Relatório de Desenvolvimento

Alexis Castro Correia (A102495)
João de Albuquerque Ferreira Vieira da Fonseca (A102512)

4 de janeiro de 2025

Resumo

Este relatório descreve o desenvolvimento de uma linguagem de programação imperativa simples, juntamente com seu compilador correspondente, conforme especificado na declaração do projeto.A linguagem foi projetada para facilitar a declaração de variáveis atômicas inteiras, a execução de operações aritméticas, relacionais e lógicas, bem como instruções de fluxo de controle (seleção e iteração). Além disso, recursos adicionais foram implementados, incluindo a manipulação de arrays unidimensionais ou bidimensionais e a definição e invocação de subprogramas sem parâmetros.

O compilador, criado com o auxílio das ferramentas PLY/Python, traduz o código-fonte escrito nesta linguagem em pseudocódigo e Assembly para execução em uma Máquina Virtual (VM). Este documento engloba a descrição do processo de implementação, exemplos de código na nova linguagem e validação através de casos de teste.

Conteúdo

Capítulo 1

Introdução

Este relatório descreve o processo de concepção e implementação de uma linguagem de programação imperativa simples, bem como o desenvolvimento de um compilador para traduzir programas escritos nessa linguagem para Assembly de uma máquina virtual (VM). Este trabalho foi realizado no contexto da disciplina *Processamento de linguagens e compiladores*, com o objetivo de consolidar os conhecimentos sobre gramáticas formais, construção de compiladores e geração de código Assembly.

A linguagem foi projetada para suportar as principais funcionalidades de linguagens de programação imperativas, incluindo a declaração de variáveis atômicas, manipulação de estruturas de controle de fluxo, execução de operações aritméticas e lógicas, leitura e escrita em *standard input/output*, e a implementação de ciclos. Além disso, foram incorporadas funcionalidades opcionais, como o suporte a *arrays* ou subprogramas, de acordo com os requisitos estabelecidos no enunciado do projeto.

O relatório está organizado da seguinte forma: inicialmente, apresenta-se o contexto e os requisitos para a definição da linguagem e do compilador. Em seguida, detalha-se o processo de construção da gramática independente de contexto (GIC) e a utilização de ferramentas como Yacc/Lex ou PLY/Python. Posteriormente, discute-se a implementação do compilador, incluindo a geração de código Assembly e a execução em uma máquina virtual. Por fim, são apresentados os testes realizados, a análise dos resultados obtidos e as conclusões gerais sobre o trabalho.

Este projeto oferece uma experiência prática em todas as etapas de desenvolvimento de uma linguagem de programação e de um compilador, reforçando conceitos fundamentais da teoria da computação e da engenharia de software.

Capítulo 2

Contextualização

2.1 Descrição do Problema

O objetivo deste projeto é criar e implementar instruções simples que permitam aos programadores realizar tarefas simples, como declarar variáveis, executar código, controlar o fluxo de opções e padrões de iteração e ler/escrever dados. Além disso, a linguagem deve incluir recursos adicionais, como suporte para matrizes ou sub-rotinas, para fornecer uma base sólida para a resolução de problemas básicos de computador. É um pseudocódigo e um código assembly que permite que ele seja executado em uma máquina virtual. Portanto, o principal problema é criar um sistema eficaz e eficiente que atenda ao processo de ensino específico e leve em consideração as limitações e características do idioma.

2.2 Linguagem

O nosso grupo decidiu escolher a linguagem C para tradução e compilação neste projeto porque é simples, familiar e atende bem às necessidades do enunciado. A sintaxe da linguagem é simples e fácil de entender, tornando a definição da sintaxe e o compilador fáceis de usar. Além disso, C já é uma linguagem que suporta muitos conceitos como declaração de variáveis, operações aritméticas e aritméticas, expressões de controle de fluxo como if, while, for, que são importantes para os negócios. Assim. Estar familiarizado com C nos ajudará a entender melhor como funciona o processo de conversão de código em código de máquina, o que é importante em nosso trabalho com máquinas virtuais. Resumindo, C fornece uma boa estrutura para processamento simples e eficiente de análise e compilação de linguagem.

