

## 目 录

1 源语言定义 .....	1
1.1 样本语言文法定义 .....	1
1.2 单词的识别模型-有穷自动机 DFA .....	3
1.3 项目结构综述 .....	5
1.3.1 编译器 .....	5
1.3.2 IDE .....	5
2 词法分析程序的实现 .....	7
2.1 实现原理 .....	7
2.2 源文件及函数说明 .....	7
2.2.1 源文件说明 .....	7
2.2.2 数据结构及函数说明 .....	8
2.3 实例说明 .....	9
3 递归下降实现语法分析程序 .....	10
3.1 实现原理 .....	10
3.2 源文件及函数说明 .....	10
3.2.1 源文件说明 .....	10
3.2.2 数据结构及函数说明 .....	10
3.3 实例说明 .....	14
4 语义分析与中间代码生成 .....	16
4.1 中间代码定义 .....	16
4.2 实现原理 .....	16
4.2.1 回填技术 .....	16
4.3 源文件及函数说明 .....	18
4.3.1 源文件说明 .....	18
4.3.2 数据结构及函数说明 .....	18
4.4 实例说明 .....	23
5 解释程序 .....	25
5.1 实现原理 .....	25

5.2 源文件及函数说明 .....	25
5.2.1 源文件说明 .....	25
5.2.2 数据结构及函数说明 .....	26
5.3 实例说明 .....	27
6 图形用户界面 (GUI) .....	29
6.1 GUI 概述 .....	29
6.2 GUI 的组成 .....	29
6.2.1 窗口 .....	29
6.2.2 菜单 .....	29
6.2.3 状态栏 .....	30
7 集成开发环境 .....	31
7.1 集成开发环境的概述 .....	31
7.2 Awesome Studio 的基础功能 .....	31
7.2.1 文件操作 .....	31
7.2.2 代码编辑 .....	32
7.2.3 界面布局 .....	32
7.2.4 调试与编译 .....	33
7.3 IntelliSense 智能感知 .....	33
7.3.1 Structure Sense 结构感知 .....	33
7.3.2 Coding Sense 编程感知 .....	34
7.3.3 Color Sense 对象着色感知 .....	34
7.3.4 File Sense 文件感知 .....	35
8 感想 .....	36
附录 A 小组分工 .....	37
附录 B 错误信息表 .....	38
附录 C 公用库文件 .....	41
附录 D 界面展示 .....	42
参考文献 .....	44

# 1 源语言定义

## 1.1 样本语言文法定义

样本语言的文法  $G = (V_N, V_T, P, S)$ , 其中,  $S$  = 程序,  $P$  为

数字  $\rightarrow 0|1|2|3|4|5|6|7|8|9$  (1.1)

字母  $\rightarrow A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z$  (1.2)

$a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z$  (1.3)

赋值运算符  $\rightarrow =$  (1.4)

算数运算符  $\rightarrow +|-|*|/|%$  (1.5)

比较运算符  $\rightarrow <|>|<=|>=|==|!=$  (1.6)

布尔运算符  $\rightarrow \&\&||$  (1.7)

分隔符  $\rightarrow ;|,|'|"|( )|[ ]|\{ |\}$  (1.8)

访问权限  $\rightarrow private|public$  (1.9)

变量类型  $\rightarrow int|float|double$  (1.10)

函数类型  $\rightarrow int|float|double|void$  (1.11)

标识符  $\rightarrow$  标识符(字母|数字)|字母|\_ (1.12)

整形常数  $\rightarrow$  数字|数字 整形常数 (1.13)

浮点常数  $\rightarrow$  整形常数.整形常数 (1.14)

表达式  $\rightarrow$  表达式项 表达式' (1.15)

表达式'  $\rightarrow$  算数运算符 表达式| $\epsilon$  (1.16)

表达式项  $\rightarrow$  数组项|标识符|整形常数|浮点常数|(表达式) (1.17)

布尔表达式  $\rightarrow$  布尔表达式项 布尔表达式' (1.18)

布尔表达式'  $\rightarrow$  布尔运算符 布尔表达式| $\epsilon$  (1.19)

布尔表达式项  $\rightarrow$  表达式 比较运算符 表达式 (1.20)

非空参数列表  $\rightarrow$  变量类型 参数, 非空参数列表|变量类型 参数 (1.21)

参数列表  $\rightarrow$  非空参数列表| $\epsilon$  (1.22)

非空值列表  $\rightarrow$  非空值列表, (整形常数|浮点常数|表达式)| (1.23)

整形常数|浮点常数|表达式 (1.24)

- 值列表  $\rightarrow$  非空值列表  $|\epsilon$  (1.25)
- 数组项  $\rightarrow$  标识符[整形常数|表达式] (1.26)
- 数组初始化  $\rightarrow$  数组项 赋值运算符 { 值列表 } (1.27)
- 函数声明语句  $\rightarrow$  访问权限 函数类型 标识符 (参数列表){语句块} (1.28)
- 函数调用语句  $\rightarrow$  函数类型 标识符 (值列表); (1.29)
- 变量声明语句  $\rightarrow$  变量类型 (标识符|数组初始化)|  
变量声明语句, (标识符|数组初始化) (1.31)
- 赋值语句  $\rightarrow$  (标识符|数组项) 赋值运算符 表达式; (1.32)
- while*语句  $\rightarrow$  *while*(布尔表达式){语句块} (1.33)
- for*语句  $\rightarrow$  *for*(赋值语句; 布尔表达式; 赋值语句){语句块} (1.34)
- if*语句  $\rightarrow$  *if*(布尔表达式){语句块}(*else*{语句块}) $|\epsilon$  (1.35)
- return*语句  $\rightarrow$  *return*; | *return* 表达式; (1.36)
- 语句  $\rightarrow$  函数调用语句|变量声明语句; |赋值语句; |  
*if*语句|*if - else*语句|*while*语句|*for*语句|*return*语句 (1.38)
- 语句块  $\rightarrow$  语句 语句块|语句 (1.39)
- 类成员  $\rightarrow$  函数声明语句|变量声明语句 $|\epsilon$  (1.40)
- 类声明  $\rightarrow$  *class* 标识符 {类成员} (1.41)
- 程序  $\rightarrow$  类声明 (1.42)

## 1.2 单词的识别模型-有穷自动机 DFA

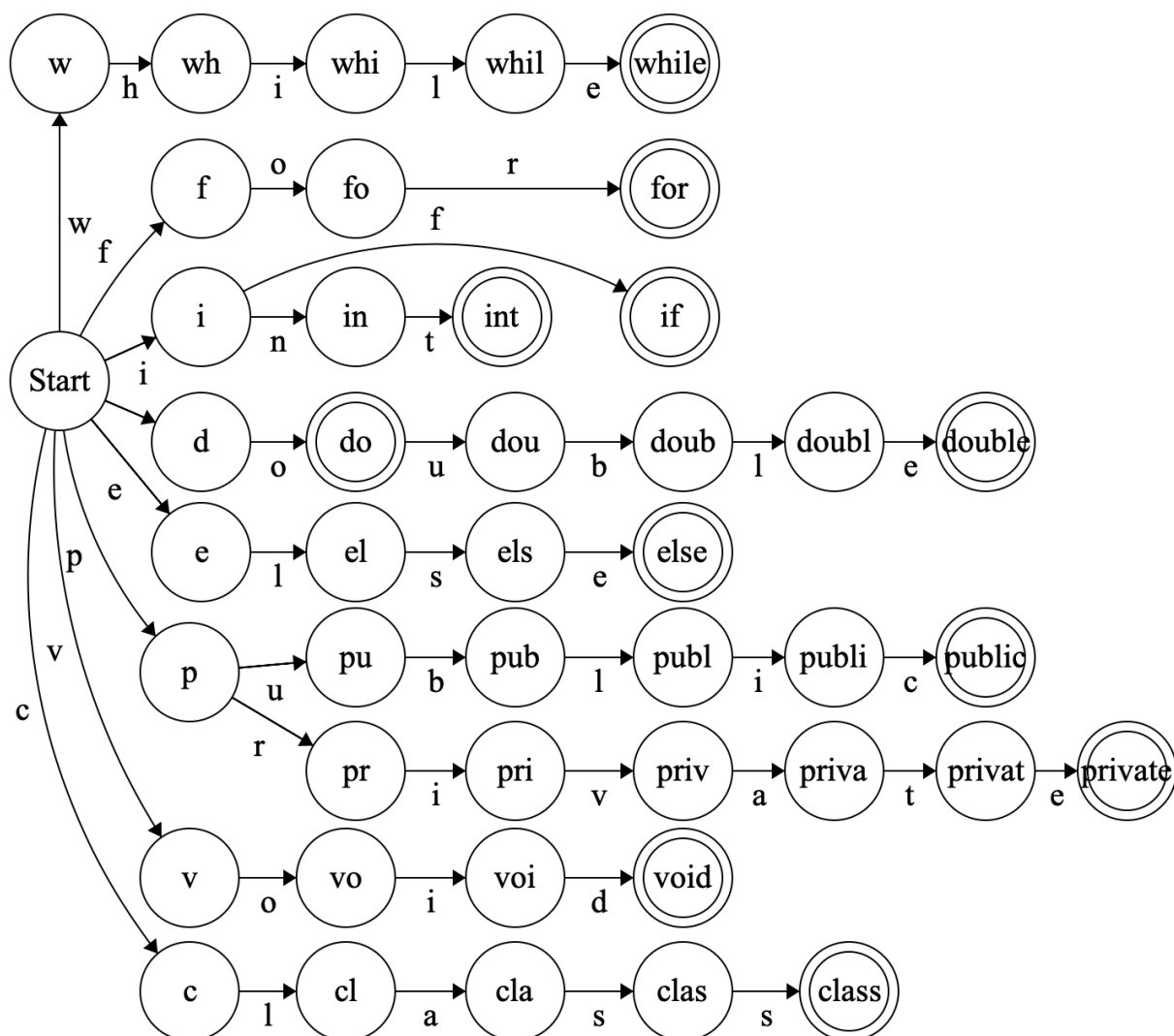


图 1.1 DFA 关键字部分

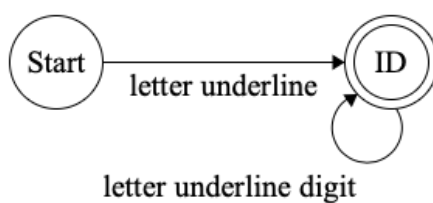


图 1.2 DFA 标识符部分

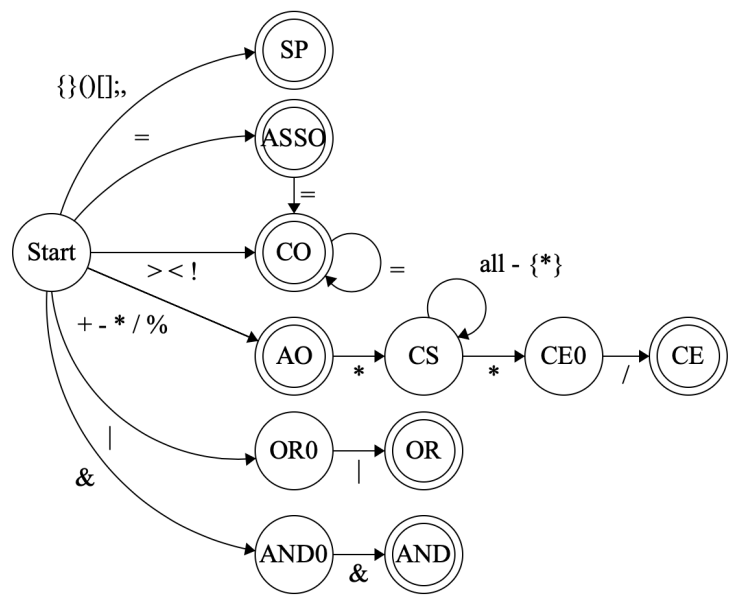


图 1.3 DFA 符号部分

其中，图中符号说明如表1.1所示。图中省略了两个地方。一是关键字部分非结束状态和结束状态如果读入了其他字母（除了关键字部分中转移的字母）、数字、下划线，就转移到 ID 状态；否则转入错误。而是符号部分，如果读入了其他字符，就将转入错误。

