Про́клятый огонь, или магия препроцессора C

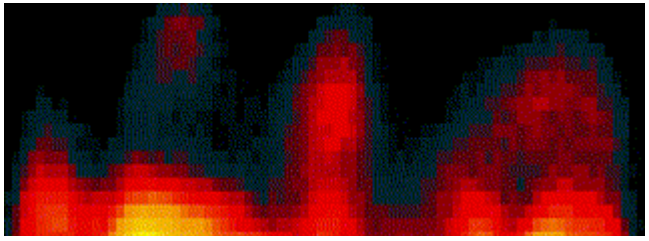
haqreu

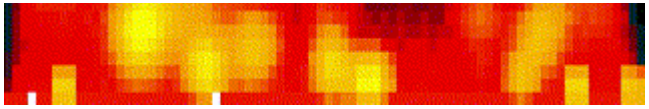
Задавались ли вы когда-нибудь вопросом, можно ли полноценно программировать при помощи директивы `#define` в языке C? Полнота по Тьюрингу шаблонов C++ известна весьма широко, например, люди пишут трассировщики лучей, делающие все вычисления во время компиляции (вместо времени исполнения). А как обстоят дела с препроцессором C? Вопрос оказался сильно нетривиальнее, и эта история является, на мой вкус, отличным анекдотом для курса лекций по теории компиляторов, что я готовлю в данный момент. В частности, для лучшего понимания происходящего здесь, рекомендую ознакомиться со второй статьёй, которую я опубликовал параллельно этой: лексер и парсер.

Чтобы не было обманутых впечатлений, предупрежу сразу, что рейтрейсера не будет, но про́клятый код будет очень даже! Итак, поехали. Для начала, почему я вообще задался этим вопросом? Если обычный код компьютерной графики вам скучен, следующий раздел можно пропустить, перематывайте до последней картинки.

Самый обычный огонь, никем не про́клятый

Как я и сказал, я пообещал написать простейший, но вполне полноценный компилятор только что придуманного мной языка wend за выходные. Написать-то дело несложное, а вот *описать* труднее. Для хорошего описания нужны красочные примеры. У меня аллергия на иллюстрации из разряда вычислений чисел Фибоначчи. Ну сколько же можно?! Поскольку wend крайне примитивен, то и примеры мне нужны простые, но всё же как можно более эффектные. И тут я вспомнил про древнюю демосцену! Вот, например, я хочу написать на своём языке программу, которая просто гоняет по кругу пламя:





Это сделать несложно: у меня нет возможности запускать графический режим, но ведь моя консоль поддерживает управляющую последовательность `\033[`, так что для отрисовки пламени мне вполне хватит одной инструкции `print`! Кстати, я слышал, что нынче даже в винде консоль поддерживает эскейп-последовательности ANSI, но лично не проверял.

Дело за малым, написать код. Поскольку я болен только на часть головы, а не на всю, писать я его буду сначала на C, и уж потом только транслировать (руками) в wend, поскольку мой компилятор - это хорошо, но всё же вокруг си инструментов самую малость больше. Да и баги в моём компиляторе никто не отменял, и мне лень думать, где именно у меня проблема. На баги gcc я уже, конечно, натыкался, но это исчезающе редкое явление.

Давайте посмотрим, как создать такой огонь, а потом продолжим разговор про препроцессор и чёрную магию. Вот так выглядит обёртка, от которой мы будем отталкиваться:

▸ Hidden text

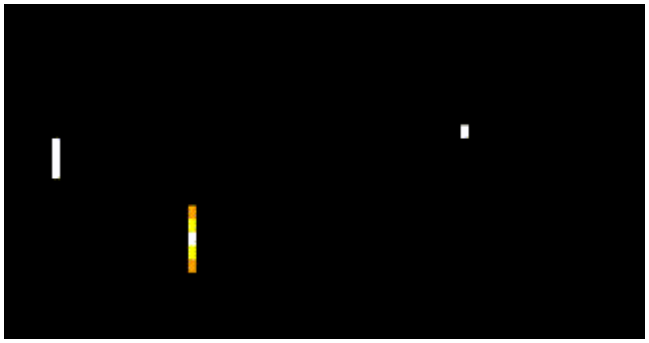
Для начала я определяю размеры моей консоли, в которой буду рисовать (80x25), Потом я определяю массив-палитру в 256 значений и буфер `fire`, который собственно и хранит картинку с (будущим) пламенем. Затем в вечном цикле `for(;;)` я отрисовываю этот буфер, в котором для начала просто заполняю белым цветом случайно выбранные пиксели. Получаем вполне ожидаемый результат:

Белые пиксели у меня будут искрами пламени. Искры довольно быстро остывают, нагревая при этом окружающую среду. Это можно промоделировать очень просто: на каждом кадре размывая картинку с предыдущего кадра. Все изменения в коде у меня будут происходить внутри блока отмеченного как `// rendering body`, так что больше код приводить целиком не буду, финальный код

можно найти в репозитории моего компилятора. Самое простое размытие картинки можно сделать, просто посчитав для каждого пикселя среднее значение среди всех его соседей. Практически все имплементации, что я встречаю, требуют создания копии всего буфера, например, посмотрите в википедии. При этом подобные фильтры сепарабельны по координатам, так что он полностью эквивалентен двум motion blur фильтрам, одному горизонтальному, второму вертикальному. Для начала давайте сделаем только вертикальное размытие:

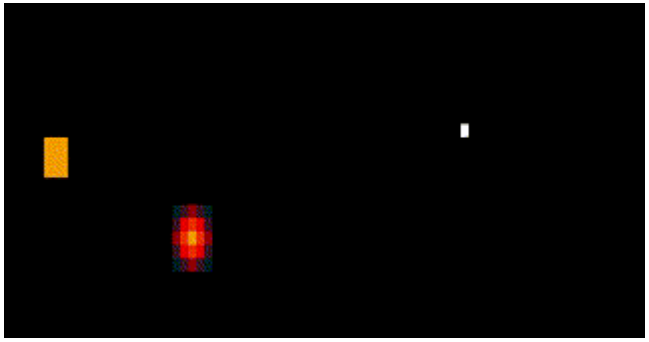
▸ Hidden text

Кольцевого буфера на три элемента мне хватает, больше никаких копий экранного буфера мне не надо. Этот код даёт следующий результат (я чуть-чуть замедлил видео, чтобы было нагляднее):



Добавим к нему горизонтальное:

▸ Hidden text



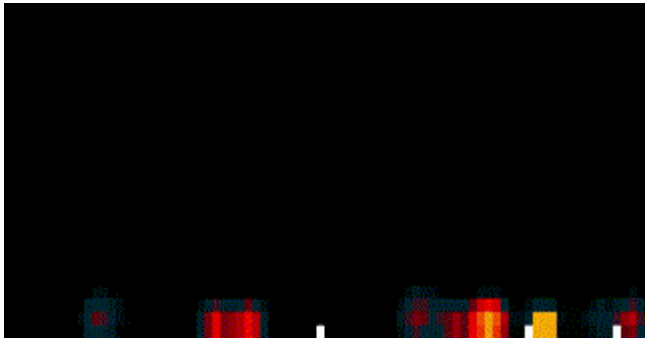
Тепло от одного пикселя быстро распределяется по всё большей и большей территории, так что перестаёт даже быть видным в моей палитре: на первой итерации один белый пиксель окружён восемью чёрными, на второй все девять пикселей имеют значение 255/9 =

28 и так далее:

```
итерация 1      итерация 2       итерация 3
                0  0  0  0 0     3  6  9  6 3
 0  0  0       0 28 28 28 0     6 12 18 12 6
 0 255 0       0 28 28 28 0     9 18 28 18 9
 0  0  0       0 28 28 28 0     6 12 18 12 6
                0  0  0  0 0     3  6  9  6 3
```

