Documentation: C translator

January, 2025
Pablo Pérez Santiago
Agustín Eneas Harispe Lucarelli

INDEX

INDEX
INTRODUCTION
- Request
- Requirements / Objectives
  - Points to Accept

GRAMMAR
- Types
- Variable Declaration
- Assignment statements
- Operations
- Functions
- Printf
- Scanf
- If/Else
- While

TRANSLATION
- Assignment statements
  - N-Dimensional array indexing
- Operations
- Variable Declaration
- IF-ELSE and WHILE statements
- Problems with translations (a review)

Using the Translator
Grammar
Requirements

INTRODUCTION

Request

It is requested to design a grammar which accepts the C language, implement the grammar and create a program that translates C files into Assembly language.
In this text we'll discuss the process in fulfilling the assignment.

Requirements / Objectives

Embracing C language as a whole would be excessive. There has been given a list of keypoints that the grammar must accept to generate a subset of C language. Additional points or features can be added too.

Points to Accept

- Assignment statements
- Comparison operators (==, <=,>=, !=)
- Logical Operators (&&, ||, !)
- Arithmetic operators (+, -, *, / and unary minus)
- Variables and numerical constants
- Variables declaration rules (only int type)
- General functions from C language.
- Global variables
- Printf and Scanf functions
- AST for operations *
- Arrays, multidimensional matrix and pointers
- Nested conditional sentences
- Nested While loops
- Process global variables
- Type checking

GRAMMAR

Note: Some of the rules shown in the implementation have the purpose of applying the appropriate semantics for the grammar to work. They are labelled 'semantic rules' and are not shown in the grammar itself, they are referenced as the logic they implement.

```
S' -> Global
Global -> Funcion Global | Declaracion ';' Global |  Global ->
<empty>
```

Our translation always starts with these rules. Funcion will be explained in function, Declaracion in Declaration and we allow empty programs.

Types

The grammar accepts global and local integer variables and N-dimensional arrays.
Other C types are not present in our grammar. At first, we wanted to add float type but after learning how to translate it we decided to not add it.

Variable Declaration

We allow declaring int and N-dimensional arrays which can be declared alone or in packs. Ex.: “Int a;” or “Int a,b,c,d;” While declaring you can assign a value through the allowed operations, explained later.

```
Declaracion -> TIPO_ID '=' Operation
Declaracion -> TIPO_ID CORCHETES
Declaracion -> TIPO_ID
Declaracion -> Declaracion2 Declaracion3

Declaracion2 -> Declaracion2 Declaracion3 ','
Declaracion2 -> TIPO_ID '=' Operation ','
Declaracion2 -> TIPO_ID CORCHETES ','
Declaracion2 -> TIPO_ID ','

Declaracion3 -> ID '=' Operation
Declaracion3 -> ID CORCHETES
Declaracion3 -> ID
```

We check that new variables are not declared in this scope, otherwise the file is not accepted. We had a problem with Tipo Id so we turned it into the Tipo_ID rule, this is explained later in the function rules.

Assignment statements

```
Line -> Assign Operation
Assign -> Assign ID posCorchete '=' | <empty>
```

Assigning a value to an already declared variable is simple, it can be concatenated. Ex.: a = b = c = 4;
For arrays, brackets are required and will be checked in the grammar to ensure that the variable is assigned correctly.

Operations

```
Operation -> andOp OR Operation | andOp
andOp -> equalOp AND andOp | equalOp
equalOp -> compOp equalSymbol equalOp |compOp
compOp -> addOp compSymbol compOp | addOp
equalSymbol -> NOT_EQ | EQUAL | GR_EQ | LE_EQ
addOp -> prodOp | prodOp '-' addOp | prodOp '+' addOp
prodOp -> fact |  fact '/' prodOp | fact '*' prodOp
fact -> fcall | ID | ID CORCHETES | '(' Operation ')' | '!' fact
     | '-' fact | NUM
```

We allow OR, AND, ==, !=, <=, >=, +, -, *, /, ! and unary -. Operation is called through Assign, Declaration, Operation, IF_ELSE and WHILE. You can concatenate as many operations as you want, use literal int numbers, variables or function calls as the operating values.

Functions

```
   Funcion -> TIPO_ID '(' variables ')' '{' Input RETURN Operation ';' '}'
   Funcion -> VOID ID '(' variables ')' '{' Input '}'
```

Rule 'Funcion' allows the use of functions of type void (with no return value) and with every type allowed in our grammar ('int').
Example:

```
int fun_example(int a, int b) {
/logic/
return return_value}
```

TIPO_ID / VOID ID:
There was a need for an additional rule 'TIPO_ID -> TIPO ID' since otherwise a shift/reduce conflict appeared, trying to reduce to the set of declaration rules that starts with 'TIPO ID' or shift and continue reading for the current rule explained here.

