Всероссийский конкурс исследовательских и проектных работ школьников «Высший пилотаж»

Создание макропроцессора для поддержки метапрограммирования в языке C

Проект
Направление ”Computer Science”

Архипов Тимофей Сергеевич
учащийся 11 класса,
ЧОУ ОО МШСО, г. Москва

2025 г.

Содержание

1 Введение

Язык C (а именно, самый популярный его стандарт C89 ”ANSI”[4]) был введен в 1989 году. Позднее появлялись многие другие его стандарты, но, исторически, макропроцессор в нем всегда являлся *текстовым препроцессором*.

Текстовый препроцессор раскрывает макросы, не проводя при этом синтаксического разбора, основной его задачей является примитивная подстановка текста.

1.1 О недостатках препроцессора

У подобного подхода есть некоторые минусы. Рассмотрим классическую ситуацию, когда требуется использовать в теле макроса набор операторов:

```
#define FOO(f) { printf("calling " #f "\n"); f(); }

void myfunc() { /* ... */ }

int main(void) {
    FOO(myfunc)
    return 0;
}
```

Однако пример ниже уже не скомпилируется:

```
if (condition)
    FOO(myfunc);
else
    /* ... */
```

Раскрыв макрос причина ошибки становится ясна:

```
if (condition)
    { printf("calling myfunc\n"); myfunc(); };
else
    /* ... */
```

Из этой ситуации придумали выходить следующим образом:

```
#define FOO(f)\
    do { \
        printf("calling " #f " \n");\
        f();\
    } while(0)
```

Теперь предыдущий пример компилируется, но, очевидно, что писать длинные макросы таким образом становится крайне неудобно.

1.2 О современных макропроцессорах

В качестве примера, рассмотрим макропроцессор в языке Rust.

В Rust макросы являются частью синтаксического дерева, а процедура их раскрытия – преобразованием узлов дерева.

Таким образом макросы могут быть использованы на месте строго определенного множества синтаксических элементов.

В случае Rust макросы могут быть использованы на месте:

1. Паттернов
2. Операторов
3. Выражений
4. Элементов языка (в т.ч. `impl`)
5. Типов

Но ***не могут*** замещать идентификаторы, поля структуры, варианты оператора `match`[6].

1.3 О разнице в подходах

Рассмотрим следующий пример на Rust:

```
macro_rules! some_expr
{
    () => {
        2 + 5
    }
}

fn main() {
    println!("{}", 2 * some_expr!());
}
```

Чтобы понять, каким будет результат выполнения кода, проанализируем действия компилятора.

Для выражения `2 * some_expr!()` компилятор посторит следующее синтаксическое дерево:

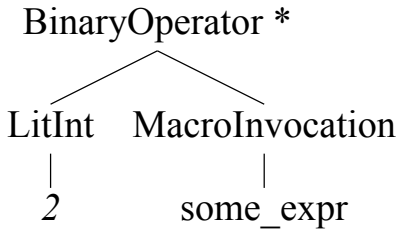


Рис. 1: Дерево до раскрытия макросов

После раскрытия макроса, дочерний узел будет преобразован следующим образом:

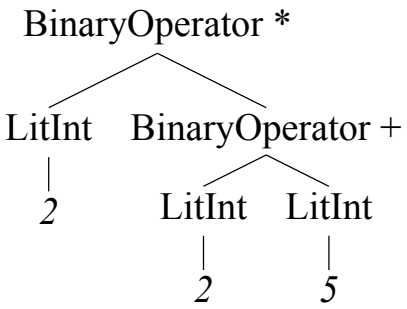


Рис. 2: Дерево после раскрытия макросов

Таким образом, результатом вычисления станет число 14.

Аналогичный пример рассмотрим на языке C:

```
#define SOME_EXPR() 2 + 7

int main(void)
{
    printf("%d", 2 * SOME_EXPR());
    return 0;
}
```

После раскрытия макросов получаем:

```
int main(void)
{
    printf("%d", 2 * 2 + 7);
    return 0;
}
```

Соответственно программа выведет число 11.

Обобщая суть подходов, можно сказать, что преобразование макросов в C – *текстовая операция*, а преобразование макросов в Rust – *структурная*.

