**БУ ВО Ханты-Мансийского автономного округа – Югры**

**« СУРГУТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**Политехнический институт**

**Кафедра автоматики и компьютерных систем**

***ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА***

***(БАКАЛАВРСКАЯ РАБОТА)***

***по направлению подготовки 27.03.04 «Управление в технических системах»***

***по направлению подготовки 09.03.04 «Программная инженерия»***

***на тему: Ассемблирование кода Intel 8080 и генерация листинга на его основе***

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Студент(ка) группы 605-11 |  |  |  |  |  | *Давыдов Кирилл Александрович* |
|  |  | (дата) |  | (подпись) |  | (фамилия, имя, отчество) |
| Руководитель |  |  |  |  |  | *Запевалова Лариса Юрьевна* |
|  |  |  |  |  |  |  |
| Нормоконтроль |  |  |  |  |  | *Кривицкая Марина Александровна* |
|  |  |  |  |  |  |  |
| «Допустить к защите» заведующий кафедрой |  |  |  |  |  | *Запевалов Андрей Валентинович* |

Сургут

2025

БУ ВО Ханты-Мансийского автономного округа – Югры

«Сургутский государственный университет»

|  |  |
| --- | --- |
| «УТВЕРЖДАЮ»: |  |
| Зав. кафедрой |  |
|  | (подпись) |

ЗАДАНИЕ

на выполнение выпускной квалификационной работы

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | Выдано студенту |  | |
| |  |  | | --- | --- | | 1. Тема ВКР |  | |
|  |
|  |
|  |
| |  |  |  |  | | --- | --- | --- | --- | | утверждена приказом по вузу № |  | от |  | |
| |  |  | | --- | --- | | 2. Дата завершения ВКР | *ХХ.06.20ХХ г.* | |
| |  |  | | --- | --- | | 3. Исходные данные к ВКР |  | |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| |  |  | | --- | --- | | 4. Содержание пояснительной записки (перечень подлежащих разработке вопросов) |  | |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| |  |  | | --- | --- | | 5. Перечень графического материала (с указанием относящихся к нему разделов) |  | |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| |  |  | | --- | --- | | 6. Консультанты по ВКР (с указанием относящихся к ним разделов) |  | |
| *ФИО (Название раздела)* |
|  |
|  |
|  |
|  |
|  |
| |  |  |  |  | | --- | --- | --- | --- | | 7. Дата выдачи задания | *ХХ.ХХ.20ХХ г.* |  |  | |
| |  |  |  |  | | --- | --- | --- | --- | | Руководитель |  |  |  | |
| |  |  |  |  | | --- | --- | --- | --- | |  | подпись |  | (Фамилия И. О.) | |
| |  |  |  |  | | --- | --- | --- | --- | | Задание принял к исполнению |  |  |  | |
| |  |  |  |  | | --- | --- | --- | --- | |  | подпись |  | (Фамилия И.О.) | |

Аннотация

Учебно-методический микропроцессорный стенд УМПК-80 до сих пор находит применение в учебных целях, в частности на нескольких направлениях обучения в Сургутском государственном университете. Выполнение лабораторных работ с его использованием предполагает ввод программ в машинном коде микропроцессора Intel 8080, а также составление отчёта с детальным листингом, включающего в себя используемые адреса, машинный и эквивалентный ассемблерный код, а также комментарии. Наличие листинга значительно упрощает и процесс разработки, но составление и коррекция программы ассемблирования и пересчёта адресов. Вручную или с помощью типичного ассемблера, составление листинга затруднительно и занимает много времени.

В связи с этим была поставлена цель написания ассемблера и генератора листингов в виде модуля для некого текстового процессора. Такой модуль должен позволить автору составить программу для стенда в ассемблерном коде в самом отчёте или в подгружаемых файлах и в результате генерировать детальный листинг прямо в документе. Исправления кода должны отражаться в листинге.

Процесс разработки следует стандартной схеме и включает анализ доступных расширяемых текстовых процессоров, изучение выбранного текстового процессора, изучение языка ассемблера микропроцессоров Intel 8080, написание парсера, ассемблера и инструментов форматирования для генерации листинга. Был выбран текстовый процессор ConTeXt, так как он использует язык Lua, имеет для него библиотеку для составления парсеров и имеет типографический функционал достаточный, чтобы заместить более распространённые текстовые процессоры, например Microsoft Word.

Для изучения языка ассемблера применялось выпущенное в 1977 году официальное руководство. Для составления парсера была применена библиотека LPeg для парсеров на основе РВ-грамматики, включённая в доступные по умолчанию в текстовом процессоре ConTeXt библиотеки. Демонстративные инструменты форматирования используют интерфейс для взаимодействия с ConTeXt через Lua, но в будущем предполагается интерфейс, более приближенный к самому текстовому процессору, и как следствие более удобный.

ConTeXt является весьма непопулярным текстовым процессором. Это можно объяснить отсутствием маркетинга, необходимостью составления документов на специальном (предметно-ориентированным) языке программирования, усиленным фокусом на качество вёрстки ценой сложности инструмента. В связи с этим возможности внедрения разработанного ассемблера практически отсутствуют, пока требования к качеству вёрстки в академической среде ограничены возможностями более популярных текстовых процессоров, прежде всего Microsoft Word.

Полученное на момент написания пояснительной записки программное решение имеет функционал, достаточный для использования его в указанных целях. Автор отчёта может использовать модуль для вызова ассемблера, указав исходный код программы, адрес начала и известные символы, и ассемблер соберёт программу и выдаст на месте вызова листинг. Цель была достигнута, но разработка на этом не заканчивается. Модуль требует доработки в полноценное дополнение для текстового процессора для более удобной работы. В частности, нужно реализовать поддержку ассемблерных директив, составления программ отдельными сегментами, а также предоставить возможность гибкой настройки вида листинга.

Annotation

The UMPK-80, an educational microprocessor system, is being used to this day, particularly within Surgut State University as part of several courses. Laboratory assignments using the system include program entry in Intel 8080 machine code and report preparation. The report requires a detailed code listing that includes addresses used, machine code and its assembly code equivalent, as well as comments. Having such a listing greatly simplifies the development process as well, however any changes to the program require assembly. Done manually or using a separate assembler, the process of making a code listing is time consuming and tedious.

As such, a goal has been set to develop an assembler that doubles as a code listing generator and works as a module for a text processor. Such a module must enable the author to provide a program in assembly code in the report or in auxiliary files and as a result the module must generate a code listing right in the document. Any changes in code must be reflected in the listing.

The development process follows standard procedure and includes the analysis of extensible text processors available, studying the chosen text processor, studying the Intel 8080 assembly language, writing a parser, assembler and formatter capable of generating a code listing. ConTeXt has become the text processor of choice given that it uses the Lua programming language, has a parser builder library for it and it is typographically capable enough to be a replacement for more common text processors such as Microsoft Word.

The official 1977 manual has been the only reference for the study of the Intel 8080 assembly language. The developed parser uses the LPeg library for making PEG parsers as it is included in the ConTeXt installation. A demonstrative formatter has been written which uses simplistic tools to interface with ConTeXt through Lua, however it is planned that it will be replaced with an interface more closely tied to the text processor itself and as such an easier one.

ConTeXt является весьма непопулярным текстовым процессором. Это можно объяснить отсутствием маркетинга, необходимостью составления документов на специальном (предметно-ориентированным) языке программирования, усиленным фокусом на качество вёрстки ценой сложности инструмента. В связи с этим возможности внедрения разработанного ассемблера практически отсутствуют, пока требования к качеству вёрстки в академической среде ограничены возможностями более популярных текстовых процессоров, прежде всего Microsoft Word.

ConTeXt is largely unpopular. It may be caused by a lack of marketing, usage of a domain-oriented language being a requirement for writing documents, the complexity of the tool coming as a price for high quality typesetting. It comes as no surprise that there is little hope for integrating the developed software, not until academic typesetting requirements grow beyond the possibilities of much more popular text processing software such as Microsoft Word.

At the moment of writing this explanatory note the developed software solution has enough functionality to achieve the main goal. A report author can use the module to call the assembler with given source code, starting address and known symbols, and the assembler will build a program and generate a listing in its place. The goal has been reached, but the development process continues as the module requires more work to become easier to use. In particular, the module requires assembly directive support, literate programming style source code splitting and more flexible code listing typesetting settings.

Обозначения и сокращения

|  |  |  |
| --- | --- | --- |
| АСД | — | абстрактное синтаксическое дерево. Структура данных, отражающая структуру (фрагмента) программы |
| РВ‑грамматика | — | грамматика, разбирающая выражение. Один из типов формальной грамматики. Применяется для обработки компьютерных языков (см. также контекстно-свободную грамматику) |
| Текстовый процессор | — | программа, предназначенная для работы над форматированным текстом (составление, печать, стилизация, структуризация и т. д.) |
| УМПК‑80 | — | учебно-методический стенд, предназначенный для практического изучения микропроцессорных систем на начальном уровне. В основе лежит микропроцессор Intel 8080. Используется в Сургутском государственном университете преподавателями и студентами на направлениях «Управление в технических системах» и «Программная инженерия» |
| ConTeXt | — | текстовый процессор на основе системы компьютерной вёрстки TeX. Разрабатывается нидерландской компанией PRAGMA ADE, занимающейся автоматизированной высококачественной вёрсткой. Использует собственно разработанную новейшую версию TeX под названием LuaMetaTeX, имеющую множество улучшений, в том числе встроенный интерпретатор Lua |
| Intel 8080 | — | второй восьмибитный микропроцессор от Intel, выпуск которого начался в апреле 1974 года |
| LPeg | — | Lua Parsing Expression Grammar, библиотека для составления парсеров на основе РВ-грамматики в Lua |
| Lua | — | легко встраиваемый и расширяемый интерпретируемый язык программирования. Нередко используется в полноценных программных решениях как способ пользователю расширить функционал программы. Отличается простотой, скоростью работы интерпретатора и широким применением хеш-таблиц |

Содержание

[Введение 10](#_Toc200915653)

[1 Анализ инструментов и предметной области 13](#_Toc200915654)

[1.1 Существующие решения проблемы 13](#_Toc200915655)

[1.2 Синтаксис языка ассемблирования Intel 8080 15](#_Toc200915656)

[1.3 Технические требования к ПО 18](#_Toc200915657)

[2 Проектирование модуля 20](#_Toc200915658)

[2.1 Доступные текстовые процессоры 20](#_Toc200915659)

[2.2 ConTeXt и Lua: CLD 25](#_Toc200915660)

[2.3 Общая структура модуля 26](#_Toc200915661)

[2.4 Структура данных о командах ассемблера 27](#_Toc200915662)

[2.5 Иерархия парсеров 28](#_Toc200915663)

[2.6 Ассемблирование кода 30](#_Toc200915664)

[2.7 Форматирование результата 31](#_Toc200915665)

[3 Написание модуля 33](#_Toc200915666)

[3.1 РВ-грамматика и LPeg 33](#_Toc200915667)

[3.2 Данные о командах ассемблера 34](#_Toc200915668)

[3.3 Парсинг ассемблерного кода 35](#_Toc200915669)

[3.4 Процесс ассемблирования 38](#_Toc200915670)

[3.5 Форматирование результата 38](#_Toc200915671)

[4 Тестирование программного решения 39](#_Toc200915672)

[4.1 Парсер чисел 39](#_Toc200915673)

[4.2 Парсер выражений 39](#_Toc200915674)

[4.3 Ассемблер и генератор листинга 39](#_Toc200915675)

[5 Сопровождение ПО 40](#_Toc200915676)

[Заключение 41](#_Toc200915677)

[Conclusion 42](#_Toc200915678)

[Список использованных источников 43](#_Toc200915679)

[Приложение А Характеристика текстового процессора ConTeXt 45](#_Toc200915680)

[Приложение Б Команды микропроцессора Intel 8080 46](#_Toc200915681)

[Приложение В Листинг модуля 47](#_Toc200915682)

1. Введение

Выполнение лабораторных работ с учебно-методическим микропроцессорным стендом УМПК-80 включает в себя анализ проблемы, проектирование программного решения, последующую реализацию и тестирование с возможными доработки по мере необходимости. Это проводится в целях закрепления учебного материала по дисциплине «Организация ЭВМ». Ввод программ осуществляется в машинном коде микропроцессора в выбранную студентом область памяти. [\in[item: umpk-80]] По окончании работы необходимо привести отчёт, одним из обязательных элементов которого является детальный листинг в виде таблицы, включающей в себя адреса каждого байта машинного кода программы, метки для наглядности, сам машинный код и эквивалентный ассемблерный код, а также комментарии (таблица 1)

Таблица 1 — Пример листинга

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Адрес | Метка | Машинный  код | Ассемблерный код | Комментарии |
| 0990H | memcpy: | 1A | LDAX D | Байт считывается из источника… |
| 0991H |  | 77 | MOV M, A | … и он записывается по адресу назначения |
| 0992H |  | 13 | INX D | Следующая ячейка памяти источника… |
| 0993H |  | 23 | INX H | Следующая ячейка памяти назначения… |
| 0994H |  | 0D | DCR C | Одним байтом меньше, декремент счётчика |
| 0995H |  | C2 | JNZ memcpy | Если есть ещё байты, продолжить копирование |
| 0996H |  | 90 |
| 0997H |  | 09 |
| 0998H |  | C9 | RET | Выход из подпрограммы |

Как видно, каждый байт занимаемой (под)программой памяти имеет заранее определённый адрес, так как программист сам выбирает, где в адресном пространстве расположить программу. В листинге метка имеет ограниченную пользу, так как все адреса известны заранее, но с помощью них можно сделать переходы более наглядными. Ассемблерный код и комментарии делают листинг понятнее.

