《编译原理》课程项目

词义分析工具实验报告

205XXXX GTY

205XXXX CXX

205XXXX CWQ

指导教师：WZH 老师

2022 年 12 月

目录

一、实验内容

在 LR(1)分析方法得到的语法分析器基础上，编写一个类 C 语言的语义分析程序

1. 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）；

2. 注意静态语义错误的诊断和处理；

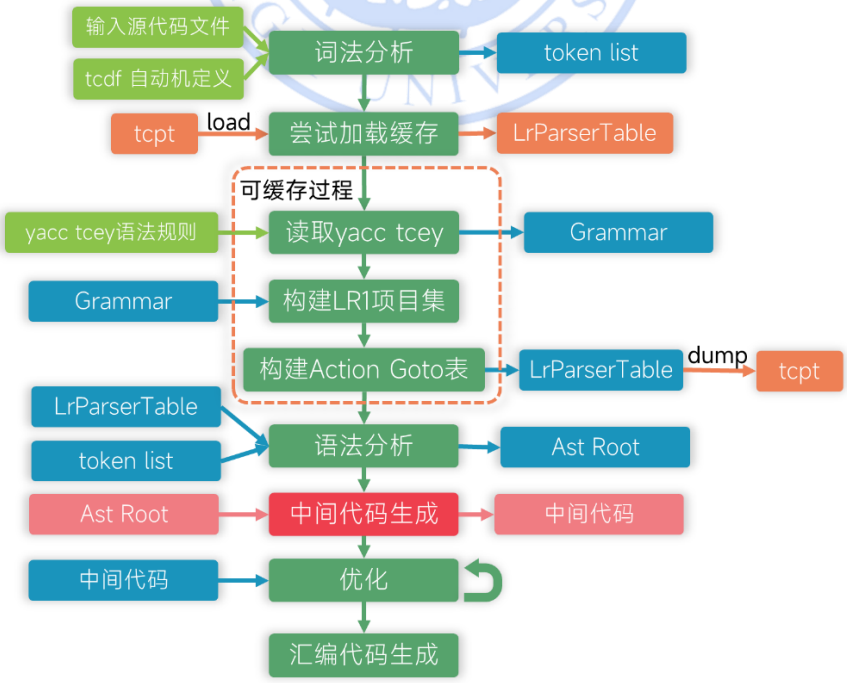
3. 在此基础上，考虑更为通行的高级语言的语义检查和中间代码生成所需要注意的内容，并给出解决方案。

二、需求分析

2.1 前阶段提供的基础

在此前的项目中，我们成功实现 C++11 语言词法分析，并按照 C99 文法 [4] 完成语法树构建（文法规则见附录 1），得到语法树的根节点。该节点类型为“翻译单元（translation unit）”，沿该节点遍历，可以得到整个源程序的结构，含每个符号的内容和位置，及它在语法中的含义。

2.2 本阶段在整个项目中的位置



上图为本项目的整体任务流程，需要对源代码文件进行词法分析，根据特定文法做语法分析，将语法树转为中间代码，在优化后转换成汇编代码。

本阶段在图中以茶花红颜色标注，接收语法分析阶段产生的语法树，生成中间代码，后者将在后续项目中经历多轮优化，最终转换为汇编代码。

2.3 本阶段任务

本阶段的输入内容是一棵语法树的根节点，输出目标是中间代码。由于我们采用的 C99 语法生成的语法树较为复杂，树上优化变得很困难，我们决定以尽量好的方式生成中间代码，旨在给优化器带来较少压力的同时，尽量避免在树上直接进行复杂的分析。

在经过对四元式的研究，我们认为，四元式表示高维数组访问的能力较弱，不易表示链接符号，且缺少对多种复杂数据类型的支持。对此，我们设计一种能力更强且依旧易用的中间代码（详见附录 2）。对于输入的语法树，中间代码生成器尝试将其转换为我们自己设计的中间代码，并在生成过程中及时报告遇到的错误，包括但不限于引用未定义的符号（变量、函数等），并报告重复定义等不严重但可能是隐患的错误，如代码块内符号与外部符号重名。

三、概要设计

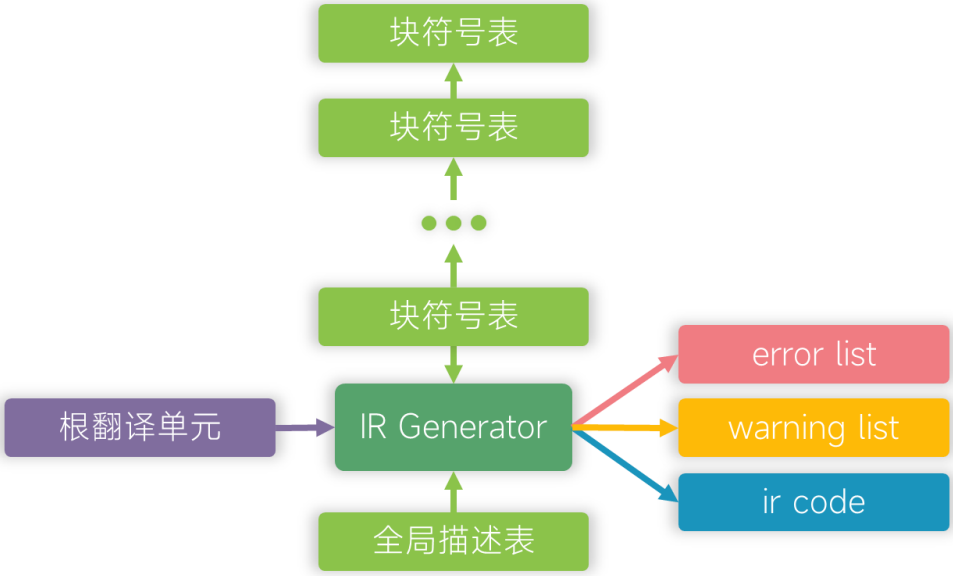
3.1 中间代码形式

经过对四元式的研究，我们认为，该形式在表示多维数组、函数调用、结构体和外部链接符号时，显得心有余而力不足。因此，我们自行设计一种三地址中间代码，我们称之为 tcir（详见附录 2）。

我们设计的中间代码格式更倾向于 x86 汇编，在其中引入待分配变量和虚拟寄存器等结构，在简洁明了的同时提供较好的可优化性。同时，tcir 内部保留内部导出和外部关联信息，并保存完整的块符号表信息，为后续优化器等模块提供冷加载中间代码并进行处理的能力，进一步解开模块之间的耦合。

本项目的中间代码生成器将以 tcir 为生成目标，分析语法树，输出中间代码，并在遇到错误时及时报告。

3.2 结构一览



中间代码生成器连接在语法分析器之后，输入内容为一个翻译单元节点。经过语法分析器的分析，我们可以认为，输入的节点引出的语法树严格遵循 C99 语法（若不，则程序已在此前步骤中输出报错信息，并提前结束）。

中间代码生成器内部维护全局描述表，便于提供全局函数和全局变量导入导出信息，供未来的链接器使用。同时，它管理一个“当前块符号表”结构。该结构内有一个指向上级符号表结构的指针，提供多层嵌套内定义同名变量和向外寻找已定义变量的能力。

中间代码生成器内部针对 C99 文法定义多个语法树节点处理模块，每个模块都十分简洁。对于叶节点（终结符）处理模块，进行简单处理后返回需要的结果；对于非叶节点（非终结符）处理模块，进行简单处理后，将需要的内容送入其他节点处理模块处理，并进行汇总处理。

3.3 整体工作流程

开始工作前，生成器先清空内部已有数据，防止多翻译模块分析时前一翻译模块留下的缓存对本翻译模块带来的干扰（实际测试时不涉及这种情况）。

由于输入的语法树节点由语法分析器精心准备，我们默认其已经遵循 C99 文法，且输入节点已经是“翻译单元（translation unit）”。我们将该节点送往翻译单元处理子模块进行处理，后者会根据情况递归处理语法树，最终在内部生成中间代码的二进制内容。

外部调用者在 IR 生成器工作完毕后，首先读取其中产生的报错信息，并告知用户。如果有“致命”级别报错，生成器内保存的中间代码是不可用的。只有当不存在错误，或仅存

在“警告”级别错误时，外部可以通过生成器提供的“导出”能力，获取以 ASCII 字符格式描述的中间代码内容。

四、详细设计

4.1 中间代码格式设计

为满足对符号链接、更多种数据类型等功能的需求，我们选择基于四元式的思想，自己设计一种中间语言格式，称为 tcir。

tcir 较为接近 x86 汇编，但保留了函数参数原始信息及变量原本信息，以便在后期实现不同方式的函数调用，且为优化器提供更多优化空间。

四元式中的临时计算结果存放于临时变量，需要依赖优化器转换为寄存器存储。在 tcir 中，我们设定 2 个虚拟寄存器，借助它们和虚拟栈即可描述几乎所有运算操作。同时，这样的设计让数据跟踪算法变得十分简单，减轻优化器设计压力。

