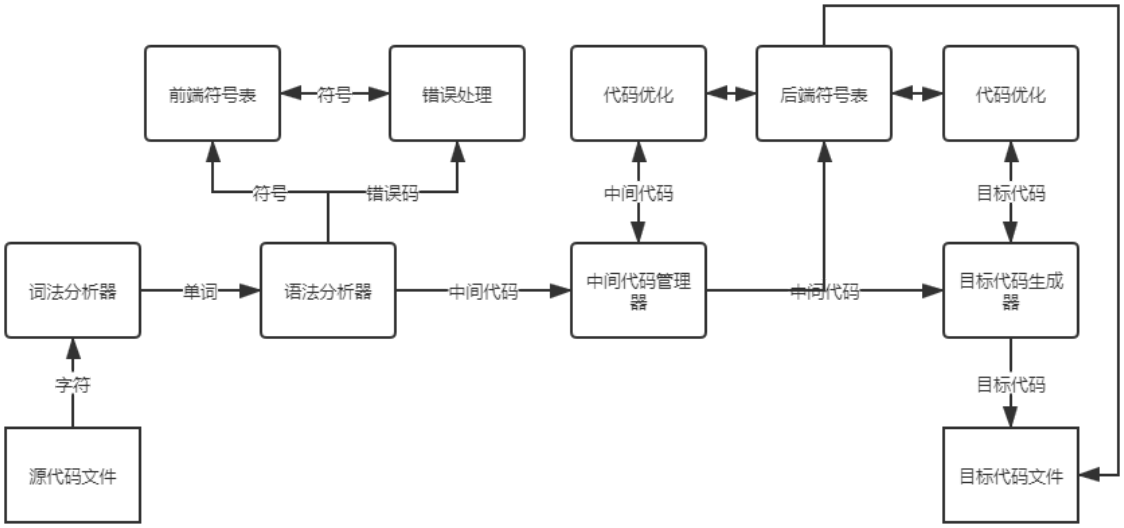
编译课设总设计文档

18373584 甘天淳

一、整体架构

整体上采用前后端分离的架构，前端主要由词法分析、语法分析、错误处理、符号表管理、中间代码生成几部分组成，词法分析器从输入文件中不断读入字符形成单词并提供给语法分析器，语法分析器进行递归下降的语法成分分析并同步进行错误探测与处理和填读符号表操作，同时针对各语句生成相应的四元式中间代码。后端主要由代码优化和目标代码生成部分组成，后端从前端生成的中间代码开始，首先构造记录各变量内存偏移量的后端符号表，再对中间代码进行多遍扫描以进行优化和目标代码生成。有图示如下：



本架构中涉及到的主要实体的定义及结构如下：

单词：用于词法分析器和语法分析器的交互，具体形式如下：

```
struct word
{
    std::string symbol;
    std::string content;
    int line;
};
```

符号：文法中定义的符号主要有整型常量、字符常量、整型变量（一维、二维数组）、字符变量（一维、二维数组）、返回值为整型或字符的函数、无返回值函数。因此采用 `identifierType` 和 `identifierBaseType` 分别描述上述符号的属性，此外函数需要单独存储其参数表，变量需要存储其维数及各维范围，其具体形式如下：

```
enum identifierType
{
    CONST,          // 常量
    VAR,            // 变量
    ARRAY1,         // 一维变量数组
```

```
    ARRAY2,          // 二维变量数组
    FUNC,            // 函数
    PARAM,           // 函数参数
    NOTEXISTS        // *不存在*，仅作符号不存在时返回使用
};

enum identifierBaseType
{
    INT,             // 整型
    CHAR,            // 字符型
    VOID,            // 无返回值型，在本文法中仅无返回值函数
    NOEXISTS         // *不存在*，仅作符号不存在时返回使用
};

struct Symbol
{
    std::string name;                    // 符号名
    identifierType type;                 // 符号类型
    identifierBaseType baseType;         // 符号基础类型
    std::vector<Symbol> params;          // 函数参数表，非函数则为空
    int dim1;                            // 第一维上限，若非一维数组则为1
    int dim2;                            // 第二维上限，若非二维数组则为1
    int value;                           // 具体值，仅常量使用
    int offset;                          // 内存偏移量，仅后端符号表使用
};
```

