编译原理课程设计说明书

—　类C语言编译器

学　　院　电子与信息工程学院

专　　业　计算机科学与技术

授课老师　卫志华

学　　号　1853790

姓　　名　庄镇华

完成日期　2021.06.14

目录

一、 需求分析

1.1 背景概述

编译器是编译系统的核心，主要负责解析源程序的语义，生成目标机器代码。一般情况下，编译流程包含词法分析、语法分析、语义分析和代码生成四个阶段。符号表管理和错误处理贯穿于整个编译流程。如果编译器支持代码优化，那么还需要优化器模块。

下图展示了一个优化编译器结构。

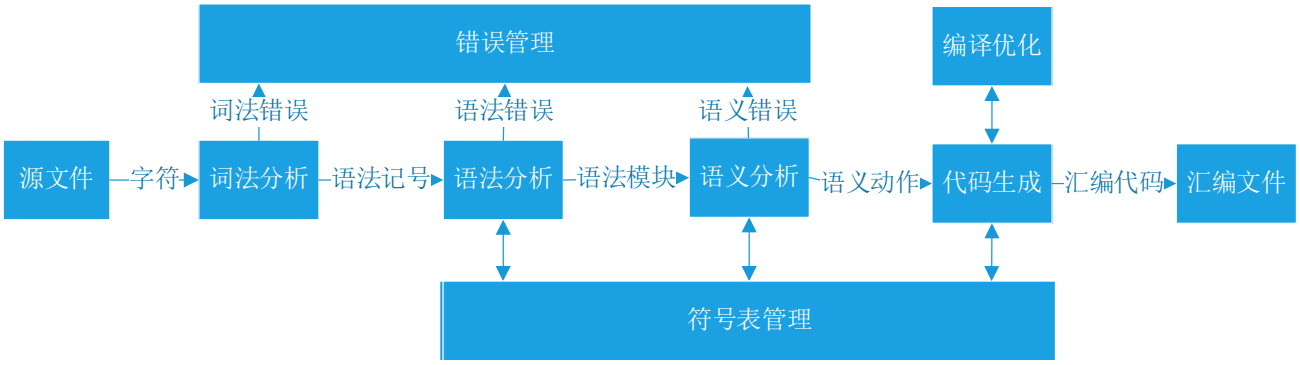


图 1-1 编译器结构

1.2 任务要求

- 使用高级程序语言作为实现语言，实现一个类 C 语言的编译器。编码实现编译器的组成部分。
- 要求的类 C 编译器是个一遍的编译程序，词法分析程序作为子程序，需要的时候被语法分析程序调用;
- 使用语法制导的翻译技术，在语法分析的同时生成中间代码，并保存到文件中。
- 要求输入类 C 语言源程序，输出中间代码表示的程序;
- 要求输入类 C 语言源程序，输出目标代码(可汇编执行)的程序。
- 实现过程、函数调用的代码编译。
- 拓展类 C 语言文法，实现包含数组的中间代码以及目标代码生成。

1.3 用户需求

高级计算机语言便于人编写，阅读交流，维护。机器语言是计算机能直接解读、运行的。编译器将汇编或高级计算机语言源程序（Source program）作为输入，翻译成目标语言（Target language）机器代码的等价程序。

汇编器和链接器的出现大大提高了编程效率，降低了编程和维护的难度。但是人们对汇编语言的能力并不满足，于是就出现了如今形形色色的高级编程语言。这样就面临着一个问题——如何将高级语言翻译为汇编语言？这正是编译器所做的工作。编译器比汇编器复杂得多。汇编语言的语法比较单一，它与机器语言有基本的对应关系。而高级语言形式比较自由，计算机识别高级语言的含义比较困难，而且它的语句翻译为汇编语言序列时有多种选择，如何选择更好的序列作为翻译结果也是比较困难的，所以这些问题都亟待解决。

为了解决以上需求，实现人们使用简洁易懂的编程语言与计算机交流的目的，高级语言编译器的实现是十分必要的。

1.4 功能需求

1.4.1 任务输入及其范围

程序输入为类 C 语言程序，其要求为符合如下表格的词法及语法规则。**最高难度：**一段带过程调用以及数组的代码段。**一般难度：**符合语法规则的简单组合语句。

【词法规则】：

关键字：int | void | if | else | while | return

标识符： 字母（字母|数字）* （**注：不与关键字相同**）

数值：数字（数字）*

赋值号： =

算符： + | - | * | / | = | == | > | >= | < | <= | !=

界符： ；

分隔符： ，

注释号： /* */ | //

左括号： （

右括号： ）

左中括号： [

右中括号：]

左大括号： {

右大括号： }

字母：| a |....| z | A |....| Z |

数字：0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

结束符：#

【语法规则】：（注： { }中的项表示可重复若干次）

Program ::= <声明串>

<声明串> ::=<声明>{<声明>}

<声明> ::=int <ID> <声明类型> | void <ID> <函数声明>

<声明类型>::=<变量声明> | <函数声明> | <数组声明>

<变量声明> ::= ；

<函数声明> ::= ‘(’ <形参> ‘)’ <语句块>

<数组声明> ::= ‘[’ (数字)+ ‘]’ { ‘[’ (数字)+ ‘]’ }

<形参>::= <参数列表> | void

<参数列表> ::= <参数> {, <参数>}

<参数> ::= int <ID>

<语句块> ::= ‘{’ <内部声明> <语句串> ‘}’

<内部声明> ::= 空 | <内部变量声明>;{<内部变量声明>;}

<内部变量声明>::=int <ID>

<语句串> ::= <语句>{ <语句> }

<语句> ::= <if 语句> |< while 语句> | <return 语句> | <赋值语句>

<赋值语句> ::= <ID> ‘=’ <表达式>; | <数组> ‘=’ <表达式>;

<return 语句> ::= return [<表达式>] ;（**注：[ ]中的项表示可选**）

<while 语句> ::= while ‘(’ <表达式> ‘)’ <语句块>

<if 语句> ::= if ‘(‘<表达式> ‘)’ <语句块> [else <语句块>]（**注：[ ]中的项表示可选**）

<表达式>::=<加法表达式>{ relop <加法表达式> } （**注：relop-> <|<=|>|>=|==|!=**）

<加法表达式> ::= <项> {+ <项> | -<项>}

<项> ::= <因子> {* <因子> | /<因子>}

<因子> ::=num | ‘(’ <表达式> ‘)’ |<ID> FTYPE |<数组>

FTYPE ::= <call> | 空

<call> ::= ‘(’ <实参> ‘) ’

<数组>::=<ID> ‘[’ <表达式> ‘]’ | <数组> ‘[’ <表达式> ‘]’

<实参> ::=<实参列表> | 空

<实参列表> ::= <表达式>{, <表达式>}

<ID>::=字母(字母|数字)*

【代码实例】：(最高难度)

```
int program(int a,int b,int c)
{
    int i;
    int j;
    i=0;
    if(a>(b+c))
    {
        j=a+(b*c+1);
    }
    else
    {
        j=a;
    }
    while(i<=100)
    {
        i=j*2;
    }
```

```
    return i;
}
int demo(int a)
{
    a=a+2;
    return a*2;
}
void main(void)
{
    int a[2][2];
    a[0][0]=3;
    a[0][1]=a[0][0]+1;
    a[1][0]=a[0][0]+a[0][1];
    a[1][1]=program(a[0][0],a[0][1],demo(a[1][0]));
    return;
}
```

1.4.2 输出形式

输出形式包含屏幕输出和文件输出：

1. 屏幕输出包含词法分析结果，语法分析结果，中间代码生成结果，目标代码生成结果。
2. 输出到文件的形式包含两个，分别为中间代码生成结果和目标代码生成结果。

部分示例如下：

	operation	arg1	arg2	result
1	f1	:	_	_
2	pop	_		t1
3	pop	_		t2
4	pop	_		t3
5	-	fp		fp
6	:=	0	_	t4
7	+	t2	t3	t6
8	>	t1	t6	t10
9	j>	t10	0	l1
10	j	_	_	l2
11	l1	:	_	_
12	*	t2	t3	t7
13	+	t7	1	t8
14	+	t1	t8	t9
15	:=	t9	_	t5
16	j	_	_	l3
17	l2	:		

图 1-2 中间代码输出结果

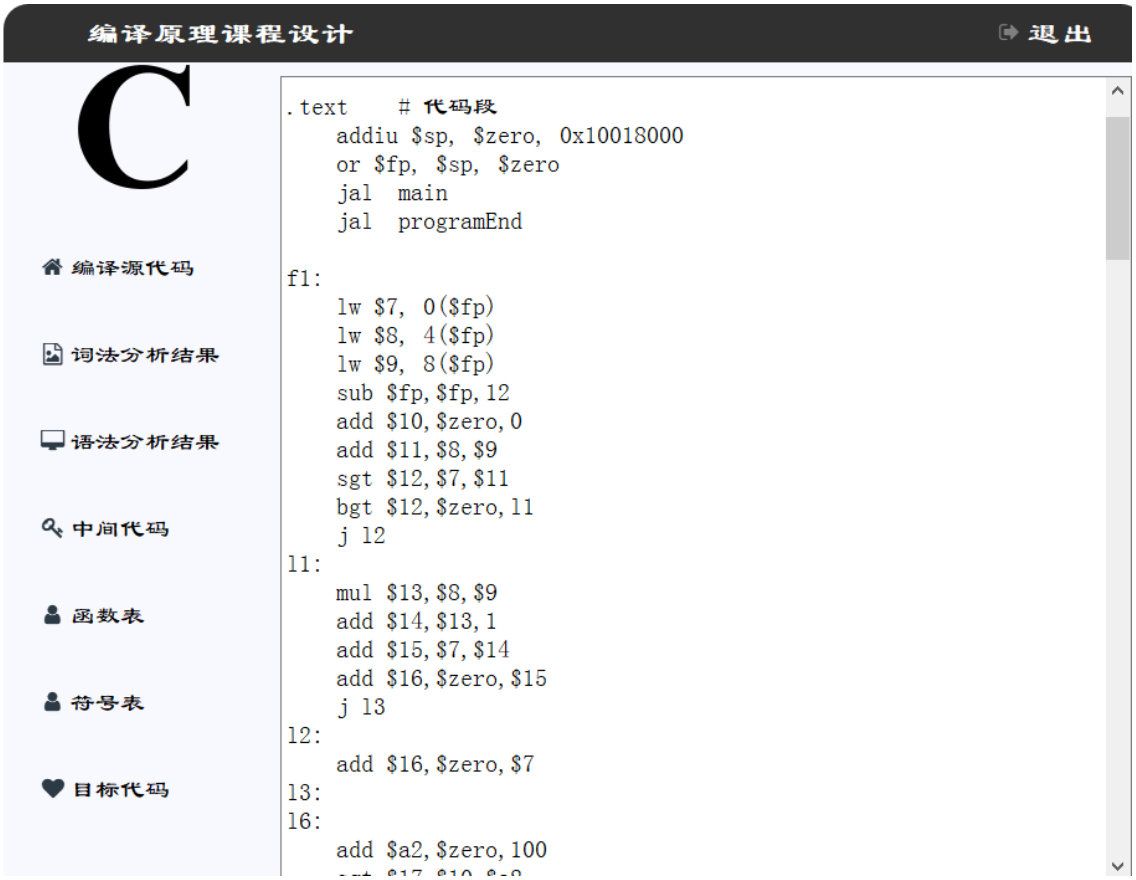


图 1-3 目标代码输出结果

1.4.3 程序功能

1. 能够使用语法制导的翻译技术，在语法分析的同时生成中间代码，并保存到文件中。
2. 输入类 C 语言源程序，能够输出中间代码表示的程序。
3. 输入类 C 语言源程序，能够输出目标代码(可汇编执行)的程序。
4. 能够实现过程、函数调用、多维数组的代码编译。

