同济大学

TONGJI UNIVERSITY

《编译原理》
设计说明

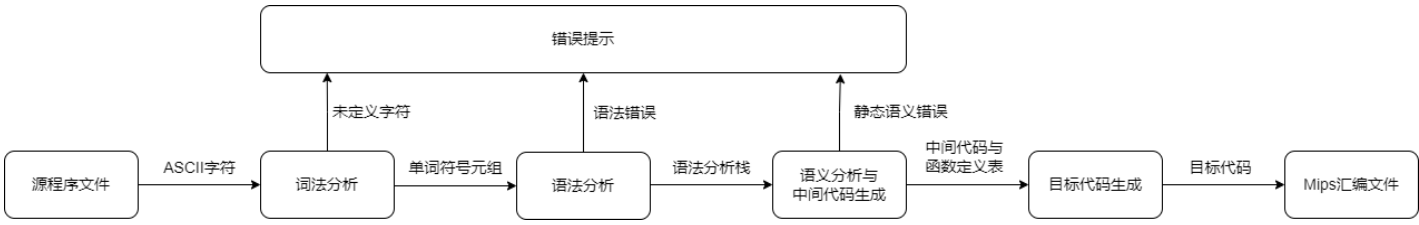
作业名称	类 C 程序编译器
姓　　名	张博文
学　　号	2154071
学院（系）	电子与信息工程学院
专　　业	计算机科学与技术
任课教师	丁志军
日　　期	2023 年 5 月 20 日

目录

一、 需求分析

本编译器采取一遍扫描的翻译方式，实现了对包括过程调用的类 C 语言源程序的翻译，能够生成 Mips 格式的汇编目标代码。

编译流程共分为 4 个主要阶段，分别是词法分析、语法分析、语义分析与中间代码生成、目标代码生成。下图展示了编译器的结构。



编译器程序按照这 4 个阶段分为对应的 4 个主要模块。词法分析作为子程序，在需要的时候被语法分析调用；语义分析与语法分析同时进行，分析完成后生成中间代码，并且可以选择保存中间代码；通过中间代码，生成可汇编执行的 Mips 汇编目标代码。

本编译器的输入是类 C 语言源程序，可以通过输入框输入，也可以通过文件打开，可以在文本框进行编辑，并且提供了保存和另存接口；中间代码、目标代码输出在屏幕上，用户也可以将代码保存到本地文件。

装 订 线

二、 概要设计

按照本编译程序划分成的 4 个模块，分别叙述各模块的功能。

2.1 词法分析

词法分析的任务是以空格为分隔，逐个读入待编译源程序的单词，并将其转换成单词种别+属性值的形式传递给语法分析器。

2.2 语法分析

语法分析的任务是判断源程序是否符合文法，也就是检测源程序是否有在语法层面上的错误，如果有，编译器程序需要给出错误提示。为了实现这一目标，在语法分析阶段还需要根据文法生成 action 表和 go 表，利用这两个表对输入符号串进行规约，以完成语法分析过程。

2.3 语义分析与中间代码生成

语义分析与语法分析是同时进行的，并且在分析的过程 中还需要生成中间代码。

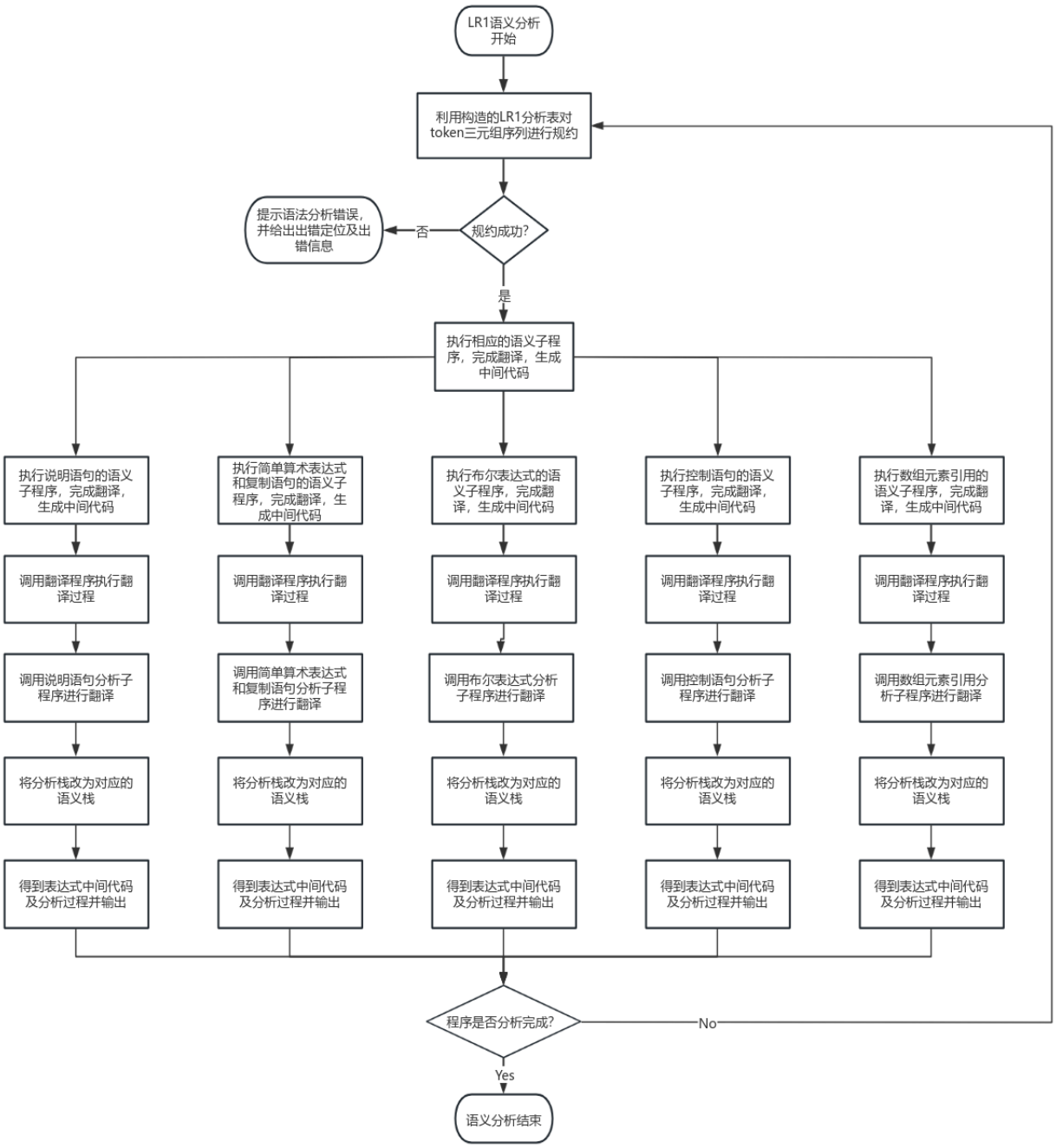
语义分析的任务是判断源程序在语义层面上是否存在错误，比如对 C 程序来说，使用了未定义的变量属于语义错误，即程序虽然符合语法规范，但程序表示的含义有误。

语法分析和语义分析是通过规约进行的，在规约时遵循属性文法的翻译模式，就可以生成对应的中间代码。将中间代码全部保存下来，以供后续生成目标代码。

2.4 目标代码生成

本编译器的最终目标是生成可汇编执行的目标代码。选取的是 Mips 格式的汇编代码，包括取数存数、条件与无条件转移以及加减乘除运算等指令。与教材中提到的目标代码不同，Mips 指令的运算指令只能有寄存器与立即数参与，不能通过内存访问的方式直接进行运算操作，这会涉及到更加复杂的取数存数逻辑，在后续会详细介绍。生成的代码（包括中间代码）会显示在屏幕上，用户也可以选择将其保存至指定位置。

给出到语义分析为止的程序执行流程。



装 订 线

三、 详细设计

这里同样按照四个模块分别进行叙述，并详细叙述各模块间的调用关系。

3.1 词法分析

3.1.1 数据结构设计

词法分析将源程序的单词转换成种别+属性值的形式传递给语法分析。这里的种别指单词的种类，比如所有的变量名、函数名都属于标识符，用字符串”idt”表示，属性值是单词原本的形式。因此可以用 pair<string, string>类型的二元组来表示单词符号，前者是单词的类型，后者是其内容。此外，还需要存储单词行号这一信息，用于报错提示时定位，将行号与二元组一起组成单词的信息结构。行号的值可以通过读到的换行符数量来判断。

