

2024-2025第1学期

《编译原理课程设计》报告

|  |  |
| --- | --- |
| **题 目：** | 类GO语言的编译器实现 |
| **学 院：** | 计算机学院 |
| **专 业：** | 计算机科学与技术实验班 |
| **年级班级：** | 2022级 |
| **课题成员：** | 202283290436韩铭佳 |
| 202283290140张宇 |
| 202283290159张瑞晨 |
| **任课教师：** | 郑关胜 |

**二Ｏ二二 年 十一 月 日**

|  |  |
| --- | --- |
| **实践小组成员分工及职责** | |
| 学号姓名 | 任务分工 |
| 202283290436韩铭佳 | 文法设计  词法语法框架构建  变量赋值语义实现 |
| 202283290140张宇 | 语法树构建  可视化分析程序 |
| 202283290159张瑞晨 | 控制结构设计  语义功能实现 |

类GO语言的编译器实现

摘要

本文深入探讨了编译器中的词法分析、语法分析和语义分析三个核心阶段。在词法分析阶段，我们利用Flex工具将源代码分解成标记（tokens），其中包括关键字、标识符、常量和运算符等。为了更有效地管理这些标记，我们采用了单词链结构，这是一种高效的数据结构，能够优化标记的存储和检索。特别地，我们实现了lookup\_word函数，该函数使用线性搜索在单词链中查找单词，但在单词链很长时可能会效率低下。为了提高查找效率，可以考虑使用散列表（hash table）代替链表。散列表通过将键（这里是单词）映射到表中的一个位置来实现快速查找，这通常可以提供接近常数时间的查找性能。

语法分析阶段使用Yacc/Bison工具，通过定义产生式规则来描述语言的语法结构，并构建解析器，该解析器读取标记序列并尝试根据文法规则构建抽象语法树（AST）。

语义分析阶段与语法分析并行进行，通过在Yacc/Bison的产生式规则中嵌入语义动作来实现，这些动作包括更新符号表、计算表达式的值、生成中间代码和错误处理等。如果源代码符合语法规则，解析器将构建出AST。错误处理是编译过程中的一个重要环节，它在语义分析阶段被包含，用于报告和定位代码中的问题，例如类型不匹配或未定义的标识符。 我们的代码集中在编译过程的中间阶段，确保源代码在结构和语义上的正确性，并通过嵌入语义动作实现语法制导翻译，有效将源代码转换为目标代码。语法制导翻译（Syntax-Directed Translation, SDT）的流程涉及将源代码中的语法结构转换为一系列语义动作，这些动作实现了源代码的语义。语法制导翻译将语法分析和语义分析结合在一起，使得这两个步骤是交织在一起的，而不是传统编译过程中分开的两个步骤。

关键词：词法分析、语法分析、抽象语法树、语法制导翻译、单词链结构

目 录

[1. 课程设计需求分析 4](#_Toc736535965)

[1. 设计语言的基本语法结构 4](#_Toc597827702)

[2. 实现词法分析和语法分析 4](#_Toc417995336)

[3. 测试和结果可读性 4](#_Toc1250920501)

[2.编译过程 5](#_Toc294961309)

[1. 词法分析（Lexical Analysis） 5](#_Toc1433697941)

[2. 语法分析（Syntax Analysis） 5](#_Toc1526588904)

[3. 语义分析（Semantic Analysis） 5](#_Toc802291524)

[4. 错误处理（Error Handling） 5](#_Toc353646859)

[3.编译程序的结构 6](#_Toc2101831949)

[3.1词法分析阶段 6](#_Toc322947770)

[3.1.1 token识别 6](#_Toc405967298)

[3.1.2单词链生成 7](#_Toc1298072894)

[3.1.3性能考虑 9](#_Toc1996845725)

[3.2语法分析阶段 9](#_Toc1142905337)

[3.2.1终结符和非终结符 9](#_Toc920573203)

[3.2.2规约规则 11](#_Toc1939515306)

[3.3语义分析阶段 13](#_Toc2042756316)

[4.结果显示与解释评估 17](#_Toc1654070984)

[5. 小结 21](#_Toc1482114749)

[参考文献 21](#_Toc433756766)

[致谢 22](#_Toc470807148)

[附录 23](#_Toc1818323515)

[1. 词法分析部分代码 zzh.l 23](#_Toc1520006463)

[2. 语法分析部分代码 zzh.y 27](#_Toc868096981)

[3. 可视化语法树构建 41](#_Toc2105638151)

# 课程设计需求分析

1. 设计语言的基本语法结构

定义语言规范：首先需要定义一个简单的编程语言的基本语法结构。这可以包括但不限 于变量声明、赋值语句、条件语句、循环语句、函数定义等。

语法规则：明确语言的语法规则，包括终结符和非终结符的定义以及它们的产生式规则。

2. 实现词法分析和语法分析

自动生成或手动实现：词法分析器和语法分析器可以利用现有的工具自动生成（如使用 Flex和Bison），也可以自行构造实现。

词法分析器：

负责将源代码字符流转换为标记（tokens）。

需要处理关键字、标识符、常量、运算符等的识别。

应能识别并报告词法错误（如非法字符、未闭合的括号等）。

语法分析器：

负责根据语言的语法规则分析标记序列，并构建抽象语法树（AST）。

需要实现错误恢复机制，以处理语法错误并继续分析。

应能识别并报告语法错误（如类型不匹配、缺少分号等）。

3. 测试和结果可读性

测试用例：设计测试用例，包括正常案例和异常案例，确保测试覆盖所有语言特性。

测试执行：执行测试用例，确保编译器能正确处理各种输入，并生成正确的输出。

结果验证：验证编译器生成的AST或其他中间表示是否正确，确保它们能够准确地反 映源代码的结构。

结果可读性：确保编译器的输出（如AST、错误信息等）是可读的，便于调试和验证。

# **2.编译过程**

1. 词法分析（Lexical Analysis）

步骤：使用Flex工具，源代码被转换成一系列的标记（tokens）。这些标记包括关键 字、标识符、常量、运算符等。

输出：标记序列，作为语法分析的输入。

2. 语法分析（Syntax Analysis）

步骤：使用Yacc/Bison进行语法分析。它定义了一系列的产生式规则，这些 规则描 述了语言的语法结构。

工具：Yacc/Bison根据这些规则构建一个解析器，该解析器读取标记序列，并尝试根 据文法规则构建抽象语法树（AST）。

输出：如果源代码符合语法规则，解析器将构建出AST。

3. 语义分析（Semantic Analysis）

步骤：在语法分析的同时，代码执行语义分析。这是通过在Yacc/Bison的产生式 规 则中嵌入语义动作来实现的，这些动作在解析过程中被执行。

语义动作：包括更新符号表、计算表达式的值、生成中间代码和错误处理等。

输出：经过语义检查和处理的AST，为代码生成做准备。

4. 错误处理（Error Handling）

步骤：在语义分析阶段包含了错误处理。如果检测到语义错误，如类型不匹配或未定义 的标识符，编译器将报告错误并可能停止编译过程。

输出：错误信息，帮助用户定位和修复代码中的问题。

我们的代码主要集中在编译过程的中间阶段，即词法分析，语法分析和语义分析。这些阶段是编译器的核心，它们确保源代码不仅在结构上正确，而且在语义上也是合法的。通过在产生式规则中嵌入语义动作，我们的代码实现了语法制导翻译，这是一种将源代码转换为目标代码的有效方法。

# **3.编译程序的结构**

## 3.1词法分析阶段

词法分析器的实现使用Flex工具编写。Flex是一个强大的工具，用于生成词法分析器，它能够根据定义的词法规则自动生成相应的C代码。我们在词法阶段识别token并存入，并把标识符使用单词链数据结构进行存储。

