编译技术课程设计设计文档

一、项目背景及文法

本项目是2020年北航计算机学院编译原理课程的课程设计作业，能够分析符合给定文法要求的程序，并将其编译为可运行的MIPS代码。

项目文法类似c0文法，详细信息如下：

```
<加法运算符> ::= +|-
<乘法运算符>  ::= *|/
<关系运算符>  ::=  <|<=|>|>=|!=|==
<字母>   ::= _|a|...|z|A|...|Z
<数字>   ::= 0|1|...|9
<字符>    ::=  '<加法运算符>'|'<乘法运算符>'|'<字母>'|'<数字>'
<字符串>   ::=  "{十进制编码为32,33,35-126的ASCII字符}"
<程序>    ::= [<常量说明>][<变量说明>]{<有返回值函数定义>|<无返回值函数定义>}<主函数>
<常量说明> ::=  const<常量定义>;{ const<常量定义>;}
<常量定义>   ::=   int<标识符>=<整数>{,<标识符>=<整数>}
                 | char<标识符>=<字符>{,<标识符>=<字符>}
<无符号整数>  ::= <数字>{<数字>}
<整数>        ::= [+|-]<无符号整数>
<标识符>    ::=  <字母>{<字母>|<数字>}
<声明头部>   ::=  int<标识符> |char<标识符>
<常量>   ::=  <整数>|<字符>
<变量说明>  ::= <变量定义>;{<变量定义>;}
<变量定义> ::= <变量定义无初始化>|<变量定义及初始化>
<变量定义无初始化>  ::= <类型标识符>(<标识符>|<标识符>'['<无符号整数>']'|<标识符>'['<无符号整数>']''['<无符号整数>']'){,(<标识符>|<标识符>'['<无符号整数>']'|<标识符>'['<无符号整数>']''['<无符号整数>']' )}
<变量定义及初始化>  ::= <类型标识符><标识符>=<常量>|<类型标识符><标识符>'['<无符号整数>']'='{'<常量>{,<常量>}'}'|<类型标识符><标识符>'['<无符号整数>']''['<无符号整数>']'='{''{'<常量>{,<常量>}'}'{, '{'<常量>{,<常量>}'}'}'}'
<类型标识符>      ::=  int | char
<有返回值函数定义>  ::=  <声明头部>'('<参数表>')' '{'<复合语句>'}'
<无返回值函数定义>  ::= void<标识符>'('<参数表>')''{'<复合语句>'}'
<复合语句>   ::=  [<常量说明>][<变量说明>]<语句列>

<参数表>    ::=  <类型标识符><标识符>{,<类型标识符><标识符>}| <空>
```

```
<主函数>     ::= void main‘(’‘)’ ‘{’<复合语句>‘}’
<表达式>     ::= [＋|－]<项>{<加法运算符><项>}   //[+|-]只作用于第一个<项>
<项>       ::= <因子>{<乘法运算符><因子>}
<因子>      ::= <标识符>|<标识符>'['<表达式>']'|<标识符>'['<表达式>']''['<表达式>']'|'('<表达式>')'|<整数>|<字符>|<有返回值函数调用语句>
<语句>      ::= <循环语句>|<条件语句>| <有返回值函数调用语句>;  |<无返回值函数调用语句>;|<赋值语句>;|<读语句>;|<写语句>;|<情况语句>|<空>;|<返回语句>; | '{'<语句列>'}'
<赋值语句>    ::=  <标识符>=<表达式>|<标识符>'['<表达式>']'=<表达式>|<标识符>'['<表达式>']''['<表达式>']' =<表达式>

<条件语句>   ::= if '('<条件>')'<语句>[else<语句>]
<条件>      ::=  <表达式><关系运算符><表达式>

<循环语句>    ::=  while '('<条件>')'<语句>| for'('<标识符>=<表达式>;<条件>;<标识符>=<标识符>(+|-)<步长>')'<语句>

<步长>::= <无符号整数>
<情况语句>   ::=  switch ‘(’<表达式>‘)’ ‘{’<情况表><缺省>‘}’
<情况表>    ::=  <情况子语句>{<情况子语句>}
<情况子语句>   ::=  case<常量>：<语句>

<缺省>    ::=  default :<语句>

<有返回值函数调用语句> ::= <标识符>'('<值参数表>')'
<无返回值函数调用语句> ::= <标识符>'('<值参数表>')'
<值参数表>    ::= <表达式>{,<表达式>}|<空>

<语句列>    ::= {<语句>}
<读语句>     ::=  scanf '('<标识符>')'
<写语句>     ::= printf '(' <字符串>,<表达式> ')'| printf '('<字符串> ')'| printf '('<表达式>')'
<返回语句>    ::=  return['('<表达式>')']
```

项目编译成MIPS代码的功能实现分为如下几个阶段，词法分析，语法语义分析，中间代码生成，中间代码优化，目标代码生成这五个主要阶段。下面将对各个阶段的实现细节作进一步介绍。