2 Анализ существующих решений

Единственным аналогичным языком, частично выполняющим поставленные задачи, является язык C++.

В C++ (начиная со стандарта C++11) присутствует библиотека для метапрограммирования, однако, язык, все же не имеет **полной** обратной совместимости, а также стоит заметить, что C++, помимо парадигмы, предлагает намного больше функционала, чем это, порой, требуется.

3 Постановка задачи

Задачей данного проекта является частичное интегрирование функционала макропроцессора Rust в язык C для повышения функциональных возможностей языка.

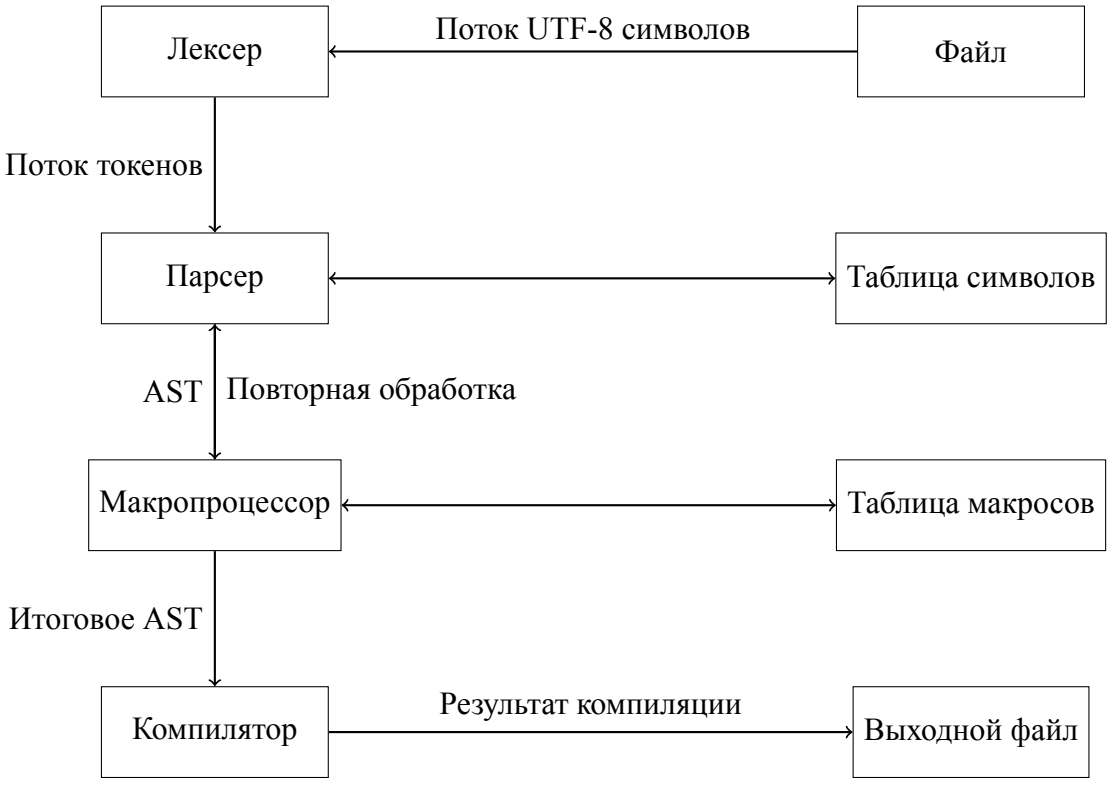
4 Реализация

Для реализации проекта был выбран язык Rust. Весь исходный код доступен в открытом репозитории на GitHub[1].

Обработка языка разделена на следующие этапы:

1. Лексический анализ
2. Синтаксический анализ (здесь и далее – парсинг)
3. Раскрытие макросов (см. 4.3)
4. Ре-парсинг раскрытых макросов
5. Компиляция

Схематично это можно изобразить следующим образом:



[1] https://github.com/timar07/cmex/

4.1 Анализ языка C

Для синтаксической обработки языка C был выбран алгоритм рекурсивного спуска [7, с. 104]. При реализации парсера не было использовано сторонних библиотек, исходный код доступен в директории `compiler/cmex_parser`[2].

