

## Задание 17. Генератор кода для машины 8080

Дополните ваш транслятор заключительным фрагментом – генератором кода, который по списку атомов выдает код для машины 8080.

**Вход/выход:** Входные данные (программа на языке MiniC) считываются из файла, ассемблерный код на языке машины 8080 записывается в результирующий файл. Ваш транслятор должен вывести: список атомов, таблицу символов, таблицу строк и результирующий код на языке ассемблера 8080.

Перед выполнением задания ознакомьтесь с комментариями в конце упражнения.

### Пример работы программы

*Обратите внимание: в примере вывода ниже ассемблерный код записан в три колонки. Это сделано для экономии места. Ваш транслятор может так не делать.*

Программа в файле prog.minic	
<pre>int sqRoots(int x, int y, int z){     int result;     result = y*y - 4*x*z;     if (result &lt; 0){         out "No real roots";     } else {         if (result == 0)             out "One root";         else             out "Two roots";     }     return result; }  int main(){     int a, b, c, d;     in a;     in b;     in c;     d = sqRoots(a, b, c);     return 0; }</pre>	
Вывод в файл prog.asm	
Syntax OK	
ATOMS	
-----	
0	(MUL, 2, 2, 5)
0	(MUL, '4', 1, 7)
0	(MUL, 7, 3, 8)
0	(SUB, 5, 8, 6)
0	(MOV, 6, , 4)
0	(MOV, '1', , 9)
0	(LT, 4, '0', L2)
0	(MOV, '0', , 9)
0	(LBL, , , L2)
0	(EQ, 9, '0', L0)
0	(OUT, , , S0)
0	(JMP, , , L1)
0	(LBL, , , L0)
0	(MOV, '1', , 10)
0	(EQ, 4, '0', L5)
0	(MOV, '0', , 10)
0	(LBL, , , L5)
0	(EQ, 10, '0', L3)

```

0      (OUT, , , S1)
0      (JMP, , , L4)
0      (LBL, , , L3)
0      (OUT, , , S2)
0      (LBL, , , L4)
0      (LBL, , , L1)
0      (RET, , , 4)
0      (RET, , , '0')
11     (IN, , , 12)
11     (IN, , , 13)
11     (IN, , , 14)
11     (PARAM, , , 12)
11     (PARAM, , , 13)
11     (PARAM, , , 14)
11     (CALL, 0, , 16)
11     (MOV, 16, , 15)
11     (RET, , , '0')
11     (RET, , , '0')

```

#### SYMBOL TABLE

```

-----
code   name      kind   type   len   init   scope  offset
0      sqRoots  func   int    3     0      -1     -1
1      x        var    int    None  0      0      20
2      y        var    int    None  0      0      18
3      z        var    int    None  0      0      16
4      result  var    int    None  0      0      12
5      [tmp1]  var    int    None  0      0      10
6      [tmp2]  var    int    None  0      0      8
7      [tmp3]  var    int    None  0      0      6
8      [tmp4]  var    int    None  0      0      4
9      [tmp5]  var    int    None  0      0      2
10     [tmp6]  var    int    None  0      0      0
11     main    func   int    0     0      -1     -1
12     a        var    int    None  0      11     8
13     b        var    int    None  0      11     6
14     c        var    int    None  0      11     4
15     d        var    int    None  0      11     2
16     [tmp7]  var    int    None  0      11     0

```

#### STRING TABLE

```

-----
0      No real roots\n
1      One root\n
2      Two roots\n

```

#### ASM 8080 code

```

-----
ORG 8000H
str0: DB 'No real roots', 0
str1: DB 'One root', 0
str2: DB 'Two roots', 0
ORG 0
LXI H, 0
SPHL
CALL main
END
@MULT:
; Code for MULT library function
@PRINT:
; Code for PRINT library function

sqRoots:
MVI A, 0
CMP B

; (MOV, '0', , 9)
MVI A, 0
LXI H, 2
DAD SP
MOV M, A
; (LBL, , , L2)
LBL2:
; (EQ, 9, '0', L0)
MVI A, 0
MOV B, A
LXI H, 2
DAD SP
MOV A, M
CMP B

main:
LXI B, 0
PUSH B
PUSH B
PUSH B
PUSH B
PUSH B
IN 0
LXI H, 8
DAD SP
MOV M, A
; (IN, , , 12)
IN 0
; (IN, , , 13)
IN 0