二、词法分析

2.1 词法

2.1.1 符号类型

根据项目文法，可以为程序中可能出现所有符号进行词法规则的生成，可能出现的符号有以下几种：

标识符，整型常量，字符常量，字符串，保留字，单字符分界符，双字符分界符。

2.1.2 词法生成规则

```
<标识符>  ::=  <字母>{<字母>|<数字>}
<整形常量> ::= <数字>{<数字>}
<字符常量>  ::= '<加法运算符>'|'<乘法运算符>'|'<字母>'|'<数字>
<字符串>  ::= "{十进制编码为32,33,35-126的ASCII字符}"
 < 保留字 >  ::= const  |  int  |  char  |  void  |  main  |  if  |
 else  |  switch  |  case  |  default  |  while  |  for |  scanf  |
 printf  |  return
< 单字符分界符 > ::= '+'  |  '-'  |  '*'  |  '/'  |  '<'  |  '>'  |
 ':'  |  ';'  |  '='  |  ','  |  '('  |  ')'  |  '['  |  ']'  |
 '{'  |  '}'
< 双字符分界符 > ::= "<="  |  '>='  |  '=='  |  '!='
```

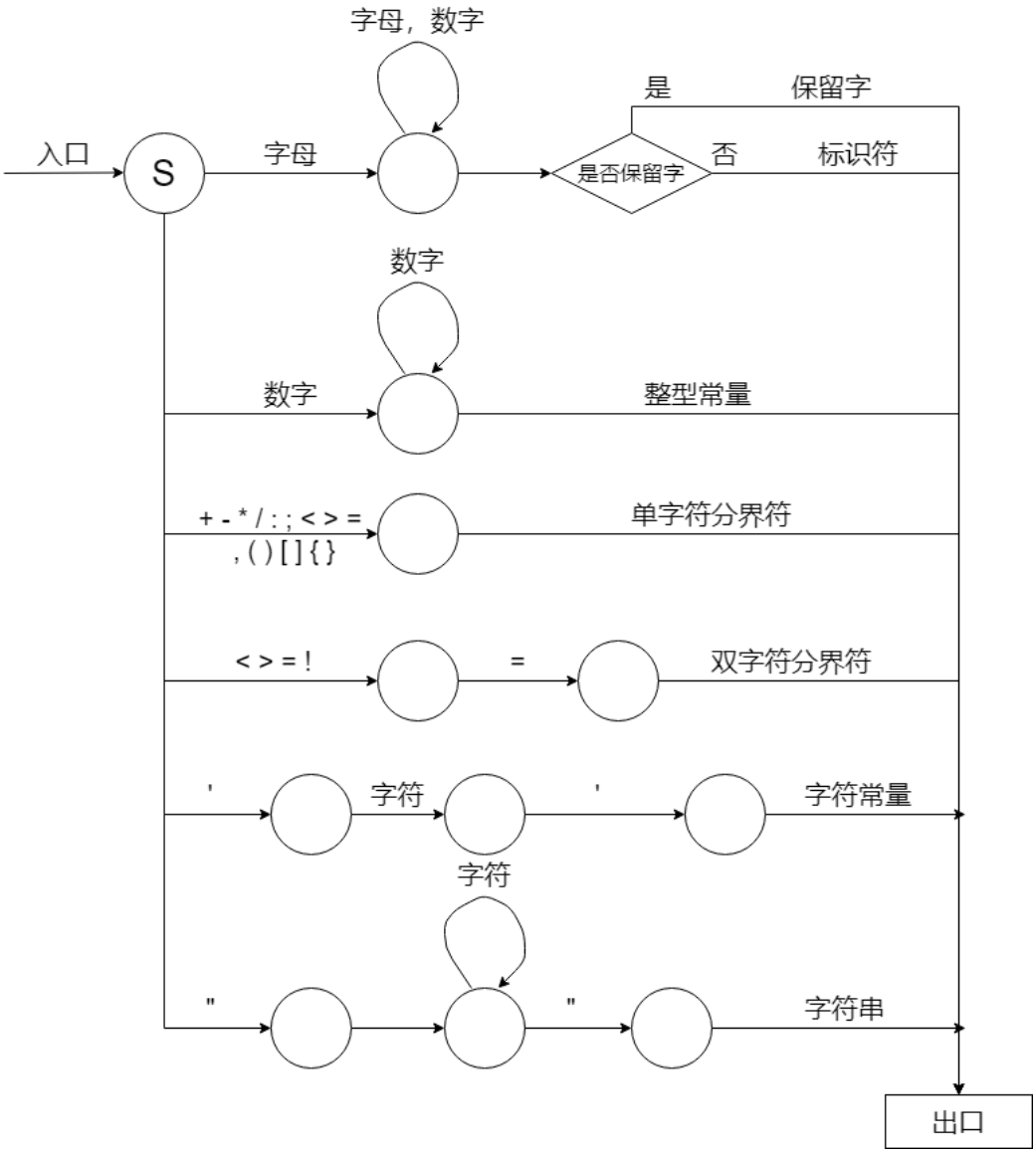
2.1.3 符号类别码

对程序中每一个符号，都需要一个唯一的类别码对其进行标记，以方便后续阶段的分析和处理。

单词名称	类别码	单词名称	类别码	单词名称	类别码
标识符	IDENFR	default	DEFAULTTK	>=	GEQ
整型常量	INTCON	while	WHILETK	==	EQL
字符常量	CHARCON	for	FORTK	!=	NEQ
字符串	STRCON	scanf	SCANFTK	:	COLON
const	CONSTTK	printf	PRINTFTK	=	ASSIGN
int	INTTK	return	RETURNTK	;	SEMICN
char	CHARTK	+	PLUS	,	COMMA
void	VOIDTK	-	MINU	(	LPARENT
main	MAINTK	*	MULT	)	RPARENT
if	IFTK	/	DIV	[	LBRACK
else	ELSETK	<	LSS	]	RBRACK
switch	SWITCHTK	<=	LEQ	{	LBRACE
case	CASETK	>	GRE	}	RBRACE

