DOTTER.
БИБЛИОТЕКА ДЛЯ ВИЗУАЛИЗАЦИИ ДЕРЕВЬЕВ

Данила Байгушев*[1]*, 9 класс

Научный руководитель: И.Р. Дединский, МФТИ

При работе нелинейными структурами данных часто возникает не всегда тривиальная задача их визуализации. В частности, визуализация (распечатка, дамп) применяется для контроля правильности данных в структуре. В работе рассматривается техническая визуализация двоичных деревьев синтаксического разбора (AST). Для облегчения этой задачи была разработана библиотека, которая поможет максимально просто создать понятную и красивую визуализацию таких деревьев. Библиотека основывается на приложении dot пакета GraphWiz и облегчает описание графов на языке dot.

Введение

Во время восхождения на лестницу знаний, рано или поздно программист сталкивается с такой структурой данных, как двоичное, или бинарное, дерево. Удобнее всего работать с таким деревом при помощи специального класса. У каждого надежного объекта (класса) должны быть диагностические функции проверки состояния (назовем ее `OK`, она возвращает `true`, если с объектом все в порядке) и визуализации или распечатки (`Dump`, она выдает полную техническую информацию об объекте для поддержки отладки, на программистском жаргоне «дамп»). И если реализация первой функции не вызывает трудностей ~~(если, конечно, руки из правильного места растут)~~, то со второй возникают разные вопросы. Для примера мы будем разбирать деревья, полученные в результате синтаксического разбора выражений или программ (так называемые AST [1]).

Зачем нужна визуализация

В основном визуализацию используют для отлова ошибок. Вот пример типичной ошибки, допущенной программой синтаксического разбора при построении дерева (рис. 1):

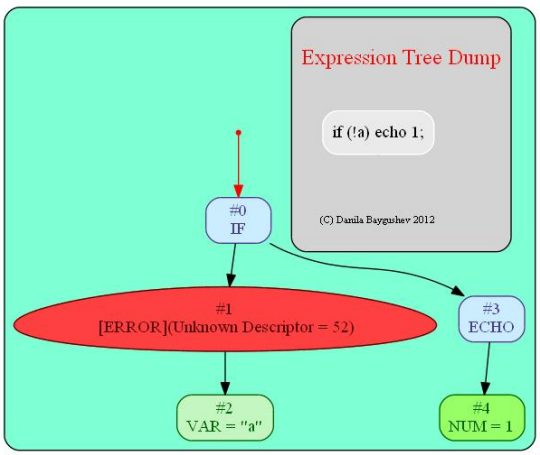

Рис. 1. Ошибочное дерево синтаксического разбора.

[1] E-mail для связи: *IDanila24@gmail.com*.

Из этой иллюстрации сразу видно, что ошибка допущена при разборе выражения «`!a`». Правильный вариант дерева должен был выглядеть так (рис. 2):

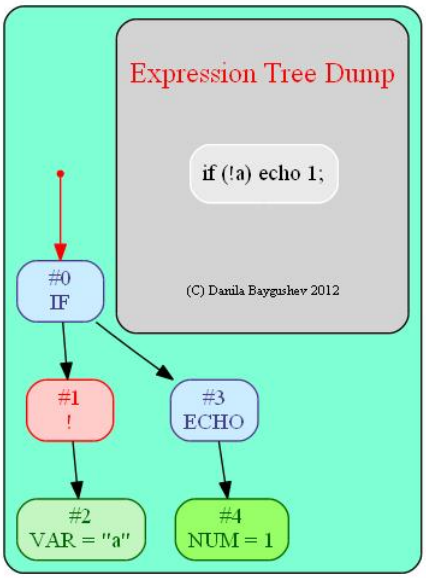

Рис. 2. Правильное дерево разбора для предыдущего примера.

Если ошибочную структуру дерева увидеть сразу и в наглядной форме, то ошибку легче найти и исправить. Это экономит часы (а иногда и дни) программистского труда.

