МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ  
федеральное государственное бюджетное образовательное учреждение  
 высшего образования «Казанский национальный исследовательский   
технический университет им. А.Н. Туполева-КАИ»  
(КНИТУ-КАИ)

Институт компьютерных технологий и защиты информации  
 (наименование института (факультета), филиала)

Кафедра Прикладной математики и информатики  
 (наименование кафедры)

(09.04.01) Программная инженерия   
(шифр и наименование направления подготовки (специальности))

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА   
К РАСЧЕТНО-ГРАФИЧЕСКОЙ РАБОТЕ**

**по дисциплине**

**«Системное программирование»**

**Вариант 61**

Выполнил:

Проверил: доцент каф. ПМИ Сотников С.В.

Оценка\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подпись\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

“\_\_\_” \_\_\_\_\_\_\_\_\_\_\_\_\_2022г.

КАЗАНЬ 2022

Оглавление

[Постановка задачи. 3](#_Toc294912415)

[Цель работы 3](#_Toc294912416)

[Требования к программе 3](#_Toc294912417)

[Требования к макропроцессору 3](#_Toc294912418)

[Теоритический раздел. 4](#_Toc294912419)

[Основные понятия. 4](#_Toc294912420)

[Возможности макроязыка 7](#_Toc294912421)

[Структуры данных Макропроцессора 16](#_Toc294912422)

[Разработанные алгоритмы. 18](#_Toc294912423)

[Руководство пользователя. 21](#_Toc294912424)

[Описание макроязыка. 25](#_Toc294912425)

[Описание модулей. 26](#_Toc294912426)

[Описание разработанных тестов. 28](#_Toc294912427)

[Список использованной литературы. 34](#_Toc294912428)

[Листинг программы. 35](#_Toc294912429)

# Постановка задачи.

Цель работы: разработать макропроцессор.

Требования к программе:

* Программа должна работать в двух режимах: консольном и GUI режиме (графический пользовательский интерфейс);
* Переключение между режимами осуществляется в зависимости от наличия параметров во входной строке программы: если входная строка не пуста, то работа программы в консольном режиме, иначе в GUI режиме;
* Как минимум три группы ключей должна уметь обрабатывать программа:

1. Ключи, определяющие входные данные программы;
2. Ключи, определяющие выходные данные программы;
3. Ключ, использующийся для вызова справки по командной строке программы.

Требования к макропроцессору:

* Опережающее описание макроса: нет;
* Количество просмотров: 1;
* Вложенные макроопределения: да;
* Наличие макровызовов внутри макроса: нет;
* Параметры вызова макроса: смешанные;
* Метки внутри макроса: да;
* Условная макрогенерация: AIF, AGO, IF, ELSE, ENDIF, WHILE, ENDW;
* Работа в составе основного алгоритма Ассемблера: нет.

# Теоретический раздел.

## Основные понятия.

Макропроцессор - модуль системного ПО, позволяющий расширить возможности языка Ассемблера за счет предварительной обработки исходного текста программы.

Определение, которое дает ГОСТ не представляется удачным, так как оно говорит только о сокращении объема записи, а это лишь одна из возможностей, обеспечиваемых Макропроцессором. Хотя Макропроцессоры являются обязательным элементом всех современных языков Ассемблеров, аналогичные модули (Препроцессоры) могут быть и для других языков, в том числе и для языков высокого уровня. Для одних языков (Pascal, PL/1) применение средств препроцессора является опционным, для других (C, C++) - обязательным.

Важно понимать, что Макропроцессор осуществляет обработку исходного текста. Он "не вникает" в синтаксис и семантику операторов и переменных языка Ассемблера, не знает (как правило) имен, употребляемых в программе, а выполняет только текстовые подстановки. В свою очередь, Ассемблер обрабатывает исходный текст, не зная, написан тот или иной оператор программистом "своей рукой" или сгенерирован Макропроцессором. По тому, насколько Препроцессор (Макропроцессор) и Транслятор (Ассемблер) "знают" о существовании друг друга, их можно разделить на три категории:

1. *Независимые*. Препроцессор составляет отдельный программный модуль (независимую программу), выполняющую просмотр (один или несколько) исходного модуля и формирующую новый файл исходного модуля, поступающий на вход Транслятора (пример - язык C).
2. *Слабосвязанные*. Препроцессор составляет с Транслятором одну программу, но разные секции этой программы. Если в предыдущем случае Препроцессор обрабатывает весь файл, а затем передает его Транслятору, то в этом случае единицей обработки является каждый оператор исходного текста: он обрабатывается секцией Препроцессора, а затем передается секции Транслятора. (Пример - HLASM для S/390).
3. *Сильносвязанные*. То же распределение работы, что и в предыдущем случае, но Препроцессор использует некоторые общие с Транслятором структуры данных. Например, Макропроцессор может распознавать имена, определенные в программе директивой EQU и т.п. (Пример - MASM, TASM).

Основные термины, связанные с данными, обрабатываемыми Макропроцессором: макровызов (или макрокоманда), макроопределение, макрорасширение.

*Макровызов* или *макрокоманда* или *макрос* - оператор программы, который подлежит обработке Макропроцессором (как мы дальше увидим, Макропроцессор обрабатывает не все операторы, а только ему адресованные).

*Макроопределение* - описание того, как должна обрабатываться макрокоманда, макроопределение может находиться в том же исходном модуле, что и макрокоманда или в библиотеке макроопределений.

*Макрорасширение* - результат выполнения макровызова, представляющий собой один или несколько операторов языка Ассемблера, подставляемых в исходный модуль вместо оператора макровызова. Пример обработки макровызова показан на рисунке.



Оператор макровызова в исходной программе имеет тот же формат, что и другие операторы языка Ассемблера: В нем есть метка (необязательно), мнемоника и операнды. При обработке исходного текста если мнемоника оператора не распознается как машинная команда или директива, она считается макрокомандой и передается для обработки Макропроцессору.

Макроопределение описывает, как должна обрабатываться макрокоманда. Средства такого описания составляют некоторый Макроязык. Для Макропроцессоров 1-й и 2-й категорий средства Макроязыка могут быть достаточно развитыми. Для Макропроцессоров 3-й категории средства Макроязыка могут быть довольно бедными, но в составе языка Ассемблера может быть много директив, применяемых в макроопределениях (возможно, - только в макроопределениях). В теле макроопределения могут употребляться операторы двух типов:

* операторы Макроязыка, которые не приводят к непосредственной генерации операторов макрорасширения, а только управляют ходом обработки макроопределения;
* операторы языка Ассемблера (машинные команды и директивы), которые переходят в макрорасширение, возможно, с выполнением некоторых текстовых подстановок.

Поскольку макроопределение, обрабатывается перед трансляцией или вместе с ней, макрокоманда, определенная в исходном модуле, может употребляться только в этом исходном модуле и "не видна" из других исходных модулей. Для повторно используемых макроопределений обычно создаются библиотеки макроопределений. В некоторых системах (например, z/OS) макрокоманды обеспечивают системные вызовы и существуют богатейшие библиотеки системных макроопределений.

Самое очевидное применение макрокоманд - для сокращения записи исходной программы, когда один оператор макровызова заменяется на макрорасширение из двух и более операторов программы. В некоторых случаях макрорасширение может даже содержать и единственный оператор, но просто давать действию, выполняемому этим оператором более понятную мнемонику. Но возможности Макропроцессора гораздо шире. Так, одна и та же макрокоманда с разными параметрами может приводить к генерации совершенно различных макрорасширений - и по объему, и по содержанию.

Сравнение макросредств и подпрограмм

Использование макросредств во многом подобно использованию подпрограмм: в обоих случаях мы сокращаем запись исходного текста и создаем повторно используемые фрагменты кода. (Например, в C/C++ вызов псевдофункции неотличим от вызова функции.) Принципиальные различия между подпрограммами и макросредствами:

* Команды, реализующие подпрограмму, содержатся в кода загрузочного модуля один раз, а команды, реализующие макровызов, включаются в программу для каждого применения макровызова (макросредства требуют больше памяти).
* Выполнение подпрограммы требует передачи управления с возвратом - команды типа CALL и RET, а команды макрорасширения включаются в общую последовательность команд программы (макровызовы выполняются быстрее).
* Если в многофункциональной подпрограмме имеется разветвление в зависимости от значений параметров, то в загрузочный модуль включается код подпрограммы в полном объеме, даже если в конкретной программе реально используется только одна из ветвей алгоритма; в макровызове в каждое макрорасширение включаются только операторы, определяемые фактическими значениями параметров макровызова (экономия и времени и объема в макровызовах).

Общий итог сравнения: макросредства обеспечивают несколько большее быстродействие при несколько больших затратах памяти. Поэтому обычно макросредства применяются для оформления сравнительно небольших фрагментов повторяющегося кода.

## Возможности макроязыка

Ниже мы описываем некоторые возможности макроязыка, в той или иной форме реализованные во всех Макропроцессорах. Мы, однако, ориентируемся прежде всего на Макропроцессор, независимый от Ассемблера, потому что в этой категории функции Макропроцессора легче определить.

**Заголовок макроопределения**

Макроопределение должно как-то выделяться в программе, поэтому оно всегда начинается с заголовка.

Заголовок имеет формат, подобный следующему:

*имя\_макрокоманды* MACRO *список формальных параметров*

*имя\_макрокоманды* является обязательным компонентом. При макровызове это имя употребляется в поле мнемоники оператора. Имена макроопределений, имеющихся в программе, должны быть уникальны. Обычно при распознавании макровызова поиск по имени макрокоманды ведется сначала среди макроопределений имеющихся в программе, а затем (если в программе такое макроопределение не найдено) - в библиотеках макроопределений. Таким образом, имя макрокоманды, определенной в программе, может совпадать с именем макрокоманды, определенной в библиотеке, в этом случае макрокоманда, определенная в программе, заменяет собой библиотечную.

Формальные параметры играют ту же роль, что и формальные параметры процедур/функций. При обработке макровызова вместо имен формальных параметров в теле макроопределения подставляются значения фактических параметров макровызова.

В развитых Макроязыках возможны три формы задания параметров: позиционная, ключевая и смешанная. При использовании *позиционной* формы соответствие фактических параметров формальным определяется их порядковым номером. (Позиционная форма всегда применяется для подпрограмм).

Пример:

|  |  |  |
| --- | --- | --- |
| Заголовок макроопределения | Макровызов | Результат подстановки |
| M1 MACRO A,B,C | M1 X,Y,Z | A=X, B=Y, C=Z |

В позиционной форме количество и порядок следования фактических параметров макровызова должны соответствовать списку формальных параметров в заголовке макроопределения. При использовании *ключевой* формы каждый фактический параметр макровызова задается в виде:

*имя\_параметра*=*значение\_параметра*

В таком же виде они описываются и в *списке формальных параметров*, но здесь *значение\_параметра* может опускаться. Если *значение\_параметра* в *списке формальных параметров* не опущено, то это - значение по умолчанию. В макровызове параметры могут задаваться в любом порядке, параметры, имеющие значения по умолчанию, могут опускаться.

Пример:

|  |  |  |
| --- | --- | --- |
| Заголовок макроопределения | Макровызов | Результат подстановки |
| M1 MACRO A=Q,B=,C=R | M1 C=Z,B=X | A=Q, B=X, C=Z |

В *смешанной* форме первые несколько параметров подчиняются правилам позиционной формы, а остальные - ключевые.

Пример:

|  |  |  |
| --- | --- | --- |
| Заголовок макроопределения | Макровызов | Результат подстановки |
| M1 MACRO A,B,C=Q,D=,E=R | M1 X,Y,Z,D=T,E=S | A=X,B=Y,C=Q,D=T,E=S |

В некоторых Макропроцессорах имена параметров начинаются с некоторого отличительного признака (например, амперсанда - &), чтобы Макропроцессор мог отличить "свои" имена (имена, подлежащие обработке при обработке макроопределения) от имен, подлежащих обработке Ассемблером. Для Макропроцессоров, которые мы отнесли к категории сильносвязанных такой признак может и не быть необходимым, так как такой Макропроцессор обрабатывает как свои имена, так и имена Ассемблера. В любом случае возникает проблема распознавания имени в теле макроопределения. Например, если макроопределение имеет формальный параметр &P, а в макровызове указано для него фактическое значение 'X', то как должна обрабатываться подстрока '&PA' в теле макроопределения? Должна ли эта подстрока быть заменена на 'XA' или оставлена без изменений?

Логика, которой следует большинство Макропроцессоров в этом вопросе, такова. &PA является именем в соответствии с правилами формирования имен. Поэтому оно не распознается как имя &P и остается без изменений. Если мы хотим, чтобы подстановка в этой подстроке все-таки произошла, следует поставить признак, отделяющий имя параметра от остальной части строки. Обычно в качестве такого признака используется точка - '.': '&P.A' заменяется на 'XA'.

**Окончание макроопределения**

Если у макроопределения есть начало (оператор MACRO), то у него, естественно, должен быть и конец. Конец макроопределения определяется оператором MEND. Этот оператор не требует параметров. Макроопределение, взятое в "скобки" MACRO - MEND может располагаться в любом месте исходного модуля, но обычно все макроопределения размещают в начале или в конце модуля.

**Локальные переменные макроопределения**

Поскольку генерация макрорасширения ведется по некоторому алгоритму, описанному в макроопределении, реализация этого алгоритма может потребовать собственных переменных. Эти переменные имеют силу только внутри данного макроопределения, в макрорасширении не остается никаких "следов" переменных макроопределения.

Переменные макроопределения могут использоваться двумя способами:

* их значения могут подставляться вместо их имен в тех операторах макроопределения, которые переходят в макрорасширение;
* их значения могут проверяться в условных операторах макроязыка и влиять на последовательность обработки.

При подстановке значений переменных макроопределения в макрорасширение работают те же правила, что и при подстановки значений параметров.

Для сильносвязанных Макропроцессоров необходимости в локальных переменных макроопределения, вместо них могут использоваться имена программы (определяемые директивой EQU). Для сильносвязанных и независимых процессоров переменный макроопределения и имена программы должны различаться, для этого может применяться тот же признак, что и для параметров макроопределения. Объявление локальной переменной макроопределения может иметь, например, вид:

*имя\_переменной* LOCL *начальное\_значение* (последнее необязательно)

**Присваивание значений переменным макроопределения**

Присваивание может производиться оператором вида:

*имя\_переменной* SET *выражение*

или

*имя\_переменной* = *выражение*

Выражения, допустимые при присваивании, могут включать в себя имена переменных и параметров макроопределения, константы, строковые, арифметические и логические операции, функции. Основной тип операций - строковые (выделение подстроки, поиск вхождения, конкатенация. etc.), так как обработка макроопределения состоит в текстовых подстановках. Строковые операции обычно реализуются в функциях. Однако, в некоторых случаях может потребоваться выполнение над переменными макроопределения операций нестрокового типа. Как обеспечить выполнение таких операций? Можно предложить два варианта решения этой проблемы:

1. Ввести в оператор объявления переменной макроопределения определение ее типа. При выполнении операций должно проверяться соответствие типов.
2. Все переменные макроопределения имеют строковый тип, но при вычислении выражений автоматически преобразуются к типу, требуемому для данной операции (при таком преобразовании может возникать ошибка). Результат выражения автоматически преобразуется в строку.

Как правило, операции присваивания могут применяться к параметрам макроопределения точно так же, как и к переменным макроопределения.

