Procesadores del Lenguaje

Curso 2020/2021

Global Practise

AUTOR1: Alejandro García Hermoso
AUTOR2: Allae Ghayat Zayoun

Puerto Real, 2020

Índice

1. Introduction

In this practise, we implemented a translator from C to assembler code (X86), and a Python interpreter of C code, that let us evaluate the generated AST by the parser. The whole project has been developed in Python, using the SLY library as the backbone of the implementation. First, we are going to explain the grammar used, then we will talk about some details of the code, like the entries that it accepts and which ones it rejects. Also, we will talk about the problems that we found while developing the code and its corresponding solutions.

2. Grammar

The grammar used has left and right recursion. When needed, rules has been factorized in order to avoid ambiguities. Sometimes, factorization has been avoided, when we considered that the factorization would make the code less legible.

This grammar accepts all the necessary rules for a correct implementation of the translator. With the proposed grammar we can write quite complete and complex programs that accept assignment instructions, arithmetic and logical operations, conditional statements, while-loops and int type declaration, functions are implemented too and local and global variables are differentiated successfully. The grammar is included in a attached txt file.

3. Problems found and their respective solutions

The majority of our problems have been found in the development of the Python Interpreter of C code, because it has been our main focus. Thanks to that, we have tested our AST representation a lot, and we now have the perfect structure to implement the x86 translation part of the project easily.

The first big problem that we found was the control flow in conditional statements, loops and functions. It was solved thanks to the addition of the class *StatementNode*, that is used to represent a C statement. Statements can be considered assignments, if and else instructions, calls to functions, expressions, etc. *StatementNode* stores two elements, the first is the nodes required to execute the statement itself, and the second one being the next statement to execute. This way, we can make lists of statements. If we have an if else block, there will be a list of statements for the if block and another one for the else block. Once the expression of the conditional statement is evaluated, the program chooses one of the paths. For functions, the idea is similar. Each function has a list of statements, and each time it is called, the list of statements is iterated through.

The second big problem was the scope statement in while-loops, if/else statements and functions. It was difficult to implement because we must to consider that when we enter and then get out from a scope, the variables created in it must be destroyed. The solution to that was adding a new dictionary(we

use dictionaries to save the variables from each scope) to the stack (the stack is composed of these dictionaries), so when we get out from this scope we eliminate this dictionary from the stack's top to eliminate the variables created at this scope. We save the actual state of the stack before entering the scope of a function and a new empty stack is created, where we add a dictionary with the values passed as arguments to the function. When we get out from the scope of the function, the stack is restored to its previous state. For creating, reading, and modifying variables in memory, we created a Python module called *MemoryOperations.py*.

4. Entries that accept the code

4.1. Python interpreter of C code

```c
int x = 3;
int y;

int fact(int n) {
    int acum = 1;

    while (n > 1){
        printf("n = %d",n);
        acum = acum * n;
        n = n - 1;
    }

    return acum;
}

int main(){
    y = 4;
    // z = 17; // Comentado porque daría error si no lo estuviera

    int x = 5;
    return fact(y);
}
```

Figura 1: First C source code example.

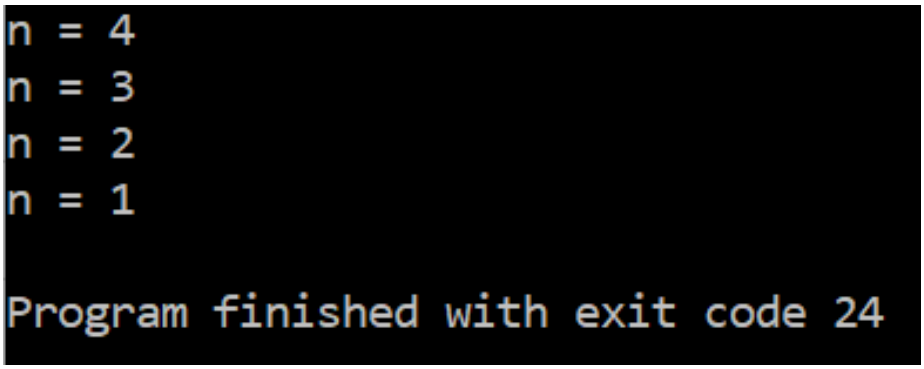
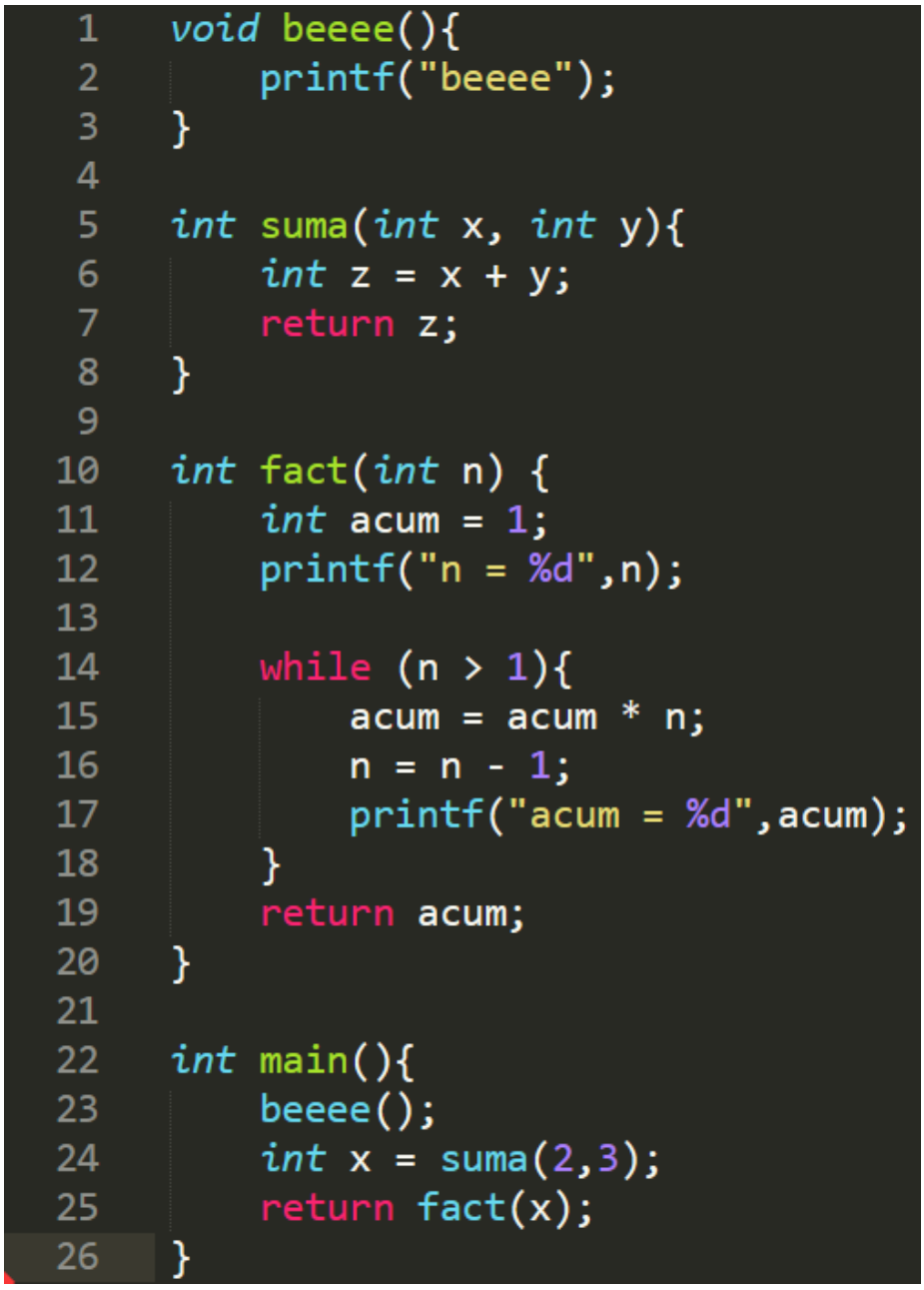