2.2 词法分析流程

利用自动机来对程序进行处理，从程序中逐个读入字符，并进行相应的状态转移，对符合词法的符号进行输出，不符合文法的符号进行错误处理。



2.3 函数定义

词法分析阶段是整个编译过程的第一阶段，在本项目中，词法分析和语法分析同时进行，因此要预留接口以供后续阶段的使用：

```
int get_symbol();//从文件中读取一个符号，将符号保存到token中，将类型保存到curr_symbol中，若成功则返回0，否则返回负值
```

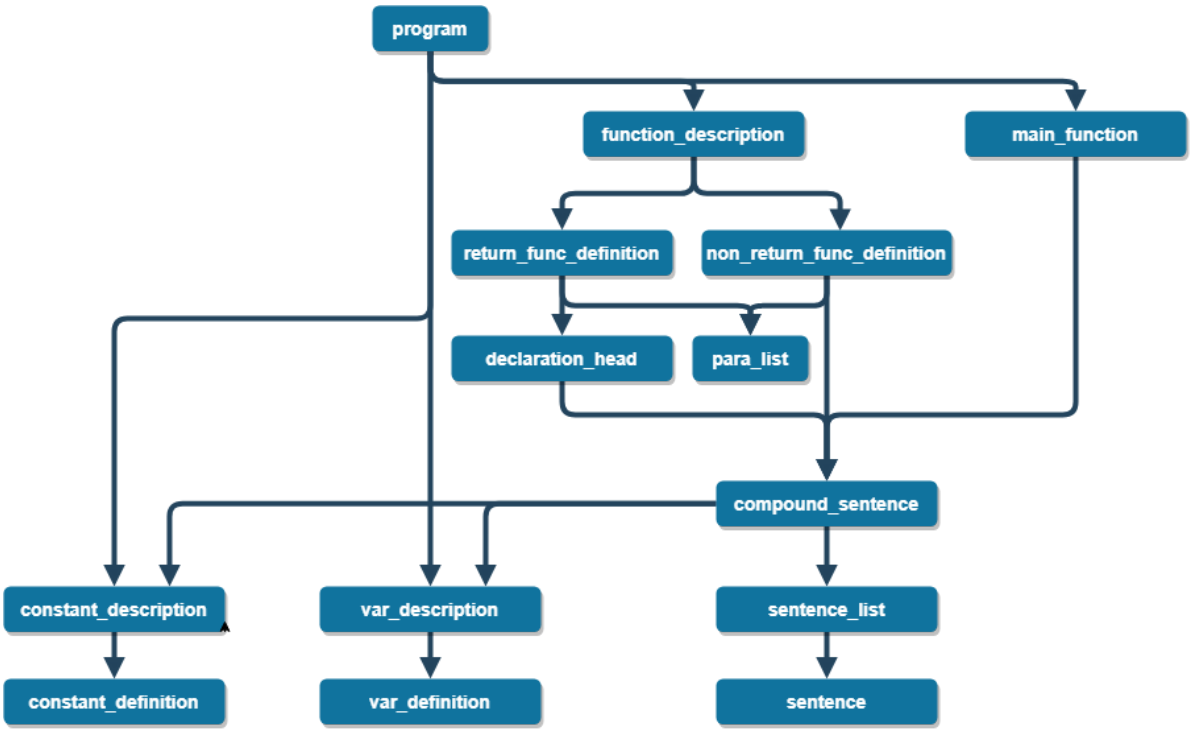
三、语法分析及语义分析

语法分析及语义分析是编译过程的第二个阶段，通过分析符号串来识别相应的语法成分，分析方法采用递归子程序法对程序符号串进行分析。

语法分析和词法分析的过程是同时进行的，每当递归子程序分析过程中需要读入字符时，就调用词法分析程序来读取一个符号。

3.1 函数调用结构

递归子程序法需要对每个语法成分设置一个函数进行分析，其调用结构如下：



3.2 代码结构

语法分析通过调用词法分析相关函数拓展而来，只通过上图的函数调用关系来实现整个语法分析程序较为复杂，因而在编码时，增加额外的函数和宏来进行代码结构的简化。

3.2.1 函数定义

```
unsigned int unsigned_integer(); //解析无符号整数
int integer(); // 解析一个整数
char character(); // 解析一个字符常量
std::string a_string(); // 解析一个字符串常量
void constant(); // 解析一个常量
void factor(); // 解析一个因子
void term(); // 解析一个项
void expression(); // 解析一个表达式
void constant_assign(Symbol symbol); // 常量赋值
void constant_definition(); // 常量定义
void constant_description(); // 常量说明
void linear_array_assign(Symbol symbol, unsigned int length); //
一维数组赋值
void two_dimension_array_assign(Symbol symbol, unsigned int x,
unsigned int y); //二维数组赋值
bool var_assign(Symbol symbol); // 变量赋值
void var_definition(); // 变量定义
void var_description(); // 变量说明
```

```
void declaration_head(); // 声明头部
void para_list(); // 解析参数表
void condition(); // 解析条件
void while_sentence(); // 解析while循环语句
void step(); // 解析for循环中的步长
void for_sentence(); // 解析for循环语句
void loop_sentence(); // 解析循环语句
void condition_sentence(); // 解析条件语句
void default_sentence(); // 解析缺省
void case_sentence(); // 解析情况子语句
void switch_table(); // 解析情况表
void switch_sentence(); // 解析情况语句
void scan_sentence(); // 解析读语句
void print_sentence(); // 解析写语句
void return_sentence(); // 解析返回语句
void assign_sentence(); // 解析赋值语句
void value_list(); // 解析值参数表
void call_return_func_sentence(); // 解析有返回值的函数调用语句
void call_function_sentence(); // 解析函数调用语句
void sentence(); // 解析语句
void sentence_list(); // 解析语句列
void compound_sentence(); // 解析复合语句
void return_func_definition(); // 解析有返回值函数的定义
void non_return_func_definition(); // 解析无返回值函数的定义
