Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Системное программирование

Аникин Егор Георгиевич

Реализация расширенного препроцессора для проекта РуСи

Бакалаврская работа

Научный руководитель:
зав. каф. СП, д.ф.-м.н., проф. А.Н. Терехов

Рецензент:
Директор ООО ”Новые Мобильные Технологии” В.В. Оносовский

Санкт-Петербург
2020

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
System Programming

Anikin Egor Georgievich

Implementing an Advanced Preprocessor for a RuC Project

Bachelor's Thesis

Scientific supervisor:
chair head, D.Sc, prof. Andrey Terekhov

Reviewer:
CEO LLC New Mobile Technologies Valentin Onossovski

Saint-Petersburg
2020

Содержание

Введение

Языки программирования – область, которая активно развивается на протяжении всей истории развития IT, начиная с 50-х годов прошлого века. Наряду с ультрасовременными языками, основанными на последних разработках, такими как Swift или Kotlin, продолжается и развитие традиционных языков, существующих с 80-х годов прошлого века. Это в полной мере относится к языку C, простота и эффективность которого сделала его «фактическим стандартом» для программирования встроенных систем и систем реального времени.

На кафедре системного программирования профессором А.Н.Тереховым в течение последних лет разрабатывается язык РуСи [7] [8], который является развитием стандартного языка C в сторону повышения надежности и безопасности программирования. Перспективным направлением является использование РуСи в проектах, связанных с разработкой встроенного ПО реального времени для различных специальных применений.

По сравнению со стандартным C, РуСи обладает рядом преимуществ: поддержка кириллического алфавита, возможность обходиться без использования потенциально опасных конструкций (например, арифметики указателей) и значительно более подробная диагностика ошибок. Существенным недостатком текущей версии транслятора языка РуСи является невозможность построения сложной архитектуры кода, так как не поддерживается работа с проектами, включающими в себя множество файлов исходных текстов.

Важной компонентой большинства С-подобных языков является препроцессор, основная задача которого — первоначальное преобразование кода, упрощение до основной части синтаксического анализа. Это реализуется путём выполнения специальных команд — препроцессорных директив. Часто используются такие директивы, как добавление файлов и библиотек, условная компиляция, подстановка макросов. В больших промышленных проектах могут потребоваться более специализированные средства, например циклы, сложные условия, переназначение макросов. Отдельного внимания заслуживает разработка переназначения макросов с помощью арифметического счёта. Это позволяет проводить генерацию результата вычисления на этапе препроцессирования.

К сожалению, в текущей версии транслятора РуСи препроцессор отсутствует, что и обусловило актуальность данной работы.

Целью данной дипломной работы является создание препроцессора для языка РуСи. Для достижения этой цели были поставлены следующие задачи.

- Проектирование архитектуры препроцессора.
- Реализация функциональности, что включает в себя следующие директивы:
  - #include,
  - #define,
  - #if, #ifdef и #ifundef,
  - #macro,
  - #eval,
  - #while,
  - #set,
  - #undef.
- Разработка нескольких способов добавления файлов для корректной работы с большими проектами.
- Реализация возможности детектирования ошибок в директивах макроязыка.
- Проведение тестирования препроцессора.

Базой для данного исследования послужили работы Терехова А.Н. [7] [8], Тернера К.Д. [5], Столлмана Р.М. [4].

Данная работа состоит из обзорной главы, в которой подробно рассматриваются особенности существующих препроцессоров, и трех глав, описывающих общую архитектуру, реализацию и тестирование.

Терминология

В данной работе используются следующие термины.

- Директива препроцессора представляет собой инструкцию, записанную в коде и выполняемую до трансляции.
- Макрос или макровыражение — это лексема, последовательность символов, которую можно заранее определить и впоследствии использовать, также у неё могут быть параметры.
- Идентификатор макроса – имя, назначенное макросу.
- Препроцессирование – алгоритм, просматривающий входные ”.c” файлы, исполняющий в них директивы препроцессора и включающий другие файлы, указанные в директивах #include. В результате получается код, в котором не содержатся директивы препроцессора, и все используемые макросы раскрыты.
- Лексический анализатор – элемент компилятора, который обрабатывает поток символов исходной программы, объединяя их в значащие последовательности символов, лексемы. Он обрабатывает код, полученный после работы препроцессора.