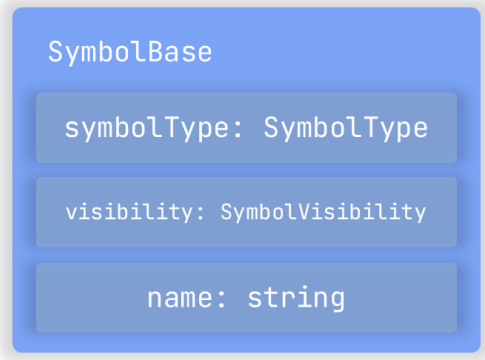
现代程序通常由多个模块组成，程序之间通过全局符号链接。为满足该需求，我们将全局符号信息直接登记到 tcir 中的指定区域内，便于生成汇编代码时标记导出导入信息。

tcir 中保留代码块的基本信息，有利于处理变量重复利用。

详细设计请看附录 2。

4.2 符号表及其他基本数据结构

4.2.1 符号基类



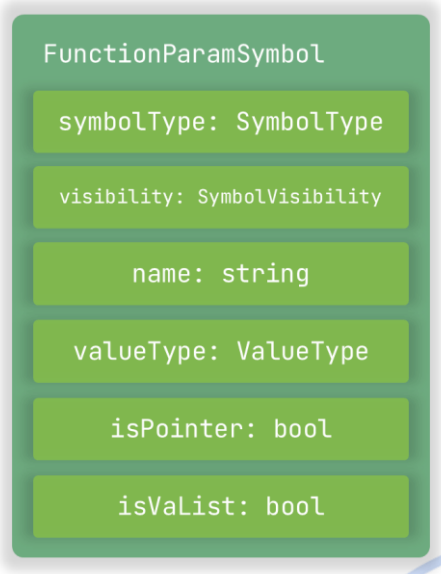
我们将程序内出现的函数、局部变量、全局变量、函数参数皆视为“符号”。他们公有两个属性：名称、可见性。同时，考虑到 C++语言没有运行时反射机制，为区分不同符号类型，额外加入“符号类型”枚举值。

可见性分为内部和外部。内部可见的符号不会登记到中间代码的“导入导出符号”区，无法被其他程序链接使用；外部符号会被登记到该区域，可以被其他程序链接使用。当然，它也可以登记希望从其他程序链接使用的代码。

符号名称的保留对全局符号是至关重要的，链接器的工作十分依赖它。

我们将上述三个值组合在一起，构成“符号基类”，其他符号类皆自它派生而来。

4.2.2 函数参数符号



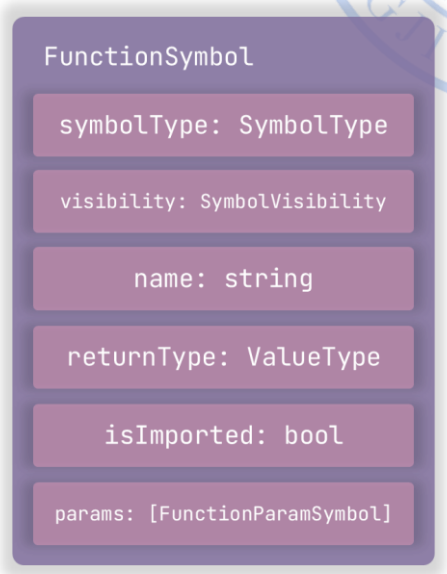
函数参数符号保存函数参数信息。每个函数符号内需要登记一个函数参数符号表。

我们在符号基类的基础上，添加“数据类型”、“是否为指针”、“是否为变长参数标记”信息。

“数据类型”目前设置有 8 位、16 位、32 位整数，皆分为有符号和无符号。此外，还有“空”，即为 void。若“是否为指针”被标为“是”，表示该参数是一个指针，其对应的可能是数组。当然，目前的设计暂时没有处理高纬数组的问题，但基于该结构，只需加入纬度向量即可实现。

若“变长参数”被设为“是”，则表明从该位置开始，后续可能跟随 0 个至多个参数，需要通过其他手段处理。由于变长参数往往都在栈上传递，处理时一般通过操作系统提供的库函数实现，编译器项目不需要过多纠结该问题。

4.2.3 函数符号

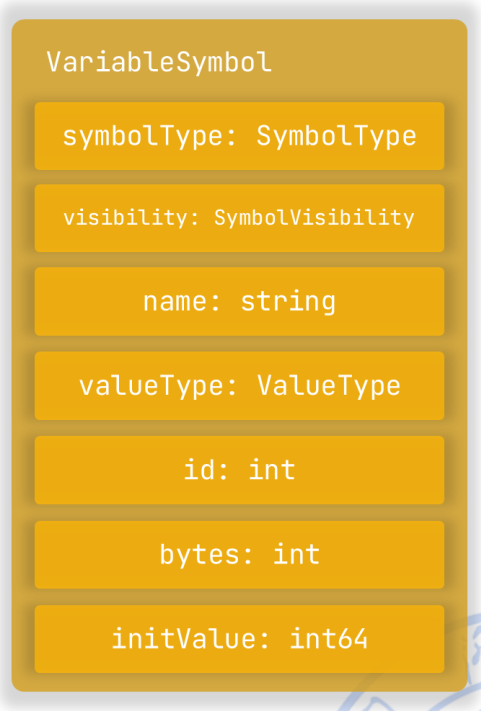


函数符号用于表示程序内定义的函数。其在符号基类的基础上，添加返回类型信息，以便生成汇编代码时处理返回信息寄存器（如 x86 汇编中的 eax 寄存器）。

真实环境下，函数很可能是从其它可重定位文件关联得到的。isImported 标记该函数是否为外部导入函数，即是否是从外部链接得到的。如果是，则需要在汇编代码内添加关联标记，如 extern。

此外，函数符号内还需要维护一个参数表，表中每个元素都是一个“函数参数符号”。对函数参数信息的保留有利于灵活处理参数在栈上的分配方式。

4.2.4 变量符号



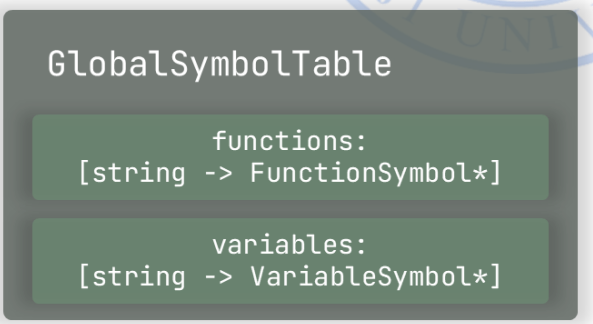
“变量符号”指复合语句内定义的变量和及全局范围内定义的符号。

由于复合语句内变量的名称并不重要，我们选择以 id 形式管理它们，在编译后不再存储其原始名称。由于“名称”保留在基础类内，变量符号结构虽然不一定需要，但只能皆包含它。对于不需要保留名称的变量，我们依旧在其中存入原始名称，即使该内容已经无用。

为满足对数组的处理，我们将字节数存入符号结构内，结合数据类型，可以推知其可能表示的一维或高维数组信息。

对于全局变量，由于其值需要在编译期得到，我们将计算得到的值存放到符号结构内。对于未赋值的全局变量，我们统一将其值设置为 0。若其赋值表达式明显无法在编译期得到，我们将输出一条错误信息，提示用户修改代码。

4.2.5 全局符号表

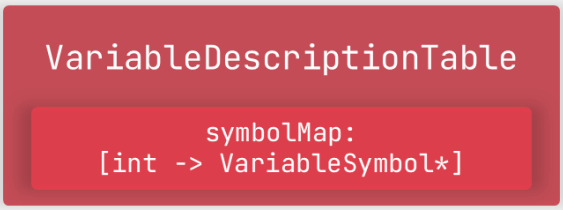


众所周知，我们的程序在大多数时候会作为一个更大的项目的一部分存在。我们需要调用其他模块的函数与变量，也可能需要将自己的函数和变量交给其他部分使用。对此，我们设计全局符号表结构。

全局符号表负责登记模块内所有导入导出符号信息。我们将模块内定义的函数和需要从外部导入的符号全部填写在该表内，通过“函数符号”内的成员值判断是否应该生成导入导出指令。对全局变量的处理类似。

按照设计，为便于管理不同作用域内同名变量，我们将块符号表内登记的变量符号同步设置到全局符号表内。结构所用内存由全局符号表负责管理。

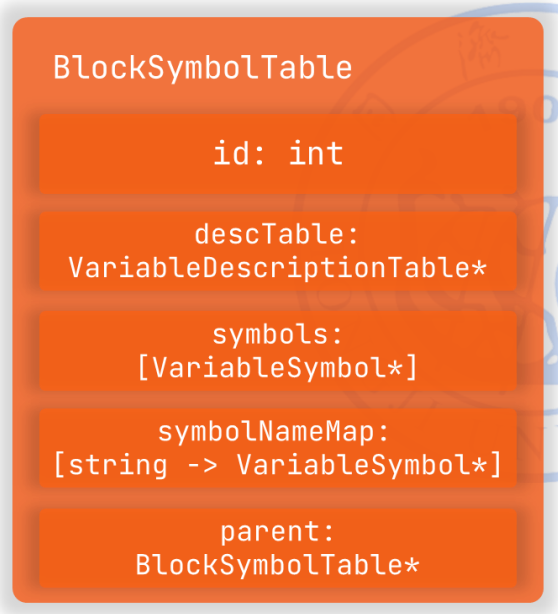
4.2.6 符号描述表



由于我们在代码块为每个原始名称不重要的变量分配唯一的 id，并在后期通过 id 管理它们，我们需要同步维护一个 id 到变量符号结构的映射，以便获取该 id 对应的变量具有的额外信息，如数据类型，字节宽度。这样的设计令我们的优化器可以获得关于变量的更多信息，以便分配内存和优化代码生成。

