



同濟大學  
TONGJI UNIVERSITY

## 同济大学编译原理课程 中间代码生成器设计与实现实验报告

组    长	<u>2154312 郑博远</u>
组    员	<u>2152496 郭桢齐</u>
组    员	<u>2154318 蔡一锴</u>
专    业	<u>计算机科学与技术</u>
授课老师	<u>丁志军</u>

---

## 1. 实验概述

### 1.1. 实验目的

本次实验的目的是在完成词法分析和语法分析的基础上，实现中间代码生成器的设计与实验。通过该实验，我们将深入理解编译器在词法、语法分析的基础上进行中间代码生成的过程，通过实践掌握对四元式这一形式的中间代码的使用，以更便捷地表示源代码的语义结构。此外，我们还将重点关注对静态语义错误的诊断处理。通过本次实验，把理论知识应用到实际的编译任务中，深化对编程语言语法和语义的理解，以及加深对编译原理课程内容的掌握。

### 1.2. 主要任务

**基本功能：**在词法分析和语法分析的基础上给出源程序的语义分析结果，实现中间代码生成（建议以四元式的形式作为中间代码）；注意静态语义错误的诊断和处理；在此基础上，考虑更为通行的高级语言的语义检查和中间代码生成所需要注意的内容，并给出解决方案。

**扩展功能：**不限。

### 1.3. 需求分析

#### 1.3.1. 程序输入

- 选择一个类 C 的源程序文件输入；
  - 进行词法分析，输出词法分析的结果；
  - 从文件读入文法基本信息，即文法的产生式集合等（文法的起始符号由产生式集合中第一个产生式的左边的文法符号），产生 LR（1）分析表 action 表和 goto 表；
  - 根据之前词法分析的结果，测试该源程序文件是不是一个符合既定文法
-

的源程序。若符合既定文法则进行移进规约过程和显示最终的语法树，若不符合既定文法则给出错误提示。

· 在移进规约过程中，针对该程序做语义分析，并同步进行中间代码生成，若存在语义 **warning** 则会进行可视化展示，若存在语义 **error**，则会在进行可视化展示的同时终止中间代码的生成。

程序输入范例如下：

```
/*
    这是一个简单的乘法函数，接受一个整数和一个浮点数，
    返回它们的乘积。
*/
float multiplyNumbers(int a, float b) {
    float result = 0.0; // 初始化结果为 0.0

    // 使用 while 循环进行乘法运算
    while (a > 0) {
        result += b; // 将浮点数累加到结果中
        a -= 1;      // 递减整数 a，控制循环次数
    }

    return result; // 返回最终乘积
}
```

## 1.3.2. 文法输入

相比于上次实验的文法设计，我们修改了 **if**、**while** 与函数调用等部分的内容，并添加了 **epsilon** 空转移来进行简化，修改后的文法输入如下：

```
Program -> S DeclarationString
DeclarationString -> Declaration Declarations | Declaration
Declarations -> Declaration Declarations | Declaration
Declaration -> int identifier DeclarationType | float identifier
DeclarationType | void identifier P FunctionDeclaration
DeclarationType -> VarDeclaration | P FunctionDeclaration
VarDeclaration -> ;
```

```
FunctionDeclaration -> ( HeadVarStatements ) Block | ( ) Block |
( void ) Block
HeadVarStatements -> HeadVarStatements , VarStatement |
VarStatement
VarStatement -> int identifier | float identifier
Block -> { InnerStatement StatementString } | { InnerStatement }
| { StatementString }
InnerStatement -> VarStatement ; InnerVarStatements |
VarStatement ; | epsilon
InnerVarStatements -> VarStatement ; InnerVarStatements |
VarStatement ;
StatementString -> StatementString M Statement | Statement
Statement -> IfStatement | WhileStatement | ReturnStatement |
AssignStatement | FunctionCallStatement
AssignStatement -> identifier = RelopExpression ;
ReturnStatement -> return RelopExpression ; | return ;
WhileStatement -> while M ( RelopExpression A ) M Block
IfStatement -> if ( RelopExpression A ) M Block | if
( RelopExpression A ) M Block N else M Block
FactorExpression -> identifier | integer_constant |
floating_point_constant | ( RelopExpression ) | FactorExpression
/ FactorExpression | FactorExpression * FactorExpression |
FunctionCallExpression
AddExpression -> AddExpression + AddExpression | AddExpression -
AddExpression | FactorExpression
RelopExpression -> RelopExpression < RelopExpression |
RelopExpression > RelopExpression | RelopExpression ==
RelopExpression | RelopExpression <= RelopExpression |
RelopExpression >= RelopExpression | RelopExpression !=
RelopExpression | AddExpression
FunctionCallStatement -> FunctionCallExpression ;
FunctionCallExpression -> identifier ( Arguments ) | identifier
( )
Arguments -> Arguments , RelopExpression | RelopExpression
M -> epsilon
N -> epsilon
```

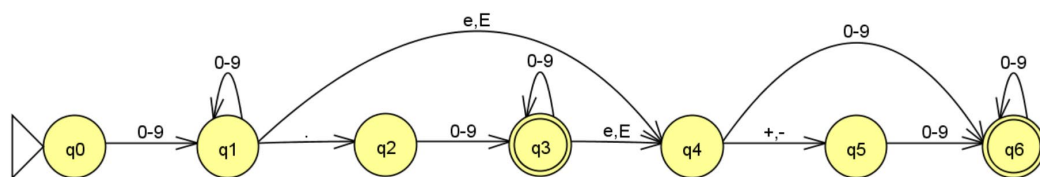
```
A -> epsilon
S -> epsilon
P -> epsilon
```

## 1.4. 亮点与扩展功能

### 1.4.1. 浮点数与类型转换

在上一次实验中，我们实现了对浮点数常量的识别以及对 float 类型变量的支持。支持的浮点数格式包括：普通形式，如 1.25, 34.2 等；或科学计数法形式，如 3e4, 5.4e-9, 8E12 等。

具体来说，通过如下的状态转移图，构造 DFA 以识别浮点数常量：



在本次实验中，我们在先前的基础上实现了对浮点数与整型数的类型转换。当进行加减、乘除运算时，若两个操作数分别为整型和浮点型，会对整型变量或常量进行类型转换；在进行赋值运算时，也会进行对应的类型转换，并在浮点型转整型时抛出可能丢失数据的 warning。例如：

```
int main() {
    int a;
    a = 1 + 3.2;
    return 0;
}
```

Warning at (3,14): 从 float 转换到 int, 可能丢失数据

地址	四元式
100	(j, _, _, 101)
101	(intofloat, 1, _, T0)
102	(float+, T0, 3.2, T1)
103	(intoint, T1, _, T2)
104	(=, T2, _, a)
105	(ret, 0, _, _)

## 1.4.2. 过程调用

在上次实验支持过程调用的基础上，我们参照了课本与课件中对于过程调用以及保留作用域信息部分的介绍，实现了对过程调用部分中间代码生成的支持，以及对应的静态语义错误检测。

**首先，介绍全局变量与局部变量的管理。**参照课件中保留作用域信息的部分，我们修改了如下文法，插入了一些  $\epsilon$  产生式：

```
Program -> S DeclarationString
DeclarationString -> Declaration Declarations | Declaration
Declarations -> Declaration Declarations | Declaration
Declaration -> int identifier DeclarationType | float
identifier DeclarationType | void identifier P
FunctionDeclaration
DeclarationType -> VarDeclaration | P FunctionDeclaration
VarDeclaration -> ;
S -> epsilon
P -> epsilon
```

对于每个过程，语义分析器会维护一个 **Process** 类，用一张 **words\_table** 记录其中的局部变量（文法不支持过程嵌套）；此外，还有一张全局 **words\_table** 记录全局变量。每个 **Process** 类的创建在规约产生式 “**P** -> epsilon” 时完成。这样，在此后的函数体语句的规约时，**Process** 类已经形成，可以修改或访问其中的 **words\_table**。

**其次，介绍过程调用与函数返回。**我们参照了课件中对于过程调用部分的四元式生成，设计了如下的文法：

```
FunctionCallExpression -> identifier ( Arguments ) | identifier ( )
Arguments -> Arguments , RelopExpression | RelopExpression
```

在 **Arguments** 规约过程中，会记录一个 **queue** 属性，从而记录下函数调用部分传入的所有表达式（及其类型）。在规约成 **FunctionCallExpression** 时，会逐一生成 “param” 四元式。与课件直接 “call 过程名” 不同的是，我们修改了

对函数调用“call”四元式的定义，改为了“(call, 被调用过程入口地址, \_, 返回值存储的临时变量 (void 则为空))”这一格式。对于函数返回，由于没有找到课件中的相关样例，我们自行定义了“(ret, 返回值 (void 则为空), \_, \_)”的四元式格式。具体样例如下：

<pre> int sum(int a, int b){     return a + b; } int main(){     int a;     a = sum(5, 7);     return 0; } </pre>	<table> <tr> <th>地址</th><th>四元式</th></tr> <tr> <td>100</td><td>(j, _, _, 103)</td></tr> <tr> <td>101</td><td>(int+, a, b, T0)</td></tr> <tr> <td>102</td><td>(ret, T0, _, _)</td></tr> <tr> <td>103</td><td>(param, 5, _, _)</td></tr> <tr> <td>104</td><td>(param, 7, _, _)</td></tr> <tr> <td>105</td><td>(call, 101, _, T1)</td></tr> <tr> <td>106</td><td>(=, T1, _, a)</td></tr> <tr> <td>107</td><td>(ret, 0, _, _)</td></tr> </table>	地址	四元式	100	(j, _, _, 103)	101	(int+, a, b, T0)	102	(ret, T0, _, _)	103	(param, 5, _, _)	104	(param, 7, _, _)	105	(call, 101, _, T1)	106	(=, T1, _, a)	107	(ret, 0, _, _)
地址	四元式																		
100	(j, _, _, 103)																		
101	(int+, a, b, T0)																		
102	(ret, T0, _, _)																		
103	(param, 5, _, _)																		
104	(param, 7, _, _)																		
105	(call, 101, _, T1)																		
106	(=, T1, _, a)																		
107	(ret, 0, _, _)																		

此外，在过程调用的部分还会检查调用与声明的变量类型、个数是否匹配，具体的错误分析将在之后的报告中详细介绍。

对于 main 函数的入口跳转，我们进行了如下处理：在 Program 程序初始的“S -> epsilon”规约时，在空的四元式表中插入一句“(j, \_, \_, \_)”无条件跳转语句。当规约到 main 函数的函数体时，再将 main 的起始地址回填到四元式表首的最后一个单元中，从而实现对 main 函数的跳转。

### 1.4.3. 关系表达式

由于本次实现的文法中，RelopExpression 可以通过“FactorExpression -> ( RelopExpression )”被规约为 FactorExpression，从而继续参与表达式计算；因此，对于关系表达式，不能采用记下真假出口的 truelist、falselist 再回填的方式（因为这样就没有临时变量存放表达式的值了）。因此，我们借鉴了课件中的“数值表示法”，即计算布尔表达式如同计算算术表达式一样一步步算。此时，在作为 if、while 语句的分支或循环条件时，则需要额外的两条跳转指令来完成跳转。但当归约到 IfStatement 或 WhileStatement 时，其内部的 block 已经产生了很多新的四元式，这两条跳转指令在此时插入则会位置错误。因

