УНИВЕРСИТЕТ ИТМО

факультет программной инженерии и компьютерной техники

Направление подготовки 09.04.04 Системное и прикладное программное обеспечение

Системное програмное обеспечение Семестр 1

Лабораторная работа No1

Студент
Хуссейн, Авин
P4114

Преподаватель
Кореньков Ю.Д

Санкт-Петербург, 2024 г.

Goals

Implement a module for a syntax analysis of the given language in variant number 3, using ANTLR 3.4. The implementation should support syntax tree construction having nodes of the kinds according to the given syntax model of the language. Write some representation of the tree to the output file using data format which supports graphical visualization.

Subtasks

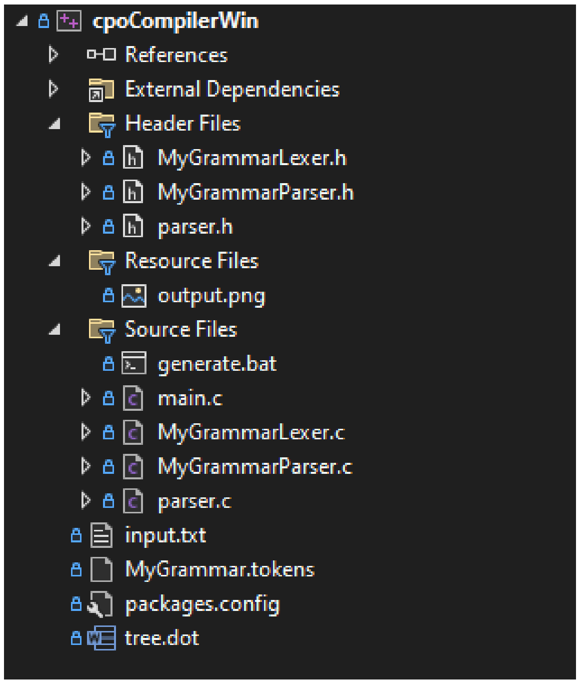
- Learning about ANTLR 3 and how to construct a correct grammar using it.
- Write a grammar in antlr, and test it on different examples.
- Prepare the environment to implement the solution and test it on different operating systems using remote debugging and a virtual machine.
- Implement the internal module, responsible for parsing the grammar and returning a structure containing the AST and the errors- if they occur.
- Implement a visualizing graph of the output tree.
- Implement a testing program, to verify the correctness of the solution.

Solution description

modules the solution consists of 3 projects

1. **C** a build of the antlr 3.4 tool for C language.
2. **cpoCompiler(Linux)** our project that will get tested on Linux (Debian) using remote building with VM.
3. **cpoCompilerWin** our project that will be tested on my windows machine.

For each of the 2 and 3 projects we have the following structure shown in the picture bellow



```
cpoCompilerWin
    References
    External Dependencies
    Header Files
        MyGrammarLexer.h
        MyGrammarParser.h
        parser.h
    Resource Files
        output.png
    Source Files
        generate.bat
        main.c
        MyGrammarLexer.c
        MyGrammarParser.c
        parser.c
    input.txt
    MyGrammar.tokens
    packages.config
    tree.dot
```

Санкт-Петербург, 2024 г.

Both the modules: MyGrammarParser and MyGrammarLexer are autogenerated when we generate from the grammar: MyGrammar.g (located in the linux project). They can be generated from the developer powershell by calling the bat file “generate.bat”.

The parsing unit is “Parser.c”. In it’s associated header file “Parser.h”. And the main function, responsible for getting the input from the input file “input.txt” and handing it to the parsing unit, and also receiving the parse results from it, that would be “main.c”.

Implementation details and source code examples

We define the structure of the output, which is “ParseResult”. This output format consists of the tree and the errors. The tree is stored in “ParseTree”, which is a linked list of ParseTrees- containing information about the tree’s children and their count. While, the errors go in “ErrorInfo” which is a linked list containing information about the line and position of the error.

The main function used is

```
ParseResult parse(char* text, size_t size, char* name);
```

Which is responsible for parsing a given text and returning the parse results.

Finally, freeErrorInfo, is a function used to free the resources after the task is done.

To create a visual representation of the tree, dot format is used. For starters I create a .dot file representing the AST, and then I use the Graphviz tool to create an image of the tree in the .dot file.