图中符号	说明
Start	开始状态
ASSO	赋值预算符
CO	比较运算符
AO	算数运算符
OR	逻辑运算符
AND	逻辑运算符 &&
CS	注释开始
CE	注释结束
ID	标识符
letter	字母
digit	数字
underline	下划线

表 1.1 DFA 说明

## 1.3 项目结构综述

### 1.3.1 编译器

编译器使用 c++ 编写，总行数 5034，包括空行和注释。项目结构如图1.4所示。

- ./awesomeCC，项目代码
  - ./awesomeCC/front-end，前端
  - ./awesomeCC/back-end，后端
  - ./awesomeCC/lib，共用库
- ./awesomeCC\_test，测试代码
- ./demo，测试用例

其中头文件 (.h) 都放在 include 文件夹中，源文件 (.cc) 都放在 src 文件夹中。全局遵守 Google C++ Code Style。使用 Google test 进行单元测试，travisCI 进行集成测试。使用 Cmake 进行编译，生成可执行文件 awesomeCC。

### 1.3.2 IDE

IDE 使用 c# 编写，总行数 9294。项目结构如图1.5所示。

- ./AwesooomeStudio，项目代码
  - ./AwesooomeStudio/bin，二进制文件
    - \* ./AwesooomeStudio/bin/Debug，Debug 模式的编译输出
    - \* ./AwesooomeStudio/bin/Release，Release 模式的编译输出
  - ./AwesooomeStudio/obj，链接对象
  - ./AwesooomeStudio/Properties，项目工程属性
  - ./AwesooomeStudio/Resources，图片等资源文件
- ./Binary，编译出来的成品文件

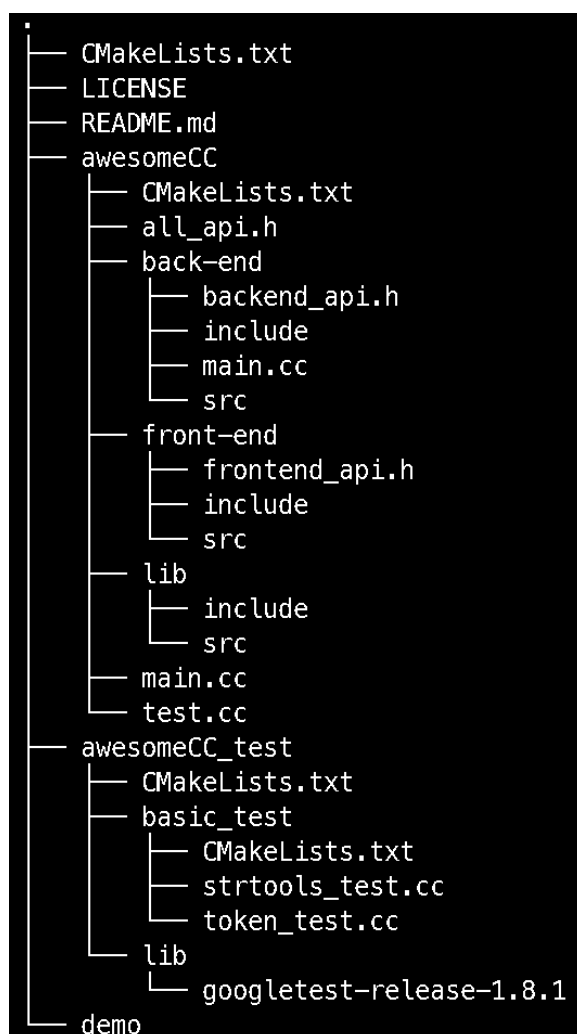


图 1.4 项目结构



图 1.5 项目结构



## 2 词法分析程序的实现

### 2.1 实现原理

如 1.2 节中的 DFA 所示。分析时，从 `start` 状态开始。通过词法分析，我们得到 Token 列表。

- 当前 `in_comment` 为 `true` 如果后两个字符是 `*/`，读取后两个字符，将 `in_comment` 设为 `false`，跳出注释；否则，继续读取。
- 当前字符为空格或者换行符  
检验是否是终止状态，如果是则加入到 `token` 列表，回到开始状态；如果不是，则报错。
- 当前字符不是字母、数字、下划线  
按照图 1.3 进行匹配，检查下一个字符是否能匹配上的两位运算符的终止状态，如果能则加入列表，如果不能则退回匹配一位运算符的终止状态，匹配不上则报错。匹配成功则加入 `token` 列表，回到开始状态。
- 当前字母是数字  
如果目前是 `Start`，就进入常数的匹配，使用 `dot_cnt` 记录小数点个数，如果超过 1 个则报错；否则继续进入常数的匹配。如果下一个不是数字或者小数点，则加入 `token` 列表，回到开始状态。
- 当前字符是下划线  
如果是 `start` 或者标识符状态，则进入标识符状态；否则报错。如果下一个字符不是数字、字母、下划线，则加入 `token` 列表，回到开始状态。
- 当前字符是字母  
向后看，和每个关键字进行匹配，如果匹配上了则加入 `token` 列表，回到开始状态；如果下一个字符是数字、字母、下划线则继续进入标识符状态；否则加入 `token` 列表，回到开始状态。

### 2.2 源文件及函数说明

#### 2.2.1 源文件说明

词法分析器涉及到的文件及其作用如下所示。

- awesomeCC/front-end/include/lexical\_analyzer.h: 词法分析器头文件。
- awesomeCC/front-end/src/lexical\_analyzer.cc: 词法分析器源文件。
- awesomeCC/lib/include/token.h: Token 类头文件。
- awesomeCC/lib/src/token.cc: Token 类源文件。

### 2.2.2 数据结构及函数说明

- 枚举量 `TOKEN_TYPE_ENUM`, 包含 6 个大类 (关键字、分隔符、标识符、数字常量、字符串常量、运算符) 和关键字的 16 个具体分类、运算符的 19 个具体分类、分隔符的 11 个具体分类。
- Token 类
  - `value`, 字符串类型, 记录该 Token 的值。
  - `line_number`, `int` 类型, 记录该 Token 所在行号。
  - `pos`, `int` 类型, 记录该 Token 所在列号。
  - `type`, `TOKEN_TYPE_ENUM` 类型, 记录 Token 的类型。
  - 输出流, 便于格式化输出 Token。
- LexicalAnalyzer 类
  - `tokens`, Token 数组类型, token 列表。
  - `sentence`, 字符串类型, 当前分析的句子。
  - `len`, `int` 类型, 当前分析的句子长度。
  - `cur_pos`, `int` 类型, 当前分析的位置。
  - `cur_line_number`, `int` 类型, 当前分析的句子行号。
  - `in_comment`, `bool` 类型, 是否在注释中。
  - `analyze` 函数, 供外部调用的词法分析执行过程, 包括分析、错误处理。
  - `getAllTokens` 函数, 供外部调用, 返回 Token 列表的。
  - `_analyze` 函数, 按照 2.1 节的逻辑, 分析一个句子。从开始状态开始处理, 每次加入 token 列表后回到开始状态, 直到 `cur_pos >= len`。
  - `_skipBlank` 函数, 用于跳过空白和注释。

类中的一些静态成员函数和私有函数是用于简化判断过程的, 此处不赘述。

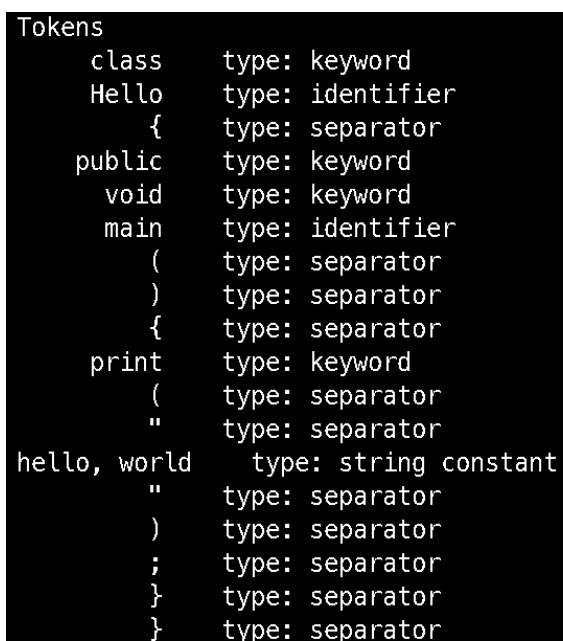
## 2.3 实例说明

词法分析的 api 封装在 awesomeCC/front-end/frontend\_api.h 中，输入的参数是文件路径，如果没有错误，得到 token 列表；如果有错误，则输出错误。

如下所示的 hello world 代码，使用命令 `./acc hello_world.ac -l` 得到 token 列表，如图2.1所示。

```
1 class Hello {  
2     public void main() {  
3         print("hello, world");  
4     }  
5 }
```

Listing 2.1 hello\_world.ac



Tokens	
class	type: keyword
Hello	type: identifier
{	type: separator
public	type: keyword
void	type: keyword
main	type: identifier
(	type: separator
)	type: separator
{	type: separator
print	type: keyword
(	type: separator
"	type: separator
hello, world	type: string constant
"	type: separator
)	type: separator
;	type: separator
}	type: separator
}	type: separator