void function_description(); // 解析函数说明
void main_function(); // 解析主函数
void program(); // 解析一个程序

Symbol pre_read(); // 预读一个符号，并返回其类型
std::vector<Symbol> pre_read(int pre_read_num); // 预读
pre_read_num个符号，并返回到一个数组中
```

3.2.2 具体实现细节

文法中不存在左递归，可以通过预读符号来进行下一个语法成分的判别，避免回溯情况，因此在语法分析阶段，我将词法分析接口进行包装，将预读取符号和读取符号两项功能分离来避免回溯。

预读符号的实现

通过增加一个预读队列来实现预读功能，通过调用词法分析中的`get_symbol()`来进行预读，并将预读后得到的信息存入预读队列，每次预读一定个数符号时，就从当前预读队列中获取，或是通过`get_symbol()`函数来扩充预读队列来预读足够个数的符号。

在递归子程序中出现多个语法成分选择的情况下，预读符号而不是直接读取符号能够在保证语法成分选择正确性的前提下，不损失每个语法成分中的符号。

对下一个语法成分的判别

对于一个非终结符号，往往有多个语法成分选择，为了避免回溯，需要对这些选择的头符号集进行判别，准确进入下一个语法成分分析函数。可以通过预读一个或多个符号进行模式的匹配，判定是否进入当前判定的语法成分。通过定义宏来简化判别过程。

```
#define IS_TYPE_TK(s) (s == INT_TK || s == CHAR_TK)
#define IS_IDEN_TK(s) (s == IDENTIFIER_CON)
#define IS_SIGN_TK(s) (s == PLUS || s == MINUS)
#define IS_MULT_TK(s) (s == MULTI || s == DIV)
#define IS_INT_CON(s) (s == INT_CON)
#define IS_CHAR_CONS(s) (s == CHAR_CON)
#define IS_LEFT_SB(s) (s == LPARENT)
#define IS_FUNC_HEAD(s) (s == INT_TK || s == CHAR_TK || s == VOID_TK)
#define IS_VOID_TK(s) (s == VOID_TK)
#define IS_MAIN_TK(s) (s == MAIN_TK)
#define IS_SENTENCE_FIRST(s) (s == FOR_TK  || s == WHILE_TK || s == IF_TK || s == IDENTIFIER_CON || s == SCANF_TK ||  \
                                s == PRINTF_TK || s == SWITCH_TK || s == SEMICN || s == RETURN_TK || s == LBRACE)
#define IS_REL_SIGN(s) (s == LSS || s == LEQ || s == GRE || s == GEQ || s == EQL || s== NEQ)
#define IS_EXPS_FIRST(s) (s == IDENTIFIER_CON || s == PLUS || s == MINUS || s == INT_CON || s == CHAR_CON || s == LPARENT)

std::vector<Symbol> symbols = pre_read(3);
if (symbols.empty()) return;
if (IS_TYPE_TK(symbols[0]) && IS_IDEN_TK(symbols[1]) && !IS_LEFT_SB(symbols[2])) {
    var_description();
}
```

四、符号表生成

中间代码的生成、目标码的生成都建立在完成符号表的基础上，同时，需要我们进行编译的程序并不一定能够保证完全的正确性，对错误的处理也离不开查符号表、建符号表。符号表的建立在语法分析的过程中进行，符号表中需要存储程序中出现的函数信息，变量信息。

4.1 符号表定义

符号表分为全局的符号表包含程序中定义的全局常量、全局变量、函数信息，对于每个函数，还需要建立额外的局部符号表来存储函数中定义的常变量、临时变量等信息。

```
enum Location {
    GLOBAL_, PARTIAL_
};

