



**UNIVERSIDADE ESTADUAL DO CEARÁ
CENTRO DE CIÊNCIAS E TECNOLOGIA**

**KAIO RODRIGUES DA CUNHA
MATEUS MATOS DA COSTA**

**RELATÓRIO
COMPILADOR MINIJAVA**

**FORTALEZA, CEARÁ
2022**

1.INTRODUÇÃO

Esse trabalho apresenta o projeto de implementação de um compilador escrito na linguagem Java, feito a partir do código fonte inicial do compilador Minijava fornecido pela Universidade de Washington.

MiniJava é um subconjunto de Java e o significado de um programa MiniJava é dada pelo seu significado como um programa Java. Comparado ao pleno Java, um programa MiniJava consiste em uma única classe que contém um método estático main seguido de zero ou mais classes. Não existem outros métodos estáticos ou variáveis. Classes podem estender outras classes, e método de substituição está incluído, mas a sobrecarga de método não é.

Todas as classes em um programa MiniJava estão incluídos em um único arquivo de origem. Os únicos tipos disponíveis são int, tipos boolean, int [] e de referência (classes). "System.out.println (...)"; é uma indicação e só pode imprimir números inteiros - não é uma chamada de método normal, e não se refere a um método estático de uma classe. Todos os métodos são-retornando valor e deve terminar com uma única instrução de retorno. As únicas outras declarações disponíveis são, se, ao mesmo tempo, e atribuição. Há outras simplificações para manter o tamanho do projeto razoável.

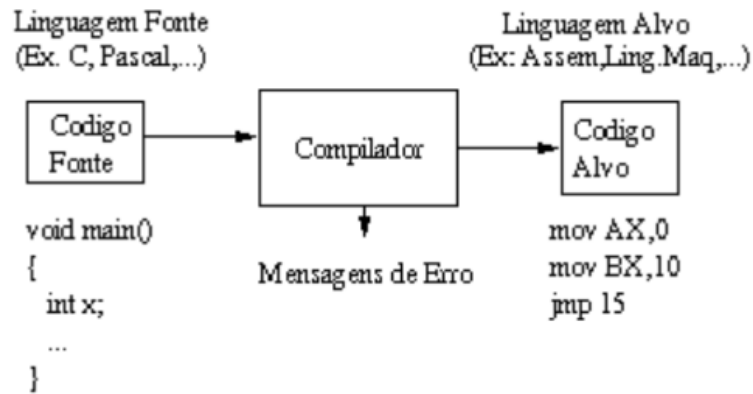
2. OBJETIVO

Aplicar o conhecimento relacionado à construção e funcionamento de um compilador, a fim de proporcionar uma melhor fixação do mesmo, por meio do entendimento de cada etapa do processo executado por um compilador.