此，我们修改了文法如下：

```
WhileStatement -> while M ( RelopExpression A ) M Block
IfStatement -> if ( RelopExpression A ) M Block | if
( RelopExpression A ) M Block N else M Block
M -> epsilon
N -> epsilon
A -> epsilon
```

产生式“A -> epsilon”规约时，会插入 jnz、j 两条跳转四元式，判断条件与转移地址都待回填，并将这两条四元式的地址记录在 truelist、falselist 属性中。这样，当归约到 IfStatement 或 WhileStatement 时，就可以根据该地址进行转移条件与转移地址的回填。程序示例如下：

```
int main(){
    int a;

    a = 10;

    while(a > 0){
        a = a - 1;
    }

    return 0;
}
```

地址	四元式
100	(j, _, _, 101)
101	(=, 10, _, a)
102	(j>, a, 0, 105)
103	(=, 0, _, T0)
104	(j, _, _, 106)
105	(=, 1, _, T0)
106	(jnz, T0, _, 108)
107	(j, _, _, 111)
108	(int-, a, 1, T1)
109	(=, T1, _, a)
110	(j, _, _, 102)
111	(ret, 0, _, _)

## 2. 使用说明

### 2.1. 环境配置说明

以下脚本需要在代码目录的命令行中执行。

需要先行安装 pnpm, Python 及 pip。建议使用 virtual env。



## 2.1.1. 安装依赖

```
pnpm install
pip install -r requirements.txt
```

## 2.1.2. 运行

```
pnpm dev
```

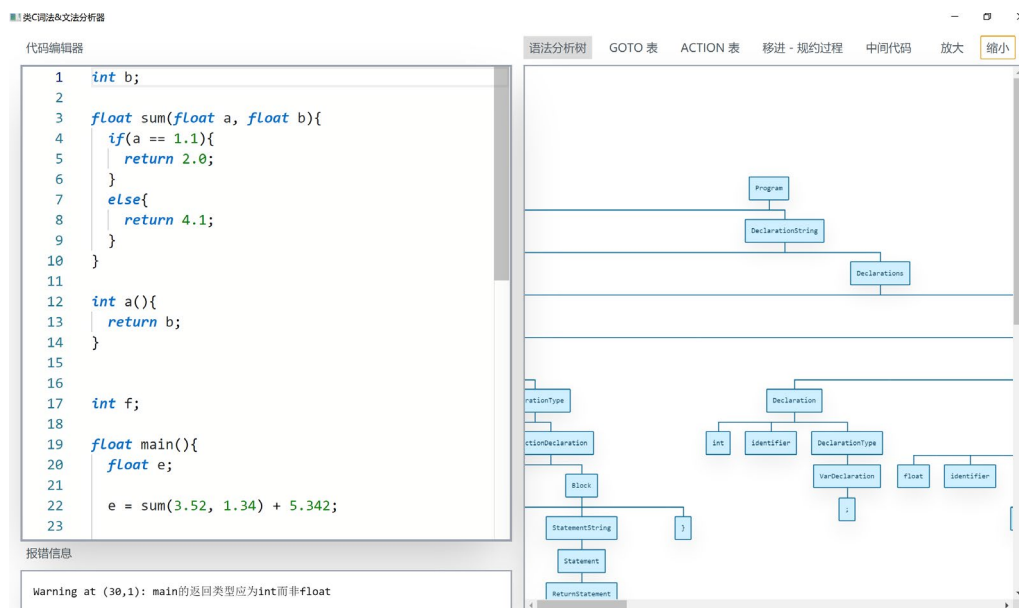
## 2.1.3. 打包

```
pnpm build
```

## 2.2. 整体设计

### 2.2.1. 初始界面

初始界面展示，左侧的代码是从与 `main.py` 同目录下的 `test.c` 文件中读取的，我们的分析器支持读取文件和左侧编辑器输入，可以看到左侧的代码是带有语法高亮功能的，我们将构建好的 `action` 表和 `goto` 表打包成静态文件，每次需要时进行读取，如果不存在则需要从头构建，这样大大提升了程序运行效率，初始界面如下所示，代码下方有报错位置和信息，右侧可以选择查看语法分析树、分析表、移进规约过程和中间代码。



## 2.2.2. 基础功能展示

通过下面的例子来介绍不包含错误的代码的中间代码生成方式。首先需要  
一个无条件跳转语句跳转到 `main` 函数，开始执行 `main` 函数的代码（对应本例  
115 地址的语句），变量 `ans` 类型为 `float` 但赋值为 `int` 值 0，因此需要做类型转  
换（对应本例的 116 地址的语句）使用一个临时变量 `T5` 来存储类型转换后的结  
果，并将其值赋给 `ans` 变量。

接着执行 `while` 语句。在每次循环中，如果布尔表达式 `n>0` 的值为真，就  
将临时变量 `T6` 的值赋为 1（对应本例的 121 地址的语句），否则赋为 0（对应  
本例的 119 地址的语句），然后根据 `T6` 的值判断下一条语句的位置，如果 `T6`  
非 0，则跳转到循环体内的赋值语句开始执行（对应本例的 124 地址的语  
句），执行块内内容后跳转到循环语句开始的位置（对应本例的 118 地址的语  
句），如果 `T6` 为 0，则跳转到循环语句的下一条语句的位置继续执行（对应本  
例的 132 地址的语句）。

在循环体内部存在函数调用语句，其首先使用 `param` 语句传递实参给函数  
体，然后跳转到函数体首地址开始执行（对应本例的 101 地址的语句），在执  
行完毕后使用 `ret` 语句将返回值传递给 `main` 函数，并跳转回 `main` 函数中下一条  
语句的位置继续执行。

类C文法中间代码生成器

代码编辑器

```

1  int a;
2  float b;
3  float sum(int a, float b){
4      float c;
5      c = a + b;
6      if(c > 2.1){
7          return c;
8      }
9      else{
10         return 0 - c;
11     }
12 }
13 int main(){
14     int n;
15     float ans;
16     n = 10;
17     ans = 0;
18     while (n > 0){
19         n = n - 1;
20         ans = ans + sum(n, 1.2);
21     }
22     return 0;
23 }
24

```

报错信息

0 errors, 0 warnings.

语法分析树 GOTO 表 ACTION 表

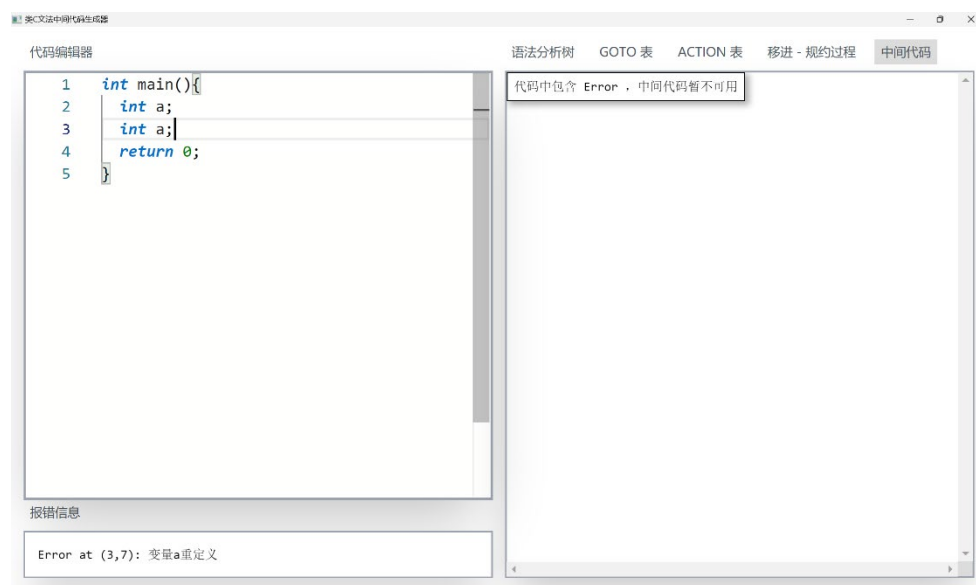
地址	四元式
100	(j, -, -, 115)
101	(intofloat, a, -, T0)
102	(float+, T0, b, T1)
103	(=, T1, -, c)
104	(j>, c, 2.1, 107)
105	(=, 0, -, T2)
106	(j, -, -, 108)
107	(=, 1, -, T2)
108	(jnz, T2, -, 110)
109	(j, -, -, 112)
110	(ret, c, -, -)
111	(j, -, -, 0)
112	(intofloat, 0, -, T3)
113	(float-, T3, c, T4)
114	(ret, T4, -, -)
115	(=, 10, -, n)
116	(intofloat, 0, -, T5)
117	(=, T5, -, ans)
118	(j>, n, 0, 121)
119	(=, 0, -, T6)
120	(j, -, -, 122)
121	(=, 1, -, T6)
122	(jnz, T6, -, 124)
123	(j, -, -, 132)
124	(int-, n, 1, T7)
125	(=, T7, -, n)
126	(param, n, -, -)
127	(param, 1.2, -, -)
128	(call, 101, -, T8)
129	(float+, ans, T8, T9)
130	(=, T9, -, ans)
131	(j, -, -, 118)
132	(ret, 0, -, -)

## 2.3. 错误处理

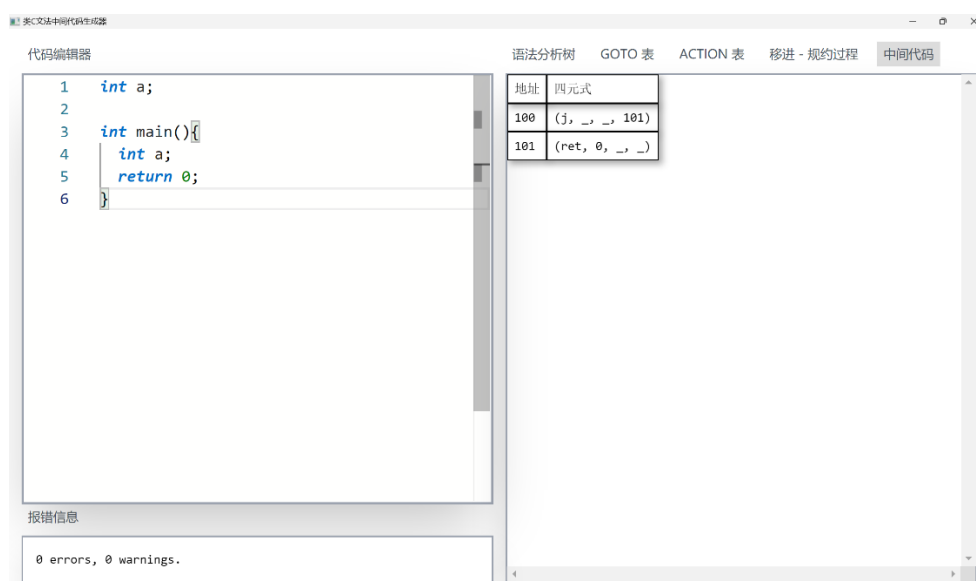
本次项目的重点之一就是对于静态语义分析中的错误检查。我们将错误处理分为 warning 和 error 两种报错形式。如果语义分析时遇到 error，则不继续生成代码四元式。接下来将按语义错误类型分四个部分做简单介绍。

### 2.3.1. 变量定义问题

在同一作用域内变量重定义，报错为 error，因此不生成中间代码。



如果同名变量中一个为全局变量，按低层屏蔽高层的原则处理，不报错。



全局变量可以在任意一个函数体内使用，不报错。

The screenshot shows the 'C语言中间代码生成器' (C Language Intermediate Code Generator) interface. The '代码编辑器' (Code Editor) contains the following C code:

```

1  int a;
2
3  int b;
4
5  int main(){
6      int a;
7
8      b = 3 + 2;
9
10     a = b + 4;
11
12     return 0;
13 }
    
```

The '报错信息' (Error Information) section at the bottom left shows: 0 errors, 0 warnings.

