

# C 程序形式化验证学习系统 用户使用手册

安徽中科国创高可信软件有限公司

Copyright © 2020 版权所有

# 目录

<b>1</b>	<b>软件概述</b>	<b>4</b>
1.1	软件介绍	4
1.2	背景介绍	4
<b>2</b>	<b>一般操作步骤</b>	<b>4</b>
2.1	系统结构	4
2.1.1	验证平台的服务器构成	4
2.1.2	验证平台的使用环境	5
2.2	系统使用	5
2.3	用户注册	6
2.4	匿名用户登入	8
2.5	注册用户登入	9
2.6	用户密码重置	11
2.7	登出	13
2.8	注册用户信息修改	13
2.9	注册用户密码修改	14
2.10	文件添加	14
2.11	本地文件装载	15
2.12	注册用户所有文件打包下载	17
2.13	文件重命名	17
2.14	文件下载	18
2.15	文件删除	18
2.16	文本编辑器	19
2.17	程序文件提交服务器	21

2.18	验证开始 .....	22
2.19	撤销验证 .....	23
2.20	验证结果查看 .....	23
2.21	帮助 .....	27
<b>3</b>	<b>验证结果说明 .....</b>	<b>31</b>
3.1	错误检查 .....	31
3.2	验证结果 .....	32
3.2.1	函数前后条件 .....	32
3.2.2	循环不变式 .....	33
3.2.3	大验证条件 .....	34
3.2.4	小验证条件 .....	35
3.2.5	函数调用 .....	36
3.3	形状图演算结果 .....	37
3.3.1	主要验证点对应的形状图 .....	37
3.3.2	语句演算结束时的形状图 .....	40
3.4	虚拟变量的命名规则 .....	42
3.4.1	在程序变换中引入的虚拟变量 .....	42
3.4.2	量化断言的约束变元变换引入的虚拟变量 .....	42
3.4.3	范域词名称变换引入的虚拟变量 .....	43
3.4.4	赋值语句演算时引入的虚拟变量 .....	43
3.4.5	没有 const 修饰符的指针形参, 其指向数据块命名时引入的虚拟变量 .....	44
3.4.6	证明循环语句终止性时引入的虚拟变量 .....	44
3.4.7	字符串常量、字符串谓词拆分时引入的虚拟变量 .....	44
3.4.8	部分内建构造在演算中变换引入的虚拟变量 .....	45
3.4.9	递归谓词或者逻辑函数多次展开时引入的虚拟变量 .....	45

3.4.10 函数调用受 assigns 影响的表达式代换引入的虚拟变量.....	46
3.4.11 形状图断言变换引入的虚拟变量.....	46
3.4.12 形状图构建引入的虚拟变量.....	46
3.4.13 堆指针数据域赋值时引入的虚拟变量.....	47
3.4.14 删除节点 (free 语句等) 时引入的虚拟变量.....	48
3.4.15 函数调用时堆指针形参引入的虚拟变量.....	48
3.4.16 函数调用时堆指针数据域实参引入的虚拟变量.....	48
3.4.17 smt2 翻译时引入的虚拟变量.....	49

# 1 软件概述

## 1.1 软件介绍

安徽中科国创高可信 C 程序形式化验证学习系统（简称**科创验证学习平台**，或**验证平台**）是一个 C 语言程序的形式化验证学习平台，它包含了验证程序的编辑、提交、验证、验证结果展示，以及验证文件的上传、下载，和用户信息维护等功能。在线的帮助提供了安全 C 语言使用手册等使用文档和验证程序样例供大家参考、学习。

## 1.2 背景介绍

形式化验证是目前已知的保证程序正确的最可靠的方法。借助于对程序行为的形式化描述、定理证明和逻辑推理的数学证明手法，使得通过验证的程序满足设计者的要求，安全可靠。形式化验证因其对验证者的要求高，验证难度大，缺乏有效的验证工具等原因，使得目前的验证工作主要用于如操作系统的内核、系统控制和嵌入式应用程序的核心算法等关键程序的验证。科创验证学习平台基于安全 C 语言和安全 C 规范语言（SCSL），采用正向的演绎推理方法，为初学者学习和掌握形式化验证方法与技术提供了一个便捷的学习平台，为编程人员提高软件代码质量、保障软件的高可靠性和信息安全提供新的方法和技术。

# 2 一般操作步骤

## 2.1 系统结构

### 2.1.1 验证平台的服务器构成

1. 数据库服务器  
用于存放用户登录信息、验证程序和验证结果。
2. 应用程序服务器  
用于验证任务的分发、负载均衡和与前端的交互。
3. 验证服务器  
用于程序的验证。

## 2.1.2 验证平台的使用环境

用户通过浏览器使用科创验证服务，目前支持的浏览器为 Chrome（版本 77 及以上）和 Firefox（版本 69 及以上）。使用 Microsoft Edge、360 极速浏览器、QQ 浏览器和搜狗高速浏览器也可以访问验证平台，但不做推选。

## 2.2 系统使用

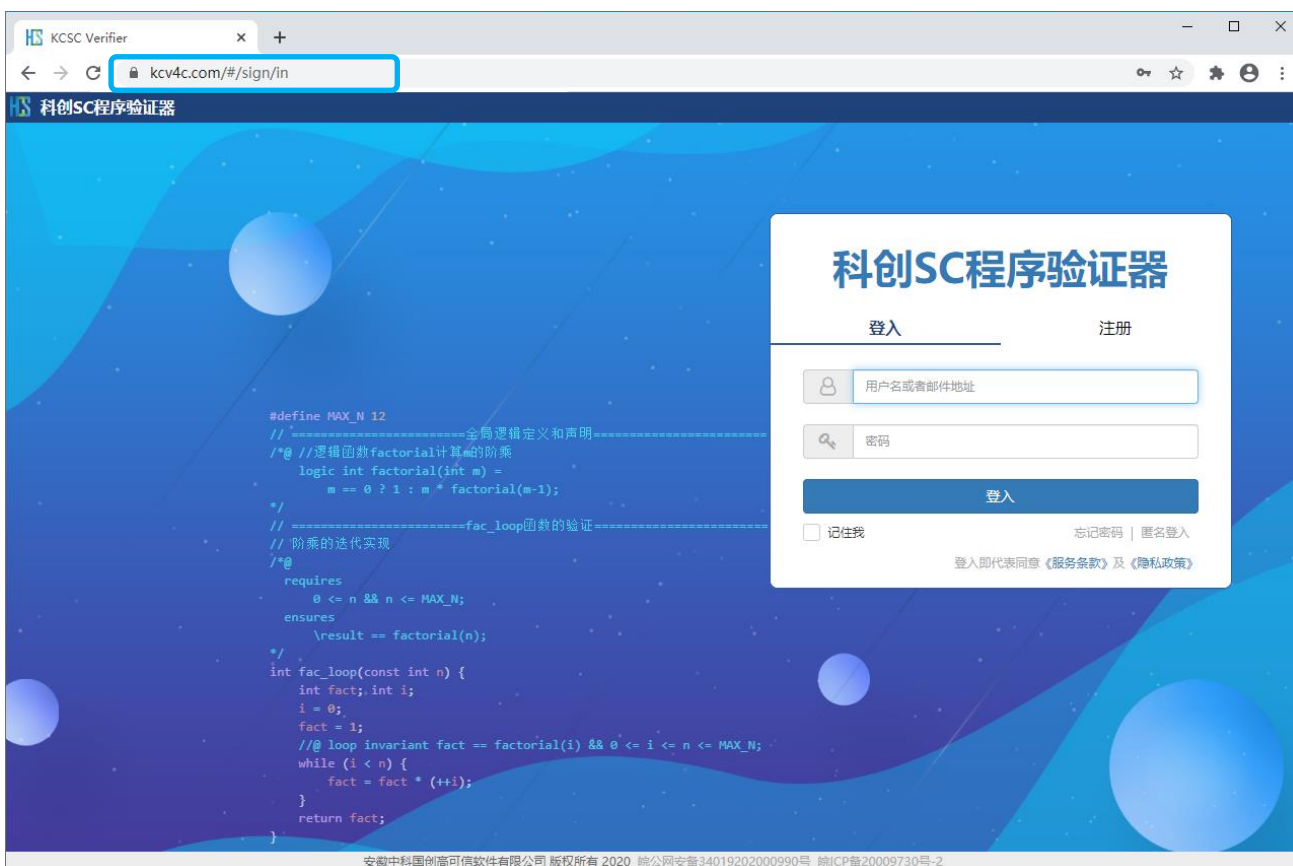
打开浏览器，使用 URL 地址 `https://www.kcv4c.com/` 进行访问。

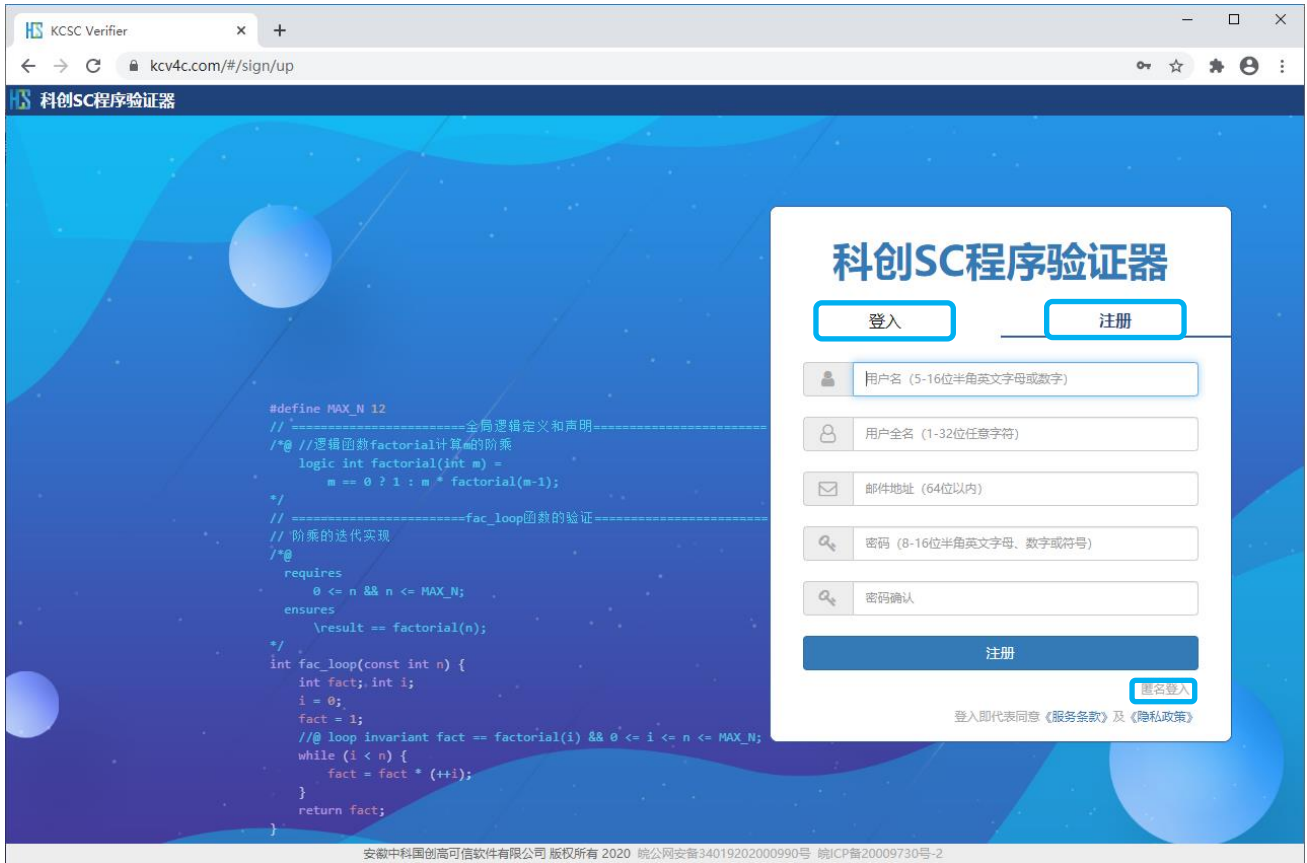
进入系统后，首先打开的是用户登录界面。已注册的用户输入【用户名/登录邮箱】和【密码】进行登入，未注册的用户可以选择先注册再登入；或选择【匿名登入】直接登入。

首页上可以进行用户的登入、注册、密码重置以及匿名登入。

系统为注册用户提供了信息保存功能，使用户拥有更好的学习体验。

**⚠注意：**在网络环境拥挤的情况下，验证平台的应答有时会比较慢。此时，可尝试重新加载页面（刷新）加以改善。





## 2.3 用户注册

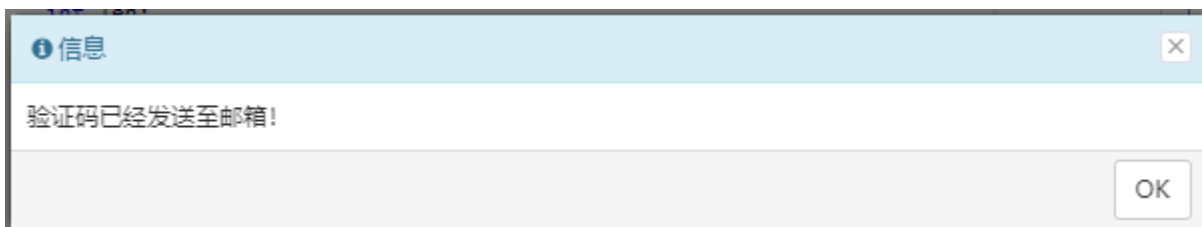
1、系统首页中，切换至【注册】区域，输入对应用户信息，进行注册；需要用户提供最基本的信息，如：注册账号，用户名，密码，以及邮箱。



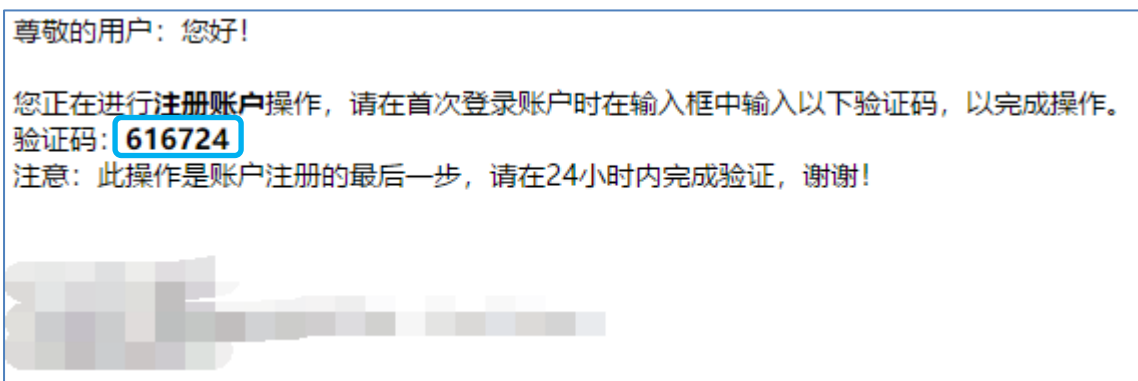
2、若填写信息不符合要求，在点击【注册】时，会提示不符合填写要求的项目，以及对应项目需要的信息。



3、填写符合要求的信息后，点击【注册】按钮进行注册。系统将发送一封包含验证信息的邮件至用户填写的邮箱。这时注册的账户处于待激活状态。用户在用该账户首次登入时，需要提供验证信息中的验证码进行激活。注册按钮点击后的效果如下：



验证邮件样例如下：



⚠ 注：

- 1) 用户需要提供可使用的邮箱地址，如果地址不正确或者无法正常接收邮件，则会影响后续的正常登入以及密码修改和找回操作。
- 2) 因为在登入时邮箱地址等同于用户名，所以一个邮箱地址只能注册一个用户账户。



