

题目

基于 LR(1)分析方法实现的编译器

实验目的和要求

实验目的：

在该实验中，学生将综合利用汇编语言、高级程序设计语言、数据结构与算法、计算机原理、操作系统、软件工程、编译原理等课程中学到的原理、技术和方法，设计并实现一个将高级程序设计语言编写的源程序翻译为可执行文件的较为完整的小型编译器。该课程将培养学生设计复杂大型软件系统的能力；培养学生应用知识解决问题的能力；培养学生的实践动手能力；培养学生的系统分析与设计能力；培养学生的工程素质。

实验要求：

- 1、编译程序的基本功能
 - 1) 词法分析
 - 2) 语法分析
 - 3) 语义分析和汇编代码生成
 - 4) 符号表管理和存储分配
- 2、编译程序所处理的源程序应具有下列基本功能
 - 1) 变量和一维数组（均 int 型）声明和使用；
 - 2) 赋值语句；
 - 3) 布尔表达式；
 - 4) 三种典型的控制流语句（while, for, if-else；
 - 5) 输入输出语句（scanf、printf）；
 - 6) 算术表达式的运算包括+、-、*、/、++、--。
- 3、编译程序首先能将源程序生成汇编代码，再经过汇编器的编译生成可执行程序，要求生成的可执行程序能够正确运行。

源语言的形式描述

词法的形式化描述：

```
#define hash_key 11
#define var 12
#define digit 13
#define lr_brac 14
#define rr_brac 15
```

```

#define le_add 16
#define le_sub 17
#define le_div 18
#define le_mul 19
#define equal 20
#define no_less 21
#define less 22
#define bigger 23
#define no_biger 24
#define semicolon 25
#define star 26
#define no_equal 27
#define assign 28
#define comma 29
#define ls_brac 30
#define rs_brac 31
#define q_mark 32
#define p_mark 33
#define log_AND 34
#define log_OR 35
#define addone 36
#define subone 37
#define lb_brac 38
#define rb_brac 39
#define le_printf 40
#define le_scanf 41
#define le_string 42
#define le_main 43
#define le_ad 44
#define end_eof 45
char *key_word[] = { "char", "do", "double", "else", "float", "for", "int", "if", "return",
"void", "while" };

```

语法规则的形式描述:

```

$-><S>
<S>-><R> main ( ) { <B> <RET> }
<RET>->return <FH> ;
<RET>->return
<FH>->dig
<R>->int
<R>->void
<B>-><BS> <B>
<B>-><BS>

```

<BS>-><D>
 <BS>-><F>
 <BS>-><X>
 <BS>-><W>
 <BS>-><Y>
 <BS>-><Z> ;
 <Z>->var <A> ++
 <Z>->var ++
 <Z>->var <A> --
 <Z>->var --
 <D>->int var <A> <DF> <DS> ;
 <D>->int var <A> <DF> ;
 <D>->int var <DF> <DS> ;
 <D>->int var <DF> ;
 <D>->int var <A> <DS> ;
 <D>->int var <A> ;
 <D>->int var <DS> ;
 <D>->int var ;
 <F>->var <A> <DF> ;
 <F>->var <DF> ;
 <A>->[var]
 <A>->[dig]
 <A>->[<M>]
 <DF>->= <M>
 <DF>->= { <NS> }
 <NS>->dig
 <NS>->dig <CN>
 <CN>->, dig <CN>
 <CN>->, dig
 <DS>->, var <A> <DF> <DS>
 <DS>->, var <DF> <DS>
 <DS>->, var <A> <DS>
 <DS>->, var <DS>
 <DS>->, var <A> <DF>
 <DS>->, var <DF>
 <DS>->, var <A>
 <DS>->, var
 <M>-><MC> <MS> <MF>
 <M>-><MC> <MS>
 <M>-><MC> <MF>
 <M>-><MC>
 <MF>->+ <M>
 <MF>->- <M>
 <MS>->* <MC> <MS>

```

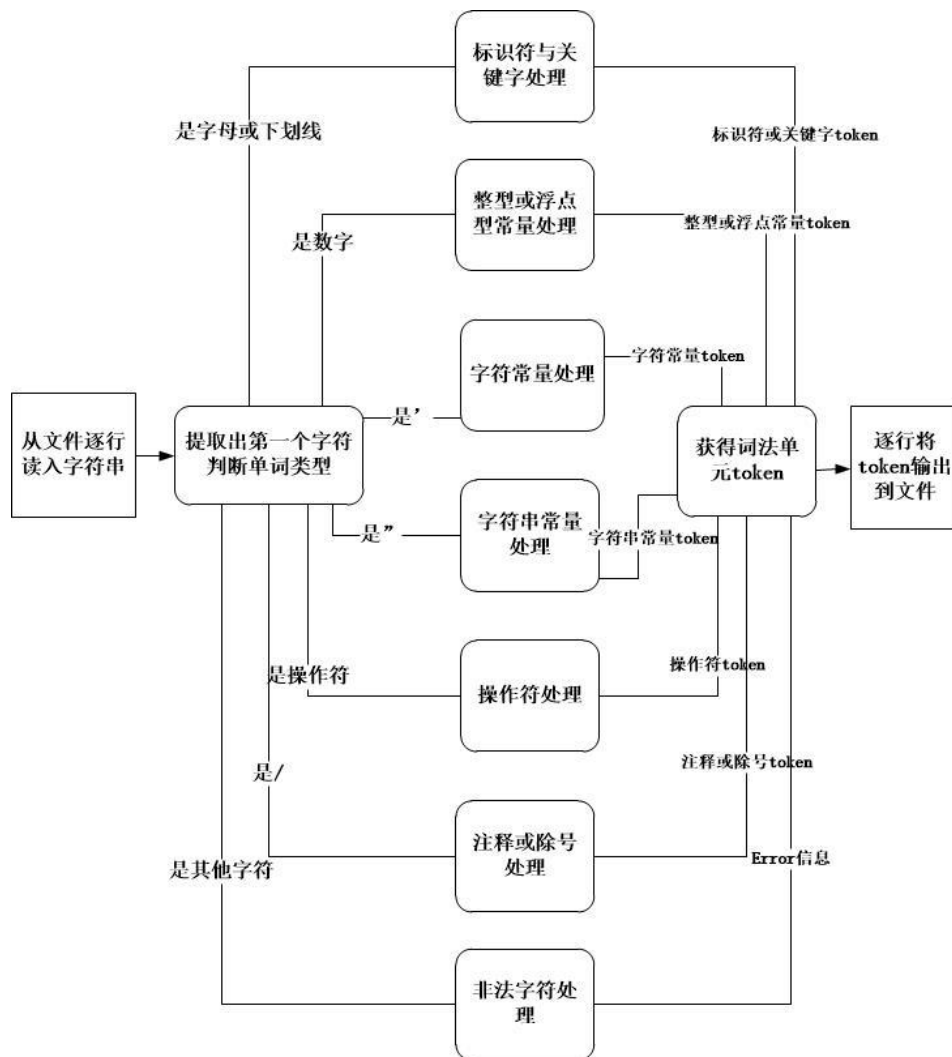
<MS>->/ <MC> <MS>
<MS>->* <MC>
<MS>->/ <MC>
<MC>->( <M> )
<MC>->dig
<MC>->str
<MC>->var <A>
<MC>->var
<X>-><FORX>
<X>-><WHILEX>
<FORX>->for ( <FOD> <FOT> <FOF> ) { <B> }
<FORX>->for ( <FOD> <FOT> ) { <B> }
<FOD>->;
<FOD>-><F>
<FOT>-><T> ;
<FOT>->;
<FOF>->var <A> = <M>
<FOF>->var = <M>
<FOF>-><Z>
<T>->( <M> <GK> <M> <TS> )
<T>->( <M> <GK> <M> )
<T>-><M> <GK> <M>
<TS>->&& <T>
<TS>->|| <T>
<GK>->!=
<GK>->==
<GK>->>=
<GK>-><=
<GK>->>
<GK>-><
<WHILEX>->while <T> { <B> }
<W>-><IF> <IF_E>
<W>-><IF>
<IF>->if <T> { <B> }
<IF_E>-><ELSE>
<ELSE>->else <ELSE_E>
<ELSE_E>-><IF> <IF_E>
<ELSE_E>-><IF>
<ELSE_E>->{ <B> }
<Y>->print ( str , <PRLI> ) ;
<Y>->print ( str ) ;
<Y>->scanf ( str , <SCLI> ) ;
<SCLI>-><SCA>
<SCLI>-><SCLI> , <SCA>