```

```
enum Basic_type {
    INT_, CHAR_, VOID_, CHAR_VAR, CHAR_FUNC, CHAR_CONS
};

enum Type {
    CONST_, VAR_, ARRAY_, ARRAY2_, FUNC_, PARAM_, TMP_
};

typedef struct {
    std::string name;
    Basic_type basic_type; // int char void
    Type type; // const, var, array, func, param
    int value; // const -> value, var -> null, array -> length, func -> para_num
    // array2 -> 1 size
    int value2; // array2 -> 2 size
    Location location;
} identifier_info;

extern std::map<std::string, identifier_info> global_table;//全局符号表
extern std::map<std::string, identifier_info> partial_table;
extern std::map<std::string, std::vector<identifier_info>> func_para_list; //函数参数表
extern std::map<std::string, std::map<std::string, identifier_info>> func_table; //函数的符号表
```

4.2 符号表的生成与查询

符号表的生成过程伴随语法分析过程进行，也即在语法分析的过程中，对属于标识符定义的语法成分，如常量、变量、数组、函数的定义，需要将这些标识符插入到符号表中。而对程序中需要引用已定义标识符的地方，则需要查询已经生成的符号表。以下为符号表的插入与查询操作代码：

```
int insert_ident(Location l, std::string identifier_name, identifier_info ii) {

 std::transform(identifier_name.begin(),identifier_name.end(),identifier_name.begin(),tolower);
    if (l == GLOBAL_) {
        ii.location = GLOBAL_;
        return insert_global_ident(identifier_name,ii);
    } else {
        ii.location = PARTIAL_;
        return insert_partial_ident(identifier_name,ii);
    }
}

```

```
identifier_info *look_up_ident(Location l, std::string identifier_name) {

 std::transform(identifier_name.begin(),identifier_name.end(),identifier_name.begin(),tolower);

    if (l == GLOBAL_) {
        return look_up_global_ident(identifier_name);
    } else if (l == PARTIAL_) {
        return look_up_partial_ident(identifier_name);
    }
    return nullptr;
}
```

五、错误处理

错误处理伴随在编译过程的每个阶段，错误处理能够帮助我们准确定位程序中出现的错误和类型，帮助我们进行修改，也能够避免出现错误时继续分析程序而生成错误的目标代码。

5.1 错误类型定义

根据课程组给出的错误情况，可以将错误类型细分为以下几类：

```
enum ERROR_TYPE {
    //lexical_error
    INT_OVERFLOW, UNEXPECTED_CHAR_IN_INT, INVALID_CHAR_CON, INVALID_STRING_CON,
    UNKNOWN_SYMBOL,
    //syntactic_error
    IDEN_REDEFINITION, IDEN_NO_DEFINITION, FUNC_NO_DEFINITION, FUNC_PARA_NUM_DISPATCH,
    FUNC_PARA_TYPE_DISPATCH, CONDITION_TYPE_DISPATCH, RETURN_TYPE_DISPATCH,
    EXPECT_RETURN_SENTENCE, RETURN_EXP_IN_VOID_FUNC, ARRAY_INDEX_TYPE_DISPATCH,
    ASSIGN_TO_CONSTANT, EXPECT_SEMICN, EXPECT_RPARENT, EXPECT_RBRACK,
    INVALID_NUM_IN_ARRAY_INIT, INVALID_TYPE_IN_ARRAY_INIT, CONSTANT_TYPE_DISPATCH,
    EXPECT_DEFAULT_SENTENCE
};
```

错误类型分为两大类，为词法错误和语义错误，其中词法错误会在词法分析阶段进行，语义错误会在语法分析阶段进行，

5.2 处理过程

5.2.1 词法错误

对于词法错误，主要做法是在逐个读取字符，并判断是否为终结符的过程中，对合法终结符之外的情况进行分类处理，可以将词法错误细分到整数溢出，非法整数，非法字符，非法符号串以及未定义符号五种。

5.2.2 语义错误

对于语法错误，根据作业要求，可以将错误类型归结到符号表处理错误，表达式类型匹配错误，语法成分缺失错误，不合法定义错误四大类，之所以分成这四类，主要是因为这些错误在本质上可以归结到不同的语法分析元素中，并且可以进行类似的处理。

1）符号表处理错误，主要体现在标识符的重定义或未定义上，需要建立具有详细信息的符号表，在语法分析的过程中通过查符号表来判断。

2）表达式具有不同的返回类型，在我们的作业中，表达式的返回值被限定到了字符型和整型两大类，怎么在递归分析表达式的过程中准确得到表达式的返回类型是这部分的主要工作点。