## 2.4 匿名用户登入

1、匿名用户可以有两种方式进行登入，按钮的具体位置如下：

方式一：登入界面

科创SC程序验证器

登入 注册

用户名或者邮件地址

密码

登入

记住我 [忘记密码](#) [匿名登入](#)

登入即代表同意《服务条款》及《隐私政策》

方式二：注册界面

科创SC程序验证器

登入 注册

用户名 (5-16位半角英文字母或数字)

用户全名 (1-32位任意字符)

邮件地址 (64位以内)

密码 (8-16位半角英文字母、数字或符号)

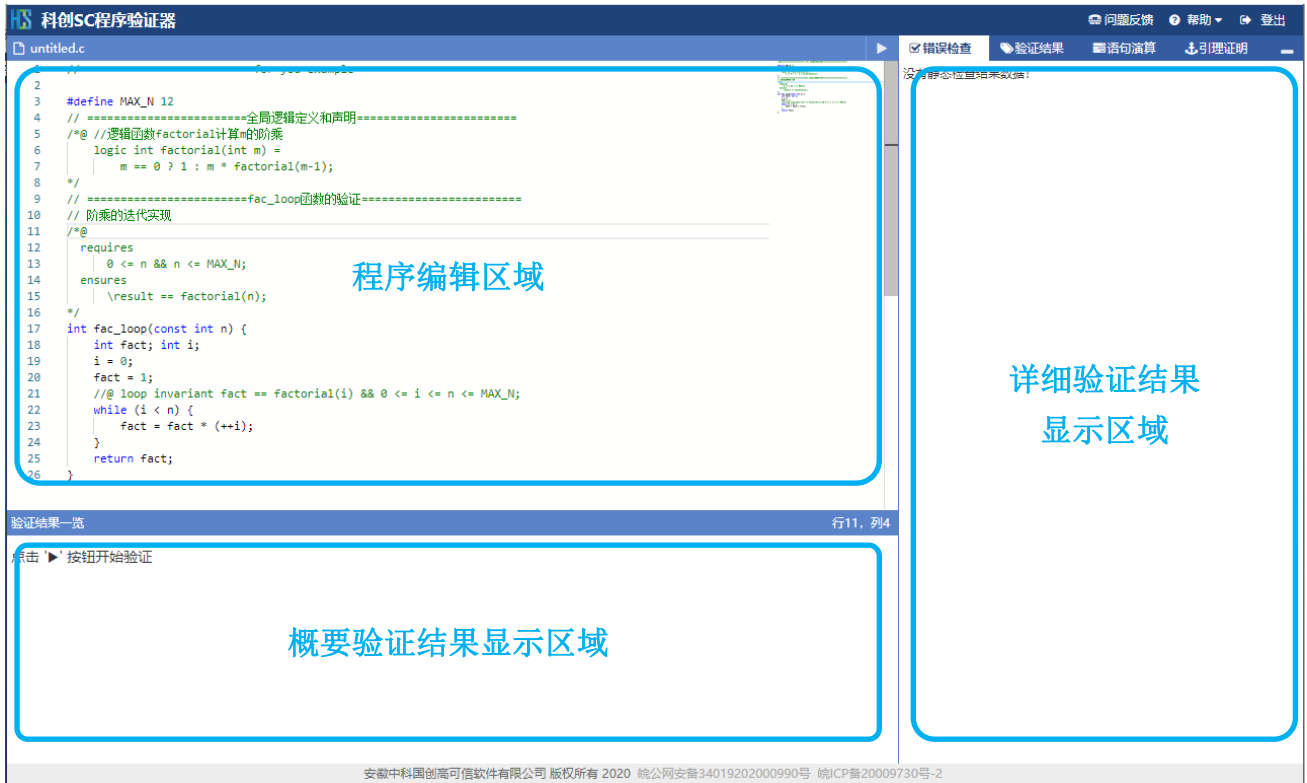
密码确认

注册

[匿名登入](#)

