系。每个结点的孩子数量是不统一的，我们设计一个 sibling 链表，或者一个宽裕的子节点数组，记录继承关系。

```
void TreeNode::addSibling(TreeNode* sibling){
    TreeNode* p = this;
    while (p->sibling != nullptr) {
        p = p->sibling;
    }
    p->sibling = sibling;
}
```

6.3.3 lex 与 yacc 的连结，个人项目文件的引入

lex 与 yacc 的连结依靠某些全局变量，如 yytext 等。此外，此次实验我们希望将自己的数据结构定义放在独立的.c/h 文件之中，供 lex 与 yacc 使用，因此需要正确的区分头文件与源文件中代码的内容，避免编译时出现 link error。

7 类型检查

主要作用是检查语法是否符合语法规则，编译器提示的错误信息多在此处产生。

7.1 实现原理

类型检查是编译过程的重要一步，以确保操作对象与操作符相匹配。每一个表达式都有相关联的类型，如比较运算表达式的类型为布尔型，而计算表达式的类型一般为整型或浮点型等。类型检查的目的在于找出源代码中不符合类型表达式规定的代码，在最终的代码生成之前报错，使得程序员根据错误信息对源代码进行修正。

7.2 实现步骤

首先要规定一些基本类型，包括 int，char，bool，void。

类型检查在建立语法树的过程中完成（tree.cpp 中的 typecheck()），我们需要为语法树结点增加一项以表示该节点代表的表达式类型。类型检查也是自底向上的过程，父节点需要检查孩子结点的类型，并根据孩子结点类型确定自身类型。有一些表达式可以在语法制导翻译时就确定类型，比如整数就是整型，这些表达式通常是语法树的叶结点。而有些表达式则依赖其子表达式确定类型，这些表达式则是语法树中的内部节点。举例而言，对于加法运算，其类型是整形还是浮点性取决于其子表达式，而如果其子表达式类型为字符型，则发生了类型错误。

```
void TreeNode::typeCheck() {
    //具体内容非常多，不全部展示
}
```

typecheck 函数：重新遍历语法树，进行类型检查和检查诸如 break，continue 是否处于循坏提内部等语法分析难以检查的语法错误。以 while 为例：首先记录循环层数，为 continue 和 break 提供循环外错误检查

```
if (nodeType == NODE_STMT && (stype == STMT_FOR || stype == STMT_WHILE)) {
    cycleStackTop++;
}
```

先遍历子节点进行 type 计算

```
TreeNode *p = this->child;
while (p != nullptr) {
    p->typeCheck();
    p = p->sibling;
}
```

分情况检查类型错误并对部分情况进行强制类型转换

```
switch (this->nodeType)
{
    case NODE_FUNCALL:
    //......
}
```

8 生成汇编程序

将语法树通过遍历生成目标代码，目标代码要求为汇编语言程序。

8.1 实现原理及步骤

8.1.1 label 生成

我们知道在遇到分支条件语句的时候，程序会根据条件语句的真假执行不同的语句，在汇编语言中则会根据条件得值进行跳转，跳转到不同的标号所在的语句。但是生成目标代码的程序是如何知道应该跳转到的 label 的值多少呢？这就需要我们在生成汇编代码之前遍历语法树，提前给相应的结点生成编号。

```
void TreeNode::get_label() {
    string temp;
    switch (nodeType)
    {
        //......
    }
}
```

在 get_label 函数中，可以通过判断节点的不同类型（包括 stmt 和 op）进行不同的处理。

```
switch (stype)
{
    case STMT_FUNCDECL:
        this->label.begin_label = this->child->sibling->var_name;
        // next为return和局部变量清理
        this->label.next_label = ".LRET_" + this->child->sibling->var_name;
        break;
    case STMT_IF:
        this->label.begin_label = new_label();
        this->label.true_label = new_label();
        this->label.false_label = this->label.next_label = new_label();
        this->child->label.true_label = this->label.true_label;
        this->child->label.false_label = this->label.false_label;
        break;
    //......
}
```

