同济大学

TONGJI UNIVERSITY

《编译原理》
设计说明

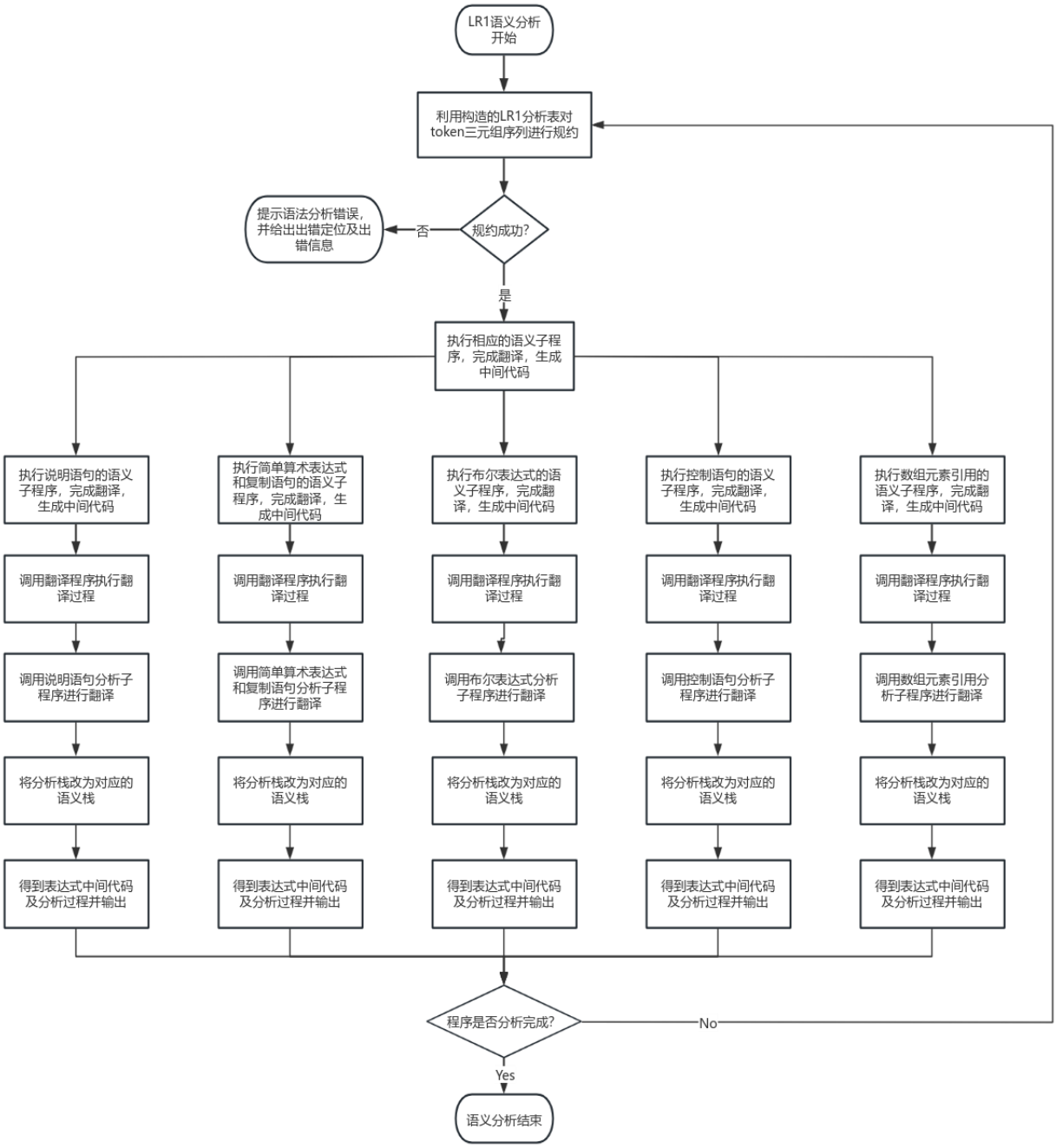
作业名称	中间代码生成工具
小组成员	2153603 王政尧
	2153683 郭嘉
	2154071 张博文
学院（系）	电子与信息工程学院
专　　业	计算机科学与技术
任课教师	丁志军
日　　期	2023 年 12 月 24 日

装

订

线

一、 整体设计



本次实验要求：

（1）在前面实验的基础上（词法、语法分析），进行语义分析和中间代码生成器的设计，输入类 C 源程序，输出等价的中间代码四元式序列。

（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、 语义分析和中间代码生成

2.1 概述

语义分析的任务是在语法分析的基础上，考虑代码的含义与功能，并将类 C 语言源代码转化成另一种更简洁、通用的中间代码的形式。常见的中间代码有逆波兰式、三元式、间按三元式、四元式等，本作业选用的是四元式。

在分析的过程中，还需要检查语义错误，如使用未定义变量、或变量重定义等错误。

2.2 数据结构设计

语义分析需要针对一些特定的语句生成中间代码，这同样通过产生式的规约，语法分析和语义分析使用同一套产生式，但变元表示的含义有所不同，为此，需要新设计一种表示适用于语义分析过程的变元的数据结构，将其命名为 State。

根据语义分析的属性文法，为 State 添加如下成员：

```
struct State
{
    string place;
    vector<int> true_list;
    vector<int> false_list;
    vector<int> next_list;
    int next_quad;
    int line_count;
}
```

这些是所有属性文法产生式涉及到的属性，我们将其整合在一起，按需取用。其中，各属性表示的含义是：

place：表示变元的名称

true_list：在涉及布尔表达式的产生式中使用，表示变元的真出口链表

false_list：同 true_list，表示变元的假出口链表。

next_list：表示语句块内需要进行跳转的四元式的编号链表。

next_quad：用于记录某一位置的待产生四元式的编号。
line_count：表示变元对应的原标识符在源代码中的行数，由词法分析提供，在语义检查出错时用于提示错误位置。

2.3 详细设计过程

2.3.1 实现思路

语义分析和语法分析是一起进行的。为此需要添加一个栈，来存放在 2.2 中提到的属性文法变元 State。

在移进操作中，在原先移进语法状态和符号的基础上，还需要将属性文法变元入栈，令该变元的 place 为输入符号的 token，此时改变元的 place 更倾向于表示输入字符串的内容，而非变元真正的 place，在之后可能还需要被修改。

在规约过程中，需要利用 ACTION 表获得规约使用的产生式序号，语义分析需要对其中一些特定的产生式，根据给定的翻译模式做语义翻译。由于这里的翻译并没有像语法分析构造 ACTION 表和 GOTO 表这样高度集成的算法，因此需要对每条产生式单独讨论。

2.3.2 一般的产生式

语法分析的产生式一共有 69 条，我们只需要对其中的约 30 条做处理，利用 map 记录它们的序号，其余的产生式我们称为“一般的产生式”。对于这些一般的产生式，其规约出的父变元在属性文法上是没有任何意义的，它的存在只是为了在规约时保证栈顶的若干变元与产生式右部一致。由于许多一般产生式的右部只有一个变元，因此在每次规约前将栈顶的变元赋值给一个新的 State，记为 fa，表示产生式的左部，在规约完成后将栈顶相应个数的变元弹出，再将 fa 压入栈。这样的好处是同时可以处理只需要进行变元继承的产生式，如 Term->Factor，按之前的处理可以将 Factor 的内容直接继承给 Term，省去了更多的判断。

2.3.3 M 产生式

产生式形式：M -> ϵ

翻译模式：{M.quad = nextquad}

由于 M 生成空，因此在文法中添加 M 并不会影响正常的规约过程。M 的作用是记录此时下一条或者说将要生成的四元式的序号，在布尔表达式的回填操作中要使用到。

2.3.4 N 产生式

产生式形式：N -> ϵ

翻译模式：{ N.nextlist:=makelist(nextquad);emit(j,-,-,-) }

N 同样生成空。该翻译模式生成了一条跳转语句，用来控制 N 之前的语句块执行到此处时需要跳转至的位置。

2.3.5 声明产生式

产生式形式：Parameter -> int idt

这个产生式的功能是声明变量，由于我们只考虑了 int 类型的数据，因此这里是 int 而非 Ftype。声明语句可能在函数的形参中出现，也可能在代码块的前半部分 InternalDeclaration 中出现。规约这条产生式时，需要记录声明了什么变量，为此，需要一个 string 类型的数据，来记录源代码声明了哪些变量。由此，可以对源代码中出现的所有标识符进行判断，如果标识符出现在声明语句中，且该标识符已经被声明过，那么就出现了重定义的错误；如果标识符不在声明语句中且被使用，那么就出现了未定义的错误。

2.3.6 语句合并产生式

产生式形式：StatementSequence -> StatementSequence M Statement

对应课本中给出的属性文法：L -> L1 M S

翻译模式：{ backpatch(L1.nextlist, M.quad); L.nextlist:=S.nextlist }

这里要用 M.quad 回填 L1.nextlist，也就是将 L1 的跳转出口指定至 S 的入口，并将 S.nextlist 继承给 L，这样就实现了两段代码块的合并。

2.3.7 if 表达式产生式

产生式形式：IfStatement -> if (BoolExpression) M StatementBlock

对应课本中给出的属性文法：S→if E then M S1

翻译模式：{ backpatch(E.truelist, M.quad); S.nextlist:=merge(E.falselist, S1.nextlist) }

