Государственное бюджетное общеобразовательное учреждение

города Москвы "Школа № 2098 "Многопрофильный образовательный центр" имени Героя Советского Союза Л.М. Доватора"

Конференция «Инженеры будущего»

**Проект**

**Написание языка программирования**

Выполнил:

Кононцов Леонид, 10 Г класс

Руководитель команды:

Кононцова Яна Сергеевна,

учитель информатики и ИКТ, 89680348581, kononcova@yandex.ru

Москва, 2019

**Содержание**

[Введение 3](#_30j0zll)

[Основные этапы работы 4](#_3znysh7)

[Этап 1.Дизайн языка 4](#_2et92p0)

[Этап 2.Разработка парсера 4](#_tyjcwt)

[Этап 3.Обход дерева 5](#_3dy6vkm)

[Заключение 6](#_1t3h5sf)

[Список используемой литературы и интернет ресурсов 7](#_4d34og8)

# Введение

В мире существует уже много языков программирования, которые решают свои проблемы, например, на C пишет низкоуровневые программы, на Erlang используется для распределённых систем, Javascript для написания сайтов. Но, к сожалению, я не нашёл языка программирования, который я бы смог использовать для решения повседневных задач автоматизации. По этому я решил написать свой язык программирования на языке Clojure.

Цель: Написать синтаксис и создать интерпретатор языка собственного программирования.

Задачи:

* разработать удобный синтаксис;
* написать парсер и исполнитель языка;
* написать встроенную библиотеку;
* реализовать repl и запуск файлов.

## Этап 1. Дизайн языка

Сначала был придуман синтаксис языка. Выбор пал на lisp из-за своей простоты в освоении и частично из-за лёгкости реализации. В отличие от common lisp, мой язык программирования должен быть более однообразным, там не должно быть возможности добавить свой синтаксис, тем самым испортить его.

Также стоял вопрос о выборе встроенных типов данных. Во-первых, я решил не включать возможность написать свой тип данных, то есть нет возможности проверить типы данных. Во-вторых, список из встроенных типов данных должен быть максимально коротким, нельзя допустить, чтобы большое количество типов данных перегружало сложность языка. В итоге, на данный момент используется восемь обычных типов данных, а именно: number, bool, string, quotedlist, keyword, hashmap и fn (тип fn есть двух видов) и два необычных (word и list). Свойство этих типов данных, в том, что они “исполняются”, т.е., как только интерпретатор встречает word он заменяет его на какие-нибудь данные, которые находятся в памяти, если исполнитель видит тип list, то первый элемент списка считается функция, а все следующие аргументами. Также было принято решение о добавлении трёх, пока что нигде не используемых типа данных, int, float и ratio, они были добавлены для проверки типов на будущее.

## Этап 2. Разработка парсера

После придумывания синтаксиса шёл этап его исполнения. Для написания парсера было принято решение использовать парсер комбинаторы, эта технология показала себя с лучшей стороны, у неё есть много плюсов. Во-первых, в отличии от регулярных выражений, заниматься поиском ошибок в парсер комбинаторах на много проще, чем в регулярных выражениях, так как последние используют крайне странный синтаксис, больше похожий на кашу из символов. Что ещё хуже, чем больше становиться проект и чем сложнее становиться синтаксис, который необходимо обработать регулярные выражения растут в размере, после чего не то что ловить ошибки, но и просто смотреть на них становиться невыносимой мукой. Во-вторых, конечный автомат показался мне слишком громоздким, ситуация с ним на много лучше чем с регекспами, но всё ещё слишком некрасиво, и когда я узнал о парсер комбинаторах, я переписал весь свой парсер на них. В-третьих, можно было использовать всевозможные генераторы парсеров, они отличаются тем, что парсер генерируется из правил грамматики, которые описываются в специальном файле. Данный способ плох по двум причинам: во-первых, он генерирует не понятный или обфусцированный код, во-вторых, это особый DSL, тонкости которого надо знать. Таким образом превосходство парсер комбинаторов для этого проекта доказано. Для языка clojure, на котором я реализовывал интерпретатор, оказалась одна хорошая библиотека под названием kern. Любой может без труда найти её в интернете, либо по ссылкам в разделе Литература. У данной библиотеки оказалась очень хорошая документация и все функции для реализации языка. После парсинга возвращается хешмапа в которой гарантированно есть два поля :type и :value, то есть абстрактное синтаксическое дерево.

## Этап 3. Обход дерева

Исполнение языка - это просто проход по АСТ, который был сгенерирован с помощью парсер комбинатора. Проход по дереву осуществляется с помощью функции read-ast. Она принимает хешмапу, смотрит на тип данных, который обёрнут и если это не особый тип данных, просто отправляет копию объекта, иначе пытается исполнить то, что находится внутри хешмапы. К сожалению, нет никаких оптимизаций в моём интерпретаторе, т.к. у меня не достаточно опыта с этим.

Выше я говорил про то, что у меня два вида функций: встроенные и посторонние. Встроенные функции исполняются функцией execute-built-in-function. Выполнение посторонних функций состоит из нескольких этапов. Первый этап - подстановка, специальная функция проходит по дереву и подставляет вместо названия аргумента то, что передали в функцию, вставляется всё дерево, это значит, что значения не вычисляются сразу.

# Заключение

Самыми трудными этапами оказались: придумавыние синтаксиса и проход по дереву. Хотя это и всего три этапа, иногда, приходится делать необычные решения в дизайне языка, чтобы возможно было написание проходчика по дереву. С другой стороны, иногда нужен сложный код для того, чтобы функции не ломала старые.

# Список используемой литературы и интернет ресурсов

1. [https://github.com/blancas/kern](https://github.com/blancas/kern/) - библиотека, позволяющая использовать комбинатор парсеров
2. <https://github.com/blancas/kern/wiki> - документация библиотеки
3. [https://www.graalvm.org](https://www.graalvm.org/) - компилятор в нативный код
4. [https://leiningen.org](https://leiningen.org/) - система сборки проектов на языке clojure