1.4.4 测试数据

正确输入数据已经在上文 1.4.1 节给出，故此处不再赘述。错误输入数据如下：

变量重定义

```
int t1;
int t1;
```

使用未声明的变量

```
int main()
{
  t2 = 1;
  return 0;
}
```

使用未定义的函数

```
int main()
{
   int t3 = demo3(3 * 9);
   return 0;
}
```

变量赋值时类型错误

```
int main()
{
   int t4 = 1;
   void t5 = t4;
   return 0;
}
```

函数形参和实参不匹配

```
int demo()
{
    return 16;
}
int main()
{
   int t6 = demo(6 * 9);
   return 0;
}
```

1.5 可行性分析

1.5.1 经济可行性

由于是本地开发，不需要办公场所设施；单人开发，无需开发人员工资；至于资料费和办公消耗，均在可承受范围内。并且该系统使用后可以得到非定量收益，高效编译程序，充分发挥计算机设备功能，提高工作效率。

权衡利弊，该系统开发带来的收益高出成本支出，因此开发该系统具有经济可行性。

1.5.2 技术可行性

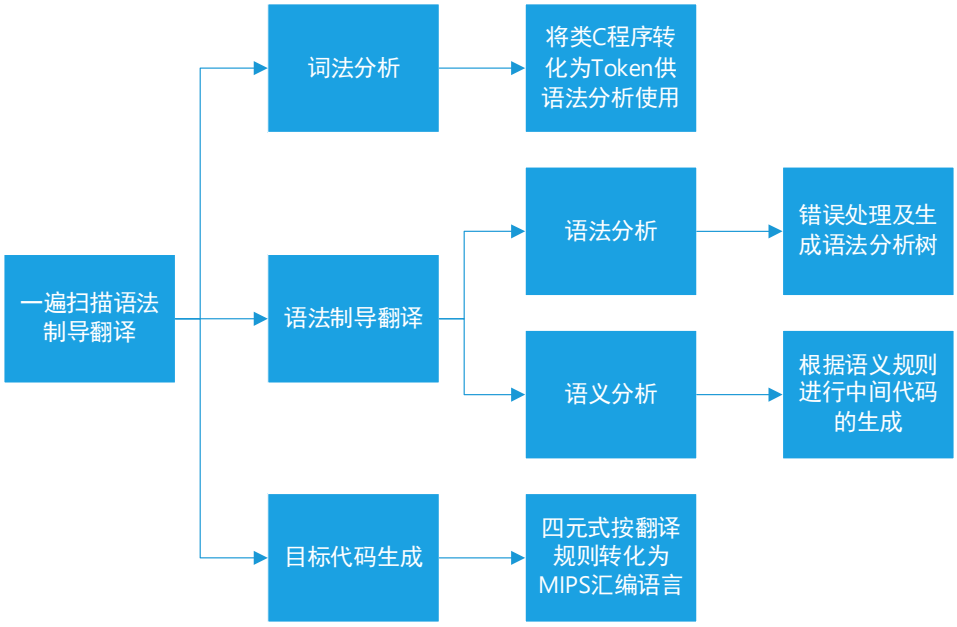
由于上学期编译原理已经使用 LR(1) 的分析方法完成了类 C 语言程序的词法分析、语法分析，并且使用 LR(1) 分析方法进行编译器实现已有前人铺路；

数组、中间代码和目标代码生成方面。在语义分析过程中生成中间代码、由中间代码生成目标代码可由语义分析和目标代码生成章节中的理论实现，数组的加入在语法分析、语义分析和目标代码生成章节也有理论实现，因此认为该系统在技术方面具有可行性。

二、 概要设计

2.1 任务的分解

一般来讲，编译器主要有 5 个步骤：词法分析、语法分析、语义分析、中间代码生成和目标代码生成，根据任务指导书给出的要求，LR(1)方法是适用的。词法分析作为子程序被语法分析调用，语法分析的过程中，语义分析、中间代码生成也在同步进行。



2.2 数据类型定义

使用面向对象的方法，将每个阶段看作一个对象，对该阶段的操作就是该对象的方法。这既符合程序开发的一般流程又简洁优雅，大大方便程序的开发和维护。对于每一部分，又需要不同的数据结构来维护，接下来将详细说明。

词法分析类 LexicalAnalyzer：

重点函数或变量	功能与意义
CURRENT_LINE	词语当前行数
pInputStr	当前扫描指针
Reserved： {'if': 'IF', 'else': 'ELSE', 'while': 'WHILE', 'int': 'INT', 'return': 'RETURN', 'void': 'VOID'}	保留字
Type： ['seperator', 'operator', 'identifier', 'int']	类别
Token： {class, row, colum, name, data, type}	词法分析结果，包括种类（是否为终结符）、行数、列数、名字、数据、类型（具体种类）

语法规则类 ItemSet：

重点函数或变量	功能与意义
Items	单个项目
Reserved	保留字
NonTerminalSymbols	变元
TerminalSymbols	终结符
OriginStartSymbol	原起始符
StartSymbol	广义起始符
calFirstSet	计算 First 集
scanLine	调用词法分析函数，扫描单行
generateTokens	调用词法分析函数，生成 Token
loadGrammer	读取文法

语法规则类 ItemSetFamily：

重点函数或变量	功能与意义
itemSets	DFA 的状态
prods	项目集规范族的 DFA 的产生式
getLeftNT	获取某个非终结符的产生式
getLR1Closure	计算闭包
getFirstSet	获取字符串的 First 集
buildFamily	构建项目集规范族

语法分析类 SyntacticAnalyzer：

重点函数或变量	功能与意义
ACTION[s, a]	当状态 s 面临输入符号 a 时，下一步采取的动作
GOTO[s, X]	当状态 s 面对文法符号 X 时，下一步转移的状态
isRecognizable	判断一个字符串是否能被识别
getTables	计算 ACTION 和 GOTO 数组
item2prodIdx	返回项目对应的产生式编号

语义分析中间代码生成类 SemanticAnalyser：

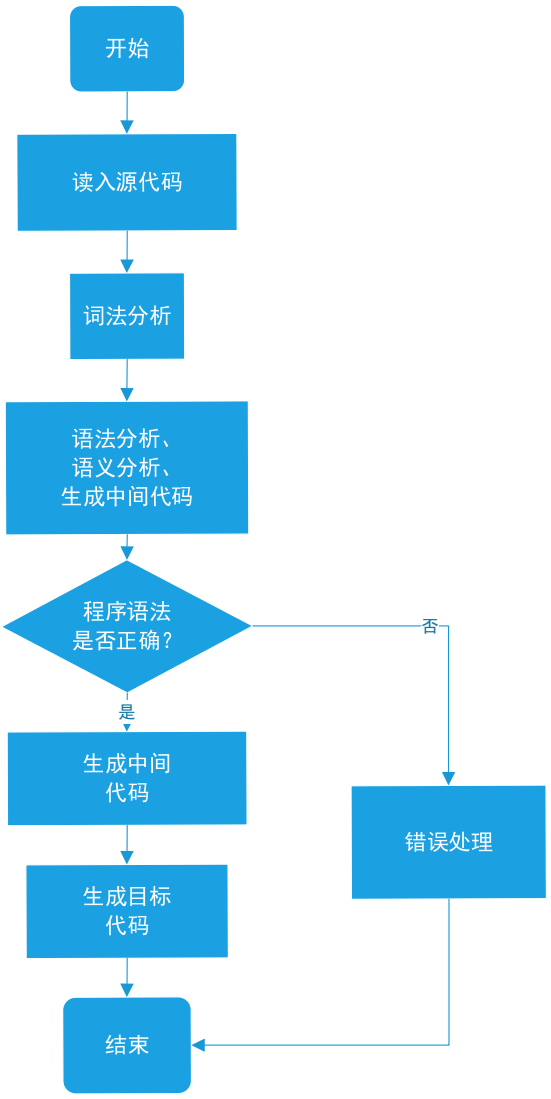
重点函数或变量	功能与意义
sStack	语义分析栈
symbolTable	符号表
funcTable	函数表
curTempId	中间变量名序号

curLabelId	当前标号名序号
semanticAnalyze	产生式规约时进行语义分析，并生成对应中间代码

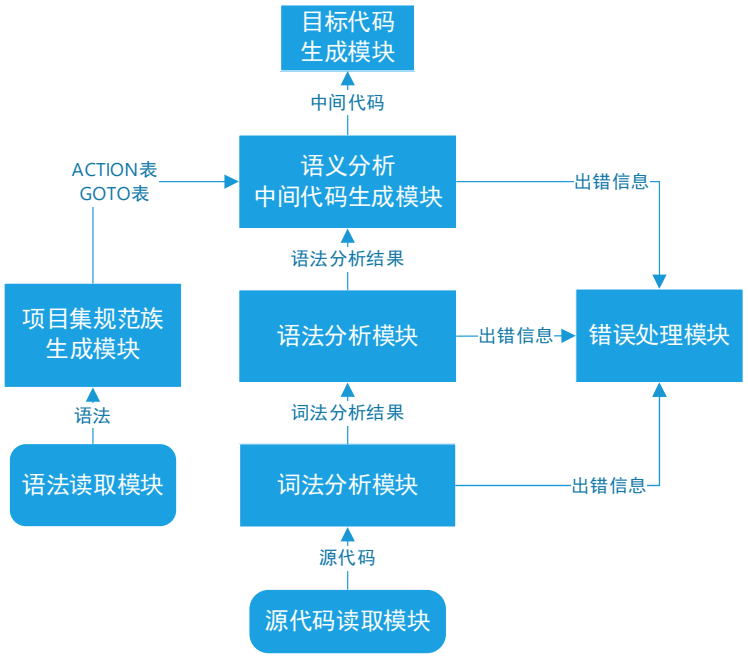
目标代码生成类 ObjectCodeGenerator：

重点函数或变量	功能与意义
mipsCode	存储目标代码
regTable	记录寄存器内部存的是哪个变量的值
varStatus	记录变量是在寄存器中还是内存中
getRegister	获取寄存器
freeRegister	释放寄存器
genMips	生成目标代码

2.3 主程序流程



2.4 模块间的调用关系



三、 详细设计

3.1 词法分析设计

3.1.1 词法分析器功能及结构

编译器工作之前，需要将高级语言书写的源程序作为输入。它顺序扫描源文件内的字符，通过与词法记号的有限自动机进行匹配，产生各式各样的词法记号。我们使用类 C 语言定义高级语言。词法分析器通过对源文件的扫描获得高级语言定义的词法记号。所谓词法记号（也称为终结符），反映在高级语言语法中就是对应的标识符、关键字、常量，以及运算符、逗号、分号等界符。如下图：

