北京航空航天大學

BEIHANG UNIVERSITY

编译原理课程设计说明文档

院（系）名称	计算机学院
专业名称	计算机科学与技术
学号	16061170
姓名	宋卓洋

2019 年 1 月

第一部分 需求说明

1. 文法说明

1.1 符号集

{ ‘+’, ‘-’, ‘*’, ‘/’, ‘<’, ‘<=’, ‘>’, ‘>=’, ‘==’, ‘!=’, ‘=’, ‘(’, ‘)’, ‘[‘, ‘]’, ‘{‘, ‘}’, ‘,’, ‘;’, ‘:’, ‘’’, ‘””, ‘a’, ... , ‘z’, ‘A’, ... , ‘Z’, ‘0’, ... , ‘9’, ‘_’}

1.2 保留字集

{ const, void, int, char, if, while, switch, case, default, return, main, printf, scanf}

1.3 文法解读

- *＜加法运算符＞ ::= + | -*

  解释：加法运算符可以为’+’或’-’。

- *＜乘法运算符＞ ::= * | /*

  解释：乘法运算符可以为’*’或’/’。

- *＜关系运算符＞ ::= < | <= | > | >= | != | ==*

  解释：关系运算符可以为’<’、’<=’、’>’、’>=’、’!=’、’==’。

- *＜字母＞ ::= _ | a | ... | z | A | ... | Z*

  解释：字母可以为大小写字母以及下划线’_’。

- *＜数字＞ ::= 0 | ＜非零数字＞*

  解释：数字可以为 0 或非零数字。

- *＜非零数字＞ ::= 1 | ... | 9*

  解释：非零数字可以为 1 至 9 中任意一个。

- *＜字符＞ ::= '＜加法运算符＞' | '＜乘法运算符＞' | '＜字母＞' | '＜数字＞'*

  解释：规定了字符的格式。字符由单引号标识边界，内部可以为单个的加法运算符、乘法运算符、字母或数字。要注意字符的不包含其他的 ASCII 字符，与字符串的范围进行区别。

- *＜字符串＞ ::= "｛十进制编码为 32,33,35-126 的 ASCII 字符｝"*

  解释：字符串由双引号标识边界，内部为任意数量的十进制编码为

32,33,35-126 的 ASCII 字符。注意不存在转义字符的概念。如“\n”按照原样输出，而不将其转义为换行符。

- ***＜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＞***

解释：规定了程序的基本结构。程序由常量说明、变量说明、函数定义、主函数依次按照顺序组成，规定了程序的整体结构。其中除主函数为必选项外其他都为可选部分。常量说明和变量说明都分别最多存在一个。函数定义分为有返回值函数定义和无返回值定义，在文法上两者间无顺序要求，且都可存在多个。注意这几个部分之间的顺序关系。

- ***＜常量说明＞ ::= const＜常量定义＞;{ const＜常量定义＞;}***

解释：规定了常量说明的整体结构。常量说明由一至多个“常量说明部分”构成。每个“常量说明部分”都要由保留字“const ”作为前缀，后接一个常量定义，最后以分号作为该部分的结尾。

- ***＜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＞}

解释：规定了定义每个常量时的结构。常量定义分为整型常量定义和字符型常量定义。二者分别由保留字“int”和“char”作为前缀。整型常量定义可以对一至多个整型常量进行定义。定义的格式为：*＜标识符＞＝＜整数＞*。每个定义间以逗号作为分隔。字符型常量定义可以对一至多个字符型常量进行定义。定义的格式为：*＜标识符＞＝＜字符＞*。每个定义间以逗号作为分隔。注意每次仅能对同一种常量进行定义，并且需要对常量进行赋值操作。

- ***＜无符号整数＞ ::= ＜非零数字＞｛＜数字＞｝| 0***
- ***＜整数＞ ::= ［＋ | －］＜无符号整数＞***

解释：

无符号整数可为 0 或以非零数字作为前缀后接多个数字。含有前缀零的数字为非法。

整数由无符号前接至多一个加号或减号构成。

- ***＜标识符＞ ::= ＜字母＞｛＜字母＞ | ＜数字＞｝***

解释：标识符以字母开头，后面由任意个字母或数字构成。

- ***<声明头部> ::= int<标识符> |char<标识符>***

解释：声明头部由“int”或“char”作为开头，以单个标识符作为结尾。

- ***<变量说明> ::= <变量定义>;{<变量定义>;}***

解释：变量说明由至少一个变量定义构成，每个变量定义间由分号分隔。

- ***<变量定义> ::= <类型标识符>(<标识符>|<标识符>'['<无符号整数>']'){,(<标识符>|<标识符>'['<无符号整数>']')}***

解释：变量定义以类型标识符作为开头。后面可以定义至少一个变量或数组。若定义变量则由标识符构成；若定义数组则其文法格式为：*<标识符>'['<无符号整数>']'*。注意需要判断数组索引部分为大于零的无符号整数，不能为表达式，也不能为带“+”的整数。

- ***<常量> ::= <整数>|<字符>***

解释：常量为整数或字符。

- ***<类型标识符> ::= int | char***

解释：类型标识符为“int”或“char”。

- ***<有返回值函数定义> ::= <声明头部>'('<参数表>')' '{'<复合语句>'}'***

解释：规定了有返回值函数定义的结构——有返回值函数定义由声明头部、参数表、复合语句三部分组成。声明头部作为该函数定义的开头，其后为由小括号括住的参数表，最后为花括号包裹的复合语句。

- ***<无返回值函数定义> ::= void<标识符>'('<参数表>')''{'<复合语句>'}'***

解释：规定了无返回值函数定义的结构——无返回值函数定义以“void”作为开头，后面依次由函数名、参数表、复合语句三部分组成。函数名由标识符构成，参数表由小括号包围，复合语句由花括号包裹。

- ***<复合语句> ::= [<常量说明>][<变量说明>]<语句列>***

解释：复合语句由常量说明、变量说明、语句列依次构成。除语句列为必选项外，常量说明和变量说明为可选项，至多有一个。注意各部分的顺序。

- ***<参数表> ::= <参数>{,<参数>}| <空>***

解释：参数表可以为空或者由至少一个构成，每个参数间由逗号分隔。

- ***＜参数＞ ::= ＜类型标识符＞＜标识符＞***

解释：参数由类型标识符加标识符构成。

注意：由以上几条规则可以看出函数声明的参数表部分无法表示数组结构，即类似于“int searchIn(int list[])”的结构是非法的。函数只能接受变量、常量形式的输入。

- ***＜主函数＞ ::= void main'('')''{'＜复合语句＞'}'***

解释：主函数由“***void main'('')***”作为前部，后接由花括号包围的复合语句。

- ***＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞}***

解释：表达式由至少一个项组成。第一个项的前面可以由一个“+”或“-”，且该符号进作用于第一个项。其他项之间由加法运算符分隔。

- ***＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}***

解释：项由至少一个因子构成，每个因子间由乘法运算符分隔。

- ***＜因子＞ ::= ＜标识符＞｜＜标识符＞'['＜表达式＞']'|'('＜表达式＞')'｜＜整数＞|＜字符＞｜＜有返回值函数调用语句＞***

解读：因子可以为六种情况：

1. 单个标识符；
2. 标识符后接由中括号包围的表达式（数组）；
3. 由括号包围的表达式（嵌套表达）；
4. 整数；
5. 字符；
6. 有返回值函数的调用语句；