由于块符号表内的信息会同步登记到本表内，我们决定由本表负责变量符号结构的内存管理。

4.2.7 块符号表



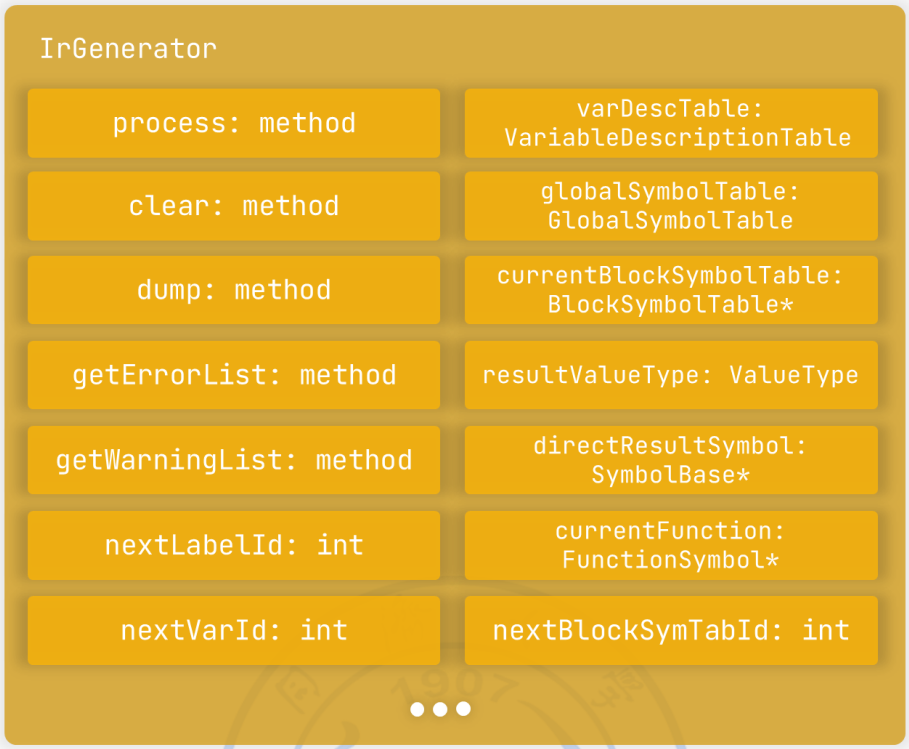
几乎所有人在编写代码时都要与代码块打交道。我们知道，代码块将代码分层，同层下往往不允许定义同名变量，但在不同级别下定义同名变量是允许的。同时，在子代码块内定义的变量，在代码块外不可使用。

为维护该逻辑，我们设置“块符号表”结构。该结构内含有指向其上级块符号表的指针，总体形成链式结构。分析器只需要抓住当前所在块的块符号表，后期可以通过递归查找以判断某变量是否定义，是在何处定义。

存入该表的符号都是局部变量，皆分配有 id。因此，在向该表登记符号时，会同步将符号登记到符号描述表内。

编译期间，我们从代码文件内得到的都是变量名，而不是内部分配的 id。为加速查找过程，我们额外添加一个变量名到变量符号结构的映射，可将缩小查找目标的时间复杂度从线性降低到对数级。

4.3 中间代码生成器总体工作流程



本项目的核心部分是“中间代码生成器”。我们在前文中已经描述，生成器的输入是语法树根节点，输出是错误列表和中间代码。

中间代码生成器对外提供六个方法：

· 处理：传入语法树根节点，令生成器从该结点开始分析代码，寻找代码中的错误，并在一切就绪后准备好中间代码；

· 清空：清除生成器内暂存的分析结果。该结果可能来自上一次分析；

· 导出：以字符串形式导出中间代码，送入指定输出流中；

· 获取错误列表：获取上次处理完毕后，分析器内登记的错误信息表。错误信息含触发报错的语法树节点，和简短的提示信息；

· 获取警告列表：类似获取错误列表。但是，错误列表中登记有内容意味着中间代码不可用。警告列表只起警告作用。有警告而无错误时，中间代码仍然可用，只是存在隐患。

此外，分析器内维护分析过程需要记录的辅助信息。我们需要记录当前正在处理的函数的函数符号，深入寻找符号时找到的结果（用于处理赋值表达式）。我们还需要记录一

个 continue 目标栈和一个 break 目标栈，以处理当遇到 break 或 continue 时，生成跳转到何处的语句，或者是报错。

当然，分析器内需要保存分析得到的中间内容。这些中间内容会在后续导出成中间代码。

分析器内部设有针对各个语法节点的处理模块。“处理”方法输入的语法树节点指向的一定是“翻译单元”，我们将其送入对应处理模块，后者会根据其子节点情况继续决定如何向下分发处理指令。

4.4 模块处理逻辑

说明：由于 C99 文法较为庞大，我们在本项目中专注实现基础架构，而非更完整的功能。因此，我们选择仅实现整数、void、循环等简单语句，移位等相对高级语句待后续完善。你可以在附录 1 看到 C99 文法。

4.4.1 翻译单元（translation unit）

```
translation_unit
    : external_declaration
    | translation_unit external_declaration
    ;
```

语法树的根节点一定是一个翻译单元。

如果它只有一个子节点，直接按照外部定义处理规则处理其孩子即可。否则，将其第一个孩子递归送入本模块，再将第二个孩子送入外部定义处理模块。

4.4.2 外部定义（external declaration）

```
external_declaration
    : function_definition
    | declaration
    ;
```

根据孩子类型，分别送入两个处理模块即可。

4.4.3 函数定义（function declaration）

```
function_definition
    : declaration_specifiers declarator declaration_list compound_statement
    | declaration_specifiers declarator compound_statement
    ;
```

我们选择不支持第一种语法。

对于第二种语法，根据 declaration specifiers 解析其函数返回类型等信息，再根据 declarator 获取函数名和参数表。基于这些信息，生成函数符号，绑定到生成器内“当前正在处理的函数”上。之后，将复合语句送入复合语句处理模块即可。

4.4.4 复合语句（compound statement）

```
compound_statement
    : '{' '}'
    | '{' block_item_list '}'
    ;
```

复合语句即代码块。

若其只有 2 个孩子，即为空代码块，不执行任何动作。

若其拥有三个孩子，为其生成一张块符号表，新表的父表设为分析器当前直接管理的表。若分析器目前不管理任何表，新表的父表设为自己。将新表绑定为分析器直接管理的表后，处理块项目列表即可。

4.4.5 块项目表（block item list）

```
block_item_list
    : block_item
    | block_item_list block_item
    ;
```

根据孩子数量的不同，分别选择递归调用自己和将孩子送入块项目处理模块的动作。

4.4.6 块项目（block item）

```
block_item
    : declaration
    | statement
    ;
```

根据孩子类型分别处理即可。

4.4.7 语句（statement）

```
statement
    : labeled_statement
    | compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    ;
```

根据孩子类型，分别送入不同的处理模块。我们暂不支持标签语句的处理。

4.4.8 选择语句（selection statement）

```
selection_statement
    : IF '(' expression ')' statement
    | IF '(' expression ')' statement ELSE statement
    | SWITCH '(' expression ')' statement
    ;
```

我们暂不支持 switch 语句的处理。

对于 if 语句，我们选择生成标签的方法，巧妙利用递归执行过程，躲避四元式中的回填过程。

对于不含 else 的 if 语句，我们生成一个“end”标签。在它前面，从前往后，我们依次分析 expression，令其生成代码，并在其后加入一个“je end”命令，实现 if 条件不满足时的跳出。跳出语句后，放置 statement 分析得到的代码。

对于含 else 的语句，我们生成“else”和“end”标签。语句生成顺序依次为：

· 判断条件 expression

· 0 号虚拟寄存器值为 0 时，跳转到 else 标签的语句

· 条件为 true 时执行的语句

· 跳转到 end 标签的语句

· else 标签

· 条件为 false 时执行的语句

· end 标签

当然，每个标签都以一个英文句号开头，汇编器会将这样的标签认为是内部标签。同时，标签名后跟随一个唯一编号，防止标签重名。

4.4.9 迭代语句（iteration statement）

```
iteration_statement
    : WHILE '(' expression ')' statement
    | DO statement WHILE '(' expression ')' ';'
    | FOR '(' expression_statement expression_statement ')' statement
    | FOR '(' expression_statement expression_statement expression ')' statement
    | FOR '(' declaration expression_statement ')' statement
    | FOR '(' declaration expression_statement expression ')' statement
    ;
```

处理迭代语句时，需要适当增减 break 目标栈和 continue 目标栈内的内容。标签在放置前提前创建，并在合适时机送入相应栈；语句处理完毕，将符号从栈内弹出。

