

编译器设计专题实验六：

中间代码生成

计算机 2101 田濡豪 2203113234

1 实验内容（必做）

1.1 实验要求（用户需求）

目标：中间代码生成

输入：输入抽象语法树信息或其他，token，源程序等等。

输出：输出中间代码表示，三元式、四元式、三地址方式均可。

备注：文法是上一个实验的文法。

```
P -> D' S' # 程序入口
D' -> epsilon | D' D; # 声明表
D -> T d | T d[i] | T d(A'){D' S'} # 单个声明
T -> int | float | void; # 类型
A' -> epsilon | A' A; # 形参
A -> T d | T d[] | T d(T) # 单个形参
S' -> S | S' ; S # 语句表
S -> d = E | d[E] = E | if (B) S | if (B) S else S | while (B)
S | return E | {S'} | d(R') # 语句
E -> num | flo | d | d[E] | E + E | E * E | (E) | d(R') # 常规
算数表达式
B -> E r E | E # 布尔表达式
R' -> epsilon | R' R, # 实参表
R -> E | d[] | d() # 单个实参
```

1.2 实验设计

备注：由于代码量过大，相关文件的相对路径使用红字标出。为照顾阅读体验不在文中直接给出代码。本实验的默认路径为 lab6/

1.2.1 需求分析

先前实验已经完成了语法/语义分析并填充了符号表。本次实验的目的是中间代码生成。特别的，实验没有对中间代码表示做明确要求，因此本次实验主要需要完成：

1. 设计一个中间代码表示及对应的模拟环境，比如模拟一个自定义逻辑寄存器配置、内存空间以及指令架构
2. 增强现有 AST 属性：现有 AST 属性完全是为了语法和语义分析设计的，没有考虑到中间代码生成。由于中间代码生成本质上也是遍历 AST 树，所以需要在其中补充有利代码生成的相关信息。
3. 实现中间代码生成工具，该工具通过遍历 AST 树、利用符号表等信息生成中间代码。期间它需要维护寄存器、内存空间使用情况等。

1.2.2 虚拟生成环境设计

参见：[include/interm_code/interm_code_model.h](#)

1.2.2.1 寄存器

考虑到时间和实现复杂性，假设逻辑寄存器分为两类：

- T 通用寄存器：用于存储临时变量和中间结果
- R 通用寄存器：专门用于函数调用过程，如传递参数、返回值等
- RA 通用寄存器：专门用于函数的返回地址。

假设各类寄存器数量无限。

寄存器配合不同作用域的工作流程如下：

- 当进入到一个新作用域后，立刻断定目前所有的 T 寄存器都是空的。随后，将 R 寄存器（即传入的参数，如果有的话）依次加载到 T 寄存器中。此后 R 寄存器将不再使用，直到当前作用域结束。
- 当从当前作用域返回父作用域时，将返回值结果（如果有的话）存储到 R 寄存器中。然后根据 RA 寄存器的内容跳转到父作用域的返回地址。此时 RA 寄存器将不再使用，直到下一个函数调用。
- 在执行当前作用域语句时，维护寄存器的使用情况。如果遇到函数调用，将目前使用的 T 寄存器的内容按照编号顺序依次存入栈中。最后在栈顶压入 RA 寄存器的内容（这是当前函数——而不是要进入的函数——要返回的地址）。
- 在函数调用结束后，首先将 RA 寄存器从栈顶中弹出并复原，之后将栈中的寄存器内容依次恢复到 T 寄存器中，然后从 R 寄存器中取出返回值（如果有的话）。

比如一个程序的运行结构如下

```
main() {  
    // 使用 T0
```

```

    // 使用 T1
    T2 = func1(T0, T1); // 调用 func1
    // 使用 T2
    T2 = func2(T2); // 调用 func2
    // 结束
}
func1(R0, R1) {
    // 使用 T0
    // 使用 T1
    T2 = func3(T0, T1); // 调用 func3
    // 返回 T2
}
func2(R0) {
    // 使用 T0
    // 返回 T0
}
func3(R0, R1) {
    // 使用 T0
    // 使用 T1
    // 返回 T0
}

```

在上面的例子中，寄存器使用情况如下：

1. 在`main`函数中，首先 T0 和 T1 被使用。随后调用`func1`，将 T0 和 T1 分别赋值给 R0 和 R1。之后查看当前作用域使用情况，发现 T0 和 T1 被使用，因此将 T0 和 T1 的内容依次存入栈中，记录其保存了 2 个寄存器。将 RA 寄存器压入栈顶，接着进入`func1`函数。
2. 进入 func1 函数后，首先将 R0 和 R1 的内容依次加载到 T0 和 T1 中。随后 T2 被使用，调用`func3`函数，将 T0 和 T1 传入。此时检查当前作用域使用情况，发现 T0 和 T1 被使用，因此将它们的内容依次存入栈中，记录其保存了 2 个寄存器。之后在栈顶添加当前的 RA 寄存器。接着进入`func3`函数。此时栈中依次是：func1 的 RA、func1 的 T1、func1 的 T0、main 的 RA（整个程序的总返回地址）、main 的 T1、main 的 T0。
3. 进入`func3`函数后，首先将 R0 和 R1 的内容依次加载到 T0 和 T1 中。进行各种操作，期间未经打断，最后将 T0 的内容作为返回值存入 R0 中。此时栈中依次是：func1 的 RA、func1 的 T1、func1 的 T0、main 的 T1、main 的 T0。
4. 从`func3`函数返回`func1`后，查看现场，发现有 2 个寄存器被保存，因此将栈中的 T0 和 T1 依次恢复到 T0 和 T1 中（即使后续不使用）。随后将 R0 的内容（即 func3 的返回值）存入 T2 中。此时栈中依次是：main 的 RA、main 的 T1、main 的 T0。之后将 T2 的内容存入 R0 中，准备返回到`main`函数。