登入即代表同意《服务条款》及《隐私政策》

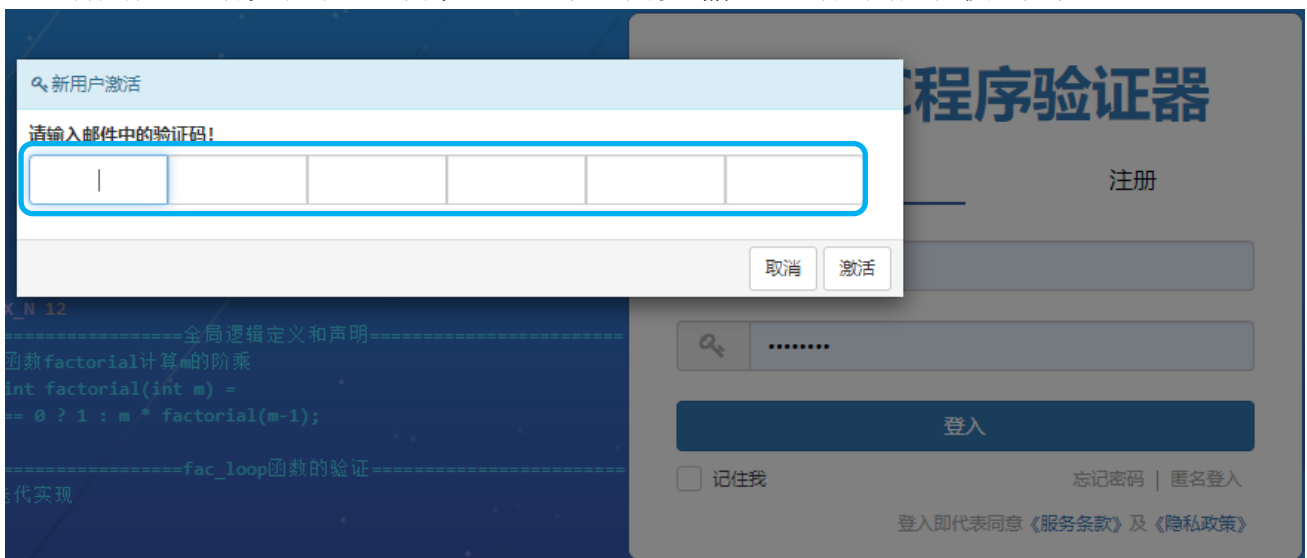
匿名登入时的界面如下，用户可以在线编辑程序，进行验证，并对验证结果进行确认。



匿名登入可以省去注册时的一点时间，但因为没有任何文件保存功能，验证程序和验证结果都无法保存到系统中。建议大家注册使用，以获得更好的学习体验。

## 2.5 注册用户登入

新用户注册完成后，初次登入时，需要输入注册邮箱中收到的验证码。



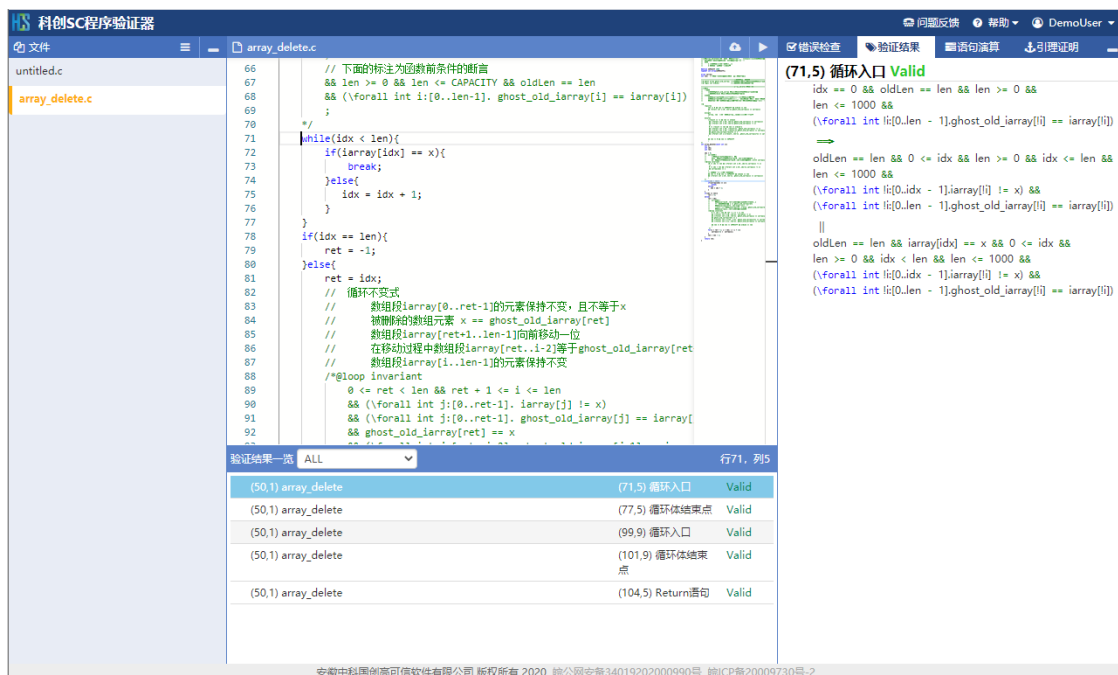
成功激活后，进入程序验证主画面。对于已经激活的用户，登入时不再要求提供验证码。

注册用户登入成功后的界面如下。



注册用户首次登入主画面时，系统会为用户默认创建一个名为“untitled.c”的样例程序文件。

以后登入时，系统将显示保存的文件列表，打开上次登出时选中的验证程序文件和该文件最近一次的验证结果，如下所示。

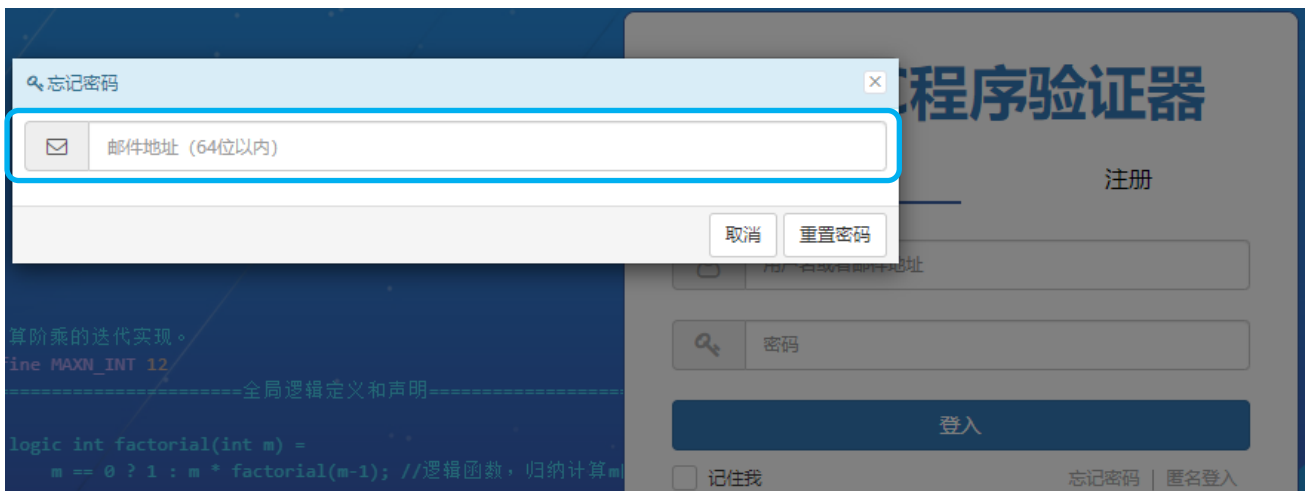


## 2.6 用户密码重置

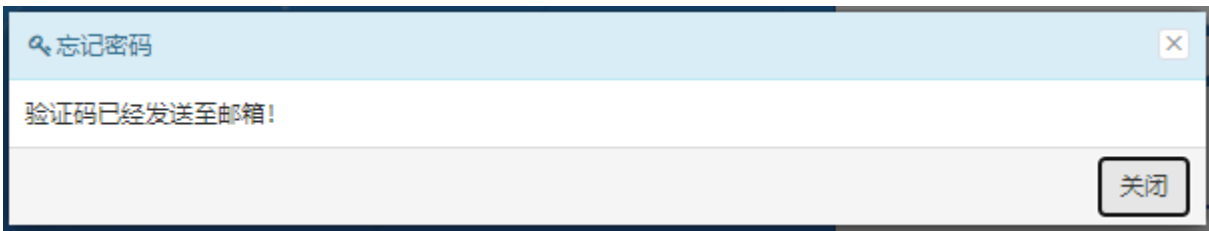
登入界面中点击【忘记密码】按钮，可以进行用户密码的重置。



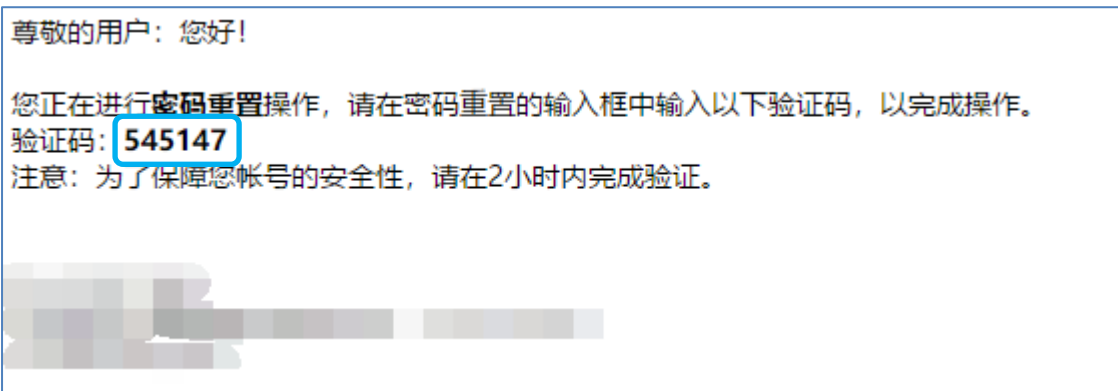
填写注册时提供的邮箱地址，用来接收密码重置的验证信息。



验证信息发送成功后，查看邮箱中相关的验证码信息。



用户密码重置相关的验证邮件样例如下：

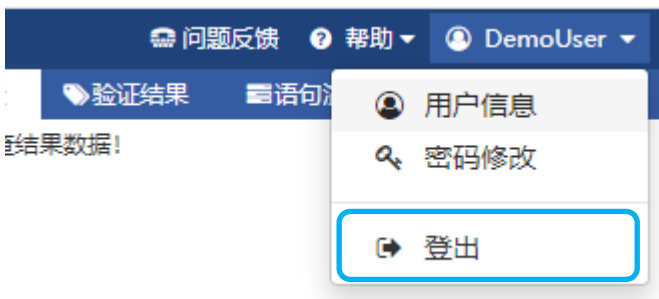


输入验证邮件中的验证码，以及新密码，点击确认按钮，完成密码重置操作。



## 2.7 登出

注册用户登入成功后，点击页面右上角的用户名区域，在弹出的对话框中选择【登出】按钮，退出当前登入的账户。

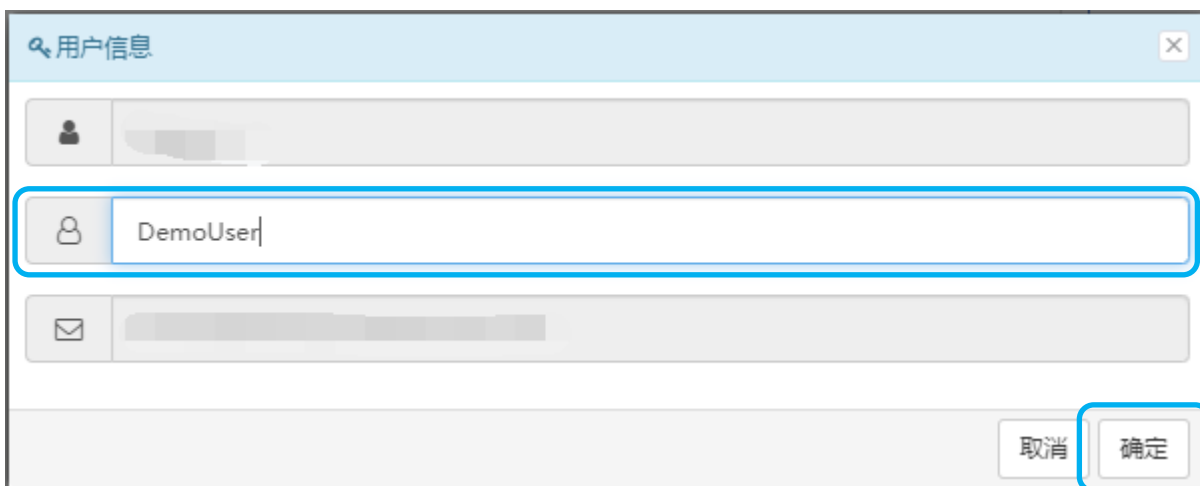


## 2.8 注册用户信息修改

注册用户登入成功后，点击页面右上角的用户名区域，在弹出的对话框中选择【用户信息】按钮。



用户信息对话框中进行用户名修改，点击确定按钮保存相关信息。

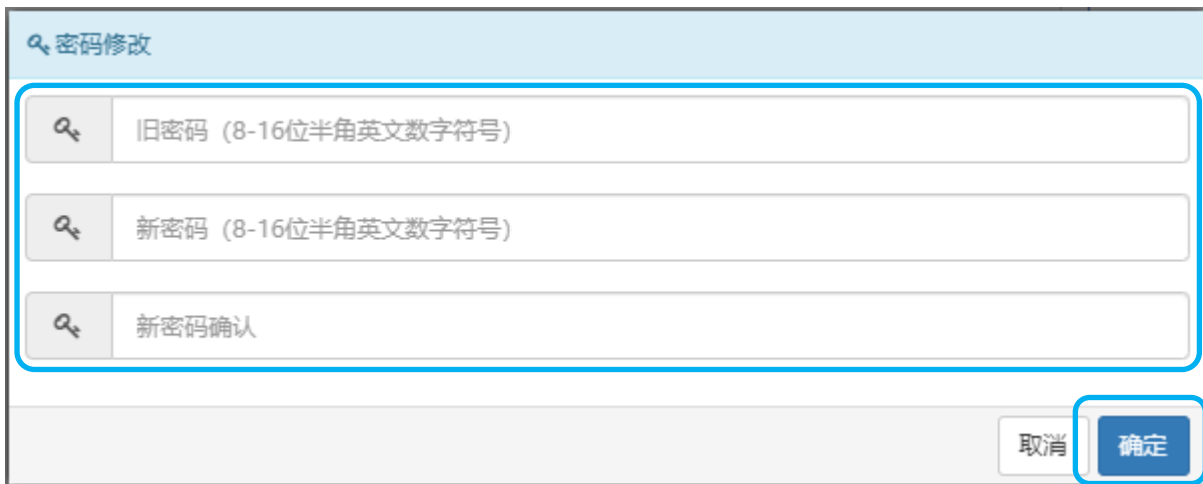


## 2.9 注册用户密码修改


注册用户登入成功后，点击页面右上角用户名区域，在弹出的对话框中选择【密码修改】按钮。

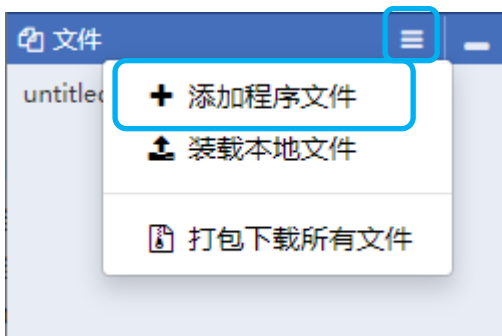


密码修改对话框中进行密码的修改，点击确定按钮保存相关信息。



## 2.10 文件添加

注册用户登入的情况下，点击文件列表上方（左侧）的文件处理图标 ，在弹出的文件菜单中点击【添加程序文件】，创建一个新的验证程序文件。

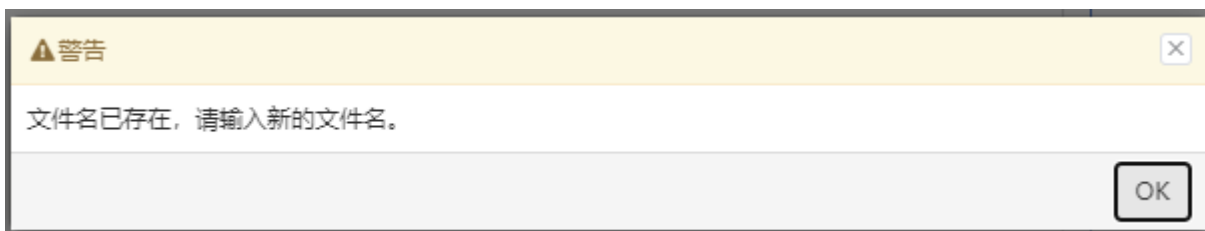


对新添加的文件进行命名，命名完成后点击保存按钮，进行程序文件的添加。


注意：文件名由英字母、数字、下划线（\_）和英文句点（.）构成，长度不超过 64 个字符。（目前（.）只用于文件名后缀，如“.c”。）

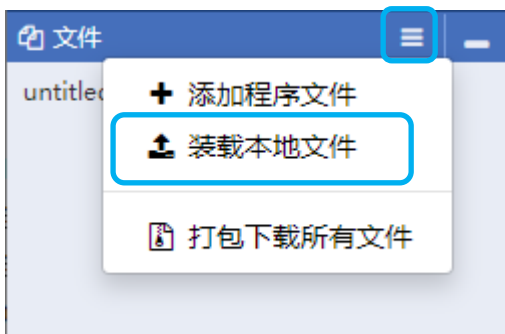


新添加的文件名不能与文件列表中现有文件名同名。重名时系统给出提示信息，要求重新命名。



## 2.11 本地文件装载

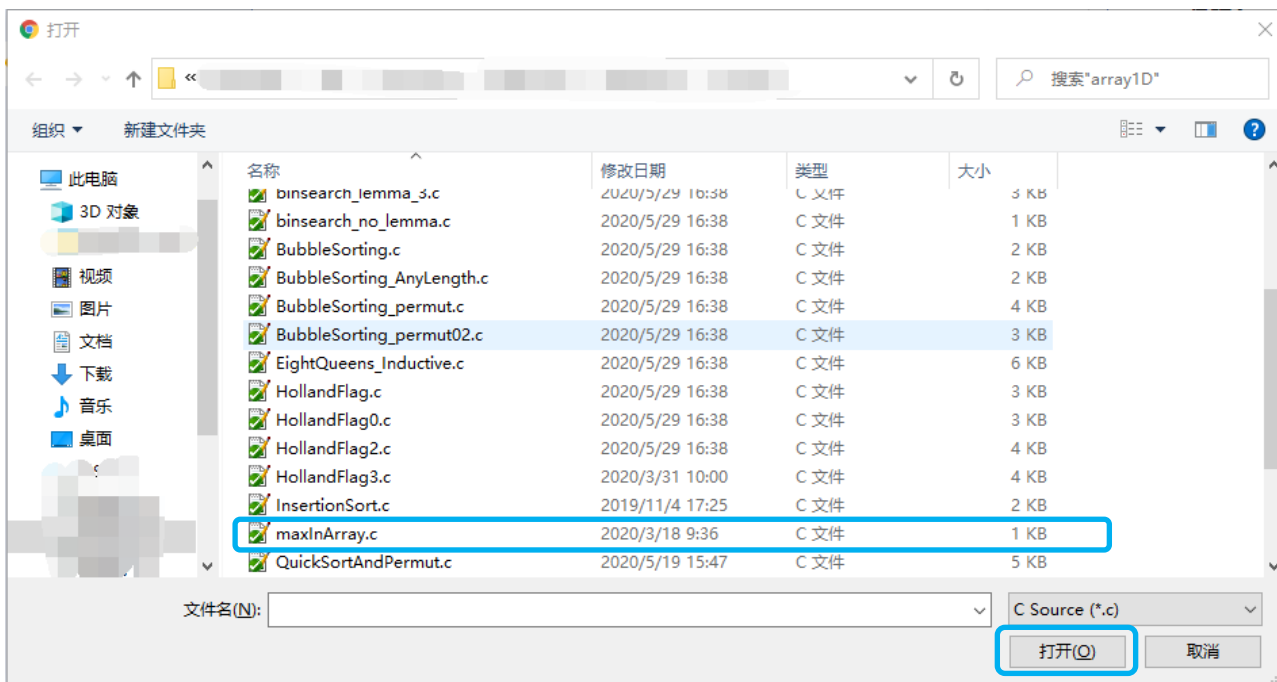
注册用户登入的情况下，点击左上侧的文件处理图标 ，在弹出的文件菜单中点击【装载本地文件】，进行本地程序文件的上传和添加。



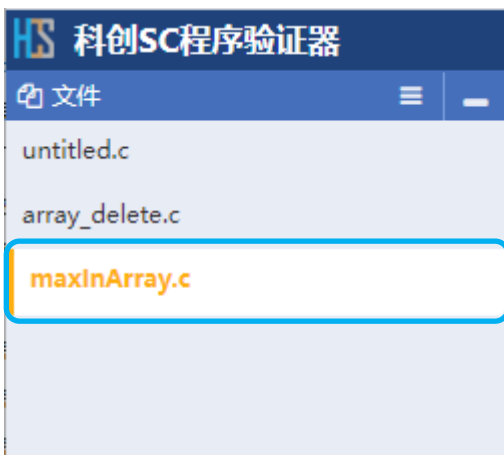
通过文件选择对话框选择需要上传的本地文件。

各浏览器显示的文件选择对话框可能不尽相同，以下以 Chrome（谷歌）浏览器为例，说明相关内容。

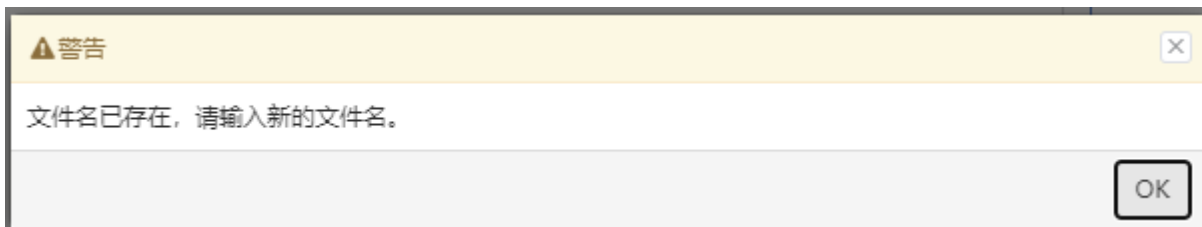




点击【打开】按钮，选择需要上传的文件，这里选择“maxInArray.c”作为演示例。本地文件上传处理成功后，文件列表中则会显示相应的记录。

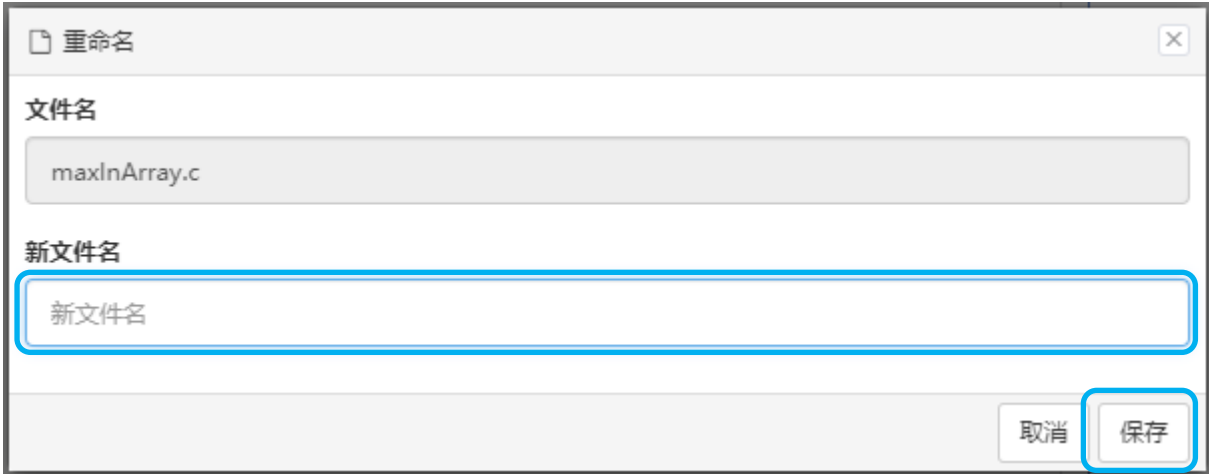


如果上传的文件名和用户已有的文件的名称相同，提示文件名重复




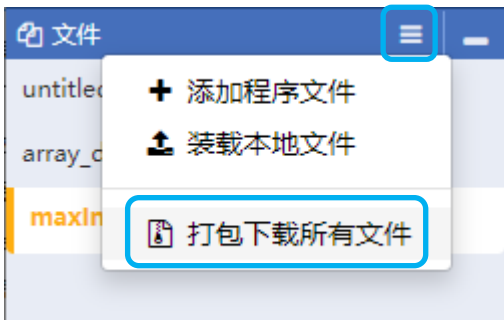
对于出现重复的情况，需要提供新的文件名，通过对新添加的文件进行重新命名，完成文件的上传和添加。

