1. 介绍

《s・C・ratch 高级开发框架》是一款由不同之者设计的，基于 C++和 scratch 编程语言实现的，用于提高 scratch 创作效率的开发框架。

该框架包含两个组件，一个是基于 scratch 实现的 FSC Plus，另一个是基于 C++ 实现的 C 语言编译器。

该框架的使用流程是：使用 C 语言编写程序→用该 C 语言编译器将程序编译成 FSC Plus 类汇编语言→在 scratch 中编写 C 语言中调用了的 scratch 函数→运行 FSC。

该框架使用的 C 语言并非传统的 C 语言，而是有着一套新的标准。请在使用前详细阅读标准。

本项目为开源项目，如需使用可自行操作。

本文档主要叙述该 C 语言的标准，FSC Plus 的标准见附件。

1.1 快速入门

完整介绍该项目需要较大篇幅，接下来将阐述一些概要以便快速入门。

1.1.1 从 C 语言到 s • C • ratch

快速入门 s • C • ratch 要求用户有 C 语言基础。只需注意几个与 C 语言有明显区别的地方。

C 语言有很多变量类型，但是 s • C • ratch 只有 **var** 类型，即万能变量。

比如，在 C 语言中，定义变量是这样的

```
int a;
```

但是在 s • C • ratch 中，你需要这样

```
var a;
```

在 C 语言中，一般使用 **printf** 来输出，而且需要引入头文件；而在 s • C • ratch 中，不能也不需要引入头文件，输出可以使用 **display** 函数，只有一个参数。输出到 scratch 的 **console** 列表中。

1.1.2 C 语言与 scratch 的交互

如果光是运行 C 语言，那这个项目便失去了意义。s • C • ratch 的核心功能是 C 语言与 scratch 的交互。

但自定义交互的门槛较高，而自正式版起，s • C • ratch 会不断添加各种库，这些库可以让你用 C 语言轻松与 scratch 交互。比如，你希望用 C 语言操控 scratch 角色移动到(100, 100)，则只需要先包含头文件

```
#include <graphics.h>
```

然后调用函数

```
move_to(100, 100);
```

更多的功能请参考 **7. 库**。

1.1.3 编写你的 C 语言代码

开发环境一定要选择 Microsoft Visual Studio，否则该工具的入门门槛将极高。

先运行“C 语言开发工具”文件夹下的“自动安装 Visual Studio 模板.bat”。

然后打开 Visual Studio，创建新项目，搜索模板 **sCr_Template** ，强烈建议使用 Pro 版，因为该版使用门槛更低，0 基础即可轻松使用。

创建完成后，展开“源文件”，打开 **source.c** ，这个文件就是你编写 C 语言代码所在的文件。

请不要更改任何模板中自带文件的文件名。

1.1.4 编译你的 C 语言代码

写好 C 语言代码后，若所用的是非 Pro 版框架，编译方法请查看测试版标准的快速入门部分。若所用的是 Pro 版框架，编译将变得无比轻松，你只需在菜单栏的“生成”中点击“重新生成解决方案”，程序便会自动处理各项工作并用记事本打开一个存储着 FSC 代码的 .txt 文档，该代码便是你的程序的编译结果。

注意：您需要关闭记事本才能再次编译代码。

1.1.5 运行你的代码

将编译结果复制，粘贴到 scratch 程序 **FSC Plus with Full Library** 中的列表 **original_code** 中，运行 scratch 程序即可。

1.1.5 Q&A

Q：提示“错误：列 0 你的代码有语法结构错误，但是做这个编译器的人太菜，不知道你哪错了。你自己把代码拿到别的编译器测测吧。”

A：由于多方面因素，该编译器不能准确报出所有错误，因此你需要利用其他编译器自行检查。（很大可能是你写了 **int** 而不是 **var**）

Q：写了 **main** 函数，但是仍然提示“错误：列 0 没有找到 main 函数。”