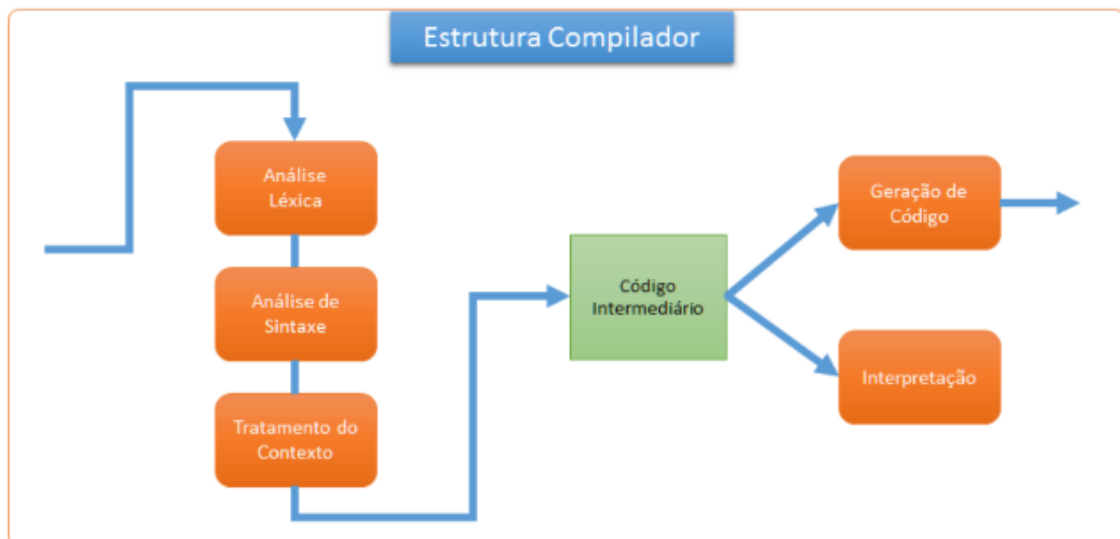
3. REFERENCIAL TEÓRICO

3.1. COMPILADOR

Um compilador é um programa (ou um conjunto de programas) que traduz um código fonte para uma linguagem de mais baixo nível (a linguagem alvo, que tem uma forma binária conhecida como código objeto). Normalmente, o código fonte é escrito em uma linguagem de programação de alto nível, com grande capacidade de abstração, e o código objeto é escrito em uma linguagem de baixo nível, como uma sequência de instruções a ser executada pelo processador. O compilador é um dos dois tipos mais gerais de tradutores, juntamente com o interpretador.



A construção de um compilador é dividida em partes, cada uma com uma função específica, sendo elas: O scanner (Análise Léxica), o parser (Análise Sintática), a Análise Semântica e o gerador de código de máquina.



3.2. SCANNER

No scanner é feita a análise léxica onde ocorre a quebra do código fonte em pequenos pedaços chamados tokens (Symbols). Cada token é uma única unidade atômica da linguagem, por exemplo, uma instância de palavra-chave, identificador ou símbolo. A sintaxe dos tokens é tipicamente uma linguagem regular, de modo que um autômato de estados finitos construídos a partir de uma expressão regular pode ser utilizado para reconhecê-lo. Este pode não ser um passo separado - ele pode ser combinado com a etapa de análise em análise sintática sem varredura, caso em que a análise é feita no nível de caractere, e não no nível de token.

3.3. PARSER

No parser ocorre a análise sintática, que envolve a análise da sequência de sinal para identificar a estrutura sintática do programa. Esta fase geralmente constrói uma árvore de análise (Parse), que substitui a sequência linear de

tokens com uma estrutura de árvore construída de acordo com as regras de uma gramática formal que definem a sintaxe da linguagem. A árvore de parse é frequentemente analisada, aumentada, e transformada por fases posteriores do compilador.

3.4. ANÁLISE SEMÂNTICA

A análise semântica é a fase em que o compilador adiciona informações semânticas para a árvore de análise e constrói a tabela de símbolos. Esta fase realiza verificações de semântica, como a verificação de tipo (verificação de erros de tipo), ou objeto ligação (associando referências de variáveis e função com suas definições), ou cessão definitiva (exigindo que todas as variáveis locais para ser inicializado antes de usar), rejeitando programas incorretos ou emissão avisos. análise semântica requer geralmente uma árvore de análise completa, o que significa que esta fase segue logicamente a fase de análise, e logicamente precede a fase de geração de código, embora muitas vezes é possível dobrar várias fases em uma passagem sobre o código em uma implementação do compilador.

3.5. GERADOR DE CÓDIGO

Por fim temos o gerador de código onde é produzido o código na linguagem de máquina. Para isso ocorrem três processos, são eles:

- A) **Análise:** Esta é a recolha de informações sobre o programa a partir da representação intermediária derivado da entrada; análise de fluxo de dados é usada para construir uso-definir correntes, juntamente com a análise da dependência, a análise de alias, análise de ponteiro, escapar análise, etc. análise precisa é a base para qualquer otimização do compilador. O gráfico de chamadas e gráfico fluxo de controle são geralmente também construídos durante a fase de análise.
- B) **Otimização:** a representação de linguagem intermediária é transformada em formas funcionalmente equivalentes, mas mais rápida e/ou menor. As otimizações mais comuns são a eliminação de código morto, constante de propagação, a transformação de loop, alocação de registro e até mesmo a paralelização automática.
- C) **Geração de código:** a linguagem intermediária transformada é traduzida para a linguagem de saída, geralmente a linguagem de máquina nativa do sistema. Trata-se de decisões de recursos e de armazenamento, como decidir quais variáveis para caber em registos e memória e a seleção e programação de instruções de máquina apropriadas juntamente com os seus modos de endereçamento associados. Dados de depuração também podem precisar ser gerados para facilitar a depuração.