```

<SCA>->& <MC>
 <SCA>-><MC>
 <PRLI>-><PRA>
 <PRLI>-><PRLI> , <PRA>
 <PRA>-><M>

词法分析器的设计

(1) 词法分析器的总体结构



概要设计说明：

本词法分析器采用从文件读取的方式获得输入的代码，逐行读取，以分隔符为界，判断每个分隔开的字符串的第一个字符。根据该字符的不同情况（字母及下划线、数字、单引号、双引号、操作符、左斜线、其他字符）转入不同的词法分析流程，进入相应单词的有限状态机，判断退出状态是否是终态。如果退出状态是终态，则满足合法的词法单元形式，于是输出 token 到文件；如果发现错误，就输出错误信息到文件。

在技术方面，采用了 C 语言实现本词法分析器。

(2) 数据结构设计：符号表、分析表等

符号表是采用 C 的 struct 结构体实现的，结构体中包含五个数据字段：

1. varList: char 型的二维数组，第一维表示变量在符号表中的位置，varList[i] 得到的是第 i 个变量的名字。
2. type: int 型数组，存储第 i 个变量的类型。
3. num: int 型数组，存储第 i 个变量包含的数据个数（如果不是数组则这一项为 1）。
4. addr: int 型数组，存储第 i 个变量的地址，在本编译器的设计中这里用偏移量来表示。
5. end: int 型数据，存储当前符号的结束位置。

具体定义如下：

```
52 struct varList
53 {
54     char varList[100][128]; //允许有100个变量名
55     int type[100];
56     int num[100]; //变量包含的数据个数
57     int addr[100]; //变量的地址（偏移值）
58     int end;
59 }; //这里把函数名与变量名等同对待
```

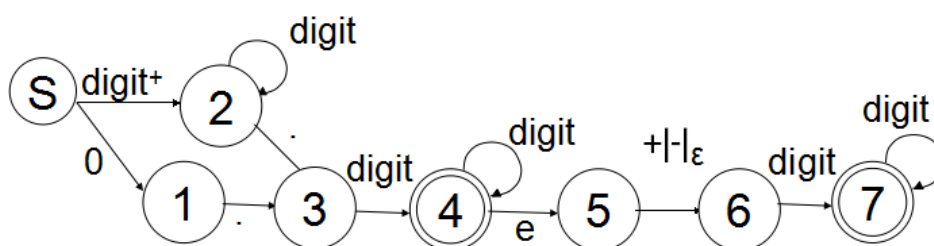
词法返回的 token 采用 C 语言结构体实现，具体定义如下：

```
46 struct lexical
47 {
48     int token_id; //识别出来的词语的id号
49     int value; //词语的有效值
50 };
```

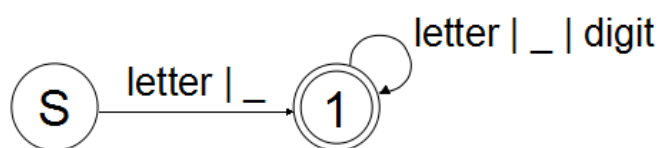
(3) 状态转换图与程序流程图等

● 状态转换图

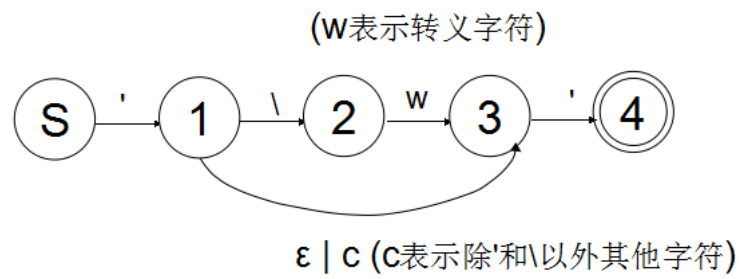
整数和浮点数常量：(d 代表数字)