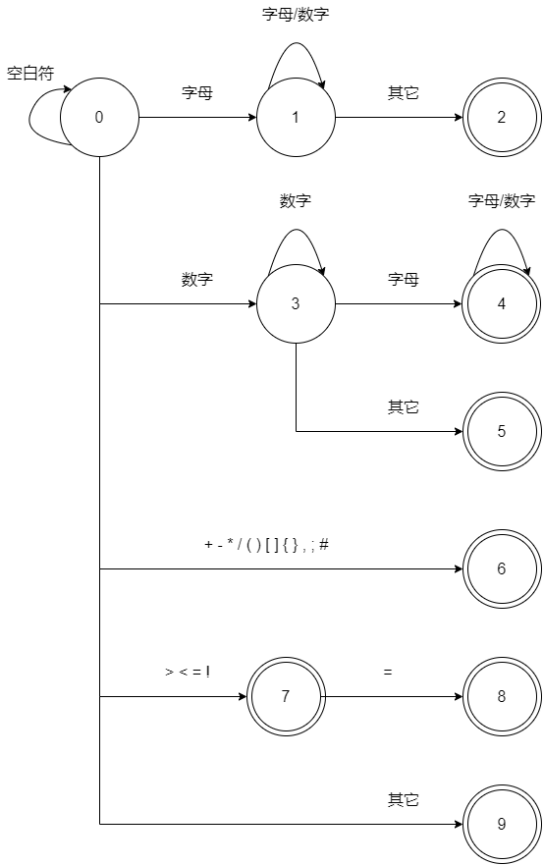
3.1.2 主要算法

词法分析模块的主要算法用于解决如何将单词分离出来并判断其类别，本质上是一个自动机模型。

c++的文件流可以轻易实现以空格/换行符为分隔，读入一个单词的功能。将读入的单词作为输入，设计一个 DFA 来判断单词的类别。类 C 源程序中所有可能出现的单词分为 3 中类型：标识符、数字和符号。关于标识符，C 语言的规定是：以下划线或字母为开头的、只包含字母、数字、下划线的单词，这里省略下划线，规定标识符是：以字母为开头的、只包含字母与数字的单词；关于数字，本编译器只考虑整数，不考虑小数,因此数字单词仅包含 0~9 的阿拉伯数字;符号包含+-*/等算数运算符号、与或非等逻辑运算符号、><=等比较运算符号及其组合、三种括号、///**/等注释符号以及逗号、分号等分隔符号。规定所有标识符属于同一种别、数字属于同一种别、每种符号各自成为一个种别，因为对于符号需要分类讨论其不同的功能，而标识符和数字不需要。

自动机模型可用下图来表示。

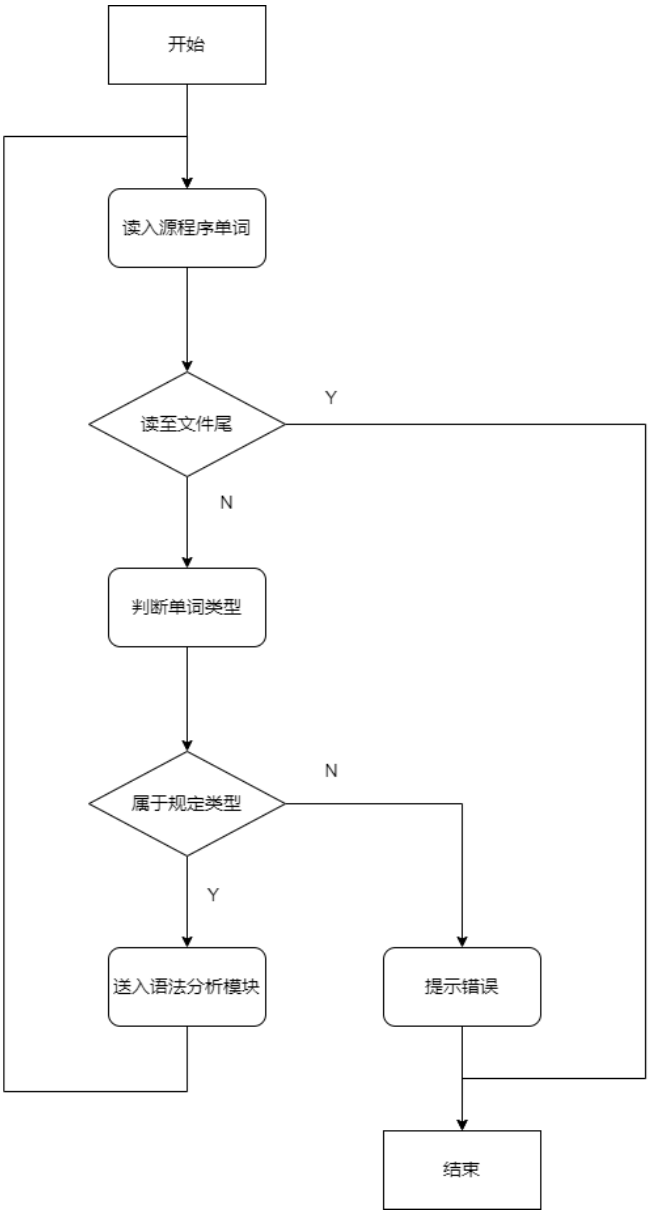
装 订 线



使用循环与分支配合就可以实现该自动机。如图，处于状态 0 时，根据下一个读入的字符就可以转移至不同的状态，并且彼此间没有交织，也就是说，根据单词的首字符就能决定单词的类别，后续读入的字符只是验证或拓展（>和>=的关系），一旦读入了不合预期的字符，就认为读到了未定义字符，比如以数字开头的单词中存在字母等情况。循环结束后，根据所处的分支能够判断出单词的类别，赋予元组对应的种别与单词本身的值，并将其送入语法分析模块。

词法分析工作时，每次从文件中取一个单词，并通过自动机模型判断其类别，直到读到文件尾为止。其工作过程可用如下流程图来表示：

装 订 线



3.2 语法分析

3.2.1 数据结构设计

- Production

用 Production 结构来表示文法的一条产生式。类 C 源程序的文法是上下文无关文法，产生式的左部只有一个非终结符变元，右部包含终结符或非终结符组成的若干变元。用 string 来表示变元，则 Production 可定义为如下形式：

```
struct Production
{
    string left;          // 左部
    vector<string> right; // 右部
};
```

右部用一 vector 数组来表示若干变元组成的符号串。

- LR1Item

本编译器使用一遍扫描的 LR(1)分析法进行语法分析。LR(1)分析法会将产生式拓展成若干 LR(1)项目：在产生式右部的某个位置加上一个点，并且引入一个向前查看符号来解决移进-规约冲突。因此 LR(1)项目的定义如下：

```
struct LR1Item
{
    Production production; // 产生式
    int dotIndex;          // 点的位置
    string lookahead;      // 向前查看符号
};
```

- FIRST 集

非终结符的 FIRST 集是非终结符可推导出的所有句子中开头的终结符的集合，形式化定义如下：

$$FIRST(V) = \{a \mid V \overset{*}{\Rightarrow} a \ldots\}$$

因此可用如下形式表示 FIRST 集：

```
map<string, set<string>> First;
```

键 string 表示非终结符，值 set 表示终结符的集合。

- 状态转换函数 GO

GO 函数的定义是：$GO(I,X) = CLOSURE(J)$，其中 I、J 表示项目集，X 表示文法符号，表示 I 可以通过 X 转移至 J。则 Go 的数据结构表示如下：

```
struct Go
{
    int pre;
    int next;
```

```
    string changeStr;
};
```

其中 pre 表示 I，next 表示 J，changeStr 表示 X。

- ACTION 表

$ACTION[s,a]$的定义是：当状态 s 面对输入符号 a 时，需要采取的动作。动作包括 4 种：移进、规约、接受和报错。其中，状态 s 指项目集的序号，输入 a 指词法传来的单词符号；移进操作有第二个参数，表示移进状态的序号；规约操作也有第二个参数，表示用来规约的产生式序号。综上，ACTION 表的定义如下：

```
vector<map<string, pair<int, int>>> ACTION;
```