Несмотря на то, что формально листинг необходим лишь как элемент документации, особенности ввода программ на учебном стенде [\in[item: umpk-80]] делают его необходимым ещё на этапе реализации и даже реализации решения, прежде всего позволяя сопоставить машинный код и адресное пространство системы для последующего ввода программы в систему. Вариантов составления его не так много:

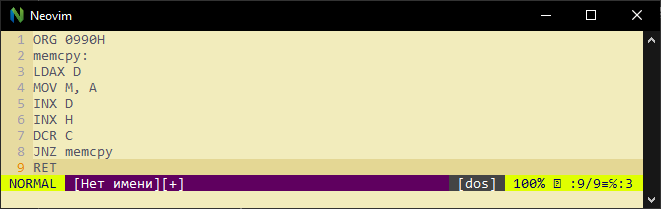
* Листинг можно составить вручную, что делает большинство студентов. Как правило, для этого используются электронные таблицы (в частности, Microsoft Office Excel). При аккуратном ведении листинга можно обеспечить правильную последовательность используемых адресов и даже простоту исправления неверных адресов и пропущенных команд, но процесс ассемблирования придётся выполнять вручную. По очевидным причинам это несёт за собой вероятность совершения ошибок.
* Можно воспользоваться существующими ассемблерами в дополнении к электронным таблицам. Как правило, для них нужно иметь исходный код в формате, представленном на рисунке 1. Для наглядности представлена программа из таблицы 1.  
  

Рисунок 1 — Пример исходного кода программы для типичного ассемблера для Intel 8080

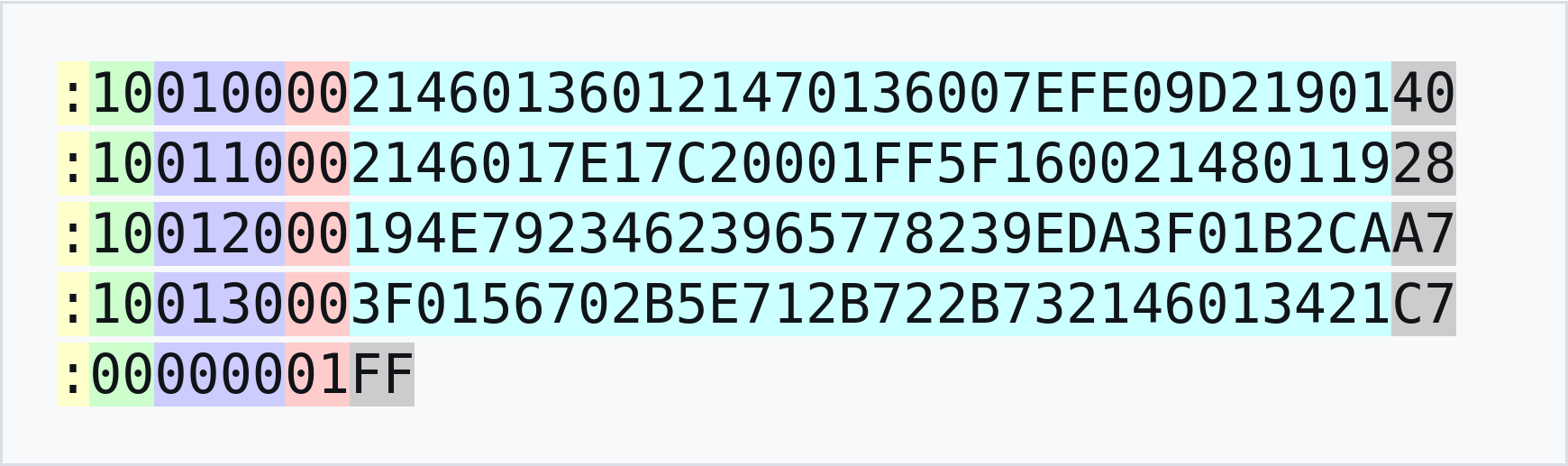
В лучшем случае с помощью них можно получить ассемблерный код в виде файла формата Intel HEX (рисунок 2). Как можно увидеть, он является достаточно неудобным для копирования. Несмотря на это, ассемблеры значительно упрощают работу, самостоятельно генерируя машинный код и отсчитывая адреса.  


Рисунок 2 — Формат Intel HEX. Выделены по порядку: количество байт, адрес, тип записи, данные, контрольная сумма

* В редком случае можно найти инструменты, генерирующие листинг в неком виде. В частности, в работе name [\in[item: umpk emu]] представлен эмулятор, имеющий также функционал для генерации листингов в формате, приведённом в таблице 1. Это значительно облегчает составление отчёта, но требуют повторной генерации вне текстовых процессоров в случае изменения программы.

Необходимо отметить, что решения с применением ассемблера чаще всего предполагают наличие отдельного от текстового процессора ПО. В связи с этим была поставлена цель составить внедрённое в текстовый процессор решение, позволяющее генерировать листинги прямиком в документе. Такой подход позволяет легко составить листинг корректного вида (не требующий никаких стилистических изменений) для использования как при реализации и тестировании программы, так и при составлении документации к ней.

Проблема составления листинга является актуальной, но второстепенной. Это во многом связано с тем, что ПО рассчитано на лиц, которые нуждаются в нём лишь на ограниченное время, хоть эти лица и будут продолжать появляться. Проблема усугубляется и выбором технологий для решения проблемы.

1. Анализ инструментов и предметной области
   1. Существующие решения проблемы

Как уже было упомянуто, есть три основных пути решения проблемы составления листинга: вручную, с применением стандартных ассемблеров, и с помощью специализированного ПО.

Первый способ предполагает использование электронных таблиц для составления листинга. Такой способ не требует установки никакого дополнительного программного обеспечения и подходит для работ, где не ожидается большой объём кода. Среди популярных электронных таблиц можно отметить Microsoft Excel и Google Sheets. Можно также упомянуть и LibreOffice Calc.

Наиболее полезным функционалом при использовании электронных таблиц является автозаполнение адресов. В Excel, например, значительную помощь окажет применение функций ДЕС.В.ШЕСТН и ШЕСТН.В.ДЕС. [\in[item: excel hex2dec], \in[item: excel dec2hex]] Эквивалентные функции можно найти и для Google Sheets. [\in[item: sheets hex2dec], \in[item: sheets dec2hex]] С помощью них можно выполнить нумерацию адресов в шестнадцатеричной системе счисления. Далее ручной работой возможно составить ассемблерный код и найти эквивалентный машинный код, например, используя справочник. Результирующая таблица будет выглядеть как на рисунке 3.

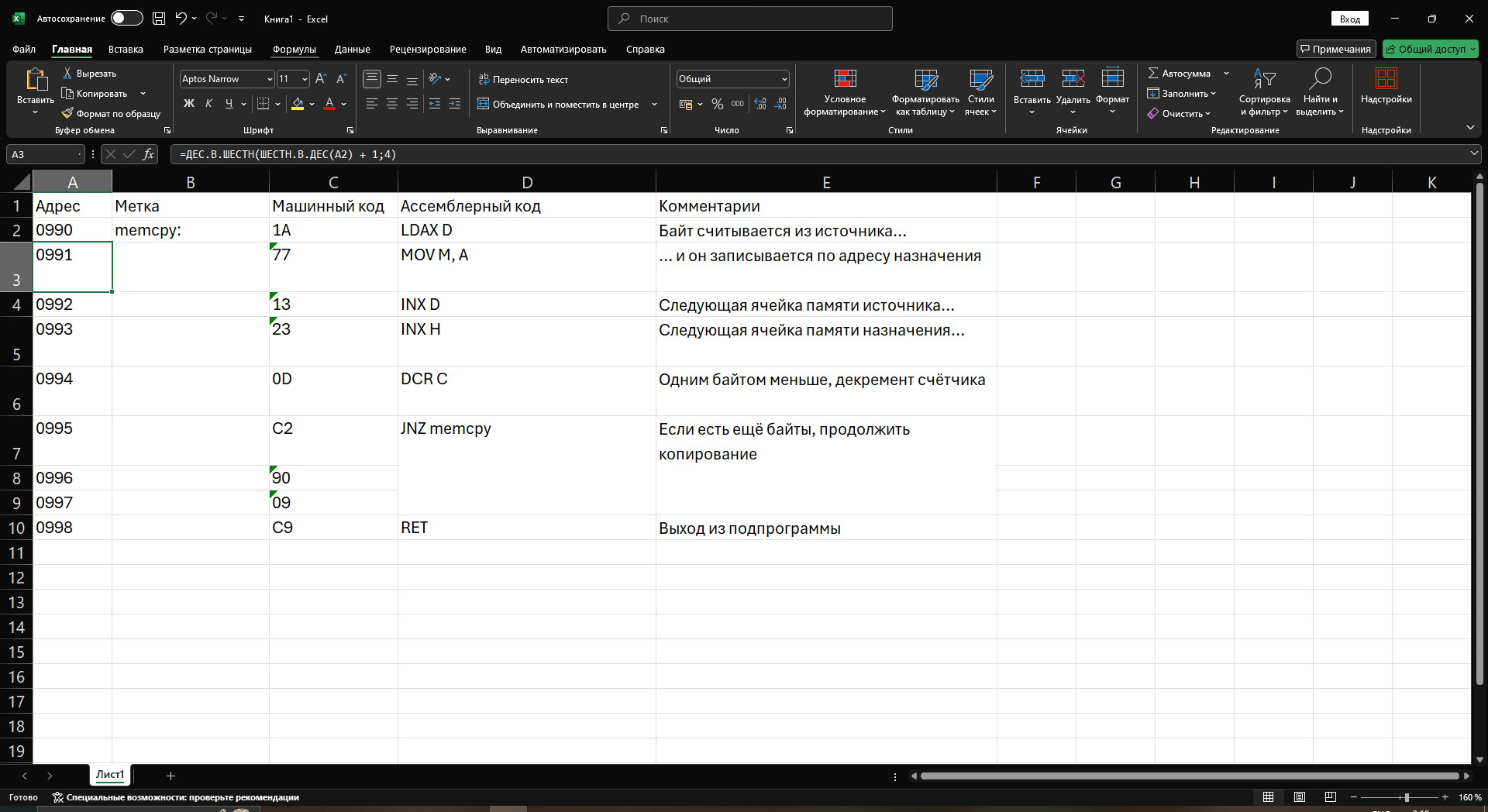


Рисунок 3 — Применение ДЕС.В.ШЕСТН и ШЕСТН.В.ДЕС позволяют выполнить нумерацию адресов в шестнадцатеричном виде

Такой подход лишает пользователя необходимости вести адреса вручную, но при исправлении программы адреса, представленные в коде (по большей части переходы), придётся редактировать вручную. Применение меток способно в действительности усугубить ситуацию, так как они могут навести на ложную мысль совпадения адреса в машинном коде и адреса в ассемблерном коде, но одновременно с этим метки делают проще проверку на их правильность. Весьма вероятно, что преобладающая часть студентов будет пользоваться электронными таблицами и ручным ассемблированием несмотря на другие существующие решения.

Более доступной, но требующей дополнительные программы альтернативой является использование существующих ассемблеров для Intel 8080 вместе с электронными таблицами. Например, есть платформа emuStudio [\in[item: emuStudio]] для эмуляции ряда старых компьютерных систем, а также онлайн ассемблер Pretty Intel 8080 Assembler [\in[item: pretty assembler]]. В случае emuStudio ассемблер для Intel 8080 идёт как элемент набора инструментов для разработки ПО для компьютеров MITS Altair 8800 — представляется возможным запустить его с командной строки, передав ему путь к файлу с ассемблерным кодом (рисунок 4).