- ***＜语句＞ ::= ＜条件语句＞｜＜循环语句＞| '{'＜语句列＞'}'| ＜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＞;｜＜空＞;|＜情况语句＞｜＜返回语句＞;***

解释：语句分为 10 种情况。分别为条件语句、循环语句、语句块（由花括号包围的语句列）、函数调用语句（有返回值函数调用和无返回值函数调用）、赋值语句、读语句、写语句、情况语句、返回语句和空语句。注意其中函数调

用语句、赋值语句、读写语句、情况语句、返回语句和空语句后部需以分号结尾，因为这些为单个的语句。

- ***＜赋值语句＞ ::= ＜标识符＞＝＜表达式＞|＜标识符＞'['＜表达式＞']'= ＜表达式＞***

解释：赋值语句可将表达式的值赋给标识符和数组元素。表达式在右部，标识符和数组元素在左部，之间用等号连接。数组元素的结构为标识符后接用中括号包围的表达式。此外，该语句后部无分号。

- ***＜条件语句＞ ::= if '('＜条件＞')'＜语句＞***

解释：条件语句由判断部分和行为部分组成。判断部分由“if”作为前部，后接由括号包围的条件。行为部分为语句。注意条件语句只有“如果”部分，无“否则”部分。

- ***＜条件＞ ::= ＜表达式＞＜关系运算符＞＜表达式＞ | ＜表达式＞***

解释：条件语句由一至两个条件构成。若为两个表达式，则两者间由关系运算符分隔。

- ***＜循环语句＞ ::= while '('＜条件＞')'＜语句＞***

解释：循环语句由判断部分和循环体组成。判断部分由“while”作为前部，后接由括号包围的条件。循环体为语句。

- ***＜情况语句＞ ::= switch '('＜表达式＞')' '{'＜情况表＞＜缺省＞ '}'***

解释：情况语句由关键字声明和分类讨论两部分组成。情况语句由“switch”作为前部，后接由括号包围的表达式。分类讨论部分为情况表和缺省被花括号包围构成。

- ***＜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＞|＜空＞***

解释：情况表由至少一个情况子语句组成。情况子语句由判断标签和行为部分组成。判断标签的结构为：“case”作为前部，分号作为后部，中间为常量。行为部分为语句。缺省可以为空，也可以为“default:”后接语句。与 C 语言不同的是在执行每个情况的行为部分语句后无需“break;”语句来忽略后面的情况，该文法默认只允许选择包含缺省在内的所有情况中的一个。“default”语

句可以缺少。

- *＜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＞*

解释：函数调用语句结构为标识符后接由括号包围的值参数表。值参数表可以为空也可以为至少一个表达式，每个表达式以逗号作为分隔。

- *＜语句列＞ ::= ｛＜语句＞｝*

解释：语句列为任意个语句，可以为空。

- *＜读语句＞ ::= scanf '('＜标识符＞{,＜标识符＞}')'*

解释：读语句为对函数“scanf”的调用语句。其中值参数表不能为空，而且表达式只能为标识符（用于存储读取到的数据）。

- *＜写语句＞ ::= printf '(' ＜字符串＞,＜表达式＞ ')'| printf '('＜字符串＞ ')'| printf '('＜表达式＞')'*

解释：写语句为对函数“printf”的调用语句，写 1 至 2 个参数。可以单独写一个字符串或表达式；也可以同时依次写一个字符串和一个表达式（两者间以逗号分隔且顺序固定）。但是无法同时输出多个字符串或多个表达式。

- *＜返回语句＞ ::= return['('＜表达式＞')']*

解释：返回语句可以仅为“return”，或者在其之后接由括号包围的表达式。

2. 目标代码说明

- *addu rd, rs, rt*

操作：rs := rt + rd。

- *subu rd, rs, rt*

操作：rs := rt – rd。

- *mult rd, rs, rt*

操作：计算 rs * rt，将高 32 为赋值给 HI，将低 32 位赋值给 LO 和 rd。

- *div rs, rt*

操作：计算 rs / rt，将整除结果赋值给 LO，将余数赋值给 HI。

- *mflo rd / mfhi rd*

操作：将 LO / HI 寄存器中的值至传给 rd。

- *slt rd, rs, rt / sltu rd, rs, rt*

操作：若 rs < rt，则 rd := 1；否则 rd := 0。后者处理无符号数。

- *sll rd, rt, num*

操作：将 rt 中的值左移 num 位，并赋值给 rd。

- *srl rd, rt, num / sra rd, rt, num*

操作：将 rt 中的值右移 num 位，并赋值给 rd。后者根据符号位补全左部。

- *beq rs, rt, label*

操作：rs 等于 rt 跳转至 label。

- *blt rs, rt, label*

操作：rs 小于 rt 跳转至 label。

- *bge rs, rt, label*

操作：rs 大于等于 rt 跳转至 label。

- *bne rs, rt, label*

操作：re 不等于 rt 跳转至 label。

- *bgtz rs, label*

操作：rs 大于 0 跳转至 label。

- *blez rs, label*

操作：rs 小于等于 0 跳转至 label。

- *slt rd, rs, rt*

操作：rs 小于 rt 置 rd 为 1，否则置零。

- *sw rs, imme(rt)*

操作：将 rs 的值存至内存中地址为 rt 加偏移量 imme 的位置。

- *lw rs, imme(rt)*

操作：将 rt 加偏移量 imme 的位置的值存至 rs。

- *la rs, label*

操作：将 label 的地址存至 rs。

- *j label*

操作：无条件跳转至 label。

- *jal label*

  操作：无条件跳转至 label，并将 PC+4 存至$ra（无延迟槽）。

- *jr rs*

  操作：PC := rs。

- *syscall*

  操作：系统调用。

3. 优化方案

3.1 数据流分析

在函数内对划分好的基本块进行活跃变量分析，推导出每个基本块的“in”、“out”集合。为跨基本块的优化提供数据流信息的支持。

3.2 基本块内公共子表达式删除

为基本块构建 DAG 图，删除公共子表达式，并结合数据流分析的结果将 DAG 图导出。

3.3 全局寄存器分配

根据数据流分析的结果，以函数为单位构造冲突图，并运用图着色算法分配全局寄存器。

3.4 临时寄存器使用

采用临时寄存器池的结构，使用 LRU 调度算法进行临时寄存器的分配。

3.5 常量合并与复写传播

对于常量的计算进行合并，将常量内联至代码。完成赋值语句即类赋值语句的传播。

3.6 死代码删除

删除程序中的无用代码，如冗余的代码分支、无用的计算语句等。

第二部分 详细设计

1. 程序结构

1.1 词法分析单元

词法分析单元接受语法分析单元的调用，并且能够使用错误处理单元提供的接口。

词法分析单元负责读取源程序文件，按照文法规定的词法从源程序文件中解析出词素，并将词素的信息输出到相应的变量中供语法分析单元使用。在词法分析的过程中，词法分析单元需要对源文件中不符合词法的部分进行识别、归类，并通过调用错误处理单元提供的接口将错误信息传至错误处理单元。词法单元能够对一部分简单词法错误按照恐慌策略进行恢复操作。