Вот еще один пример дерева, полученного при разборе арифметического выражения (рис. 3):

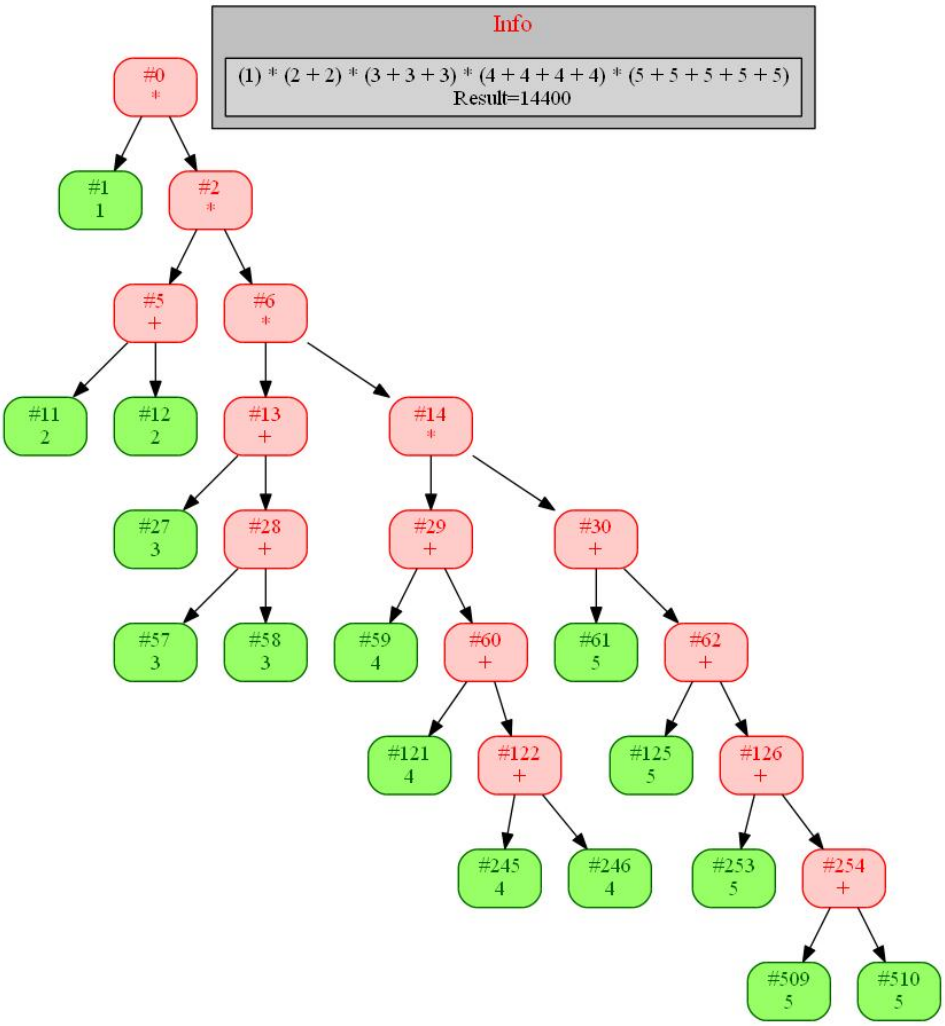

Рис. 3. Дерево разбора арифметического выражения.

Именно по этому нужно сделать визуализацию максимально наглядной и интуитивно понятной, иначе поиск ошибок становится сложным и долгим.

«Плохая» и «хорошая» визуализация

Чаще всего визуализация класса делается в форме текстовой распечатки, как в примере ниже (см. рис. 4). Но из этого же примера видно, что это не очень хорошо подходит для дерева, так как оно не является линейной структурой:

```
newTree // Good
{
    Size_         = 3;
    CurrentNode_  = 0;
    Data_ =
    {
        {+1/ \-1 Descriptor = 3 (NEW), Data = -1, Str = “” }, // [0]
        {-1/ \+2 Descriptor = 2 (VAR), Data = -1, Str = “a”}, // [1]
        {-1/ \-1 Descriptor = 1 (NUM), Data = 10, Str = “” }  // [2]
    };
}
```

Рис. 4. Пример текстовой распечатки дерева.

Что все же есть хорошего в этой «плохой» визуализации?

1. По комментарию возле `newTree` видно состояние класса. Для этого используется функция `OK`, входящая в класс узла дерева (см. рис. 5).
2. Выписаны значения всех элементов класса.
3. Распечатка выполнена в стиле C++. Вследствие этого, потом будет легко написать функцию, которая воссоздаёт (загружает) объект класса по такой распечатке.

```
bool newTree::OK()
{
    if (Size_ < 0)              return false; // Неверный размер
    if (CurrentNode_ < 0)       return false; // Неверный номер узла
    if (CurrentNode_ >= Size_) return false; // Номер узла за границей массива

    return Data_.OK();                        // Data_ - вектора со своим OK().
}
```

Рис. 5. Реализация функции `Tree::OK()`.