The '中间代码' (Intermediate Code) section on the right displays the generated quadruples in a table:

地址	四元式
100	(j, _, _, 101)
101	(int+, 3, 2, T0)
102	(=, T0, _, b)
103	(int+, b, 4, T1)
104	(=, T1, _, a)
105	(ret, 0, _, _)

未声明的标识符，报错为 error，不生成中间代码。

The screenshot shows the 'C语言中间代码生成器' (C Language Intermediate Code Generator) interface. The '代码编辑器' (Code Editor) contains the following C code:

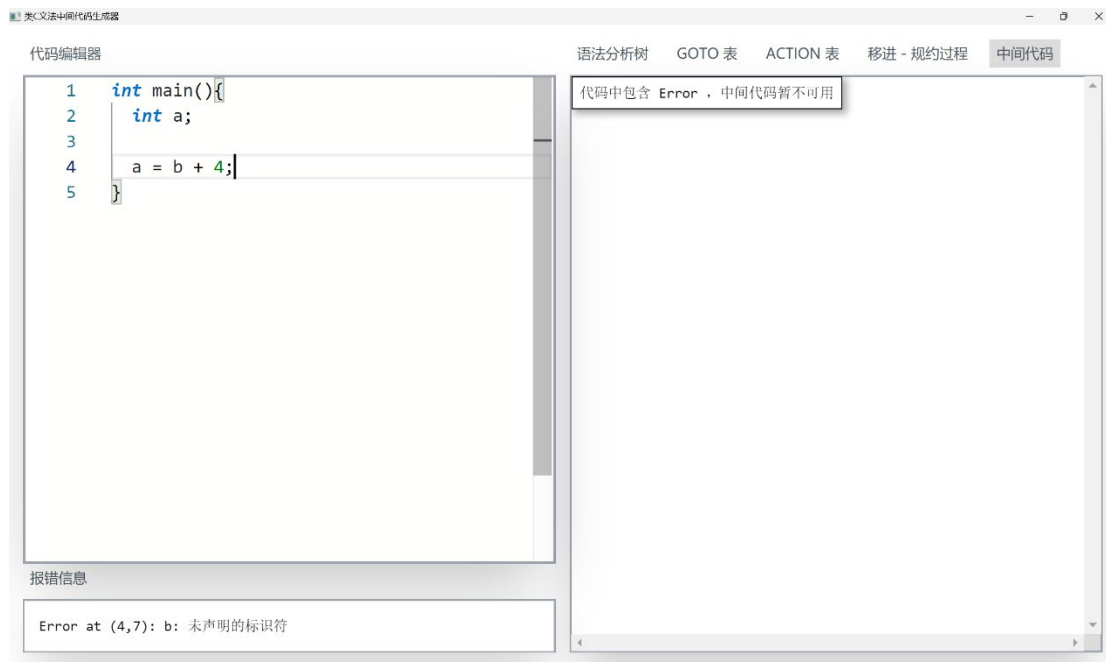
```

1  int main(){
2      int a;
3      b = 5;
4  }
    
```

The '报错信息' (Error Information) section at the bottom left shows: Error at (3,8): b: 未声明的标识符 (b: undeclared identifier).

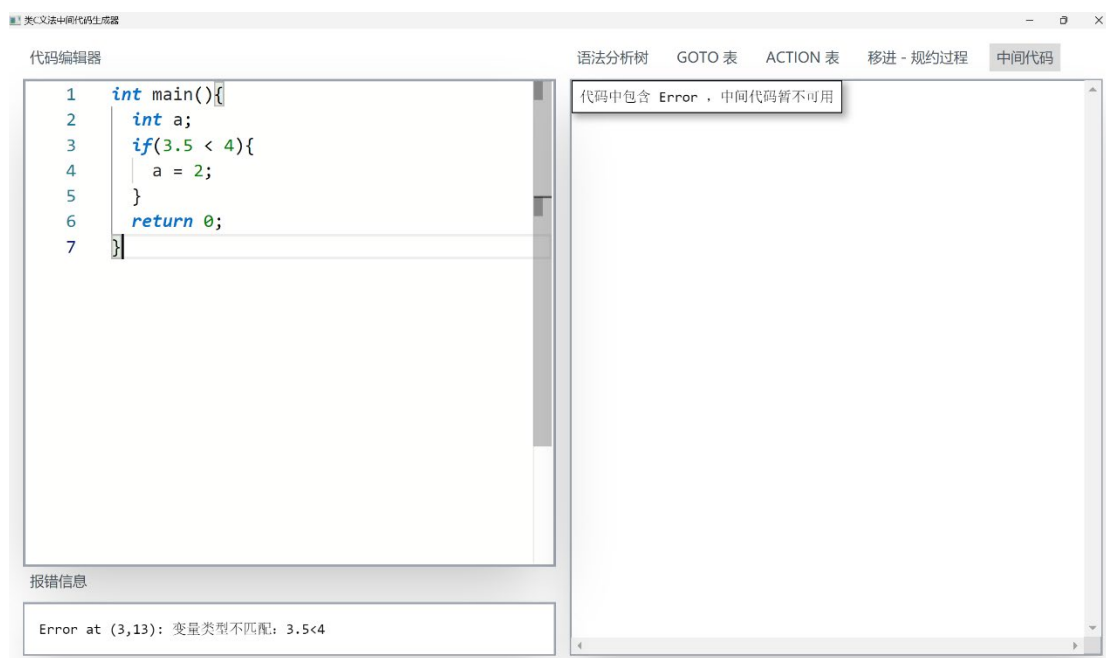
The '中间代码' (Intermediate Code) section on the right displays the message: 代码中包含 Error，中间代码暂不可用 (Code contains Error, intermediate code is temporarily unavailable).