四元式：每一个四元式拥有四个域，分别是一个操作和三个符号，其具体形式如下:

```
struct MiddleCode
{
    Operator op;
    Symbol id1;
    Symbol id2;
    Symbol id3;
};
```

二、词法分析

词法分析部分主要作用是从源程序（输入文件）中识别出单词，记录其单词类别和单词值，并记录下来以提供给语法分析部分使用。因此我们将词法分析器实现为单独的类，其构造参数为输入文件流，主要功能为不断读取输入文件字符并利用状态机识别单词，记录单词类别与单词值，形成结构体 `word` 并填入 `wordList` 链表中，主函数可调用其进行单词表的输出。在具体的识别过程中，应进行读入和识别字符的约定，在本架构中我们约定：

1. 识别单词之前均预先读入一个字符.
2. 在某些需要继续读入以判断单词类别的情况下使用 `peek()` 函数，避免回退。
3. 记录单词值时将字母直接转为小写。

在语法分析作业过程中对词法分析部分架构进行了修改，使得每次调用词法分析函数时仅返回下一个单词的类别与值，而不需要将单词保存。修改后词法分析器模块主要函数及功能如下：

`LexicalAnalyzer::LexicalAnalyzer(std::ifstream &input)`：词法分析器的构造函数，接

收输入文件流作为参数，这是因为整个架构中仅词法分析部分需要读取输入文件，其他部分均不与输入

文件交互。

`LexicalAnalyzer::analyze()`：词法分析的主要函数，依次读入字符并依照文法识别单词，识别到一个单词后便返回。

三、语法分析

语法分析部分从词法分析程序中不断获取单词，使用递归下降的方式进行各语法成分的分析。语法分析器类将词法分析器作为其私有变量，主函数可调用语法分析器从 `<程序>` 开始进行分析。

3.1 语法分析器构造

语法分析器 `SyntaxAnalyzer` 模块是编译器前端的主要模块，其主体是各递归下降分析函数，在递归分析过程中同时进行错误记录与处理、符号表填读等多种操作，同时还能够进行语义分析以生成中间代码，本架构中语法分析器主要函数功能如下：

`SyntaxAnalyzer(LexicalAnalyzer &newLexicalAnalyzer)`：语法分析器构造函数，接收词法分析器作为参数，使得词法分析器只与语法分析器进行交互。

`void analyze()`：语法分析的入口函数，做递归下降前的准备工作（即按照约定首先读入一个单词）后从语法成分 `<程序>` 开始分析。

`void *()`：针对每一个非终结符的递归下降分析函数，需要输出当前读到的单词，根据文法FIRST集进入下一个递归，遵循约定在退出前输出语法成分并多读下一个单词。针对可能发生冲突的函数定义等递归子程序，通过预读方式避免回溯，设置 `int mode` 作为参数，改变其值可以在具体分析过程中忽略已经预读的成分，具体方式参见第三节“约定及特殊情形处理方法”。

3.2 递归子程序的一般构造方法

本文法大多数非终结符已经消除左递归且FIRST集不相交，可以较为暴力地构造递归子程序，具体方式为：若下一个单词应是终结符，则直接判断当前单词是否为该终结符并输出再多读一个单词，若下一个单词应是非终结符，则直接进入该非终结符所对应的递归子程序，若下一单词可以是多个非终结符，则根据 `FIRST` 集进入不同的递归子程序，若下一个单词可有可无，则进行判断，若下一个单词可以出现多次，则用 `while` 语句块将其包裹，用下一终结符作为退出循环的条件。对于 `<变量定义><变量定义及初始化><变量定义无初始化>` 等需要预读较多的文法，可直接修改文法将 `<类型标识符><标识符>` 语法成分提出来，再根据以上方法构造子程序。