这里按照课本给出的翻译模式进行翻译即可，将布尔表达式的真链出口指定至 M 记录的序号，并将布尔表达式的假链出口和 S1 语句块的出口合并，作为整个 if 表达式的出口。值得一提的是 if 和 while 语句的表达式使用了非终结符 BoolExpression 而不是 Expression，作这个区分的原因是 if、while 等语句中的表达式最终是不需要通过一个中间变量来记录它的值的，只需要通过真假链进行跳转即可，而参与运算的 Expression 是需要一个变量来记录它的值的，可能是声明的变量，也可能是临时记录的中间变量。这样进行区分之后，就可以避免 if、while 等语句的布尔表达式在翻译时生成冗余的四元式。

if-else 表达式在此基础上多加了一些操作，但基本思想一致，这里不再赘述。

2.3.8 while 表达式产生式

产生式形式：WhileStatement -> while M (BoolExpression) M StatementBlock

对应课本中给出的属性文法：E -> while M1 E do M2 S1

翻译模式：{backpatch(S1.nextlist, M1.quad); backpatch(E.truelist, M2.quad);
S.nextlist:=E.falselist; emit(‘j,－,－,’ M1.quad) }

简单想象一下 while 表达式的执行过程，就可以看懂 while 的翻译模式。用 M1.quad 回填 S 语句块的 nextlist，即语句块跳出后重新判断布尔表达式；用 M2.quad 回填布尔表达式的 truelist，即判断布尔表达式为真后，执行 S 语句块；布尔表达式的假出口就是整个 while 表达式的出口；并在最后添加一条跳转语句，跳转至判断布尔表达式的位置，在语句块正常执行完未发生跳转的情况下，进行循环。

2.3.9 赋值表达式产生式

产生式形式：AssignmentStatement -> idt = Expression ;

翻译模式：{emit(:=, Expression.place, _, idt.place)}

这里很好理解，就是生成将 Expression 赋值给 idt 的语句的四元式。但这里有两点需要注意的情况：

a) 需要判断等号左端标识符是否为未定义标识符的问题。

b) 需要保证等号右端 Expression 必须有 place 的问题，这主要涉及对布尔表达式的处理。举个例子，对于赋值语句 i＝a＞b; ，之前我们提到过，不希望布尔表达式生成冗余的中间变量，因此将 a＞b 语句规约至 Expression 时，这个表示 Expression 的变元是没有 place 的，因此我们需要在生成赋值语句之前做一次检查，保证 Expression 变元有 place，如果没有，就在此处生成一个中间变量，这种情况只有在涉及到布尔表达式的时候才会出现，因此赋予中间变量就是将布尔值转化为 0/1 值的问题。实现这个功能需要四条四元式，两条赋值语句，为一个新的中间变量分别赋值为 1 和 0，并且用各自的序号回填 Expression 的 truelist 和 falstlist；两条无条件跳转语句，分别跟在两句赋值语句之后，作为 Expression 新的 truelist 和 falselist。代码如下：

```
s.place = "v" + to_string(vars.size());
vars.push_back(s.place);
int t = next_quad();
emit(":=", "1", "", s.place);
emit("j", "", "", "0");
int f = next_quad();
emit(":=", "0", "", s.place);
emit("j", "", "", "0");
backpatch(s.true_list, t);
backpatch(s.false_list, f);
s.true_list = {t + 1};
s.false_list = {f + 1};
```

2.3.10 逻辑或表达式产生式

产生式形式：OrExpression -> OrExpression || M AndExpression

对应课本中给出的属性文法：E→E1 or M E2

翻译模式：{ backpatch(E1.falselist, M.quad); E.truelist:=merge(E1.truelist, E2.truelist);
E.falselist:=E2.falselist }

在这里可以更明显地体会到，布尔表达式不生成四元式的思想。利用 M 来记录 E2 的首地址，E1 判断失败后，用 M.quad 回填其 falselist，整个表达式的 truelist 是 E1 和 E2 的 truelist 的并集；falselist 是 E2 的 falselist，这与 or 的定义相符。

2.3.11 逻辑与表达式产生式

产生式形式：AndExpression -> AndExpression && M NotExpression

对应课本中给出的属性文法：E→E1 and M E2

翻译模式：{ backpatch(E1.truelist, M.quad); E.truelist:=E2.truelist;
E.falselist:=merge(E1.falselist,E2.falselist) }

与的处理类似。E1 判断成功后需要继续判断 E2，用 M.quad 回填其 truelist，整个表达式的 truelist 是 E2 的 truelist；falselist 的 E1 和 E2 的 falselist，与 and 的定义相符。

2.3.11 逻辑非表达式产生式

产生式形式：NotExpression -> ! CompareExpression

对应课本中给出的属性文法：E → not E1

翻译模式：{ E.truelist:=E1.falselist; E.falselist:=E1.truelist}

这里将 E1 的 truelist 和 falselist 交换后继承给 E，也就是完成逻辑取反的任务。

2.3.12 布尔表达式产生式

产生式形式：CompareExpression -> AdditiveExpression

对应课本中给出的属性文法：E → id

翻译模式：{ E.truelist:=makelist(nextquad); E.falselist:=makelist(nextquad+1);
emit(‘jnz’ ‘,’ id .place ‘,’ ‘—’ ‘,’ ‘0’)；emit(‘ j, -, -, 0’) }

这里对应的是将加法表达式直接作为布尔表达式使用的情况，比如 if(a)中的 a。由于此时这里的加法表达式并未经过与或非以及比较表达式，因此该变元是没有真假链的，所以我们需要在这里进行补充，就按照如上的翻译模式，创建真链和假链，并生成两条跳转语句，分别与真链和假链对应。

2.3.13 比较表达式产生式

产生式形式：CompareExpression -> Expression relop AdditiveExpression

对应课本中给出的属性文法：E → id1 relop id2

翻译模式：{ E.truelist:=makelist(nextquad); E.falselist:=makelist(nextquad+1);
emit(‘j’ relop.op ‘,’ id 1.place‘,’ id 2.place ‘,’ ‘0’); emit(‘j, -, -, 0’) }

这里会为新生成的变元生成真假链，并生成两条跳转语句与真假链对应，根据比较表达式的结果决定跳转目的地。需要注意的是，这里与赋值表达式相同，需要 id1 与 id2，也就是 Expression 和 AdditiveExpression 都具有 place 属性，因为这里涉及值的比较，如果 Expression 从布尔表达式规约上来，需要按照真为 1、假为 0 的规则为布尔表达式赋予中间变量，用于进行比较。

2.3.13 运算表达式产生式

产生式形式：OperatorTerm -> Factor operator Factor

对应课本中给出的属性文法：E → id1 operator id2

翻译模式：{ emit(operator, ‘id1.place’, ‘id2.place’, newtemp) }

这里对应加减乘除四条产生式，它们的规约规则相同，在文法中通过不同非终结符来决定它们的优先级，乘除高于加减。只需生成一条运算语句，将 id1 和 id2 的运算结果给到一个新的中间变量即可。

2.3.14 标识符产生式

产生式形式：Factor -> idt

在这条产生式规约时，需要检查标识符是否出现过，如果标识符未出现过，需要提示使用未声明变量的错误。

以上就是所有涉及中间代码生成或进行语义检查的产生式。

三、更为通行的高级语言的语义检查和中间代码生成

3.1 语义分析的复杂性

3.1.1 类型系统

高级语言通常有更复杂的类型系统，包括强类型、静态类型等。需要检查变量、表达式和函数调用之间的类型兼容性。

3.1.2 作用域和符号解析

需要处理变量作用域、函数作用域以及符号解析的规则，确保正确的标识符解析和使用。

3.1.3 解决方案

设计并实现一个强大的类型检查系统，包括类型推断、类型转换和作用域解析机制。这需要建立符号表、类型检查器以及作用域管理器等组件。

3.2 控制流和数据流分析

3.2.1 控制流语句

高级语言包括条件语句、循环语句等，需要分析和处理控制流转移。

3.2.2 数据流分析

考虑变量的生命周期、数据依赖和数据流转移。

3.2.3 解决方案

实现一个控制流图和数据流图的生成器，以便分析代码中的控制流和数据流。这可以帮助进行常量传播、死代码消除和优化等操作。

3.3 错误处理和异常处理

3.3.1 语义错误处理

需要检测并报告语义错误，如类型不匹配、未声明的变量等。

3.3.2 异常处理

高级语言通常有异常处理机制，需要考虑如何生成代码以支持异常的抛出和捕获。

3.3.3 解决方案

实现一个强大的错误处理机制，能够捕获和报告语义错误，并提供合适的错误信息。同时，设计支持异常处理的中间代码表示和生成方式。

3.4 面向对象和函数式编程支持

3.4.1 面向对象特性

如果语言支持面向对象编程，需要处理类、对象、继承、多态等概念。

3.4.2 函数式编程

如果支持函数式编程，需要处理高阶函数、闭包、纯函数等特性。

3.4.3 解决方案

针对面向对象或函数式编程特性，需要设计对应的中间代码表示形式，并编写转换规则以支持这些特性。

3.5 最优化和性能提升

3.5.1 优化技术

考虑在中间代码生成阶段应用基本的优化技术，例如常量折叠、公共子表达式消除等。

3.5.2 性能考虑

考虑生成高效的中间代码，使得最终生成的目标代码能够在性能上有所优势。

3.5.3 解决方案

实现一些基本的代码优化技术，例如局部优化和全局优化，以及对代码生成过程的性能考量。

对于更为通行的高级语言，需要在语义分析和中间代码生成器中整合更多的特性和复杂性。设计一个灵活而强大的系统来处理类型、作用域、控制流、异常、面向对象/函数式编程等方面的功能，并实现基本的代码优化以提高最终生成代码的质量和性能。