同理，表达式情况的未声明标识符，报错为 **error**，不生成中间代码。

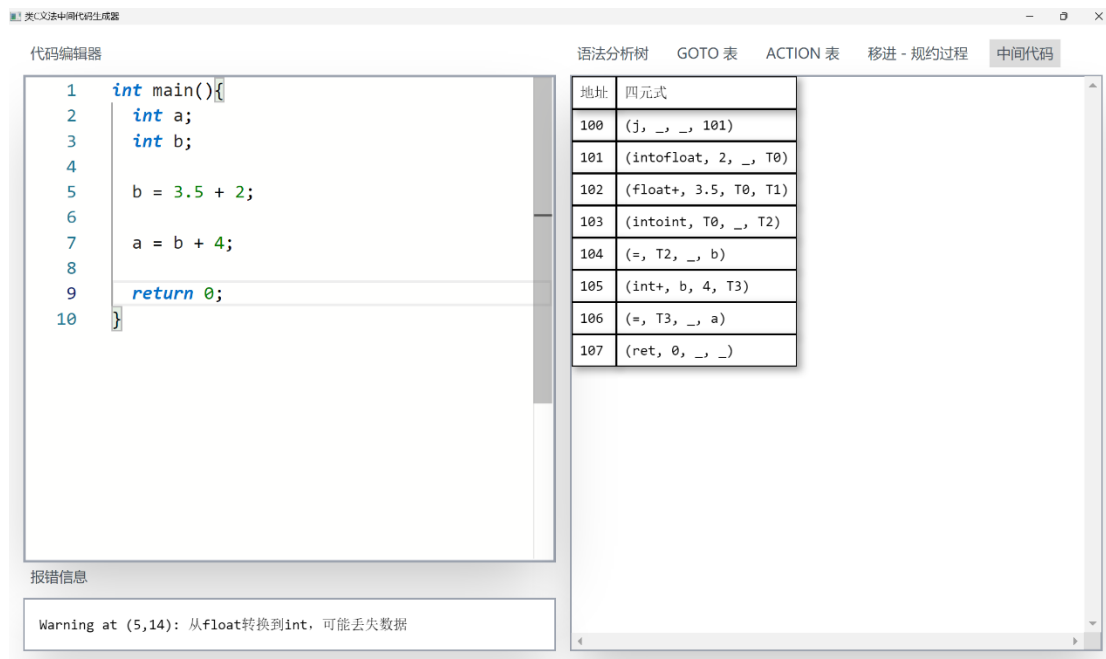


## 2.3.2. 变量类型问题

布尔表达式左右两侧类型不匹配，报错为 **error**，不生成中间代码。

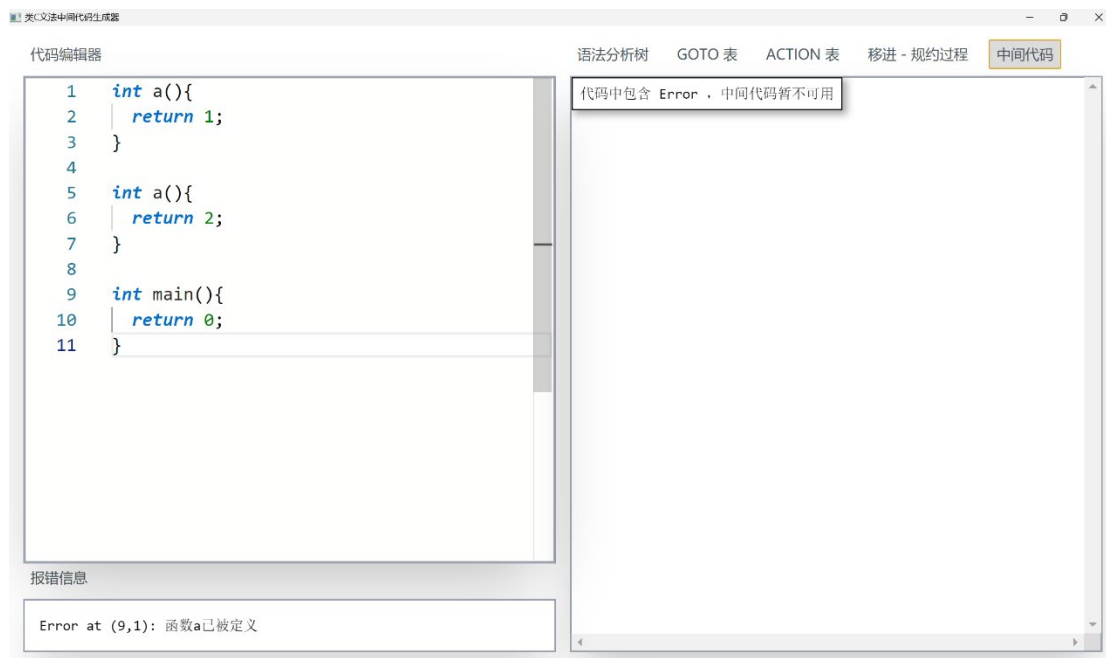


类型转换从 float 转为 int，报错为 **warning**，不影响中间代码生成。



### 2.3.3. 函数定义问题

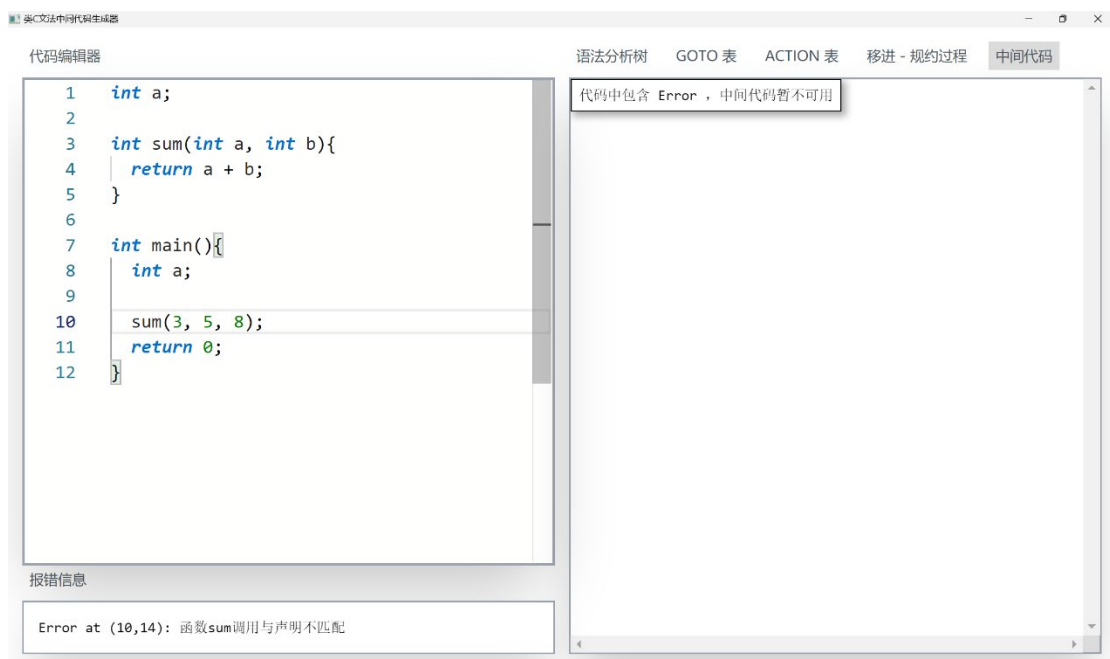
函数重定义问题，报错为 **error**，不生成中间代码。



函数未定义问题，找不到标识符，报错为 **error**，不生成中间代码。

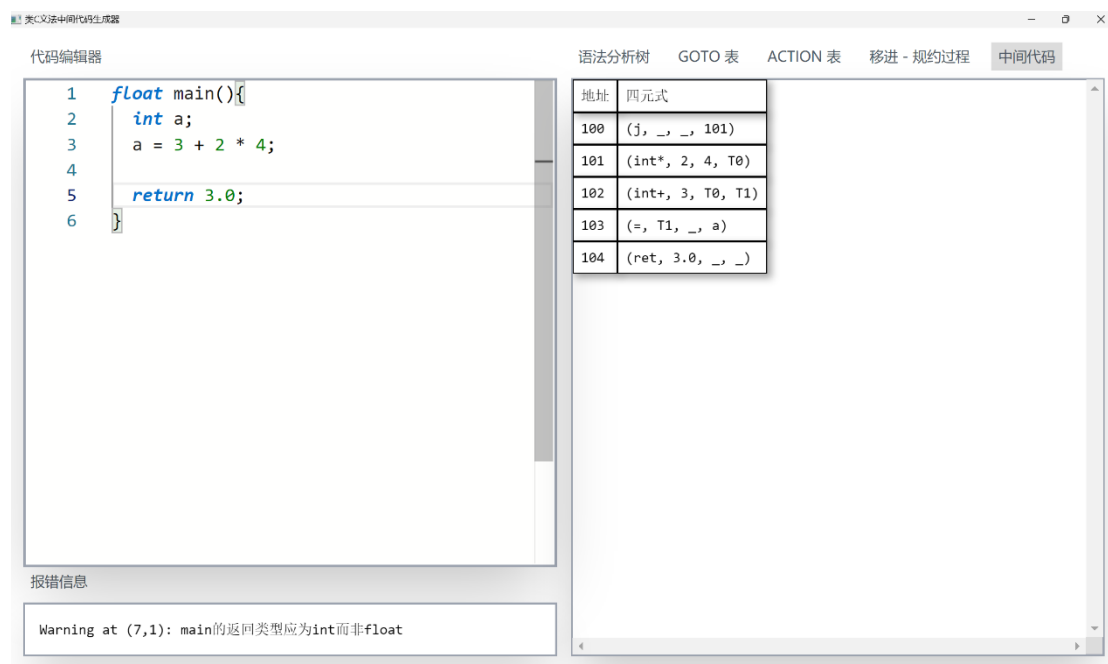


函数声明与调用不匹配问题，报错为 **error**，不生成中间代码。



## 2.3.4. 函数返回值问题

main 函数返回类型为 float，报错为 **warning**，不影响中间代码生成。



代码编辑器

```

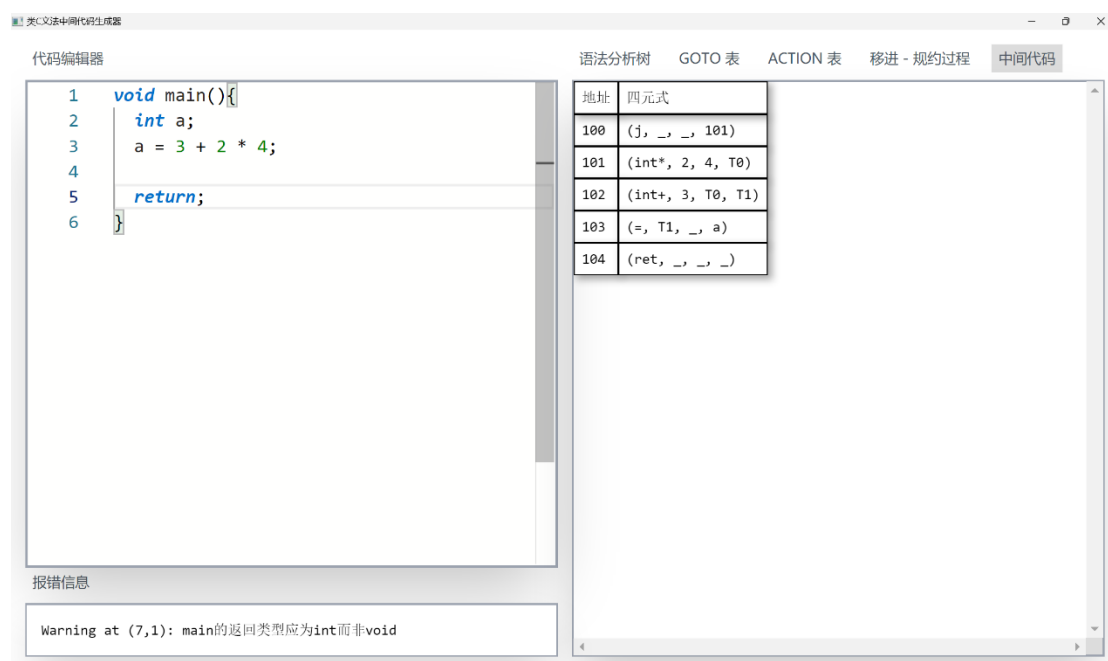
1 float main(){
2     int a;
3     a = 3 + 2 * 4;
4
5     return 3.0;
6 }
    
```

报错信息

Warning at (7,1): main的返回类型应为int而非float

地址	四元式
100	(j, _, _, 101)
101	(int*, 2, 4, T0)
102	(int+, 3, T0, T1)
103	(=, T1, _, a)
104	(ret, 3.0, _, _)

main 函数返回类型为 void，报错为 warning，不影响中间代码生成。



代码编辑器

```

1 void main(){
2     int a;
3     a = 3 + 2 * 4;
4
5     return;
6 }
    
```

报错信息

Warning at (7,1): main的返回类型应为int而非void

地址	四元式
100	(j, _, _, 101)
101	(int*, 2, 4, T0)
102	(int+, 3, T0, T1)
103	(=, T1, _, a)
104	(ret, _, _, _)

返回类型为 int/float 类型的函数没有返回值的情况，报错为 error，不生成中间代码。



The screenshot shows a code editor with the following C code:

```

1  int a;
2
3  int q(){
4
5  }
6
7  int main(){
8      int a;
9      return 0;
10 }
    
```

The error message at the bottom states: "Error at (7,1): 函数q必须返回一个值" (Error at (7,1): function q must return a value).

The parse tree on the right shows the structure of the code, with nodes for Program, DeclarationString, Declaration, and Identifier. The tree structure is as follows:

```

graph TD
    Program --> DeclarationString
    DeclarationString --> Declaration
    Declaration --> Identifier
    Identifier --> int
    
```

返回类型为 void 的函数有返回值的情况，报错为 error，不生成中间代码。

The screenshot shows a code editor with the following C code:

```

1  int a;
2
3  void q(){
4      return 5;
5  }
6
7  int main(){
8      int a;
9      return 0;
10 }
    
```

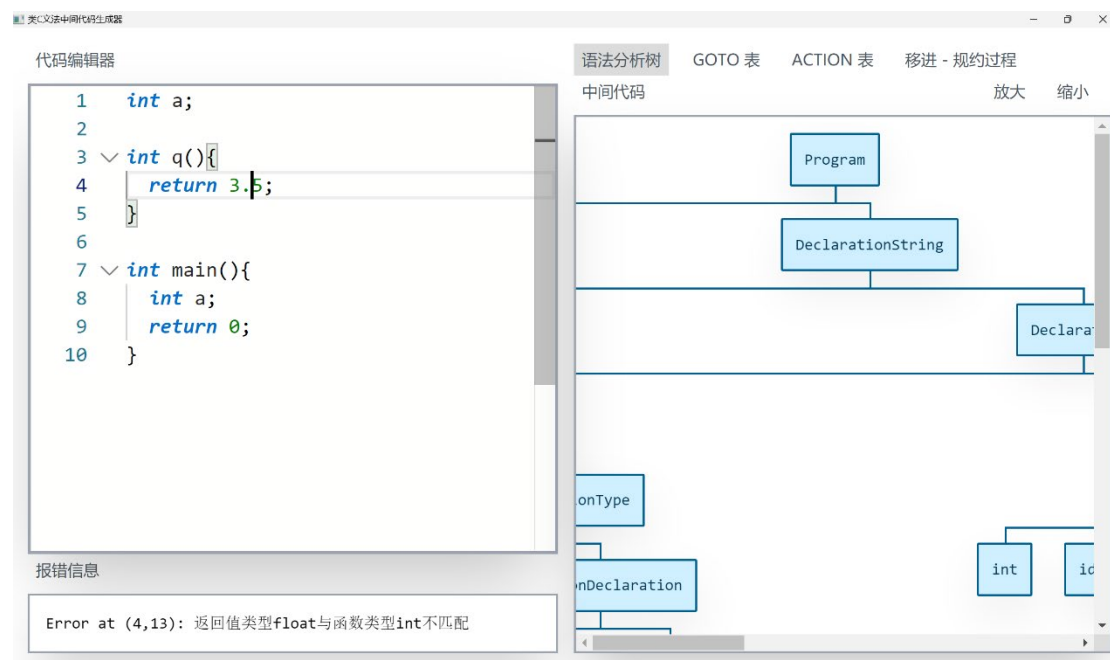
The error message at the bottom states: "Error at (4,11): 返回值类型int与函数类型void不匹配" (Error at (4,11): return value type int does not match function type void).