**Глобальные переменные макроопределения**

Значения локальных переменных макроопределения сохраняются только при обработке данного конкретного макровызова. В некоторых случаях, однако, возникает необходимость, чтобы значение переменной макроопределения было запомнено Макропроцессором и использовано при следующей появлении той же макрокоманды в данном модуле. Для этого могут быть введены глобальные переменные макроопределения (в сильносвязанных Макропроцессорах в них опять-таки нет необходимости).

Объявление глобальной переменной макроопределения может иметь, например, вид:

VAR *имя\_переменной* *начальное\_значение* (последнее необязательно)

Присваивание значений глобальным переменным макроопределения выполняется так же, как и локальным.

**Уникальные метки**

В некоторых случаях операторы машинных команд, имеющихся в макроопределении, должны быть помечены, например, для того чтобы передавать на них управление. Если применить для этих целей обычную метку, то может возникнуть ошибочная ситуация. Если метка в макроопределении имеет обычное имя, и в модуле данная макрокоманда вызывается два раза, то будет сгенерировано два макрорасширения, и в обоих будет метка с этим именем. Чтобы избежать ситуации неуникальности меток, в макроязыке создается возможность определять метки, для которых формируются уникальные имена. Обычно имя такой метки имеет тот же отличительный признак, который имеют параметры и переменные макроопределения. Каждую такую метку Макропроцессор заменяет меткой с уникальными именем.

Уникальное имя метки может формироваться формате, подобном следующему:

*имя*.*nnnnnn:*

где - *nnnnnn* - число, увеличивающееся на 1 для каждой следующей уникальной метки.

Другой возможный способ формирования, например:

*имя*&SYSNDX

где SYSNDX - предустановленное имя, имеющее числовое значение, начинающееся с 00001 и увеличивающееся на 1 для каждой следующей уникальной метки.

Следующие операторы Макроязыка влияют на последовательность обработки операторов макроопределения. В тех или иных Макропроцессорах имеется тот или иной набор таких операторов.

**Оператор безусловного перехода и метки макроопределения**

Возможный формат оператора:

AGO *макрометка*

Концептуально важным понятием является *макрометка*. Макрометка может стоять перед оператором Макроязыка или перед оператором языка Ассемблера. Макрометки не имеют ничего общего с метками в программе. Передача управления на макрометку означает то, что при обработке макроопределения следующим будет обрабатываться оператор, помеченный макрометкой. Макрометки должны иметь какой-то признак, по которому их имена отличались бы от имен программы и переменных макроопределения. Например, если имена переменных макроопределения начинаются с символа :, то имя макрометки может начинаться с %.

**Оператор условного перехода**

Возможный формат оператора:

AIF *условное\_выражение макрометка*

Если *условное\_выражение* имеет значение "истина", обработка переходит на оператор, помеченный *макрометкой*, иначе обрабатывается следующий оператор макроопределения. Условные выражения формируются по обычным правилам языков программирования. В них могут употребляться параметры и переменные (локальные и глобальные) макроопределения, константы, строковые, арифметические и логические операции и, конечно же, операции сравнения. Кроме того, в составе Макроязыка обычно имеются специальные функции, позволяющие распознавать тип своих операндов, например: является ли операнд строковым представлением числа, является ли операнд именем, является ли операнд именем регистра и т.п.

**Условные блоки**

Возможный формат оператора:

IF *условное\_выражение*

*операторы\_макроопределения\_блок1*

ENDIF

ELSE

*операторы\_макроопределения\_блок2*

ENDIF

Если *условное\_выражение* имеет значение "истина", обрабатываются операторы макроопределения от оператора IF до оператора ENDIF, иначе обрабатываются операторы макроопределения от оператора ESLE до оператора ENDIF. Как и в языках программирования блок ELSE - ENDIF не является обязательным.

Условные выражения описаны выше. Обычно предусматриваются специальные формы:

IFDEF *имя*

IFNDEF *имя*

Операторы условных блоков довольно часто являются не операторами Макроязыка, а директивами самого языка Ассемблера.

**Операторы повторений**

Операторы повторений Макроязыка (или директивы повторений языка Ассемблера) заставляют повторить блок операторов исходного текста, возможно, с модификациями в каждом повторении. Операторы повторений играют роль операторов цикла в языках программирования, они не являются обязательными для макроязыка, так как цикл можно обеспечить и условным переходом.

Как и в языках программирования, в Макроязыке может быть несколько форм операторов повторения, приведем некоторые (не все) из возможных форм:

WHILE *условное\_выражение*

*блок\_операторов\_макроопределения*

ENDW

обработка *блока операторов* повторяется до тех пор, пока значение *условного\_выражения* - "истина".

**Макроопределения внутри макроопределений**

Честно говоря, необходимость в таких средствах сомнительна. Она может возникнуть при создании большого макроопределения, в котором есть повторяющиеся фрагменты. Вложенное макроопределение действительно только внутри того макроопределения, в которое оно вложено.

Против такого средства можно привести 2 соображения:

1. макроопределение не бывает слишком большим - иначе не срабатывают его преимущества над подпрограммой (следует, однако, признать, что могут существовать довольно большие макроопределения, которые генерируют разнообразные варианты небольших макрорасширений);
2. в языке Pascal допускаются вложенные процедуры, а в языке C - нет; и C прекрасно обходится без них, да и современная практика программирования на Pascal их практически не использует.

Тем не менее, если вложенные макроопределения все же необходимы, можно предложить следующий вариант их реализации:

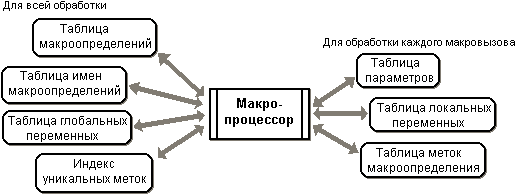
* 1-й проход Макропроцессора работает почти по тому же алгоритму. Принципиально важно, однако, что Таблица макроопределений и Таблица имен макроопределений имеют последовательную структуру, элементы в них записываются в порядке их поступления.
* В Макропроцессоре есть некоторая целая переменная - глубина вложенности. Ее исходное значение - 0, при каждом появлении оператора MACRO это значение увеличивается на 1, при каждом появлении оператора MEND - уменьшается на 1.
* Если при глубине вложенности 0 появляется оператор MACRO, в Таблицу имен макроопределений заносится новый элемент, и текст макроопределения записывается в Таблицу макроопределений - до тех пор, пока глубина вложенности не станет равной 0.
* Появление оператора MACRO при глубине вложенности, большей 0 не приводит к созданию нового элемента в Таблице имен макроопределений.
* Таким образом, в Таблице имен макроопределений имеется строка только для самого внешнего макроопределения, а все вложенные пока "не видны" и находятся внутри текста внешнего в Таблице макроопределений.
* 2-й проход Макропроцессора при обработке макровызова считывает текст макроопределения в некоторый буфер и прежде всего рекурсивно вызывает для его обработки Макропроцессор.
* Для вложенного вызова Макропроцессора доступны Таблица макроопределений и Таблица имен макроопределений, новые макроопределения, обнаруженные рекурсивным вызовом заносятся в конец этих таблиц.
* При возврате из рекурсивного вызова макроопределения, дописанные им, удаляются из таблиц.

**Макрокоманды внутри макроопределений**

В отличие от предыдущего, это средство может быть весьма полезным. Прежде всего - для часто употребляемых макрокоманд, могут быть включены в библиотеки макроопределений - системные или пользовательские. Это может весьма упростить создание новых макроопределений.

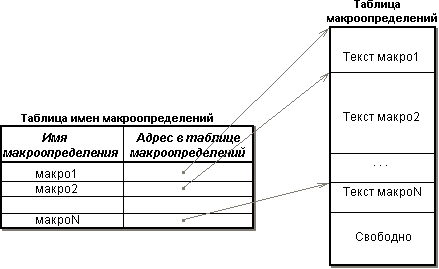
Для обеспечения такой возможности достаточно сделать рекурсивным только 2-й проход Макропроцессора. В нем несколько усложняется анализ операторов макроопределения. 2-й проход Макропроцессора должен распознавать макрокоманду и, если оператор - макрокоманда, вызывать сам себя. Распознавание макрокоманды - методом исключения: если оператор - не оператор Макроязыка, не директива Ассемблера и не машинная команда, то он считается макрокомандой и ищется в Таблице имен макроопределений. Для рекурсивного вызова создается новая Таблица локальных переменны (и параметров). Таблица глобальных переменных и индекс уникальных меток используются общие.

## Структуры данных Макропроцессора



*Таблица макроопределений*, строго говоря, не таблица, а просто массив строк, в который записываются тексты всех макроопределений (от оператора MACRO до оператора MEND), найденных в обрабатываемом модуле.

*Таблица имен макроопределений* содержит имена макроопределений и указатель на размещение текста макроопределения в таблице макроопределений, как показано на рисунке.



*Таблица глобальных переменных* имеет такую структуру:

http://khpi-iip.mipk.kharkiv.edu/library/sp/sp2/ht3_1.gif

Все таблицы имеют переменный размер и заполняются в процессе работы.

*Индекс уникальных меток* - число, используемое для формирования уникальной части имен меток, встречающихся в макроопределениях

Для обработки каждого макровызова создаются:

*Таблица параметров*, содержащая информацию о параметрах макроопределения.

*Таблица локальных переменных*, содержащая информацию о локальных переменных макроопределения.

Структура этих таблиц - такая же, как и таблицы глобальных переменных, эти две таблицы могут быть объединены в одну таблицу параметров и локальных переменных.

*Таблица меток макроопределения*, структура которой:

http://khpi-iip.mipk.kharkiv.edu/library/sp/sp2/ht3_2.gif

# Разработанные алгоритмы.

**1) Алгоритм 1 прохода.**

1. Обнуляем счетчик макроопределений.
2. Организуем цикл по обработке строк исходного файла:
   1. Если в поле МКОП встретилась директива END, то проверяем значение счетчика макроопределений:
      1. Если> 0 – ошибка, макрос не описан до конца
      2. Иначе происходит завершение 1 прохода и вывод результата
   2. Иначе если в поле МКОП встретилась директива VAR и счетчик макроопределения = 0, обрабатываем эту строку по алгоритму 3 и переходим к следующей строке
   3. Иначе если в поле МКОП встретилась директива SET и счетчик макроопределения = 0, обрабатываем эту строку по алгоритму 4 и переходим к следующей строке
   4. Иначе если в поле МКОП встретилась директива INC и счетчик макроопределения = 0, обрабатываем эту строку по алгоритму 6 и переходим к следующей строке
   5. Иначе если в поле МКОП встретилась директива MEND
      1. Если счетчик макроопределения> 0, занести строку в тело описываемого макроса
      2. Уменьшаем счетчик на 1
      3. Организуем цикл по всем строкам, подставляя вместо вызова данного макроса его макротело
   6. Иначе если в поле МКОП встретилась директива MACRO, то проверяем значение флага макроопределения:
      1. Если счетчик макроопределений = 0, организуем следующие проверки
         1. Если поле метки не пустое – ошибка
         2. Если макрос уже находится в ТМО – ошибка
         3. Если имя макроса уже существуют как глобальная переменная или имя макроса в ТМО - ошибка
         4. Если ошибки не было выявлено, добавляем имя макроса в ТМО
      2. Если счетчик макроопределения > 0, занести строку в тело описываемого макроса
      3. В любом случае увеличить счетчик макроопределений
   7. Иначе проверяем значение счетчика макроопределений
      1. Если он > 0, дописываем строку в тело описываемого макроса
      2. Иначе если встретили директиву макровызова, осуществляем макровызов
      3. Иначе дописываем в результат текущую строку

**3) Алгоритм обработки директивы VAR**

1. На вход алгоритму поступает строка с директивой VAR в поле МКОП
2. Организуем следующие проверки:
   1. Если поле метки не пустое – ошибка
   2. Выполним поиск данной глобальной переменной в таблице глобальных переменных, среди имен макросов в ТМО.
      1. Если поиск удачен – ошибка
   3. Если количество операндов – 2, выполняются следующие проверки
      1. Если второй операнд – не целое число – ошибка, некорректное значение глобальной переменной
   4. Если количество операндов не 1 и не 2 – ошибка, некорректное количество операндов в директиве VAR
3. Если ошибки не выявлено, проверяется количество операндов
   1. Если 1 операнд, в таблицу глобальных переменных заносится имя глобальной переменной без значения
   2. Иначе в нее заносится имя переменной со значением

**4) Алгоритм обработки директивы SET**

1. На вход алгоритму поступает строка с директивой SET в поле МКОП
2. Организуем следующие проверки:
   1. Если поле метки не пустое – ошибка
   2. Если количество операндов – 2, выполняются следующие проверки
      1. Выполним поиск данной глобальной переменной в таблице глобальных переменных.
         1. Если поиск удачен – ошибка
      2. Если второй операнд – не целое число – ошибка, некорректное значение глобальной переменной
3. Иначе ошибка – некорректное количество операндов в директиве SET
4. Если ошибки не выявлено, выполняется поиск в таблице глобальных переменных данной переменной и присвоение ей указанного значения

**5) Алгоритм макрогенерации**

1. На вход алгоритму поступает строка макровызова с найденным в ТМО макросом.
2. Организуется цикл по обработке строк тела макроса
   1. Проверка на наличие меток в макросе, преобразование к уникальным меткам
3. Организуем цикл по обработке строк макроса
   1. Обработка условной макрогенерации, т. е. директив IF-ELSE-ENDIF, WHILE-ENDW
   2. Строка обрабатывается алгоритмом первого прохода
4. Результат цикла передается основной программе в виде строк ассемблерного кода

**6) Алгоритм обработки директивы INC**

1. На вход алгоритму поступает строка с директивой INC в поле МКОП
2. Организуем следующие проверки:
   1. Если поле метки не пустое – ошибка
   2. Если количество операндов – 1, выполняются следующие проверки
      1. Выполним поиск данной глобальной переменной в таблице глобальных переменных.
         1. Если поиск неудачен – ошибка
      2. Если данной глобальной переменной не задано значение – ошибка
3. Иначе ошибка – некорректное количество операндов в директиве INC
4. Если ошибки не выявлено, выполняется поиск в таблице глобальных переменных данной переменной и увеличение ее значения на 1

# Руководство пользователя.

Разработанная программа работает в двух режимах: консольном и визуальном. Если строка параметров пуста программа запускается в визуальном режиме. Для запуска макропроцессора в консольном режиме необходимо наличие хотя бы одного параметра запуска.

Работа в консольном режиме.

