Introduction to language theory and compiling : Project Part 3

Abou Zaidi Ahmed 427196, Sefu Kevin 418790

December 19, 2018

1 Introduction

During this last part of the project, we were asked to improve our recursive-descent LL(1) parser by adding to it the ability of code generation that corresponds to the semantic of the Super-Fortran language. The output of our algorithm had to be a code written in the LLVM intermediary language.

2 Implementation

2.1 How we managed scoping

To manage scoping we decided to use a structure that looks a little bit to a tree. Meaning that our compiler manage several symbols table, on for each scope.
To enter in a new scope we call `enterContext` which is equivalent to enter in a child of the current node. And exiting the context with `exitContext` which is equivalent to go back (back to the parent).
As said before all those symbol tables are arranged in a tree, in order to reflect the nesting of scopes. When the compiler met a token corresponding to an identifier with a certain name, it looks up for this name in several symbol tables: first, the current symbol table, thenif the identifier has not been, it will look in the father scope and will continue like this until we reach the scope corresponding to the identifier. This browsing can continue until we reach the root node, which correspond to the symbol table of the global scope.

2.2 Data Structures

List : We used ArrayList in this third part, to handle the list of LLVM intructions that will be executed and to handle the list of variables.

HashSet : We used HashSet to check if the same variables has been declared several times in the super-fortran source code and to store variable that has been declared in a specific scope.

2.3 Construction of the abstract syntax tree for expression and conditions

We build binary AST only for arithmetical expression and conditions.
To make this implementation we first convert the sub-tree that contains the expressions to Left-child right-sibling tree. We decided to use Left-child right-sibling tree because this structure allowed to manage easily the code generation for arithmetical expressions. The figure 1 gives an illustrated explanation of our algorithm.

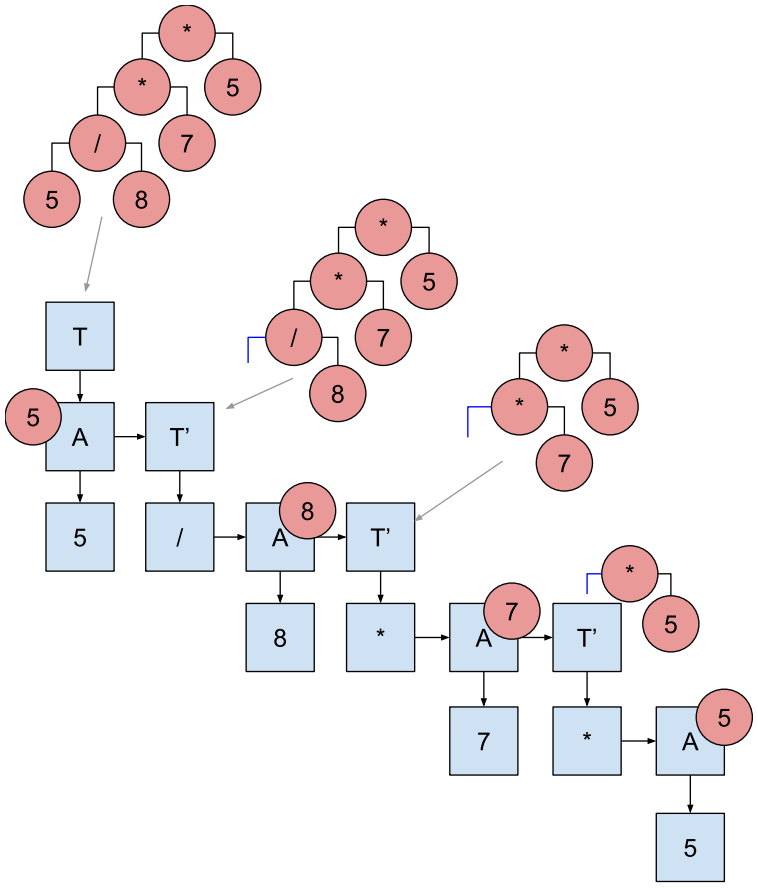

Figure 1: Converting expression `Parsetree` to `BinaryTree`

Image explanation : Firstly, we distinguished two kind of trees, full trees and plug-able trees.
Full trees are trees return by variables : Atom, Term and Expr. Plug-able trees are the ones returned by TermPrim and ExprPrim. (cf our grammar.java file)

When a plug-able tree is returned, it is used by the parent to plug its other children.

This algorithm allows us to parse the expression in a way that no nodes need to take attribute from brothers or uncles. But only from children. It also keep the left associative property of operators.

2.4 Hypotheses

Variables initialization : We decided to initialize all variable with zero. This assignation is made before running user instructions.

2.5 Exceptions categories thrown

In order to make the debugging easy for the user, In addition to syntax error we implemented two categories of exceptions that can be thrown :

1. Accessing to a variable that have not been declared
2. Declaring twice the same variable

2.6 Test files description

We created 29 test files. Those files browse in general all the implementations choices that we made during the whole project. Among those 29 files, there are also some files that generate compilation errors (ex : err-Cond).
The name of the files has been chosen in a way that, those name explain a little bit what the code does.

2.7 Javadoc

We were asked also to provide the javadoc of our program. In fact writing the Javadoc is a good habit because it helps us and our reader to understand easily the implementation.

2.8 Bonus implemented

We implemented two bonus :

1. FOR loops with negative increment: We implement this bonus by using an incrementor that we add to the iterator at each iteration. To know the value of the incrementor we firstly do a comparison between the start value and the target value, if the target value is bigger than the start value, then the incrementor value is 1 otherwise if the target value is less than the start value, the incrementor value will -1.
   Before the first iteration, the incrementor should be added to the target value to include it in the range. This bonus can be tested by using the file named 29-ForDesc.sf

2. Variable scoping : This bonus has been implemented has explain in the section `How we managed scoping`
   This bonus can be tested by using the file named 28-ScopeError.sf and 27-ScopeFor.sf

3 Difficulties faced during implementation

3.1 AST Contruction

During the implementation of our code, a difficulty that we faced was the construction of the abstract synthax tree. In fact, even by doing research on internet we were not able to find a good tutorial that explained well what we were looking for. To solve this problem we finally implement our on algorithm for AST generation.

4 Conclusion

Working on the Super-Fortran allowed us to understand very deeply how compiler works. This project was very interesting because it allowed us to enrich our knowledge on the different compilation phases. About this third part of the project, it brought us new and very important knowledge that are used in the real life, among those knowledge we distinguished :

1. The way to use LLVM and we discover that LLVM wasn't not just an academic tool but it was used also by the programming language C.
2. The creation of abstract synthax trees (AST).

In summary as said in the lines above, it was a very rewarding project.