文件上传成功时，新添加的文件在文件列表中当前光标所在的位置。以后界面刷新时，文件列表将按字母序升序排序。




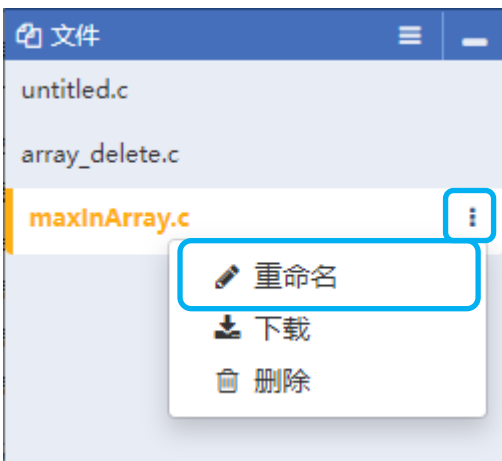
## 2.12 注册用户所有文件打包下载

注册用户登入的情况下，点击左上侧的文件处理图标 ，在弹出的文件菜单中点击【打包下载所有文件】，用户当前文件列表中的所有程序文件将打包下载到浏览器的缺省下载目录。

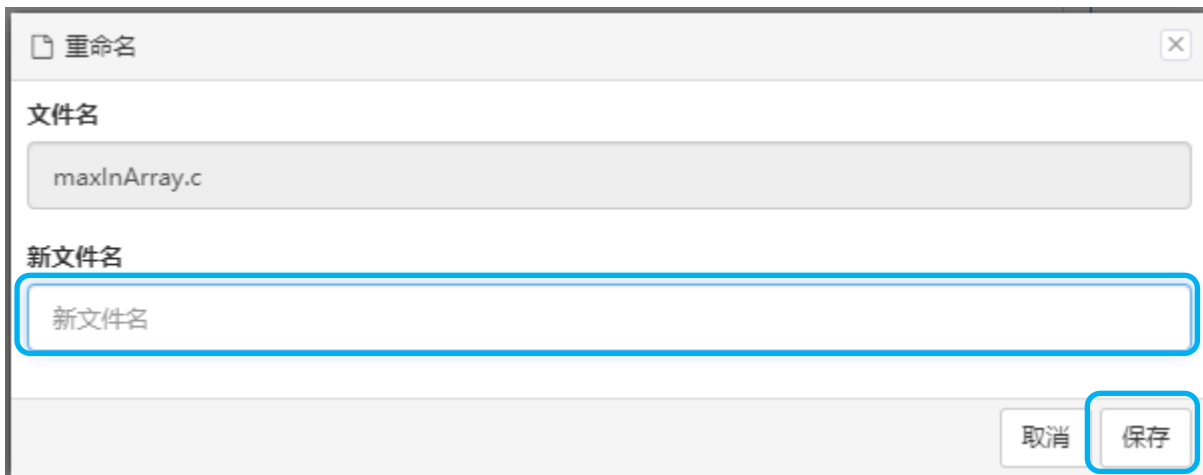


## 2.13 文件重命名


鼠标悬停在文件列表中的文件名上，左侧出现文件操作图标 。点击图标，在弹出的文件菜单中点击【重命名】按钮，进行文件的重命名操作。

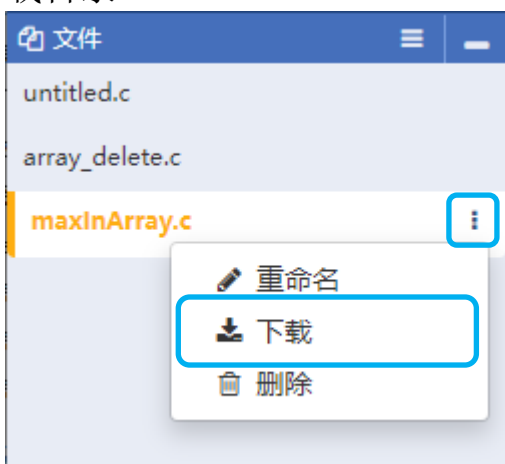


重命名对话框中，输入新的文件名，点击【保存】按钮，进行当前选中文件的重命名。文件名的长度不超过 64 字符，命名规则可参见 2.11 节。




## 2.14 文件下载

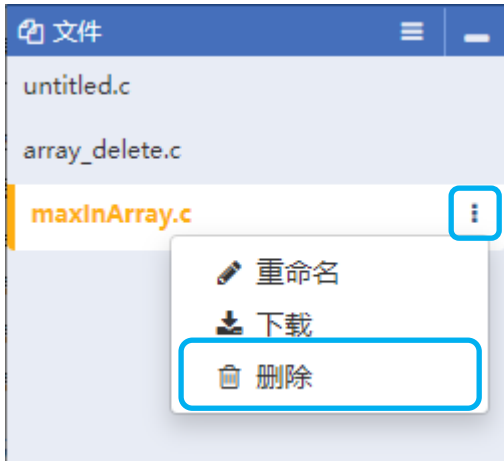
鼠标悬停在文件列表中的文件上，点击文件操作图标 ，在弹出的文件菜单中点击【下载】按钮，进行文件的下载操作。文件将下载到浏览器的缺省下载目录。



## 2.15 文件删除

鼠标悬停在文件列表中的文件上，点击文件操作图标 ，在弹出的文件菜单中点击【删除】按钮，进行文件的删除操作。

 注：文件删除后无法进行恢复，请在详细确认无误后再进行删除操作。

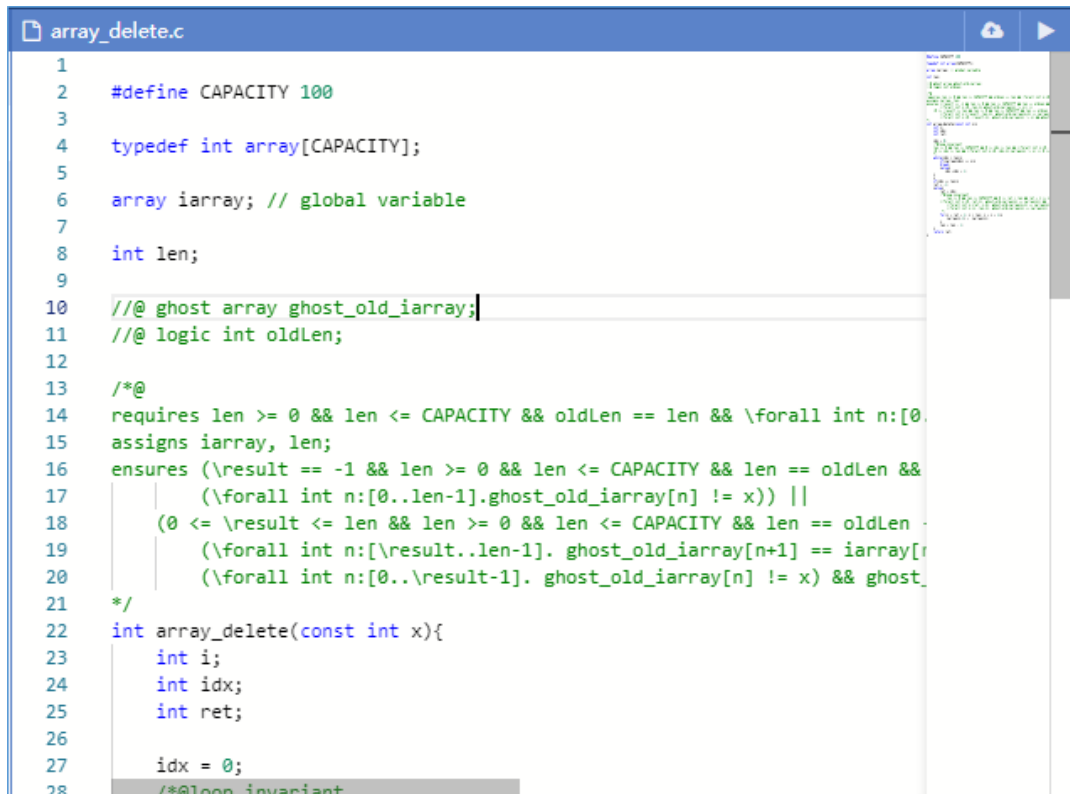


确认删除文件

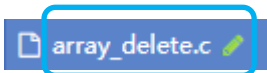


## 2.16 文本编辑器

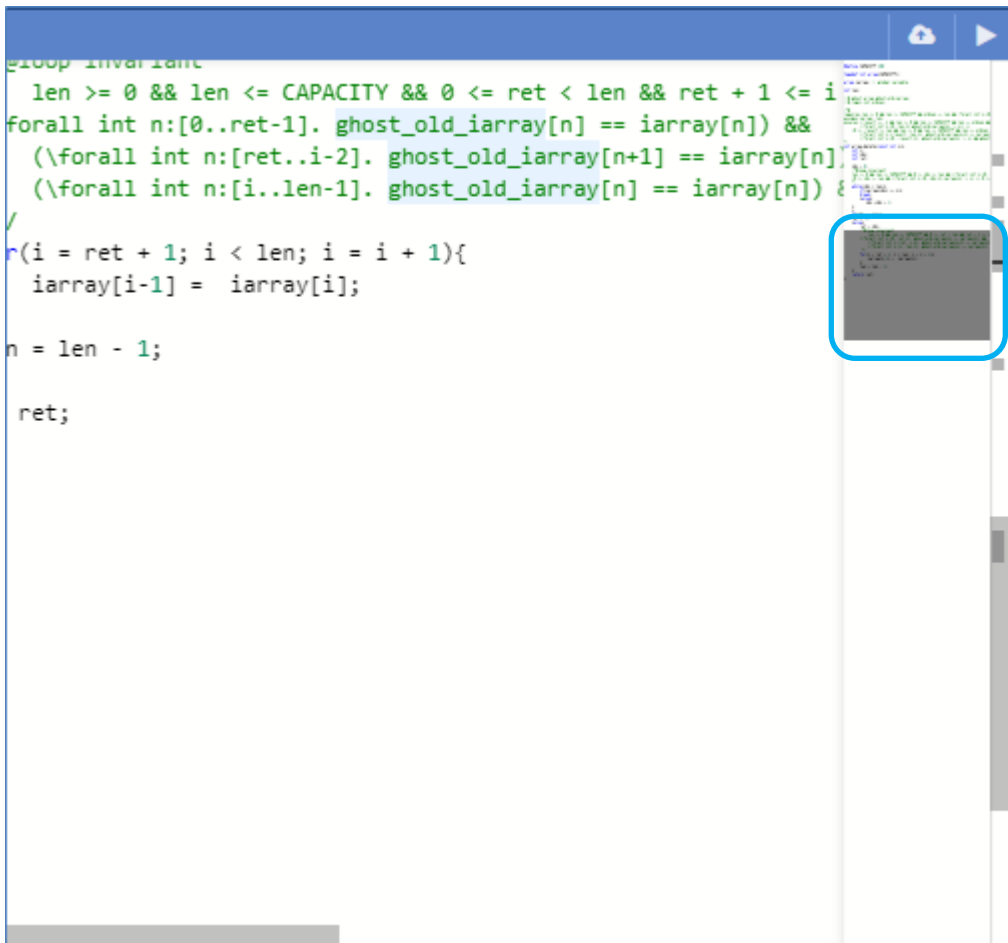
主画面的中间区域为验证程序的编辑区，用户可以创建新文件，或对添加、上传的文件进行编辑。