4.4.10 do-while 循环语句

对于 do while 循环，按照以下方式生成语句：

· 创建 b 标签，设为 break 目标

· 处理 statement，生成代码

· 创建 c 标签，并设为 continue 目标

· 处理 expression，生成代码

· 创建语句：jne b

4.4.11 while 循环语句

对于 while 循环，按照以下方式生成语句：

· 创建 beg 标签，并设为 continue 目标

· 生成 expression 的代码

· 生成跳转语句：je end

· 生成 statement 的代码

· 生成跳转语句：j beg

· 创建 end 标签，并设为 break 目标

4.4.12 for 循环语句

对于 for 循环，我们直接认为其圆括号内有三个语句，按照以下方式生成语句：

· 处理圆括号内第一个语句

· 创建跳转指令：j e_two

· 创建标签 e_three，并设为 continue 目标

· 处理圆括号内第三条语句

· 创建标签 e_two

· 创建跳转语句：je end

· 处理 statement

· 创建跳转指令：j e_three

· 创建标签 end，并设为 break 目标

进入 for 循环处理逻辑前，外部根据语法树实际结构判断圆括号内三个元素分别是什么。当然，圆括号内第三个元素可以为空。由于前两个语句在语法内已经支持空指令，不用做特殊处理。

4.4.13 跳转语句（jump statement）

```
jump_statement
    : GOTO IDENTIFIER ';'
    | CONTINUE ';'
    | BREAK ';'
    | RETURN ';'
    | RETURN expression ';'
    ;
```

本部分较为简单。

与 return 有关的两条指令，需要生成一条 ret 指令。对于带表达式的返回语句，需要先进行表达式计算。由于表达式计算的结果默认存放于 0 号虚拟寄存器，此处不用做额外处理。

4.4.14 变量定义（declaration）

```
declaration
    : declaration_specifiers ';'
    | declaration_specifiers init_declarator_list ';'
    ;
```

首先解析限定符。之后，处理函数声明表。支持一组限定符定义多个变量。

特别注意，全局变量不能含有动态计算的部分，且部分细节处理与局部变量不同。后续相关模块皆需要接收“是否为全局范围”参数，以处理不同情况。

4.4.15 变量声明符（variable init declarator）

```
init_declarator
    : declarator
    | declarator '=' initializer
    ;
```

首先，处理 declarator，获取变量名信息。暂不支持函数和数组等复杂情况，但这些情况需要在下一阶段的项目中考虑。

如果没有 initializer，则给全局变量赋默认值 0，局部变脸不做赋值操作。

如果有 initializer，则调用其处理模块，并最终将处理结果移动到虚拟变量内。注意，当变量在全局时，得到的必须是常数，该值直接登记到变量符号结构的“初始值”内。

4.4.16 赋值表达式（assigment expression）

```
assignment_expression
    : conditional_expression
    | unary_expression assignment_operator assignment_expression
    ;
```

该表达式有两种类型。其中，带单个子节点的只负责计算，带三个子节点的需要产生新变量，或改变已有变量的值。对于第一种情况，直接调用相应处理模块即可；对第二种情况，先处理单目运算，得到值改变目标，再处理第三个子节点，得到值，最后根据赋值运算符判断需要做何种处理。

4.4.17 条件表达式（conditional expression）

```
conditional_expression
    : logical_or_expression
    | logical_or_expression '?' expression ':' conditional_expression
    ;
```

节点只有一个子节点时，调用相应处理模块即可。后续多个表达式皆包含相似情况，接按照相似方式处理。

节点具有三个子节点时，先处理逻辑或表达式，根据其结果分别执行“表达式”或“条件表达式”。

4.4.18 逻辑或表达式（logical or expression）

```
logical_or_expression
    : logical_and_expression
    | logical_or_expression OR_OP logical_and_expression
    ;
```

较简单。注意，当逻辑或运算符前的表达式结果为 true 时，后方表达式不应被执行。

4.4.19 逻辑与表达式（logical and expression）

```
logical_and_expression
    : inclusive_or_expression
    | logical_and_expression AND_OP inclusive_or_expression
    ;
```

较简单。注意，当逻辑与运算符前的表达式结果为 false 时，其后的表达式不应被计算。

4.4.20 或表达式（inclusive or expression）

```
inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression '|' exclusive_or_expression
    ;
```

较简单。注意，或运算符不应做短路处理。

4.4.21 异或表达式（exclusive or expression）

```
exclusive_or_expression
    : and_expression
    | exclusive_or_expression '^' and_expression
    ;
```

较简单。计算时，前一表达式的结果可以先存入栈，待后一表达式计算完毕，从栈中取出结果，再做计算。这样产生的代码效率可能较差，但优化器可以很方便地改进它。

4.4.22 与表达式（and expression）

```
and_expression
    : equality_expression
    | and_expression '&' equality_expression
    ;
```

同样，与表达式不应考虑短路运算问题。

4.4.23 等式表达式（equality expression）

```
equality_expression
    : relational_expression
    | equality_expression EQ_OP relational_expression
    | equality_expression NE_OP relational_expression
    ;
```

该表达式需要区分等于和不等于运算符，做不同处理。

4.4.24 关系表达式（relational expression）

```
relational_expression
    : shift_expression
    | relational_expression '<' shift_expression
    | relational_expression '>' shift_expression
    | relational_expression LE_OP shift_expression
    | relational_expression GE_OP shift_expression
    ;
```

该表达式需要判断较多运算符号。

4.4.25 移位表达式（shift expression）

```
shift_expression
    : additive_expression
    | shift_expression LEFT_OP additive_expression
    | shift_expression RIGHT_OP additive_expression
    ;
```

移位运算。

4.4.26 加法表达式（additive expression）

```
additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;
```

含加法和减法。

4.4.27 乘法表达式（multiplicative expression）

```
multiplicative_expression
    : cast_expression
    | multiplicative_expression '*' cast_expression
    | multiplicative_expression '/' cast_expression
    | multiplicative_expression '%' cast_expression
    ;
```

含除法和取模运算。

4.4.28 类型转换表达式（cast expression）

```
cast_expression
    : unary_expression
    | '(' type_name ')' cast_expression
    ;
```

该部分涉及数据类型转换，实际情况会很复杂，我们选择暂不支持，只处理节点含一个子节点的情况。

4.4.29 表达式语句（expression statement）

```
expression_statement
    : ';'
    | expression ';'
    ;
```

节点只有一个子节点时，不做任何操作；否则，将第一个子节点送入表达式处理模块处理。

4.4.30 表达式（expression）

```
expression
    : assignment_expression
    | expression ',' assignment_expression
    ;
```

注意，逗号表达式的每一个表达式都需要经过计算，但整体的结果为最后一个表达式的结果。

4.4.31 单目表达式（unary expression）

```
unary_expression
    : postfix_expression
    | INC_OP unary_expression
    | DEC_OP unary_expression
    | unary_operator cast_expression
    | SIZEOF unary_expression
    | SIZEOF '(' type_name ')'
    ;
```

注意，前缀自增和自减符号需要改变后方的单目表达式处理得到的变量。如果后方表达式处理得到的不是可变变量，需要报错。

4.4.32 后缀表达式（postfix expression）

```
postfix_expression
    : primary_expression
    | postfix_expression '[' expression ']'
    | postfix_expression '(' ')'
    | postfix_expression '(' argument_expression_list ')'
    | postfix_expression '.' IDENTIFIER
    | postfix_expression PTR_OP IDENTIFIER
    | postfix_expression INC_OP
    | postfix_expression DEC_OP
    | '(' type_name ')' '{' initializer_list '}'
    | '(' type_name ')' '{' initializer_list ',' '}'
    ;
```

注意，后缀自增和自减表达式需要改变原变量，但返回的值是改变前的。

4.4.33 基础表达式（primary expression）

```
primary_expression
    : IDENTIFIER
    | CONSTANT
    | STRING_LITERAL
    | '(' expression ')'
    ;
```

该表达式内需要处理字符串常量问题，如登记到全局字符串表。同时，需要处理变量注册或查找的事情。

4.4.34 声明限定符（declaration specifiers）

```
declaration_specifiers
    : storage_class_specifier
    | storage_class_specifier declaration_specifiers
    | type_specifier
    | type_specifier declaration_specifiers
    | type_qualifier
    | type_qualifier declaration_specifiers
    ;
```

变量声明符号可能存在不同种类，如“const unsigned int”内包含三个符号。同一个符号也可能出现多次。同时，有可能有指针和函数的信息出现。情况过于复杂，该部分的处理模块难以知道外部想要什么，所以它的任务是提取所有符号，以列表形式返回，供外部选择如何处理。

五、调试分析

5.1 命令行使用说明

为引导用户使用，我们设计简单易懂的使用说明。

使用 -help 标签，即可看到使用说明：

```
flowerblack@ubuntu-mibookprox-gty:~/code-repo/ToyCompile/build$ ./ToyCompile sUniCli -help

ToyCompile Unified CommandLine

params:
  fname:[x]      : specify input file 'x'.
  help           : get help.

  dump-tokens    : dump tokens.

  rebuild-table  : reload parser table from tcey file.
                   if not set, parser would try to load cache
                   to improve performance.
  no-store-table : don't store built table to file.
  cache-table:[x]: specify cache table file.
  tcey:[x]       : set tcey file 'x'.
  dump-ast       : dump parser result.
  dot-file:[x]   : store parser result to file 'x'.

  dump-ir        : dump toycompile ir code.
  ir-to-file:[x] : store ir code to file.
  disable-color  : disable color to log output stream.

must have:
  fname:[x]

examples:
  ParserCli -fname:resources/test/easy.c.txt -rebuild-table

flowerblack@ubuntu-mibookprox-gty:~/code-repo/ToyCompile/build$
```