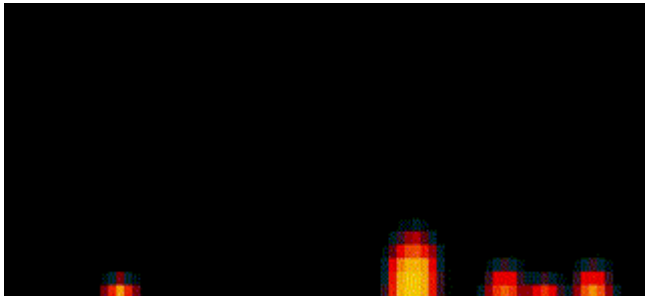
Тут я искры разбросал по всему экрану, но в реальности у нас тепло идёт непосредственно из костра, так что давайте чуточку подправим код, разрешив генерировать горячие пиксели только на самой нижней строчке экрана:

▸ Hidden text



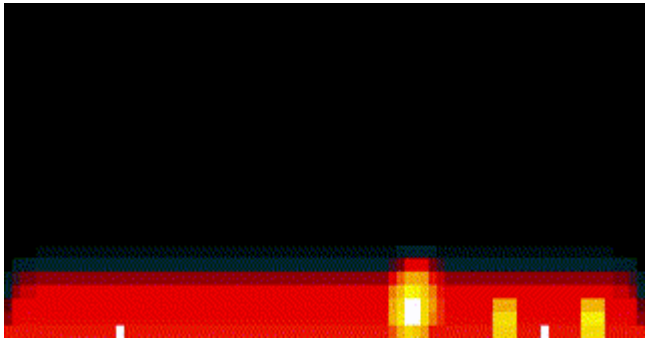
Изображение стало менее интересным, зато небо перестало греться без причины. Чего нам тут не хватает, так это конвекции! Давайте на каждом этапе просто сдвигать предыдущий кадр на одну строчку вверх:

▸ Hidden text



Уже сильно более похоже на правду! Но ведь у костра есть подушка углей, искры не возникают просто так, так что давайте закрасим постоянным цветом (а значит, и добавим тепла) нижнюю строчку картинки:

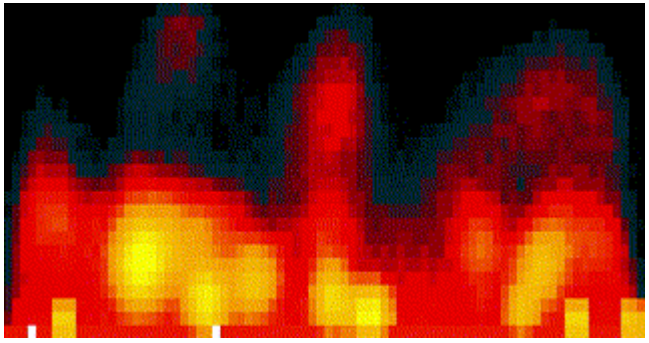
▸ Hidden text



Почти совсем хорошо, но тепла что-то очень много, давайте в качестве последнего штриха добавим охлаждение:

▸ Hidden text

Ну, собственно, и всё, обычный, никем не проклятый огонь, готов. Давайте посмотрим на него ещё раз:



Наверное, можно переводить код на wend? И тут как раз в кустах случайно стоит рояль, я могу сыграть!

Рояль в кустах

Внимательный читатель мог заметить, что в этой демке я всю отрисовку провожу в массиве `fire[]`, а в моём языке wend нет массивов!

И отрисовывать пиксели независимо один от другого не получится никак, поскольку рассеивание тепла (да и конвекция) требуют знания состояния соседних пикселей. Но это не беда, ведь у меня есть функции. Давайте предположим, что мне нужен массив в восемь элементов. Его можно сэмулировать восемью разными переменными, и двумя функциями - геттером/сеттером:

```
uint8_t fire0, fire1, fire2, fire3, fire4, fire5, fire6, fire7;
uint8_t get_fire(int i) {
    if (i==0) return fire0;
    if (i==1) return fire1;
    if (i==2) return fire2;
    if (i==3) return fire3;
    if (i==4) return fire4;
    if (i==5) return fire5;
    if (i==6) return fire6;
    if (i==7) return fire7;
}
void set_fire(int i, uint8_t v) {
  [...]
}
```