3）语法成分缺失主要是指在语法分析的过程中没有读到期待的正确语法成分，例如括号的丢失，各种语句的丢失等，这部分的处理比较简单，通过预读符号类型来判断即可。

4）定义错误，主要体现在不符合前置条件的语法成分，例如常变量定义中初始化值类型不匹配，数组初始化时元素个数和类型不匹配，函数定义、调用时参数个数、类型不匹配等。

5.2.3 表达式返回值类型判断

表达式是通过表达式->项->因子这条递归链来定义的，在判断表达式类型时，从因子开始判断类型，并将结果逐层上传。最终得到表达式的类型。

具体做法如下：

```
Basic_type factor() {
    Basic_type ret = INT_;
    if (symbols[0] == IDENTIFIER_CON) {
        if (symbols[1] == LPARENT) {
            ret = call_return_func_sentence();
        } else if (symbols[1] == LBRACK) {
            ret = (p->basic_type == INT_) ? INT_ : CHAR_VAR;
        } else {
            ret = (p->basic_type == INT_) ? INT_ : CHAR_VAR;
        }
    } else if (IS_SIGN_TK(symbols[0]) || IS_INT_CON(symbols[0])) {
        ret = INT_;
    } else if (IS_CHAR_CONS(symbols[0])) {
        ret = CHAR_CONS;
```

```
    } else if (symbols[0] == LPARENT) {
        ret = INT_;
    }
    return ret;
}

Basic_type term() {
    Basic_type ret = INT_;
    std::vector<Symbol> symbols = pre_read(1);
    if (IS_EXPS_FIRST(symbols[0])) {
        ret = factor();
        while (true) {
            symbols = pre_read(1);
            if (symbols.empty()) break;
            if (IS_MULT_TK(symbols[0])) {
                get_symbol(); // * /
                Basic_type bt = factor();
                if (bt == CHAR_VAR && ret == CHAR_VAR) {
                    ret = CHAR_VAR;
                } else {
                    ret = INT_;
                }
            } else break;
        }
    }
    return ret;
}

Basic_type expression() {
    Basic_type ret = INT_;
    std::vector<Symbol> symbols = pre_read(1);
    if (IS_EXPS_FIRST(symbols[0])) {
        if (IS_SIGN_TK(symbols[0])) {
            get_symbol();
        }
        ret = term();
        while (true) {
            symbols = pre_read(1);
            if (symbols.empty()) break;
            if (IS_SIGN_TK(symbols[0])) {
                get_symbol(); // + -
                Basic_type bt = term();
                if (bt == CHAR_VAR && ret == CHAR_VAR) {
                    ret = CHAR_VAR;
                } else {
                    ret = INT_;
                }
            } else break;
        }
```

```
    }
    if (ret == CHAR_VAR || ret == CHAR_FUNC || ret == CHAR_CONS) {
        ret = CHAR_;
    }
    return ret;
}
```

如此做法使得语法分析过程中的每一个表达式都具有具体的返回值类型，对于表达式类型不匹配这类的错误就能得到很好的处理。

5.2.4 语法成分缺失的处理

对于语法成分的缺失，可以使用预读判断的方法来进行实现，例如如下对分号缺失的错误处理：

```
symbols = pre_read(1);
if (symbols[0] == SEMICN) {
    get_symbol();
} else {
    log_error(EXPECT_SEMICN, curr_symbol_line);
}
//get_symbol();//;
```

这类错误的处理具有很大的相似性，可以将这类错误处理抽象成一个函数来进行，使代码更加简洁。

5.2.5 定义不合法时的处理

定义不合法的情况主要有三类，函数调用时参数类型或个数不匹配，数组初始化时元素类型或个数不匹配，常量类型不一致。对于该类型错误的处理，将处理方法分发到各个处理过程中。即将所有的函数参数表存起来，每遇到函数调用时，就去分析函数调用语句的值参数表是否和该函数参数表匹配，否则报错。

六、中间代码设计

在进行完语法分析之后，需要根据语法分析结果来生成相应的中间代码序列，中间代码居于源程序和目标码之间，能够帮助我们进行源程序的进一步优化，并且能够根据中间代码敏捷生成不同的目标代码，因此中间代码的设计在代码生成阶段具有重要的作用。在课程设计中，编译器将会在语法分析的过程中生成源程序对应的四元式中间代码序列。

6.1 中间代码编码

在生成目标代码之前，根据文法来对生成过程中需要生成的中间代码进行编码，具体编码如下：

```
enum mid_opcode {
    CONST, VAR, PARA, ARRAY, ARRAY2, //常量，变量，参数，数组的声明
    ADD, SUB, MUL, DIV, NEG, //运算符，加减乘除和取负
    BEQ, BNE, BGEZ, BGTZ, BLEZ, BLTZ, //关系运算符 与MIPS代码语义保持一致
    READ, WRITE, WRITE_NEW_LINE, //读写操作，分别对应读入变量、打印数据、换行
    ASSIGN, //变量的赋值操作
    ARR_ASSIGN, ARR_INIT, USE_ARR, // 数组操作，对应数组元素赋值，数组元素初始化，数组元素引用
    GEN_LABEL, JUMP, //生成标签，跳转至某标签
    MAIN_START, MAIN_END, FUNC_START, FUNC_END, //主函数和其他函数的开始位置和结束位置
    RETURN, //返回语句，可以返回值也可以直接返回
    PUSH, CALL, GET_RET //函数调用操作 分别对应 参数入栈，调用函数，获取函数的返回值
};
```

6.2 中间代码生成过程

中间代码在语法分析的过程中生成，根据文法，中间代码的生成难点主要集中在以下几个地方：

表达式中间代码的生成

采用递归下降的方法来分析表达式，并生成中间代码，主要操作如下：

对递归终点的每个因子生成一个操作数对象，在回溯的过程中，若在项的分析中出现多个因子，就生成一条中间代码，进行运算符指定的运算，并生成一个新的临时变量，将运算结果存入该变量中，并将该临时变量作为结果再次进行回溯，表达式的操作同理。

