Задание 16.2 Построение полного транслятора с языка MiniC на язык атомов

Для транслирующей грамматики языка MiniC построить транслятор на язык атомов методом рекурсивного спуска. После списка атомов ваша программа должна печатать таблицу символов с вашими и временными переменными, а также таблицу строковых констант. Транслятор должен использовать лексический анализатор языка MiniC.

Описания языка атомов и транслирующей грамматики языка MiniC можно загрузить с сайта по адресу http://prog.tversu.ru/cs4.html

Вход/выход: Входные данные считываются из файла, список атомов записывается в результирующий файл.

Пример работы программы

Программа в файле prog.minic

```
int sqRoots(int x, int y, int z){
      int result;
      result = y*y - 4*x*z;
      if (result < 0){
        out "No real roots\n";
      } else {
        if (result == 0)
           out "One root\n";
        else
           out "Two roots\n";
      }
      return result;
}

int main(){
  int a, b, c, d;
  in a;
  in b;
  in c;
  d = sqRoots(a, b, c);
  return 0;
}
```

Программа в файле prog.atom

```
Syntax OK
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       -1
2       y       var     int     None    0       0       -1
3       z       var     int     None    0       0       -1
4       result  var     int     None    0       0       -1
5       [tmp1]  var     int     None    0       0       -1
6       [tmp2]  var     int     None    0       0       -1
7       [tmp3]  var     int     None    0       0       -1
8       [tmp4]  var     int     None    0       0       -1
9       [tmp5]  var     int     None    0       0       -1
10      [tmp6]  var     int     None    0       0       -1
11      main    func    int     0       0       -1      -1
12      a       var     int     None    0       11      -1
13      b       var     int     None    0       11      -1
14      c       var     int     None    0       11      -1
15      d       var     int     None    0       11      -1
16      [tmp7]  var     int     None    0       11      -1
STRING TABLE
------------
0       No real roots\n
1       One root\n
2       Two roots\n
```

План выполнения задания

Этап 1. Доработка таблицы символов

1. Создайте внутри структуры `TableRecord` перечисления для видов символов: `enum class RecordKind {unknown, var, func}` и типов символов: `enum class RecordType {unknown, integer, chr}`.
2. Нам потребуется хранить идентификаторы области видимости. Такой идентификатор является целым числом, но для того, чтобы при работе с аргументами функций и переменными было понятно, что это не просто число, а именно идентификатор области видимости, введём для него специальный тип `Scope`, объявив его синонимом типа `int` с помощью оператора `typedef`:

   ```
   typedef int Scope;
   ```

   Добавьте эту строчку в начало файла SymbolTable.h. После неё определите константу для глобальной области видимости:

   ```
   const Scope GlobalScope = -1;
   ```
3. Добавьте в структуру `TableRecord` записей таблицы символов новые поля:
   a) `RecordKind _kind` – вид символа, значение по умолчанию: `unknown`;
   b) `RecordType _type` – тип символа, значение по умолчанию: `unknown`;
   c) `int _len` – число аргументов функции, значение по умолчанию: -1;
   d) `int _init` – начальное значение для переменных, значение по умолчанию: 0;

a) `Scope _scope` – контекст, значение по умолчанию: `GlobalScope`;
b) `int _offset` – смещение (будет использовано на этапе генерации кода, пока -1).

4. Скорректируйте оператор вывода и тесты для таблицы символов.

Этап 2. Дополнительные классы атомов

1. Аналогично атомам, реализованным в прошлом задании, создайте три новых класса для атомов: `CallAtom`, `RetAtom` и `ParamAtom`, унаследовав их от класса `Atom`:
   a) атом `CallAtom` получает и сохраняет два операнда типа `MemoryOperand` – вызываемую функцию и переменную для размещения результата;
   b) атом `RetAtom` получает операнд типа `RValue`, который определяет возвращаемое значение;
   c) атом `ParamAtom` получает операнд типа `RValue`, который задаёт значение, передаваемое в функцию.
2. Реализуйте тесты для этих атомов и протестируйте их.

Этап 3. Контекст / область видимости

1. Добавьте параметр `Scope scope` в метод `generateAtom` транслятора. Преобразуйте вектор `_atoms` в `std::map<Scope, std::vector<std::unique_ptr<Atom>>>`, где ключом является номер области видимости. Атомы должны добавляться в список, соответствующий переданной области видимости.
2. Добавьте параметр `Scope scope` в метод `alloc()` таблицы символов, и во все функции парсера, соответствующие правилам грамматики в трансляторе. Везде, где потребуется (в том числе в тестах), установите этот параметр в `GlobalScope`.
3. Добавьте в существующие тесты assert'ы, которые бы проверяли, что создаваемые атомы в списке атомов транслятора и переменные в таблице символов приписываются к контексту `GlobalScope`.
4. Сделайте тест, который моделирует разбор выражения не в глобальном контексте: напишите выражение на языке MiniC, в результате которого должны быть сформированы несколько атомов и созданы временные переменные. Вызовите для его разбора функцию `E()` с контекстом, отличным от `GlobalScope`, и проверьте, что атомы и переменные отнесены к правильному контексту.
5. Реализуйте в классе `SymbolTable` методы (тестируйте их в процессе добавления):
   a)
   ```
   std::shared_ptr<MemoryOperand> addVar(const std::string & name,
                   const Scope scope,
                   const TableRecord::RecordType type,
                   const int init = 0);
   ```
   Вызывается для объявления переменной или параметра функции.
   Параметры:
   • name – имя идентификатора;
   • scope – контекст, в котором происходит объявление: `GlobalScope` для глобального или код функции в таблице символов для локального;
   • type – тип переменной;
   • init – начальное значение.
   Проверяет, что в таблице символов нет идентификатора с совпадающими scope и именем name. Если это так, то создается новая запись, для которой прописываются переданные type, init, scope и `_kind = RecordKind::var`, после чего для этой записи конструируется `MemoryOperand` и возвращает указатель `std::shared_ptr` на него. Иначе возвращается `nullptr` (идентификатор уже занят в данном контексте).