To report errors (this work can be found in parser.c in the errors region), I have overloaded a function “MyreportError” which overloads the function “reportError”, present in the built-in recognizer inside the parser tool. It adds errors in the typical way of linked lists, except for that it is called whenever an error occurs and then through this function

```
ErrorInfo* createErrorInfoNode(const char* message, int line, int position)
```

It creates a new error node, and through this function

```
void addError(ErrorInfo** errors, ErrorInfo* newError)
```

it adds it to the errors list.

Some of the extra work involved in the implementation

While working with the grammar, I faced a problem with constructing trees of multidimensional arrays. The problem was caused by two factors, firstly- the nature of the grammar of the my variant (3) supports multidimensional arrays that are left recursive, which are defined as the following

```
typeRef: {

|builtin: 'bool'|'byte'|'int'|'uint'|'long'|'ulong'|'char'|'string';

|custom: identifier;

|array: typeRef '(' (',')* ')';
```

Secondly, I have chosen the antlr tool, which does not support left recursion and also parses left to right, so rewriting the grammar was not going to help make it parse correctly. The only solution was to reconstruct the grammar manually.

The solution of this problem can be found in the "reconstructing the tree" region of the parser.c module.

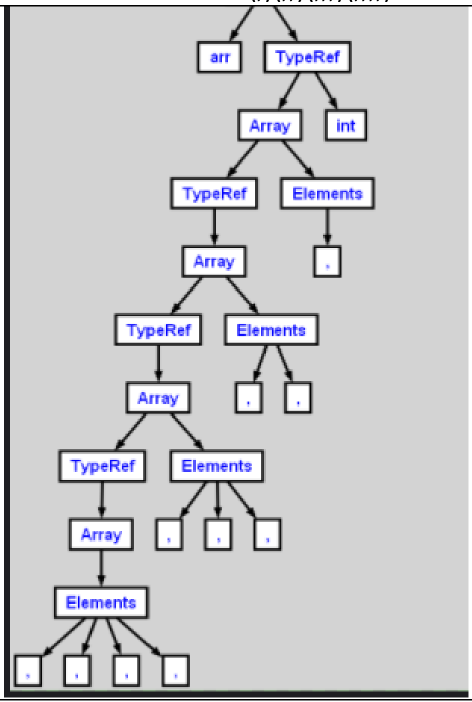
But in short, the solution was implemented through two functions

```
 int run2(pANTLR3_BASE_TREE tree) which is the function reconstructing the tree
void runChildren(pANTLR3_BASE_TREE tree) which is calling run2 recursively on all of
it's children.
```

In the first function, I get the token of the tree and handle different cases, so that the reconstruction only happens if we find the multidimensional array case that we talked about, as follows.

```
if (tree) {
        pANTLR3_COMMON_TOKEN tok = tree->getToken(tree);


        if (tok) {
            pANTLR3_BASE_TREE full_tree = antlr3BaseTreeNew;
            switch (tok->type) {


            case TypeRef: {
                if (tree->children != NULL) {
                    pANTLR3_BASE_TREE elements = tree;
                    if (tree->children->count > 1) {

                        //reconstruct (** to be explained)
                                }
                   runChildren(tree);
                    return 0;
                }

            }

            default: {
                runChildren(tree);
                break;
            }
```

(**) when the reconstruction needs to happen the graph of the tree looks something like this for the following grammar

arr as int(,)(,,)(,,,)(,,,,)


I am doing 2 things, firstly, I am taking the type node at the very beginning and adding it to the last array child (every array must have two children , it's type, and elements. If we look closely at the bottom array, it has no type.)

The second thing is to change the order of the elements. The order should be completely flipped to get the correct tree. This has been done, by looping twice on that portion of the tree and changing the order of elements sequentially and then adding the last element to the top. This was done with the help of these functions from the built-in library

```
tree->children->count  // to get the child count

type_tree->addChild(type_tree, type_node); // add a child node

tree->deleteChild(tree, index); // delete a child node

pANTLR3_BASE_TREE type_tree = tree->dupNode(tree); // duplicate a node

pANTLR3_BASE_TREE type_node = getChild(tree, index); // get a child node at an
index, this is not built- in
```