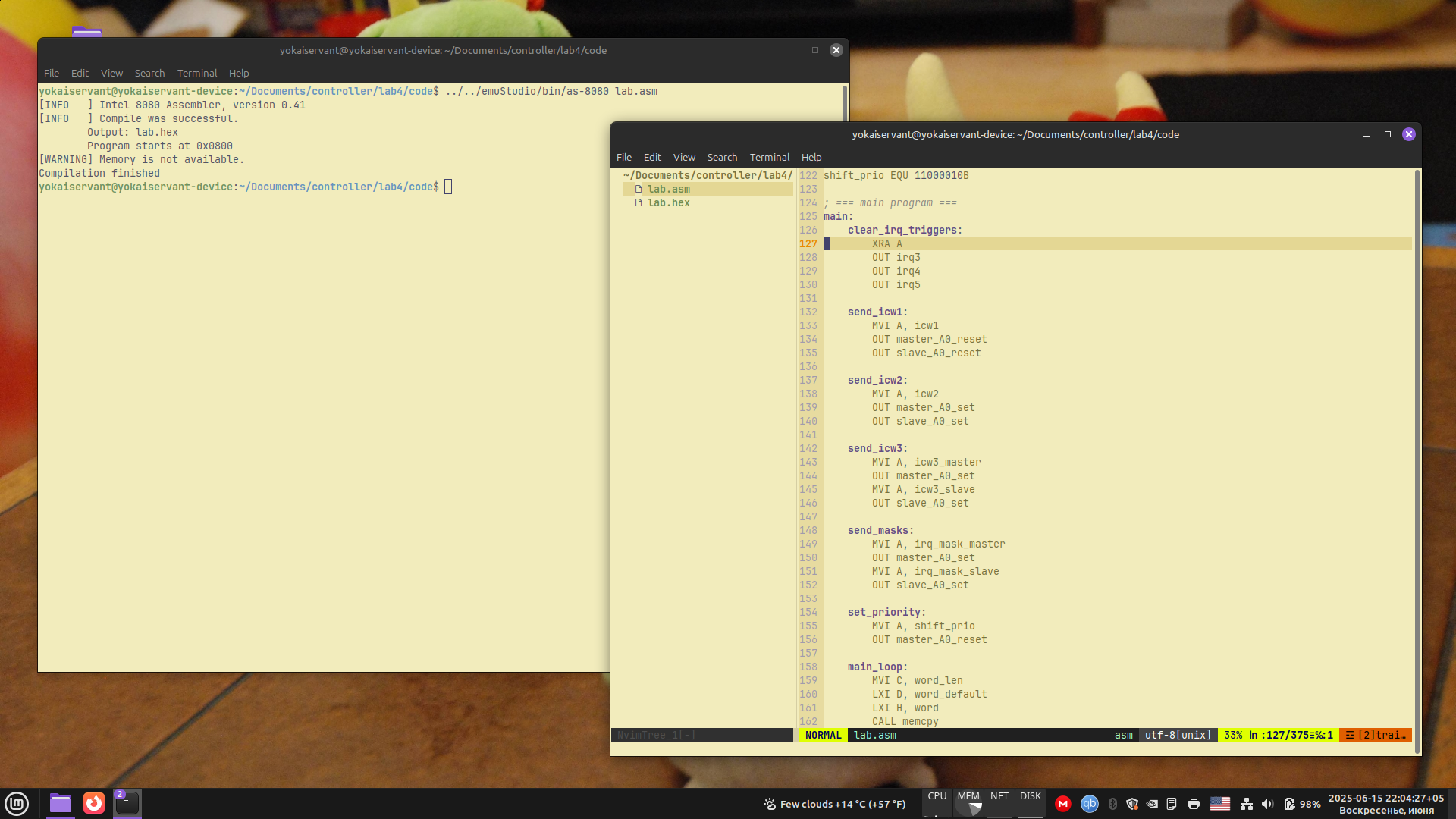


Рисунок 4 — emuStudio обработал файл lab.asm (справа) и выдал файл lab.hex. Сообщение об этом приведено слева

Редактировать код придётся в собственном текстовом редакторе. Ассемблер выдаёт файл формата Intel HEX (см. введение), что делает перенос машинного кода и адресов в листинг затруднительным. В частности, можно потеряться среди полей типа данных и байтов адресов. Сбить с толку может и контрольная сумма в конце каждой строки.

Pretty Intel 8080 Assembler является онлайн решением, обладающим достаточно удобным встроенным текстовым редактором. Наиболее полезной функцией данного решения является отображение машинного и ассемблерного кода бок о бок, что значительно упрощает перенос программы в электронную таблицу и затем в отчёт (рисунок 5). К сожалению, данное решение требует доступа в Интернет, но в большинстве случаев он имеется. Возможным представляется и вариант запуска собственного сервера для локального хостинга.

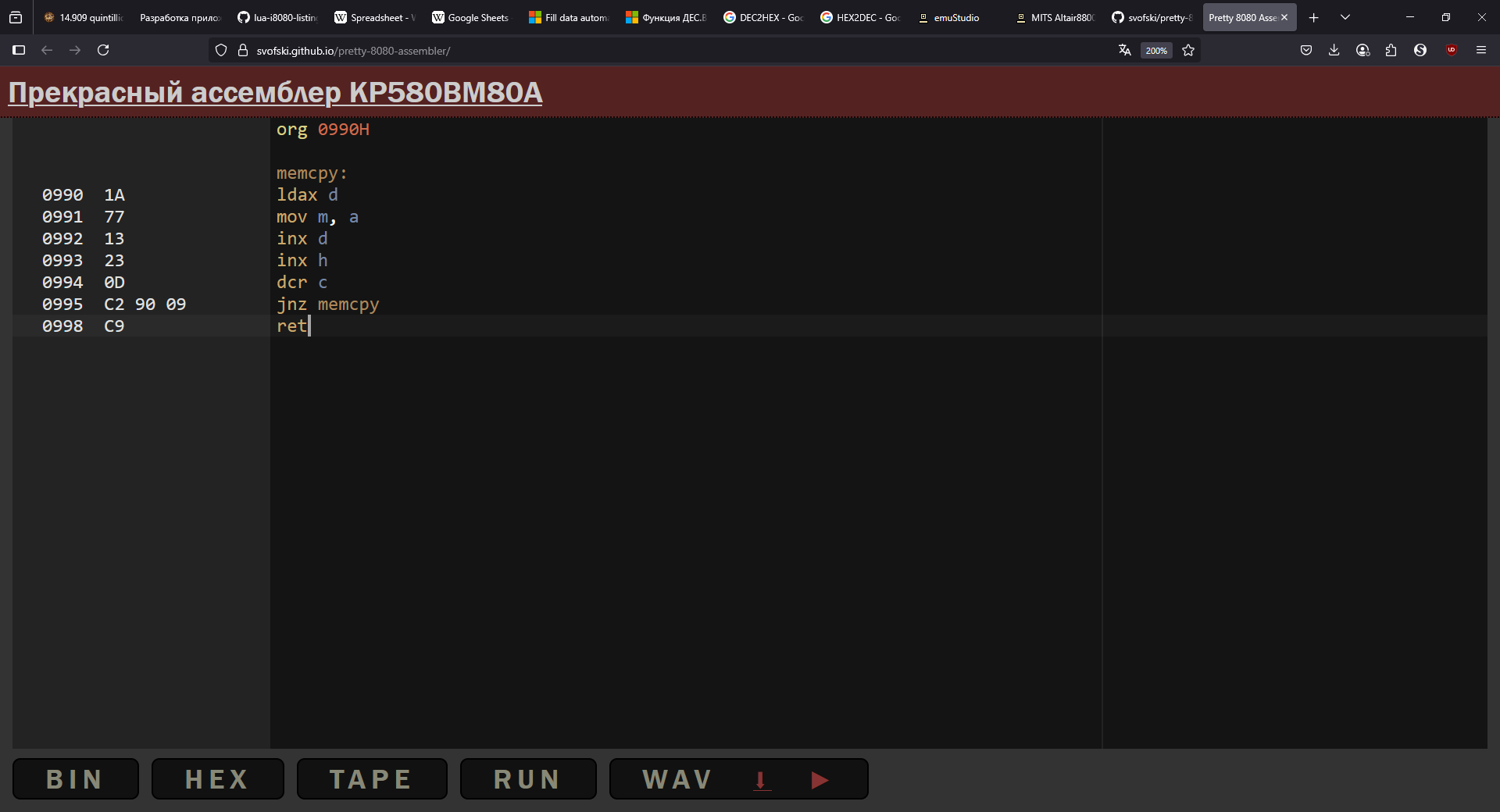


Рисунок 5 — Слева можно увидеть адреса и машинный код. Справа представлен введённый ассемблерный код

У всех представленных программных решений есть недостаток в виде необходимости в работе с несколькими программами для выполнения работы. Данная работа представляет собой попытку составить дополнение к существующему текстовому процессору для того, чтобы было возможным иметь ассемблер прямо там, где составляется отчёт. Теоретически это должно ускорить написание программ для стенда УМПК-80.

В частности, в работе name [\in[item: umpk emu]] представлен эмулятор, имеющий также функционал для генерации листингов в формате

* 1. Синтаксис языка ассемблирования Intel 8080

В данном подразделе приводится описание наиболее важных для репликации аспектов языка ассемблирования микропроцессоров Intel 8080. Необходимо отметить, что в связи со спецификой работы с УМПК-80 отпадает необходимость в большинстве ассемблерных директив, а также макросах. [\in[item: umpk 80]] Функционал оставшихся директив можно повторить вне самого языка. Приведённый ниже материал взят из официального руководства 1977 года. [\in[item: i8080]]

Программы будут предоставляться в ассемблерном коде (см. введение). Исходный файл состоит из строк, каждая из которых может содержать до четырёх полей следующего формата в следующем порядке:

1. Метка или идентификатор. Метки и идентификаторы (вместе символы) мало отличаются друг от друга в плане технической реализации, представляя собой способ обращения к переменным, содержащим беззнаковые целочисленные именованные значения на 16 бит, доступные во время составления программы. Метки сохраняют за собой адрес, с которого будет начинаться следующая команда или результат директивы. Это возможно, так как ассемблер ведёт счётчик позиции, указывающий, где в адресном пространстве будет помещена следующая команда. Идентификаторы позволяют обращаться к произвольным численным значениям, которые можно присвоить с помощью директив EQU и SET. Чтобы пометить следующую команду или результат некоторых директив, необходимо указать название новой метки и поставить двоеточие. При использовании упомянутых директив идентификатор указывается аналогично, но без двоеточия. [\in[item: i8080]]
2. Команда или директива (без операндов). Указывается команда для микропроцессора или директива, побуждающая ассемблер изменить своё состояние или определить содержимое программы вне команд — изменить текущее значение счётчика позиции, установить значение переменной, выделить участок адресного пространства программы под произвольные данные и так далее. Все команды и директивы состоят лишь из латиницы. [\in[item: i8080]]
3. Операнды. Если команда или директива нуждается в операндах, то они идут после и разделяются запятыми. Операнды бывают нескольких видов: регистры, регистровые пары и выражения. [\in[item: i8080]]
4. Комментарии. Начинаются с символа точки с запятой и заканчиваются вместе со строкой. Игнорируются стандартными ассемблерами и поэтому могут содержать любые символы, кроме перехода на следующую строку. [\in[item: i8080]]

Из директив полезными могут оказаться по большей части лишь EQU и ORG. Последняя позволяет установить счётчик позиции, но этот функционал проще вынести на уровень вызова функции ассемблирования в виде параметра. Замена первой возможна аналогичным образом.

До, после и между этими полями может располагаться любое количество символов пустого пространства (кроме переходов на новую строку) — например, пробелов. Стоит отметить множество ограничений, которые имеются у официального ассемблера. Например, должны отсутствовать полностью пустые строки; разрешены лишь символы ASCII; символы не могут совпадать по названию с регистрами и регистровыми парами. [\in[item: i8080]] По возможности будут сниматься ограничения там, где это облегчит работу.

Метки должны состоять из не более шести символов латиницы или цифр [\in[item: i8080]], но имеет смысл увеличить возможную длину и разрешить ряд других символов, например подчёркивание. Таблица символов следит за рядом атрибутов меток и идентификаторов, которые не имеют смысла при работе с УМПК-80, поэтому имеет смысл хранить лишь сами символьные адреса и присвоенные значения.

Регистры, регистровые пары и выражения позволяют представить необходимые для оператора данные. Доступны регистры A, B, C, D, E, H, L, M, регистровые пары B, D, H, SP, PC, PSW. Команды различаются по тому, какие регистры (регистровые пары) могут использоваться. Выражения необходимы там, где нужно получить адрес, номер порта или непосредственно данные. В выражениях могут участвовать численные данные в шестнадцатеричном, десятичном, восьмеричном и двоичном форматах; счётчик позиции; символы (метки и идентификаторы). [\in[item: i8080]]

Для записи числа в шестнадцатеричном формате нужно начать значение с десятичной цифры, после чего оставшиеся цифры могут быть шестнадцатеричными. Вероятно, что это ограничение будет снято. Шестнадцатеричные числа оканчиваются на постфикс H. Десятичное число записывается десятичными цифрами и может оканчиваться на постфикс D. Восьмеричные числа оканчиваются на постфикс O или Q, двоичные — на постфикс B. [\in[item: i8080]]

Текущее значение счётчика позиции (т. е. во время размещения команды с операндом-выражением) доступно в выражениях как символ $. Возможно обратиться и к меткам/идентификаторам — для этого необходимо, чтобы за два обхода ассемблерного кода было известно хранящееся за символом значение. Так можно использовать метки, определённые после выражений, в которых они используются. [\in[item: i8080]]

Выражения состоят из выше представленных данных, объединённых операторами. Порядок следования следующий:

* Выражения в скобках;
* Операции извлечения старшего и младшего байтов HIGH и LOW соответственно;
* Умножение \*, деление /, остаток от деления MOD, поразрядные сдвиги SHL, SHR;
* Сложение + и вычитание -;
* Операторы отношений: равенство EQ, неравенство NE, меньше LT, больше GT, меньше или равно LE, больше или равно GE;
* Логическое отрицание NOT;
* Конъюнкция AND;
* Дизъюнкция обыкновенная OR и исключающая XOR.

Как можно догадаться, промежуточные результаты также являются беззнаковыми целочисленными значениями на два байта. Операции одного порядка выполняются слева направо, т. е. они лево ассоциативны. [\in[item: i8080]]

Стоит отметить, что такой порядок делает логическое отрицание одной из последних операций. Это непривычно, учитывая, что в современных языках программирования наблюдается противоположное. В связи с этим имеет смысл поменять приоритет логического отрицания.