### 3.1.1 token识别

词法分析是编译器前端的一个步骤，它的任务是将源代码文本分解成一系列的标记（tokens），这些标记是构成程序的基本元素，比如关键字、标识符、常量、运算符等。

以下是词法分析器识别的token的表格，包括每个标记的正则表达式、描述和返回值：

表 1 token识别表

| 正则表达式 | 描述 | 返回值 |
| --- | --- | --- |
| [\n]+ | 匹配一个或多个换行符，更新行号 | 无 |
| [ \t\r]+ | 忽略空白字符 | 无 |
| "var" | 匹配关键字 var | VAR |
| "if" | 匹配关键字 if | IF |
| "else" | 匹配关键字 else | ELSE |
| "for" | 匹配关键字 for | FOR |
| "while" | 匹配关键字 while | WHILE |
| "int" | 匹配关键字 int | INT |
| "double" | 匹配关键字 double | DOUBLE |
| "string" | 匹配关键字 string | STRING |
| "bool" | 匹配关键字 bool | BOOL |
| "print" | 匹配关键字 print | PRINT |
| ":" | 匹配冒号 : | COLON |
| "=" | 匹配赋值运算符 = | ASSIGN |
| "==" | 匹配等于运算符 == | EQ |
| "!=" | 匹配不等于运算符 != | NE |
| "<=" | 匹配小于等于运算符 <= | LE |
| ">=" | 匹配大于等于运算符 >= | GE |
| "<" | 匹配小于运算符 < | LT |
| ">" | 匹配大于运算符 > | GT |
| "[" | 匹配左方括号 [ | LBRACKET |
| "]" | 匹配右方括号 ] | RBRACKET |
| "(" | 匹配左圆括号 ( | LPAREN |
| ")" | 匹配右圆括号 ) | RPAREN |
| "{" | 匹配左花括号 { | LBRACE |
| "}" | 匹配右花括号 } | RBRACE |
| "," | 匹配逗号 , | COMMA |
| "+" | 匹配加号 + | PLUS |
| "-" | 匹配减号 - | MINUS |
| "\*" | 匹配乘号 \* | MULT |
| "/" | 匹配除号 / | DIV |
| -?[0-9]+ | 匹配整数 | NUMBER |
| -?[0-9]+\.[0-9]+ | 匹配浮点数 | DNUM |
| `"([^"\\n] | \.)\*"` | 匹配带引号的字符串 |
| [A-Za-z\_][A-Za-z0-9\_]\* | 匹配标识符 | ID |
| \/\/[^\n]\* | 忽略单行注释 | 无 |
| . | 匹配其他字符，表示词法错误 | 无 |

VAR、IF、ELSE 等都是由 Yacc/Bison 定义的标记代码，用于在语法分析阶段识别相应的关键字。NUMBER、DNUM、STR 和 ID 分别代表数值、浮点数、字符串和标识符。其他符号+、-、\*、/ 等都是运算符或特殊字符。对于词法错误，代码会输出错误信息。

### 3.1.2单词链生成

在词法分析阶段，我们生成单词链（word chain）作为单词表的数据结构，用于存储和查找识别的单词（tokens）或标识符。

单词链单元结构定义包含如下内容：

1.char \*word\_name;

作用：存储单词的名称或标识符的字符串。

类型：字符指针。

重要性：标识符的唯一标识，用于后续的语义分析和代码生成。

2.int word\_type;

作用：标识单词的类型，如整数、浮点数、字符串或关键字等。

类型：整型。

重要性：决定了单词在语法和语义分析中的角色。

3.union { int int\_value; double double\_value; char \*string\_value; } value;

作用：存储单词的值，根据单词类型不同，可能是整数、浮点数或字符串。

类型：联合体，包含三种可能的值类型。

重要性：存储实际的数据值，对于常量和变量的进一步处理至关重要。

4.struct word \*next;

作用：指向链表中下一个struct word节点的指针。

类型：指向struct word的指针。

重要性：维护符号表的链表结构，允许顺序访问和动态扩展。

以下是添加单词函数和查找单词函数的解释：

（1）add\_word函数

add\_word函数用于向单词链中添加一个新的单词（word）。它的参数是一个类型 int type和一个指向单词字符串的指针char \*word。函数的步骤如下：

1.检查重复：使用lookup\_word函数检查单词是否已经存在于单词链中。如果返回 值非零，表示单词已存在，函数返回0。

2.分配内存：如果单词不存在，使用malloc为新的struct word结构体分配内存。

3.初始化结构体：设置新单词的next指针，使其指向当前的word\_list头部。为单词 名称分配内存，并使用strcpy复制单词名称。设置单词的类型word\_type。将新单词 插入到单词链的头部，更新word\_list指针。

4.返回成功：如果单词成功添加，函数返回1。

（2）lookup\_word函数

lookup\_word函数用于在单词链中查找一个给定的单词。它的参数是一个指向要查找 的单词字符串的指针char \*word。函数的步骤如下：

1.遍历单词链：从word\_list开始遍历单词链。

2.比较单词：对链表中每个单词用strcmp函数比较word\_name字段和给定的单词。

3.返回结果：如果找到匹配的单词，返回1。如果遍历完整个链表都没有找到，返回0。

### 3.1.3性能考虑

lookup\_word函数使用线性搜索，这在单词链很长时可能会效率低下。如果需要提高查找效率，可以考虑使用散列表（hash table）代替链表。散列表通过将键（这里是单词）映射到表中的一个位置来实现快速查找，这通常可以提供接近常数时间的查找性能。总结来说，这两个函数共同实现了一个简单的单词链表，用于存储和查找词法分析阶段识别的单词。这种数据结构在编译器设计中非常常见，用于符号表的实现。

## 3.2语法分析阶段

语法分析是编译过程中的关键步骤，负责检查源代码的语法结构是否符合语言规范，并构建抽象语法树（AST）。本报告使用Bison展示语法分析器的设计和实现。我们此处使用LR（1）分析器进行语法分析。（Bison是生成LR(1)分析器的工具）。

LR(1)分析器通过构造一个LR(1)项目集族和状态转移图来进行解析，它能够处理复杂的语法结构，并在解析过程中跟踪一个状态栈和一个符号栈。LR(1)分析器在编译器设计中非常流行，因为它能够处理大多数上下文无关文法，并且相对容易实现。

在语法分析阶段，我们通过词法分析阶段获得的token进行LR(1)分析下的规约检查语法正确性。其中我们规定有终结符和非终结符内容。

### 3.2.1终结符和非终结符

1.终结符表：

表 2 终结符表

| 标记名称 | 数据类型 | 描述 |
| --- | --- | --- |
| ID | char\* | 标识符 |
| NUMBER | int | 整数 |
| DNUM | double | 浮点数 |
| STR | char\* | 字符串 |
| VAR | - | 关键字 var |
| IF | - | 关键字 if |
| ELSE | - | 关键字 else |
| FOR | - | 关键字 for |
| WHILE | - | 关键字 while |
| INT | - | 关键字 int |
| DOUBLE | - | 关键字 double |
| STRING | - | 关键字 string |
| BOOL | - | 关键字 bool |
| PLUS | - | 加号 + |
| MINUS | - | 减号 - |
| MULT | - | 乘号 \* |
| DIV | - | 除号 / |
| ASSIGN | - | 赋值符号 = |
| EQ | - | 等于 == |
| NE | - | 不等于 != |
| LT | - | 小于 < |
| GT | - | 大于 > |
| LE | - | 小于等于 <= |
| GE | - | 大于等于 >= |
| COLON | - | 冒号 : |
| COMMA | - | 逗号 , |
| LBRACKET | - | 左方括号 [ |
| RBRACKET | - | 右方括号 ] |
| LPAREN | - | 左圆括号 ( |
| RPAREN | - | 右圆括号 ) |
| LBRACE | - | 左花括号 { |
| RBRACE | - | 右花括号 } |
| PRINT | - | 关键字 print |

终结符是文法中最小的单位，它们不能再被分解，通常是关键字、标识符、常量和操作符等。

1. 非终结符 (Non-terminals)

表 3 非终结符表

| 标记名称 | 数据类型 | 描述 |
| --- | --- | --- |
| expression | int | 表达式 |
| expression2 | char\* | 字符串表达式 |
| id\_list | char\* | 标识符列表 |
| type | char\* | 数据类型 |
| array\_dimensions | char\* | 数组维度 |
| statement\_list | char\* | 语句列表 |
| condition | int | 条件表达式 |
| if\_statement | char\* | if语句 |
| print\_statement | char\* | 打印语句 |