В качестве грамматического подмножества была выбрана грамматика стандарта ANSI. Из модификаций:

1. Разрешено смешение деклараций и операторов, как в поздних стандартах языка:

```
void foo() {
    int a;
    bar();
    /* ... */
    int b;
}
```

2. Не поддерживается устаревший стиль деклараций функций (т.н. K&R style):

```
void foo(a, b)
    char a;
    int b;
{
    /* ... */
}
```

3. Поддерживается GNU расширение ”statement expression”[3]

```
void foo() {
    printf("%d", ({
        int a = 5;
        a += 2;
        /* ... */
        a;
    });
}
```

[2]https://github.com/timar07/cmex/tree/main/compiler/cmex_parser
[3]https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html

4.2 Исследование абстрактного синтаксического дерева

Для проверки корректности результирующего синтаксического дерева, можно запустить программу с флагом `-ast-dump`, например для следующего ввода:

```
macro_rules! four {
    () => { 3 + 1 };
}

int main() {
    printf("%d", 2 * four!());
    return 0;
}
```

В стандартный поток вывода будет выведен следующий результат:

```
TranslationUnit
|-Macro `four`
`-FuncDecl `main`
  `-CompoundStmt
    |-ExprStmt
    | `-CallExpr
    |   |-PrimaryExpr
    |   `-BinaryOperator `*`
    |     |-PrimaryExpr
    |     `-Invocation `four`
    `-ReturnStmt
```

Если вместе с флагом `-ast-dump` подать флаг `-E`, то будет выведено дерево после макропреобразований:

```
TranslationUnit
|-Macro `four`
`-FuncDecl `main`
  `-CompoundStmt
    |-ExprStmt
    | `-CallExpr
    |   |-PrimaryExpr
    |   `-BinaryOperator `*`
    |     |-PrimaryExpr
    |     `-BinaryOperator `+`
    |       |-PrimaryExpr
    |       `-PrimaryExpr
    `-ReturnStmt
```

4.3 Интегрирование макропроцессора в семантику языка

В Rust доступен следующий список метапеременных:

- `block`: блок, т.н. «составной оператор», однако в Rust может также быть выражением
- `expr`: выражение
- `ident`: идентификатор или ключевое слово
- `item`: элемент языка, такой как функция, структура, реализация и т.д.
- `lifetime`: время жизни (например `'a`, `'static`)
- `literal`: литерал (например 123, ”string ’a’)
- `meta`: метаэлемент, используемый внутри атрибута (`#[....]`)
- `pat`: паттерн
- `path`: путь (например `::std::fmt::Display`)
- `stmt`: оператор
- `tt`: дерево токенов
- `ty`: тип
- `vis`: квалификатор видимости (например `pub`, `pub(crate)`, ...)

Для соответствия семантике языка C, были исключены следующие метапеременные:

- `lifetime`: в языке отсутствует система заимствований
- `meta`: в языке отсутствуют атрибуты
- `pat`: см. 8
- `path`: в языке отсутствует система модулей
- `vis`: избыточно, с учетом того, что единственный доступный квалификатор видимости – `static`

В остальном оригинальная грамматика макропроцессора Rust [5] тронута не была.

4.4 Алгоритм сопоставления паттерна с образцом

Парсинг паттерна в Rust реализуется алгоритмом, по своей сути очень схожим с алгоритмом Эрли[1][2]. Рассмотрим суть работы алгоритма:

1. Паттерн делится на отдельные составляющие, с помощью которых легко можно отслеживать текущее положение парсера (здесь и далее, текущее положение парсера будет изображаться символом •). Например, следующий паттерн:

   ```
   a a $(a),+ b; $(c)*
   ```

   будет преобразован в такой набор «особых токенов» (каждый отдельный «токен» взят в кавычки):

   ```
   'a' 'a' '$(' 'a' ')' ',' '+' 'b' ';' '$(' 'c' ')' '*' <END>
   ```

   Здесь и далее коллекция терминалов и нетерминалов будет обозначаться греческой буквой, единичный нетерминал – $\mathcal{A}, \mathcal{B}, \mathcal{C}$..., единичный терминал – a, b, c... Нотация взята из источника [3, с. 192]

2. Для любого положения парсера k создается набор возможных следующих положений $S(k+1)$ по данным правилам:

   - Для любого терминального символа a, если он соответствует символу из паттерна, передвинуть положение на одну позицию вперед. Пример:
     Паттерн: a
     Ввод: a
     Возможные позиции: a •
   - Для последовательностей вида $(α)* или $(α)? создается два возможных положения – $(•$\alpha$) и $(α) •:
   - Для последовательностей $(α)•+ и $(α)•* можно либо пропустить повторение, либо попробовать еще раз: $(α)+ •, $(•$\alpha$)+

   - Если в последовательности есть разделитель, то можно либо пропустить его, вместе с оператором повторения, либо, если текущий токен соответствует b, переместить положение вперед на одну позицию:
     $(α) b + •
     $(α) b •+
   - Для операторов повторения + и * после разделителя, возможен только один переход – на начало повторения:

- Наконец, для любого единичного нетерминального символа $\mathcal{A}$ производится перемещение на одну позицию вперед только тогда, когда парсинг данного символа завершился успешно: $\bullet\mathcal{A} \to \alpha\bullet$

Итак, имея функцию следующего положения, для каждой текущей позиции вызывается функция $S(k)$, стек переходов заполняется следующим образом ($\vdash$ обозначает конец ввода):

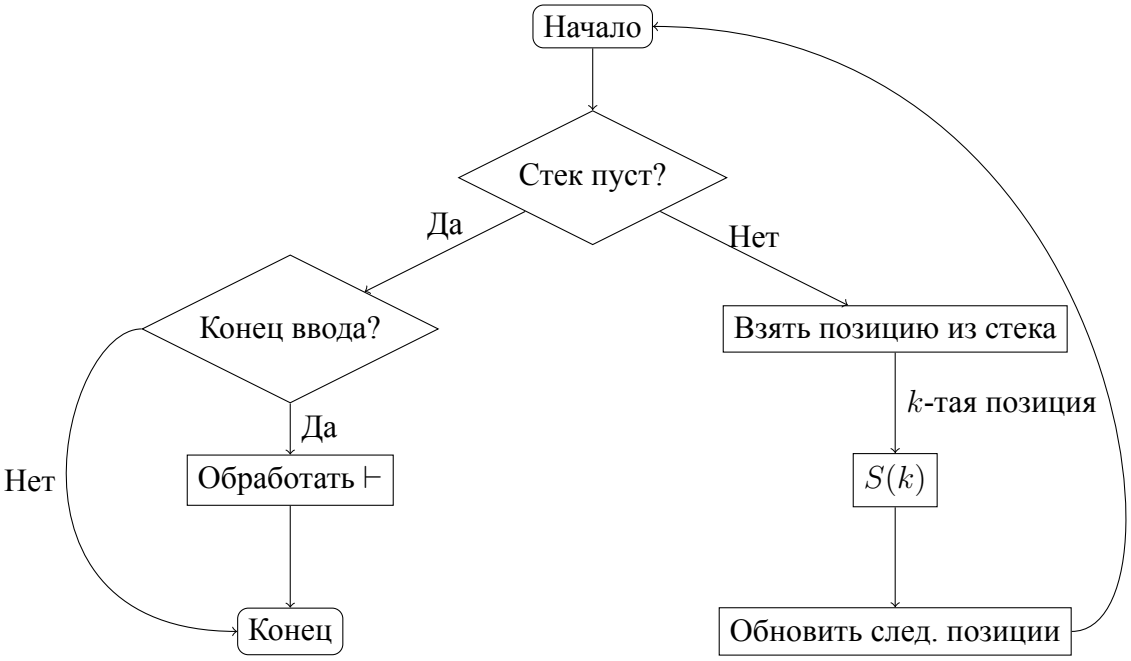


Рис. 3: Заполнение стека возможных переходов

После заполнения, рассматривается состояние стека:

- Если стек пуст – ошибка;
- Найдены возможные следующие позиции – добавить в стек;
- Найден нетерминальный символ – вызывать парсер, добавить в стек;
- Найден нетерминальный символ, при этом также найдены возможные другие позиции – ошибка неоднозначности.

