2023-2024

INFOF403 - Introduction to Language Theory and Compilation

Generating IR for PascalMaisPresque

Ferreira Brenno

Mutkowski Philippe

Gilles GEERAERTS

2023

Table des matières

1 Generating IR

1.1 Introduction

The third part of this project consists of the generation of LLVM IR code (intermediate representation) that correspond to the semantics of PASCALMaisPresque. LLVM IR code is a powerful language that can be converted to machine code, offering the strength and speed of assembly without the need for hardware-dependent code.

To achieve this, two specialized classes, ASTGenerator and LLVMGenerator, were implemented. The first class, ASTGenerator, generates an abstract syntax tree (AST) from the ParseTree generated by the parser. The resulting AST contains the code's logic and structure, containing only the necessary information for the upcoming code generation process.

The second class, LLVMGenerator, utilizes recursive functions to navigate to each AST node and generate code based on the language's semantics. This approach ensures that the generated LLVM IR accurately reflects the PASCALMaisPresque source code. The following sections delve into the details of these implementations.

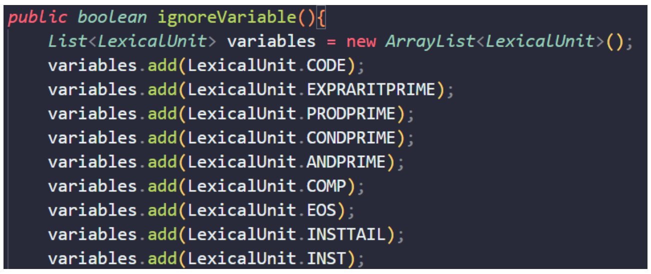
1.2 AST

The resulting ParseTree output of Parser class contains too much information, containing nodes that were crucial for ensuring the correct syntax but that are not needed in the following steps of compilations. The idea is to generate an AST that will mirror the original structure while removing unnecessary nodes.

The abstract syntax tree of the source code must first be generated by the ASTGenerator class in order to be used by LLVMGenerator class. This approach ensures a clear separation of concerns, simplifying the process of refactoring and modifying the AST.

1.2.1 Ignoring Variables : `ignoreVariable()`

This function basically contains a list with all the variables that are not wanted in the AST. It returns `True` or `False` depending if the variable is unwanted or not. Typically, unwanted variables are those without children (resulting in ϵ) or intermediary variables identified as unnecessary during implementation.



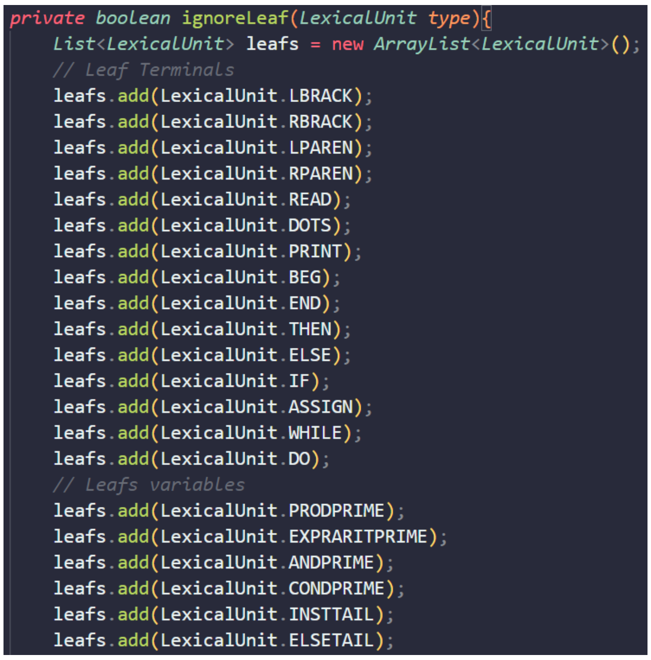
```
public boolean ignoreVariable(){
    List<LexicalUnit> variables = new ArrayList<LexicalUnit>();
    variables.add(LexicalUnit.CODE);
    variables.add(LexicalUnit.EXPRARITPRIME);
    variables.add(LexicalUnit.PRODPRIME);
    variables.add(LexicalUnit.CONDPRIME);
    variables.add(LexicalUnit.ANDPRIME);
    variables.add(LexicalUnit.COMP);
    variables.add(LexicalUnit.EOS);
    variables.add(LexicalUnit.INSTTAIL);
    variables.add(LexicalUnit.INST);
```

FIGURE 1.1 – The variables that are unwanted inside the AST.