The parse tree on the right shows the structure of the code, with nodes for Program, DeclarationString, Declaration, and Identifier. The tree structure is as follows:

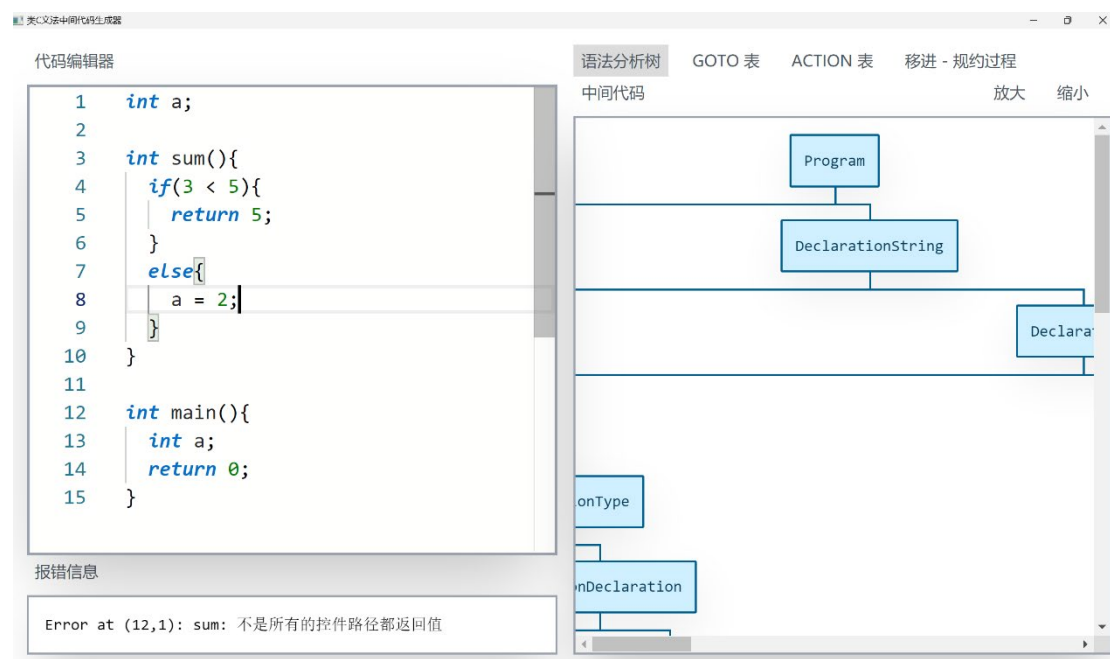
```

graph TD
    Program --> DeclarationString
    DeclarationString --> Declaration
    Declaration --> Identifier
    Identifier --> int
    
```

返回类型与返回值不匹配的情况，如本例返回类型是 int 但返回值是 float，报错为 error，不生成中间代码。



函数返回类型为 int/float，但存在 if 的某些分支没有返回值的情况，不是所有控件路径都返回值，报错为 error，不生成中间代码。



对于函数返回类型为 void 的处理方式，则在规约函数时 emit 一个 ret 语句（对应本例的 109 地址的语句），将所有 nextlist 跳转地址都回填到此处。

The screenshot shows the 'C语言中间代码生成器' (C Language Intermediate Code Generator) interface. The '代码编辑器' (Code Editor) contains the following C code:

```

1  int a;
2
3  void test(){
4      if(3 < 5){
5      }
6  } else{
7      a = 2;
8  }
9
10
11 int main(){
12     int a;
13     return 0;
14 }
    
```

The '报错信息' (Error Information) section shows: 0 errors, 0 warnings.

The '中间代码' (Intermediate Code) section displays the following table:

地址	四元式
100	(j, _, _, 110)
101	(j<, 3, 5, 104)
102	(=, 0, _, T0)
103	(j, _, _, 105)
104	(=, 1, _, T0)
105	(jnz, T0, _, 107)
106	(j, _, _, 108)
107	(j, _, _, 109)
108	(=, 2, _, a)
109	(ret, _, _, _)
110	(ret, 0, _, _)

## 2.3.5. 词法分析或语法分析出错

除上述语义分析报错外，如果代码不符合规定的文法规则或在词法分析阶段出错，同样也会报错为 **error**，不生成中间代码。

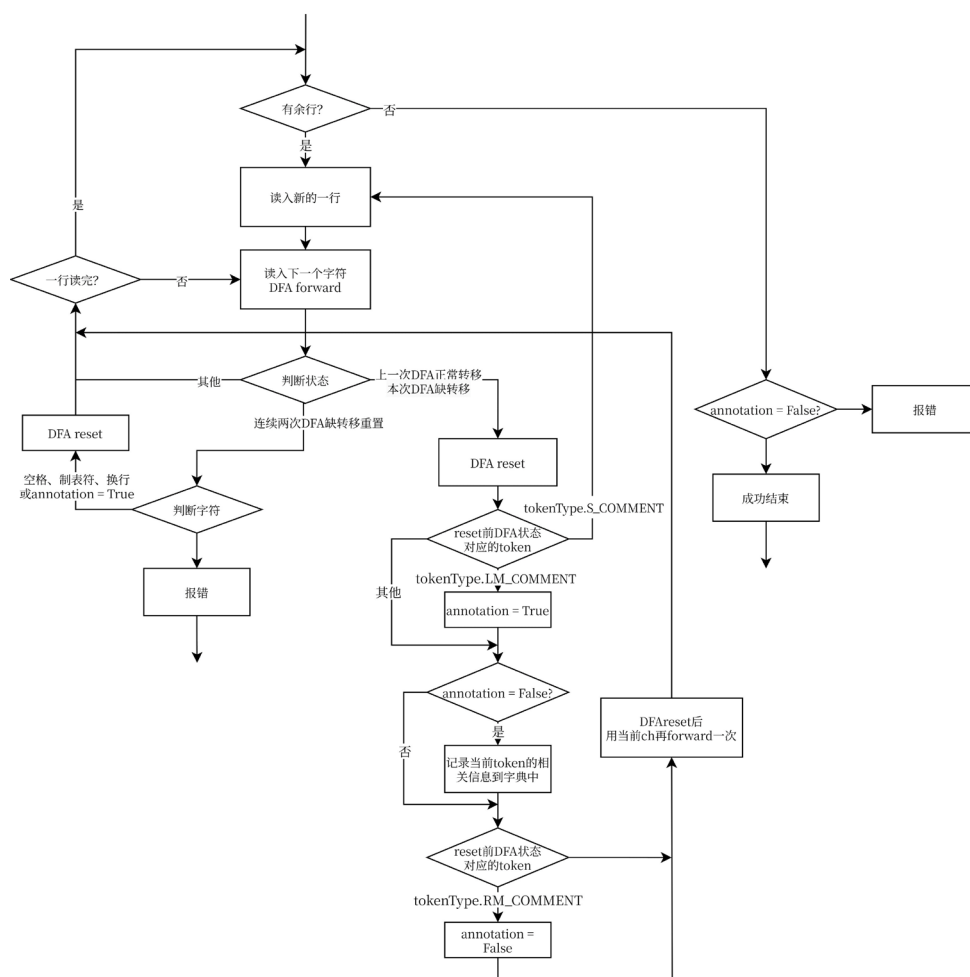
The screenshot shows the 'C语言中间代码生成器' (C Language Intermediate Code Generator) interface. The '代码编辑器' (Code Editor) contains the following C code:

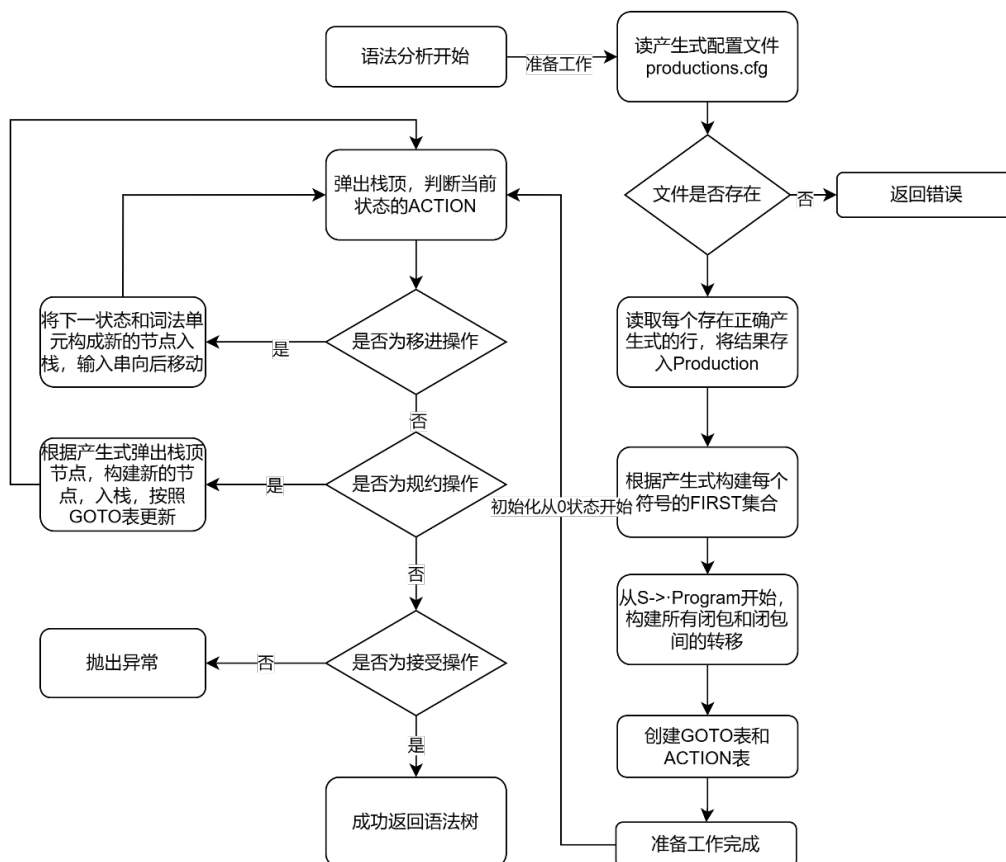
```

1  int main(){
2      return 0; ¥
3  }
    
```