Сеттер описывать не буду, его структура будет такой же, как у геттера. Код тривиальный, но по факту я сделал связный список вместо массива, и чтобы добраться до двухтысячного элемента, мне придётся сделать две тысячи сравнений. Оно, конечно, абсолютно пофиг на таком размере данных, но что-то неспокойна моя душа. Впрочем, ничто не мешает искать нужную переменную при помощи дихотомии, сведя сложность от линейной к логарифмической:

```
uint8_t get_fire(int i) {
    if (i<4) {
        if (i<2) {
            if (i<1) {
```

```
                return fire0;
            } else {
                return fire1;
            }
        } else {
            if (i<3) {
                return fire2;
            } else {
                return fire3;
            }
        }
    } else {
        if (i<6) {
            if (i<5) {
                return fire4;
            } else {
                return fire5;
            }
        } else {
            if (i<7) {
                return fire6;
            } else {
                return fire7;
            }
        }
    }
}
```

Это уже сильно приятнее. И как раз вот этот код и дал толчок к данной статье. Как мне сгенерировать эту функцию? Руками писать сильно неохота :)

Мой язык местами сильно похож на си, так что если сделать сишный код, то его можно протестировать сначала нормально, а потом скопировать в исходник на wend.

Консоль у меня 80x25, так что мне нужна память на две тысячи ячеек. 2048 очень близко к 2000, и при этом является точной степенью двойки, так что с минимальным оверхедом мне не надо себе ломать голову над граничными условиями, и можно сделать полностью сбалансированное двоичное дерево поиска. Очевидно, что я мог взять любой язык программирования, и сгенерировать соответствующую строку текста, однако мне почему-то захотелось сделать в оригинальном исходнике с огнём простой переключатель на дефайне, который бы переключал между обычным массивом и эрзац-массивом: это был бы простой способ убедиться в том, что я нигде не напортачил. И вот тут я задался вопросом: а нельзя ли эту самую функцию сгенерировать именно при помощи препроцессора C?

Для этого мне пришлось бы написать рекурсивный `#define`. Можно ли это сделать, и если да, то как? Разумеется, я пошёл задавать вопрос гуглу. И среди прочего наткнулся на [любопытный тред](#) на cplusplus.com, давайте я его даже заскриню:

▸ Hidden text

Наш коллега задался ровно тем же самым вопросом, что и я, и ему трижды ответили, что рекурсия на дефайнах невозможна. Ха.

Имейте в виду, что это не я такой умный, я нашёл [правильную ссылку](#) на stackoverflow, и попытался лишь скомпилировать полученное знание.

Как работает препроцессор или почему макрокоманда - это не функция

Давайте разбираться с тем, как работает препроцессор C. В вышеприведённом коде с огнём мы уже встречались с дефайном `#define WIDTH 80`, это вполне стандартная практика определения констант (примечание: не делайте так в C++, `constexpr` - это хорошо! У дефайнов есть много неприятных моментов, которые начисто убираются при помощи `constexpr`). И когда лексер встречает лексему `WIDTH`, он её подменяет на 80 ещё **до** запуска непосредственно компилятора. Макрокоманды бывают ещё и похожими на функции, например, знаменитый `#define MIN(X, Y) (((X) < (Y)) ? (X) : (Y))`. Примечание: не делайте так, на третьем десятке

двадцать первого века я не могу придумать ни одной причины, чтобы продолжать использовать код из семидесятых.

"Выполнение", а точнее, разворачивание макрокоманд является чисто текстовым. Препроцессор не понимает язык Си, поэтому, если вы ему дадите `MIN(habracadabra, circ][(beg+++))`, то он её радостно преобразует в `(((habracadabra) < (circ][(beg+++))) ? (habracadabra) : (circ][(beg+++))`! Проверьте сами при помощи gcc -E source.c.