8.1.2 汇编代码生成

在生成完 label 后我们可以进行汇编代码的生成，汇编代码的生成也是一个对于语法树的遍历的过程，根据结点类型进行相应的输出。

以 while 循环为例：

```
case STMT_WHILE:
    get_label();
    cycleStack[++cycleStackTop] = this;
```

```
    cout << label.next_label << ":" << endl;
    this->child->genCode();
    cout << label.true_label << ":" << endl;
    this->child->sibling->genCode();
    cout << "\tjmp\t\t" << label.next_label << endl;
    cout << label.false_label << ":" << endl;
    cycleStackTop--;
    break;
```

当遇到 while 语句的时候，我们首先输出该语句的 lable 标号，之后递归遍历该节点的两个孩子（循环条件与循环体），最后跳转回循环体开始的 label。

9 实验结果展示

经过上述的努力，我们最终实现了一个简单的 C 语言编译器，接下来开始检验实验结果。以下的测试样例由助教提供。

9.1 测试样例源程序

9.1.1 基本功能的检验

基本功能的检验有两个样例 case1.c 和 case2.c 需要编译。

```c
//#include<stdio.h>

int main() {
    int a = 50;
    int sum = -1;
    // outer loop
    for (int i = 0; i < a * 2; i++) {
        // inner loop
        for (int j = 0; j < a ; j++) {
            if (j < a / 2 && j % 2 != 0 ) {
                sum = sum +  i * j / 2 + i * j;
            } else {
                while (sum > j || !sum != 0) {
                    sum = sum - 3 / 2;
                }
            }
        }
    }

    // print the value of `sum`
    printf("%d\n", sum);

    return 0;
}
```

图 9.1: case1.c

```c
//#include<stdio.h>

int main() {
    int a = 1, b = 1;
    int c;

    // Fibonacci
    for (int i = 2; i < 20; i++) {
        c = a;
        a = a + b;
        b = c;
    }

    printf("%d %d\n", a, b);
    return 0;
}
```

图 9.2: case2.c

9.1.2 函数调用的检查

```c
//#include<stdio.h>

int fib(int a, int b) {
    if (a > 1000) {
        return a * b % 5 - -1;
    }
    return fib(a + b, a) + 1;
}

int main() {
    printf("%d\n", fib(1, 1));
    return 0;
}
```

图 9.3: function.c

9.1.3 一维与多维数组的检查

```c
//#include<stdio.h>


int main(){
  int b[4];
  b[2] = 6;
  b[1] = 2;
  int a[2][3];
  a[0][1] = 2;
  a[1][2] = 3;
  int c;
  c =  b[2];
  int d = b[1];
  c = c+d;
  printf("%d\n", c);
  c = a[0][1] + a[1][2];
  printf("%d\n", c);
  return 0;
}
```

图 9.4: array1D.C

9.1.4 一维指针的检查

```c
//#include<stdio.h>

int main()
{
    int a = 2;
    int b = 3;
    printf("%d\n", a);
    printf("%d\n", b);
    int* pa = &a;
    int* pb = &b;
    int t = *pb;
    pb = pa;
    t = t**pb**pa / *pb;
    printf("%d\n", *pa);
    printf("%d\n", *pb);
    printf("%d\n", t);
    return 0;
}
```

图 9.5: pointer1D.c

9.2 测试样例汇编程序

通过指令 make asm，会把 test 文件夹下所有的测试样例均编译为相应的汇编程序。

由于汇编程序都比较长，以下代码截图只是展示效果，没有覆盖全部汇编代码，完整的汇编程序在检查时已经发送给助教了。

9.2.1 基本功能的检查

```
.LC0:
        .string "%d\n"
        .text
        .globl  main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $28, %esp
        movl    $50, %eax
        movl    %eax, -12(%ebp)
        movl    $1, %eax
        negl    %eax
        movl    %eax, -16(%ebp)
        movl    $0, %eax
        movl    %eax, -20(%ebp)
.L0:
        movl    -20(%ebp), %eax
        pushl   %eax
        movl    -12(%ebp), %eax
        pushl   %eax
        movl    $2, %eax
        popl    %ebx
        imull   %ebx, %eax
        popl    %ebx
        cmpl    %eax, %ebx
        setl    %al
        jl              .L1
        jmp             .L2
```