非终结符是可以被进一步分解的符号，它们表示文法中的复合结构，如表达式、语句等。在Yacc/Bison中，%type声明指定了非终结符的预期数据类型。

### 3.2.2规约规则

1. 程序规则

program → statement\_list

描述：程序由一系列语句（statement\_list）组成。

1. 语句列表规则

statement\_list → statement statement\_list

描述：语句列表（statement\_list）由一个语句（statement）跟另一个语句列表递归组成。

statement\_list → ε

描述：语句列表（statement\_list）也可以为空（ε表示空字符串），表示没有语句。

3.语句规则

statement → variable\_declaration

描述：语句（statement）可以是一个变量声明（variable\_declaration）。

statement → assignment

描述：语句（statement）可以是一个赋值操作（assignment）。

statement → if\_statement

描述：语句（statement）可以是一个条件语句（if\_statement）。

statement → loop\_statement

描述：语句（statement）可以是一个循环语句（loop\_statement）。

statement → print\_statement

描述：语句（statement）可以是一个打印语句（print\_statement）。

4.变量声明规则

variable\_declaration → VAR id\_list type

描述：变量声明（variable\_declaration）由关键字VAR开始，后跟一个标识符列表（id\_list）和一个类型（type）。

variable\_declaration → VAR id\_list type ASSIGN expression

描述：变量声明（variable\_declaration）也可以包括一个赋值表达式（expression），格式为VAR id\_list type = expression。

5.标识符规则

id\_list → ID

描述：标识符列表（id\_list）由单个标识符（ID）组成。

id\_list → ID COMMA id\_list

描述：标识符列表（id\_list）也可以由一个标识符后跟逗号和另一个标识符列表组成。

6.类型规则

type → INT

描述：类型（type）可以是整数类型（INT）。

type → DOUBLE

描述：类型（type）可以是双精度浮点数类型（DOUBLE）。

type → STRING

描述：类型（type）可以是字符串类型（STRING）。

type → BOOL

描述：类型（type）可以是布尔类型（BOOL）。

type → INT array\_dimensions

描述：类型（type）可以是整数数组类型，由INT后跟数组维度（array\_dimensions）组成。

type → DOUBLE array\_dimensions

描述：类型（type）可以是双精度浮点数数组类型，由DOUBLE后跟数组维度（array\_dimensions）组成。

7.数组规则

array\_dimensions → LBRACKET NUMBER RBRACKET

描述：数组维度（array\_dimensions）由左方括号、一个数字和右方括号组成。

array\_dimensions → LBRACKET NUMBER RBRACKET array\_dimensions

描述：数组维度（array\_dimensions）也可以由多个这样的维度组成，表示多维数组。

1. 赋值语句规则

assignment → ID ASSIGN expression2

描述：赋值语句（assignment）由一个标识符（ID）、赋值符号（ASSIGN）和一个字符串表达式（expression2）组成。

assignment → ID ASSIGN expression

描述：赋值语句（assignment）由一个标识符（ID）、赋值符号（ASSIGN）和一个数值表达式（expression）组成。

9.条件判断语句规则

if\_statement → IF LPAREN condition RPAREN LBRACE statement\_list RBRACE

描述：条件语句（if\_statement）由关键字IF开始，后跟左圆括号、条件表达式（condition）、右圆括号和一个语句块（statement\_list）组成。

if\_statement → IF LPAREN condition RPAREN LBRACE statement\_list RBRACE ELSE LBRACE statement\_list RBRACE

描述：条件语句（if\_statement）也可以包括一个ELSE分支，由关键字IF开始，后跟左圆括号、条件表达式（condition）、右圆括号、一个语句块、关键字ELSE和另一个语句块组成。

10.循环语句规则

loop\_statement → FOR ID COLON ID LBRACE statement\_list RBRACE

描述：循环语句（loop\_statement）可以是增强型FOR循环，由关键字FOR开始，后跟一个标识符、冒号、另一个标识符和一个语句块组成。

loop\_statement → WHILE LPAREN condition RPAREN LBRACE statement\_list RBRACE

描述：循环语句（loop\_statement）也可以是WHILE循环，由关键字WHILE开始，后跟左圆括号、条件表达式（condition）、右圆括号和一个语句块组成。

这些规则定义了程序的结构和组成，包括程序、语句、变量声明、类型声明、数组维度、赋值、条件语句和循环语句等。每个规则后面的描述解释了规则的结构和含义。

## 3.3语义分析阶段

我们使用语法制导翻译的语义分析。语法制导翻译（Syntax-Directed Translation, SDT）的流程涉及将源代码中的语法结构转换为一系列语义动作，这些动作实现了源代码的语义。语法制导翻译将语法分析和语义分析结合在一起。在传统的编译过程中，这两个步骤是分开的：首先进行语法分析以构建语法树，然后进行语义分析以检查语义正确性并执行语义动作。而在语法制导翻译中，这两个步骤是交织在一起的。

以下是我们的语义分析阶段。

Program

规则：program: statement\_list

语义动作：将程序结构写入抽象语法树文件，表示程序由语句列表组成。

描述：在语法制导翻译中，程序入口点是语句列表，表示程序的执行从一系列语句开始。

Statement List

规则1：statement\_list: statement statement\_list

语义动作：递归地将每个语句写入抽象语法树文件，表示语句列表由多个语句组成。

描述：这允许程序处理一系列语句，每个语句后跟另一个语句列表形成语句的层次结构。

规则2：statement\_list: /\* 空 \*/

语义动作：表示语句列表可以为空，即没有语句。

描述：这处理了程序中没有执行语句的情况。

Statement

规则1：statement: variable\_declaration

语义动作：将变量声明写入抽象语法树文件。

描述：这表示程序可以声明变量，为后续操作提供存储空间。

规则2：statement: assignment

语义动作：将赋值操作写入抽象语法树文件。

描述：这表示程序可以执行赋值操作，更新变量的值。

规则3：statement: if\_statement

语义动作：将条件语句写入抽象语法树文件。

描述：这表示程序可以根据条件执行不同的代码路径。

规则4：statement: loop\_statement

语义动作：将循环语句写入抽象语法树文件。

描述：这表示程序可以执行循环操作，重复执行一段代码。

规则5：statement: print\_statement

语义动作：将打印语句写入抽象语法树文件。

描述：这表示程序可以输出信息到控制台。

Variable Declaration

规则1：variable\_declaration: VAR id\_list type

语义动作：将变量声明的标识符列表和类型写入抽象语法树文件。

描述：这表示程序可以声明多个同类型的变量。

规则2：variable\_declaration: VAR id\_list type ASSIGN expression

语义动作：将变量声明的标识符列表、类型和初始值写入抽象语法树文件。

描述：这表示程序可以声明变量并同时初始化。

Type

规则1：type: INT

语义动作：将整型写入抽象语法树文件，并推送类型到符号表。

描述：这表示程序可以处理整型数据。

规则2：type: DOUBLE

语义动作：将双精度浮点型写入抽象语法树文件，并推送类型到符号表。

描述：这表示程序可以处理双精度浮点型数据。

规则3：type: STRING

语义动作：将字符串类型写入抽象语法树文件，并推送类型到符号表。

描述：这表示程序可以处理字符串数据。

规则4：type: BOOL

语义动作：将布尔型写入抽象语法树文件，并推送类型到符号表。

描述：这表示程序可以处理布尔型数据。

Array Dimensions

规则1：array\_dimensions: LBRACKET NUMBER RBRACKET

语义动作：将数组维度写入抽象语法树文件。

描述：这表示程序可以定义数组的大小。

规则2：array\_dimensions: LBRACKET NUMBER RBRACKET array\_dimensions

语义动作：将多维数组维度写入抽象语法树文件。

描述：这表示程序可以定义多维数组。

Assignment

规则1：assignment: ID ASSIGN expression2

语义动作：将赋值操作的左侧标识符和右侧字符串表达式的值写入抽象语法树文件，并 更新符号表中的值。

描述：这表示程序可以将字符串表达式的值赋给变量。

规则2：assignment: ID ASSIGN expression

语义动作：将赋值操作的左侧标识符和右侧数值表达式的值写入抽象语法树文件，并更 新符号表中的值。

描述：这表示程序可以将数值表达式的值赋给变量。

If Statement

规则1：if\_statement: IF LPAREN condition RPAREN LBRACE statement\_list RBRACE

语义动作：将条件语句写入抽象语法树文件，并根据条件的真假决定执行哪个语句列表。

描述：这表示程序可以根据条件执行不同的代码块。

规则2：if\_statement: IF LPAREN condition RPAREN LBRACE statement\_list RBRACE ELSE LBRACE statement\_list RBRACE