1. Обзор

1.1. Существующие препроцессоры

Для C/C++ наиболее популярным является препроцессор CPP[1], используемый в проекте gcc. Среди других распространенных препроцессоров представляют интерес PP[2], который весьма распространён и удобен в работе с различными типами данных, и m4, хорошо подходящий для создания сложных макросов. Весьма интересен оригинальный препроцессор, созданный Антоном Москалём, который тоже работает со сложными макросами и даёт вариативность работы с разделителями. Рассмотрим эти препроцессоры более подробно.

Создатели проекта gcc сформулировали ряд базовых требований к препроцессору CPP, которые можно рассматривать как основу для сравнения.

Согласно официальной документации gcc [4], CPP обладает следующими свойствами:

- распространённостью;
- понятным и привычным синтаксисом;
- базовой функциональностью, включающей в себя специальные символы и директивы:
  - добавления файлов,
  - объявления макросов,
  - условной компиляции;
- большим набором предопределённых макросов;
- открытым исходным кодом.

[1] CPP — препроцессор языка C.

[2] PP (generic Preprocessor (with Pandoc[3] in mind)) — языко-независимый препроцессор, объединяющий функциональностью GPP[4] и DPP[5].

[3] Pandoc — универсальный конвертер документов

[4] GPP — препроцессор общего назначения с настраиваемым синтаксисом.

[5] DPP — препроцессор диаграмм (с учетом pandoc).

Существует расширенный препроцессор для C / C++, созданный Антоном Москалём [6], на кафедре системного программирования, который по сравнению с вышеописанным обладает следующими дополнительными возможностями:

- описание многострочных макрокоманд, содержавших внутри себя операторы препроцессора;
- использование произвольного числа аргументов и произвольные разделители между ними;
- поддержка различных типов скобок;
- вычисление арифметических и логических операций;
- циклы по спискам;
- присваивание переменных;
- перегрузка макроопределений как по типу скобок, так и по числу аргументов.

Использование произвольных разделителей, различных типов скобок и перегрузки макросов является уникальной особенностью данного препроцессора, отсутствующей в других рассматриваемых продуктах.

Наиболее полное описание препроцессора PP представлено в [3]. Его от CPP отличает:

- синтаксис, не похожий на синтаксис более распространённых препроцессоров;
- дополнительная функциональность, включающая, в частности:
  - вычисление арифметических и логических операций,
  - набор директив для работы с файлами,
  - набор директив для работы с форматами данных,
  - поддержку написанных на Bash, Cmd, Power Shell, Python, Lua, Haskell и R скриптов;
- большой набор предопределённых макросов.

Тернер Кеннет в [5] предоставляет подробное описание макропроцессора m4. Этот препроцессор обладает следующими характерными отличиями:

- язык директив этого препроцессора является самостоятельным макроязыком, использующимся независимо от языка программирования;
- вычисление арифметических и логических операций;
- нестандартный и менее распространенный синтаксис;
- имеется отдельный набор директив для работы со строками;
- возможность определения макросов в момент использования других макросов;
- рекурсивное переопределение макросов.

При реализации транслятора РуСи рассматривались различные варианты включения препроцессора, в частности, варианты интеграции с одним из существующих.

Синтаксис языка РуСи схож с Си, поэтому целесообразно использовать соответствующий препроцессор. CPP был подходящим по синтаксису, но не обладал достаточной функциональностью, например, в нём отсутствуют препроцессорные циклы. Попытка изменения препроцессоров с открытым кодом под необходимые условия требует подробного изучения принципов работы этих препроцессоров, что значительно сложнее реализации нового продукта.

Наиболее подходящим кандидатом являлся препроцессор Антона Москаля, так как его функциональность больше всего подходила требованиям, однако эта идея была отклонена из-за отсутствия открытого исходного кода, недостаточной документации и менее привычного синтаксиса. После ознакомления с описанием использования директив некоторые из них были реализованы в рамках текущей работы. Варианты использования препроцессоров PP и m4 также были отклонены из-за неподходящего синтаксиса.