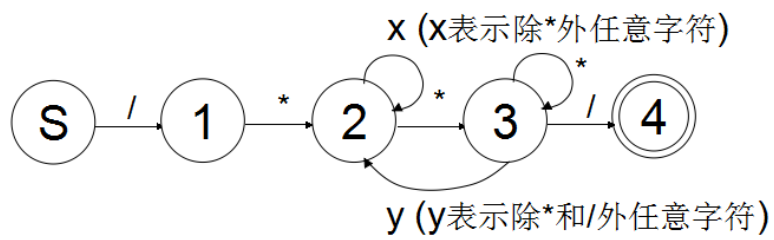
字符串常量：



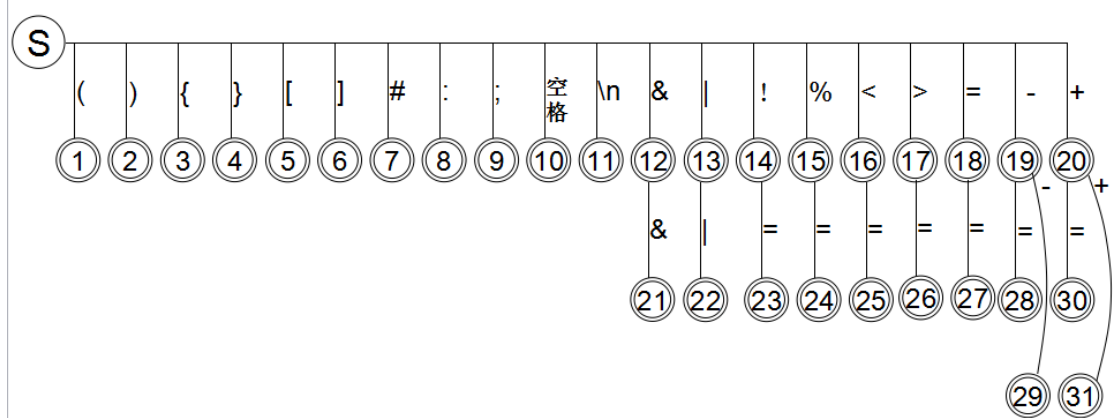
字符常量:



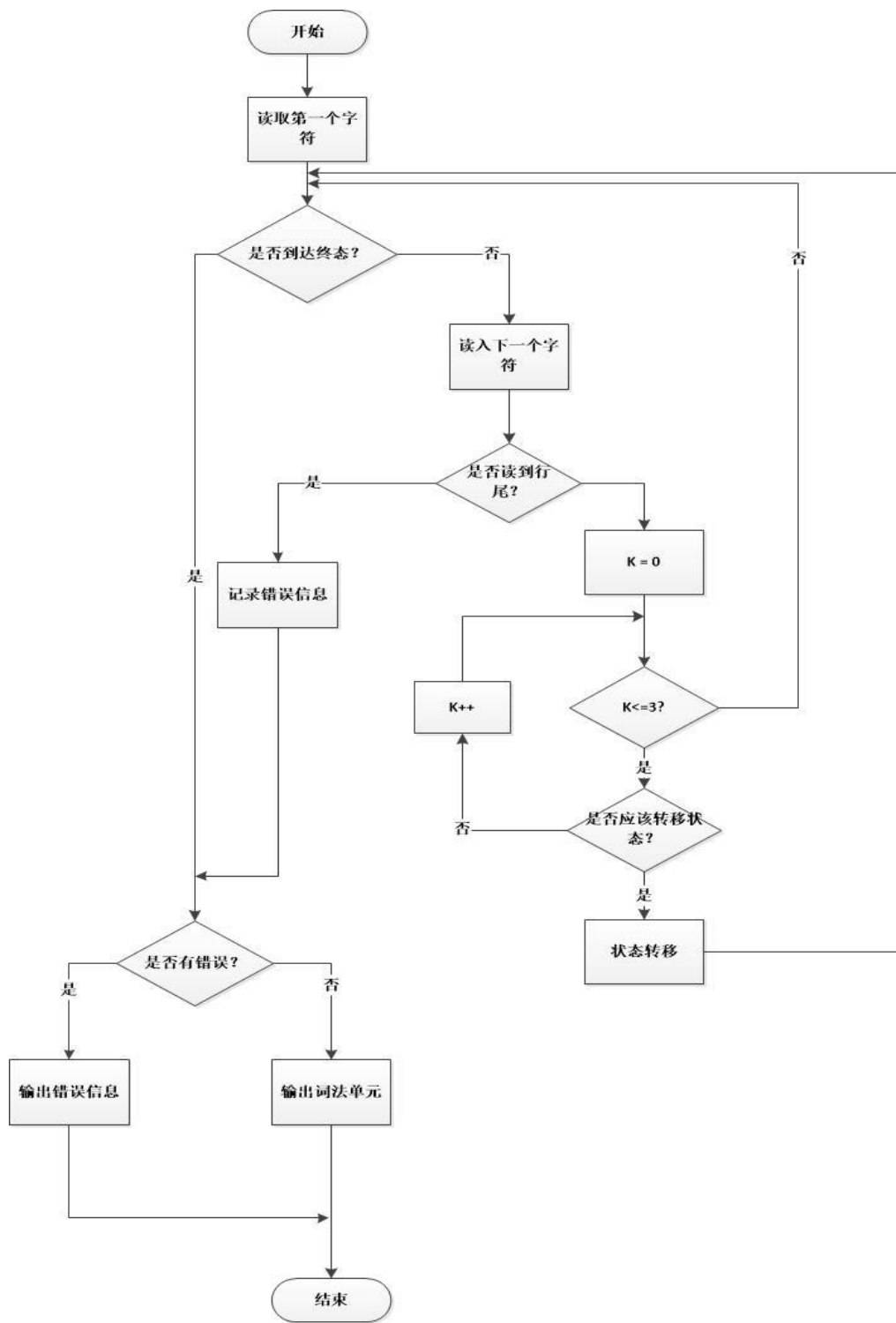
注释:



单界符与双界符:



- 流程图



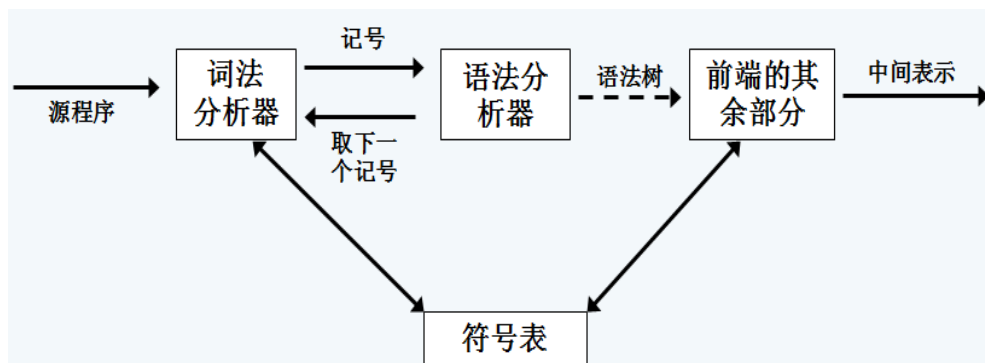
语法分析器的设计与实现

(1) 语法分析器的总体结构

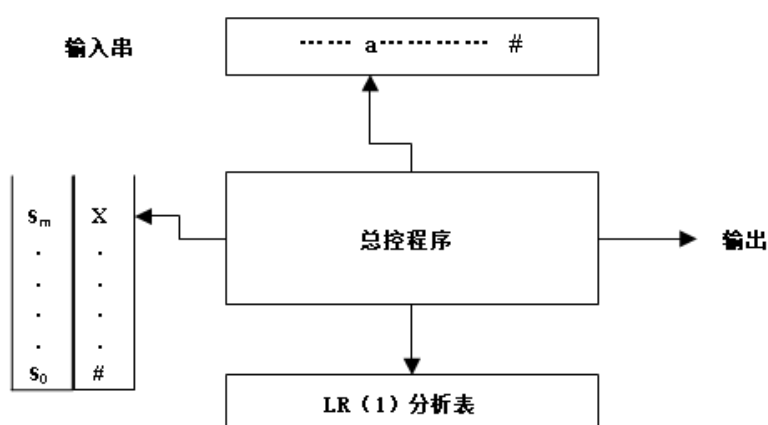
语法分析器的功能：

1. 接收词法分析器提供的记号串。

2. 检查记号串是否能由源程序语言的文法产生。
 3. 输出产生源代码所使用到的产生式。
- 语法分析器在整个编译器中的结构：



语法分析器的系统框架图：



总控程序分为：

文法解析模块：根据文法求解 first 集，利用 first 集以及文法产生式的闭包运算来构造 LR(1)项目集规范族，最后由 LR(1)项目集构造 LR(1)分析表，分析表用于处理模块对输入串进行分析。

处理模块（上图控制程序部分）：根据文法解析模块得到的 Action 表和 Goto 表对输入串进行识别，并采取相应的语法动作，移进操作或者归约操作，但采取归约操作时输出归约使用的产生式。

（2）数据结构设计：分析栈、分析表等

分析栈采用的是双栈模式，一个栈用于存放状态 S，另外一个栈用于存放语法符号，其数据结构定义如下：

```

82     struct stack
83     {
84         int state[9000];
85         Attr sym[9000];
86         int num1;
87         int num2;
88     };
  
```

LR(1)分析表的数据结构是一个二维结构体数组，第一维表示当前处理器处于的状态，第二维表示识别到的文法符号，结构体中有两项，at 字段是一个

char 型数据用于表明是移进还是归约操作，gt 字段是一个 int 型数据用于表明跳转到的状态号。数据结构定义如下：

```
61 struct Entry
62 {
63     char at;
64     int gt;
65 };
66 Entry Action[700][46];
```

构造 first 集，LR(1)项目集规范族以及 LR(1)分析表时，使用 python 中的数据类型字典和 list 来表示。字典即哈希表，字典的 key 是一个由 LR(1)项目集规范族编号和文法符号构成的元组，value 是一个由语义动作和转到的 LR(1)项目集规范族的编号。

(3) LR(1)分析表的构造

主要功能函数说明：

closure(I): 构造文法集 I 的闭包，对于 I 中的每个形如 $[A \rightarrow \alpha.B\beta, a]$ 将文法 G 中形如 $B \rightarrow \eta$ 的产生式和 First($\beta\alpha$)中的每个终结符 b，如果 $[B \rightarrow \eta, b]$ 不在 I 中则将 $[B \rightarrow \eta, b]$ 加入 I 中，直到 I 不在增大。这部分功能在 python 中是用 list 数据结构来表示集合 I 的，用 for 循环遍历集合 I。

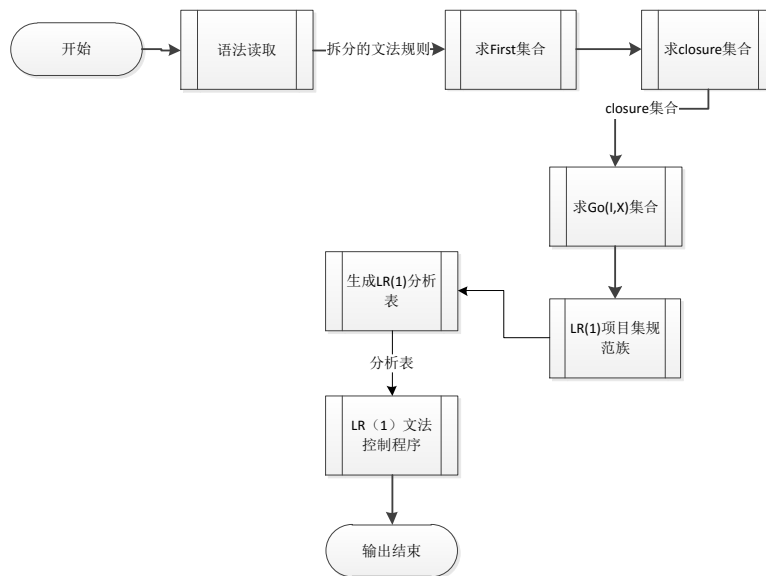
First_str(x): 求字符串 x 的 first 集，用于选用的文法产生式中没有空产生式，所以该函数等价于直接求 x 的第一个字符 a 的 first 集，调用的是 First_sig(a)。First_sig 函数采用递归的方式求解，初始化 first 集为空，如果是终结符则将该终结符加入 first 集中然后直接返回，对于文法 G 中的所有 $a \rightarrow \alpha \dots$ ，递归调用 First_sig(α)，函数返回集合 A，将 A 并入 first 集中。

Go(I,X): 构造文法集 I（一个状态）在输入文法符号为 X 的情况下转移生成的文法集。初始集合 J 为空，对于 I 中所有形如 $[A \rightarrow \alpha.X\beta, a]$ 的项目，将 $[A \rightarrow \alpha.X\beta, a]$ 并入集合 J 中，最后返回 closure(J)。

CreatLR: 构造 LR(1)项目集规范族。先初始化集合 C 为空，然后求 $C \cup \text{closure}([S' \rightarrow \cdot S, \#])$ ，在对于 C 中的每个项 I 用 Go(I,X)求其在文符号转移得到集合 A，将集合 A 并入 C 中。

CreatTable: 构造 Action 表和 Goto 表。根据 CreatLR 生成的 LR(1)项目集规范族来构建状态转移表。

(4) 程序流程图等



语义子程序与汇编代码生成器的设计与实现

(1) 目标语言(汇编语言)的简单定义

语义动作采用直接翻译成汇编代码的形式，所以在我实现的编译器程序里是没有中间代码生成的。汇编代码用的是 AT&T 汇编。AT&T 汇编的简单定义如下：

①定义段空间

```
.section .data      有初值数据段
.section .bss       无初值数据段
.section .text      代码段
```

②程序开始和结束的定义

```
.section .data
a:      .int      4          定义变量 a 值为 4
.section .bss
.lcomm   b,       2          定义变量 b 大小为 2 字节
.section .text
.globl   _start            程序的开始
_start:
movl     $1,      %eax      程序的结束
movl     $0,      %ebx
int      $0x80
```

③寻址的简单定义

base_address(offset_address, index, size)

得到地址为 $\text{base_address} + \text{offset_address} + \text{index} * \text{size}$

例如 $-4(\text{\%eax}, \text{\%ebx}, 2)$ 得到地址为 $-4 + \text{\%eax} + \text{\%ebx} * 2$