语义动作：将条件语句及其else分支写入抽象语法树文件，并根据条件的真假决定执 行哪个语句列表。

描述：这表示程序可以处理条件语句及其else分支。

Loop Statement

规则1：loop\_statement: FOR ID COLON ID LBRACE statement\_list RBRACE

语义动作：将增强型for循环写入抽象语法树文件。

描述：这表示程序可以执行循环操作，其中循环变量的声明和迭代是一体的。

规则2：loop\_statement: WHILE LPAREN condition RPAREN LBRACE statement\_list RBRACE

语义动作：将while循环写入抽象语法树文件。

描述：这表示程序可以执行循环操作，只要条件为真。

Condition

规则1：condition: expression EQ expression

语义动作：将等值条件写入抽象语法树文件，并设置条件标志。

描述：这表示程序可以比较两个表达式是否相等。

规则2：condition: expression NE expression

语义动作：将不等条件写入抽象语法树文件，并设置条件标志。

描述：这表示程序可以比较两个表达式是否不相等。

规则3：condition: expression LT expression

语义动作：将小于条件写入抽象语法树文件，并设置条件标志。

描述：这表示程序可以比较一个表达式是否小于另一个表达式。

规则4：condition: expression LE expression

语义动作：将小于等于条件写入抽象语法树文件，并设置条件标志。

描述：这表示程序可以比较一个表达式是否小于等于另一个表达式。

规则5：condition: expression GT expression

语义动作：将大于条件写入抽象语法树文件，并设置条件标志。

描述：这表示程序可以比较一个表达式是否大于另一个表达式。

规则6：condition: expression GE expression

语义动作：将大于等于条件写入抽象语法树文件，并设置条件标志。

描述：这表示程序可以比较一个表达式是否大于等于另一个表达式。

Expression

规则1：expression: NUMBER

语义动作：将数值写入抽象语法树文件。

描述：这表示程序可以处理数值常量。

规则2：expression: ID

语义动作：将标识符及其值写入抽象语法树文件。

描述：这表示程序可以处理变量和其对应的值。

规则3：expression: LPAREN expression RPAREN

语义动作：将括号内的表达式写入抽象语法树文件。

描述：这表示程序可以处理括号内的表达式，确保正确的运算顺序。

规则4：expression: expression PLUS expression

语义动作：将加法表达式写入抽象语法树文件，并计算结果。

描述：这表示程序可以执行加法运算。

# **4.结果显示与解释评估**

使用下述代码进行测试：

//this is declaration

var x,a,y int

var str string

var a int

var i int

var aa int[5][6]

//this is assignment

str = "abcbd"

a = 10

x = 20 + b \* 2

//this is if

if(a == 10){

a = a+10

print(a)

print(a \* 10)

}else{

x = x/10

print(x)

print("i love" - " o")

}

print("i love compiler" - "o")

输出的运行结果如图所示：

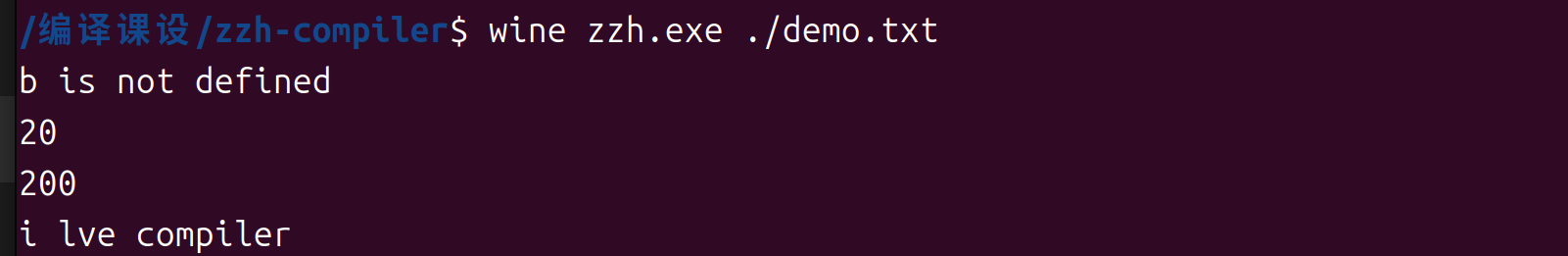


图 1 代码运行输出

由于b未定义，程序在编译时会报错，并且不会执行任何打印操作。因此，实际上这段代码的运行结果是编译错误，无法得到任何输出。

输出语句中的"i lve compiler"是由于字符串"i love compiler"减去"o"得到的，即移除了所有的"o"字符。

得到的token表结果如下：

表 4 识别到的token表

| Line | Token | Name |
| --- | --- | --- |
| 1 | no meaning | //this is declaration |
| 2 | Key Word | var |
| 2 | ID | x |
| 2 | Specail Word | , |
| 2 | ID | a |
| 2 | Specail Word | , |
| 2 | ID | y |
| 2 | Key Word | int |
| 3 | Key Word | var |
| 3 | ID | str |
| 3 | Key Word | string |
| 4 | Key Word | var |
| 4 | ID | a |
| 4 | Key Word | int |
| 5 | Key Word | var |
| 5 | ID | i |
| 5 | Key Word | int |
| 6 | Key Word | var |
| 6 | ID | aa |
| 6 | Key Word | int |
| 6 | Specail Word | [ |
| 6 | NUM | 5 |
| 6 | Specail Word | ] |
| 6 | Specail Word | [ |
| 6 | NUM | 6 |
| 6 | Specail Word | ] |
| 7 | no meaning | //this is assignment |
| 8 | ID | str |
| 8 | Operator | = |
| 8 | STR | abcbd |
| 9 | ID | a |
| 9 | Operator | = |
| 9 | NUM | 10 |
| 10 | ID | x |
| 10 | Operator | = |
| 10 | NUM | 20 |
| 10 | Operator | + |
| 10 | ID | b |
| 10 | Operator | \* |
| 10 | NUM | 2 |
| 11 | no meaning | //this is if |
| 12 | Key Word | if |
| 12 | Specail Word | ( |
| 12 | ID | a |
| 12 | Operator | == |
| 12 | NUM | 10 |
| 12 | Specail Word | ) |
| 12 | Specail Word | { |
| 13 | ID | a |
| 13 | Operator | = |
| 13 | ID | a |
| 13 | Operator | + |
| 13 | NUM | 10 |
| 14 | Key Word | print |
| 14 | Specail Word | ( |
| 14 | ID | a |
| 14 | Specail Word | ) |
| 15 | Key Word | print |
| 15 | Specail Word | ( |
| 15 | ID | a |
| 15 | Operator | \* |
| 15 | NUM | 10 |
| 15 | Specail Word | ) |
| 16 | Specail Word | } |
| 16 | Key Word | else |
| 16 | Specail Word | { |
| 17 | ID | x |
| 17 | Operator | = |
| 17 | ID | x |
| 17 | Operator | / |
| 17 | NUM | 10 |
| 18 | Key Word | print |
| 18 | Specail Word | ( |
| 18 | ID | x |
| 18 | Specail Word | ) |
| 19 | Key Word | print |
| 19 | Specail Word | ( |
| 19 | STR | i love |
| 19 | Operator | - |
| 19 | STR | o |
| 19 | Specail Word | ) |
| 20 | Specail Word | } |
| 21 | Key Word | print |
| 21 | Specail Word | ( |
| 21 | STR | i love compiler |
| 21 | Operator | - |
| 21 | STR | o |
| 21 | Specail Word | ) |

得到的单词表如下图所示：

表 5 识别到的单词表

| Name | Type | Value |
| --- | --- | --- |
| i | int | 0 |
| str | string | abcbd |
| y | int | 0 |
| a | int | 20 |
| x | int | 7680692 |