Когда мы хотим развернуть макрокоманду, похожую на функцию, то видя похожий синтаксис, большинство программистов считают, что и работает это как функция, то есть, сначала мы оценим значения параметров, и передадим в тело родительской макрокоманды. И чаще всего такая интуиция не противоречит тому, что мы видим. Но препроцессор - это не сам С, и макросы ведут себя совсем не так, и вышепривённый пример с `MIN` тому подтверждение.

Давайте я приведу правила подстановки для макрокоманд (в порядке их выполнения):

Приведение к строке (оператор #, в этой статье не встречается)

Подстановка текста аргументов вместо имён параметров (без разворачивания лексем)

Склеивание лексем (оператор ##, в этой статье встречается повсеместно)

Разворачивание лексем параметров

Повторное сканирование и разворачивание результата.

Настали тёмные времена, или чёрная магия

Давайте рассмотрим простейший пример (пока на си) хвостовой рекурсии:

```
void recursion(int d) {
    printf("%d ", d);
    if (d!=0) recursion(d - 1);
}
```

Если мы вызовем `recursion(3)`, то на экране будет выведено `3 2 1 0`. Нам нужно научиться сделать подобное исключительно на макросах. Итак, перечислим необходимые нам ингредиенты в порядке возрастающей сложности:

нужно уметь делать декремент

нужно уметь делать ветвление

нужно уметь проверить числовое значение на равенство нулю

ну и научиться делать непосредственно рекурсивный вызов.

Декремент

Давайте начнём с самого первого, с декремента. Препроцессор довольно тупой. Он выполняет подстановку текста, и ничего больше. Временами это раздражает, но это также позволяет вам манипулировать частями выражений, если вы того хотите, так что в этом есть определенный смысл.

Тот факт, что препроцессор ничего не знает про арифметику, делая исключительно текстовые подстановки, несколько затрудняет жизнь. Давайте сделаем первую попытку:

```
ssloy@home:~$ gcc -P -E - <<<'
#define DEC(n) n - 1
DEC(3)
'
3 - 1
```

Для наглядности я буду запускать gcc сразу на коде из командной строки, так что вы видите и сам код, и результат работы препроцессора. Итак, макрокоманда `DEC(3)`, разворачивается не в желаемую константу `2`, а в выражение `3-1`.

Не беда, давайте попробуем чуть схитрить, используя возможности склеивания лексем:

```
ssloy@home:~$ gcc -P -E - <<<'
#define DEC(n) DEC_##n
#define DEC_0 0
#define DEC_1 0
```

```
#define DEC_2 1
#define DEC_3 2
DEC(3)
DEC(DEC(3))
'

2
DEC_DEC(3)
```

Когда мы разворачиваем `DEC(3)`, то в происходит склеивание лексем `DEC_` и `3`, и создаётся новая лексема `DEC_3`, которая развёртывается в `2`, отлично! А вот с `DEC(DEC(3))` фокус не прошёл, почему? Не зря я правила разворачивания макрокоманд приводил, склеивание происходит на этап раньше развёртывания параметров, и поэтому код в реальности делает не то, что кажется на первый взгляд: мы приклеиваем лексему `DEC_` к неразвёрнутому тексту параметра `DEC(3)`, и на этом всё останавливается. Этому горю помочь несложно, достаточно спрятать склеивание на один уровень глубже:

```
ssloy@home:~$ gcc -P -E - <<<'
#define CONCAT(a,b) a##b
#define DEC(n) CONCAT(DEC_,n)
#define DEC_0 0
#define DEC_1 0
#define DEC_2 1
#define DEC_3 2

DEC(3)
DEC(DEC(3))
DEC(DEC(DEC(3)))
> '
2
1

```

```
0
```

Я объявил макрокоманду склеивания двух лексем `CONCAT`, и все проблемы пропали, декремент прекрасно работает, оперируя сразу численными константами, а не выражениями. Обратите внимание, что в данном коде я не могу декрементировать, например, 4. Вполне правомерен вопрос: а насколько это вообще разумно, хотеть определять по макрокоманде на каждое численное значние? Краткий ответ: про какую разумность может идти речь при программировании чисто на лексере?! Развёрнутый ответ: в данном случае декремент идёт по глубине рекурсии, и очень редко она бывает нужна дальше десятка-двух уровней.