图 2.1 词法分析输出的 token 列表

## 3 递归下降实现语法分析程序

### 3.1 实现原理

语法分析采用递归下降分析方法，在语法分析之前，先调用词法分析，然后使用 `getAllTokens()` 获得 token 列表，然后开始语法分析。这样便于区别词法报错和语法报错，也便于测试。通过语法分析，我们得到语法树。

### 3.2 源文件及函数说明

#### 3.2.1 源文件说明

词法分析器涉及到的文件及其作用如下所示。

- `awesomeCC/front-end/include/syntax_analyzer.h`: 语法分析器头文件。
- `awesomeCC/front-end/src/syntax_analyzer.cc`: 语法分析器源文件。
- `awesomeCC/lib/include/syntax_tree.h`: 语法树类头文件。
- `awesomeCC/lib/src/syntax_tree.cc`: 语法树类源文件。

#### 3.2.2 数据结构及函数说明

- `SyntaxTreeNode` 类
  - `value`, `string` 类型，记录该节点的值。
  - `type`, `string` 类型，记录该节点类型。
  - `extra_info`, `string` 类型，记录该节点的其他备注。
  - `line_number`, `int` 类型，记录该节点所在行号。
  - `pos`, `int` 类型，记录该节点所在列号。
  - `left`, `SyntaxTreeNode *` 类型，记录该节点的左兄弟<sup>1</sup>。
  - `right`, `SyntaxTreeNode *` 类型，记录该节点的右兄弟。
  - `father`, `SyntaxTreeNode *` 类型，记录该节点的父亲节点。
  - `first_son`, `SyntaxTreeNode *` 类型，记录该节点的第一个子节点。

---

<sup>1</sup>我们使用孩子-兄弟表示法来存语法树

- `true_list`, `int` 数组类型, 如果该节点类型为 `Bool-Expression`, 则记录这个节点为 `true` 时的需要回填的集合<sup>1</sup>。
- `false_list`, `int` 数组类型, 如果该节点类型为 `Bool-Expression`, 则记录这个节点为 `false` 时的需要回填的集合。
- `next_list`, `int` 数组类型, 如果该节点类型为 `Block`, 则记录这个节点执行完后的需要回填的集合。

- `SyntaxTree` 类

- `root`, `SyntaxTreeNode *` 类型, 记录语法树的根节点。
- `cur_node`, `SyntaxTreeNode *` 类型, 记录语法树的最新插入的节点。
- `addNode` 函数, 用于向语法树中插入节点。
- `display` 函数, 用于格式化输出语法树。

这里参考了 `*nix` 下以树形展示文件结构的命令行插件 `tree` 的方式来展现语法树<sup>2</sup>。我们通过深度优先搜索语法树, 使用一个整数 `status` 来记录状态, `status` 转为二进制的从低位到高位第 `i` 位表示树的第 `i` 层的输出状态。如果某节点 `right` 为空, 则是输入完成状态, 此时 `status` 的第 `i` 位为 1, 否则为 0。每次向下一层搜索的时候, 更新 `status`。

- 枚举量 `SENTENCE_PATTERN_ENUM`, 包含声明语句、赋值语句等 10 个语句种类。

- `SyntaxAnalyzer` 类

- `index`, `int` 类型, 目前分析的 `token` 的下标。
- `len`, `int` 类型, `token` 列表的长度。
- `tokens`, `Token` 数组类型, `token` 列表。
- `tree`, `SyntaxTree *` 类型, 语法树。
- `_judgeSentencePattern` 函数, 使用最长匹配判断句子的种类。
- `analyze` 函数, 供外部调用进行语法分析, 包括生成语法树和错误处理。
- `getSyntaxTree` 函数, 供外部调用, 返回语法树。

---

<sup>1</sup>我们使用拉链回填来处理布尔表达式

<sup>2</sup>图 1.4 就是使用该插件生成的

- `_analyze` 函数，按照1.1节的文法定义，判断当前句子的类别，再转发给具体的函数进行处理。
- `_statement` 函数，处理变量声明语句。  
根据式1.31，对变量声明语句进行处理。先匹配类型，计入 `type`。然后每次匹配到标识符，判断下一个 `token` 的类型，如果下一个是 `‘`，则读取，然后继续匹配标识符；如果下一个是 `‘;`，则读取结束；如果下一个是 `‘[`，则按照数组的定义的方式进行匹配，这个过程中，处理数组初始化的值列表的每一项时，递归调用 `_expression` 函数，处理表达式。
- `_functionStatement` 函数，处理函数声明语句。  
根据式1.28，对函数声明语句进行匹配，这个过程中，遇到 `‘{`就递归调用 `_block` 来处理。
- `_functionCall` 函数，处理函数调用语句。  
根据式1.29，对函数调用语句进行匹配，这个过程中，处理值列表的每一项时，递归调用 `_expression` 函数。
- `_assignment` 函数，处理赋值语句。  
根据式1.32，对赋值语句进行匹配。这个过程中，如果遇到数组项则递归调用 `_expression` 函数处理数组下标；`‘=`右边也递归调用 `_expression` 函数。
- `_while` 函数，处理 `while` 语句。根据式1.33来匹配 `while` 语句，处理代码块的时候递归调用 `_block` 函数。
- `_for` 函数，处理 `for` 语句。  
根据式1.34，为了方便之后的翻译过程，这里使用 `pseudo while` 来代替 `for` 语句。即将

```

1  for ( 赋值语句1; 布尔表达式; 赋值语句2 ) {
2      语句块
3  }
```

Listing 3.1 for

改造为

```

1  赋值语句1;
2  while ( 布尔表达式 ) {
```

```

3     语句块
4     赋值语句2;
5 }
```

Listing 3.2 pseudo\_while

然后调用 `_assignment` 函数、`_while` 函数来处理。

- `_print` 函数，处理使用 `print` 函数的语句。  
处理值列表的时候，递归调用 `_expression` 函数。
- `_if` 函数，处理 `if` 语句。  
根据式1.35来用匹配 `while` 语句。递归调用 `_expression` 函数处理 ‘()’ 中的布尔表达式，`_block` 处理语句块。处理完后，如果下一个 `token` 值为 ‘else’，则调用 `_else` 函数。
- `_else` 函数，处理 `else` 语句。  
如果下一个 `token` 的值 ‘if’，则调用 `_else_if` 函数进行处理；否则，调用 `_block` 函数。
- `_else_if` 函数，处理 `else if` 语句。递归调用 `_expression` 函数处理 ‘()’ 中的布尔表达式，`_block` 处理语句块。
- `_block` 函数，处理语句块。  
类似 `_analyze` 函数，判断当前句子的类别，再转发给具体的函数进行处理，最后匹配 ‘}’，如果没有，则报错。
- `_return` 函数，处理返回语句。  
根据式1.36来匹配 `return` 语句。如果下一个 `token` 是 ‘;’，则加入 `Void-Return` 类型子树；否则递归调用 `_expression` 函数，处理返回值。
- `_expression` 函数，处理表达式。  
根据式1.18和式1.15处理表达式。根据当前 `token` 的不同类型，进行如下操作。
  - \* 数字常量，加入 `Expression-Constant` 子树
  - \* 标识符  
如果下一个 `token` 是 ‘=’，则加入 `Expression-Variabl` 子树；否则加入 `Expression-ArrayItem` 子树，递归调用 `_expression`，插入到子树的 `Array-Index` 子树中。

- \* 括号，进行对应的弹栈、压栈操作。
- \* 运算符，判断优先级，进行对应的弹栈、压栈操作。

### 3.3 实例说明

语法分析的 api 封装在 awesomeCC/front-end/frontend\_api.h 中，输入的参数是文件路径，如果没有错误，得到语法树；如果有错误，则输出错误。

如下所示的简单求和代码，其中包括函数定义、变量声明的两个产生式、赋值语句、for 循环改造成 pseudo while、数组项处理、print 语句处理，能够比较全面的测试语法分析器。使用命令 `./acc sum.ac -p` 得到语法树，如图3.1所示。

```
1 class Sum {
2     public void main() {
3         int i, sum, a[5] = {99, 80, 81, 70, 92};
4
5         sum = 0;
6         for (i = 0; i < 5; i = i + 1) { sum = sum + a[i]; }
7
8         print(sum);
9     }
10 }
```

Listing 3.3 sum.ac



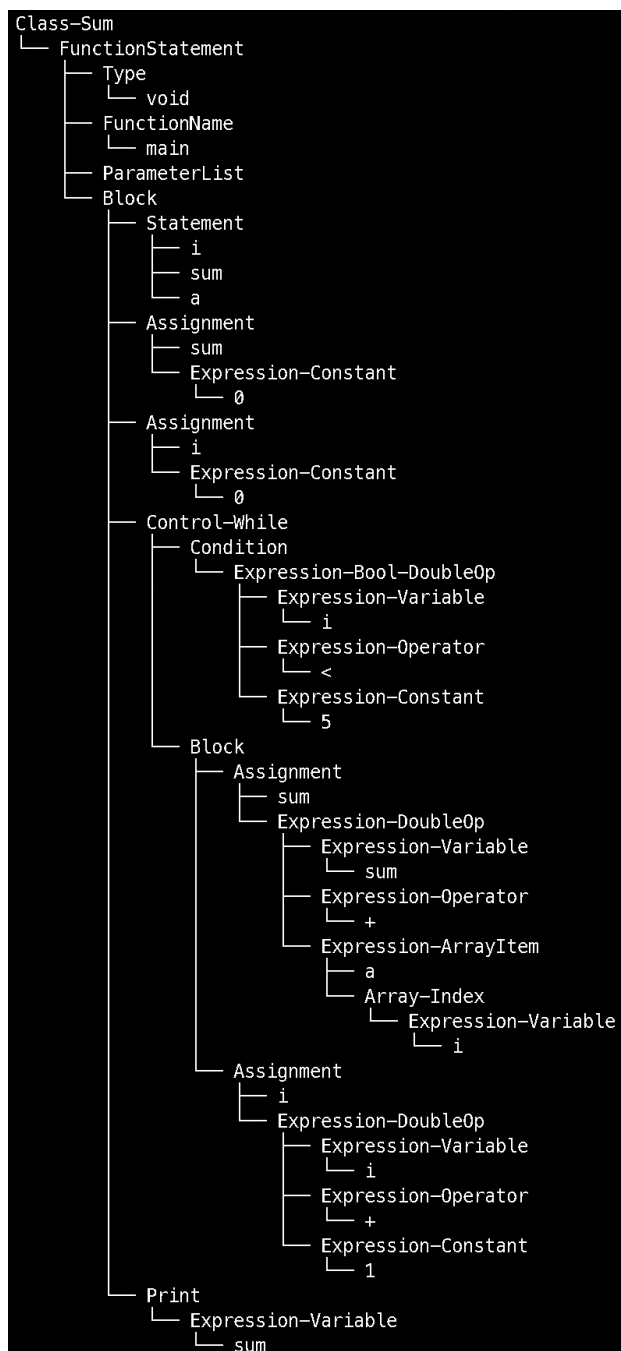


图 3.1 语法分析输出的语法树

## 4 语义分析与中间代码生成

### 4.1 中间代码定义

我们使用一种类似汇编的四元式作为中间代码，其语法如表4.1所示。变量使用 `string` 类型 "v下标" 来表示，临时变量使用 "t下标" 来表示。对于数组项，使用相对寻址，用 "v下标[偏移量]" 来表示<sup>1</sup>。