5. 从`func1`函数返回`main`后，查看现场，发现有 2 个寄存器被保存，因此将栈中的 T0 和 T1 依次恢复到 T0 和 T1 中。随后将 R0 的内容（即 func1 的返回值）存入 T2 中。此时栈中为空。

6. 在`main`函数中，T2 被使用，调用`func2`函数，将 T2 的内容存入 R0 中。此时检查当前作用域使用情况，发现 T0、T1、T2 被使用，因此将它们的内容依次存入栈中，记录其保存了 3 个寄存器。接着进入`func2`函数。此时栈中依次是：main 的 RA、main 的 T2、main 的 T1、main 的 T0。

7. 进入`func2`函数后，首先将 R0 的内容加载到 T0 中。进行各种操作，期间未经打断，最后将 T0 的内容作为返回值存入 R0 中。此时栈中依次是：main 的 RA、main 的 T2、main 的 T1、main 的 T0。

8. 从`func2`函数返回`main`后，查看现场，发现有 3 个寄存器被保存，因此将栈中 RA 恢复，并将栈中的 T0、T1、T2 依次恢复到 T0、T1、T2 中。随后将 R0 的内容（即 func2 的返回值）存入 T2 中（此时 T2 被复写，这就是为什么要先恢复寄存器再处理返回值）。此时栈中为空。

1.2.2.2 内存空间

中间代码生成的逻辑空间分为 3 个部分：

- 程序代码段：存储中间代码指令
- 数据段：存储全局变量、静态变量等
- 栈段：存储函数调用时的寄存器现场、局部变量等

其地址彼此独立。对程序段，地址以语句为单位，每条语句都有一个唯一地址。对数据段，地址的单位是上个实验中定义的“最小内存单元”。对栈段，每个地址和一个寄存器一一对应。

1.2.2.3 指令架构

需要满足实验文法，至少需要以下指令：

- 寄存器赋值：将一个寄存器的值赋给另一个寄存器
- 寄存器加载：从内存中加载一个值到寄存器
- 寄存器存储：将寄存器的值存储到内存中
- GOTO 跳转：无条件跳转到指定地址
- 条件跳转：根据布尔表达式的结果跳转到指定地址
- 寄存器相加：将两个寄存器的值相加并存储到一个寄存器中

- 寄存器相乘：将两个寄存器的值相乘并存储到一个寄存器中
- 小于比较：比较一个寄存器的值是否小于另一个寄存器的值，并将结果存储到一个寄存器中
- 相等比较：比较一个寄存器的值是否等于另一个寄存器的值，并将结果存储到一个寄存器中
- 小于等于比较：比较一个寄存器的值是否小于等于另一个寄存器的值，并将结果存储到一个寄存器中
- 空指令：用于占位或表示无操作

