Lisp. Транслятор и модель

- Шляпников Александр Дмитриевич
- `lisp -> asm | acc | neum | hw | instr | struct | stream | port | pstr | prob1 | cache`
- Базовый вариант.

Язык программирования

```
program ::= exp
          | exp exp

space ::= \s
line-break :== \n
str ::= "<any chars except spaces and line-breaks"

const ::= str
        | <any digits>

exp ::= (symbol)
      | (symbol const)
      | (symbol exp)

symbol ::= set input print print_int deproc if loop + - / * % and or = != > <

comment ::= <any symbols after: -->
```

Код выполняется последовательно. Операции:

- `(set <name> <value-exp>)` -- создать или обновить переменную
- `(input <variable name>)` -- ввести значение (строку) и сохранить в переменную
- `(input)` -- ввести значение (один символ)
- `(print <exp>)` -- напечатать значение являющееся результатам вычисления выражения (как строку)
- `(print_int <exp>)` -- напечатать значение являющееся результатам вычисления выражения (как число)
- `(deproc <name> <body-exp1..N>)` -- объявить процедуру с именем <name> и телом <body-exp1..N>
- `(if <cond> <body-exp1..N>)` -- условное выражение если <cond> сделать <body-exp1..N>
- `(loop <end-cond> <body-exp1..N>)` -- цикл с предусловием пока <cond> выполнять <body-exp1..N>
- `(+ <exp1> <exp2> ...)` -- последовательное суммирование всех операндов
- `(- <exp1> <exp2> ...)` -- последовательное вычитание всех операндов
- `(/ <exp1> <exp2> ...)` -- последовательное деление всех операндов
- `(* <exp1> <exp2> ...)` -- последовательное умножение всех операндов
- `(% <exp1> <exp2> ...)` -- последовательное взятие остатка всех операндов
- `(and <exp1> <exp2> ...)` -- последовательное логическое И всех операндов
- `(or <exp1> <exp2> ...)` -- последовательное логическое ИЛИ всех операндов
- `(= <exp1> <exp2> ...)` -- проверка, что все операнды равны
- `(!= <exp1> <exp2> ...)` -- проверка, что все операнды не равны
- `(> <exp1> <exp2> ...)` -- проверка, что все операнды последовательно уменьшаются
- `(< <exp1> <exp2> ...)` -- проверка, что все операнды последовательно растут
- `-- <any symbols \ text \ etc>` -- комментарий

Все операторы, а также операции принимающие на вход поддерживают потенциально любой уровень вложенности (при достатке памят)

Математические операторы работают только с числами, при попытке использовать в качестве операнда строку - поведение не определено

Все инструкции кроме `set` и `input` поддерживают любое кол-во последовательных exp

`(+ 1 (+ 2 3) (and 2 3))` - корректный код

`(deproc proc1 (set a 2) (set b (+ 1 (+ 2 3) (and 2 3)) )` - корректный код

Операция input может использоваться как операнд, но не может сама принимать на вход никакие аргументы кроме переменных

Корректный код:

- `(input a) (print a)`
- `(print (input))`

НЕ корректный код: (аргумент input НЕ переменная)

- `(input (+ 1 2))`

Память выделяется статически, при запуске модели.

Видимость данных -- глобальная.

Поддержка констант -- присутсвует.

```
(set b 1000)

(set c "Test")
```

Символы пробела и переноса строки в строках поддерживаются через плейсхолдеры \s \n соотвественно:

`(print "Hello\sworld")` --> **stdout:** Hello world

`(print "Hello\nworld")`--> **stdout:** Hello

world

Область видимости у всех переменных глобальная, в том числе у аргументов процедур.