1.2 语法分析单元

语法单元能够使用词法分析单元和错误处理的单元提供的接口。

语法单元通过调用词法分析单元提供的接口，依次获取源文件中的词素，并按照文法对源文件进行语法分析，识别并检查源文件中的语法成分，保证源程序结构正确性。语法分析单元在语法分析构成中需要对语法错误的部分进行识别归类，对于部分简单的语法错误按照恐慌策略进行回复，并将错误信息通过调用错误处理单元的接口将错误信息传至错误处理单元。

1.3 语义分析及中间代码生成单元

语义分析及中间代码生成单元采取语法制导翻译策略，接受语法分析单元的调用，并能够使用错误处理单元提供的接口。

语法分析及中间代码生成单元建立于语法分析单元的基础之上，在进行语法分析的时候接受语法分析单元的调用，完成各种用于管理源程序语义信息的数据结构的建立，根据上下文检查源程序语义的正确性，并运用属性翻译文法进行语法制导翻译，将源程序转换为中间代码（四元式）。语法分析部分需要对语义错误的部分进行识别归类，并通过调用接口将错误信息传至错误处理单元。

1.4 中间代码优化单元

1.4.1 基本块操作单元

基本块操作单元负责完成基本块的划分、存储基本块相关的信息以及对存储基本块信息的数据结构进行维护，提供对基本块进行操作的接口。

1.4.2 数据流分析单元

使用活跃变量分析算法，对基本块进行数据流分析，更新基本块的数据流信息。

1.4.3 基本块内优化单元

根据基本块的中间代码，生成相应的 DAG 图，并根据基本块的变量活跃信息将 DAG 图导出为中间代码。

1.4.4 冲突关系分析单元

以函数为单位，根据活跃变量分析的结果生成变量间的冲突图，并维护保存冲突图的数据结构，供目标代码生成单元进行全局寄存器分配时使用。

1.4.5 复写传播与死代码删除单元

扫描中间代码，完成变量和常量的复写传播，并将冗余代码删除。

1.5 MIPS 汇编生成单元

1.5.1 全局寄存器分配单元

根据中间代码优化单元构造的冲突图，运用图着色算法为变量分配全局寄存器。记录全局寄存器的分配情况，维护相关的数据结构。

1.5.2 临时寄存器分配单元

维护临时寄存器池，运用 LRU（最近最少使用）管理临时寄存器，为未分配到全局寄存器的变量分配临时寄存器，并记录临时寄存器的分配情况，维护相关的数据结构。

1.5.3 代码生成单元

代码生成单元负责根据寄存器分配的情况将生成的中间代码转换成 MIPS 汇编代码，并实现目标代码对其运行时环境的维护。在转换过程中，选择适宜的 MIPS 汇编代码进行机器相关优化。

1.6 错误处理单元

错误处理单元定义编译过程中的错误种类，为词法分析、语法分析、语义分析单元提供调用接口，接受错误的信息，并将错误进行格式化输出。

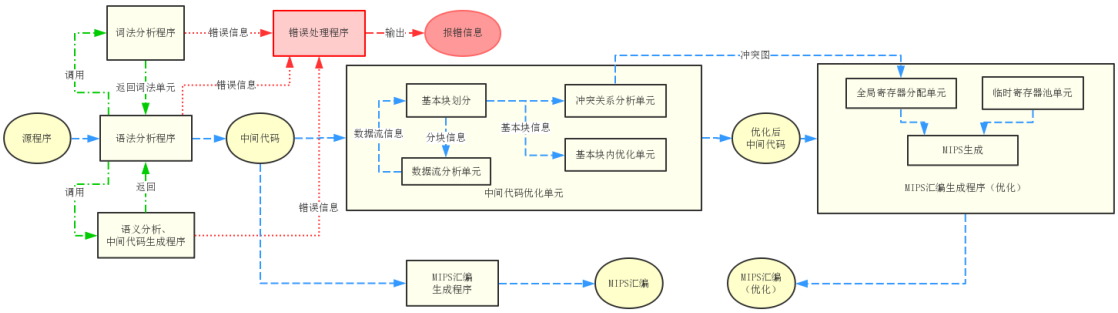

图 1：编译器程序结构

2. 函数功能

2.1 词法分析单元

2.1.1 set_input_file (char*)

接受保存源程序的文件的路径，初始化词法分析单元各参数，完成词法分析开始前的准备。

2.1.2 get_sym ()

供语法分析单元调用。每次调用时会继上次词法分析结束位置解析出一个词素，并根据种类的不同将词素的信息保存至指定的几个变量中。

2.1.3 fetch_int ()

当识别整型量前缀（数字）时调用此函数，完成对整型量的解析。

2.1.4 fetch_ident ()

当识别标识符前缀（字母）时调用此函数，完成对标识符的解析。

2.1.5 fetch_char ()

当识别字符型常量前缀（单引号）时调用此函数，完成对字符型常量的解析。

2.1.6 fetch_string ()

当识别字符串常量前缀（双引号）时调用此函数，完成对字符串常量的解析。

2.1.7 move () / flush_buffer ()

实现对源程序的缓存读取，维护缓存区与读取指针。

2.1.8 str2simple (char) / str2symbol (char*)

实现字符串向相应枚举类的映射。

2.2 语法分析单元

语法分析单元采取递归下降子程序方法进行语法分析，并为每个产生式编写了相对应的分析程序。所有分析子程序如下所示：

非终结符	分析子程序名
<程序>	analyze_program ()
<常量说明>	analyze_const_dec ()
<常量定义>	analyze_const_def (
<无符号整数>	analyze_unsigned_int ()
<整数>	analyze_int ()
<声明头部>	analyze_def_head ()
<变量说明>	analyze_var_dec ()
<变量定义>	analyze_var_def ()
<常量>	analyze_const ()
<有返回值函数定义>	analyze_refunc_def ()
<无返回值函数定义>	analyze_func_def ()
<复合语句>	analyze_complex_sentence ()
<参数表>	analyze_para_list ()
<参数>	analyze_para ()
<主函数>	analyze_main ()
<表达式>	analyze_exp ()
<项>	analyze_item ()

<因子>	analyze_factor ()
<语句>	analyze_sentence ()
<赋值语句>	analyze_assignment ()
<条件语句>	analyze_if ()
<条件>	analyze_condition ()
<循环语句>	analyze_while ()
<情况语句>	analyze_switch ()
<情况表>	analyze_case_list ()
<情况子语句>	analyze_case ()
<缺省>	analyze_default ()
<有返回值函数调用语句>	analyze_refunc_call ()
<无返回值函数调用语句>	analyze_func_call ()
<值参数表>	analyze_value_list ()
<语句列>	analyze_block ()
<读语句>	analyze_scanf ()
<写语句>	analyze_printf ()
<返回语句>	analyze_return ()

表 1 语法分析子程序表

2.3 语义分析及中间代码生成单元

由于本编译器采用语法制导翻译以及一遍扫描策略，在语法分析的同时会根据相应的属性翻译文法进行语义分析以及中间代码生成。本单元需要为属性翻译文法中的每一个动作（更新符号表、查询符号表、生成中间代码等）提供相应的函数，从而实现其功能。此外还要对语法分析子程序的传入和返回值进行改写，从而实现综合属性和继承属性的传递。