Запуск макропроцессора:

SystemSoftware.exe [-input\_file <имя файла>] [-output\_file <имя файла>] [ -help]

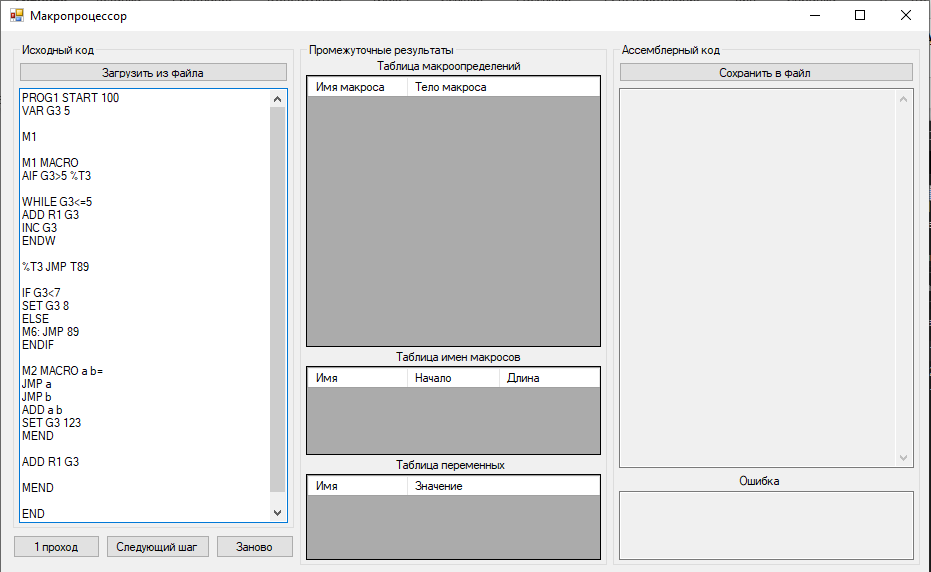
Параметры:

-input\_file <имя файла> - задает имя входного файла, по умолчанию input.txt;

-output\_file <имя файла> - задает имя выходного файла, по умолчанию output.txt;

-help - вывод справочной информации.

**Работа в визуальном режиме.**



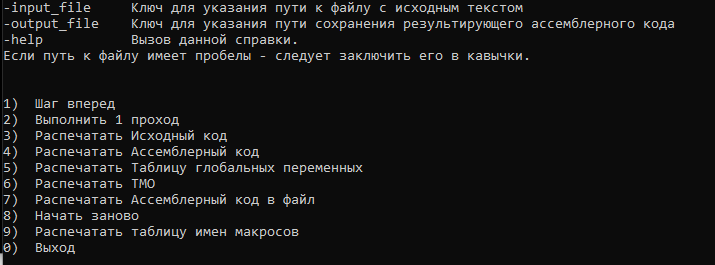
Интерфейс программы в визуальном режиме организован следующим образом:

Блок «Исходные данные» служит для загрузки и вывода содержимого входного файла макропроцессора. Снизу располагаются кнопки управления работой макропроцессора. Блок «Промежуточные результаты» служит для вывода содержимого таблицы глобальных переменных и ТМО. Блок «Результат» служит для вывода результирующего ассемблерного кода на экран и сохранении его в файле

Работа программы в визуальном режиме организована следующим образом. При запуске программы, входным файлом по умолчанию является файл «input.txt». Для загрузки другого входного файла необходимо нажать кнопку «Загрузить из файла» и выбрать новый входной файл. До запуска макропроцессора есть возможность изменять входной файл и сохранять его, нажатием на кнопке «Сохранить в файл» (кнопки располагаются соответственно над текстом исходного кода и результата).

Для запуска макропроцессора необходимо нажать кнопку «Первый проход» или «Следующий шаг». Если был установлен пошаговый режим работы, то за один шаг будет обработана только одна строка входного файла. Для принудительного завершения работы макропроцессора, нужно нажать кнопку «Заново». При этом очистятся все таблицы и поля, кроме исходного текста.

**Работа в консольном режиме**



Работа программы в визуальном режиме организована следующим образом. При запуске программы, входным файлом по умолчанию является файл, указанный в качестве параметра input\_file. Из него данные загружаются в программу и появляется меню выбора действия. Выбрать другой файл с исходными данными нельзя. При выборе пользователем некоторого пункта меню программа будет выполнять выбранное действие и выдавать результат его выполнения. После каждого действия вновь появляется меню выбора, пока пользователь не выйдет из программы (выбор «0» в меню).

Для запуска макропроцессора необходимо выбрать пункт 1 или 2. Если был установлен пошаговый режим работы, то за один шаг будет обработана только одна строка входного файла. При этом пункт меню 2 не будет выполнять своих функций. Если изначально пошаговый режим выбран не был, а был выполнен 1 проход, пошаговый режим также блокируется.

Для принудительного завершения работы макропроцессора и обновления данных, нужно выбрать пункт 8. Для просмотра таблицы глобальных переменных или ТМО необходимо выбрать пункты 7 и 6 соответственно. Для просмотра исходного кода или результирующего ассемблерного кода надо выбрать пункты 3 или 4 соответственно. В любой момент времени доступна функция распечатки результата в файл, указанный в качестве параметра output\_file, после которой он откроется в блокноте на просмотр.

# Описание макроязыка.

**Определение макроса.**

Макрос описывается с помощью директивы MACRО, перед которой указывается имя макроса. Имя макроса не должно содержать специальных символов, а так же не должно являться зарезервированным словом, метки перед ним быть не должно. Завершается описание макроса директивой MEND. Имена макросов не должны дублироваться и совпадать с глобальными переменными. Параметры в макросе смешанные.

*<Имя макроса>* ***MACRO <****Список параметров****>***

*<Список выражений>*

***MEND***

**Директива IF-ELSE-ENDIF.**

В качестве параметров могут выступать только инициализированные глобальные переменные или целые числа. Директива ENDIF обязательна, а блок ELSE может отсутствовать.

***IF*** *<параметр1><*>|<|=|<=|>=|!=*> <параметр2>*

*<тело IF>*

***ELSE***

*<тело ELSE>*

***ENDIF***

**Директива WHILE-ENDW.**

В качестве параметров операции сравнения директивы WHILE могут выступать только инициализированные глобальные переменные или целые числа. Директива ENDW обязательна.

***WHILE*** *<параметр1><***>|<|=|<=|>=|!=***> <параметр2>*

*<тело WHILE>*

***ENDW***

**Директива VAR.**

Имя переменной должно начинаться с символа и может содержать латинские буквы и цифры. Оно не должно содержать специальных символов, а так же не должно являться зарезервированным словом.

***VAR*** *<имя переменной> [<значение>]*

**Директива SET.**

Глобальная переменная должна быть описана заранее. Значение глобальной переменной может быть целым числом.

***SET*** *<имя переменной> <значение>*

**Директива INC.**

Глобальная переменная должна быть описана и инициализирована заранее. Значение глобальной переменной увеличивается на 1.

***INC*** *<имя переменной>*

Директива **AIF**.

В качестве параметров могут выступать только инициализированные глобальные переменные или целые числа. Переход осуществляется только по макрометке.

**AIF** <параметр1><>|<|=|<=|>=|!=> <параметр2> <макрометка>

Директива **AGO**.

Переход осуществляется только по макрометке.

**AGO** <макрометка>

# Описание модулей.

В модуле «Program.cs» в зависимости от наличия ключей происходит инициализация GUI программы или консольной посредством обращения к соответствующим конструкторам.

Модуль «ConsoleProgram.cs» реализует API для работы программы в консольном режиме, делегируя все вызовы классу Processor, являющемуся ядром программы через методы NextStep (следующий шаг выполнения программы), FirstRun (первый проход). Он также имеет методы для распечатывания меню выбора и справки о программе в консоль.

Модуль «VisualApp.cs» - реализует подобным же образом API для работы программы в GUI режиме, также делегируя все вызовы классу Processor.

Модуль «Processor.cs» является основным в программе. Он содержит 4 класса. Класс SourceCode реализует работу со списком строк исходников посредством методов, основными из которых являются FirstRunStep (шаг первого прохода). Остальные 3 класса – CheckSourceEntityFirstRun (проверка строк исходного кода при первом проходе), CheckBody (проверка строк тела макроса при макрогенерации) являются статическими и реализуют статические методы проверки строк кода на корректность и обработки директив WHILE-ENDW, IF-ELSE-ENDIF, VAR, SET, INC.

Модуль «CodeEntity.cs» осуществляет работу с представлением строки входного текста в объектах, соответственно инкапсулируют ее данные в поля модуля CodeEntity – Label, Operation, Operands (поля метки, МКОП и операндов). Также тут есть метод представления объекта в виде строки (переопределенный метод ToString).

Модуль «Helpers.cs» содержит вспомогательные методы и данные, используемые другими модулями, например для проверки строки на ключевое слово (IsKeyWord) или пригодность строки для использования в качестве метки (IsLabel), выполнение операции сравнения для директивы WHILE (Compare) и т.п.

Модули «MacrosStorage.cs», «Macro.cs» представляют классы для работы с ТMО. Класс MacrosStorage содержит список элементов Macro, представляющих собой описание макроса. В классах есть методы для поиска информации в ТMО (метод SearchInTMO), обновления (метод Refresh), проверки, есть ли элемент в ТМО (IsInTMO).

Модуль «MainForm.cs» является необходимым для визуального представления программы, является контейнером для графических элементов и выполняет роль контроллера – передачи пользовательских команд исполняющим классам, содержит только обработчики стандартных событий.

# Описание разработанных тестов.

Тестовый исходный код выглядит следующим образом:

PROG1 START 100

VAR G3 5

M1

M1 MACRO

AIF G3>5 %T3

WHILE G3<=5

ADD R1 G3

INC G3

ENDW

%T3 JMP T89

IF G3<7

SET G3 8

ELSE

M6: JMP 89

ENDIF

M2 MACRO a b=

JMP a

JMP b

ADD a b

SET G3 123

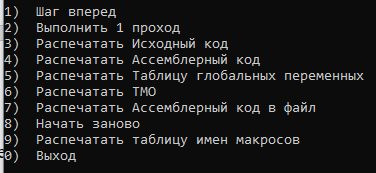
MEND

ADD R1 G3

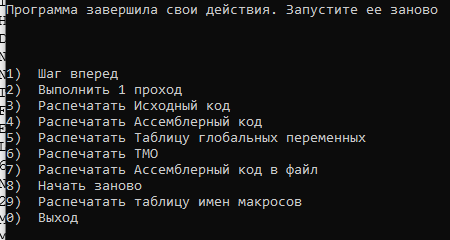
MEND

END

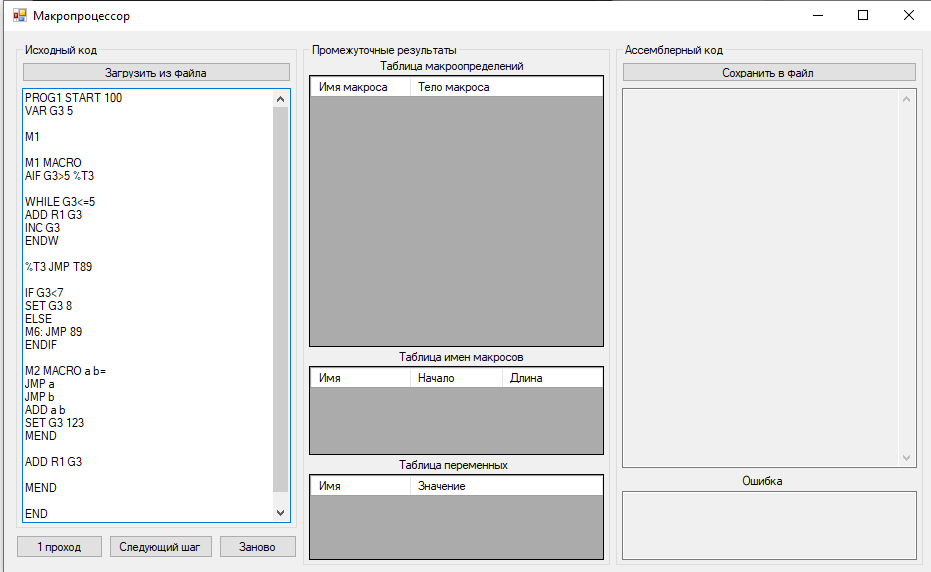
Запуск программы с параметрами входного и выходного файлов:



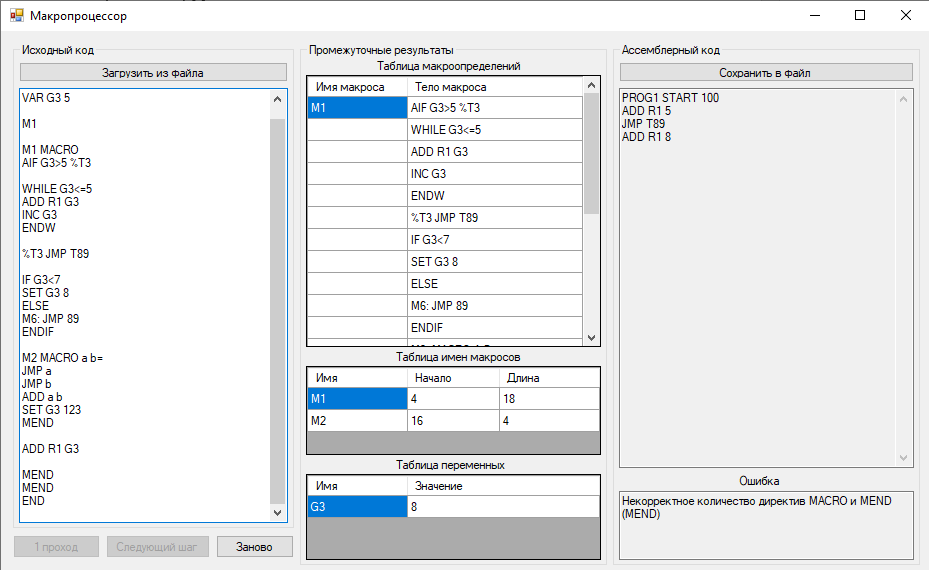
1 проход:



Запуск программы без ключей:



Выполнение обработки программы в оконном режиме с неверной командой.