Для иллюстрации работы транслятора с правилом "everything is expression" был написан тест expression_common.yml

```
  (print_int (set x 1))
  (print (if (= (+ 0 1) 1) "T"))
  (print (if (= (if (= x 1) (+ 0 1)) 1) (input)))
  (print "\n")
  (print_int x)
  (print "\n")
  (loop (or (< (+ x 1) 3) (= (+ x 1) 3))
      (set x (+ x 1))
      (print_int x)
      (print "\n")
  )
  stdin: dudo
  stdout: 1Td
  1
  2
  3
```

В данном примере видно как язык способен обрабатывать вложенные выражения

Для иллюстрации работы с процедурами, программа для алгоритма prob1 была написана с использование процедуры

```
(deproc prob1 (bound)
    (set sum 0)
    (set i 0)
    (loop (< i bound)
        (if (= (% i 3) 0)
            (set sum (+ sum i))
        )
        (if (and (= (% i 5) 0) (!= (% i 3) 0))
            (set sum (+ sum i))
        )
        (set i (+ i 1))
    )
    (print_int sum)
)
(prob1 1000)

stdin: EOF
stdout: 233168
```

Реализация успешно обрабатывает вызов процедуры, в этом примере также используются вложенные выражения, так что он может быть использован для иллюстрации поддержки принципа everything is expression

Организация памяти

Модель памяти процессора:

Память команд и данных находятся в едином адресном пространстве (не разделены) Команды хранятся как объекты описывающие инструкции Данные хранятся как список чисел В случае переполнения регистров доступной памяти в начале выполнения будет выброшено исключение

Обработка сложных операторов

`(<op> <exp> <exp>)`, где любой оператор, а сложные выражения требующие вычисления (не константы)

Для обработки таких команд каждый из вложенных предварительно вычисляет, под его хранение в памяти выделяется ячейка.

После того как все промежуточные значения вычислены выполняется верхняя команда `<op>` в качестве аргументов для нее используются ячейки с вычисленными ранее промежуточными значениями

Пример:

```
  (print_int (+ 1 2 (- 4 3)))
```

Транслируется в:

```
0: JP 6
1: MEM 1
2: MEM 2
3: MEM 4
4: MEM 3
5: MEM 0 <-- Ячейка промежуточного результата
6: LD 3  | <-- Выполнение расчёта промежуточного оператора (вычитание)
7: SUB 4 |
8: ST 5 <-- Сохранение промежуточного резуьтата
9: LD 1   | <-- Выполнение основной операции (сложение)
10: ADD 2 |
11: ADD 5 | <-- Сложение с промежуточным результатом
```

```
12: OUT_PURE 0
13: HLT
```

Обработка констант

Константы сохраняются в память и адреса для них проставляются на этапе трансляции.

```
(print_int 4)

0: JP 2
1: MEM 4 <-- Место константы в памяти
2: LD 1 <-- Константа загружается в аккумулятор
3: OUT_PURE 0
4: HLT
```

Строковые константы сохраняются в память переменных виде указателя на ячейку с длиной

А также в память строк в виде ячейки с длиной и последовательности ячеек символов

Регистры в памяти 32 разрядные, один разряд резервируется как флаг, поэтому максимально хранимое значение 2^31 Наличие старшего разряда показывает что перед нами указатель, а сами указатели адресуются по логике: `addr = pointer_val - 2^31`

```
(print "foo")

0:  JP 2
1:  MEM 2147483653 <-- Указатель
2:  LD 1
3:  OUT 0
4:  HLT
5:  MEM 3   |<-- Указывает сюда (2147483653 - 2^31 = 5)
6:  MEM 102 |
7:  MEM 111 |
8:  MEM 111 |
...
```

Буферизированное чтение

Буферизированное чтение достижимо при использовании `input` инструкции с аргументом в виде переменной

В таком случае на этапе трансляции будет выделена память под эту переменную размером в 64 машинных слова - это ограничение при чтении

```
(input a)

0:  JP 2
1:  MEM 2147483652 <-- Указатель на переменную
2:  IN 1 1 <-- Чтение в переменную по указателю
3:  HLT
4:  MEM 0 | <-- Выделенная память
5:  MEM 0 |
6:  MEM 0 |
7:  MEM 0 |
...       |
69: MEM 0 |
```