1.2.2 Ignoring Terminals : `ignoreLeaf()`

This functions is analogous to `ignoreVariable()`. However, leaf nodes are treated differently due to their role as the edge case for recursion. Unlike variables at other levels, which only trigger recursive calls, leaf nodes mark the point where the recursive cascade returns.

The choice of final variables and terminals in the AST was made during the implementation of the LLVMGenerator class. While implementing functionalities into the code generation process, various combinations of variables and terminals were tested in order to identify the ones that would streamline LLVM code generation. The chosen nodes were those providing crucial information or aiding in establishing the priority of operations.The design decision behind AST generation facilitated straightforward refactoring.



```java
private boolean ignoreLeaf(LexicalUnit type){
    List<LexicalUnit> leafs = new ArrayList<LexicalUnit>();
    // Leaf Terminals
    leafs.add(LexicalUnit.LBRACK);
    leafs.add(LexicalUnit.RBRACK);
    leafs.add(LexicalUnit.LPAREN);
    leafs.add(LexicalUnit.RPAREN);
    leafs.add(LexicalUnit.READ);
    leafs.add(LexicalUnit.DOTS);
    leafs.add(LexicalUnit.PRINT);
    leafs.add(LexicalUnit.BEG);
    leafs.add(LexicalUnit.END);
    leafs.add(LexicalUnit.THEN);
    leafs.add(LexicalUnit.ELSE);
    leafs.add(LexicalUnit.IF);
    leafs.add(LexicalUnit.ASSIGN);
    leafs.add(LexicalUnit.WHILE);
    leafs.add(LexicalUnit.DO);
    // Leafs variables
    leafs.add(LexicalUnit.PRODPRIME);
    leafs.add(LexicalUnit.EXPRARITPRIME);
    leafs.add(LexicalUnit.ANDPRIME);
    leafs.add(LexicalUnit.CONDPRIME);
    leafs.add(LexicalUnit.INSTTAIL);
    leafs.add(LexicalUnit.ELSETAIL);
```

FIGURE 1.2 – The leaf node variables that are unwanted inside the AST.

1.2.3 generateAST()

The function is build around a for loop that goes trough every child of a tree and checks if the child needs to be added in the ast.
There are 2 main cases : when the child is a leaf and when the child is a node.

1. When it's a leaf, by default it's added in the AST. However, the `ignoreLeaf()` function determines if it needs to be ignored. There are a few terminals that are not needed in the AST and all the variables that results into epsilon.
2. When it's a node, it checks if the variable can be ignored with the `ignoreVariable()` function. If so, it generates the grandchildren and recursively calls `generateAST()` to each one of them. If the node isn't ignored, it's just added in the ast.

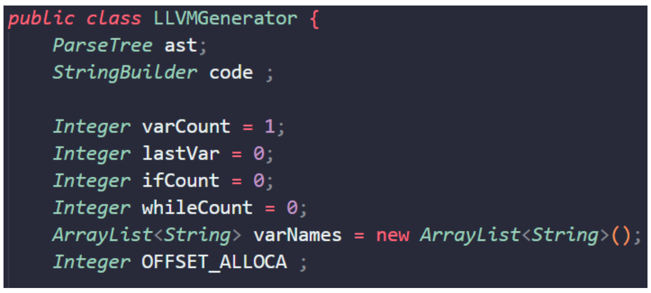
1.3 LLVM IR Code Generation

Using the AST as input, the LLVMGenerator class recursively traverses each node of the tree, generating the corresponding LLVM IR code that it represents.

About the implementation design : Every node has a process function. The function checks every child one level below of the tree that was given in input and for every token found, the correct function is called with that child.

There are nodes with a known number of child nodes. In this case, instead of utilizing a for loop, elements are selected 'manually' by their index. This concerns `processProgram()`, `processIf()`, `processElseTail()`, `processWhile()`, `processCondAtom()`, `processPrint()` and `processRead()`. For other nodes that don't have a fixed number of children, a for loop is employed to iterate trough every child.