A：这个错误很多情况下都会报，很可能是你的代码结构错误，比如少了大括号、少了分号等等。

2. 编译过程

编译大体分为以下三个过程：词法分析，语法分析，语义分析及目标代码生成。

2.1 词法分析

通过词法分析得到 token 序列。每个 token 有两个属性：type 和 content。type 表示该 token 的类型，分为 KEYWORD、IDENTIFIER、CONSTANT、OPERATOR 和 SEPARATOR。content 表示 token 的内容，其值为代码本身。

2.2 语法分析

语法分析器基于 ebnf 进行枚举生成抽象语法树。

该 C 语言的 ebnf 见附件。

该语法分析非预测分析，因此无法报出语法结构错误具体位置。

2.3 语义分析及目标代码生成

该编译器不生成中间代码，在语义分析过程中直接生成目标代码。

目标代码为 FSC 代码，需使用 FSC Plus 运行。

该语义分析器不具有纠错功能，因此一次最多报出一处错误。

3. 系统

受 scratch 特性影响，该 C 语言与传统 C 语言的系统有很大差异。

3.1 数据

数据类型只有 var 一种，代表 scratch 变量的数据类型。但若作为指针，则既可以使 var* 类型的，也可以是 void* 类型的，但其本质仍然是 var。

3.2 内存

每个整数对应一个内存地址，每个地址存储一个 var 类型变量。

3.2.1 内存空间

受 FSC Plus 特性影响，存储空间大小固定为 200000，地址从 0 到 199999。

地址 [1, 40000) 为静态存储区，存放全局变量、全局数组、字符串常量。

地址 [40000, 99999) 为堆区，存放动态申请的数据。

地址 [100005, 199999）为栈区，存放函数参数、局部变量、函数返回值。

3.2.2 内存空间分配

该 C 语言编译器不使用传统 C 语言的对齐式分配。所有数据会紧凑地按地址从

小到大填满空间。

函数在栈上的空间分配按地址从小到大分别为参数，返回值，局部变量，控制链和机器状态。参数严格按照传递的从左到右的顺序在内存中由低到高紧密排列。

3.2.3 特殊空间分配

[0] 为常量 0，不可修改。

[99999, 100004]，[199999] 为系统关键区域，修改可能会引发程序崩溃。

3.3 运算

若该运算符在 scratch 中存在，则直接使用 scratch 的运算符模块运算。

部分运算符在 scratch 中不存在：

次幂运算：$a^x = e^{x \ln a}$

逻辑布尔运算：使用 > 运算符将 **var** 类型转换为 **bool** 类型。该转换并非传统 C 语言的转换，而是大于 0 的 **var** 类型变量为真，小于 0 的 **var** 类型变量为假。

二进制位运算：不支持

3.4 优化

3.4.1 常量优化

检测相邻指令之间，下句使用的寄存器是在上句被 **mov** 赋值过的已知量的情况，用数值代替寄存器。

3.4.2 重复优化

检测相邻指令之间，同一个寄存器在上句被修改，但是在下句又被 **mov** 赋值，且所赋值不涉及自身寄存器的情况，删除上句。

3.4.3 加减运算优化

检测相邻指令之间，同一个寄存器在上句被 **add** 或 **sub** 修改，在下句又被 **add** 或 **sub** 修改的情况，合并运算。

3.4.4 乘除运算优化

检测相邻指令之间，同一个寄存器在上句被 **mul** 或 **div** 修改，在下句又被 **mul** 或 **div** 修改的情况，合并运算。

3.4.5 拓扑优化

将每行代码视为一个节点，1 为原点，每一句到下一句建立有向边，**j** 类型语句额外到跳转目标点建立有向边，构成一个有向图，进行广度优先搜索判断节点可达性，将不可达的行删除。