### 4.2 实现原理

我们使用语法制导翻译，遍历语法树的时候直接嵌入语义动作进行翻译。除了控制语句之外的语义动作在4.3.2节详细描述。控制语句的翻译在4.2.1节详细描述。

#### 4.2.1 回填技术

我们使用回填技术<sup>[1]</sup>来翻译布尔表达式。对于形如  $B_1 || B_2$ 、 $B_1 \&\& B_2$ 、 $(B_1)$ 、 $!B_1$  的布尔表达式，引入一个非终结符号  $M$ ，将文法改造为：

$$B \rightarrow B_1 || M B_2 | \quad (4.1)$$

$$B_1 \&\& M B_2 | \quad (4.2)$$

$$!B_1 | ( B_1 ) \quad (4.3)$$

$$M \rightarrow \epsilon \quad (4.4)$$

遇到  $M$  时，触发执行图4.1所示的语义动作。

对于 `while` 语句  $S \rightarrow \text{while } (B) S$ ，也采取同样的方法，引入非终结符  $M_1, M_2$ ，改写为

$$S \rightarrow \text{while } M_1 (B) M_2 S \quad (4.5)$$

$$M_1 \rightarrow \epsilon \quad (4.6)$$

$$M_2 \rightarrow \epsilon \quad (4.7)$$

此时遇到  $M$  触发执行图4.2所示的语义动作。图中的 *instr* 是对应语句翻译的四元式的起始编号；*S.nextlist* 是一个包含跳转指令的下标列表，对应的指令应该跳转到紧跟在  $S$  的代码之后的指令；*B.truelist*、*B.falselist* 也是包含跳转指令的下标列表，这

<sup>1</sup>现在想想，使用临时变量做数组偏移量，然后通过临时变量寻址更加方便

四元式指令	说明
$ADD, s_1, s_2, d$	$d \leftarrow s_1 + s_2$ $d$ 只能为变量或临时变量 $s_1, s_2$ 可为临时变量、变量、立即数
$SUB, s_1, s_2, d$	$d \leftarrow s_1 - s_2$ $d$ 只能为变量或临时变量 $s_1, s_2$ 可为临时变量、变量、立即数
$MUL, s_1, s_2, d$	$d \leftarrow s_1 * s_2$ $d$ 只能为变量或临时变量 $s_1, s_2$ 可为临时变量、变量、立即数
$DIV, s_1, s_2, d$	$d \leftarrow s_1 / s_2$ $d$ 只能为变量或临时变量 $s_1, s_2$ 可为临时变量、变量、立即数
$MOD, s_1, s_2, d$	$d \leftarrow s_1 \% s_2$ $d$ 只能为变量或临时变量 $s_1, s_2$ 可为临时变量、变量、立即数
$J, \_, \_, d$	无条件跳转到 $d$ $d$ 可为临时变量、变量、立即数
$JE, s_1, s_2, d$	如果 $s_1 = s_2$ 则跳转到 $d$ $s_1, s_2, d$ 可为临时变量、变量、立即数
$JNE, s_1, s_2, d$	如果 $s_1 \neq s_2$ 则跳转到 $d$ $s_1, s_2, d$ 可为临时变量、变量、立即数
$JL, s_1, s_2, d$	如果 $s_1 < s_2$ 则跳转到 $d$ $s_1, s_2, d$ 可为临时变量、变量、立即数
$JG, s_1, s_2, d$	如果 $s_1 > s_2$ 则跳转到 $d$ $s_1, s_2, d$ 可为临时变量、变量、立即数
$MOV, s, \_, d$	$d \leftarrow s$ $s$ 可为临时变量、变量、立即数 $d$ 为临时变量或变量
$PRINT, s, \_, \_, s$	打印 $s$ $s$ 可为临时变量、变量、立即数、字符串
$POP, \_, \_, d$	弹出栈顶元素并将值放入 $d$ $d$ 为临时变量或变量
$PUSH, \_, \_, s$	将 $s$ 的值入栈 $d$ 为临时变量、变量、立即数

表 4.1 四元式定义

些下标对应的指令需要回填为  $B$  的真出口值或  $B$  的假出口值<sup>1</sup>。

<sup>1</sup>在3.2节中的 SyntaxTreeNode 类中有讲述

1) $B \rightarrow B_1 \parallel M B_2$	{ <i>backpatch</i> ( <i>B</i> <sub>1</sub> . <i>false</i> <i>list</i> , <i>M.instr</i> ); <i>B.true</i> <i>list</i> = <i>merge</i> ( <i>B</i> <sub>1</sub> . <i>true</i> <i>list</i> , <i>B</i> <sub>2</sub> . <i>true</i> <i>list</i> ); <i>B.false</i> <i>list</i> = <i>B</i> <sub>2</sub> . <i>false</i> <i>list</i> ; }
2) $B \rightarrow B_1 \&\& M B_2$	{ <i>backpatch</i> ( <i>B</i> <sub>1</sub> . <i>true</i> <i>list</i> , <i>M.instr</i> ); <i>B.true</i> <i>list</i> = <i>B</i> <sub>2</sub> . <i>true</i> <i>list</i> ; <i>B.false</i> <i>list</i> = <i>merge</i> ( <i>B</i> <sub>1</sub> . <i>false</i> <i>list</i> , <i>B</i> <sub>2</sub> . <i>false</i> <i>list</i> ); }
3) $B \rightarrow ! B_1$	{ <i>B.true</i> <i>list</i> = <i>B</i> <sub>1</sub> . <i>false</i> <i>list</i> ; <i>B.false</i> <i>list</i> = <i>B</i> <sub>1</sub> . <i>true</i> <i>list</i> ; }
4) $B \rightarrow ( B_1 )$	{ <i>B.true</i> <i>list</i> = <i>B</i> <sub>1</sub> . <i>true</i> <i>list</i> ; <i>B.false</i> <i>list</i> = <i>B</i> <sub>1</sub> . <i>false</i> <i>list</i> ; }
5) $B \rightarrow E_1 \text{ rel } E_2$	{ <i>B.true</i> <i>list</i> = <i>makelist</i> ( <i>nextinstr</i> ); <i>B.false</i> <i>list</i> = <i>makelist</i> ( <i>nextinstr</i> + 1); <i>gen</i> ('if' <i>E</i> <sub>1</sub> . <i>addr</i> <i>rel.op</i> <i>E</i> <sub>2</sub> . <i>addr</i> 'goto -'); <i>gen</i> ('goto -'); }
6) $B \rightarrow \text{true}$	{ <i>B.true</i> <i>list</i> = <i>makelist</i> ( <i>nextinstr</i> ); <i>gen</i> ('goto -'); }
7) $B \rightarrow \text{false}$	{ <i>B.false</i> <i>list</i> = <i>makelist</i> ( <i>nextinstr</i> ); <i>gen</i> ('goto -'); }
8) $M \rightarrow \epsilon$	{ <i>M.instr</i> = <i>nextinstr</i> ; }

图 4.1 布尔表达式的翻译方案

## 4.3 源文件及函数说明

### 4.3.1 源文件说明

语义分析及中间代码生成涉及到的文件及其作用如下所示。

- awesomeCC/front-end/include/inter\_code\_generator.h: 语义分析及中间代码生成头文件。
- awesomeCC/front-end/src/inter\_code\_generator.cc: 语义分析及中间代码生成源文件。
- awesomeCC/lib/include/quadruple.h: 四元式类头文件。
- awesomeCC/lib/src/quadruple.cc: 四元式类源文件。

### 4.3.2 数据结构及函数说明

- 枚举类 INTER\_CODE\_OP\_ENUM, 四元式类型枚举量, 包含4.1中所有种类。
- Quadruple 类: 四元式类

- 1)  $S \rightarrow \text{if}(B) M S_1$  { *backpatch*(*B.true**list*, *M.instr*);  
                              *S.nextlist* = *merge*(*B.false**list*, *S<sub>1</sub>.nextlist*); }
- 2)  $S \rightarrow \text{if}(B) M_1 S_1 N \text{ else } M_2 S_2$   
      { *backpatch*(*B.true**list*, *M<sub>1</sub>.instr*);  
      *backpatch*(*B.false**list*, *M<sub>2</sub>.instr*);  
      *temp* = *merge*(*S<sub>1</sub>.nextlist*, *N.nextlist*);  
      *S.nextlist* = *merge*(*temp*, *S<sub>2</sub>.nextlist*); }
- 3)  $S \rightarrow \text{while } M_1 (B) M_2 S_1$   
      { *backpatch*(*S<sub>1</sub>.nextlist*, *M<sub>1</sub>.instr*);  
      *backpatch*(*B.true**list*, *M<sub>2</sub>.instr*);  
      *S.nextlist* = *B.false**list*;  
      *gen*('goto' *M<sub>1</sub>.instr*); }
- 4)  $S \rightarrow \{ L \}$            { *S.nextlist* = *L.nextlist*; }
- 5)  $S \rightarrow A ;$            { *S.nextlist* = *null*; }
- 6)  $M \rightarrow \epsilon$            { *M.instr* = *nextinstr*; }
- 7)  $N \rightarrow \epsilon$            { *N.nextlist* = *makelist*(*nextinstr*);  
                              *gen*('goto -'); }
- 8)  $L \rightarrow L_1 M S$        { *backpatch*(*L<sub>1</sub>.nextlist*, *M.instr*);  
                              *L.nextlist* = *S.nextlist*; }
- 9)  $L \rightarrow S$            { *L.nextlist* = *S.nextlist*; }

图 4.2 语句的翻译

- op, INTER\_CODE\_OP\_ENUM 类型, 表示四元式种类。
  - arg1, string 类型, 第一个操作数。
  - arg2, string 类型, 第二个操作数。
  - res, string 类型, 目标操作数。
  - 输出流, 便于格式化输出四元式。
- 枚举类 VARIABLE\_INFO\_ENUM, 表示变量类型, 如 int、void 等。
  - Info 类, 符号表中内容的基类。
    - name, string 类, 符号保存的地点名字。

