УНИВЕРСИТЕТ ИТМО

Факультет программной инженерии и компьютерной техники

Направление подготовки 09.04.04 Программная инженерия

Дисциплина «Системное программное обеспечение»

Лабораторная работа №2

Вариант 4

Студент

Данилов П. Ю.

P4114

Преподаватель

Кореньков Ю. Д.

Санкт-Петербург, 2024 г.

Цель

Реализовать построение графа потока управления посредством анализа дерева разбора для набора входных
файлов. Выполнить анализ собранной информации и сформировать набор файлов с графическим
представлением для результатов анализа.

Задачи

1 Описать структуры данных, необходимые для представления информации о наборе файлов, наборе подпрограмм и графе потока управления, где:

a. Для каждой подпрограммы: имя и информация о сигнатуре, граф потока управления, имя исходного файла с текстом подпрограммы.

b. Для каждого узла в графе потока управления, представляющего собой базовый блок алгоритма подпрограммы: целевые узлы для безусловного и условного перехода (по мере необходимости), дерево операций, ассоциированных с данным местом в алгоритме, представленном в исходном тексте подпрограммы

2 Реализовать модуль, формирующий граф потока управления на основе синтаксической структуры текста подпрограмм для входных файлов

a. Программный интерфейс модуля принимает на вход коллекцию, описывающую набор анализируемых файлов, для каждого файла – имя и соответствующее дерево разбора в виде структуры данных, являющейся результатом работы модуля, созданного по заданию 1 (п. 3.b).

b. Результатом работы модуля является структура данных, разработанная в п. 1, содержащая информацию о проанализированных подпрограммах и коллекция с информацией об ошибках

c. Посредством обхода дерева разбора подпрограммы, сформировать для неё граф потока управления, порождая его узлы и формируя между ними дуги в зависимости от синтаксической конструкции, представленной данным узлом дерева разбора: выражение, ветвление, цикл, прерывание цикла, выход из подпрограммы – для всех синтаксических конструкций по варианту (п. 2.b)

d. С каждым узлом графа потока управления связать дерево операций, в котором каждая операция в составе текста программы представлена как совокупность вида операции и соответствующих операндов (см задание 1, пп. 2.d-g)

e. При возникновении логической ошибки в синтаксической структуре при обходе дерева разбора, сохранить в коллекции информацию об ошибке и её положении в исходном тексте

3 Реализовать тестовую программу для демонстрации работоспособности созданного модуля

a. Через аргументы командной строки программа должна принимать набор имён входных файлов, имя выходной директории

b. Использовать модуль, разработанный в задании 1 для синтаксического анализа каждого входного файла и формирования набора деревьев разбора

c. Использовать модуль, разработанный в п. 2 для формирования графов потока управления каждой подпрограммы, выявленной в синтаксической структуре текстов, содержащихся во входных файлах

d. Для каждой обнаруженной подпрограммы вывести представление графа потока управления в отдельный файл с именем “sourceName.functionName.ext” в выходной директории, по-умолчанию размещать выходной файлы в той же директории, что соответствующий входной

e. Для деревьев операций в графах потока управления всей совокупности подпрограмм сформировать граф вызовов, описывающий отношения между ними в плане обращения их друг к другу по именам и вывести его представление в дополнительный файл, по-умолчанию размещаемый рядом с файлом, содержащим подпрограмму main.

f. Сообщения об ошибке должны выводиться тестовой программной (не модулем, отвечающим за анализ!) в стандартный поток вывода ошибок

4 Результаты тестирования представить в виде отчета, в который включить:

a. В части 3 привести описание разработанных структур данных

b. В части 4 описать программный интерфейс и особенности реализации разработанного модуля

c. В части 5 привести примеры исходных анализируемых текстов для всех синтаксических конструкций разбираемого языка и соответствующие результаты разбора

Описание работы

Текстовый файл с исходным текстом разбираемой программы. Пример содержимого файла:

```
def printf(p (of string array[10])) end
def x() end
def y() end
def z() end

def main(argc (of int), argv (of string array[10] array[10]))
   printf("1");

   while(1) {
       printf("2");

       while(2) {
           printf("3");

           if(3) then {
               printf("break\n");
           } else if (4) then {
               printf("continue\n");
           } else {
               printf("labb:\n");
               printf("4");
           }
       } end
```

```
      printf("5");

      if(5) then {
          printf("break\n");
      } else if (6) then {
          printf("continue\n");
      } else {
          printf("6");
      }

      printf("7");

      {
          printf("8");
          if (9) then
              printf("break\n");
      } while(10);

      printf("11");

      if(11) then {
          printf("break\n");
      } else if (12) then {
          printf("continue\n");
      } else {
          printf("12");
          printf("goto labb\n");
      }

      printf("13");

  } end

  printf("14");

  if (a > 3) then {
      printf("1\n");
  } else {
      printf("2\n");
      printf("goto lab\n");
  }
  printf("debug!\n");

  while(10) {
      if (a > b) then {
           i = i + 2;
      } else if (2 * 3) then {
          printf("break\n");
      } else {
          printf("continue\n");
      }
  } end

  if (1) then {
      if(2) then {
          if(3) then {
              x();
          } else {
              y();
          }
```

```
        } else {
            z();
        }
    }

    if (a > 3) then {
        printf("1\n");
    } else {
        printf("2\n");
        printf("22\n");
        printf("222\n");
    }
    printf("debug!\n");

    while (a < 3) {
        printf("3\n");
    } end

    printf("Hello!\n");
end
```