1.2.3 关键数据模型设计

1.2.3.1 地址格式

为了表示不同的逻辑内存空间，地址格式由两部分组成：逻辑空间名+编号。

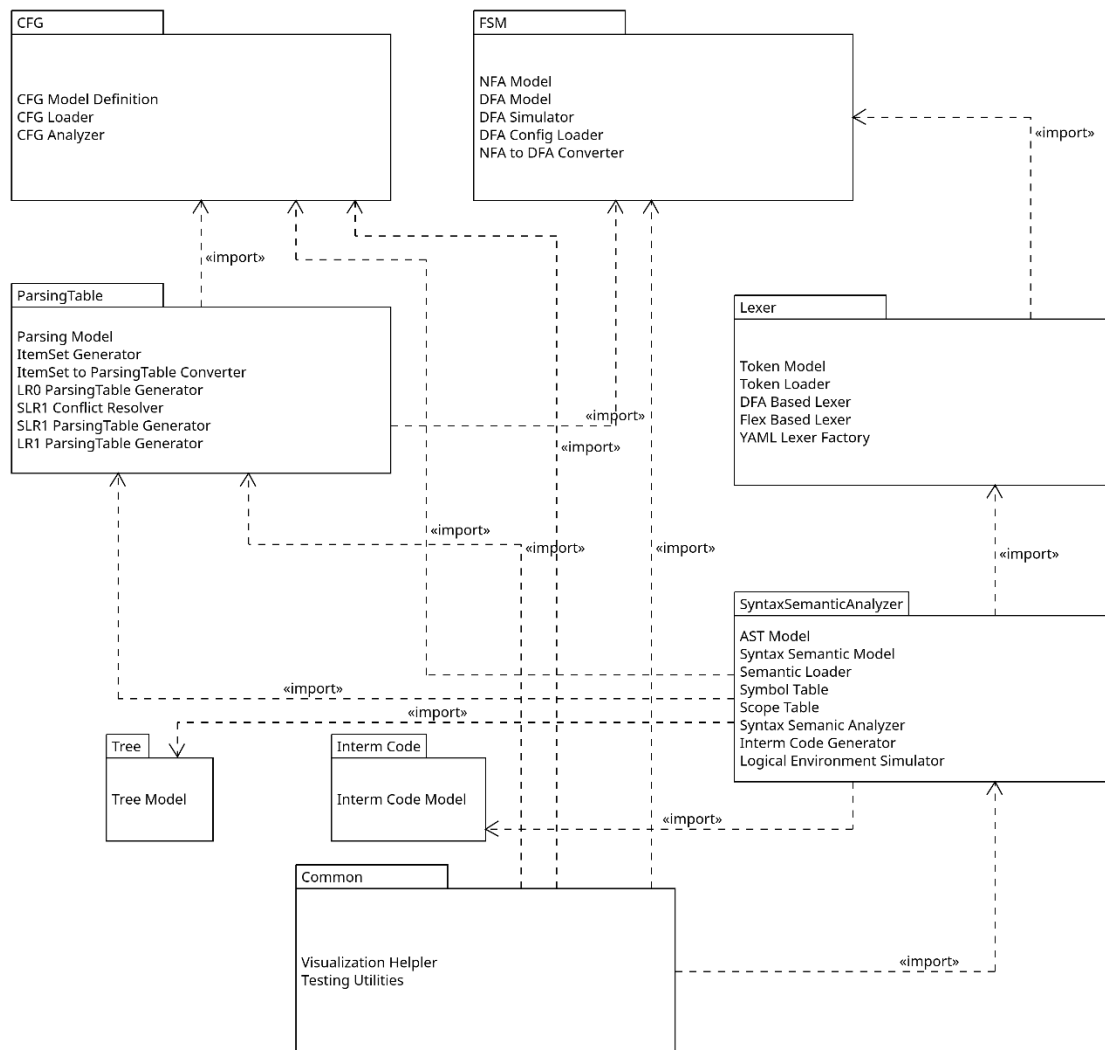
1.2.3.2 中间代码指令格式

规定指令格式为：操作码 + 操作数 1 + 操作数 2 + 操作数 3。其中操作数是可选的，对为空的情况，使用`-`表示。

此外还允许使用标签（label）来标记指令地址，以便进行跳转。标签为可选属性。

1.2.4 整体架构

基本维持实验 5 不变。



1.3 实现细节

1.3.1 AST 模型增强

1.3.1.1 AST 属性增强

参见: [include/syntax_semantic_analyzer/ast_model.h](#)

期望在语法语义分析结束后，遍历一遍 AST 树即可生成所需要的代码。这也就是说。当遍历到一个节点时，要用于中间代码生成的所有信息都已经准备好了。

观察实验给定的文法，并不是每个节点都会产生新的中间代码，产生代码的 AST 节点一定发生在语句表 S' 及其子节点中（作用域内语句），或是 D'/D 节点（声明）。

特别说明：此处的需求指节点所需要的内部信息（AST 本身的属性信息），不包含外部的寄存器、内存空间等信息。

S 节点产生语句的需求

- 对于 IF/WHILE 跳转类语句，B 节点应该将结果（值或寄存器）准备好用于判断。此外，S 节点应当准备好其所有对应中间代码（不能在后根遍历时直接写入代码，否则 IF/WHILE 语句无法进行跳转判断）。事实上，这个要求间接指明所有 S 节点及产生代码的其子节点（E、B、S'）都需要准备好中间代码。
- 对于赋值类语句，其需要的信息是变量信息和要赋值的值（或者寄存器）。变量名已经在符号表中准备好了，因此 E 节点需要将其结果（值或寄存器）在遍历时准备好。
- 对于函数调用语句，语义分析已经很有先见地维护了实参表到每个实参 R 的链接，但是每个 R，如果其类型为表达式（即子节点为 E），应当准备好对应的结果/寄存器。

E 节点产生语句的需求

同样，其子节点为 E、R' 应准备好相应的信息。

B 节点产生语句的需求

同上，需要 E 节点的结果信息。

D 节点产生语句的需求

- 变量和数组声明：需要的变量名、类型等都已经准备好了。
- 函数声明：需要子节点 D' 和 S' 的所有中间代码片段，因此 D' 和 S' 的子节点需要准备所有相应的代码。

1.3.2 AST 节点代码生成函数

参见：[src/syntax_semantic_analyzer/ast_interm_code_gen.cpp](#)

每个 AST 节点都需要一个代码生成函数，用于生成对应的中间代码。该函数需要根据节点类型和属性，生成相应的中间代码指令，并利用符号表和逻辑环境模拟器进行寄存器和内存的操作。

该函数实现为 AST 节点的一个方法，其利用以下信息：

- 直接输入：逻辑环境模拟器接口，用于节点和外部信息进行沟通。例如获取新的寄存器、将生成的代码写入中间代码指令列表等。
- 间接输入：AST 节点的内部信息，如节点类型、子节点等。这些信息可以通过 AST 节点的属性访问。

下面设计每个 AST 节点的代码生成函数的逻辑功能。

非 P、D'、D、S'、S、E、R'、R、B 的其他节点

生成函数直接返回空，因为这些节点不需要生成中间代码。

P 节点

$P \rightarrow D' S'$ # 程序入口

汇总所有子节点的中间代码片段即可。

D' 节点

$D' \rightarrow \text{epsilon} \mid D' D$; # 声明表

主要用于汇总所有子节点的中间代码片段。

如果子节点为 epsilon，初始化中间片段列表。

如果子节点为 D' D，先复制 D' 的中间片段列表，然后将 D 的中间代码片段添加到列表末尾。

D 节点

$D \rightarrow T d \mid T d[\text{num}] \mid T d(A')\{D' S'\}$ # 单个声明，分别是声明变量、数组和函数

如果为变量/数组声明，无需生成中间代码，直接返回。这是因为符号表已经存在，登记变量寄存器信息/数组内存信息可以通过遍历符号表完成。

如果为函数声明：

1. 初始化中间代码片段。
2. 获取函数对应的 label。
3. 在符号表中查找函数定义中的参数量，添加将对应数据从 R 形寄存器移到 T 形寄存器的中间代码。将第一条代码的标签设置为函数入口地址。
4. 从 D' 节点获取所有形参的中间代码片段，并添加到中间代码片段列表中。
5. 从 S' 节点获取所有函数体的中间代码片段，并添加到中间代码片段列表中。

S'节点

$S' \rightarrow S \mid S'; S \#$ 语句表

主要用于汇总所有子节点的中间代码片段。

如果子节点为 S，直接将 S 的中间代码片段添加到列表中。

如果子节点为 S'; S，先复制 S'的中间代码片段列表，然后将 S 的中间代码片段添加到列表末尾。

S 节点

$S \rightarrow d = E \mid d[E] = E \mid \text{if}(B) S \mid \text{if}(B) S \text{ else } S \mid \text{while}(B) S \mid \text{return } E \mid \{S'\} \mid d(R') \#$ 语句

如果为 $d = E$:

1. 获取变量的寄存器地址。
2. 获取 E 的结果寄存器。
3. 从 E 中取出中间代码片段作为当前节点的中间代码片段。
4. 生成寄存器存储指令，将 E 的结果寄存器的值存储到变量的寄存器地址中。将该指令添加到当前节点的中间代码片段列表中。

如果为 $d[E] = E$:

1. 获取第三个子节点——第一个 E 的代码列表，添加到当前节点的中间代码片段列表中。
2. 获取第六个子节点——第二个 E 的代码列表，添加到当前节点的中间代码片段列表中。
3. 获取列表的内存地址和第三个子节点的结果寄存器。
4. 添加一条 MUL 指令，将结果寄存器的值乘以数组元素大小，得到数组元素的内存地址偏移量。
5. 添加一条 ADD 指令，将数组基地址和偏移量相加，得到最终的数组元素内存地址。
6. 获取第六个子节点的结果寄存器。
7. 添加一条 STORE 指令，将结果寄存器的值存储到数组元素的内存地址中。

如果为 $\text{if}(B) S$:

1. 将 B 的中间代码列表添加到当前节点的中间代码片段列表中。

2. 在当前作用域下获取两个新的标签，一个用于进入 if 语句块，一个用于跳过 if 语句块。
3. 获取 B 的结果寄存器。
4. 添加一条条件跳转指令，比较 B 的结果寄存器，跳转地址是进入语句块的标签。
5. 添加一条无条件跳转指令，跳转到跳过语句块的标签。
6. 将 S 的中间代码片段的第一个指令打上进入语句块的标签。
7. 将 S 的中间代码片段添加到当前节点的中间代码片段列表中。
8. 添加一条空指令，并将其打上跳过语句块的标签。

如果为 if (B) S else S:

1. 将 B 的中间代码列表添加到当前节点的中间代码片段列表中。
2. 在当前作用域下获取两个新的标签，一个用于进入 if 语句块，一个用于进入 else 语句块。
3. 获取 B 的结果寄存器。
4. 添加一条条件跳转指令，比较 B 的结果寄存器，跳转地址是进入语句块的标签。
5. 添加一条无条件跳转指令，跳转到 else 语句块的标签。
6. 将第五个节点——第一个 S 的中间代码片段的第一个指令打上进入语句块的标签，并将其添加到当前节点的中间代码片段列表中。
7. 将第六个节点——第二个 S 的中间代码片段的第一个指令打上进入 else 语句块的标签，并将其添加到当前节点的中间代码片段列表中。

如果为 while (B) S:

1. 将 B 的中间代码列表添加到当前节点的中间代码片段列表中。
2. 在当前作用域下获取三个新的标签，一个用于进入 while 语句块，一个用于跳过 while 语句块，最后一个用于跳转到 while 判断的标签。
3. 获取 B 的结果寄存器。
4. 添加一条条件跳转指令，比较 B 的结果寄存器，跳转地址是进入语句块的标签，并且将该指令的标签设置为 while 判断的标签。
5. 添加一条无条件跳转指令，跳转到跳过语句块的标签。
6. 将 S 的中间代码片段的第一个指令打上进入语句块的标签，并将其添加到当前节点的中间代码片段列表中。
7. 添加一条无条件跳转指令，跳转到 while 判断的标签。

8. 添加一条空指令，并将其打上跳过语句块的标签。

如果为 return E:

1. 将 E 的中间代码列表添加到当前节点的中间代码片段列表中。
2. 获取 E 的结果寄存器。
3. 添加一条寄存器存储指令，将 E 的结果寄存器的值（或是 E 的 value）存储到 R 寄存器中（如果有返回值）。

如果为 {S'}:

将 S' 的中间代码片段添加到当前节点的中间代码片段列表中。（注意在这个文法中，该候选式不会产生新作用域）

如果为 d(R'):

0. 将 R' 节点的中间代码片段添加到当前节点的中间代码片段列表中。
1. 调用逻辑环境模拟器，将所有当前作用域的寄存器内容保存到栈中（T 和 RA 寄存器），将其生成的代码片段添加到当前节点的中间代码片段列表中。
2. 从 R' 节点中对应的实参列表中获取相关寄存器地址（如果是 E，直接获取其寄存器/值，如果是列表，在符号表——寄存器映射中查找），将这些值按照参数列表顺序依次写入到 R 寄存器中。
3. 获取当前作用域的一个新临时标签，作为函数调用的返回地址。
4. 添加一条寄存器赋值指令，将 RA 寄存器的值设置刚刚获取的新临时标签。
5. 在符号表中查找函数名对应的函数入口地址，并添加一条 GOTO 跳转指令，跳转到该函数入口地址。
6. 添加一条空指令，将其标签设置为函数调用的返回地址。
7. 调用逻辑环境模拟器，将相关寄存器内容从栈中恢复到 T 寄存器中（如果有返回值，将 R 寄存器的值恢复到 T 寄存器中），将其生成的代码片段添加到当前节点的中间代码片段列表中。

E 节点

$E \rightarrow \text{num} \mid \text{flo} \mid d \mid d[E] \mid E + E \mid E * E \mid (E) \mid d(R')$ # 常规算数表达式

如果为 num 或 flo:

1. 获取当前节点的值（num 或 flo）。
2. 获取一个新的 T 寄存器。
3. 添加一条寄存器赋值指令，将 T 寄存器的值设置为当前节点的值。

4. 将这条指令设置为当前节点的中间代码片段。将刚刚获取的寄存器设置为结果寄存器。

如果为 d :

不需要生成中间代码，直接从符号表中获取变量的寄存器的地址，并将其设置为结果寄存器。

如果为 $d[E]$:

1. 获取第三个子节点——第一个 E 的代码列表，添加到当前节点的中间代码片段列表中。
2. 获取列表在当前作用域的内存地址和第三个子节点的结果寄存器。
3. 添加一条 MUL 指令，将结果寄存器的值乘以数组元素大小，得到数组元素的内存地址偏移量。
4. 添加一条 ADD 指令，将数组基地址和偏移量相加，得到最终的数组元素内存地址。
5. 获取一个新的 T 寄存器作为结果寄存器。
6. 添加一条寄存器加载指令，从数组元素的内存地址加载值到结果寄存器。将该指令添加到当前节点的中间代码片段列表中。

如果为 $E + E$ 或 $E * E$:

1. 获取第一个子节点的中间代码片段列表，添加到当前节点的中间代码片段列表中。
2. 获取第三个子节点的中间代码片段列表，添加到当前节点的中间代码片段列表中。
3. 获取第一个子节点的结果寄存器和第三个子节点的结果寄存器。
4. 获取一个新的 T 寄存器作为结果寄存器。
5. 如果是 $E + E$ ，添加一条寄存器相加指令，将第一个子节点的结果寄存器和第三个子节点的结果寄存器相加，并将结果存储到结果寄存器中。
6. 如果是 $E * E$ ，添加一条寄存器相乘指令，将第一个子节点的结果寄存器和第三个子节点的结果寄存器相乘，并将结果存储到结果寄存器中。
7. 将结果寄存器设置为当前节点的结果寄存器。
8. 将刚刚生成的指令添加到当前节点的中间代码片段列表中。

如果为 (E) :

只需要传递子节点的中间代码片段列表和结果寄存器即可。

如果为 $d(R')$:

0. 将 R' 节点的中间代码片段添加到当前节点的中间代码片段列表中。

1. 调用逻辑环境模拟器，将所有当前作用域的寄存器内容保存到栈中（T 和 RA 寄存器），将其生成的代码片段添加到当前节点的中间代码片段列表中。
2. 从 R' 节点中对应的实参列表中获取相关寄存器地址（如果是 E，直接获取其寄存器/值，如果是列表，在符号表——寄存器映射中查找），将这些值按照参数列表顺序依次写入到 R 寄存器中。
3. 获取当前作用域的一个新临时标签，作为函数调用的返回地址。
4. 添加一条寄存器赋值指令，将 RA 寄存器的值设置刚刚获取的新临时标签。
5. 在符号表中查找函数名对应的函数入口地址，并添加一条 GOTO 跳转指令，跳转到该函数入口地址。
6. 添加一条空指令，将其标签设置为函数调用的返回地址。
7. 调用逻辑环境模拟器，将相关寄存器内容从栈中恢复到 T 寄存器中（如果有返回值，将 R 寄存器的值恢复到 T 寄存器中），将其生成的代码片段添加到当前节点的中间代码片段列表中。
8. 将 R 寄存器的值设置为当前节点的结果寄存器。