- VarInfo 类, Info 类的派生类, 用于保存变量信息。
  - type, VARIABLE\_INFO\_ENUM 类型, 用于标示变量的类型。
  - place, int 类型, 用于记录符号表中的下标, 便于随机访问。
- FuncInfo 类, Info 类的派生类, 用于保存函数信息。
  - ret\_type, VARIABLE\_INFO\_ENUM 类型, 用于标示返回的类型。
  - start\_place, int 类型, 用于记录函数的开始位置。
  - end\_place, int 类型, 用于记录函数的结束位置。
- InterCodeGenerator 类
  - tree, SyntaxTree \* 类型, 保存语法树。
  - temp\_var\_index, int 类型, 作为临时变量栈的栈顶指针。
  - var\_index, int 类型, 作为变量栈的栈顶指针。
  - table, map<string, VarInfo> 类型, 变量表, 键是变量名, 值是 VarInfo。
  - func\_table, map<string, FuncInfo> 类型, 函数表, 键是函数名, 值是 FuncInfo。
  - func\_backpatch, map<string, vector<int>> 类型, 每个函数需要回填的语句下标, 键为函数名, 值为下标数组。
  - inter\_code, Quadruple 数组类型, 记录生成的中间代码。
  - analyze 函数, 供外部调用的中间代码生成函数, 包括生成中间代码和报错。
  - \_lookUpVar 函数, 在符号表中寻找某一变量的存放位置, 参数接收变量名的字符串, 如果不存在则报错。
  - \_backpatch 函数, 用于回填, 参数接收一个下标数组和一个回填值, 将下标数组中对应下标的中间代码都回填上回填值。
  - \_emit 函数, 生成一个四元式。
  - \_analyze 函数, 处理类的成员。

根据文法规则1.40, 类下的能够直接定义的只有变量和函数。在 \_analyze 函数中, 如果遇到函数定义, 转发给 \_functionStatement 函数处理; 遇到变量定义, 转发给 \_statement 函数处理; 否则报错。遇到函数的时候, 先翻译主函数, 然后再按顺序翻译别的函数, 如图4.3所示。

main函数
J 程序结束
函数1
...
函数n
程序结束

图 4.3 函数声明的翻译顺序

- `_expression` 函数，翻译表达式，返回结果所在的地址字符串。
  - \* 数字常量  
压入临时变量栈，返回地址。
  - \* 字符串常量  
压入临时变量栈，返回地址。
  - \* 数组项  
调用 `_lookUpVar` 函数查找数组的基地址，接着递归调用 `_expression` 得到数组下标存放的地址。返回基地址与偏移量之和。
  - \* 算数双目表达式  
递归调用 `_expression` 函数处理左运算数和右运算数，得到它们的地址。通过相应的计算获得值，压入临时变量栈。
  - \* 布尔双目表达式  
递归调用 `_expression` 函数处理左运算数和右运算数，得到它们的地址，然后生成比较跳转的四元式。由于四元式中只有 `JE`, `JG`，所以遇到 `>=`, `<=` 符号就交换操作数，改成 `<`, `>` 符号。按照4.2.1节中图4.1所描述的翻译动作翻译和回填操作。
- `_voidReturn` 函数，翻译空的返回语句。  
弹出栈顶元素，保存到定时变量中并跳转到此变量对应的指令。
- `_block` 函数，翻译语句块。  
判断语句类型，转发给 `_print` 函数、`_statement` 函数、`_assignment` 函数、`_if` 函数、`_while` 函数、`_functionCall` 函数进行处理；如果不是上述类型，则报错。由于变量的作用域被 `{}` 限定，故在函数执行前需要对变量表做备份，函数

执行后对变量表进行恢复<sup>1</sup>。

– `_print` 函数，翻译 `print` 语句。

调用 `_expression` 函数来处理值列表里的每一项，得到存储地址，每一项都产生 `PRINT, , , 值` 的代码。最后产生 `PRINT, , , 代码`，用于输出换行。

– `_statement` 函数，翻译声明语句。

如果记录 `type` 是 `int` 或者 `double`，则加入符号表，压入变量栈，分配 1 个单位的储存地址。如果记录的 `type` 是 `array`，读取数组大小信息 `n`，数组名入栈，分配 `n` 个单位的储存地址<sup>2</sup>。

– `_assignment` 函数，翻译赋值语句。

在符号表里查找左值，如果存在对应变量，则记录变量储存的地址，否则报错。调用 `_expression` 函数处理右值，保存返回的地址。生成四元式 `MOV, 右值, , 左值的地址`。

– `_if` 函数，翻译 `if` 语句。

调用 `_expression` 函数处理条件语句，`_block` 函数翻译语句块。判断是单独的 `if`，还是有 `else` 跟随的 `if`，按照 4.2.1 节中图 4.2 所述的回填规则进行回填。

– `_while` 函数，翻译 `while`。

调用 `_expression` 函数处理条件语句，`_block` 函数翻译语句块。按照 4.2.1 节中图 4.2 所述的回填规则进行回填。

– `_functionCall` 函数，翻译函数调用语句。

该函数调用之前需要保存变量表，执行完成后恢复变量表，以弹出函数作用域中的局部变量。先讲 `pc` 指针的值入栈，保存下标，等待回填。接着处理参数，通过栈传递，调用 `_expression` 函数处理值列表中的每一项，保存返回的地址，逆序压栈。生成跳转语句，保存下标，加入调用函数的函数回填表中。使用此时的 `pc` 值回填到一开始的 `pc` 指针入栈的地方。栈的情况如图 4.4 所示。

– `_functionStatement` 函数，翻译函数声明语句。

按照参数 1 ~ 参数 `k` 的顺序弹栈，保存到变量表中。然后调用 `_block` 函数处理语句块。最后弹出栈顶保存的返回位置到一个临时变量中，并且跳转<sup>3</sup>。

<sup>1</sup>也是栈式存储符号表

<sup>2</sup>此处没有在变量表中查询，因为希望实现同名情况下局部变量优先于全局变量的特性。此处没有查找变量表而是直接覆盖即可做到，并且每个作用域都会恢复符号表，故不会丢失上一级作用域中变量的值

<sup>3</sup>等价于在函数结束的地方自动调用了 `_voidReturn` 函数



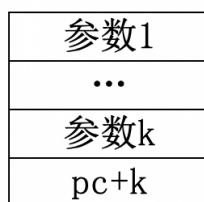


图 4.4 函数调用的入栈顺序

## 4.4 实例说明

语义分析及中间代码生成的 api 封装在 awesomeCC/front-end/frontend\_api.h 中，输入的参数是文件路径，如果没有错误，得到中间代码；如果有错误，则输出错误。

如下所示的判断闰年代码，能够方便的测试编译器翻译的逻辑。使用命令 `./acc leapYear.ac -a` 得到中间代码，如图4.5所示。

```
1 class LeapYear {
2     public void main() {
3         int year;
4         year = 2000;
5
6         if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0) {
7             print(year, "is leap year.");
8         }
9         else {
10            print(year, "is not leap year.");
11        }
12    }
13 }
```

Listing 4.1 leapYear.ac

这里为了更好的说明翻译的中间代码，为中间代码绘制了流程图，如4.6所示。

```

Generated 18 inter codes
#0 (MOV , 2000 , , v0 )
#1 (MOD , v0 , 4 , t0 )
#2 (JE , t0 , 0 , 4 )
#3 (J , , , 14 )
#4 (MOD , v0 , 100 , t1 )
#5 (JNE , t1 , 0 , 10 )
#6 (J , , , 7 )
#7 (MOD , v0 , 400 , t2 )
#8 (JE , t2 , 0 , 10 )
#9 (J , , , 14 )
#10 (PRINT , v0 , , )
#11 (PRINT , "is leap year.", , )
#12 (PRINT , , , )
#13 (J , , , 17 )
#14 (PRINT , v0 , , )
#15 (PRINT , "is not leap year.", , )
#16 (PRINT , , , )
#17 (J , , , 18 )
    
```

图 4.5 语义分析及中间代码生成输出的中间代码

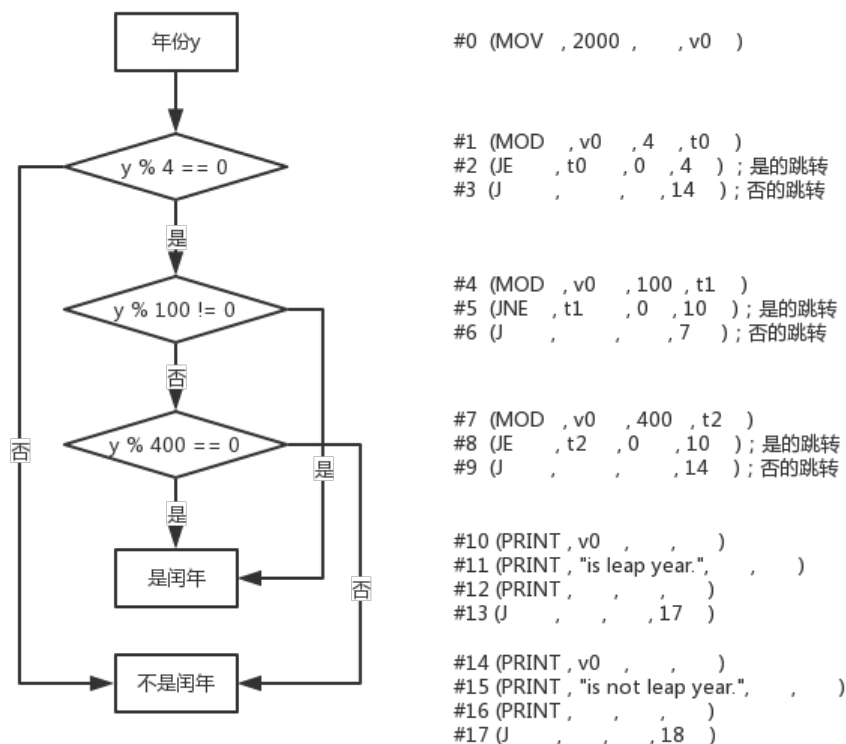


图 4.6 判断闰年的逻辑

## 5 解释程序

### 5.1 实现原理

我们使用经典的如图5.1所示动态储存区、静态储存区结构。将代码放在静态区。动态区的堆栈实现，我们是通过3个不同的数组实现。v\_stack是变量栈，等价于图中所示堆区；t\_stack是临时变量栈，activity是活动记录栈，这两个数组一起构成了图中所示的栈区<sup>1</sup>。

