Scanning and Parsing of YACLL: Yet Another C Like Language

Group ID: 15

1. Introduction

We propose a Yet Another C-Like Language (YACLL) which is syntactically and semantically similar to C language with some modifications. These modifications vary in complexity, like from just changing ‘float’ to ‘fp’ up to like providing a powerful construct for tensor (multi-dimensional matrix) operations.

We also introduce some little nuance features like digit separators (from Python) and optional labeled loops (from Rust) which are very much liked by the programmers community (atleast me).

Currently YACLL does not support preprocessing and other useful constructs which make C very flexible, however we plan to revise YACLL and add more features as we go through the development of the compiler.

2. Lexical Structure

2.1 Identifiers

- An identifier starts with a letter (a-z, A-Z) or underscore (_)
- Followed by letters, digits, or underscores
- Case-sensitive

2.2 Keywords

- Reserved words: int, fp, char, text, void, bool, struct, const, sizeof, return
- Cannot be used as variable, function, or struct names.

2.3 Literals

- **Integer (int):** `10`, `-42`, `0`
- **Floating point (fp):** `3.14`, `-0.001`, `10.0`
- **Character (char):** `'a'`, `'\n'`

- **String (text):** `"hello"`, `"pointer world"`
- **Boolean (bool):** `true`, `false`

2.4 Data Types

a. **Primitive Types**
   - int → signed integer
   - fp → floating-point number
   - char → single character
   - text → string
   - bool → boolean (true/false)
   - void → no value

b. **Pointer Types**
   - Any type can be a pointer: int *p;, char *c;
   - Multiple levels allowed: int **pp;
   - Struct pointers allowed: struct Node *n;

c. **const Qualifier**
   - Declared using const
   - Must be initialized during declaration
   - Example: const int x = 5;

2.5 Operators

a. **Arithmetic**
   - `+ - * / % **`

b. **Relational**
   - `< > <= >= == !=`

c. **Logical**
   - `&& || !`

d. **Assignment**
   - `=`

e. **Pointer**
   - ***** → dereference
   - **&** → address-of

2.6 Expressions

- Follow C-style precedence.
- Parentheses allowed for grouping.
- Function calls allowed: sum(a,b)

- **`sizeof`** allowed: **`sizeof(int)`**, **`sizeof(x)`**

2.7 Statements

a. **Expression statement**
   - `a = b + c;`
   - `foo(x,y);`

b. **Declaration statement**
   - `int x;`
   - `fp y = 3.14;`
   - `char *p;`

c. **Return statement**
   - `return;`
   - `return x;`

2.8 Blocks and Scope

- All blocks must use { }
- No single-line blocks allowed.

Valid:

```
if (x > 0) {
        return x;
    }
```

2.9 Control Flow

a. **Conditionals**
   - `if (condition) { }`
   - `if-else`
   - `if-else-if`
   - Condition must be a valid expression.

b. **Loops**
   - Loop constructs supported (syntax defined separately if implemented).

2.10 Functions

a. **Declaration**
   - `int sum(int a, int b);`

b. **Definition**

```
int sum(int a, int b) {
   return a + b;
}
```

c. **Function Call**
   - `sum(10,20);`
   - `foo();`
   - `bar(x+y, *p);`

2.11 Structures (struct)

a. **Definition**

```
struct Point {
   int x;
   int y;
};
```

b. **Variables**
   - `struct Point p;`
   - `struct Point *ptr;`

c. **Member Access**
   - Dot: `p.x = 10;`
   - Arrow: `ptr->y = 20;`

2.12 sizeof Operator

- `sizeof(int)` - Works on type or expression.