3.6 LLVM

3.6.1 LLVM 概述

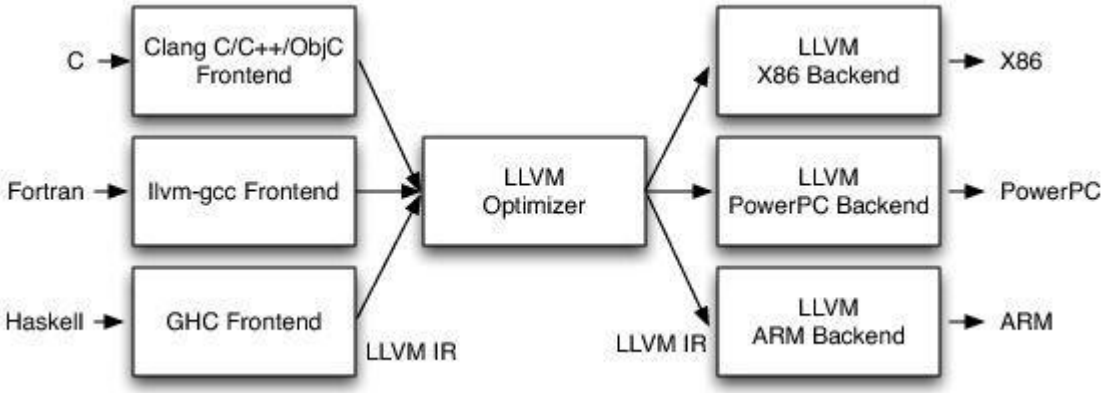
LLVM 是 Low Level Virtual Machine（底层虚拟机）的缩写。LLVM 通过在编译时、链接时、运行时和运行之间的空闲时间为编译器转换提供高级信息，从而支持对任意程序进行透明的、终身的程序分析和转换。LLVM 以静态单赋值（SSA）的形式定义了一种常见的低级代码表示形式，具有如下特性：一个简单的、与语言无关的类型系统，它公开了通常用于实现高级语言特性的原语；一种用于输入地址运算的指令；一个简单的机制，可以用来实现高级语言的异常处理特性。

3.6.2 LLVM 体系架构

LLVM 是一个开源的编译器基础设施，它被设计用于优化编译，支持各种编程语言，并提供灵活的中间表示（IR，Intermediate Representation）。LLVM 不仅仅是一个

编译器，它也是一个编译器基础设施的集合，包括一系列的工具、库以及用于构建编译器的核心组件。

LLVM 为了简化编译器开发流程，将这三个阶段分离开来，用统一的中间语言 IR 将这三个阶段串联起来。这样就可以无视输入语言的不同带来的问题，只 需要设计一个通用的优化器，对于后端模块来说，把 IR 作为它的输入，就能为特定的硬件平台开发位移的后端代码生成器。因此要使用 LLVM 框架开发编译器，需要开发的仅仅是一个输入为 IR 的前端，其他的模块可以不用考虑。LLVM 编译器的结构如下图：



图：LLVM 架构

3.6.3 中间代码 IR

LLVM 的中间语言 IR（Intermediate Representation，中间表示）是 LLVM 编译器基础设施中的一种抽象的低级编程语言。LLVM IR 被设计为一种与平台无关的、面向类型的中间表示形式，它在编译过程中作为编译器前端和后端之间的中间层。

• 低级别和面向类型：LLVM IR 比高级语言更接近硬件的层次，但仍然比汇编语言更抽象。它包括类型信息，并具有高级语言不具备的底层操作，例如指针算术和内存管理。

• 静态单赋值形式（SSA）： LLVM IR 采用静态单赋值形式，其中每个变量只被赋值一次。这种形式简化了对程序的分析和优化。

• 模块化和扩展性：LLVM IR 被设计成可以轻松地扩展和添加新的指令，这使得 LLVM IR 适用于不同的编程语言和编译器优化。

• 平台无关性：IR 是与平台无关的，这意味着它不针对特定的硬件架构，可以通过后端编译成针对不同架构的机器码。

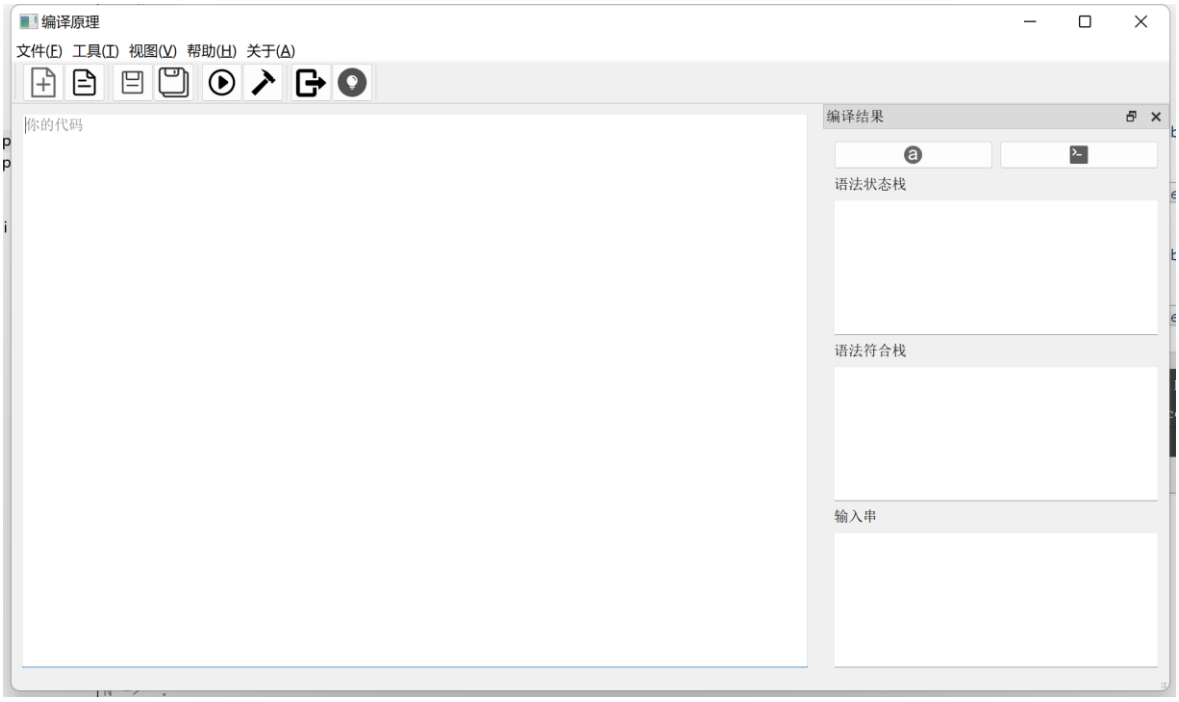
•优化友好：由于其低级别的特性，LLVM IR 很容易被各种优化技术处理。LLVM 的优化器可以对 IR 进行各种优化，以改善代码的性能和效率。

LLVM IR 通常用于在编译器的不同阶段之间传递信息。编译器的前端会将源代码翻译成 LLVM IR，然后通过 LLVM 提供的优化器对 IR 进行优化，最后由后端将 LLVM

IR 转换成目标机器的机器码。因为 LLVM IR 提供了一个抽象的、高度优化的表示形式，所以它在编译器工具链中发挥着关键作用，让开发者能够进行跨平台的编译优化，提高代码的性能和可移植性。

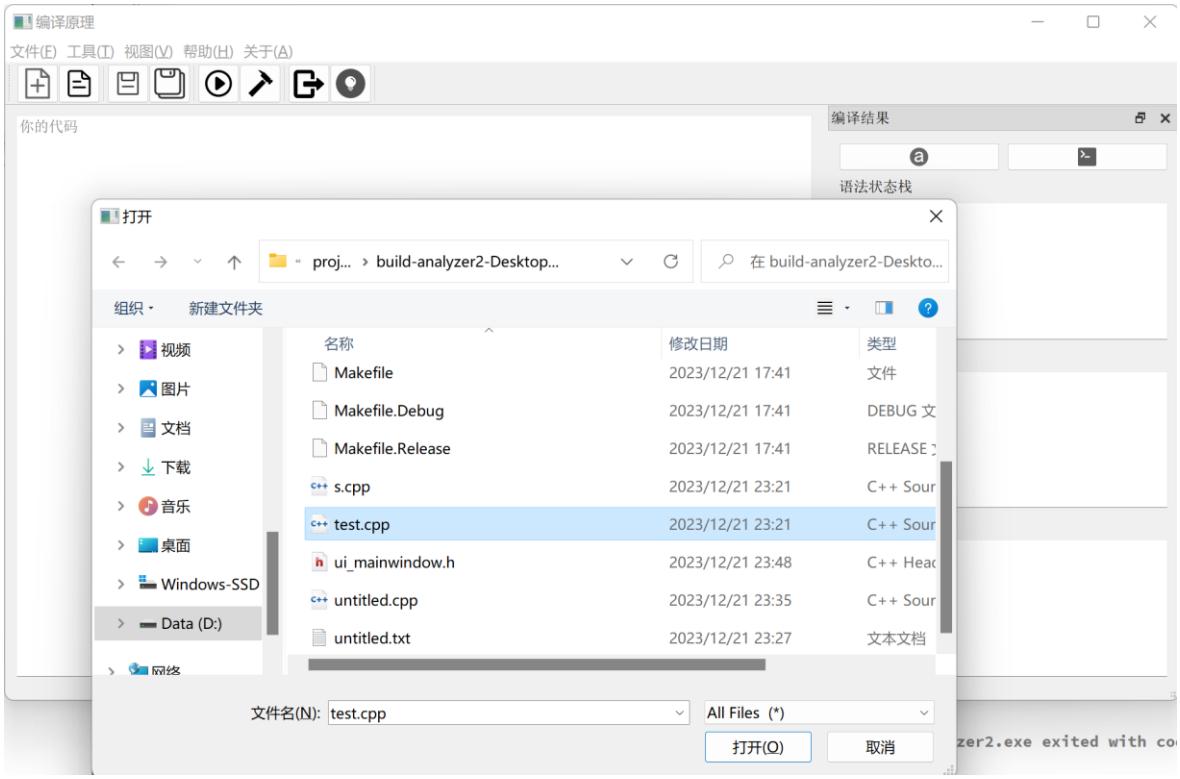
总体来说，LLVM 的设计使得它能够适用于各种不同的编程语言，并且提供了一个强大且灵活的编译器基础设施，使得开发者可以构建高效、可扩展的编译器和工具，值得我们的研究学习和参考借鉴。