На рисунке 6 можно увидеть все варианты представления данных и несколько выражений для примера. В приложении Б представлены все команды микропроцессора Intel 8080.

Рисунок 6

* 1. Технические требования к ПО

Выдвигается достаточно ограниченный перечень требований к разрабатываемому решению:

* ПО должно представлять собой дополнение к существующему текстовому процессору. ПО не может быть отдельно запускаемой программой.
* ПО должно иметь возможность вести несколько программ в одном документе. Ассемблирование одной программы не может привести к состоянию ПО, где ассемблирование другой программы представляется невозможным. Для этого достаточно избегать использования глобальных переменных для хранения результатов ассемблирования и форматирования.
* ПО должно хранить первоначальный ассемблерный код, на основе которого был сгенерирован листинг. Возможности составляемого модуля в данном аспекте ограничены возможностями текстового процессора, в связи с чем является разумным рассмотреть доступные текстовые процессоры. Данное требование означает, что не допускается, например, спрашивать у пользователя ассемблерный код, использовать его для составления листинга и затем удалять его из памяти. Составляемое программное решение должно помогать составлять программу и поэтому оно должно хранить ассемблерный код для дальнейшего его написания пользователем. Предпочтение отдаётся текстовым процессорам, позволяющим хранить код прямо там, где будет листинг, так как это повышает наглядность. При этом не исключается вариант использования дополнительных файлов.
* Так как реализуемое программное решение будет представлять собой модуль к некому текстовому редактору, появляются ограничения по доступным библиотекам. Поэтому ПО должно использовать минимальное количество внешних библиотек, чтобы не было необходимым следить за обновлениями каждой из них. Со стороны текстового процессора это значит, что предпочтительным является наличие необходимого для написания ассемблера функционала в пределах самого текстового процессора. Это упростит написание и сопровождение модуля.

Текстовые процессоры являются достаточно нересурсоёмким программным обеспечением, поэтому нереалистично применение затратных по памяти и вычислительным ресурсам технологий, например искусственного интеллекта. Также не имеет смысл чрезмерный функционал, например автозаполнение при написании ассемблерного кода. Это лишь модуль, единственная цель которого — обрабатывать ассемблерный код и делать из него листинги в виде таблицы.

1. Проектирование модуля
   1. Доступные текстовые процессоры

Основным фактором при выборе текстового процессора в данном случае будет расширяемость за счёт встроенного в него языка программирования. Предпочтение отдаётся популярным языкам программирования, так как у тех больше пользовательских библиотек, в том числе и библиотек для составления парсеров. Надежда есть на то, что количество намеренно установленных библиотек будет весьма ограниченным. Это не единственный фактор — так как вероятно то, что пользователю придётся переключаться на новый текстовый процессор для того, чтобы он мог воспользоваться модулем, лучше выбрать наиболее подходящий для академических целей текстовый процессор.

Текстовые процессоры можно поделить на две группы по идеологии: более распространённую WYSIWYG (англ. *what you see is what you get*, «получишь, что видишь») и более редкую WYSIWYM (англ. *what you see is what you mean*, «имеешь в виду то, что видишь»).

Первая группа включает в себя более популярные опции, как Microsoft Word. Как следует из названия идеологии, текстовые процессоры данного вида представляют пользователю документ в том виде, в котором он будет представляться читателям. Они имеют собственные редакторы, позволяющие представить форматированный текст с различными комбинациями кеглей, шрифтов, стилей, отступов и т. д. Эти редакторы позволяют автору выбрать конкретную семью шрифтов и при вводе текста увидеть, как она будет смотреться в документе. [\in[item: wysiwyg]]

Из текстовых процессоров идеологии WYSIWYG были рассмотрены:

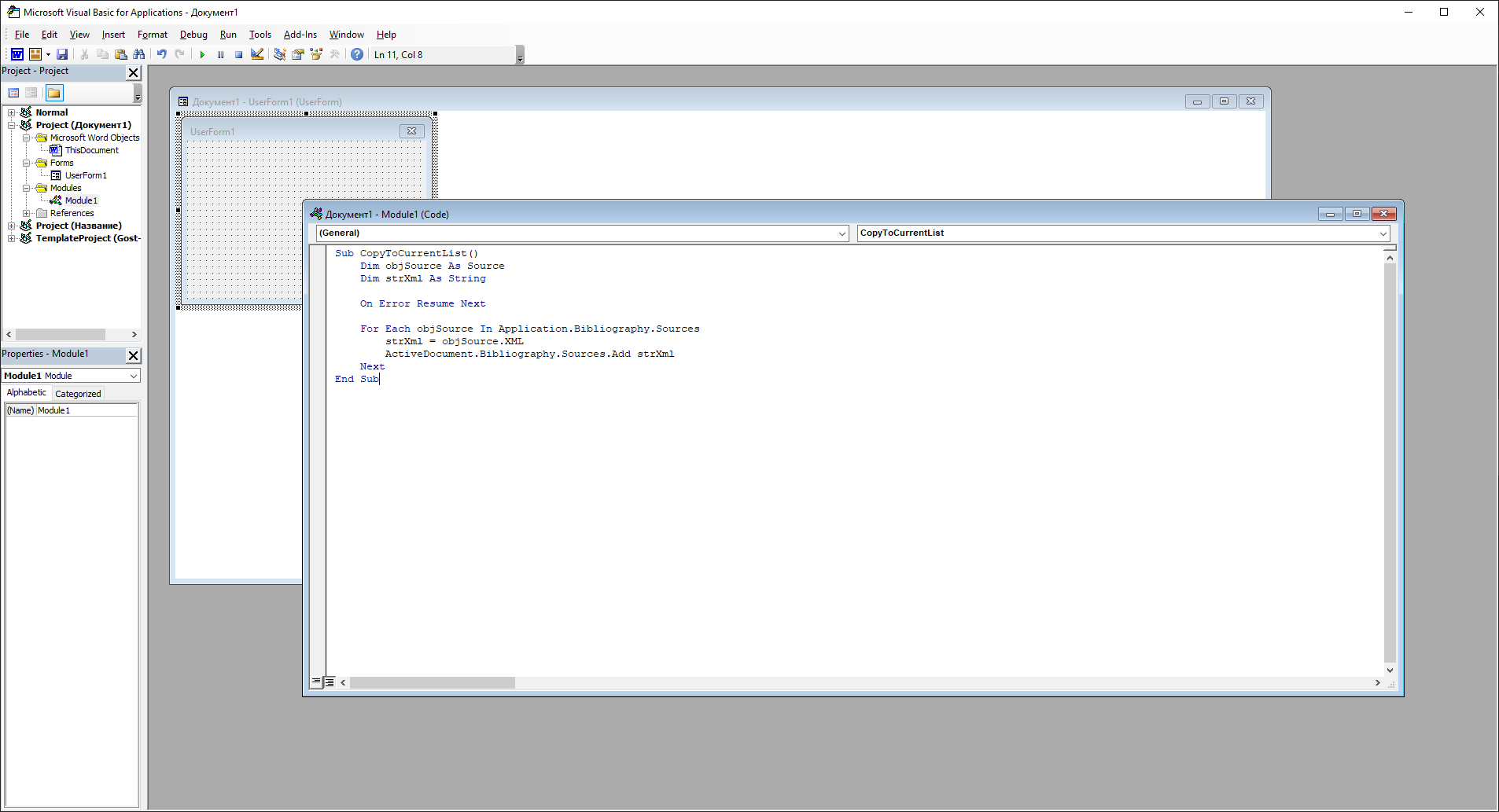
* Microsoft Word. Имеет поддержку макросов на собственном диалекте языка программирования Basic под названием Visual Basic for Applications. Он является упрощённой версией Visual Basic. На сайте Microsoft можно найти достаточно подробную документацию по языку. Язык имеет доступный даже непрограммистам синтаксис и имеет широкие возможности по взаимодействию с другим ПО от Microsoft. [\in[item: ms vba help]] На рисунке 7 можно увидеть окно редактора макросов Word.  
  

Рисунок 7 — Редактор позволяет создавать как модули кода, так и формы для взаимодействия с пользователем

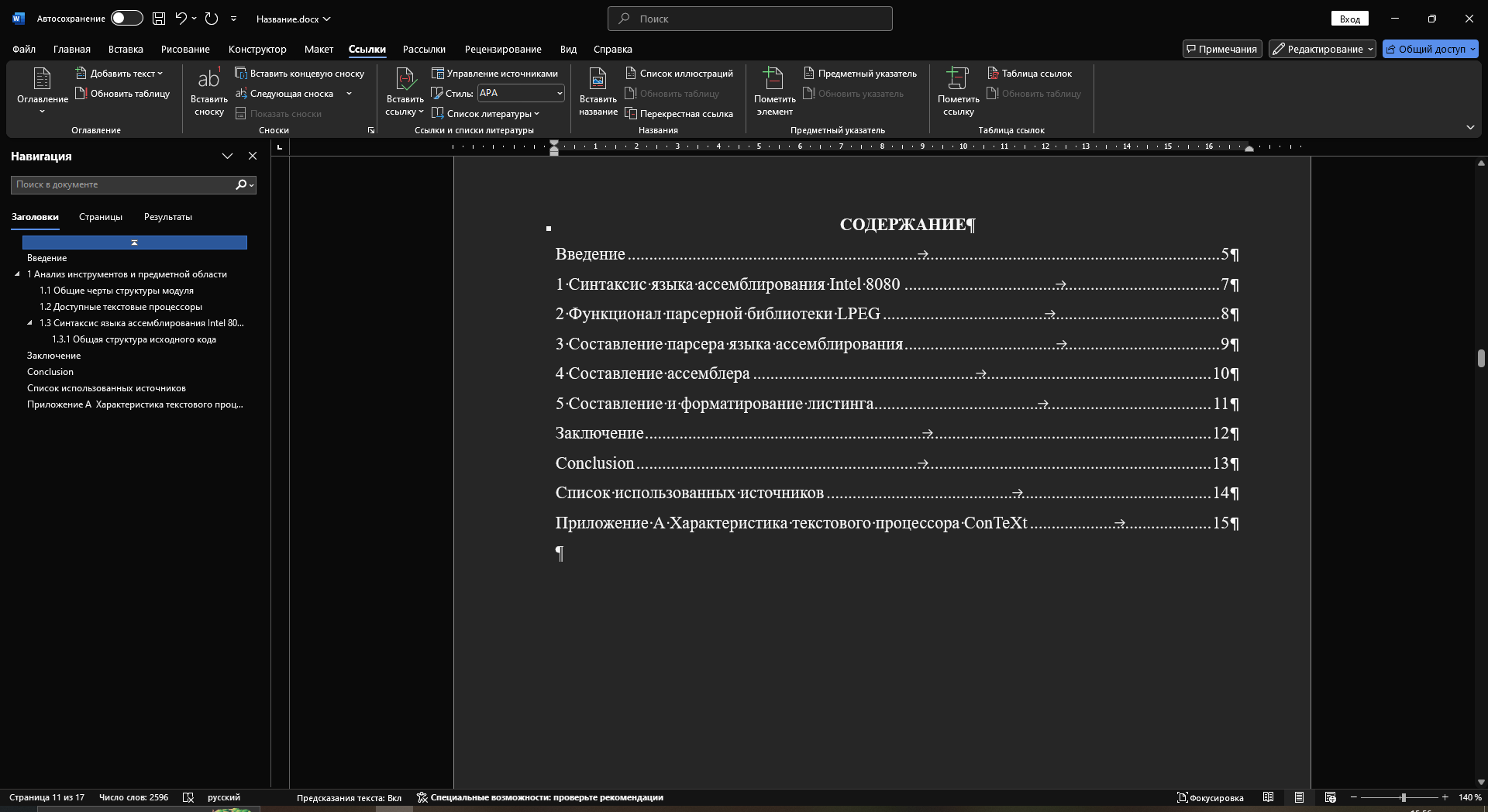
Представляется возможным написать ряд макросов для генерации листинга из исходного кода, но возникает ряд проблем. Во-первых, язык макросов VBA не обладает значительным объёмом библиотек для составления парсеров. В частности, была найдена лишь одна малоизвестная библиотека, последнее обновление которой было пять лет назад. [\in[item: vba peg]] В связи с этим представляется необходимым составлять парсер вручную. Во-вторых, в связи с упором на визуальную составляющую хранить ассемблерный код в тексте документа не представляется возможным — остаётся лишь располагать его в иных местах. Например, можно воспользоваться переменными документа [\in[item: vba doc vars]], или сохранять код в отдельных файлах. В-третьих, результирующий модуль будет требовать от пользователя обновлять таблицу вручную при каждом изменении исходного кода. Такое поведение аналогично тому у оглавлений в Word (рисунок 8); ручные обновления плохо сочетаются с подходом «немедленной готовности» документов после редактирования в текстовых процессорах WYSIWYG.  


Рисунок 8 — Содержание документа в тексте (справа) отстаёт от действительного (панель навигации справа). Необходимо обновить содержание в тексте через выпадающее меню

Стоит отметить, что составлять документы с достаточно сложной структурой в Microsoft Word представляется затруднительным. В частности, создание разделов, подразделов и т. д. требует использования многоуровневых списков с определёнными стилями на верхних уровнях, что порой мешает созданию произвольных списков внутри них. Это усложняет его использование в академических кругах.