其中 base_address 可以为变量或者常量，offset_address 和 index 必须为寄存器

④数据类型的简单定义

.ascii	字符串	
.short	短整型	16 位 2 字节
.int .long	长整型	32 位 4 字节
.byte	一个字节	
.float	浮点单精度	
.double	浮点双精度	
.comm	名称,长度	长度为字节大小, 定义一个为初始化的变量放在 bss 中

(2) 语义子程序的设计

1. 属性的设计

属性包含以下字段:

key: int 型数据。用于表示文法符号的编号,表明当前符号是哪一个文法符号。

str: char 型数组。用于表示当前文法符号的一些信息,例如,值为“reg”时表示当前文法符号的值存储在寄存器中。

value: int 型数据。用于表示当前文法符号的值。

array: int 型数组。当一个文法变量表示一个数组时用于表示这个数值中各个数据的值。

message: int 型数组。用于记录一些额外的补充信息,使得语法树的父节点能根据这些信息翻译出对应的汇编代码。其具体为:0 位表示运算类型,1 位为第一个操作数的值,2 位为第二个操作数的值,3 位代表目的操作数,4 位表示一个操作数的类型(0 表示是数值,1 表示是变量并且存储的是变量的地址,2 表示是寄存器),5 位表示第二个操作数的类型,6 为表示跳转的位置。

isT: bool 型数据。用于表示何时翻译成汇编代码。

具体代码如下:

```
72 struct Attr
73 {
74     int key;
75     char str[200];
76     int value;
77     int array[100];
78     int message[10];
79     bool isT;
80 };
```

2. 存储分配子程序的设计

在语法的识别过程中如果识别到一条产生式是一个声明语句,这对变量分配相应的内存空间。具体做法是在上一个变量的地址的基础上加上当前变量的个数(单个变量个数为 1,数组变量的个数为其包含的元素的个数)与 4(都是 int 型数据)的乘积。具体代码如下:

```

214     varlist.type[nt] = 6;
215     varlist.num[nt] = num;
216     if(nt > 0)
217         varlist.addr[nt] = varlist.addr[nt-1]+4*num;
218     else
219         varlist.addr[nt] = 4*num;

```

(3) 汇编代码生成器的设计与实现

由于本编译器程序没有生成中间代码，当语法分析器识别出一个产生式后采取相应的语义动作将源程序翻译成相应的汇编代码。以下是产生式对应的语义动作：

变量说明

<D>->int var <DS> ;	lookup(var.value);//查表 var.type = int; editTable(var)//在符号表中修改变量的信息。
<CN>->, dig	将 dig 的值添加到<CN>.array 数组中
<CN>->, dig <CN> ₁	将 dig 的值以及<CN> ₁ .array 添加到<CN>.array 中
<NS>->dig <CN>	将 dig 的值以及<CN>.array 添加到<NS>.array 中
<DF>->= { <NS> }	将 <NS>.array 添加到<DF>.array 中。
<D>->int var <A> <DF> ;	根据<DF>.array 中的值翻译数组变量，将 array 中的值赋值给数组变量中的每个元素。 lookup(var.value);//查表 var.type = int; editTable(var)//在符号表中修改变量的信息。

• 赋值

<F>->var = <M>	lookup(var.value);//查表获取 var 的偏移量。 movl <M>.value var 的偏移量(%ebp);//根据产生式直接生成汇编代码。
----------------	--

• 表达式

<M>-><MC> <MF>	<MC>.value;//通过 value 值获得<MC>所代表的变量的地址偏移量。 翻译成 movl 汇编语句，将<MF>
----------------	---

	中的值赋给<MC>所表示的变量。
<MF>->+ <M> - <M>	<MF>.message[1] = <M>.value; <MF>.message[4] = <M>.message[4]; <MF>.message[0] = +/-OP;
<MS> ->* <MC> <MS> <MC> <MS>	<MS>.message[1] = <MC>.value; <MF>.message[4] = <MC>.message[4]; <MS>.message[2] = <MS>.value; <MF>.message[5] = <MS>.message[5]; <MF>.message[0] = *//OP;
<MC>->var <A>	根据<A>所包含的信息，以 var 的地址为基地址加上<A>所表示的偏移量，得到数组元素的地址信息。翻译成 movl 汇编语句，find(reg);//寻找一个尚未使用的寄存器，将元素的值拷贝到寄存器中。 <MC>.value = reg;//将寄存器的编号赋值给<MC>
<MC>->var	Addr =lookup(var.value);//查找 var 变量的地址偏移量 <MC>.value = Addr;
<MC>->dig	<MC>.value = dig.value;//将 dig 的值赋值给<MC>

• 循环

<X>-><WHILEX>	cout<<"jmp " <<"L" <<nt<<endl;//翻译成一句无条件跳转汇编代码
<T>->(<M> <GK> <M> ₁)	<T>.message[1] = <M>.value; <T>.message[2] = <M> ₁ .value; <T>.message[0] = <GK>.OP;
<WHILEX>->WHILE <T> { }	#处理 while 条件语句 temp = newTemp() ans_out[addr] = temp+'':= '+ifStack[0]+ifStack[1]+ifStack[2]

	<pre> addr+=1 whileStart = addr ans_out[addr] = 'if '+temp+' goto '+repr(addr+2) addr+=1 whileHuiTian = addr ans_out[addr] = 'goto ' addr+=1 ifStack = [] } { #while true 代码结束处 ans_out[addr]='goto '+repr(whileStart) addr+=1 ans_out[whileHuiTian]+=repr(a ddr) } </pre>
<FOT>-><T> ;	<pre> cout<<"cml "<<<T>.message[1]<<<T>.mess age[2];//比较两个量的值 switch(<T>.message[0]) { case equal: cout<<"jne"<<"L"<<jumpN<<e ndl; break; case no_less: cout<<"jb"<<"L"<<jumpN<<en dl; break; case less: cout<<"jae"<<"L"<<jumpN<<e ndl; break; case biger: cout<<"jbe"<<"L"<<jumpN<<e ndl; break; case no_biger: cout<<"ja"<<"L"<<jumpN<<en dl; </pre>

	<pre>break; }</pre>
<pre><FORX>->for (<FOD> <FOT> <FOF>) { }</pre>	<pre>if(newattr.message[0] == 1) { cout<<"addl \$0x1,-0x"<<hex<<newattr.mess age[1]<<"(%ebp)"<<endl; } else if(newattr.message[0] == 2) { cout<<"subl \$0x1,-0x"<<hex<<newattr.mess age[1]<<"(%ebp)"<<endl; } }</pre>
<pre><X>-><FORX></pre>	<pre>cout<<"jmp L"<<dec<<jumpN-1<<endl; //翻 译成一句无条件跳转汇编代码</pre>

