UNIVERSITEIT VAN AMSTERDAM

COMPILER REPORT

SAIF RASHED (15063658)
PRE-MASTER SOFTWARE ENGINEERING

Table of Contents

1. INTRODUCTION

The importance of programming languages and compilers cannot be overstated as compilers are essential tools that translate high-level programming languages into machine code. This report delves into the design and implementation of a compiler for the CiviC programming language, a C-like language designed for educational purposes. In this compiler, we designed support for arrays and multi-dimensional arrays up to the context analysis stage and nested functions up to the type analysis stage, but we were only able to complete the core language. By outlining the various stages involved in the process, such as lexical analysis, syntactic analysis, semantic analysis, and code generation, we aim to provide a comprehensive understanding of the inner workings of our CiviC compiler. Furthermore, we reflect on the challenges faced and lessons learned during this project.

2. LEXICOGRAPHIC ANALYSIS

The construction of our compiler began with the creation of a lexer using the Flex tool. A Flex file was created to define the tokenization rules for the input source code. Keywords and symbols utilized in the Civicc language were identified, and corresponding regular expressions were devised to recognize them in the input stream.

One challenge encountered during the development process was the lack of an effective method to test and debug our scanner. To overcome this challenge, the lexer was tested in a separate file, utilizing the flex command to compile and run a scanner. Simple print statements were implemented to output the tokens recognized by the lexer for a given input. Following this process, once confidence in the accuracy of token recognition was established, the lexer was integrated into our compiler, facilitating the advancement to the syntactic analysis phase.

3. SYNTACTIC ANALYSIS

This section describes the order of operations used in building the grammar.

Constants and expressions. Before starting with the higher-level grammars, we began by defining the constants and expressions as specified in the CiviC document. As a result, we created *unary_expr*, *arithmetic_expr*, *comparison_expr*, and *logical_expr* grammars. We also added expressions such as funcall and cast.

Statements, parameters and arguments. In defining these, we created three types of grammars for each: singular, one or many, and optional many grammars. We saw these types as being beneficial, for example, in function calls where it might be optional to have arguments. This could also be the case for parameters where function definitions might not have any parameters. The same goes for statements where a block could be empty.

Function definition, variable declaration. In the function definition, we attempted to unify all parts into one grammar. Unfortunately, this approach did not work out as

the grammar became too large, and we encountered ambiguities resulting in shift/reduce conflicts. To resolve this issue, we split the function definition into multiple parts, including funheader, funbody, and fundef. This approach made the grammar more manageable and allowed us to use the parts in other contexts, such as localfundef, for which we also implemented a grammar.

However, we still encountered ambiguities with vardecls, which was a known issue that had been discussed in the announcements. We resolved this conflict by making the vardecls left-recursive, but this resulted in the vardecls being reversed. To address this, we implemented a quick and easy solution that allowed us to continue:

```
vardecls: vardecls vardecl  // example: int a = 5; int b = 4; int c;
        {
          node_st *current = $1; // take head of vardecls

          while(VARDECL_NEXT(current) != NULL) { // traverse it till the end
            current = VARDECL_NEXT(current);
          }

          VARDECL_NEXT(current) = $2; // in the end we place the given vardecl

          $$ = $1;
        }
        | vardecl
        {
          $$ = $1;
        };
```

Global definition / declaration. After completing all the individual parts, we moved on to the global definition and declaration. This section was relatively straightforward, and we encountered no significant challenges or noteworthy elements worth mentioning in this report.

Top level declarations and program. The same applies to this since they were relatively simple grammars that did not introduce any noteworthy elements, so we will not delve into further detail here.

4. SEMANTIC ANALYSIS

During the semantic analysis stage, we designed a symbol table and implemented type checking. Each process will be described, with particular attention paid to the unique aspects. The following section will describe the order of operations used in implementing this.

Strategy

The general idea was to devise a symbol table that could be used for type checking and code generation. To do this, we needed to store all declarations in a data structure, and we chose to use a SYMTBL and STE node, which would in turn become a linked list attached to the program and fundef nodes.