其中，vector 数组表示每个项目集对应一个 ACTION 表表项，map 的键 string 表示输入符号 a，值 pair 表示动作类型与第二个参数值。

- GOTO 表

$GOTO[s,X]$的定义是：当状态 s 面对文法符号 X 时，下一状态是什么。这里的文法符号 X 可以是终结符，也可以是非终结符。GOTO 表的定义如下：

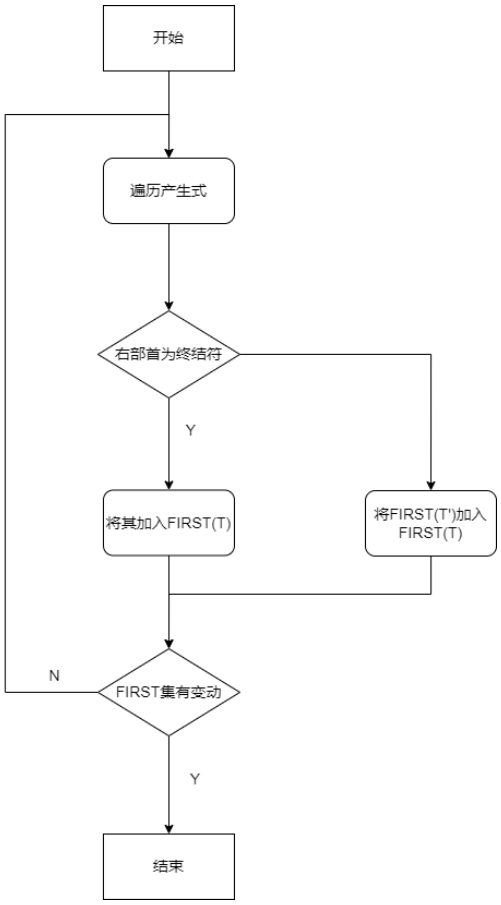
```
vector<map<string, int>> GOTO;
```

其中，vector 数组表示每个项目集对应一个 GOTO 表表项，map 的键 string 表示文法符号 X，值 int 表示下一状态的序号。

3.2.2 主要算法

- 计算 FIRST 集

需要计算所有非终结符的 FIRST 集。算法是不断循环并尝试修改 FIRST 集，如果在某次循环没有修改，则说明计算完毕，退出循环。在循环中，遍历所有的产生式，如果产生式的右部首为终结符，则将其加入左部 T 的 FIRST 集；如果右部首为非终结符 T’，则将 FIRST(T’)加入 FIRST(T)。FIRST 集使用的是 set 容器，对于重复的值，set 不会重复添加，借此可以知道原先集合中是否含有某值，以判断 FIRST 是否被修改，避免死循环。算法可表示为如下流程图：



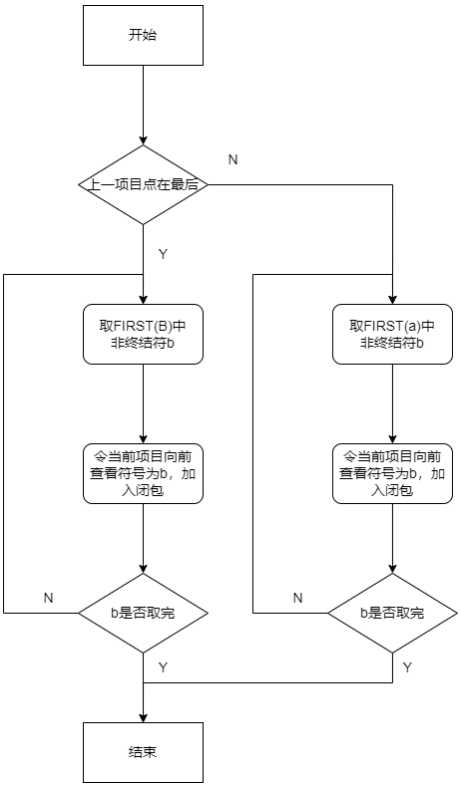
- 计算闭包

闭包是对于项目集而言的。对于项目集 I，I 的所有项目属于闭包，并且如果项目$[A \to \alpha \cdot B\beta, a]$属于闭包，那么对于 B 的每个产生式$B \to \gamma$与$FIRST(\beta a)$中的每个终结符 b，项目$[B \to \cdot \gamma, b]$也属于闭包。

将闭包的计算分为两种情况：一种是项目已经在当前闭包中；另一种是项目不在当前闭包，从另一闭包或外部转移过来的。

对于内部算闭包，当前项目的向前查看符号由上一项目决定。如果上一项目的点不在最后，令上一项目点后的文法符号为 B，对于$FIRST(B)$中的所有非终结符 b，令当前项目的向前查看符号为 b，并加入闭包；如果上一项目的点在最后，令上一项目的向前查看符号为 a，对于$FIRST(\alpha)$中的所有非终结符 b，令当前项目的向前查看符号为 b，并加入闭包。

上述过程可用如下流程图来表示：



对于外部算闭包，将当前项目加入闭包，如果当前项目点后是非终结符 T，则遍历项目集，将所有点处于开头，并且左部等于 T 的项目加入闭包，此处调用 inclosure，上一项目参数即为当前项目。

- 计算 Go 表

对于 Go 转移关系的计算，主要参考了课本 P.115 的构造算法：

关于文法 G′的 LR(1)项目集族 C 的构造算法是：

```
BEGIN
    C: = {CLOSURE({[S′→·S, # ]})};
    REPEAT
        FOR   C 中的每个项目集 I 和 G′的每个符号 X      DO
            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
```

首先遍历所有已存在的项目规范集，随后遍历每个符号，查找是否存在 dot 后为当前符号的项目，找到后该项目进入 outclosure 外部计算闭包。注意到，这样产生的闭包可能会有重复，无法正确获得项目集的包含关系和转移关系，所以在每一个每一

装 订 线

族生成一族新的规范集后，都要进行是否属于已存在项目集的操作。

检查是否属于的过程中，如果确实存在，则将转移关系的 next 赋值为已存在的项目集号，同时，删除新计算得到的闭包和项目集。如果不属于已存在的项目集，则正常关系进栈即可，无需其他操作。一直循环计算到项目集规范族不再增大为止。

- 构造 ACTION 表和 GOTO 表

  遍历项目集族的每个项目集中的每个产生式，按照如下规则进行操作：

- 如果项目$[A \to \alpha \cdot a\beta, b]$属于项目集 k，且$Go(k, a) = j$，a 为终结符，则置 ACTION[k, a]为”sj”，这里 s 指移进，j 指下一状态。
- 如果项目$[A \to \alpha \cdot, a]$属于项目集 k，则置 ACTION[k, a]为”rj”，这里 r 指规约，j 指该项目对应产生式的序号。
- 如果项目$[S' \to S \cdot, \#]$属于项目集 k，则置 ACTION[k, #]为”acc”，表示接受态。
- 如果 Go(k, A)=j，则置 GOTO[k, A]=j
- 其余栏均为空白，表示出错。

之前的一切操作都属于语法分析任务的初始化，有了 ACTION 表和 GOTO 表就可以开始语法分析的工作。

- 分析过程

由于语法分析与语义分析是同时进行的，因此在语义分析处一并阐述语法分析的工作原理。

3.3 语义分析与中间代码生成

3.3.1 数据结构设计

1. State

用 State 结构来表示语法/语义分析过程中出现的文法符号，可以指代终结符和非终结符，其定义如下：

```
struct State
{
    string place;
    vector<int> true_list;
    vector<int> false_list;
```

```
    vector<int> next_list;
    int next_quad;
    int line_count;
};
```

place：在属性文法中，有的变元需要拥有一个“名字”，比如一个表示数字的变元“num”，num 是变元的名称，而它表示的是“1”这个值。

true_list：在涉及布尔表达式的产生式中使用，表示变元的真出口链表。

false_list：同 true_list，表示变元的假出口链表。

next_list：表示语句块内需要进行跳转的四元式的编号链表。

next_quad：用于记录某一位置的待产生四元式的编号。

line_count：表示变元对应的原标识符在源代码中的行数，由词法分析提供，在语义检查出错时用于提示错误位置。

2. Code

Code 结构表示中间代码，是四元式形式。

```
struct Code
{
    string op;
    string src1;
    string src2;
    string dst;
};
```

op：表示指令的运算符。

src1：表示源操作数 1。

src2：表示源操作数 2。

dst：表示目标。

3. FuncInfo

FuncInfo 表示函数的信息。由于本编译器需要处理过程调用语句，因此需要记录下来函数的信息。FuncInfo 的定义如下：

```
struct FuncInfo
{
    string type;
```

```
    string name;
    int startAdd;
};
```

type：函数的返回类型，有 int 和 void 两种。

name：函数名称。

startAdd：函数的第一条中间代码的序号，用以生成表示函数的标签。

3.3.2 主要算法

1. 规约分析

LR(1)分析法使用 ACTION 表和 GOTO 表，借助栈来进行分析。定义两个栈，一个栈存放整数，数代表状态的编号，用于语法分析，令其为 S1；一个栈存放 State，State 表示变元，用于语义分析，令其为 S2。

初始，S1 将 0 进栈，0 表示初始状态，S2 栈为空。在分析过程中，会不断地从词法分析器那里取得文法符号，这些文法符号都是终结符。假设当前 S1 栈顶地状态为 k，传入的文法符号为 a，那么考虑 ACTION[k, a]的值。如果是 acc 或是空白，则接受源程序或者出错，那么分析都会结束；否则分为以下两种情况：