# Список использованной литературы.

1. Фельдман С.К. Системное программирование на персональном компьютере. – 2е изд. – М.: Букпресс, 2006.— 512 с.
2. Бек Л. Введение в системное программирование. М. Мир. 1988г. 448 с.
3. А.С. Деревянко «Системное программирование»

# Листинг программы.

using System;

using System.Collections.Generic;

using SystemSoftware.Common;

using SystemSoftware.MacroProcessor;

namespace SystemSoftware.Interface

{

public abstract class AbstractApp

{

public static readonly string InitialInputFile = Helpers.CurrentDirectory + "\\input.txt";

public static readonly string InitialOutputFile = Helpers.CurrentDirectory + "\\output.txt";

public virtual string InputFile { get; set; } = InitialInputFile;

public virtual string OutputFile { get; set; } = InitialOutputFile;

public int SourceCodeIndex { get; set; }

public Processor SourceCode { get; set; }

public List<string> SourceStrings { get; set; }

[Obsolete("Всегда 1 проход")]

public RunMode RunMode { get; } = RunMode.FirstRun;

}

}

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using SystemSoftware.Common;

using SystemSoftware.MacroProcessor;

using SystemSoftware.Resources;

namespace SystemSoftware.Interface

{

public class ConsoleProgram : AbstractApp

{

public bool IsFirstRunEnded { get; set; }

[Obsolete("Только 1 проход", true)]

public bool IsSecondRunEnded { get; set; }

public bool IsProcessingBySteps { get; set; }

public bool IsProcessingEnded { get; set; }

public ConsoleProgram(string[] args)

{

switch (args.Length)

{

case 1:

if (args[0].ToUpper() == "-HELP")

{

Console.WriteLine(GetUserGuide());

}

else

{

throw new CustomException(ConsoleMessages.Error\_WrongCommandLineArguments);

}

break;

case 2:

if (args[0].ToUpper() == "-INPUT\_FILE")

{

InputFile = args[1];

}

else if (args[0].ToUpper() == "-OUTPUT\_FILE")

{

OutputFile = args[1];

}

else

{

throw new CustomException(ConsoleMessages.Error\_WrongCommandLineArguments);

}

break;

case 4:

if (args[0].ToUpper() == "-INPUT\_FILE")

{

InputFile = args[1];

if (args[2].ToUpper() == "-OUTPUT\_FILE")

{

OutputFile = args[3];

}

else

{

throw new CustomException(ConsoleMessages.Error\_OutputFileArgument);

}

}

else if (args[0].ToUpper() == "-OUTPUT\_FILE")

{

OutputFile = args[1];

if (args[2].ToUpper() == "-INPUT\_FILE")

{

InputFile = args[3];

}

else

{

throw new CustomException(ConsoleMessages.Error\_InputFileArgument);

}

}

else

{

throw new CustomException(ConsoleMessages.Error\_WrongCommandLineArguments);

}

break;

default:

throw new CustomException(ConsoleMessages.Error\_WrongArgumentsCount);

}

Refresh();

}

public void Refresh()

{

try

{

var temp = File.ReadAllLines(InputFile);

SourceCode = new Processor(temp);

SourceStrings = new List<string>(temp);

IsFirstRunEnded = false;

IsProcessingBySteps = false;

IsProcessingEnded = false;

SourceCodeIndex = 0;

}

catch (Exception)

{

throw new CustomException(ConsoleMessages.Error\_InputFileNotFound);

}

}

public void NextStep()

{

try

{

if (!IsProcessingEnded)

{

SourceCode.FirstRunStep(SourceCode.SourceCodeLines[SourceCodeIndex++], MacrosStorage.Root);

}

}

catch (ArgumentOutOfRangeException)

{

IsProcessingEnded = true;

}

catch (CustomException ex)

{

IsProcessingEnded = true;

Helpers.WriteInConsole($@"{ConsoleMessages.Prefix\_ErrorInstring} ""{SourceCode.SourceCodeLines[SourceCodeIndex - 1].ToString()}"": {ex.Message}");

}

if (SourceCodeIndex == SourceCode.SourceCodeLines.Count)

{

Helpers.WriteInConsole(ConsoleMessages.ApplicationRunEnded);

IsProcessingEnded = true;

return;

}

}

public void FirstRun()

{

if (IsFirstRunEnded == true)

{

Helpers.WriteInConsole(ConsoleMessages.FirstRunEnded);

return;

}

SourceCodeIndex = 0;

try

{

while (true)

{

NextStep();

if (SourceCodeIndex == SourceCode.SourceCodeLines.Count)

{

IsFirstRunEnded = true;

IsProcessingEnded = true;

return;

}

if (IsProcessingEnded) return;

}

}

catch (CustomException ex)

{

IsProcessingEnded = true;

Helpers.WriteInConsole($"{ConsoleMessages.ErrorPrefix} :{ex.Message}");

}

}

public static string GetUserGuide()

{

var n = Environment.NewLine;

var t = Separator.Tab;

return

$"{ConsoleMessages.Help\_Title}{n}" +

$"{t}{ConsoleMessages.Help\_AllowedKeys}: [-input\_file] [-output\_file] [-help]{n}" +

$"-input\_file{t}{ConsoleMessages.Help\_InputFileKey}{n}" +

$"-output\_file{t}{ConsoleMessages.Help\_OutputFileKey}{n}" +

$"-help{t}{t}{ConsoleMessages.Help\_HelpKey}.{n}" +

$"{ConsoleMessages.Help\_Footer}.{n}";

}

public static string GetProgramGuide()

{

var n = Environment.NewLine;

var s = Separator.Bracket;

var t = Separator.Space + Separator.Space;

return

$"1{s}{t}{ConsoleMessages.Menu\_NextStep}{n}" +

$"2{s}{t}{ConsoleMessages.Menu\_FirstRun}{n}" +

$"3{s}{t}{ConsoleMessages.Menu\_PrintSourceCode}{n}" +

$"4{s}{t}{ConsoleMessages.Menu\_PrintAssemblerCode}{n}" +

$"5{s}{t}{ConsoleMessages.Menu\_PrintVariablesTable}{n}" +

$"6{s}{t}{ConsoleMessages.Menu\_PrintMacrosTable}{n}" +

$"7{s}{t}{ConsoleMessages.Menu\_SaveAssemblerCodeIntoFile}{n}" +

$"8{s}{t}{ConsoleMessages.Menu\_Refresh}{n}" +

$"9{s}{t}Распечатать таблицу имен макросов{n}" +

$"0{s}{t}{ConsoleMessages.Menu\_Exit}{n}";

}

public void PrintAssemblerCode()

{

Helpers.WriteInConsole(ConsoleMessages.Print\_AssemblerCode);

foreach (var se in SourceCode.AssemblerCode)

{

Console.WriteLine(se.ToString());

}

}

public void PrintTmo()

{

Helpers.WriteInConsole(ConsoleMessages.Print\_MacrosTable);

foreach (var e in MacrosStorage.Entities)

{

Console.WriteLine($"{Separator.Tab}{e.Name}:");

for (int i = 0; i < e.Body.Count; i++)

{

Console.WriteLine(e.Body[i].ToString());

}

}

}

public void PrintVariablesTable()

{

Helpers.WriteInConsole(ConsoleMessages.Print\_VariablesTable);

foreach (Variable e in VariablesStorage.Entities)

{

Console.WriteLine($"{e.Name} = {e.Value?.ToString() ?? string.Empty}");

}

}

public void PrintMacroNameTable()

{

Helpers.WriteInConsole("Таблица имен макросов");

foreach (var e in MacrosStorage.Entities)

{

var startIndex = SourceCode.SourceCodeLines.IndexOf(SourceCode.SourceCodeLines.FirstOrDefault(x => x.SourceString.ToUpper().Contains($"{e.Name} MACRO".ToUpper()))) + 1;

Console.WriteLine($"{e.Name}. Начало: {startIndex}. Длина: {e.Body?.Count ?? 0}");

}

}

}

}

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using SystemSoftware.Common;

using SystemSoftware.MacroProcessor;

using SystemSoftware.Resources;

namespace SystemSoftware.Interface

{

public class ConsoleProgram : AbstractApp

{

public bool IsFirstRunEnded { get; set; }

[Obsolete("Только 1 проход", true)]

public bool IsSecondRunEnded { get; set; }

public bool IsProcessingBySteps { get; set; }

public bool IsProcessingEnded { get; set; }

public ConsoleProgram(string[] args)

{

switch (args.Length)

{

case 1:

if (args[0].ToUpper() == "-HELP")

{

Console.WriteLine(GetUserGuide());

}

else

{

throw new CustomException(ConsoleMessages.Error\_WrongCommandLineArguments);

}

break;

case 2:

if (args[0].ToUpper() == "-INPUT\_FILE")

{

InputFile = args[1];

}

else if (args[0].ToUpper() == "-OUTPUT\_FILE")

{

OutputFile = args[1];

}

else

{

throw new CustomException(ConsoleMessages.Error\_WrongCommandLineArguments);

}

break;

case 4:

if (args[0].ToUpper() == "-INPUT\_FILE")

{

InputFile = args[1];

if (args[2].ToUpper() == "-OUTPUT\_FILE")

{

OutputFile = args[3];

}

else

{

throw new CustomException(ConsoleMessages.Error\_OutputFileArgument);

}

}

else if (args[0].ToUpper() == "-OUTPUT\_FILE")

{

OutputFile = args[1];

if (args[2].ToUpper() == "-INPUT\_FILE")

{

InputFile = args[3];

}

else

{

throw new CustomException(ConsoleMessages.Error\_InputFileArgument);

}

}

else

{

throw new CustomException(ConsoleMessages.Error\_WrongCommandLineArguments);

}

break;

default:

throw new CustomException(ConsoleMessages.Error\_WrongArgumentsCount);

}

Refresh();

}

public void Refresh()

{

try

{

var temp = File.ReadAllLines(InputFile);

SourceCode = new Processor(temp);

SourceStrings = new List<string>(temp);

IsFirstRunEnded = false;

IsProcessingBySteps = false;

IsProcessingEnded = false;

SourceCodeIndex = 0;

}

catch (Exception)

{

throw new CustomException(ConsoleMessages.Error\_InputFileNotFound);

}

}

public void NextStep()

{

try

{

if (!IsProcessingEnded)

{

SourceCode.FirstRunStep(SourceCode.SourceCodeLines[SourceCodeIndex++], MacrosStorage.Root);

}

}

catch (ArgumentOutOfRangeException)

{

IsProcessingEnded = true;

}

catch (CustomException ex)

{

IsProcessingEnded = true;

Helpers.WriteInConsole($@"{ConsoleMessages.Prefix\_ErrorInstring} ""{SourceCode.SourceCodeLines[SourceCodeIndex - 1].ToString()}"": {ex.Message}");

}

if (SourceCodeIndex == SourceCode.SourceCodeLines.Count)

{

Helpers.WriteInConsole(ConsoleMessages.ApplicationRunEnded);

IsProcessingEnded = true;

return;

}

}

public void FirstRun()

{

if (IsFirstRunEnded == true)

{

Helpers.WriteInConsole(ConsoleMessages.FirstRunEnded);

return;

}

SourceCodeIndex = 0;

try

{

while (true)

{

NextStep();

if (SourceCodeIndex == SourceCode.SourceCodeLines.Count)

{

IsFirstRunEnded = true;

IsProcessingEnded = true;

return;

}

if (IsProcessingEnded) return;

}

}

catch (CustomException ex)

{

IsProcessingEnded = true;

Helpers.WriteInConsole($"{ConsoleMessages.ErrorPrefix} :{ex.Message}");

}

}

public static string GetUserGuide()

{

var n = Environment.NewLine;

var t = Separator.Tab;

return

$"{ConsoleMessages.Help\_Title}{n}" +

$"{t}{ConsoleMessages.Help\_AllowedKeys}: [-input\_file] [-output\_file] [-help]{n}" +

$"-input\_file{t}{ConsoleMessages.Help\_InputFileKey}{n}" +

$"-output\_file{t}{ConsoleMessages.Help\_OutputFileKey}{n}" +

$"-help{t}{t}{ConsoleMessages.Help\_HelpKey}.{n}" +

$"{ConsoleMessages.Help\_Footer}.{n}";

}

public static string GetProgramGuide()

{

var n = Environment.NewLine;

var s = Separator.Bracket;

var t = Separator.Space + Separator.Space;

return

$"1{s}{t}{ConsoleMessages.Menu\_NextStep}{n}" +

$"2{s}{t}{ConsoleMessages.Menu\_FirstRun}{n}" +

$"3{s}{t}{ConsoleMessages.Menu\_PrintSourceCode}{n}" +

$"4{s}{t}{ConsoleMessages.Menu\_PrintAssemblerCode}{n}" +

$"5{s}{t}{ConsoleMessages.Menu\_PrintVariablesTable}{n}" +

$"6{s}{t}{ConsoleMessages.Menu\_PrintMacrosTable}{n}" +

$"7{s}{t}{ConsoleMessages.Menu\_SaveAssemblerCodeIntoFile}{n}" +

$"8{s}{t}{ConsoleMessages.Menu\_Refresh}{n}" +

$"9{s}{t}Распечатать таблицу имен макросов{n}" +

$"0{s}{t}{ConsoleMessages.Menu\_Exit}{n}";

}

public void PrintAssemblerCode()

{

Helpers.WriteInConsole(ConsoleMessages.Print\_AssemblerCode);

foreach (var se in SourceCode.AssemblerCode)

{

Console.WriteLine(se.ToString());

}

}

public void PrintTmo()

{

Helpers.WriteInConsole(ConsoleMessages.Print\_MacrosTable);

foreach (var e in MacrosStorage.Entities)

{

Console.WriteLine($"{Separator.Tab}{e.Name}:");

for (int i = 0; i < e.Body.Count; i++)

{

Console.WriteLine(e.Body[i].ToString());

}

}

}

public void PrintVariablesTable()

{

Helpers.WriteInConsole(ConsoleMessages.Print\_VariablesTable);

foreach (Variable e in VariablesStorage.Entities)

{

Console.WriteLine($"{e.Name} = {e.Value?.ToString() ?? string.Empty}");

}

}

public void PrintMacroNameTable()

{

Helpers.WriteInConsole("Таблица имен макросов");

foreach (var e in MacrosStorage.Entities)

{

var startIndex = SourceCode.SourceCodeLines.IndexOf(SourceCode.SourceCodeLines.FirstOrDefault(x => x.SourceString.ToUpper().Contains($"{e.Name} MACRO".ToUpper()))) + 1;

Console.WriteLine($"{e.Name}. Начало: {startIndex}. Длина: {e.Body?.Count ?? 0}");

}

}

}

}

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Windows.Forms;

using SystemSoftware.Common;

using SystemSoftware.MacroProcessor;

namespace SystemSoftware.Interface

{

public partial class MainForm : Form

{

private VisualApp \_program;

private bool prepareFlag = false;

public MainForm()

{

InitializeComponent();

Show();

Activate();

}

private void OnNextStep(object sender, EventArgs e)

{

Action HandleError = delegate ()

{

SetButtonsDisabled();

btnRefreshAll.Enabled = true;

btnFirstRun.Enabled = false;

\_program.PrintTmo(dgvTmo);

\_program.PrintVariablesTable(dgvVariables);

\_program.PrintMacroNameTable(dgvMacroNames);

};

if (!prepareFlag)

{

var sourceCodeText = tbSourceCode.Text.Split('\n');

\_program = new VisualApp(sourceCodeText);

prepareFlag = true;

}

try

{

btnRefreshAll.Enabled = true;

if (\_program.SourceCode.SourceCodeLines.Count == 0)

{

throw new CustomException("Исходный текст должен содержать хотя бы одну строку");

}

if (\_program.SourceCodeIndex == 0)

{

tbError.Clear();

}

var isLastStep = \_program.SourceCodeIndex + 1 == \_program.SourceCode.SourceCodeLines.Count

&& \_program.RunMode == RunMode.FirstRun;

if (isLastStep)

{

btnNextStep.Enabled = false;

CheckSourceEntity.CheckEnd(new CodeEntity(), 0);

btnFirstRun.Enabled = false;

}

\_program.NextFirstStep(tbAssemblerCode);

\_program.PrintAssemblerCode(tbAssemblerCode);

\_program.PrintTmo(dgvTmo);

\_program.PrintVariablesTable(dgvVariables);

\_program.PrintMacroNameTable(dgvMacroNames);

}

catch (CustomException ex)

{

tbError.Text = ex.Message;

HandleError();

}

catch (Exception)

{

tbError.Text = "Произошла ошибка при попытке совершить следующий шаг программы";

HandleError();

}

}

private void OnFirstRun(object sender, EventArgs e)

{

btnFirstRun.Enabled = false;

btnNextStep.Enabled = false;

btnRefreshAll.Enabled = true;

if(!prepareFlag)

{

var sourceCodeText = tbSourceCode.Text.Split('\n');

\_program = new VisualApp(sourceCodeText);

prepareFlag = true;

}

while (true)

{

var isLastStep = \_program.SourceCodeIndex == \_program.SourceCode.SourceCodeLines.Count

&& \_program.RunMode == RunMode.FirstRun;

if (!string.IsNullOrEmpty(tbError.Text) || isLastStep)

{

break;

}

OnNextStep(sender, e);

}

}

private void OnFormLoad(object sender, EventArgs e)

{

tbInputFile.Text = \_program?.InputFile ?? VisualApp.InitialInputFile;

tbOutputFile.Text = \_program?.OutputFile ?? VisualApp.InitialOutputFile;

SetButtonsDisabled();

if (!string.IsNullOrEmpty(tbInputFile.Text))

{

try

{

FillSourceTextBoxFromFile(tbInputFile.Text, tbSourceCode);