1.2. Проект РуСи

Язык РуСи подробно описан в [7], [8]. Описание процесса трансляции приведено в [1], [2]. РуСи имеет следующие особенности:

- большое количество типов отлавливаемых ошибок, таких как ошибка выхода индекса за пределы массива;
- высокая скорость работы;
- фиксированный объём оперативной памяти;
- переносимость;
- поддержка русского языка в идентификаторах и ключевых словах;
- расширение возможностей ввода/вывода, а именно автоматическое определение типа вводимой или выводимой переменной и приём или отображение информации в соответствующем виде;
- отказ от потенциально опасных операций, в частности, отсутствует арифметика указателей.

Препроцессор использует следующие инструменты РуСи:

- функции работы с русскими символами из лексического анализатора,
- утилита получения следующего символа,
- утилита записи символа в результирующий текст.

2. Архитектура

2.1. Функциональные требования к препроцессору

Разрабатываемый препроцессор должен удовлетворять следующим функциональным требованиям:

- объявлять и использовать макросы, которые могут содержать входные параметры и сложную внутреннюю структуру, в которую могут входить другие директивы, в том числе и директива объявления макроса;
- переопределять в рамках одного файла макросы с простой структурой и без параметров;
- создавать блоки условной компиляции, которые могут быть вложенными или взаимоисключающими;
- создавать циклы, выполняющиеся на этапе препроцессора, с поддержкой вложенности;
- вычислять арифметические выражения с использованием макросов;
- обеспечить контроль ошибок, связанный с использованием директив;
- сопоставлять местоположение в исходном файле с местом, указанным компилятором при выводе ошибки кода РуСи;
- обеспечивать объединение компилируемых файлов и их передачу лексическому анализатору.

2.2. Проектирование

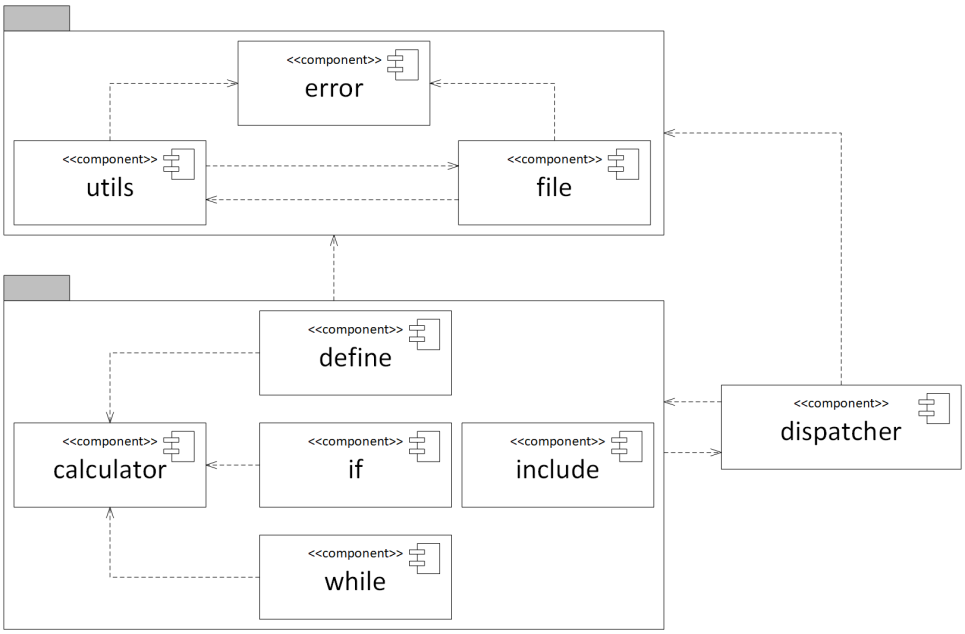

Рис. 1 Архитектура препроцессора

Диаграмма компонентов (Рис. 1) отражает общую архитектуру препроцессора. Компоненты препроцессора разделены на три следующие большие группы.

- Первая — вспомогательные. Каждый элемент этого блока используется другими компонентами. Данный блок содержит следующие компоненты:
  - error, отвечающий за корректный вывод ошибок препроцессора;
  - file, реализующий чтение информации из файла и запись результатов обработки препроцессора;
  - utils — часто используемые утилиты.