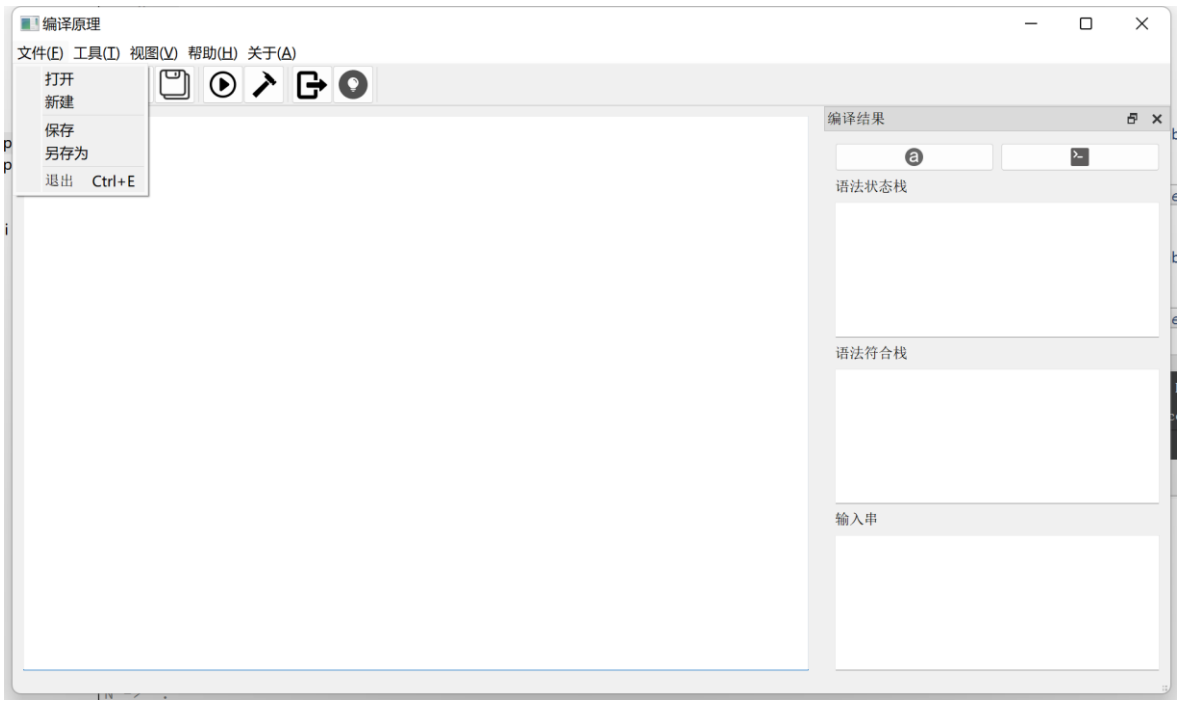
四、 界面设计与实验结果



装 订 线

（1） 用户点击打开按钮从主界面选择导入文件，或着点击新建按钮新建文件，工具栏中文件部分中也有打开和新建功能。





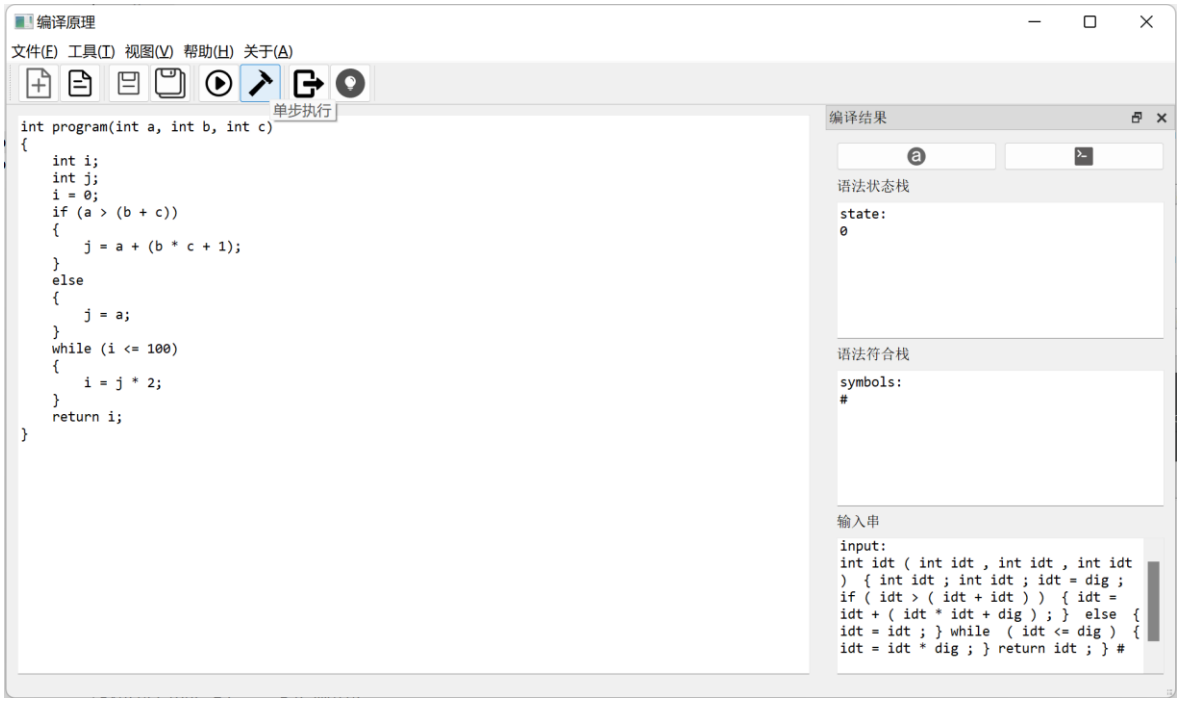
装 订 线

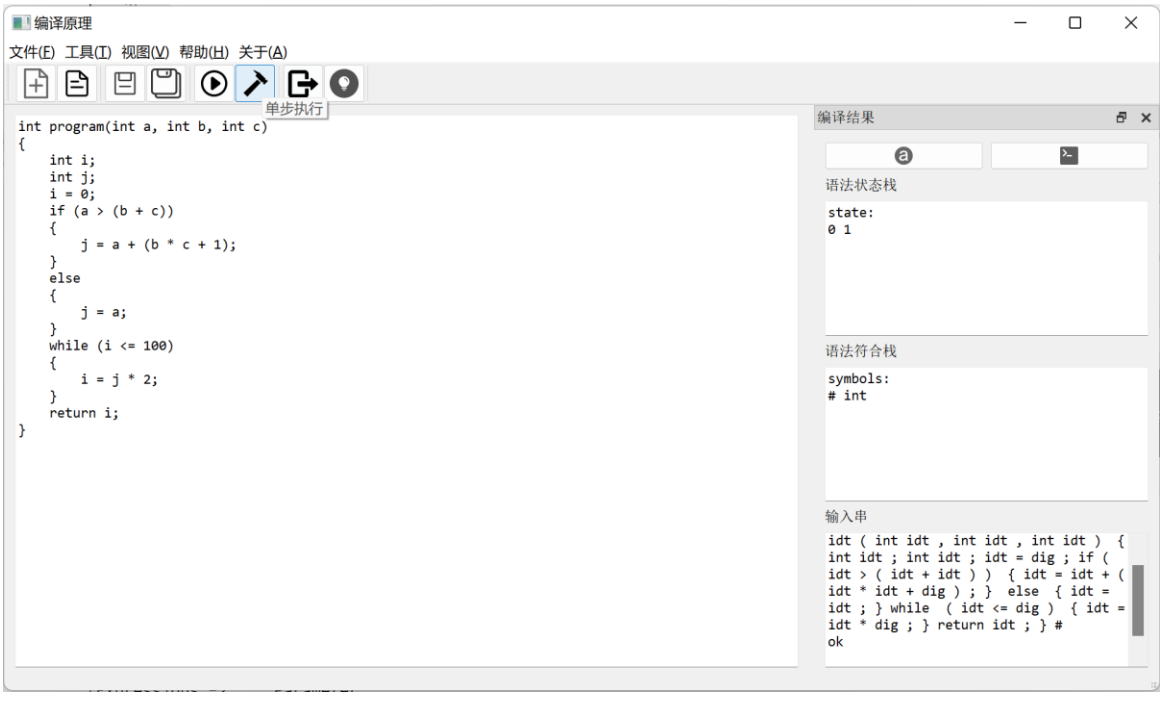
（2） 导入文件后，your code 界面自动显示文件内容

装 订 线

（3） 点击工具栏上的“单步执行”按钮或者“全部执行”按钮，进行词法与语法分析。单步执行可以支持一步一步走，全部执行可以直接一步完成所有分析。右边的编译结果窗口显示编译结果。其中显示了语法状态栈，语法符号栈，输入串和中间代码窗口。编译结果窗口中也有工具栏，实现在语法词法分析和中间代码分析的切换，同时编译结果窗口也是一个 dock 窗口，可以自由拖拽。