Variables:

```
variables -> empty | listavars
listavars -> listavars ',' TIPO ASTERISCO ID posCorchete| TIPO ASTERISCO ID
posCorchete

Funcion -> TIPO_ID setcurrentAmbito '(' variables ')' setcurrentFunction '{'
Input RETURN Operation ';' '}'
```

Two additional semantic rules are introduced. First, `setCurrentAmbito`, which recovers the scope of the variables in the `variables` rule to allow checking if the variable has been declared. Second, `setCurrentFunction`, which adds the ID of the current function to a dictionary, enabling recursion to work.

Function Calls:

We ensure that a function is declared before it is called; you cannot call a function without it being implemented. Additionally, it is worth noting that there is no standalone declaration per se – each function is defined immediately, meaning it is both declared and implemented at the same time.

We check whether the entry values are declared in the current scope. If they are not, we check if they are global. If neither condition is met, the values are not translated, as this indicates an error. Entry values can include integers, N-dimensional arrays, and integer pointers.

```
fcall -> ID '(' entradaID ')'
entradaID -> listaID | <empty>
listaID -> listaID ',' AMPERSAN ID posCorchete | AMPERSAN ID
posCorchete
AMPERSAN -> '&' | <empty>
posCorchete -> CORCHETES | <empty>
```

Printf

```
Line -> PRINTF '(' CADENA_SCANF , AuxPrintf ')'
Line -> PRINTF '(' CADENA ',' AuxPrintf ')'
Line -> PRINTF '(' CADENA ')' | Declaracion | Assign Operation
AuxPrintf -> AuxPrintf ',' ID posCorchete | ID posCorchete
```

CADENA is a token where any character combination between “ is valid. There is only one exception which is CADENA_SCANF which is (%(d|i|u|f|s)|)+ between “. We have three rules to use printf. First with a string like CADENA_SCANF. Second with a string like CADENA and using variables. Last one is for only string like CADENA when there is no %(d|i|u|f|s). We check %(d|i|u|f|s) type is the same as variable. AuxPrintf allows using more than one variable on a printf.

CADENA is a token that represents any character combination enclosed within quotation marks ("). The only exception is CADENA_SCANF, which specifically represents a pattern of (%(d|i|u|f|s)|)+ enclosed within quotation marks.
We have three rules for using printf:

The first rule applies when the format string is like CADENA_SCANF.
The second rule is for format strings like CADENA that include variables.
Last rule is for strings like CADENA when there are no format specifiers such as %(d|i|u|f|s). We ensure that the type specified in %(d|i|u|f|s) matches the corresponding variable's type. Additionally, AuxPrintf allows the use of multiple variables in a single printf call.

Scanf

```
Line -> SCANF '(' CADENA_SCANF ',' AuxScanf ')'
AuxScanf -> AuxScanf ',' ID | ID | AuxScanf ',' '&' ID | '&' ID
```

Similar idea to the printf rule but there is no need to allow strings that are not like CADENA_SCANF. We check variables are declared before using them.

If/Else

```
Condicional -> IF '(' Operation ')'; Condicional_ELSE
Condicional -> IF '(' Operation ')' '{' Input '}' Condicional_ELSE
Condicional -> IF '(' Operation ')' Line ';' Condicional_ELSE
Condicional_ELSE -> ELSE '{' Input '}' |  ELSE Line ';' | <empty>
```

We allow a conditional with no rule, a block of rules or a single rule. You can use an else after any of them with a block of rules or a single rule. You can use all previous rules but declaration in an if_else structure does not create an if_else scope. They work as if they were outside the structure meaning they still exist after you exit the structure. If_Else can be nest.

While

```
Bucle -> WHILE '(' Operation ')' '{' Input '}'
     | WHILE '(' Operation ')' Line ';'
     | WHILE '(' Operation ')' ';'
```

We allow a while loop with no rule, a block of rules or a single rule. Same as in If_Else structures all previous rules except declarations which are supported by the grammar to work but they do not work as expected. You can nest as many loops and if_else structures as you want.

TRANSLATION

Our translator reduces from bottom to top meaning the first rules to run will be the token values. That is why we decided that after a token value it will be put in the stack and it is the next rule's responsibility to know how many values they have to take from the stack.