The symtbl node would contain the head (child) of the symbol table and an outer (child) node that would reference the parent table. To fill this symbol table, we would need to traverse all declarations and store them with an *STstore* function. This function would create a new STE node, add it to the end of the SYMTBL, and link the STE to this declaration. If an STE was already defined, we would give an error message.

After some transformations, we would perform name binding, which would pass all Varlet and Var nodes and search for the name in the symbol table with *NBlookup*. This function would search through all symbol tables by climbing up the parents until there is no parent left. Let's go into more detail.

Symbol table

The symbol table traversal will visit the following nodes: program, fundef, globdecl, globdef, vardecl, param, and ids.

The goal of this traversal is to store all declarations in the symbol table of the scope where each declaration is located. To keep track of the scope, we use travdata to store the program or fundef node. Additionally, we define three functions that assist us in the traversals:

STstore. This function creates a new STE, adds it at the end of a symbol table, creates a link with the given declaration, and adds the type to the STE.

STlookup. We also need to determine whether a declaration has already been defined in the symbol table so that we can produce a correct error message. We accomplish this using *STlookup*, which searches for an identifier in a symbol table. Note that we have the symbol table stored in our scope node, which has been saved in the travdata.

STgeneratesignature. We want to encode function arity so we generate a function signature which contains the parameter amount. A function called *foo(int a, int b)* would get this signature: *foo_2.*

Array initialization

This part was implemented in an optimistic moment in our construction when we thought that arrays were still achievable. We still managed to do this transformation but this was arguably the hardest transformation to implement. There are several parts to this traversal:

Systematic Traversal of Declarations:
In the context analysis phase, we systematically traverse all fundefs, and manually loop through all declarations. If we encounter a vardecl with dimensions, we begin the process of extracting the dimensions.

Extracting Dimensions:
After identifying a vardecl with dimensions, we extract the dimensions and replace them with temporary variable names. To maintain the proper order, we use the Aladd_before function to add the new vardecls in the correct position. We also add them before the symbol table entry to maintain consistency within the table.

Determining Expression Type (Array Expression or Scalar Value):
We now need to determine the type of expression, as it will influence the next steps.

a) Array Expression: If the expression is an array, we create a list of assign statements. These statements will be added to the statement list of the fundef.