4. IMPLEMENTAÇÃO

A implementação do projeto foi feita a partir do projeto inicial do compilador Minijava, onde foram implementados o scanner, o parser, a AST, construção da tabela e a verificação de tipo:

4.1. SCANNER

No desenvolvimento do scanner, foi realizada a edição do arquivo “minijava.flex”, onde foram colocadas palavras reservadas, delimitadores, os identificadores e os operadores da Linguagem. Adicionamos algumas palavras reservadas como por exemplo: static, public e class; alguns operadores: <, -, *; delimitadores: {, }, [,]. Após essa edição, executamos o jflex, onde usamos o arquivo de saída “scanner.java”. A função symbolToString() é utilizada para imprimir os tokens.

4.2. PARSER

O Parser reconhece o que as sequências de caracteres são atribuídas (declarações, identificadores, entre outros), ou seja, como elas são combinadas para que possam fazer sentido na linguagem, o que caracteriza a análise sintática. A gramática foi modificada para LALR(1), para isso, no Parser a ordem de precedência utilizada para as operações aritméticas foram, respectivamente: multiplicação(MULT), soma(PLUS) e subtração(MINUS) e para operações booleanas foram, respectivamente: negação(NOT), && (ANDAND) e menor que(<)(LT).

No desenvolvimento do PARSER, editamos o arquivo minijava.cup com as palavras reservadas de acordo com o que foi adotado no carregamento do minijava.flex. Desta maneira, identificamos os não terminais da linguagem, da mesma forma que a gramática do minijava implementa.

A fim de evitar erros de shift-reduce, implementamos a ordem de precedência entre as operações, onde finalizamos esta parte com a implementação das produções. Finalizando esta parte, efetuamos a geração dos arquivos parser.java, parser_actions.java e sym.java.

4.3. AST

Na geração do AST, o visitor percorre cada uma das classes, entra nos statements e métodos do código, onde se adiciona um nó para cada e guarda as respectivas variáveis e parâmetros.

4.4. CONSTRUÇÃO DA TABELA

Na construção da Tabela de símbolos, o visitor faz a mesma coisa do AST, porém não há necessidade da visita mais profunda ao programa, pois se limitarmos à visita do visitor até os métodos, será o suficiente para o mapeamento completo do código.

Nesta parte construímos as seguintes classes: Classes, Methods e Variables, para serem utilizadas durante o processo de checagem de tipo.