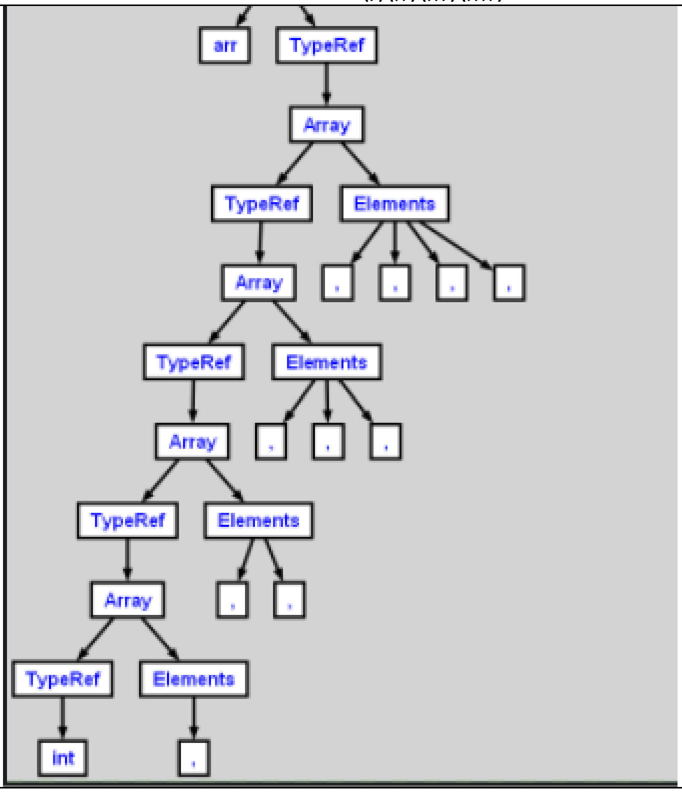
Finally, the "main.c" module. This module is reading the input from an input file "input.txt", calling the parser, getting the parse results and errors and printing the errors one by one as follows

```
ErrorInfo* current = result.errors;
	while (current != NULL) {
		printf("%s", current->message, " at line: ", current->line, " and pos:
", current->position);
			current = current->next;
	}
	freeErrorInfo(result.errors);
```

Санкт-Петербург, 2024 г.

Results

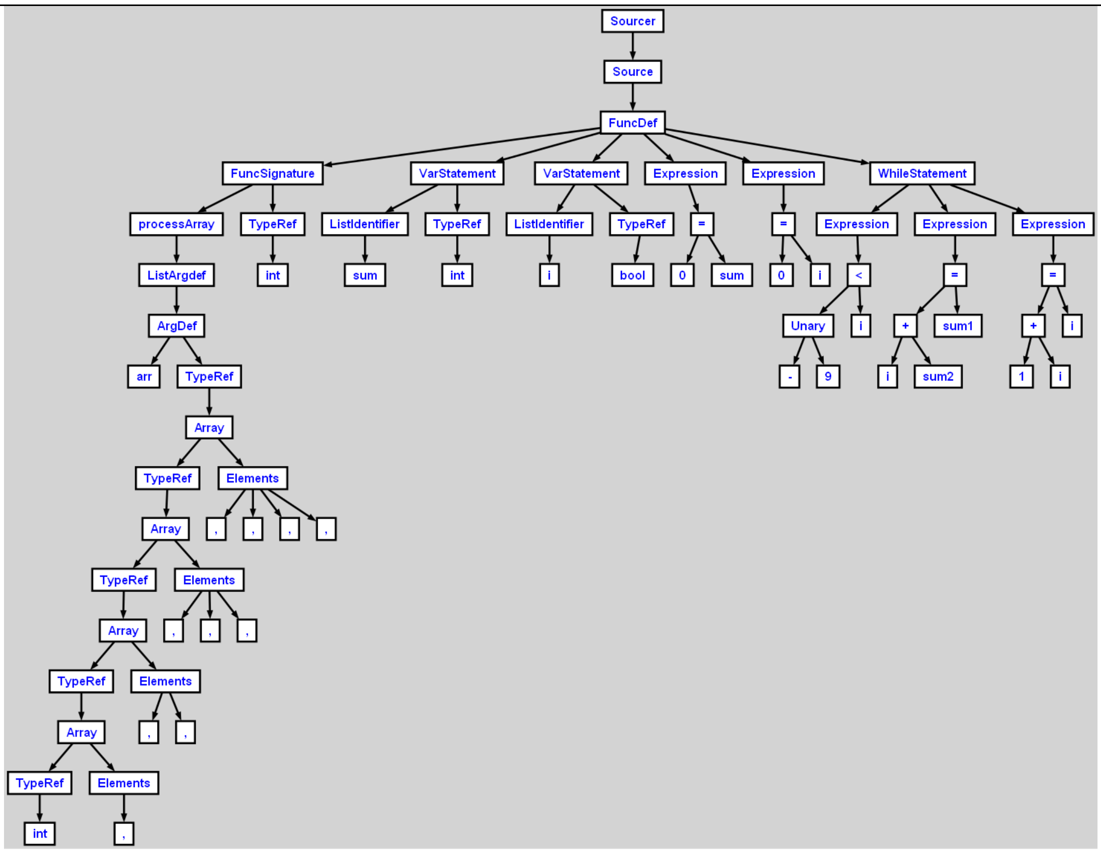
The result of the last example discussed (multi-D array) become as we can see in the picture bellow.

