C-ássio
Uma linguagem de programação

Gabriel Rodrigues[1], **Giovanni G. De Giacomo**[1]

[1]Centro de Ciências Computacionais – Universidade Federal do Rio Grande (FURG)
Caixa Postal 474 – 96.201-900 – Rio Grande – RS – Brazil

Resumo. *Este artigo tem como objetivo apresentar a linguagem de programação C-ássio que busca permitir a utilização de elementos de baixo e alto nível para uma programação que ofereça performance e abstração. A linguagem será descrita em suas formas gramaticais formais e informais. No fim da apresentação será apresentado um exemplo de um código utilizando os vários elementos oferecidos pelo C-ássio.*

1. Introdução

A linguagem de programação C-ássio, cujo objetivo é misturar elementos de alto e baixo nível para obter uma linguagem híbrida capaz de proporcionar abstração e performance em um único pacote, será apresentada no decorrer deste documento. Na próxima seção, a gramática do C-ássio será apresentada de maneira informal e na seção seguinte sua Turing-Completude será justificada. Após isso, serão apresentadas as notações formais de Backus-Naur e Wirth. Por fim, será apresentado um código de exemplo.

2. Descrição Informal

Nesta seção será apresentada a descrição informal da linguagem de programação C-ássio. Apos isso, na seção seguinte serão apresentadas as gramaticas formais em duas notações distintas.

programa: data *declarações* code *funções*
tipo: byte, word, dword, qword
argumento: *nome*, [*nome*], *registrador*
registrador: rax, rbx, rcx, rdx, rdi, rsi
declarações:
- simples: *tipo nome*;
- vetor: *tipo nome* ($\mathbb{Z}^+$);
- estruturas: struct *nome* { *declarações* };

parâmetros:
- forma 1: *tipo nome*
- forma 2: *registrador*

funções: function *nome* (*parâmetros*) ← *tipo* { *comandos* };
comando de atribuição: *argumento* ← *expressão*;
comando condicional: if (*condição*) { *comandos* };
comando iterativo: while (*condição*) { *comandos* };
comando de entrada: *argumento* ← input();
comando de saída: output(*argumento*);

3. Turing-Completude

A linguagem C-ássio é turing-completa devido ao seu suporte a instruções sequenciais, instruções de controle (if) e seu suporte a instruções de iteração (while). Portanto, pelo Teorema de Böhm Jacopini, pode-se afirmar que o C-ássio é turing-completo. [Mogensen 2017]

4. Descrição Formal

Nesta seção são apresentadas as descrições formais nas formas de Backus-Naur e Wirth para a linguagem de programação C-ássio. [Hopcroft 2013]

4.1. Forma de Backus-Naur

⟨*Program*⟩ ::= data ⟨*Declaration*⟩ code ⟨*Function*⟩

⟨*Capacity*⟩ ::= ϵ
| (⟨*Number*⟩)

⟨*Declaration*⟩ ::= ϵ
| byte ⟨*Identifier*⟩⟨*Capacity*⟩; ⟨*Declaration*⟩
| word ⟨*Identifier*⟩⟨*Capacity*⟩; ⟨*Declaration*⟩
| dword ⟨*Identifier*⟩⟨*Capacity*⟩; ⟨*Declaration*⟩
| qword ⟨*Identifier*⟩⟨*Capacity*⟩; ⟨*Declaration*⟩
| struct ⟨*Identifier*⟩ { ⟨*Declaration*⟩ }; ⟨*Declaration*⟩

⟨*Function*⟩ ::= ϵ
| function ⟨*Identifier*⟩ (⟨*Parameter*⟩) ⟨*Return Type*⟩ { ⟨*Instruction*⟩ }; ⟨*Function*⟩

⟨*Return Type*⟩ ::= ϵ
| ← ⟨*Type*⟩

⟨*Parameter*⟩ ::= ϵ
| byte ⟨*Identifier*⟩⟨+*Parameter*⟩
| word ⟨*Identifier*⟩⟨+*Parameter*⟩
| dword ⟨*Identifier*⟩⟨+*Parameter*⟩
| qword ⟨*Identifier*⟩⟨+*Parameter*⟩
| rax⟨+*Parameter*⟩
| rbx⟨+*Parameter*⟩
| rcx⟨+*Parameter*⟩
| rdx⟨+*Parameter*⟩
| rdi⟨+*Parameter*⟩
| rsi⟨+*Parameter*⟩

⟨+*Parameter*⟩ ::= ϵ
| , ⟨++*Parameter*⟩

⟨++*Parameter*⟩ ::= byte ⟨*Identifier*⟩⟨+*Parameter*⟩
| word ⟨*Identifier*⟩⟨+*Parameter*⟩
| dword ⟨*Identifier*⟩⟨+*Parameter*⟩
| qword ⟨*Identifier*⟩⟨+*Parameter*⟩
| rax⟨+*Parameter*⟩

| rbx⟨+*Parameter*⟩
| rcx⟨+*Parameter*⟩
| rdx⟨+*Parameter*⟩
| rdi⟨+*Parameter*⟩
| rsi⟨+*Parameter*⟩

