1. ВТОРОЙ ОТБОРОЧНЫЙ ЭТАП

Задачи второго этапа

1.1. Задачи по информатике

Задача 1.1.1. Компилятор для EVM (31 баллов)

Предположим, есть язык программирования F-stoke, разработанный для написания контрактов для платформы Ethereum.

F-stroke – язык, поддерживающий функциональную парадигму (http://bit.ly/2YLQ6WE). Являясь упрощенной и модифицированной версией языка Lisp, он берет из этого языка базовый синтаксис и семантику.

Структура программы и определения

Программа на языке F-stroke – последовательность элементов. Элементы – это либо атомы (*идентификаторы*), либо *константы*, либо *списки*.

Константа – это заданное в явном виде значение. В языке F-stroke все константы только безнаковые целые числа.

Атомы похожи на переменные в традиционных языках программирования. С атомом ассоциирован идентификатор. Атом может иметь значение. Значение атома – это либо значение соответствующей константы, либо список.

Списки – это последовательности элементов, разделенные пробелами и заключенные в круглые скобки.

Исполнение программы начинается с самого первого элемента, расположенного точке входа (будет определена позже). Каждый элемент обрабатывается в соответствии со своей семантикой:

- Если текущий элемент – атом, используется его значение
- Если текущий элемент – список, он обрабатывается как вызов функции, имя которой – это первый элемент списка, а последующие элементы (если они есть) рассматриваются как аргументы функции.

Некоторые списки имеют специальную назначение. Если список начинается с одного из приведенных ниже ключевого слова, то это называется *специальной формой* и обрабатывается одним способов описанных далее.

Ключевые слова для специальных форм:

```
setq func prog cond while return break
```

Специальные формы

Специальные формы определяют инструментарий языка программирования и поэтому каждая имеет свое собственное поведение. Каждая специальная форма – список, где первый элмент списка - ключевое слово. Остальная часть специальной формы содержит число элементов, специфичное для каждой формы.

`* ( setq Atom Element )`

Эта форма выполняет присваивание значения атому. Сначала обрабатывается третий элемент списка и полученный результат определяет новое значение атома из второго элемента списка, заменяя его предыдущее значение. Если в текущем контексте исполнения нет атома с данным именем, тогда он создается и добавляется в контекст.

Поскольку результатом вычисления формы `setq` целиком является неопределенное значение, то использование формы в составе арифметических или логических операций не допускается.

Примеры:

```
( setq x 5 )
( setq y ( plus 1 2 ) ) // atom y gets the value of 3
```

`* ( func Atom List Element )`

Эта форма используется для задания определенной пользователем функции. Второй элемент списка становится именем функции. Третий элемент списка должен содержать атомы, которые будут расцениваться, как параметры функции. Четвертый элемент – тело функции.

Следует отметить, что каждая определенная пользователем функция вводит свой собственный локальный контекст исполнения. Это значит, что атомы, представляющие параметры функции, рассматриваются локальными к функции, также как атомы объявленные `setq` формой. Если `setq` форма объявляет атом с таким же именем, как атом из внешнего контекста, то локально объявленный атом "скрывает"атом из внешнего контекста. Локальный контекст функции пропадает после исполнения функции.

Тело функции может состоять из списка элементов, выраженных также списками. Тогда при выполнении тела функции такие элементы обрабатываются последовательно.

Результатом исполнения функции будет являться результат последнего обработанного элемента.

Функция должна быть определена до использования в теле другой функции.

Функция не может входить в тело других форм.

Примеры:

```
( func Cube ( arg ) ( times ( times arg arg ) arg ) )
( func One () 1 )
```

```
( func dec ( i ) ( minus i 1 ) )
( func pow ( arg1 arg2 ) (
  ( setq res 1 )
  ( while ( nonequal arg2 0 )
```

```
5           ( ( setq res ( times res arg1 ) )
6             ( setq arg2 ( dec arg2 ) )
7           )
8     )
9     ( return res ) )
10  )
```

`* ( prog ( List { List } ) )`

Эта форма объявляет точку входа, с которой начинается выполнение программы. Во втором элементе списка – список элементов, которые будут выполняться последовательно.

В программе может быть только одна форма `prog`. Она не может входить в тело других форм.

Самым последним элементом, выполняющимся в форме `prog`, должна быть специальная форма `return`.

Определенные пользователем функции должны быть определены до формы `prog`.