如果 ACTION[k, a]是”si”，表示当前需要进行移进操作，将 i 压入 S1，将 a 构造成 State 压入 S2，构造的方法是令 State 的 place 值等于 a 的属性值，即其原本的字符串表示。

如果 ACTION[k, a]是”rj”，表示当前需要进行规约操作，先从文法表中找到序号 j 对应的产生式 P，令其左部的非终结符为 L，右部文法符号的数量为 n。对于 S1，弹出栈顶的 n 个状态，找到 GOTO 表中 L 对应的整数 s，将 s 压入栈中。语法分析需要完成的就是这个任务，重复这个过程直到接受/报错或输入完毕，对于最后者，如果此时 S1 栈不空，则认为源程序语法存在错误。

对于语义分析，在规约操作时还需要完成更多的任务。此时已经通过 ACTION 表得到了产生式，那么此时 S2 栈顶的 n 个变元就是该产生式右部的 n 个变元，语义分析需要通过产生式的种类以及这 n 个变元中存储的信息来完成中间代码的生成工作。下面对于不同的产生式依次做讨论。

3.3.3 功能函数（针对不同产生式）

- 一般的产生式

语法分析的产生式一共有 70 多条，我们只需要对其中的 30 多条做处理，利用 map 记录它们的序号，其余的产生式我们称为“一般的产生式”。对于这些一般的产生式，其规约出的父变元在属性文法上是没有任何意义的，它的存在只是为了在规约时保证栈顶的若干变元与产生式右部一致。由于许多一般产生式的右部只有一个变元，因此在每次规约前将栈顶的变元赋值给一个新的 State，记为 fa，表示产生式的左部，在规约完成后将栈顶相应个数的变元弹出，再将 fa 压入栈。这样的好处是同时可以处理只需要进行变元继承的产生式，如 Term->Factor，按之前的处理可以将 Factor 的内容直接继承给 Term，省去了更多的判断。

- M 产生式

  产生式形式：M -> ϵ

  翻译模式：{M.quad = nextquad}

  由于 M 生成空，因此在文法中添加 M 并不会影响正常的规约过程。M 的作用是记录此时下一条或者说将要生成的四元式的序号，在布尔表达式的回填操作中要使用到。

- N 产生式

  产生式形式：N -> ϵ

  翻译模式：{ N.nextlist:=makelist(nextquad);emit(j,-,-,-) }

  N 同样生成空。该翻译模式生成了一条跳转语句，用来控制 N 之前的语句块执行到此处时需要跳转至的位置。

- 声明产生式

  产生式形式：Parameter -> int idt

这个产生式的功能是声明变量，由于我们只考虑了 int 类型的数据，因此这里是 int 而非 Ftype。声明语句可能在函数的形参中出现，也可能在代码块的前半部分 InternalDeclaration 中出现。规约这条产生式时，需要记录声明了什么变量，为此，需要一个 string 类型的数据，来记录源代码声明了哪些变量。由此，可以对源代码中

出现的所有标识符进行判断，如果标识符出现在声明语句中，且该标识符已经被声明过，那么就出现了重定义的错误；如果标识符不在声明语句中且被使用，那么就出现了未定义的错误。

- 语句合并产生式

产生式形式：StatementSequence -> StatementSequence M Statement

对应课本中给出的属性文法：L -> L1 M S

翻译模式：{ backpatch(L1.nextlist, M.quad); L.nextlist:=S.nextlist }

这里要用 M.quad 回填 L1.nextlist，也就是将 L1 的跳转出口指定至 S 的入口，并将 S.nextlist 继承给 L，这样就实现了两段代码块的合并。

- if 表达式产生式

产生式形式：IfStatement -> if (BoolExpression) M StatementBlock

对应课本中给出的属性文法：S→if E then M S1

翻译模式：{ backpatch(E.truelist, M.quad); S.nextlist:=merge(E.falselist, S1.nextlist) }

这里按照课本给出的翻译模式进行翻译即可，将布尔表达式的真链出口指定至 M 记录的序号，并将布尔表达式的假链出口和 S1 语句块的出口合并，作为整个 if 表达式的出口。值得一提的是 if 和 while 语句的表达式使用了非终结符 BoolExpression 而不是 Expression，作这个区分的原因是 if、while 等语句中的表达式最终是不需要通过一个中间变量来记录它的值的，只需要通过真假链进行跳转即可，而参与运算的 Expression 是需要一个变量来记录它的值的，可能是声明的变量，也可能是临时记录的中间变量。这样进行区分之后，就可以避免 if、while 等语句的布尔表达式在翻译时生成冗余的四元式。

if-else 表达式在此基础上多加了一些操作，但基本思想一致，这里不再赘述。

- while 表达式产生式

产生式形式：WhileStatement -> while M (BoolExpression) M StatementBlock

对应课本中给出的属性文法：E -> while M1 E do M2 S1

翻译模式：{backpatch(S1.nextlist, M1.quad); backpatch(E.truelist, M2.quad); S.nextlist:=E.falselist; emit(‘j,－,－,’ M1.quad) }

简单想象一下 while 表达式的执行过程，就可以看懂 while 的翻译模式。用 M1.quad 回填 S 语句块的 nextlist，即语句块跳出后重新判断布尔表达式；用 M2.quad 回填布尔表达式的 truelist，即判断布尔表达式为真后，执行 S 语句块；布尔表达式的假出口就是整个 while 表达式的出口；并在最后添加一条跳转语句，跳转至判断布尔表达式的位置，在语句块正常执行完未发生跳转的情况下，进行循环。

- 赋值表达式产生式

产生式形式：AssignmentStatement -> idt = Expression ;

翻译模式：{emit(:=, Expression.place, _, idt.place)}

这里很好理解，就是生成将 Expression 赋值给 idt 的语句的四元式。但这里有两点需要注意的情况：

a) 需要判断等号左端标识符是否为未定义标识符的问题。

b) 需要保证等号右端 Expression 必须有 place 的问题，这主要涉及对布尔表达式的处理。举个例子，对于赋值语句 i=a>b; ，之前我们提到过，不希望布尔表达式生成冗余的中间变量，因此将 a>b 语句规约至 Expression 时，这个表示 Expression 的变元是没有 place 的，因此我们需要在生成赋值语句之前做一次检查，保证 Expression 变元有 place，如果没有，就在此处生成一个中间变量，这种情况只有在涉及到布尔表达式的时候才会出现，因此赋予中间变量就是将布尔值转化为 0/1 值的问题。实现这个功能需要四条四元式，两条赋值语句，为一个新的中间变量分别赋值为 1 和 0，并且用各自的序号回填 Expression 的 truelist 和 falstlist；两条无条件跳转语句，分别跟在两句赋值语句之后，作为 Expression 新的 truelist 和 falselist。代码如下

```
s.place = "v" + to_string(vars.size());
vars.push_back(s.place);
int t = next_quad();
emit(":=", "1", "", s.place);
emit("j", "", "", "0");
int f = next_quad();
emit(":=", "0", "", s.place);
emit("j", "", "", "0");
backpatch(s.true_list, t);
backpatch(s.false_list, f);
```

```
s.true_list = {t + 1};
s.false_list = {f + 1};
backpatch(s.true_list, next_quad());
backpatch(s.false_list, next_quad());
```

- 逻辑或表达式产生式

  产生式形式：OrExpression -> OrExpression || M AndExpression

  对应课本中给出的属性文法：E→E1 or M E2

  翻译模式：{ backpatch(E1.falselist, M.quad); E.truelist:=merge(E1.truelist, E2.truelist); E.falselist:=E2.falselist }

  在这里可以更明显地体会到，布尔表达式不生成四元式的思想。利用 M 来记录 E2 的首地址，E1 判断失败后，用 M.quad 回填其 falselist，整个表达式的 truelist 是 E1 和 E2 的 truelist 的并集；falselist 是 E2 的 falselist，这与 or 的定义相符。

- 逻辑与表达式产生式

  产生式形式：AndExpression -> AndExpression && M NotExpression

  对应课本中给出的属性文法：E→E1 and M E2

  翻译模式：{ backpatch(E1.truelist, M.quad); E.truelist:=E2.truelist; E.falselist:=merge(E1.falselist,E2.falselist) }

  与的处理类似。E1 判断成功后需要继续判断 E2，用 M.quad 回填其 truelist，整个表达式的 truelist 是 E2 的 truelist；falselist 的 E1 和 E2 的 falselist，与 and 的定义相符。

- 逻辑非表达式产生式

  产生式形式：NotExpression -> ! CompareExpression

  对应课本中给出的属性文法：E → not E1

  翻译模式：{ E.truelist:=E1.falselist; E.falselist:=E1.truelist}

  这里将 E1 的 truelist 和 falselist 交换后继承给 E，也就是完成逻辑取反的任务。

- 布尔表达式产生式

  产生式形式：CompareExpression -> AdditiveExpression

  对应课本中给出的属性文法：E → id

翻译模式：{ E.truelist:=makelist(nextquad); E.falselist:=makelist(nextquad+1); emit(‘jnz’ ‘,’ id .place ‘,’ ‘－’ ‘,’ ‘0’)；emit(‘ j, -, -, 0’) }