* LibreOffice Writer. Текстовый процессор имеет два основных языка для написания макросов — Python и собственный диалект Basic, частично повторяющий функционал Visual Basic for Applications. [\in[item: libreoffice basic]] Скрипты Python в LibreOffice Writer могут использовать пользовательские библиотеки, которые можно установить как в общую папку, так и в пользовательскую папку. Также скрипты могут быть встроены в документ. [\in[item: libreoffice python]] Для Python легче найти библиотеки и инструменты для составления парсера, которые можно использовать для создания генератора листингов. [\in[item: pegen], \in[item: parsimonious]] Несмотря на это, проблема обновления листинга, хранения ассемблерного кода и неудобства структуризации документа всё так же применимы в связи с тем, как работает WYSIWYG. На рисунке 9 представлен редактор макросов LibreOffice Writer.

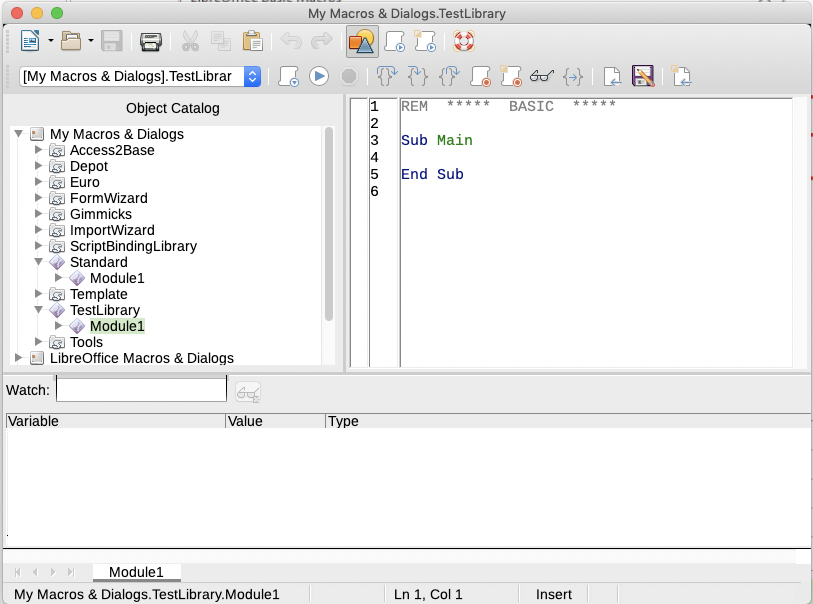


Рисунок 9 — Редактор макросов LibreOffice отчасти похож на тот у Microsoft Word

Альтернативу текстовым процессорам группы WYSIWYG составляют процессоры WYSIWYM. Вместо того, чтобы текстовый процессор немедленно отражал то, как выглядит документ, он позволяет лишь составить его структуру. Нередко подобные программы имеют собственный язык разметки или даже программирования для составления текстов. Такой подход требует конвертирования из исходного файла в формат для просмотра (HTML, PDF и др.), что является как положительной, так и отрицательной особенностью. Проверить состав текста, его структуру и стилизацию на правильность сложнее без экспорта, но становится гораздо проще автоматизировать процесс генерирования структуры и текста, например для составления документации из комментариев к коду. В случае более сложных текстовых процессоров данной группы могут быть доступны продвинутые инструменты для составления структуры в документе. [\in[item: wysiwym]]

При использовании текстовых процессоров WYSIWYM представляется возможным хранить ассемблерный код прямо в документе, так как финальный вид результирующего PDF-файла определяется выполняемыми командами при обработке.

Из текстовых процессоров данной идеологии были рассмотрены:

* LaTeX. Несмотря на то, что LaTeX (и остальные решения ниже) обычно не называют текстовыми процессорами, а, например, системами компьютерной вёрстки, функционал пересекается достаточно для рассмотрения. LaTeX является системой подготовки документов высокого типографического качества, являющейся пакетом макросов для TeX. Чаще всего она используется для составления средних или больших научно-технических документов. LaTeX поддерживает структуризацию документов на разделы с автоматическим нумерованием, вёрстку сложных математических формул, размещение рисунков и таблиц с автоматическим нумерованием, легко обновляемые перекрёстные ссылки на вышеописанные элементы. [\in[item: latex]]

Широкий список пакетов значительно расширяют LaTeX, позволяя автоматизировать ведение библиографий [\in[item: ctan biblatex]], составление графиков [\in[item: ctan pgfplots]], произвольной графики [\in[item: ctan pgf]] и т. д. LaTeX имеет собственный ненастраиваемый шаблон для статей и ожидается, что его не нужно менять; по необходимости можно скачать дополнительные, более гибкие шаблоны. Один из наиболее популярных наборов настраиваемых шаблонов — KOMA-script. [\in[item: ctan koma-script]]. LaTeX имеет активное сообщество, но страдает от малого объёма документации. Несмотря на сложность работы в LaTeX (на рисунке 10 можно увидеть пример работы с исходным файлом), система является распространённой опцией в западных академических кругах.

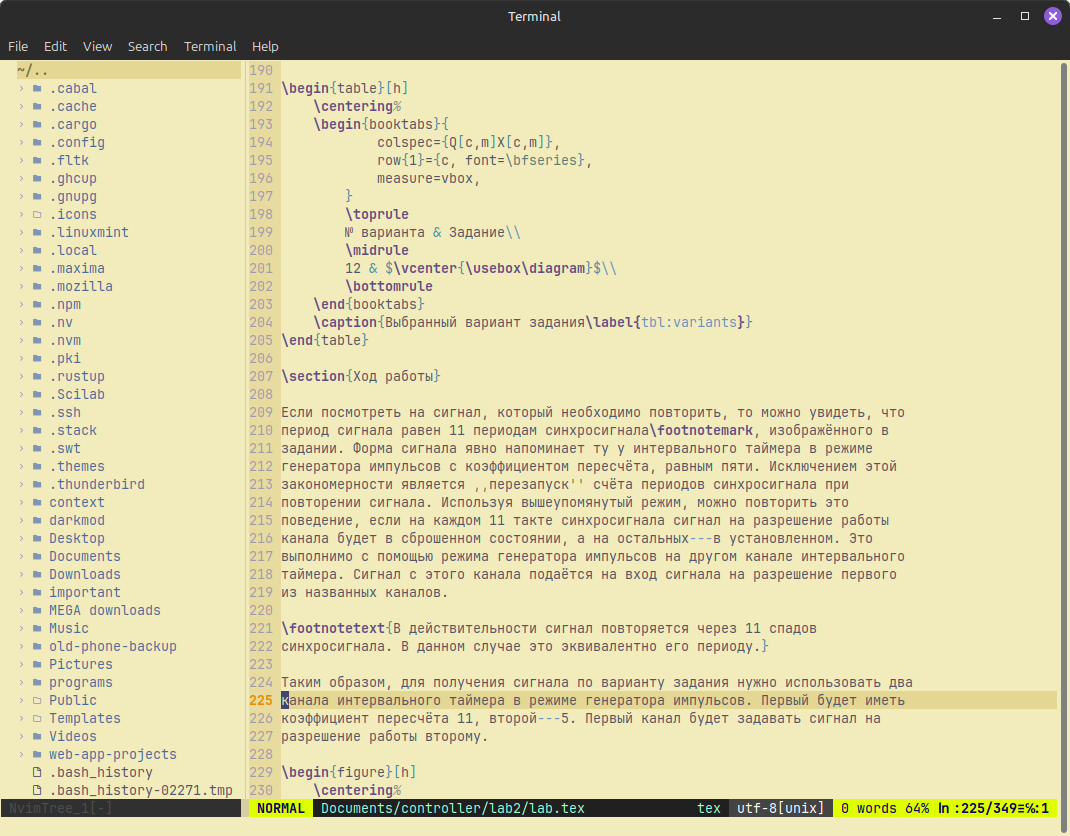


Рисунок 10 — Исходный файл некой лабораторной работы в LaTeX

Наиболее продвинутым вариантом LaTeX является LuaLaTeX, использующий систему компьютерной вёрстки LuaTeX. [\in[item: lualatex]] Наиболее важным нововведением является встроенный интерпретатор Lua с API для взаимодействия с документом. Представляется возможным составить пакет для LuaLaTeX, использующий, например, библиотеку LPeg [\in[item: lpeg]] для реализации парсера.

* SILE. Эту систему вёрстки можно назвать современником TeX. Во многом повторяет функционал TeX и LaTeX, но также поддерживает Lua. SILE способен принимать файлы на собственном языке аналогично TeX/LaTeX, или же в формате XML. Не обладает широкой поддержкой ввиду своей новизны. Как и с LaTeX, представляется возможным использовать Lua для написания пакета.
* ConTeXt. Система вёрстки, которая разрабатывается параллельно с LaTeX. В отличие от LaTeX, который стремится избавить пользователя от необходимости настраивать стиль документа (если не использовать отдельные пакеты и шаблоны), ConTeXt пытается дать пользователю полный контроль над стилем собираемого текста. ConTeXt стремится быть функционально полным решением, замещая собой всё то, для чего бы в LaTeX пришлось использовать пакеты. ConTeXt имеет также и основную часть функционала LaTeX. К сожалению, всё это делает эту систему сложнее для изучения и использования. [\in[item: not so short context]] К счастью, помощь можно найти в официальной почтовой рассылке [\in[item: context mail]], в многочисленных справочниках в инсталляции и на сайте разработчика [\in[item: pragma ade]] и на официальной вики [\in[item: context garden]].

ConTeXt также построена вокруг LuaTeX (самые новые версии используют LuaMetaTeX) и в действительности эта система обладает более гибким API в Lua для работы с документом. Также ConTeXt включает вышеупомянутый пакет LPeg в инсталляции, что делает составление модуля проще. [\in[item: luametatex]]

После рассмотрения текстовых процессоров было решено, что целесообразнее выбрать одно из решений группы WYSIWYM. Предпочтение было отдано ConTeXt вместо LaTeX из-за наличия библиотеки LPeg и более гибкой настройки структуры документа, что лучше подходит для академических целей.

* 1. ConTeXt и Lua: CLD

Типичный пользователь ConTeXt будет составлять входные файлы в формате .tex, использующий команды и макросы TeX и ConTeXt. Они начинаются с обратной наклонной черты. Некоторые из этих команд идут парами — например, \starttext и \stoptext позволяют начать видимую часть документа и завершить её соответственно. Аргументы командам могут передаваться в фигурных скобках. [\in[item: not so short context]]

Выполнить произвольный фрагмент кода Lua можно с помощью \startluacode и \stopluacode помимо других команд. Lua замещает ConTeXt там, где его предметно-ориентированный язык недостаточен. Перейдя к выполнению Lua, можно выдавать команду системе вёрстки посредством интерфейса CLD (англ. *ConTeXt Lua Document*, Lua-документ ConTeXt). Через этот интерфейс практически каждая команда/макрос имеет эквивалент в виде функции Lua, содержащейся в таблице context. Так, можно воспользоваться командой \mathematics или функцией context.mathematics(…) для перехода в режим вёрстки формул. \in[item: cld]

При обработке файла формата .tex возможно подгружать дополнительные файлы разными способами. Основной интерес представляют механизм модулей и так называемые файлы среды. Первый позволяет создавать и использовать полноценные модули, которые можно использовать в любом проекте; это наиболее правильный способ сделать модуль для ConTeXt. Второй способ больше подходит для хранения настроек отдельного документа, от стилей до функционала; файлы среды проще составлять и тестировать, поэтому лучше подходят для подтверждения работоспособности кода.

На рисунке 11 представлен пример входного файла .tex. Можно увидеть размещение текста в \starttext … \stoptext и несколько команд управления шрифтом как через сами команды TeX, так и через интерфейс CLD. Результирующие фрагменты текста будут эквивалентны по стилю. Больше про команды можно найти в приложении А.