- Вторая — основной блок, каждый компонент которого отвечает за реализацию близких по смыслу директив:
  - calculator отвечает за препроцессорный счёт, выполняющийся посредством директивы #eval, которая может быть использована в других директивах;
  - define осуществляет работу с макросами:
    * #define — объявление простого макроса,
    * #macro — объявление макроса со сложной структурой,
    * #undef — удаление простого макроса без параметров,
    * #set — изменение простого макроса без параметров;
  - if ответственен за работу с ветками условной компиляции с помощью директив #if, #ifdef, #ifundef и #else, а также отвечает за контроль вложенности блоков условной компиляции;
  - while сохраняет препроцессорный цикл #while и воспроизводит тело цикла, пока выполняется условие;
  - include отвечает за добавление содержимого файлов в то место, где прописана директива #include.
- Третья состоит из одного главного компонента dispatcher, который инициализирует и контролирует всю работу препроцессора. Он обрабатывает компилируемые файлы. Если встречаются ключевые слова препроцессора, вызывается компонент, отвечающий за работу с соответствующей директивой. dispatcher может использовать элементы второго блока во время обработки текста внутри тела директивы.

3. Реализация базовой и расширенной функциональности

Функциональность препроцессора можно разделить на базовую и расширенную. В базовую функциональность входят часто используемые директивы, добавление файлов, объявление макросов, условная компиляция. Расширенной является функциональность, которая не входит в базовые возможности и присутствует не во всех препроцессорах. Она даёт дополнительные способы использовать препроцессор.

В настоящее время язык РуСи планируется использовать в качестве одного из входных языков в проекте создания промышленной технологии программирования с большим количеством строк кода. Это накладывает определённые требования, в том числе на функциональность препроцессора. Таким образом, в данной работе расширениями являются функции, циклы, объявление макросов со сложной внутренней структурой.

3.1. Реализация директив

3.1.1. Описание директив

Далее приведены реализованные директивы. Здесь используются только англоязычные варианты ключевых слов, но в препроцессоре (как и в языке РуСи в целом) имеются их русскоязычные аналоги.

#define

#define <имя> <список формальных параметров> <тело макроса>

<имя> — может являться любым идентификатором.

<тело макроса> — текст замены, который идёт до конца строки. Внутри могут быть использованы другие макросы и директива *#eval* (Пример 1).

<список формальных параметров> — имена параметров. Макрос может быть как с ними, так и без (Примеры 2 и 3).

Если в конце строки присутствует символ ‘/’, то следующая строка обрабатывается как продолжение текущей (Пример 4). Определение макроса можно удалить с помощью директивы *#undef* (Пример 5).

```
#define A #eval(5 + 10) // A = 15 а не (5 + 10)
```

Пример 1

```
#define N 5 // N = 5
```

Пример 2

```
#define S(A, B) (A + B) // S(N, 7) = (5 + 7)\\
```

Пример 3

```
#define PI 3.1415/
9265359 // PI = 3.14159265359
```

Пример 4

```
#undef N // N = null
```

Пример 5

#if

<условный оператор> <условие>

<ветка условной компиляции>

#elif <условие>

<ветка условной компиляции>

…

#else

<ветка условной компиляции>

#endif

<условный оператор> — *#if*, *#ifdef* или *#ifundef*.

<условие> — если используется условный оператор *#if* или *#elif*, условие может содержать логические операторы, операторы сравнения, макросы, директиву *#eval* (пример 6). Если *#ifdef* или *#ifundef*, условие — это имя макроса (пример 7). Условие истинно если:

- *#if*: выражение верно,
- *#ifdef*: макрос определён,
- *#ifundef*: макрос не определён.

<ветка условной компиляции> — код, отправляемый на компиляцию, если условие верное, или в случае *#else* , когда все предыдущие условия не верны. *#elif*, *#else* – создают дополнительные ветки, могут, но не обязаны присутствовать. *#elif* может быть использован только когда условным оператором является *#if* (пример 8).

Блоки условной компиляции поддерживают вложенность.

```
#if (A > #eval(10 / 3 )) && (A == 5) // если условие истинно
#define B 0 // определить B как 0
#endif // если ложно ничего не делаем и идём дальше
```

Пример 6

```
#ifdef N // если N определён
#define B 0 // определить B как 0
#endif
```

Пример 7

```
#if N  == 0 // если N определён
#define B 0
#elif N > 5
#define B 0 //на компиляцию идёт эта ветка если,  N не равен 0 и бол
#endif
```

Пример 8

#macro

#macro используется для определения сложных макросов. Данная директива похожа на директиву *#define*, только текст замены идёт до *#endm* и может содержать любые другие директивы препроцессора (Пример 9).

```
#macro Sig(X)
#If X >= 0
1 // макрос будет заменён на 1 если X > 0
#else
(-1) // и на (-1) если меньше 0
#endm
```

Пример 9

#set

#set используется для переопределения простого макроса без параметров. Синтаксис такой же как, у *#define*, только идентификатор должен быть определён на момент использования данной директивы.

```
#set N  100// N = 100
```

Пример 10

#while

#while <условие>

<тело цикла>

#endw

<условие> — аналогично условию при *#if*.