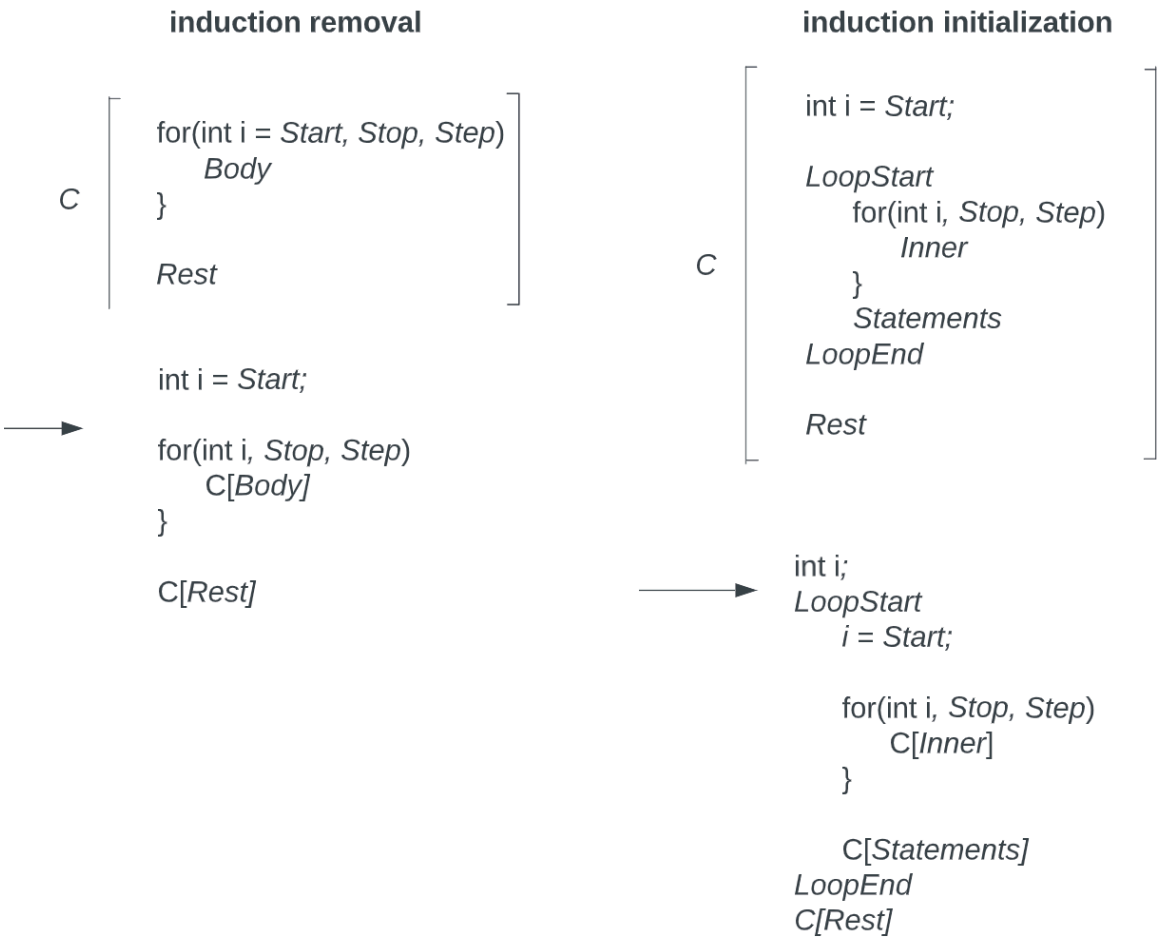
b) Scalar Value: If the expression is a scalar value, we create a nested for loop. The amount of nesting is determined by the number of dimensions defined in the vardecl.

We unfortunately didn't manage to get the order of the assignments correct which means that they are now reversed. This was because we did not build the stmts list correctly. Despite our best efforts, we were unable to correct this mistake in a timely manner due to time constraints. As a result, we were forced to move on to the next stage of the project, with the hope of addressing this issue at a later point.

INDUCTION REMOVAL (AND INITIALIZATION)

The goal of this transformation is to extract the induction variable into a variable declaration and remove it from the for loop. Initially, we performed the extraction, but later on, during code generation, we encountered an issue. While the for loops worked correctly, they did not work as expected when nested in another loop. This is because the transformation to an assignment, which is done during variable initialization, did not account for nested loops. To address this, we regard the two transformations as one operation but in two separate traversals to ensure proper functionality in both cases.

To visualize this process we attempt to devise a compilation scheme:

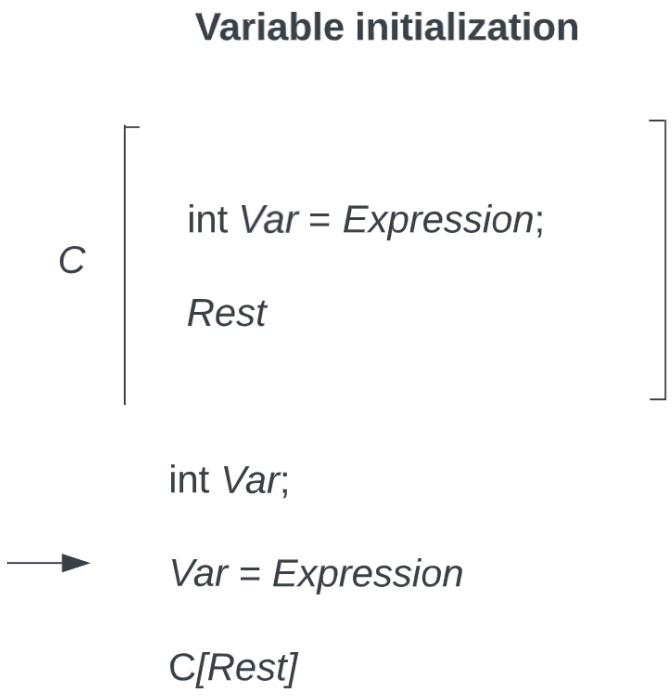


In the induction removal transformation, we extract the start expression into a separate variable declaration and recursively apply this transformation to the body statements and post statements. After extracting the induction variable, we perform an assignment right before the for loop statement for nested loops to ensure that the variable is assigned the start value each time the loop iterates. This helps prevent issues, such as those explained above, from arising.

Variable initialization

In this code transformation process, we perform two main tasks. Firstly, we initialize global variables by extracting their initialization values and inserting them into an assignment statement within the "__init" function. Secondly, we initialize function variable declarations by extracting their initialization values and inserting them into a new assignment statement at the beginning of the function statement list. If the variable declaration contains a "var", we traverse it to determine the correct associated declaration.

To visualize this process we attempt to devise a compilation scheme:



However, we encountered an issue with determining the correct scope for a variable. Our symbol table lookup function searches the entire table, regardless of position. This poses a problem when a variable uses a declaration that is further down the list or even within a "varlet". In this case, it should actually point to a global variable, if it exists. To solve this, we systematically check if a variable has a declaration before, and if not, we search the outer scope for a global variable. Otherwise, we throw an undefined reference error.

Name binding

During this traversal, we focus on searching varlet, var, and funcall nodes. This is the final pass in the context analysis, which was a satisfying moment because it meant that the symbol table would finally be useful for type checking analysis. For var and varlet nodes, nothing much changed - we simply use the lookup function to search

all symbol tables from inner to outer and skip any nodes that already have an entry due to preceding transformations.

However, for funcall nodes, things become more interesting. We used the *STgeneratesignature* function during symbol table creation to encode the arity of the fundef node. Now, we need to create a version that generates a signature for function calls by counting the arguments. We use this signature to search the symbol table, and if it exists, we have a match on arity. If it does not exist, we can conclude that the function signature does not match any other function.

During this process, we encountered a problem when considering implementing overloaded functions. It was problematic because we did not know how to determine the types of each argument without actually performing type checking. One possible solution would be to TRAVdo the arguments in case any of the arguments are vars. After that, we could infer the types of each argument and generate a correct signature, allowing for function overloading. However, we did not pursue this solution and continued into type analysis.

Type checker

In the type checker, our goal was to check for type mismatches and provide meaningful error messages when statements, functions, or variable declarations had disallowed types. In other words, we aimed to ensure that the types of all expressions, variables, and functions were compatible with their expected types, as per the rules of the CiviC language.

To achieve this, we followed a similar approach to that used in the syntactical analysis of grammars. We began with the lowest level nodes, such as constants and variables, and gradually progressed to monop/binops and other expressions. At each of these stages, we tracked an *inferred* variable that changed based on the type of expression, providing the correct type when traversed. This helped us in higher-level nodes like statements and function definitions.

Constants

In this part of type checking, we simply pass the num, float, and bool nodes and assign the inferred variable to one of these types. For var and varlet nodes, we do the same, but now we access the symbol table entry node and read the type from there.

Expressions

For the binops and monops, we use a switch case, which suffices for our goal. First, we check the type of operator, and then depending on the combination of types, we infer a certain type. If none of the cases are met, we return an error message. For funcalls, we iterate through all arguments and simultaneously test them with the parameters of their function definition. Eventually, we infer the function definition

type by accessing it in the symbol table. The cast node does the same, but here we already insert an ternary node for booleans that are casted to either float or num.

Statements
For return statements, we use our trick of storing the scope in travdata to access the function definition. Now, we can test the types and send error messages if necessary. In the assign node, we traverse both sides and compare the types. However, it should be noted that assign type checking also includes globaldef and vardecl type checking. We also perform an indices check. For the rest of the statements, we simply check if the types are correct in place. For example, in a while loop, we would check if the condition is boolean.