- 分支

<pre><IF>->if (<T>) { }</pre>	<pre>{ #处理 if 条件语句 temp = newTemp() ans_out[addr] = temp+' := '+ifStack[0]+ifStack[1]+ifStack[2] addr+=1 ans_out[addr] = 'if '+temp+' goto '+repr(addr+2) addr+=1 huiTian = addr ans_out[addr] = 'goto ' addr+=1 ifStack = [] } {</pre>
--	--

	#回填 ans_out[huiTian]+=repr(addr) } cout<<"jmp L"<<jumpF<<endl; cout<<"L"<<jumpN<<":"<<endl; //根据回填的信息 jumpF 和 jumpN 填写 for 循环的地址。
<ELSE_E>->{ }	<ELSE_E>.value = .next;

• 函数返回

<RET>->return <FH> ;	cout<<"movl \$0x"<<hex<<newattr.value<<," %eax"<<endl; cout<<"leave"<<endl<<"ret"<<e ndl;
----------------------	---

测试

(1) 测试样例

```

#include<stdio.h>
int main(){
    int score[8] = {76, 82, 90, 86, 79, 62,90,88};
    int credit[8] = {2, 2, 1, 2, 2, 3,2,2};
    int mean, sum, tmp, i;

    tmp = 0;
    sum = 0;
    for( i = 0; i < 8; i++)/*bukaixin 不开心 */
    {
        sum = sum + score[i] * credit[i];
    }

    while(i != 0) {
        tmp = tmp + credit[i - 1];
        i--;
    }

    mean = sum / tmp;

    if(mean >= 60){
        mean = mean - 60;
        printf("Your score is %d higher than 60!\n", mean);
    }
}

```

```

    }
    else{
        mean = 60 - mean;
        printf("Your score is %d lower than 60!\n", mean);
    }
    return 0;
}

```

(2) 测试结果

- 语法分析结果（输出产生式）

```

产生式: 5    <R>->int
产生式: 57   <MC>->dig
产生式: 49   <M>-><MC>
产生式: 31   <A>->[ <M> ]
产生式: 37   <CN>->, dig
产生式: 36   <CN>->, dig <CN>
产生式: 36   <CN>->, dig <CN>
产生式: 36   <CN>->, dig <CN>
产生式: 36   <CN>->, dig <CN>
产生式: 36   <CN>->, dig <CN>
产生式: 36   <CN>->, dig <CN>
产生式: 36   <CN>->, dig <CN>
产生式: 35   <NS>->dig <CN>
产生式: 33   <DF>->= { <NS> }
产生式: 20   <D>->int var <A> <DF> ;
产生式: 9    <BS>-><D>
产生式: 57   <MC>->dig
产生式: 49   <M>-><MC>
产生式: 31   <A>->[ <M> ]
产生式: 37   <CN>->, dig
产生式: 36   <CN>->, dig <CN>
产生式: 36   <CN>->, dig <CN>
产生式: 36   <CN>->, dig <CN>
产生式: 36   <CN>->, dig <CN>
产生式: 36   <CN>->, dig <CN>
产生式: 36   <CN>->, dig <CN>
产生式: 35   <NS>->dig <CN>
产生式: 33   <DF>->= { <NS> }
产生式: 20   <D>->int var <A> <DF> ;
产生式: 9    <BS>-><D>
产生式: 45   <DS>->, var
产生式: 41   <DS>->, var <DS>
产生式: 41   <DS>->, var <DS>
产生式: 25   <D>->int var <DS> ;
产生式: 9    <BS>-><D>

```

产生式: 57	<MC>->dig
产生式: 49	<M>-><MC>
产生式: 32	<DF>->= <M>
产生式: 28	<F>->var <DF> ;
产生式: 10	<BS>-><F>
产生式: 57	<MC>->dig
产生式: 49	<M>-><MC>
产生式: 32	<DF>->= <M>
产生式: 28	<F>->var <DF> ;
产生式: 10	<BS>-><F>
产生式: 57	<MC>->dig
产生式: 49	<M>-><MC>
产生式: 32	<DF>->= <M>
产生式: 28	<F>->var <DF> ;
产生式: 66	<FOD>-><F>
产生式: 60	<MC>->var
产生式: 49	<M>-><MC>
产生式: 82	<GK>-><
产生式: 57	<MC>->dig
产生式: 49	<M>-><MC>
产生式: 74	<T>-><M> <GK> <M>
产生式: 67	<FOT>-><T> ;
产生式: 16	<Z>->var ++
产生式: 71	<FOF>-><Z>
产生式: 60	<MC>->var
产生式: 60	<MC>->var
产生式: 49	<M>-><MC>
产生式: 31	<A>->[<M>]
产生式: 59	<MC>->var <A>
产生式: 60	<MC>->var
产生式: 49	<M>-><MC>
产生式: 31	<A>->[<M>]
产生式: 59	<MC>->var <A>
产生式: 54	<MS>->* <MC>
产生式: 47	<M>-><MC> <MS>
产生式: 50	<MF>->+ <M>
产生式: 48	<M>-><MC> <MF>
产生式: 32	<DF>->= <M>
产生式: 28	<F>->var <DF> ;
产生式: 10	<BS>-><F>
产生式: 8	-><BS>
产生式: 63	<FORX>->for (<FOD> <FOT> <FOF>) { }
产生式: 61	<X>-><FORX>
产生式: 11	<BS>-><X>