5.2 正确的样例：简单代码

```c
int a;

int f(int x) {
    int y = x + 1;
    return a;
}

```

使用下方命令运行：

```
./ToyCompile sUniCli -fname:resources/test/easy2.c -dump-ir
```

得到结果如下：

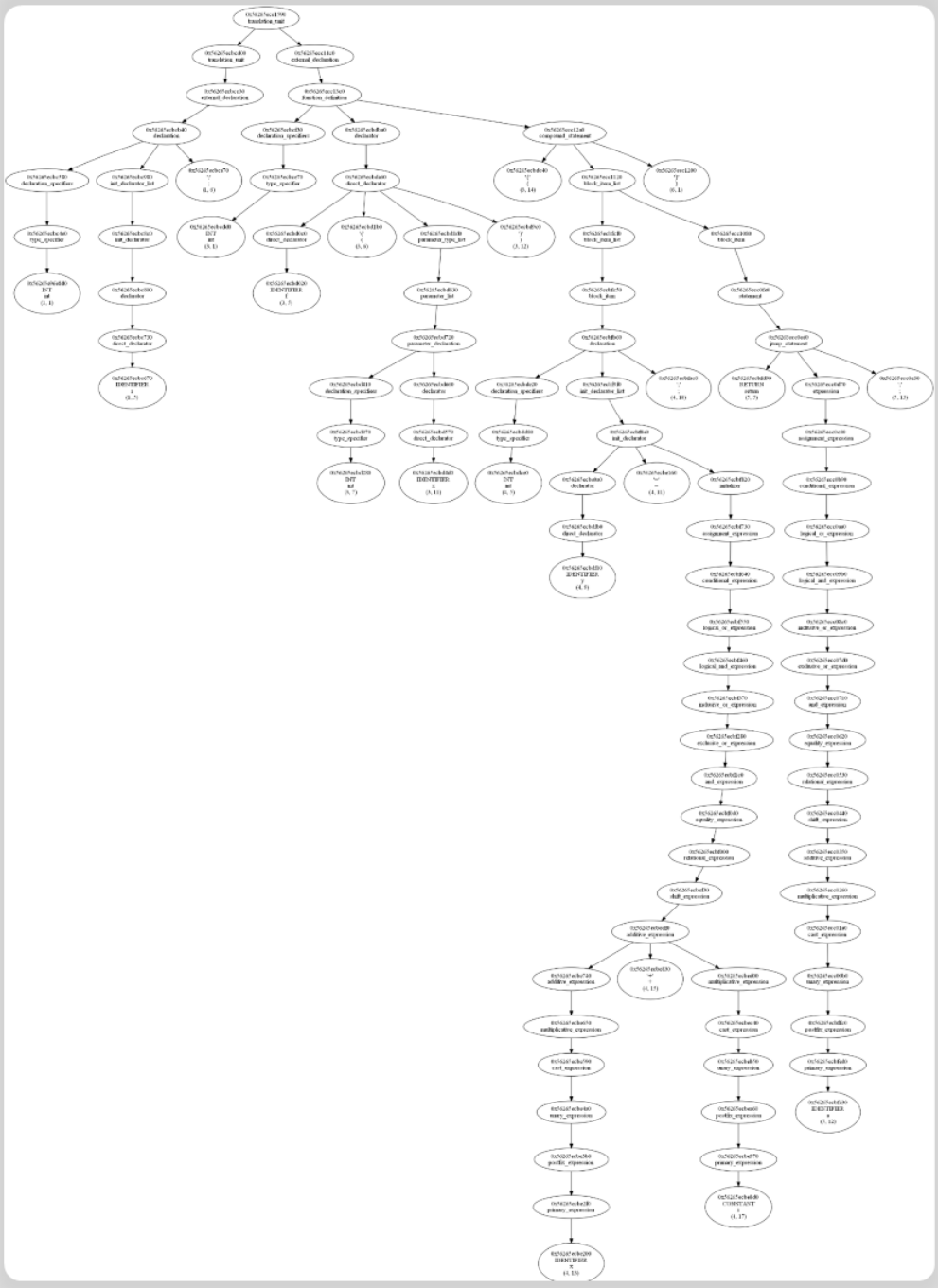
```
flowerblack@ubuntu-mibookprox-gty:~/code-repo/ToyCompile/build$
./ToyCompile sUniCli -fname:resources/test/easy2.c -dump-ir
@ begin of extlink
export f fun
export a var
@ end of extlink

@ begin of static-data
int var a s32 0
@ end of static-data

@ begin of global-symtab
fun visible f 1 s32
  s32 value x
var 1 y s32 4
@ end of global-symtab

@ begin of block-symtab
% begin
tab-id 1
parent-tab-id 1
var 1 y s32 4
% end

@ end of block-symtab

@ begin of instructions
label f
mov vreg 0 fval x
push 4 vreg 0
mov vreg 0 imm 1
pop 4 vreg 1
add vreg 1 vreg 0
xchg vreg 0 vreg 1
mov val 1 vreg 0
mov vreg 0 val a
ret
ret
@ end of instructions
```

粉色区域为符号导入导出信息。可以看到，中间代码内将函数 f 和变量 a 标记为导出变量，供链接器与外部程序链接。

紫色部分登记全局变量 a 的信息。由于源代码内没做赋值，我们给它设置默认值 0。

橙色部分为生成的中间代码，遵循我们设计的 tcir 格式。由于中间代码未被转换为汇编代码，无法与外部驱动程序链接运行，我们只能通过手算模拟运行。经手动验证，我们认为，该结果是正确的。不难注意到，该中间代码内存在部分效率较低的写法，这些内容有待后续优化器优化处理。

这段代码对应的语法树如下所示：



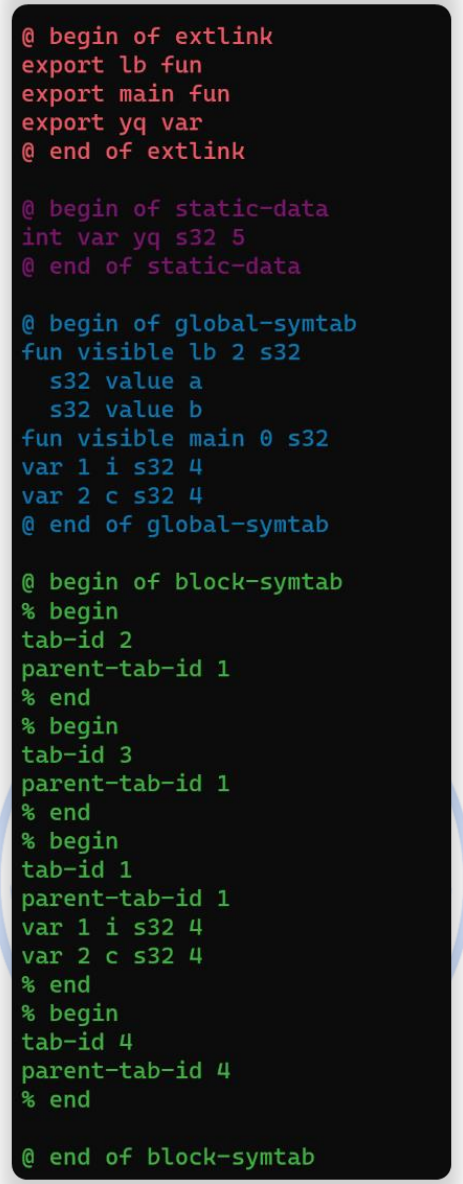
高清原图见：https://www.gardilily.com/files/compile-1.png

5.3 正确的样例：循环与选择

我们选择一个相对复杂的程序进行测试：

```
1
2 int yq = 3 + 2;
3
4 int lb(int a, int b) {
5
6     if (yq) {
7         yq = yq + 1;
8     }
9
10    int i;
11    for (i = 0; i < 10; i++) {
12        yq = yq + i;
13    }
14
15    int c = a + 2;
16    return c - b;
17 }
18
19 int main() {
20
21    return 0;
22 }
23
```

由于程序处理生成的中间代码内容较多，我们分段看。除直接指引工作的代码外，生成的辅助内容如下：



```
@ begin of extlink
export lb fun
export main fun
export yq var
@ end of extlink

@ begin of static-data
int var yq s32 5
@ end of static-data

@ begin of global-symtab
fun visible lb 2 s32
  s32 value a
  s32 value b
fun visible main 0 s32
var 1 i s32 4
var 2 c s32 4
@ end of global-symtab

@ begin of block-symtab
% begin
tab-id 2
parent-tab-id 1
% end
% begin
tab-id 3
parent-tab-id 1
% end
% begin
tab-id 1
parent-tab-id 1
var 1 i s32 4
var 2 c s32 4
% end
% begin
tab-id 4
parent-tab-id 4
% end

@ end of block-symtab
```