2.3.1 enter ()

将标识符加入符号表。

2.3.2 enter_var ()

将变量名登陆至符号表。

2.3.3 enter_array ()

将数组信息登陆至数组信息表。

2.3.4 enter_para ()

将函数说明中的形参登陆至符号表。

2.3.5 enter_func ()

将函数名字登陆至符号表。

2.3.6 enter_str ()

将字符串常量添加至符号表。

2.3.7 search ()

搜索标识符在符号表中的位置。

2.3.8 gen_mid ()

生成四元式。

2.4 中间代码优化单元

2.4.1 create_blocks()

将中间代码转换为基本块。

2.4.2 search_next()

搜索后继基本块。

2.4.3 mark_blocks()

标记基本块前驱后继。

2.4.4 build_blocks()

划分基本块。

2.4.5 enter_use_def()

生成 use 集合、def 集合。

2.4.6 refresh_live()

计算活跃变量 in 集合和 out 集合。

2.4.7 gen_live()

活跃变量分析。

2.4.8 live_variable_analysis()

外部调用接口。

2.4.9 init_conflict()

初始化冲突图。

2.4.10 add_conflict()

增加冲突。

2.4.11 gen_conflict_in()

冲突关系分析。

2.4.12 init_dag()

初始化 DAG 图。

2.4.13 create_node()

初始化结点。

2.4.14 gen_opt_code()

生成中间代码至缓存区。

2.4.15 enter_node()

将结点加入至结点表，不存在则新建表项。

2.4.16 need_export()

判断结点是否需要导出。

2.4.17 search_node()

搜索 DAG 图结点，不存在则新建结点。

2.4.18 is_exportable()

检查所有父节点是否已导出。

2.4.19 export_node()

导出结点。

2.4.20 export_dag()

导出已生成的 DAG 图。

2.4.21 gen_dag()

生成 DAG 图。

2.4.22 write_back()

写回优化后的中间代码。

2.4.23 remove_common_exp()

块内公共子表达式消除，外部调用接口。

2.4.24 check_rewrite(char** name)

检查是否能被复写。

2.4.25 rewrite()

跨基本块复写传播。

2.4.26 delete_dead()

死代码删除。

2.5 MIPS 汇编代码生成单元

2.5.1 start_generator()

代码生成程序启动函数。

2.5.2 init_generator()

初始化目标代码生成器。

2.5.3 gen_mips()

生成 MPIS 汇编代码。

2.5.4 get_unused_temp_reg()

获取未被使用的寄存器。

2.5.5 get_offset()

获取变量和参量相对偏移、获取数组首元素的相对偏移。

2.5.6 alloc_reg_global()

获取全局寄存器。

2.5.7 alloc_reg_resident()

申请驻留寄存器。

2.5.8 assign_global_reg()

分配全局寄存器（着色算法）。

2.5.9 gen_data()

生成.data 段。

2.5.10 gen_init_stack()

初始化栈空间。

2.5.11 gen_save_env()

保存环境。

2.5.12 gen_func_def()

函数声明。

2.5.13 gen_func_call()

函数调用。

2.5.14 gen_return()

函数返回。

2.5.15 gen_B_jump()

选择跳转。

2.5.16 gen_cul()

计算语句。

2.5.17 gen_scanf()

输入语句。

2.5.18 gen_printf()

输出语句。

2.5.19 gen_text()

生成代码段。

2.6 错误处理单元

2.6.1 error_stand()

输出系统错误。

2.6.2 commit_error_gramma()

输出语法错误。

2.6.3 commit_error_lex()

输出词法错误。

2.6.4 commit_error_sema()

输出语义错误。

2.6.5 commit_jump_info()

跳读信息输出。

2.6.7 commit_error_mips_gen()

输出 MIPS 生成错误。

2.6.8 commit_error_block()

输出分块错误。

3. 调用依赖关系

如下图。

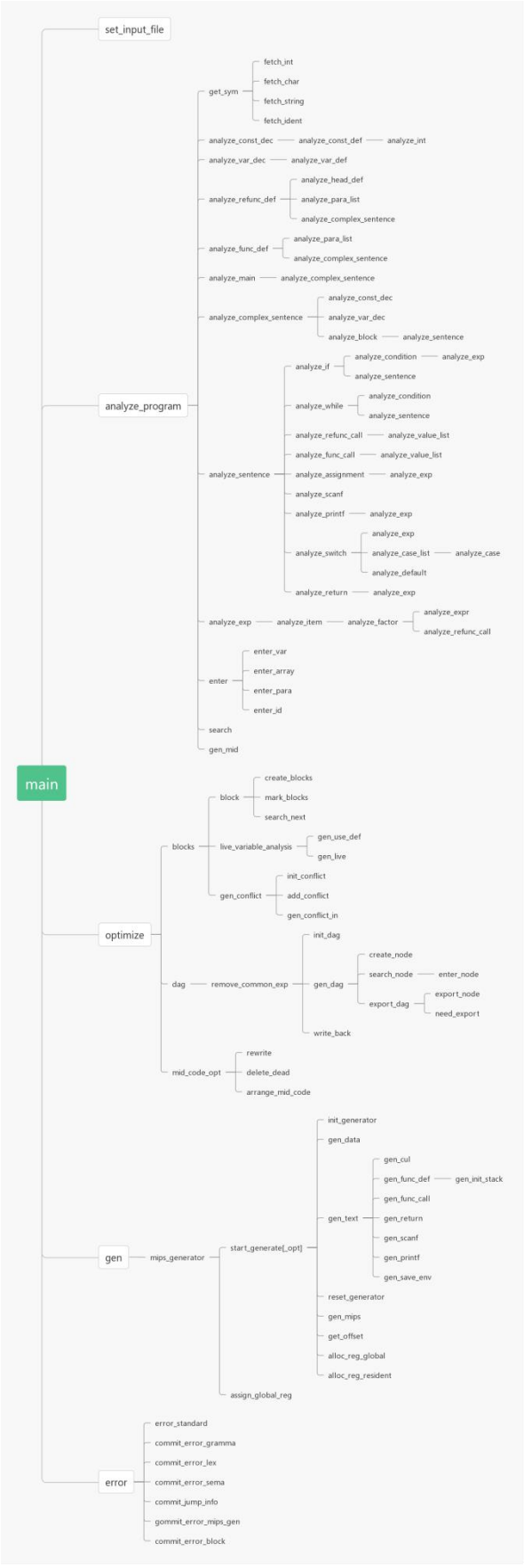
main
set_input_file
analyze_program
get_sym
fetch_int
fetch_char
fetch_string
fetch_ident
analyze_const_dec
analyze_const_def
analyze_int
analyze_var_dec
analyze_var_def
analyze_refunc_def
analyze_head_def
analyze_para_list
analyze_complex_sentence
analyze_func_def
analyze_para_list
analyze_complex_sentence
analyze_main
analyze_complex_sentence
analyze_complex_sentence
analyze_const_dec
analyze_var_dec
analyze_block
analyze_sentence
analyze_sentence
analyze_if
analyze_condition
analyze_exp
analyze_sentence
analyze_while
analyze_condition
analyze_sentence
analyze_refunc_call
analyze_value_list
analyze_func_call
analyze_value_list
analyze_assignment
analyze_exp
analyze_scanf
analyze_printf
analyze_exp
analyze_switch
analyze_exp
analyze_case_list
analyze_case
analyze_default
analyze_return
analyze_exp
analyze_exp
analyze_item
analyze_factor
analyze_expr
analyze_refunc_call
enter
enter_var
enter_array
enter_para
enter_id
search
gen_mid
optimize
blocks
block
create_blocks
mark_blocks
search_next
live_variable_analysis
gen_use_def
gen_live
gen_conflict
init_conflict
add_conflict
gen_conflict_in
dag
remove_common_exp
init_dag
gen_dag
create_node
search_node
enter_node
export_dag
export_node
need_export
write_back
mid_code_opt
rewrite
delete_dead
arrange_mid_code
gen
mips_generator
start_generate[_opt]
init_generator
gen_data
gen_text
gen_cul
gen_func_def
gen_init_stack
gen_func_call
gen_return
gen_scanf
gen_printf
gen_save_env
reset_generator
gen_mips
get_offset
alloc_reg_global
alloc_reg_resident
assign_global_reg
error
error_standard
commit_error_gramma
commit_error_lex
commit_error_sema
commit_jump_info
gommit_error_mips_gen
commit_error_block