Ветвление

Итак, мы познакомились с самым главным: мы можем генерировать новые лексемы путём склеивания кусков, и эти лексемы могут быть именами других макрокоманд! В таком случае, ветвление труда не составит. Давайте посмотрим на следующий код:

```
ssloy@home:~$ gcc -P -E - <<<'
#define IF_ELSE(b) CONCAT(IF_,b)
#define IF_0(i) ELSE_0
#define IF_1(i) i ELSE_1
#define ELSE_0(e) e
#define ELSE_1(e)
IF_ELSE(1)(then body)(else body)
IF_ELSE(0)(then body)(else body)
'
then body
else body
```

`IF_ELSE` - это макрокоманда, принимающая в качестве аргумента исключительно 0 или 1, и которая генерирует либо лексему `IF_0`, либо лексему `IF_1` путём тривиального склеивания. IF_0 - это команда, которая генерирует лексему ELSE_0, по пути просто съедая свои собственные аргументы. Ну а `ELSE_0` - это просто тождественное отображение. Давайте проследим всю цепочку развёртывания

`IF_ELSE(0)(then body)(else body)`:

```
IF_ELSE(0)(then body)(else body)
IF_0(then body)(else body)
ELSE_0(else body)
(else body)
```

Разворачивание с аргументом 1 происходит совершенно аналогично.

Проверка на равенство нулю

Теперь вы закалённые метапрограммисты, и не испугаетесь простой проверки на ноль :)

```
ssloy@home:~$ gcc -P -E - <<<'
#define SECOND(a, b, ...) b
#define TEST(...) SECOND(__VA_ARGS__, 0)
#define ISZERO(n) TEST(ISZERO_ ## n)
#define ISZERO_0 ~, 1
ISZERO(0)
ISZERO(3)
'

1
0
```

Давайте разбираться. Когда мы развёртываем `ISZERO(n)`, то первое, что мы делаем (ещё раз смотрим в порядок разворачивания макрокоманд) - это склейка. В данном примере я тестировал `ISZERO(0)` и `ISZERO(3)`. Во втором случае мы генерируем лексему `ISZERO_3`, а вот в первом `ISZERO_0`, которая является именем уже существующей макрокоманды! А вот она, в свою очередь, разворачивается в список `~,1` . Это ключевой момент: Только передавая ноль в команду `ISZERO`, мы внутри получим что-то, содержащее запятую. При передаче любого другого числа получится просто несуществующий токен. Нет, конечно, ничто нам не

помешает засунуть в аргументы `ISZERO` не число, а что-то, разворачивающееся так же, как и `ISZERO_0`, но это же си, тут каждый может выстрелить себе в ногу, если ему так хочется :)

Оставшуюся работу делает вариативная макрокоманда `SECOND`, которая возвращает свой второй аргумент. Мы точно знаем, что она получит на вход как минимум два аргумента: `ISZERO(3)` разворачивается в `SECOND(ISZERO_3, 0)`, и в итоге получается 0. Ну а `ISZERO(0)` разворачивается в `SECOND(~,1,0)` и сводится к 1. Значок тильды ~ здесь - это не магическая команда, он просто выбран от балды, поскольку вероятнее всего создаст синтаксическую ошибку при баге в наших макросах.

Рекурсия

Итак, мы научились делать декремент, научились ветвить код по равенству нулю, остался последний рывок. Если вы сумели продраться через равенство нулю, вам теперь должно быть море по колено.

Очень важно понимать, что все подстановки в макросах выполняются во время работы лексера, ещё **ДО** запуска парсера. И именно парсер по идее должен заниматься рекурсиями (привет, полнота по Тьюрингу у темплейтов). А создатели лексера приложили значительные усилия, чтобы **рекурсивных вызовов не допустить**.