数组元素操作

数组元素操作主要有数组赋值和数组元素引用两种形式，对于两种形式的数组，其取值和复制的操作都可以转化为对数组基址上一定偏移位置处数据的操作，二维数组的偏移位置确定需要额外增加乘除指令来计算。

条件语句

条件语句需要生成标签，其形式为：在条件成立时执行的语句体末尾增加标签，若条件不满足，则直接跳至尾部标签，因此每个条件语句的条件跳转指令都是反向的。若条件语句存在else语句，则需要额外生成标签和跳转语句。

循环语句

循环语句在入口处设置开始标签，并在进入标签后判断条件，若不满足条件则立即跳转至出口标签，否则执行循环体并在执行结束后跳转回开始标签。两种循环结构都是如此，但是for循环需要在末尾执行循环变量增减的操作。

函数

对于一个函数结构，需要设置函数的入口和出口，函数的入口位于函数声明处，出口则位于函数体末尾或者return语句处。

6.3 中间码格式

OP_CODE	op_num1	op_num2	result
CONST 定义常量			const_info
VAR 定义变量			var_info
PARA 定义参数			para_info
ARRAY 定义一维数组			array_info
ARRAY2 定义二维数组			array2_info
ADD result=opnum1+opnum2	opnum1	opnum2	result
SUB result=opnum1-opnum2	opnum1	opnum2	result
MUL result=opnum1*opnum2	opnum1	opnum2	result
DIV result=opnum1/opnum2	opnum1	opnum2	result
NEG result= -opnum1	opnum1		result
READ scanf(read_var)			read_var
WRITE printf(write_var)			write_var
ASSIGN dst=src	src		dst
ARR_ASSIGN array_name[index]=value	array_name	index	value
ARR_INIT array_name[index]=value	array_name	index	value
USE_ARR var=array_name[index]	array_name	index	var

OP_CODE	op_num1	op_num2	result
WRITE_NEW_LINE printf('\n')			
MAIN_START main函数起始位置			
MAIN_END main函数结尾			
GEN_LABEL 生成标签			label_info
JUMP 跳转至标签			label
BEQ 两操作数相等则跳转	opnum1	opnum2	label
BNE 两操作数不等则跳转	opnum1	opnum2	label
BGEZ 操作数大于等于0则跳转	opnum1		label
BGTZ 操作数大于0则跳转	opnum1		label
BLEZ 操作数小于等于0则跳转	opnum1		label
BLTZ 操作数小于0则跳转	opnum1		label
FUNC_START 函数开始位置			func_name
FUNC_END 函数结束位置			func_name
RETURN 返回语句			var/null
PUSH 参数入栈			var/const
CALL 调用函数			func_name

OP_CODE	op_num1	op_num2	result
GET_RET 获取函数的返回值			var_info

七、目标码生成

生成的中间代码逐条翻译成为目标代码，最终生成的目标码需要符合规范。在生成MIPS代码的过程中，需要重新扫描生成的中间代码，并配合符号表进行内存分配和寄存器分配。

7.1 地址分配

按照全局和局部进行地址分配，对于全局变量和数组，统一置放于gp区，局部变量全部存在sp区。

首先对全局符号表进行扫描，全局符号表中存储了所有全局变量和函数信息，扫描符号表并将所有信息存入寄存器信息表中，以方便在生成代码时分配寄存器，寄存器表的结构如下：

```
std::map<std::string, Record*> reg_map;
struct Record {
    Reg reg; //使用的寄存器
    std::vector<Reg> func_used;
    bool is_global; // 存放区域是gp还是sp
    int gp_offset; // 相对gp的偏移
    int sp_offset; // 相对sp的偏移
    int stack_size; // 函数的栈空间
};
```

使用名称来唯一确定一个变量的寄存器分配，对于全局变量，初始化该变量的gp_offset，对于函数，扫描它的参数表和符号表，初始化其sp_offset，函数栈的分配从上到下按照局部变量，临时变量，fp寄存器和ra寄存器值，参数值进行存储。

例如对下面函数int fun(int x, int y)的分配：

```
int fun(int x,int y) {
    int z,w;
    int p = 10;
    p = 1 + 1 + 2;
    z = p;
    return (z);
}
分配结果
FUNC fun VAR p At 0
FUNC fun VAR w At 4
FUNC fun VAR z At 8
FUNC fun TMP t0 At 12
```

```
FUNC fun TMP t1 At 16
FUNC fun fp rec  At 20
FUNC fun ra rec  At 24
FUNC fun PARA x At 44
FUNC fun PARA y At 40
FUNC fun para num:2 stack size: 48
该偏移是相对于当前fp寄存器的
```

由于函数中存在的调用关系，在处理有参数的函数时，每遇到一个函数参数，都需要压栈将其存入，此时若通过sp寄存器以及偏移来访问数据，会出现数据错位，无法获取正确数据的情况，因此在每进入到一个函数时，都会将当前的栈顶存入fp寄存器中，保证数据访问是的正确性。