图 2 调用依赖关系图

4. 符号表管理方法

4.1 符号表项结构

源程序中的每个标识符都需要在符号表中登记一个符号表项，用于保存与该标识符有关的信息。以下为符号表项所拥有的域以及其意义。

(1) Id:

标识符名称。

(2) kind:

标识符种类，分为常量、变量、函数、参数。

(3) type:

标识符类型，分为整型、字符型。

(4) Return_type:

函数类型，分为无返回值、返回整型、返回字符型。

(5) Const_value:

存储常量值。

(6) Array_length:

存储数组长度。

(7) addr:

在符号表中的索引。

(8) Field_num:

所属域的编号。

4.2 字符串信息表

字符串信息表用于记录编译过程中的字符串常量信息。以下为每个域的意义。

(1) Strings: 所有字符串内容。

(2) Sum: 字符串总数。

4.3 中间代码表

中间代码表用于记录生成的中间代码。表中的每一项存储一条中间代码（四元式）。

5. 存储分配方案

5.1 运行栈

运行栈由高地址向低地址生长。每一栈帧的布局如下表所示。

全局寄存器环境 （保存调用者的全局寄存器环境）	本栈帧
临时变量 （保存本函数的临时变量）	
局部变量 （保存本函数的局部变量）	
参数 （调用者传入参数）	上一栈帧
返回地址 （调用者的返回地址）	

表 2：运行栈布局

当目标代码进行函数调用时，由于调用者为非叶函数，首先需要保存自身的返回地址，再将参数依次压入栈中，并将$sp 寄存器指向此时的栈顶。进入被调用函数后，被调用依次将局部变量与临时变量设置于栈顶上相应位置。由于被调用函数需要进行全局寄存器的分配与使用，还需将需要使用的全局寄存器的旧值存放值栈帧中相应位置，在返回时再进行恢复操作。

5.2 静态区

全局变量保存在静态区，由$gp 寄存器统一寻址。字符串常量统一保存在.data 段，通过 Label 寻址。

6. 四元式设计

6.1 standard operation

GEI // GET a b c ----- a = b[c]

AEI // AEI a b c ----- a[b] = c

ADD // ADD a b c ----- a = b + c

SUB // SUB a b c ----- a = b - c

MUL // MUL a b c ----- a = b * c

DIV // DIV a b c ----- a = b / c

ASN // ASN a b ------- a = b

6.2 compare operation

LES // LES a b ------- a < b

LEQ // LEQ a b ------- a <= b

GTR // GTR a b ------- a > b

GEQ // GTE a b ------- a >= b

EQL // EQL a b ------- a == b

NEQ // NEQ a b ------- a != b

6.3 jump operation

LABEL // LABEL LABEL_1 -- LABEL_1:

GOTO // GOTO LABEL_1 -- jump to LABEL_1

BZ // BZ LABEL_1 --- if condition is false then jump to LABEL_1

6.4 function operation

FUNC // FUNC type name - declare a function named "name" returning a value in "type"

PARA // PARA type a ---- declare a parameter "a" in "type"

PUSH // PUSH a --------- transmit value para

CALL // CALL a --------- call a function named "a"

RET // RET a ---------- return (a)

6.5 declare operation

CONST // CONST INT a 10 - declare a constant and assign to 10

VAR // VAR INT a ------ declare a variable

ARRAY // ARRAY INT a ---- declare a array of integer

6.6 I/O operation

READ // READ a INT ----- read a in type of integer

PRINT // PRINT a INT ---- print a in type of integer

6.7 control operation

END // END ------------ end of program

NOP // NOP ------------ nop

7. 目标代码生成方案

7.1 数据段生成

数据段生成时需要将程序中所有的字符串常量存储至目标代码的.data 段，并为每个字符串常量添加 Label。

7.2 代码段生成

7.2.1 运算指令生成

根据运算符种类的不同生成不同的 MIPS 汇编代码。由于在生成中间代码时已经将常量内联至中间代码中，生成运算指令时需要识别中间代码中的常量，并调整常量位置，生成相应的立即数运算指令。若中间代码的两个操作数都为常量，则生成对于结果的赋值指令。

7.2.2 跳转指令生成

由于中间代码仅存在不满足跳转语句，在生成跳转指令时需要对选择相反

的判断条件。如果中间代码存在常量，需要将常量调整至正确的位置。此外在跳转指令的前部需要生成相关的存储指令来保护跳转前的环境。

7.2.3 函数调用指令生成

在生成函数调用指令时，首先需要生成存储指令，将参数压入运行栈供被调用者使用。然后需要生成存储指令，将环境中的临时寄存器的值写回内存中的相应位置，保护现场。最后生成跳转指令跳转并链接至被调用函数的位置。

7.2.4 函数定义指令生成

在生成函数定义指令时，首先需要生成函数相应的 Label，再对函数的全局寄存器进行分配，然后生成存储指令保存需要使用的全局寄存器原先的值。若存在分配到全局寄存器的函数的传入参数则需要生成相应的指令将参数在运行栈中的值存入该全局寄存器。

7.2.5 函数返回指令生成

在生成函数返回指令时，如果函数有返回值，则需要将返回值传至$v0 寄存器。然后生成内存读取指令恢复调用前全局寄存器的值。最后生成跳转指令跳转至调用函数的返回地址。

7.2.6 I/O 指令生成

在生成输出指令时，首先需要将要输出的值传至$a0，再根据输出值的类型生成不同的系统调用指令完成输出。

在生成输入指令时，需要根据输入值的类型生成相应的系统调用指令，再将$v0 寄存器得到的值传至存储变量相应的位置。

8. 优化方案

8.1 活跃变量分析

（1）划分基本块

首先遍历一次中间代码序列，根据中间代码操作符的种类标记基本块的起始位置与结束位置，以及函数块的起始位置。

然后再次遍历一次中间代码序列，根据标记生成函数块和基本块的存储结构，将基本块的的基本信息存储至相应的数据结构中。不同的基本块按照其所属的函数进行分类存储。此外在生成函数块时需要为其构造入口块和出口块。