<тело цикла> — код, который сохраняется и воспроизводится, пока условие верно (Пример 11).

Циклы поддерживают вложенность (Пример 12).

```
#define J 0
#while J < K  //пока  J  < K
#eval(J + 10) + h++; // вставляемое выражение
#set J  #eval(J+1) // изменить счётчик
#endw
#undef J
```

Пример 11

```
#define J 0
#while J < N
#define K 0
#while K < N
A(J, K)   // вставляемое выражение
#set K  #eval(K+1)
#endw
#undef K
#set J  #eval(J+1)
#endw
#undef J
```

Пример 12

#eval

#eval используется для вычислений на этапе работы препроцессора (Пример 13). Если *#eval* используется не внутри директивы *#macro*, то все значения его макросов должны быть определены на момент обработки директивы *#eval* препроцессором (Пример 14).

```
#define S #eval(1.25 * (1.3 - 1) + 31/5) //  S = 6.575
```

Пример 13

```
#macro LIST (K, M)
#define J 0
#while J < K
#eval(J + M + 1) // на момент определения LIST, M не определён
#set J  #eval(J+1)
#endw
#undef J // удалить J (например чтобы потом использовать как
#endm  // переменную)
```

Пример 14

#include

#include используется для добавления файлов. Допустимо прописывание относительного пути до имени файла через “/”, для выхода из текущей папки используется “..” (Пример 15). Более подробно *#include* рассматривается в разделе 3.2

```
#include ’’../libs/lib1.h’’
/* выйти из текущий папки, зайти в libs открыть lib1.h */
```

Пример 15

3.1.2. Особенности реализации конструкции #if

Данная директива реализуется следующим образом:

- условие сразу обрабатывается функцией `calculator`, которая преобразует инфиксную нотацию в постфиксную и реализует счёт на основе обратной польской записи;
- если условие истинно в этой ветви, идёт обычная обработка кода, пока не встретится лексема `#elif`, `#else` или `#endif`;
- если ещё раз встретилась лексема `#if`, `#ifdef` или `#ifundef`, то препроцессор повторно входит в функцию обработки условия, делает всё необходимое и возвращается из рекурсивного вложения функции;
- если условие ложно, код до конца этой ветки игнорируется, а также отслеживается одинаковое количество входов и выходов из условной компиляции;
- если есть ещё одна ветвь, процесс повторяется;
- если после обработки ветви с истинным условием остались другие, они пропускаются, при этом отслеживается вложенность.

3.1.3. Особенности реализации конструкции #while

Данная директива реализована следующим образом:

- сохранение цикла:
  - условие сохраняется в локальную переменную препроцессора `ifstring`,
  - в переменную `wstring` записывается специальный символ начала цикла, далее значение индекса `ifstring`, с которого начинается условие, потом пропускается один символ, затем идёт тело цикла до лексемы *#endw*,
  - если встречается директива *#while*, процесс сохранения цикла повторяется рекурсивно,
  - когда встретится лексема *#endw*, в пропущенный символ записывается количество символов, сохранённых в `wstring` на данный момент;
- воспроизведение цикла:
  - проверяется условие цикла, считанное с `ifstring` от указателя до символа конца строки (в ближайших символах `wstring` хранится информация о соответствующем условии и размер цикла),
  - если условие верно, тело цикла обрабатывается до конца как обычный код, а после указатель на обрабатываемый символ возвращается в начало цикла,
  - если встретится специальный символ начала цикла, функция воспроизведение цикла вызывается рекурсивно,
  - если условие ложно, обработка продолжается с конца текущего цикла,
  - кода обработается последний символ строки `wstring`, цикл будет завершён, и обработка когда продолжится в обычном режиме.

3.1.4. Особености реализации конструкции #eval

`#eval` имеет следующие особенности.