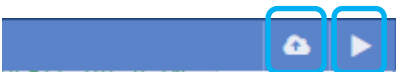
编辑区域的左上角显示文件名，以及当前文档的修改状态。



编辑区域右边为整体代码的缩略图，通过拖拽遮罩的小块，可以快速滚动到对应位置的内容。

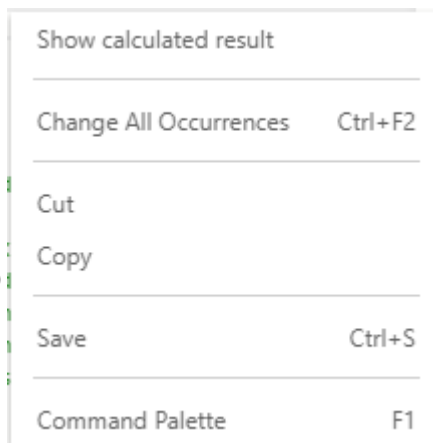


编辑区域的右上角为文件保存（提交服务器）按钮和验证按钮。



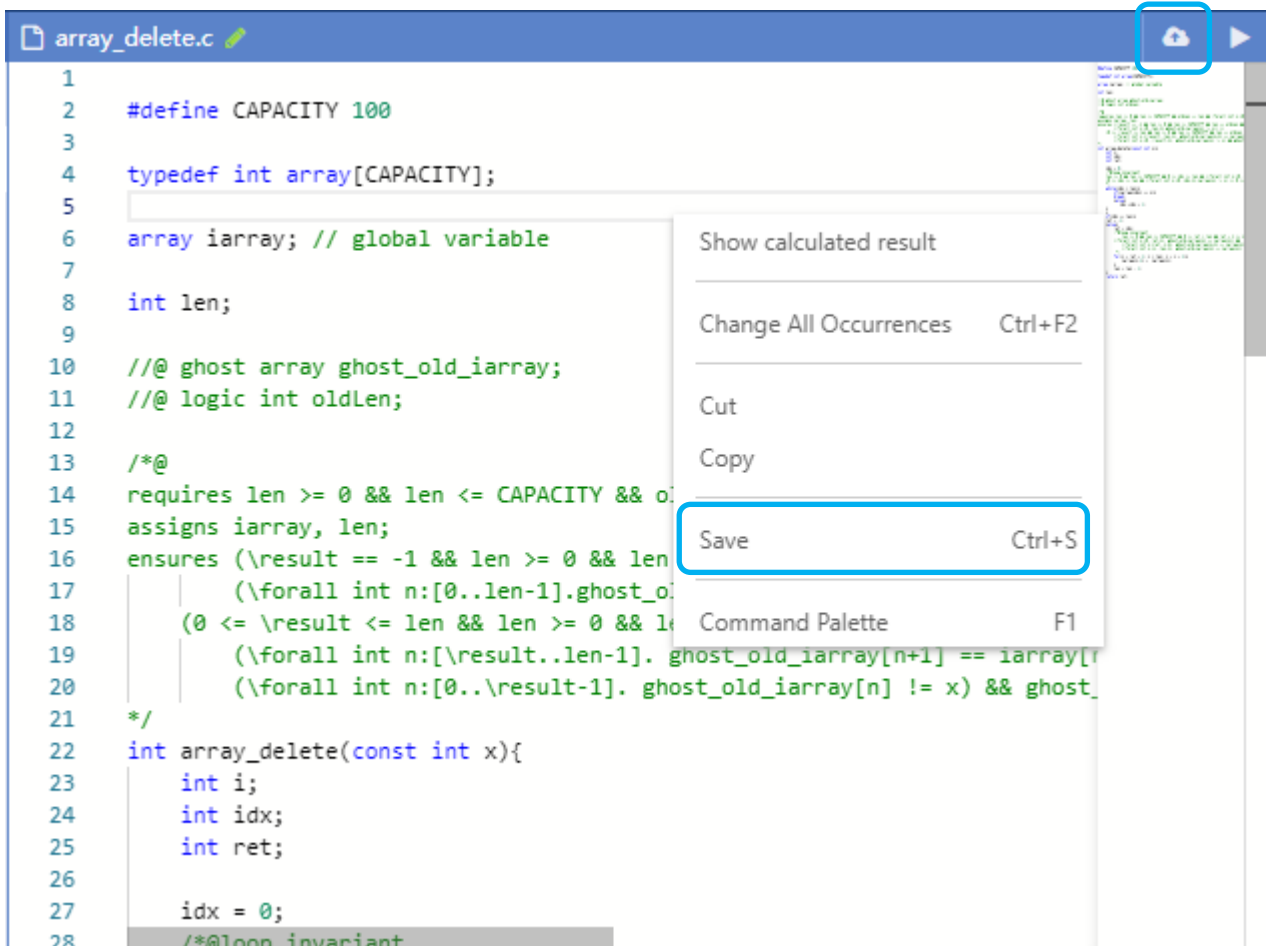
通过编辑区域的右键菜单选项，进行一些常规的如剪切、拷贝修改动作。

还有，一些常用的编辑快捷键，如复制（Ctrl+C）、剪切（Ctrl+X）、粘贴（Ctrl+V）、撤消（Ctrl+Z）、恢复（Ctrl+Y）等都可以使用。

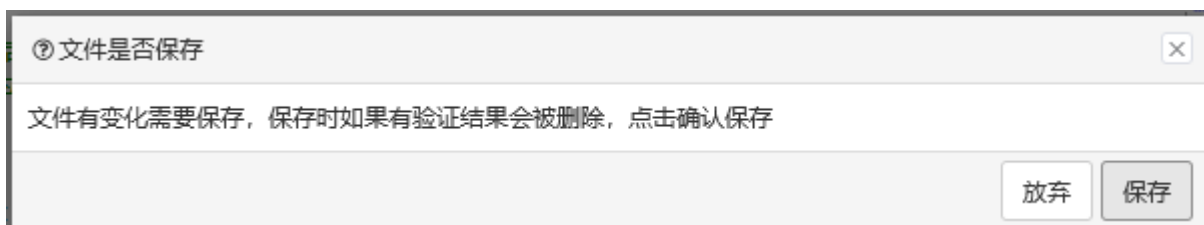


## 2.17 程序文件提交服务器

修改后的程序文件，通过点击【保存】按钮，或者编辑区域的邮件菜单中的【Save】选项，或者使用快捷键“Ctrl + S”进行文件的上传保存。



若文件内容已修改，且当前程序存在验证结果，则系统将提示保存动作会引起验证结果的删除，确认是否保存。




选择【保存】则文件上传保存，同时清空最近一次验证结果。选择【放弃】则放弃本次文件保存的操作。

若文件未修改，则系统显示“文件无变化”的信息。


若文件已修改，且希望放弃修改，则可通过撤消（Ctrl+Z）操作退回到修改前的状态，或切换到其他文件时放弃保存。

## 2.18 验证开始

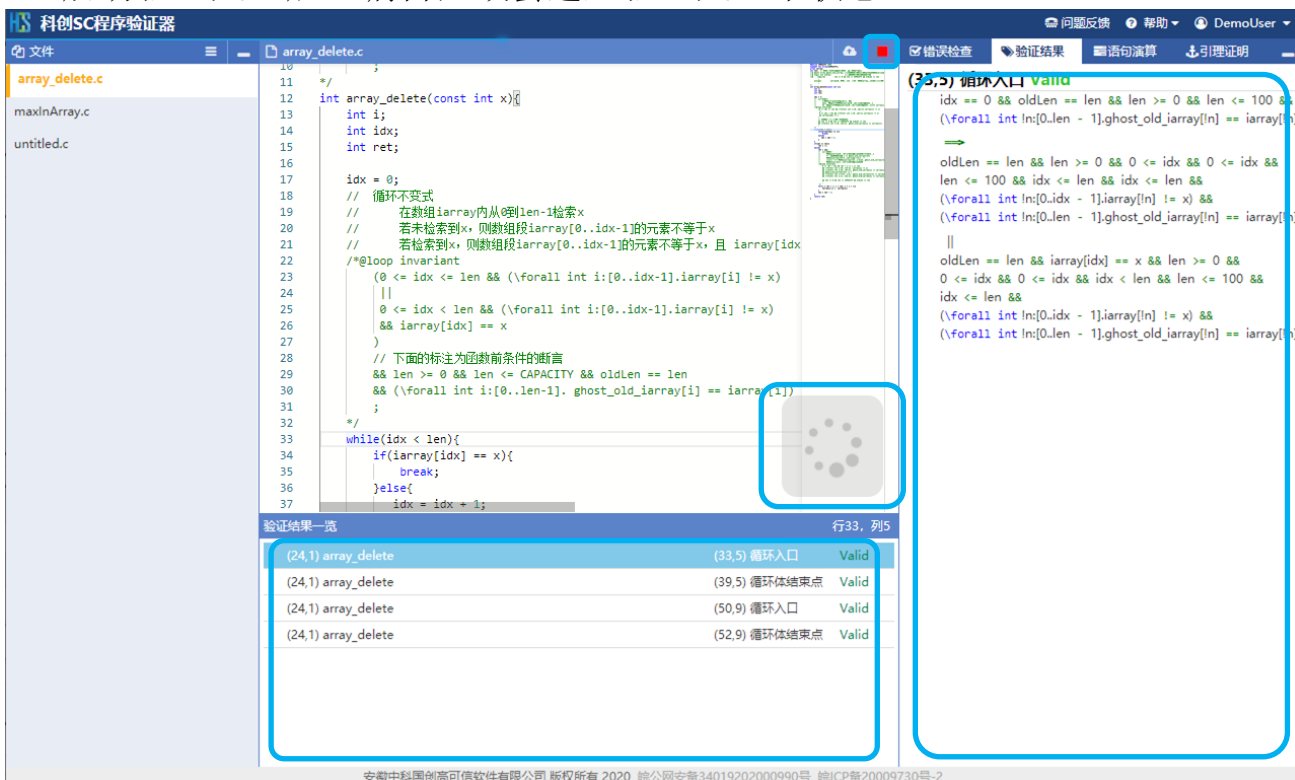
点击编辑区域右上角的【验证】按钮，启动程序验证。

在点击验证按钮时，如果当前选中的文件存在变更，1) 若不存在验证结果，系统将提示是否保存当前文件内容。2) 若已经存在验证结果，系统将提示保存动作会引起验证结果的删除，确认是否保存。





对于 1)，点击【保存】按钮，保存文件并启动验证；点击【放弃】按钮，则放弃保存并启动验证。对于 2) 点击【保存】按钮，保存文件、清空验证结果并启动验证。点击【放弃】按钮，则放弃保存、清空验证结果并启动验证。若放弃本次验证，则选择  关闭提示窗口。

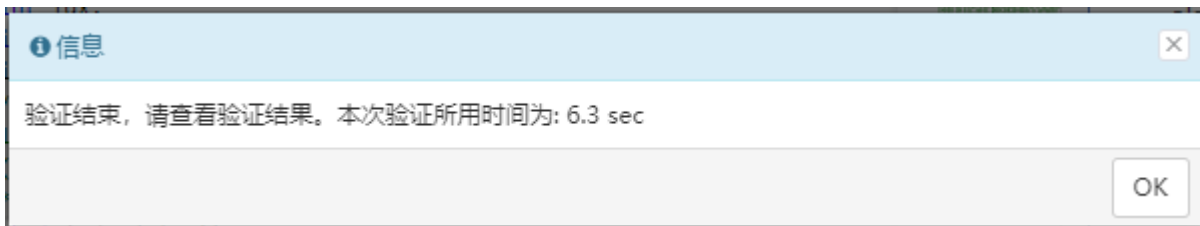
启动验证处理后，编辑区域会进入验证处理中状态。



验证处理中状态，主画面做以下变化：

1. 验证按钮由“开始验证”状态 ，切换至“撤销验证”状态 ；
2. 验证结果展示区的内容被清空；编辑区域切换至为不可修改；
3. 验证结果概要区域加载当前验证中已完成的验证结果，并选中第一条；
4. 验证结果详细区域加载概要栏中选中记录的验证详细结果。

验证完成时，系统提示验证已完成。主画面恢复至验证前的状态。



## 2.19 撤销验证

系统进入验证中状态时，通过点击编辑区域右上方的【撤销验证】按钮 ，中断并取消验证处理。



验证撤销后，主画面回到验证前的状态，同时加载撤销前已经验证完成的部分结果信息。

## 2.20 验证结果查看

验证结果有概要验证结果和详细验证结果两个区域组成。

1. 概要结果显示汇总后的验证结果

通过选中概要结果中的记录，切换详细验证结果中的信息。

切换验证结果右侧的下拉框选项，可对结果按函数名进行筛选显示。

验证结果一览		ALL	行33, 列5
(24,1) array_delete	(33,5) 循环入口	Valid	
(24,1) array_delete	(39,5) 循环体结束点	Valid	
(24,1) array_delete	(50,9) 循环入口	Valid	
(24,1) array_delete	(52,9) 循环体结束点	Valid	
(24,1) array_delete	(55,5) Return语句	Valid	



2. 详细结果由四个部分组成，错误检查、验证结果、语言演算和引理证明。具体显示如下：

错误检查

<input checked="" type="checkbox"/> 错误检查	<input type="checkbox"/> 验证结果	<input type="checkbox"/> 语句演算	<input type="checkbox"/> 引理证明
(15,22)	Error	expected an lvalue expression	
(44,17)	Error	invalid operands to binary expression ('int' and 'boolean')	
(51,13)	Warning	implicitly declaring library function 'exit' with type 'void (int) __attribute__((noreturn))'	
(51,13)	Note	include the header <stdlib.h> or explicitly provide a declaration for 'exit'	
(51,13)	Error	the function protocol must include the exits clause	

验证结果

<input checked="" type="checkbox"/> 错误检查	<input checked="" type="checkbox"/> 验证结果	<input type="checkbox"/> 语句演算	<input type="checkbox"/> 引理证明
<b>(51,9) 循环入口 Valid [2]</b>			
<b>分支1 Valid</b>			
<pre>ret == idx &amp;&amp; i == ret+1 &amp;&amp; oldLen == len &amp;&amp; iarray[idx] == x &amp;&amp; len &gt;= 0 &amp;&amp; 0 &lt;= idx &amp;&amp; idx &lt; len &amp;&amp; len &lt;= 100 &amp;&amp; idx &lt;= len &amp;&amp; idx != len &amp;&amp; (\forall int !n:[0..idx-1],iarray[!n] != x) &amp;&amp; (\forall int !n:[0..len-1],ghost_old_iarray[!n] == iarray[!n]) ⇒ oldLen == len &amp;&amp; ghost_old_iarray[ret] == x &amp;&amp; len &gt;= 0 &amp;&amp; 0 &lt;= ret &amp;&amp; i &lt;= len &amp;&amp; ret &lt; len &amp;&amp; len &lt;= 100 &amp;&amp; ret+1 &lt;= i &amp;&amp; (\forall int !n:[0..ret- 1],iarray[!n] != x) &amp;&amp; (\forall int !n:[0..ret- 1],ghost_old_iarray[!n] == iarray[!n]) &amp;&amp; (\forall int !n:[i..len-1],ghost_old_iarray[!n] == iarray[!n]) &amp;&amp; (\forall int !n:[ret..i-2],ghost_old_iarray[!n+1] == iarray[!n])</pre>			
<b>分支2 Valid</b>			
<pre>ret == idx &amp;&amp; i == ret+1 &amp;&amp; oldLen == len &amp;&amp; iarray[idx] == x &amp;&amp; len &gt;= 0 &amp;&amp; 0 &lt;= idx &amp;&amp; idx &lt; len &amp;&amp; len &lt;= 100 &amp;&amp; idx &lt;= len &amp;&amp; idx != len &amp;&amp; (\forall int !n:[0..idx-1],iarray[!n] != x) &amp;&amp; (\forall int !n:[0..len-1],ghost_old_iarray[!n] == iarray[!n]) ⇒ oldLen == len &amp;&amp; ghost_old_iarray[ret] == x &amp;&amp; len &gt;= 0 &amp;&amp; 0 &lt;= ret &amp;&amp; i &lt;= len &amp;&amp; ret &lt; len &amp;&amp; len &lt;= 100 &amp;&amp; ret+1 &lt;= i &amp;&amp; (\forall int !n:[0..ret- 1],iarray[!n] != x) &amp;&amp; (\forall int !n:[0..ret- 1],ghost_old_iarray[!n] == iarray[!n]) &amp;&amp; (\forall int !n:[i..len-1],ghost_old_iarray[!n] == iarray[!n]) &amp;&amp; (\forall int !n:[ret..i-2],ghost_old_iarray[!n+1] == iarray[!n])</pre>			

### 验证结果中的形状图显示

错误检查 验证结果 语句演算 引理证明

**(30,13) Return语句 Invalid**

`k == n && n >= 0 && n <= 100`

**Graph list**

**形状子图1 Invalid**

`$result == \null && n >= 0 && n <= 100`

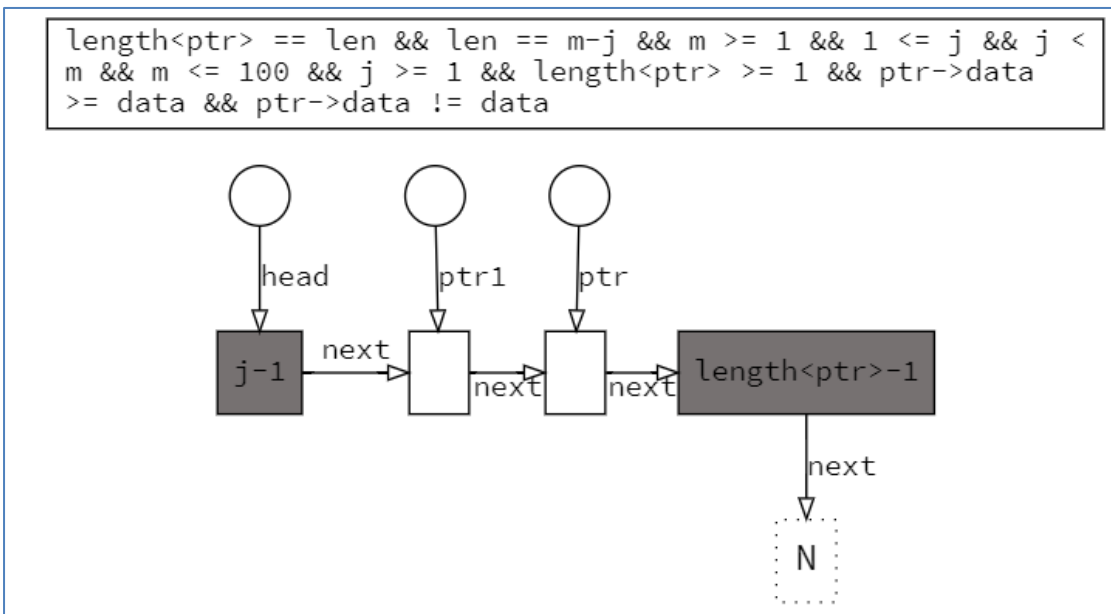
`n == 0`

**Invalid assert**

SVG [Pre-graph] SVG [Post-graph]

形状图中显示对应的断言状态，并以图的方式进行呈现。点击【SVG】按钮，查看形状图，形状图显示效果如下：

### 形状图 SVG



语句演算

错误检查
验证结果
语句演算
引理证明

**(33,26) If语句 演算结果断言** ◀前语句 后语句▶

所属协议: **bsearch(22,1)**

**分支1**

```
m == 100 && j == k-1 && i == k+1 && k == %2i+(%2j-
%2i)/2 && 0 <= %2i && %2i <= m && -1 <= %2j && %2j
<= m-1 && %2i <= %2j && val <= a[k] && val >= a[k]
&& %2j-%2i >= -1 && (\forall int !n:[0..%2i-1].val >
a[!n]) && (\forall int !n:[0..m-2].a[!n] < a[!n+1]) &&
(\forall int !n:[%2j+1..m-1].val < a[!n])
```

**分支2**

```
m == 100 && i == k+1 && k == %2i+(j-%2i)/2 && -1 <= j
&& 0 <= %2i && %2i <= m && j <= m-1 && %2i <= j
&& val > a[k] && j-%2i >= -1 && val >= a[k] && (\forall int
!n:[j+1..m-1].val < a[!n]) && (\forall int !n:[0..%2i-
1].val > a[!n]) && (\forall int !n:[0..m-2].a[!n] < a[!n+1])
```

**分支3**

```
m == 100 && j == k-1 && k == i+(%2j-i)/2 && 0 <= i &&
i <= m && i <= %2j && -1 <= %2j && %2j <= m-1 &&
val < a[k] && %2j-i >= -1 && val <= a[k] && (\forall int
!n:[0..i-1].val > a[!n]) && (\forall int !n:[0..m-2].a[!n] <
a[!n+1]) && (\forall int !n:[%2j+1..m-1].val < a[!n])
```

引理证明

错误检查
验证结果
语句演算
引理证明

**(6, 1) sorted\_array\_1 True**

```
\forall array !b.\forall int !value.\forall int !k:[0..m-1].
(\forall int !n:[0..m-2].!b[!n] < !b[!n+1]) && !value > !b[!k] ==>
(\forall int !n:[0..!k].!value > !b[!n])
```

**直接证明 Unknown**

**Assertion for proof:**

```
\forall array !b, int !value, int !k.(0 <= !k && !k <= m-1 &&
(\forall int !n.0 <= !n && !n <= m-2 ==> (!b[!n] < !b[!n+1]))
&& !value > !b[!k]) ==> (\forall int !n.0 <= !n && !n <= !k
==> (!value > !b[!n]))
```