入口块和出口块不包含中间代码。

最后遍历每个函数块中的基本块，根据每个基本块最后一条中间代码寻找该基本块的后继基本块，更新基本块的前驱和后继。所有函数返回基本块的后继设置为出口块，入口块的后继设置为第一个基本块。

（2）生成“use 集合”和“def 集合”

遍历每个基本块内的中间代码，根据每条中间代码对变量的定义与使用情况更新基本块的“use 集合”和“def 集合”。入口块的“def 集合”设置为函数传入的参数。

（3）迭代生成“in 集合”和“out 集合”

首先置所有 in 集合”和“out 集合”为空集，再遍历每个函数块内的基本块，根据基本块后继的“in 集合”以及自身的“in 集合”和“out 集合”，按照公式：

$$in = use \cup (out - def)$$

更新基本块的“in 集合”，直至所有“in 集合”和“out 集合”不再改变。

8.2 局部公共子表达式删除

（1）生成 DAG 图

遍历每个基本块内的中间代码，将运算代码按照 DAG 图生成算法生成 DAG 图结点，并更新结点表中的记录。当遍历到函数调用语句时，需要将 DAG 图进行一次导出，再生成相应的压参和调用语句。当遍历至输入语句时，由于对该变量进行了一次赋值操作，该变量已离开原来在 DAG 图中的结点，因而需要从结点表中删除与该变量相关的记录。最后遍历至基本块结尾时将 DAG 导出。

（2）DAG 图导出

DAG 图的导出采用类似树的后序遍历的方式。先选取最先加入 DAG 图的未导出的根节点，开始后序遍历以该节点为根节点的子树，若子节点都已导出则生成该节点的中间代码，否则采取同样的方式递归导出子节点。将该根节点导出完毕后，继续处理下一未导出的根节点，直至将所有节点导出。在导出时优先选择跨块的变量作为结点的被赋值变量进行导出。在导出完成后需要检查

是否所有跨块变量都已导出，若存在未导出的跨块变量则需生成相应的中间代码导出该变量。

8.3 常量合并与传播

在编译阶段，编译器将识别到的常量内联至中间代码中，并将可计算的中间代码进行修改成相应的赋值语句。在优化阶段，编译器会遍历中间代码将含常量的赋值语句中的常量传播至后续代码，替换被赋值的变量。

8.4 复写传播

在优化阶段，编译器会识别赋值语句和等价于赋值语句的中间代码，将后续代码中的该变量替换。

8.5 死代码删除

在优化阶段，编译器会遍历中间代码，识别可计算的比较语句，删除不会运行到的代码分支。此外，编译器还会识别中间代码中会运行到但该语句被定义的变量不会被使用的语句，并将其删除。复写传播和死代码删除迭代交替进行，直至中间代码趋于稳定。

8.6 全局寄存器分配

（1）生成冲突图

首先将每个基本块的“out 集合”的变量设置为在其基本块出口活跃。再逆向遍历每个基本块中的基本代码，根据中间代码的操作符种类更新活跃的变量，并在变量的定义点将此时活跃的变量与被定义的变量设置为冲突，更新冲突图。此外，对于函数的传入参数，由于其相对于函数的定义点在函数的入口块，则设置传入参数相互冲突。

（2）图着色算法分配寄存器

首先进行冲突图的导出。对于每个冲突图，计算其每个阶段的度。若存在度小于全局寄存器个数的结点，则取度最小的结点加入分配栈，移出冲突图；否则将度最大的从冲突图中删除，并重新计算其他节点的度。重复以上步骤至所有变量被移出冲突图。

最后进行全局寄存器的分配。逆序遍历分配栈，依次为变量分配寄存器。在分配的同时需要保证该变量分配到的寄存器不应已分配寄存器的与其冲突的变量的寄存器相同。

8.7 临时寄存器池

所有寄存器统一管理，构成寄存器池。当需要分配临时寄存器时，编译器首先寻找是否有未分配的临时寄存器。若有则将其分配；否则选择最近最少使用的临时寄存器分配。分配已使用的临时寄存器时需要对原寄存器进行写回判断。若该寄存器存储的数据为脏数据且在之后会被使用，就将其写回内存；否则不写回。

8.8 指令选择优化

指令选择时使用 MIPS 汇编的基础指令，以减少伪指令转换为基础指令时产生的冗余指令。

8.9 其他优化

8.9.1 运算强度减弱

编译器识别代码中有关 2 的幂次的乘法和除法，将其转换为逻辑左移与右移以减轻极端强度。

9. 出错处理

9.1 词法错误和语法错误

在词法分析和语法分析阶段，编译程序如果发现不符合文法规定的成分时会采用恐慌恢复策略，对源程序进行适当的忽略以保证分析的继续进行。与此同时，还需将错误的信息进行记录与输出。

9.2 语义错误

在语义分析阶段，编译程序如果发现不符合语义的成分，比如变量重复声明、类型不一致等，将会输出所有的错误记录，并终止编译。

9.3 错误信息及含意

9.3.1 系统错误

```
#define  ILLEGAL_NUM        1   // 函数传入参数错误
FILE_NOT_FOUND              2   // 文件打开失败
END_FILE                    3   // 读取至文件尾部，结束程序
OVERFLOW_MID_CODE           4   // 中间代码超限
```

9.3.2 词法错误

```
LEX_ERROR                       1   // 词法错误
TOO_LARGE_INT                   2   // 输入数字溢出
ILLEGAL_INT                     3   // 非法数字格式
TOO_LONG_STRING                 4   // 输入字符串溢出
MISSMATCH_SYM_QUOTE             5   // 单引号匹配错误
MISSMATCH_SYM_DOUBLE_QUOTE      6   // 双引号匹配错误
MISSMATCH_SYM_NOT_EQUAL         7   // 不等号匹配错误
MISSMATCH_CHAR                  8   // 字符匹配错误
ILLEGAL_CHAR                    9   // 非法标识符
EXCEPTIONAL_END                 10  // 异常结束
```

9.3.3 语法错误

```
MISSMATCH_CONST_INT             1   // 未匹配到整型
MISSMATCH_IDENTIFIER            2   // 未匹配到标识符
MISS_MAIN                       3   // 未匹配到主函数
MISSMATCH_INT                   4   // 未匹配到无符号整数
MISSMATCH_CONST_CHAR            5   // 未匹配到字符常量
MISSMATCH_TYPE_DEF              6   // 未匹配到类型声明保留字“int”
“char”
MISSMATCH_VOID                  7   // 未匹配到“void”
MISSMATCH_KEY_CONST             8   // 未匹配到“const”
MISSMATCH_MAIN                  9   // 未匹配到“main”
MISSMATCH_IF                    10  // 未匹配到“if”
```

```
MISSMATCH_SCANF                 11  // 未匹配到“scanf”
MISSMATCH_PRINTF                12  // 未匹配到“printf”
MISSMATCH_SWITCH                13  // 未匹配到“switch”
MISSMATCH_CASE                  14  // 未匹配到“case”
MISSMATCH_DEFAULT               15  // 未匹配到“default”
MISSMATCH_RETURN                16  // 未匹配到“return”
MISSMATCH_SEMICOLON             17  // 未匹配到分号
MISSMATCH_RIGHT_BRACKET         18  // 未匹配到右中括号
MISSMATCH_LEFT_PARENT           19  // 未匹配到左括号
MISSMATCH_RIGHT_PARENT          20  // 未匹配到右括号
MISSMATCH_LEFT_BRACE            21  // 未匹配到左大括号
MISSMATCH_RIGHT_BRACE           22  // 未匹配到右大括号
MISSMATCH_BECOME                23  // 未匹配到赋值号
MISSMATCH_COMPARE               24  // 未匹配到关系运算符
MISSMATCH_COLON                 25  // 未匹配到冒号
MISSMATCH_PARALIST_END          26  // 参数表结束符匹配错误
MISSMATCH_COMMA                 27  // 未匹配到逗号
INT_OVERFLOW                    28  // 整型越界
MISSBRANCH_PROGRAM              101 // <程序>中分支错误
MISSBRANCH_SENTENCE             102 // <语句>中分支错误
MISSBRANCH_FACTOR               103 // <因子>中分支错误
EXCESSIVE_PROGRAM               110 // 源程序超出<程序>
RECOVER_RIGHT                   140 // 右括号恢复。
RECOVER_LEFT                    141 // 左括号恢复。
```