The '报错信息' (Error Information) section shows: Error at (2,12): 无法识别的字符/字符串

The '中间代码' (Intermediate Code) section shows a message: 代码中包含 Error，中间代码暂不可用





## 3.3. 语义分析与中间代码生成

### 3.3.1. 数据结构

```

class Word:
    def __init__(self, id=0, name=""):
        """
        Word 类的构造函数。

        参数：
        - id (int): 单词的唯一标识符。
        - name (str): 与单词相关联的名称或值。
        """
        self.id = id
        self.name = name
        self.type = ""
    
```

```
def __repr__(self):  
    """  
    Word 对象的字符串表示。  
  
    返回：  
    str: 包含有关 Word 对象信息的格式化字符串。  
    """  
    return f"<Word 对象 (ID:{self.id}, 名称:{self.name}, 类  
    型:{self.type})>"
```

```
class Quaternion:  
    def __init__(self, op="", src1="", src2="", tar=""):  
        """  
        Quaternion 类的构造函数。  
  
        参数：  
        - op (str): 四元式的操作符。  
        - src1 (str): 第一个操作数。  
        - src2 (str): 第二个操作数。  
        - tar (str): 结果存放的目标操作数。  
        """  
        self.op = op  
        self.src1 = src1  
        self.src2 = src2  
        self.tar = tar  
  
    def __repr__(self):  
        """  
        Quaternion 对象的字符串表示。  
  
        返回：  
        str: 包含有关 Quaternion 对象信息的格式化字符串。  
        """  
        return f"({self.op}, {self.src1}, {self.src2},  
        {self.tar})"
```

```
class Attribute:  
    def __init__(self):
```

```
"""
Attribute 类的构造函数。

属性：
- type (str): 值类型，可以是 "int"、"float"、"word" 或
"tmp_word"。
- place: 存储位置。
- quad: 下一条四元式位置。
- truelist (list): true 条件跳转目标的列表。
- falselist (list): false 条件跳转目标的列表。
- nextlist (list): 顺序执行下一目标的列表。
- queue (list): 队列，用于函数参数。

注意: place、quad、truelist、falselist、nextlist 和 queue 的
初始值为 None 或空列表。
"""

self.type = ""
self.place = None
self.quad = None
self.truelist = []
self.falselist = []
self.nextlist = []
self.queue = []

def __repr__(self):
    """
    Attribute 对象的字符串表示。

    返回：
    str: 包含有关 Attribute 对象信息的格式化字符串。
    """
    return f"<Attribute 对象 (Type:{self.type},
Place:{self.place}, Truelist:{self.truelist},
Falselist:{self.falselist}, Nextlist:{self.nextlist},
Quad:{self.quad})>"

class Process:
    def __init__(self, start_address):
        """
```

Process 类的构造函数。

参数：

- start\_address: 过程的起始地址。

属性：

- name (str): 过程的名称。

- return\_type (str): 过程的返回类型。

- actual\_returns (list): 过程的实际返回值列表。

- start\_address: 过程的起始地址。

- words\_table (list): 用于存储 Word 对象的词法表，初始包含一个空的 Word 对象。

- param (list): 过程的参数列表。

注意: name、return\_type、actual\_returns 的初始值为空字符串或空列表。

```

"""
self.name = ""
self.return_type = ""
self.actual_returns = []
self.start_address = start_address
self.words_table = [Word()]
self.param = []

def __repr__(self):
    """
    Process 对象的字符串表示。

    返回:
    str: 包含有关 Process 对象信息的格式化字符串。
    """
    return f"<Process 对象 (Name:{self.name}, Return
Type:{self.return_type}, Start Address:{self.start_address},
Params:{self.param})>"
    
```

### 3.3.2. 重点函数

#### 1. 查找单词编号 (checkup\_word) :



```
def checkup_word(self, word_name):  
    """  
    检查词法表中是否存在指定名称的单词。  
  
    参数：  
    - word_name (str): 要检查的单词名称。  
  
    返回：  
    int: 如果在当前作用域内找到，返回该单词在当前作用域的索引；  
        如果在全局变量中找到，返回负值表示在全局变量中的索引的相反数；  
        如果未找到，返回 0。  
    """  
    words_table = self.process_table[-1].words_table  
  
    # 在作用域内找到  
    for i, word in enumerate(words_table):  
        if word.name == word_name:  
            return i  
  
    # 全局变量  
    for i, word in enumerate(self.words_table):  
        if word.name == word_name:  
            return -i  
  
    return 0
```

## 2. 查找单词的类型 (checkup\_word\_type) :

```
def checkup_word_type(self, word_name):  
    """  
    获取指定单词名称的数据类型。  
  
    参数：  
    - word_name (str): 要获取数据类型的单词名称。  
  
    返回：  
    str or None: 如果在当前作用域内找到，返回该单词的数据类型；  
        如果在全局变量中找到，返回该单词的数据类型；  
        如果未找到，返回 None。  
    """
```

```
"""
words_table = self.process_table[-1].words_table

# 在当前作用域内找到
word_type = next((word.type for word in words_table if
word.name == word_name), None)

# 低层屏蔽高层，再去全局变量找
if word_type is None:
    word_type = next((word.type for word in
self.words_table if word.name == word_name), None)

return word_type
```

### 3. 获取指定位置处的单词（get\_word）：

```
def get_word(self, place):
    """
    获取指定位置处的单词。

    参数：
    - place (int)：要获取的单词的位置。

    返回：
    Word or None：如果位置大于 0，在当前作用域词法表中返回对应单词；
                   如果位置小于等于 0，在全局变量的词法表中返回对应单词；
                   如果未找到，返回 None。
    """
    if place > 0:
        return self.process_table[-1].words_table[place]
    else:
        return self.words_table[-place]
```

### 4. 创建单词（create\_word）：

```
def create_word(self, word):
    """
    在当前作用域的词法表中创建一个新的单词。
```

```
参数：
- word (Word)：要创建的单词对象。

返回：
无返回值。
"""
words_table = self.process_table[-1].words_table
word.id = len(words_table)
words_table.append(word)
```

## 5. 报告错误信息 (raise\_error) :

```
def raise_error(self, type, loc, msg):
    """
    报告错误信息。

    参数：
    - type (str)：错误类型，例如 "Error"。
    - loc (dict)：错误位置信息，包括行号和列号。
    - msg (str)：错误消息。

    返回：
    无返回值。
    """
    if type == "Error":
        self.error_occur = True
        self.error_msg.append(f"{type} at
({loc['row']},{loc['col']}) : {msg}")
```

## 6. 语义分析 (analyse) :

本函数是本实验的核心代码，其目的是针对每个产生式做各自对应的分析，由于代码较长，不在报告中展示，下面将针对每一个文法产生式的处理方法做单独介绍。

### a. 语法规则：VarStatement

这个规则表示了声明语句，可以声明整型或浮点型的变量。

在语法分析的过程中，当遇到这个产生式时，从语法分析栈中取出产生式右侧的终结符对应的变量名，获取产生式右侧变量的类型，检查是否重复定义了同名的变量，如果已经定义，则报错。如果变量名未重复定义，创建一个新的变量对象，设置其名称和类型，并在当前作用域的词法表中添加这个变量。为新变量创建属性对象并设置其类型，将属性对象关联到当前语法树节点。

## b. 文法规则：FactorExpression

这个表达式可以是标识符、整数常量、浮点数常量、带括号的关系表达式、除法操作、乘法操作或函数调用。

如果 FactorExpression 被识别为 FunctionCallExpression（函数调用），则获取并检查返回属性，如果返回类型不是临时词（tmp\_word），则引发无效操作的错误，因为操作数是 void 类型；如果 FactorExpression 是标识符，则检查标识符是否已声明，如果没有，则引发未声明标识符的错误，然后，属性 tmp 被填充为有关标识符的信息；如果 FactorExpression 是整数或浮点数常量，则相应地设置属性 tmp；如果 FactorExpression 是形如（RelopExpression）的形式，则传播内部表达式的属性；如果 FactorExpression 包含乘法或除法，它执行类型检查，并在需要时插入用于类型转换的临时词和四元组。

在处理上述结果后，计算得到的属性存储在 tmp 属性中，其中包括类型信息、位置信息以及关于四元组的信息，并对使用 void 类型操作数和引用未声明的标识符等情况进行了错误检查，在表达式评估过程中使用临时词和四元组来管理中间结果。

## c. 文法规则：AddExpression

处理语法中的 AddExpression 表达式，涉及到加法和减法运算。

在语法分析过程中，它首先检查当前处理的产生式是否为 AddExpression，然后创建一个临时属性对象 tmp，用于存储表达式的属性信息。如果产生式右

---

侧只有一个元素，直接将该元素的属性信息赋给 `tmp`；如果右侧有两个元素，表示是加法或减法表达式，进行语义检查，获取操作数的类型和名称。如果需要类型转换，生成临时变量和相应的四元组，并更新操作数的名称。接着，创建一个新的临时变量 `tmp_word`，用于存储表达式的结果，并更新符号表和四元组表。

最后，更新 `tmp` 的属性信息，包括类型、位置、四元组信息以及真值和假值列表。整个过程涵盖了对加法和减法表达式的语义分析，包括类型检查、类型转换和生成相应的临时变量和四元组。最终，将计算结果的属性信息存储在 `item["attribute"]` 中，以便后续语法分析阶段使用。

#### d. 文法规则：RelopExpression

处理语法中的 `RelopExpression` 表达式，涉及到关系运算符（比较运算符）以及可能的条件跳转。

在语法分析阶段，首先检查当前处理的产生式是否为 `RelopExpression`，然后创建一个临时属性对象 `tmp`，用于存储表达式的属性信息。如果产生式右侧只有一个元素，表示是 `RelopExpression -> AddExpression` 的情况，直接将该元素的属性信息赋给 `tmp`。如果右侧有两个元素，表示涉及关系运算符的情况，会进行如下处理：首先，创建一个临时变量 `tmp_word`，用于存储表达式的结果。接着，生成相应的四元组，包括操作符、操作数等信息，并更新符号表。进行语义检查，如果操作数类型不匹配则报错（本次实验未对关系表达式进行类型转换的处理）。随后，生成用于条件跳转的四元组，包括条件跳转的目标地址。最后，更新 `tmp` 的属性信息，包括类型、位置以及四元组信息。

最终，将计算结果的属性信息存储在 `item["attribute"]` 中，以便后续语法分析阶段使用。整个过程涵盖了关系运算符表达式的语义分析，包括类型检查、生成相应的四元组以及条件跳转的处理。

#### e. 文法规则：AssignmentStatement

处理语法中的 `AssignStatement`，用于赋值语句的语义分析。

在语法分析阶段，首先检查当前处理的产生式是否为 `AssignStatement`，然后创建一个临时的四元组对象 `tmp_quaternion`，其中操作符为 `=`，第二操作数为下划线（表示无需使用），用于表示赋值操作，接着确定赋值语句的目标标识符的类型，如果该标识符未声明，则报错，然后获取赋值语句右侧表达式的类型和名称。如果目标标识符的类型与右侧表达式的类型不一致，创建一个新的临时变量 `tmp_word`，用于存储类型转换后的值，根据类型，生成相应的四元组，将右侧表达式的值转换为目标类型，并更新右侧表达式的名称，同时，发出一个警告，提示可能会丢失数据。