这里对应的是将加法表达式直接作为布尔表达式使用的情况，比如 if(a)中的 a。由于此时这里的加法表达式并未经过与或非以及比较表达式，因此该变元是没有真假链的，所以我们需要在这里进行补充，就按照如上的翻译模式，创建真链和假链，并生成两条跳转语句，分别与真链和假链对应。

- 比较表达式产生式

  产生式形式：CompareExpression -> Expression relop AdditiveExpression

  对应课本中给出的属性文法：E → id1 relop id2

  翻译模式：{ E.truelist:=makelist(nextquad); E.falselist:=makelist(nextquad+1); emit(‘j’ relop.op ‘,’ id 1.place‘,’ id 2.place ‘,’‘0’); emit(‘j, -, -, 0’) }

这里会为新生成的变元生成真假链，并生成两条跳转语句与真假链对应，根据比较表达式的结果决定跳转目的地。需要注意的是，这里与赋值表达式相同，需要 id1 与 id2，也就是 Expression 和 AdditiveExpression 都具有 place 属性，因为这里涉及值的比较，如果 Expression 从布尔表达式规约上来，需要按照真为 1、假为 0 的规则为布尔表达式赋予中间变量，用于进行比较。

- 运算表达式产生式

  产生式形式：OperatorTerm -> Factor operator Factor

  对应课本中给出的属性文法：E → id1 operator id2

  翻译模式：{ emit(operator, ‘id1.place’, ‘id2.place’, newtemp) }

这里对应加减乘除四条产生式，它们的规约规则相同，在文法中通过不同非终结符来决定它们的优先级，乘除高于加减。只需生成一条运算语句，将 id1 和 id2 的运算结果给到一个新的中间变量即可。

- 标识符产生式

  产生式形式：Factor -> idt

在这条产生式规约时，需要检查标识符是否出现过，如果标识符未出现过，需要提示使用未声明变量的错误。

- 函数定义产生式

  产生式形式：Program -> Type M idt (FormalParameters) M StatementBlock

  这条产生式用于记录函数的信息，Type 对应函数的返回值类型，idt 对应函数名称，第一个 M 的 next_quad 记录了函数的第一条中间代码的地址。将这些信息记录在 FuncInfo 的数组中。

- 实参调用产生式

  产生式形式：Expressions -> Expressions Expression 和 Expressions -> Expression

  这两条产生式在函数调用时给出实参时使用，在这里要将 Expression 的 place 放入 Expressions 的 param_list 存储起来，在之后一起生成 par 指令。不能直接生成 par 指令的原因是，实参可能是函数的值，那就需要先为这个函数调用生成中间代码，用其返回值作为实参，如果在这里直接生成 par 指令，就无法得知之前生成的 par 指令属于外层函数还是内层函数，因此要后置输出。

- 函数调用产生式

  产生式形式：Factor -> M idt FTYPE

  函数调用操作需要生成 call 指令，对象是 idt 的 place，即函数名。在之后如果函数有返回值，需要为返回值生成一条独特的赋值语句。在 Mips 汇编中，返回值存入特定的寄存器，因此这里可以使用一个特殊的标识符如“@”来表示返回值，将其赋值给一个临时变量，并将 Factor 的 place 赋值为临时变量的值，这样就能让返回值参与后续计算。

- return 产生式

  产生式形式：ReturnStatement -> return ;, ReturnStatement -> return Expression ;

  中间代码的形式也是 return，如果有返回值，则用 Expression 的 place 作为返回内容。

- get 产生式

  产生式形式：Parameters -> Parameters Parameter, Parameters -> Parameter

  这里是形参声明的变量，目标代码需要知道哪些变量是形式参数，以分配内存，因此生成 get 语句来告知，规约时生成 get Parameter.place 即可。

以上就是全部需要生成中间代码的产生式。语法、语义分析的任务到此为止，生成的所有中间代码存储在 codes 数组中，函数信息存储在 FuncInfo 数组中，接下来就可以开始生成目标代码。

3.4 目标代码生成

3.4.1 数据结构设计

- 活跃、待用信息

活跃与待用信息是四元式中的变量的信息，随着四元式变化而变化，不同四元式中相同变量的活跃与待用信息不同。

活跃信息指该变量在之后是否要被用到，是一个 bool 值；待用信息指该变量在之后的哪句中间代码最先被用到，如果为-1，代表不活跃，否则为一个正整数。用 pair<int, bool>来表示<待用信息，活跃信息>，命名为 info。

- 带信息的中间代码

带信息指为中间代码中的变量附加上了待用与活跃信息，可用如下形式来表示：

```
struct info_code
{
    int num;
    info dst, l, r;
};
```

num：四元式的索引。

l、r、dst：分别表示源操作数 1、源操作数 2、目的操作数的信息。

- 基本块 Block

在生成目标代码前，需要先将中间代码划分成若干基本块。基本块的定义如下：

```
struct Block
{
public:
    vector<info_code> icodes;
    int next1; // 跳转
    int next2; // 后继
};
```

icodes：带有信息的中间代码数组，表示基本快内的若干中间代码。

next1：如果基本块以跳转语句为结尾，则 next1 表示跳转至的基本块的序号，否则为-1。

next2：表示直接跟在该基本块后的基本块序号。

- 目标代码 ObjectCode

ObjectCode 表示目标代码，遵循 Mips 格式。其定义如下：

```
struct ObjectCode
{
    string op;
    string src1;
    string src2;
    string src3;
};
```

op：表示指令的运算符。

src1、src2、src3：表示三个操作数。

- 目标代码生成器：ObjectCodeGenerator

ObjectCodeGenerator 是一个数据和方法的集成类，这里介绍它的数据成员。

```
class ObjectCodeGenerator
{
public:
    static const int regs_num = 32;
    map<string, string> jmp_map;
    map<string, vector<Block>> funcBlocks;
    map<string, vector<map<string, vector<info>>>> funcSymbol;
    map<string, set<int>> Avalue;
    map<int, set<string>> Rvalue;
    map<string, vector<set<string>>> funcOut;
    map<string, vector<set<string>>> funcIn;
    string nowFunc;
    FuncInfo nowFuncInfo;
    set<int> freeReg;
    vector<Block>::iterator nowBlock;
    vector<info_code>::iterator nowCode;
    vector<ObjectCode> objCodes;
    map<string, int> varOffset;
    vector<pair<string, bool>> parList;
```


装 订 线

```
    int top;
};
```

jmp_map：目标代码的转移语句用的是 j L1 这种标签的形式，该 map 的功能是将中间代码的地址映射到标签。

funcBlocks：为每个函数安排一个基本块数组，以函数名为索引。

funcSymbols：为每个函数的每个基本块安排一个符号表，符号表以变量的名字为索引，里面记录了变量从基本块内最后一句代码向前的信息变化链，其构造和使用方法后续介绍。

Avalue：Avalue 记录了变量的值位于哪些地方，包括寄存器和内存。

Rvalue：Rvalue 记录了寄存器内的值是哪些变量的值。

funcOut：为每个函数的每个基本块安排一个 out 表，表中记录了出基本块后哪些变量是活跃的。

funcIn：为每个函数的每个基本块安排一个 in 表，表中记录了进基本块前哪些变量在内存中有值存储。

nowFunc：记录当前处理的函数名。

nowFuncInfo：记录当前处理的函数的信息。

freeReg：记录哪些寄存器是空闲的。

nowBlock：记录当前处理的基本块。

nowCode：记录当前处理的四元式。

objCodes：存储生成的目标代码。

varOffset：记录变量在内存中的位置。

parList：记录函数调用时的参数列表。

top：记录当前栈顶指针的值。

3.4.2 主要算法

- 划分基本块

首先挑选出入口语句，作为每个基本块的起始语句，挑选入口语句遵循如下原则：

1) 函数的第一条语句

2) 能由跳转语句转移到的语句。

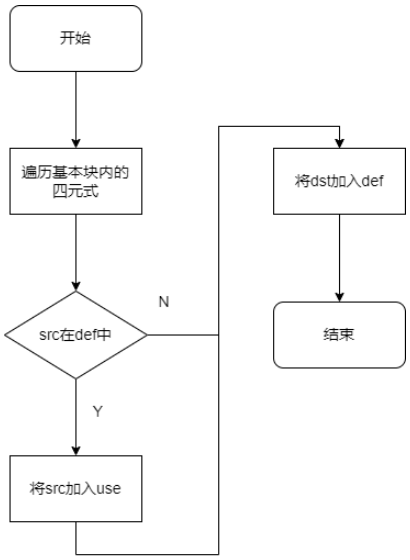
3) 紧跟在条件转移语句后面的语句。

每个入口语句到下个入口语句（不包括）或下个转移语句（包括）之间的语句构成基本块，每一函数的入口语句到下一函数的入口语句（不包括）之间的基本块属于该函数。这样就能为每个函数划分好基本块。