图 3-1 词法分析器功能

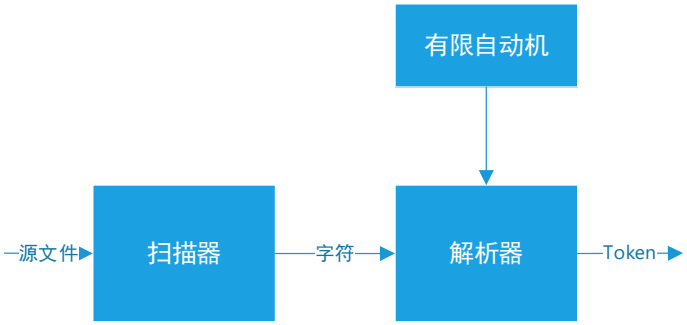


图 3-2 词法分析器结构

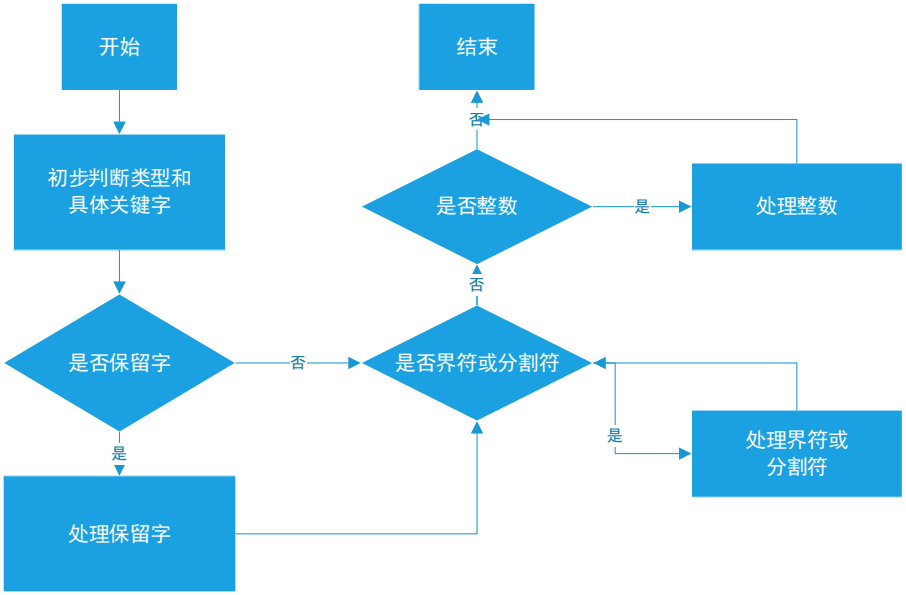


图 3-3 实际词法分析器设计

3.1.2 扫描器

扫描器读取源文件，按序返回文件内的字符，直到文件结束。比较高效的方式是使用一块缓冲区保存后续的多个字符，每次调用时首先从缓冲区内按序获取字符，只有缓冲区为空时才会读取磁盘重新加载缓冲区，类似于 Linux 文件系统的预读机制。

```
def scan_line(self, line):  # 对一行进行重复扫描，获得一组 token
    tokens = []
    result = line.strip().strip('\t')
    origin = result
    while True:
        if result == "":
            break
        before = result
        result = self.scan(result)
        if result['regex']:
            # 保留字，对应文法中->不加引号，认定为终结符
            if result['data'] in self.reserved:
                pass
            # 操作符或者界符，对应文法中->加引号，认定为终结符
            if token['name'] == "operator".upper() or token['name'] ==
"seperator".upper():
                pass
            if token['name'] == "INT":
                pass
            tokens.append(token)
return tokens
```

3.1.3 Token 语法记号

关键字：int、void、if、else、while、return

算符：+、-、*、/、%、 ==、 >=、 <=、 >、<、 !=

界符：;

分割符：,

注释号：// /* */

结束符：#

左括号：(

右括号：)

左中括号：[

右中括号：]

左大括号：{

右大括号：}

数值：数字(数字)*

赋值号：=

3.1.4 有限自动机

有限自动机识别的语言称为正则语言，有限自动机分为 DFA 和 NFA 两组。DFA 和 NFA 都可以描述正则语言，DFA 规定只能有一个开始符号，且转移标记不能为空，其代码实现较为方便。由于每个 NFA 都可以转化为一个 DFA，我们设计的词法分析器统一使用 DFA 描述前面定义的所有词法记号。

由于 python 有丰富的库提供调用，因此我们使用 re 正则表达式库来定义识别单词的 DFA。

```
# 词法分析所使用的正则表达式
self.regexs = [
    '\{|\}|\[|\]|\(|\)|,|;'          # 界符
    , '\+|-|\*|/|==|!=|>=|<=|>|<|='  # 操作符
    , '[a-zA-Z][a-zA-Z0-9]*'         # 标识符
    , '\d+'                          # 整数
]
```

如果匹配成功，返回匹配的具体关键字并初步判断该关键词的类型，随即生成词法分析结果 Token 供语法分析对象调用。

3.2 LR(1)语法分析设计

3.2.1 语法分析功能及结构

语法分析器获取词法分析器提供的 Token，根据高级语言文法的结构，识别不同的语法模块。经过语法分析器的处理后，Token 序列形成一棵完整的抽象语法树，抽象语法树的子树（包括抽象语法树本身）也称为语法模块。高级语言的文法直接影响语法分析器的结构。本次设计采用 LR(1)文法，下图描述了语法分析器的结构。

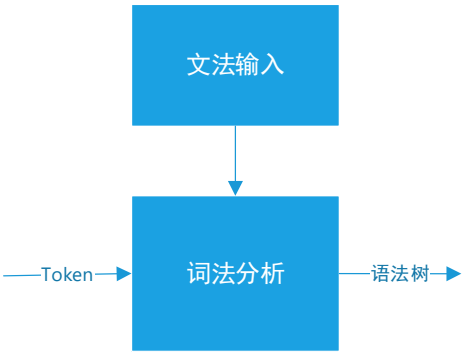


图 3-4 语法分析器功能及结构

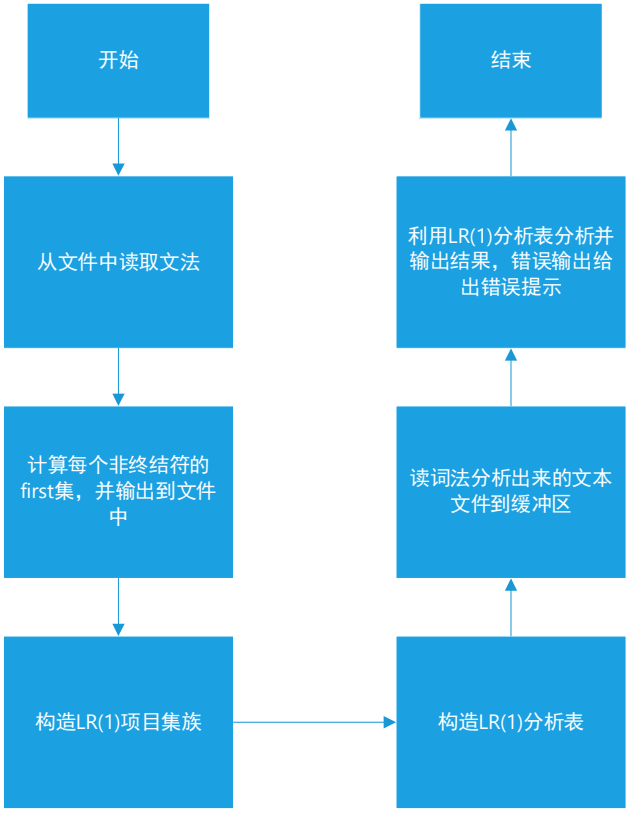


图 3-5 实际语法分析器设计

3.2.2 语法分析重要原理

1. LR(1)文法定义：

LR 文法: 对于一个文法，如果能够构造一张 LR 分析表, 使得它的每个入口均是唯一确定，则该文法称为 LR 文法。在进行自下而上分析时，一旦栈顶形成句柄，即可归约。

LR(k)文法：对于一个文法，如果每步至多向前检查 k 个输入符号，就能用 LR 分析器进行分析。则这个文法就称为 LR(k)文法。

大多数适用的程序设计语言的文法不能满足 LR(0) 文法的条件，因此使用 LR(0)规范族中冲突的项目集（状态）用向前查看一个符号的办法进行处理，以解决冲突，即 LR(1)。

2. First 集构造算法：

使用如下规则，直至每一个非终结符的 FIRST 集合不再增大为止：

✧ 若 X 属于 VT，则 FIRST（X）= {X}

✧ 若 X 属于 VN，且有产生式 X -> a…，则把 a 加入 FIRST；若 S -> ε也是产生式，则把 ε 也加入到 FIRST（X）中

✧ 若 X -> Y……是一个产生式且 Y 属于 VN，则把 FIRST（Y）中所有的非空元素都加入到 FIRST（X）中，若 X -> $Y_1Y_2\cdots Y_k$ 是一个产生式，$Y_1,Y_2,\cdots Y_{i-1}$ 都是非终结符，而且对于任何满足 1 <= j <= i - 1，first（Y_j）都含有ε，则把 FIRST（Y_i）中所有的非 ε元素都加入到 FIRST（X）中，特别的，若所有 FIRST（Y_j）对于 j = 1，2，…, k 都含有 ε，则将 ε也加入到 FIRST(X)中。

3. 项目集闭包构造算法：

假定 I 是一个项目集，它的闭包 CLOSURE（I）可按如下方式构造：

✧ I 的任何项目都属于 CLOSURE（I）

✧ 若项目[A -> α. $B\beta$, a] 属于 CLOSURE（I），B -> γ是一个产生式，那么，对于 FIRST（βa）中的每个终结符 b，如果[B -> . γ, b]原来不在 CLOSURE（I）中，则把它加进去；

✧ 重复执行步骤 2）直至 CLOSURE（I）不再增大为止。 在实际代码编写中，我们主要采用广度优先搜索算法求解项目的闭包，在确定项目的展望字符时，需要用到之前计算的 FIRST 集。

4. 项目集族构造算法：

构造有效的 LR（1）项目集族的办法本质上和构造 LR（0）项目集规范族的办法是一样的。都需要两个函数 CLOSURE 和 GO，定义如下：

令 I 是一个项目集，X 是一个文法符号，函数 GO（I, X）定义为 GO（I, X）= CLOSURE（J），其中 J = {任何形如[A -> αB. β, a]的项目 | [A -> α. $B\beta$, a] ∈ I}。

关于文法 G 的 LR（1）项目集族的构造算法是：

```
BEGIN
   C := {CLOSURE([S->.S,#])};
   REPEAT:
         FOR C 中的每一个项目集 I 和 G 中的每个符号 X
               IF GO(I,X)非空且不属于 C，THEN
               把 GO(I,X)加入 C 中
   UNTIL C 不再增大
END
```

具体实现代码时，我们仍采用 BFS 算法求解项目集族，实际上最后构造的项目集转移图也是一个 DFA，我们采用记录图的方式来记录这个 DFA。

5. LR（1）分析表构造算法：

从文法的 LR（1）项目集族 C 构造分析表的算法如下：

假定 C = {I_0，I_1，…，I_n}，令每一个 I_k 的下标 k 为分析表的状态。令那个含有[S’ -> .S, #]的 I_k 的 k 作为分析器的初态。动作 ACTION 和状态转换 GOTO 可构造如下：