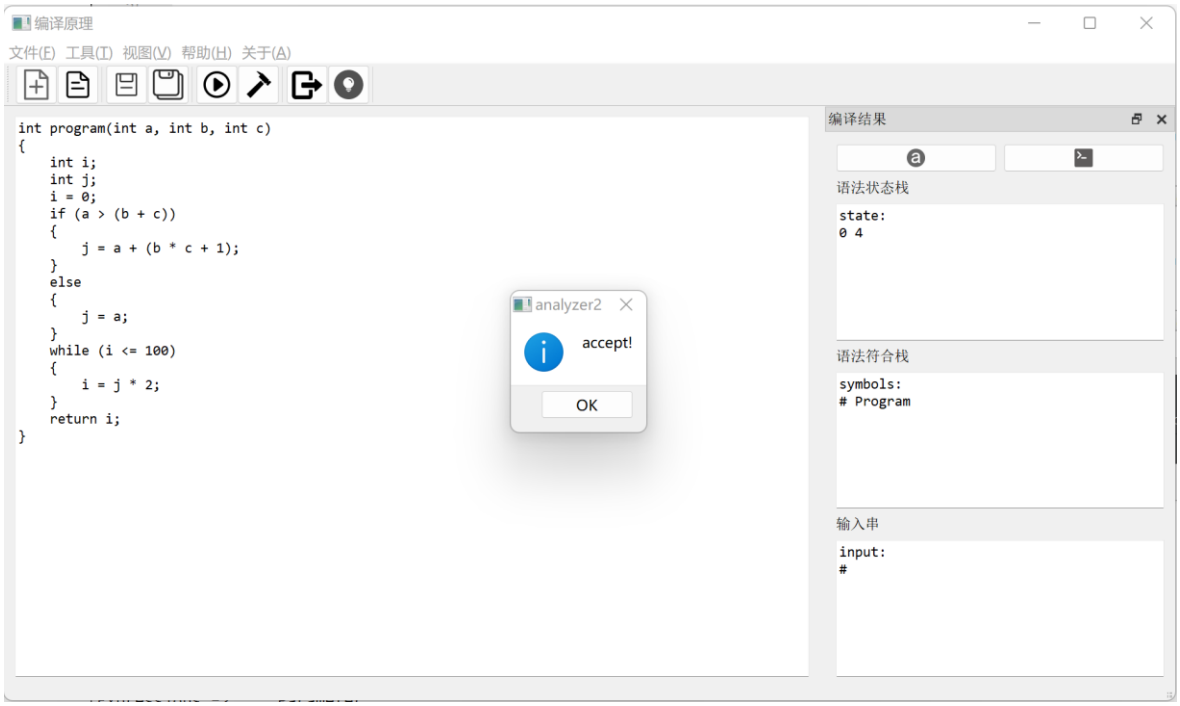
单步执行与编译结果窗口展示：

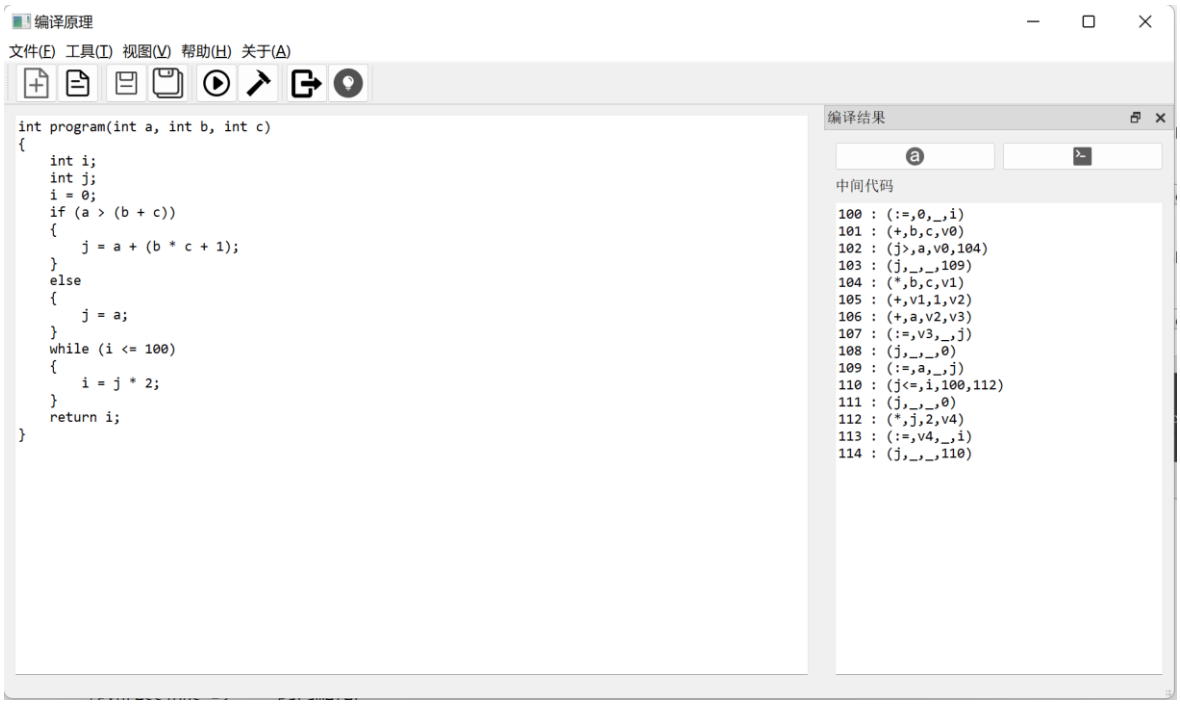




装 订 线

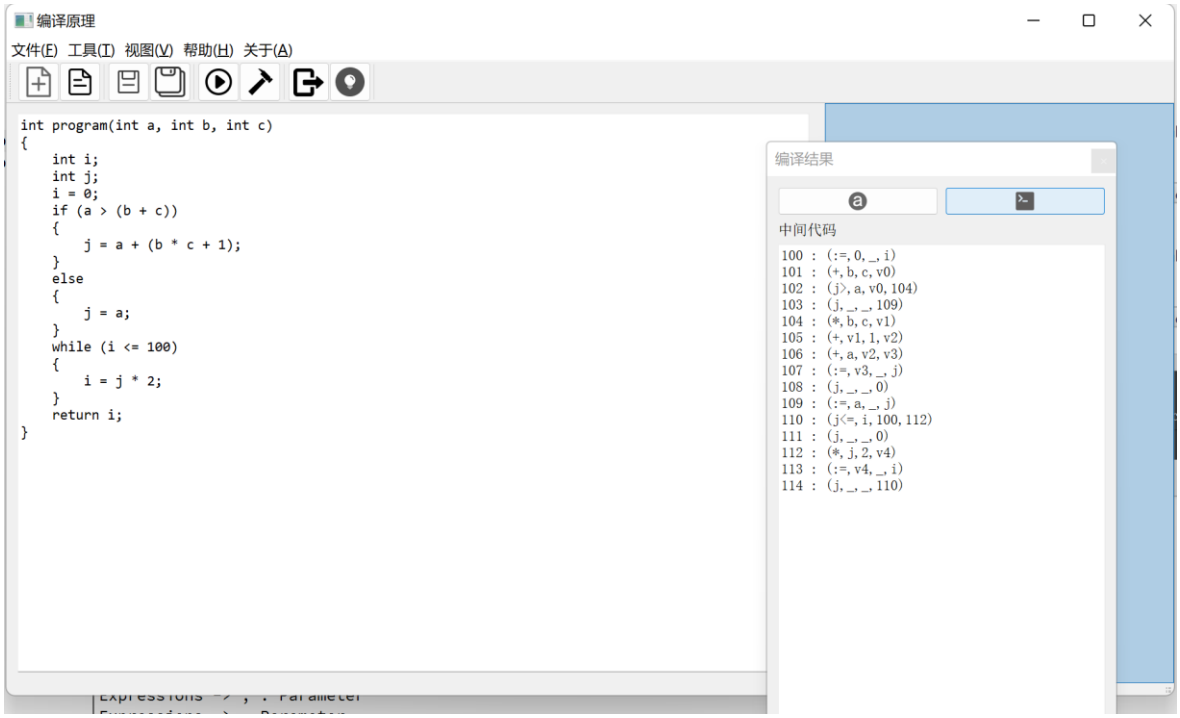
全部执行与编译结果展示：



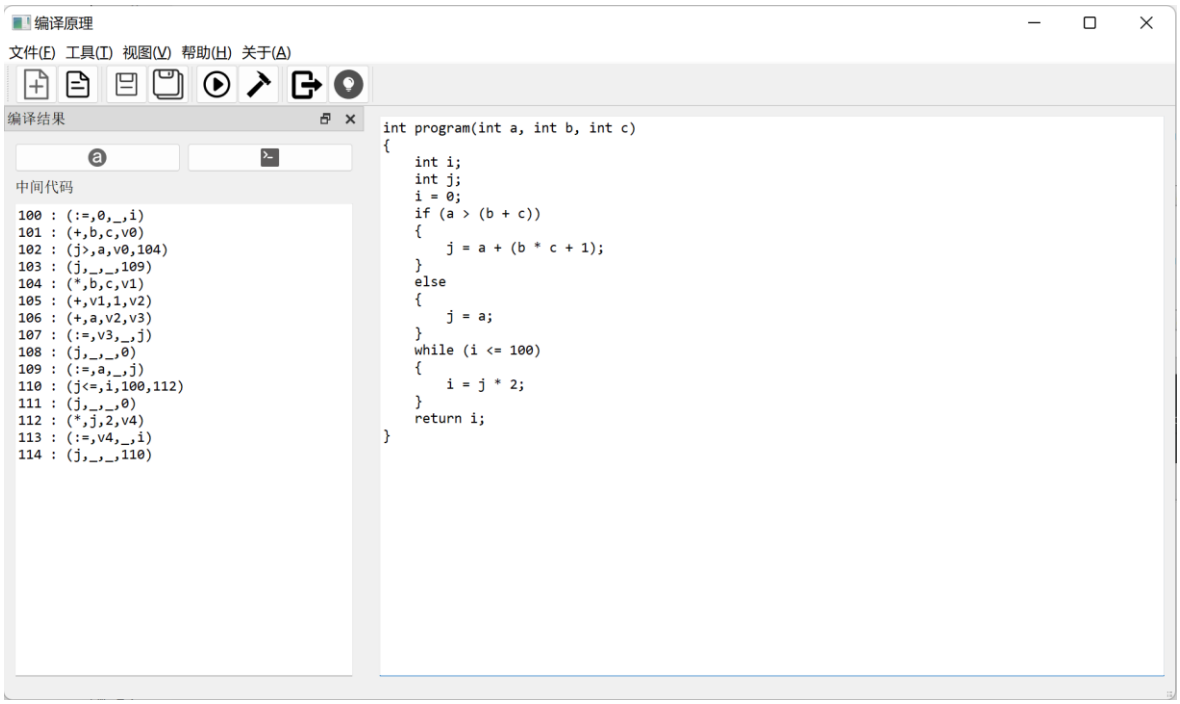


装 订 线

Dock 窗口展示：



换到左边：

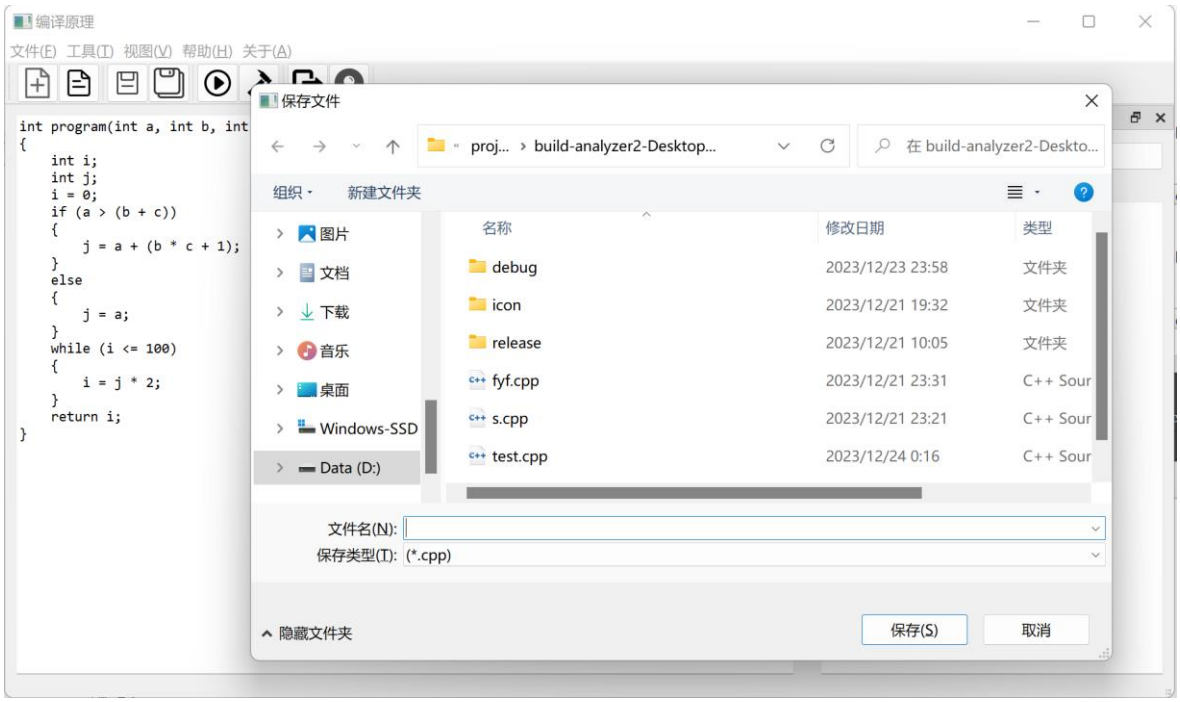


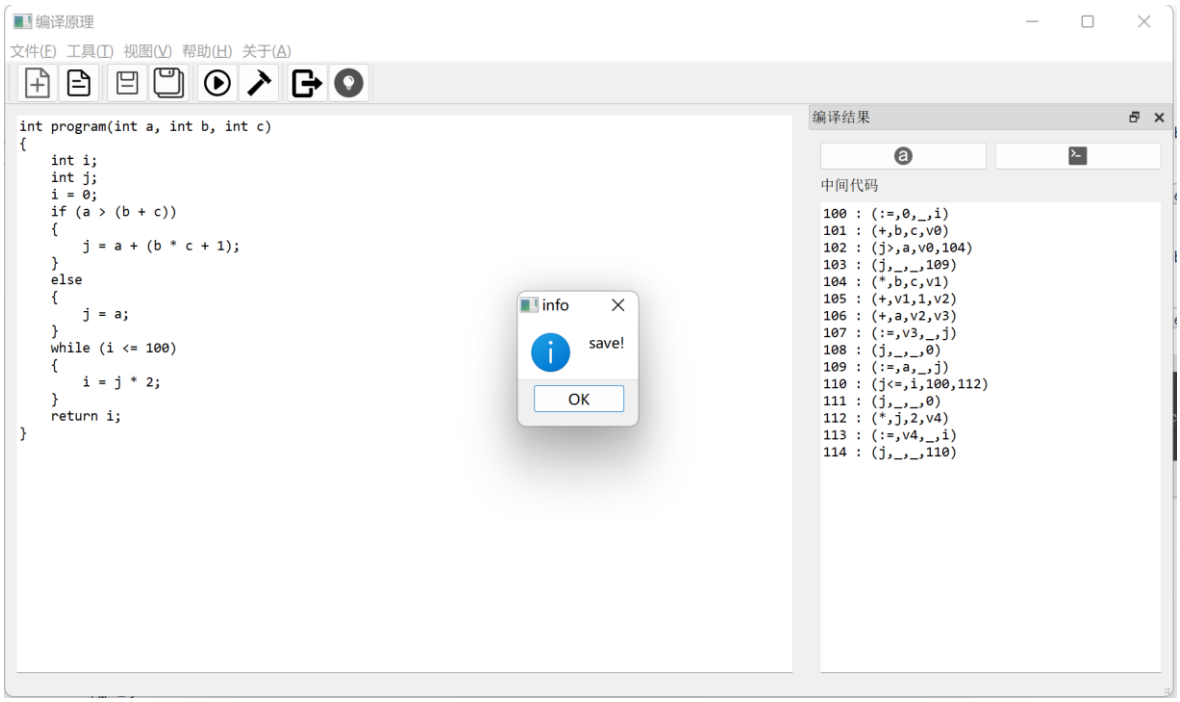
装

订

线

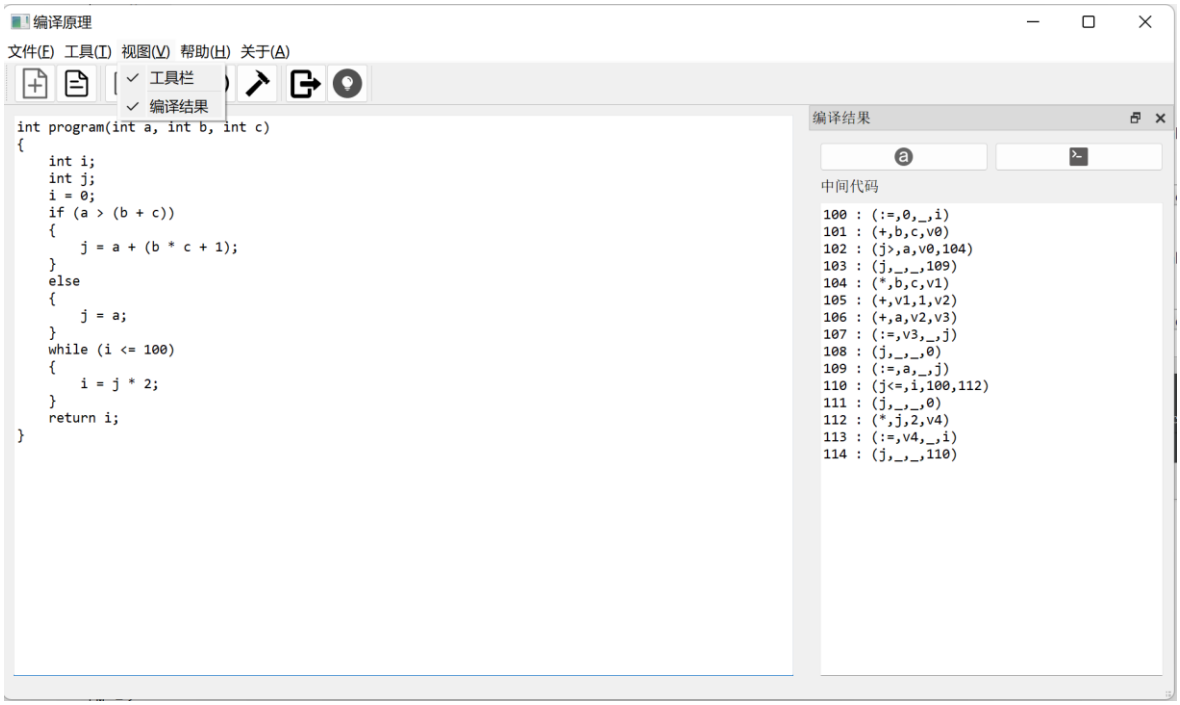
（4） 工具栏中可以实现保存和另存为功能。



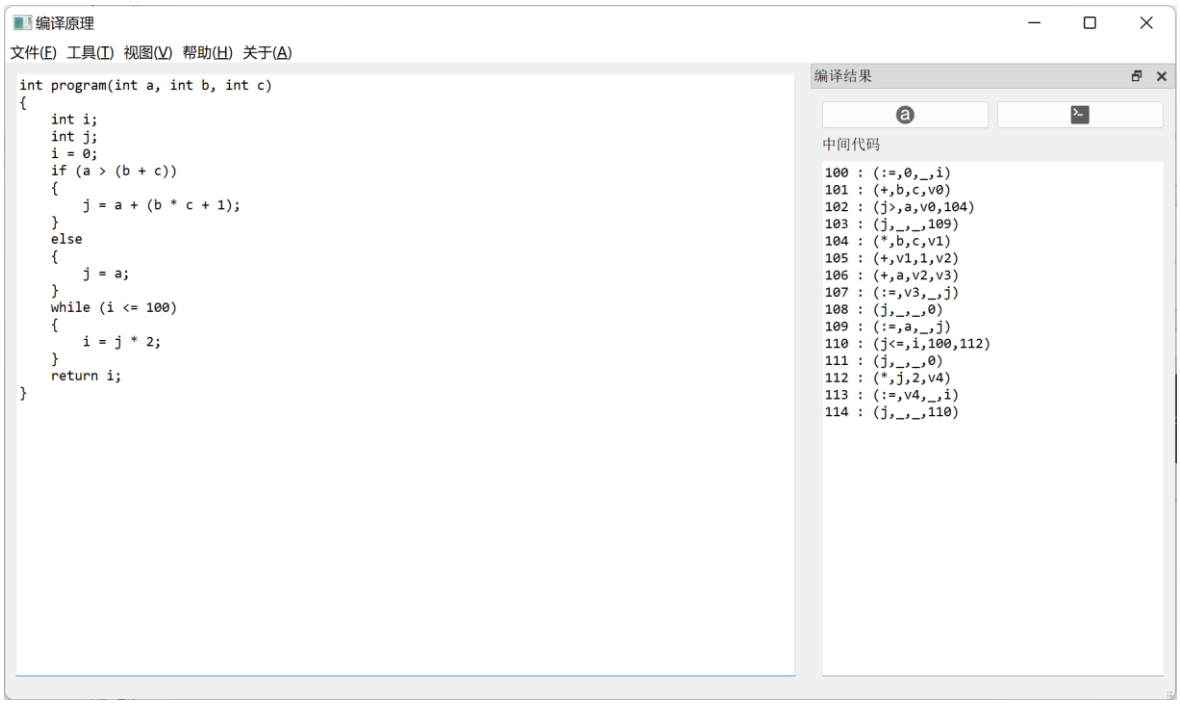


装 订 线

（5） 工具栏中视图部分可以选择隐藏工具栏和编译结果窗口。



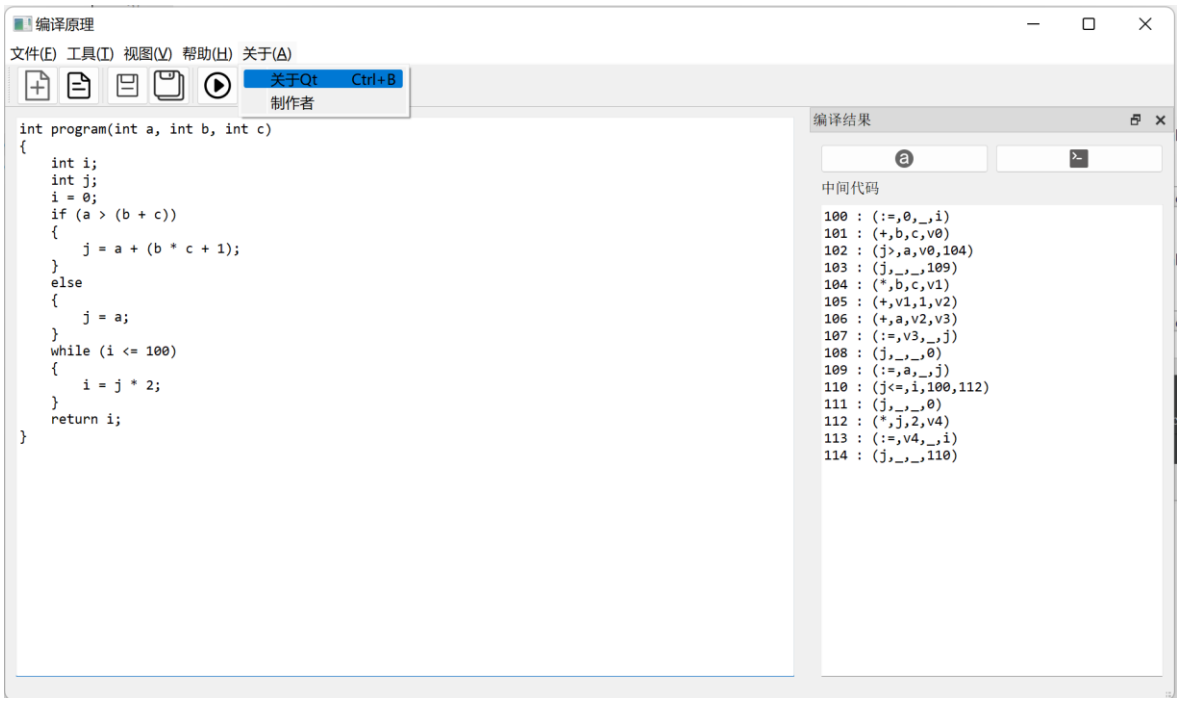
隐藏工具栏：



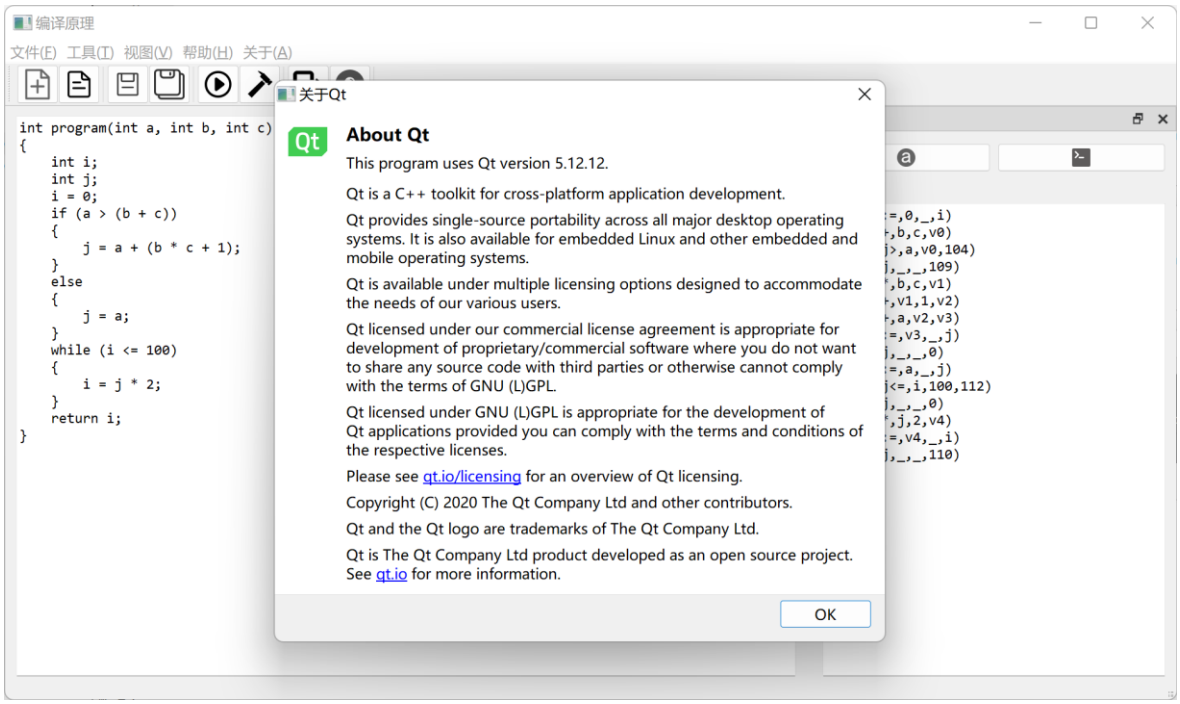
装 订 线

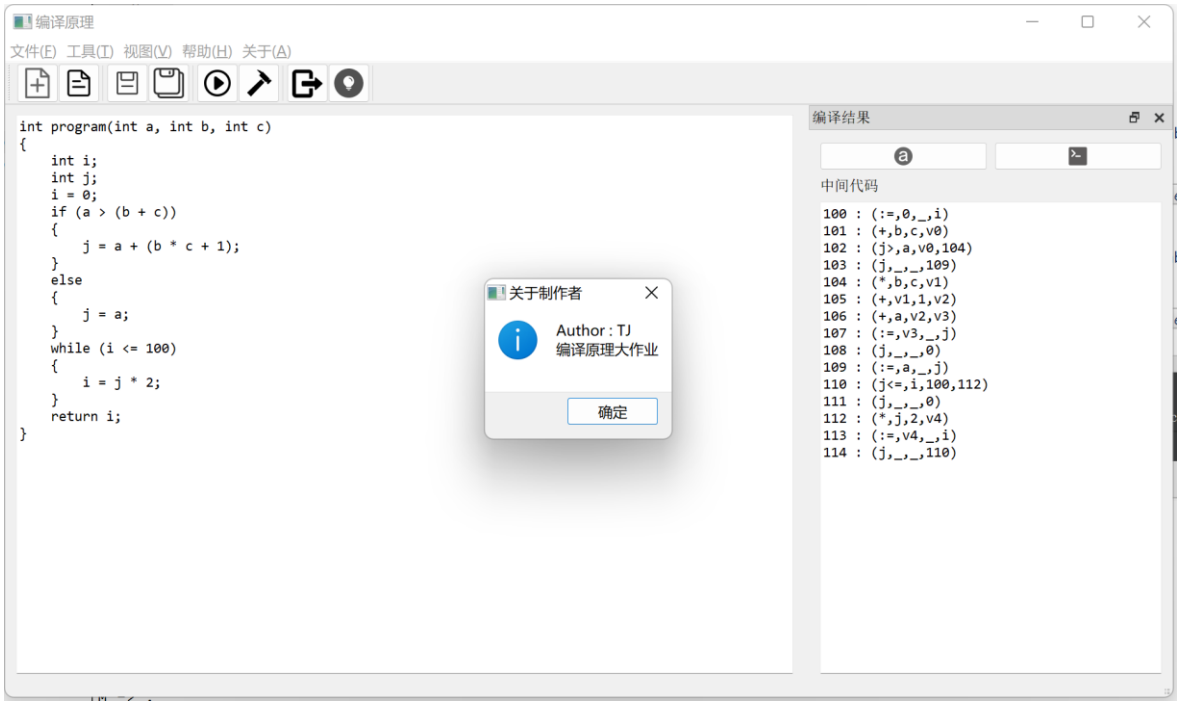
隐藏编译结果窗口：

（6） 关于部分也有关于 Qt 和关于作者



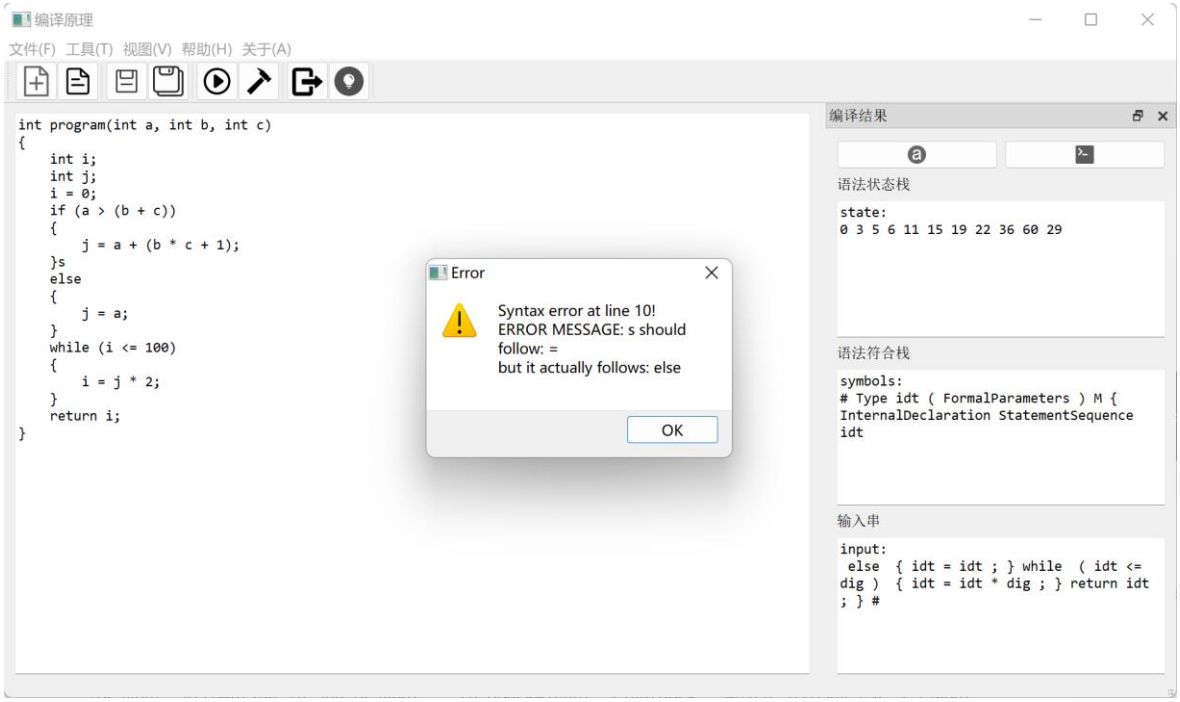