**SMT2**

**自然数引理证明 [!value] Unknown**

**归纳变元:**

!value

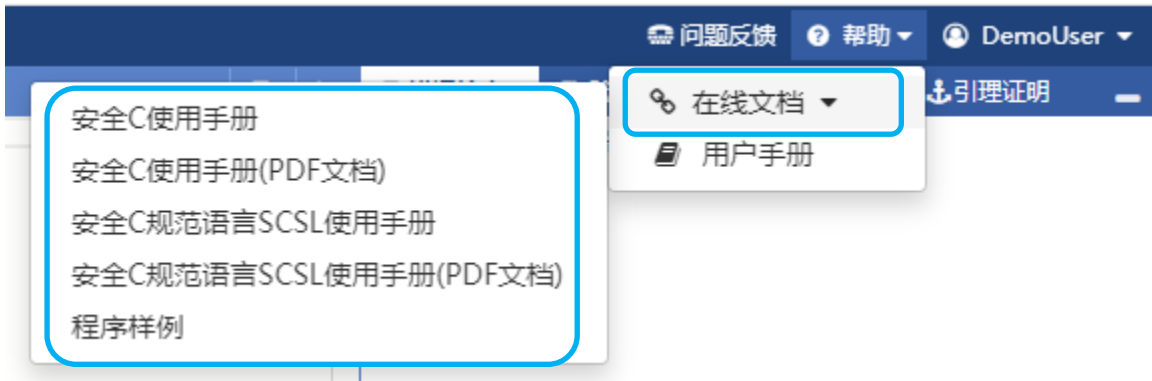
**约束区间:**

```
\forall array !b, int !k.(0 <= !k && !k <= m-1 && (\forall int
!n.0 <= !n && !n <= m-2 ==> (!b[!n] < !b[!n+1])) && !value
> !b[!k]) ==> (\forall int !n.0 <= !n && !n <= !k ==> (!value
> !b[!n]))
```

**归纳基始证明: Unknown**

## 2.21 帮助

主画面中点击左上角的【帮助】按钮，弹出帮助菜单。选择[在线文档]，弹出二级菜单，项目有安全 C 使用手册及手册的 PDF 版、安全 C 规范语言 SCSL 使用手册及手册的 PDF 版、验证程序样例五个选项。举例如下：



### 安全 C 语言使用手册

## 安全C语言使用手册

安徽中科国创高可信软件有限公司  
科创验证器® 2020年6月

---

### 前言

本手册供熟悉C语言，并准备用安全C语言的验证器（名字：科创验证器），进行程序验证的软件技术人员学习安全C语言时使用。当然，使用者还需要学习标注语言手册，才能逐步胜任程序验证工作。

第一章介绍怎样用尽量少的编程约束，把不安全的C99语言限制成安全C语言的设计思路。这主要是把程序验证中使用赋值公理时关注的是否存在变量别名，和安全语言设计中关注的类型是否可靠，这两件表面上面向不同需求但实质上紧密相关的事情综合起来，给出设计方案。本章主要通过一些简单的实例来展示变量别名和类型可靠两者的相关性。

第二章以第一章的设计方案为基础，对C语言的各种类型，介绍具体的编程约束。从中可以看到，为保证安全性，代码中缺少的一些信息，是由程序员在程序标注中进行补充而得到的。程序标注是程序员为程序代码写一些说明，作为给验证器的提示，以提高验证器判断程序安全性和正确性的能力。本章还简略介绍了各种主要的程序标注。

第三章介绍为易变数据结构设计的形状系统。各种易变数据结构的名称就是它们形状的名称。形状系统仿照类型系统，把形状分成基本形状和构造形状，构造形状分成嵌套形状和含附加指针的形状。本章围绕单态命名基本形状和形状的分类，对形状系统进行初步介绍，让大家对形状系统有大体的了解，并了解形状图在验证操作易变数据结构的代码上的优点。以方便学习标注语言和为代码写标注。

程序验证是以C的源文件为单位，各源文件分别验证，然后再集成验证。对于每个源文件，验证器对源文件的每个函数，自上而下地按照各种语句的推理规则进行演绎推理，并在重要的程序点产生验证条件。若一个函数的所有验证条件都得证，则该函数得证。

**重要备注：**对于第三章介绍为易变数据结构设计的形状系统，目前已实现3.1节介绍的单态命名基本形状。3.2节和3.3节的内容在相关形状的实现过程中可能还会修改。

安全C语言使用手册下载

安全 C 是 C 语言 C99 标准的一个子集，它对 C 语言中的一些使用做了限制，在不影响 C 语言表现能力的前提下，提高了 C 语言的安全性，同时有益于 C 语言程序的形式化验证。手册给出了安全 C 的使用说明。

## 安全 C 规范语言使用手册

### 安全C规范语言使用手册

安徽中科国创高可信软件有限公司 科创SC验证器®  
2020年6月

#### 前言

本手册是安全C语言的规范语言SCSL (*Safe C Specification Language*) 的参考手册，供熟悉ISO C编程语言，有些了解安全C语言，并准备用安全C语言的验证器 - 科创SC验证器，进行程序验证的软件技术人员学习如何描述程序功能时使用。

SCSL是安全C语言的行为接口规范语言 (*behavioral interface specification languages*)，可用于编写程序标注，程序标注加在程序注释中，用来描述安全C源代码的行为性质，因此SCSL又称为标注语言。该语言的设计参考了ACSL语言 (*ANSI/ISO C Specification Language*) [1]，也体现出了中国科学技术大学计算机学院相关实验室多年来的研究成果[2, 3, 5]。

源文件中很多地方都有标注，这些标注是对相关程序片段行为的描述，它们综合起来就构成程序行为的描述。从宏观上看，标注分成两大部分：全局标注和语句标注。


**全局标注** 主要有下面几种。

1. 函数协议  
描述函数的行为，其标注正好插在函数的声明或定义之前。
2. 类型不变式  
描述结构体和共用体等类型的不变性，其标注正好出现在被描述对象的定义或声明之后。
3. 变量不变式  
描述外部变量，静态外部变量和静态局部变量的不变性也是出现在对应的全局或局部声明的层次上。
4. 全局逻辑声明  
逻辑的常量、类型、变量、函数、谓词及引理等的定义或声明可标注在允许全局声明出现的地方有些也可以出现在允许局部声明出现的地方。

**语句标注** 有下面几种。

1. 程序点断言  
程序点断言标注在C标号可以出现的任何地方，或者正好在程序块的闭括号之前，描述该程序点状态必须满足的断言。
2. 循环标注

安全 C 的规范语言是一种用于对程序的行为性质（程序的功能）进行形式化描述的标注语言。它以注释的形式添加在程序体中，和源代码一起构成科创验证器可以接受的待验证的程序。源文件中很多地方都有标注，这些标注是对相关程序片段行为的描述，它们综合起来就构成程序行为的描述。

 注：在程序需要添加标注的地方（如函数协议、循环不变式等）未给出标注时，则默认标注为“`!true`”。

程序验证样例给出了算法级别的程序及标注的书写示例，并按照其特点进行了分类。样例程序页面给出了程序的算法概要、标注说明、程序代码和下载按钮。读者可以通过下载程序，或复制代码到验证器进行验证、学习。

安全C语言程序验证样例

一些常见算法的安全C语言程序通过了科创验证器的验证，这里所列出的是从中选出的一部分程序作为学习样例。程序中的标注是我们对程序行为性质的形式化描述，是学习程序验证的重点。所有样例程序仅供学习之用，可自由下载。用户可以通过 [科创验证器学习平台](#) 尝试程序验证的乐趣。

**By Topic**

- 简单程序 (Simple Programs) [12 examples]
- 一维数组 (One-Dimensional Array) [18 examples]
- 二维数组 (Two-Dimensional Array) [10 examples]
- 易变数据结构 (Mutable Data Structures) [30 examples]
- 字符串
- PL/O

**By Algorithm**

- 排序
- 查找
- 数值计

**By Keyword**

- 引理
- 归纳谓
- 幽灵谓
- 逻辑变
- 形状谓
- 范域谓

## 简单程序(Simple Programs)

Examples of verified algorithms

- 黑与白问题 (Black And White Problem)
- 求阶乘 (Find Factorial of a Number)
- 求最大公约数 (Find Greatest Common Divisor)
- 魔方问题 (Magic Square Problem)
- 数
- 乘
- 求
- 自
- W
- 杨
- 杨
- 杨

阶乘函数  
计算阶乘函数的验证  
本文件有2个函数，分别是计算阶乘的迭代和递归实现。 (验证时间约19sec)

[Back to Index]

验证特点: 迭代，递归，逻辑函数  
标注说明: 函数的返回值为 n 的阶乘。为预防计算结果溢出，函数前条件给出了 n 的上限。

程序样例

```
1 //本文件有2个函数，分别是计算阶乘的迭代和递归实现。
2
3 #define MAXN_LONG 20
4 ////////////////////////////////////////////////////////////////////全局逻辑定义和声明//////////////////////////////////////////////////////////////////
5 /**
6  | logic long factorial(int n) = //逻辑函数factorial计算n的阶乘
7  |   n == 0 ? 1 : n * factorial(n-1);
8  */
9
10 //1. 迭代计算阶乘函数的验证
11 //要点:
12 // 循环迭代计算 n 的阶乘
13 //关键字: 逻辑函数, 循环迭代
14 ////////////////////////////////////////////////////////////////////fac_loop函数的验证//////////////////////////////////////////////////////////////////
15 //函数协议:
16 //前条件: 给出 n 的上限; 后条件: 函数的返回值为 n 的阶乘。
17
18 /**
19  | requires
20  |   0 <= n && n <= MAXN_LONG
21  |   ;
22  | ensures
23  |   \result == factorial(n)
24  |   ;
25  */
26 long fac_loop(const int n) {
27   long fact; int i;
28   i = 0;
29   fact = 1;
30   //@ loop invariant fact == factorial(i) && 0 <= i <= n <= MAXN_LONG;
31   while (i < n) {
32     fact = fact * (i + 1);
33     i = i + 1;
34   }
35   return fact;
36 }
```

### 程序样例-详细界面

#### 删除数组元素

查找数组中第一个与给定值x相等的数组元素。若找到则删除该元素，并将数组中其后的所有元素向前移动一位，返回删除元素的数组下标。若找不到，则返回-1。（验证时间约45sec）

[\[Back to Index\]](#)

**验证特点：**幽灵数组，逻辑变量，assigns子句，量化断言

**标注说明：**函数前条件：用一个幽灵数组保存实参数组在函数入口处的值，并用逻辑变量记录实参数组的长度；  
函数后条件：分二种情况表述，即

- 1) 没有找到要删除的元素，返回值=-1，数组元素的值没有改变。
- 2) 找到，返回值=找到元素的数组下标；该元素被删除，其前的元素没有变化，其后的所有元素都前移了一位。

**程序样例**

```
1 //函数array_delete查找并删除数组iarray中第一个与给定值x相等的数组元素；
2 //删除后数组中将该元素之后的所有元素逐个向前移一位
3 // 要点
4 //      1 幽灵数组在验证中的辅助使用
5 //      2 量化断言对数组性质的描述
6
7 #define CAPACITY 1000
8 typedef int array[CAPACITY];
9
10 array iarray;
11 int len; // 数组中有效元素的个数（简称为数组长度）
12
13 ////////////////////////////////////////////////////////////////////幽灵变量、逻辑变量//////////////////////////////////////////////////////////////////
14 //@ ghost array ghost_old_iarray; //幽灵数组，记录函数入口处的数组iarray的内容
15 //@ logic int oldLen; //逻辑变量，记录数组的长度
16
17 ////////////////////////////////////////////////////////////////////array_delete函数的验证//////////////////////////////////////////////////////////////////
18 //函数协议
19 // 前条件
20 //     幽灵变量ghost_old_iarray 记录函数入口时的数组iarray的元素
21 //     逻辑变量oldLen 记录函数入口时的数组长度len
22 // 后条件
23 //     若数组iarray内不存在x，程序返回值为-1，数组未发生改变
24 //     若数组iarray内存在x，程序返回值为第一次发现x的位置，删除该位置的数组元素
25 //     数组内该位置前的数组元素未发生改变，该位置后的数组元素向前移动一位，数组长度减1
26
27 /*@
28     requires
29         len >= 0 && len <= CAPACITY && oldLen == len
```

样例详细界面中包含了当前选择程序样例的功能说明、验证特点、标注说明和样例代码。

点击程序样例右侧的下载图标，可以下载程序样例对应的 C 程序文件。

同样，通过点击右上角的【Back to Index】链接，可以快速回到大分类界面。



## 3 验证结果说明

在 2.21 节给出了验证结果在系统界面的分布与表示，本章将对不同类型的验证结果给出详细的说明，以及如何根据验证的 Log 输出发现程序或标注的错误。

### 3.1 错误检查

【错误检查】标签页（Tab）中的信息来源于两个方面的检查结果：1) 安全 C 语言及规范语言（程序标注）的约束检查。2) 待验证的 C 语言程序及其标注的词法、语法及语义检查。因此，在验证前，系统会对这些约束及错误进行检查。下面以阶乘函数为例，介绍错误检查中的错误输出和修改。

```

1  #define N 12
2  /*@ //逻辑函数 factorial 计算 m 的阶乘
3      logic int factorial(int m) =
4          m == 0 ? 1 : lfactorial(m-1) * m;
5  */
6  /*@ //函数协议
7      requires    n >= 0 & n <= N;
8      ensures     \result = factorial(n);
9  */
10 int factorial(const int n) {
11     int fact; int i;
12     i = 0;
13     fact = 1;
14     /*@ loop invariant fact == lfactorial(i) && 0 <= i <= n <= N && fact >= 1;
15     while (i < n){
16         i = i +1;
17         fact = fact * i;
18     }
19     return fact;
20 }
```

对上述阶乘函数进行验证，系统在画面右侧的【错误检查】Tab页中报告如下五个错误：

错误检查	验证结果	语句演算	引理证明
(4,22)	Error	use of undeclared lemma,predicate,inductive,logic-function	
(7,24)	Error	invalid operands to binary expression ('boolean' and 'boolean')	
(8,25)	Error	expected '=='	
(14,9)	Error	unkown key-words	
(3,15)	Error	redefinition of factorial	
(10,6)	Note	previous declaration is here	



第一列给出的是错误位置（行，列）。第二列是信息分类。第一条错误是说：第 4 行使用的逻辑函数“lfactorial”没有定义，需要将第 3 行的逻辑函数名称改为“lfactorial”；第二条错误是说：第 7 行断言中的逻辑与运算符错误（“&”应写成“&&”）；其他的四条错信息分别是：第 8 行断言中运算符使用错误（“=”应写成“==”）；第 14 行的循环不变式的关键字存在拼写错误（“invarant”应写成“invariant”）；第 3 行的逻辑函数名和 C 函数名重名（逻辑函数名 factorials 与第 10 行的函数名重名）。最后一条 Note 级别信息，提示重名发生的位置。经过修改，正确的阶乘函数定义如下：

```

1  #define N 12
2  /*@ //逻辑函数 lfactorial 计算 m 的阶乘
3      logic int lfactorial(int m) =
4          m == 0 ? 1 : lfactorial(m-1) * m;
5  */
6  /*@ //函数协议
7      requires    n >= 0 && n <= N;
8      ensures     \result == lfactorial(n);
9  */
10 int factorial(const int n) {
11     int fact; int i;
12     i = 0;
13     fact = 1;
14     //@ loop invariant fact == lfactorial(i) && 0 <= i <= n <= N && fact >= 1;
15     while (i < n){
16         i = i +1;
17         fact = fact * i;
18     }
19     return fact;
20 }
```

## 3.2 验证结果

### 3.2.1 函数前后条件

C 语言程序中函数待验证的性质使用函数前后条件来描述，通常称为函数协议。函数的前后条件使用一阶逻辑公式表示。函数前条件逻辑公式中的自由变量只能是函数形参和全局变量（如果全局变量和函数形参同名，该自由变量指函数形参）。函数后条件逻辑公式中的自由变量只能是函数形参常量和全局变量。如果函数有返回值，函数后条件中使用\result 表示该返回值。函数前条件主要描述形参和全局变量的基本信息（数组的大小，指针的偏移，数值型变量的取值范围等）和该函数调用入口处满足的条件。函数后条件主要描述函数输

入和输出之间的关系以及该函数调用出口处满足的性质（通常就是待验证的性质）。下面以上述的阶乘函数为例，介绍函数前后条件的描述。

```

    /*@ requires n >= 0 && n <= N;
       ensures  \result == factorial(n);
    */