- 分析基本块

这一步骤的任务是确定每一条语句中变量的活跃与待用信息，以及每一基本块的 in 和 out 表。以函数为单位，各函数之间互不干涉。

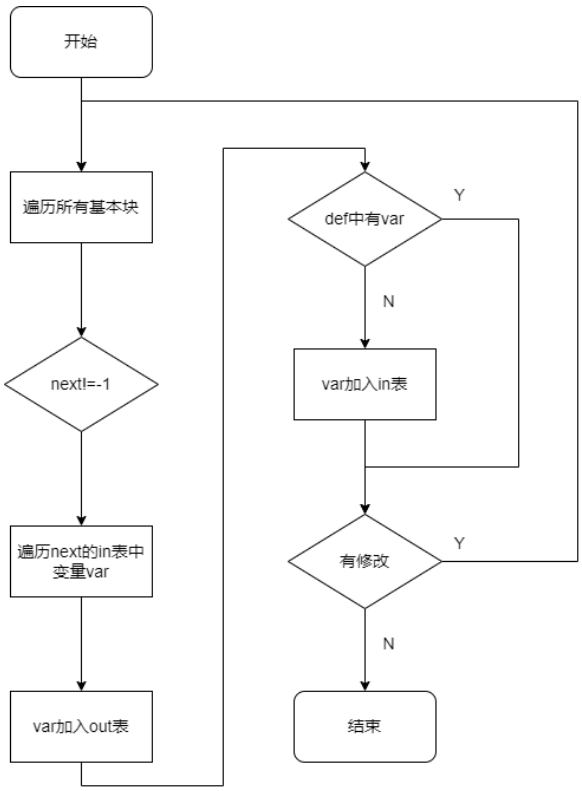
首先构建 in 表和 out 表，为此需要先明确 def 和 use 集合。def 与 use 是在基本块层面上的集合，def 记录了基本块内有哪些变量在使用前就被赋值了，use 记录了基本块内有哪些变量在使用时还未赋值。这需要按顺序遍历基本块内的所有四元式，对于源操作数，如果在 def 集合中没有该变量，则加入 use 集合；对于目的操作数，加入 def 集合。可用如下流程图表示：



显然，use 集合属于 in 集合，由于它们在使用前未被赋值，因此这些变量在进入基本块时在内存中需要有值，否则无处取到它们的值，因此将 use 集合加入 in 集合。再按照如下办法构造 in 表和 out 表。

对于每个基本块，如果其 next1 或 next2 不为-1，表示该基本块存在后继基本块，遍历后继基本块的 in 表中的变量，将其加入本基本块的 out 表，表示该变量出本基本

块后活跃，并且如果本基本块的 def 集合中没有该变量，即该变量在本基本块内未被赋值，则将该变量加入本基本块的 in 表。重复这一过程直到所有基本块的 in 表和 out 表均不再改变。可用如下流程图来表示：



之后确定基本块中各变量的活跃与待用信息。对于每个基本块，给其 out 表中的变量赋予{-1, true}的信息，true 表示出基本块后活跃，-1 表示在该块内无待用。随后从后向前遍历四元式，对于源操作数，将变量的信息附加到四元式上，并将变量的信息改为{i, true}，其中 i 是四元式的序号；对于目的操作数，将变量的信息附加到四元式上，并将变量的信息改为{-1, false}。

至此完成了基本块的分析任务。

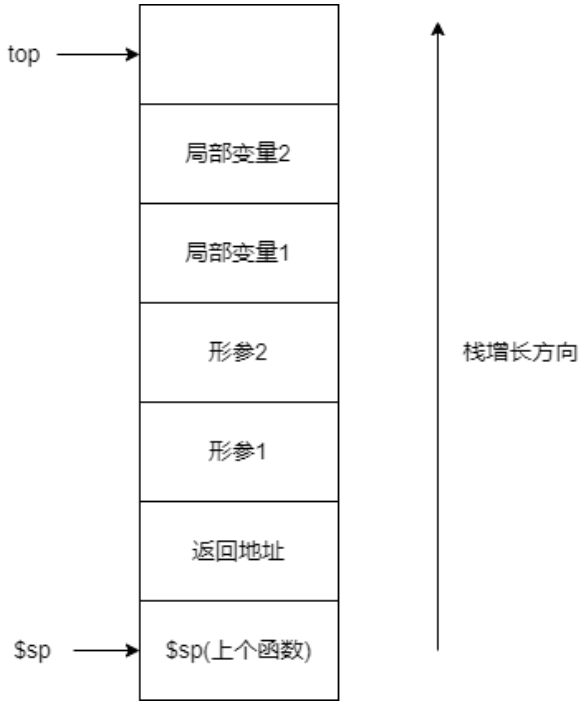
- 栈空间的结构

  由于本编译器涉及过程调用的处理，因此需要先明确有关函数调用的栈空间的结构。

每个函数都享有内存中的一段连续空间，使用栈的方式来管理内存。整数为 4 字节，4 字节的内存空间表示存储一个整数。使用$sp 寄存器来表示栈底指针，$sp 指针

装 订 线

指向的内存中存储的是上个函数的$sp 值。4($sp)存储的是函数的返回地址。从 8 开始的若干个内存空间保存的是函数的参数的值，再接下来的若干个内存空间属于函数内部定义的局部变量，栈顶指针用 top 来表示。可用下图表示函数栈空间的结构：



- 目标代码的生成

  先考虑单个四元式的目标代码。

1) 寄存器：Mips 指令的源操作数必须是寄存器或立即数的形式。对于变量，不能直接让内存中的存储值直接参与运算，只能先将值装载到寄存器中。为此，设计了两个寄存器分配函数 alloc_reg 和 get_reg，分别供源操作数和目的操作数使用。这样对于任何要用到寄存器的 Mips 指令，都可以调用这两个函数来获取分配到的寄存器。
2) 跳转语句：中间代码给出的跳转目标是四元式的地址，在划分基本快的过程中已经建立好了四元式到基本块标签的映射关系，因此使用 jmp_map 获取跳转地址即可。
3) 赋值语句：赋值语句不需要生成目标代码，只需要修改 dst 的 Avalue 以及源操作数寄存器的 Rvalue 值即可。
4) par 语句：par 四元式声明实参列表，处理时将变量存入 parList 即可。

5) call 语句：处理过程调用语句时需要完成如下任务：将 parList 中的变量压栈，内存地址从 8($sp)开始、将$sp 的值存入内存 0($sp)、移动$sp 到$sp+top*4 的位置、用 jal 语句完成过程调用、以及装载回$sp 的值，这步是从函数中返回时执行的。

6) return 语句：如果有返回值，需要将返回值存入$2 号寄存器，然后从 4($sp)中取出返回地址的值存入$ra，使用 jar 指令回到函数被调用之前的位置。

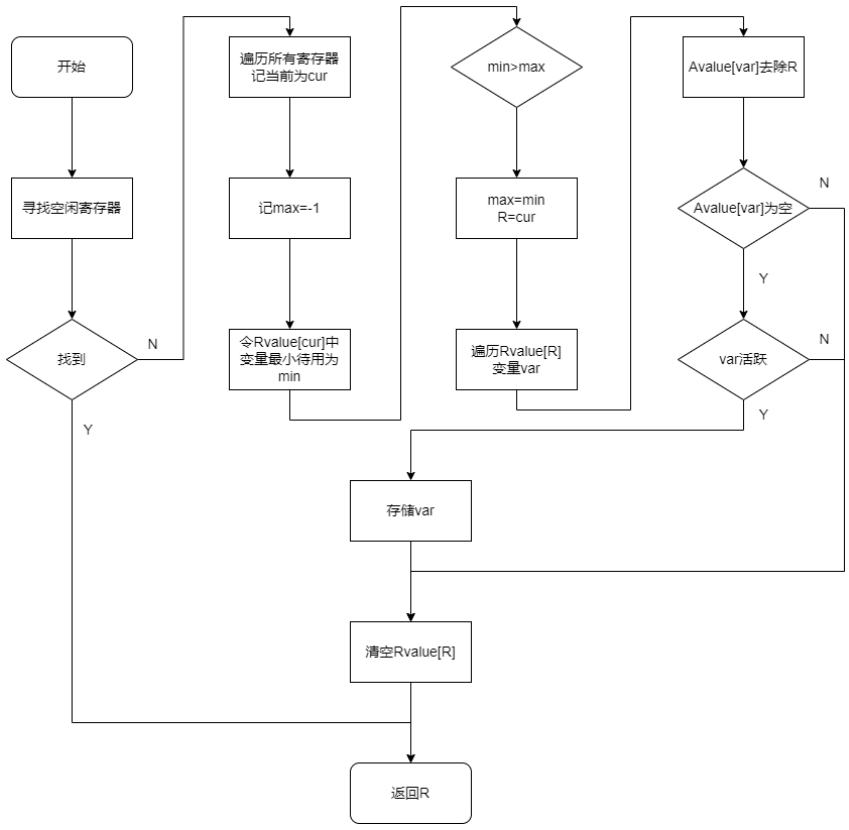
3.4.3 功能函数

- 无参 alloc_reg 函数

该函数的功能是分配一个寄存器给操作数。首先寻找有无空闲寄存器，如果有直接返回即可。如果没有则需要考虑抢占。

遍历所有寄存器，令当前处理的寄存器为 R，考虑 Rvalue[R]中的变量，首先希望所有变量都在除该寄存器之外的地方有值的拷贝，这样将寄存器不会对任何变量有影响。