Алгоритм сопоставления последовательно применяется к каждой из ветвей `macro_rules` до тех пор, пока хотя бы одна из ветвей не будет соответствовать вводу. Например:

```
macro_rules! ordering {
    (test) => {
        printf("1");
    };
    ($a:ident) => {
        printf("2");
    }
```

```
}

/* ... */
ordering!(test);
```

Хотя ввод и может соответствовать обеим ветвям, но сопоставление останавливается на *первой подходящей ветви*.

Полная реализация макропроцессора находится в директории `compiler/cmex_macros`[4]

4.5 Ограничения макропроцессора

Ввиду всего вышеперечисленного, рассмотрим пример ошибки сопоставления:

```
macro_rules! ambiguity {
    ($($a:ident)* $b:ident) => { /* ... */ }
}
```

Для примера рассмотрим сопоставление со строкой `a b c`:

1. Текущие позиции: •`a b c`
2. $S(1) = \{$ `$( •$a:ident)* $b:ident`, `$($a:ident)* •$b:ident` $\}$
3. На данном этапе переменой `$a` и `$b` могут соответствовать разные значения: `$a` $= \{$ `a` $\}$ или `$a` $= \{\}$, соответственно компилятор выводит ошибку неоднозначности (см. 6.2)

5 Архитектура программы

Все модули программы оформлены как библиотеки и находятся в директории `compiler`.
Программа состоит из следующих модулей:

- `cmex_driver`: главный модуль компилятора;
- `cmex_ast`: структура абстрактного синтаксического дерева, а также некоторые полезные процедуры, такие как дамп дерева (см. 4.2);
- `cmex_lexer`: лексический анализатор;
- `cmex_parser`: парсер языка C а также расширений синтаксиса;
- `cmex_macros`: модуль, ответственный за анализ и раскрытие макросов. В нем также реализован собственный парсер синтаксиса макросов;
- `cmex_symtable`: таблица символов. Используется также на этапе синтаксического анализа, так как от определения имени зависит выбор грамматического правила;

[4]https://github.com/timar07/cmex/tree/main/compiler/cmex_macros

- `cmex_errors`: модуль для построения ошибок и диагностик;
- `cmex_span`: вспомогательный модуль для определения положения лексемы. Используется на этапе диагностики и выведения ошибок;
- `cmex_source`: также вспомогательный модуль для эффективного взаимодействия с обрабатываемым файлом.

5.1 Отладка компилятора

В проекте использовалась библиотека `tracing`[5] позволяющая легко производить логирование.

Логи для конкретного модуля можно включить переменной окружения `RUST_LOG=[<path-to-module>]`.

6 Использование макропроцессора

Иллюстративные примеры исходных файлов можно найти в директории `examples`[6]

6.1 Примеры использования

Рассмотрим типичный пример создания вектора на `cmex`:

```
macro_rules! vec {
    [] => {
        ({
            /* Empty vector */
            struct vec v;
            v;
        })
    };
    [capacity = $cap:expr; $($x:expr),*] => {
        ({
            struct vec v;
            vec_init_capacity(&v, $cap);
            $(vec_push(&v, $x);)*
            v;
        })
    };
    [$($x:expr),*] => {
        ({
```

[5] https://docs.rs/tracing/latest/tracing/

[6] https://github.com/timar07/cmex/tree/main/examples

```
                struct vec v;
                vec_init(&v);
                $(vec_push(&v, $x);)*
                v;
            })
        }
    }
```

Теперь можно удобно создавать вектор:

```
struct vec *a = vec![1, 2, 3];
struct vec *b = vec![capacity = 5; 1, 2, 3, 4, 5];
```

Для примера рассмотрим результат компиляции данного примера (результат отформатирован):

```
int main(void)
{
    struct vec a = ({
        struct vec v;
        vec_init(&v);
        vec_push(&v, 1);
        vec_push(&v, 2);
        vec_push(&v, 3);
        v;
    });
    struct vec b = ({
        struct vec v;
        vec_init_capacity(&v, 5);
        vec_push(&v, 1);
        vec_push(&v, 2);
        vec_push(&v, 3);
        vec_push(&v, 4);
        vec_push(&v, 5);
        v;
    });
    return 0;
}
```

Рассмотрим более сложный пример для создания словаря (хеш-таблицы):

```
macro_rules! dict {
    ($($key:ident => $val:expr),*) => {
```

```
        ({
            struct dict *d = init_dict();
            $(dict_set(d, $key, $val);)*
            d;
        })
    };
}
```

Теперь данным макросом можно создавать хеш-таблицы с удобным синтаксисом как в высокоуровневых языках:

```
struct dict *table = dict!{
    "foo" => 5,
    "bar" => 10
};
```

6.2 Диагностика

Любая ошибка выводится в следующем формате:

```
<path> <error tag>: <message>
<context>
```

Где `path` – путь к обрабатываемому файлу, `error tag` – тег ошибки, обозначающий фазу компиляции, в процессе которой произошла ошибка, `message` – текст ошибки, `context` – обрабатываемый участок кода.