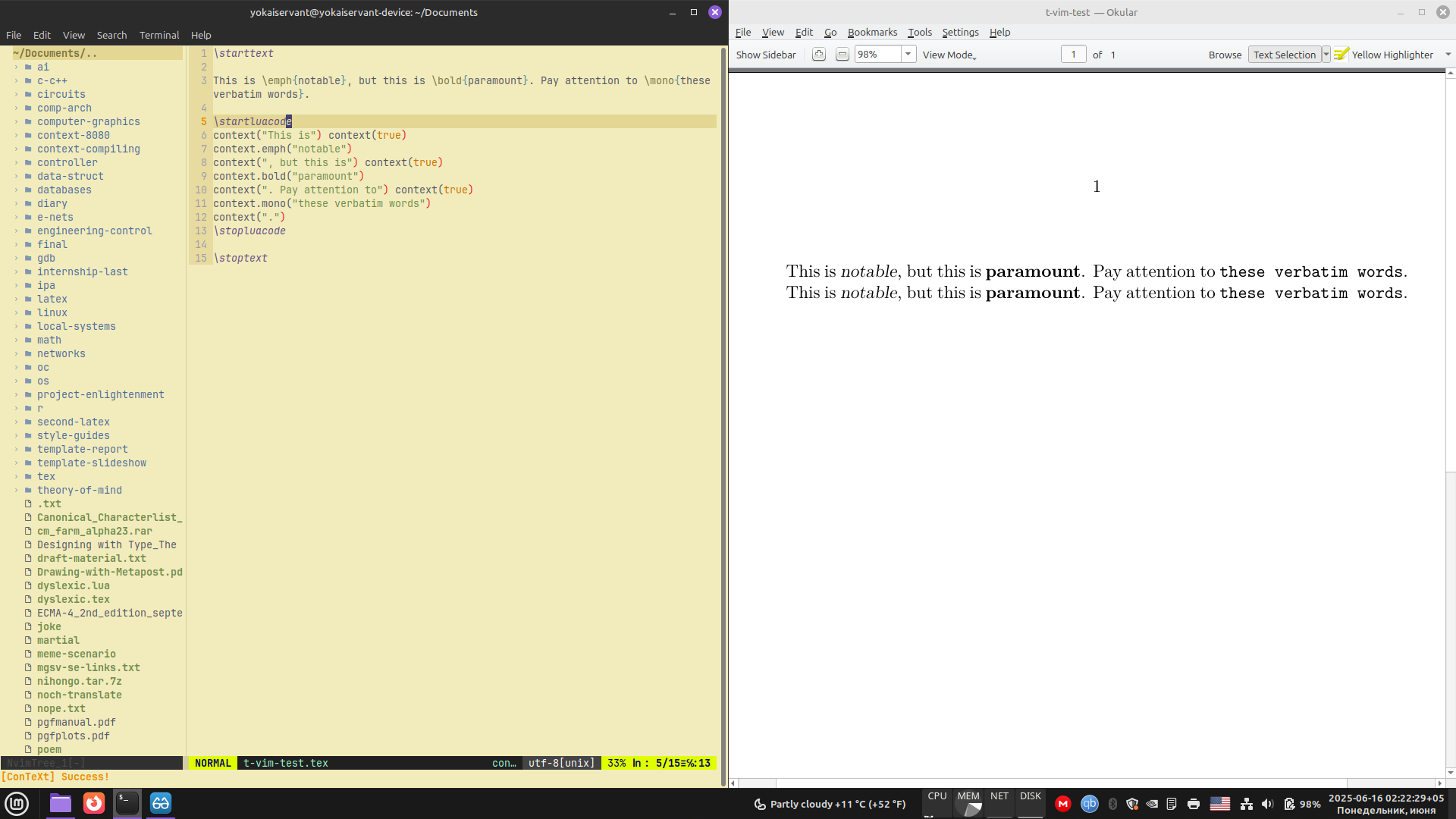


Рисунок 11 — Слева представлено два способа составить один и тот же текст (справа). Сперва используются команды TeX, затем используется CLD

* 1. Общая структура модуля

Любой модуль Lua представляет собой таблицу (структуру данных), содержащую в себе функции, переменные и так далее. Lua широко использует таблицы для хранения гетерогенных данных, аналогично структурам из языка программирования C, но при этом по функционалу таблицы Lua являются объединением хеш-таблиц и динамических массивов. Это позволяет использовать как непрерывные сегменты данных, так и ассоциативную память. Наиболее логичным представляется разбить генератор листинга на три (четыре, если учесть вспомогательные данные) части, представимые в виде таблиц и располагаемые внутри одной большой таблицы intel8080:

* Элементы парсинга. Таблица intel8080.parsing будет содержать объекты LPeg, используемые для обработки ассемблерного кода. Для составления полноценного парсера имеет смысл начать с простых элементов (регистры, регистровые пары, численные значения и т. д.) и строить более сложные последовательности из них, используя операции РВ-грамматики. Стоит также отметить, что Intel 8080 обладает достаточно большим количеством команд, поэтому целесообразно сгенерировать объект LPeg из некой структуры данных, содержащей все команды. Для этого была создана отдельная таблица intel8080.langdef, в которой была размещена таблица команд intel8080.langdef.opcodes. С помощью локальной функции generate\_cmds эта таблица обрабатывается и собирается парсер для обработки команд intel8080.parsing.cmd. Присутствуют и функции, преобразовывающие вывод объектов LPeg для удобной работы с ним, прежде всего для создания АСД, по которому можно составить листинг. Главным парсером является intel8080.parsing.listing, обрабатывающим весь исходный файл. Он зависит от intel8080.parsing.line для обработки строки и так далее.
* Функции для ассемблирования. Содержимое таблицы intel8080.assembly ответственно за вычисление результатов выражений и дальнейшее ассемблирование кода. Уникальным является то, что результатом ассемблирования будет дополненное машинным кодом и адресами АСД. intel8080.assembly.assemble выполняет ассемблирование и зависит от функции вычисления выражения intel8080.assembly.evaluate\_expression. Она использует ряд функций, расположенных в таблицах intel8080.assembly.binary\_operation и intel8080.assembly.unary\_operation, для применения операций к АСД выражения.
* Функции для форматирования. В данной части модуля будут присутствовать функции для форматирования отдельных элементов листинга и всего листинга в целом. Здесь можно найти функции intel8080.formatting.format\_expression и intel8080.formatting.make\_listing для форматирования выражения из АСД и форматирования листинга из представленного строкой кода соответственно.

Структуру модуля можно увидеть в виде блок-схемы на рисунке 12. Части модуля образуют одну большую цепочку зависимостей от парсинга до форматирования.

Рисунок 12

Планируется разработка следующей части модуля intel8080.settings, позволяющей настроить отдельные элементы модуля — например, функцию форматирования выражений.

Для демонстрации достижения цели минимальным функционалом был запланирован простейший интерфейс взаимодействия с модулем, предполагающий прямой вызов функции конструирования листинга intel8080.formatting.make\_listing через блок выполнения кода Lua (\startluacode … \stopluacode). Функция расположена в таблице для функций форматирования. Она будет принимать исходный код в виде строки, а также начальное значения счётчика позиции (адреса) и заранее известные метки. Функция выполняет ассемблирование и выдаёт листинг на месте вызова функции.

На данный момент вплоть до стабилизации интерфейса модуля тот будет представлен как один файл среды, основная масса которого является кодом Lua между \startluacode … \stopluacode.

* 1. Структура данных о командах ассемблера

Первым делом было предпринято составить таблицу команд intel8080.langdef.opcodes. Такая таблица должна отражать название команды, первый её байт и возможные аргументы. Так как соответствие между первым байтом и частью аргументов однозначно — например, LXI D, d8 в любом случае будет иметь первый байт 1116. Это вне зависимости от того, чему равняется байт d8, так как эти данные определяют значения следующих байт. Исключением являются команды системных переходов RST n. Несмотря на то, что n представляет собой аргумент команды в ассемблерном коде, он никак не отражается в последующих байтах команды. В этом плане аргумент можно назвать «псевдоаргументом» или «фальшивым аргументом». Этот аргумент выполняет выбор между разными командами, т. е. влияет на первый бит. Аргумент должен быть выражением, значение которого находится от нуля до семи включительно.

В связи с этим было решено использовать название команды как ключ к таблице, по которому можно получить все команды с данным названием, но отличающимися аргументами и первыми байтами. Это выгодно, так как программа ориентирована на ассемблерный код, опирающийся на названия команд. Пример такой таблицы приведён в таблице 2. Приведён лишь фрагмент для наглядности.

Таблица 2 — Использование названия команды как ключа представляется более наглядным и правильным с точки зрения соответствия между названием команды и возможными первыми байтами

|  |  |  |
| --- | --- | --- |
| Ключ (название команды) | Первый байт команды | Аргументы |
| PUSH | C5 | Регистровая пара B |
| D5 | Регистровая пара D |
| E5 | Регистровая пара H |
| F5 | Регистровая пара PSW (слово состояния) |
| MVI | 06 | Регистр B, байтовое значение |
| 16 | Регистр D, байтовое значение |
| 26 | Регистр H, байтовое значение |
| RST | C7 | Фальшивый аргумент в виде численного значения. Должно быть не меньше нуля и не больше семи |
| CF |
| D7 |
| DF |

Это задаёт очевидное соответствие между именем команды и первыми байтами, что облегчает проверку команды RST. Если как ключ бы использовались первые байты, то искать варианты RST было бы тяжелее. Вероятно, что пришлось бы делать дополнительную структуру данных.

* 1. Иерархия парсеров

Наиболее базовыми можно назвать выражения для обработки отдельных элементов ассемблерного кода. Для пустого пространства между полями строки был выделен парсер intel8080.parsing.blank. Имеет смысл сделать так, чтобы он определял не только пробелы, но и табуляцию и возможно даже пробелы разных размеров, что есть в Unicode. Разделитель операндов (запятая) представляется парсером intel8080.parsing.arg\_sep. Двоеточие после метки определяется intel8080.parsing.label\_delim, и аналогично точка с запятой для начала комментария определяется intel8080.parsing.comment\_delim. Для новой строки был выделен парсер intel8080.parsing.newline. Все эти парсеры объединяет то, что ими полученное значение положения этих символов или групп символов не представляет пользу. Важно то, что эти символы разделяют собой.

Далее желательно определить парсеры для простых аргументов. Для регистров и регистровых пар были созданы таблицы intel8080.parsing.register и intel8080.parsing.register\_pair. Было решено, что команды и аргументы к ним могут записываться как строчными, так и прописными буквами, но регистр (прописные и строчные буквы) не может чередоваться в одном элементе кода (команде или аргументе). Для упрощения работы имеет смысл сразу преобразовать все строчные буквы в полученном регистре (регистровой паре) в большие, чтобы случайно не допустить ситуации, где прописные (строчные) буквы допускаются, но не их противоположность по регистру.

Численные значения будут представляться как таблицы с полем для значения в некой системе счисления и полем для хранения самой системы счисления. Это позволяет избежать предположений о системе счисления. Для этого было составлено четыре парсера intel8080.parsing.numeric.bin, intel8080.parsing.numeric.oct, intel8080.parsing.numeric.dec, intel8080.parsing.numeric.hex для двоичных, восьмеричных, десятичных и шестнадцатеричных значений соответственно. Получаемый результат преобразовывается в вышеописанные таблицы.

Обработка меток и счётчика позиции происходит с помощью парсеров intel8080.parsing.label и intel8080.parsing.location\_counter соответственно. В частности, метки будут допускать не только латиницу и цифры, а также подчёркивание и дефис.

Для обработки выражений необходимо иметь возможность определить каждую из операций. Для этого в intel8080.parsing.operation были помещены парсеры для каждого из возможных операторов — логических конъюнкции и дизъюнкции, сложения и вычитания, умножения и вариантов деления и так далее. Операторы, записываемые словами (MOD для остатка деления, SHR для сдвига вправо и множество других), обрабатываются аналогично регистрам — их возможно записать всеми прописными или всеми строчными, но после получения оператора все буквы преобразовываются в прописные.

Наиболее сложными для парсинга являются выражения. Математические выражения в естественном виде имеют рекурсию, так как операндами могут выступать выражения в скобках, что сами могут иметь операнды в виде выражений. Вложенность является практически неограниченной. Необходимым представляется реализовать парсер intel8080.parsing.expression.rule с рекурсией. Порядок рассмотрения выражения в соответствии с правилами языка ассемблера показан в таблице 3. В действительности порядок отличается от истинного тем, что логическое отрицание теперь обладает наибольшим приоритетом. Это лучше соответствует устоявшемуся в современных языках программирования порядку выполнения операций.

Каждый уровень выражения представляет собой таблицу, содержащую упорядоченные список операций одного уровня и список операндов. Операнды сами являются таблицами подобного вида или же неразделимыми операндами, также хранящимися в таблицах — числами, метками и т. п. В результате получается полноценное АСД математического выражения. Так, для выражения такая таблица будет иметь вид наподобие {операнды = {7, {операции = {\*}, операнды = {5, 3}}}, операции = {+}}. Такую структуру несколько тяжело читать человеку, но легко обойти рекурсивной функцией, что и планируется.

Таблица 3 — Можно заметить, как уровни ниже определяют вид операндов уровней выше. Рекурсия задаётся на последнем уровне, где операндом может являться вложенное выражение, возвращая обработку на первый уровень

|  |  |  |
| --- | --- | --- |
| Уровень (равно приоритетность операции) | Операция, операнды | Пример |
| 1 — выражение | Дизъюнкции (ИЛИ, исключающее ИЛИ), дизъюнкты | 0150H XOR 7FFFH |
| 2 — дизъюнкт | Конъюнкция (И), конъюнкты | 15AFH AND 00FFH |
| 3 — конъюнкт | Сравнение (равенство, неравенство, меньше, больше и др.), сравниваемые значения | $ EQ 0900H |
| 4 — сравниваемое значение | Сложение/вычитание, слагаемые | $ + 10D |
| 5 — слагаемое | Произведение/деление и близкие к ним операции (сдвиги влево/вправо, остаток деления), значения | result SHL 2 |
| 6 — значение | Унарные операции (взятие старшего/младшего байтов, логическое отрицание), численные значения/значения меток/значение счётчика позиции/выражение в скобках | NOT 3DH |

Не менее сложным представляется составление парсера команд. К счастью, его можно сгенерировать исходя из таблицы команд intel8080.langdef.opcodes с помощью локальной функции generate\_cmds. Генерация происходит наиболее простым образом: для каждой вариации некой команды INS составляется отдельный парсер, ожидающий аргументы конкретного вида (некий регистр, выражение и так далее). Отдельно обрабатываются фальшивые аргументы. Затем возвращается парсер в виде упорядоченного выбора всех полученных парсеров вариаций команд. Результат этой функции помещается в intel8080.parsing.cmd. Несмотря на неэффективность алгоритма с точки зрения выполнимых действий, заметного влияния на производительность обнаружено не было.