图 9.6: case1.s

```
.LC0:
        .string "%d %d\n"
        .text
        .globl  main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $28, %esp
        movl    $1, %eax
        movl    %eax, -12(%ebp)
        movl    $1, %eax
        movl    %eax, -16(%ebp)
        movl    $2, %eax
        movl    %eax, -24(%ebp)
.L0:
        movl    -24(%ebp), %eax
        pushl   %eax
        movl    $20, %eax
        popl    %ebx
        cmpl    %eax, %ebx
        setl    %al
        jl              .L1
        jmp             .L2
.L1:
        movl    -12(%ebp), %eax
        movl    %eax, -20(%ebp)
        movl    -12(%ebp), %eax
```

图 9.7: case2.s

9.2.2 函数调用的检查

```
.LC0:
        .string "%d\n"
        .text
        .globl  fib
        .type   fib, @function
fib:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $12, %esp
.L0:
        movl    8(%ebp), %eax
        pushl   %eax
        movl    $1000, %eax
        popl    %ebx
        cmpl    %eax, %ebx
        setg    %al
        jg              .L1
        jmp             .L2
.L1:
        movl    8(%ebp), %eax
        pushl   %eax
        movl    12(%ebp), %eax
        popl    %ebx
        imull   %ebx, %eax
        pushl   %eax
        movl    $5, %eax
        movl    %eax, %ebx
        popl    %eax
        cltd
        idivl   %ebx
        movl    %edx, %eax
```

图 9.8: function.s

9.2.3 一维和多维数组的检查

```
.LC0:
        .string "%d\n"
        .text
        .globl  main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $60, %esp
        movl    $6, %eax
        pushl   %eax
        movl    $2, %eax
        popl    %ebx
        movl    %ebx, -12(%ebp,%eax,4)
        movl    $2, %eax
        pushl   %eax
        movl    $1, %eax
        popl    %ebx
        movl    %ebx, -12(%ebp,%eax,4)
        movl    $2, %eax
        pushl   %eax
        movl    $0, %eax
        pushl   %eax
        movl    $3, %eax
        popl    %ebx
        imull   %ebx, %eax
        pushl   %eax
        movl    $1, %eax
        popl    %ebx
        addl    %ebx, %eax
```

图 9.9: array1D.s

9.2.4 一维指针的检查

```
.LC0:
        .string "%d\n"
        .text
        .globl  main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $32, %esp
        movl    $2, %eax
        movl    %eax, -12(%ebp)
        movl    $3, %eax
        movl    %eax, -16(%ebp)
        movl    -12(%ebp), %eax
        pushl   %eax
        movl    $.LC0, %eax
        pushl   %eax
        call    printf
        addl    $8, %esp
        movl    -16(%ebp), %eax
        pushl   %eax
        movl    $.LC0, %eax
        pushl   %eax
        call    printf
        addl    $8, %esp
        leal    -12(%ebp), %eax
        movl    %eax, -20(%ebp)
```

图 9.10: pointer1D.s

9.3 利用汇编器生成可执行文件

通过指令 make exe 可以利用 GCC，将当前目录下的所有.s 汇编程序转化为可执行文件。如下图所示。

```
root@VM-16-2-ubuntu:~/CompilePrinciple/Compiler# make exe
gcc case1.s -m32 -o case1
gcc case2.s -m32 -o case2
gcc function.s -m32 -o function
gcc array1D.s -m32 -o array1D
gcc pointer1D.s -m32 -o pointer1D
```

图 9.11: 生成可执行文件

9.4 运行最终程序

按顺序执行前文生成的可执行文件，最终运行结果正确，代表实验顺利完成。

```
root@VM-16-2-ubuntu:~/CompilePrinciple/Compiler# ./case1
24
root@VM-16-2-ubuntu:~/CompilePrinciple/Compiler# ./case2
6765 4181
root@VM-16-2-ubuntu:~/CompilePrinciple/Compiler# ./function
20
root@VM-16-2-ubuntu:~/CompilePrinciple/Compiler# ./array1D
9
5
root@VM-16-2-ubuntu:~/CompilePrinciple/Compiler# ./pointer1D
2
3
2
2
6
```

图 9.12: 运行最终程序

10 源码仓库

本项目的源代码地址见https://gitee.com/yang-wanli0417/compiler.git