Как видно из примера, несмотря на все усилия, эта распечатка всё равно не очень понятна, так как не очень ясны связи между узлами. Только представьте, сколько бы вы искали ошибку дереве, полученном в результате синтаксического разбора несложной программы из четырех коротких функций (см. рис. 6).

Такие деревья лучше визуализировать графически, что позволяет максимально упростить создание понятного дампа для двоичного дерева.

Задача визуализации графов весьма старая и общая, и для ее решения было разработано множество средств. В частности, существуют языки описания графов, и программы, строящие по таким описаниям визуализации. Однако в этих описаниях не всегда просто разобраться.

Существуют и готовые программы для визуализации синтаксических деревьев. Одна из них, в частности, разработана учениками 10 класса А. Ивановым и Е. Никитенко в ходе изучения курса структур данных и основ машинной трансляции в 2011 г. [2]. Однако работа с такими программами требует соответствия форматам их входных данных, что по разным причинам бывает не всегда удобно.

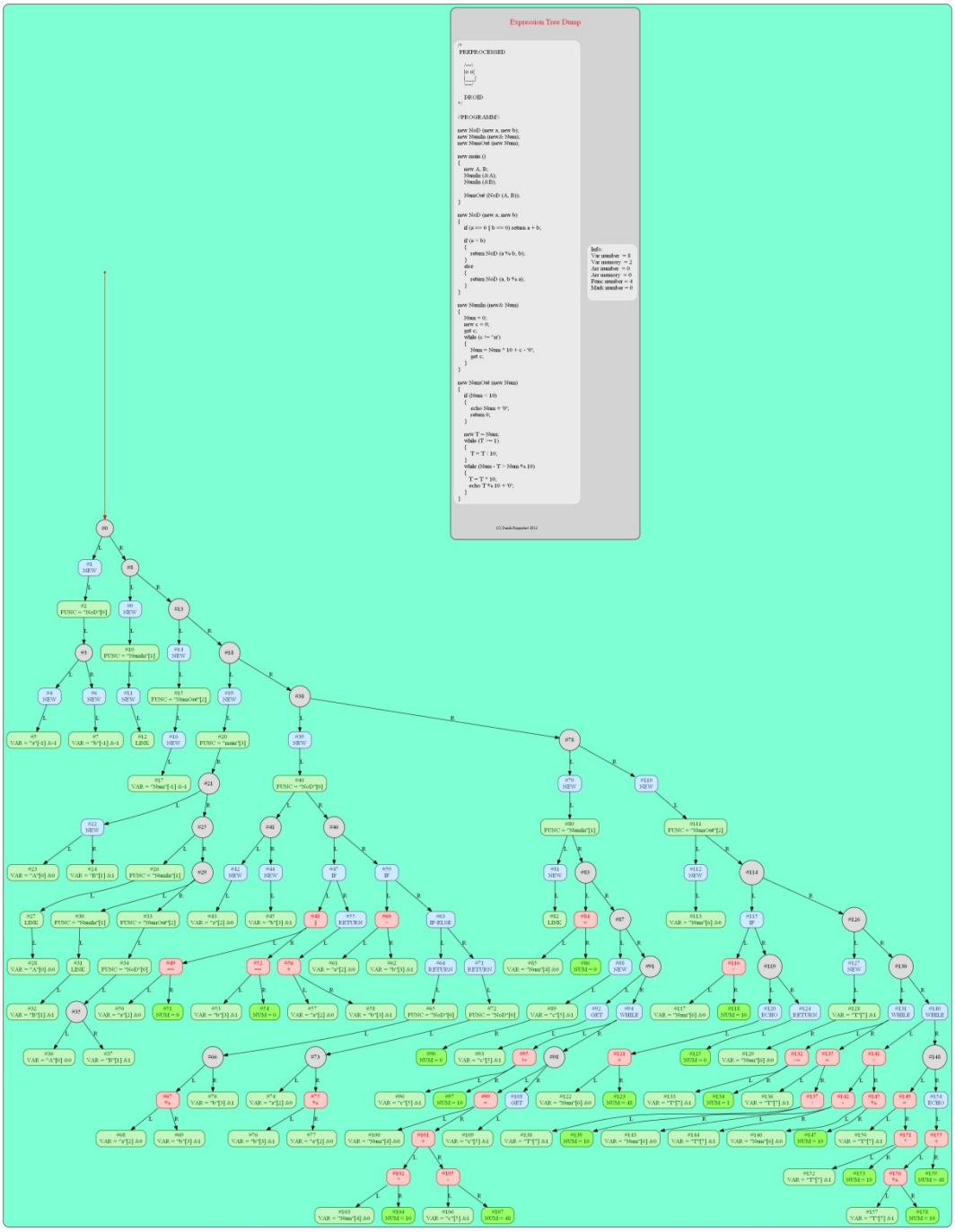

Рис. 6. Синтаксическое дерево простой программы (текст программы см. на рисунке).

