Университет ИТМО

Факультет ПИиКТ

Системное программное обеспечение

Лабораторная работа №1

Работу выполнил:

Абузов Ярослав

Группа:

P4114

Вариант:

1

Преподаватель:

Кореньков Ю. Д.

Санкт-Петербург

2024

Содержание

1 Цели

Использовать выбранное средство синтаксического анализа для реализации модуля, который выполняет разбор текста в соответствии с заданным языком. Разработать систему, которая строит синтаксическое дерево на основе исходного файла с текстом. Узлы синтаксического дерева должны соответствовать элементам синтаксической модели языка. Результирующее синтаксическое дерево должно быть сохранено в файл в формате, поддерживающем графическое представление дерева.

2 Задачи

1) Изучить выбранное средство синтаксического анализа
    a. Средство должно поддерживать программный интерфейс, совместимый с языком Си
    b. Средство должно параметризоваться спецификацией, описывающей синтаксическую структуру разбираемого языка
    c. Средство может функционировать посредством кодогенерации и/или подключения необходимых для его работы дополнительных библиотек
    d. Средство может быть реализовано с нуля, в этом случае оно должно использовать обобщённый алгоритм, управляемый спецификацией
2) Изучить синтаксис разбираемого по варианту языка и записать спецификацию для средства синтаксического анализа, включающую следующие конструкции:
    a. Подпрограммы со списком аргументов и возвращаемым значением
    b. Операции контроля потока управления – простые ветвления if-else и циклы или аналоги

c. В зависимости от варианта – определения переменных
d. Целочисленные, строковые и односимвольные литералы
e. Выражения численной, битовой и логической арифметики
f. Выражения над одномерными массивами
g. Выражения вызова функции

3) Реализовать модуль, использующий средство синтаксического анализа для разбора языка по варианту
   a. Программный интерфейс модуля должен принимать строку с текстом и возвращать структуру, описывающую соответствующее дерево разбора и коллекцию сообщений ошибке
   b. Результат работы модуля – дерево разбора – должно содержать иерархическое представление для всех синтаксических конструкций, включая выражения, логически представляющие собой иерархически организованные данные, даже если на уровне средства синтаксического анализа для их разбора было использовано линейное представление
4) Реализовать тестовую программу для демонстрации работоспособности созданного модуля
   a. Через аргументы командной строки программа должна принимать имя входного файла для чтения и анализа, имя выходного файла записи для дерева, описывающего синтаксическую структуру разобранного текста
   b. Сообщения об ошибке должны выводиться тестовой программной (не модулем, отвечающим за анализ!) в стандартный поток вывода ошибок
5) Результаты тестирования представить в виде отчета, в который включить:
   a. В части 3 привести описание структур данных, представляющих результат разбора текста (3a)

b. В части 4 описать, какая дополнительная обработка потребовалась для результата разбора, предоставляемого средством синтаксического анализа, чтобы сформировать результат работы созданного модуля
c. В части 5 привести примеры исходных анализируемых текстов для всех синтаксических конструкций разбираемого языка и соответствующие результаты разбора

3 Описание работы

Разработанная тестовая программа генерирует AST для файла с исходным кодом и создаёт файл формата dot для графического представления полученного дерева. Через аргументы командой строки программа принимает имя входного файла с исходным кодом (ключ `-i`), имя выходного файла для записи графического представления дерева (ключ `-o`) и флаг для вывода отладочных сообщений (ключ `-d`). Для получения справки необходимо ввести ключ `--help` (или `-?`).

```
root@1c2f4a9b58c5:/workspaces/sppo/lab-1#./build/main --help
Usage: main [OPTION...]
MyLang Parser Program

  -d, --debug                Enable debug output
  -i, --input=INPUTFILE      Input file
  -o, --output=OUTPUTFILE    Output file
  -?, --help                 Give this help list
      --usage                Give a short usage message

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.
```

Вывод справки по аргументам

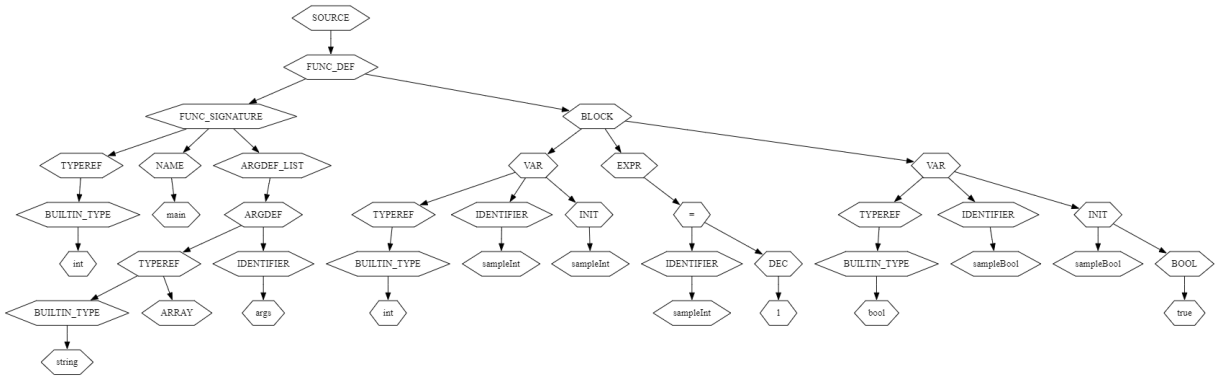
При указании корректного входного и выходного файлов программа создаст наполненный dot файл. Пример далее.

```
root@1c2f4a9b58c5:/workspaces/sppo/lab-1#./build/main -i ./testInput2 -o ./test.dot
```