装 订 线


编译原理
文件(F) 工具(T) 视图(V) 帮助(H) 关于(A)
关于Qt
About Qt
This program uses Qt version 5.12.12.
Qt is a C++ toolkit for cross-platform application development.
Qt provides single-source portability across all major desktop operating systems. It is also available for embedded Linux and other embedded and mobile operating systems.
Qt is available under multiple licensing options designed to accommodate the needs of our various users.
Qt licensed under our commercial license agreement is appropriate for development of proprietary/commercial software where you do not want to share any source code with third parties or otherwise cannot comply with the terms of GNU (L)GPL.
Qt licensed under GNU (L)GPL is appropriate for the development of Qt applications provided you can comply with the terms and conditions of the respective licenses.
Please see qt.io/licensing for an overview of Qt licensing.
Copyright (C) 2020 The Qt Company Ltd and other contributors.
Qt and the Qt logo are trademarks of The Qt Company Ltd.
Qt is The Qt Company Ltd product developed as an open source project. See qt.io for more information.
OK
int program(int a, int b, int c)
{
int i;
int j;
i = 0;
if (a > (b + c))
{
j = a + (b * c + 1);
}
else
{
j = a;
}
while (i <= 100)
{
i = j * 2;
}
return i;
}


编译原理
文件(F) 工具(T) 视图(V) 帮助(H) 关于(A)