如果达不成这个条件，再希望这些变量越远用到越好，即希望它们的待用信息尽可能大。寄存器最近一次被使用取决于待用信息最小的变量，即我们需要找出最小待用信息最大的寄存器，令其为 R。对于 Rvalue[R]中的变量，需要将 R 从 Avalue 中去除，如果 R 是该变量的唯一存储位置，则考虑是否要为该变量生成存数指令。如果该变量在基本块内还需要被使用，且使用前未被赋值，或该变量在基本块的 out 表内，则需要为该变量生成存数指令。最后将 Rvalue[R]清空，因为它存储的值即将改变，不再是这些变量的值。可用如下流程图来表示：



- 有参 alloc_reg(src)函数

该函数的功能是获取一个装有 src 值的寄存器。先遍历 Avalue[src]集合，如果有寄存器则直接返回，没有则调用无参 alloc_reg 获取一个寄存器，生成一条取数指令 lw，将 src 的值从内存放入寄存器，再更新 Avalue 和 Rvalue 值即可。

- get_reg 函数

这个函数专用于给三操作数的四元式的 dst 分配寄存器。参考的是书上的算法：

首先考虑如果存在某个独属于 src1 的寄存器，并且 src1 与 dst 相等或 src1 不活跃，则可以将该寄存器分配给 dst；否则使用无参 alloc_reg 函数为 dst 分配寄存器，并更新 Rvalue 与 Avalue 值。

- store_var 函数

store_var 函数的功能是生成存数指令，将变量的值从给定寄存器存进内存。在 varOffset 中记录了变量的地址（相对于$sp 的偏移量），如果 varOffset 中没有记录变量地址，则将 top 指向的 4 个字节空间分配给该变量，记录进 varOffset，并将 top 的值+4。使用 sw 指令生成存数指令。

- release_var 函数

  该函数的功能是释放变量，对于 Avalue 中的寄存器，从其 Rvalue 中剔除变量，并清除变量的 Avalue 值。及时清除变量能够腾出更多的寄存器，减少抢占发生的次数。

四、 调试分析

4.1 测试数据

使用《编译原理课程设计任务书》中给出的含过程调用的程序实例作为测试源程序，并且改掉了原本的死循环，检查生成的中间代码与目标代码的正确性，并将源程序改成错误的，测试编译器能否正确检查出错误。源程序如下：

```
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;
       j = i;
    }
    return i;
}
int demo(int a)
{
    a = a + 2;
    return a * 2;
}
int main(void)
{
    int a;
    int b;
    int c;
    a = 3;
    b = 4;
    c = 2;
```

```
    a = program(a, b, demo(c));
    return 0;
}
```

4.2 测试结果

该源程序生成的中间代码如下：

```
100 : (get,_,_,a)
101 : (get,_,_,b)
102 : (get,_,_,c)
103 : (:=,0,_,i)
104 : (+,b,c,v0)
105 : (j>,a,v0,107)
106 : (j,_,_,112)
107 : (*,b,c,v1)
108 : (+,v1,1,v2)
109 : (+,a,v2,v3)
110 : (:=,v3,_,j)
111 : (j,_,_,113)
112 : (:=,a,_,j)
113 : (j<=,i,100,115)
114 : (j,_,_,118)
115 : (*,j,2,v4)
116 : (:=,v4,_,i)
117 : (j,_,_,113)
118 : (return,_,_,i)
119 : (get,_,_,a)
120 : (+,a,2,v0)
121 : (:=,v0,_,a)
122 : (*,a,2,v1)
123 : (return,_,_,v1)
124 : (:=,3,_,a)
125 : (:=,4,_,b)
126 : (:=,2,_,c)
127 : (par,c,_,_)
128 : (call,_,_,demo)
129 : (:=,@,_,v0)
130 : (par,a,_,_)
131 : (par,b,_,_)
132 : (par,v0,_,_)
133 : (call,_,_,program)
134 : (:=,@,_,v1)
135 : (:=,v1,_,a)
136 : (return,_,_,0)
```

经检查，中间代码与源代码在语义上完全一致。

生成的目标代码如下：

装 订 线

```
j main
program:
sw $ra, 4($sp)
lw $23, 12($sp)
lw $22, 16($sp)
add $21, $23, $22
sw $0, 20($sp)
lw $20, 8($sp)
bgt $20, $21, L1
L0:
j L2
L1:
lw $23, 12($sp)
lw $22, 16($sp)
mul $23, $23, $22
addi $22, $0, 1
add $23, $23, $22
lw $21, 8($sp)
add $20, $21, $23
sw $20, 24($sp)
j L3
L2:
lw $23, 8($sp)
sw $23, 24($sp)
L3:
lw $23, 20($sp)
addi $22, $0, 100
ble $23, $22, L5
L4:
j L6
L5:
lw $23, 24($sp)
addi $22, $0, 2
mul $21, $23, $22
sw $21, 20($sp)
j L3
L6:
add $2, $0, $0
lw $ra, 4($sp)
jr $ra
demo:
sw $ra, 4($sp)
lw $23, 8($sp)
addi $22, $0, 2
add $23, $23, $22
addi $21, $0, 2
mul $20, $23, $21
add $2, $0, $20
lw $ra, 4($sp)
jr $ra
main:
sw $ra, 4($sp)
addi $23, $0, 3
addi $22, $0, 4
addi $21, $0, 2
sw $23, 8($sp)
sw $22, 12($sp)
sw $21, 24($sp)
sw $sp, 16($sp)
addi $sp, $sp, 16
jal demo
lw $sp, 0($sp)
L7:
lw $23, 8($sp)
sw $23, 24($sp)
lw $23, 12($sp)
sw $23, 28($sp)
sw $2, 32($sp)
sw $sp, 16($sp)
addi $sp, $sp, 16
jal program
lw $sp, 0($sp)
L8:
addi $2, $0, 0
lw $ra, 4($sp)
jr $ra
```

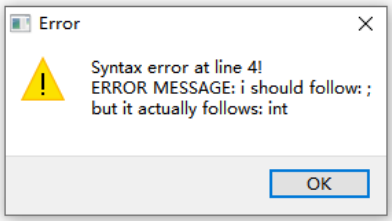
在 Mars 汇编器中运行的结果如下：

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000001
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000003
$s5	21	0x0000000c
$s6	22	0x00000064
$s7	23	0x000000c0
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00000000
hi		0x00000000
lo		0x000000c0

23 号寄存器中是 i 的值，192 是预期结果，说明执行正确。

将源程序的语法改成错的，查看结果：

```
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;
}
int main(void)
{
    int a;
    int b;
    int c;
    a = 3;
    b = 4;
    c = 2;
    a = program(a, b, demo(c));
    return 0;
}
```

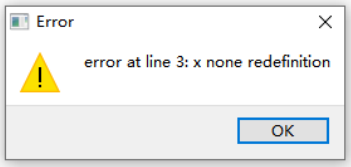


可以看到检测出了没有分号的错误。

装 订 线

使用未定义的变量 x，查看结果：

```
int program(int a, int b, int c)
{
    int i;
    i = x;
    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;
}
int main(void)
{
    int a;
    int b;
    int c;
    a = 3;
    b = 4;
    c = 2;
    a = program(a, b, demo(c));
    return 0;
}
```