Пример запуска программы с входных и выходным файлами без вывода отладочных сообщений

```
int main(string[] args) {
    int sampleInt;
    sampleInt = 1;
    bool sampleBool = true;
}
```

Пример входящего файла с исходным кодом



Визуализированное дерево, полученное в результате разбора входящего файла выше

```
root@1c2f4a9b58c5:/workspaces/sppo/lab-1# ./build/main -i ./testInput2 -o ./test.dot
-d
(SOURCE (FUNC_DEF (FUNC_SIGNATURE (TYPEREF (BUILTIN_TYPE int)) (NAME main)
(ARGDEF_LIST (ARGDEF (TYPEREF (BUILTIN_TYPE string) ARRAY) (IDENTIFIER args))))
(BLOCK (VAR (TYPEREF (BUILTIN_TYPE int)) (IDENTIFIER sampleInt) (INIT sampleInt))
(EXPR (= (IDENTIFIER sampleInt) (DEC 1))) (VAR (TYPEREF (BUILTIN_TYPE bool))
(IDENTIFIER sampleBool) (INIT sampleBool (BOOL true))))))
1
node SOURCE_0 [label="SOURCE"]
  node FUNC_DEF_1 [label="FUNC_DEF"]
    node FUNC_SIGNATURE_2 [label="FUNC_SIGNATURE"]
      node TYPEREF_3 [label="TYPEREF"]
        node BUILTIN_TYPE_4 [label="BUILTIN_TYPE"]
          node int_5 [label="int"]
      node NAME_6 [label="NAME"]
        node main_7 [label="main"]
      node ARGDEF_LIST_8 [label="ARGDEF_LIST"]
        node ARGDEF_9 [label="ARGDEF"]
          node TYPEREF_10 [label="TYPEREF"]
            node BUILTIN_TYPE_11 [label="BUILTIN_TYPE"]
              node string_12 [label="string"]
            node ARRAY_13 [label="ARRAY"]
          node IDENTIFIER_14 [label="IDENTIFIER"]
            node args_15 [label="args"]
    node BLOCK_16 [label="BLOCK"]
      node VAR_17 [label="VAR"]
        node TYPEREF_18 [label="TYPEREF"]
          node BUILTIN_TYPE_19 [label="BUILTIN_TYPE"]
            node int_20 [label="int"]
        node IDENTIFIER_21 [label="IDENTIFIER"]
          node sampleInt_22 [label="sampleInt"]
        node INIT_23 [label="INIT"]
          node sampleInt_24 [label="sampleInt"]
      node EXPR_25 [label="EXPR"]
        node ASSIGN_26 [label="="]
```

```
          node IDENTIFIER_27 [label="IDENTIFIER"]
            node sampleInt_28 [label="sampleInt"]
          node DEC_29 [label="DEC"]
            node 1_30 [label="1"]
      node VAR_31 [label="VAR"]
        node TYPEREF_32 [label="TYPEREF"]
          node BUILTIN_TYPE_33 [label="BUILTIN_TYPE"]
            node bool_34 [label="bool"]
        node IDENTIFIER_35 [label="IDENTIFIER"]
          node sampleBool_36 [label="sampleBool"]
        node INIT_37 [label="INIT"]
          node sampleBool_38 [label="sampleBool"]
          node BOOL_39 [label="BOOL"]
            node true_40 [label="true"]
```

Пример запуска программы с входных и выходным файлами с выводом отладочных сообщений

Тестовая программа использует два разработанных ранее модуля:

- Модуль разбора текста по заданной грамматике с построением AST
- Модуль отображения полученного AST в термины формата dot для визуализации

Более подробное описание модулей будет в аспектах реализации. Также внутри тестовой программы реализован разбор аргументов командой строки с помощью `argp`.

4 Аспекты реализации

4.1 Основной модуль разбора текста

Для реализации модуля разбора текста использовалось средство синтаксического анализа ANTLR v3. Для получения основного кода модуля разбора для генератора ANTLR была написана грамматика в соответствующих терминах необходимого языка. В качестве источника вдохновения служила грамматика для языка C из репозитория с примерами грамматик (examples-v3).

```
sourceItem
    : funcSignature (statementBlock | ';') -> ^(FUNC_DEF funcSignature
statementBlock?)
    ;

funcSignature
    : typeRef? identifier '(' argDefList ')' -> ^(FUNC_SIGNATURE typeRef? ^(NAME
identifier) argDefList)
```

```
    ;

argDefList
    : (argDef (',' argDef)*)? -> ^(ARGDEF_LIST argDef*)
    ;

argDef
    : typeRef? identifier -> ^(ARGDEF typeRef? ^(IDENTIFIER identifier))
    ;
```

Часть разработанной грамматики ANTLR

После использования генератора кода разбора из инструментария ANTLR на выходе были получены файлы для лексера и парсера разработанной грамматики – `MyLangLexer.c`, `MyLangLexer.h`, `MyLangParser.c`, `MyLangParser.h`, где MyLang – название грамматики.