4.5. VERIFICAÇÃO DE TIPO

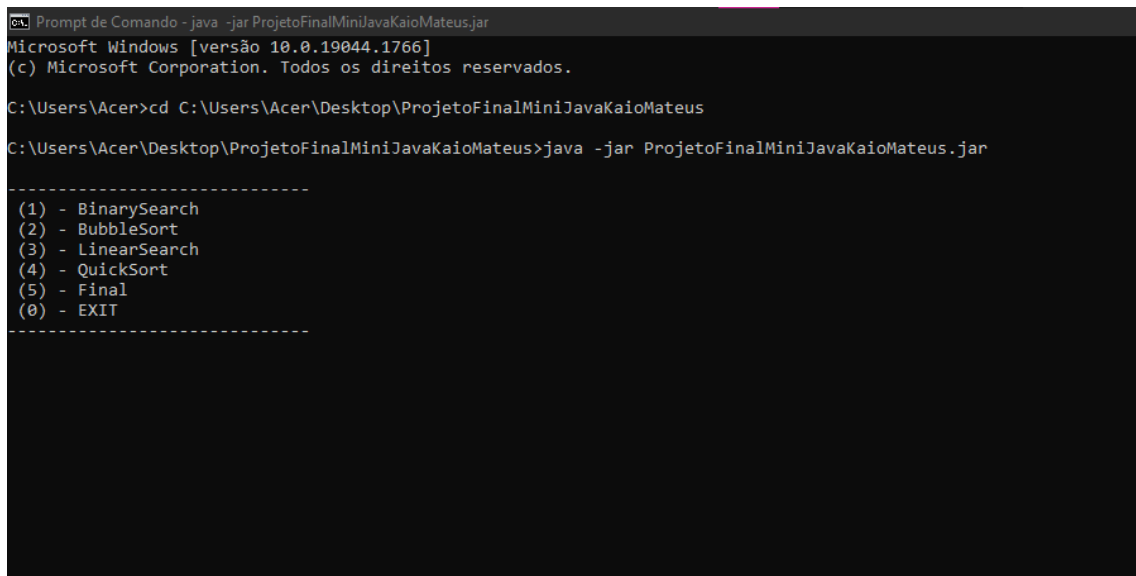
Na parte de verificação de tipo fizemos a construção de um novo visitor, onde ficam armazenadas as informações acerca da classe e método analisados atualmente, e para saber se realmente houve declaração onde estão sendo requisitados os métodos. Outra implementação realizada foi a verificação da corretude e compatibilidade dos tipos retornados com as operações.

5. EXECUÇÃO

Para que se inicie a execução do compilador, dentro da pasta do projeto, executar o comando:

Java – jar ProjetoFinalMiniJavaKaioMateus.jar

Em seguida, selecionar uma das opções de arquivos na linguagem MiniJava que será analisado (Exemplo: BinarySearch.java).



```
C:\> Prompt de Comando - java -jar ProjetoFinalMiniJavaKaioMateus.jar
Microsoft Windows [versão 10.0.19044.1766]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\Acer>cd C:\Users\Acer\Desktop\ProjetoFinalMiniJavaKaioMateus
C:\Users\Acer\Desktop\ProjetoFinalMiniJavaKaioMateus>java -jar ProjetoFinalMiniJavaKaioMateus.jar

-----
(1) - BinarySearch
(2) - BubbleSort
(3) - LinearSearch
(4) - QuickSort
(5) - Final
(0) - EXIT
-----
```

Após a escolha do arquivo, o compilador irá fazer todo o processo de tradução para a linguagem de máquina, passo a passo:

Prompt de Comando - java -jar ProjetoFinalMiniJavaKaiomateus.jar

```
class BS {
    int [] number;
    int size;
    public int Start (int sz) {
        int aux01;
        int aux02;
        aux01 = this.Init(sz);
        aux02 = this.Print();
        if (this.Search(8))
            System.out.println(1);
        else System.out.println(0);
        if (this.Search(19))
            System.out.println(1);
        else System.out.println(0);
        if (this.Search(20))
            System.out.println(1);
        else System.out.println(0);
        if (this.Search(21))
            System.out.println(1);
        else System.out.println(0);
        if (this.Search(37))
            System.out.println(1);
        else System.out.println(0);
        if (this.Search(38))
            System.out.println(1);
        else System.out.println(0);
        if (this.Search(39))
            System.out.println(1);
        else System.out.println(0);
        if (this.Search(50))
            System.out.println(1);
        else System.out.println(0);
        return 999;
    }
    public boolean Search (int num) {
        boolean bs01;
        int right;
        int left;
        boolean var_cont;
        int medium;
        int aux01;
        int nt;
        aux01 = 0;

```

Prompt de Comando - java -jar ProjetoFinalMiniJavaKaiomateus.jar

```
        right = (medium - 1);
        else left = (medium + 1);
        if (this.Compare(aux01, num))
            var_cont = false;
        else var_cont = true;
        if ((right < left))
            var_cont = false;
        else nt = 0;
    }
    if (this.Compare(aux01, num))
        bs01 = true;
    else bs01 = false;
    return bs01;
}
    public int Div (int num) {
        int count01;
        int count02;
        int aux03;
        count01 = 0;
        count02 = 0;
        aux03 = (num - 1);
        while ((count02 < aux03)) {
            count01 = (count01 + 1);
            count02 = (count02 + 2);
        }
        return count01;
    }
    public boolean Compare (int num1, int num2) {
        boolean retval;
        int aux02;
        retval = false;
        aux02 = (num2 + 1);
        if ((num1 < num2))
            retval = false;
        else if (!(num1 < aux02))
            retval = false;
        else retval = true;
        return retval;
    }
    public int Print () {
        int j;
        j = 1;
        while ((j < size)) {
            System.out.println(number[j]);