Организация памяти при выполнении

Исполнение начинается с нулевой ячейки с которой идёт безусловный переход к первой "значимой" инструкции Этот JP как и все остальные рассчитывается на этапе трансляции

```
Memory

0:  JP (N + 1) <-- Начало исполнения
1:  MEM ...  | <-- Память числовых констант \ переменных и указателей
...          |
N:  MEM ...  |


N + 1:  <OP> <ARG> | <-- Память инструкций (первая значимая инструкция тут)
...                |
N + M:  <OP> <ARG> |


N + M + 1: MEM ... | <-- Память строковых переменных \ констант
...                |
...                |
```

Система команд

Особенности процессора:

- Машинное слово -- 32 бит, знаковое.
- Доступ к памяти данных осуществляется по адресу, хранящемуся в специальном регистре `data_address`.
- Доступ к текущей инструкции осуществляется по адресу, хранящемуся в специальном регистре `program_address`.
- Напрямую установка адреса недоступна, осуществляется неявно внутри команд операторов, команд переходов и внутри команд загрузки, выгрузки памяти.

- Загрузка констант возможна при помощи команды `MEM <const>`
- Обработка данных происходит при помощи операторов `ADD`, `SUB`, `MUL`, `DIV`, `MOD`, `AND`, `OR`, `EQ`, `NEQ`, `GR`, `LW` каждый из них принимает на вход адрес второго операнда (первый - аккумулятор), а также команд загрузки выгрузки памяти `LD`, `ST`
- Поток управления:
  - инкремент `PC` после каждой инструкции;
  - условный (`JPZ`) и безусловный (`JP`) переходы
  - `counter` для выполнения команд более требующих больше одного такта
- Устройства ввода вывода используется при помощи портов не адресуются напрямую в память
  - Для доступа к устройству вывода используется команда OUT
  - Для доступа к устройству ввода используется команда IN

Набор инструкций

Инструкция	Кол-во тактов	Описание
LD `<addr>`	2	Загрузить в аккумулятор значение из ячейки по адресу `<addr>`
ST `<addr>`	2	Сохранить значение из аккумулятора в ячейке по адресу `<addr>`
ADD `<addr>`	3	Выполнить операцию сложения аккумулятора с значение ячейки по адресу `addr`
SUB `<addr>`	3	Выполнить операцию вычитания аккумулятора с значение ячейки по адресу `addr`
MUL `<addr>`	3	Выполнить операцию умножения аккумулятора с значение ячейки по адресу `addr`
DIV `<addr>`	3	Выполнить операцию деления аккумулятора с значение ячейки по адресу `addr`
MOD `<addr>`	3	Выполнить операцию взятия остатка аккумулятора с значение ячейки по адресу `addr`
AND `<addr>`	3	Выполнить операцию логического И аккумулятора с значение ячейки по адресу `addr`
OR `<addr>`	3	Выполнить операцию логического ИЛИ аккумулятора с значение ячейки по адресу `addr`
EQ `<addr>`	3	Проверить равенство аккумулятора с значение ячейки по адресу `addr`
NEQ `<addr>`	3	Проверить не равенство аккумулятора с значение ячейки по адресу `addr`
GR `<addr>`	3	Сравнить значение аккумулятора с значение ячейки по адресу `addr`
LW `<addr>`	3	Сравнить значение аккумулятора с значение ячейки по адресу `addr`
IN `<port> [pointer]`	1+	Сохранить значение из порта `port` в аккумуляторе или по указателю `[pointer]`
OUT `<port>`	1+	Подать на выход значение в аккумуляторе на порт `port` (строка)
OUT_PURE `<port>`	1	Подать на выход значение в аккумуляторе на порт `port` (напрямую, как число)
jmp `<addr>`	1	безусловный переход
jz `<addr>`	1	переход, если в аккумуляторе 0
HLT	-	остановить тактовый генератор
MEM `<const>`	-	преобразуется в константу в памяти по адресу индекса строки в которой указана инструкция