Примеры:

```
1  ( prog ( ( return ( pow ( read 0 ) ) ) ) )
```

```
1  ( prog (
2      ( setq v ( read 0 ) )
3      ( setq b ( read 1 ) )
4      ( setq l ( read 2 ) )
5      ( setq res 0 )
6      ( while ( equal 1 1 ) (
7              ( cond
8              ( equal ( minus v ( times ( divide v b ) b ) ) l )
9              ( setq res ( plus res 1 ) )
10         )
11         ( setq v ( divide v b ) )
12         ( cond ( equal v 0 ) ( return res ) )
13       )
14     )
15   )
16 )
```

`* ( cond List List { List } )`

Данная форма – аналог условного оператора в традиционных языках программирования. Она состоит из двух или трех аргументов. Получение результа для этой формы начинается с выполнения функции, вызываемой в первом аргументе. Функция должна вернуть истину или ложь. Если функция вернула истину, то вычисляется второй аргумент. Иначе вычисляется третий аргумент, если он есть.

Поскольку результатом вычисления формы целиком является неопределенное значение, то использование формы в составе арифметических или логических операций не допускается.

`* ( while List List )`

Форма используется для выполнения повторяющихся действий. Сначала вычисляется второй элемент списка. Если результат вычисления истина, тогда вычисляется третий элемент списка, а затем контроль возвращается снова для вычисления

второго элемента списка. Иными словами, второй элемент списка вычисляется перед каждой итерацией. Если результат вычисления – ложь, то форма `while` завершается.

Содержимое третьего элемента списка (если определено также в виде списка) обрабатываются последовательно.

Поскольку результатом вычисления формы целиком является неопределенное значение, то использование формы в составе арифметических или логических операций не допускается.

`* ( return Element )`

Эта форма применима только внутри форм `func` и `prog`. Она вычисляет аргумент и прерывает выполнение той формы, внутри которой выполняется.

`* ( break )`

Эта форма применима только внутри формы `while`. Она безусловно прерывает выполнение ближайшей формы, в которой выполнилась форма `break`.

Предопределенные функции

Все предопределенные функции выполняют какие-то действия над переданными аргументами и возвращают какой-то результат. Результатом выполнения предопределенных функций может быть целое число, либо логический тип внутреннего использования (не может быть возвращен из пользовательских функций и формы `prog`). Вызовы функций представлены в виде списков, где первый элемент – атом (идентификатор), определяющий какая функция будет вызвана.

Все предопределенные функции за исключением `not` и `read` принимают по два аргумента. Перед передачей управления в тело функции первым действием вычисляются все аргументы: сначала первый, потом второй.

Арифметические операции

- `( plus Element Element )` – складывает два аргумента
- `( minus Element Element )` – вычитает из первого второй аргумент
- `( times Element Element )` – умножает два аргумента
- `( divide Element Element )` – возвращает результат целочисленного деления первого аргумента на второй

Сравнение

- `( equal Element Element )` – возвращает истину, если два аргумента равны
- `( nonequal Element Element )` – возвращает истину, если два аргумента не равны
- `( less Element Element )` – возвращает истину, если первый аргумент меньше второго
- `( lesseq Element Element )` – возвращает истину, если первый аргумент меньше либо равен второму
- `( greater Element Element )` – возвращает истину, если первый аргумент больше второго
- `( greatereq Element Element )` – возвращает истину, если первый аргумент больше либо равен второму

Все функции сравнения возвращают логический тип внутреннего использования, который не может быть возвращен из пользовательских функций и формы `prog`.

Логические операции

В качестве аргументов данных функций могут быть использованы только результаты функций сравнения или функций, выполняющих логические операции.

- `( and Element Element )` – возвращает истину, если результаты вычисления обоих элементов – истина
- `( or Element Element )` – возвращает истину, если результат вычисления хотя бы одного из выражений – истина
- `( not Element )` – возвращает истину, только если результат вычисления элемента – ложь

Результат функций, выполняющих логические операции, – логический тип внутреннего использования.

Получение данных

Специальная функция позволяет получать целое число, которое было передано в качестве параметра при вызова кода контракта:

`( read Element )`

В качестве аргумента данная функция принимает целое число – индекс параметра в передаваемых в контракт данных. Нумерация индексов начинается с 0. Параметр – это всегда безнаковое целое число размерностью в 256 бит.

Примеры:

```
( prog  ( ( return ( plus ( read 0 ) ( read 1 ) ) ) ) )
```

Данный контракт вернет сумму двух чисел, которые были переданы ему в качестве параметра.