报错信息如下：

error: a is already declared at line: 4

b is not defined at line: 10

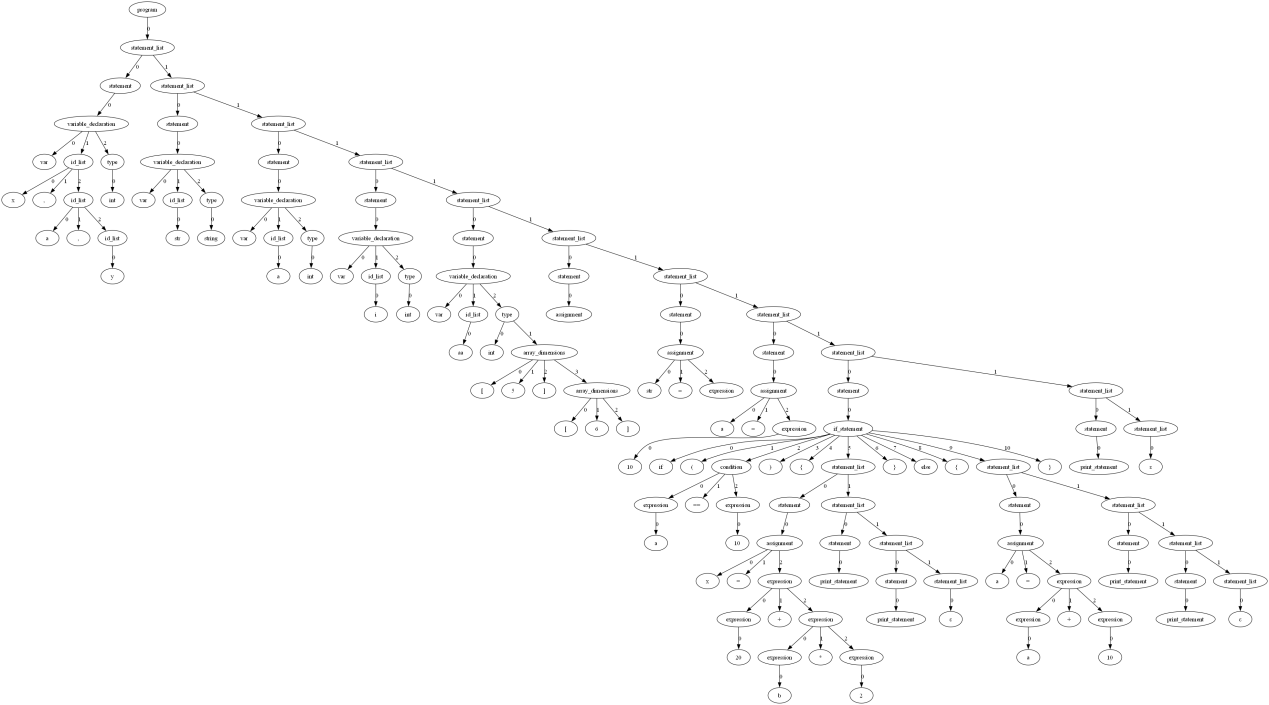


图 2 通过规约构建的语法树

# 小结

本文详细阐述了编译器中的词法分析、语法分析和语义分析三个核心阶段，并探讨了这些阶段中使用的关键技术和数据结构。在词法分析阶段，我们使用Flex工具将源代码分解成标记，并采用单词链结构来存储这些标记，同时提出了使用散列表来提高查找效率的可能性。语法分析阶段通过Yacc/Bison工具实现，构建解析器并生成抽象语法树（AST）。语义分析阶段则在语法分析的同时进行，通过嵌入语义动作来执行符号表更新、表达式计算等任务。此外，我们还讨论了语法制导翻译（SDT），它将语法分析和语义分析紧密结合，提高了编译过程的效率。通过这些技术的应用，我们确保了源代码在结构和语义上的正确性。

# **参考文献**

[1]John R. Levine..Flex & bison = Flex与bison/ John R. Levine..东南大学出版社,.2010.

[2](美)Alfred V. Aho...[等]著.编译原理:principles, techniques and tools.机械工业出版社.2009

# 致谢

在本学期编译原理课程与设计的学习旅程即将画上句点之际，我谨以此文向我们的编译原理老师——郑老师，表达我最深切的感激之情。

首先，我要感谢郑老师在课堂上的辛勤耕耘。您的讲解清晰透彻，将编译原理的复杂概念层层剖析，让我们能够逐步理解和掌握这门学科的核心知识。您不仅传授知识，更激发了我们对编译原理的兴趣和热情，让我们在学习中不断探索和发现新的知识领域。

其次，我要感谢郑老师在课程设计中的悉心指导。在项目实践中，您耐心地解答我们的疑惑，引导我们独立思考和解决问题。每当我们遇到难题时，您总是不厌其烦地提供帮助，让我们受益匪浅。您的专业指导和严谨态度，对我们的学术成长和职业发展都有着不可估量的影响。

最后，我们要感谢郑老师在个人成长方面的关心和鼓励。您不仅是我们的老师，更是我们的朋友和导师。您的鼓励和支持，让我们在学术道路上更加自信和坚定。

郑老师，您的教诲和影响将伴随我们一生。在此，我再次向您表示最诚挚的感谢。愿您工作顺利，身体健康，家庭幸福！

# 附录

## 1.词法分析部分代码 zzh.l

%{

#include "y.tab.h"

#include <string.h>

void yyerror(char \*msg);

extern int linecount;

extern int isStatement;

extern int isif;

extern int iselse;

extern int if\_flag;

extern int idNum;

extern FILE \*errorFile;

extern FILE \*tokenFile;

extern int isPushType;

int num = 0;

struct word {

char \*word\_name;

int word\_type;

union {

int int\_value;

double double\_value;

char \*string\_value;

} value;

struct word \*next;

};

extern struct word \*word\_list;

int add\_word(int type, char \*word);

int lookup\_word(char \*word);

%}

%%

[\n]+ { linecount++; }

[ \t\r]+ /\* 忽略空白字符 \*/

"var" { num = 0; isStatement = 1; fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Key Word",yytext); return VAR; }

"if" { isif=1; fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Key Word",yytext); return IF; }

"else" { if\_flag=!if\_flag; fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Key Word",yytext); return ELSE; }

"for" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Key Word",yytext); return FOR; }

"while" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Key Word",yytext); return WHILE; }

"int" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Key Word",yytext); return INT; }

"double" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Key Word",yytext); return DOUBLE; }

"string" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Key Word",yytext); return STRING; }

"bool" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Key Word",yytext); return BOOL; }

"print" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Key Word",yytext); return PRINT; }

":" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Specail Word",yytext); return COLON; }

"=" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Operator",yytext); return ASSIGN; }

"==" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Operator",yytext); return EQ; }

"!=" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Operator",yytext); return NE; }

"<=" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Operator",yytext); return LE; }

">=" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Operator",yytext); return GE; }

"<" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Operator",yytext); return LT; }

">" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Operator",yytext); return GT; }

"[" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Specail Word",yytext); return LBRACKET; }

"]" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Specail Word",yytext); return RBRACKET; }

"(" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Specail Word",yytext); return LPAREN; }

")" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Specail Word",yytext); return RPAREN; }

"{" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Specail Word",yytext); return LBRACE; }

"}" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Specail Word",yytext); return RBRACE; }

"," { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Specail Word",yytext); return COMMA; }

"+" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Operator",yytext); return PLUS; }

"-" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Operator",yytext); return MINUS; }

"\*" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Operator",yytext); return MULT; }

"/" { fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"Operator",yytext); return DIV; }

-?[0-9]+ { yylval.intVal = atoi(yytext); fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"NUM",yytext); return NUMBER; }

-?[0-9]+\.[0-9]+ { yylval.doubleVal = atof(yytext); fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"NUM",yytext); return DNUM; }

\"([^\"\\\n]|\\.)\*\" { // 匹配带引号的字符串

int len = strlen(yytext);

// 去掉两端的引号

yytext[len - 1] = '\0'; // 去掉结尾的引号

memmove(yytext, yytext + 1, len - 1); // 移动字符以去掉开头的引号

yylval.strVal = strdup(yytext); // 去掉引号后的字符串传递给语法分析器

fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"STR",yytext);

return STR;