产生式: 60	$\langle MC \rangle \rightarrow \text{var}$
产生式: 49	$\langle M \rangle \rightarrow \langle MC \rangle$
产生式: 77	$\langle GK \rangle \rightarrow !=$
产生式: 57	$\langle MC \rangle \rightarrow \text{dig}$
产生式: 49	$\langle M \rangle \rightarrow \langle MC \rangle$
产生式: 73	$\langle T \rangle \rightarrow (\langle M \rangle \langle GK \rangle \langle M \rangle)$
产生式: 60	$\langle MC \rangle \rightarrow \text{var}$
产生式: 60	$\langle MC \rangle \rightarrow \text{var}$
产生式: 57	$\langle MC \rangle \rightarrow \text{dig}$
产生式: 49	$\langle M \rangle \rightarrow \langle MC \rangle$
产生式: 51	$\langle MF \rangle \rightarrow - \langle M \rangle$
产生式: 48	$\langle M \rangle \rightarrow \langle MC \rangle \langle MF \rangle$
产生式: 31	$\langle A \rangle \rightarrow [\langle M \rangle]$
产生式: 59	$\langle MC \rangle \rightarrow \text{var} \langle A \rangle$
产生式: 49	$\langle M \rangle \rightarrow \langle MC \rangle$
产生式: 50	$\langle MF \rangle \rightarrow + \langle M \rangle$
产生式: 48	$\langle M \rangle \rightarrow \langle MC \rangle \langle MF \rangle$
产生式: 32	$\langle DF \rangle \rightarrow = \langle M \rangle$
产生式: 28	$\langle F \rangle \rightarrow \text{var} \langle DF \rangle ;$
产生式: 10	$\langle BS \rangle \rightarrow \langle F \rangle$
产生式: 18	$\langle Z \rangle \rightarrow \text{var} --$
产生式: 14	$\langle BS \rangle \rightarrow \langle Z \rangle ;$
产生式: 8	$\langle B \rangle \rightarrow \langle BS \rangle$
产生式: 7	$\langle B \rangle \rightarrow \langle BS \rangle \langle B \rangle$
产生式: 83	$\langle WHILEX \rangle \rightarrow \text{while} \langle T \rangle \{ \langle B \rangle \}$
产生式: 62	$\langle X \rangle \rightarrow \langle WHILEX \rangle$
产生式: 11	$\langle BS \rangle \rightarrow \langle X \rangle$
产生式: 60	$\langle MC \rangle \rightarrow \text{var}$
产生式: 60	$\langle MC \rangle \rightarrow \text{var}$
产生式: 55	$\langle MS \rangle \rightarrow / \langle MC \rangle$
产生式: 47	$\langle M \rangle \rightarrow \langle MC \rangle \langle MS \rangle$
产生式: 32	$\langle DF \rangle \rightarrow = \langle M \rangle$
产生式: 28	$\langle F \rangle \rightarrow \text{var} \langle DF \rangle ;$
产生式: 10	$\langle BS \rangle \rightarrow \langle F \rangle$
产生式: 60	$\langle MC \rangle \rightarrow \text{var}$
产生式: 49	$\langle M \rangle \rightarrow \langle MC \rangle$
产生式: 79	$\langle GK \rangle \rightarrow > =$
产生式: 57	$\langle MC \rangle \rightarrow \text{dig}$
产生式: 49	$\langle M \rangle \rightarrow \langle MC \rangle$
产生式: 73	$\langle T \rangle \rightarrow (\langle M \rangle \langle GK \rangle \langle M \rangle)$
产生式: 60	$\langle MC \rangle \rightarrow \text{var}$
产生式: 57	$\langle MC \rangle \rightarrow \text{dig}$
产生式: 49	$\langle M \rangle \rightarrow \langle MC \rangle$
产生式: 51	$\langle MF \rangle \rightarrow - \langle M \rangle$

[illegible]

产生式: 2 <RET>->return <FH> ;

- 汇编代码生成

```
.section .text
.globl _main
_main:
pushl %ebp
movl %esp, %ebp
subl $0x50, %esp
movl $0x4c,-0x20(%ebp)
movl $0x52,-0x1c(%ebp)
movl $0x5a,-0x18(%ebp)
movl $0x56,-0x14(%ebp)
movl $0x4f,-0x10(%ebp)
movl $0x3e,-0xc(%ebp)
movl $0x5a,-0x8(%ebp)
movl $0x58,-0x4(%ebp)
movl $0x2,-0x40(%ebp)
movl $0x2,-0x3c(%ebp)
movl $0x1,-0x38(%ebp)
movl $0x2,-0x34(%ebp)
movl $0x2,-0x30(%ebp)
movl $0x3,-0x2c(%ebp)
movl $0x2,-0x28(%ebp)
movl $0x2,-0x24(%ebp)
movl $0x0,-0x4c(%ebp)
movl $0x0,-0x48(%ebp)
movl $0x0,-0x50(%ebp)
L0:cmpl $0x8,-0x50(%ebp)
jae L1
movl -0x50(%ebp),%ebx
mov -0x40(%ebp,%ebx,4),%ecx
movl -0x50(%ebp),%ebx
mov -0x20(%ebp,%ebx,4),%eax
imull %ecx,%eax
addl -0x48(%ebp),%eax
movl %eax,-0x48(%ebp)
addl $0x1,-0x50(%ebp)
jmp L0
L1:
L2:cmpl $0x0,-0x50(%ebp)
jz L3
movl -0x50(%ebp),%ebx
subl $0x1,%ebx
mov -0x40(%ebp,%ebx,4),%edx
```

```

    addl -0x4c(%ebp),%edx
    movl %edx,-0x4c(%ebp)
    subl $0x1,-0x50(%ebp)
    jmp L2
L3:
    movl -0x48(%ebp),%eax
    cltd
    idivl -0x4c(%ebp)
    movl %eax,-0x44(%ebp)
L4:cmpl $0x3c,-0x44(%ebp)
    jb L5
    movl -0x44(%ebp),%ebx
    subl $0x3c,%ebx
    movl %ebx,-0x44(%ebp)
    pushl -0x44(%ebp)
    pushl $text6
    call _printf
    jmp L6
L5:
    movl $0x3c,%ebx
    subl -0x44(%ebp),%ebx
    movl %ebx,-0x44(%ebp)
    pushl -0x44(%ebp)
    pushl $text7
    call _printf
L6:
    movl $0x0,%eax
    leave
    ret

.section .data
text6: .ascii "Your score is %d higher than 60!\n\0"
text7: .ascii "Your score is %d lower than 60!\n\0"

```

- 符号表

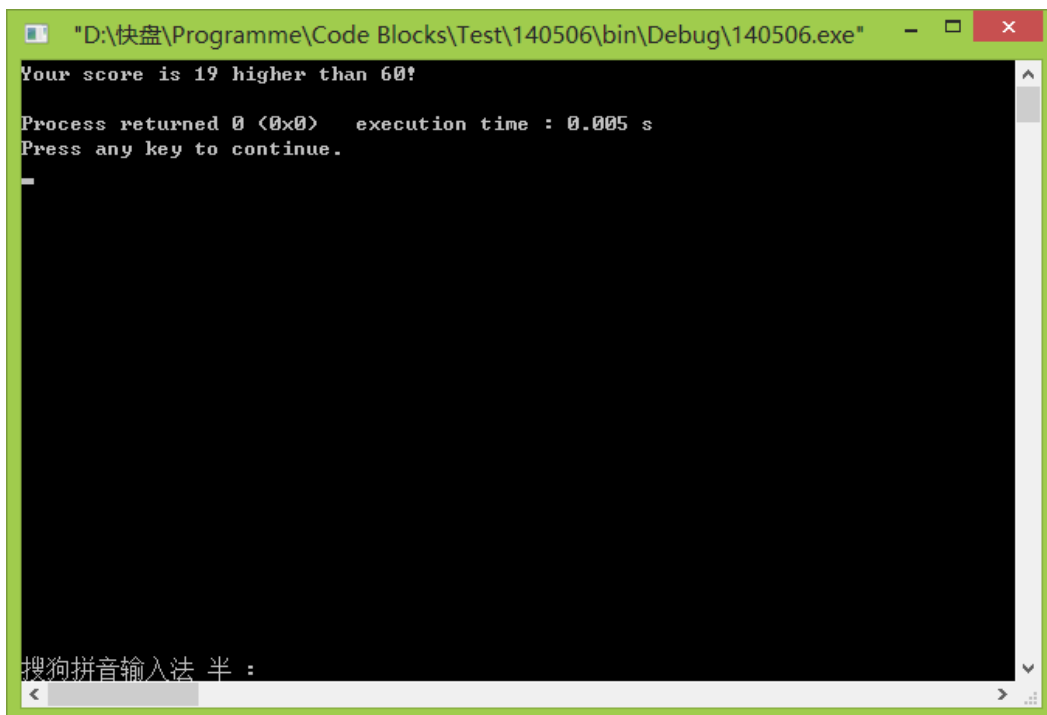
```

0: score   8   32
1: credit  8   64
2: mean    1   68
3: sum     1   72
4: tmp     1   76
5: i       1   80
6: Your score is %d higher than 60!\n   0   0
7: Your score is %d lower than 60!\n   0   0