3.3 约定及特殊情形的处理方法

1. 递归下降应遵循一定的约定，在本架构中，约定进入每一个递归子程序前已将单词读入，在每一个递归子程序退出前应多读下一个单词。
2. 变量声明与有返回值函数声明避免回溯的问题。在本文法中，变量声明和有返回值函数声明均可以 `<类型标识符><标识符>` 开头，为避免回溯，本架构中如下处理： `<程序>` 中分析 `<常量声明>` 结束后，若下一单词类别为 `INTTK | CHARTK` 则进入变量声明递归子程序，在变量声明子程序中先记录 `<类型标识符><标识符>` 的值而不输出，若读到下一个字符为 `LPARENT` 则按需输出 `<变量声明><声明头部>` 语法成分，再以 `mode 1` 进入函数定义子程序（不再分析声明头部），否则输出后进入变量定义递归子程序，在变量定义递归子程序中不再分析 `<类型标识符><标识符>` 成分。

四、符号表管理

由于前后端分离架构的限制，在编译全过程中共需使用两个符号表，其中前端符号表主要服务于中间代码生成过程以及错误处理过程，后端符号表主要用于记录各符号偏移量以生成 `mips` 目标代码，二者结构相似且后端符号表体积更小，因此在此仅对前端符号表做出解释。

4.1 符号表结构

一般而言，符号表需要形成链式结构以区分不同作用域中定义的符号，然而观察文法可以发现并不需要这么复杂的符号表结构设计：

```
<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>'}'
<主函数>    ::= void main‘(’‘)’ ‘{’<复合语句>‘}’
<复合语句>   ::=  [<常量说明>] [<变量说明>] <语句列>
```

通过观察以上文法可以发现，局部变量定义仅在 `<复合语句>` 中出现，而 `<复合语句>` 仅作为函数主体出现。即变量定义仅在全局或函数体开头出现，其作用域分别为全局和当前函数，因此我们仅需两个栈分别存储全局与当前函数符号表即可，在本架构中其分别为 `vector<Symbol> globalTable` 和 `vector<Symbol> tmpTable` 。具体实现上，将全局定义的常量、变量及函数存入全局符号表中，将局部定义的常量及变量存入当前符号表中（函数参数需要另存一份到全局符号表的对应函数参数中），当前函数的递归下降语法分析完成后清空当前符号表。

4.2 符号表对外接口

符号表主要为语法递归下降分析部分提供增删查改接口，此外还有一些为了方便而增加的特殊接口。

`bool insert(std::string name, identifierType type, identifierBaseType baseType)` ：符号表的插入操作。将标识符为 `name` ，类型为 `type` ，基础类型为 `baseType` 的符号填入符号表中，填入前需要在当前符号表中搜索该标识符以进行重复定义检测，若未重复定义则返回 `true` ，重复定义则返回 `false` 。

`void clearClosetFunc()` ：清空当前符号表。函数定义的递归下降分析结束前清空 `tmpTable` 以防止对后续函数的分析造成影响。

`bool findInNowFunc(std::string name), identifierType findInAllFunc(std::string name), identifierBaseType findBaseInAllFunc(std::string name)` ：数个查询函数。接收标识符的字符串形式，分别在当前符号表、所有符号表中进行查询并返回 `type` 或 `baseType` 。

`std::vector<Symbol> getFuncParams(std::string name)` ：查询函数参数表并返回。

`void changeVarType(int dim, int dim1, int dim2)` ：将当前正在分析的符号（一定处于当前符号表的末尾）的维数及各维上限分别改成 `dim, dim1, dim2` 。

`identifierBaseType findClosetBaseType()` ：将当前正在分析的符号（一定处于当前符号表的末尾）的 `baseType` 返回。