Именно данные модули выполняют основную работу по разбору входящего текста, по определению ошибок и по построению AST. О последнем немного подробнее далее.

В ANTLR (в отличии, например, от bison) AST может быть построено автоматически, если в настройках грамматики указать соответствующий пункт (`output=AST`). Далее необходимо внутри грамматики «подсказать» ANTLR как именно строить дерево, в противном случае получится простой список токенов, а не иерархическое представление. Это реализуется с помощью таких операторов как: «^» для создания корневого узла (или же - поддерева), «!» для исключения токена из дерева. Также для вставки «несуществующих» токенов (для информативности внутри дерева) и перестановки поступающих токенов используется механизм rewriting rules (оператор «`->`»).

```
funcCall
    : identifier '(' exprList? ')' -> ^(FUNC_CALL ^(NAME identifier) exprList?)
    ;
```

Пример использования механизма rewriting rules

В качестве результата работы сгенерированный парсер возвращает дерево разбора `pANTLR3_BASE_TREE`. Для отлова ошибок необходимо заменить стандартную функцию обработчика и/или принтера ошибок

согласно документации ANTLR. Однако для начала необходимо создать структуры для хранения списка ошибок.

```
typedef struct ErrorNode {
  char *errorText;
  unsigned int errorLine;
  int errPosInLine;
  char *errTokenText;
  struct ErrorNode *next;
} ErrorNode;

typedef struct __attribute__((packed)) ErrorContext {
  unsigned int errorCount;
  ErrorNode *head;
} ErrorContext;
```

Структуры для хранения ошибок

Ошибки хранятся в связном списке. Для каждого нового вхождения создаётся структура `ErrorNode`, которая содержит:

- `errorText` – текст ошибки
- `errorLine` – номер строчки, где найдена ошибка
- `errPosInLine` – номер символа, где найдена ошибка
- `errTokenText` – описание ошибки от ANTLR
- `next` – ссылка на следующую ошибку

Для доступа к началу списка ошибок существует структура `ErrorContext`, которая содержит:

- `errorCount` – количество ошибок
- `head` – ссылка на первую ошибку в списке

Помимо указанных структур в файле `errorUtils.h` описаны объявления функций для создания списка ошибок, его удаления, для добавления ошибки, для отлова ошибок в парсере и лексере.

```
void initErrorContext(ErrorContext *context);

void destroyErrorContext(ErrorContext *context);

void addError(ErrorContext *context, const char *errorMsg,
              unsigned int errorLine, int errPosInLine, const char *errTokenText);
```

```
void printErrors(ErrorContext *context);

void extractRecognitionError(pANTLR3_BASE_RECOGNIZER recognizer,
                             pANTLR3_UINT8 *tokenNames);

void reportLexerError(pANTLR3_BASE_RECOGNIZER recognizer);
```

Функции в файле `errorUtils.h`

Для того, чтобы парсер мог добавлять ошибки в список, необходимо предоставить ему ссылку на структуру `ErrorContext` и заменить стандартную функцию вывода ошибок `displayRecognitionError`.

```
  parser->pParser->rec->state->userp = &result->errorContext;
  parser->pParser->rec->displayRecognitionError = extractRecognitionError;
```

Подготовка парсера для кастомной обработки ошибок

После выполнения разбора текста дерево и список ошибок будут сохранены в структуре `MyLangResult`, которая содержит:

- `tree` – полученное AST формата `MyAstNode` (создаётся путём копирования структуры и наполнения дерева ANTLR, описывать не буду)
- `errorContext` – список ошибок
- `isValid` – флаг наличия ошибок

Структура описана в файле `myLang.h`, который является верхнеуровневым модулем в данной цепочке. Он также содержит определения функций для запуска процесса парсинга с получением результата и освобождения памяти от структуры `MyLangResult`.

```
#include "ast/myAst.h"
#include "errorsUtils/errorUtils.h"
#include <stdbool.h>

typedef struct MyLangResult {
    MyAstNode *tree;
    ErrorContext errorContext;
    bool isValid;
} MyLangResult;

void parseMyLangFromFile(MyLangResult *result, char *filename, bool debug);
```

```
void parseMyLangFromText(MyLangResult *result, const char *text, bool debug);

void destroyMyLangResult(MyLangResult *result);
```

Файл `myLang.h`

Для хранения дерева разбора используется собственная структура MyAstNode, которая содержит:

- `children` – список детей узла
- `childCount` – количество детей узла
- `label` – текст узла
- `line` – номер строки с токеном в тексте
- `pos` – позиция в строке у данного токена
- `isImaginary` – является ли токен настоящим

```
typedef struct __attribute__((packed)) MyAstNode {
  struct MyAstNode **children;
  uint32_t childCount;
  const char *label;
  uint32_t line;
  uint32_t pos;
  bool isImaginary;
} MyAstNode;
```

Структура MyAstNode

4.2 Модуль преобразования AST в термины dot формата

Для визуализации дерева был выбран формат dot. Механизм преобразования включает в себя следующее:

- Копирование структуры дерева
- Присвоение уникального идентификатора каждому узлу
- Сохранение информации о количестве детей в узле
- Присвоение имени узлу для отображения