观察紫色区域。我们在源代码内，定义全局变量 yq 时，使用的是表达式“3+2”。然而，全局变量的计算不应在运行期完成。因此，编译过程中，表达式“3+2”被替换成“5”，标记给全局变量。

蓝色区域详细标注了函数 lb 的信息。该函数需要接受 2 个参数，分别是 a 和 b，皆为 4 字节长度。字节长度信息对希望链接本程序的程序非常有用，变量名则为后续处理提供辅助信息。

最后，我们看生成的执行指令代码：

```
@ begin of instructions
label lb
mov vreg 0 val yq
je .if_end_1
mov vreg 0 val yq
mov vreg 0 val yq
push 4 vreg 0
mov vreg 0 imm 1
pop 4 vreg 1
add vreg 1 vreg 0
xchg vreg 0 vreg 1
mov val yq vreg 0
label .if_end_1
mov vreg 0 val 1
mov vreg 0 imm 0
mov val 1 vreg 0
label .for_loop_estmt_2
mov vreg 0 val 1
push 4 vreg 0
mov vreg 0 imm 10
pop 4 vreg 1
cmp vreg 1 vreg 0 l
je .for_loop_end_2
mov vreg 0 val yq
mov vreg 0 val yq
push 4 vreg 0
mov vreg 0 val 1
pop 4 vreg 1
add vreg 1 vreg 0
xchg vreg 0 vreg 1
mov val yq vreg 0
label .for_loop_exp_2
mov vreg 0 val 1
mov vreg 0 val 1
add val 1 imm 1
jmp .for_loop_estmt_2
label .for_loop_end_2
mov vreg 0 fval a
push 4 vreg 0
mov vreg 0 imm 2
pop 4 vreg 1
add vreg 1 vreg 0
xchg vreg 0 vreg 1
mov val 2 vreg 0
mov vreg 0 val 2
push 4 vreg 0
mov vreg 0 fval b
pop 4 vreg 1
sub vreg 1 vreg 0
xchg vreg 0 vreg 1
ret
ret
label main
mov vreg 0 imm 0
ret
ret
@ end of instructions
```

借助 label 信息，我们可以清楚地找到 lb 函数和 main 函数的位置。从中，我们也可以看出，中间代码生成器正确地处理完毕 for 循环和 if 选择语句。

生成的中间代码具有不少效率较低的指令，这些有待优化器做处理。

源代码对应的语法树图片见：https://www.gardilily.com/files/compile-2.png

5.4 错误的样例：引用未定义符号

测试代码如下：

```c
int x;
int q = 3;
int x;

int f() {

    int c = 1;
    y = x + 1 + c;

    return z;

}

```

观察该代码，可以注意到以下问题：

1. 全局区域，变量 x 重复定义；

2. 函数 f 内，引用了未定义的变量 y 和 z。

我们尝试对其进行中间代码生成，得到输出结果如下：

```
flowerblack@ubuntu-mibookprox-gty:~/code-repo/ToyCompile/build$
 ./ToyCompile sUniCli -fname:resources/test/easy2.c
warning: symbol redefined: x
  token: x
  loc  : (3, 5)
error: symbol not found: y
  token: y
  loc  : (8, 5)
error: symbol not found: z
  token: z
  loc  : (10, 12)
flowerblack@ubuntu-mibookprox-gty:~/code-repo/ToyCompile/build$
```

可见，中间代码生成器成功发现这些错误，并给出提示信息。由于源文件存在致命错误（error），其得到的中间代码不可用。

5.5 全局变量数值编译期计算测试

测试代码如下：

```
1 int a = 1 + 3 + 5;
2 int b = 2 - 3 + 4;
3 int c = 2 - (3 + 4);
4 int d = 5 && 7 + 8;
5 int e = 0 || 7 && 3;
6 int f = 8 && 3 && 0 && 4;
7 int g = (1, 2, 3, 4);
8 int h = (1, 2 + 3, 4 + 5);
9 int x = 5, y = 7, z = 2 + 90;
```

执行得到结果如下（仅展示静态数据区）：

```
@ begin of static-data
int var a s32 9
int var b s32 3
int var c s32 -5
int var d s32 15
int var e s32 3
int var f s32 0
int var g s32 4
int var h s32 9
int var x s32 5
int var y s32 7
int var z s32 92
@ end of static-data
```

对于变量 a 和变量 b，较为简单，无需解释。

对于变量 c，程序正确地处理括号带来的优先级改变。

对于 d，e 和 f，成功完成逻辑表达式的短路计算。注意到，d 和 e 的结果应该是 true。按照 c 语言规范，任何非 0 数都可以代表 true，我们这种赋值方式是正确的。

g 和 h 展示对逗号表达式的处理能力，即取表达式内最后一个结果作为整体返回结果。

x，y 和 z 定义在同一行，共享同一个 int 符号。我们的程序正确处理这种写法。

综上，可见，所有变量的编译期计算结果皆正确。

六、总结与收获

6.1 项目收获

本项目中，我们使用 C++17 语言完成中间代码生成器的开发，它是整个编译器系统内的一大重要部分。研究过程中，我们不断巩固在课堂上学到的知识，并应用到实践。同时，我们也在努力优化项目结构，增强代码可读性，制作能够指引学弟学妹们参考学习的产品。通过本项目的开发，我们亦对 C++语言收获更多理解，并看到无数前人努力的身影。现在的我们拥有很幸福的开发环境。当我们打下一个字符，语言服务器会自动为我们提供输入提示，并进行潜在错误的分析。多文件项目的构建变得几乎无感。这一切都是前人们努力的成果。

6.2 遇到的问题及其解决

研发过程中，我们坚持先规划后开发，在逻辑层面未遇到任何问题。然而，C99 文法过于庞大复杂，综合考虑工期分配问题，我们决定优先框架开发，降低功能点的优先级。于是，我们选择将大部分功能拓展相关的任务延到课程设计项目内完成，包含但不限于连续赋值、函数调用、长整型、字符串。

6.3 课程认识

《编译原理》作为计算机系的一大重要课程，在整个教学体系内扮演不可替代的作用。作为计算机系的学生，学习该课程是研究计算机工作原理时不可少的一部分。

课程上，我们学习到的是相对理论的知识，是对实际应用的抽象。而在课程设计项目中，我们将理论应用到实践，巩固课堂所学的知识，感受前人留下的智慧，收获颇丰。

正如枫铃树在文章《红黑树详解》结尾处所属，上世纪的智者将一个个伟大的火把交到我们手中，我们要牢牢握住，并用它照亮前方的路，点燃远方的灯[12]。上世纪的贤者们开创形式语言相关理论，提出相关方法，制作大量工具。他们的努力为我们带来无比幸福的学习研究环境。我们接过他们的理论，重温经典，力求在经典之上有所创新，将知识的火把一代代传递下去。

七、获取产品

7.1 获取源码

本项目在 GitHub 开源。前往项目仓库以获取源代码：

https://github.com/FlowerBlackG/ToyCompile

7.2 构建运行

项目使用 CMake 管理，依赖 Boost 库。对于 Windows GCC 环境，Boost 库已经内置。对于其他环境，需自行解决依赖问题。

程序运行依赖词法自动机定义文件（tcdf）和语法定义文件（tcey）。这些文件已经放置在 resources 文件夹内，默认情况会自动加载。

编译构建，启动构建产物 ToyCompile.exe，即可体验产品功能。

参考资料

[1] 陈火旺等. 程序设计语言编译原理（第 3 版）. 国防工业出版社, 2000

[2] Maoyao233. ToyCC. https://github.com/Maoyao233/ToyCC

[3] GQT. 词法分析器. 2021

[4] Jutta Degener. ANSI C Yacc grammar. 2012

[5] 王爽. 汇编语言（第 3 版）. 清华大学出版社，2013

[6] LLVM. LLVM Tutorial. https://llvm.org/docs/tutorial/

[7] Evian Zhang. llvm ir tutorial. https://github.com/Evian-Zhang/llvm-ir-tutorial

[8] 踌躇月光. 操作系统实现. bilibili

[9] ZingLix. 8086 汇编指令集整理. ZingLix Blog, 2018

[10] 华为. MAPLE IR Specification. https://gitee.com/openarkcompiler/OpenArkCompiler/blob/master/doc/en/MapleIRDesign.md