Capítulo 3

Análisador Léxico

Para a resolução do problema construímos o analisador léxico seguindos os seguintes passos

```
1   import ply.lex as lex
2
3   literals = ['(' , ')' , '{' , '}', ';' , ',' , '[' , ']', '&']
4
5   tokens = ('ID','CHAR', 'INT', 'FLOAT', 'TIPO',
6           'STRING', 'ADD','SUB', 'MUL', 'DIV',
7           'EQ', 'NEQ', 'LT', 'LE', 'GT', 'GE',
8           'WRITE','READ', 'INCLUDE', 'BIBLIO',
9           'IF', 'ELSE', 'FOR', 'WHILE','RETURN',
10          'COMENT', 'ATRIBUICAO', 'NOT', 'AND', 'OR')
11
12  def t_COMENT(t):
13      r'//[^\n]*|(/\*(.*)\*/)'
14      return t
15
16  def t_BIBLIO(t):
17      r'<[A-z0-9][A-z0-9_-]*\.h>'
18      return t
19
20  def t_ADD(t):
21      r'\+'
22      return t
23
24  def t_SUB(t):
25      r'-'
26      return t
27
28  def t_MUL(t):
29      r'\*'
30      return t
31
32  def t_DIV(t):
```

```
33          r'/'
34          return t
35
36      def t_EQ(t):
37          r'=='
38          return t
39
40      def t_NEQ(t):
41          r'\!='
42          return t
43
44      def t_NOT(t):
45          r'\!(?!=)'
46          return t
47
48      def t_LT(t):
49          r'<'
50              return t
51
52      def t_GT(t):
53          r'>'
54          return t
55
56      def t_LE(t):
57          r'<='
58          return t
59
60      def t_GE(t):
61          r'>='
62          return t
63
64      def t_AND(t):
65          r'&&'
66          return t
67
68      def t_OR(t):
69          r'\|\|'
70          return t
71
72      def t_ATRIBUICAO(t):
73          r'=(?!=)'
74          return t
75
76      def t_TIPO(t):
77          r'char|int|float|void'
78          return t
79
80      def t_INCLUDE(t):
```

```
81          r'\#[ ]?include'
82          return t
83
84      def t_IF(t):
85          r'if'
86          return t
87
88      def t_ELSE(t):
89          r'else'
90          return t
91
92      def t_FOR(t):
93          r'for'
94          return t
95
96      def t_WHILE(t):
97          r'while'
98          return t
99
100     def t_RETURN(t):
101         r'return'
102         return t
103
104     def t_WRITE(t):
105         r'printf'
106         return t
107
108     def t_READ(t):
109         r'scanf'
110         return t
111
112     def t_CHAR(t):
113         r'\"[A-z]\"'
114         return t
115
116     def t_STRING(t):
117         r'\".+\"'
118         return t
119
120     def t_INT(t):
121         r'[0-9]+(?!\.)'
122         return t
123
124     def t_FLOAT(t):
125         r'[0-9]+\.[0-9]+'
126         return t
127
128     def t_ID(t):
```

```
129         r'[A-z][A-z0-9_]*'
130         return t
131
132     t_ignore = ' \n\t'
133
134     def t_error(t):
135         print('Illegal character: ', t.value[0])
136         t.lexer.skip(1)
137
138     lexer = lex.lex()
```

Primeiramente impou-se a biblioteca que nos permite criar o analisador léxico,seguida de dois conceitos distintos:

Literals → São defenidos caracteres específicos ((,), , , ;, ,, [,],&,:) que são diretamente reconhecidos como tokens. Esses símbolos são usados frequentemente na linguagem C para delimitação ou controle.

Tokens → São listados todos os tipos de tokens que o analisador léxico precisa identificar no código C. Cada token representa uma unidade significativa na linguagem, como palavras-chave, operadores, literais e identificadores.

ID → Representa identificadores, que são nomes definidos pelo programador para variáveis, funções, ou outros elementos.

CHAR → Representa caracteres literais delimitados por aspas simples (').

INT → Representa números inteiros.

FLOAT → Representa números de ponto flutuante.

TIPO → Representa as palavras-chave de tipo na linguagem C, como int, char e float.

STRING → Representa cadeias de caracteres delimitadas por aspas duplas (").

ADD → Representa a operação soma.

SUB → Representa a operação subtração.

MUL → Representa a operação multiplicação.

DIV → Representa a operação divisão.

EQ → Representa a comparação igualdade.

NEQ → Representa a comparação diferença.

LT → Representa a comparação menor que.

LE → Representa a comparação menor ou igual.

GT → Representa a comparação maior que.

GE → Representa a comparação maior igual a.

WRITE → Representa a função padrão (em C) write.

READ → Representa a função padrão (em C) read.

INCLUDE → Indica uma inclusão.

BIBLIO → Representa bibliotecas padrão(em C).