```

LXI B, 0	JZ LBL0	LXI H, 6
PUSH B	; (OUT, , , S0)	DAD SP
PUSH B	LXI A, str0	MOV M, A
PUSH B	CALL @print	; (IN, , , 14)
PUSH B	; (JMP, , , L1)	IN 0
PUSH B	JMP LBL1	LXI H, 4
PUSH B	; (LBL, , , L0)	DAD SP
PUSH B	LBL0:	MOV M, A
; (MUL, 2, 2, 5)	; (MOV, '1', , 10)	; (CALL, 0, , 16)
LXI H, 18	MVI A, 1	PUSH B
DAD SP	LXI H, 0	PUSH D
MOV A, M	DAD SP	PUSH H
MOV D, A	MOV M, A	PUSH PSW
LXI H, 18	; (EQ, 4, '0', L5)	LXI B, 0
DAD SP	MVI A, 0	PUSH B
MOV A, M	MOV B, A	LXI B, 0
CALL @MULT	LXI H, 12	LXI H, 4
MOV A, C	DAD SP	DAD SP
LXI H, 10	MOV A, M	MOV A, M
DAD SP	CMP B	MOV C, A
MOV M, A	JZ LBL5	PUSH B
; (MUL, '4', 1, 7)	; (MOV, '0', , 10)	LXI B, 0
LXI H, 20	MVI A, 0	LXI H, 6
DAD SP	LXI H, 0	DAD SP
MOV A, M	DAD SP	MOV A, M
MOV D, A	MOV M, A	MOV C, A
MVI A, 4	; (LBL, , , L5)	PUSH B
CALL @MULT	LBL5:	LXI B, 0
MOV A, C	; (EQ, 10, '0', L3)	LXI H, 8
LXI H, 6	MVI A, 0	DAD SP
DAD SP	MOV B, A	MOV A, M
MOV M, A	LXI H, 0	MOV C, A
; (MUL, 7, 3, 8)	DAD SP	PUSH B
LXI H, 16	MOV A, M	CALL sqRoots
DAD SP	CMP B	POP B
MOV A, M	JZ LBL3	POP B
MOV D, A	; (OUT, , , S1)	POP B
LXI H, 6	LXI A, str1	POP B
DAD SP	CALL @print	MOV A, B
MOV A, M	; (JMP, , , L4)	LXI H, 0
CALL @MULT	JMP LBL4	DAD SP
MOV A, C	; (LBL, , , L3)	MOV M, A
LXI H, 4	LBL3:	POP PSW
DAD SP	; (OUT, , , S2)	POP H
MOV M, A	LXI A, str2	POP D
; (SUB, 5, 8, 6)	CALL @print	POP B
LXI H, 4	; (LBL, , , L4)	; (MOV, 16, , 15)
DAD SP	LBL4:	LXI H, 0
MOV A, M	; (LBL, , , L1)	DAD SP
MOV B, A	LBL1:	MOV A, M
LXI H, 10	; (RET, , , 4)	LXI H, 2
DAD SP	LXI H, 12	DAD SP
MOV A, M	DAD SP	MOV M, A
SUB B	MOV A, M	; (RET, , , '0')
LXI H, 8	LXI H, 22	MVI A, 0
DAD SP	DAD SP	LXI H, 12
MOV M, A	MOV M, A	DAD SP
; (MOV, 6, , 4)	POP B	MOV M, A
LXI H, 8	POP B	POP B
DAD SP	POP B	POP B
MOV A, M	POP B	POP B
LXI H, 12	POP B	POP B
DAD SP	POP B	POP B

<pre> MOV M, A ; (MOV, '1', , 9) MVI A, 1 LXI H, 2 DAD SP MOV M, A ; (LT, 4, '0', L2) MVI A, 0 MOV B, A LXI H, 12 DAD SP MOV A, M CMP B JM LBL2 </pre>	<pre> POP B RET ; (RET, , , '0') MVI A, 0 LXI H, 22 DAD SP MOV M, A POP B POP B POP B POP B POP B POP B POP B POP B POP B POP B RET </pre>	<pre> RET ; (RET, , , '0') MVI A, 0 LXI H, 12 DAD SP MOV M, A POP B POP B POP B POP B POP B RET </pre>
--	--	--

## План выполнения задания

### Этап 1. Таблица символов

Дополните класс таблицы символов двумя вспомогательными функциями:

- `int getM(int scope) const` – возвращает количество локальных и временных переменных в области видимости `scope`;
- `void calculateOffset()` – подсчитывает и сохраняет значения поля `offset` для всей таблицы символов;
- `std::vector<std::string> functionNames() const` – возвращает список с именами всех функций, зарегистрированных в таблице символов.