int program(int a, int b, int c)
{
int i;
int j;
i = 0;
if (a > (b + c))
{
j = a + (b * c + 1);
}
else
{
j = a;
}
while (i <= 100)
{
i = j * 2;
}
return i;
}
关于制作者
Author : TJ
编译原理大作业
确定
编译结果
中间代码
100 : (:=,0,_,i)
101 : (+,b,c,v0)
102 : (j>,a,v0,104)
103 : (j,_,_,109)
104 : (*,b,c,v1)
105 : (+,v1,1,v2)
106 : (+,a,v2,v3)
107 : (:=,v3,_,j)
108 : (j,_,_,0)
109 : (:=,a,_,j)
110 : (j<=,i,100,112)
111 : (j,_,_,0)
112 : (*,j,2,v4)
113 : (:=,v4,_,i)
114 : (j,_,_,110)

装 订 线

（7）错误处理弹窗



（8）退出按钮退出程序

装 订 线

五、 小组成员分工

2153603 王政尧：词法、语法分析错误提示功能，研究更为通行的高级语言的语义检查方法，对应部分报告撰写。

2153683 郭嘉： UI 界面设计与实现，对应部分报告撰写。

2153603 张博文：中间代码生成器设计与实现，对应部分报告撰写。

六、 参考资料

[1] 陈火旺 .程序设计语言编译原理：国防工业出版社，2020

装

订

线