Это необходимо для того, чтобы избежать бесконечной рекурсии при разворачивании макросов. Например, рассмотрим такой случай:

```
ssloy@home:~$ gcc -P -E - <<<'
#define FOO F BAR
#define BAR B FOO
 FOO
BAR
'
F B FOO
B F BAR
```

Работает это так: препроцессор знает, какие макросы он разворачивает, и если во время раскрытия одного из макросов он снова его встречает, то "помечает синим цветом" (жаргон компиляторщиков) повторное появление лексемы, и оставляет её дальше как есть.

Допустим, мы хотим развернуть лексему FOO. Препроцессор заходит в состояние "разворачиваем FOO", обрабатывает лексему F . Но когда он разворачивает макрос BAR, то снова встречает лексему FOO , и сразу же её помечает, запрещая дальнейшее раскрытие. Примерно такая же история при раскрытии макроса BAR.

А теперь давайте хитрить!

```
ssloy@home:~$ gcc -P -E - <<<'
#define FOO() F BAR
#define BAR() B FOO
FOO()()()()()()()()()
'
F B F B F B F B F BAR
```

Любопытно, любопытно. А что же произошло? А случилась крайне интересная вещь: мы превратили макрокоманду FOO в команду с параметрами. Давайте проследим два уровня развёртывания:

```
FOO()()()()()()()()()
F BAR()()()()()()()()
F B FOO()()()()()()() <- вот тут FOO не помечен синим!
```

Когда мы второй раз встретились с FOO, лексер его не узнал, поскольку сгенерировал лексему FOO **без параметров**. И на этом обработка одной FOO закончилась. А затем лексер продолжил, обнаружил скобки, и вызвал FOO во второй раз. А потом в третий. И четвёртый...

И тут мне пошла карта. Только вот рекурсия бывает не только хвостовой, но даже для хвостовой мне не хочется генерировать заранее все параметры :)

Напоминаю, что прямой вызов FOO() внутри BAR не работает, нужно именно остановить контекст развёртывания макрокоманды прежде, чем встретится лексема этой же макрокоманды с параметрами. И, оказывается, это несложно сделать. Для начала давайте добавим пустую макрокоманду EMPTY() :

```
ssloy@home:~$ gcc -P -E - <<<'
#define EMPTY()
#define FOO() F BAR EMPTY() ()
#define BAR() B FOO EMPTY() ()
FOO()

#define EVAL(x) x
EVAL(FOO())
EVAL(EVAL(FOO()))
EVAL(EVAL(EVAL(FOO())))
EVAL(EVAL(EVAL(EVAL(FOO()))))
'
F BAR ()
F B FOO ()
F B F BAR ()
F B F B FOO ()
F B F B F BAR ()
```

Теперь при попытке разворачивания `FOO()` лексер создаёт лексему `F`, лексему `BAR`, не совпадающую с именем макрокоманды `BAR()`, обрабатывает пустую лексему, оставляет скобки `()` как есть. Всё, контекст `FOO()` завершился, и ничто не было покрашено синим! Только вот разворачивается `FOO()` в весьма скучное `F BAR ()`, ровно как и раньше. Что же мы выиграли? А смотрите дальше по коду. Я определил макрокоманду тождественного отображения `#define EVAL(x) x`, и вот уже развёртывание `EVAL(FOO())` сильно интереснее. Давайте проследим всю цепочку (освежите в памяти правила развёртывания, особенно последнее):

```
EVAL(FOO()) <- заходим в контекст EVAL
FOO()       <- на этом этапе происходит развёртывание лексем параметров EVAL
F BAR ()    <- единственный параметр EVAL развёрнут
F B FOO ()  <- ПОВТОРНОЕ СКАНИРОВАНИЕ после разворачивания параметров EVAL!
```