Библиотека Dotter

Для облегчения работы по визуализации синтаксических деревьев автором была разработана библиотека `Dotter`. Она доступна для бесплатного некоммерческого использования по адресу https://dl.dropbox.com/u/16586422/PUBLIC_FILES/Dotter_Files/DotterInstaller.exe.

`Dotter` использует свободно распространяемую программу GraphWiz [3], строящую растровые изображения по текстовым описаниям на специальном языке dot [4]. Все иллюстрации к этой статье были созданы с ее помощью. Пользуясь ей, можно очень просто создать понятный графическую визуализацию вашего дерева. Как именно это сделать, можно посмотреть в документации библиотеки. В ней приведены примеры, один из которых показывает, как составить дерево выражения и создать его графическое представление. Вот пример, строящий небольшое дерево (рис. 7):

```
#include "Dotter.h"

int main()
{
    dtBegin ("Example3.dot");          // Начало dot-описания графа

    dtNodeStyle ("box");               // Устанавливаем стиль узлов
    dtLinkStyle ("dashed");            // Устанавливаем стиль связей

    dtNode (0, "i = 0\n value = 1");   // Рисуем узел 0
    dtNode (1, "i = 1\n value = 3");   // Рисуем узел 1
    dtNode (2, "i = 2\n value = 9");   // Рисуем узел 2
    dtNode (3, "i = 3\n value = 5");   // Рисуем узел 3
    dtNode (4, "i = 4\n value = 6");   // Рисуем узел 4
    dtNode (5, "i = 5\n value = 4");   // Рисуем узел 5
    dtNode (6, "i = 6\n value = 4");   // Рисуем узел 6

    dtLink (0, 1);                     // Рисуем связь узла 0 с 1
    dtLink (0, 2);                     // Рисуем связь узла 0 с 2
    dtLink (0, 3);                     // Рисуем связь узла 0 с 3
    dtLink (3, 4);                     // Рисуем связь узла 3 с 4
    dtLink (3, 5);                     // Рисуем связь узла 3 с 5
    dtLink (5, 6);                     // Рисуем связь узла 5 с 6

    dtLinkStyle ("bold", "red");       // Другой стиль связи
    dtLink (4, 6, "Error");            // Рисуем связь узла 4 с 6

    dtEnd (file);                      // Конец dot-описания графа

    dtRender ("Example3.dot");         // Делаем из файла картинку
}
```

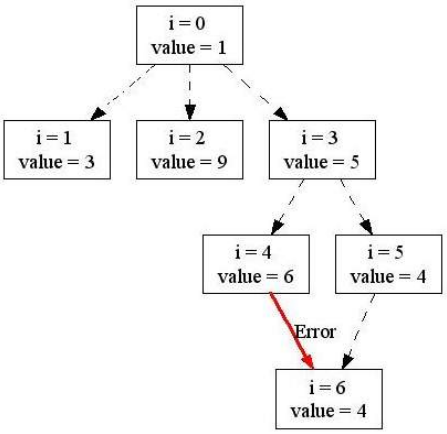

Рис. 7. Вверху: пример применения библиотеки для визуализации дерева. Внизу: результат визуализации.

Заключение

Автор надеется, что его работа поможет многим сэкономить время для поиска ошибок в сложных структурах данных. Скачать библиотеку можно по адресу https://dl.dropbox.com/u/16586422/PUBLIC_FILES/Dotter_Files/DotterInstaller.exe. Если у читателей возникнут пожелания и предложения, или кто-то найдет в ней ошибки, просьба писать автору по электронной почте IDanila24@gmail.com.

Литература

1. Абстрактное синтаксическое дерево. Материал из Википедии – свободной энциклопедии. http://ru.wikipedia.org/wiki/AST.
2. А. Иванов, Е. Никитенко. Визуализатор синтаксических деревьев. Неопубликованные данные.
3. Graphviz: Graph Visualization Software. http://www.graphviz.org.
4. DOT (язык). Материал из Википедии – свободной энциклопедии. http://ru.wikipedia.org/wiki/DOT_%28%D1%8F%D0%B7%D1%8B%D0%BA%29.