IF → Representa a estrutura de controle de fluxo if.
ELSE → Representa a estrutura de controle de fluxo else.
FOR → Representa a estrutura de controle de fluxo for.
WHILE → Representa a estrutura de controle de fluxo while.
RETURN → Representa a estrutura de controle de fluxo return.
COMENT → Representa comentários no código
ATRIBUICAO → Representa o operador de atribuição(=).
NOT → Representa negação lógica.
AND → Representa e lógico.
OR → Representa ou lógico.

Com o analisador léxico pronto temos agora uma base sólida para construir um BNF que represente a linguagem C juntamente com o analisador sintático.

Capítulo 4

Análisador Sintático

No desenvolvimento do BNF, nosso objetivo foi criar uma representação simplificada e acessível da linguagem C, que fosse clara e fácil de compreender. Com isso em mente, apresentamos o BNF passo a passo.

```
Programa ::= Imports Funcs
```

O programa é composto por duas partes principais:

Imports → Representa os cabeçalhos de biblioteca importados.

Funcs → Contém as funções definidas no programa.

```
Imports ::= Import
          | Import Imports
```

Representa a lista de importações podendo haver uma única importação (Import) ou várias (Import Imports).

```
Import ::= INCLUDE BIBLIO
```

INCLUDE → Palavra-chave para inclusão de bibliotecas.

BIBLIO → Nome da biblioteca a ser incluída

```
Funcs ::= Func
        Func Funcs
```

Representa uma função ou uma sequência de funções

```
Func ::= Tipos ID '(' Params ')' '{' Lines Output '}'
```

Cada função tem:

Tipos → O tipo de retorno da função (ex.: int, void).

ID → O nome da função.

Params → Os parâmetros da função, listados entre parênteses.

Lines → O corpo da função, composto por várias linhas de código.

Output → A declaração de retorno da função (return).

```
Tipos ::= Tipo
```

```
        | VOID
Tipo ::= INT
       | CHAR
       | FLOAT
```

Tipos → Representa os tipos básicos de variáveis podendo ser int.char ou float.

```
Params ::= Param
         |Param ',' Params
         |
Param ::= Tipo ID
```

Params → Lista de parâmetros, podendo ser única (Param) ou múltipla (Param ',' Params).
Param → Um parâmetro tem um tipo e um identificador (Tipo ID).

```
Lines ::= Line ';'
        | Line ';' Lines
```

Lines → Representa o corpo de uma função, que é uma sequência de linhas de código.

```
Line ::= Declaration
       | Atribuition
       | DecAt
       | Math
       | Call
       | Select
       | Cicle
       | Read
       | Write
       | COMENT
```

Uma linha pode ser uma declaração, atribuição, chamada de função, estrutura de controle, entre outros.

```
Declaration ::= Tipo VarList
VarList ::= ID Index
          | ID Index ',' VarList
```

Para declarar variaveis é necessário o tipo e uma lista de variáveis no qual pode só conter uma única variável (ID Index) ou várias separadas por vírgula.

```
Index ::= '[' INT ']'
        |
```

Representa a indexação de arrays. Pode ser vazia, se a variável não for um array.

```
Atribuition ::= EqList '=' ID Index
              | EqList '=' Value
```

Atribui um valor ou outra variável a um identificador.

```
EqList ::= ID Index
         | ID Index '=' EqList
```

Representa uma lista de atribuições em cadeia.

```
DecAt ::= Tipo ID '=' ID Index
        | Tipo ID '=' Value
```

Essa produção permite declarar uma variável de um determinado tipo e, ao mesmo tempo, atribuir um valor inicial a ela.

```
Values ::= Value
         | Value ',' Values
Value ::= INT
        | FLOAT
        | CHAR
        | Array
Array ::= '{' Values '}'
```

Essa produção permite a definição de múltiplos valores, separados por vírgulas do tipo int,float,char e até mesmo um coleção de valores representada por um array.

```
Math ::= ID '=' Expression
```

Realiza operações matemáticas e armazena o resultado em uma variável.

```
Expression ::= Expression ADD Expression
             | Expression SUB Expression
             | Expression MUL Expression
             | Expression DIV Expression
             | '(' Expression ')'
             | ID
             | INT
             | FLOAT
```

Representa expressões matemáticas, incluindo operações básicas, valores numéricos e variáveis.

```
Call ::= ID '(' Inputs ')'
```

Representa a chamada de uma função na linguagem.

```
Inputs ::= Input
        | Input ',' Inputs
```

Define os parâmetros ou valores passados para uma função durante sua chamada.

```
Input ::= ID Index
        | Value
```