arr as int(,)(,,)(,,,)(,,,,)
 arr TypeRef Array TypeRef Elements Array TypeRef Elements Array TypeRef Elements Array TypeRef Elements int

The overall result, was done through a mix of rewrite rules in the grammar and some manual tree reconstruction, it looks like this for the following examples.

An example with a while loop

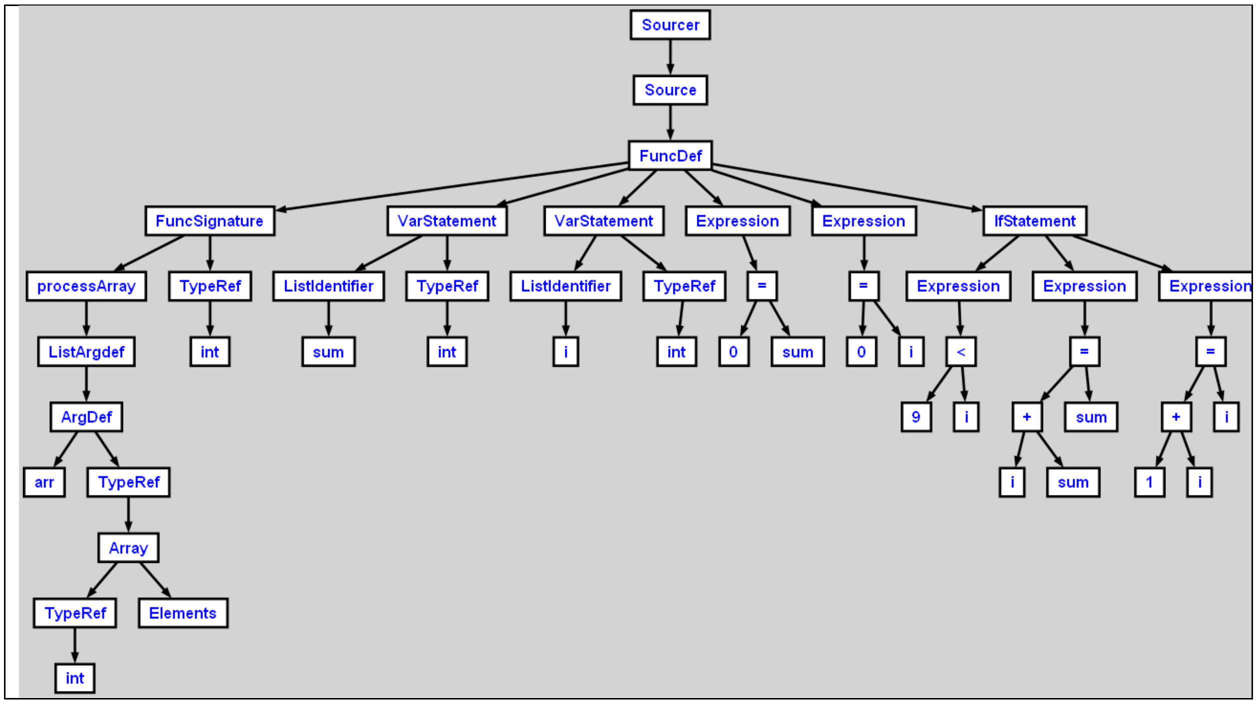
```
function processArray(arr as int(,)(,,)(,,,)(,,,,)) as int
    dim sum as int
    dim i as bool
    sum = 0;
    i = 0;
    while i < -9
        sum1 = sum2 + i;
        i = i + 1;
        wend
end function
```



An example with a while loop

```
function processArray(arr as int()) as int
    dim sum as int
    dim i as int
    sum = 0;
    i = 0;
    if i < 9 then
        sum = sum + i;
        i = i + 1;
end if
end function
```