最后，更新 `tmp_quaternion` 的第一操作数为右侧表达式的名称，将这个四元组添加到四元组表中，创建一个临时属性对象 `tmp`，设置 `quad` 为当前四元组的数量，并将其存储在 `item["attribute"]` 中，以便后续语法分析阶段使用。

## f. 文法规则：Block

处理语法中的 `Block`，用于表示代码块的语义分析。

在语法分析阶段，首先检查当前处理的产生式是否为 `Block`，然后创建一个临时的属性对象 `tmp`。根据产生式右侧元素的数量，如果右侧有 4 个元素，表示 `{ InnerStatement StatementString }` 形式的代码块。在这种情况下，将 `tmp.nextlist` 设置为 `InnerStatement` 生成的代码块的 `nextlist`；如果右侧有 3 个元素且第二个元素的根节点是 `StatementString`，表示 `{ StatementString }` 形式的代码块，将 `tmp.nextlist` 设置为 `StatementString` 生成的代码块的 `nextlist`。

最后，将属性信息存储在 `item["attribute"]` 中，以便后续语法分析阶段使用。整个过程涵盖了代码块的语义分析，主要是设置 `nextlist`，该属性在后续的语义处理中可能用于控制流程。

## g. 文法规则：StatementString

处理语法中的 `StatementString`，用于表示一系列语句的语义分析。

在语法分析阶段，首先检查当前处理的产生式是否为 `StatementString`，然

---

后创建一个临时的属性对象 `tmp`。根据产生式右侧元素的数量，如果右侧有一个元素，表示 `StatementString -> Statement` 形式的语句序列。在这种情况下，将 `tmp.nextlist` 设置为 `Statement` 生成的语句的 `nextlist`；如果右侧有三个元素，表示 `StatementString -> StatementString M Statement` 形式的语句序列。在这种情况下，遍历 `StatementString` 生成的语句序列的 `nextlist`，对每个元素执行 `backpatch` 操作，将其指向 `M.quad + self.start_address`，然后将 `tmp.nextlist` 设置为 `Statement` 生成的语句的 `nextlist`。

最后，将属性信息存储在 `item["attribute"]` 中，以便后续语法分析阶段使用。整个过程涵盖了语句序列的语义分析，其中涉及到 `nextlist` 的设置和 `backpatch` 操作，用于在控制流程中处理跳转目标。

## **h. 文法规则：Statement**

处理语法中的 `Statement`，用于表示各种语句的语义分析。

在语法分析阶段，首先检查当前处理的产生式是否为 `Statement`，然后创建一个临时的属性对象 `tmp`。将 `tmp.nextlist` 设置为第二个元素（可能是 `IfStatement`、`WhileStatement` 等）生成的语句的 `nextlist`。最后，将属性信息存储在 `item["attribute"]` 中，以便后续语法分析阶段使用。

## **i. 文法规则：epsilon**

处理一系列空产生式，即产生式右侧为空的情况。

处理空产生式 `M -> epsilon`：创建一个临时属性对象 `tmp`，设置 `tmp.quad` 为当前四元组表的长度（`len(self.quaternion_table)`），表示当前位置的四元组的地址，将 `tmp` 存储在 `item["attribute"]` 中。

处理空产生式 `N -> epsilon`：创建一个临时属性对象 `tmp`，设置 `tmp.quad` 为当前四元组表的长度（`len(self.quaternion_table)`），表示下一条语句的地址，创建一个新的条件跳转四元组，将其添加到四元组表中，将 `tmp` 存储在 `item["attribute"]` 中。

处理空产生式  $A \rightarrow \epsilon$ : 创建一个临时属性对象 `tmp`, 将 `tmp` 添加到 `tmp.truelist` 中, 表示真值条件跳转的地址, 将 `len(self.quaternion_table) + 1` 添加到 `tmp.falselist` 中, 表示假值条件跳转的地址, 创建两个条件跳转四元组, 一个用于真值条件跳转, 一个用于假值条件跳转, 并将它们添加到四元组表中, 将 `tmp` 存储在 `item["attribute"]` 中。

处理空产生式  $S \rightarrow \epsilon$ : 在四元组表中插入一条条件跳转的空白语句, 将跳转地址设置为下划线, 待后续回填。

处理空产生式  $P \rightarrow \epsilon$ : 创建一个新的进程, 将其插入到进程表中, 起始地址为当前四元组表的长度加上程序的起始地址。

## j. 语法规则: IfStatement

处理语法中的 IfStatement, 用于表示条件语句的语义分析。

在语法分析阶段, 首先检查当前处理的产生式是否为 IfStatement, 然后创建一个临时的属性对象 `tmp`。

如果右侧为 `if ( RelopExpression ) M Block N else M Block` 形式的条件语句, 对 `A.truelist` 中的每个地址执行 `backpatch` 操作, 将其中的跳转地址的 `src1` 设置为 `RelopExpression` 的值, `tar` 设置为 `else` 语句块的起始地址, 对 `A.falselist` 中的每个地址执行 `backpatch` 操作, 将其中的跳转地址的 `tar` 设置为 `else` 语句块的结束地址, 将 `S.nextlist` 设置为 `if` 语句块、`else` 语句块和 `M2` 语句块的合并结果。

如果右侧为 `IfStatement  $\rightarrow$  if ( RelopExpression ) M Block` 形式的条件语句, 对 `A.truelist` 中的每个地址执行 `backpatch` 操作, 将其中的跳转地址的 `src1` 设置为 `RelopExpression` 的值, `tar` 设置为 `if` 语句块的起始地址, 将 `S.nextlist` 设置为 `A.falselist` 和 `if` 语句块的合并结果。

最后, 将属性信息存储在 `item["attribute"]` 中, 以便后续语法分析阶段使用。整个过程涵盖了条件语句的语义分析, 其中主要涉及到 `backpatch` 操作、



跳转地址的设置以及 nextlist 的合并。

## k. 文法规则: WhileStatement

处理语法中的 WhileStatement, 用于表示循环语句的语义分析。

在语法分析阶段, 首先检查当前处理的产生式是否为 WhileStatement, 然后创建一个临时的属性对象 tmp。对 S1.nextlist 中的每个地址执行 backpatch 操作, 将其中的跳转地址的 tar 设置为 while 语句块的起始地址, 对 A.truelist 中的每个地址执行 backpatch 操作, 将其中的跳转地址的 src1 设置为 RelopExpression 的值, tar 设置为 while 语句块的结束地址, 将 S.nextlist 设置为 A.falselist, 表示循环结束后执行的下一条语句, 向四元式表中添加一条 j 指令, 将跳转目标设置为 while 语句块的起始地址。

最后, 将属性信息存储在 item["attribute"] 中, 以便后续语法分析阶段使用。整个过程涵盖了循环语句的语义分析, 其中主要涉及到 backpatch 操作、跳转地址的设置以及 nextlist 的更新。

## l. 文法规则: DeclarationType

处理语法中的 DeclarationType, 用于区分声明的类型。

在语法分析阶段, 首先检查当前处理的产生式是否为 DeclarationType, 然后创建一个临时的属性对象 tmp, 在 DeclarationType -> VarDeclaration 形式的情况下, 表示变量声明, 将 tmp.declaration\_type 设置为 "var", 在 P Function Declaration 形式的情况下, 表示函数声明, 将 tmp.declaration\_type 设置为 "func", 并将参数列表 queue 从 FunctionDeclaration 的属性中传递给 tmp.queue。最后, 将属性信息存储在 item["attribute"] 中, 以便后续使用。

## m. 文法规则: Declaration

处理语法中的 Declaration, 用于声明变量或函数。

首先, 检查产生式右侧的形式来确定是函数声明还是变量声明, 如果产生式表示函数声明。在这种情况下, 将函数的名称、参数列表和返回类型存储到

`self.process_table` 中，并检查函数是否已被定义。同时，检查 `main` 函数的返回类型是否为 `int`。如果产生式表示变量声明。在这种情况下，创建一个新的 `Word` 对象表示变量，设置其名称、类型，并将其添加到 `self.words_table` 中。

## **n. 语法规则：HeadVarStatements**

处理语法中的 `HeadVarStatements`，即函数声明中的参数列表。

如果只有一个参数的情况，将其类型添加到参数列表中。如果存在多个参数，递归地将前面的参数列表与新的参数类型合并。

## **o. 语法规则：FunctionDeclaration**

处理语法中的 `FunctionDeclaration`，即函数的声明。

如果其表示有参数的函数声明，将参数的类型列表保存在 `tmp.queue` 中，如果其表示没有参数的函数声明，如果其表示没有参数的函数声明，其中 `void` 表示没有参数。

## **p. 语法规则：Arguments**

处理语法中的 `Arguments`，即函数的参数列表。

如果函数只有一个参数，在这种情况下，将参数的类型和名称添加到 `tmp.queue` 中。如果函数有多个参数的情况。在这种情况下，将之前参数的列表和新参数的类型和名称添加到 `tmp.queue` 中。

## **q. 语法规则：FunctionCallExpression**

处理 `FunctionCallExpression`，即函数的调用。

对于有参函数调用，需要检查传递的参数类型和数量是否与函数声明一致，如果不一致，报错。对于无参函数调用，需要检查函数声明中是否有参数，如果有，报错。如果函数有返回值，则在调用后创建一个临时变量来保存返回值，并生成相应的四元式。如果函数返回类型为 `void`，则直接生成调用函数的四元式。

---

## r. 文法规则：ReturnStatement

处理 ReturnStatement，表示返回语句。

创建 ReturnStatement 的属性对象 tmp，生成 ret 操作码的四元式。这个四元式表示返回语句，将函数的控制权返回给调用者，如果有返回值，将返回值的类型和位置信息添加到当前函数的 actual\_returns 列表中，以用于后续检查函数返回类型是否与声明一致。将 tmp 对象设置为当前语法树节点的属性。

## 3.4. 界面设计

### 3.4.1. 页面总设计图



### 3.4.2. 重点函数

#### 1. 语法分析器计算部分和用户界面的异步加载

由于 Parser 需要一段比较长的时间读取 production.cfg 并完成初始化（大约需要十几秒），为了保证用户体验，这里使用 Qt 的信号-信号槽模型，通过一个 worker QThread 来完成 Parser 的初始化，并将准备完成的 Parser 传递