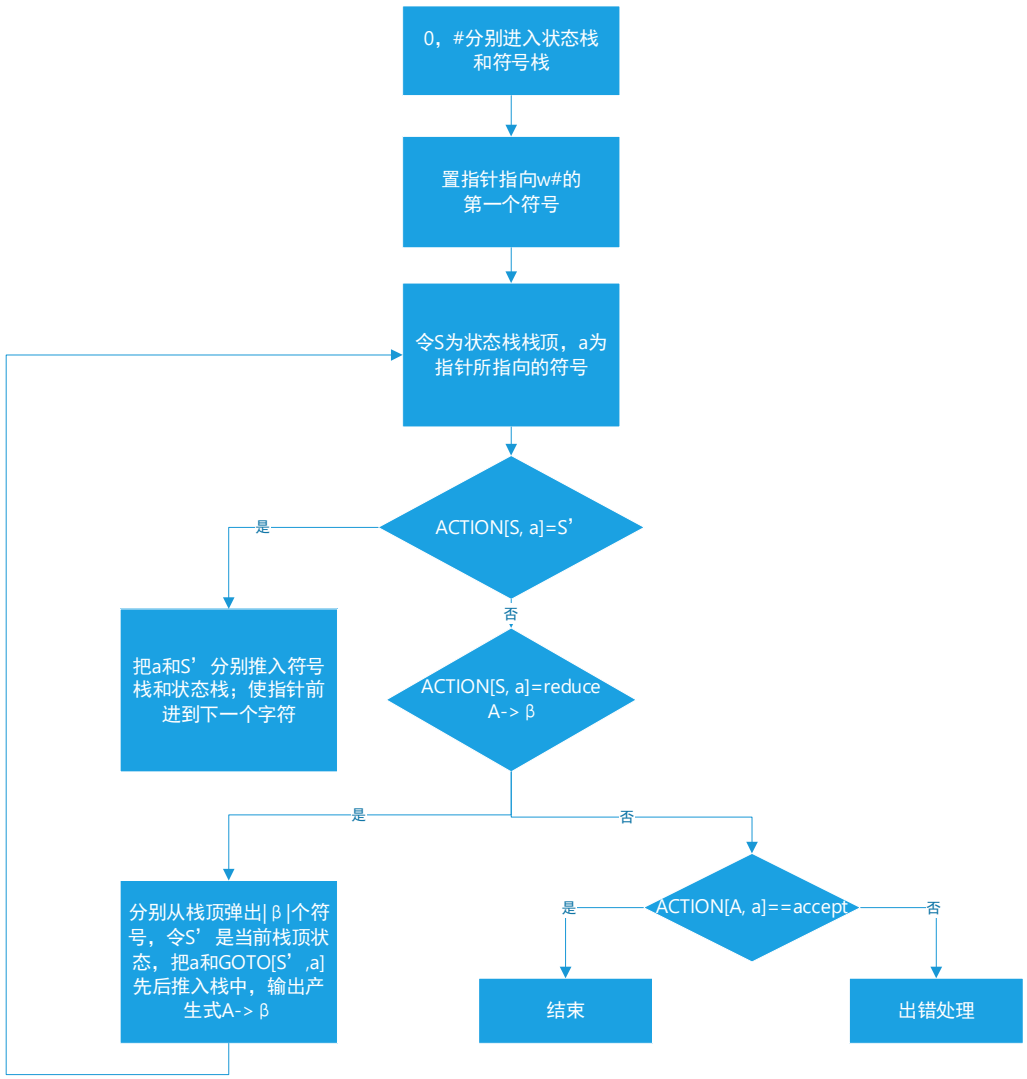
✧ 若项目[A -> α. $a\beta$, b]属于 I_k 且 GOTO（I_k, a）= I_j，a 为终结符，则置 ACTION[k, a]为“把状态 j 和符号 a 移进栈”，简记为“a_j”。

✧ 若项目[A -> α., a]属于 I_k，则置 ACTION[k, a]为“用产生式 A -> α”规约”，简记为“r_j”；其中假定 A -> α为文法 G’的第 j 个产生式。

✧ 若项目[S’ -> S., #]属于 Ik，则置 ACTION[k, #]为“接受”，简记为“acc”

✧ 若 GO（I_k, a）= I_j，则置 GOTO[k, A] = j。

6. LR（1）总控程序算法：



3.2.3 语法规则的设计

参考语法规则如下，进行类 C 语法规则的设计和读取，为接下来的语法分析工作做准备。

Program ::= <声明串>

<声明串> ::=<声明>{<声明>}

<声明> ::=int <ID> <声明类型> | void <ID> <函数声明>

<声明类型>::=<变量声明> | <函数声明> | <数组声明>

<变量声明> ::= ;

<函数声明> ::= ‘(’ <形参> ‘)’ <语句块>

<数组声明> ::= ‘[’ (数字)+ ‘]’ { ‘[’ (数字)+ ‘]’ }

<形参>::= <参数列表> | void

<参数列表> ::= <参数> {, <参数>}

<参数> ::= int <ID>

<语句块> ::= ‘{’<内部声明> <语句串> ‘}’

<内部声明> ::= 空 | <内部变量声明>;{<内部变量声明>;}

<内部变量声明>::=int <ID>

<语句串> ::= <语句>{ <语句> }

<语句> ::= <if 语句> |< while 语句> | <return 语句> | <赋值语句>

<赋值语句> ::= <ID> ‘=’<表达式>; | <数组> ‘=’<表达式>;

<return 语句> ::= return [<表达式>] ;（**注：[ ]中的项表示可选**）

<while 语句> ::= while ‘(’ <表达式> ‘)’ <语句块>

<if 语句> ::= if ‘(‘<表达式> ‘)’ <语句块> [else <语句块>]（**注：[ ]中的项表示可选**）

<表达式>::=<加法表达式>{ relop <加法表达式> } （**注：relop-> <|<=|>|>=|==|!=**）

<加法表达式> ::= <项> {+ <项> | -<项>}

<项> ::= <因子> {* <因子> | /<因子>}

<因子> ::=num | ‘(’<表达式> ‘)’ |<ID> FTYPE |<数组>

FTYPE ::= <call> | 空

<call> ::= ‘(’<实参> ‘) ’

<数组>::=<ID> ‘[’<表达式> ‘]’ |<数组> ‘[’<表达式> ‘]’

<实参> ::=<实参列表> | 空

<实参列表> ::= <表达式>{, <表达式>}

<ID>::=字母(字母|数字)*

为了方便后续语法规则读入与语义分析规则设计，将原始语法规则进行更改，更改后的结果如下所示：

产生式左部	**产生式右部**
program	declarationChain
declarationChain	declaration declarationChain
typeSpecifier	int
	void
declaration	typeSpecifier id ;
	completeFunction
	typeSpecifier id arrayDeclaration ;
arrayDeclaration	[num]
	[num] arrayDeclaration

completeFunction	declareFunction block
declareFunction	typeSpecifier id (formalParaList)
formalParaList	para
	para , formalParaList
	void
para	typeSpecifier id
block	{ statementChain }
statement	declaration
	ifStatement
	iterStatement
	returnStatement
	assignStatement
array	id [expression]
	array [expression]
assignStatement	id = expression ;
	array = expression ;
returnStatement	return expression ;
iterStatement	while (expression) block
ifStatement	if (expression) block
	if (expression) block else block
expression	primaryExpression
	primaryExpression operator expression
primaryExpression	num
	(expression) (expression)
	id (actualParaList)
	id
	array

operator	+ | - | * | / | < | <= | >= | > | == | !=
actualParaList	expression
	expression , actualParaList

3.2.4 LR(1)语法分析的设计

语法规则类 ItemSet：

重点函数或变量	功能与意义
Items	单个项目
Reserved	保留字
NonTerminalSymbols	变元
TerminalSymbols	终结符
OriginStartSymbol	原起始符
StartSymbol	广义起始符
calFirstSet	计算 First 集
scanLine	调用词法分析函数，扫描单行
generateTokens	调用词法分析函数，生成 Token
loadGrammer	读取文法

这一部分是对与语法的处理。众所周知，LR（1）语法中的单个项目 Item 是加点的，在读入所有的语法规则后，getDotItems 函数给所有产生式的所有位置加点，形成所有的单个项目。

然后，对于所有单个符号（包括终结符和非终结符）都有 First 集，终结符的 First 集就是自己，但非终结符的 First 集需要计算。因此在之后的语法分析步骤中，需要计算非终结符 First 集的函数 calNTFirstSet。

```
def calNTFirstSet(self, symbol):
    eps = {'class': 'T', 'name': '', 'type': self.Epsilon}
    hasEpsAllBefore = -1
    prods = [prod for prod in self.prods if prod.left == symbol]
    if len(prods) == 0:
        return

    is_add = 1
    while (is_add):
        is_add = 0
        for prod in prods:
            hasEpsAllBefore = 0

            for right in prod.right:
                # 2. 若X∈VN，且有产生式X->a…，a∈VT，则 a∈FIRST(X)
                #    X->ε,则ε∈FIRST(X)
                if right['class'] == 'T' or (right['type'] == self.Epsilon and
len(prod.right) == 1):  # A->epsilon
                    # 有就加
                    if right['type'] not in self.firstSet[symbol]:
                        self.firstSet[symbol].append(right['type'])
                        is_add = 1
                    break
```

```
                # 3. 对NT, 之前已算出来过, 但有可能是算到一半的
                if len(self.firstSet[right['type']]) == 0:
                    if right['type'] != symbol:  # 防止陷入死循环
                        self.calNTFirstSet(right['type'])

                # X->Y…是一个产生式且Y ∈VN  则把FIRST(Y)中的所有非空符号串ε元素都加入到FIRST(X)中。
                if self.Epsilon in self.firstSet[right['type']]:
                    hasEpsAllBefore = 1

                for f in self.firstSet[right['type']]:
                    if f != self.Epsilon and f not in self.firstSet[symbol]:
                        self.firstSet[symbol].append(f)
                        is_add = 1

            # 到这里说明整个产生式已遍历完毕 看是否有始终能推出eps
            # 中途不能退出eps的已经break了
            # 所有right(即Yi) 能够推导出ε,(i=1,2,…n)，则
            if hasEpsAllBefore == 1:
                if self.Epsilon not in self.firstSet[symbol]:
                    self.firstSet[symbol].append(self.Epsilon)
                    is_add = 1

return
```

语法规则类 ItemSetFamily：

重点函数或变量	功能与意义
itemSets	DFA 的状态
prods	项目集规范族的 DFA 的产生式
getLeftNT	获取某个非终结符的产生式
getLR1Closure	计算闭包
getFirstSet	获取字符串的 First 集
buildFamily	构建项目集规范族
GO	状态转移函数

通过前文中 LR1 的算法构建项目集族 DFA。

语法分析类 SyntacticAnalyzer：

重点函数或变量	功能与意义
ACTION[s, a]	当状态 s 面临输入符号 a 时，下一步采取的动作
GOTO[s, X]	当状态 s 面对文法符号 X 时，下一步转移的状态
isRecognizable	判断一个字符串是否能被识别
getTables	计算 ACTION 和 GOTO 数组
item2prodIdx	返回项目对应的产生式编号