Кодирование инструкций

- Машинный код сериализуется в список команд объекты класса MemoryCell
- Один строка списка -- одна инструкция.
- Строки считаются с 0
- Номер строки -- адрес инструкции. Используется для команд перехода.
- При загрузке в память команды MEM заменяются на значение их аргумента - это память данных

Пример:

```
  0: JP 4                0: JP 4
  1: MEM 223             1: 223
  2: MEM 12              2: 12
  3: MEM 3         --->  3: 3
  4: LIT 0               4: LIT 0
  5: ST 2                5: ST 2
  6: LD 2                6: LD 2
  7: ST 3                7: ST 3
  8: IN 1                8: IN 1
```

Транслятор

Интерфейс командной строки:

```
Usage: translator.py [OPTIONS] SOURCE_CODE MACHINE_CODE

  Translator run control interface

  SOURCE_CODE - Path to source code file

  MACHINE_CODE - Path to output file, where machine code will be placed

Options:
  --help  Show this message and exit.
```

Реализовано в модуле: translator

Этапы трансляции (функция `main`):

1. Очистка комментариев и лишних пробелов
2. Парсинг исходного кода (см. parser) и преобразование его в дерево объектов Instruction (см. instruction)
3. Использование дерева для генерации машинного кода.

Правила генерации машинного кода:

- Процедуры поддерживаются только на уровне языка, в машинном коде происходит инлайнинг всех инструкций внутри тела функции
- Вложенные математические операции выполняются последовательно изнутри наружу, каждый этап вложенности вычисляется отдельно и сохраняется в ячейку. После вычисления всех уровней идёт последовательное применение оператора к полученному списку вычисленных значений
- Переменные также, как и константы отображаются на память статически, адреса рассчитываются при трансляции и больше не меняются