Грамматика языка F-stroke

При определении специальных форм и предопределенных функций выше использовались следующие сущности, заданные в грамматике языка F-stroke:

```
Program : Element { Element }
List : ( Element { Element } )
Element : Atom | Literal | List
Atom : Identifier
Literal : Integer
Identifier : Letter { Letter | DecimalDigit }
Letter : Any Unicode character that represents a letter
Integer : DecimalDigit { DecimalDigit }
DecimalDigit : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Виртуальная машина Ethereum

Виртуальная машина (http://bit.ly/38Fww34) Ethereum (Ethereum Virtual Machine, EVM) имеет стековую архитектуру (http://bit.ly/2Eiy4BU), оперирует данными размером 256 бит, представленными в формате *big-endian*. EVM используется для выполнения контрактов, сохраненных в блокчейн Ethereum.

Контракты подобны обычным Ethereum аккаунтам за исключением того, что в тот момент как аккаунт получает входящую транзакцию, EVM выполняет байткод

ассоциированный с данным аккаунтом, позволяя производить вычисления и последующие транзакции. Здесь под транзакциями подразумевается не столько закодированная последовательность байт с цифровой подписью, сколько запрос на изменение состояния аккаунта или на доступ к данным аккаунта.

Транзакции могут содержать данные, которые используются для передачи в EVM информации (например, входных параметров), расширяя контекст выполнения контракта.

Исполнение контракта начинается с самой первой инструкции.

Каждая EMV команда (опкод) кодируется одним байтом за исключением команды `PUSH`, которая принимает дополнительное значение.

Обслуживаемый EVM стек состоит из 1024 слов длиной по 256 бит. Стек используется для хранения локальных переменных, аргументов вызываемых функций, адресов возврата.

Все команды извлекают свои операнды из стека, смещая его вершину, и помещают результат своего выполнения на стек.

Для примера рассмотрим небольшой фрагмент контракта и то, как каждая команда изменяет состояние стека:

```
PC| OPCODE     | Stack representation
--|----------- |----------------------
00: PUSH1 0x0a | | 0x0a | ...
02: PUSH1 0x0f | | 0x0f | 0x0a | ...
04: DUP2       | | 0x0a | 0x0f | 0x0a | ...
05: ADD        | | 0x19 | 0x0a | ...
06: MUL        | | 0xfa | ...
```

Рассмотренный выше фрагмент контракта представляется в виде набора байт, шестнадцатиричная запись которого выглядит следующим образом: `600a600f810102`.

Более подробно о том, как именно каждая команда EVM работает со стеком и какой код соответствует какой команде можно получить в справочнике "Ethereum Virtual Machine Opcodes"(https://ethervm.io/).

Виртуальная машина также может оперировать с памятью, которая представлена массивом из однобайтовых элементов. Содержимое этой памяти не сохраняется между вызовами контракта. Размер доступной памяти теоретически - неограничен. Фактически он ограничен размером средств, которые могут быть потрачены на выполение контракта работающего с большими объемами данных в памяти. В данной задаче объем памяти ограничен системой проверки задачи на Stepik. Для работы с памятью существует три команды: `MLOAD`, `MSTORE` и `MSTORE8`.

Транзакции, вызывающие контракт на выполнение, могут передавать в контекст выполнения контракта внешние данные. Внутри контракта эти данные могут быть считаны с помощью команд `CALLDATALOAD` и `CALLDATACOPY`.

В реальных контрактах Ethereum существует также сохранять данные между вызовами контракта в хранилище (*storage*), ассоциированном с каждым контрактом. Хранилище - это набор записей `key:value`, где `key` – беззнаковое целое число длиной 256 бит, и `value` – также беззнаковое целое число длиной 256 бит.

Задача

Ознакомившись с описанием языка высокого уровня F-stroke и основами функционирования виртуальной машины Ethereum, напишите программу, которая бы представляла бы собой компилятор из программы на F-stroke в программу (контракт), выраженную в виде байткода EVM. Базовое представление о том, как можно написать такой компилятор можно почерпнуть в статье *"Стековые и регистровые машины"* (https://ps-group.github.io/compilers/stack_and_register). Обратите внимание на упражнения из этой статьи.

Помимо свойств виртуальной машины Ethereum, описанных выше, в данной задаче для упрощения введены также следующие ограничения:

- Примененная в данной задаче реализация EVM не будет использовать следующие команды:

  `SHA3, ADDRESS, BALANCE, ORIGIN, CALLER, CALLVALUE, CODESIZE, CODECOPY, GASPRICE, EXTCODESIZE, EXTCODECOPY, RETURNDATASIZE, RETURNDATACOPY, EXTCODEHASH, BLOCKHASH, COINBASE, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, GAS, LOG0, LOG1, LOG2, LOG3, LOG4 CREATE, CALL, CALLCODE, DELEGATECALL, CREATE2, STATICCALL, REVERT, SELFDESTRUCT`

- Выполнение контракта завершается командой `RETURN`
- Даже если при исполнении контракта использовалась команда `SSTORE`, то по завершению исполнения команд контракта (успешному или досрочному) реальное изменение в хранилище контракта не происходит. Иными словами, код контракта можно рассматривать как чистую (*pure*) функцию в терминах функционального программирования.
- Обращение к элементу за границами стека (например, `PUSH32` при заполненном стеке или `POP` при пустом стеке) приведет к досрочному завершению программы. Длина возвращаемого набора данных (`RETURNDATASIZE`) в этом случае — 0.
- Обращение к непроинициализированной памяти через `MLOAD` или `SLOAD` возвращает 0
- Переход с помощью `JUMP` или `JUMPI` должен происходить только к команде `JUMPDEST`. Иначе — досрочное завершение программы. Длина возвращаемого набора данных (`RETURNDATASIZE`) в этом случае — 0
- Попытка выполнить нераспознанную или нереализованную команду приведет к досрочному завершению программы. Длина возвращаемого набора данных (`RETURNDATASIZE`) в этом случае — 0.

Также специфика виртуальной машины Ethereum вносит следующее ограничение на использование функций в F-stroke: в функцию не может быть передано больше, чем 15 аргументов. В данной же задаче функции в предлагаемых для компиляции программах на F-stroke не будут иметь больше, чем 4 аргумента.

Проверка решения будет происходить выполнением программы в байткоде с различными входными данными.

Программы на языке F-stroke, которые будут подаваться на вход вашему решению, будут синтаксически верные.

Полезные ссылки:

- *"Изучение работы EVM на примере разбора Solidity контракта"* (http://bit.ly/35pdzzA)

- *"Bytecode to opcodes"* (https://etherscan.io/opcode-tool)
- *"Solidity Decompiler"* (https://ethervm.io/decompile)

Примеры

Пример 1 – сумма двух чисел, переданных в контракт в качестве параметров