重要函数是 getTables、isRecognizable，通过 getTables 函数可以生成 ACTION 表和 GOTO 表，进而通过总控程序 isRecognizable 对输入语句进行 LR（1）语法分析，在语法分析的同时进行语义分析生成中间代码。

```
    # 总控程序
    def isRecognizable(self, originCode):
        inputStr = []       # 输入串
        inputStr += self.lex.getTokensOfOneLine(originCode)
        sys.stdout.flush()
        stateStack = []     # 栈内状态序列
        shiftStr = []       # 移进规约串
        self.parseRst = [] # 记录步骤

        # 开始
        wallSymbol = {'class': 'T', 'type': '#'}
        shiftStr.append(wallSymbol)
        stateStack.append('s0')
        X = inputStr[0]     # X 为当前单词
        while (True):
            if len(inputStr) <= 2:
                tmpInputStr = self.lex.getTokensOfOneLine(originCode)
                if len(tmpInputStr) == 0:
                    inputStr.append(wallSymbol)
                else:
                    inputStr += tmpInputStr

            self.parseRst.append({'stateStack': copy.deepcopy(stateStack),
                                  'shiftStr': copy.deepcopy(shiftStr),
                                  'inputStr': copy.deepcopy(inputStr)})

            act = self.M[stateStack[-1]][X['type']].split(' ')[0]
            target = self.M[stateStack[-1]][X['type']].split(' ')[1] if
len(self.M[stateStack[-1]][X['type']].split(' ')) == 2 else None

            if act == 'shift':      # 移进操作
                stateStack.append(target)
                inputStr.pop(0)
                shiftStr.append(X)
                X = inputStr[0]
            elif act == 'goto':     # 转移操作
                stateStack.append(target)
                shiftStr.append(X)
                X = inputStr[0]
            elif act == 'reduce':   # 规约操作
                prodIdx = int(target)
                prod = self.prods[prodIdx]
                self.semantic.semanticAnalyze(prod, shiftStr)
                if False == self.semantic.semanticRst:
                    return False
                rightLen = len(prod.right)
                stateLen = len(stateStack)
                if rightLen == 1 and prod.right[0]['type'] == '$':
                    # 是空串，有问题
                    dst = self.M[stateStack[-1]][prod.left].split(' ')[1]
                    stateStack.append(dst)
                    shiftStr.append({'class': 'NT', 'type': prod.left})
                    X = inputStr[0]
                else:# 不是空串
```

```
                    stateStack = stateStack[0 : stateLen - rightLen]
                    shiftStr = shiftStr[0 : stateLen - rightLen]
                    X = {'class': 'NT', 'type': prod.left}
            elif act == 'acc':       # 接受操作
                print('语法、语义分析成功!')
                self.semantic.semanticAnalyze(self.prods[1], shiftStr)
                return True
            else:
                print('语法、语义分析出错!')
                self.syntacticRst = False
                sys.stdout.flush()
                self.syntacticErrMsg = "语法分析错误: " + str(X['row']) + "行" +
str(X['colum']) + "列"
    return False
```

3.3 语义分析及中间代码生成设计

3.3.1 语义分析功能及结构

紧接在词法分析和语法分析之后，编译程序要做的工作就是进行静态语义检查和翻译。静态语义检查通常包括：

(1) 类型检查。如果操作符作用于不相容的操作数，编译程序必须报告出错信息。

(2) 控制流检查。控制流语句必须使控制转移到合法的地方。在 C 语言中 break 语句使控制跳离包括该语句的最小 while、for 或 switch 语句。如果不存在包括它的这样的语句,则应报错。

(3) 名字的作用域分析也是静态语义分析的工作。

虽然源程序可以直接翻译为目标语言代码，但是许多编译程序却采用了独立于机器的复杂性介于源语言和机器语言之间的中间语言。这样做的好处是：

(1) 便于进行与机器无关的代码优化工作;

(2) 使编译程序改变目标机更容易;

(3) 使编译程序的结构在逻辑上更为简单明确。以中间语言为界面，编译前端和后端的接口更清晰。

静态语义分析和中间代码产生在编译程序中的地位如图所示。

3.3.2 语义分析重要原理

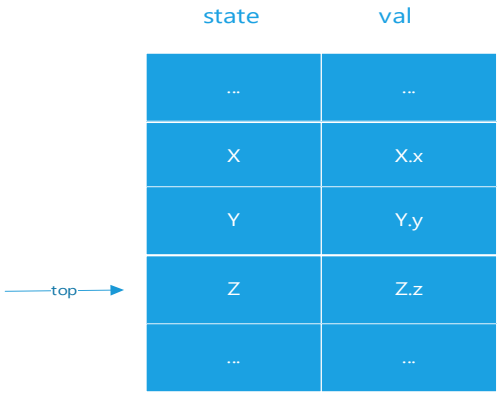
1. S-属性文法及自底向上扫描原理：

S-属性文法只含有综合属性。综合属性可以在分析输入符号串的同时由自下而上的分析器来计算。分析器可以保存与栈中文法符号有关的综合属性值，每当进行归约时，新的属性值就由栈中正在归约的产生式右边符号的属性值来计算。我们可以扩充分析器中的栈来存放这些综合属性值。

S-属性文法的翻译器通常可借助于 LR（1）分析器实现。在 S-属性文法的基础上，LR（1）分析器可以改造为一个翻译器，在对输人串进行语法分析的同时对属性进行计算。

我们讨论分析栈中的综合属性。在自底向上的分析方法中,我们使用一个栈来存放已经分析过的子树的信息。现在我们可以在分析栈中使用一个附加的域来存放综合属性值。下图表示的是一个带有一个属性值

空间的分析栈的例子。我们假设栈是由一对数组 state 和 val 来实现的。每一个 state 元素都是一个指向 LR 分析表的指针。



设当前的栈顶由指针 top 指示。我们假设综合属性是刚好在每次归约前计算的。假设语义规则 A.a := f(X.x, Y.y, Z.z)是对应于产生式 A→XYZ 的。在把 XYZ 归约成 A 以前，属性 Z.z 的值放在 val[top]中，Y.y 的值放在 val[top - 1] 中，X.x 的值放在 val[top - 2] 中。如果一个符号没有综合属性，那么数组 val 中相应的元素就不定义。归约以后，top 值减 2，A 的状态存放在 state[top]中(也就是 x 的位置)，综合属性 A.a 的值存放在 val[top]中。

在上面描述的实现中，代码段刚好在归约以前执行。归约提供了一个“挂钩”，便得代码段中的动作能够与之相联。也就是说，我们可以允许用户把一个语义动作与一个产生式联系起来，这个动作是当利用该产生式进行归约时要被执行的。

语义规则的计算可能产生代码、在符号表中存放信息，给出错误的信息或执行其他动作。对输入符号串的翻译就是根据语义规则进行计算的结果。S-属性文法是只含有综合属性的属性文法。而综合属性可以在分析输入符号串的同时由下而上的分析器计算。分析器可以保存预展中文法符号有关的综合属性值，每当进行归约时，新的属性值由栈中正在归约的产生式右边符号的属性值来计算。

2. 用综合属性代替继承属性：

有时，改变基础文法可能避免继承属性。例如，一个 Pasecal 的说明由一标识符序列后跟类型组成,如，m, n：integer。这样的说明的文法可由下面形式的产生式构成

D→L:T

T→integer | char

L→L, id | id

因为标识符由 L 产生而类型不在 L 的子树中，我们不能仅仅使用综合属性就把类型与标识符联系起来。事实上，如果非终结符 L 从第一个产生式中它的右边 T 中继承了类型，则我们得到的属性文法就不是 L-属性的，因此，基于这个属性文法的翻译工作不能在语法分析的同时进行。

一个解决的方法是重新构造文法，使类型作为标识符表的最后一个元素：

D→id L

L→, id L | :T

T→integer | char

这样，类型可以通过综合属性 L. type 进行传递，当通过 L 产生每个标识符时，它的类型就可以填入到符号表中。

3. 中间代码和四元式：

语义分析最终需要的结果是中间代码，而源程序的中间表示方法包括：后缀式，三地址代码（包括三元式，四元式，间接四元式），DAG 图表示。

三地址代码由下面一般形式的语句构成的语句序列：X := Y OP Z。其中 x，y，z 为名字，常数或编译时产生的临时变量；op 代表运算符号如定点运算符，浮点运算符，逻辑运算符等等。每个语句的右边只能有一个运算符。四元式属于三地址语句的一种，一个四元式通常是一个带有四个域的记录结构。这四个域通畅被称为 op， arg1， arg2 ，result。域 op 包含一个代表运算符的内部码，三地址语句 x:= y op z 可表示为：将 y 置于 arg1 域，z 置于 arg2 域，x 置于 result 域，:= 是运算符。

4. 符号表和函数表：

符号表内记录了编译过程中产生的关键信息，在符号表信息更新和代码生成过程中，语义分析需要检查代码语义信息的正确性。而语义分析和代码生成则需要从符号表内读取所需的信息，进行相关的语义检查和代码的翻译。代码生成阶段，产生的临时变量也需要保存到符号表。

根据设计的类 C 语言特性，变量、函数和字符串常量的信息是关键的符号信息。另外，由于允许在不同的作用域定义、使用相同的符号名，因此需要在变量符号内保存作用域的信息以区分同名的变量。符号表内最终需要记录的信息包含变量、函数、字符串常量和作用域信息等。

函数管理涉及函数对象的创建、将函数对象添加到函数表和从函数表取出函数对象。相比而言，函数对象的添加较为复杂，是因为需要考虑函数定义和函数声明的不同。获取函数对象时，除了提供函数名外，还需要提供实际参数列表，以方便符号表对函数参数类型进行检查。无论是函数定义还是函数声明，它们在文法级别具有公共的首部。

由于函数对象是直接插入函数表中的，因此使用函数名可以唯一确定函数对象。在语法分析中，访问函数对象的时机是在函数调用的时候，因此获取函数对象时还需要额外检查函数调用的实际参数类型是否与函数声明的形式参数类型匹配。主要需要实现以下三个功能：根据函数名获取局部变量表；根据函数名获取函数表；根据函数名从局部变量表中获取参数数量。

3.3.3 重点函数与重点变量

变量表类 Symbol：

重点函数或变量	功能与意义
name	符号的标识符
type	类型
size	占用字节数
offset	内存偏移量
place	对应的中间变量
function	所在函数

函数表类 FunctionSymbol：

重点函数或变量	功能与意义
name	函数的标识符

type	返回值类型
label	入口处的标签
params	形参列表

语义分析中间代码生成类 SemanticAnalyser：

重点函数或变量	功能与意义
sStack	语义分析栈
symbolTable	符号表
funcTable	函数表
curTempId	中间变量名序号
curLabelId	当前标号名序号
semanticAnalyze	产生式规约时进行语义分析，并生成对应中间代码

关于重点函数 semanticAnalyze，主要进行了语义规则的设计，由于代码较长，因此在下一小节将采用表格和语句的方式进一步解释。

3.3.4 语义规则的设计

这里仅选取几个重要的语义规则设计进行展示，其中有关过程、函数调用和数组的语义规则将在下几章进行专门介绍。

1. 变量声明语句：

语法规则：

typeSpecifier	int
	void
declaration	typeSpecifier id ;

对应的语义规则：

declaration.type = int;
declaration.size = 4;