Ну, собственно, и всё. Обернув в два EVAL, пройдём дальше. В целом нужно (примерно) столько же обёрток EVAL, какая у нас глубина рекурсии, это можно обеспечить всего несколькими строчками кода. Давайте соберём всё вместе, напоминаю, что нам нужно получить подобие вот этого сишного кода, но с рекурсией на макросах!

```
ssloy@home:~$ gcc -xc - <<<'
#include <stdio.h>
void foo(int d) {
    printf("%d ", d);
    if (d!=0) foo(d - 1);
}
int main() {
    foo(3);
    return 0;
}
' && ./a.out
3 2 1 0 ssloy@home:~$
```

Да не вопрос вообще! Это простой копи-пейст вышеприведённых кусков кода. Единственный момент, с которым пришлось быть аккуратным, это ещё один уровень задержки на строчке 29, поскольку лексема BAR генерируется внутри команды ветвления, и нам нужно подождать ещё одну итерацию EVAL, чтобы BAR не покрасился.

```
ssloy@home:~$ gcc -xc - <<<'
#include <stdio.h>
#define CONCAT(a,b) a##b

#define DEC(n) CONCAT(DEC_,n)
#define DEC_0 0
#define DEC_1 0
#define DEC_2 1
```

```
#define DEC_3 2

#define IF_ELSE(b) CONCAT(IF_,b)
#define IF_0(i) ELSE_0
#define IF_1(i) i ELSE_1
#define ELSE_0(e) e
#define ELSE_1(e)

#define SECOND(a, b, ...) b
#define TEST(...) SECOND(__VA_ARGS__, 0)
#define ISZERO(n) TEST(ISZERO_ ## n)
#define ISZERO_0 ~, 1

#define EMPTY()
#define FOO(d) \
    printf("%d ", d); \
    IF_ELSE(ISZERO(d)) \
    ( ) \
    ( BAR EMPTY EMPTY() () (DEC(d)) )
#define BAR(d) FOO EMPTY() (d)

#define EVAL(x)  EVAL1(EVAL1(EVAL1(x)))
#define EVAL1(x) EVAL2(EVAL2(EVAL2(x)))
#define EVAL2(x) EVAL3(EVAL3(EVAL3(x)))
#define EVAL3(x) x

int main() {
    EVAL(FOO(3))
    return 0;
}
```

```
' && ./a.out
3 2 1 0 ssloy@home:~$
```

Ну а исходник проклятого огня можно найти тут. Как и обещал, весь фреймбуфер хранится в скопище отдельных переменных, никаких массивов!

Полон ли по Тьюрингу препроцессор C?

В разговорной речи термин "Тьюринг-полный" означает, что любой реальный компьютер общего назначения или компьютерный язык может приблизительно моделировать вычислительные аспекты любого другого реального компьютера общего назначения или компьютерного языка. Ни одна реальная система не может иметь бесконечную память, но если пренебречь ограничением конечной памяти, то большинство языков программирования в остальном являются Тьюринг-полными.

У препроцессора конечна не только память, но и количество уровней ре-сканирования лексем (которое мы задаём при помощи `EVAL`), но ведь это всего-навсего одна из форм ограничений по памяти, так что в обывательском смысле препроцессор вполне себе полон по Тьюрингу.

Уже после написания этой статьи, я нашёл ещё одну ссылку, где граждане натурально забабахали метаязык программирования исключительно на дефайнах, но это уже психопатство запредельного уровня, и, к сожалению, слишком сложное для первого знакомства с чорной магией препроцессора. Трюки, которые показал я, вполне ещё могут пойти в продакшн, а вот metalang99 - сомневаюсь :)

Бонус

Современные оптимизирующие компиляторы - это, пожалуй, самое сложное и впечатляющее творение человечества в области программной инженерии. Но это не значит, что там есть что-то магическое. Обычный человеческий мозг в несколько секунд скажет, во что должен скомпилироваться нижеприведённый весьма тривиальный код, а вот gcc понадобится куча экзабайт памяти и много-много лет, для того, чтобы дать ответ.

▸ Hidden text

Послесловие

Не забудьте выйти из хаба "ненормальное программирование", и вернуться ко вполне нормальному. Have fun!

Если эта публикация вас вдохновила и вы хотите поддержать автора — не стесняйтесь нажать на кнопку