Assignment statements

Once an assignment rule is reduced we take a single value from the stack which will be the value assigned to the variable in the rule. Ex.:
a = b = 0, will be translated as

```
popl %eax
movl %eax, -8(%ebp)
pushl -8(%ebp)    # this last push allows next variables to take the correct value
popl %eax
movl %eax, -4(%ebp)
pushl -4(%ebp) # this last push allows next variables to take the correct value
popl %eax   # at the end of the assignments we clear the stack residual value
```

We took into account that an assign rule can be empty and then the top value in the stack is incorrect. That is why we add a ‘popl %eax’ if you do not assign or use, for example in a condition, the partial result.

N-Dimensional array indexing

For addressing an N-dimensional array, we need to convert its **N-dimensional subindexes** into a **linear index**, as assembly stores arrays in a contiguous block of memory (a one-dimensional structure). This way we can map the multidimensional representation of the array into the linear memory layout used by the computer.

The conversion is achieved using a formula that calculates the **linear index** from the given subindexes and the dimensions of the array.

Example:

For a 2D array of size D1×D2 the element at Subindex_i=(r,c) is calculated as: **Linear_Index** = r x D2 + c

For a 3D array for Subindex_i=(r,c,k) the formulation would be as following: **Linear_index** = r x (D2 x D3) + c x D3 + k

N-D Array: With this idea, we generate a general formula...

$$\text{Linear\_index} = \sum_{i=0}^{N-1} (Subindex_i \; x \prod_{j+i+1}^{N-1} Dimension_j)$$

Note: It is necessary to translate for the stack pointer (multiply by -4) to address the elements correctly.

Operations

Operations take two values from the stack and once the operation is over they push the partial result in the stack. All tags have a counter which increases as more are used.All operations translations:

Add:
popl %ebx
popl %eax
addl %ebx, %eax
pushl %eax

Multiplication:
popl %ebx
popl %eax
imull %ebx, %eax
pushl %eax

Unary Subtraction:
popl %ebx
movl &0,%eax
subl %ebx, %eax
pushl %eax

Subtraction:
popl %ebx
popl %eax
subl %ebx, %eax
pushl %eax

Division:
popl %ebx
popl %eax
cdq
divl %ebx
pushl %eax

Equal:

```
        popl %ebx
        popl %eax
        compl &ebx,%eax
        jne cond_false0
        pushl &1
        jmp cond_final0
cond_false0:
        pushl &0
cond_final0:
```

Not Equal:

```
        popl %ebx
        popl %eax
        compl &ebx,%eax
        je cond_false0
        pushl &1
        jmp cond_final0
cond_false0:
        pushl &0
cond_final0:
```

Greater or Equal:

```
        popl %ebx
        popl %eax
        compl &ebx,%eax
        jl cond_false0
        pushl &1
        jmp cond_final0
cond_false0:
        pushl &0
cond_final0:
```

Lower or Equal:

```
        popl %ebx
        popl %eax
        compl &ebx,%eax
        jg cond_false0
        pushl &1
        jmp cond_final0
cond_false0:
        pushl &0
cond_final0:
```

Negation:

```
        popl %eax
        cmpl $0, %eax
        jne cond_final0
        movl $0, %eax
cond_final0:
        pushl %eax
```

And:

```
        popl %eax
        compl &0,%eax
        je cond_false0
        popl %eax
        compl &0,%eax
        je cond_false0
        pushl &1
        jmp cond_final0
cond_false0:
        pushl &0
cond_final0:
```

Or:

```
        popl %eax
        compl &0,%eax
        jne cond_true0
        popl %eax
        compl &0,%eax
        je cond_true0
        pushl &0
        jmp cond_final0
cond_true0:
        pushl &1
cond_final0
```

Variable Declaration

Variable Declaration can be done everywhere in the code. We made each Declaration into a valid translation but it would be better to reserve all memory at the same point. Ex.:

int a,b;

```
subl $4, %esp
subl $4, %esp
```

int a = 2, b = 4;

```
subl $4, %esp
popl %eax
movl %eax, -12(%ebp)
subl $4, %esp
popl %eax
movl %eax, -16(%ebp)
```

IF-ELSE and WHILE statements

The translator correctly manages to produce distinct tags for the conditions required in these blocks, as well as the tags needed for the jumps.

While Statements:

- Use tags like `start_whileX` and `final_whileX` to manage loops.
- Evaluate the condition at the beginning and loop back if true.

If Statements:

- Use tags like `elseX` and `if_else_finalX` to manage control flow.
- Separate true and false blocks with conditional jumps.

In this implementation, variables declared inside `while` or `if` blocks **do not have a separate scope**. They are treated as if declared outside the block, remain accessible even after the block ends, and are managed using the same memory location throughout the function.