SetButtonsDisabled(false);

tbError.Text = string.Empty;

}

catch (Exception ex)

{

tbError.Text = ex.Message;

SetButtonsDisabled();

}

}

}

private void OnRefreshAll(object sender, EventArgs e)

{

prepareFlag = false;

tbError.Clear();

tbAssemblerCode.Clear();

\_program.PrintTmo(dgvTmo);

\_program.PrintVariablesTable(dgvVariables);

\_program.PrintMacroNameTable(dgvMacroNames);

VariablesStorage.Refresh();

MacrosStorage.Refresh();

MacrosStorage.macros = new List<string>();

Processor.assemblyMarks = new List<StructMarks>();

SetButtonsDisabled(false);

}

private void OnLoadFromFile(object sender, EventArgs e)

{

try

{

OpenFileDialog ofd = new OpenFileDialog();

ofd.Filter = "Text Files (.txt)|\*.txt";

ofd.InitialDirectory = Helpers.CurrentDirectory;

if (ofd.ShowDialog() == DialogResult.OK)

{

tbSourceCode.Clear();

tbInputFile.Text = ofd.FileName;

FillSourceTextBoxFromFile(ofd.FileName, tbSourceCode);

}

SetButtonsDisabled(false);

tbError.Text = string.Empty;

}

catch (Exception ex)

{

tbError.Text = ex.Message;

SetButtonsDisabled();

}

}

private void OnSaveIntoFile(object sender, EventArgs e)

{

SaveFileDialog sfd = new SaveFileDialog();

sfd.Filter = "Text Files (.txt)|\*.txt";

sfd.InitialDirectory = Helpers.CurrentDirectory;

if (sfd.ShowDialog() == DialogResult.OK)

{

tbInputFile.Text = sfd.FileName;

List<string> temp = tbAssemblerCode.Text.Split('\n').ToList();

StreamWriter sw = new StreamWriter(sfd.FileName);

foreach (string str in temp)

{

sw.WriteLine(str);

}

sw.Close();

}

}

public void FillSourceTextBoxFromFile(string file, TextBox tb)

{

try

{

var temp = string.Empty;

StreamReader sr = new StreamReader(file);

while ((temp = sr.ReadLine()) != null)

{

tb.AppendText(temp + Environment.NewLine);

}

sr.Close();

}

catch

{

throw new CustomException("Не удалось загрузить данные с файла. Возможно путь к файлу указан неверно");

}

}

private void SetButtonsDisabled(bool disabled = true)

{

btnNextStep.Enabled = !disabled;

btnRefreshAll.Enabled = !disabled;

btnFirstRun.Enabled = !disabled;

}

}

}

namespace SystemSoftware.Common

{

public enum ConditionalDirective

{

IF,

ELSE,

ENDIF,

AGO,

WHILE,

ENDW,

Empty

}

}

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using SystemSoftware.MacroProcessor;

using SystemSoftware.Resources;

namespace SystemSoftware.Common

{

public static class Helpers

{

public static readonly string[] Symbols = {"%", "#", "$", "", "!", "@", "^", "&", "\*", "-", "[", "\"", "\*", "(", ")", "\\",

"/", "?", "№", ";", ":", "+", "=", "[", "]", "{", "}", "|", "<", ">", "`", "~", ".", ",", "'", " " };

public static readonly string[] AssemblerDirectives = { "BYTE", "RESB", "RESW", "WORD" };

public static readonly string[] MacroGenerationDirectives = { "START", "END", "MACRO", "MEND" };

public static readonly string[] ConditionDirectives = { Directives.While, Directives.Endw, Directives.If, Directives.Else, Directives.EndIf };

public static readonly string[] VariableDirectives = { Directives.Variable, Directives.Set, Directives.Increment };

public static readonly string[] Keywords = { "ADD", "SAVER1", "SAVER2", "LOADR1", "JMP" };

public static readonly string[] AllKeywords = MacroGenerationDirectives.Concat(ConditionDirectives).Concat(VariableDirectives).Concat(Keywords).ToArray();

public static readonly string RussianLetters = "АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ";

public static readonly string[] ComparisonSigns = { "==", ">=", "<=", "!=", ">", "<" };

public static readonly string CurrentDirectory = new DirectoryInfo(Environment.CurrentDirectory).Parent.Parent.Parent.FullName;

public static bool IsAssemblerDirective(string operation)

{

return operation != null && operation.In(AssemblerDirectives);

}

public static bool IsLabel(string label)

{

if (!IsOperation(label))

{

return false;

}

if (AllKeywords.Contains(label))

{

return false;

}

if (!IsNotRussian(label))

{

return false;

}

if (MacrosStorage.IsInTMO(label))

{

return false;

}

if (VariablesStorage.IsInVariablesStorage(label))

{

return false;

}

return true;

}

public static bool IsOperation(string operation)

{

if (string.IsNullOrEmpty(operation)) return false;

if (char.IsDigit(operation[0])) return false;

if (operation.Any(x => x.ToString().In(Symbols))) return false;

if (IsRegister(operation)) return false;

if (IsAssemblerDirective(operation)) return false;

return true;

}

public static bool IsKeyWord(string operation)

{

return operation != null && operation.In(Keywords);

}

public static bool IsRegister(string operation)

{

for (int i = 0; i < 16; i++)

{

if ("R" + i.ToString() == operation.Trim().ToUpper())

return true;

}

return false;

}

public static bool IsNotRussian(string word)

{

return !word.Any(x => x.ToString().In(RussianLetters));

}

public static void ParseCondition(string str, out int first, out int second, out string sign)

{

string[] arr;

first = 0;

second = 0;

sign = "";

int temp;

foreach (string sgn in ComparisonSigns)

{

if ((arr = str.Split(new string[] { sgn }, StringSplitOptions.None)).Length > 1)

{

if (VariablesStorage.IsInVariablesStorage(arr[0]))

{

if (VariablesStorage.Find(arr[0]).Value == null)

{

throw new CustomException($"{ProcessorErrorMessages.EmptyVariableInComparison} ({arr[0]})");

}

else

{

first = (int)VariablesStorage.Find(arr[0]).Value;

}

}

else if (int.TryParse(arr[0], out temp) == false)

{

throw new CustomException($"{ProcessorErrorMessages.UndefinedComparisonPart} ({arr[0]})");

}

else

{

first = int.Parse(arr[0]);

}

if (VariablesStorage.IsInVariablesStorage(arr[1]))

{

if (VariablesStorage.Find(arr[1]).Value == null)

{

throw new CustomException($"{ProcessorErrorMessages.EmptyVariableInComparison} ({arr[1]})");

}

else

{

second = (int)VariablesStorage.Find(arr[1]).Value;

}

}

else if (int.TryParse(arr[1], out temp) == false)

{

throw new CustomException($"{ProcessorErrorMessages.UndefinedComparisonPart} ({arr[1]})");

}

else

{

second = int.Parse(arr[1]);

}

sign = sgn;

return;

}

}

throw new CustomException(ProcessorErrorMessages.UndefinedComparisonSign);

}

public static void PushConditionArgs(string str, Macro te)

{

int first, second;

string sign;

Helpers.ParseCondition(str, out first, out second, out sign);

string[] arr;

List<string> list = new List<string>();

if ((arr = str.Split(new string[] { sign }, StringSplitOptions.None)).Length == 2)

{

if (VariablesStorage.IsInVariablesStorage(arr[0]))

{

list.Add(arr[0]);

}

if (VariablesStorage.IsInVariablesStorage(arr[1]))

{

list.Add(arr[1]);

}

}

Dictionary<List<string>, Macro> dict = new Dictionary<List<string>, Macro>();

dict.Add(list, te);

VariablesStorage.WhileVar.Push(dict);

}

public static bool Compare(string str)

{

int first;

int second;

string sign;

ParseCondition(str, out first, out second, out sign);

switch (sign)

{

case ">":

return first > second;

case "<":

return first < second;

case ">=":

return first >= second;

case "<=":

return first <= second;

case "==":

return first == second;

case "!=":

return first != second;

default:

return false;

}

}

public static void CheckNames(string name)

{

List<string> list = new List<string>();

foreach (Variable glob in VariablesStorage.Entities)

{

list.Add(glob.Name);

}

foreach (Macro te in MacrosStorage.Entities)

{

list.Add(te.Name);

}

foreach (var p in MacrosStorage.Entities.SelectMany(e => e.Parameters))

{

list.Add(p.Name);

}

if (list.Contains(name))

{

throw new CustomException($"{ProcessorErrorMessages.NameIsAleradyUsed} ({name})");

}

}

public static CodeEntity Print(CodeEntity str)

{

CodeEntity newStr = str.Clone() as CodeEntity;

for (int j = 0; j < newStr.Operands.Count; j++)

{

if (VariablesStorage.IsInVariablesStorage(newStr.Operands[j]))

{

if (VariablesStorage.Find(newStr.Operands[j]).Value.HasValue)

newStr.Operands[j] = VariablesStorage.Find(newStr.Operands[j]).Value.Value.ToString();

else

throw new CustomException($"{ProcessorErrorMessages.NullVariable} ({newStr.Operands[j]})");

}

}

return newStr;

}

public static void WriteInConsole(string message)

{

Console.WriteLine($"{Environment.NewLine}{message}{Environment.NewLine}");

}

}

public class CustomException : Exception

{

public CustomException(string message)

: base(message)

{

}

}

}

using System.Collections.Generic;

using System.Linq;

using SystemSoftware.MacroProcessor;

namespace SystemSoftware.Common

{

public class PseudoMacroTableSingletone

{

private static List<CodeEntity> instance;

private PseudoMacroTableSingletone() { }

public static List<CodeEntity> Instance

{

get

{

if (instance == null)

{

instance = new List<CodeEntity>();

}

return instance;

}

}

public static void AddSourceEntity(CodeEntity se)

{

Instance.Add(se);

}

public static void RemoveSourceEntity(CodeEntity se)

{

Instance.Remove(se);

}

public static void Clear()

{

instance = null;

}

public static void CheckPseudoMacro()

{

foreach (CodeEntity se in Instance)

{

if (!MacrosStorage.IsInTMO(se.Operation) &&

!VariablesStorage.IsInVariablesStorage(se.Operation) &&

!Helpers.IsAssemblerDirective(se.Operation) &&

!Helpers.IsKeyWord(se.Operation) &&

!Helpers.MacroGenerationDirectives.Contains(se.Operation) &&

!(se.Operands.Count > 0 && se.Operands[0] == "START"))

{

throw new CustomException("Операция \"" + se.Operation + "\" не является ни оператором языка Ассемблера, ни оператором Макроязыка, ни макросом.");

}

}

}

}

}

namespace SystemSoftware.Common

{

public enum RunMode

{

FirstRun = 0,

SecondRun = 1

}

}

namespace SystemSoftware.Common

{

public class Separator

{

public const string Comma = ",";

public const string Tab = "\t";

public const string Space = " ";

public const string Bracket = ")";

}

}

using System.Collections.Generic;

using System.Linq;

namespace SystemSoftware.MacroProcessor

{

public static class VariablesStorage

{

public static List<Variable> Entities { get; set; } = new List<Variable>();

public static Stack<Dictionary<List<string>, Macro>> WhileVar { get; set; } = new Stack<Dictionary<List<string>, Macro>>();

public static void Refresh()

{

Entities = new List<Variable>();

WhileVar = new Stack<Dictionary<List<string>, Macro>>();

}

public static Variable Find(string name)

{

return Entities.SingleOrDefault(x => x.Name == name);

}

public static bool IsInVariablesStorage(string name)

{

return Find(name) != null;

}

}

}

namespace SystemSoftware.MacroProcessor

{

public class Variable

{

public string Name { get; set; }

public int? Value { get; set; }

public Variable(string name, int? value)

{

Name = name;

Value = value;

}

}

}

namespace SystemSoftware.MacroProcessor

{

public class StructMarks

{

public string Name { get; set; }

public bool Used { get; set; }

public string Macro { get; set; }

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using SystemSoftware.Common;

using SystemSoftware.Resources;

namespace SystemSoftware.MacroProcessor

{

public class Processor

{

public static string ConditionDirectives = "IF,ELSE,ENDIF,AIF,AGO";

private int \_macroCount { get; set; }

public string \_currentMacroName { get; set; }

private List<CodeEntity> PseudoMacroCalls { get; set; } = new List<CodeEntity>();

public List<CodeEntity> SourceCodeLines { get; set; }

public List<CodeEntity> AssemblerCode { get; set; } = new List<CodeEntity>();

public readonly MacroParametersParser MacroParametersParser;

public static string cm;

public static List<StructMarks> assemblyMarks = new List<StructMarks>();

public Processor(IEnumerable<string> strs)

{

SourceCodeLines = ParseSourceCode(strs);

foreach (CodeEntity se in SourceCodeLines)

{

se.Sources = this;

}

PseudoMacroTableSingletone.Clear();

MacroParametersParser = new MixedMacroParametersParser(this);

}

public Processor(List<CodeEntity> strs)

{

SourceCodeLines = strs;

foreach (CodeEntity se in SourceCodeLines)

{

se.Sources = this;

}

MacroParametersParser = new MixedMacroParametersParser(this);

}

public void MacroSubstitution(CodeEntity se, Macro te)

{

List<CodeEntity> localMbMacroCall = new List<CodeEntity>();

foreach (CodeEntity mc in PseudoMacroCalls)

{

if (mc.Operation == \_currentMacroName)

{

Macro currentTe = MacrosStorage.SearchInTMO(\_currentMacroName);

currentTe.PreviousMacros = te;

CheckSourceEntity.CheckMacroSubstitution(mc, currentTe);

CheckBody.CheckMacroRun(mc, te, currentTe);

var processedMacroBody = ProcessMacroParams(currentTe, mc.Operands);

var calledProcessor = currentTe.CallMacro(processedMacroBody);

List<CodeEntity> macroSubs = new List<CodeEntity>();

foreach (CodeEntity str in calledProcessor.AssemblerCode)

{

CodeEntity macroSubsEntity = Helpers.Print(str);

macroSubs.Add(macroSubsEntity);

if (!Helpers.IsAssemblerDirective(macroSubsEntity.Operation) && !Helpers.IsKeyWord(macroSubsEntity.Operation))

{

localMbMacroCall.Add(macroSubsEntity);

macroSubsEntity.IsRemove = true;

List<CodeEntity> list = PseudoMacroCalls.FindAll(x => x.Operation == macroSubsEntity.Operation);

if (list.Count > 0)

{

int n = list.Max(x => x.CallNumber);

macroSubsEntity.CallNumber = n++;

}

}

}

for (int i = 0; i < AssemblerCode.Count; i++)

{

if (AssemblerCode[i].Operation == mc.Operation && AssemblerCode[i].IsRemove == true && mc.CallNumber == AssemblerCode[i].CallNumber)

{

AssemblerCode.Remove(AssemblerCode[i]);

AssemblerCode.InsertRange(i, macroSubs);

i += macroSubs.Count - 1;

}

}

}

}

foreach (CodeEntity str in localMbMacroCall)

{

PseudoMacroCalls.Add(str);

PseudoMacroTableSingletone.AddSourceEntity(se);

}

foreach (CodeEntity sourceEntity in PseudoMacroCalls)

{

Macro currentTe = MacrosStorage.SearchInTMO(sourceEntity.Operation);

if (MacrosStorage.IsInTMO(sourceEntity.Operation))

{

CheckBody.CheckMacroRun(sourceEntity, te, currentTe);