2.13 Grammar Constraints & Notes

- Every statement must end with **;**, except:
  - Block statements
  - if / else blocks
- No preprocessing (**#include, #define**)

2.14 Out-of-Scope (Not Supported)

- Ternary operator (?:)
- Function pointers
- Preprocessor directives
- Memory management semantics

3. Additional Features

3.1 Variadic Functions

Inspired by Rust, we add support for optional loop labels so that we can specify which loop we want to 'break' or 'continue'.
The loop label has to be a valid identifier name;
A loop label is only valid in the context of the same loop that is defined by the block structure. So using a loop label which has not been declared must give a compile time error.
When defining a loop label, the label is prefixed with an apostrophe (‘`’).
Example:

```
‘loop_label1:
     while(True){
          int i = 0;
          while(True){
               if(i > 5){
                    break loop_label1;
               }
               i = i + 1;
          }
```

As in Python, numbers with many digits can be separated by '_' for much better readability. We intend to implement the same capability in our language for numeric data types that is **int** and **float**.

3.2 Variadic Functions

We propose a newer syntax for variadic functions (functions taking variable number of arguments).
In C, the method to use variadic functions is by declaring the function with '...' as the last parameter, and in the definition, **va_list** and **va_start** are used to iterate over the arguments.
We explain the new syntax with the following example:

```
void greet(char* person1, char* person2, ...){
     int arg_count = greet.args_count;
     void *args = greet.args_list;
}
```

1. For functions we define special attributes when which are accessible in the direct scope of the function. For now we limit these attributes to **args_count** and **args_list**.
2. These attributes can be accessed with struct-like member access notation.
3. The data type of **args_count** is 8 bit unsigned integer, which in C typically corresponds to unsigned char. (I mean, who would want even that much variadicity?)
4. The data type of **args_list** is void*

3.3 Tensor Data Type

```
tensor<T, d1, d2, ..., dn> a;
```

T is the data type of the elements in the tensor, there are n dimensions in it, and d1 to dn are the extents of the respective dimensions, a is the name of the tensor.

There is a provision of the keyword item to iterate through tensors without writing nested loops. This is associated with the keyword axis to control the order of iteration.

```
for item in a axis(a0,a1,a2,..,an)
{
      T x = item.value;
      int i = item.indices[0];
}
```

No axis should appear more than once and they should all lie between 0 and n, default will be 0 to n, if you specify less than the remaining will be auto-filled in ascending order. Axis-specification does not change the index tuple order.

The indices will be in order, they can also be given to a list of length n for ease.

Operation	**Operator**	**Type checking**
Indexing	`[i][j][k]`, for three dimensional tensor	Index should be in range, number of indices should be equal to number of dimensions
Slicing	`[i, : , k]` for three or more dimensional tensor	Indices should be in range, number of elements in slice operator should be equal to

		number of dimensions, if less, it is assumed that the rest are “ : ”
Element-wise addition	.+	Both operands must be tensors of identical shape; base type must be numeric. Result shape equals operand shape.
Element-wise subtraction	.-	Both operands must be tensors of identical shape; base type must be numeric. Result shape equals operand shape.
Element-wise multiplication	.*	Both operands must be tensors of identical shape; base type must be numeric. Result shape equals operand shape.
Element-wise division	./	Both operands must be tensors of identical shape; base type must be numeric. Result shape equals operand shape. Run-time error in case of division by zero
Tensor contraction (dot product for n=1, matrix multiplication for n=2)	@	In case of n>1, the last dimension of the first tensor should be equal to the first dimension of the second tensor; base type must be numeric.
Tensor product	@*	Base type of both tensors must be numeric

4. Example

```
int num1 = 3;
int num2 = 4;

struct Pair {
    int a;
    int b;
};

int add(struct Pair *p) {
    return p->a + p->b;
}

int main() {
    struct Pair *x;
    x->a = num1;
    x->b = num2;
    return add(x);
}
```

OUTPUT

```
***parsing successful***
#global_declarations = 5
#function_definitions = 2
#struct_definitions = 1
#pointer_declarations = 2
#if_statements = 0
#for_statements = 0
```

```
#function_calls = 1
***parsing completed***
```