Варианты реализации этих функций описаны в Комментариях ниже.

### Этап 2. Вспомогательные функции генерации кода

Добавьте в класс `RValue` чистую виртуальную функцию `virtual void load(std::ostream & stream) const = 0`; Перегрузите этот метод в классах `MemoryOperand` и `NumberOperand`. Методы должны вывести в поток последовательность инструкций 8080, загружающих в регистр A соответствующее значение операнда. Протестируйте эти методы.

Добавьте в класс `MemoryOperand` метод `void save(std::ostream & stream) const`, который должен выводить в поток последовательность инструкций 8080, записывающих значение регистра A в месте, на которое указывает операнд. Протестируйте метод.

Добавьте в таблицу символов метод `void generateGlobals(std::ostream & stream) const`, который для каждой глобальной переменной из таблицы символов выводит строку

```
varX: DB N
```

где X – код идентификатора, N – значение по умолчанию. Протестируйте работу метода.

Добавьте в таблицу строк метод `void generateStrings(std::ostream & stream) const`, который для каждой строки из таблицы выводит в поток строку

```
strX: DB 'значение_строки', 0
```

где X – порядковый номер строки, значение строки дается в одинарных кавычках, после него идет нулевой байт – признак конца строки. Протестируйте метод.

Дополните класс транслятора следующими вспомогательными функциями:

- `void saveRegs(std::ostream & stream)` – генерирует инструкции 8080, сохраняющие значения всех четырех регистровых пар в стек;
- `void loadRegs(std::ostream & stream)` – генерирует инструкции 8080, восстанавливающие со стека значения всех четырех регистровых пар.

### Этап 3. Полиморфные атомы

Дополните класс `Atom` чистым виртуальным методом `virtual void generate(std::ostream & stream) const = 0`, который, будучи вызванным для конкретного объекта-атома, сгенерирует нужный код на языке 8080, используя функции с этапа 2.

Теперь следует реализовать этот метод для всех атомов. Чтобы можно было проводить тестирование проекта рекомендуется сначала добавить во все атомы пустую реализацию этой функции (чтобы проект компилировался), а затем по очереди реализовывать и тестировать её для разных атомов.

Обратите внимание, что генерация кода для различных бинарных операторов отличается: для операций ADD, SUB, AND и OR генератор должен сгенерировать код из одной инструкции ассемблера (не считая кода загрузки операндов и сохранения результата, который генерируется соответствующими методами классов `MemoryOperand` и `NumberOperand`), в то время как для реализации атомов MUL и DIV нужно генерировать код для вызова библиотечной функции. Поэтому, создайте два новых класса – класс `SimpleBinaryOpAtom` и `FnBinaryOpAtom`, унаследовав их от класса `BinaryOpAtom`. В классе `BinaryOpAtom` объявите чистый виртуальный метод `virtual void generateOperation(std::ostream & stream) const = 0`, который будет генерировать код, непосредственно выполняющий операцию. Перегрузите и реализуйте этот метод в классах `SimpleBinaryOpAtom` и `FnBinaryOpAtom`. Обратите внимание, что названия ассемблерных инструкций и имена функций совпадают с именем атома, которое хранится в поле `_name`, это сводит генерацию кода к подстановке этого имени в соответствующую строку.

В классе `BinaryOpAtom` реализуйте метод `generate()`, который должен:

- 1) выдать комментарий с именем атома (см. комментарий 1);
- 2) сгенерировать код загрузки второго операнда, используя его метод `load()`;
- 3) добавить инструкцию перемещения значения из регистра A в регистр B;
- 4) сгенерировать код загрузки первого операнда;
- 5) вызвать только что реализованный метод `generateOperation()` для генерации инструкций выполнения операции;
- 6) вызвать метод `save()` для генерации кода сохранения результата.

Не забывайте тестировать методы генерации кода по мере их реализации.