[11] 同济大学计算机系. 第七章 语义分析和中间代码产生. 同济大学计算机系

[12] 枫铃树. 红黑树详解（下）（红黑树的删除操作）. CSDN, 2022

附录 1：C99 文法

```
/*

ANSI C Yacc grammar (This yacc file is accompanied by a matching lex file.)

In 1985, Jeff Lee published his Yacc grammar for the April 30, 1985 draft version of the ANSI C standard.  Tom Stockfisch reposted it to net.sources in 1987; that original, as mentioned in the answer to question 17.25 of the comp.lang.c FAQ, used to be available via ftp from ftp.uu.net as usenet/net.sources/ansi.c.grammar.Z

The version you see here has been updated based on an 1999 draft of the standards document. It allows for restricted pointers, variable arrays, "inline", and designated initializers. The previous version's lex and yacc files (ANSI C as of ca 1995) are still around as archived copies.

I want to keep this version as close to the 1999 Standard C grammar as possible; please let me know if you discover discrepancies. (If you feel like it, read the FAQ first.)

Jutta Degener, 2012

*/

%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token TYPEDEF EXTERN STATIC AUTO REGISTER INLINE RESTRICT
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
%token BOOL COMPLEX IMAGINARY
%token STRUCT UNION ENUM ELLIPSIS

%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%start translation_unit
%%

primary_expression
    : IDENTIFIER
    | CONSTANT
    | STRING_LITERAL
    | '(' expression ')'
    ;

postfix_expression
    : primary_expression
    | postfix_expression '[' expression ']'
    | postfix_expression '(' ')'
    | postfix_expression '(' argument_expression_list ')'
    | postfix_expression '.' IDENTIFIER
    | postfix_expression PTR_OP IDENTIFIER
    | postfix_expression INC_OP
    | postfix_expression DEC_OP
    | '(' type_name ')' '{' initializer_list '}'
    | '(' type_name ')' '{' initializer_list ',' '}'
    ;

argument_expression_list
    : assignment_expression
    | argument_expression_list ',' assignment_expression
    ;

unary_expression
    : postfix_expression
    | INC_OP unary_expression
    | DEC_OP unary_expression
    | unary_operator cast_expression
    | SIZEOF unary_expression
    | SIZEOF '(' type_name ')'
    ;
```

```
unary_operator
    : '&'
    | '*'
    | '+'
    | '-'
    | '~'
    | '!'
    ;

cast_expression
    : unary_expression
    | '(' type_name ')' cast_expression
    ;

multiplicative_expression
    : cast_expression
    | multiplicative_expression '*' cast_expression
    | multiplicative_expression '/' cast_expression
    | multiplicative_expression '%' cast_expression
    ;

additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;

shift_expression
    : additive_expression
    | shift_expression LEFT_OP additive_expression
    | shift_expression RIGHT_OP additive_expression
    ;

relational_expression
    : shift_expression
    | relational_expression '<' shift_expression
    | relational_expression '>' shift_expression
    | relational_expression LE_OP shift_expression
    | relational_expression GE_OP shift_expression
    ;

equality_expression
    : relational_expression
    | equality_expression EQ_OP relational_expression
    | equality_expression NE_OP relational_expression
    ;

and_expression
    : equality_expression
    | and_expression '&' equality_expression
    ;

exclusive_or_expression
    : and_expression
    | exclusive_or_expression '^' and_expression
    ;

inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression '|' exclusive_or_expression
    ;

logical_and_expression
    : inclusive_or_expression
    | logical_and_expression AND_OP inclusive_or_expression
    ;

logical_or_expression
    : logical_and_expression
    | logical_or_expression OR_OP logical_and_expression
    ;
```

```
conditional_expression
    : logical_or_expression
    | logical_or_expression '?' expression ':' conditional_expression
    ;

assignment_expression
    : conditional_expression
    | unary_expression assignment_operator assignment_expression
    ;

assignment_operator
    : '='
    | MUL_ASSIGN
    | DIV_ASSIGN
    | MOD_ASSIGN
    | ADD_ASSIGN
    | SUB_ASSIGN
    | LEFT_ASSIGN
    | RIGHT_ASSIGN
    | AND_ASSIGN
    | XOR_ASSIGN
    | OR_ASSIGN
    ;

expression
    : assignment_expression
    | expression ',' assignment_expression
    ;

constant_expression
    : conditional_expression
    ;

declaration
    : declaration_specifiers ';'
    | declaration_specifiers init_declarator_list ';'
    ;

declaration_specifiers
    : storage_class_specifier
    | storage_class_specifier declaration_specifiers
    | type_specifier
    | type_specifier declaration_specifiers
    | type_qualifier
    | type_qualifier declaration_specifiers
    | function_specifier
    | function_specifier declaration_specifiers
    ;

init_declarator_list
    : init_declarator
    | init_declarator_list ',' init_declarator
    ;

init_declarator
    : declarator
    | declarator '=' initializer
    ;

storage_class_specifier
    : TYPEDEF
    | EXTERN
    | STATIC
    | AUTO
    | REGISTER
    ;

type_specifier
    : VOID
    | CHAR
    | SHORT
    | INT
```

```
    | LONG
    | FLOAT
    | DOUBLE
    | SIGNED
    | UNSIGNED
    | BOOL
    | COMPLEX
    | IMAGINARY
    | struct_or_union_specifier
    | enum_specifier
    | TYPE_NAME
    ;

struct_or_union_specifier
    : struct_or_union IDENTIFIER '{' struct_declaration_list '}'
    | struct_or_union '{' struct_declaration_list '}'
    | struct_or_union IDENTIFIER
    ;

struct_or_union
    : STRUCT
    | UNION
    ;

struct_declaration_list
    : struct_declaration
    | struct_declaration_list struct_declaration
    ;

struct_declaration
    : specifier_qualifier_list struct_declarator_list ';'
    ;

specifier_qualifier_list
    : type_specifier specifier_qualifier_list
    | type_specifier
    | type_qualifier specifier_qualifier_list
    | type_qualifier
    ;

struct_declarator_list
    : struct_declarator
    | struct_declarator_list ',' struct_declarator
    ;

struct_declarator
    : declarator
    | ':' constant_expression
    | declarator ':' constant_expression
    ;

enum_specifier
    : ENUM '{' enumerator_list '}'
    | ENUM IDENTIFIER '{' enumerator_list '}'
    | ENUM '{' enumerator_list ',' '}'
    | ENUM IDENTIFIER '{' enumerator_list ',' '}'
    | ENUM IDENTIFIER
    ;

enumerator_list
    : enumerator
    | enumerator_list ',' enumerator
    ;

enumerator
    : IDENTIFIER
    | IDENTIFIER '=' constant_expression
    ;

type_qualifier
    : CONST
    | RESTRICT
```

```
    | VOLATILE
    ;

function_specifier
    : INLINE
    ;

declarator
    : pointer direct_declarator
    | direct_declarator
    ;

direct_declarator
    : IDENTIFIER
    | '(' declarator ')'
    | direct_declarator '[' type_qualifier_list assignment_expression ']'
    | direct_declarator '[' type_qualifier_list ']'
    | direct_declarator '[' assignment_expression ']'
    | direct_declarator '[' STATIC type_qualifier_list assignment_expression ']'
    | direct_declarator '[' type_qualifier_list STATIC assignment_expression ']'
    | direct_declarator '[' type_qualifier_list '*' ']'
    | direct_declarator '[' '*' ']'
    | direct_declarator '[' ']'
    | direct_declarator '(' parameter_type_list ')'
    | direct_declarator '(' identifier_list ')'
    | direct_declarator '(' ')'
    ;

pointer
    : '*'
    | '*' type_qualifier_list
    | '*' pointer
    | '*' type_qualifier_list pointer
    ;

type_qualifier_list
    : type_qualifier
    | type_qualifier_list type_qualifier
    ;

parameter_type_list
    : parameter_list
    | parameter_list ',' ELLIPSIS
    ;

parameter_list
    : parameter_declaration
    | parameter_list ',' parameter_declaration
    ;

parameter_declaration
    : declaration_specifiers declarator
    | declaration_specifiers abstract_declarator
    | declaration_specifiers
    ;

identifier_list
    : IDENTIFIER
    | identifier_list ',' IDENTIFIER
    ;

type_name
    : specifier_qualifier_list
    | specifier_qualifier_list abstract_declarator
    ;

abstract_declarator
    : pointer
    | direct_abstract_declarator
    | pointer direct_abstract_declarator
    ;
```

```
direct_abstract_declarator
    : '(' abstract_declarator ')'
    | '[' ']'
    | '[' assignment_expression ']'
    | direct_abstract_declarator '[' ']'
    | direct_abstract_declarator '[' assignment_expression ']'
    | '[' '*' ']'
    | direct_abstract_declarator '[' '*' ']'
    | '(' ')'
    | '(' parameter_type_list ')'
    | direct_abstract_declarator '(' ')'
    | direct_abstract_declarator '(' parameter_type_list ')'
    ;

initializer
    : assignment_expression
    | '{' initializer_list '}'
    | '{' initializer_list ',' '}'
    ;

initializer_list
    : initializer
    | designation initializer
    | initializer_list ',' initializer
    | initializer_list ',' designation initializer
    ;

designation
    : designator_list '='
    ;

designator_list
    : designator
    | designator_list designator
    ;

designator
    : '[' constant_expression ']'
    | '.' IDENTIFIER
    ;

statement
    : labeled_statement
    | compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    ;

labeled_statement
    : IDENTIFIER ':' statement
    | CASE constant_expression ':' statement
    | DEFAULT ':' statement
    ;

compound_statement
    : '{' '}'
    | '{' block_item_list '}'
    ;

block_item_list
    : block_item
    | block_item_list block_item
    ;

block_item
    : declaration
    | statement
    ;
```

```
expression_statement
    : ';'
    | expression ';'
    ;

selection_statement
    : IF '(' expression ')' statement
    | IF '(' expression ')' statement ELSE statement
    | SWITCH '(' expression ')' statement
    ;

iteration_statement
    : WHILE '(' expression ')' statement
    | DO statement WHILE '(' expression ')' ';'
    | FOR '(' expression_statement expression_statement ')' statement
    | FOR '(' expression_statement expression_statement expression ')' statement
    | FOR '(' declaration expression_statement ')' statement
    | FOR '(' declaration expression_statement expression ')' statement
    ;

jump_statement
    : GOTO IDENTIFIER ';'
    | CONTINUE ';'
    | BREAK ';'
    | RETURN ';'
    | RETURN expression ';'
    ;

translation_unit
    : external_declaration
    | translation_unit external_declaration
    ;

external_declaration
    : function_definition
    | declaration
    ;

function_definition
    : declaration_specifiers declarator declaration_list compound_statement
    | declaration_specifiers declarator compound_statement
    ;

declaration_list
    : declaration
    | declaration_list declaration
    ;
%%
```