R' 节点

$R' \rightarrow \text{epsilon} \mid R' R, \# \text{ 实参表}$

只需要汇总所有子节点的中间代码片段即可。其语义动作已经构建了通向各个具体实参的指针。

如果为空，则初始化中间代码片段列表。

如果为 $R' R$ ，则先复制 R' 的中间代码片段列表，然后将 R 的中间代码片段添加到列表末尾。

R 节点

$R \rightarrow E \mid d[] \# \text{ 单个实参，分别是表达式运算结果、数组和函数调用}$

如果为 E：

1. 获取 E 的中间代码片段列表，添加到当前节点的中间代码片段列表中。
2. 获取 E 的结果寄存器，并将其设置为当前节点的结果寄存器。

如果为 $d[]$ ：

1. 获取 d 的内存基地址
2. 获取一个新的 T 寄存器作为结果寄存器。

3. 添加一条寄存器赋值指令，将 d 的内存基地址加载到结果寄存器中。
4. 将当前作用域内 d 和 T 寄存器通过逻辑环境模拟器注册。

1.3.3 中间代码生成工具设计

中间代码生成工具主要包括以下几个部分：

- 生成逻辑环境模拟器：

参见： `include/syntax_semantic_analyzer/logical_env_simulator.h`

- 寄存器管理：维护 T 寄存器和 R 寄存器的使用情况，处理寄存器的加载、存储、恢复等操作。这其中包含寄存器和作用域的关系，例如每个作用域内的寄存器使用情况。

- 内存空间管理：维护程序代码段、数据段和栈段的地址分配和使用情况。

- 变量和函数管理：维护符号表中变量和函数的信息，比如其对应的寄存器、内存地址等。

- 中间代码生成器：

参见： `include/syntax_semantic_analyzer/interm_code_generator.h`

- 遍历 AST 树：使用深度优先遍历（DFS）算法遍历 AST 树，生成中间代码。

- 生成中间代码：根据 AST 节点的类型和属性，生成对应的中间代码指令，并利用生成的逻辑环境模拟器进行寄存器和内存的操作。

- 输出结果：将生成的中间代码指令输出到指定格式（如文本文件或其他数据结构）。

2 实验结果

从实验 1 到实验 6 总共实现 62 个测试。

所有测试： `test/`

仅针对语法语义分析/中间代码生成的集成测试：

`test/syntax_semantic_analyzer/syntax_semantic_analyzer_tests.cpp`

仅针对语法语义分析/中间代码生成的集成测试数据：

`test/data/syntax_semantic_analyzer/syntax_semantic_analyzer`

选取有代表性的结果展示中间代码生成。

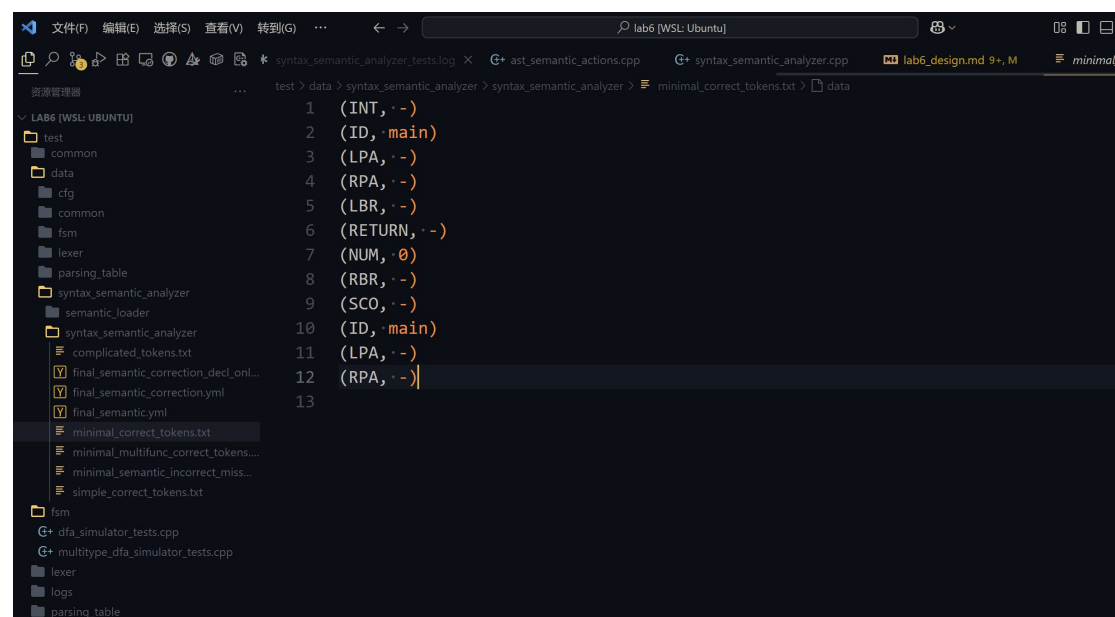
2.1 复现说明

1. LAB6 新建 build 文件夹
2. 安装 CMakeList 所需依赖到本地路径，确保 Cmake 可访问（GTEST、SPDLOG、YAML-CPP、TABULATE、BOOST GRAPH LIBRARY）
3. 在 build 文件夹执行 cmake ..
4. 在 build 文件夹执行 make
5. 在 build 文件夹运行 test_all，所有测试 log 和结果（语法树可视化、符号表、中间代码生成）均出现在当前目录

2.2 最小 DEMO

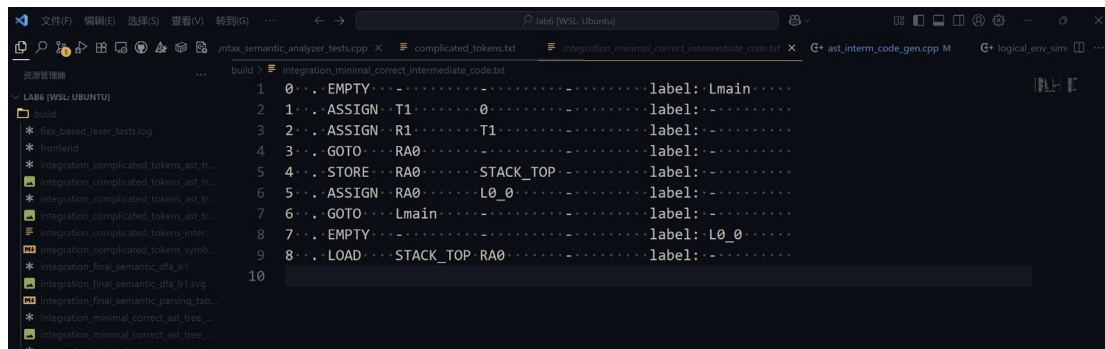
代码：int main(){return 0}; main()

对应 Token：



```
1 (INT, --)
2 (ID, main)
3 (LPA, --)
4 (RPA, --)
5 (LBR, --)
6 (RETURN, --)
7 (NUM, 0)
8 (RBR, --)
9 (SCO, --)
10 (ID, main)
11 (LPA, --)
12 (RPA, --)
13
```

该 DEMO 同样出现在上衣实验测试中，语法树、符号表详见实验五报告。直接给出中间代码生成结果：



```
1 0... EMPTY...label: Lmain...
2 1... ASSIGN...T1...0...label: ...
3 2... ASSIGN...R1...T1...label: ...
4 3... GOTO...RA0...label: ...
5 4... STORE...RA0...STACK_TOP...label: ...
6 5... ASSIGN...RA0...L0_0...label: ...
7 6... GOTO...Lmain...label: ...
8 7... EMPTY...label: L0_0...
9 8... LOAD...STACK_TOP...RA0...label: ...
10
```

可见：

- 首行声明 main 函数，通过标签 Lmain 定位。
- 0 作为常量使用立即数加载到 T1
- REUTRN0 通过 T1 赋值专用传递参数的 R1 寄存器实现
- 返回通过 GOTO 配合专门放置返回地址的 RA 寄存器实现
- 编号为 4 的语句是程序开始，对应 main()，此时发现是一个 STAT_FUNC_CALL 节点，因此首先分配返回地址标签 L0_0（含义为作用域 0 的第 0 个标签），将其存入 RA0。之后进入状态保护流程，检查当前作用域寄存器使用情况，发现只有 RA0 被使用，因此 RA 入栈。
- 跳转使用 GOTO Lmain 完成。
- L0_0 被分配到编号 7 的空语句，紧接着 GOTO 之后，用于定位返回地址。接着进入状态恢复阶段，发现要恢复的只有 RA 寄存器，因此使用 LOAD 寄存器将 RA 从栈中恢复。此时程序结束，main 返回值存储在 R1。

2.3 多函数跳转 DEMO

一个略微复杂一些的例子，代码：int func1(){return 0}; int main(){return func1();}
main()