- Данная директива преобразует выражения в скобках в постфиксную нотацию, после чего реализует арифметический счёт на основе обратной польской записи.
- Помимо цифр и арифметических операций внутри выражения могут быть макросы. Все макросы должны быть определены до этого момента, за исключением того случая, когда директива используется внутри тела директивы `#macro`.
- Макросы раскрываются сразу в момент обнаружения идентификатора внутри выражения и обрабатываются, как и всё остальное. Если после раскрытия макросов встречается что-либо отличное от цифр или арифметических операций, выдается сообщение об ошибке. На этом этапе отлавливаются макросы, результат раскрытия которых не является арифметическим выражением.
- По умолчанию вычисления считаются целочисленными, однако, если хотя бы одно из двух чисел является числом с плавающей точкой, то и результат их бинарных операций тоже будет числом с плавающей точкой.

Данный способ позволяет сразу проводить вычисления и избежать ситуации хранения выражения вместо числа и счёта каждый раз при появлении соответствующего идентификатора. Например, в цепочке макросов, где каждый следующий является результатом вычисления предыдущих, описанная ситуация сильно бы замедлила работу (пример 16).

```
#define A 5
#define B #eval(A + 3) // 8, а не (5+3)
#define C #eval(A + B) // 13, а не (5+(5+3))
```

Пример 16

3.1.5. Особености реализации конструкции #macro

Особенностями `#macro` являются:

- как и директива `#define`, `#macro` может быть либо с параметрами, либо без;
- тело директивы располагается под идентификатором до `#endm`;
- при объявлении данной директивы её тело никак не обрабатывается, сразу же сохраняется;
- когда встречается идентификатор, итератор над входным потоком переключается на место сохранения тела макроса и обрабатывают его так, как был бы обработан обычный код внутри файла;
- внутри директивы могут быть любые другие директивы: `#define`, условная компиляция, циклы, и даже другая директива `#macro`;
- директива позволяет создавать сложные, повторяющееся с небольшими изменениями выражения, зависящие от параметров. (пример 14).

3.2. Включение файлов

В проекте РуСи на этапе препроцессора происходит склеивание всех файлов в один текст, который потом подается на вход лексического анализатора. Это необходимо поскольку текущая версия транслятора РуСи обрабатывает только один входной файл. При этом возникает проблема контроля области видимости. Необходимо определить, в каком файле какие написанные на РуСи функции могут использоваться. Реализация проверки области видимости не входит в задачи данной работы. Однако, чтобы обозначить начало нового файла, в общий текст добавляется специальный маркер, а также ссылка на информацию о файле.

3.2.1. Способ добавления

Предусмотрено три способа добавления файлов.

Первый — параметром передать в РуСи путь к файлу конфигурации, в котором указаны все ".c" файлы. Сначала идёт предварительная обработка их содержимого, в ходе которой запоминаются все директивы `#include`, ссылающиеся на файлы заголовков. Также в файле конфигурации прописаны пути, которые используются для поиска include файлов. После предварительной обработки идёт полная обработка препроцессором, сначала всех заголовочных ".h" файлов, а затем — ".c" файлов с исходными текстами.В последствии, файлу конфигурации может быть использован для создания собственной системы сборки РуСи.

Второй способ добавления — это уже упомянутая директива `#include`. Заголовочные файлы должны быть сохранены на предварительном этапе обработки и последовательно собраны в начале результирующего текста. Их необходимо прописать в начале файла, чтобы избежать некоторых ситуаций. Например, такой ситуацией может быть наличие директивы `#include` внутри блока условной компиляции, который не обрабатывается на предварительном этапе. Директива `#include` позволяет решить эту задачу. Файлы с исходным кодом, включаемые через `#include` не требуют перемещения и подстав-

ляются на место использования данной директивы.

Третий способ добавлять файлы — напрямую передавать список их имен в составе строки параметров при вызове транслятора РуСи. Обработка будет идти так же, как и из файла конфигурации.

3.2.2. Реализация добавления и обработки файлов

При предварительной обработке каждый компилируемый файл записывается во временную строку. По ходу обработки проверяется наличие директив `#include`, ссылающихся на заголовочные файлы, и сохраняются макросы.

