Первая фаза
Вторая фаза

Участники:
Лебедев Егор
Худяков Юрий
Карнаухов Кирилл
Сурков Петр

Начнём с описания классов из которых будет состоять наше AST, на диаграмме они представлены в центре в блоке *Tree.* Главный абстрактный класс – *Command*. Это общий класс для любого поддерживаемого выражения в нашем баше. Его наследники – конкретные возможные варианты, а именно: *CatCommand* (содержит список файлов), *EchoCommand* (содержит список аргументов – строчек), *PwdCommand*, и т.д.

Теперь разберём класс *Grammar*. Он содержит правила для лексера и парсера, с помощью которых better-parse сможет лексить и парсить выражения, он наследуется от класса *Grammar<Program>* из этой библиотеки.
Про *Grammar:*

- описание правил лексера
  - *catToken* должен матчить “cat”
  - *pipeToken* должен матчить “|”
  - …
- описание правил парсера для термов
  - *catTerm* должен парсить команду *cat* с проивзольным числом аргументов
  - echoTerm должен парсить команду *echo* с произвольным числом аргументов
  - exitTerm должен парсить команду *exit* без аргументов *(хотя можно было бы указывать код возврата)*
  - …
- более высокоуровневые правила парсера
  - *term* принимает любой возможный конкретный терм, то есть *catTerm* и остальные
  - *pipeChain* должно строить дерево (почти бамбук) для цепочки пайпов

Но этот класс не будет использоваться напрямую в пайплайне. У нас будут некие обёртки. Лексер представим интерфейсом *Lexer* с единственной требуемой функцией – *tokenize(String): List<Token>,* то есть получать по строке список токенов. Реализовывать интерфейс *Lexer* будет *LexerAdapter* уже как раз используя *Grammar*. Аналогично для парсинга: интерфейс *Parser* требует единственную функцию – parse, которая по списку токенов возвращает верхний элемент AST – *Command*. Этот интерфейс будет реализовывать класс *ParserAdapter* опять же используя *Grammar.* Таким образом мы абстрагируемся от конкретной библиотеки для лексинга и парсинга, что позволит легче заменить её при необходимости или проводить тестирование.

Лексинг проводится в 2 этапа.
Во время первого этапа, мы проходимся по строчке. Если мы находимся не внутри одинарных кавычек, то при обнаружении строки $var_name смотрим в уже накопленный ранее Environment (о нём написано ниже). Если в нём есть нужная переменная, то мы заменяем эту строку на значение из Environment, иначе заменяем на пустую строку.
Во время второго этапа: разбиваем строку на токены в соответствии с правилами выше. Получили список токенов.

С кавычками будем делать следующее:

- Первый лексинг найдёт нам quoteToken (текст в одинарных кавычках) и doubleQuoteToken (текст в двойных кавычках)
- После чего мы посмотрим на содержимое doubleQuoteToken и там применим все подстановки (найдя все доллары внутри, можем ещё учесть экранирование, посчитав сколько перед долларом бэкслешей)
- Затем мы переведём набор токенов снова в текст, его лексим, парсим и далее

Маленькое уточнение: "the cat says 'meow'" – один большой doubleQuoteToken.

После этого список токенов передаётся в парсер, который строит дерево (что-то вроде бамбука в наших ограничениях) в соответствии с правилами, описанными выше.

Теперь обсудим, что делать с построенным деревом. У каждой команды есть *execute(context: IoContext, env: Environment): ExecutionResult,* отвечающая за исполнение команды. Из первого аргумента каждая команда будет знать, откуда получать данные, куда отправлять данные и куда писать при получении ошибок. Второй аргумент хранит всю информацию о переменных, а также текущую директорию (что полезно, например, для *PwdCommand)*. С помощью *PipeCommand* мы объединяем в одну цепочку все написанные команды. Она отвечает за то, чтобы создать фиктивный *OutputStream* для левой команды, потом превратить его в *InputStream* правой команды и т.д.

Также каждая команда хранит специфичную для нее информацию. Например, *EchoCommand* хранит список того, что надо вывести.

ExecutionResult - sealed class, который отвечает за коды возврата и завершение исполнения. В нашем случае у него 2 наследника: *CodeResult* и *ExitResult*

- *CodeResult* возвращается в большинстве случаев и содержит в себе код возврата. На данный момент нет поддержки передачи кода возврата с следующую команду, так как нет команд, использующих это. Но это можно легко поддержать, сохраняя код возврата в environment, или немного изменить *Command.execute*. Сейчас же код возврата используется только в *CommandExecutor*, и может например сохранятся (если вдруг *$?* появится в будущем) или завершать исполнение всей программы с тем же кодом возврата.
- ExitResult возвращается в случае команды exit и завершает исполнение, если этот ExecutionResult был получен в CommandExecutor (то есть был последним в цепочке pipe-ов), иначе игнорируется

IOContext – data class (== POJO) с тремя основными потоками: stdout, stdin, stderr

Environment – окружение исполнения, который хранит все описанные ранее переменные (в Map) и рабочую директорию

CommandExecutor – класс, создающийся в Main. Обладает единственным методом execute(command: String): ExecutionResult, который выполняет команду (лексит, парсит, вызывает run у команды). Поля CommandExecutor – лексер и парсер, environment и iocontext (смотри выше)

Внешние команды запускаются с помощью ProcessBuilder. У созданных Process есть getInputStream, getOutputStream, getErrorStream, и полученные потоки используются в PipeCommand.

Теперь поговорим об общем потоке исполнения. Main создает CommandExecutor, отдает ему input/output/err и запускает бесконечный цикл, который делает следующее:

- Спрашивает пользователя новую команду
- Отправляет ее CommandExecutor и получает результат
- Если она вернула ExitResult, то цикл заканчивается, иначе все сначала