}

[A-Za-z\_][A-Za-z0-9\_]\* {

if(isStatement==1){

num++;

idNum = num;

yylval.id = strdup(yytext);

if(add\_word(0, yytext) == 0){

//printf("error: %s is already declared\n", yytext);

fprintf(errorFile, "error: %s is already declared at line: %d\n", yytext,linecount);

isPushType = 0;

}

}else{

yylval.id = strdup(yytext);

}

fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"ID",yytext);

return ID;

}

\/\/[^\n]\* { /\* 忽略注释 \*/ fprintf(tokenFile, "%-15d%-15s%-15s\n",linecount,"no meaning",yytext); }

. {

//printf("%s: lexical error\n", yytext);

fprintf(errorFile, "%s: lexical error at line: %d\n", yytext,linecount);

}

%%

extern void \*malloc();

int add\_word(int type, char \*word){

struct word \*wp;

if(lookup\_word(word) != 0){

/\* 该变量已经声明 \*/

return 0;

}

/\* 添加变量 \*/

wp = (struct word \*)malloc(sizeof(struct word));

wp->next = word\_list;

wp->word\_name = (char\*)malloc(strlen(word)+1);

strcpy(wp->word\_name,word);

wp->word\_type = type;

word\_list = wp;

return 1;

}

/\* 搜索变量名是否存在 ！！！若查找效率低，可以使用散列表代替链表 \*/

int lookup\_word(char \*word){

struct word \*wp = word\_list;

while(wp!=NULL){

if(strcmp(wp->word\_name, word) == 0){

return 1;

}

wp = wp->next;

}

return 0;

}

## 2.语法分析部分代码 zzh.y

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/stat.h>

#include <sys/types.h>

#ifdef \_WIN32

#include <direct.h>

#include <io.h>

#define ACCESS \_access

#define MKDIR(a) \_mkdir((a))

#else

#include <unistd.h>

#define ACCESS access

#define MKDIR(a) mkdir((a), 0777) // 0777 是权限设置，表示所有用户可读写执行

#endif

#define TYPE\_NAME(x) \_Generic((x), \

int: 1, \

double: 2, \

char\*: 3 )

void yyerror(char \*msg);

int yylex();

extern FILE \*yyin;

int linecount = 1;

int isif=0;

int iselse=0;

int isStatement = 0;

int isPushType = 1;

int idNum = 0;

int if\_flag=1;

FILE \*treeFile;

FILE \*errorFile;

FILE \*tokenFile;

struct word {

char \*word\_name;

int word\_type;

union {

int int\_value;

double double\_value;

char \*string\_value;

} value;

struct word \*next;

};

struct word \*word\_list = NULL;

void pushType(int idNum,int type);

void free\_list(struct word \*head);

struct word \*isExist(char \*word);

char\* string\_add(const char\* str1, const char\* str2);

char\* string\_subtract(const char\* str1, const char\* str2);

void outputIDS();

void createOutputFolder();

%}

%union {

int intVal;

double doubleVal;

char\* strVal;

char\* id;

}

/\* 终结符 \*/

%token <id> ID

%token <intVal> NUMBER

%token <doubleVal> DNUM

%token <strVal> STR

%token VAR IF ELSE FOR WHILE INT DOUBLE STRING BOOL

%token PLUS MINUS MULT DIV

%token ASSIGN EQ NE LT GT LE GE COLON COMMA LBRACKET RBRACKET LPAREN RPAREN LBRACE RBRACE PRINT

/\* 优先级 \*/

%left EQ NE

%left LT LE GT GE

%left PLUS MINUS

%left MULT DIV

/\* 非终结符 \*/

%type <intVal> expression

%type <strVal> expression2

%type <id> id\_list type array\_dimensions

%type <strVal> statement\_list

%type <intVal> condition

%type <strVal> if\_statement

%type <strVal> print\_statement

%%

program: statement\_list{ fprintf(treeFile, "program --> statement\_list\n");}

;

statement\_list: statement statement\_list{ fprintf(treeFile, "statement\_list --> statement statement\_list\n");}

| /\* 空 \*/{ fprintf(treeFile, "statement\_list --> ε\n");}

;

statement: variable\_declaration{ fprintf(treeFile, "statement --> variable\_declaration\n");}

| assignment{ fprintf(treeFile, "statement --> assignment\n");}

| if\_statement{ fprintf(treeFile, "statement --> if\_statement\n");}

| loop\_statement{ fprintf(treeFile, "statement --> loop\_statement\n");}

| print\_statement{ fprintf(treeFile, "statement --> print\_statement\n"); }

;

variable\_declaration: VAR id\_list type{ fprintf(treeFile, "variable\_declaration --> var id\_list type\n");}

| VAR id\_list type ASSIGN expression{ fprintf(treeFile, "variable\_declaration --> var id\_list type = expression\n");}

;

id\_list: ID { isStatement = 0; fprintf(treeFile, "id\_list --> %s\n",$1);}

| ID COMMA id\_list { fprintf(treeFile, "id\_list --> %s , id\_list\n",$1);}

;

type: INT{

pushType(idNum,1);

fprintf(treeFile, "type --> int\n");

}

| DOUBLE{

pushType(idNum,2);

fprintf(treeFile, "type --> double\n");

}

| STRING{

pushType(idNum,3);

fprintf(treeFile, "type --> string\n");

}

| BOOL {

pushType(idNum,4);

fprintf(treeFile, "type --> bool\n");

}

| INT array\_dimensions{ fprintf(treeFile, "type --> int array\_dimensions\n");}

| DOUBLE array\_dimensions{ fprintf(treeFile, "type --> double array\_dimensions\n");}

;

array\_dimensions: LBRACKET NUMBER RBRACKET{ fprintf(treeFile, "array\_dimensions --> [ %d ]\n",$2);}

| LBRACKET NUMBER RBRACKET array\_dimensions{ fprintf(treeFile, "array\_dimensions --> [ %d ] array\_dimensions\n",$2);}

;

assignment: ID ASSIGN expression2{

if(!isif){

char \* name = $1;

struct word \*w = isExist(name);

int type = TYPE\_NAME($3);

if(w != NULL){

if(w->word\_type == type){

w->value.string\_value = (char\*)malloc(strlen($3) + 1); // +1 是为了 '\0'

if (w->value.string\_value == NULL) {

printf("%s's Memory allocation failed\n",w->word\_name);

fprintf(errorFile, "%s's Memory allocation failed at line: %d\n",w->word\_name,linecount);

}

strcpy(w->value.string\_value, $3);

//printf("%s:%s\n",$1,w->value.string\_value);

}else{

printf("error: %s's type mismatch\n",name);

fprintf(errorFile, "error: %s's type mismatch at line: %d\n",name,linecount);

}

}

fprintf(treeFile, "assignment --> %s = expression\n",$1);

}

else if(if\_flag!=0){

char \* name = $1;

struct word \*w = isExist(name);

int type = TYPE\_NAME($3);

if(w != NULL){

if(w->word\_type == type){

w->value.string\_value = (char\*)malloc(strlen($3) + 1); // +1 是为了 '\0'

if (w->value.string\_value == NULL) {

printf("%s's Memory allocation failed\n",w->word\_name);

fprintf(errorFile, "%s's Memory allocation failed at line: %d\n",w->word\_name,linecount);

}

strcpy(w->value.string\_value, $3);

//printf("%s:%s\n",$1,w->value.string\_value);

}else{

printf("error: %s's type mismatch\n",name);

}

}

fprintf(treeFile, "assignment --> %s = expression\n",$1);

}

}

| ID ASSIGN expression{

if(!isif){

char \* name = $1;

struct word \*w = isExist(name);

int type = TYPE\_NAME($3);

if(w != NULL){

if(w->word\_type == type){

w->value.int\_value = $3;

//printf("%s:%d\n",$1,w->value.int\_value);

}else{

printf("error: %s's type mismatch\n",name);

fprintf(errorFile, "error: %s's type mismatch at line: %d\n",name,linecount);

}

}

fprintf(treeFile, "assignment --> %s = expression\n",$1);