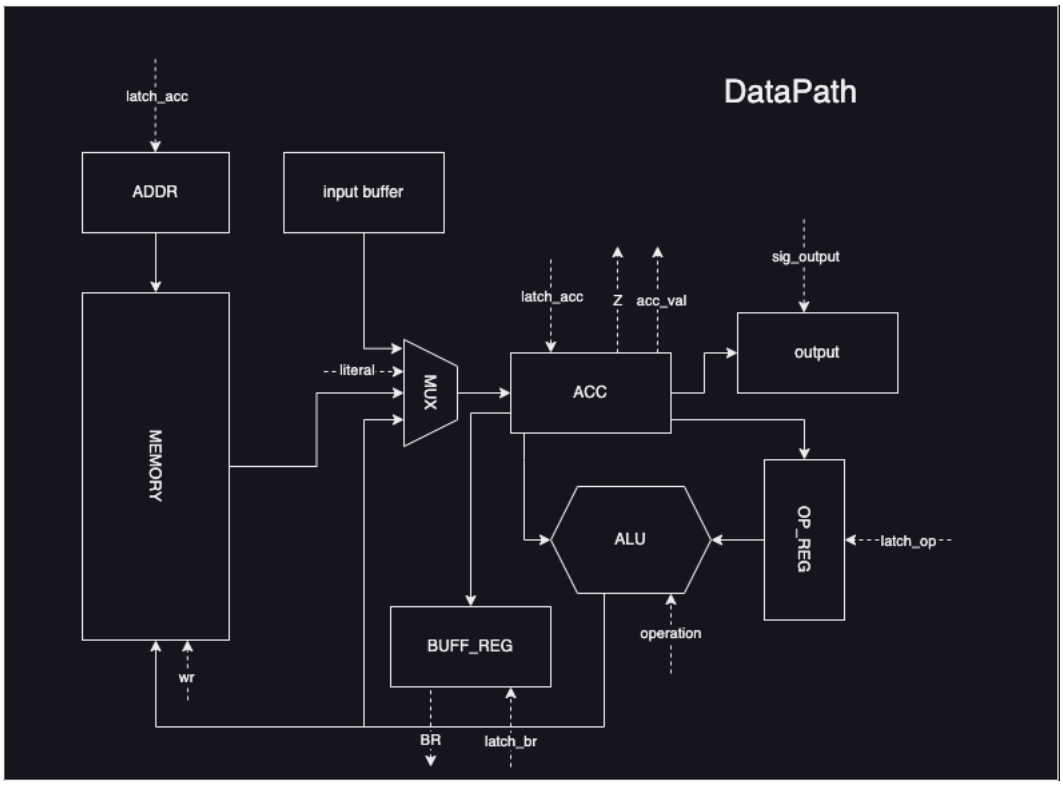
Модель процессора

Интерфейс командной строки:

```
Usage: machine.py [OPTIONS] CODE INPUT_BUFFER

  Simulator run control interface

  CODE - Machine code file path

  INPUT_BUFFER - Path to file which will emulate input buffer

Options:
  --log-output-file TEXT   File to place machine working logs; Default: Not
                           set
  --verbose-instr INTEGER  Amount of instructions to be verbose logged;
                           Default: Not set
  --logging-level TEXT     DEBUG, INFO, WARN, ERROR, CRITICAL; Default: DEBUG
  --help                   Show this message and exit.
```

Реализовано в модуле: machine.

DataPath



Реализован в классе `DataPath`.

`memory` -- однопортовая память, поэтому либо читаем, либо пишем.

Сигналы (обрабатываются за один такт, реализованы в виде методов класса):

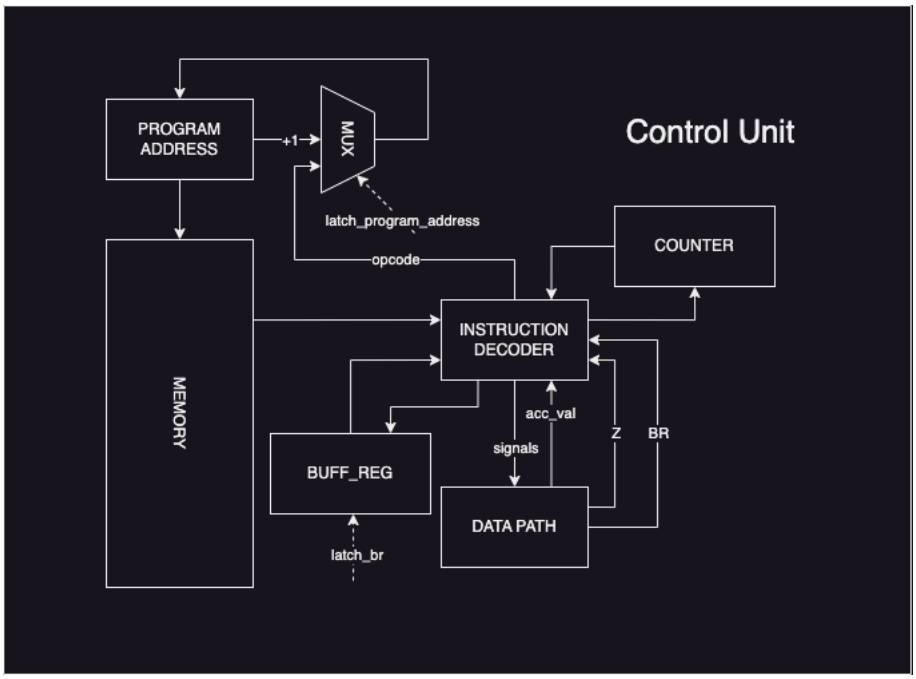
- `latch_addr` -- защёлкнуть выбранное значение в `data_addr`;
  - адрес приходит как управляющий сигнал ControlUnit
- `latch_acc` -- защёлкнуть в аккумулятор выбранное значение;
  - выход из памяти
  - константу
  - результат операции в алу
  - значение из порта ввода
- `latch_op` -- защёлкнуть значение регистра операнда
  - Берётся значение аккумулятора