1.3.1 Global variables



```java
public class LLVMGenerator {
    ParseTree ast;
    StringBuilder code ;

    Integer varCount = 1;
    Integer lastVar = 0;
    Integer ifCount = 0;
    Integer whileCount = 0;
    ArrayList<String> varNames = new ArrayList<String>();
    Integer OFFSET_ALLOCA ;
```

FIGURE 1.3 – The variables of the LLVMGenerator class

1. `ast` : input AST generated from the ParseTree given by the Parser class.
2. `code` : this is where every function will write llvm code. This is what is written in the stdout.
3. `varCount`, `lastVar` : these are used to keep track of the last used unnamed LLVM variables. They are incremented every time an unnamed variable is created.
4. `ifCount`, `whileCount` : these are used for writing labels for the if and while functions. It's incremented every time there's a if or while.
5. `varNames` : it's a list with all variable names of the .pmp program. It's used to check if a variable is already initialized while assigning it, aiding in determining when to allocate memory.
6. `OFFSET_ALLOCA` : it's a number used to know the beginning of the main function. It's initialized when all the basic functions have been written. It contains the file position in where the allocation line should be written.

1.3.2 processFunctions

generate()

This function adds all the basic functions in the LLVM code, sets the `OFFSET_ALLOCA` and calls `processProgram()` which starts AST analysis and code generation.

processProgram()

This function doesn't write anything. Only used to call processInstList.

processInstList()

This function does not write anything. Instead, it iterates trough every children, and calls the appropriate process function. The children are either a type of instruction or an `INSTLIST`. If it is `INSTLIST`, the function calls `processInstlist()` again. If it is an instruction, it calls the correct `process...()` function. Instruction nodes include `ASSIGN`, `IF`, `WHILE`, `PRINT`, and `READ`. It is worth noting that `INSTRUCTION` was one of the ignored variables.

processAssign()

The children of an `ASSIGN` node are always a `VARNAME` and an `EXPRARIT`. First of all, the function needs to process `EXPRARIT` by using the function `processExprArit()`.

For the `VARNAME` node, it first checks if the variable needs to be allocated by verifying that its name is not in the global `varNames` list. If it isn't, the variable is allocated at the beginning of the LLVM main function. The variable's allocation position is determined by the OFFSET_ALLOCA, which is incremented by the length of the allocation string. While the increment is not strictly necessary, it adds clarity and it's more intuitive to have the variables in order of appearance.

Allocating variables at the beginning of the main functions was needed to avoid the "Instruction does not dominate all uses!" error. This error happens when a variable is allocated in a code block that may not be executed, but the variable is used in another code block later in the program.

While allocating memory for a variable, there is an additional condition : The variable should be initialized before use. Thus, the function checks if the upcoming `EXPRARIT` contains the variable being currently assigned before doing the assignment. If such condition is met, the generator raises an error. Without this verification, the code was compiled and ran successfully, but the default LLVM behaviour was assuming the uninitialized variable is equal to 0, leading to unintended behavior.

processExprArit()

To understand how `processExprArit()` work, it is important to know how the AST is structure for this node. The Figure 1.4 shows the AST for the following line : `a := 7 + 4/2 + 1*2 - 11`.

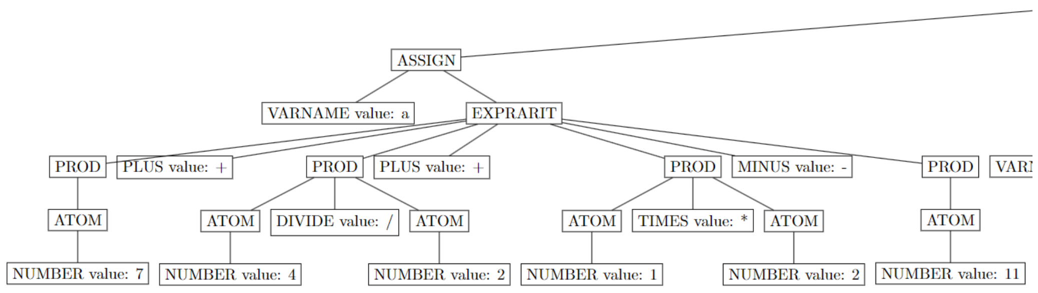


FIGURE 1.4 – AST branch showing an Assign node and a ExprArit one.