Define os possíveis tipos de argumentos que podem ser passados para uma função.

```
Select ::= IF '(' Conditions ')' '{' Lines '}'
         | IF '(' Conditions ')' '{' Lines '}' ELSE '{' Lines '}'
```

Estrutura condicional if-else, que executa blocos de código baseados em condições.

```
Cicle ::= WHILE '(' Conditions ')' '{' Lines '}'
        | FOR '(' ID ATRIBUICAO INT ';' Conditions ';' Maths ')' '{' Lines
            '}'
```

Representa os ciclos while e for, usados para repetir blocos de código.

```
Conditions ::= Condition
             | Condition AND '(' Conditions ')'
             | Condition OR '(' Conditions ')'
```

Representa condições lógicas compostas por operadores como AND, OR, e NOT.

```
Condition ::= Expression EQ Expression
            | Expression NEQ Expression
            | Expression LT Expression
            | Expression LE Expression
            | Expression GT Expression
            | Expression GE Expression
            | NOT Condition
```

Condições baseadas em operadores de comparação.

```
Maths ::= Math
        | Math ',' Maths
```

Lista de operações matemáticas

```
Read ::= READ '(' STRING ',' Addresses ')'
Addresses ::= Address
            | Address ',' Addresses
Address ::= '&' ID
```

Este trecho reprensenta a função read e os juntamente com os endereços que esta função requere

```
Write ::= WRITE '(' STRING ')'
        | WRITE '(' STRING ',' Values ')'
```

Representa a função write.

```
Output ::= RETURN Ret
Ret ::= ID Index
      | Value
      |
```

Corresponde ao return em C.

Com isto tudo temos o seguinte BNF.

```
Programa ::= Imports Funcs
Imports ::= Import
          | Import Imports
```

```
Import ::= INCLUDE BIBLIO
Funcs ::= Func
        | Func Funcs
Func ::= Tipos ID '(' Params ')' '{' Lines Output '}'
Tipos ::= Tipo
        | VOID
Tipo ::= INTT
       | CHART
       | FLAOTT
Params ::= Param
         | Param ',' Params
         |
Param ::= Tipo ID
        |
Lines ::= Line ';'
        | Line ';' Lines
Line ::= Declaration
       | Atribuition
       | DecAt
       | Math
       | Call
       | Select
       | Cicle
       | Read
       | Write
       | COMENT
Declaration ::= Tipo VarList
VarList ::= ID Index
          | ID Index ',' VarList
Index ::= '[' INT ']'
        |
Atribuition ::= EqList '=' ID Index
              | EqList '=' Value
EqList ::= ID Index
         | ID Index '=' EqList
DecAt ::= Tipo ID '=' ID Index
        | Tipo ID '=' Value
Values ::= Value
         | Value ',' Values
Value ::= INT
        | FLOAT
        | CHAR
        | Array
Array ::= '{' Values '}'
Math ::= ID '=' Expression
Expression ::= Expression ADD Expression
             | Expression SUB Expression
             | Expression MUL Expression
```

```
52              | Expression DIV Expression
53              | '(' Expression ')'
54              | ID
55              | INT
56              | FLOAT
57  Call ::= ID '(' Inputs ')'
58  Inputs ::= Input
59           | Input ',' Inputs
60  Input ::= ID Index
61          | Value
62  Select ::= IF '(' Conditions ')' '{' Lines '}'
63           | IF '(' Conditions ')' '{' Lines '}' ELSE '{' Lines '}'
64  Cicle ::= WHILE '(' Conditions ')' '{' Lines '}'
65          | FOR '(' ID ATRIBUICAO INT ';' Conditions ';' Maths ')' '{' Lines
                '}'
66  Conditions ::= Condition
67               | Condition AND '(' Conditions ')'
68               | Condition OR '(' Conditions ')'
69  Condition ::= Expression EQ Expression
70              | Expression NEQ Expression
71              | Expression LT Expression
72              | Expression LE Expression
73              | Expression GT Expression
74              | Expression GE Expression
75              | NOT Condition
76  Maths ::= Math
77          | Math ',' Maths
78  Read ::= READ '(' STRING ',' Addresses ')'
79  Addresses ::= Address
80              | Address ',' Addresses
81  Address ::= '&' ID
82  Write ::= WRITE '(' STRING ')'
83          | WRITE '(' STRING ',' Values ')'
84  Output ::= RETURN Ret
85  Ret ::= ID Index
86        | Value
87        |
```

Capítulo 5

Testes e Resultados

5.1 Testes e Resultados

Resultados dos testes realizados com exemplos.

Capítulo 6

Conclusão

Resumo do trabalho realizado, análise de resultados e propostas futuras.