9.3.4 语义错误

```
#define  DUPLICATE              1   // 重复定义
WRONG_TYPE                      2   // 引用时类型错误
ILLEGAL_COMPARE_TYPE            3   // 比较式类型错误
UNDEFINED                       4   // 未定义
```

```
MISTAKE_SUM_VALUE_PARA      5   // 函数调用值参数表数量错误
ILLEGAL_FUNC_CALL           6   // 无函数调用
ILLEGAL_ARRAY_SIZE          7   // 数组长度定义错误
INDEX_OVERFLOW              8   // 数组下标越界
ILLEGAL_FUNC_CALL_RET       9   // 非法调用无返回值函数
WRONG_ASSIGN_TYPE           10  // 赋值类型不匹配
WRONG_KIND                  11  // 非法种类
WRONG_INDEX_TYPE            12  // 非法数组下标类型
NO_RETURN                   13  // 函数无有效返回语句
WRONG_RETURN_TYPE           14  // 返回值类型错误
```

第三部分 操作说明

1. 运行环境

VS 2017

MARS 4-5

2. 操作说明

使用 VS-2017 运行编译器，输入源程序地址，开始编译。

编译结果存储在工程目录下的“mips.txt”和“mips_opt.txt”文件中，复制至 MARS 即可运行。

第四部分 测试报告

1. 测试程序及测试结果

1.1 测试程序一

测试程序见“test_assign_exp.txt”

测试结果：

```
**********************************************
Start testing assign & exp:
1
40320
-2
1
3
a
_
a
_
903264
**********************************************
```

1.2 测试程序二

测试程序见“text_branch.txt”。

测试结果：

```
**********************************************
Start testing if & while:
+
-
+
-
+
/
*
/
*
/
**********************************************
Start testing switch:
a
```

1
b
3

_
****

1.3 测试程序三

测试程序见“test_global_IO.txt”。

测试结果：

****
Start testing global:
12345679
0
0
-12345679
9

_
+
*
12345679
-12345679
9
*
0
1
1
2
3
1
1
2
6
2
a
A
z
Z

_
+
-
*
/

6
****

1.4 测试程序四

测试程序见“test_recursion.txt”。

测试结果：

****
Start testing recursion:
take
1
from
a
to
b
take
2
from
a
to
c
take
1
from
b
to
c
take
3
from
a
to
b
take
1
from
c
to
a
take
2
from
c

to
b
take
1
from
a
to
b
take
4
from
a
to
c
take
1
from
b
to
c
take
2
from
b
to
a
take
1
from
c
to
a
take
3
from
b
to
c
take
1
from
a
to
b

```
take
2
from
a
to
c
take
1
from
b
to
c
**********************************************
```

1.5 测试程序五

测试程序见“test_ret_para.txt”。

测试结果：

```
**********************************************
Start testing return:
22
a
b
c

_
**********************************************
Start testing parameter:
INPUT of func_ret_int_1:
12345679
-12345679
A
Z
OPERATE of func_ret_int_1:
-12345679
12345679
Z
A
-12345679
**********************************************
```

1.6 测试程序六

测试程序见“test_error_lex.txt”。

程序需要检测出的错误：输入数字溢出、输入数字不符合文法、单引号匹配错误、双引号匹配错误、不等号匹配错误、字符常量不符合文法、非法标识符。

测试结果：

ERROR: TO_LARGE_INT at line 4, char 18
ERROR: TO_LARGE_INT at line 4, char 19
ERROR: TO_LARGE_INT at line 4, char 20
ERROR: ILLEGAL_INT at line 5, char 10
ERROR: ILLEGAL_INT at line 5, char 11
ERROR: ILLEGAL_INT at line 5, char 12
ERROR: MISSMATCH_SYM_NOT_EQUAL at line 6, char 11
ERROR: MISSMATCH_COMPARE at line 6
ERROR: MISSMATCH_SYM_DOUBLE_QUOTE at line 9, char 16
ERROR: MISSMATCH_RIGHT_PARENT at line 10
ERROR: MISSMATCH_SEMICOLON at line 10
ERROR: MISSBRANCH_SENTENCE at line 10
IGNORE: line 10 become
ERROR: MISSMATCH_SYM_QUOTE at line 10, char 11
IGNORE: line 10 const_char
WARNING: NO_RETURN at line 11
END

1.7 测试程序七

测试程序见“text_error_gramma_1.txt”。

程序主要检测变量和常量声明出现的错误，需要检测出的错误：整型数字不匹配、字符型常量不匹配、标识符不匹配、类型声明保留字不匹配、赋值符号未匹配。

测试结果：

ERROR: MISSMATCH_INT at line 1
IGNORE: line 1 const_char
ERROR: MISSMATCH_IDENTIFIER at line 1
IGNORE: line 1 become
IGNORE: line 1 const_int
ERROR: MISSMATCH_CONST_CHAR at line 2
IGNORE: line 2 const_int
ERROR: MISSMATCH_TYPE_DEF at line 3
IGNORE: line 3 ident
ERROR: MISSMATCH_BECOME at line 4
ERROR: MISSMATCH_CONST_CHAR at line 4

ERROR: MISSMATCH_INT at line 7
IGNORE: line 7 const_char
ERROR: MISSMATCH_IDENTIFIER at line 7
IGNORE: line 7 become
IGNORE: line 7 const_int
ERROR: MISSMATCH_CONST_CHAR at line 8
IGNORE: line 8 const_int
ERROR: MISSMATCH_TYPE_DEF at line 9
IGNORE: line 9 ident
ERROR: MISSMATCH_BECOME at line 10
ERROR: MISSMATCH_CONST_CHAR at line 10
ERROR: MISSBRANCH_SENTENCE at line 12
IGNORE: line 12 const
IGNORE: line 12 int
IGNORE: line 12 ident
IGNORE: line 12 become
IGNORE: line 12 const_int
FUNC: main
END