给主 Qt 线程。

这样就实现了 Parser 的异步加载，使得用户可以先看到启动完成的 Qt 界面以及“Parser 正在加载中”的提示，等待一段时间后由主 Qt 线程主动刷新界面，展示 Parser 结果。

```
class ParserLauncher(QObject):
    returnParser = pyqtSignal(Parser)
    finished = pyqtSignal()

    def run(self):
        self.returnParser.emit(Parser()) # 加载完成后传递给主线程
        self.finished.emit() # 通知主线程销毁该子线程

# Compiler 类中与此相关的部分
class Compiler(QObject):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.lexer = Lexer()
        self.parser = None
        self.parser_init_begin = time.time()
        self.getParserNonBlock() # 非阻塞加载 Parser

    def getParserNonBlock(self):
        self.pl_thread = QThread()
        self.pl_worker = ParserLauncher()
        self.pl_worker.moveToThread(self.pl_thread)
        self.pl_thread.started.connect(self.pl_worker.run)
        self.pl_worker.returnParser.connect(self.setParser)
        self.pl_worker.finished.connect(self.pl_thread.quit)
        self.pl_thread.finished.connect(self.pl_thread.deleteLater)

        self.pl_thread.start()

    @pyqtSlot(Parser)
    def setParser(self, parser):
        self.parser = parser # 接收子线程传递过来的 Parser 对象
        delta = time.time() - self.parser_init_begin
        print(f"self.parser is set to {self.parser}")
        print(f"Parser took {delta} seconds to initialize")
```

```

        self.goto_table = self.parser.get_goto_table() # 计算
goto 表 (不随输入变化, 只需计算一次)
        self.action_table = self.parser.get_action_table() # 计
算 action 表 (不随输入变化, 只需计算一次)
        self.parent().page().runJavaScript("window.cpf.flush();
") # 主动刷新用户界面

```

## 2. 语法分析器和用户界面的通信, 以及对具体计算函数的调用

由于用户界面和语法分析器也是异步运行的, 并且用户界面代码由 JavaScript 编写, 并且运行在 QWebEngineView 这个类似浏览器的环境中, 所以它们二者之间的通信也使用了 Qt 的信号-信号槽模型。UI 代码可以通过 QWebChannel 调用 Compiler.process() 函数, 间接调用语法分析函数, 并获取其结果。

```

# 省略了 Parser 加载的 Compiler 类的剩余部分
class Compiler(QObject):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.lexer = Lexer()
        self.parser = None
        self.parser_init_begin = time.time()
        self.getParserNonBlock() # 省略了 Parser 加载的剩余部分

    @pyqtSlot(str, result=str)
    def process(self, code_str):
        token_list, lexer_success = self.getLex(code_str)
        parse_result = self.getParse(token_list)
        return json.dumps(
            {
                "lexer": self.dumpTokenList(token_list),
                "lexer_success": lexer_success,
                **parse_result,
            }
        )

    def dumpTokenList(self, token_list):
        def dumpToken(r):

```

```

        r["prop"] = r["prop"].value
        return r

    return list(map(dumpToken, token_list))

def getParse(self, token_list):
    if self.parser is not None:
        parsed_result = self.parser.getParse(token_list)
        return { # 如果 Parser 已经加载完成, 返回 Parser 结果
            "ast": parsed_result,
            "goto": self.goto_table,
            "action": self.action_table,
            "process": self.parser.parse_process_display,
            "semantic_quaternation": self.parser.semantic_quaternation,
            "semantic_error_occur":
self.parser.semantic_error_occur,
            "semantic_error_message":
self.parser.semantic_error_message,
        }
    else:
        launching = "Parser 正在启动, 请稍等。"
        return { # 如果 Parser 未加载完成, 返回提示信息
            "ast": {
                "root": launching,
                "err": "parser_not_ready",
            },
            "goto": [[launching]],
            "action": [[launching]],
            "process": [[launching]],
        }

def getLex(self, code_str: str):
    return self.lexer.getLex(code_str.splitlines())

```

### 3. QWebEngineView 与 aiohttp.web.Application 双进程并行

用户界面代码由 HTML + JavaScript + CSS 写成, 但 QWebEngineView 本身并没有直接加载它们的能力, 需要一个 HTTP Server 来 serve 这些静态文件。我们通过在子进程中分别运行 aiohttp Server 和主 Qt 进程, 并通过环回

地址来完成 QWebEngineView 对本地静态文件的访问。

```
class MainWindow(QWebEngineView):
    def __init__(self):
        super(MainWindow, self).__init__()
        self.load(QUrl(f"http://localhost:{serverPort}/index.html")) # 通过环回地址访问 Server 提供的本地静态文件
        self.webChannel = QWebChannel(self.page()) # 初始化 PyQt 侧的 QWebChannel
        self.webChannel.registerObject("compiler", Compiler(self)) # 初始化 Compiler 对象, 绑定到 QWebChannel 上
        self.page().setWebChannel(self.webChannel) # 应用 QWebChannel, 触发 JavaScript 侧 QWebChannel 初始化

def ServerProcess(application_path):
    # serve 本地静态文件
    app = web.Application()
    app.add_routes([
        web.static(
            "/",
            os.path.join(application_path, serverDirectory),
            show_index=True,
            follow_symlinks=False,
            append_version=True,
        )
    ])
    web.run_app(app, host="localhost", port=serverPort) # 监听环回地址, 响应对本地静态文件的 HTTP 请求

def QtProcess():
    app = QApplication(sys.argv)
    window = MainWindow()
    window.showMaximized()
    app.exec_()
```

## 4. 主用户界面代码

这部分代码主要工作是：初始化 `QWebChannel` 以及用户界面主体（包括左侧的 `Monaco Editor`）。值得一提的是，虽然我们使用的 `Monaco Editor` 自带代码高亮功能，但是我们通过使用纯文本模式将其禁用，然后应用了我们自己根据我们的语法分析器结果生成的代码高亮。

用户界面控制逻辑基本写在 `CPF` 类中，该类会在窗口加载完成后实例化，并且在 `QWebChannel` 初始化完成后从 `Python` 端接受一个 `compiler` 对象（`Python` 代码中的 `Compiler` 类），来完成 `JavaScript` 端和 `Python` 端的连接。

这个类会在编辑器中代码发生变化时，取得其中的文本内容，传递给语法分析器，获取词法分析、语法分析结果，以及其他需要展示的内容，并且调用代码高亮、语法分析树以及各个表格的绘制代码。由于代码过于冗长，此处不进行展示，详见对应文件中。

## 4. 结语

### 4.1. 实验总结

在本次实验中，我们在上次词法、语法分析器的基础上，成功地实现了类 `C` 语言的中间代码生成器，并聚焦了静态语义错误的处理。在代码的编辑界面中，我们采用了左侧编辑器输入的方式，并加入了语法高亮功能，使用户能够更清晰地看到源代码的结构和错误提示。整个界面设计注重用户友好性，提供了方便的输入方式和直观的可视化展示，使用户能够轻松地查看语法分析树、分析表、移进规约过程和中间代码的生成过程。此外，我们还将构建好的分析表打包成静态文件，允许程序在需要时快速读取从而减少构建分析表的开销。

在语法分析的基础上，我们扩展实现了对浮点数与整型数的类型转换，以及对过程调用的支持。在过程调用中，我们引入了全局变量和局部变量的作用域管理，使得过程内外变量的访问得以正确实现。过程调用中间代码生成的逻辑清晰，参数传递、过程体执行和返回值传递等方面都得到了有效处理。

在静态语义错误检查方面，我们实现了对变量定义问题、变量类型问题、

---



函数定义问题和函数返回值问题的全面检测。通过引入 `warning` 和 `error` 两种报告形式，我们既实现了清晰易懂的可视化报错界面，又能阻止生成可能导致错误的目标代码，保证了最终生成的中间代码的正确性。通过详细而清晰的错误信息（报错信息含错误位置、错误原因等详细信息），我们生成的程序能够引导使用者迅速定位问题并进行代码逻辑修复。

总体而言，通过这次实验，我们不仅深入理解了编译器在各个阶段的工作原理，还提高了对中间代码生成以及静态语义错误检测的实际处理能力。这次实验既是对之前课程内容的巩固，又是对编译原理知识的进一步拓展，为我们更深层次地理解编译原理奠定了坚实的基础。

## 4.2. 通用高级语言的语义检查和中间代码生成的设计

在完成第二次大作业的基础上，我们还思考了通用高级编程语言的语义检查和中间代码生成设计。首先，需要深入关注类型检查的细节，确保语义分析时在运算和赋值操作中，涉及的变量和表达式的类型都符合语言规范。引入一个强大的类型系统，不仅能进行类型推断，还能有效地匹配各种运算和操作，从而在编译阶段就捕捉到潜在的类型错误。这种设计决策有助于提高代码的可读性和可维护性，确保程序在运行时不会受到类型相关的错误干扰。

其次，作用域和生命周期的检查是确保变量在正确上下文中使用的关键因素。通过引入并维护符号表，能够记录变量的声明、引用以及其在程序执行期间的生命周期。通过符号表的维护，我们能够在编译过程中严格检查变量的作用域，防止未定义变量的引用，从而提高程序的健壮性和可维护性。这一步骤对于语言的静态分析和编译优化都起到了关键的作用。本次实验中，由于我们的文法不支持过程嵌套，因此变量作用域较为简单。若要进一步支持过程嵌套，则要考虑更复杂的变量作用域的维护。此外，变量的初始化也至关重要。最好需要明确规定在使用变量之前必须进行初始化从而避免未初始化变量导致的不确定行为，提高程序的稳定性和可靠性。

再者，关于函数调用的检查也至关重要。我们需要检查函数调用时传递的参数数量、类型以及函数返回值是否与函数声明相匹配。这种一致性检查有助于防止由于函数调用不当而导致的潜在错误，保障程序在执行过程中的可靠性。同时，这也为编译器提供了更多的优化可能性，从而提高程序的性能。

最后，在处理错误时，生成清晰、详细的错误消息是提高开发效率和减少调试时间的关键。错误消息应该包含有关错误位置的准确信息，以便程序员能够迅速定位并修复问题。

## 5. 参考文献

- [1] Generation I C. Intermediate Code Generation[J]. 2006.
  - [2] Wilhelm R, Seidl H, Hack S. Compiler design: syntactic and semantic analysis[M]. Springer Science & Business Media, 2013.
  - [3] Hoe A V, Sethi R, Ullman J D. Compilers—principles, techniques, and tools[J]. 1986.
  - [4] 张素琴, 吕映芝. 编译原理[M]., 清华大学出版社
  - [5] 蒋立源、康慕宁等, 编译原理（第2版）[M], 西安: 西北工业大学出版社
  - [6] 陈火旺,刘春林,谭庆平,等.程序设计语言编译原理:第3版 [M].国防工业出版社, 2008
  - [7] 孙冀侠, 迟呈英, 李迎春. LR (1) 语法分析的自动构造[J]. 鞍山科技大学学报, 2003, 26(2): 90-92.
  - [8] Microsoft (2021). Monaco Editor. Microsoft. <https://microsoft.github.io/monaco-editor/>
  - [9] Muhammad Yahya. (2021). Dynamically Building Nested List from JSON Data and Tree View with CSS3. Medium. <https://medium.com/oli-systems/dynamically-building-nested-list-from-json-data-and-tree-view-with-css3-2ee75b471744>
  - [10] Mozilla Developer Network. (2018). QWebEngineView.html. Mozilla Developer Network. <https://developer.mozilla.org/en-US/search?q=QWebEngineView.html>
  - [11] 戴桂兰, 张素琴, 田金兰, 等. 编译系统中间代码的一种抽象表示[J]. 电子学报, 2002, 30(S1): 2134.
  - [12] 李磊. C 编译器中间代码生成及其后端的设计与实现[D]. 电子科技大学, 2016.
-