3.4.6 连续跳转优化

对于一个 **j** 类型指令，如果其跳转目标是该指令的下一行，则删除该指令。

3.4.7 单条无效指令优化

对于一条指令，以下三种情况视为无效，并删除：

1. **mov** 指令的两个参数完全相同；
2. **add** 和 **sub** 指令的第二个参数为常数 0；
3. **mul** 和 **div** 指令的第二个参数为常数 1。

4. 语法规则

该 C 语言与传统 C 语言语法大体类似，但存在一些差别，主要是为了适应 scratch 环境而做出的调整。

4.1 词法

4.1.1 KEYWORD

KEYWORD 为关键字类型。

关键字包括：**var**、**struct**、**void**、**if**、**else**、**switch**、**case**、**default**、**break**、**for**、**while**、**do**、**continue**、**const**、**return**、**sizeof**、**__asm**。

4.1.2 IDENTIFIER

IDENTIFIER 为标识符类型。

标识符必须仅由数字、字母和下划线组成，并且必须以字母或下划线开头。

标识符不能是关键字。

4.1.3 CONSTANT

CONSTANT 为常量类型。

常量包括：数字、字符、字符串。

数字前加 0x、0b 分别表示十六进制、二进制。

在两个单引号之间只允许有一个字符或转义字符。

4.1.4 OPERATOR

OPERATOR 为运算符类型。

运算符包括：**->**、**++**、**--**、**<=**、**>=**、**==**、**!=**、**&&**、||、**+=**、**-=**、***=**、**/=**、**%=**、**(**、**)**、**[**、**]**、**.**、**!**、**+**、**-**、***、**/**、**%**、**<**、**>**、**^**、**?**、**:**、**=**、**,**。

4.1.5 SEPARATOR

SEPARATOR 为分界符。

分界符包括：**<空格>**、**{**、**}**、**;**。

注意：分界符不包括换行，代码中不应当出现换行。

补充说明：之所以您可以正常地在代码中使用换行符并通过编译，是因为 Pro 版的模板含有自动处理程序，会帮您将原始代码处理成编译器能识别的代码。

4.2 语法

具体语法规则请参考该项目的 ebnf 文件。

5. 语义规则

语义规则较为复杂，但大部分与 C 语言一致，故此处只介绍与 C 语言有差别的规则。

5.1 类型转换

任何类型的数据只要大小一致就可以相互转换，转换方式为内存拷贝。

5.2 函数

函数参数数量必须固定。

函数参数可以传递指针，但删除了传递数组的写法。

不支持函数指针。

5.3 结构体

不支持结构体初始化。

不支持在定义结构体的同时定义该结构体类型的变量。

不支持匿名结构体。

初始化列表与结构体的结构必须完全一致。

5.4 表达式与运算

5.4.1 赋值运算

不考虑数据类型，只考虑数据大小。

5.4.2 次幂运算

^ 运算符不再表示按位异或，而是表示次幂运算，其优先级高于乘法运算且低于前置单目运算符。

5.4.3 sizeof

sizeof 不能用于检测数组总占用空间。

5.5 流程控制

删除了 **switch** 语句。

5.6 内联汇编

s・C・ratch 支持在 C 语言中内联 FSC 代码。

格式为：__asm(<op>,<arg1>,<arg2>);

分别对应 FSC 代码的三个参数。

填入<op>的数据必须是字符串常量。

填入<arg1>和<arg2>的数据有两种选择，一种是字符串常量，一种是表达式。

若<arg2>不填，则自动填 0。

比如，你希望将变量 **a** 的值赋值给寄存器 **$ex**，则可以使用如下代码：

```
__asm("mov", "$ex", a);
```

6. 内部实现函数

该项目存在内部实现的函数。

6.1 display

```
void display(var arg);
```

在 scratch 的列表 **console** 中打印数字。

每调用一次该函数会自动在列表中换行一次。

6.2 scr_func

```
var scr_func(var id, var arg);
```

在 scratch 中调用编号为 **id** 的函数，并向其传递一个参数 **arg**。返回值由 scratch 代码决定。

6.3 exit

```
void exit(var val);
```

以返回值 **val** 退出程序。

6.4 数学函数

包含 **abs**，**floor**，**ceil**，**sqrt**，**sin**，**cos**，**tan**，**asin**，**acos**，**atan**，**ln**，**log10**，**exp**，**rand**。以下用 **xxx** 表示统称。

```
var xxx(var val);
```