Описанные выше данные и ссылку на детей узла хранит структура `DotNode`.

```
typedef struct __attribute__((packed)) DotNode {
    struct DotNode** children;
    uint64_t id;
    uint32_t childCount;
```

```
    const char* label;
} DotNode;
```

Структура DotNode

В файле `dotUtils.h` также представлены определения функций для создания Dot-дерева, добавления узлов, копирования структуры из ANTLR дерева или MyAstNode дерева, генерации dot файла.

```
DotNode* newDotNode(uint64_t id, const char* label, uint32_t childCount);

void destroyDotNodeTree(DotNode* root);

DotNode* createDotTreeFromAntlrTree(pANTLR3_BASE_TREE root, uint64_t layer, uint64_t
*id, bool debug);

DotNode* createDotTreeFromMyTree(MyAstNode *root, uint64_t layer, uint64_t *id, bool
debug);

void writeTreeToDot(FILE *file, DotNode* root);

int generateDotFile(DotNode* root, const char *filename);

int generateDotFileFromAntlrTree(pANTLR3_BASE_TREE tree, const char *filename, bool
debug);

int generateDotFileFromMyTree(MyAstNode *tree, const char *filename, bool debug);
```

Функции в файле `dotUtils.h`

В файле `dotUtils.c` помимо реализаций вышеназванных функций, определены функции, которые производят определённые преобразования со значениями узлов оригинального дерева при копировании для соответствия возможностям dot файлов – замена специальных символов, удаление кавычек. В качестве примера приведена основная функция копирования структуры дерева по принципы preOrderTraversal.

```
DotNode *createDotTreeFromMyTree(MyAstNode *root, uint64_t layer, uint64_t *id,
                                 bool debug) {
  if (root == NULL) {
    return NULL;
  }

  uint64_t currentId = (*id)++;

  if (debug) {
```

```
    for (uint32_t i = 0; i < layer; i++) {
      printf("  ");
    }
    char *tokenText = removeQuotes(root->label);
    const char *nodeName = postProcessingNodeToken(tokenText);
    printf("node %s_%lu [label=\"%s\"]", nodeName, currentId, tokenText);
    printf("\n");
    free(tokenText);
  }

  char *label = removeQuotes(root->label);

  DotNode *newNode =
      newDotNode(currentId, (const char *)label, root->childCount);

  free(label);

  for (uint32_t i = 0; i < root->childCount; i++) {
    newNode->children[i] = createDotTreeFromMyTree(root->children[i], layer + 1, id,
debug);
  }

  return newNode;
}
```

Функция копирования структуры дерева

Для создания файла dot необходимо вызвать функцию `generateDotFileFromMyTree`, указав путь до выходного файла.

5 Результаты

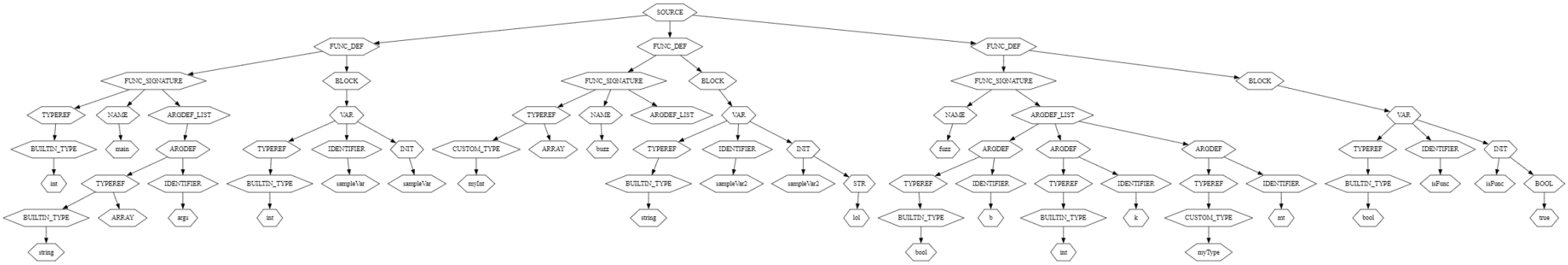
1) Подпрограммы со списком аргументов и возвращаемым значением

```
int main(string[] args) {
    int sampleVar;
}

myInt[] buzz() {
    string sampleVar2 = "lol";
}

fuzz(bool b, int k, myType mt) {
    bool isFunc = true;
}
```