One can note the structure of this tree : `EXPRARIT` has a sum or difference of `PROD` and the same for `PROD`, where it has a multiplication or division of `ATOM`. This symmetry implies that the code for `PROD` will mirror the one for `EXPRARIT`.

The implemented approach involves iterating through every child, starting from the leftmost one. This ensures the application of the left associativity principle for operations with the same level of priority. Then, once both first and second term of the operation were read, the function writes down the operation and sets the result as the first term of the next operation.

If the node is a `PROD`, the function calls `processProd()`, meaning that any expression within `PROD` will be written and executed first, giving priority to multiplication/division over addition/-subtraction.

If the node is a terminal, such as `PLUS` or `MINUS`, the function stores it in memory using the variable `operation` until the second term of the addition/subtraction is read. This approach ensures the evaluation of the expression from left to right by pair of terms. For example, the line of code `a := 7 + 4/2 + 1*2 - 11` will be executed as `a := (((7 + (4/2)) + (1*2)) - 11)`.

processProd() As said before, `processProd()` is analogous to `processExprArith()`. However, the terminals are now `TIMES` and `DIVIDE` and the variable `ATOM` is processed with `processAtom()`. The same principles of `processExprArit()` are applied here. For example, the line : `a := 7+ 4/2*(2/3)` is executed as `a := (7+ ((4/2)*(2/3)))`.

processAtom() The `ATOM` has four cases and a special operation, the unary minus. In the first case, `ATOM` gives a `NUMBER`. This number is instantiated into an unnamed variable using a LLVM trick : `%0 = add i32 0, number`. This trick stores the value of the number inside the unnamed variable by mean of an addition with zero.

In the second case, where the node is a `VARNAME`, the only action required is to load the variable's value. Since the compiler enforces variable initialization before use, there is no need for additional precautions.

The third case is there to treat the case `a := (4+5)` where the elements inside parenthesis is seens by the parser as an `EXPRARIT`. It just call `processExprArit()` on it.

In the last case, the function handles the special operation of unary minus. The corresponding tree structure consists of an `ATOM` node pointing to a `MINUS` and another `ATOM`. When `processAtom()` encounters the `MINUS` node, it activates the unary minus operation by setting the variable `is_unary_minus()` to `True`. The following element is expected to be another `ATOM`, and the function processes it within this same function.

Before returning any result, `processAtom()` checks if `is_unary_minus()` is `True`. If true, the function multiplies the last unnamed variable containing the previous result by `-1`, correctly implementing the unary minus operation.

processRead()

The function calls `@readInt` to read from the stdin (defined in the basic functions). Then, it acts just like the `processAssign()` in order to assign the value read from stdin to the variable name specified in the code. The function checks whether the variable has already been declared. If it already is declared and allocated, the value is directly assigned to the variable; otherwise, memory is allocated before the assignment.

processPrint()

This function loads a value of the `VARNAME` linked to the `PRINT` node and then calls `@println` (defined in the basic functions).

processIf()

The structure of the AST for a `IF` node is : first child is a `COND` with the condition, the second child could be a `INSTLIST` or a single instruction and the last child, if present, is an `ELSETAIL` node.

Before processing nodes, this functions increments the `ifCount` variable to obtain its index, which it is then stored in `memory_if_count`. This variable is utilized to generate unique labels for the `IF` construct : `if_true_{memory_if_count}`, `if_false_{memory_if_count}` and `if_end_{memory_if_`

After this, the condition is processed with `processCond()` and the conditional jump is written.

Prior to processing the true or the false block, the function writes the corresponding label. Then, at the end of the block, it writes the jump to `if_end_{memory_if_count}`, representing the continuation of code after the `IF`.

Finally, the function concludes by writing the label `if_end_{memory_if_count}`, signifying the end of the `IF` construct.

processCond()

A `COND` node has a structure similar to `EXPRARIT` and it is possible to draw an equivalence between nodes :

— `EXPRARIT` ↔ `COND`

— `PROD` ↔ `AND`

— `ATOM` ↔ `CONDATOM`

This implies that the code of equivalents are similar.