Function definition
Here, we use a *TChas_return* function to check if a function has a return statement or not. If it does not have a return statement when it is required, we send an error message. In addition to this, we simply traverse the children.

5. CODE GENERATION

In this part we describe the necessary transformations for code generation and the code generation itself. In this section we really got to fixing some design mistakes.

Strategy

To implement code generation, we had to make some transformations that would allow the code generator to produce code easily. Once the transformations were done, we used a file pointer in the travdata to write to a file. For each of the nodes we traversed, we would use fprintf to output the correct instructions. But before going into more detail, let's first go through the transformations.

Parameter passing

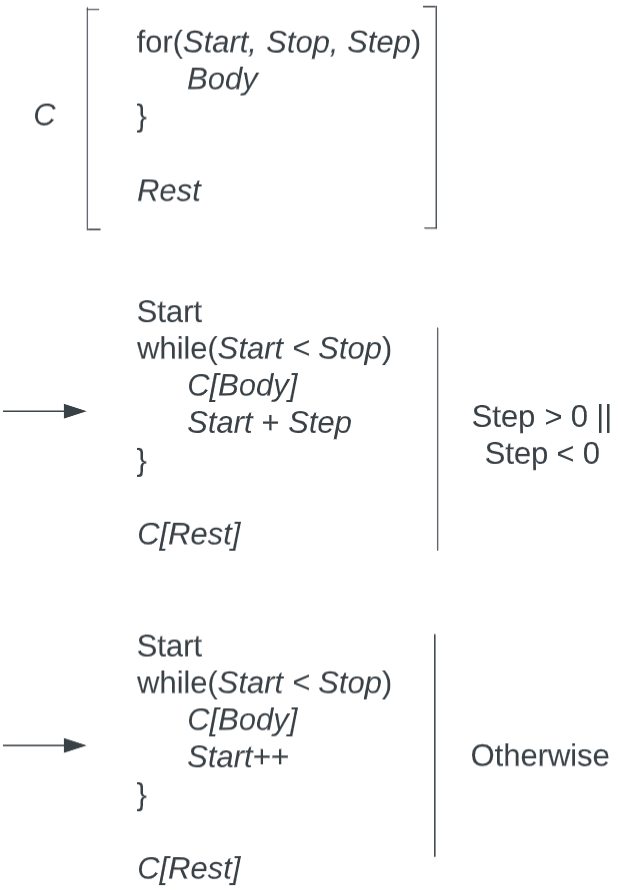
The parameter passing transformation passes through all fundef and funcall nodes and uses *PPadd_expr_before_arg* and *PPadd_id_before_arg* to insert new arguments in the funcall. For parameters we use *PPadd_before_param* function to add new parameters. We did not encounter any problems when creating this traversal so therefore we can keep it short.

Loop Transform

In the code generation we want to keep things simple and not overcomplicate things. That also means that we want to restrict ourselves to generating one type of loop. This means transforming the for loop into an equivalent while loop.

To visualize this process we attempt to devise a compilation scheme:

Loop transform



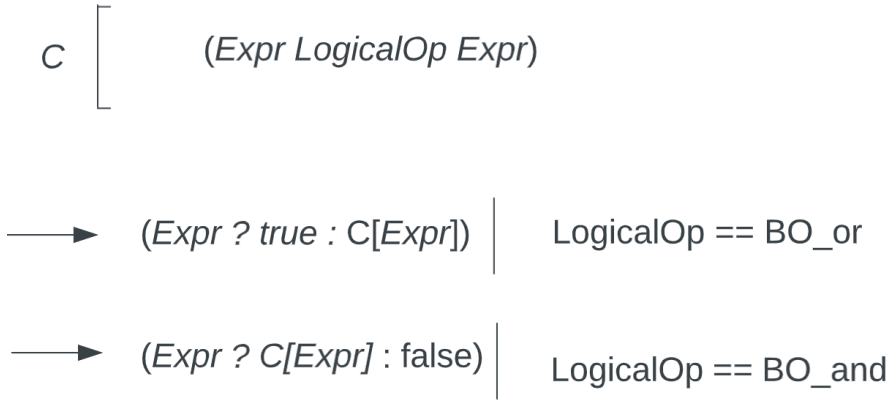
To do this we should take in consideration that a for loop can have a negative step value. This changes the behavior and we quickly figured out that the ternary can be used to encode this logic. This is how we implemented it:

```
node_st *condition = ASTternary(
    ASTbinop(CCNcopy(step), ASTnum(0), BO_gt),
    ASTbinop(ASTvar(NULL, STRcpy(FOR_VAR(node))), CCNcopy(FOR_STOP(node)), BO_lt),
    ASTbinop(ASTvar(NULL, STRcpy(FOR_VAR(node))), CCNcopy(FOR_STOP(node)), BO_gt));
```

BOOLEAN TRANSFORMATION

This transformation is quite straightforward and doesn't require much explanation but simply said we use the ternary node to recreate AND and OR operators by doing the following transformation:

Boolean transformation



Code generator

In this part, we initially encountered a lot of trouble because the concept of assembly and the VM was still unclear to us. The first thing we did was to produce assembly code with the reference compiler. By using the -o flag, we managed to generate an output file, which was crucial for generating code since it allowed us to compare both sides. The second step was to actually understand how the CiviC VM worked, and we struggled a lot with understanding what a constant runtime pool actually means in relation to the CiviC VM.

After some time, we understood that the CiviC VM consists of multiple tables, including import, export, global, and constant tables, and that each maps to a type of node. For example, a global definition would always be added to the global table, and each constant node above 2 would always be added to the constant table.

After understanding these nuances, we started building out the code generator. To keep track of the indexes of each table, we keep track of globals:

```
int global_index = 0;    // index for global table
int constant_index = 0;  // index for constant table
int importfun_index = 0; // index for import function table
int importvar_index = 0; // index for import variable table

char *instruction_string = ""; // to be shown in the end
```

As you can see, we also have a string that we build up. This string will contain all the pseudo instructions for the tables. The next step is to actually increment these values in the globdef, vardecl, param, and globdecl nodes. Before incrementing, we

need to store the index somewhere so that var/varlets can have access to the correct table entry.

To do this, we create a new STE attribute called 'assembly_index' which will store this information for us. Because all var/varlets are linked with the STE's, we can access the correct table index, but only have to determine the correct instruction. This is simply a set of if statements that check the node type.

Everything afterwards was quite straightforward. Eventually, after testing and comparing the reference compiler bytecode with ours, we created our own tests. These captured some edge cases:

- core.cvc: Contains our civic core.cvc code (without the rest since we did not manage the extensions)
- complex_if.cvc: Contains a complex if statement where we test if-else statements
- casting.cvc: We further tested casting values to make sure the values are still the same
- nested_loops.cvc: We nest different types of loops in each other to make sure we did not do anything wrong in the loop transform

REFLECTION

We started the semantic phase in a very improvised manner without really designing the process. Fortunately, we got the hang of this midway through the context analysis, where we designed and thought out the entire process before proceeding. This meant taking a helicopter view and looking further down the process to determine what would be useful and what was redundant. The result was that we were able to create useful functions like the lookup functions (which come in two variants) that saved us a lot of time. We could now also order the traversals in the most optimal way, such as for example initializing the induction variables from induction removal in the var initialize traversal.

We recognize that there were opportunities for improvement in our approach. For example, we dedicated a lot of unnecessary time to array initialization, which could have been better allocated towards implementing function overloading. Despite this setback, we believe that our decision to design for all extensions (and attempt them) from the start was the right choice. Each component of our project builds upon one another, and not doing so would have been a risky move since there was a possibility that we were able to do one or more extensions if time allowed for it. Overall, we are pleased with our end result.