Исходный код



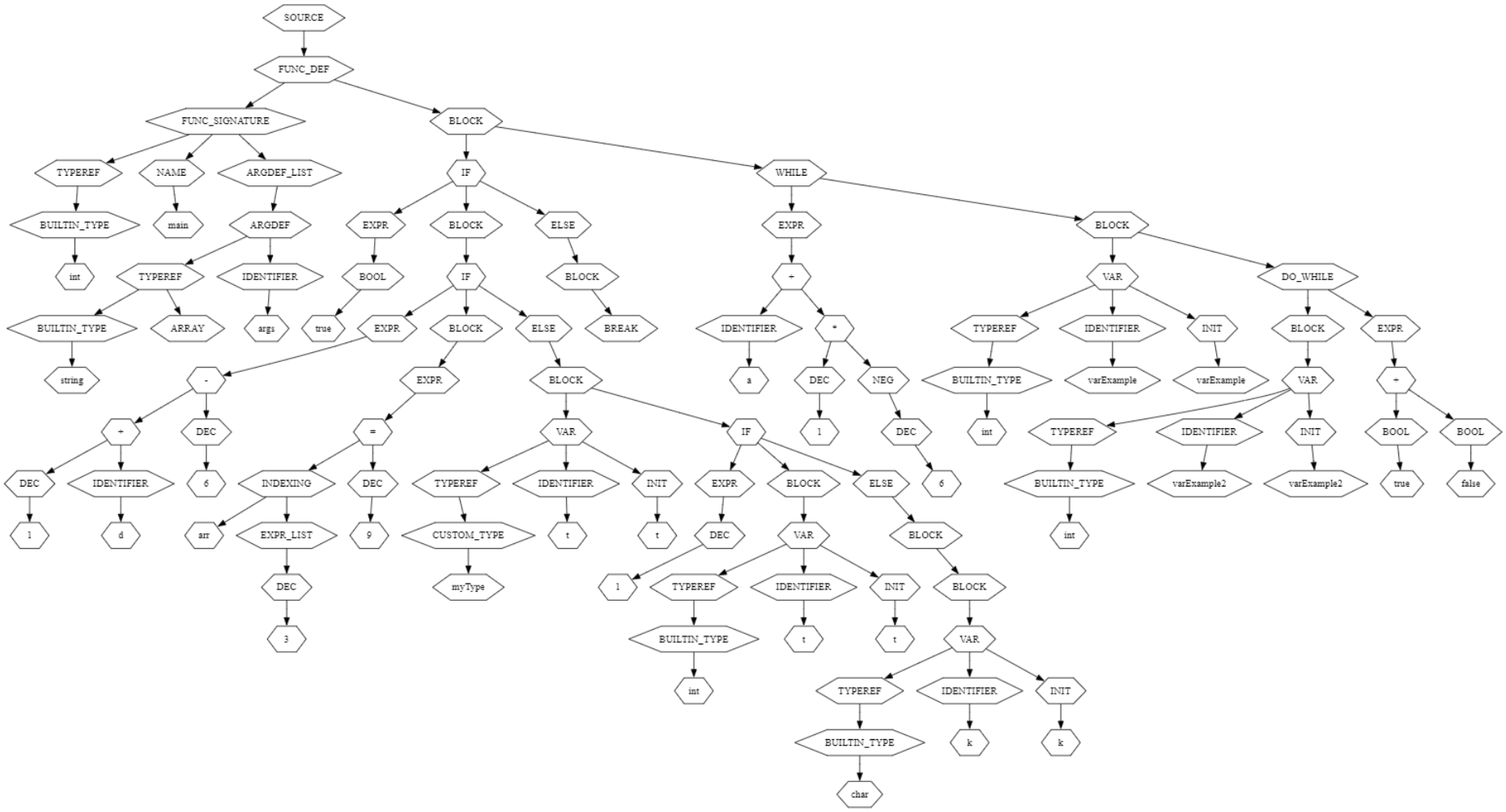
Визуализированное дерево

2) Операции контроля потока управления – простые ветвления if-else и циклы или аналоги

```
int main(string[] args) {
    if(true) {
        if (1 + d - 6) {
           arr[3] = 9;
        } else {
            myType t;
            if (1) {
                int t;
            } else {
                {
                    char k;
                }
            }
        }
    } else {
        break;
    }

    while(a + 1 * -6) {
        int varExample;
        do {
            int varExample2;
        } while (true + false);
    }
}
```

Исходный код

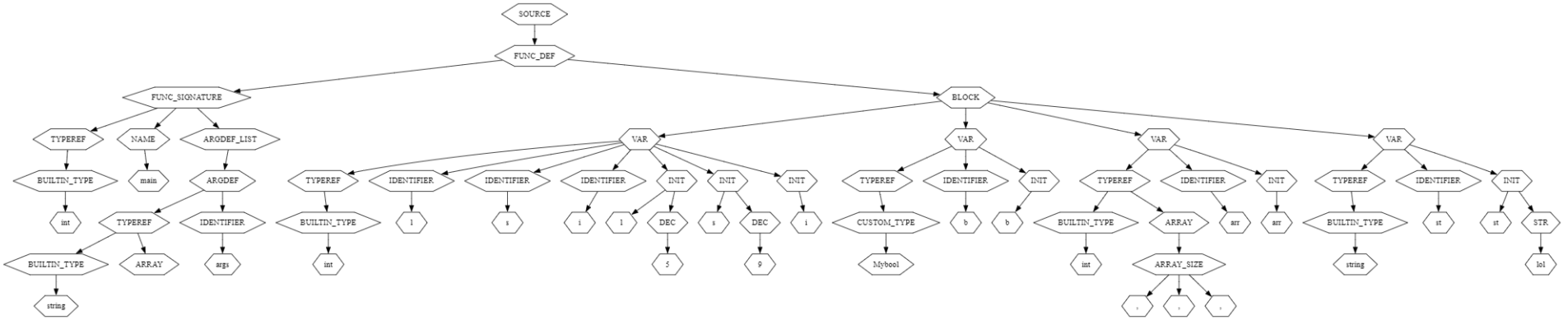


Визуализированное дерево

3) Определения переменных

```
int main(string[] args) {
    int l = 5, s = 9, i;
    Mybool b;
    int[,,,] arr;
    string st = "lol";
}
```