In the structure, `COND` is basically ’or’ operations between `AND` nodes which, at its turn, are ’and’ operations between `CONDATOM` nodes. When evaluating a `COND`, the function iterates from left to right, first evaluating `AND` nodes, consistent with left associativity and priority principles. Notably, `processCond()` writes ’or’ operations every time it processes the left and right elements of the atomic condition. Ultimately, the result is set as the left element for the subsequent ’or’ operation for longer conditons.

processAnd() This function is almost identical to the one before. However, it processes ’and’ operations and `CONDATOM` variables.

processCondAtom() Two cases are possible :

— `CONDATOM` gives another `COND`. This corresponds to when the condition contains brackets.

— `CONDATOM` gives a left `EXPRARIT`, a operation and a right `EXPRARIT`.

For the first case, the function recalls `processCond()` to reevaluate the condition inside the brackets.

In the last and most general case, it is possible to directly extract the left and right `EXPRARIT` nodes and process them. After processing each `EXPRARIT`, the result is stored in the respective variable, `leftVar` or `rightVar`. Finally, the operation, which could also easily be extracted from the tree, is written down using `leftVar` and `rightVar` variables.

processElseTail()

This function doesn't generate any output; it only calls `processInstList()` to manage else instruction block. However, this separation is necessary to distinguish instructions within the true code block from those in the else block.

processWhile()

`WHILE` nodes are similar to `IF` nodes. Indeed, both contains a `COND` node and a true block to be executed if the condition is verified. The differences are that `WHILE` does not have a false block and that the condition is reevaluated at the end of the true block.

Similarly to `IF`, the function starts by incrementing the `whileCount` to obtain its index, which generates unique labels : `while_loop_{memory_while_count}`, `while_end_{memory_while_count}`. The condition is then processed and written in the current block. This is crucial because determining whether to jump, inside the loop or at the end, relies on checking the condition beforehand.

The label for the loop is set, and its block is processed. However, inside the loop block, the function reprocesses and rewrites the condition. This ensures that the code will reevaluate the condition at the end of each loop, deciding whether to continue with the loop or jump to the end accordingly.

Finally, the end while label is write down, signifying the end of the `WHILE` construct.

1.4 Main

The Main class was adjusted to first, makes the LLVM code generation, and then by generating the desired output.

The command implemented in the last part, which generated a .tex file containing the ParseTree, was removed, despite being heavily used during implementation for checking and studying the optimal AST structure.

1.5 Tests

1.5.1 euclid.pmp

This code returns the GCD of two values hard coded at the beginning of the code using an assign. This is the very first example of code in PASCALMaisPresque given in the project statements.

1.5.2 fibonacci.pmp

This test writes the 10 first terms of the fibonacci sequence. This code tests assignments, prints as well as while statements.

1.5.3 guessingGame.pmp

This test is a game where the user should guess what is the hidden value. Users has 10 tries and after one try, the program tells if the hidden value is higher, by sending 1 into the output or lower by sending -1. If the user correct guess, the while loop is break through the flag variable and a victory integer is printed. In the other case the loss victory is then printed.

This code tests almost every aspect of the language and it is very complex. This code shows the need for new elements to the language as booleans and strings, which unfortunately will not be implemented.

1.5.4 primeNumbers.pmp

This test generates the first 10 prime numbers. To achieve this, the logic for finding a prime number and performing a modulo operation was implemented directly in place. This is a comprehensive test, and the accuracy of the implementation can be quickly confirmed by examining the code output.

1.5.5 Conclusion

These tests aim to maximize the utilization of PASCALMaisPresque, covering nearly every functionality in the base language. The complexity of the test suite also highlights the potential for improvement by incorporating additional operators, data types, and function definitions. Recognizing the challenges associated with introducing new functionalities, the decision was made to prioritize studying and optimizing code generation. These tests confirm the successful implementation of PASCALMaisPresque.

1.6 Conclusion

This project provided a great opportunity to elucidate concepts from lectures and extend our understanding by applying them to the development of PASCALMaisPresque. It deepened our knowledge of machine languages, their functioning, and the associated challenges. Implementing PASCALMaisPresque was a unique and enriching experience, which nobody outside the domain of computer science will understand.