Пример исходного текста программы на разбираемом языке

Функция, осуществляющая разбор исходного файла в структуру, содержащую деревья:

```
Array *executionGraph(FilenameParseTree *input, int size);

struct ExecutionNode {
    char *desc;
    TreeNode *operationTree;
    ExecutionNode *defaultBranch;
    ExecutionNode *conditionalBranch;
    int id;
    int printed;
};

struct ExecutionInfo {
    char *name;
    TreeNode *funCalls;
    ExecutionNode *nodes;
    char *filename;
    TreeNode *funcSignature;
    char **errors;
    int errorsCount;
};
```

Функция разбора, а также структура результата

Аспекты реализации

Текстовый файл с описанием дерева операций введенной программы в формате flowchart. Пример:

```
flowchart TB
node340([Value: call: printf]) --> node341([Value: const: 1])
node345([Type: linked execution node id, Value: 339]) --> node340([Value:
call: printf])
node346([Type: linked execution node id, Value: 336]) --> node337([Value:
const: 1])
node176([Value: call: printf]) --> node177([Value: const: 14to lab])
node347([Type: linked execution node id, Value: 175]) --> node176([Value:
call: printf])
node171([Type: read]) --> node172([Value: value place 'a'])
node170([Type: GT]) --> node171([Type: read])
node170([Type: GT]) --> node173([Value: const: 3])
node348([Type: linked execution node id, Value: 169]) --> node170([Type:
GT])
node159([Value: call: printf]) --> node160([Value: const: 2\no l])
node349([Type: linked execution node id, Value: 158]) --> node159([Value:
call: printf])
node153([Value: call: printf]) --> node154([Value: const: goto lab\])
node350([Type: linked execution node id, Value: 152]) --> node153([Value:
call: printf])
node144([Value: call: printf]) --> node145([Value: const: debug!\n])
node351([Type: linked execution node id, Value: 143]) --> node144([Value:
call: printf])
node352([Type: linked execution node id, Value: 140]) --> node141([Value:
const: 10])
node353([Type: linked execution node id, Value: 99]) --> node100([Value:
const: 1])
...
```

Пример дерева операций в формате flowchart

Текстовый файл с описанием графа потока управления введенной программы в формате flowchart. Пример:

```
flowchart TB
node447([Type: source]) --> node446([Type: sourceItemList])
node446([Type: sourceItemList]) --> node13([Type: sourceItem])
node13([Type: sourceItem]) --> node12([Type: funcSignature, Value: printf])
node12([Type: funcSignature, Value: printf]) --> node10([Type: argList])
node10([Type: argList]) --> node9([Type: argListItems])
node9([Type: argListItems]) --> node8([Type: arg])
node8([Type: arg]) --> node2([Type: IDENTIFIER, Value: p])
node8([Type: arg]) --> node7([Type: array, Value: 10])
node7([Type: array, Value: 10]) --> node4([Type: TYPEDEF, Value: string])
node446([Type: sourceItemList]) --> node445([Type: sourceItemList])
node445([Type: sourceItemList]) --> node18([Type: sourceItem])
node18([Type: sourceItem]) --> node17([Type: funcSignature, Value: x])
node445([Type: sourceItemList]) --> node444([Type: sourceItemList])
node444([Type: sourceItemList]) --> node23([Type: sourceItem])
node23([Type: sourceItem]) --> node22([Type: funcSignature, Value: y])
node444([Type: sourceItemList]) --> node443([Type: sourceItemList])
node443([Type: sourceItemList]) --> node28([Type: sourceItem])
...
```

Пример графа потока управления в формате flowchart

Текстовый файл с описанием графа вызовов введенной программы в формате flowchart. Пример:

```
flowchart TB
node18([Type: operationTreeId, Value: 16]) --> node19([Type: call, Value:
printf])
node344([Type: function, Value: main]) --> node18([Type: operationTreeId,
Value: 16])
node27([Type: operationTreeId, Value: 25]) --> node28([Type: call, Value:
printf])
node344([Type: function, Value: main]) --> node27([Type: operationTreeId,
Value: 25])
node38([Type: operationTreeId, Value: 36]) --> node39([Type: call, Value:
printf])
node344([Type: function, Value: main]) --> node38([Type: operationTreeId,
Value: 36])
node47([Type: operationTreeId, Value: 45]) --> node48([Type: call, Value:
printf])
node344([Type: function, Value: main]) --> node47([Type: operationTreeId,
Value: 45])
node53([Type: operationTreeId, Value: 51]) --> node54([Type: call, Value:
printf])
node344([Type: function, Value: main]) --> node53([Type: operationTreeId,
Value: 51])
node59([Type: operationTreeId, Value: 57]) --> node60([Type: call, Value:
printf])
...
```

Пример графа вызовов в формате flowchart

Результаты

В результате выполнения работы:

1. Были изучены структуры разбора исходного текста: дерево операций, граф потока выполнения, граф вызовов функций.
2. Был реализован модуль формирования указанных графов на основе исходного текста программы.
3. Модуль был протестирован на различных примерах исходных текстов.

Исходный код разработанного решения: https://github.com/47iq/spo

Выводы

В итоге цель работы можно считать успешно выполненной. Модуль был реализован и протестирован. В ходе выполнения работы был изучен материал по рассматриваемым темам.