Исходный код



Визуализированное дерево

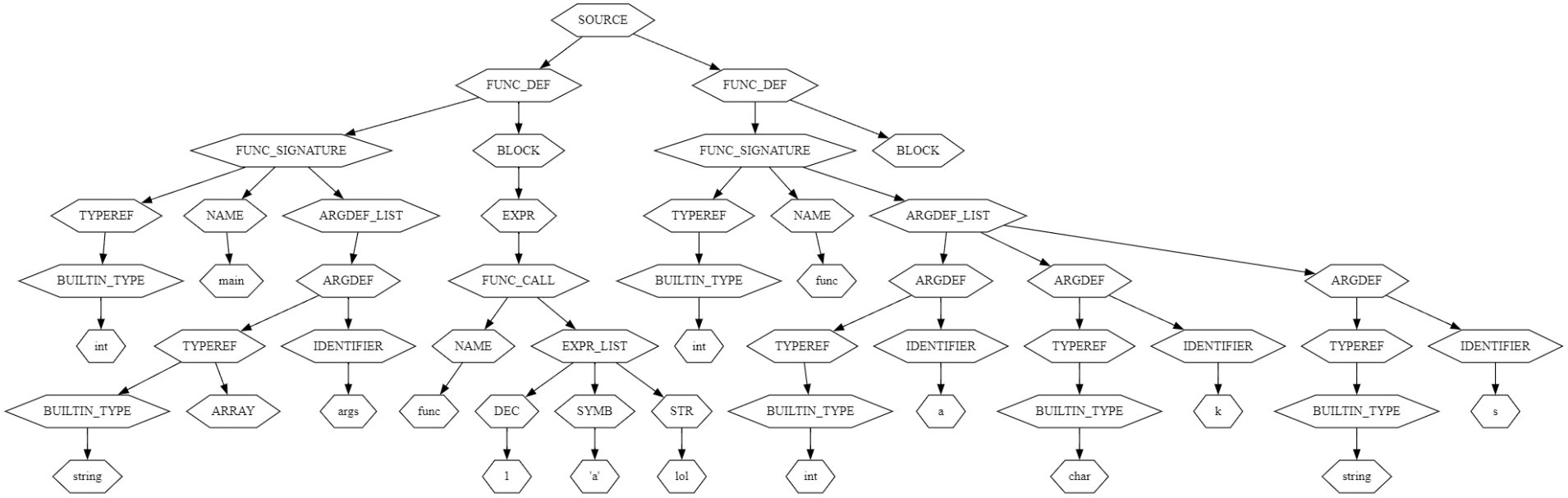
4) Целочисленные, строковые и односимвольные литералы

```
int main(string[] args) {
    func(1, 'a', "lol");
}

int func(int a, char k, string s) {

}
```

Исходный код

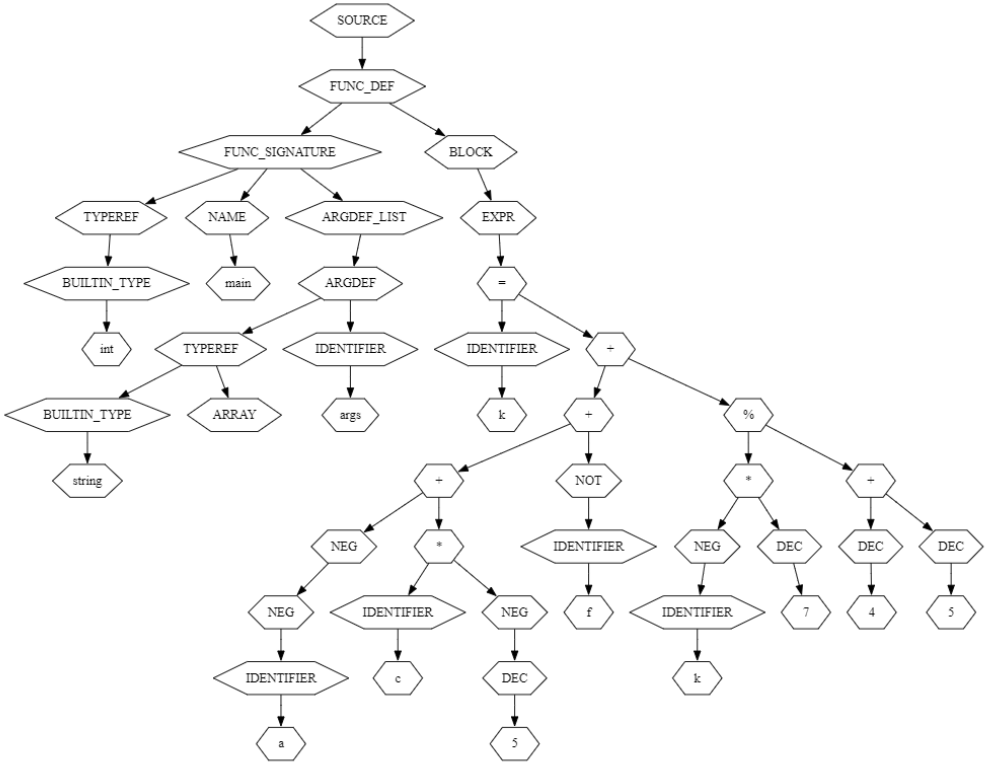


Визуализированное дерево

5) Выражения численной, битовой и логической арифметики

```
int main(string[] args) {
    k = --a + c * -5 + !f + -k * 7 % (4 + 5);
}
```