主要流程：

- 使用语义规则获取变量的类型，大小。
- 检查符号表中是否存在同名的变量。
- 若不存在，创建一个新的数据项，保存该变量的名称，类型大小。

2. 赋值语句：

语法规则：

assignStatement	id = expression ;
	array = expression ;
expression	primaryExpression
	primaryExpression operator expression
primaryExpression	num
	(expression) (expression)
	id (actualParaList)
	id
	array

主要流程：

- 检查表达式左端变量是否在符号表中，不存在则报错。
- 判断变量是常数还是标识符。
- 进行中间代码生成。
  - ✓ 找到赋值语句左端的变量 s
  - ✓ 新建一个中间变量 n
  - ✓ 基于表达式 expression，得到对 n 赋值的中间代码。
  - ✓ 将计算得到的 n 的值赋给 s。

3.条件语句：

语法规则：

ifStatement	if (expression) block
	if (expression) block else block

主要流程：

if (expression) block

- 建立真值跳转地址和结束地址
- 获得 block 在状态栈中的内容和 expression 在状态栈中的内容
- 计算条件表达式，生成相应的中间代码
- 判断是否表达式是否为真，若为真，设置真值跳转地址，并跳转。若为假，跳转到结束地址
- 生成语句中间代码，结束后，设置结束地址

if (expression) block else block

- 建立真值地址，假地址，结束地址
- 获取假植表达式，真值表达式，条件表达式

- 计算条件表达式
- 和没有 else 语句，类似，根据计算结果进行跳转
- 在真值表达式后加入跳转到结束地址的指令

4. 循环语句：

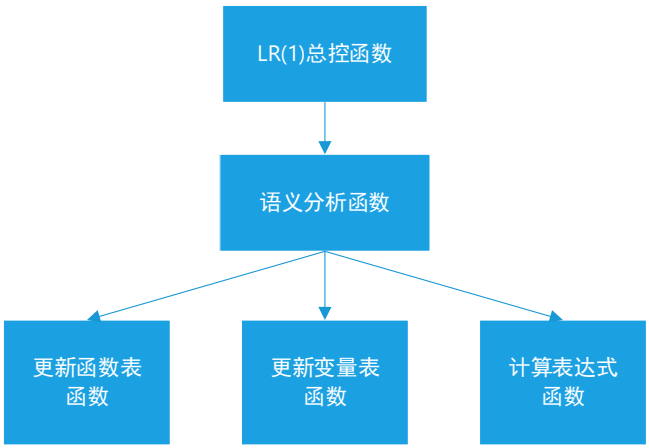
语法规则：

iterStatement	while (expression) block

主要流程：

- 建立四个分支入口：true、false、begin、end
- 获取循环内容状态
- 获取条件表达式
- 跳转到 begin 地址
- 计算条件表达式并给出中间代码
- 若表达式为真，跳转到 true 地址
- 为假，跳转到 false 地址
- 生成循环内容的中间代码。注意，break 则跳转到 false 地址，continue 则跳转到 true 地址
- 最后，跳转到 begin 地址。设置 false 地址

3.3.5 函数调用图



3.4 目标代码生成设计

3.4.1 目标代码生成功能及结构

编译模型的最后一个阶段是代码生成。它以源程序的中间代码作为输入，并产生等价的目标程序作为输出，如下图所示。

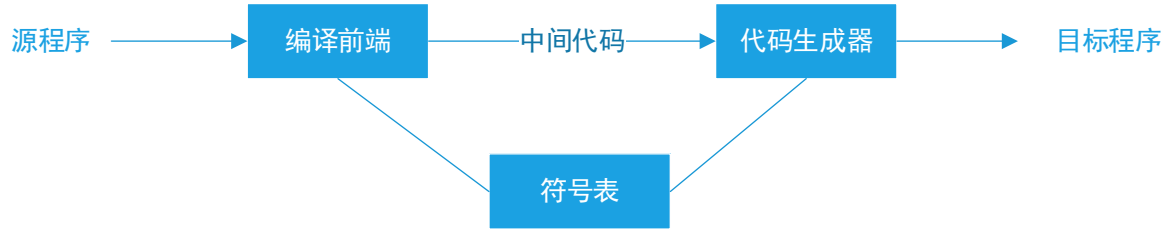
代码生成器的输人包括中间代码和符号表中的信息。代码生成是把语义分析后或优化后的中间代码变换成目标代码。目标代码一般有以下三种形式。

(1) 能够立即执行的机器语言代码，所有地址均已定位(代真)。

(2) 待装配的机器语言模块。当需要执行时，由连接装入程序把它们和某些运行程序连接起来，转换成能执行的机器语言代码。

(3) 汇编语言代码，尚需经过汇编程序汇编，转换成可执行的机器语言代码。（本次课设采用此种方法）

代码生成要着重考虑两个问题： 一是如何使生成的目标代码较短；另一是如何充分利用计算机的寄存器，减少目标代码中访问存储单元的次数。这两个问题都直接影响目标代码的执行速度。



3.4.2 目标代码生成重要原理

1. 代码块的基本生成算法：

为简单起见，假设基本块中每个中间代码形为 A := B op C。如果基本块中含有其它形式的中间代码，也不难仿照下述算法写出对应的算法。基本块的代码生成算法如下。

对每个中间代码 i: A := B op C，依次执行下述步骤。

(1) 以中间代码 i: A := B op C 为参数，调用函数过程 GETREG(i: A := B op C)。当从 CETREG 返回时，我们得到一个寄存器 R，它将用作存放 A 现行值的寄存器。

(2)利用地址描述数组 AVALUE[B]和 AVALUE[C]，确定出变量 B 和 C 现行值的存放位置 B'和 C'。如果其现行值在寄存器中，则把寄存器取作 B'和 C'。

(3) 如果 B' ≠ R,则生成目标代码:

```
LD R, B’
op R, C’
```

否则生成目标代码 op R, C ;如果 B’或 C’为 R，则删除 AVALUE[B]或 AVALUE[C]中的 R。

(4)令 AVAIUE[A]= {R} ,并令 RVAUE[R]= {A} ,以表示变量 A 的现行值只在 R 中并且 R 中的值只代表 A 的现行值。

(5)如果 B 和 C 的现行值在基本块中不再被引用,它们也不是基本块出口之后的活跃变量(由该中间代码 i 上的附加信息知道)，并且其现行值在某寄存器 R_k 中,则剥除 RVAIUE[R_k]中的 B 或 C 以及 AVALUE[B]中的 R_k ,使该寄存器不再为 B 或 C 所占用。

CETREG 是一个函数过程，GETREG(i:A : = B op C)给出一个用来存放 A 的当前值的寄存器 R，其中要用到中间代码 i 上的待用信息，GETREC 的算法如下。

(1)如果 B 的现行值在某寄存器 R_i 中，RVALUE[R_i]只包含 B，此外，或者 B 与 A 是同一标识符，或者 B 的现行值在执行中间代码 A : = B op C 之后不会再引用(此时,该中间代码 i 的附加信息中，B 的待用信息和活跃信息分别为“非待用”和“非活跃”)，则选取 R_i 为所需的寄存器 R，并转 4。

(2)如果有尚未分配的寄存器，则从中选取一个 R_i，为所需的寄存器 R，并转 4。

(3)从已分配的寄存器中选取一个 R_i 为所需的寄存器 R。最好使 R_i 满足以下条件:

占用 R_i 的变量的值，也同时存放在该变量的主存单元中，或者在基本块中要在最远的将来才会引用到或不会引用到(关于这一点可从有关中间代码 i 上的待用信息得知)。

对 RVALUE[R_i]中每一变量 M，如果 M 不是 A，或者如果 M 是 A 又是 C，但不是 B 并且 B 也不在 RVALUE[R_i]中，则

(1)如果 AVALUE[M]不包含 M，则生成目标代码 ST R_i，M;

(2)如果 M 是 B，或者 M 是 C 但同时 B 也在 RVALUE[R_i]中，则令 AVALUE[M] = {M，R} ,否则令 AVALUE[M] = {M};

(3)删除 RVALUE[R_i]中的 M。.

(4)给出 R_i 返回。

3.4.3 重点函数与重点变量

目标代码生成类 ObjectCodeGenerator：

重点函数或变量	功能与意义
mipsCode	存储目标代码
regTable	记录寄存器内部存的是哪个变量的值
varStatus	记录变量是在寄存器中还是内存中
getRegister	获取寄存器
freeRegister	释放寄存器
genMips	生成目标代码

重点函数中获取寄存器 getRegister 实现思路和 3.4.2 小节中的代码块的基本算法实现原理类似，主要思路为：1. 若该变量已经存在在寄存器中，返回相应寄存器。2. 若不然，按序找到空的寄存器，返回该寄存器。3. 若没有空寄存器，根据释放寄存器算法释放一个寄存器并范回。关于释放寄存器算法，优先释放之后不会使用的寄存器和之后使用最少最远使用的寄存器。

3.4.4 四元式转汇编语言规则

四元式	汇编语言
:=, {}, _,{}	add {},$zero,{}
{}, :, _, _	{}:
store, _, {}, {}	add $a0,$zero,{} sw {}, {}($sp)
load, _, {}, {}	lw {}, {}($sp)
j, _, _, {}	j {}
j>, {}, 0, {}	bgt {},$zero,{}
+, {}, {}, {}	add {},{},{}

-, {}, {}, {}	sub {},{},{}
*, {}, {}, {}	add $a1,$zero,{} mul {},{},{}
>, {}, {},{}	add $a1,$zero,{} sgt {},{},{}

3.5 函数调用的中间代码生成和目标代码生成设计

3.5.1 语法规则

declaration	completeFunction
completeFunction	declareFunction block
declareFunction	typeSpecifier id (formalParaList)
formalParaList	para , formalParaList
primaryExpression	id (actualParaList)

3.5.2 中间代码生成

函数声明时，检查函数表中是否存在同名的函数，如果存在报错；利用函数名称和参数列表更新函数表。在规约参数列表语句时，将参数存放到到 formalParaList.stack 中；函数调用时，从 actualParaList 的 stack 属性中读出输入参数，在函数表中寻找名字是 id 的函数，比对参数是否符合，基于传值的方法，将参数存放到属性表中，写明变量所在的函数。

3.5.3 目标代码生成

四元式	汇编语言
call	jal {}
push	sw $ra, {}($fp)
pop	lw $ra, {}($fp)
return	jr $ra

3.6 数组的语法分析、中间代码生成和目标代码生成设计

3.6.1 语法规则

数组相关语句主要结构如下：

declaration	typeSpecifier id arrayDeclaration ;
arrayDeclaration	[num]
	[num] arrayDeclaration
array	id [expression]
	array [expression]

3.6.2 中间代码生成

数组定义时和变量定义类似，需要先查变量表检查是否有重复，若没有，则定义变量类型为 array，获得当前数组维数，插入变量表；对于数组的访问，首先，计算数组偏移地址表达式的值，然后获取数组变量基址名，最终获取数组当前访问位置。