Последним конструируются парсеры строки и целого листинга. Первый просто пытается получить необязательные метку, команду с операндами и комментарий. Результат обработки строки помещается в таблицу с полями для меток, команд и комментариев соответственно. Листинг, соответственно, представляется как множество строк, разделённых переходами на новую строку. АСД листинга выглядит как последовательность АСД каждой из строк. Запрет на пустые строки был снят, что полезно для разбиения кода на части по смыслу.

На рисунке 13 можно увидеть дерево, демонстрирующее цепочки зависимости сложных парсеров от простых.

Рисунок 13

* 1. Ассемблирование кода

Прежде, чем использовать АСД листинга для ассемблирования, необходимо рассчитать значения аргументов команд. Для этого нужна функция высчитывания выражений intel8080.assembly.evaluate\_expression. Та в свою очередь зависит от таблиц функций, задающих бинарные и унарные операции intel8080.assembly.binary\_operation и intel8080.assembly.unary\_operation соответственно. Те в свою очередь используют функцию intel8080.assembly.to\_unsigned\_word для ограничения переданного численного значения к беззнаковому двухбайтовому целому числу (от 0 до 65535 включительно). Возможным является перезаполнение. Так как численные значения, возвращаемые парсером, представляют собой таблицу с полями для значения и системы счисления, также имеется функция intel8080.assembly.match\_to\_number для перехода от такой таблицы к простому численному значению.

intel8080.assembly.evaluate\_expression получает АСД выражения, текущее значение счётчика положения и известные метки. Она по рекурсии проходит все операнды, выполняя операции над ними, пока операнды не будут неразделимыми (численные значения, метки и прочее). Результатом является число, полученное как результат вычисления выражения.

intel8080.assembly.assemble получает ассемблерный код в виде строки, а также первоначальное значение счётчика позиции и известные метки. Выполняется парсинг с помощью intel8080.parsing.listing, после чего начинаются обходы результирующего АСД. Как и официальный ассемблер, данная функция выполняет два обхода кода. На первом выясняются значения меток, размещённых в коде. На втором выполняется с помощью предыдущей функции находятся значения выражений для нахождения остальных байт команд (первого байта в случае RST) и, соответственно, формирования машинного кода. Машинный код размещается в АСД, являющимся результатом ассемблирования.

* 1. Форматирование результата

Для форматирования было составлено две функции: intel8080.formatting.format\_expression для форматирования выражений, полученных в виде АСД, и intel8080.formatting.make\_listing для генерации листинга. Первая получает лишь АСД как аргумент, так как для форматирования это является достаточным. Она работает аналогично intel8080.assembly.evaluate\_expression, только конструирует строку, эквивалентную полученному АСД.

Вторая получает ассемблерный код в виде строки, а также начальное значение счётчика позиции и известные метки. Она выполняет ассемблирование с помощью intel8080.assembly.assemble, после чего начинает обработку АСД программы. Так как строк без команд в листинге в виде таблицы не может быть, все метки и комментарии на строках без самого кода программы собираются вместе и добавляются к первой строке, на которой есть команда. Преобразованное АСД обходится для выполнения вывода в документ во время его обработки. Для этого используется функционал интерфейса CLD.

1. Написание модуля

В данном разделе будет опущен немногочисленный код на TeX для перехода к Lua. Далее демонстрируется лишь код на языке Lua.

* 1. РВ-грамматика и LPeg

Как было сказано в подразделе 2.1, ConTeXt идёт с библиотекой LPeg для составления парсеров на основе грамматики, разбирающей выражения (РВ-грамматики). РВ-грамматика представляет собой альтернативу контекстно-свободной грамматике (КС-грамматике), часто используемой для определения языков программирования. В действительности, РВ-грамматика и КС-грамматика стилистически похожи. Первая имеет практически те же операторы, что и вторая, а также имеет операторы регулярных выражений. [\in[item: peg roberto], \in[item: peg bryan]]

РВ-грамматика предлагает следующие операторы для формирования выражений:

* Создание правила (стрелка влево). Позволяет присвоить имя выражению. E ← «foo» создаст правило E, соответствующее строке «foo».
* Последовательность. Сперва применяется одно выражение, затем следующее. A B соответствует поочерёдному применению правил A и B.
* Упорядоченный выбор (косая черта). Перебор выражений/правил, объединённых этой операцией, пока одно из них не будет соответствовать строке. «foo» / «bar» пытается сопоставить «foo» с проверяемой строкой, и если это не удаётся, то это повторяется с «bar».
* Образование класса символов (квадратные скобки). [А-Я] представляет собой любой один символ от заглавной А до заглавной Я. [2-5] представляет собой одну оценку, которую могут поставить в российских школах — «2», «3», «4» или «5».
* Формирование произвольной последовательности символов (кавычки). «int» и «for», например, являются такими строками и соответствуют указанным символам в указанном порядке.
* Совпадение с любым символом (точка).
* Обыкновенные скобки для группирования операций.
* Необязательное совпадение (вопросительный знак) — «else»? соответствует нулю или одному появлению строки «else».
* Повторение выражения от нуля и более раз (астериск). [а-я]\* соответствует любой строке из строчных букв кириллицы произвольной длины, включая пустую строку.
* Повторение выражения от одного и более раз (плюс). [а-я]+ соответствует любой строке из строчных букв кириллицы произвольной длины, но не пустую строку.
* Предусловие И (амперсанд) — пытается сопоставить операнд с проверяемой строкой, не поглощая его. &[1-9] [0-9]+ сперва проверяет строку на то, чтобы она начиналась с ненулевой цифры, но не трогает её. При соответствии применяется следующая часть выражения, соответствующая любой цифре.
* Предусловие НЕ (восклицательный знак). Аналогично предыдущему, но проверяет на несоответствие. Предыдущий пример можно переписать как !0 [0-9]+.

Основное отличие от КС-грамматики заключается в том, что оператор выбора является упорядоченным, а не произвольным. Это удобно, если работа ведётся над достаточно простыми языками. Также РВ-грамматика обеспечивает лишь локальный поиск с возвратом при выборе — как только одно из правил в выборе подошло, нельзя будет попробовать непроверенные правила в случае дальнейших провалов при сопоставлении. [\in[item: peg roberto]] Как бонус, все РВ-грамматики можно разобрать в линейном времени. [\in[item: peg bryan]] Поэтому её применение имеет смысл для несложных языков программирования. В частности, Python использует парсер на основе РВ-грамматики. [\in[item: python parser]]

LPeg позволяет составлять парсеры на основе РВ-грамматики. Эта библиотека предоставляет функции для создания и объединения последовательностей (правил), а также составления грамматик, поддерживающих рекурсию. Больший интерес представляют захваты — последовательности, возвращающие некие значения (семантическую информацию) в зависимости от того, к чему последовательность подошла. В особенности важны захваты с группированием и табличные захваты. [\in[item: lpeg]] Они позволяют формировать АСД из захваченных строк, что необходимо для ассемблирования и построения листинга. Находят применение и захваты с обработкой функцией.

* 1. Данные о командах ассемблера

Сперва была вручную составлена массивная таблица всех команд, intel8080.langdef.opcodes. На рисунке 14 можно увидеть её фрагмент. Как видно, на каждую команду есть несколько записей в виде таблиц с полями cmd для первого байта конкретной вариации и args для аргументов. args представляет собой таблицу, содержащую таблицы, каждая из которых имеет лишь одно поле. Если это register, то в этой позиции ожидается регистр, чьё имя хранится в поле. Аналогично с register\_pair (регистровые пары). Если ожидается произвольный байт, то будет поле byte. Его значение не используется (лишь проверяется на неравность nil), но «data» в byte соответствует байту данных, «address» — номеру порта. Подобное выполняется и для аргумента размером в слово (два байта); будет наблюдаться поле word. Всю таблицу целиком можно увидеть в приложении В.

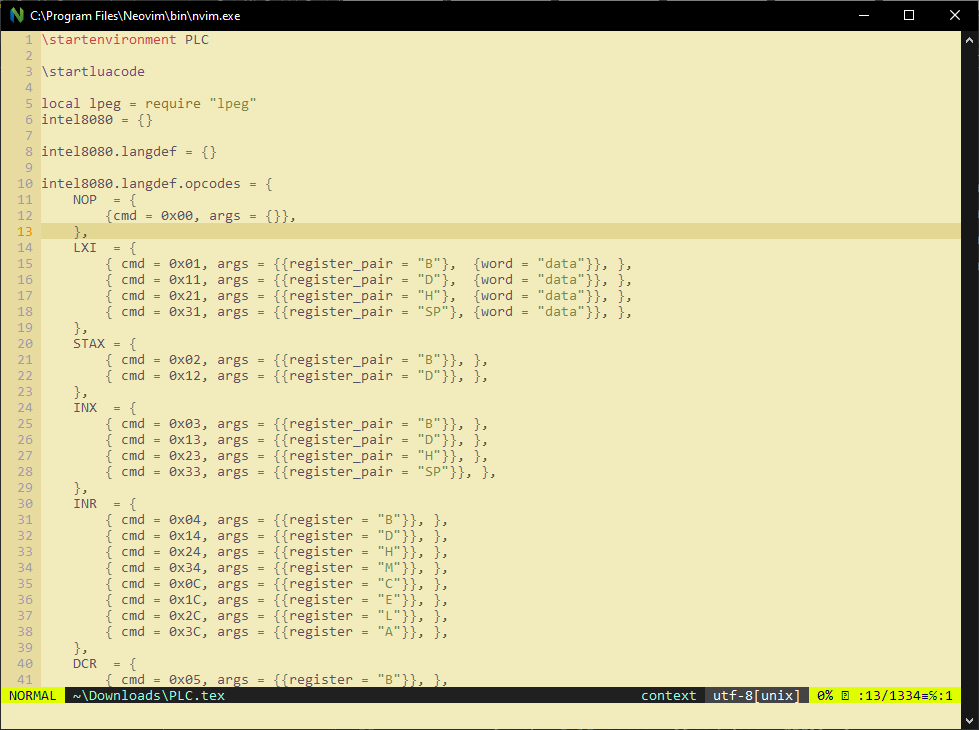


Рисунок 14 — cmd соответствует первому байту команды, args содержит аргументы

* 1. Парсинг ассемблерного кода

Для парсинга понадобится LPeg. Сперва были определены парсеры наиболее простых элементов ассемблерного кода (рисунок 15). lpeg.P является наиболее простой из доступных функций составления парсеров. Когда её аргумент является строкой, получается парсер, пытающийся сопоставить текст с аргументом. lpeg.S позволяет выполнить сопоставление с множеством символов (любым символом из переданной строки). В данном случае используется встроенная библиотека utf8, с помощью которой можно получить строку из символов, имея конкретные кодовые точки. Очевидно, что 2016 соответствует пробелу, 916 — символу горизонтальной табуляции. Это было сделано так, потому что среды разработки могут заменить табуляцию на множество пробелов.

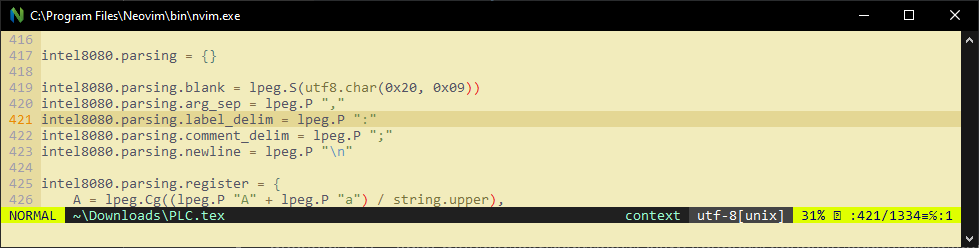


Рисунок 15 — lpeg.P позволяют составить самые простые правила сопоставления. lpeg.S подходит для выбора из множества символов

Далее были определены парсеры регистров и регистровых пар, как видно на рисунке 16. С помощью упорядоченного выбора (сложение объектов lpeg.P) были получены парсеры, способные распознать регистры и регистровые пары полностью прописными и полностью строчными буквами, но не вперемешку. Захваченное значение передаётся функции string.upper для переключения регистра букв на верхний. Операция деления в данном случае обозначает захват с обработкой функцией. Это инструмент LPeg, не формальный элемент РВ-грамматики.