Исходный код

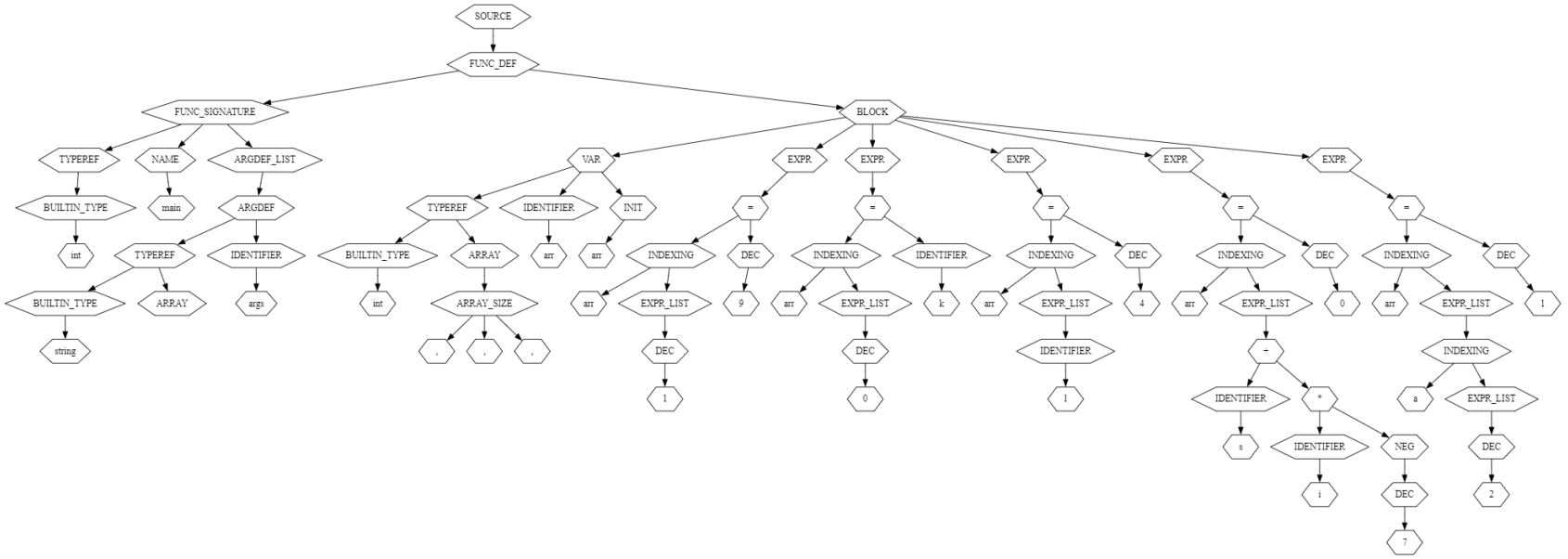


Визуализированное дерево

6) Выражения над одномерными массивами

```
int main(string[] args) {
    int[,,,] arr;
    arr[1] = 9;
    arr[0] = k;
    arr[1] = 4
    arr[s + i * -7] = 0;
    arr[a[2]] = 1;
}
```

Исходный код



Визуализированное дерево

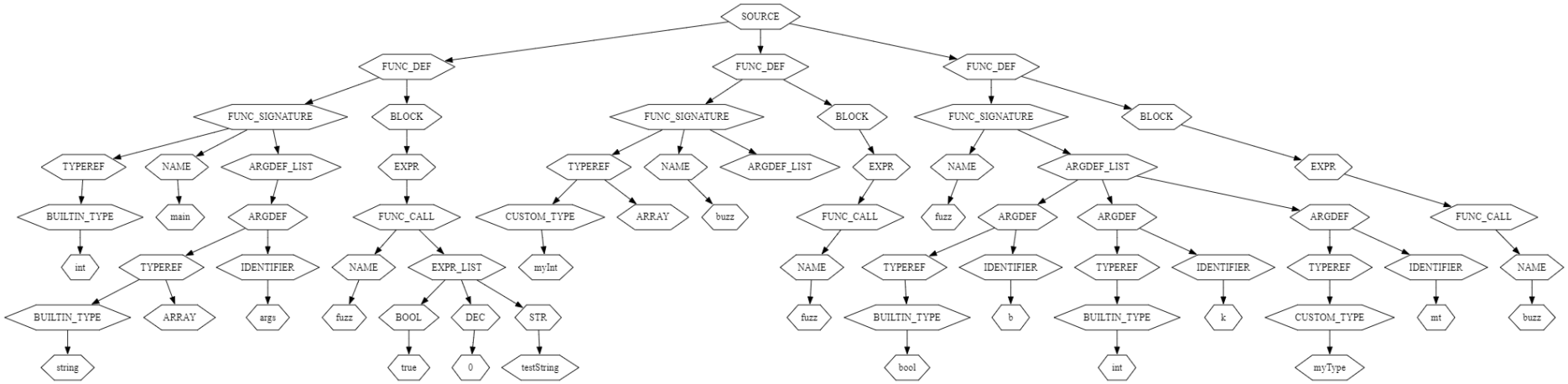
7) Выражения вызова функции

```
int main(string[] args) {
    fuzz(true, 0, "testString");
}

myInt[] buzz() {
    fuzz();
}

fuzz(bool b, int k, myType mt) {
    buzz();
}
```