⟨*Type*⟩ ::= byte
| word
| dword
| qword

⟨*Instruction*⟩ ::= ϵ
| if (⟨*Expression*⟩) { ⟨*Instruction*⟩ }; ⟨*Else*⟩ ⟨*Instruction*⟩
| while (⟨*Expression*⟩) { ⟨*Instruction*⟩ }; ⟨*Instruction*⟩
| [_a-zA-Z][_a-zA-Z0-9]+⟨*Child*⟩⟨+*Instruction*⟩; ⟨*Instruction*⟩
| [⟨*Identifier*⟩⟨*Child*⟩]⟨*Assignment*⟩; ⟨*Instruction*⟩
| rax⟨*Assignment*⟩; ⟨*Instruction*⟩
| rbx⟨*Assignment*⟩; ⟨*Instruction*⟩
| rcx⟨*Assignment*⟩; ⟨*Instruction*⟩
| rdx⟨*Assignment*⟩; ⟨*Instruction*⟩
| rdi⟨*Assignment*⟩; ⟨*Instruction*⟩
| rsi⟨*Assignment*⟩; ⟨*Instruction*⟩
| return ⟨*Expression*⟩; ⟨*Instruction*⟩

⟨+*Instruction*⟩ ::= ϵ
| ←⟨*Expression*⟩
| (⟨*Argument*⟩)

⟨*Else*⟩ ::= ϵ
| else { ⟨*Instruction*⟩ };

⟨*Argument*⟩ ::= ϵ
| [_a-zA-Z][_a-zA-Z0-9]+⟨*Child*⟩⟨*Call*⟩⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩⟨*MoreArgume*
| [⟨*Identifier*⟩⟨*Child*⟩]⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩⟨*MoreArgument*⟩
| rax⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩⟨*MoreArgument*⟩
| rbx⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩⟨*MoreArgument*⟩
| rcx⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩⟨*MoreArgument*⟩
| rdx⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩⟨*MoreArgument*⟩
| rsi⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩⟨*MoreArgument*⟩
| rdi⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩⟨*MoreArgument*⟩
| [0-9]+⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩⟨*MoreArgument*⟩
| (⟨*Expression*⟩)⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩⟨*MoreArgument*⟩
| not ⟨*Expression*⟩⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩⟨*MoreArgument*⟩
| ”[a-zA-Z_]*”⟨*MoreArgument*⟩

⟨*MoreArgument*⟩ ::= ϵ
| ,⟨*Value*⟩⟨*MoreArgument*⟩

⟨*Assignment*⟩ ::= ϵ
| ←⟨*Value*⟩

⟨*Value*⟩ ::= ϵ
| [_a-zA-Z][_a-zA-Z0-9]+⟨*Child*⟩⟨*Call*⟩⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| [⟨*Identifier*⟩⟨*Child*⟩]⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| rax⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| rbx⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| rcx⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| rdx⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| rsi⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| rdi⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| [0-9]+⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| (⟨*Expression*⟩)⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| not ⟨*Expression*⟩⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| ”[a-zA-Z_]*”

⟨*Call*⟩ ::= ϵ
| (⟨*Argument*⟩)

⟨*Child*⟩ ::= ϵ
| .⟨*Identifier*⟩⟨*Child*⟩

⟨*Expression*⟩ ::= [_a-zA-Z][_a-zA-Z0-9]+⟨*Child*⟩⟨*Call*⟩⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| [⟨*Identifier*⟩⟨*Child*⟩]⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| rax⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| rbx⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| rcx⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| rdx⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| rsi⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| rdi⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| [0-9]+⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| (⟨*Expression*⟩)⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩
| not ⟨*Expression*⟩⟨*Low*⟩⟨*Math*⟩⟨*Logical*⟩⟨*Boolean*⟩

⟨*Boolean*⟩ ::= ϵ
| and ⟨*Expression*⟩
| or ⟨*Expression*⟩

⟨*Logical*⟩ ::= ϵ
| == ⟨*Expression*⟩
| != ⟨*Expression*⟩
| $>$⟨*Expression*⟩
| $<$⟨*Expression*⟩
| $>=$ ⟨*Expression*⟩
| $<=$ ⟨*Expression*⟩

⟨*Math*⟩ ::= ϵ
| + ⟨*Expression*⟩
| - ⟨*Expression*⟩

⟨*Low*⟩ ::= ϵ
| * ⟨*Expression*⟩
| / ⟨*Expression*⟩

| % ⟨*Expression*⟩

⟨*Identifier*⟩ ::= [_a-zA-Z][_a-zA-Z0-9]+

⟨*Number*⟩ ::= [0-9]+

5. Código

```
data
    struct Point {
        dword x;
        dword y;
    };

    qword num1;
    qword num2;

code
    function sum(qword x, qword y) <- qword {
        return [x] + [y];
    };

    function cassio() {
        [num1] <- inputi();
        [num2] <- inputi();

        if (([num1] + [num2]) > 10) {
            rax <- sum([num1], [num2]);
            outputi(rax);
        };
        else {
            outputs("Prefiro me comunicar com as maquinas.");
        };
    };
```

Referências

Hopcroft, J. E. (2013). *Introduction to Automata Theory, Languages and Computation: For VTU, 3/e*. Pearson Education India.

Mogensen, T. Æ. (2017). *Introduction to compiler design*. Springer.