3.6.3 目标代码生成

四元式	汇编语言

[]=	la $v1, {}
	mul {}, {}, 4
	addu {}, {}, $v1
	sw {}

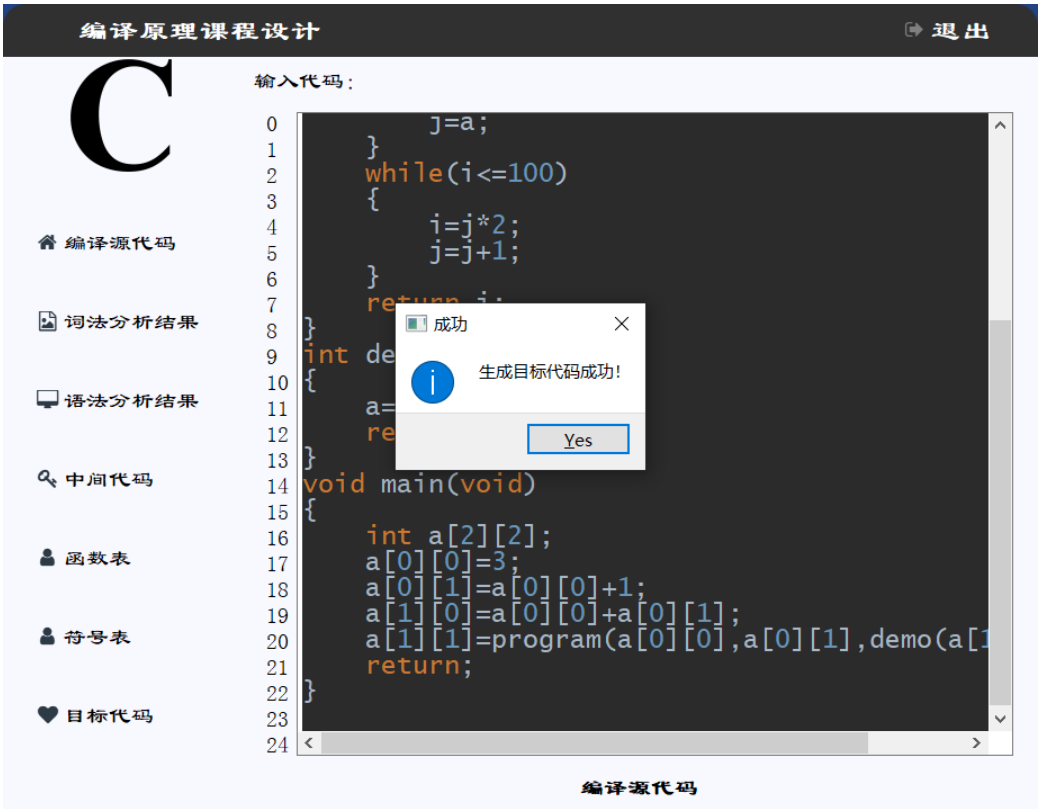
=[]	la $v1, {}
	mul {}, {}, 4
	addu {}, {}, $v1
	lw {}

四、 调试分析

4.1 正确用例

```
int program(int a,int b,int c)
{
    int i;
    int j;
    i=0;
    if(a>(b+c))
    {
        j=a+(b*c+1);
    }
    else
    {
        j=a;
    }
    while(i<=100)
```

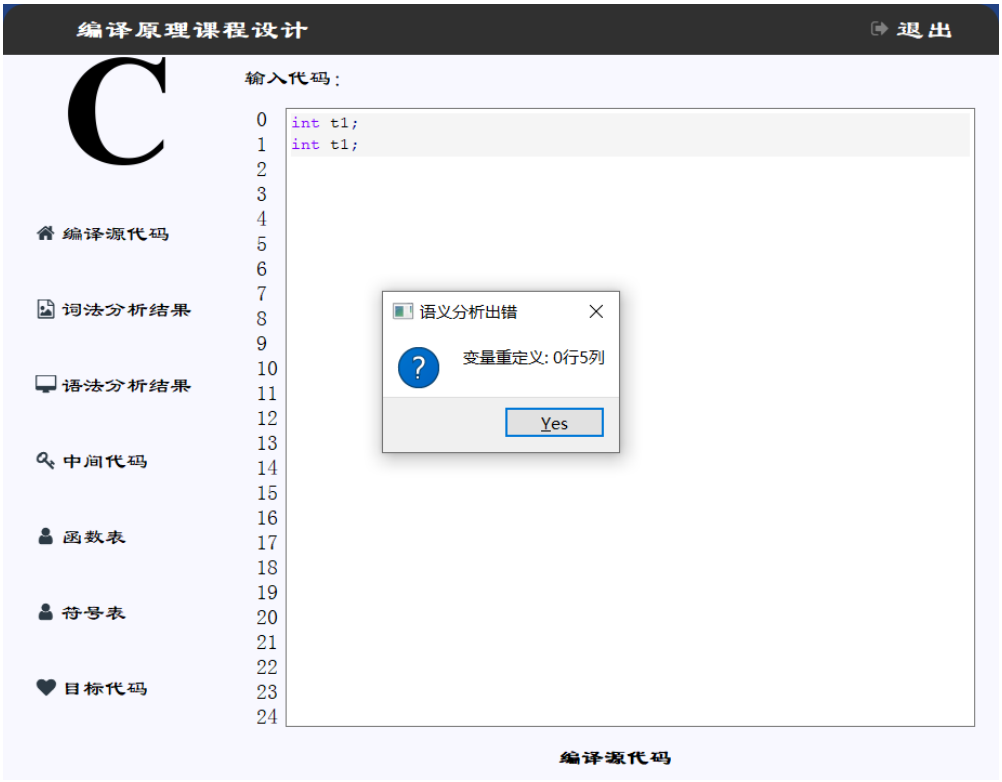
```
    {
        i=j*2;
    }
    return i;
}
int demo(int a)
{
    a=a+2;
    return a*2;
}
void main(void)
{
    int a[2][2];
    a[0][0]=3;
    a[0][1]=a[0][0]+1;
    a[1][0]=a[0][0]+a[0][1];
    a[1][1]=program(a[0][0],a[0][1],demo(a[1][0]));
    return;
}
```



4.2 错误用例

变量重定义

```
int t1;
int t1;
```



使用未声明的变量

```
int main()
{
   t2 = 1;
   return 0;
}
```

使用未定义的函数

```
int main()
{
   int t3 = demo3(3 * 9);
   return 0;
}
```

变量赋值时类型错误

```
int main()
{
   int t4 = 1;
   void t5 = t4;
   return 0;
}
```

函数形参和实参不匹配

```
int demo()
{
    return 16;
}
int main()
{
   int t6 = demo(6 * 9);
   return 0;
}
```

4.3 问题思考

关于报错处理，各个地方需要注意的细节都比较多，实现起来比较复杂，并且后期修改起来也比较麻烦，没有形成系统的报错处理机制，这个是需要后期进一步打磨的，这种系统能力设计能力还是需要注意平时培养。