返回对应数学函数的计算结果。

7. 库

该项目包含标准库，也鼓励用户提供第三方库。若想使用库，只需将“库”文件夹中的头文件复制到与 **source.c** 相同的目录下，然后拖入 Visual Studio 的“头文件”文件夹，然后在 **source.c** 中包含你希望使用的库的头文件。

7.1 io 库

功能概述：读取用户各种形式的信息输入，并以多种形式输出数据。

7.1.1 putchar

```
void putchar(const var ch);
```

打印一个字符 **ch** 到 **console** 的最后一行的末尾。

7.1.2 print_str

```
void print_str(const var *str);
```

打印一个字符串 **str** 到 **console** 的最后一行的末尾

7.1.3 print_num

```
void print_num(const var num);
```

打印一个数字 **num** 到 **console** 的最后一行的末尾

7.1.4 input_str

```
void input_str();
```

调用 scratch 的“询问并等待”模块，并将“回答”写入缓冲区。

7.1.5 clear_buffer

```
void clear_buffer();
```

清空缓冲区。

7.1.6 getchar

```
var getchar();
```

从缓冲区中读取一个字符并返回该字符的 ASCII 码。

有可能会读取到换行符。

7.1.7 getnum

```
var getnum();
```

从缓冲区中读取一个数字并返回该数字。

忽略数字前的空格和换行符，读取到空格或换行符为止。

若待读取的字符串不是数字，则不改变缓冲区，并返回 0。

7.1.8 scan_str

```
void scan_str(const var* str);
```

从缓冲区读取一个字符串并将该字符串写入 **str** 指向的字符串。

忽略字符串前的空格和换行符，读取到空格或换行符为止。

若读取成功，则返回 1。

若缓冲区为空，则返回 -1；

7.1.9 scan_str_getline

```
void scan_str_getline(const var* str);
```

从缓冲区读取一整行字符串并将该字符串写入 **str** 指向的字符串。

忽略字符串前的换行符，读取到换行符为止。

若读取成功，则返回 1。

若缓冲区为空，则返回 -1。

7.1.10 scan_num

```
void scan_num(const var *num_p);
```

从缓冲区读取一个数字并将该数字写入 **num_p** 指向的变量。

忽略数字前的空格和换行符，读取到空格或换行符为止。

若读取成功，则返回 1。

若待读取的字符串不是数字，则不改变缓冲区，并返回 0。

若缓冲区为空，则返回 -1。

7.1.11 get_key

```
var get_key();
```

获取当前正在按下的按键，返回按键对应号码。

若按下上、下、右、左方向键，则分别返回 1、2、3、4。

若按回车键，则返回 5。

若按下其他键，则返回该键的 ASCII 码。

若没有按下任何键或按下的键不受支持，则返回 0。

同时按下多个键时调用此函数是未定义行为。该行为不会引发异常，但是你并不知道该函数返回的是你按下的哪个键，因此若有按下多个键的需要，请不要使用此函数。

7.1.12 start_key_detection_service

```
void start_key_detection_service();
```

启用按键检测服务。

启用该服务后，服务会在后台运行，实时检测按下的键，并记录上一次按下的键。

7.1.13 stop_key_detection_service

```
void stop_key_detection_service();
```

关闭按键检测服务。

7.1.14 get_last_key

```
var get_last_key();
```

获取上一次按下的按键，返回按键对应号码。

按键对应号码见 **7.1.11 get_key**。

7.1.15 key_pressed

```
var key_pressed(const var key);
```

检测当前按键号码为 **key** 的按键是否按下，如果按下则返回 1，如果没有按下则返回 0。

按键对应号码见 **7.1.11 get_key**。

7.1.16 get_mouse_x

```
var get_mouse_x();
```

返回鼠标的 x 坐标。

7.1.17 get_mouse_y

```
var get_mouse_y();
```

返回鼠标的 y 坐标。

7.1.18 mouse_pressed

```
var mouse_pressed()
```

检测鼠标是否按下，如果按下则返回 1，如果没有按下则返回 0。

7.1.19 wheel_count

```
var wheel_count();
```

获取并返回鼠标滚轮累计滚动次数，并重新开始统计。

该函数将鼠标滚轮累计滚动次数保存在一个变量中，鼠标每往下滚一下，该变量就自增 1；鼠标每往上滚一下，该变量就自减 1。当调用该函数时，会将该变量