```

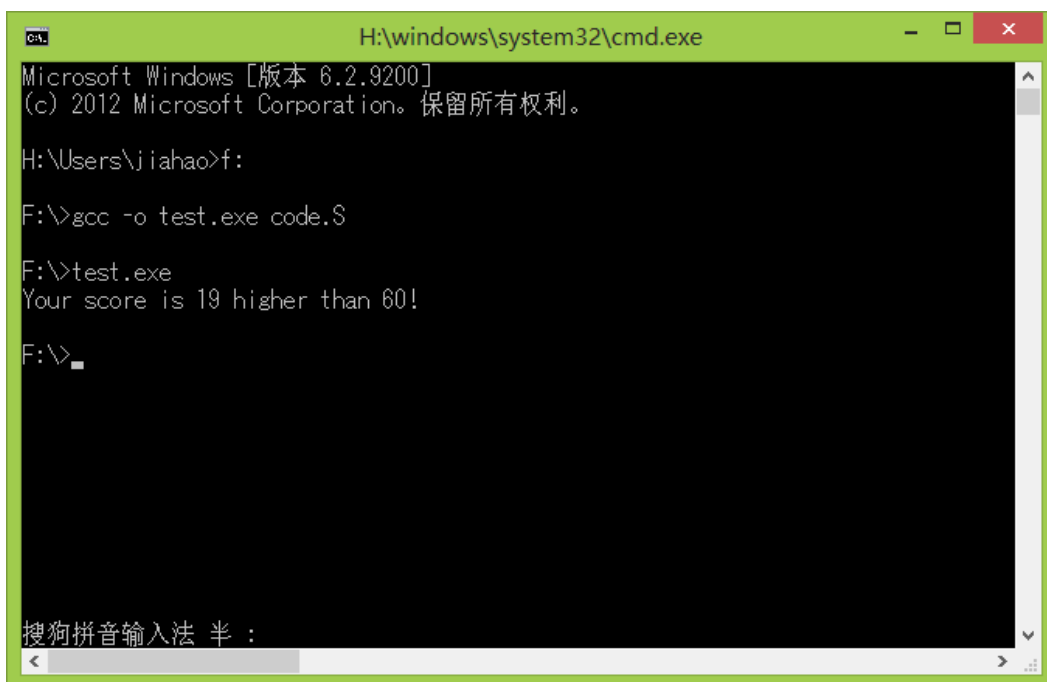
- 运行结果

Code::Blocks 编译源程序后的运行结果：



```
"D:\快盘\Programme\Code Blocks\Test\140506\bin\Debug\140506.exe"
Your score is 19 higher than 60!
Process returned 0 (0x0) execution time : 0.005 s
Press any key to continue.
搜狗拼音输入法 半 :
```

采用本编译器翻译成的汇编代码的运行结果：



```
H:\windows\system32\cmd.exe
Microsoft Windows [版本 6.2.9200]
(c) 2012 Microsoft Corporation。保留所有权利。
H:\Users\jiahao>f:
F:\>gcc -o test.exe code.S
F:\>test.exe
Your score is 19 higher than 60!
F:\>
搜狗拼音输入法 半 :
```

(3) 结果分析

词法分析部分：

通过分析实验的输出结果，可以精确地分开每一行，识别出该行的所有词法单元，其中关键字和操作符用例如<return,>的 token 表示；标识符用例如<标识符,符号表中位置 i>的 token 表示；整数或浮点数常量用例如<整数常量,0>的 token 表示；字符常量用例如<字符常量,a>的 token 表

示：字符串常量用例如<字符串常量,字符串在符号表中的位置 i>的 token 表示；注释能够分析出来并直接跳过注释部分。

语法分析部分：

- 1.实验的结果准确地输出了每读入一个词法单元后的推导产生式。
- 2.说明本语法分析程序可以处理 C 语言的函数声明语句、函数定义语句、变量声明语句、赋值语句、相当一部分表达式语句、分支结构和循环结构。
- 3.除此之外，本程序还能分析出数组的声明与赋值语句、常量、后缀表达式、自增减运算、布尔表达式等等。能区分表达式中各种运算符的优先级。
- 4.在读入某个词法单元出错时，可以分析出错误的类型，并且输出错误类型。在处理错误时不会影响到后面的分析。

代码生成部分：

- 1.本程序在读入函数声明语句时会建立符号表，在读入函数内局部变量定义语句时会填写符号表项，最后可以输出符号表内容（类型、名称、偏移量）；
- 2.本程序在读入表达式或控制语句时会产生汇编代码并输出。
- 3.支持表达式运算符的优先级区分，支持循环和分支结构的嵌套；
- 4.能进行语法级别错误的处理。

总结

通过本次实验我深入了解了从词法分析到最终汇编代码生成的整个过程，理解了 LR(1)分析方法的原理以及 LR(1)分析表的构建。虽然在整个实验过程中遇到了很多困难。比如，在构建 LR(1)分析表时就始终不对，在语法数量比较少时结果又是正确的，一旦语法数量比较大时结果就出现了错误甚至出现递归深度太大而致使程序被迫终止退出现象，这时候唯一的解决方法就是静下心来跟踪程序的执行同时手动计算 LR(1)规范项目集以及状态转换图，然后慢慢发现程序的错误之处。虽然 Debug 的过程很漫长也很痛苦，但是当错误被修复程序正确执行时那种喜悦真是溢于言表，同时也能真切的感受到自己调试程序的能力又有所提高了。