}

}

}

public void FirstRunStep(CodeEntity se, Macro te)

{

string operation = se.Operation;

string label = se.Label;

List<string> operands = se.Operands;

if (label != null && operation == null)

throw new CustomException("Метка не может быть описана без директивы ассемблера");

CheckSourceEntity.CheckLabel(se);

if (operation == "END")

{

CheckSourceEntity.CheckEnd(se, \_macroCount);

PseudoMacroTableSingletone.CheckPseudoMacro();

AssemblerCode.Add(Helpers.Print(se));

}

else if (operation == Directives.Variable && \_macroCount == 0)

{

CheckSourceEntity.CheckVariable(se);

if (operands.Count == 1)

VariablesStorage.Entities.Add(new Variable(operands[0], null));

else

VariablesStorage.Entities.Add(new Variable(operands[0], int.Parse(operands[1])));

}

else if (operation == "SET" && \_macroCount == 0)

{

CheckSourceEntity.CheckSet(se, te);

VariablesStorage.Find(se.Operands[0]).Value = int.Parse(se.Operands[1]);

}

else if (operation == "INC" && \_macroCount == 0)

{

CheckSourceEntity.CheckInc(se, te);

VariablesStorage.Find(operands[0]).Value++;

}

else if (operation == "MACRO")

{

if (\_macroCount == 0)

{

CheckSourceEntity.CheckMacro(se, \_macroCount);

Macro currentTe = new Macro()

{

Name = label,

Parameters = MacroParametersParser.Parse(operands, label)

};

currentTe.ParentMacros = te;

te.ChildrenMacros.Add(currentTe);

MacrosStorage.Entities.Add(currentTe);

CheckMacros.CheckLocalTmo();

\_currentMacroName = label;

}

else if (\_macroCount > 0)

{

MacrosStorage.SearchInTMO(\_currentMacroName).Body.Add(se);

}

else

{

throw new CustomException(ProcessorErrorMessages.IncorrectMacroMendCount);

}

\_macroCount++;

}

else if (operation == "MEND")

{

if (\_macroCount > 1)

{

MacrosStorage.SearchInTMO(\_currentMacroName).Body.Add(se);

}

CheckSourceEntity.CheckMend(se, \_macroCount);

\_macroCount--;

if (\_macroCount == 0)

{

MacroSubstitution(se, te);

}

}

else

{

if (\_macroCount > 0)

{

MacrosStorage.SearchInTMO(\_currentMacroName).Body.Add(se);

if(se.ToString().Split(' ')[0].Contains(':') || se.ToString().Split(' ')[0].Contains('%'))

MacrosStorage.macros.Add(se.ToString().Split(' ')[1]);

else

MacrosStorage.macros.Add(se.ToString().Split(' ')[0]);

for (int i = 0; i < MacrosStorage.Entities.Count; i++)

{

for (int j = 0; j < MacrosStorage.macros.Count; j++)

{

if (MacrosStorage.Entities[i].Name == MacrosStorage.macros[j])

throw new CustomException("Макровызовы запрещены в макросах");

}

}

}

else

{

if (te == MacrosStorage.Root && operation.In("IF", "ELSE", "ENDIF", "WHILE", "ENDW"))

{

throw new CustomException($"{ProcessorErrorMessages.UseOnlyInsideMacro\_1} {operation} {ProcessorErrorMessages.UseOnlyInsideMacro\_2} ({se.SourceString})");

}

if (MacrosStorage.IsInTMO(operation))

{

Macro currentTe = MacrosStorage.SearchInTMO(operation);

currentTe.PreviousMacros = te;

CheckSourceEntity.CheckMacroSubstitution(se, currentTe);

CheckBody.CheckMacroRun(se, te, currentTe);

var processedMacroBody = ProcessMacroParams(currentTe, operands);

var calledProcessor = currentTe.CallMacro(processedMacroBody);

foreach (CodeEntity sourceEntity in PseudoMacroCalls)

{

Macro teCur = MacrosStorage.SearchInTMO(sourceEntity.Operation);

if (MacrosStorage.IsInTMO(sourceEntity.Operation))

{

CheckBody.CheckMacroRun(sourceEntity, te, teCur);

}

}

PseudoMacroCalls.AddRange(calledProcessor.PseudoMacroCalls);

foreach (CodeEntity str in calledProcessor.AssemblerCode)

{

AssemblerCode.Add(Helpers.Print(str));

}

}

else

{

se = Helpers.Print(se);

if (\_macroCount == 0 && !Helpers.IsAssemblerDirective(se.Operation) && !Helpers.IsKeyWord(se.Operation))

{

se.IsRemove = true;

List<CodeEntity> list = PseudoMacroCalls.FindAll(x => x.Operation == se.Operation);

if (list.Count > 0)

{

int n = list.Max(x => x.CallNumber);

se.CallNumber = n + 1;

}

PseudoMacroCalls.Add(se);

PseudoMacroTableSingletone.AddSourceEntity(se);

}

if (se.Label != null && se.Label.Contains('%'))

se.Label = "";

AssemblerCode.Add(se);

}

}

}

for (int i = 0; i < MacrosStorage.Entities.Count; i++)

{

for (int j = 0; j < MacrosStorage.macros.Count; j++)

{

if (MacrosStorage.Entities[i].Name == MacrosStorage.macros[j])

throw new CustomException("Макровызовы запрещены в макросах");

}

}

}

public static List<CodeEntity> ParseSourceCode(IEnumerable<string> strs)

{

List<CodeEntity> result = new List<CodeEntity>();

foreach (string s in strs)

{

// пропускаем пустую строку

if (String.IsNullOrEmpty(s.Trim()))

continue;

string currentString = s.ToUpper().Trim();

CodeEntity se = new CodeEntity() { SourceString = currentString };

//разборка метки

if ((currentString.Split(' ')[0].Trim().IndexOf(":") == currentString.Split(' ')[0].Trim().Length - 1) && (!currentString.Contains("BYTE") || currentString.IndexOf(':') < currentString.IndexOf("C'")) && !currentString.Split(' ')[0].Contains('%'))

{

se.Label = currentString.Split(':')[0].Trim();

currentString = currentString.Remove(0, currentString.Split(':')[0].Length + 1).Trim();

}

else

{

if (currentString.IndexOf("%") == 0)

{

se.Label = currentString.Split(' ')[0].Trim();

var g = new StructMarks()

{

Name = se.Label,

Used = false,

Macro = cm

};

assemblyMarks.Add(g);

currentString = currentString.Remove(0, currentString.Split(' ')[0].Length).Trim();

}

}

if (currentString.Split(null as char[], StringSplitOptions.RemoveEmptyEntries).Length > 0)

{

se.Operation = currentString.Split(null as char[], StringSplitOptions.RemoveEmptyEntries)[0].Trim();

currentString = currentString.Remove(0, currentString.Split(null as char[], StringSplitOptions.RemoveEmptyEntries)[0].Length).Trim();

}

if (se.Operation == "BYTE")

{

se.Operands.Add(currentString.Trim());

}

else

{

for (int i = 0; i < currentString.Split(null as char[], StringSplitOptions.RemoveEmptyEntries).Length; i++)

{

se.Operands.Add(currentString.Split(null as char[], StringSplitOptions.RemoveEmptyEntries)[i].Trim());

}

}

if (se.Operands.Count > 0 && se.Operands[0] == "MACRO")

{

se.Label = se.Operation;

se.Operation = se.Operands[0];

cm = se.Label;

for (int i = 1; i < se.Operands.Count; i++)

{

se.Operands[i - 1] = se.Operands[i];

}

se.Operands.RemoveAt(se.Operands.Count - 1);

}

result.Add(se);

if (se.Operation == "END")

{

return result;

}

}

return result;

}

private List<CodeEntity> ProcessMacroParams(Macro macro, IEnumerable<string> passedParams)

{

int firstKeyParamIndex = -1;

if (passedParams.Count() != macro.Parameters.Count)

{

throw new CustomException(string.Format(

ProcessorErrorMessages.IncorrectMacroCallParametersCount, macro.Name, passedParams.Count(), macro.Parameters.Count));

}

var first = passedParams.FirstOrDefault(x => x.Contains("="));

if (first != null)

{

firstKeyParamIndex = Array.IndexOf(passedParams.ToArray(), first);

if (passedParams.Any(x => !x.Contains("=") && Array.IndexOf(passedParams.ToArray(), x) > firstKeyParamIndex))

{

throw new CustomException(string.Format(ProcessorErrorMessages.IncorrectMacroCallParameters, macro.Name));

}

}

if (firstKeyParamIndex != macro.FirstKeyParameterIdx)

{

throw new CustomException(string.Format(ProcessorErrorMessages.IncorrectMacroCallParameters, macro.Name));

}

Dictionary<string, int> dict = new Dictionary<string, int>();

Action ParsePositionedParams = delegate ()

{

int localFirstParamIndex = firstKeyParamIndex >= 0 ? firstKeyParamIndex : passedParams.Count();

for (int i = 0; i < localFirstParamIndex; i++)

{

string currentParam = passedParams.ToArray()[i];

var variable = VariablesStorage.Entities.FirstOrDefault(e => e.Name.EqualsIgnoreCase(currentParam));

if (variable == null && !int.TryParse(currentParam, out int temp))

{

throw new CustomException(string.Format(ProcessorErrorMessages.IncorrectMacroParameterValue, currentParam));

}

if (variable != null && variable.Value == null)

{

throw new CustomException(string.Format(ProcessorErrorMessages.MacroParameterIsEmptyVariable, currentParam));

}

int curerntParamValue = variable?.Value ?? int.Parse(currentParam);

dict.Add(macro.Parameters[i].Name, curerntParamValue);

}

};

Action ParseKeyParams = delegate ()

{

if (firstKeyParamIndex < 0) return;

for (int i = firstKeyParamIndex; i < passedParams.Count(); i++)

{

string currentParameter = passedParams.ToArray()[i];

string[] vals = currentParameter.Split('=');

if (vals.Length != 2)

{

throw new CustomException(string.Format(ProcessorErrorMessages.IcorrectMacroCallKeyParameter, macro.Name, currentParameter));

}

var macroParameter = macro.Parameters.FirstOrDefault(e => e.Name == vals[0]);

if (macroParameter == null)

{

throw new CustomException(string.Format(ProcessorErrorMessages.ParameterDoesNotExists, vals[0]));

}

if (macroParameter.Type != MacroParameterTypes.Key)

{

throw new CustomException(string.Format(ProcessorErrorMessages.IncorrectMacroParameterType, vals[0]));

}

var passedValue = vals[1];

int value = 0;

if (string.IsNullOrEmpty(passedValue))

{

value = macroParameter.DefaultValue ??

throw new CustomException(string.Format(ProcessorErrorMessages.IncorrectMacroParameterValue, currentParameter));

}

else

{

var variable = VariablesStorage.Entities.FirstOrDefault(e => e.Name.EqualsIgnoreCase(vals[1]));

if (variable == null && !int.TryParse(vals[1], out int temp))

{

throw new CustomException(string.Format(ProcessorErrorMessages.IncorrectMacroParameterValue, currentParameter));

}

if (variable != null && variable.Value == null)

{

throw new CustomException(string.Format(ProcessorErrorMessages.MacroParameterIsEmptyVariable, currentParameter));

}

value = variable?.Value ?? int.Parse(vals[1]);

}

if (dict.Keys.Contains(vals[0]))

{

throw new CustomException(string.Format(ProcessorErrorMessages.DublicateMacroCallParameter, vals[0]));

}

dict.Add(vals[0], value);

}

};

ParsePositionedParams();

ParseKeyParams();

var processedBody = macro.Body.Select(e => (CodeEntity)e.Clone()).ToList();

var macroCount = 0;

foreach (var sourceLine in processedBody)

{

if (sourceLine.Operation == "MACRO") macroCount++;

if (sourceLine.Operation == "MEND") macroCount--;

if (macroCount != 0) continue;

if (sourceLine.Operation.EqualsIgnoreCase("WHILE"))

{

if (sourceLine.Operands.Count > 0)

{

foreach (var sign in Helpers.ComparisonSigns)

{

var t = sourceLine.Operands[0].Split(new string[] { sign }, StringSplitOptions.None);

if (t.Length == 2)

{

if (macro.Parameters.Any(e => e.Name == t[0].Trim()))

{

t[0] = dict[t[0].Trim()].ToString();

}

if (macro.Parameters.Any(e => e.Name == t[1].Trim()))

{

t[1] = dict[t[1].Trim()].ToString();

}

sourceLine.Operands[0] = t[0] + sign + t[1];

break;

}

}

}

else

{

throw new CustomException(string.Format(ProcessorErrorMessages.IncorrectDirectiveUsage, sourceLine.Operation));

}

}

else if (sourceLine.Operation.In("SET", "INC"))

{

continue;

}

else

{

for (int i = 0; i < sourceLine.Operands.Count; i++)

{

var currentOperand = sourceLine.Operands[i];

if (dict.Keys.Contains(currentOperand))

{

sourceLine.Operands[i] = dict[currentOperand].ToString();

}

if (currentOperand.Contains("="))

{

string[] t = currentOperand.Split(new string[] { "=" }, StringSplitOptions.None);

if (t.Length == 2)

{

if (macro.Parameters.Any(e => e.Name == t[1].Trim()))

{

t[1] = dict[t[1].Trim()].ToString();

}

sourceLine.Operands[i] = t[0] + "=" + t[1];

}

}

}

}

}

return processedBody;

}

}

public static class CheckSourceEntity

{

public static void CheckLabel(CodeEntity se)

{

if (se.SourceString.Split(':').Length > 2 && se.Operation != "BYTE")

{

throw new CustomException($"{ProcessorErrorMessages.ExtraColonInLine} ({se.SourceString})");

}

if (se.SourceString.Split(':').Length > 1 && string.IsNullOrEmpty(se.SourceString.Split(':')[0]))

{

throw new CustomException($"{ProcessorErrorMessages.ExtraColonInLine} ({se.SourceString})");

}

}

public static void CheckMacro(CodeEntity se, int macroCount)

{

if (se.SourceString.Contains(":"))

{

throw new CustomException($"{ProcessorErrorMessages.LabesInMacroDefinition} ({se.SourceString})");

}

if (string.IsNullOrEmpty(se.Label) || !Helpers.IsLabel(se.Label))

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectMacroName} ({se.SourceString})");

}

if (MacrosStorage.IsInTMO(se.Label))

{

throw new CustomException($"{ProcessorErrorMessages.MacrosAleradyExists} ({se.Label}): {se.SourceString}");

}

Helpers.CheckNames(se.Label);

}

public static void CheckMend(CodeEntity se, int macroCount)

{

if (se.Operands.Count != 0)

{

throw new CustomException($"{ProcessorErrorMessages.MendWithParameters} ({se.SourceString})");

}

if (!string.IsNullOrEmpty(se.Label))

{

throw new CustomException($"{ProcessorErrorMessages.MendWithLabel} ({se.SourceString})");

}

if (macroCount == 0)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectMacroMendCount} ({se.SourceString})");

}

}

public static void CheckEnd(CodeEntity se, int macroCount)

{

if (macroCount != 0)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectMacroMendCount} ({se.SourceString})");

}

}

public static void CheckMacroSubstitution(CodeEntity se, Macro macro)

{

if (se.Operands.Count != macro.Parameters.Count)

{

macro.Parameters

.Where(x => x.DefaultValue.HasValue && se.Operands.All(y => !y.Contains(x.Name.ToUpper())))