附录 2：TCIR 中间代码设计

位置

ToyCompile IR（tcir）处于语法树和汇编代码之间。

我们将基于 C99 文法生成的语法树转换成 tcir，再将后者转换成汇编语言，再由汇编编译器转换为可重定向二进制文件。

文件后缀（推荐）

`tcir`

TCIR 整体结构

存储形式

使用字符形式存储，存放于文件、字符串或字符流中。文件编码必须是纯 ascii，暂不支持中文等多字节符号（特殊说明除外）。

通过换行符号标记每条指令的结尾，即：每个指令一行，每条信息一行。

每行的前缀和后缀空白符号会被过滤。

通过空格或 tab 隔离符号，即每个符号之间都要有空格，符号之间不能有逗号等奇怪的东西。

功能块设定

IR 包含几大部分，每部分有不同功能。通过以下方式标记一个区域的开始和结尾：

@ begin of ${name}

${body}

@ end of ${name}

其中，${name} 是功能区名，要按规定设置。后续会有描述。

${body} 内是该功能区的内容。具体解析方式取决于功能区名。

功能块不允许重复出现，不允许嵌套出现。

单行注释

为方便加入说明信息，tcir 支持内嵌注释。

注释格式：

// ${comment}

当检测到 // 其后（包括其本身）内容将被忽略。当然，单双引号区间内（字符串内）的双斜线不遵循此规则。

虚拟寄存器

为便于表达式计算，设计 2 个 32 位虚拟寄存器：

- t0
- t1

其中，所有表达式计算结果（含函数返回值）使用 t0 存储。

转成 x86 汇编时，可以以 eax 和 edx 表示它们。

栈增长方向

从高往低增长。例，某变量地址为 a，下一变量地址可能是 a - n。

符号关联

```
@ begin of extlink
${body}
@ end of extlink
```

该区域用于导入导出符号，相当于 8086 汇编里的 extern。

import 从外部导入

```
import ${symbol}
```

相当于 8086 汇编的 extern。

export 导出

```
export ${symbol} ${"fun" : "var"}
```

相当于 8086 汇编的 global。

注意，这里会登记全局变量，但不会登记函数。

全局数据

```
@ begin of static-data
${body}
@ end of static-data
```

字符串

```
str ${access} ${value name} "${...}"
```

其中，转义符号保留。不显式表示尾 0。value name 不含引号，需遵循变量命名规则。

access：

- val：不可变
- var：可变

例：

```
str val sayhi "hello world!\n" // const char* sayhi = "hello world\n";
```

整形

```
int ${access} ${value name} ${length} ${value}
```

其中，length 可选值如下：

内容	备注
s8	signed int 8
u8	unsigned int 8
s16	signed int 16
u16	unsigned int 16
s32	signed int 32
u32	unsigned int 32

value 为值，以 10 进制形式表示。允许有前缀负号。

例：

```
int var year s32 2022 // int year = 2022;
```

全文符号表

该表为后续代码区域提供辅助说明信息。全文不是全局的意思，该表记录所有可能用到的符号，包含那些并没有暴露给全局的符号。

```
@ begin of global-symtab
${body}
@ end of global-symtab
```

函数定义

```
fun ${visibility} ${name} ${argc} ${return-type}
    ${...args}
```

`${name}` 指定函数名。`${argc}` 指定参数个数。`${...args}` 表示参数表。

`${visibility}`:

- internal
- visible

参数表由多个三元式组成，可以写在多行。

三元式结构：

```
${type} ${ "value" | "ptr" } ${name}
```

${type} 和函数返回类型 ${return-type} 可选的值：

- u8
- s8
- u16
- s16
- u32
- s32

结构体定义

暂不支持

变量符号说明表

代码指令中的栈上变量以一个 id 记录。通过 id 在**变量符号说明表**内寻找其详细信息。

详细信息包括：

- 符号 id
- 符号名
- 符号类型
- 字节宽度

描述格式：

```
var ${id} ${name} ${type} ${bytes}
```

块符号表

块符号表可以用来辅助栈上内存分配，和判断变量作用范围（虽然这个范围在 ir 内可能很难判断）。

```
@ begin of block-symtab

% begin
${tab}
% end

% begin
```

```
${tab}
% end
...
@ end of block-symtab
```

表 id

```
tab-id ${id}
```

标明本符号表的 id。

父表 id

```
parent-tab-id ${id}
```

标明本符号表的父级表的 id。如果该表已经是祖先，则父 id 与自己的 id 相同。

符号定义

```
var ${id} ${name} ${type} ${bytes}
```

指令概述

指令区通过以下形式划出：

```
@ begin of instructions
${instructions}
@ end of instructions
```

值表示

对于常量，直接书写值。允许写负数。

对于变量，写其在**变量符号说明表**内的编号。

对于虚拟寄存器，写其后缀编号。

值种类：

- imm: 数字（立即数）
- val: 变量。数字表示变量 id，名字表示全局变量。
- vreg: 虚拟寄存器
- fval: 函数参数

访存方式：

- 不加任何标记：直接
- **暂不支持偏移寻址**

后续所有 ${value} 都遵循此规则。

例：

```
mov vreg 0 imd 2  // t0 = 2
```

指令

标签与跳转

标签定义

```
label ${name}
```

建议当成汇编里的标签看。

如果这个标签表示一个函数，在 ir 内不用考虑寄存器保护问题。后续转汇编时，结合全局函数表，判断标签是否为函数，再决定是否要保护寄存器。

直接跳转

```
jmp ${label-name}
```

条件跳转

```
j${condition} ${label-name}
```

condition 为跳转条件，根据 t0 中的值判断。

可用的条件：

```
jge: x >= 0
jg: x > 0
je: x == 0
jl: x < 0
jle: x <= 0
jne: x != 0
```

函数调用跳转

```
call ${label-name}
```

不负责处理调用栈创建与清理等事情。

需要负责完成部分寄存器保护等工作。

函数调用返回

```
ret
```

不负责处理调用栈创建与清理等事情。

需要完成部分寄存器恢复等工作。

栈操作

push 压栈

push ${bytes} ${value}

bytes: 1, 2, 4

value：常数、虚拟寄存器、变量、全局常量

pushfc 函数调用压栈

pushfc ${bytes} ${value}

pop 弹出

pop ${bytes} ${value}

pop ${bytes} // 空弹出

popfc 函数调用弹出

popfc ${bytes}

存取

mov

```
mov ${container} ${value}
```

注意，这与汇编中的 mov 是很不同的。汇编中很难直接在内存之间移动，但 tcir 允许这么做，毕竟这只是中间代码。

lea

```
lea ${container} ${value}
```

加载地址。

算术

add

```
add ${value1} ${value2}  // value1 = value1 + value2
```

sub

```
sub ${value1} ${value2}  // value1 = value1 - value2
```

neg

```
neg ${value1}  // value1 = -value1
```

乘法和除法

暂不支持

and

```
and ${value1} ${value2}  // value1 = value1 & value2
or
or ${value1} ${value2}  // value1 = value1 | value2
xor
xor ${value1} ${value2}  // value1 = value1 ^ value2
not
not ${value}  // value = !value
```

比较

```
cmp
cmp ${value1} ${value2} ${true-condition}
${true-condition}:
ge - greater or equal
le - lesser or equal
eq - equal
ne - not equal
g
l
```

比较 value1 和 value2，t0 被设为是否满足 true-condition。

其他

```
xchg
xchg ${value1} ${value2}  // swap value1, value2
```

参考

- [1] 王爽. 汇编语言（第 3 版）. 清华大学出版社，2013
- [2] LLVM. LLVM Tutorial. https://llvm.org/docs/tutorial/
- [3] Evian Zhang. llvm ir tutorial. https://github.com/Evian-Zhang/llvm-ir-tutorial
- [4] 踌躇月光. 操作系统实现. bilibili
- [5] ZingLix. 8086 汇编指令集整理. ZingLix Blog, 2018
- [6] 华为. MAPLE IR Specification. https://gitee.com/openarkcompiler/OpenArkCompiler/blob/master/doc/en/MapleIRDesign.md
- [7] 陈火旺等. 程序设计语言编译原理（第 3 版）. 国防工业出版社，2000
- [8] 同济大学计算机系. 第七章 语义分析和中间代码产生. 同济大学计算机系