- `wr` -- записать значение аккумулятора в память:
- `output` -- записать аккумулятор в порт вывода.
  - Строкой
  - Числом
- `latch_br` -- защёлкнуть значение буферного регистра
  - Берётся значение из аккумулятора

Сигналы для ControlUnit:

- `Z` -- отражает наличие нулевого значения в аккумуляторе.
- `acc_val` -- отражает значения в аккумуляторе.
- `BR` -- отражает значения в аккумуляторе.

ControlUnit



Реализован в классе `ControlUnit`.

- `program_address` - регистр указатель на исполняемую инструкцию в памяти
- Hardwired (реализовано полностью на Python).
- Метод `decode_and_execute_instruction` моделирует выполнение полного цикла инструкции.
- Counter реализован внутри метода `decode_and_execute_instruction` путем рекурсивных вызовов

Сигнал:

- `latch_program_address` -- сигнал для обновления счётчика команд в ControlUnit.
  - +1
  - Адрес из команды
- `latch_br` -- сигнал для защелкивания буферного регистра
  - Значение передается из декодера

Особенности работы модели:

- Цикл симуляции осуществляется в функции `simulate`.
- Шаг моделирования соответствует одной инструкции с выводом состояния в журнал.
- Для журнала состояний процессора используется стандартный модуль `logging`.
- Количество инструкций для моделирования лимитировано.
- Остановка моделирования осуществляется при:
  - превышении лимита количества выполняемых инструкций;
  - превышении лимита памяти
  - при превышении лимита в ячейке памяти
  - исключении `StopMachine` -- если выполнена инструкция `halt`.
  - исключении `MemoryCorrupted` -- если ControlUnit прочитал память данных вместо инструкции или не смог декодировать инструкцию

Тестирование

Тестирование выполняется при помощи golden test-ов.

Тесты реализованы в: golden_test.py. и в: prob1_golden_test.py.

Отдельный файл тестов для prob1 так как лог его работы довольно большой и внутри теста machine.py запускается с аргументами, который ограничиваю вывод для этой программы

Конфигурации:

- golden/cat_common.yml
- golden/hello_common.yml
- golden/hello_user_common.yml
- golden/prob1.yml
- golden/expression_common.yml