.ToList()

.ForEach(x => se.Operands.Add(x.Name.ToUpper() + "=" + x.DefaultValue.Value));

if (se.Operands.Count != macro.Parameters.Count)

{

throw new CustomException("Некорректное количество параметров. Введено: " + se.Operands.Count + ". Ожидается: " + macro.Parameters.Count);

}

}

if (!string.IsNullOrEmpty(se.Label))

{

throw new CustomException("При макровызове макроса не должно быть меток: " + se.SourceString);

}

}

public static void CheckVariable(CodeEntity se)

{

if (se.Operands.Count > 0 && VariablesStorage.IsInVariablesStorage(se.Operands[0]))

{

throw new CustomException($"{ProcessorErrorMessages.VariableAleradyExists} ({se.SourceString})");

}

if (!String.IsNullOrEmpty(se.Label))

{

throw new CustomException($"{ProcessorErrorMessages.VariableDefinitionWithLabel} ({se.SourceString})");

}

if (se.Operands.Count == 2)

{

if (!Helpers.IsLabel(se.Operands[0]))

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectVariableName} ({se.SourceString})");

}

int temp;

if (Int32.TryParse(se.Operands[1], out temp) == false)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectVariableValue} ({se.SourceString})");

}

}

else if (se.Operands.Count == 1)

{

if (!Helpers.IsLabel(se.Operands[0]))

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectVariableName} ({se.SourceString})");

}

}

else

{

throw new CustomException($"{ProcessorErrorMessages.VariableIncorrectOperandsCount} ({se.SourceString})");

}

Helpers.CheckNames(se.Operands[0]);

}

public static void CheckSet(CodeEntity se, Macro te)

{

if (!string.IsNullOrEmpty(se.Label))

{

throw new CustomException($"{ProcessorErrorMessages.SetWithLabel} ({se.SourceString})");

}

if (se.Operands.Count == 2)

{

if (!VariablesStorage.IsInVariablesStorage(se.Operands[0]))

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectVariableName} ({se.SourceString})");

}

int temp;

if (int.TryParse(se.Operands[1], out temp) == false)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectVariableValue} ({se.SourceString})");

}

foreach (Dictionary<List<string>, Macro> dict in VariablesStorage.WhileVar)

{

if (dict.Keys.First().Contains(se.Operands[0]) && dict.Values.First() != te)

{

throw new CustomException($"{ProcessorErrorMessages.VariableIsLoopCounter} (Переменная: {se.Operands[0]}): {se.SourceString}");

}

}

}

else

{

throw new CustomException($"{ProcessorErrorMessages.SetIncorrectOperands} ({se.SourceString})");

}

}

public static void CheckInc(CodeEntity se, Macro te)

{

if (!string.IsNullOrEmpty(se.Label))

{

throw new CustomException($"{ProcessorErrorMessages.IncWithLabel} ({se.SourceString})");

}

if (se.Operands.Count != 1)

{

throw new CustomException($"{ProcessorErrorMessages.IncIncorrectOperandsCount} ({se.SourceString})");

}

if (!VariablesStorage.IsInVariablesStorage(se.Operands[0]))

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectVariableName} ({se.SourceString})");

}

if (VariablesStorage.Find(se.Operands[0]).Value == null)

{

throw new CustomException($"{ProcessorErrorMessages.NullVariable} ({se.Operands[0]})");

}

foreach (Dictionary<List<string>, Macro> dict in VariablesStorage.WhileVar)

{

if (dict.Keys.First().Contains(se.Operands[0]) && dict.Values.First() != te)

{

throw new CustomException($"{ProcessorErrorMessages.VariableIsLoopCounter} (Переменная: {se.Operands[0]}) {se.SourceString}");

}

}

}

}

public static class CheckBody

{

public static void CheckMacroRun(CodeEntity se, Macro parent, Macro child)

{

Macro current = parent;

List<Macro> list = new List<Macro>();

while (current.PreviousMacros != null)

{

if (list.Contains(current))

{

throw new CustomException(ProcessorErrorMessages.Recursion);

}

list.Add(current);

current = current.PreviousMacros;

}

if (MacrosStorage.IsInTMO(child.Name) && parent.Name == child.Name)

{

throw new CustomException($"{ProcessorErrorMessages.SelfMacroCall} (Макрос: {child.Name})");

}

if (MacrosStorage.IsInTMO(child.Name) && !parent.LocalTmo.Contains(child))

{

throw new CustomException($"{ProcessorErrorMessages.MacroScopeLocal} (Макрос: {child.Name})");

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using SystemSoftware.Common;

using SystemSoftware.Resources;

namespace SystemSoftware.MacroProcessor

{

public abstract class MacroParametersParser

{

protected readonly Processor Processor;

public MacroParametersParser(Processor processor)

{

Processor = processor;

}

public abstract List<MacroParameter> Parse(List<string> operands, string macroName);

}

public class PositionMacroParametersParser : MacroParametersParser

{

public PositionMacroParametersParser(Processor processor)

: base(processor)

{

}

public override List<MacroParameter> Parse(List<string> operands, string macroName)

{

if (!(operands?.Any() ?? false))

{

return new List<MacroParameter>();

}

foreach (string currentOperand in operands)

{

Helpers.CheckNames(currentOperand);

if (operands.Count(x => x == currentOperand) > 1)

{

throw new CustomException(string.Format(ProcessorErrorMessages.MacroParameterDublicate, currentOperand));

}

if (!Helpers.IsLabel(currentOperand))

{

throw new CustomException(string.Format(ProcessorErrorMessages.IncorrectMacroParameterName, currentOperand));

}

}

return operands

.Select(e => new MacroParameter(e, MacroParameterTypes.Position))

.ToList();

}

}

public class KeyMacroParametersParser : MacroParametersParser

{

public KeyMacroParametersParser(Processor processor)

: base(processor)

{

}

public override List<MacroParameter> Parse(List<string> operands, string macroName)

{

if (!(operands?.Any() ?? false))

{

return new List<MacroParameter>();

}

var parameters = new List<MacroParameter>();

foreach (string currentOperand in operands)

{

Helpers.CheckNames(currentOperand);

var vals = currentOperand.Split('=');

if (vals.Length != 2)

{

throw new CustomException(string.Format(ProcessorErrorMessages.IncorrectMacroDefinitionParameter, currentOperand));

}

var parameterName = vals[0];

var defaultValue = vals[1];

if (parameters.Any(e => e.Name == parameterName))

{

throw new CustomException(string.Format(ProcessorErrorMessages.MacroParameterDublicate, currentOperand));

}

if (!Helpers.IsLabel(parameterName))

{

throw new CustomException(string.Format(ProcessorErrorMessages.IncorrectMacroDefinitionParameter, currentOperand));

}

if (!string.IsNullOrEmpty(defaultValue) && !int.TryParse(defaultValue, out int temp))

{

throw new CustomException(string.Format(ProcessorErrorMessages.IncorrectMacroDefinitionParameterDefault, currentOperand));

}

parameters.Add(new MacroParameter

{

Name = parameterName,

Type = MacroParameterTypes.Key,

DefaultValue = !string.IsNullOrEmpty(defaultValue)

? int.Parse(defaultValue)

: (int?)null

});

}

return parameters;

}

}

public class MixedMacroParametersParser : MacroParametersParser

{

public MixedMacroParametersParser(Processor processor)

: base(processor)

{

}

public override List<MacroParameter> Parse(List<string> operands, string macroName)

{

if (!(operands?.Any() ?? false))

{

return new List<MacroParameter>();

}

var parameters = new List<MacroParameter>();

var firstKeyParameterIdx = Array.IndexOf(operands.ToArray(), operands.FirstOrDefault(x => x.Contains("=")));

if (firstKeyParameterIdx != -1)

{

if (operands.Any(x => !x.Contains("=") && Array.IndexOf(operands.ToArray(), x) > firstKeyParameterIdx))

{

throw new CustomException(string.Format(ProcessorErrorMessages.IncorrectMacroDefinitionParameters, macroName));

}

}

var lastPositionParameterIdx = firstKeyParameterIdx >= 0 ? firstKeyParameterIdx : operands.Count();

for (int i = 0; i < lastPositionParameterIdx; i++)

{

var currentOperand = operands.ToArray()[i];

Helpers.CheckNames(currentOperand);

if (parameters.Any(e => e.Name == currentOperand))

{

throw new CustomException(string.Format(ProcessorErrorMessages.MacroParameterDublicate, currentOperand));

}

if (!Helpers.IsLabel(currentOperand))

{

throw new CustomException(string.Format(ProcessorErrorMessages.IncorrectMacroParameterName, currentOperand));

}

parameters.Add(new MacroParameter

{

Name = currentOperand,

Type = MacroParameterTypes.Position

});

}

if (firstKeyParameterIdx >= 0)

{

for (int i = firstKeyParameterIdx; i < operands.Count(); i++)

{

var currentOperand = operands.ToArray()[i];

Helpers.CheckNames(currentOperand);

var vals = currentOperand.Split('=');

if (vals.Length != 2)

{

throw new CustomException(string.Format(ProcessorErrorMessages.IncorrectMacroDefinitionParameter, currentOperand));

}

var parameterName = vals[0];

var defaultValue = vals[1];

if (parameters.Any(e => e.Name == parameterName))

{

throw new CustomException(string.Format(ProcessorErrorMessages.MacroParameterDublicate, currentOperand));

}

if (!Helpers.IsLabel(parameterName))

{

throw new CustomException(string.Format(ProcessorErrorMessages.IncorrectMacroDefinitionParameter, currentOperand));

}

if (!string.IsNullOrEmpty(defaultValue) && !int.TryParse(defaultValue, out int temp))

{

throw new CustomException(string.Format(ProcessorErrorMessages.IncorrectMacroDefinitionParameterDefault, currentOperand));

}

parameters.Add(new MacroParameter

{

Name = parameterName,

Type = MacroParameterTypes.Key,

DefaultValue = !string.IsNullOrEmpty(defaultValue)

? int.Parse(defaultValue)

: (int?)null

});

}

}

return parameters;

}

}

}

using System.Collections.Generic;

using System.Linq;

namespace SystemSoftware.MacroProcessor

{

public static class MacrosStorage

{

public static List<Macro> Entities { get; set; } = new List<Macro>();

public static Macro Root { get; } = new Macro() { ParentMacros = null, ChildrenMacros = new List<Macro>(), IsRootMacro = true };

public static List<string> macros { get; set; } = new List<string>();

public static void Refresh()

{

Entities = new List<Macro>();

}

public static Macro SearchInTMO(string name)

{

return Entities.SingleOrDefault(x => x.Name == name);

}

public static bool IsInTMO(string name)

{

return SearchInTMO(name) != null;

}

}

}

namespace SystemSoftware.MacroProcessor

{

public class MacroParameter

{

public string Name { get; set; }

public MacroParameterTypes Type { get; set; }

public int? DefaultValue { get; set; }

public MacroParameter() { }

public MacroParameter(string name, MacroParameterTypes type, int? defaultValue = null)

{

Name = name;

Type = type;

DefaultValue = defaultValue;

}

}

public enum MacroParameterTypes

{

Position,

Key

}

}

using System.Collections.Generic;

using System.Linq;

namespace SystemSoftware.MacroProcessor

{

public static class MacrosStorage

{

public static List<Macro> Entities { get; set; } = new List<Macro>();

public static Macro Root { get; } = new Macro() { ParentMacros = null, ChildrenMacros = new List<Macro>(), IsRootMacro = true };

public static List<string> macros { get; set; } = new List<string>();

public static void Refresh()

{

Entities = new List<Macro>();

}

public static Macro SearchInTMO(string name)

{

return Entities.SingleOrDefault(x => x.Name == name);

}

public static bool IsInTMO(string name)

{

return SearchInTMO(name) != null;

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using SystemSoftware.Common;

using SystemSoftware.Resources;

namespace SystemSoftware.MacroProcessor

{

public class Macro

{

public string Name { get; set; }

public List<CodeEntity> Body { get; set; } = new List<CodeEntity>();

public List<MacroParameter> Parameters { get; set; }

public int FirstKeyParameterIdx =>

Array.IndexOf(Parameters?.ToArray() ?? new MacroParameter[] { }, Parameters?.FirstOrDefault(e => e.Type == MacroParameterTypes.Key));

public Macro ParentMacros { get; set; }

public List<Macro> ChildrenMacros { get; set; } = new List<Macro>();

public List<Macro> LocalTmo { get; set; } = new List<Macro>();

public Macro PreviousMacros { get; set; }

public bool IsRootMacro { get; set; }

public const int INFINITE\_LOOP\_COUNT = 50000;

private int \_uniqueLabelCounter { get; set; }

private int \_counter { get; set; }

public Processor CallMacro(List<CodeEntity> source)

{

Processor macroSourceCode = new Processor(source);

CheckMacros.CheckMacroLabels(this);

CheckMacros.CheckLocalTmo();

List<string> localMacroLabels = new List<string>();

int macroCount = 0;

foreach (CodeEntity currentLine in macroSourceCode.SourceCodeLines)

{

if (currentLine.Operation == "MACRO") macroCount++;

if (currentLine.Operation == "MEND") macroCount--;

if (macroCount != 0) continue;

if (!string.IsNullOrEmpty(currentLine.Label))

{

var str = currentLine.Label;

if (currentLine.Label.IndexOf("%") == 0)

str = currentLine.Label.Remove(0, 1);

if (currentLine.Label.Contains("%") && currentLine.Label.Contains(":"))

throw new CustomException("Невозможно определить тип метки");

if (!Helpers.IsLabel(str))

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectLabelInMacro} (Метка {currentLine.Label}, макрос {currentLine.SourceString})");

throw new CustomException($"{ProcessorErrorMessages.DuplicateLabelInMacro} (Метка {currentLine.Label}, макрос {currentLine.SourceString})");

}

Helpers.CheckNames(currentLine.Label);

if (localMacroLabels.Contains(currentLine.Label))

{

throw new CustomException($"{ProcessorErrorMessages.DuplicateLabelInMacro} (Метка {currentLine.Label}, макрос {currentLine.SourceString})");

}

localMacroLabels.Add(currentLine.Label);

}

}

macroCount = 0;

for (int i = 0; i < macroSourceCode.SourceCodeLines.Count; i++)

{

CodeEntity currentLine = macroSourceCode.SourceCodeLines[i];

if (currentLine.Operation == "MACRO")

{

macroCount++;

}

else if (currentLine.Operation == "MEND")

{

macroCount--;

}

if (macroCount == 0)

{

if (currentLine.Label != null && currentLine.Label.Contains("%"))

continue;

if (!string.IsNullOrEmpty(currentLine.Label))

{

currentLine.Label = currentLine.Label + "\_" + Name + "\_" + \_uniqueLabelCounter;

}

for (int j = 0; j < currentLine.Operands.Count; j++)

{

if (currentLine.Operands[j].Contains("%"))

continue;

if (localMacroLabels.Contains(currentLine.Operands[j]) )

{

currentLine.Operands[j] = currentLine.Operands[j] + "\_" + Name + "\_" + \_uniqueLabelCounter;

}

}

}

}

\_uniqueLabelCounter++;

Stack<bool> runStack = new Stack<bool>();

runStack.Push(true);

Stack<ConditionalDirective> commandStack = new Stack<ConditionalDirective>();

Stack<int> whileStack = new Stack<int>();

macroCount = 0;

for (int i = 0; i < macroSourceCode.SourceCodeLines.Count; i++)