}

else if(if\_flag!=0){

//printf("if语句中的赋值:");

char \* name = $1;

struct word \*w = isExist(name);

int type = TYPE\_NAME($3);

if(w != NULL){

if(w->word\_type == type){

w->value.int\_value = $3;

//printf("%s:%d\n",$1,w->value.int\_value);

}else{

printf("error: %s's type mismatch\n",name);

fprintf(errorFile, "error: %s's type mismatch at line: %d\n",name,linecount);

}

}

fprintf(treeFile, "assignment --> %s = expression\n",$1);

}

}

;

if\_statement: IF LPAREN condition RPAREN LBRACE statement\_list RBRACE{

isif=0;

fprintf(treeFile, "if\_statement --> if ( condition ) { statement\_list }\n");

if ($3) { // 如果条件为真

if\_flag=1;

// printf("执行if\n");

$$=$6;

}

else{

// printf("if\_statement不执行\n");

$$="NULL";}

}

| IF LPAREN condition RPAREN LBRACE statement\_list RBRACE ELSE LBRACE statement\_list RBRACE

{ isif=0;

fprintf(treeFile, "if\_statement --> if ( condition ) { statement\_list } else { statement\_list }\n");

if ($3) { // 如果条件为真

if\_flag=1;

// printf("执行if\n");

$$=$6;

} else {

if\_flag=0;

// printf("执行else\n");

$$=$10;

}

}

;

loop\_statement: FOR ID COLON ID LBRACE statement\_list RBRACE /\* 增强型 for 循环 \*/{ fprintf(treeFile, "loop\_statement --> for %s : %s { statement\_list }\n", $2, $4);}

| WHILE LPAREN condition RPAREN LBRACE statement\_list RBRACE{ fprintf(treeFile, "loop\_statement --> while ( condition ) { statement\_list }\n");}

;

condition: expression EQ expression { fprintf(treeFile, "condition --> expression == expression\n");if\_flag=($1 == $3);$$ = ($1 == $3);}

| expression NE expression { fprintf(treeFile, "condition --> expression != expression\n");if\_flag=($1 != $3);$$ = ($1 != $3); }

| expression LT expression { fprintf(treeFile, "condition --> expression < expression\n");if\_flag=($1 < $3);$$ = ($1 < $3); }

| expression LE expression { fprintf(treeFile, "condition --> expression <= expression\n");if\_flag=($1 <= $3);$$ = ($1 <= $3);}

| expression GT expression { fprintf(treeFile, "condition --> expression > expression\n"); if\_flag=($1 > $3);$$ = ($1 > $3);}

| expression GE expression { fprintf(treeFile, "condition --> expression >= expression\n");if\_flag=($1 >= $3);$$ = ($1 >= $3); }

;

expression: NUMBER { fprintf(treeFile, "expression --> %d\n", $1); $$ = $1; }

| ID {

fprintf(treeFile, "expression --> %s\n", $1);

struct word \*w = isExist($1);

if(w != NULL){

if(w->word\_type == 1){

$$ = w->value.int\_value;

}else{

printf("error: %s's type mismatch\n",$1);

fprintf(errorFile, "error: %s's type mismatch at line: %d\n",$1,linecount);

}

}else{

printf("%s is not defined\n",$1);

fprintf(errorFile, "%s is not defined at line: %d\n",$1,linecount);

}

}

| LPAREN expression RPAREN { fprintf(treeFile, "expression --> ( expression )\n"); $$ = $2; }

| expression PLUS expression { fprintf(treeFile, "expression --> expression + expression\n"); $$ = $1 + $3;}

| expression MINUS expression { fprintf(treeFile, "expression --> expression - expression\n"); $$ = $1 - $3; }

| expression MULT expression { fprintf(treeFile, "expression --> expression \* expression\n"); $$ = $1 \* $3; }

| expression DIV expression { fprintf(treeFile, "expression --> expression / expression\n"); $$ = $1 / $3; }

;

expression2: STR { fprintf(treeFile, "expression2 --> %s\n", $1); strcpy($$, $1); }

| LPAREN expression2 RPAREN { fprintf(treeFile, "expression2 --> ( expression2 )\n"); strcpy($$, $2); }

| expression2 PLUS expression2 { fprintf(treeFile, "expression2 --> expression2 + expression2\n"); $$ = string\_add($1, $3); } // 拼接字符串

| expression2 MINUS expression2 { fprintf(treeFile, "expression2 --> expression2 - expression2\n"); $$ = string\_subtract($1, $3); } // 减去字符串

;

print\_statement:PRINT LPAREN STR RPAREN

// printf("isif=%d isflag=%d ",isif,if\_flag);

{

if(isif==0)

printf("%s\n", $3); // 打印字符串内容;

else

{

if(if\_flag!=0&&isif==1){

// printf("if语句中的输出:");

printf("%s\n", $3); // 打印字符串内容

}

}

}

| PRINT LPAREN expression RPAREN

{

if(isif==0)

printf("%d\n", $3); // 打印字符串内容;

else

{ if(if\_flag!=0)

{ printf("%d\n", $3); // 打印表达式的值

}

}

}

| PRINT LPAREN expression2 RPAREN

{

{

if(isif==0)

printf("%s\n", $3); // 打印字符串内容;

else

{

if(if\_flag!=0&&isif==1){

printf("%s\n", $3); // 打印字符串内容

}

}

}

}

;

%%

char\* string\_add(const char\* str1, const char\* str2) {

size\_t len1 = strlen(str1);

size\_t len2 = strlen(str2);

char\* result = (char\*)malloc(len1 + len2 + 1); // 分配内存

strcpy(result, str1); // 复制第一个字符串

strcat(result, str2); // 拼接第二个字符串

return result;

}

// 字符串减法函数

char\* string\_subtract(const char\* str1, const char\* str2) {

// 查找str2在str1中的位置

char\* pos = strstr(str1, str2);

if (!pos) {

// 如果没有找到匹配的部分，直接返回原字符串

return strdup(str1); // strdup会复制并返回一个新的字符串

}

// 计算删除str2后，剩余部分的长度

size\_t len1 = strlen(str1);

size\_t len2 = strlen(str2);

size\_t result\_len = len1 - len2;

// 为结果分配内存，包含终止符

char\* result = (char\*)malloc(result\_len + 1);

if (!result) {

// 内存分配失败时处理

fprintf(stderr, "Memory allocation failed\n");

exit(1);

}

// 将str1中str2之前的部分复制到result

size\_t prefix\_len = pos - str1;

strncpy(result, str1, prefix\_len);

// 将str2之后的部分复制到result

strcpy(result + prefix\_len, pos + len2);

return result;

}

void yyerror(char \*msg) {

//printf("msg = %s at line %d\n",msg,linecount);

fprintf(stderr, "Error: %s at line %d\n", msg, linecount);

fprintf(errorFile, "Error: %s at line %d\n", msg, linecount);

}

int yywrap() {

return 1;

}

void pushType(int idNum,int type){

if(isPushType == 1){

struct word \*current = word\_list;

for(int i=0;i<idNum;i++){

if(current == NULL){

printf("error: IDList is too large\n");

fprintf(errorFile, "error: IDList is too large at line: %d\n",linecount);

return;

}

current->word\_type = type;

switch(type){

case 1:

current->value.int\_value = 0;

break;

case 2:

current->value.double\_value = 0;

break;

case 3:

current->value.string\_value = NULL;

break;

case 4:

current->value.int\_value = 1;

break;

}

//printf("%d: id = %s , type = %d\n",linecount,current->word\_name,current->word\_type);

current = current->next;

}

}else{

isPushType = 1;

}

}

void free\_list(struct word \*head) {

struct word \*current = head;

struct word \*next;

while (current != NULL) {

next = current->next; // 保存下一个节点的指针

// 如果节点中有动态分配的字段（如字符串），先释放它

if (current->word\_name != NULL) {

free(current->word\_name);

}

// 释放当前节点

free(current);

// 移动到下一个节点

current = next;

}

}

struct word \* isExist(char \*word){

struct word \*wp = word\_list;

while(wp!=NULL){

if(strcmp(wp->word\_name, word) == 0){

return wp;

}

wp = wp->next;

}

return NULL;