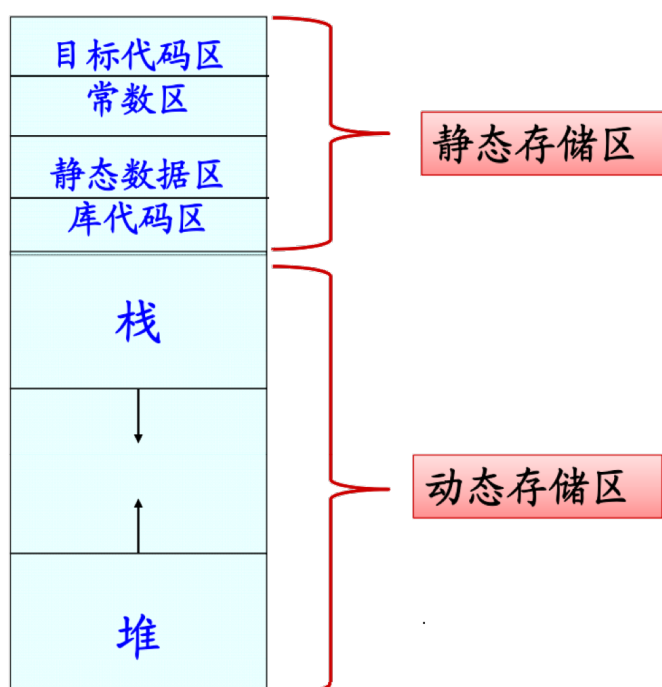


图 5.1 运行时的存储组织

### 5.2 源文件及函数说明

#### 5.2.1 源文件说明

解释程序生成涉及到的文件及其作用如下所示。

- awesomeCC/back-end/include/interpreter.h: 解释器头文件。
- awesomeCC/back-end/src/interpreter.cc: 解释器源文件。

<sup>1</sup>分开设计的目的是便于 debug

## 5.2.2 数据结构及函数说明

- Interpreter 类

- index, int 类型, 相当于 pc 指针。
- v\_size, int 类型, 变量栈大小<sup>1</sup>。
- t\_size, int 类型, 临时变量栈大小。
- code, Quadruple 数组类型, 所有的中间代码。
- v\_stack, double 数组类型, 变量栈。
- t\_stack, double 数组类型, 临时变量栈。
- activity, string 类型, 活动记录栈。
- execute 函数, 供外部调用的执行四元式的函数。包括初始化变量栈、pc 指针置 0、在 `index < code.size()` 的条件下一直执行。
- \_execute 函数, 判断四元式种类, 转发给 \_calc、\_print、\_assign、\_jump、\_pop、\_push 函数处理。
- \_getValue 函数, 参数接收字符串, 返回一个 double。  
如果是地址则去地址那里取值返回; 如果是常量就转换为 double 返回; 如果是数组项的相对寻址, 对于数组使用 \_getAddress 方法寻找基地址, 对于偏移量递归调用 \_getValue 函数获得, 最后取值返回。
- \_getAddress 函数, 参数接收字符串, 返回改变量所在的地址。
- \_calc 函数, 执行 ADD, SUB, MUL, DIV, MOD 类型的四元式。  
调用 \_getValue 函数处理 arg1, arg2, 调用 \_getAddress 函数处理 res, 然后存入变量表或者临时变量表中。
- \_print 函数, 执行 PRINT 四元式。  
如果 arg1 不为空, 则输出 arg1, 否则输出换行。
- \_assign 函数, 执行 MOV 四元式。  
对 res 调用 \_getAddress, 对 arg1 调用 \_getValue, 然后存入变量表或者临时变量表中。

---

<sup>1</sup>这里为了方便随机访问, 使用数组实现的栈, 所以记录栈以便在空间不足的时候动态增加空间

- `_jump` 函数，执行J，JE，JNE，JG，JL四元式。  
对于J直接调用 `_getValue` 获得跳转地址直接跳转。对于其他则比较arg1，arg2是否满足跳转条件，如果满足，则调用 `_getValue` 获得跳转地址进行跳转，否则 `index = index + 1`。
- `_pop` 函数，执行POP四元式。  
对于res，调用 `_getAddress` 获得地址，然后将activity栈顶元素弹出，存入res的地址。
- `_push` 函数，执行PUSH四元式。  
对于res，调用 `_getValue` 获得其值，然后压入activity栈。

### 5.3 实例说明

解释器的api封装在 `awesomeCC/back-end/backend_api.h` 中，输入的参数是中间代码的文件路径，如果读取成功，则直接执行；如果有错误，报错。

4.4节中对判断闰年的代码进行了中间代码生成，这里正好使用该中间代码进行解释执行。使用命令 `./acc leapYear.ac.ic -i` 执行4.4中得到的的中间代码文件 `leapYear.ac.ic`，结果如图5.2所示<sup>1</sup>。

```


1  MOV,2000,,v0
2  MOD,v0,4,t0
3  JE,t0,0,4
4  J,,,14
5  MOD,v0,100,t1
6  JNE,t1,0,10
7  J,,,7
8  MOD,v0,400,t2
9  JE,t2,0,10
10 J,,,14
11 PRINT,v0,,
12 PRINT,"is leap year.",,
13 PRINT,,,
14 J,,,17
15 PRINT,v0,,

```

<sup>1</sup>显示的代码中行号从1开始，但是解释的时候四元式的index是从0开始的

```
16 PRINT,"is not leap year.",,  
17 PRINT,,,  
18 J,,,18
```

Listing 5.1 leapYear.ac.ic



```
~/Workspace/CompilePrinciple/awesomeCC/demo ➤ master • ./acc leap_year_simple.ac.ic -i  
2000 is leap year.
```

图 5.2 判断闰年的中间代码执行结果

## 6 图形用户界面 (GUI)

### 6.1 GUI 概述

在图形用户界面 (Graphical User Interface, 简称 GUI) 中, 计算机画面上显示窗口、图标、按钮等图形, 表示不同目的之动作, 用户通过鼠标等指针设备进行选择。本次课程设计的 GUI 是一个强大的集成开发环境 (IDE), 我们将它取名为 **Awesome Studio**。从设计之初, 我们就以高标准为原则, 高品质为目标, 用心打磨每一个像素, 竭尽全力创造一个让程序员满意的 IDE。

### 6.2 GUI 的组成

#### 6.2.1 窗口

在窗口中, 根据各种数据/应用程序的内容设有标题栏, 一般放在窗口的最上方, 并在其中设有最大化、最小化 (隐藏窗口, 并非消除数据)、最前面、缩进 (仅显示标题栏) 等动作按钮, 可以简单地对窗口进行操作。

**Awesome Studio** 的 GUI 介于单一文件界面 (Single Document Interface) 和多文件界面 (Multiple Document Interface) 之间, 在窗口中, 可以选择一个数据在一个窗口内完成的方式, 也可以选择在一个窗口之内进行多个数据管理的方式。

**Awesome Studio** 的所有窗口都进行了渲染与重绘。**Wndproc** 是 Windows 操作系统向应用程序发送一系列消息之一, 每个窗口会有一个窗口过程的回调函数, 分别是窗口句柄、消息 ID、**WPARAM**、**LPARAM**。**Awesome Studio** 通过重写 **Wndproc**, 来实现一部分窗口的拖拽与自定义大小功能。

**Awesome Studio** 在窗口边缘设计了呼吸灯特效。呼吸灯是指灯光在程序的控制之下完成由亮到暗的逐渐变化, 感觉好像是人在呼吸, 起到一个通知提醒的作用。我们以分割条 (splitter) 控件为呼吸灯的边缘, 通过设置一个时钟 **Timer** 来实现颜色变化。当窗口失去焦点的时候, 便会将该时钟 **Timer** 禁用。一旦窗口获得新的焦点, 将会自动调用该时钟 **Timer**, 来实现窗口边缘的 RGB 颜色渐变, 直到变化为设定的最终颜色为止。

#### 6.2.2 菜单

**Awesome Studio** 的 GUI 设计了即时菜单 (又称功能表、上下文菜单 (Context Menu))。与应用程序准备好的层次菜单不同, 在菜单栏以外的地方, 通过鼠标的第二按钮调出的菜单称为“即时菜单”。根据调出位置的不同, 菜单内容即时变化, 列出所指示的对象目前可以进行的操作。

Awesome Studio 的所有菜单都进行了渲染与重绘。在 `eilRenderer.cs` 文件中，您可以看到相关渲染与重绘代码。

### 6.2.3 状态栏

Awesome Studio 的状态栏是一些代码关键信息的载体，例如，在状态栏中，会显示代码 Debug 的简要结果反馈信息、文档编码格式、当前行号等等。



## 7 集成开发环境

### 7.1 集成开发环境的概述

在集成开发环境（Integrated Development Environment，简称 IDE，也称为 Integration Design Environment、Integration Debugging Environment）是一种辅助程序开发人员开发软件的应用软件，在开发工具内部就可以辅助编写源代码文本、并编译打包成为可用的程序，有些甚至可以设计图形接口。

IDE 通常包括编程语言编辑器、自动构建工具、通常还包括调试器。有些 IDE 包含编译器或解释器，如微软的 Microsoft Visual Studio，有些则不包含，如 Eclipse、SharpDevelop 等，这些 IDE 是通过调用第三方编译器来实现代码的编译工作的。有时 IDE 还会包含版本控制系统和一些可以设计图形用户界面的工具。许多支持面向对象的现代化 IDE 还包括了类别浏览器、对象查看器、对象结构图。虽然目前有一些 IDE 支持多种编程语言（例如 Eclipse、NetBeans、Microsoft Visual Studio），但是一般而言，IDE 主要还是针对特定的编程语言而量身打造（例如 Visual Basic）。

一个优秀的集成开发环境，在于能够将程序员从繁重的大量重复而无意义的劳动中解放出来，让程序员充当软件的大脑，具体的行为动作应该由 IDE 来帮助程序员实现。为此，我们在设计 Awesome Studio 的时候，除了让 Awesome Studio 具有基本的代码编辑功能之外，还开创性地增加了四大智能感知功能：Structure Sense 结构感知、Coding Sense 编程感知、Color Sense 对象着色感知和 File Sense 文件感知功能。

### 7.2 Awesome Studio 的基础功能

#### 7.2.1 文件操作

Awesome Studio 可以打开任意类型的文件，默认是以 UTF-8 编码方式打开的。Awesome Studio 可以记住最近打开过的文件，方便用户快捷打开上一次未完成的工作，提高编程效率。

Awesome Studio 同样可以自由保存编辑好的文件。在 Windows 7 或更高的版本中，相对于 Windows XP 系统，为了在“系统管理员”和“标准用户”两者之间的操作权限及安全性上获取平衡，Microsoft 增加了用户帐户控制（User Account Control, UAC）。当用户的某些动作可能会影响系统的安全及稳定性时，例如变更系统设置、运行未经微软认证的程序等，UAC 便会弹出提示视窗，在运行前要求提供管理员的账户及密码，且该对话框之外的屏幕其他部分都会变暗，让用户不能进行其他操作提醒用户操作。如果用户的组群是“系统管理员”，则只需在弹出的对话框中选择“允许”或“不允许”，如