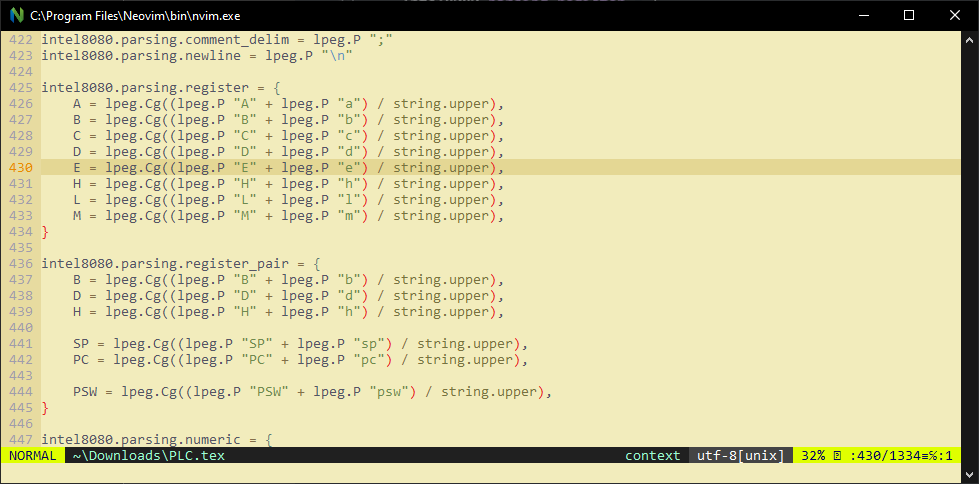


Рисунок 16 — lpeg.Cg выполняет групповой захват данных, полученных от парсера внутри

Наконец, lpeg.Cg выполняет групповой захват, возвращающий объединение всех полученных внутри значений. В данном случае это используется для избежания странного дублирования считанного регистра (если был передан регистр H как «h», то парсер мог вернуть «hH»). Вероятно, что это ожидаемое поведение, но документация не наводит на причину возникновения проблемы.

Далее составляется парсер численных значений, как показано на рисунке 17. Для каждой из систем счисления есть свой отдельный парсер. lpeg.R позволяет искать символ между первым и вторым символами переданной строки включительно — иными словами, это инструмент обработки символа в диапазоне. Функция n-арна — можно передать несколько таких строк-диапазонов. Возведение в положительную степень означает повтор от указанного числа раз. Здесь lpeg.Cg используется в иных целях. lpeg.Ct собирает результаты работы всех групповых захватов lpeg.Cg и помещает их в таблицу. Если у lpeg.Cg есть второй аргумент, то результат будет помещён в поле с этим именем. Так, получится таблица с полями value и radix, где value соответствует значению, radix — системе счисления. Результаты неименованных групп помещаются в нумерованные позиции в порядке появления. Это будет полезно при обработке всего листинга.

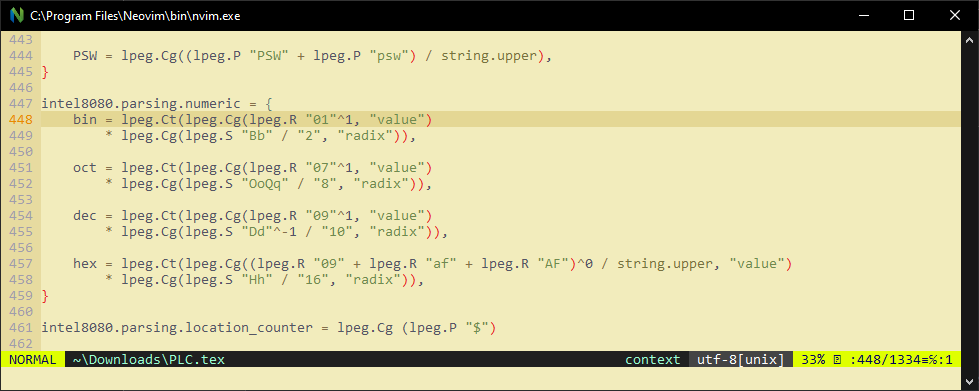


Рисунок 17 — При получении численных значений был применён табличный захват lpeg.Ct. Для его использования необходим lpeg.Cg

Стоит отметить использование деления, но не с функцией, а со строкой. Это строковый захват — захваченное значение заменяется на эту строку. lpeg.Ct и lpeg.Cg — основа для получения АСД.

Следующие парсеры — парсеры для определения метки и счётчика позиции (рисунок 18). Как видно, счётчик позиции не требует многого.

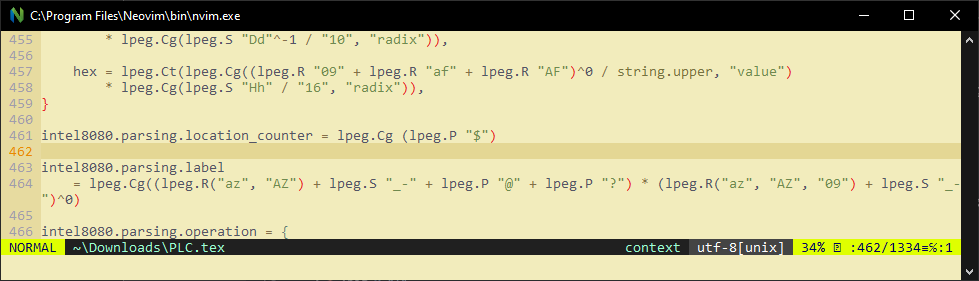


Рисунок 18 — Парсеры счётчика позиции и метки

В составе же метки можно увидеть, что первым символом может быть символ at или вопросительный знак. Они необходимы для функционала из официального ассемблера, который не будет добавляться в модуль. В дальнейшем парсер метки будет упрощён. Также необходимо отметить наличие оператора последовательного применения (астериск, «звёздочка»).

Для используемых в выражениях операциях была составлена таблица парсеров для каждого из операторов, как показано на рисунке 19. Здесь не применяются ранее не продемонстрированные функции LPeg или подходы, поэтому имеет смысл перейти к обработке выражений.



Рисунок 19 — Таблица содержит парсеры на все имеющиеся операции

Для обработки выражений был составлен парсер intel8080.parsing.expression.rule. Его фрагмент приведён на рисунке 20.

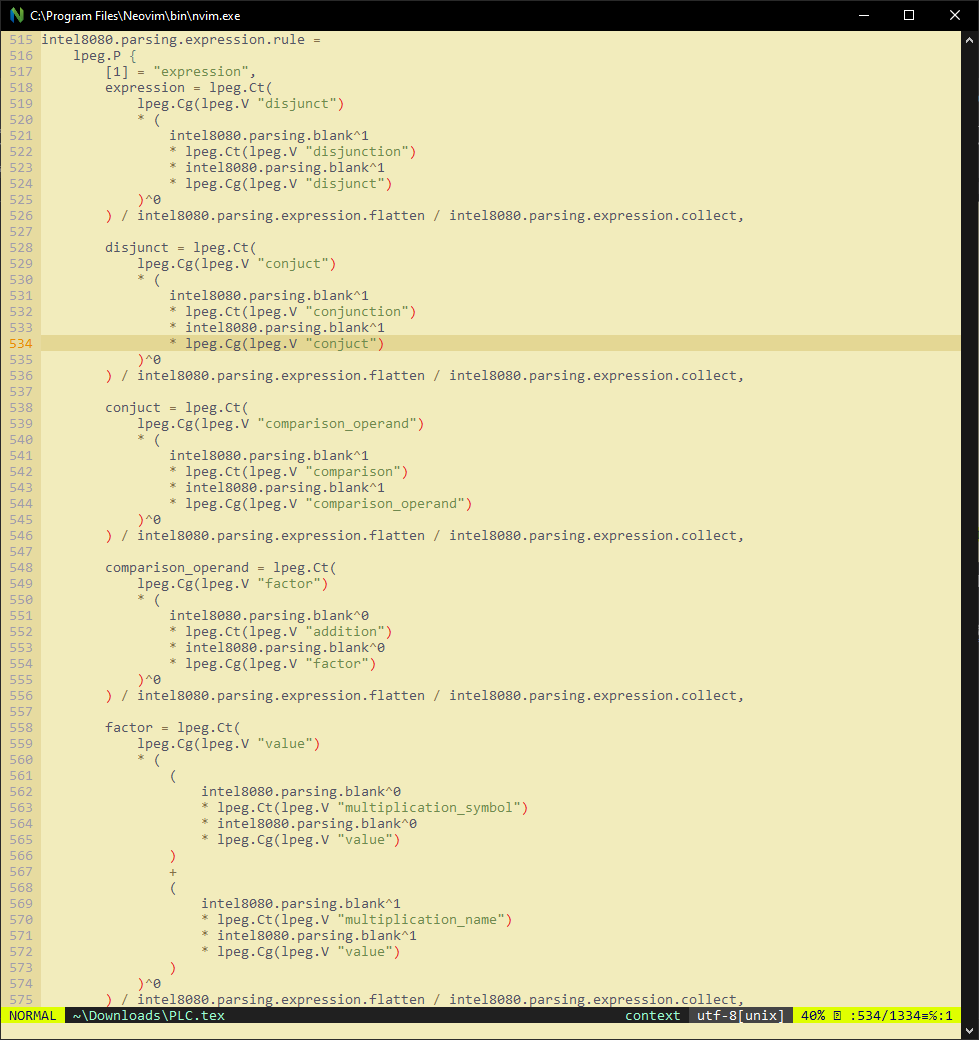


Рисунок 20 — Объекты грамматики позволяют представлять правила с рекурсией

В данном случае lpeg.P используется для создания объекта грамматики. В отличие от созданных до этого объектов lpeg, грамматики способны обрабатывать рекурсию в правилах. Для составления грамматики необходимо передать lpeg.P таблицу с правилами. lpeg.V можно использовать для того, чтобы сослаться на правила внутри самой грамматики. Это позволяет пользоваться правилами, которые технически ещё не были созданы (таблицы создаются целиком, поля не знают друг о друге).

Каждое правило представленной на рисунке 20 грамматики представляют собой один из уровней выражения (см. подраздел 2.5). Каждый уровень использует правило, определяющий уровень ниже. Так происходит вплоть до правила, определяющего базовые значения — числа, метки, счётчик позиции, а также выражение в скобках. В последнем и заключается рекурсия.

Парсер выражений широко использует табличные захваты lpeg.Ct и групповые захваты lpeg.Cg для преобразования выражения в АСД. АСД выражения тяжело визуализировать, но возможно описать. Непосредственно значение представляется как таблица Lua, имеющая лишь одно поле. По ключу можно определить тип значения — от чисел до выражений в скобках. Обвернув данную таблицу ещё таблицей, представляется возможным дополнительно хранить применённые унарные операции. Несколько таких конструкций, размещённых в массиве, а также операции, размещённые в отдельном массиве, позволяют описать операции на одном уровне приоритета, что есть выражение.

К сожалению, уровни, операции которых не присутствуют, добавляют АСД чрезмерную вложенность таблиц. Также представляется возможным воспользоваться фактом того, что все операции в ассемблерном языке Intel 8080 лево ассоциативны. Это позволяет собрать все операции и операнды на одном уровне в отдельные и удобные массивы, как описано ранее. Для этого были определены функции intel8080.parsing.expression.flatten и intel8080.parsing.expression.collect соответственно. Их применение к уровням видно на рисунке 20. Определение представлено на рисунке 21.

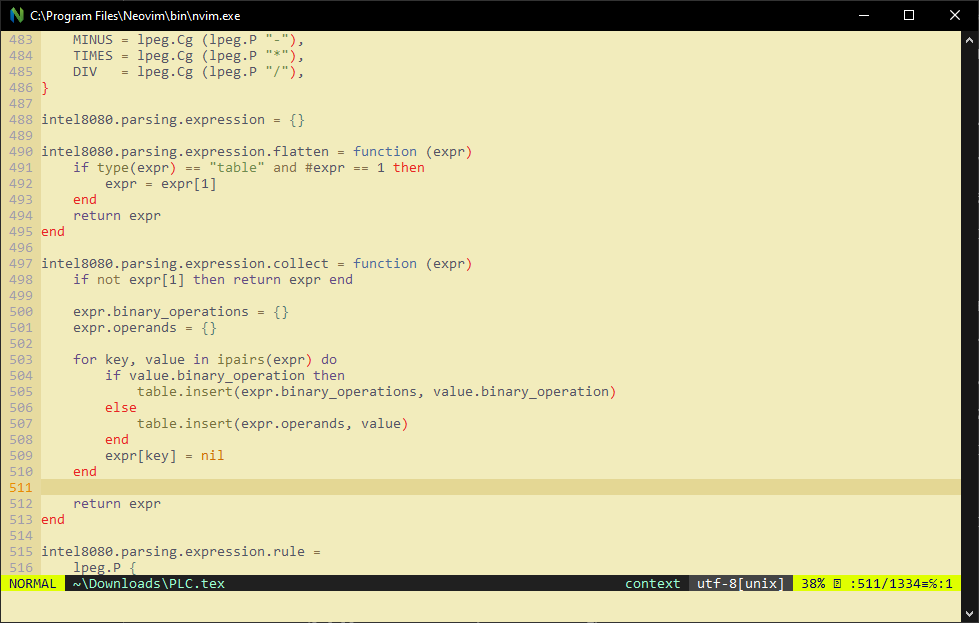


Рисунок 21 — flatten «сплющивает» АСД, collect «собирает» операнды и операции

Далее необходимо получить парсер команд. Для этого была составлена локальная функция

* 1. Процесс ассемблирования
  2. Форматирование результата

1. Тестирование программного решения
   1. Парсер чисел
   2. Парсер выражений
   3. Ассемблер и генератор листинга
2. Сопровождение ПО
3. Заключение
4. Conclusion
5. Список использованных источников
6. Характеристика текстового процессора ConTeXt
7. Команды микропроцессора Intel 8080
8. Листинг модуля