对应 Token：

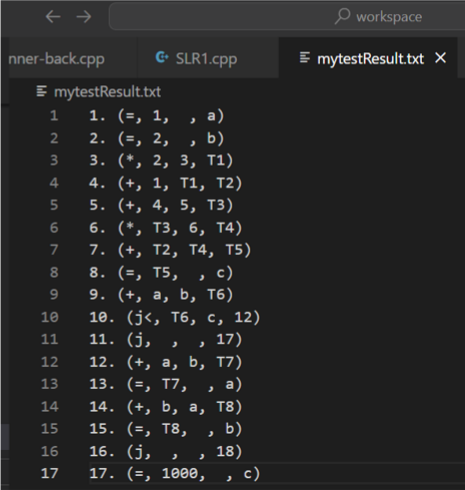
二、实验步骤和结果

实验六输入

```
int main ()
{
    a=1;
    b=2;
    c=1+2*3+(4+5)*6;

    if a+b<c
    {
        a=a+b;
        b=b+a
    }
    else
        c=1000;
```

实验六输出



```
1 1. (=, 1, , a)
2 2. (=, 2, , b)
3 3. (*, 2, 3, T1)
4 4. (+, 1, T1, T2)
5 5. (+, 4, 5, T3)
6 6. (*, T3, 6, T4)
7 7. (+, T2, T4, T5)
8 8. (=, T5, , c)
9 9. (+, a, b, T6)
10 10. (j<, T6, c, 12)
11 11. (j, , , 17)
12 12. (+, a, b, T7)
13 13. (=, T7, , a)
14 14. (+, b, a, T8)
15 15. (=, T8, , b)
16 16. (j, , , 18)
17 17. (=, 1000, , c)
```

但是该程序的语法和语义均是不正确的。比如语句表 S'要求最后一条语句后不能加分号、每个函数均要有返回值、每个变量使用前都应该声明等。因此将其修改为正确的形式：

```
int main() {
    int a;
    int b;
    int c;
    a = 1;
    b = 2;
    c = 1 + 2 * 3 + (4 + 5) * 6;
    if (a + b < c)
    {
        a = a + b
    };
    return c
};
main()
```

对应 Token 过长，给出生成结果。

符号表：

文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) ...

lab6 [WSL: Ubuntu]

integration_complicated_tokens_symbol_table.md 3

build > integration_complicated_tokens_symbol_table.md > # Symbol Table

1 # Symbol Table

2 | Symbol Name | Data Type | Kind | Scope ID | Memory Size (bytes) | Other Attributes |

3 |-----|-----|-----|-----|-----|-----|

4 | main | INT | Function | 0 | 0 | Args: [], BodyScopeID: 1 |

5 | a | INT | Variable | 1 | 4 | |

6 | b | INT | Variable | 1 | 4 | |

7 | c | INT | Variable | 1 | 4 | |

8

9

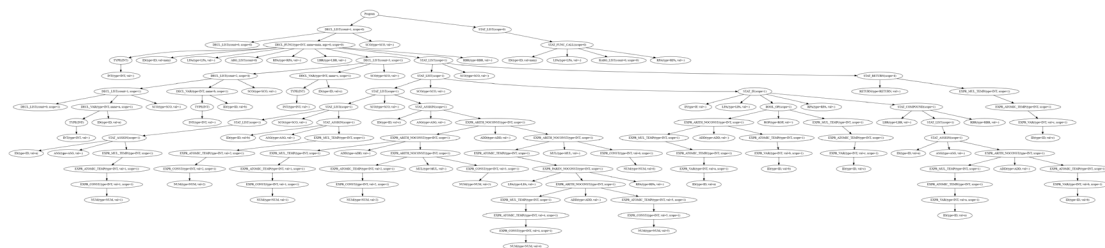
预览 integration_complicated_tokens_symbol_table.md

Symbol Table

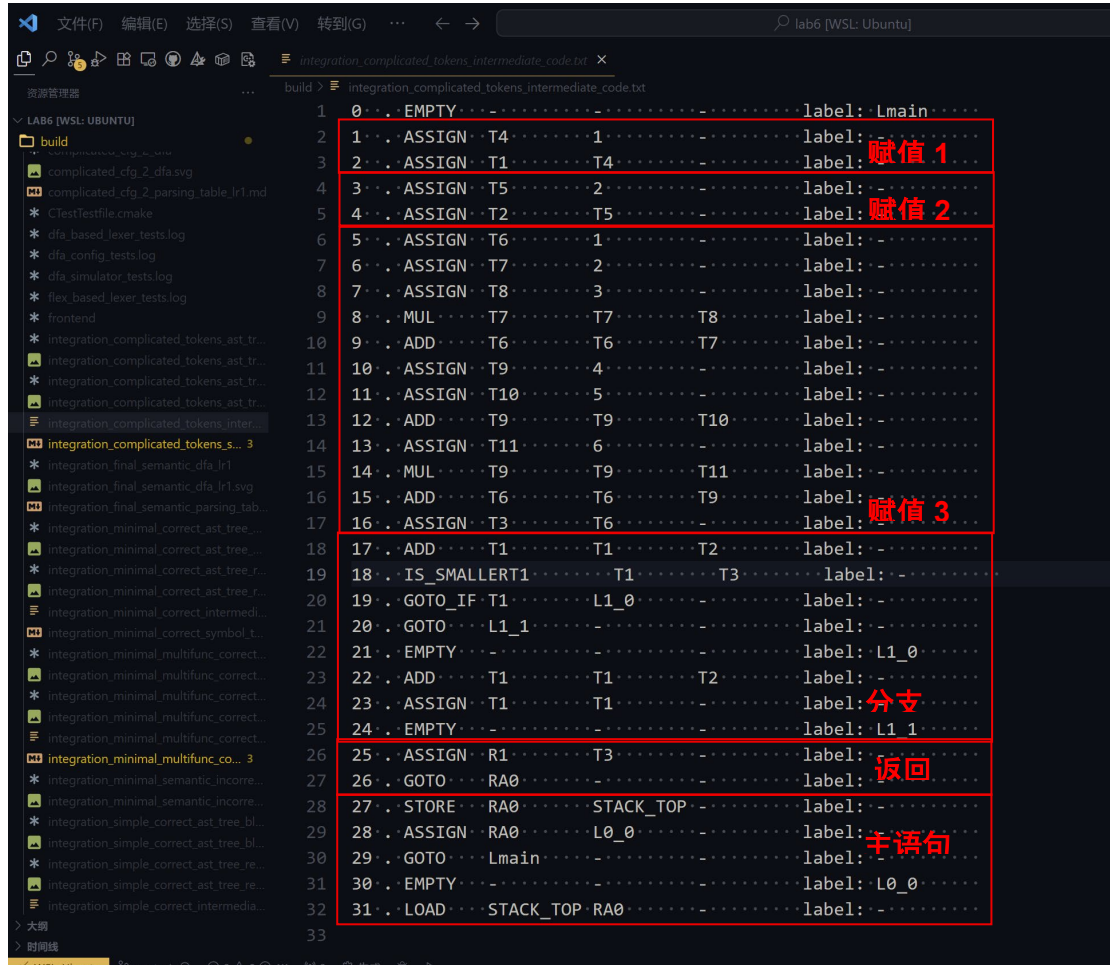
Symbol Name	Data Type	Kind	Scope ID	Memory Size (bytes)	Other Attributes
main	INT	Function	0	0	Args: [], BodyScopeID: 1
a	INT	Variable	1	4	
b	INT	Variable	1	4	
c	INT	Variable	1	4	