Problems with translations (a review)

In general, as more rules were added, it became increasingly difficult to maintain a structured approach for new additions to the program. Without a clear understanding of the final outcome and lacking experience with grammars of this complexity, many decisions made during the process were adjusted on the fly—sometimes preemptively and other times belatedly.

One example is the use of tables to store variables. The amount of information and the meaning assigned to certain default values evolved over

time, eventually leading to a structure with three fields, where an integer type could be treated as an array of size one.

Although generating highly efficient assembly code was not required—thus reducing the number of cases to handle—finding the general case became a significant challenge. There are limited references on the subject, so we relied heavily on lecture materials and logical reasoning to implement the solution.

Using the Translator

To use the translator, begin by running the script in a terminal or command prompt. Upon execution, the program will prompt you to enter the name of a C file, such as `hola.c`, which will serve as the input for translation. If no filename is provided, the translator defaults to processing a file named `prueba.c`.

If no errors are encountered during parsing, the program writes the resulting translation to a `.txt` file. This file is saved in the same directory as the script and is named using the format `traduccion<filename>.txt`. For example, if the input file is `hola.c`, the output file will be `traduccionhola.c.txt`.

In addition to generating the translated assembly code, the program provides diagnostic information. It prints details about all variables and functions defined in the input file. For variables, it includes their name, scope, type, size, and memory location relative to the stack frame (`EBP`). For functions, it displays their name, return type, and parameters.

Appendix

Grammar

```
S' -> Global
Global -> Funcion Global | Declaracion ';' Global |  Global -> <empty>
Funcion -> VOID ID '(' variables ')' '{' Input '}'
Funcion -> TIPO_ID '(' variables ')' '{' Input RETURN Operation ';' '}'
Input -> Input Bucle | Input Condicional | Input Line ';' | <empty>

Line -> SCANF '(' CADENA_SCANF ',' AuxScanf ')'
Line -> PRINTF '(' CADENA_SCANF , AuxPrintf ')'
```

```
Line -> PRINTF '(' CADENA ',' AuxPrintf ')'
Line -> PRINTF '(' CADENA ')' | Declaracion | Assign Operation

AuxPrintf -> AuxPrintf ',' ID posCorchete | ID posCorchete
AuxScanf -> AuxScanf ',' ID | ID | AuxScanf ',' '&' ID | '&' ID

Condicional -> IF '(' Operation ')'; Condicional_ELSE
Condicional -> IF '(' Operation ')' '{' Input '}' Condicional_ELSE
Condicional -> IF '(' Operation ')' Line ';' Condicional_ELSE
Condicional_ELSE -> ELSE '{' Input '}' |  ELSE Line ';' | <empty>

Bucle -> WHILE '(' Operation ')' '{' Input '}'
      | WHILE '(' Operation ')' Line ';'
      | WHILE '(' Operation ')' ';'

Declaracion -> TIPO_ID '=' Operation | TIPO_ID CORCHETES  | TIPO_ID
      | Declaracion2 Declaracion3
Declaracion2 -> Declaracion2 Declaracion3 ','
      | TIPO_ID '=' Operation ',' | TIPO_ID CORCHETES ',' | TIPO_ID ','
Declaracion3 -> ID '=' Operation | ID CORCHETES | ID
TIPO_ID -> TIPO ID
TIPO -> INT
CORCHETES -> CORCHETES '[' NUM ']' | '[' NUM ']'

Assign -> Assign ID posCorchete '=' | <empty>

Operation -> andOp OR Operation | andOp
andOp -> equalOp AND andOp | equalOp
equalOp -> compOp equalSymbol equalOp |compOp
compOp -> addOp compSymbol compOp | addOp
equalSymbol -> NOT_EQ | EQUAL | GR_EQ | LE_EQ
addOp -> prodOp | prodOp '-' addOp | prodOp '+' addOp
prodOp -> fact |  fact '/' prodOp | fact '*' prodOp
fact -> fcall | ID | ID CORCHETES | '(' Operation ')' | '!' fact
      | '-' fact | NUM

variables -> listavars | <empty>
listavars -> listavars ',' TIPO ASTERISCO ID posCorchete
      | TIPO ASTERISCO ID posCorchete
ASTERISCO -> ASTERISCO '*' | <empty>
posCorchete -> CORCHETES | <empty>
fcall -> ID '(' entradaID ')'
entradaID -> listaID | <empty>
listaID -> listaID ',' AMPERSAN ID posCorchete | AMPERSAN ID posCorchete
AMPERSAN -> '&' | <empty>
```