7.2 寄存器分配

对局部变量分配s寄存器，对临时变量分配t寄存器，按照最近最少使用的原则来进行分配。维护一个寄存器使用队列，即每次在引用一个变量之后，就将该变量对应的寄存器置于队列尾部，当所有寄存器都被占用时，从队首取出最近一段时间内使用次数最少的寄存器，用来存储新的变量。

对程序中可能出现的需要给常量分配寄存器的情况，引入两个寄存器信息const_tmp1和const_tmp2，来为常量申请寄存器。

7.3 目标码生成

目标码的生成事实上是一个机械的翻译过程，需要注意几个方面：

1. 将整个代码分为三个部分，data部分，声明函数中可能用到的字符串；init区，在程序跳转到main函数之前对一些全局变量进行初始化；text区，存放生成的目标。最后按照data，init，text的顺序输出，就可以得到最终的目标码。
2. 对字符串的输出，在目标码的data区声明这些字符串，并为其生成标签，在打印字符串时，只需要其标签信息就可以完成。
3. 对函数的调用，采用先push参数进栈，在进入函数体后再申请其余空间的方法，退出函数时则恢复申请的栈空间。
4. 每当进入新的函数时，必须立即保存之前函数的ra和fp寄存器值，这两个值分别对应外部函数的返回地址和栈指针，在执行完函数之后，则恢复这两个值。

八、代码优化

在进行完以上步骤之后，我们的编译器已经能够通过编译符合文法的程序生成可执行的MIPS代码，按部就班的生成能够保证稳定性，但不能够保证程序的效率，例如程序中出现的一些常量表达式，不可进入的条件语句，循环体内不变的表达式，忽视对这些存在优化可能性的程序部分的处理，我们的编译器在编译运行一些特殊的程序时，会生成效率极低的代码。因此对代码中存在的一些可优化部分，有必要再保证生成代码正确性的前提下进行优化。我在课程设计中完成了三项优化任务，如下：

8.1 常量传播

对程序中出现的一些常量表达式，例如a=1+2+3+4+5+6;，按照一般的生成过程，计算这样一个常量表达式需要生成5个额外的临时变量用来存储值，而事实上这种语句带进行中间代码的生成时就可以进行优化，直接令a=21，会减少很多不必要的计算，程序中对常量的引用也是如此，直接将值代入进行计算，会省去很多不必要的运算。

具体的做法是在生成的中间代码中进行扫描，将常量的运算的结果对应的临时变量进行替换，替换为运算结果，如此往复，直至程序中不存在常量运算的情况。

8.2 死代码删除

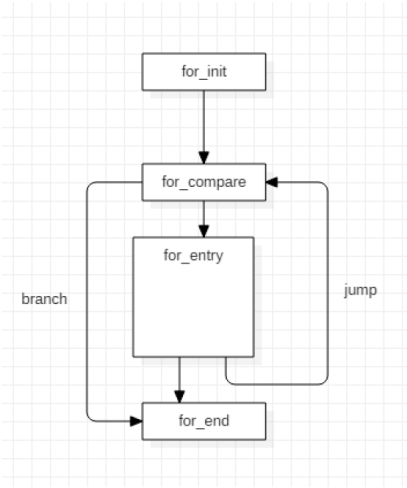
程序中出现的一些条件跳转语句，其进入条件为恒假的，就可以在中间代码中删去从条件跳转到跳转标签范围内的所有中间代码。减少代码量。

8.3 乘除法优化

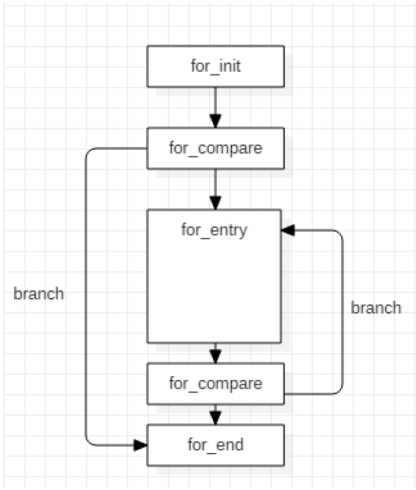
在MIPS体系结构中，乘除法不在ALU中进行计算，而在独立的乘除法器中进行，而且乘除法运算时间长，运算代价大，因此在生成的代码中，可以尽可能减少乘除法出现的频率。对程序中存在的乘除常量的语句，若除数或者乘数为2的幂，则可以将乘除法转化为移位运算，提高效率。

8.4 循环结构优化

对常规的循环结构，一般的处理策略如下图，在判断条件满足后执行循环体，执行完循环体后跳转至开头进行条件判断，这样的做法最为直观，但每次循环都会执行两次跳转语句。



若将循环结构改为如下所示，虽然第一次进入循环需要进行两次条件判断，但当循环次数较多的情况下，每次循环执行的跳转语句平均只有一条，相比上面的结构，能够减少近一半的跳转语句。



九、总结

编译技术课程设计从最简单的词法分析开始，一步步扩展和实现整个编译器的功能，并最终实现一个能够将给定程序编译为MIPS代码的编译器。通过这一个学期的学习，一步步地完成自己的编译器，我了解了实现一个编译器的基本流程，学会了设计和完成一个项目的基本方法和思路，提高了工程能力，收获颇丰。