1.8 测试程序八

测试程序见“test_error_grama_2.txt”。

程序主要检测对复合语句以及函数声明部分的报错。需要及测出的错误有：函数声明部分不符合文法以及各复合语句不符合文法。

测试结果：

ERROR: MISSMATCH_TYPE_DEF at line 3
IGNORE: line 3 left_brace
IGNORE: line 4 return
IGNORE: line 4 semicolon
ERROR: MISSMATCH_TYPE_DEF at line 6
IGNORE: line 6 ident
ERROR: WRONG_RETURN_TYPE at line7
ERROR: MISSMATCH_TYPE_DEF at line 9
ERROR: WRONG_RETURN_TYPE at line10
ERROR: MISSMATCH_SEMICOLON at line 13
ERROR: MISSMATCH_SEMICOLON at line 13
ERROR: MISSBRANCH_SENTENCE at line 13
IGNORE: line 13 const_char
WARNING: NO_RETURN at line 14
ERROR: ILLEGAL_COMPARE_TYPE at line17
ERROR: ILLEGAL_COMPARE_TYPE at line17

ERROR: MISSMATCH_COMPARE at line 18
ERROR: MISSMATCH_INT at line 19
ERROR: MISSBRANCH_FACTOR at line 19
ERROR: MISSMATCH_RIGHT_PARENT at line 20
ERROR: ILLEGAL_COMPARE_TYPE at line21
ERROR: ILLEGAL_COMPARE_TYPE at line21
ERROR: MISSMATCH_COMPARE at line 22
ERROR: MISSMATCH_INT at line 23
ERROR: MISSBRANCH_FACTOR at line 23
ERROR: MISSMATCH_RIGHT_PARENT at line 24
ERROR: MISSMATCH_INT at line 25
ERROR: MISSBRANCH_FACTOR at line 25
ERROR: MISSMATCH_CASE at line 25
ERROR: MISSMATCH_CASE at line 26
ERROR: MISSMATCH_INT at line 27
ERROR: MISSMATCH_INT at line 28
IGNORE: line 28 colon
ERROR: MISSMATCH_RIGHT_PARENT at line 29
ERROR: MISSMATCH_SEMICOLON at line 29
ERROR: MISSBRANCH_SENTENCE at line 29
IGNORE: line 29 right_parent
ERROR: MISSMATCH_RIGHT_PARENT at line 30
ERROR: MISSMATCH_LEFT_PARENT at line 31
ERROR: MISSMATCH_RIGHT_PARENT at line 32
ERROR: MISSMATCH_SEMICOLON at line 32
ERROR: MISSBRANCH_SENTENCE at line 32
IGNORE: line 32 right_parent
ERROR: MISSMATCH_RIGHT_PARENT at line 33
ERROR: MISSMATCH_LEFT_PARENT at line 34
WARNING: NO_RETURN at line 35

1.9 测试程序九

测试程序见“test_error_grama_3.txt”。

程序主要测试表达式计算的语法错误。

测试结果：

ERROR: MISSMATCH_INT at line 3
ERROR: MISSBRANCH_FACTOR at line 3
ERROR: MISSMATCH_SEMICOLON at line 4
ERROR: MISSBRANCH_SENTENCE at line 4
ERROR: MISSMATCH_INT at line 5
ERROR: MISSBRANCH_FACTOR at line 5
IGNORE: line 5 right_bracket

IGNORE: line 5 become
IGNORE: line 5 ident
ERROR: MISSMATCH_RIGHT_BRACKET at line 6
ERROR: MISSMATCH_INT at line 7
ERROR: MISSBRANCH_FACTOR at line 7
IGNORE: line 7 become
IGNORE: line 7 ident
ERROR: MISSBRANCH_SENTENCE at line 8

1.10 测试程序十

测试程序见“test_error_sema.txt”。

程序主要测试语义错误，如：重复定义、引用类型错误、比较式类型错误、未定义标识符、函数调用传参错误、数组越界、复制类型不匹配、非法数组下表类型。

测试结果：

ERROR: DUPLICATE at line14 a
ERROR: UNDEFINED at line15 fun_
IGNORE: line 15 ident
IGNORE: line 15 left_parent
IGNORE: line 15 right_parent
ERROR: MISTAKE_SUM_VALUE_PARA at line16 func_1
ERROR: WRONG_TYPE at line17
ERROR: WRONG_TYPE at line17
ERROR: WRONG_ASSIGN_TYPE at line18 d
ERROR: UNDEFINED at line19 m
IGNORE: line 19 ident
ERROR: UNDEFINED at line19 n
IGNORE: line 19 ident
ERROR: MISTAKE_SUM_VALUE_PARA at line19 func_1
ERROR: INDEX_OVERFLOW at line20 c
ERROR: INDEX_OVERFLOW at line21 c
ERROR: INDEX_OVERFLOW at line22 c
ERROR: INDEX_OVERFLOW at line23 c
ERROR: INDEX_OVERFLOW at line24 c
ERROR: INDEX_OVERFLOW at line25 c
ERROR: WRONG_ASSIGN_TYPE at line26 c
ERROR: WRONG_ASSIGN_TYPE at line27 f
ERROR: ILLEGAL_FUNC_CALL_RET at line28 func
IGNORE: line 28 ident
IGNORE: line 28 left_parent
IGNORE: line 28 right_parent

2. 测试结果分析

测试程序一至五分别对表达式计算、If-While-Switch 分支语句、全局变量及 I\O 操作语句、函数调用、函数返回及参量操作进行了测试，对于可能的语法分支进行了较好的覆盖。

测试程序六至十根据编译器对编译过程中会遇到的错误进行了复现，并对错误进行连续识别。测试程序分别复现了词法错误、语法错误以及语义错误。

第五部分 总结感想

要写好一个编译器，首先要做的事就是要对要编译的语言进行深入的理解。经过前半阶段的词法分析程序、语法分析程序、语义分析及中间代码分析程序的设计，我对于语言以及文法有了更加深入的理解。一个语句如何运行，如何一步步由最初的序列经过形式化的方法抽象转化为信息，又最终转化为目标的语言。这需要对编译器的每个阶段进行详细的设计，整理出程序的整体架构。我在最初的阶段就已经对编译器的整个架构设计有了一定的框架。这使得我在非优化阶段的基础功能实现的设计上推进得比较顺利，较好地实现了一个能够正确编译源程序的编译器。

在整个编译课程设计中，最令人印象深刻的阶段就是优化阶段了。在开始优化前，我进行了一次代码质量测试。当时的成绩是 FinalCycle 为七千万以上，这在我周围的同学中是一个较差的成绩。也就是说我在最初生成的代码相当的不理想。因而若想要程序达到更好的效果，我需要的做的优化可能要比别人的更多。我在优化阶段完成了活跃变量分析、局部公共子表达式删除、图着色法分配全局寄存器等优化功能，将评测成绩降至了两千四百万左右。虽然不及一些大佬的优化结果，但这种优化效果基本对得起个人的付出了。

在优化程序的设计过程中，我很明显地感受到了理论与实践地巨大差异。在进行相关算法具体实现细节的设计上，我遇到了相当多的困难。经过查询网上的资料以及其他同学地指点，这些问题大多被很好的解决了，同时也加深了我对于一些优化思路的认识。

由于我选择的是 C 语言实现我的编译器，在语言实现层面上，比 C++语言的难度高了不少。这在我最初选择的时候就已经做好了心理准备了。在对于编译器内的许多模型进行数据结构设计时，我运用了许多复杂的数据结构，这早

一定程度上提高了我对 C 语言的使用的能力。