Figura 2: First C source code example's output.



```c
void beeee(){
    printf("beeee");
}

int suma(int x, int y){
    int z = x + y;
    return z;
}

int fact(int n) {
    int acum = 1;
    printf("n = %d",n);

    while (n > 1){
        acum = acum * n;
        n = n - 1;
        printf("acum = %d",acum);
    }
    return acum;
}

int main(){
    beeee();
    int x = suma(2,3);
    return fact(x);
}
```

Figura 3: Second C source code example.

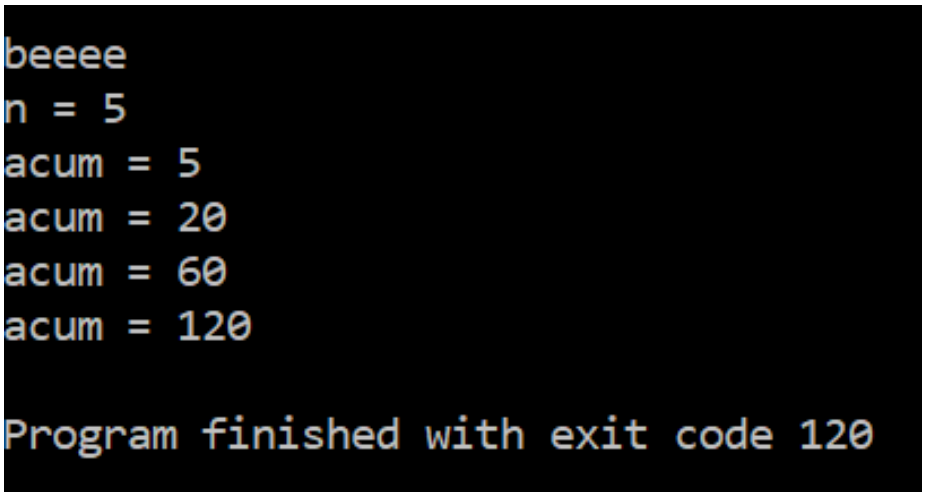


Figura 4: Second C source code example's output.

4.2. C to x86 translation

At this moment, we have done all the evaluation of the code and we only need to finish the compilation code, but we have some entries that we can prove as they are the arithmetic operations.

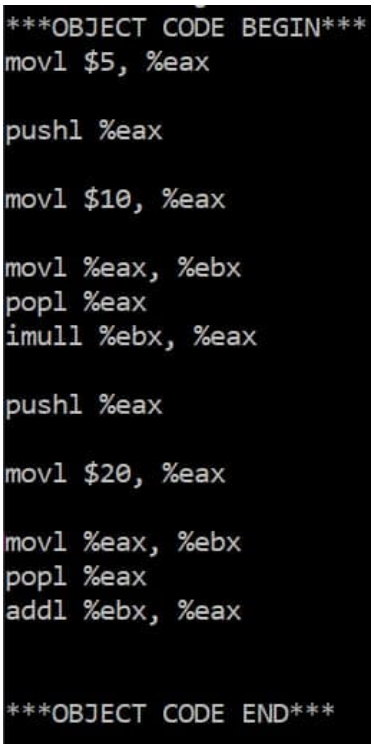


Figura 5: 5*10+20.