重置为 0。

7.2 graphics 库

功能概述：操作有关图形显示的功能。

7.2.1 struct Position

```
struct Position {
    var x, y;
};
```

坐标类型。

含有两个成员 **x** 和 **y**，分别表示 x 坐标和 y 坐标。

7.2.2 move_to

```
void move_to(const var x, const var y);
```

移动到坐标 **(x, y)**。

7.2.3 set_x

```
void set_x(const var x);
```

设置 x 坐标为 **x**。

7.2.4 set_y

```
void set_y(const var y);
```

设置 y 坐标为 **y**。

7.2.5 add_x

```
void add_x(const var x);
```

将 x 坐标增加 **x**。

7.2.6 add_y

```
void add_y(const var y);
```

将 y 坐标增加 **y**。

7.2.7 set_direction

```
void set_direction(const var dir);
```

将方向设置为 **dir**。

7.2.8 add_direction

```
void add_direction(const var dir);
```

将方向增加 **dir**。

7.2.9 get_x

```
var get_x();
```

获取并返回当前 x 坐标。

7.2.10 get_y

```
var get_y();
```

获取并返回当前 y 坐标。

7.2.11 get_direction

```
var get_direction();
```

获取并返回当前方向。

7.2.12 change_to_costume

```
void change_to_costume(const var* name);
```

将造型切换到 **name** 指向的字符串的造型。

7.2.13 change_to_costume_id

```
void change_to_costume_id(const var id);
```

将造型切换到 **id** 号。

7.2.14 set_size

```
void set_size(const var size);
```

将角色的大小设为 **size**。

7.2.15 add_size

```
void add_size(const var size);
```

将角色的大小增加 **size**。

7.2.16 set_color_effect

```
void set_color_effect(const var val);
```

将颜色特效设为 **val**。

7.2.17 add_color_effect

```
void add_color_effect(const var val);
```

将颜色特效增加 **val**。

7.2.18 set_brightness_effect

```
void set_brightness_effect(const var val);
```

将亮度特效设为 **val**。

7.2.19 add_brightness_effect

```
void add_brightness_effect(const var val);
```

将亮度特效增加 **val**。

7.2.20 set_ghost_effect

```
void set_ghost_effect(const var val);
```

将虚像特效设为 **val**。

7.2.21 add_ghost_effect

```
void add_ghost_effect(const var val);
```

将虚像特效增加 **val**。

7.2.22 show

```
void show();
```

显示角色。

7.2.23 hide

```
void hide();
```

隐藏角色。

7.2.24 get_costume_id

```
var get_costume_id();
```

获取并返回当前造型的编号。

7.2.25 get_size

```
var get_size();
```

获取并返回当前角色的大小。

7.2.26 get_color_effect

```
var get_color_effect();
```

获取并返回当前的颜色特效值。

7.2.27 get_brightness_effect

```
var get_brightness_effect();
```

获取并返回当前的亮度特效值。

7.2.28 get_ghost_effect

```
var get_ghost_effect();
```

获取并返回当前的虚像特效值。

7.2.29 erase_all

```
void erase_all();
```

清除所有画笔痕迹。

7.2.30 stamp

```
void stamp();
```

图章。

7.2.31 pen_down

```