可见 a、b、c 均属于 main 的函数体作用域 1。

AST 树：（矢量图可无限放大）



代码生成结果：



The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project named 'LAB6 [WSL: UBUNTU]' with a 'build' directory containing various files. The code editor shows the file 'integration_complicated_tokens_intermediate_code.txt' with the following assembly code:

```
1 0 . . EMPTY . . . . . label: Lmain . . . . .
2 1 . . ASSIGN . T4 . . . . . 1 . . . . . label: 赋值 1 . . . . .
3 2 . . ASSIGN . T1 . . . . . T4 . . . . . label: 赋值 2 . . . . .
4 3 . . ASSIGN . T5 . . . . . 2 . . . . . label: 赋值 2 . . . . .
5 4 . . ASSIGN . T2 . . . . . T5 . . . . . label: 赋值 2 . . . . .
6 5 . . ASSIGN . T6 . . . . . 1 . . . . . label: 赋值 2 . . . . .
7 6 . . ASSIGN . T7 . . . . . 2 . . . . . label: 赋值 2 . . . . .
8 7 . . ASSIGN . T8 . . . . . 3 . . . . . label: 赋值 2 . . . . .
9 8 . . MUL . . . T7 . . . . . T7 . . . . . T8 . . . . . label: 赋值 2 . . . . .
10 9 . . ADD . . . T6 . . . . . T6 . . . . . T7 . . . . . label: 赋值 2 . . . . .
11 10 . . ASSIGN . T9 . . . . . 4 . . . . . label: 赋值 2 . . . . .
12 11 . . ASSIGN . T10 . . . . . 5 . . . . . label: 赋值 2 . . . . .
13 12 . . ADD . . . T9 . . . . . T9 . . . . . T10 . . . . . label: 赋值 2 . . . . .
14 13 . . ASSIGN . T11 . . . . . 6 . . . . . label: 赋值 2 . . . . .
15 14 . . MUL . . . T9 . . . . . T9 . . . . . T11 . . . . . label: 赋值 2 . . . . .
16 15 . . ADD . . . T6 . . . . . T6 . . . . . T9 . . . . . label: 赋值 2 . . . . .
17 16 . . ASSIGN . T3 . . . . . T6 . . . . . label: 赋值 3 . . . . .
18 17 . . ADD . . . T1 . . . . . T1 . . . . . T2 . . . . . label: 赋值 3 . . . . .
19 18 . . IS_SMALLER . T1 . . . . . T1 . . . . . T3 . . . . . label: 赋值 3 . . . . .
20 19 . . GOTO_IF . T1 . . . . . L1_0 . . . . . label: 赋值 3 . . . . .
21 20 . . GOTO . . . L1_1 . . . . . label: 赋值 3 . . . . .
22 21 . . EMPTY . . . . . label: L1_0 . . . . .
23 22 . . ADD . . . T1 . . . . . T1 . . . . . T2 . . . . . label: 赋值 3 . . . . .
24 23 . . ASSIGN . T1 . . . . . T1 . . . . . label: 分支 . . . . .
25 24 . . EMPTY . . . . . label: L1_1 . . . . .
26 25 . . ASSIGN . R1 . . . . . T3 . . . . . label: 返回 . . . . .
27 26 . . GOTO . . . RA0 . . . . . label: 返回 . . . . .
28 27 . . STORE . . . RA0 . . . . . STACK_TOP . . . . . label: 返回 . . . . .
29 28 . . ASSIGN . RA0 . . . . . L0_0 . . . . . label: 主语句 . . . . .
30 29 . . GOTO . . . Lmain . . . . . label: 主语句 . . . . .
31 30 . . EMPTY . . . . . label: L0_0 . . . . .
32 31 . . LOAD . . . STACK_TOP . RA0 . . . . . label: 主语句 . . . . .
33
```

The code is annotated with red boxes and Chinese text: '赋值 1' (Assignment 1) for line 2, '赋值 2' (Assignment 2) for lines 3-16, '赋值 3' (Assignment 3) for lines 17-20, '分支' (Branch) for line 24, '返回' (Return) for lines 25-27, and '主语句' (Main statement) for lines 28-31.

其中赋值 1 对应 $a/b/c$ 值的计算。分支语句实现 IF 判断和跳转。返回语句将 C 从 T3 寄存器中存入 R1 作为返回值。主语句与上文相同。