   b)
   ```
   std::shared_ptr<MemoryOperand> addFunc(const std::string & name,
                   const TableRecord::RecordType type,
                   const int len);
   ```
   Вызывается для объявления имени функции. Работает только в глобальном контексте. Если идентификатор `name` уже объявлен (как функция или переменная), то возвращает `nullptr`.
   Работает аналогично `addVar`, но устанавливает `_kind = RecordKind::func` и количество параметров `len`.

   c)
   ```
   std::shared_ptr<MemoryOperand> checkVar(const Scope scope,
                                           const std::string &name);
   ```
   Вызывается для нахождения объявленной переменной с заданным именем name в текущем локальном или глобальном контекстах. Параметры:

• scope – код контекста;
• name – имя идентификатора;
Функция работает по следующему алгоритму:

i) ищет в контексте scope идентификатор с именем name; если не находит и scope != `GlobalScope`, то повторяет поиск для scope = `GlobalScope`;
ii) если найденный идентификатор не объявлен, то возвращает `nullptr`;
iii) если объявлен, но это не переменная, то возвращает `nullptr;`
iv) иначе (это переменная и она объявлена) создаёт для этой записи `MemoryOperand` и возвращает указатель `std::shared_ptr` на него.

d)
```
std::shared_ptr<MemoryOperand> checkFunc(const std::string &name,
                                         int len);
```
Вызывается для проверки, является ли переданный идентификатор именем объявленной функции.
Работает аналогично методу `checkVar()`, за исключением двух особенностей:

- поиск производится только в глобальном контексте;
- проверяется совпадение количества переданных аргументов (`len`) с количеством параметров, заданном при объявлении функции. Если количество аргументов не совпадает, возвращается `nullptr`.

6. Удалите из класса `SymbolTable` метод `add()`. Замените его использования на метод `addVar()`.
7. Проверьте работу всех тестов.

Этап 4. Доработка транслятора

При доработке транслятора следуйте методике разработке и тестирования, описанной в предыдущем задании.

1. Добавьте в класс `MemoryOperand` метод `int index() const`, возвращающий индекс записи из таблицы символов.
2. Реализуйте правила грамматики, обрабатывающие объявление переменных и функций (правила 1..14 и 16 из раздела «Транслирующая грамматика основных конструкций языка MiniC»). При реализации операции установки нового контекста (C'=addFunc(q,p,n) в грамматике) используйте метод `index()` класса `MemoryOperand`.
   Для передачи и возврата в функциях рекурсивного спуска значений атрибутов с информацией о типе объявляемой переменной используйте перечисление `RecordType`. При этом возврат значения `RecordType::unknown` означает ошибку трансляции.
   Для атрибута $name$ (имя символа) в правиле $DeclareStmt \rightarrow Type_p\ id_{name}\ DeclareStmt'_{qr}$ используйте тип данных `std::string`. Передавайте его строкой вниз функциям рекурсивного спуска, пока он не будет преобразован в `std::shared_ptr<MemoryOperand>` функцией `addFunc()` или `addVar()`.
3. Реализуйте оставшиеся нереализованными в предыдущем задании правила транслирующей грамматики выражений (правила, которые используют атомы `CallAtom`, `RetAtom`, `ParamAtom` и переменные).
   При реализации правил 28 – 31, передавайте значение наследуемого атрибута с именем переменной или функции, используя тип данных `std::string`.
   При реализации правил 32 – 35 для значений наследуемых атрибутов, использующихся для подсчёта числа аргументов в вызове функции, используйте тип `int`.
4. Реализуйте остальные правила грамматики.
   Для реализации правил, обеспечивающих работу оператора switch/case (правила 44 – 49), добавьте операторы сравнения в класс `LabelOperand`.
5. Добавьте в класс транслятора метод `bool translate()`, который будет запускать трансляцию, вызывая метод для стартового нетерминала `StmtList()`.
6. Реализуйте функцию main(), в которой вызовите транслятор для разбора данных из файла и распечатайте полученный список атомов и таблицы символов и строк.
   Если возникает синтаксическая или лексическая ошибка должно выводиться соответствующее сообщение.

Этап 5. Сообщения о синтаксических ошибках

Усовершенствуйте подсистему вывода информации о синтаксических ошибках. При возникновении ошибки транслятор должен печатать на экран правило грамматики, при разборе которого случилась ошибка, три последние корректно считанные и текущую лексемы, а также тип ошибки:

- кончились лексемы до окончания трансляции,
- закончилась трансляция, а лексемы еще имеются,
- текущая лексема не соответствует ожидаемой (необходимо выдать ожидаемую лексему).

Должна выводится ошибка, если в коде отсутствует функция main.

Все операторы (кроме объявления функций/переменных) должны быть строго внутри какой-либо функции. Если это не так, транслятор должен выдать ошибку.

Оценка задания 16.2

Задание 16.2.1. Доработка таблицы символов и дополнительные классы для атомов	10 %
Задание 16.2.2. Реализация контекста	20 %
Задание 16.2.3. Транслятор	70 %
Задание 16.2.4. Сообщения о синтаксических ошибках	10 %