五、错误处理

本次作业错误处理共有16种错误，且需要进行错误局部化处理，因此将错误处理内嵌到语法分析的递归下降程序中而不单列错误处理器，具体处理方式如下：

错误类型	错误类型码	错误判断及处理方式
非法符号或不符合词法	a	在字符与字符串分析中当单词长度不正确或含有不正确的ascii字符时报错，并跳过该符号分析
名字重定义	b	各语法成分声明/定义时通过判断符号表的插入操作返回值进行报错，并跳过该声明/定义语句
未定义的名字	c	各语法成分引用时通过判断符号表的查询操作返回值进行报错，并跳过该引用语句
函数参数个数不匹配	d	函数调用分析值参数表过程中存储传入的各参数，值参数表分析结束时将其与符号表中查出的该函数参数表作比较，若个数不匹配则报错，且跳过该函数调用语句
函数参数类型不匹配	e	函数调用分析值参数表过程中存储传入的各参数，值参数表分析结束时将其与符号表中查出的该函数参数表作比较，若类型不匹配则报错，且跳过该函数调用语句
条件判断中出现不合法的类型	f	表达式、因子、项分析过程中返回其所分析的类型（整型或字符型），条件语句分析中若出现条件左右为字符型则报错，且跳过该条件语句
无返回值的函数存在不匹配的return语句	g	使用全局变量`nowFuncBaseType`存储当前所分析函数的返回类型，返回语句分析过程中判断是否符合，若不符则报错，跳过该返回语句
有返回值的函数缺少return语句或存在不匹配的return语句	h	使用全局变量`nowFuncBaseType`存储当前所分析函数的返回类型，`hasReturn`存储是否含有正确的返回语句，返回语句分析过程中判断是否符合，若不符合则报错，符合则更新`hasReturn`，当前函数分析结束时若`hasReturn`仍为否则报错
数组元素的下标只能是整型表达式	i	表达式、因子、项分析过程中返回其所分析的类型（整型或字符型），数组下标分析表达式返回值若为字符型则报错，且跳过该含有数组下标的语句
不能改变常量的值	j	赋值语句，读语句中对将要改变值的符号进行检查，若查表结果为常量则报错，且跳过该语句
应为分号	k	在各语法成分分析中当下一单词应为`SEMICN`而非时报错，并跳过该分号
应为右小括号')'	l	在各语法成分分析中当下一单词应为`RPARENT`而非时报错，并跳过该右小括号
应为右中括号']'	m	在各语法成分分析中当下一单词应为`RBRACK`而非时报错，并跳过该右中括号
数组初始化个数不匹配	n	数组初始化时记录其各维各组数量，初始化结束前与查表所得的该数组符号一维、二维范围进行比较，若不符则报错，且跳过该声明语句

错误类型	错误类型码	错误判断及处理方式
<常量>类型不一致	o	变量定义及初始化和情况语句中对所分析符号进行查表得到其基本类型，若所分析表达式返回值与其不符则报错，且跳过该语句
缺少缺省语句	p	在情况语句分析中当下一单词应为 `RETURN` 而非时报错，并跳过缺省语句分析

六、语义分析与中间代码生成

6.1 四元式形式

每一个四元式拥有四个域，分别是一个操作和三个符号，对四元式具体形式作解释如下：

op	id1	id2	id3	解释
OpAdd OpMinus OpMulti OpDiv	左操作数符号	右操作数符号	结果符号	四则运算
OpAssign	被赋值符号	值符号		赋值
OpScan	被读符号			读
OpPrint	被写符号			写
OpLabel	标签符号			标签
OpDeclare	被声明符号			声明常量/变量/数组
OpArrayInit	数组符号	一维下标	初始化值	数组初始化
OpArrayGet	数组符号	一维下标	结果符号	取数组元素
OpArrayPush	数组符号	一维下标	写入值符号	写数组元素
OpBge OpBgt OpBle OpBlt OpBeq OpBne	左操作数符号	右操作数符号	标签符号	各类条件跳转
OpJ	标签符号			无条件跳转
OpFuncDeclare	函数符号			函数声明
OpFuncCall	函数符号	结果符号		函数调用
OpFuncReturn	返回值符号			函数返回
OpPushParam	参数符号			压入函数调用参数

得益于符号结构体已包含该符号所有信息的特点，本设计中四元式结构简单，易于设计与理解，且可以完成前后端分离的代码生成。

6.2 各语句中间代码生成策略

分支语句、循环语句、读写语句、赋值语句等语法成分在不考虑优化的情况下仅需要简单的线性的翻译即可，因此不在此赘述。下面仅解释三个比较复杂的中间代码生成策略。

6.2.1 表达式相关四元式生成

利用全局变量 `nowSymbol` 完成表达式相关四元式的生成，其作用是记录当前符号。

递归下降分析表达式时，调用项分析子程序，完成后将 `nowSymbol` 记入 `symbol1` 中，若分析到加减法运算符则再次调用项分析子程序，完成后将 `nowSymbol` 记入 `symbol2` 中，生成新的中间变量 `symbol3` 和相应加减法四元式，并将 `nowSymbol` 和 `symbol1` 均置为 `symbol3` 。