果用户的组群是“标准用户”，则需请求管理员的授权及密码。UAC 会根据四个层面触发相应的提示视窗：UAC 首先会拦截高危险性的程序及政策不容许的活动；如果该程序或活动并不违反政策或不属于高危险性，UAC 会首先衡量是否由 Windows 发行的程序，如果答案是否定的，UAC 继而会判断该程序是否来自获微软认证的合法发行者；如果 UAC 最终无法判断程序来源，就会让用户谨慎决定是否允许程序运行。因此，如果 Awesome Studio 没有获得管理员权限，将无法将文件保存至某些特定目录。用户只需要赋予 Awesome Studio 管理员权限，便可以任意保存所编辑的文件了。

同时 Awesome Studio 支持新建文件，默认是 UTF-8 编码，且以 DOS 风格的 CRLF 行尾。Awesome Studio 支持新建窗口，即快速克隆自身进程，从而应对多任务处理的相关场景。

另外，用户可以通过点击顶部文件名，迅速打开文件所在目录，方便用户进行文件管理。

### 7.2.2 代码编辑

Awesome Studio 拥有一切记事本程序的基本编辑功能。在顶部菜单栏，您可以进行撤销、重做或者是查找、替换等操作。

得益于 C# 优秀的 winform 界面设计，Awesome Studio 的查找窗口或者是替换窗口都是内嵌与代码编辑框内的，用户再也不用担心找不到用户子窗口太多的情况发生了。

在代码编辑框区域，用户可以通过打开右键菜单的方式，方便地进行更进一步的代码编辑操作。在设计 Awesome Studio 的时候，我们遵循大道至简的设计原则，尽可能删除重复的功能。用户可以在右键菜单中找到复制、粘贴、剪切、文本全选等基础功能，还可以在特定行添加与删除断点相关的标记。作为一款 IDE 软件，Awesome Studio 同样支持代码自动缩进、快速屏蔽与解除相关代码段、克隆代码行以及克隆代码行评论等强大功能。

另外，用户可以通过点击底部状态栏，进行快速代码行定位。以上几乎所有的代码编辑操作，都支持符合常规用户行为习惯的快捷键，方便用户迅速进入工作状态。

### 7.2.3 界面布局

Awesome Studio 允许用户自定义窗口界面。通过设置顶部菜单栏的视图选项，用户可以随心所欲地打开和关闭代码结构视图、代码小地图、控制台窗口。

在设计窗体时，这两个属性特别有用，如果用户认为改变窗口的大小并不容易，应确保窗口看起来显示的不是那么乱，并编写很多代码来实现这个效果，许多程序解决这个问题都是禁止给窗口重新设置大小。

这显然是解决问题的最简单的方法，但不是最好的方法，因此引入了 `Anchor` 和 `Dock` 属性。

当在 C# 项目开发中，在窗体界面的设置经常用到 `Dock` 属性值。当容器中的控件的 `Dock` 属性设置为 `Fill` 时，可能会覆盖其他 `Dock` 属性为 `Top`, `Bottom`, `Right`, `Left` 的控件。为了避免出现覆盖现象，可以将被覆盖的控件置于底层就可以（在 `Panel` 面板上“右键” - “置于底层”），如下图所示。应用 `Dock` 时，越是底层的控件，其优先级越高。

`Awesome Studio` 采用了 `Dock` 界面布局技术，通过绑定窗口大小改变事件，来动态调整窗口布局。这相比于采用时钟监听的方式，大大节省了软件对内存的占用。

#### 7.2.4 调试与编译

作为一款 IDE 程序，`Awesome Studio` 支持对代码的调试与编译。

程序调试是将编制的程序投入实际运行前，用手工或编译程序等方法进行测试，修正语法错误和逻辑错误的过程。这是保证计算机信息系统正确性的必不可少的步骤。编完计算机程序，必须送入计算机中测试。根据测试时所发现的错误，进一步诊断，找出原因和具体的位置进行修正。

编译程序（Compiler, compiling program）也称为编译器，是指把用高级程序设计语言书写的源程序，翻译成等价的机器语言格式目标程序的翻译程序。编译程序属于采用生成性实现途径实现的翻译程序。它以高级程序设计语言书写的源程序作为输入，而以汇编语言或机器语言表示的目标程序作为输出。编译出的目标程序通常还要经历运行阶段，以便在运行程序的支持下运行，加工初始数据，算出所需的计算结果。

`Awesome Studio` 通过截取命令行返回消息的方式，与编译器进行通信，从而执行代码，并且可以取回相关代码的返回信息。

`Awesome Studio` 支持从外部打开命令行控制台的功能。这一功能对于一些脚本语言例如 `Python` 语言的扩展包安装是非常有用的。

### 7.3 IntelliSense 智能感知

#### 7.3.1 Structure Sense 结构感知

`Structure Sense` 是 `Awesome Studio` 的核心功能之一，也是最重要的功能之一。`Structure Sense` 主要包括两个方面的内容。

第一个部分是对代码结构的感知。其实现原理是通过正则表达式匹配关键字，从而实现结构感知的功能。

正则表达式，又称规则表达式。（英语：Regular Expression，在代码中常简写为 `regex`、`regexp` 或 `RE`），计算机科学的一个概念。正则表达式通常被用来检索、替换那些符合某

个模式 (规则) 的文本。

正则表达式是对字符串操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑。

在 `ReBuildObjectExplorer` 函数中，通过设置 `Regex regex` 这个正则变量，来实现对关键字的筛选与过滤。

第二个部分是代码地图。如果开启了代码地图的功能，那么将会在界面的右侧显示代码地图。

代码地图和代码编辑区的滚动条是同步的。您可以直接拖动代码地图的滚动条，来实现对代码的垂直浏览。

### 7.3.2 Coding Sense 编程感知

Coding Sense 是 Awesome Studio 的核心功能之一，也是最重要的功能之一。

Coding Sense 是许多功能的总称，包括列出成员、参数信息、快速信息和完成单词。使用这些功能，可以详细了解使用的代码、跟踪键入的参数，还可以通过轻敲几个按键来添加属性和方法调用。

键入触发器字符（例如，C# 代码中的句点 (.)）后，显示类型（或命名空间）中的有效成员列表。如果继续键入字符，则会筛选该列表，以只包含以这些字符开头的成员。选择项目后，按回车键可以将其插入到代码中。如果选择一个项目并键入句点，该项目显示时后面会跟一个句点，该句点会引出另一个成员列表。如果选择一个项目，然后插入它，则你会获得该项的快速信息。在成员列表中，左边的图标表示成员类型，如命名空间、类、函数或变量。有关图标的列表，请参阅类视图和对象浏览器图标。该列表可能会很长，你可以按 `PAGE UP` 和 `PAGE DOWN` 上下移动列表内容。输入足以区分变量名、命令名或函数名的字符后，完成单词功能可以将剩余部分补充完整。

### 7.3.3 Color Sense 对象着色感知

Color Sense 是 Awesome Studio 的核心功能之一，也是最重要的功能之一。Color Sense 可以对代码进行智能着色。例如，通过正则表达式的匹配方法，可以准确区分某特定类的对象的方法与控制语句关键字。

Color Sense 定义的颜色除了部分是 C# 自带的颜色库以外，其余部分都是以十六进制网页颜色格式来定义的。由于网页 (WEB) 是基于计算机浏览器开发的媒体，所以颜色以光学颜色 RGB（红、绿、蓝）为主。网页颜色是以 16 进制代码表示，一般格式为 `#DEFABC`（字母范围从 A-F, 数字从 0-9）；如黑色，在网页代码中便是：`#000000`（在 css

编写中可简写为 #000)。当颜色代码为 #AABB11 时, 可以简写为 #AB1 表示, 如 #135 与 #113355 表示同样的颜色。RGB1、RGB4、RGB8 都是调色板类型的 RGB 格式, 在描述这些媒体类型的格式细节时, 通常会在 BITMAPINFOHEADER 数据结构后面跟着一个调色板 (定义一系列颜色)。它们的图像数据并不是真正的颜色值, 而是当前像素颜色值在调色板中的索引。以 RGB1 (2 色位图) 为例, 比如它的调色板中定义两种颜色值依次为 0x000000 (黑色) 和 0xFFFFFFFF (白色) … (每个像素用 1 位表示) 表示对应各像素的颜色为: 黑黑白白黑黑白白黑黑白白…。

对象着色感知的实现原理, 首先是要监听代码编辑区的改动事件。一旦代码编辑区的代码有所改动的时候, Color Sense 便会对改动的区域进行审查。当改变的内容与 Color Sense 设定的正则表达式相匹配的时候, 便会对改动的内容实现智能着色。

#### 7.3.4 File Sense 文件感知

File Sense 是 Awesome Studio 的核心功能之一, 也是最重要的功能之一。

File Sense 的实现原理主要是通过设置一个 Timer 来监视当前文件的变化情况, 来实现版本控制功能。

版本控制最主要的功能就是追踪文件的变更。它将什么时候、什么人更改了文件的什么内容等信息忠实地记录下来。每一次文件的改变, 文件的版本号都将增加。除了记录版本变更外, 版本控制的另一个重要功能是并行开发。软件开发往往是多人协同作业, 版本控制可以有效地解决版本的同步以及不同开发者之间的开发通信问题, 提高协同开发的效率。并行开发中最常见的不同版本软件的错误 (Bug) 修正问题也可以通过版本控制中分支与合并的方法有效地解决。一旦文件在一段时间内更新了但是没有保存, File Sense 将会在文件所在目录对文件进行智能备份, 从而实现类似于 Git 的简易版本控制功能, 保证用户在突然断电的情况下不至于损失重要文件。

## 8 感想

这次编译课设可谓是非常充实。在仅仅一个月的时间里从 0 开始编写一个编译器和编辑器。这次课设让我受益匪浅，因为过程中讲在课堂中学习的很多理论知识都落实到了代码里。之前上理论课的时候有一些地方不理解为什么要这么做、为什么要设置这些定理，在这次课程设计中都得到了解决，让我对编译原理的知识有了更加深刻的理解。

在编写报告的时候，我仔细回想每行写过的代码，想到了好多改进的地方。比如加入函数调用的四元式指令而不是在翻译的时候回填让 pc 入栈；比如可以用更方便的寻址方式；比如报错需要更加规范一点并且支持同时报多个错。总之，这个项目在未来还有很多可以改进的地方，我们不会因为课设结束就放弃掉它。