Аналогично следует поступить и при генерации кода для условных переходов: создайте класс `SimpleJumpAtom` для реализации атомов EQ, NE, GT и LT, и класс `ComplexJumpAtom` для реализации GE и LE. Оба класса унаследуйте от `ConditionalJumpAtom`. Вынесите генерацию различающейся части кода в виртуальный метод.

### Этап 4. Генерация кода

Добавьте в класс `Translator` метод `void generateProlog(std::ostream & stream)`, который генерирует инструкции загрузчика и библиотечных функций (для краткости вывода код библиотечных функций `@MULT` и `@PRINT` можно опускать):

```
ORG 0
LXI H, 0
SPHL
CALL main
END
@MULT:
; Code for MULT library function
@PRINT:
; Code for PRINT library function
```

Добавьте в класс `Translator` метод `void generateFunction(std::ostream & stream, std::string function)`, который генерирует код для функции с именем `function` следующим образом:

- a) поставить метку с именем функции;
- b) сгенерировать инструкции подготовки стека, которые положат на него `m` нулевых значений:

```
LXI B, 0
; m раз делаем PUSH B
PUSH B
```

```
; ...  
PUSH B
```

- c) сгенерировать код для каждого атома функции, вызвав методы `generate()` атомов, сгенерированных для этой функции.

Обратите внимание: после генерации кода для всех атомов больше ничего делать не нужно. Все, что было положено на стек в начале работы функции, будет снято кодом, сгенерированным для атома `RET`. Даже если в программе на языке MiniC нет оператора `return` внутри функции, в соответствии с правилом №2 транслирующей грамматики основных конструкций языка атом (`RET ,, 0`) автоматически (и независимо от наличия операторов `return`) добавляется в качестве последнего атома функции.

Дополните класс транслятора заключительным методом `void generateCode(std::ostream & stream)`, который сгенерирует ассемблерный код для всей программы:

- непосредственно перед генерацией кода вызовите метод `calculateOffset()` таблицы символов, чтобы заполнить таблицу корректными значениями смещений к локальным и временным переменным;
- сгенерируйте инструкцию `ORG 8000H`;
- вызовите метод `generateGlobals()` таблицы символов, чтобы сгенерировать строки выделения памяти под глобальные переменные;
- вызовите метод `generateStrings()` таблицы строк, чтобы создать код для инициализации строковых констант;
- вызовите метод `generateProlog()` для генерации инструкций загрузчика и библиотечных функций;
- далее в цикле вызовите метод `generateFunction()` для каждой функции из таблицы символов.

Доработайте функцию `main()` в проекте, чтобы она запускала генератор кода.

## Оценка задания 17

Этап 1. Таблица символов	20 %
Этап 2. Вспомогательные функции	30 %
Этап 3. Полиморфные атомы	30 %
Этап 4. Генерация кода	20 %

## Комментарии к заданиям

- Для удобства анализа результата ваш транслятор при генерации кода должен печатать в комментариях те атомы, для которых он генерирует код (как в примере выше).
- На сайте <http://prog.tversu.ru/cs4.html> выложен код транслятора на языке Питон. Вы можете использовать его для сравнения результатов трансляции им и вашей программой, а также для изучения его внутреннего устройства. Необходимо иметь в виду, что учебный транслятор имеет структуру классов отличную от той, которая должна быть в вашем, поэтому дословно переводить на C++ фрагменты кода из него нельзя. (Генератор кода учебного транслятора реализован **частично** – он генерирует код только для тех атомов, которые используются в примере к данному упражнению.)
- Один из вариантов работы метода `getM()`: подсчитать общее количество переменных в области видимости заданной функции и вычесть из него количество параметров этой функции, записанное в столбце `len` таблицы символов.
- Вычисление смещения, куда записывать результат работы функции при трансляции оператора `return`, выполняется по формуле:

$$res = 2(m + n + 1),$$

где  $n$  – количество входных параметров функции (берется из поля `len` таблицы символов),  $m$  – количество локальных и временных переменных функции (вычисляется методом `getM`).

5. Вычисление смещения для каждой локальной/временной переменной и входного параметра функции производится по формуле:

$$var_i = \begin{cases} 2(m + n + 1 - i), & \text{если } i \leq n, \\ 2(m + n - i), & \text{если } i > n, \end{cases}$$

где  $i$  – порядковый номер локальной/временной переменной или параметра в таблице символов (начиная с 1).