```
( prog (
  ( setq x ( read 0 ) )
  ( setq y ( read 1 ) )
  ( return ( plus x y ) ) )
)
```

Может быть оттранслирован в

```
600035602035016000526020600 0F3
```

что соответствует:

```
00: 60 00 PUSH1 0x00
02: 35    CALLDATALOAD
03: 60 20 PUSH1 0x20
05: 35    CALLDATALOAD
06: 01    ADD
07: 60 00 PUSH1 0x00
09: 52    MSTORE
0a: 60 20 PUSH1 0x20
0c: 60 00 PUSH1 0x00
0e: f3    RETURN
```

Пример 2 – сумма первых 10 членов арифметической прогрессии

```
( prog (
  ( setq sum 0 )
  ( setq i 10 )
  ( while ( nonequal i 0 ) (
    ( setq sum ( plus sum i ) )
    ( setq i ( minus i 1 ) ) )
  )
  ( return sum ) )
)
```

Может быть оттранслирован в

```
60008052600a602052600c565b60205115602a57600051602051016
000526001602051036020526 00c565b60206000f3
```

что соответствует:

```
00: 60 00 PUSH1 0x00
02: 80    DUP1
03: 52    MSTORE
04: 60 0a PUSH1 0x0a
```

```
06: 60 20 PUSH1 0x20
08: 52    MSTORE
09: 60 0c PUSH1 0x0c
0b: 56    JUMP
0c: 5b    JUMPDEST
0d: 60 20 PUSH1 0x20
0f: 51    MLOAD
10: 15    ISZERO
11: 60 2a PUSH1 0x2a
13: 57    JUMPI
14: 60 00 PUSH1 0x00
16: 51    MLOAD
17: 60 20 PUSH1 0x20
19: 51    MLOAD
1a: 01    ADD
1b: 60 00 PUSH1 0x00
1d: 52    MSTORE
1e: 60 01 PUSH1 0x01
20: 60 20 PUSH1 0x20
22: 51    MLOAD
23: 03    SUB
24: 60 20 PUSH1 0x20
26: 52    MSTORE
27: 60 0c PUSH1 0x0c
29: 56    JUMP
2a: 5b    JUMPDEST
2b: 60 20 PUSH1 0x20
2d: 60 00 PUSH1 0x00
2f: f3    RETURN
```

Пример 3 – сумма первых 10 членов арифметической прогрессии с использованием рекурсии

```
( func sum ( x ) (
    ( cond ( equal x 0 )
      ( return 0 )
      ( return ( plus x ( sum ( minus x 1 ) ) ) ) )
    )
  )
)

( prog ( ( return ( sum 10 ) ) ) )
```

Может быть оттранслирован в

```
600a6008906011565b60005260206000f35b8015829060235750602
86001820360011565b909150565b019056
```

что соответствует:

```
00: 60 0a PUSH1 0x0a
```

```
02: 60 08 PUSH1 0x08 [A]
04: 90    SWAP1
05: 60 11 PUSH1 0x11 [B]
07: 56    JUMP
08: 5b    JUMPDEST [A]
09: 60 00 PUSH1 0x00
0b: 52    MSTORE
0c: 60 20 PUSH1 0x20
0e: 60 00 PUSH1 0x00
10: f3    RETURN
11: 5b    JUMPDEST [B]
12: 80    DUP1
13: 15    ISZERO
14: 82    DUP3
15: 90    SWAP1
16: 60 23 PUSH1 23 [C]
18: 57    JUMPI
19: 50    POP
1a: 60 28 PUSH1 28 [D]
1c: 60 01 PUSH1 0x01
1e: 82    DUP3
1f: 03    SUB
20: 60 11 PUSH1 0x11 [B]
22: 56    JUMP
23: 5b    JUMPDEST [C]
24: 90    SWAP1
25: 91    SWAP2
26: 50    POP
27: 56    JUMP
28: 5b    JUMPDEST [D]
29: 01    ADD
2a: 90    SWAP1
2b: 56    JUMP
```

Дополнительная информация

Для тестирования своего решения используйте примеры программ на F-stoke, представленные ниже, а работоспособность получившегося байткода опробуйте в "реальной"сети. Для этого нужно отправить транзакцию на создание контракта, где первые 32 байта - стандартный конструктор, а затем должен идти полученный код. 19-ый и 20-ый байты в конструкторе нужно поменять на длину полученного кода (в байтах, *big endian*). Остальные байты, идущие после конструктора – байткод контракта.

Jupyter Notebook для тестирования контракта в реальной сети доступен вот по этой ссылке: http://bit.ly/2qvaAX3.

Пример 1.

Данный контракт осуществляет перевод из градусов по шкале Цельсия в градусы по шкале Фаренгейта.

```
1 ( prog ( ( return ( plus ( divide ( times ( read 0 ) 9 ) 5 ) 32 ) ) ) )
```

Пример 2.

Данный контракт осуществляет подсчет количества знаков в числе `n`, записанного в указанной системе счисления `b`.

```
( prog (
    ( setq n ( read 0 ) )
    ( setq b ( read 1 ) )
    ( cond ( equal b 0) ( return 0 ) (
        ( setq res 1 )
        ( while ( lesseq b n ) (
            ( setq n ( divide n b ) )
            ( setq res ( plus res 1 ) )
          )
        )
        ( return res )
      )
    )
  )
)
```

Пример 3.

Данный контракт в зависимости от первого входного параметра `command` определяет:

- есть ли в двоичном представлении указанного числа `v` последовательность битов со значением 0 длиной больше заданного `q`;
- определяет является ли указанное число `v`, при записи в десятичном виде, палиндром;

Контракт возвращает 1, если условие выполняется, или 0 – если не выполняется.

```
( func modulo ( a b ) ( minus a ( times ( divide a b ) b ) ) )

( func sequence ( v q n ) (
    ( setq m ( plus n 1 ) )
    ( cond ( equal v 0 )
        ( cond ( greater m q )
            ( return 1 )
            ( return 0 ) )
        ( cond ( equal v 1 )
            ( return 0 ) )
    )
    ( setq nextval (divide v 2) )
    ( cond ( equal ( modulo v 2 ) 0 )
        ( cond ( greater m q )
              ( return 1 )
              ( return ( sequence nextval q m ) ) )
        ( return ( sequence nextval q 0 ) )
    )
  )
)

( func palindrom ( v ) (
    ( cond ( equal v 0 ) ( return 0 ) )
    ( setq acc 0 )
    ( setq rest v )
    ( while ( less acc rest ) (
        ( setq acc ( plus ( times acc 10 )
```

```
28                          ( modulo rest 10 ) ) )
29          ( setq rest ( divide rest 10) )
30        )
31      )
32      ( cond ( or ( equal acc rest )
33                  ( equal ( divide acc 10 ) rest ) )
34          ( return 1 )
35          ( return 0 ) )
36    )
37  )
38
39  ( prog (
40      ( setq command ( read 0 ) )
41      ( setq res 0 )
42      ( cond ( equal command 0)
43          ( setq res ( sequence ( read 1 ) ( read 2 ) 0 ) )
44          ( cond ( equal command 1)
45              ( setq res ( palindrom ( read 1 ) ) )
46          )
47      )
48      ( return res )
49    )
50  )
```