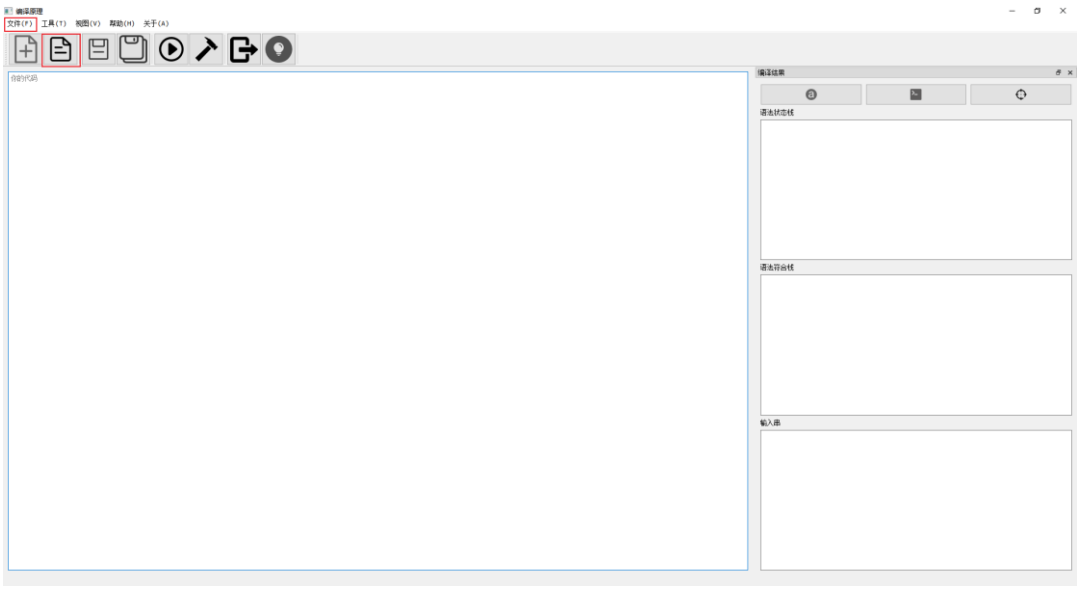
可以看到检测出了这一错误。

4.3 遇到的问题

在完成本项目的过程中，我遇到了如下问题并加以解决：

- 课本上给出的中间代码是不完备的，比如函数的形参，在中间代码处没有体现，但在目标代码需要用到，因此需要给出相应的中间代码。
- 确定变量的活跃信息与待用信息是比较难的任务，我一开始没有按照基本块，而是将全部的中间代码从后向前直接开始分析，这样做不对，因为程序的执行不一定按照代码的顺序，一旦涉及到跳转，按顺序分析就会出错，为此我才想到引入 next1 和 next2，随后一直在研究是否有遍历一遍基本块就确定全部的信息的算法，但始终没有想出来，最后参考 go 表的构建，才写了这样一个一直循环的办法。尽管复杂度很大，但确实解决了问题。

五、 用户使用说明

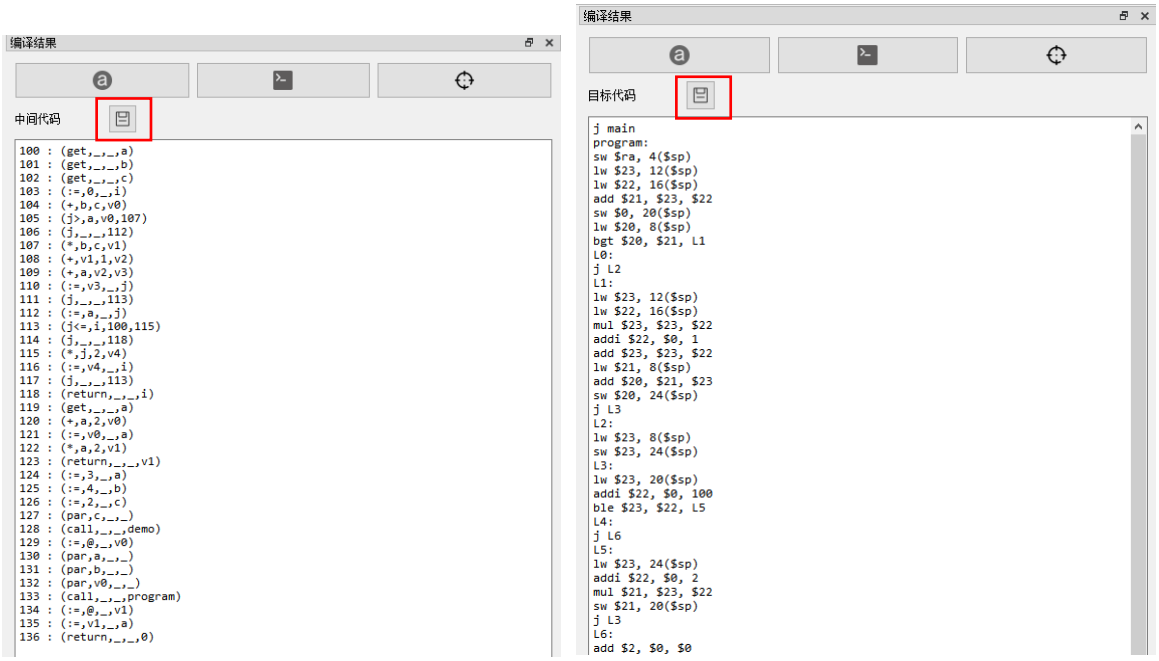


用户可以通过导航栏左侧的文件选项打开文件，也可以通过点击按纽栏第二个按钮来打开文件，也可以在左侧直接输入源程序代码。

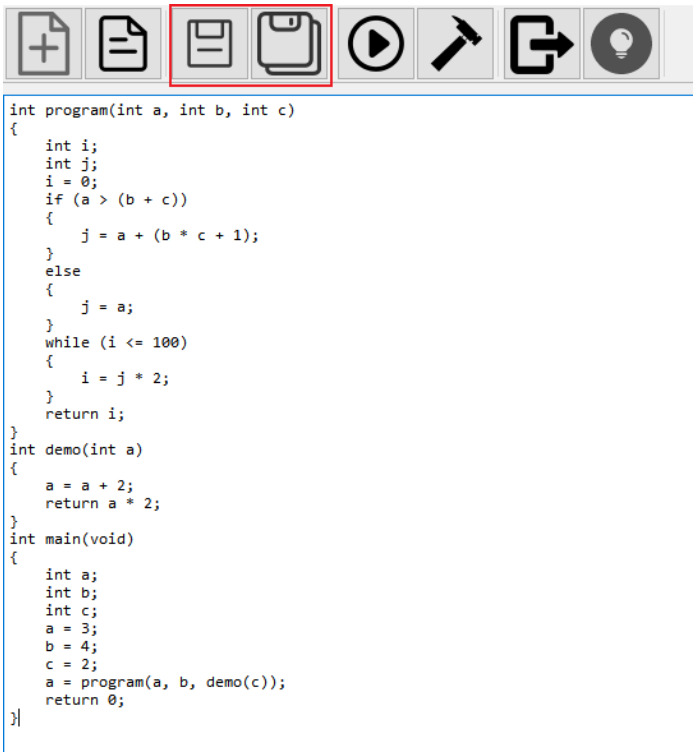
源程序准备完成后，用户可以通过点击这两个按钮来编译源程序，其中左侧是全部编译，右侧是单步编译，一次编译一个语句。

单步编译时，用户可在右侧查看语法状态栈、语法符号栈以及输入串。上面的导航栏三个按钮分别表示：查看状态栈、查看中间代码和查看目标代码。

装 订 线



源程序全部编译结束后，可查看中间代码和目标代码，并且提供了保存文件接口，用户可将代码保存至本地文件。



```
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;
}
int main(void)
{
    int a;
    int b;
    int c;
    a = 3;
    b = 4;
    c = 2;
    a = program(a, b, demo(c));
    return 0;
}
```

用户还可将修改过的源代码保存至本地或另存为其它文件。框出的两个按钮左侧表示保存，右侧表示另存。

装 订 线

六、 课程设计总结

课程认识：通过本次自主设计并实现类 C 编译器，我将编译原理课上讲过的抽象的理论知识转化为了实际上的应用。这不仅加深了我对课程知识的理解，并且在实际变成的过程中建立了各部分知识之间的联系。在编程的过程中，我了解了高级语言程序经过编译的流程与机制，让我进一步理解了软硬件的原理，对今后的学习与工作都是一份宝贵的经验。

心得体会：这学期的工作补上了上学期没有实现的过程调用，从文法到中间代码的设计，以及目标代码生成的实现。为此我又重新温习了学过的 Mips 汇编的知识以及操作系统中关于函数调用时栈的变化这一知识点，能够将不同课程的知识交织在一起实现一个项目，这让我有非常难以言说的感觉，自己的所得所学不再是孤立的个体，不再是飘渺的空中楼阁，而是能够融会贯通并用于实际的“有用”知识。今后学习计算机知识的道路仍道阻且长，我会努力坚持，争取成为一名掌握系统知识的全方位的计算机人才。