和我的队友合作的过程非常愉快，我们之前也合作过别的小型项目，不过 1w+ 行的项目是第一次。虽然过程中遇到了一些合并代码带来的 bug，也遇到了意见不合的时候，但是最后都顺利解决了，完成了本次课设。而且也大大的提升了我们的工程能力，这是我第一次用 google test 和 travisCI。

总的来说本次课程设计让我收获颇丰，不仅巩固了再编译原理课上学到的理论知识，还锻炼了工程能力、合作能力。

## 附录 A 小组分工

本次项目中，我和队友刘瀚文总的分工 50%、50%。在编译器（AwesomeCC）的实现上，我完成了 60%，我的队友完成了 40%；在编辑器（AwesomeStudio）的实现上，我完成了 40%，我的队友完成了 60%。

编译器部分，我完成了语法分析中的语法树输出、函数定义的处理、函数调用的处理、赋值语句的处理、赋值语句的处理、语句块的处理、return 语句的处理；语义分析及中间代码生成；解释执行的寻址、取值、pop 语句执行、push 语句执行。我队友完成了词法分析；语法分析中的变量声明的处理、while 语句的处理、for 语句的处理、if 语句的处理、print 语句的处理，else 语句的处理；解释执行的跳转语句执行、赋值语句执行、print 语句执行、计算语句执行。

IDE 部分，我完成了 File Sense 和 Color Sense、安全色与图标资源绘制、基本文本编辑器功能，我的队友完成了 Structure Sence 和 Coding Sense、界面响应式设计、控件渲染与重绘、Win32 API 封装与调用、FastColoredTextBox 的重构与封装、多线程并发处理设计。

## 附录 B 错误信息表

### 词法分析错误

- in digit constant, too many dots in one number  
被识别为常量的 token 中出现多个
- in string constant, lack of ' 或"  
被识别为字符串常量的 token 中引号未匹配
- unidentified symbol  
无法识别的符号

### 语法分析错误

- Everything should be wrapped in a class.  
程序中需要一个类来包裹成员函数和变量
- in main, unidentified symbol  
无法识别的类成员
- in print function, arguments should be wrapped in '()'   
print 函数需要使用括号将参数包裹
- in array initialization, expected ' , ' or '}' after a digital constant  
值列表中，数字后的字符无法识别
- in array initialization, expected '{'   
数组初始化的时候值必须放在大括号里
- in statement, unrecognized symbol  
声明语句中有无法识别的符号
- in statement, expected ']' after a statement of an array  
声明语句中数组的声明中括号未匹配
- in expression, expected '(' before ')'   
表达式缺少左小括号



- in expression, expected ‘)’ after ‘(  
表达式缺少右小括号
- in expression, unrecognized symbols  
表达式中有无法识别的符号
- in function statement’s parameter list, should be ‘,’ or ‘)’ after  
函数声明的参数列表中，形式参数后的符号无法识别
- in function statement’s parameter list, unidentified parameter type found  
函数声明的参数列表中，形式参数的类型错误
- in return, expected an expression or semicolon after ‘return’  
return 语句缺少分号或者表达式
- in block, unidentified symbols found  
语句块中有无法识别的符号
- in block, expected }  
语句块缺少右大括号
- in function call, expected ‘(’ after function name  
函数调用缺少左小括号
- in assignment, expected ‘=’ after an identifier  
赋值语句中缺少等号
- in for, Expected ‘{’ after ‘for (assignment; condition; assignment)’  
for 语句缺少左大括号
- Expected ‘(’ after ‘for’  
for 语句缺少左小括号
- Expected ‘{’ after ‘while (condition)’  
while 语句缺少左大括号
- Expected ‘(’ after ‘while’  
while 语句缺少左小括号

- in if, expected '{' after 'if (condition)'  
if 语句缺少左大括号
- in if, expected '(' after 'if'  
if 语句缺少左小括号
- in if, expected '{' after 'else' or 'if'  
else 语句缺少左大括号

## 语义分析错误

- \* is not allowed in a root of a class  
\* 不允许是类成员
- function \* is not defined before use  
函数 \* 没定义
- variable \* is not defined before use  
变量 \* 没定义

## 附录 C 公用库文件

在报告中有部分公用库文件未进行详细说明，此处补充。

### error.h

- Error 类
  - line\_number, int 类型, 错误行号
  - pos, int 类型, 错误列号
  - errorMsg, string 类型, 错误信息
  - 输出流, 用于格式化输出报错信息

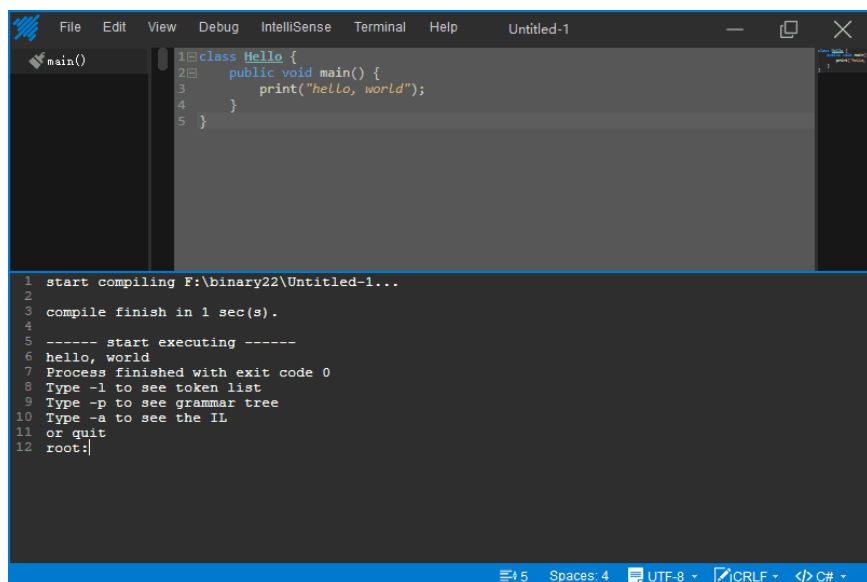
### str\_tools.h

- char2string 函数, char 转 string
- string2int 函数, string 转 int
- string2double 函数, string 转 double
- int2string 函数, int 转 string
- double2string 函数, double 转 string
- token2string 函数, TOKEN\_TYPE\_ENUM 转对应的字符串

### file\_tools.h

- readSourceFile 函数, 输入 ac 代码文件地址, 返回字符串数组
- readInterCodeFile 函数, 输入 ic 中间代码文件地址, 返回四元式数组

## 附录 D 界面展示

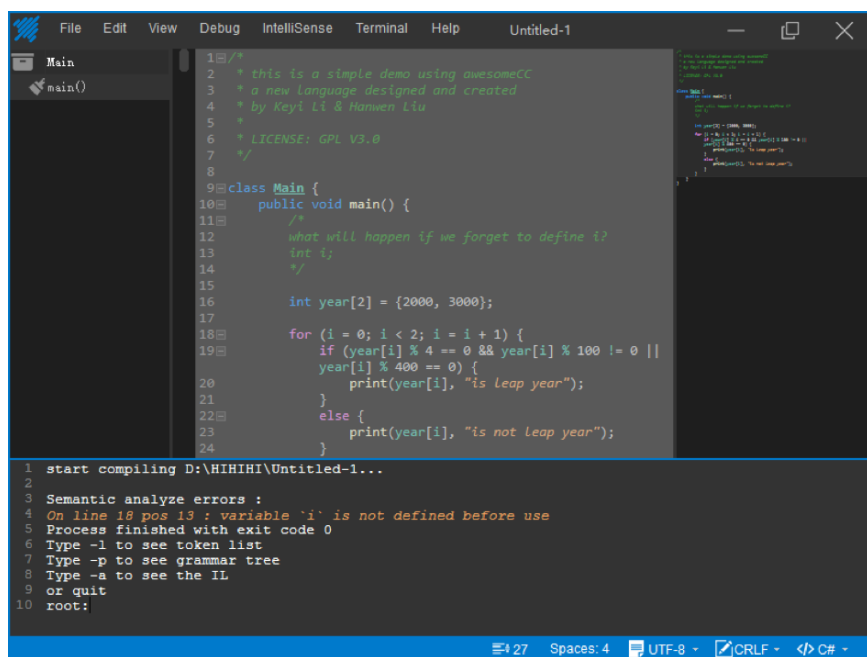


```
File Edit View Debug IntelliSense Terminal Help Untitled-1

1 class Hello {
2     public void main() {
3         print("hello, world");
4     }
5 }

1 start compiling F:\binary22\Untitled-1...
2
3 compile finish in 1 sec(s).
4
5 ----- start executing -----
6 hello, world
7 Process finished with exit code 0
8 Type -l to see token list
9 Type -p to see grammar tree
10 Type -a to see the IL
11 or quit
12 root:|
```

图 1 成功运行示例



```
File Edit View Debug IntelliSense Terminal Help Untitled-1

Main
main()

1 /*
2  * this is a simple demo using awesomeCC
3  * a new language designed and created
4  * by Keyi Li & Hanwen Liu
5  *
6  * LICENSE: GPL V3.0
7  */
8
9 class Main {
10     public void main() {
11         /*
12          * what will happen if we forget to define i?
13          * int i;
14          */
15
16         int year[2] = {2000, 3000};
17
18         for (i = 0; i < 2; i = i + 1) {
19             if (year[i] % 4 == 0 && year[i] % 100 != 0 ||
20                 year[i] % 400 == 0) {
21                 print(year[i], "is Leap year");
22             }
23             else {
24                 print(year[i], "is not Leap year");
25             }
26         }
27     }
28 }

1 start compiling D:\HIHIHI\Untitled-1...
2
3 Semantic analyze errors :
4 On line 18 pos 13 : variable 'i' is not defined before use
5 Process finished with exit code 0
6 Type -l to see token list
7 Type -p to see grammar tree
8 Type -a to see the IL
9 or quit
10 root:|
```

图 2 报错示例

```
~/Workspace/CompilePrinciple/awesomeCC/demo master • ./acc --help
OVERVIEW: awesome CC compiler, developed by Keyi Li & Hanwen Liu

USAGE: acc file_name [options] <inputs>

OPTIONS:
-a, --assembler      generate inter code(Quadruple) for a source AC file
-h, --help           get help on awesomeCC command line arguments
-i, --interpreter    interpret and execute an inter code file
-l, --lexer          lexical analyze a source AC file
-o, --output         the output file path
-p, --parser         syntax analyze a source AC file
-v, --version        display awesomeCC version

EXAMPLE:
acc source.ac -l
acc source.ac -p
acc source.ac -a
acc source.ac.ic -i
acc source.ac
acc -h
acc -v
```

图 3 在命令行单独使用 acc 示例

## 参 考 文 献

- [1] AHO A V. 编译原理 [M]. 机械工业出版社, 2009.