Рассмотрим вывод ошибки из листинга предыдущей главы (4.5):

```
./test.cmex MacroError: match error: ambiguity error
38 |     ($($a:ident)* $b:ident) => {
```

7 Заключение

В ходе проведенной работы был разработан собственный макропроцессор для языка C, позволяющий вводить элементы метапрограммирования для уменьшения объема повторяющегося кода и повышения выразительности языка.

Раскрытие макросов происходит прямо во время компиляции, поэтому введенные макросы не влияют на производительность программы.

Также была достигнута лучшая диагностика, позволяющая выявить некоторые ошибки компиляции во время синтаксического анализа.

8 Перспективы развития проекта

На данном этапе проект реализован в качестве препроцессора для языка C, однако в дальнейшем можно реализовать полноценную компиляцию в машинный код с использованием готовых бекендов (например, LLVM[7]).

Также возможна реализация обработки некоторых других метапеременных, например `pat`.

[7] https://llvm.org/

A Грамматика

A.1 Грамматика макропроцессора

⟨*macro-rules-definition*⟩ ::= ’macro_rules’ ’!’ IDENTIFIER ⟨*macro-rules-def*⟩

⟨*macro-rules-def*⟩ ::= ’{’ ⟨*macro-rules*⟩ ’}’

⟨*macro-rules*⟩ ::= ⟨*macro-rule*⟩ (’;’ ⟨*macro-rule*⟩)* ’;’?

⟨*macro-rule*⟩ ::= ⟨*macro-matcher*⟩ ’=>’ ⟨*macro-transcriber*⟩

⟨*macro-matcher*⟩ ::= ’(’ ⟨*macro-match*⟩* ’)’
| ’[’ ⟨*macro-match*⟩* ’]’
| ’{’ ⟨*macro-match*⟩* ’}’

⟨*macro-match*⟩ ::= TOKEN
| ⟨*macro-matcher*⟩
| ’$’ (IDENTIFIER | KEYWORD) ’:’ ⟨*macro-frag-spec*⟩
| ’$’ ’(’ ⟨*macro-match*⟩+ ’)’ ⟨*macro-sep*⟩? ⟨*macro-rep*⟩

⟨*macro-frag-spec*⟩ ::= ’block’ | ’expr’ | ’ident’ | ’item’ | ’literal’ | ’stmt’ | ’tt’ | ’ty’

⟨*macro-sep*⟩ ::= TOKEN

⟨*macro-rep*⟩ ::= ’+’ | ’*’ | ’?’

Список литературы

[1] Earley parser // Википедия. URL: https://en.wikipedia.org/wiki/Earley_parser (дата обращения: 14.11.2024).

[2] Earley, J., ”An efficient context-free parsing algorithm”URL: https://dl.acm.org/doi/pdf/10.1145/362007.362035

[3] Holub I. Compiler design in C

[4] Kernighan W., Ritchie M. The C Programming Language URL: https://venkivasamsetti.github.io/ebookworm.github.io/Books/cse/C%20Programming%20Language%2 (дата обращения: 22.10.2024).

[5] Macros By Example // The Rust Reference. URL: https://doc.rust-lang.org/reference/macros-by-example.html#r-macro.decl.syntax (дата обращения: 18.10.2024).

[6] Macros in the AST // The Little Book of Rust Macros. URL: https://veykril.github.io/tlborm/syntax-extensions/ast.html (дата обращения: 4.10.2024).

[7] Альфред В. Ахо, Джеффри Д. Ульман Компиляторы: принципы, технологии и инструментарий. – 2-е изд. – М. : И.Д. Вильямс, 2018. – 1184 с.