与表达式类似的，递归下降分析项时，调用因子分析子程序，完成后将 `nowSymbol` 记入 `symbol1` 中，若分析到乘除法运算符则再次调用因子分析子程序，完成后将 `nowSymbol` 记入 `symbol2` 中，生成新的中间变量 `symbol3` 和相应乘除法四元式，并将 `nowSymbol` 和 `symbol1` 均置为 `symbol3` 。

递归下降分析因子时，将因子具体分析结果（变量、函数调用结果、数组元素等）的符号赋给 `nowSymbol` 。

此外，取数组元素和有返回值函数调用分析过程中也应生成中间变量以记录结果。

6.2.2 数组相关四元式生成

在中间代码生成过程中将数组一律视为二维数组处理，并生成仅一维寻址的数组相关中间代码。

语法分析中分析到数组声明头部时即将其各维大小（若一维则 `dim2=1` ）记入该数组符号中，分析其初始化值表过程中则可通过 `offset = (tmpDim1-1)*dim2+tmpDim2-1` 转换为相对于数组头的一维寻址，数组读写分析过程中对于下标的变化与此类似，不再赘述。

6.2.3 函数调用相关四元式生成

分析语法成分 `<值参数表>` 时，为各参数表达式结果生成一句压入参数中间代码，语法成分 `<函数调用>` 分析结束前生成一句函数调用中间代码。

七、目标代码生成

由于中间代码符号已带有其全部信息，则目标代码完全由中间代码生成，与语法分析等部分无关。得益于中间代码形式较为类似 `mips` 指令，大多数中间代码可以直接翻译成目标代码。

7.1 翻译顺序

语法分析过程中生成的中间代码存储于一个容器中，首先对所有中间代码进行一次扫描，取出所有字符串并生成 `.data` 段，再生成全局变量声明及初始化部分，并记录下各函数声明标签的位置。再次扫描，为每两个函数声明标签中间的部分生成该函数对应的目标代码。

7.2 内存分配与后端符号表

为全局变量分配相对 `$gp` 寻址的内存空间，为局部变量分配相对 `$sp` 寻址的内存空间，并用两个符号表分别管理，后端符号表与前端符号表类似，提供根据变量名返回其偏移的接口如下：

```
std::pair<bool, int> getOffset(std::string name);
            // Input:  变量名
            // Output:  相对于 $gp 或 $sp 的偏移量
```

全局变量的分配是目标码生成的第一步（也因为全局变量的声明中间代码位于中间代码开头），具体方式是为每一个声明中间代码分配 `1 word` 内存并将初始化值写入，将相对于 `$gp` 偏移量记录于所声明符号内并存入全局符号表。局部变量和中间变量的分配则在函数声明产生函数标签后立即全部完成，具体分配方式与全局变量类似，不同点在于其分配的是相对于 `$sp` 的空间，存入当前符号表中。

7.3 函数调用策略

压入参数中间代码会将相应参数符号压入全局参数栈 `paramStack` 中。

函数调用时，首先取出 `paramStack` 中对应个数的参数符号，将参数存入 `$sp` 下的内存中，再往下存入需要保存的寄存器（此时只有 `$ra,$sp` 需要保存），随后将 `$sp` 寄存器下调到对应位置（这样保证了从这里开始分配新函数的局部变量不会发生冲突），随后跳转到对应函数Label，返回后取出 `$v0` 中记录的返回值即可。

函数返回时，首先将返回值（如果有的话）存入 `$v0` 中，再恢复上一函数运行现场（ `$sp, $ra` 等），最后跳转 `jr $ra` 。主函数中出现返回语句则直接结束程序。

八、代码优化

由于时间关系，代码优化部分仅尝试暴力的全局寄存器分配和临时寄存器池，但是暴力的全局寄存器分配会导致调用函数时保存现场更加复杂，表达式计算过程中也很少用到比较多的临时寄存器，因此效果均不算太好，最终全部放弃。