Исходный код



Визуализированное дерево

8) Вывод ошибок парсера

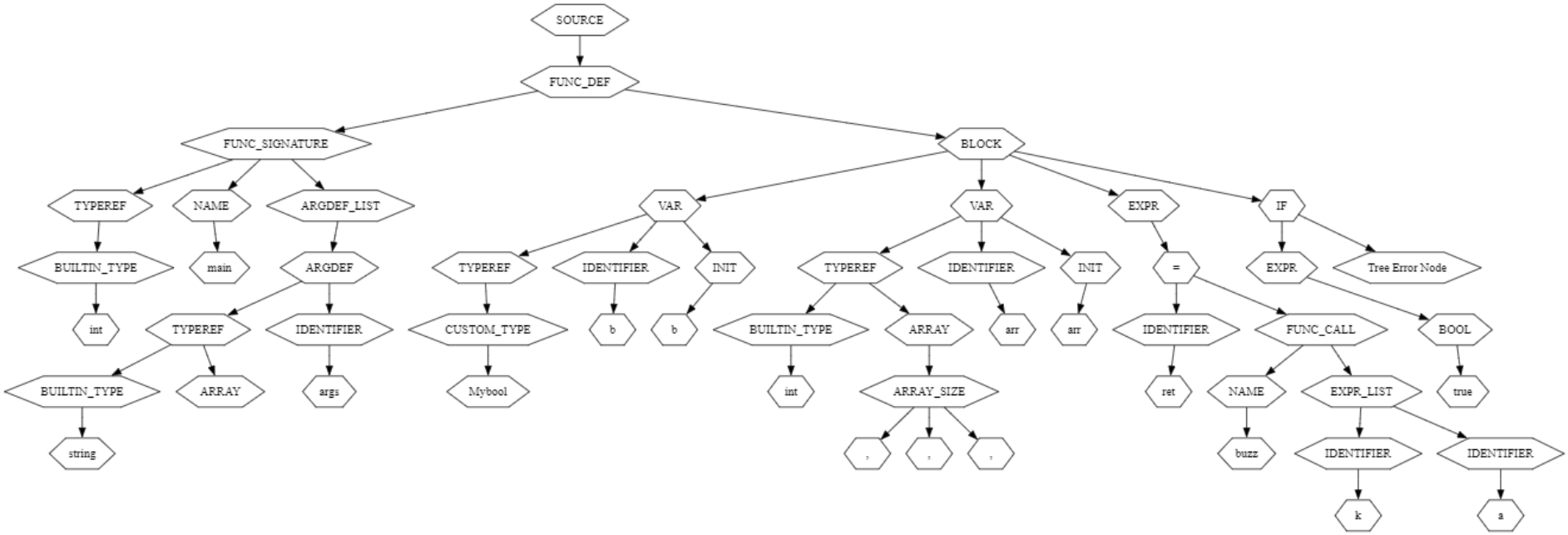
```
int main(string[] args) {
    Mybool b
    int[,,,] arr
    ret = buzz(k, a;
    if(true)

    }
}
```

Исходный код

```
Error 1 in line 3 at 4 in token '[Index: 0 (Start: 0-Stop: 0) ='<missing ';'>', type<85> Line: 3 LinePos:4]':
org.antlr.runtime.MissingTokenException
Error 2 in line 3 at 4 in token '[Index: 0 (Start: 0-Stop: 0) ='<missing ';'>', type<85> Line: 4 LinePos:4]':
org.antlr.runtime.MissingTokenException
Error 3 in line 3 at 4 in token '[Index: 0 (Start: 0-Stop: 0) ='<missing ')'>', type<83> Line: 4 LinePos:19]':
org.antlr.runtime.MissingTokenException
Error 4 in line 7 at 4 in token '[Index: 30 (Start: 344-Stop: 344) ='}', type<94> Line: 7 LinePos:4]':
Error 5 in line 7 at 4 in token '[Index: 30 (Start: 344-Stop: 344) ='}', type<94> Line: 7 LinePos:4]':
org.antlr.runtime.UnwantedTokenException
```

Полученные ошибки



Визуализированное дерево

6 Выводы

В ходе работы было изучено средство синтаксического анализа ANTLR v3, а именно способ задания грамматики языка, принципы построения AST с помощью rewriting rules и соответствующих операторов. В результате анализа получается структура, содержащая дерево и список ошибок при разборе.

Для графического представления AST был изучен и применён формат dot, для чего пришлось немного обработать полученное дерево – выдать каждому узлу уникальный номер и удалить некоторые символы.

Впервые использовалась «встроенный» в GNU C обработчик аргументов командной строки – argp, что очень упростило процесс обработки.

В качестве примера и вдохновения использовалась грамматика C, написанная в терминах ANTLR. В результате изучения примеров и немного скудной (для C target) документации выяснилось, что не все возможности доступны в C или они реализованы не так, как хотелось бы. Однако, если сравнивать опыт взаимодействия с bison, то с ANTLR в какой-то степени было проще работать т. к. не нужно было контролировать весь процесс построения AST вручную, что несомненно является плюсом в данной ситуации.