Запустить тесты: `poetry run pytest . -v`

Обновить конфигурацию golden tests: `poetry run pytest . -v --update-goldens`

CI при помощи Github Action: .github/workflows/lisp.yml

Пример использования и журнал работы процессора на примере `hello user`:

```
┌─dudosyka@MacBook-Pro-Alex-4 ~/PycharmProjects/lisp ‹master●›
└─$ cat ./test/input.txt
(print ">\sHello\swhat\sis\syour\sname\n")
(input a)
(print "<\s" a "\n>\sHi,\s" a)
┌─dudosyka@MacBook-Pro-Alex-4 ~/PycharmProjects/lisp ‹master●›
└─$ poetry run python3 ./translator.py ./test/input.txt ./test/output.txt
source LoC: 3 code instr: 13
┌─dudosyka@MacBook-Pro-Alex-4 ~/PycharmProjects/lisp ‹master●›
└─$ poetry run python3 ./machine.py ./test/output.txt ./test/input_buffer.txt
DEBUG:root:INSTR:   0 TICK:   1 PC:   5 ADDR:   0 MEM_OUT: Opcode.JP ACC: 0 OP_REG: 0 COMMAND: JP 5 @0
DEBUG:root:INSTR:   1 TICK:   3 PC:   6 ADDR:   1 MEM_OUT: 2147483665 ACC: 2147483665 OP_REG: 0 COMMAND: LD 1 @5
DEBUG:root:INSTR:   2 TICK:  58 PC:   7 ADDR:  43 MEM_OUT: 10 ACC: 10 OP_REG: 0 COMMAND: OUT 0 @6
DEBUG:root:INSTR:   3 TICK:  73 PC:   8 ADDR:  44 MEM_OUT: 8 ACC: 8 OP_REG: 0 COMMAND: IN 1 2 @7
DEBUG:root:INSTR:   4 TICK:  75 PC:   9 ADDR:   3 MEM_OUT: 2147483757 ACC: 2147483757 OP_REG: 0 COMMAND: LD 3 @8
DEBUG:root:INSTR:   5 TICK:  82 PC:  10 ADDR: 111 MEM_OUT: 32 ACC: 32 OP_REG: 0 COMMAND: OUT 0 @9
DEBUG:root:INSTR:   6 TICK:  84 PC:  11 ADDR:   2 MEM_OUT: 2147483692 ACC: 2147483692 OP_REG: 0 COMMAND: LD 2 @10
DEBUG:root:INSTR:   7 TICK: 103 PC:  12 ADDR:  52 MEM_OUT: 97 ACC: 97 OP_REG: 0 COMMAND: OUT 0 @11
DEBUG:root:INSTR:   8 TICK: 105 PC:  13 ADDR:   4 MEM_OUT: 2147483760 ACC: 2147483760 OP_REG: 0 COMMAND: LD 4 @12
DEBUG:root:INSTR:   9 TICK: 122 PC:  14 ADDR: 119 MEM_OUT: 32 ACC: 32 OP_REG: 0 COMMAND: OUT 0 @13
DEBUG:root:INSTR:  10 TICK: 124 PC:  15 ADDR:   2 MEM_OUT: 2147483692 ACC: 2147483692 OP_REG: 0 COMMAND: LD 2 @14
DEBUG:root:INSTR:  11 TICK: 143 PC:  16 ADDR:  52 MEM_OUT: 97 ACC: 97 OP_REG: 0 COMMAND: OUT 0 @15
DEBUG:root:Total instructions: 12
DEBUG:root:Total ticks: 143
DEBUG:root:Output buffer: > Hello what is your name
< dudosyka
> Hi, dudosyka
DEBUG:root:Execution stopped
> Hello what is your name
< dudosyka
> Hi, dudosyka
```

Пример проверки исходного кода:

```
┌─dudosyka@MacBook-Pro-Alex-4 ~/PycharmProjects/lisp ‹master●›
└─$ poetry run pytest . -v
============================================================================================================================
platform darwin -- Python 3.12.3, pytest-7.4.4, pluggy-1.5.0 -- /Users/dudosyka/Library/Caches/pypoetry/virtualenvs/lisp-i1
cachedir: .pytest_cache
rootdir: /Users/dudosyka/PycharmProjects/lisp
configfile: pyproject.toml
plugins: golden-0.2.2
collected 5 items

golden_test.py::test_translator_and_machine[golden/expression_common.yml] PASSED
golden_test.py::test_translator_and_machine[golden/hello_user_common.yml] PASSED
golden_test.py::test_translator_and_machine[golden/hello_common.yml] PASSED
golden_test.py::test_translator_and_machine[golden/cat_common.yml] PASSED
prob1_golden_test.py::test_translator_and_machine[golden/prob1.yml] PASSED

============================================================================================================================
┌─dudosyka@MacBook-Pro-Alex-4 ~/PycharmProjects/lisp ‹master●›
└─$ poetry run ruff check .
┌─dudosyka@MacBook-Pro-Alex-4 ~/PycharmProjects/lisp ‹master●›
└─$ poetry run ruff format .
8 files left unchanged
```

ФИО	алг	LoC	code байт	code инстр.	инстр.	такт.	вариант
Шляпников Александр Дмитриевич	hello	1	-	4	4	28	-
Шляпников Александр Дмитриевич	cat	5	-	12	31	49	-
Шляпников Александр Дмитриевич	hello_user	3	-	13	12	123	-
Шляпников Александр Дмитриевич	prob1	15	-	44	29424	64291	-