Тема: Std, STL

Вариант: 4.1.1

Задача: Реализовать интерпретатор для модельного языка программирования **DL**

Опишем **синтаксис** языка **DL** с помощью форм Бэкуса-Наура. Здесь *<integer>* и *<id>* обозначают целое число и идентификатор в обычном понимании соответственно.

```
<expression> ::= <val> | <var> | <add> | <if> | <let> |
                 <function> | <call>

<val> ::= (val <integer>)

<var> ::= (var <id>)

<add> ::= (add <expression> <expression>)

<if>  ::= (if <expression> <expression>
           then <expression>
           else <expression>)

<let> ::= (let <id> = <expression> in <expression>)

<function> ::= (function <id> <expression>)

<call> ::= (call <expression> <expression>)
```

Опишем процесс интерпретации программы на языке **DL**.

Пусть **env** – структура данных, содержащая пары *<<id>, <expression>>*, описывающие названия доступных в данный момент переменных и соответствующие им значения.

Тогда `fromEnv(<id>)` – функция, которая возвращает *<expression>*, соответствующий данному *<id>* в **env**, если таковой имеется, и провоцирует ошибку иначе.

Обозначим `getValue(<expression>)` функцию, которая возвращает соответствующий *<integer>*, если вызвана от типа *<val>*, и провоцирует ошибку иначе.

Наконец, опишем функцию `eval(<expression>)` для каждого из типов. Далее для обозначения конкретного *<expression>* используется запись вида `<"тип expression" "список параметров">`

1. `eval(<val i>) = <val i>`
2. `eval(<var id>) = fromEnv(id)`
3. `eval(<add e1 e2>) = <val getValue(eval(e1)) + getValue(eval(e1))>`
4. `eval(<if e1 e2 e_then e_else>) =`

   `eval(e_then),`
   `если getValue(eval(e1)) > getValue(eval(e2)),`

   `eval(e_else)` иначе.

5. `eval(<let id = e_value in e_body>) = eval(body),` вычисленный с учетом добавления в **env** пары: `<id, eval(e_value)>`

6. `eval(<function id expression>) =`
   `<function id expression>`

7. Пусть `f` — `<expression>` типа `<function>`. Обозначим `f.arg_id` и `f.body` параметры *`<id>`* и *`<expression>`* из описания соответственно. Тогда

   `eval(<call f_expr arg_expr>)` =

   - если `eval(f_expr)` не является *`<function>`*, то исполнение `eval` провоцирует ошибку,
   - в противном случае возвращает `eval(eval(f_expr).body)`, при этом внешний вызов `eval` происходит с **env** состоящим из пары: `<f.arg_id, eval(arg_expr)>`
   - отдельно опишем случай, когда `f_expr` является выражением типа `<var f_id>`, например:

     ```
     (let f = (function arg (add (var arg) (val 1)))
       in (call (var f) (val 0))
     )
     ```

     В таком случае, при вызове `eval(call(f_expr arg_expr))` в **env** также должна быть доступна пара `<f_id, eval(f_expr)>`, позволяющая совершить вызов по имени функции.

Интерпретацией программы на языке **DL** назовем вычисление функции `eval` от `<expression>`, содержащего представление данной программы.

В демонстрационном примере необходимо считать из файла программу на языке **DL**, создать соответствующий `<expression>` и вывести в выходной файл результат функции `eval` от него.

Замечания по реализации:

1. Различные конструкции языка стоит представить в виде иерархии классов, с базовым абстрактным классом Expression, в котором описана чистая виртуальная функция `eval`(...)

2. Для реализации **env** можно использовать либо класс std::unordered_map из стандартной библиотеку, либо свою собственную структуру данных (например, реализованную в третьей задаче).

3. В коде должно быть реализовано корректное управление динамической памятью: не должно быть утечек памяти, некорректных указателей и т. д

Входные данные:

Программа на языке **DL**.

В записи программы могут присутствовать (а могут и нет) лишние пробелы, табуляции и переносы строк для выравнивания конструкций языка, это не должно влиять на результат.

Выходные данные:

Если функция `eval` от выражения, представляющего программу, была выполнена успешно, то в выходной файл необходимо напечатать получившееся выражение.

Если во время исполнения `eval` произошла ошибка, вывести в файл строку “ERROR”.

Пример входных и выходных данных:

input.txt	output.txt

`(let K = (val 10) in`
`  (add (val 5) (var K))`
`)`
	`(val 15)`

`(let A = (val 20) in`
`  (let B = (val 30) in`
`  (if (var A) (add (var B) (val 3))`
`  then (val 10)`
`  else (add (var B) (val 1)))))`
	`(val 31)`

`(let F =`
`  (function arg (add (var arg) (val 1))) in`
`  (let V = (val -1) in`
`  (call (var F) (var V))))`
	`(val 0)`

`(add (var A) (var B))`	`ERROR`

Дополнительные задания:

Для получения автомата достаточно выполнить два из трех.

1. Добавить поддержку **lexical scope**, т.е. сделать так, чтобы при вычислении `eval` от тела функции были доступны не только аргументы функции, но и переменные, которые были доступны в точке **объявления** функции.

2. Расширить язык следующими конструкциями:

   <set> ::= (set *<id>* *<expression>*)

   При этом `eval(<set id e_val>)` добавляет в **env** пару `<id, eval(e_val)>`, либо меняет значение уже

существующей там пары с таким ключом. Возвращает при этом само выражение `<set id e_val>`

`<block> ::= (block <expression> ... <expression>)`

При этом `eval(<block e_1 ... e_n>)` вычисляет `eval` от каждого выражения из списка, а в качестве результата возвращает результат вызова `eval` от последнего.

3. Расширить язык конструкциями для работы с массивами:

   `<arr> ::= (arr <expression> ... <expression>)`

   При этом `eval(<arr e_1 ... e_n>) =`
   `<arr eval(e_1) ... eval(e_n)>`

   `--`

   `<gen> ::= (gen <expression> <expression>)`

   При этом `eval(<gen e_length e_function>)` возвращает массив из `eval(e_length)` элементов, причем i-ый элемент вычислен как `eval(<call e_function (val i)>)`

   `--`

   `<at> ::= (at <expression> <expression>)`

   При этом `eval(<at e_array e_index>)` возвращает элемент из `eval(e_array)` с индексом `getValue(eval(e_index))`, либо провоцирует ошибку, если полученный индекс выходит за границы массива или `eval(e_array)` не является выражением типа `arr`