void pen_down();
```

落笔。

7.2.32 pen_up

```
void pen_up();
```

抬笔。

7.2.33 _set_color

```
void _set_color(const var col);
```

将画笔的颜色设置为 **col**。

col 是一个 6 位 16 进制数，表示一种 RGB 颜色。

该函数不适用于直接调用。

7.2.34 rgb_color

```
var rgb_color(const var r, const var g, const var b);
```

根据 **r**，**g**，**b** 分别的值计算出并返回对应的 6 位 16 进制数。

7.2.35 set_color

```
void set_color(const var r, const var g, const var b);
```

将画笔颜色的 r，g，b 的值分别设置为 **r**，**g**，**b**。

7.2.36 set_transparency

```
void set_transparency(const var transparency);
```

将画笔的透明度设置为 **transparency**。

7.2.37 set_pen_size

```
void set_pen_size(const var size);
```

将画笔的粗细设置为 **size**。

7.3 refresh 库

功能概述：控制与刷新屏幕相关的功能。

7.3.1 enable_timing_refresh

```
void enable_timing_refresh();
```

启用 Timing Refresh 技术。

当需要同时进行前台动画和后台计算时适合启用。

7.3.2 disable_timing_refresh

```
void disable_timing_refresh();
```

禁用 Timing Refresh 技术。

当需要完成多项渲染后一次性显示出来，而渲染未完成时不显示出来时适合禁用。

7.3.3 flush_screen

```
void flush_screen();
```

刷新屏幕。

若 TR 启用，则该函数效果忽略不计；若 TR 禁用，则该函数给予了用户对屏幕刷新更强的操控性。

7.3.4 set_refresh_time

```
void set_refresh_time(const var time);
```

将屏幕刷新时间设置为 **time**。

若开启 TR，则屏幕每隔 **time** 秒会自动刷新一次。

7.4 memory 库

功能概述：进行底层快速的的内存操作。

7.4.1 memcpy

```
void* memcpy(void* dest, const void* src, var n);
```

将以 **src** 指向的内存为起点的大小为 **n** 的数据复制到以 **dest** 指向的内存为起点的大小为 **n** 的内存空间中，并返回 **src**。

源数据和目标区域不能有重叠，否则是未定义行为。

7.4.2 memmove

```
void* memmove(void* dest, const void* src, var n);
```

将以 **src** 指向的内存为起点的大小为 **n** 的数据复制到以 **dest** 指向的内存为起点的大小为 **n** 的内存空间中，并返回 **src**。

7.4.3 memset

```
void* memset(void* ptr, var value, var n);
```

将以 **ptr** 为起点的大小为 **n** 的内存中的数据全部初始化为 **value**，并返回 **ptr**。

7.4.4 memcmp

```
var memcmp(const void* ptr1, const void* ptr2, var n);
```

将分别以 **ptr1** 和 **ptr2** 为起点的大小为 **n** 的数据进行逐位比较。如果两者完全相同，则返回 0。否则：找出两者首位不相同的数据，若 **ptr1** 的值更大则返回 1，否则返回 -1。

7.4.5 malloc

```
void* malloc(var size);
```

申请一块大小为 **size** 的动态内存，并返回申请到的内存的首位地址。

若申请失败，则返回 0。

7.4.6 calloc

```
void* calloc(var num, var size);
```

申请一块大小为 <**num** 个大小为 **size** 的元素的总大小> 的内存，并将申请到的内存全部初始化为 0，并返回申请到的内存的首位地址。

若申请失败，则返回 0。

7.4.7 free

```
void free(void* ptr);
```

释放申请的位于 **ptr** 的内存。

若 **ptr** 不是之前申请到的值，则为未定义行为。

7.4.8 realloc

```
void* realloc(void* ptr, var new_size);
```

调整申请的位于 **ptr** 的内存的大小为 **new_size**。内存的地址可能会改变。并返回新的地址。