```

该函数的功能是计算形参  $n$  的阶乘。函数的前条件（`requires` 子句）说明函数形参的取值范围在 0 到  $N (=12)$  之间。对函数形参取值范围进行限制的原因是：如果形参取值超过这个范围，函数返回值可能会发生数值溢出（具体参见后面小验证条件的描述）。函数的后条件（`ensures` 子句）说明函数返回值为  $n$  的阶乘。 $n$  的阶乘的计算通过逻辑函数给出。

### 3.2.2 循环不变式

循环是程序中的一个非常重要的性质，循环的性质通过循环不变式来描述。所有循环语句都需要添加循环不变式。循环不变式也使用一阶逻辑公式表示。下面以 `while` 循环语句为例说明循环不变式和验证的步骤。

```

    /*@ loop invariant
       P
    */
    while (condition) {
        statements
    }

```

循环不变式  $P$  在每次迭代的入口处成立。即每次循环计算条件表达式 `condition` 前，循环不变式  $P$  必须成立。因此，进入循环体时， $P \ \&\& \ \text{condition}$  一定成立。循环结束退出时， $P \ \&\& \ !\text{condition}$  一定成立。

阶乘函数的循环不变式描述如下：

```

fact == factorial(i) && 0 <= i <= n <= N && fact >= 1

```

每次循环迭代，在计算条件表达式  $i < n$  前，变量 `fact` 的值都等于变量  $i$  的阶乘，且  $0 <= i < n <= N$  和  $\text{fact} >= 1$  成立。当循环结束时， $i = n$ ， $P \ \&\& \ !\text{condition}$  也成立

```

i = n && fact == factorial(n) && 0 <= n <= N && fact >= 1

```

### 3.2.3 大验证条件

大验证条件是指在程序验证过程中在程序的循环入口、循环结束、函数返回、函数调用等关键点所生产的验证条件。这些关键点的验证对掌握整个程序的验证状况有着至关重要的意义。大验证条件的生成位置和验证结果一览显示在【验证结果一览】区。如，上述的求阶乘函数的验证结果一览如下图所示。第一列是函数名及在程序文件中的位置，第二列是程序的大验证条件和在程序文件中的位置，第三列是验证结果，Valid 表示验证通过，Invalid 表示未通过。

验证结果 ALL		行15, 列5
(10,1) factorial	(15,5) 循环入口	Valid
(10,1) factorial	(18,5) 循环体结束点	Valid
(10,1) factorial	(19,5) Return语句	Valid

点击验证结果则详细的验证条件显示在【验证结果】标签页中。如，点击“(15,5)循环入口”一行，则如下图所示，验证条件显示在【验证结果】标签页中。

错误检查 验证结果 语句演算 引理证明

**(15,5) 循环入口 Valid**

```
i == 0 && fact == 1 && n >= 0 && n <= 12
==>
fact == lfactorial(i) && 0 <= i && i <= n && n <= 12 && fact >= 1
```

循环入口处生成的大验证条件，检查的是函数前条件经过赋值语句序列

```
i = 0; fact = 1;
```

演算后得到的逻辑公式是否蕴涵循环不变式。

```
i == 0 && fact == 1 && n >= 0 && n <= 12
```

==>

```
fact == lfactorial(i) && 0 <= i && i <= n && n <= 12 && fact >= 1
```

该蕴涵式的前件中红色部分断言来自于函数的前条件，蓝色部分断言来自于上述赋值语句。

循环出口处生成一个大验证条件，如下所示：

```
fact == %2fact * i && %2fact == factorial(i-1) && n <= 12
&& i-1 < n && 0 <= i-1 && i-1 <= n && %2fact >= 1
```

==>

```
fact == factorial(i) && 0 <= i && i <= n && n <= 12 && fact >= 1
```

该验证条件检查的是由循环不变式  $P1$  合取条件表达式  $i < n$ ，经过赋值语句序列  $i = i + 1; fact = fact * i;$  演算后得到的逻辑公式（前件）是否蕴涵循环不变式（后件）。该蕴涵式的前件中**红色部分**断言来自于循环不变式  $P1$ ，受赋值语句序列  $i = i + 1; fact = fact * i;$  的影响，发生了一些变换。

在正向演绎推理中，赋值语句  $t = E$  的演算规则如下：

$$VC(t = E, P) = \exists t'. P[t'/t] \wedge t == E[t'/t]$$

依据这条规则，在演算过程中并不产生存在量化断言，而是用一个新的虚拟变量代替规则中的  $t'$ 。给  $t$  赋值引入的虚拟变量的命名规则是这样的：名字分成三段： $\% +$  正整数  $n + \text{trans}(t)$ 。其中  $\%$  专用于因赋值引入的虚拟变量，正整数  $n$  表示在演算过程中第几次碰到对  $t$  的赋值， $\text{trans}(t)$  为原变量名替换特殊符号后的变换。所以**红色部分**断言中的  $fact$  改写成了  $\%2fact$ 。

赋值语句  $i = i + 1$  较为简单， $i$  的旧值可以直接使用  $i-1$  表示。所以**红色部分**断言中的  $i$  改写成了  $i-1$ 。

断言中的**蓝色部分**断言来自于循环条件表达式，**黑色部分**断言来自于赋值语句  $fact = fact * i$ 。

函数出口处生成的大验证条件，如下所示：

```
$result == fact && fact == factorial(i)
&& 0 <= i && i <= n && i >= n && n <= 12 && fact >= 1
==>
$result == factorial(n)
```

该验证条件检查的是：经过 `return` 语句演算后得到的逻辑公式（前件）是否蕴涵函数的后断言（后件）。在这里，循环结束语句是 `return` 语句的前一条语句，该蕴涵式的前件中**红色部分**断言来自于循环不变式  $P1$ 。**蓝色部分**断言来自于 `return fact;` 语句，其中 `\result` 指称函数的返回值。

所以前件是循环语句结束的循环不变式合取该循环条件表达式的否定，同时合取 `return` 语句。

### 3.2.4 小验证条件

验证过程中，验证器也会检查数值计算是否会导致数值溢出，数组下标是否越界访问，是否有除 0 运算等等。这些检查的验证条件称作小验证条件。

以前面阶乘函数为例，删除函数前条件中和循环不变式中的  $n <= 12$  断言。验证器验证结束后报告如下图所示的小验证条件错误。

(10,1) factorial	(15,5) 循环入口	Valid
(10,1) factorial	(17,9) 小验证条件	Invalid
(10,1) factorial	(18,5) 循环体结束点	Valid
(10,1) factorial	(19,5) Return语句	Valid

点击验证失败的小验证条件，在【输出窗口】的【验证结果】Tab 页中显示如下信息：

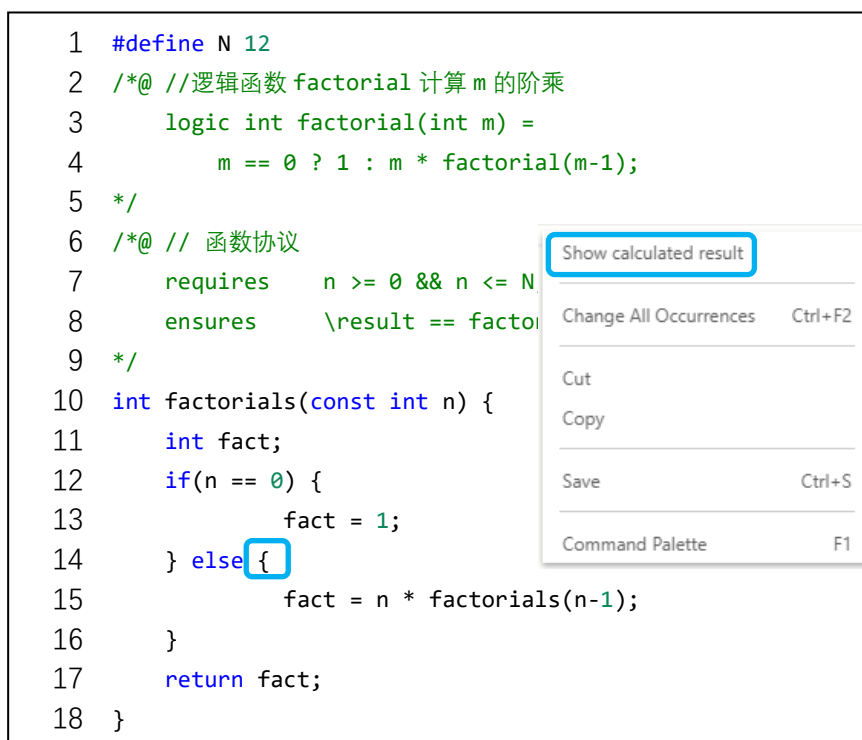


上图显示的信息说明断言  $fact * i \leq @INT\_MAX$  无法证明通过。即如果标注中不限制形参  $n$  的上界，表达式  $fact * i$  的值可能会上界溢出。用户通过点击【SMT2】按钮，可以查看验证该断言的 SMT2 文件。

### 3.2.5 函数调用

验证器对函数调用语句也会生成验证条件。如前所述，函数前条件描述函数开始执行前应满足的限制条件，函数后条件描述形参和返回值之间的关系。因此，验证器会在函数调用前检查这些限制条件是否满足，并将调用点不受被调用函数影响的断言加上被调用函数的后件作为函数调用后语句演算的断言。以阶乘函数的递归实现（如右图）为例，介绍验证器对函数调用的验证。

将鼠标光标移至第 14 行的大括号后面，点击鼠标右键，弹出菜单如右图所示。



在弹出的菜单中，点击菜单项【Show calculated result】，系统会在【语句演算】Tab 页中显示语句  $j = \text{factorials}(n-1)$ ；演算前的断言，即

$$n \geq 0 \ \&\& \ n \neq 0 \ \&\& \ n \leq 12$$

其中，**红色部分**的断言来自 `factorials` 函数前条件。**蓝色部分**的断言来自条件语句中条件表达式的非。

验证器在 `factorials(n-1)` 函数调用点产生如下验证条件。

$$\begin{aligned} & n \geq 0 \ \&\& \ n \neq 0 \ \&\& \ n \leq 12 \\ \Rightarrow & \\ & n - 1 \geq 0 \ \&\& \ n - 1 \leq 12 \end{aligned}$$

该验证条件检查 `factorials(n-1)` 函数调用前满足的条件是否蕴涵被调用函数 `factorials` 的前条件。上述蕴涵式的前件来自 `factorials(n-1)` 函数调用前的断言，后件来自于函数 `factorials` 的前条件中的形参用实参替换得到的结果。

点击【语句演算】Tab 页中【后语句】按钮，语句  $j = \text{factorials}(n-1)$ ；演算后的断言表示如下：

$$\$t15\_22 == \text{factorial}(n - 1) \ \&\& \ n \geq 0 \ \&\& \ n \neq 0 \ \&\& \ n \leq 12$$

其中，**红色部分**的断言来自函数 `factorials` 的后条件(形参 `n` 用实参 `n-1` 替换)。**蓝色部分**的断言来自函数调用 `factorials(n-1)` 点处没有被函数调用破坏的性质。

### 3.3 形状图演算结果

以程序样例中【删除二叉树根节点函数】作为示例，说明程序包含易变数据结构情况下的验证信息。以下简称样例。

#### 3.3.1 主要验证点对应的形状图

样例中包含三个主要的验证点：循环入口、循环体结束点和 Return 语句。

验证结果一览 <input type="button" value="ALL"/>		
(68,1) TREEdelete	(101,6) 循环入口	Valid
(68,1) TREEdelete	(104,6) 循环体结束点	Valid
(68,1) TREEdelete	(109,5) Return语句	Valid



点击验证结果一览中的任意记录，可以查看选中的验证点的验证详细信息。



示例中 Return 语句处存在 5 个分支的验证。

展开分支节点，可以查看每个分支对应的详细验证情况。


分支下的第一条显示当前分支中剔除易变数据结构相关信息后的断言。



※\true 代表剔除后没有其他断言。

Group list 中列举了当前分支在验证结束时拆解后的各个形状子图




每个子图中可以查看对应的验证条件，通过  划分验证添加的前条件和后条件



**Graph list**


形状子图1 Valid

```
BST(\null) && BST($result) && gt(succ, \null) &&
lt(pred, \null) && gt(succ, $result) &&
lt(pred, $result) && pred < $208oldp->d &&
succ > $208oldp->d && lt($208oldp->d, \null) &&
gt($208oldp->d, $result)
```



```
BST($result) && gt(succ, $result) && lt(pred, $result)
```

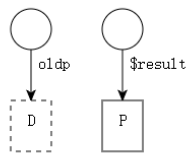
 [Pre-graph  Post-graph]

在末尾处也可以通过点击形状图查看按钮  跳转到形状图查看页面，查看前、后条件对应的形状图。

科创SC程序验证器

IMPLY-4-L\_1826\_0\_svg

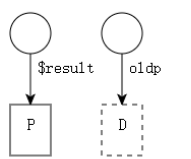
```
BST($result) && BST(\null) && gt($208oldp->d, $result) &&
lt($208oldp->d, \null) && gt(succ, $result) && gt(succ,
\ null) && succ > $208oldp->d && lt(pred, $result) &&
lt(pred, \null) && pred < $208oldp->d
```



科创SC程序验证器

IMPLY-4-R\_1827\_0\_svg

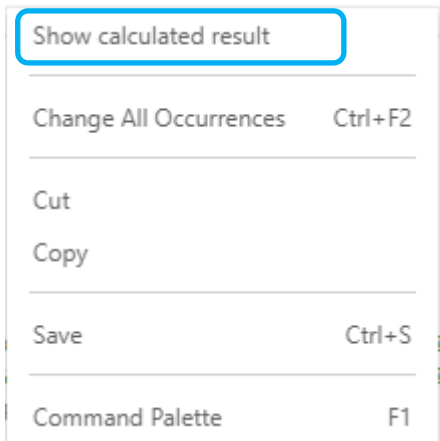
```
BST($result) && gt(succ, $result) && lt(pred, $result)
```





### 3.3.2 语句演算结束时的形状图

通过编辑区域的右键菜单中的[Show calculated result]选项，可以查看当前光标停留位置语句演算结束时的形状图。



以 106 行的 free 语句为例，点击[Show calculated result]选项，可以查看当前位置对应语句演算后的断言。通过前语句后语句按钮可以进行相邻语句的切换。

A screenshot of the IDE interface. At the top, there are tabs for '错误检查', '验证结果', '语句演算', and '引理证明'. Below the tabs, the current statement is '(106,6) 函数调用语句 之后的断言'. There are navigation buttons '前语句' and '后语句'. The main content area shows the '所属协议: TREEdelete(68,1)' and '分支1'. A large blue-bordered box contains the calculated result (invariant) for the statement: 

```
n >= 0
G : oldp->d == $1438s->d && n >= 0 && BST(q->r) &&
pred < q->d && BST(\null) && BST(oldp->r) &&
lt(pred, q->l) && lt(pred, q->r) && lt(q->d, q->r) &&
lt(pred, \null) && pred < $1438s->d &&
lt(q->d, \null) && q->d < $1438s->d &&
gt(succ, oldp->r) && BST_seg(oldp->l, q) &&
succ > $1275oldp->d && gt($1438s->d, q->r) &&
lt($1438s->d, \null) && gt($1275oldp->d, q->r) &&
gt_seg(succ, oldp->l, q) && lt_seg(pred, oldp->l, q) &&
gt($1275oldp->d, \null) &&
$1275oldp->d > $1438s->d &&
lt($1275oldp->d, oldp->r) &&
rightReaching(oldp->l, q, pred, $1275oldp->d)
```

 Below the code, there is a '形状图' button and an 'SVG' button. At the bottom, '分支2' is visible.