Работа с макросами не должна зависеть от порядка обработки файлов. Проблема возникает при изменении (`#set`) или удалении (`#undef`) макроса. Для её устранения необходимо ввести ограничения. Макросы, которые как-либо изменили своё состояние после объявления, будем называть локальными, остальные — глобальными. Локальные макросы можно объявлять, изменять, использовать или удалять только в рамках одного файла. Глобальные макросы могут быть использованы во всех файлах.

Файлы с декларацией обрабатываются один раз, поэтому пользователю необходимо убедится, что использованные в них макросы будут объявлены выше. Это можно сделать, добавив файл, в котором они объявлены посредством директивы `#include`.

При добавлении нового файла его имя сохраняется в массив, который предаётся лексическому анализатору вместе с текстом. Это необходимо для решения двух задач: контроля повторений и передачи лексическому анализатору маркеров с ссылкой на название файла. Контроль повторений обеспечивает однократную обработку заголовочных файлов. Пользователю не нужно следить за этим и использовать такие конструкции как `#ifundef` A, `#define` A … `#endif`. Маркеры используются при контроле области видимости, а также для указания исходного файла, при выводе сообщения об ошибки.

3.3. Вывод ошибок

3.3.1. Ошибки препроцессора

При использовании препроцессора необходимо убедиться, что директивы и их параметры синтаксически корректны, и соблюдён контроль типов макросов. При возникновении нарушений необходимо сообщить об ошибке с указанием файла и местоположения ошибки.

Чтобы расширенные возможности не приводили к большему числу ошибок, необходимо каждое расширение сопровождать соответствующим контролем его ошибок.

Препроцессор отлавливает следующее ошибки:

- no_space_after_preprocessor_directive — нет переноса строки после директивы препроцессора;
- no_space_after_ident — нет пробела после идентификатора;
- ident_begins_with_letters — идентификатор должен начинаться с буквы;
- must_be_endif — условный оператор препроцессора должен заканчиваться ’#ENDIF’;
- exclude_elif — в этом типе условного оператора не может использоваться ’#ELIF’ ;
- preprocessor_directive_not_found — в препроцессоре не существует данной директивы;
- not_enough_parameters — у этого идентификатора меньше параметров чем необходимо;
- function_ident_begins_with_letters — идентификатор с параметрами должен начинаться с буквы;
- functions_cannot_be_changed — идентификатор с параметрами нельзя переопределять;

- put_comma_after_function_ident — после идентификатора в функции должны быть скобки или запятая и пробел;
- incorrect_use_of_arguments — в функции аргументы должны быть описаны через запятую, в скобках;
- before_endif — перед '#ENDIF' должен стоять условный оператор препроцессора;
- repeated_ident — этот идентификатор препроцессора уже используется;
- comment_not_closed — длинный комментарий, не закрыт "/*";
- excess_parameters — у этой функции больше параметров;
- file_does_not_end_with_define — файл не может закончится до окончания команды '#DEFINE', поставьте перенос строки;
- brackets_not_closed — количество открывающих скобок не соотвествует числу закрывающих;
- put_bracket_after_eval — сразу после команды '#EVAL' должен быть символ открывающейся скобки;
- number_exceeds_limitations — слишком большое число;
- put_digit_after_exp — после экспоненты должно быть число;
- arithmetic_operations_not_allowed — внутри директивы '#EVAL()' должны быть только арифметические выражения;
- logical_operations_not_allowed — внутри команды '#EVAL()' не должно быть логических операций;
- global_macro_changed — макрос изменён не в том файле, в котором объявлен.

3.3.2. Многофайловость

Перед началом обработки нового файла препроцессор записывает в результирующий текст маркер начала файла. Также препроцессор сам хранит информацию о текущем файле. Это необходимо в случае возникновения ошибки, чтобы вывести исходный код данного файла до нужного места.

С появлением директив `#set` и `#undef` заменим необходимость контролировать область видимости макросов. Нельзя опереться на предположение, что порядок обработки файлов определён корректно с точки зрения порядка использования макросов. Поэтому все изменяемые макросы должны быть в пределах одного файла. Сообщение об ошибке выдаётся при изменении или удалении глобального макроса не в своём файле или при использовании в другом файле локального макроса.

3.3.3. Изменения вывода ошибок в РуСи

Процесс вывода ошибок в РуСи до появления препроцессора происходил следующим образом: отображалось сообщение об ошибке и написанный пользователем код до нужного символа в строке.

