CM30171 Compilers Coursework Report

Andrew Bell

December 2021

1 Introduction

In this report, the implementation of the Interpreter for C- - will be outlined and explained. My code takes an abstract syntax tree (AST) as input, and gives the value returned by the main function as output. Sample input code for the interpreter is shown throughout this document, all of these inputs are test cases included in the test suite.

2 Interpretation

2.1 Linear Programs

In order to interpret linear programs, the AST produced by the parser is traversed in an order dictated by the value of the node discovered. A function was written called 'walk' which takes a tree (*node*) as an argument and returns an integer. A switch statement with the type of the node argument was created and depending on the type of the term in question, the function was called recursively on either the left child, right child or both.

2.1.1 Values

An example of this is the 'D' type, which indicates that a function definition is the left child, and the function body is the right child. To begin with, the function definition is not dealt with, so the function calls itself on the node's right child (which walks the function body). When it reaches a leaf node, it converts it into a TOKEN type and returns its value. This allows the program to interpret programs such as:

```
int main(void){return 5;}
```

>Answer is 5

2.1.2 Operators

In order to deal with basic arithmetic, cases for +, - / and * were added, which all carry out their respective functions on the left node and right node.

2.1.3 Variables

The next problem is variable definition, which required adding the supplied data structures, *frame*, *binding* and *value*. The return type of the walk function was changed to be a value. A new parameter, the environment, was added. The value is a union, which could take on an integer, boolean or string value. The environment is a frame struct, which is essentially a list of bindings. Whenever a variable definition was encountered (indicated by ∼), a binding is added to the environment using the provided function(*declaration_method*). This binding consists of

a *value* and a *token*(name), and the provided code declares this variable by adding the name with a value of 0.

When a variable is assigned, the provided function (*assignment_method*) is called, which iterates through each frame(in this case just one) and each binding within that frame. It attempts to find a binding with the same name as the input and if found, it updates the binding to reflect the input value.

When a variable is accessed (indicated by an identifier type), the provided function (*name_method*) searches for this variable in the same manner as *assignment_method* and returns the value of the binding.

In order to deal with programs that have multiple lines, a case was added for the ';' type, which would walk both the left and right trees, and return only the real return value (indicated return value of each walk).

Using these functions and structures, the interpreter could resolve code with variables such as:

```
int main(void){
    int x;
    x=5;
    return x;
}
```

>Answer is 5

However in order to deal with programs where the the declaration and assignment of a variable is on the same line, a separate case was added to the '∼' type where if the left node is '=' it is declared and then assigned. This addition meant the interpreter could deal with lines such as:

```
int x=5;
```

2.2 Non-Linear programs

2.2.1 Conditionals

The next step was to deal with conditionals, to do so, boolean values and operators have to be implemented. Typing was dealt with in a loose manner, i.e when a boolean value is required (e.g. in a condition) it would be used as such, otherwise it could simply be considered an integer. This works because booleans and integers are not actually different types in C. For example the operator '!=' would return either a 1 or a 0 and ,a value is created with a boolean type. This is read by conditional statements as true or false, but to everything else it would just be considered an integer.

From there, the implementation of conditionals is intuitive. The only processing required other than the if statement itself is checking whether an else is present. This check is shown in the code below:

```
if (term->right->type==ELSE){
    consequent = term->right->left;
    alternative = term->right->right;
}else{
    consequent = term->right;
    alternative = NULL;
}
return if_method(condition,consequent,alternative,env);
```

2.2.2 Functions

In order to process functions, the CLOSURE struct is required, which essentially stores a piece of code (represented by a tree) and the environment in which it is defined. For functions to be lexically scoped, it is important to capture the environment in which it is defined as this is where it looks for variables or functions called within.

To call functions, the walk function has to be able to return a closure. To do this, the closure structure is included in the value union, so a value can represent a closure. This made it simple to assign a closure to a variable name, as the existing value ↔ name relationship can be utilised from the bindings. There are multiple options for which part of the code should be stored in the closure. At first it made sense to simply store the function body. This is changed out of necessity later on.

In order to call functions, the two provided functions were used in the interpreter. The first is the *lexical_call_function* which given the name of the function, retrieves the code and environment using the previously implemented *name_method*. It then calls the other provided function *extend_frame*, with the arguments and the identifiers for the given closure. This function essentially matches each parameter with each argument (making a binding), and then adds these bindings to the given environment. This function was modified significantly in order to work with my code, but it still obtains the same result: A new environment made up of the caller environment plus the arguments mapped to the parameter names. Finally the *lexical_call_method* takes this and appends it on the function definition environment as the next frame.

This gives the final environment to pass into the walk function, which will run the function body code. In this code any variables will first be found within the parameters, and then the function definition environment and finally the caller environment.

After implementing these functions, interpreter is able to process code such as:

```
int main(void){
    int double(int x){return x+x;}
    int add(int x,int y){return x+y;}
    return add(double(2),double(2));
}
```

>Answer is 8

2.2.3 Recursion

To process recursive functions, the compiler must be able to call a function from within itself. With the existing framework, a function definition will capture the environment of its enclosing block, which includes the function itself. This means recursive functions should work, as the existing walk function in the interpreter is recursive by nature. However in practice, this didn't work because only the function body was stored in the closure for the function. This meant it could not access its own parameter values and was not able to call itself. To address this issue, the entire function was stored (node with type 'D') in the closure. This includes all the definition information and the body. The code was then refactored to deal with this.

With testing, the interpreter was able to process recursive programs. The following factorial function was able to successfully calculate factorials up to the maximum integer length.

```
int main(void){
    int factorial(int x){
```

```
        if(x==0){return 1;}
        else{return x*factorial(x-1);}
    }
    return factorial(10)
}
```

>Answer is 3628800

2.2.4 Functional Return Type

Because of the loose approach to typing, the interpreter was also able to deal with functions as return types. The walk function would simply walk it regardless of whether it is a function or integer and retrieve a value. This value would contain an integer or a closure and could be set and used either way. The following test is used to demonstrate its success.

```
int main(void){
    function cplus ( int a ) {
        int cplusa ( int b ) { return a+b; }
        return cplusa ;
    }
    int f = cplus(8);
    return f(4);
}
```

>Answer is 12

2.2.5 Functional Arguments

Much like the previous problem, this is addressed immediately by the loose approach to typing. Functions were able to take other functions as arguments, and use them as required. This test shows functional arguments working in the interpreter.

```
int main(void){
    function twice (function f) {
        int g( int x) {return f(f(x));}
        return g;
    }
    int double(int x){return x+x;}
    int quadruple = twice(double);
    return quadruple(2);
}
```

>Answer is 8

3 Conclusion and Evaluation

Overall, the interpreter implementation was a success, despite not being able to implement the TAC or machine code translation phases. The interpreter's implementation could be improved by delegating processes down to other functions, as many tasks were repeated. This would also help with the fact that some processes had to look quite far down the tree to extract necessary information. If it were reimplemented, these functions would certainly be split up and made more efficient. More efficient data structures and control mechanism could also be utilised, as

the ';' and ' ' both cause issues with deciding which return value to pick. It should also be noted that instead of setting a global environment, the first environment layer is simply passed down through each walk, meaning all the sub functions can access it. Additionally, strings were not implemented in the interpreter as they are quite low on the priority list.