}

void outputIDS(){

FILE\* file = fopen("./output/IDFile.txt", "w");

if (file == NULL) {

fprintf(stderr, "failed to create IDFile.txt\n");

printf("failed to create IDFile.txt\n");

exit(1);

}else{

char \*name = "name";

char \*type = "type";

char \*value = "value";

fprintf(file, "%-15s %-15s %-15s\n", name, type, value);

struct word \*wp = word\_list;

while(wp!=NULL){

int type = wp->word\_type;

switch(type){

case 1:

fprintf(file, "%-15s %-15s %-15d\n", wp->word\_name, "int", wp->value.int\_value);

break;

case 2:

fprintf(file, "%-15s %-15s %-15f\n", wp->word\_name, "double", wp->value.double\_value);

break;

case 3:

if(wp->value.string\_value == NULL){

fprintf(file, "%-15s %-15s %-15s\n", wp->word\_name, "string", "NULL");

}else{

fprintf(file, "%-15s %-15s %-15s\n", wp->word\_name, "string", wp->value.string\_value);

}

break;

case 4:

if(wp->value.int\_value == 1){

fprintf(file, "%-15s %-15s %-15s\n", wp->word\_name, "bool", "true");

}else{

fprintf(file, "%-15s %-15s %-15s\n", wp->word\_name, "bool", "false");

}

break;

}

wp = wp->next;

}

}

fclose(file);

}

void createOutputFolder(){

const char \*folder = "output";

// 检查文件夹是否存在

if (ACCESS(folder, 0) != 0) {

// 文件夹不存在，创建文件夹

if (MKDIR(folder) != 0) {

fprintf(stderr, "failed to create Folder output\n");

printf("failed to create Folder output\n");

exit(1);

}

}

}

int main(int argc, char \*\*argv) {

createOutputFolder();

treeFile = fopen("./output/tree.txt", "w");

errorFile = fopen("./output/error.txt", "w");

tokenFile = fopen("./output/tokenFile.txt", "w");

fprintf(tokenFile , "%-15s%-15s%-15s\n", "line", "token", "name");

if (!treeFile) {

fprintf(stderr, "Could not open tree.txt\n");

printf("Could not open tree.txt\n");

return 1;

}

if (!errorFile) {

fprintf(stderr, "Could not open error.txt\n");

printf("Could not open error.txt\n");

return 1;

}

if (!tokenFile) {

fprintf(stderr, "Could not open tokenFile.txt\n");

printf("Could not open tokenFile.txt\n");

return 1;

}

if (argc > 1) {

FILE \*file = fopen(argv[1], "r");

if (!file) {

fprintf(stderr, "Could not open %s\n", argv[1]);

fclose(treeFile);

fclose(errorFile);

fclose(tokenFile);

fclose(file);

return 1;

}

yyin = file;

}

yyparse();

outputIDS();

free\_list(word\_list);

fclose(treeFile);

fclose(errorFile);

fclose(tokenFile);

return 0;

}

## 3.可视化语法树构建

import matplotlib.pyplot as plt

from graphviz import Digraph

import os

# 定义一个空字典，后续可能用于存储某种数据结构（从代码整体看可能和构建树结构相关的数据）

d1 = {}

# 初始化一个字符串变量，用于在构建树结构等操作中作为节点的标识等用途，初始值为"0"

root = "0"

def run():

def plot\_model(tree, name):

"""

函数功能：根据给定的树结构数据和名称，构建并展示一个可视化的图形（可能是语法树之类的树状结构图形）。

参数：

- tree：表示树结构的数据，通常是嵌套的字典形式，用于描述节点和节点之间的关系等信息。

- name：指定生成图形的文件名（不含后缀），后续用于保存和展示图形时的命名。

返回值：无，但会生成相应的图形并保存到指定路径（通过调用g.render()）。

"""

g = Digraph("G", filename=name, format='png', strict=False)

try:

# 尝试获取树结构中根节点（字典的第一个键对应的节点，这里假设树结构以字典形式表示）的标签

first\_label = list(tree.keys())[0]

except:

# 如果出现异常（比如树为空等情况），则直接返回，不进行后续图形构建操作

return

g.node("0", first\_label)

\_sub\_plot(g, tree, "0")

g.render(engine='dot', view=False, cleanup=True, outfile=os.path.join('./output/', name + '.png'))

def \_sub\_plot(g, tree, inc):

"""

函数功能：递归地构建图形的子节点和边，用于完善整个树状图形的结构。

参数：

- g：Digraph对象，代表正在构建的图形。

- tree：表示树结构的数据，和plot\_model函数中的tree类似，用于描述节点关系等信息。

- inc：代表当前节点的标识（字符串形式），用于构建边时确定起始节点。

返回值：无，但会在递归调用过程中不断向图形对象g中添加节点和边，完善图形结构。

"""

global root

# 获取树结构中当前节点（由传入的tree参数决定）对应的第一个子节点的标签

first\_label = list(tree.keys())[0]

ts = tree[first\_label]

for i in ts.keys():

if isinstance(tree[first\_label][i], dict):

root = str(int(root) + 1)

sub\_node\_label = list(tree[first\_label][i].keys())[0]

# 处理子节点标签为空串（用'\_EMPTY\_'标记）或者是'ε'表示空串的情况，统一设置显示为'ε'（可根据实际需求调整显示样式）

if sub\_node\_label == '\_EMPTY\_' or sub\_node\_label == 'ε':

sub\_node\_label = 'ε'

g.node(root, sub\_node\_label)

g.edge(inc, root, str(i))

\_sub\_plot(g, tree[first\_label][i], root)

else:

root = str(int(root) + 1)

leaf\_label = tree[first\_label][i]

# 同样处理叶节点标签为空串或'ε'的情况

if leaf\_label == '\_EMPTY\_' or leaf\_label == 'ε':

leaf\_label = 'ε'

g.node(root, leaf\_label)

g.edge(inc, root, str(i))

with open('./output/tree.txt', 'r', encoding='utf-8') as f:

info = f.read().splitlines()

while True:

try:

info.remove('')

except:

break

n = len(info) - 1

try:

parts = info[n].split(' --> ')

if len(parts) == 2:

x, b = parts

else:

# 如果分割后不是两部分（可能有空串情况），做相应处理，这里假设用特殊符号'\_EMPTY\_'表示空串情况

x = parts[0] if parts else '000'

b = '\_EMPTY\_'

except:

x = '000'

b = '\_EMPTY\_'

def have(idx, node):

"""

函数功能：判断在给定索引范围（从idx - 1往前到0）内，是否存在节点与传入的node相等，用于查找节点是否存在于前面的信息中。

参数：

- idx：起始索引，从这个索引往前查找。

- node：要查找的节点名称（字符串形式）。

返回值：如果找到匹配的节点则返回True，否则返回False。

"""

for i in range(idx - 1, -1, -1):

tt = info[i]

try:

a, b = tt.split(' --> ')

except:

continue

if a == node:

return True

return False

vis = [0 for i in range(len(info))]

def dfs(idx, node):

"""

函数功能：深度优先搜索函数，用于从给定索引和节点开始，递归地构建树结构相关的数据（以字典形式表示）。

参数：

- idx：当前搜索的索引位置，用于在info列表中定位信息。

- node：当前搜索的节点名称（字符串形式）。

返回值：根据搜索情况返回构建好的树结构部分数据（字典形式），或者继续递归调用返回其他相关结果。

"""

if idx == -1:

return node

if vis[idx]:

return dfs(idx - 1, node)

try:

a, b = info[idx].split(' --> ')

except:

return dfs(idx - 1, node)

di = {}

if a == node:

vis[idx] = 1

bb = b.split()

for i in range(len(bb) - 1, -1, -1):

di[i] = dfs(idx - 1, bb[i])

di = dict(

sorted(di.items(), key=lambda d: d[0], reverse=False))

return {node: di}

else:

return dfs(idx - 1, node)

d1 = dfs(n, x)

# 调用plot\_model函数，传入构建好的树结构数据d1和名称'grammar\_tree'，用于生成并保存对应的图形

plot\_model(d1, 'grammar\_tree')

if \_\_name\_\_ == '\_\_main\_\_':

run()

print("successfully save grammar\_tree.png")