В ходе работы была выявлена необходимость изменения вывода ошибок, так как после обработки кода препроцессором меняется как количество строк, так и их содержание. Требуется возможность по строке в результирующем тексте определить место в исходном файле. Для этого передаётся информация о соотношении номеров строк до и после преобразования. После каждого символа конца строки в обработанный препроцессором текст включаются специальные метаданные. Они обозначают, на сколько строк уменьшился текст при обработке директив, и сколько новых строк появилось из-за препроцессорных циклов. Во время работы лексического анализатора эти метаданные изменяют переменную, которая отвечает за количество строк в исходном файле, соответствующее обработанному на данный момент тексту.

Также из-за неправильного использования макросов может возникнуть ошибка типов. В этом и других случаях полезно понимать, что произошло после обработки препроцессором. Поэтому, если строка, в которой совершена ошибка, содержит макрозамену, пользователю выводится строка до и после обработки препроцессором.

4. Тестирование и апробация

Набор тестовых примеров был сформирован для разных случаев использования всех директив как отдельно, так и в сочетании друг с другом. Запуск осуществлялся автоматически. Помимо автоматического тестирования, проводился анализ текста, являющегося результатом работы препроцессора.

Рассмотрим работу препроцессора на конкретном примере. Необходимо в зависимости от условия объявить или 5 глобальных переменных типа int, индексация которых начинается с 3, или 7 глобальных переменных типа double, номера которых начинаются с 1.

```
#define A 5
#define B 3
#define C 7
#define D 1

#define case1(t) concat(concat(int a,t),;)
#define case2(t) concat(concat(double b,t),;)

#macro cost(M, N)
#define j 0
#while j < M
#ifdef INTEGER
case1(#eval(j + N))
#else
case2(#eval(j + N))
#endif
#set j #eval(j+1)
#endw
#undef j
#endm

#if #eval(A+B) == #eval(C+D) || A > C
```

```
#define INTEGER 0
cost(A,B)
#else
cost(C,D)
#endif

void main()
{
// cod
}
```

Помимо описанных выше директив здесь используется предопределённый макрос конкатенации concat.

Код после обработки препроцессором:

```
int a3;
int a4;
int a5;
int a6;
int a7;

void main()
{

}
```

Поскольку условие истинно, видно объявление 5 глобальных переменных типа int, как и должно быть.

5. Заключение

В ходе выполнения данной работы были получены следующие результаты.

- Спроектирована общая архитектура препроцессора и архитектура отдельных компонент.
- Реализованы следующие директивы:
  - #include,
  - #define,
  - #if, #ifdef и #ifundef,
  - #macro,
  - #eval,
  - #while,
  - #set,
  - #undef.
- Реализована возможность добавления файлов посредством передачи их в качестве параметров транслятору РуСи, через файл конфигурации и с помощью директивы #include.
- Проведена корректировка вывода ошибок РуСи и реализован вывод собственных ошибок препроцессора.
- Проведено тестирование.

Список литературы

[1] Alfred V. Aho Monica S. Lam Ravi Sethi Jeffrey D. Ullman. compilers — principles, techniques and tools. — 2007.

[2] Harold Abelson Gerald Jay Sussman Julie Sussman. Structure and Interpretation of Computer Programs. — 2016.

[3] PP - A generic Preprocessor (with Pandoc in mind) // cdsoft. — Дата обращения : 20.12.19. — Access mode: `https://cdsoft.fr/pp/`.

[4] Richard M. Stallman Zachary Weinberg. The C Preprocessor // gcc.gnu. — Дата обращения : 20.12.19. — Access mode: `https://gcc.gnu.org/onlinedocs/cpp.pdf`.

[5] Turner Kenneth J. Exploiting the m4 Macro Language // cs.stir.ac. — Дата обращения : 20.12.19. — Access mode: `http://www.cs.stir.ac.uk/~kjt/research/pdf/expl-m4.pdf`.

[6] Москаль А. Расширенный макропроцессор для языка C/C++. — 1997.

[7] Терехов А.Н. Описание входного языка проекта РуСи. — 2019.

[8] Терехов А.Н. МитеневА.В. Терехов М.А. Виртуальная машина для проекта РуСи // ipo.spb. — Дата обращения : 20.12.19. — Access mode: `http://ipo.spb.ru/journal/index.php?article/1884/`.