{

CodeEntity current = macroSourceCode.SourceCodeLines[i].Clone() as CodeEntity;

\_counter++;

if (\_counter == INFINITE\_LOOP\_COUNT)

{

throw new CustomException(ProcessorErrorMessages.InfiniteLoop);

}

if (current.Operation == "MACRO")

{

macroCount++;

}

else if (current.Operation == "MEND")

{

macroCount--;

}

if (macroCount == 0)

{

CheckMacros.CheckInner(current, commandStack);

if (current.Operation == "IF")

{

CheckMacros.CheckIf(this);

commandStack.Push(ConditionalDirective.IF);

runStack.Push(runStack.Peek() && Helpers.Compare(current.Operands[0]));

continue;

}

if (current.Operation == "ELSE")

{

CheckMacros.CheckIf(this);

commandStack.Pop();

commandStack.Push(ConditionalDirective.ELSE);

bool elseFlag = runStack.Pop();

runStack.Push(runStack.Peek() && !elseFlag);

continue;

}

if (current.Operation == "ENDIF")

{

CheckMacros.CheckIf(this);

commandStack.Pop();

runStack.Pop();

continue;

}

if (current.Operation == "WHILE")

{

CheckMacros.CheckWhile(this);

if (current.Operands.Count == 1)

Helpers.PushConditionArgs(current.Operands[0], this);

commandStack.Push(ConditionalDirective.ENDIF);

runStack.Push(runStack.Peek() && Helpers.Compare(current.Operands[0]));

whileStack.Push(i);

continue;

}

if (current.Operation == "ENDW")

{

CheckMacros.CheckWhile(this);

commandStack.Pop();

int newI = whileStack.Pop() - 1;

if (runStack.Pop())

{

i = newI;

}

continue;

}

if (current.Operation.In("AIF", "AGO"))

{

if (current.Operation == "AIF" && !Helpers.Compare(current.Operands[0])) continue;

if (runStack.Peek())

{

CheckMacros.CheckAif(this);

if(!(current.Operation == "AIF" && current.Operands[1].IndexOf("%") == 0 ) && !(current.Operation == "AGO" && current.Operands[0].IndexOf('%') == 0))

throw new CustomException("Неверное использование директивы AIF/AGO");

string label = current.Operation == "AIF" ? current.Operands[1] : current.Operands[0];

bool ready = false;

Stack<bool> agoStack = new Stack<bool>();

int localMacroCount = 0;

for (int j = i; j >= 0; j--)

{

if (macroSourceCode.SourceCodeLines[j].Operation == "MACRO") localMacroCount++;

if (macroSourceCode.SourceCodeLines[j].Operation == "MEND") localMacroCount--;

if (localMacroCount != 0) continue;

if (macroSourceCode.SourceCodeLines[j].Operation?.In("WHILE", "IF") == true)

{

if (agoStack.Count > 0)

{

agoStack.Pop();

}

}

if (macroSourceCode.SourceCodeLines[j].Operation == "ELSE")

{

if (agoStack.Count > 0)

{

agoStack.Pop();

}

agoStack.Push(false);

}

if (macroSourceCode.SourceCodeLines[j].Operation?.In("ENDIF", "ENDW") == true)

{

agoStack.Push(false);

}

if (macroSourceCode.SourceCodeLines[j].Label == label && (agoStack.Count == 0 || agoStack.Peek()))

{

i = j - 1;

ready = true;

break;

}

}

if (!ready)

{

localMacroCount = 0;

for (int j = i; j < macroSourceCode.SourceCodeLines.Count; j++)

{

// Вложенные макросы не смотрим

if (macroSourceCode.SourceCodeLines[j].Operation == "MACRO") localMacroCount++;

if (macroSourceCode.SourceCodeLines[j].Operation == "MEND") localMacroCount--;

if (localMacroCount != 0) continue;

if (macroSourceCode.SourceCodeLines[j].Operation?.In("WHILE", "IF") == true)

{

agoStack.Push(false);

}

if (macroSourceCode.SourceCodeLines[j].Operation == "ELSE")

{

if (agoStack.Count > 0)

{

agoStack.Pop();

}

agoStack.Push(false);

}

if (macroSourceCode.SourceCodeLines[j].Operation?.In("ENDIF", "ENDW") == true)

{

if (agoStack.Count > 0)

{

agoStack.Pop();

}

}

if (macroSourceCode.SourceCodeLines[j].Label == label && (agoStack.Count == 0 || agoStack.Peek()))

{

i = j - 1;

ready = true;

break;

}

}

}

if (!ready)

{

throw new CustomException("Метка " + label + " при директиве " + current.Operation +

" находится вне зоны видимости или не описана");

}

}

continue;

}

}

if (runStack.Peek())

{

macroSourceCode.FirstRunStep(current, this);

}

}

List<string> markedLabels = new List<string>();

macroCount = 0;

foreach (CodeEntity se in macroSourceCode.AssemblerCode)

{

if (se.Operation == "MACRO") macroCount++;

if (se.Operation == "MEND") macroCount--;

if (macroCount != 0) continue;

if (!string.IsNullOrEmpty(se.Label))

{

if (markedLabels.Contains(se.Label))

{

throw new CustomException($"{ProcessorErrorMessages.DuplicateLabelInMacro} (Метка {se.Label}, макрос {Name})");

}

markedLabels.Add(se.Label);

}

}

return macroSourceCode;

}

}

public static class CheckMacros

{

public static void CheckAif(Macro te)

{

List<CodeEntity> result = new List<CodeEntity>();

int macroCount = 0;

foreach (CodeEntity se in te.Body)

{

if (se.Operation == "MACRO")

{

macroCount++;

}

else if (se.Operation == "MEND")

{

macroCount--;

}

if (macroCount == 0)

{

result.Add((CodeEntity)se.Clone());

}

}

try

{

foreach (CodeEntity str in result)

{

if (str.Operation == "AIF")

{

if (str.Operands.Count != 2)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectDirectiveOperands} {Directives.Aif} ({str.SourceString})");

}

if (!string.IsNullOrEmpty(str.Label))

{

throw new CustomException($"{ProcessorErrorMessages.DirectiveWithLabel\_1} {Directives.Aif} {ProcessorErrorMessages.DirectiveWithLabel\_2} ({str.SourceString})");

}

for (int i = 0; i < Processor.assemblyMarks.Count; i++)

{

if (Processor.assemblyMarks[i].Name == str.Operands[1] && te.Name == Processor.assemblyMarks[i].Macro)

{

if (Processor.assemblyMarks[i].Used == false)

{

Processor.assemblyMarks[i].Used = true;

}

else

throw new CustomException("Найдено зацикливание или повторное обращение к метке");

}

}

}

if (str.Operation == "AGO")

{

if (str.Operands.Count != 1)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectDirectiveUsage} {Directives.Ago} ({str.SourceString})");

}

if (!string.IsNullOrEmpty(str.Label))

{

throw new CustomException($"{ProcessorErrorMessages.DirectiveWithLabel\_1} {Directives.Ago} {ProcessorErrorMessages.DirectiveWithLabel\_2} ({str.SourceString})");

}

for (int i = 0; i < Processor.assemblyMarks.Count; i++)

{

if (Processor.assemblyMarks[i].Name == str.Operands[0] && te.Name == Processor.assemblyMarks[i].Macro)

{

if (Processor.assemblyMarks[i].Used == false)

{

Processor.assemblyMarks[i].Used = true;

}

else

throw new CustomException("Найдено зацикливание или повторное обращение к метке");

}

}

}

}

}

catch (CustomException ex)

{

throw new CustomException(ex.Message);

}

catch (Exception)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectDirectiveUsage} {Directives.Aif}-{Directives.Ago}");

}

}

public static void CheckMacroLabels(Macro te)

{

var result = new List<CodeEntity>();

int macroCount = 0;

foreach (CodeEntity se in te.Body)

{

if (se.Operation == "MACRO")

{

macroCount++;

}

else if (se.Operation == "MEND")

{

macroCount--;

}

if (macroCount == 0)

{

result.Add(se.Clone() as CodeEntity);

}

}

var markedLabels = new List<string>();

foreach (var sourceLine in result)

{

if (string.IsNullOrEmpty(sourceLine.Label) || sourceLine.Operation.ToUpper() == "MACRO")

{

continue;

}

if (markedLabels.Contains(sourceLine.Label))

{

throw new CustomException($"{ProcessorErrorMessages.DuplicateLabelInMacro} (Метка {sourceLine.Label}, макрос {te.Name})");

}

markedLabels.Add(sourceLine.Label);

}

}

public static void CheckLocalTmo()

{

foreach (Macro te in MacrosStorage.Entities)

{

te.LocalTmo.Clear();

Macro current = te;

while (current != MacrosStorage.Root)

{

te.LocalTmo.AddRange(current.ChildrenMacros);

current = current.ParentMacros;

}

te.LocalTmo.AddRange(current.ChildrenMacros);

te.LocalTmo.Remove(te);

}

MacrosStorage.Root.LocalTmo = MacrosStorage.Root.ChildrenMacros;

}

public static void CheckWhile(Macro te)

{

int whileCount = 0;

List<CodeEntity> result = new List<CodeEntity>();

int macroCount = 0;

foreach (CodeEntity se in te.Body)

{

if (se.Operation == "MACRO")

{

macroCount++;

}

else if (se.Operation == "MEND")

{

macroCount--;

}

if (macroCount == 0)

{

result.Add(se.Clone() as CodeEntity);

}

}

try

{

foreach (CodeEntity str in result)

{

if (str.Operation == "WHILE")

{

if (str.Operands.Count != 1)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectDirectiveOperands} {Directives.While} ({str.SourceString})");

}

if (!string.IsNullOrEmpty(str.Label))

{

throw new CustomException($"{ProcessorErrorMessages.DirectiveWithLabel\_1} {Directives.While} {ProcessorErrorMessages.DirectiveWithLabel\_2} ({str.SourceString})");

}

whileCount++;

}

else if (str.Operation == "ENDW")

{

if (str.Operands.Count != 0)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectDirectiveOperands} {Directives.Endw} ({str.SourceString})");

}

if (!string.IsNullOrEmpty(str.Label))

{

throw new CustomException($"{ProcessorErrorMessages.DirectiveWithLabel\_1} {Directives.Endw} {ProcessorErrorMessages.DirectiveWithLabel\_2} ({str.SourceString})");

}

whileCount--;

if (whileCount < 0)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectDirectiveUsage} {Directives.While}-{Directives.Endw}");

}

}

else if ((str.Operation == "MACRO" || str.Operation == "MEND") && whileCount > 0)

{

throw new CustomException(ProcessorErrorMessages.MacroDefinitionInLoop);

}

else if (str.Operation == Directives.Variable && whileCount > 0)

{

throw new CustomException(ProcessorErrorMessages.VariablesInLoop);

}

else if (!string.IsNullOrEmpty(str.Label) && str.Operation != "MACRO" && whileCount > 0)

{

throw new CustomException(ProcessorErrorMessages.LabelsInLoop);

}

}

if (whileCount != 0)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectDirectiveUsage} {Directives.While}-{Directives.Endw}");

}

}

catch (CustomException ex)

{

throw new CustomException(ex.Message);

}

catch (Exception)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectDirectiveUsage} {Directives.While}-{Directives.Endw}");

}

}

public static void CheckIf(Macro te)

{

Stack<bool> stackIfHasElse = new Stack<bool>();

// Вложенные макросы не обрабатываем

List<CodeEntity> result = new List<CodeEntity>();

int macroCount = 0;

foreach (CodeEntity se in te.Body)

{

if (se.Operation == "MACRO")

{

macroCount++;

}

else if (se.Operation == "MEND")

{

macroCount--;

}

if (macroCount == 0)

{

result.Add(se.Clone() as CodeEntity);

}

}

try

{

foreach (CodeEntity str in result)

{

if (str.Operation == "IF")

{

if (str.Operands.Count != 1)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectDirectiveOperands} {Directives.If} ({str.SourceString})");

}

if (!string.IsNullOrEmpty(str.Label))

{

throw new CustomException($"{ProcessorErrorMessages.DirectiveWithLabel\_1} {Directives.If} {ProcessorErrorMessages.DirectiveWithLabel\_2} ({str.SourceString})");

}

stackIfHasElse.Push(false);

}

if (str.Operation == "ELSE")

{

if (str.Operands.Count != 0)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectDirectiveOperands} {Directives.Else} ({str.SourceString})");

}

if (!string.IsNullOrEmpty(str.Label))

{

throw new CustomException($"{ProcessorErrorMessages.DirectiveWithLabel\_1} {Directives.Else} {ProcessorErrorMessages.DirectiveWithLabel\_2} ({str.SourceString})");

}

if (stackIfHasElse.Peek() == true)

{

throw new CustomException($"{ProcessorErrorMessages.ExtraBranch} {Directives.Else} ({str.SourceString})");

}

else

{

stackIfHasElse.Pop();

stackIfHasElse.Push(true);

}

}

if (str.Operation == "ENDIF")

{

if (str.Operands.Count != 0)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectDirectiveOperands} {Directives.EndIf}");

}

if (!string.IsNullOrEmpty(str.Label))

{

throw new CustomException($"{ProcessorErrorMessages.DirectiveWithLabel\_1} {Directives.EndIf} {ProcessorErrorMessages.DirectiveWithLabel\_2} ({str.SourceString})");

}

stackIfHasElse.Pop();

}

}

if (stackIfHasElse.Count > 0)

{

throw new CustomException($"{ProcessorErrorMessages.DirectiveMissed} {Directives.EndIf}");

}

}

catch (CustomException ex)

{

throw new CustomException(ex.Message);

}

catch (Exception)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectDirectiveUsage} {Directives.If}-{Directives.EndIf}");

}

}

public static void CheckInner(CodeEntity current, Stack<ConditionalDirective> stack)

{

if (current.Operation == "IF")

{

return;

}

if (current.Operation == "ELSE")

{

if (stack.Count > 0 && stack.Peek() != ConditionalDirective.IF)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectDirectiveUsage} {Directives.Else}");

}

return;

}

if (current.Operation == "ENDIF")

{

if (stack.Count > 0 && stack.Peek() != ConditionalDirective.IF && stack.Peek() != ConditionalDirective.ELSE)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectDirectiveUsage} {Directives.EndIf}");

}

return;

}

if (current.Operation == "WHILE")

{

return;

}

if (current.Operation == "ENDW")

{

if (stack.Count > 0 && stack.Peek() != ConditionalDirective.ENDIF)

{

throw new CustomException($"{ProcessorErrorMessages.IncorrectDirectiveUsage} {Directives.Endw}");

}

return;

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

namespace SystemSoftware.MacroProcessor

{

public class CodeEntity : ICloneable

{

public string SourceString { get; set; }

public Processor Sources { get; set; }

public string Label { get; set; }

public string Operation { get; set; }

public List<string> Operands { get; set; } = new List<string>();

public bool IsRemove { get; set; }

public int CallNumber { get; set; }

public override string ToString()

{

string temp = "";

if (!string.IsNullOrEmpty(Label) && !Label.Contains('%'))

{

temp += Label + ": ";

}

else if (!string.IsNullOrEmpty(Label) && Label.Contains('%'))

{

temp += Label + " ";

}

temp += Operation;

foreach (string op in Operands)

{

temp += " " + op;

}

return temp;

}

public object Clone()

{

var clone = (CodeEntity)this.MemberwiseClone();

clone.Operands = new List<string>(Operands);

return clone;

}

}

}