五、用户使用说明

5.1 开发环境

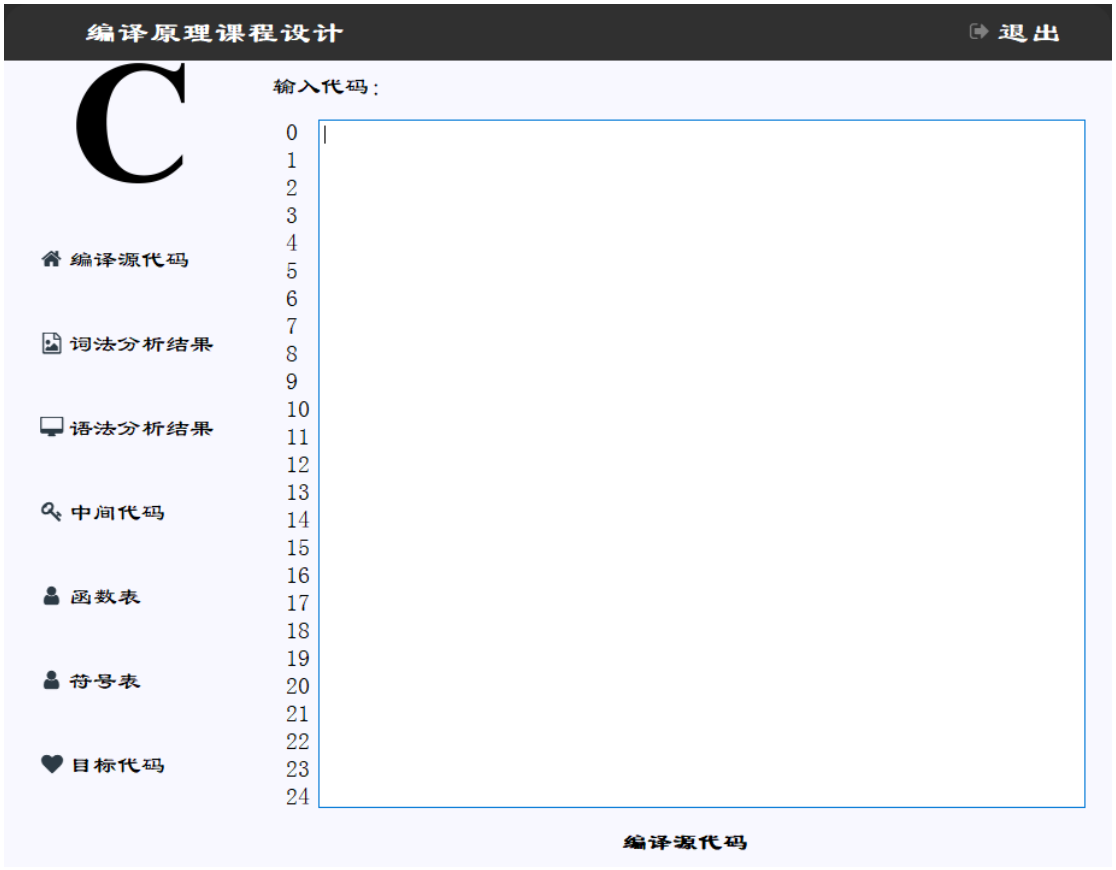
编程语言：Python 3.8

使用工具：Pycharm Community Editor

操作系统：Windows 10 家庭版

5.2 使用说明

5.2.1 初始界面：（注意一定要输入代码，否则空字符串会报错）


编译原理课程设计
退出
输入代码：
编译源代码
词法分析结果
语法分析结果
中间代码
函数表
符号表
目标代码
编译源代码

5.2.2 编译源代码

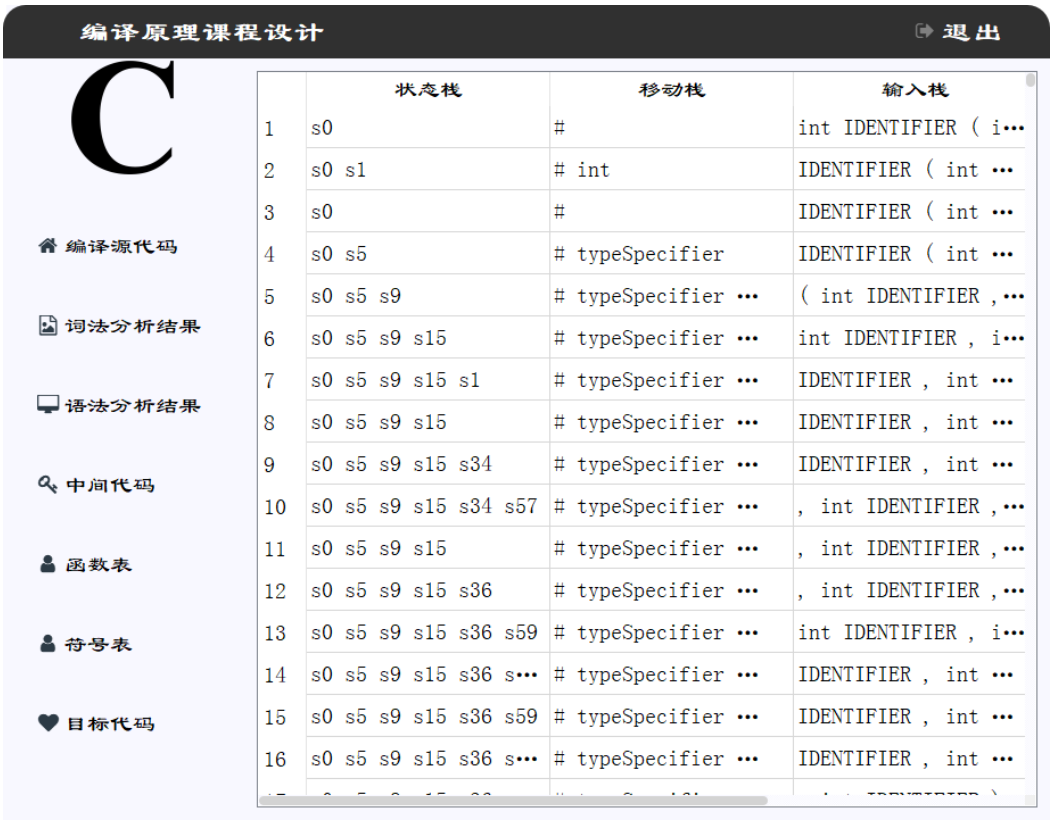

编译原理课程设计
退出
输入代码：
j=a;
}
while(i<=100)
{
i=j*2;
j=j+1;
}
return i;
}
int demo(int a)
{
a=a+2;
return a*2;
}
void main(void)
{
int a[2][2];
a[0][0]=3;
a[0][1]=a[0][0]+1;
a[1][0]=a[0][0]+a[0][1];
a[1][1]=program(a[0][0],a[0][1],demo(a[1
return;
}
编译源代码

5.2.3 结果查看

词法分析结果：

	类型	值	行	列
1	int	int	0	1
2	IDENTIFIER	program	0	5
3	(	(	0	12
4	int	int	0	13
5	IDENTIFIER	a	0	17
6	,	,	0	18
7	int	int	0	19
8	IDENTIFIER	b	0	23
9	,	,	0	24
10	int	int	0	25
11	IDENTIFIER	c	0	29
12	)	)	0	30
13	{	{	1	1
14	int	int	2	1
15	IDENTIFIER	i	2	5
16	;	;	2	6

语法分析结果：



	状态栈	移动栈	输入栈
1	s0	#	int IDENTIFIER (i…
2	s0 s1	# int	IDENTIFIER (int …
3	s0	#	IDENTIFIER (int …
4	s0 s5	# typeSpecifier	IDENTIFIER (int …
5	s0 s5 s9	# typeSpecifier …	(int IDENTIFIER ,…
6	s0 s5 s9 s15	# typeSpecifier …	int IDENTIFIER , i…
7	s0 s5 s9 s15 s1	# typeSpecifier …	IDENTIFIER , int …
8	s0 s5 s9 s15	# typeSpecifier …	IDENTIFIER , int …
9	s0 s5 s9 s15 s34	# typeSpecifier …	IDENTIFIER , int …
10	s0 s5 s9 s15 s34 s57	# typeSpecifier …	, int IDENTIFIER ,…
11	s0 s5 s9 s15	# typeSpecifier …	, int IDENTIFIER ,…
12	s0 s5 s9 s15 s36	# typeSpecifier …	, int IDENTIFIER ,…
13	s0 s5 s9 s15 s36 s59	# typeSpecifier …	int IDENTIFIER , i…
14	s0 s5 s9 s15 s36 s…	# typeSpecifier …	IDENTIFIER , int …
15	s0 s5 s9 s15 s36 s59	# typeSpecifier …	IDENTIFIER , int …
16	s0 s5 s9 s15 s36 s…	# typeSpecifier …	IDENTIFIER , int …

中间代码结果：

	operation	arg1	arg2	result
1	f1	:	_	_
2	pop	_		t1
3	pop	_		t2
4	pop	_		t3
5	-	fp		fp
6	:=	0	_	t4
7	+	t2	t3	t6
8	>	t1	t6	t10
9	j>	t10	0	l1
10	j	_	_	l2
11	l1	:	_	_
12	*	t2	t3	t7
13	+	t7	1	t8
14	+	t1	t8	t9
15	:=	t9	_	t5
16	j	_	_	l3
17	l2	:		

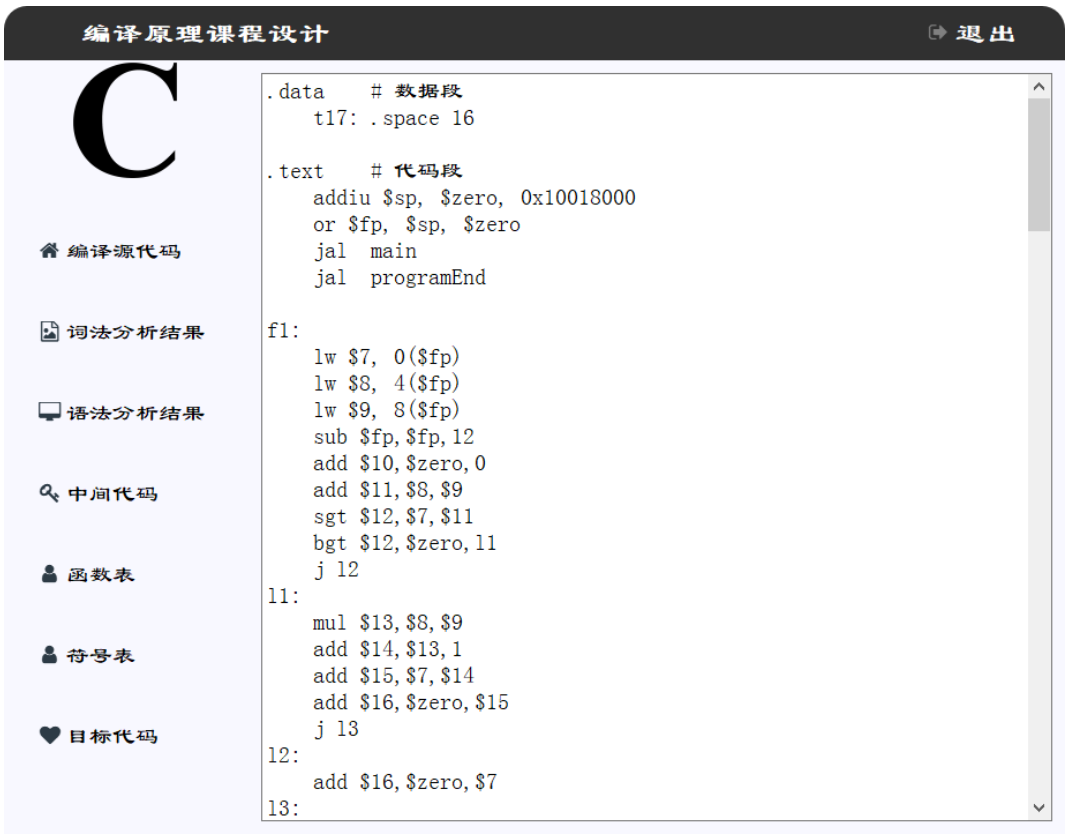
函数表：

	函数的标识符	返回值类型	入口处的标签	形参列表
1	global		global	[]
2	program	int	f1	[('a', 'int',...
3	demo	int	f2	[('a', 'int',...
4	main	void	main	[]

符号表：

	符号的标识符	类型	占用字节数	内存偏移量	对应的中间变量	所
1	a	int	4	0	t1	f1
2	b	int	4	4	t2	f1
3	c	int	4	8	t3	f1
4	i	int	4	12	t4	f1
5	j	int	4	16	t5	f1
6	a	int	4	20	t14	f2
7	a	int array	16	24	t17	mair

目标代码查看：




```
.data    # 数据段
    t17: .space 16

.text    # 代码段
    addiu $sp, $zero, 0x10018000
    or $fp, $sp, $zero
    jal  main
    jal  programEnd

f1:
    lw $7, 0($fp)
    lw $8, 4($fp)
    lw $9, 8($fp)
    sub $fp,$fp,12
    add $10,$zero,0
    add $11,$8,$9
    sgt $12,$7,$11
    bgt $12,$zero,l1
    j l2
l1:
    mul $13,$8,$9
    add $14,$13,1
    add $15,$7,$14
    add $16,$zero,$15
    j l3
l2:
    add $16,$zero,$7
l3:
```

六、课程设计总结

6.1 课程认识

通过本次大作业的编程实践，我将编译原理课堂上较为抽象的理论知识转化为实际工程上的应用，加深了对课程知识的理解，并在实际编程的过程中建立了不同编译原理知识间的联系，将零碎的知识进行了有机整合并进行应用。

在通过编程实现类 C 语言的整个编译过程中，我了解了高级编程语言程序在一台计算机上进行程序内容识别和语言文法正误检查的流程、机制，这让我更好地理解了程序设计语言深层的设计理念和设计思想，对于日后的计算机工程实践是一份宝贵的经验。

6.2 心得体会

在上学期 LR(1) 语法分析器设计之后，这学期又迎来了编译器设计过程中重要的语义分析器、目标代码生成器及数组的设计部分，相较于上学期的语法分析器，这学期的工作量明显提高了不少。

在整个实现过程中，我遇到了这样或那样的问题，例如把一些看似自然的分析过程写成一个具体的流程是比较棘手的。人工分析时，一些显然的东西有时会被自然的忽略，但对于程序设计来说都是不可或缺的。编译原理同时也是一门理论性较强的课程，其中的文法、语言到 LR(1) 文法等概念分析，基本上都是对具体问题的抽象，是需要很多时间去理解、掌握的。在真正完成类 C 语言编译器的设计后，我对所写的程序以及程序语言都有更深的本质认识，这样的认识会让我站在更高的层面去理解计算机程序，并且知道编译的过程方法理论。学习编译原理并不仅仅能够写编译器，还可以在许多其他领域进行应用，这种潜在价值是目前所无法度量的。

这段时间，我过得十分充实与紧张，虽然类 C 语言编译器的设计工作量大且比较复杂，但我也拿出自己的努力与时间，获得了一个具有演示功能的小型类 C 语言编译器。在实现类 C 语言编译器的过程中，我深入理解了类 C 语言编译器的核心工作原理，更充分地掌握了编译原理这一门重要的计算机基础学科。

七、参考文献

[1] 陈火旺,程序设计语言编译原理(第 3 版)[M].北京:国防工业出版社,1992

[2]编译器构造[M]. 机械工业出版社, (美)CharlesN.Fischer,(美)RichardJ.LeBlanc,Jr. 著, 2005

[3]C 程序设计语言[M]. 机械工业出版社 , (美)BrianW.Kernighan,(美)DennisM.Ritchie 著, 2004

[4]数据结构[M]. 清华大学出版社 , 严蔚敏,吴伟民编著, 2002

[5]王佳林,张美玲,高涵.浅析编译原理中编译工作的基本流程及其实现[J].中国新通信,2019,21(24):150.