由于函数可以同时存在多个函数协议，为了便于区分，在语句之后的断言显示时，会对不同的函数协议进行分组显示。

**(106,6) 函数调用语句 之后的断言** ◀前语句 后语句▶

所属协议: TREEdelete(68,1)

分支1

考虑到存在不同分支的可能性，语句点之后断言也会存在多分支的情况。在详细断言中存在一个特殊标记 G(graph)，以区分哪些是易变数据结构相关断言。

```

n >= 0
G: oldp->d == $1438s->d && n >= 0 && BST(q->r) &&
pred < q->d && BST(\null) && BST(oldp->r) &&
lt(pred, q->l) && lt(pred, q->r) && lt(q->d, q->r) &&
lt(pred, \null) && pred < $1438s->d &&
lt(q->d, \null) && q->d < $1438s->d &&
    
```

在末尾处也可以通过点击形状图查看按钮 **SVG** 跳转到形状图查看页面，查看当前语句之后的形状图。

科创SC程序验证器

```
gph31_0_1451_0_svg
n >= 0 && n >= 0 && rightReaching(oldp->l, q, pred,
$1275oldp->d) && gt_seg(succ, oldp->l, q) && lt_seg(pred,
oldp->l, q) && BST_seg(oldp->l, q) && BST(oldp->r) &&
gt(succ, oldp->r) && lt($1275oldp->d, oldp->r) && succ >
$1275oldp->d && pred < q->d && lt(pred, q->l) && oldp->d
== $1438s->d && gt($1275oldp->d, q->r) && gt($1275oldp->d,
\null) && $1275oldp->d > $1438s->d && lt(pred, q->r) &&
lt(pred, \null) && pred < $1438s->d && lt(q->d, q->r) &&
lt(q->d, \null) && q->d < $1438s->d && BST(q->r) &&
BST(\null) && gt($1438s->d, q->r) && lt($1438s->d, \null
)
```

通过图可以很清晰的看到 free 语句后，s 指针指向悬空节点。

### 3.4 虚拟变量的命名规则

在查看验证结果的时候，我们会常常看到诸如 `%2fact`、`%1t<i>` 的变量名。它们是验证系统在验证过程中，如前端处理的程序变换、产生验证条件的演算，以及生成 SMT2 代码的翻译等，所引入的有助于验证的变量。因为不是程序员声明的，所以这些变量被统称为虚拟变量。虚拟变量命名的原则是确保不引起名字冲突、规则简单和可读性好，同时尽可能使虚拟变量的名字与它相关的变量名字有所关联，便于验证结果的分析。

#### 3.4.1 在程序变换中引入的虚拟变量

程序变换中引入虚拟变量的命名规则如下，由 5 部分组成：

“\$” + “t” + 正整数 1 + “\_” + 正整数 2

其中\$为先导符号；t 的用意是让名字中有字母，以符合变量名的习惯；正整数 1 和正整数 2 分别为程序变换语句变量引入点的行和列号，中间以\_号连接。例如，C 程序中的++操作可能引起副作用，所以赋值语句

```
8    y = x++;
```

在程序变换时将引入新的虚拟变量，变换成类似如下的代码：

```
int $t8_9;
$t8_9 = x;
x = $t8_9 + 1;
y = $t8_9;
```

`$t8_9` 表示变换源代码第 8 行第 9 列的表达式引入的虚拟变量。

#### 3.4.2 量化断言的约束变元变换引入的虚拟变量

所有量化断言的约束变量都需要改名，其命名规则为：

“!” + 约束变元名

即统一在原来名字的基础上增加前缀!，以保证不会与程序变量以及其他情况下引入的虚拟变量出现名字冲突。

例如，`(\forall int !i:[0..j-1].a[!i] == 0)` 中的约束变元名!i 就是在原来名字 i 前增加了前缀!。

在演算过程中也可能会产生新的约束变元，其命名规则为：

“!” + 序号 + “k”

其中“序号”仅是系统为区分不同变量而生成序列号，无实际意义。

### 3.4.3 范域词名称变换引入的虚拟变量

为了确保范域词变换为逻辑函数和谓词时，名称不与其他变量和谓词函数名重复，范域词的名字也做了变换，其命名规则为：

“!” + 序号 + 范域词名

其中 序号 仅代表序列号，无实际意义。如：

`\sum(0, n, \lambda integer k. t[k])` 被变换为：  
`logic integer !1sum(integer x)=(x<0)?0:(!1sum(x-1)+t[x])`。

### 3.4.4 赋值语句演算时引入的虚拟变量

在进行赋值语句的演算时需要引入虚拟变量记录赋值前被赋值变量的值。这时需要把左值表达式中的字符 ‘(’、’)’、‘[’、‘]’ 和 ‘\’ 分别变换成 ‘-’、‘\_’、‘<’、‘>’ 和 ‘/’。把这个变换用函数称为 trans（下同）。

给变量 `expr` 赋值时引入的虚拟变量的命名规则如下：

“%” + 序号 + `trans(expr)`

其中 序号 表示在演算过程中第几次对 `expr` 的赋值。

一般情况下，因赋值语句产生的虚拟变量是针对被赋值的对象，也就是赋值语句赋值号左边的表达式 `expr`。例如：对于断言 `p[a]==1` 经过第一次赋值语句 `p[a]=3`，引入的虚拟变量为 `%1p<a>`，产生的断言为 `%1P<a>==1 && p[a]==3`。

对于共用体的一个域赋值时，虚拟变量的引入是为了记录当前断言中存在的共用体的其它域的值。这时的 `expr` 为当前断言中的共用体的其他域。例如：有共用体 `union u{int a;int b;int c;}` 和该共用体的变量 `x`，对于断言 `x.a==3 || x.c==1`，经过第一次赋值语句 `x.b=5` 后，引入的虚拟变量为 `%1x.a` 和 `%1x.c`，产生的断言为 `(%1x.a==3 || %1x.c==1) && x.b==5`。

### 3.4.5 没有 const 修饰符的指针形参，其指向数据块命名时引入的虚拟变量

对于声明为“类型\* p”的指针 p，若 p 声明为可对它赋值，并且函数中确实存在对 p 赋值的操作，或者 p 的偏移不确定是 0，则 p 所指向数据块的命名规则为：

“^” + 指针变量名

即 ^p。否则所指向数据块的名字仍然用 p。例如，字符串比较函数

```
strcmp(const char* str1, const char* str2)
```

的前条件只是说明 str1 和 str2 分别指向两个物理字符串，且逻辑指针变量 p1、p2 分别等于 str1、str2，并且没有关于 str1、p1、str2、p2 偏移为 0 的断言。这种情况下，两个数据块的命名采用 ^ + p 的形式，如下所示。

```
^p1 == \pointer(^p1, \length(p1), 0)
```

```
^p2 == \pointer(^p2, \length(p2), 0)
```

### 3.4.6 证明循环语句终止性时引入的虚拟变量

证明循环语句终止性时，须引入虚拟变量来记住某些变量在循环入口的值，其命名规则为：

“@” + trans(expr) + “-v”

### 3.4.7 字符串常量、字符串谓词拆分时引入的虚拟变量

存放字符串常量的数据块拆分时，需引入虚拟变量记录拆分前的字符串。虚拟变量的命名规则是：

“@” + 序号 + “cstr”

其中 序号 从 1 开始累加，表示验证文件中字符串常量引入的虚拟变量的个数。

例如，下列代码段首先定义了一个存放字符串常量“abcd”的字符数组，接着将该字符数组的第 3 个元素修改成 'd'。

```
char a[]="abcd";
```

```
a[2] = 'd';
```

赋值语句 a[2] = 'd' 将导致存放字符串常量的数据块被拆分。为此，演算将引入一个虚拟变量@1cstr，将拆分前的字符串常量的数据块描述成

```
\string(@1cstr, 4) == "abcd" && @1cstr == \pointer(@1cstr, 4, 0)。
```

字符串内置函数 (`\prefix`、`\suffix`、`+`、`\index`) 的相关断言受赋值运算影响时，对应虚拟字符串变量的名字分别是：

```

“@” + trans(expr) + “-p” + 序号
“@” + trans(expr) + “-s” + 序号
“@” + trans(expr1) + “-” + trans(expr2) + “-cat” + 序号
“@” + trans(expr1) + “-” + trans(expr2) + “-idx” + 序号

```

其中 序号 的含义同上。

例如，断言

```
\prefix(\string(a, 10), 3) == "012"
```

在赋值语句 `a[0] = 'a'` 后，将变换成

```

(\forall int !i:[1..3-1].@^a-p1[!i] == @2cstr[!i]) &&
\string(@2cstr, 3) == "012"。

```

### 3.4.8 部分内建构造在演算中变换引入的虚拟变量

对于 “`\result`” 和 “`\exit_status`”，由于 smt2 中标识符中不能包含 “`\`”，所以需要对其进行如下变换：

```

\result      变换为 “$result”
\exit_status 变换为 “$exit_status”

```

### 3.4.9 递归谓词或者逻辑函数多次展开时引入的虚拟变量

为了避免数组相关的递归谓词和逻辑函数在数组元素赋值受影响时被多次展开，我们引入虚拟变量将原数组做整体代换，并构建虚拟数组和原数组的元素相等断言。其命名规则为：

```
“%%” + 序号 + trans(expr)
```

如以下示例：

```

/*@ inductive f(int* arr, int n) = n < 0
    || n >= 0 && arr[n] == 1 && f(arr, n-1);
*/
int arr[20];
/*@ requires f(arr, 10);
/*@ assigns arr[0];
void test() {

```

```

arr[0]=3;
}

```

则断言 `f(arr, 10)` 在 `arr[0]=3` 赋值语句后，产生的断言为：  
`arr[0] == 3 && %%1arr[0] == %1arr<0> && f(%%1arr, 10) &&`  
`(\forall integer !i:[1..20-1].%1arr[!i] == arr[!i])`

### 3.4.10 函数调用受 assigns 影响的表达式代换引入的虚拟变量

在函数协议中需要使用 `assigns` 描述在本函数中修改的全局变量，或者形参指针指向的元素。则在函数调用后，被修改的全局变量等需要进行变换，以保留其在函数调用前的性质。其变换引入的虚拟变量命名规则为：

“&” + 序号 + `trans(expr)`

例如，有如下函数：

```

//@ assigns a;
void f() {
    a = 1;
}

```

若函数调用前有断言 `a == b`，则经过上述函数调用后产生的断言为：`&1a == b && a == 1`。

### 3.4.11 形状图断言变换引入的虚拟变量

对于标注中出现的链表长度表达式 `\length(..., ...)`，无论它出现在等式断言还是不等式断言中，都需要为每次出现的 `\length(..., ...)` 表达式使用一个新的虚拟变量。虚拟变量的名字分成三部分，命名规则如下：

“`$length<`” + `trans(expr)` + “`>`”

例如，表示形状图长度信息断言中的 `$length<head>`。

`k >= 3 && n+1 >= k && n <= 100 && $length<head>+2 == k`

### 3.4.12 形状图构建引入的虚拟变量

在形状图构建表段内置谓词时，需要为浓缩节点的长度引入虚拟变量。虚拟变量的命名方式类似于程序变换引入虚拟变量的命名方式。

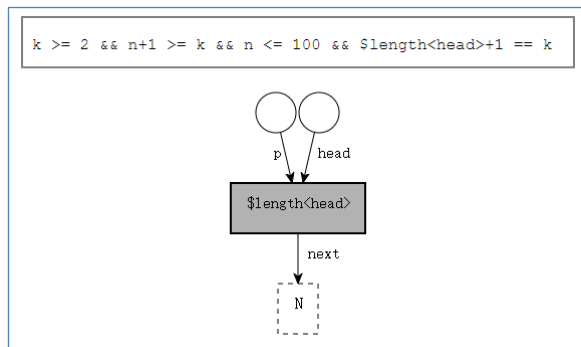
“`$length<`” + `trans(段头)` + “`-`” + `trans(段尾)` + “`>`”



在形状图构建表或树内置谓词时，需要为浓缩节点的长度引入虚拟变量。虚拟变量的命名方式类似于程序变换引入虚拟变量的命名方式，形式为

“ $\$length\langle$ ” +  $trans$ (表头或树根) + “ $\rangle$ ”

例如，右图中浓缩结点中用 $\$length\langle head\rangle$  表示其长度信息。



### 3.4.13 堆指针数据域赋值时引入的虚拟变量

堆指针数据域被赋值时，需引入虚拟变量记录被赋值前的变量信息，其命名规则为：

“ $\$$ ” + 序号 +  $trans(expr)$

其中 序号 为序列号，无实际意义， $expr$  为赋值号左边的表达式。

例如，断言  $p \rightarrow r \rightarrow data == 3$  在  $p \rightarrow r \rightarrow data = 5$  的赋值语句后，将其变换为： $\$1p \rightarrow r \rightarrow data == 3 \ \&\& \ p \rightarrow r \rightarrow data == 5$

特别的，对于赋值语句，若左侧为堆指针数据域，右侧为函数调用语句，为了临时记录函数调用语句的返回值，也需引入虚拟变量。该虚拟变量的命名规则为：

“ $\$$ ” + 序号 +  $\$result$

例如，有赋值语句  $p \rightarrow r \rightarrow data = f(\dots)$ ；且假设  $f(\dots)$  的返回值都大于 0，则该语句将变换为：

```
 $\$1\$result = f(\dots);$ 
 $p \rightarrow r \rightarrow data = \$1\$result;$ 
```

而断言  $p \rightarrow r \rightarrow data == 3$  在  $p \rightarrow r \rightarrow data = f(\dots)$  的赋值语句后，将被变换为： $\$1p \rightarrow r \rightarrow data == 3 \ \&\& \ p \rightarrow r \rightarrow data == \$1\$result \ \&\& \ \$1\$result > 0$



### 3.4.14 删除节点 (free 语句等) 时引入的虚拟变量

形状图节点被删除 (释放) 时, 需要引入虚拟变量记录该节点删除前的数据断言的信息。该虚拟变量的命名规则为:

“\$” + 序号 + trans(expr)

其中 序号 仅代表序列, 无实际意义。expr 为被删除节点表达式或指向同一节点的表达式。

例如, 有逻辑指针 oldp 和指针 p 指向同一个节点, 断言 `oldp->data == 10` 在释放语句 `free(p)` 后将被转换成 `$146oldp->data == 10` 的形式。

### 3.4.15 函数调用时堆指针形参引入的虚拟变量

堆指针相关的函数调用时会引入虚拟变量, 将调用点形状图中的堆指针实参和函数前条件的形状图中的堆指针形参用相同的虚拟变量表示。其虚拟变量的命名规则为:

“\_ \$” + 形参名

例如, 二叉排序树递归插入函数 `treeInsert1(Node* p, const int d)` 中对右子树插入递归调用点 `treeInsert1(p->r, d)` 的形状图表示如下:

```

pred+1 <= d && succ >= d+1 && oldp->d <= d && oldp->d < d
&& BST(oldp->l) && BST(_Sp) && gt(oldp->d, oldp->l) &&
lt(oldp->d, _Sp) && gt(succ, oldp->l) && gt(succ, _Sp) &&
succ > oldp->d && lt(pred, oldp->l) && lt(pred, _Sp) &&
pred < oldp->d

```



图中堆指针实参 `oldp->r` 已使用 `_Sp` 替换。

### 3.4.16 函数调用时堆指针数据域实参引入的虚拟变量

堆指针相关的函数调用时, 若实参中含有堆指针数据域, 则须用虚拟变量替换。其命名规则为:

“\$\$” + 序号 + trans(expr)

### 3.4.17 smt2 翻译时引入的虚拟变量

在将需证明的验证条件翻译成 smt2 格式时，也需要引入一些虚拟变量。

1) 需要将 `\offset(expr)` 和 `\length(expr)` 替换为虚拟变量，其命名规则分别为：

“/\$\_offset\_” + trans(expr)

“/\$\_length\_” + trans(expr)

2) 在翻译字符串常量时引入的虚拟变量的命名规则为：

“logicString\_” + 序号

其中 序号 仅代表序列，无实际意义。