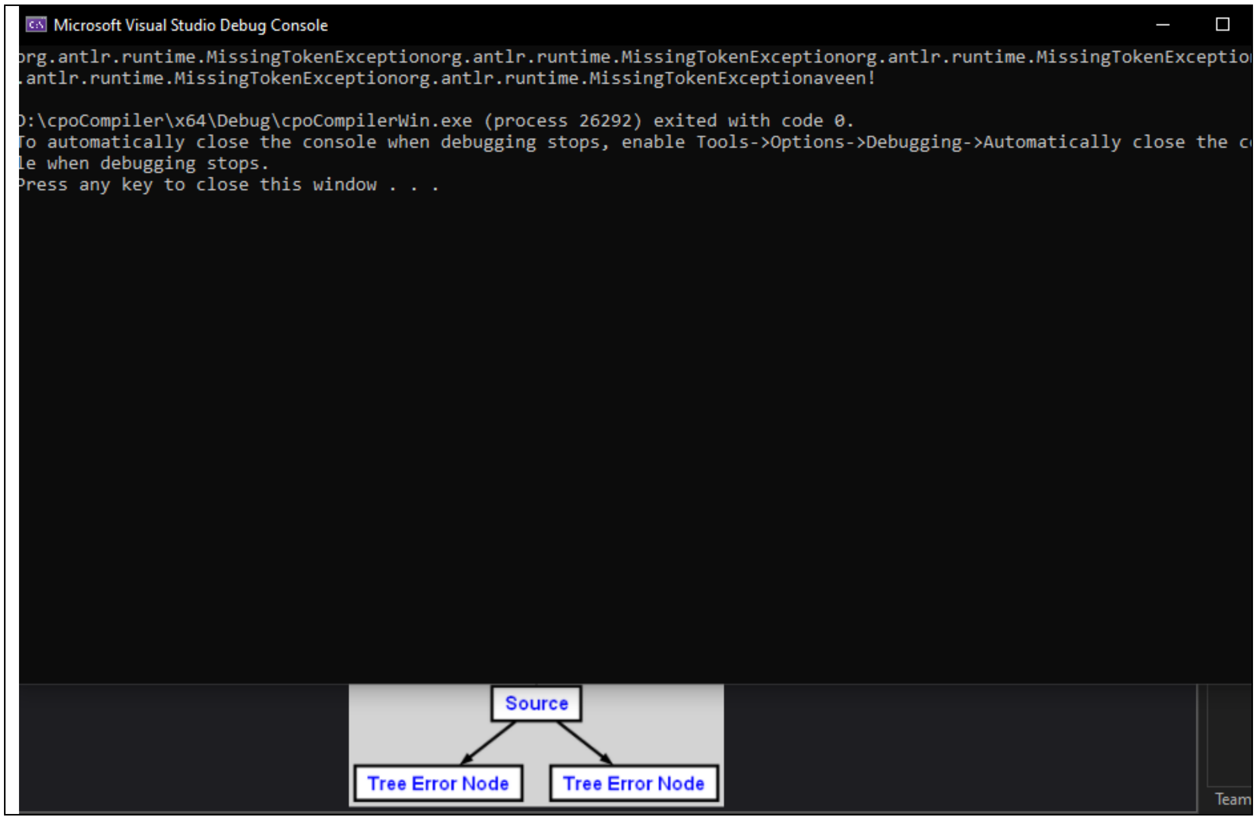
Санкт-Петербург, 2024 г.



An example with an error input

```
function processArray(arr a int()) as int
    dim sum as int
    dim i as int
    sum = 0;
    i = 0;
    if i < 9 then
        sum = sum + i;
        i = i + 1;
end if
end function
```

Санкт-Петербург, 2024 г.



```
org.antlr.runtime.MissingTokenExceptionorg.antlr.runtime.MissingTokenExceptionorg.antlr.runtime.MissingTokenExceptio
.antlr.runtime.MissingTokenExceptionorg.antlr.runtime.MissingTokenExceptionaveen!

D:\cpoCompiler\x64\Debug\cpoCompilerWin.exe (process 26292) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the c
le when debugging stops.
Press any key to close this window . . .
```

Conclusions

I believe the results of this lab work align with the goals. Antlr tool was chosen and studied, the grammar was constructed, and it suffices all the cases mentioned in the point 2 of the task description in the study material. The parse module was implemented and tested and the tree hierarchy was achieved and represented graphically. The test module was also implemented, and it is the one responsible for outputting the errors received from the parse module. The solution is supported with examples corresponding to different test cases.

I have learned a lot about how the antlr tool works, and how LL parsers work in general. And about remote debugging and about how grammars for programming languages generally need to look like, and why, and the different structures and types inside a programming language and their relationship with one another.

Link to the project on gitlab

https://gitlab.se.ifmo.ru/Aveena/cpoCompiler

Санкт-Петербург, 2024 г.