```

```

Classes:
BS
4
  Variaveis Globais:
    number -> AbstractSyntaxTree.IntArrayType@46ee7fe8 -> 4
    size -> AbstractSyntaxTree.IntegerType@7506e922 -> 8
  Metodos:
    Div -> AbstractSyntaxTree.IntegerType@4ee285c6
      Parametros:
        num -> AbstractSyntaxTree.IntegerType@621be5d1 -> 8
      Locais:
        count02 -> AbstractSyntaxTree.IntegerType@573fd745 -> 8
        count01 -> AbstractSyntaxTree.IntegerType@15327b79 -> 4
        aux03 -> AbstractSyntaxTree.IntegerType@4f2410ac -> 12
    Print -> AbstractSyntaxTree.IntegerType@722c41f4
      Parametros:
      Locais:
    Init -> AbstractSyntaxTree.IntegerType@5d6f64b1
      Parametros:
        sz -> AbstractSyntaxTree.IntegerType@32a1bec0 -> 8
      Locais:
        aux01 -> AbstractSyntaxTree.IntegerType@22927a81 -> 16
        j -> AbstractSyntaxTree.IntegerType@78e03bb5 -> 4
        k -> AbstractSyntaxTree.IntegerType@5e8c92f4 -> 8
        aux02 -> AbstractSyntaxTree.IntegerType@61e4705b -> 12
    Start -> AbstractSyntaxTree.IntegerType@50134894
      Parametros:
        sz -> AbstractSyntaxTree.IntegerType@2957fcb0 -> 8
      Locais:
        aux01 -> AbstractSyntaxTree.IntegerType@1376c05c -> 4
        aux02 -> AbstractSyntaxTree.IntegerType@51521cc1 -> 8
    Search -> AbstractSyntaxTree.BooleanType@1b4fb997
      Parametros:
        num -> AbstractSyntaxTree.IntegerType@deb6432 -> 8
      Locais:
        var_cont -> AbstractSyntaxTree.BooleanType@28ba21f3 -> 16
        left -> AbstractSyntaxTree.IntegerType@694f9431 -> 12
        nt -> AbstractSyntaxTree.IntegerType@f2a0b8e -> 28
        aux01 -> AbstractSyntaxTree.IntegerType@593634ad -> 24
        right -> AbstractSyntaxTree.IntegerType@20fa23c1 -> 8
        medium -> AbstractSyntaxTree.IntegerType@3581c5f3 -> 20
        bs01 -> AbstractSyntaxTree.BooleanType@6aa8ceb6 -> 4

```

CL Prompt de Comando - java -jar ProjetoFinalMiniJavaKaioMateus.jar

```

Search -> AbstractSyntaxTree.BooleanType@1b4fb997
  Parametros:
    num -> AbstractSyntaxTree.IntegerType@deb6432 -> 8
  Locais:
    var_cont -> AbstractSyntaxTree.BooleanType@28ba21f3 -> 16
    left -> AbstractSyntaxTree.IntegerType@694f9431 -> 12
    nt -> AbstractSyntaxTree.IntegerType@f2a0b8e -> 28
    aux01 -> AbstractSyntaxTree.IntegerType@593634ad -> 24
    right -> AbstractSyntaxTree.IntegerType@20fa23c1 -> 8
    medium -> AbstractSyntaxTree.IntegerType@3581c5f3 -> 20
    bs01 -> AbstractSyntaxTree.BooleanType@6aa8ceb6 -> 4
Compare -> AbstractSyntaxTree.BooleanType@2530c12
  Parametros:
    num1 -> AbstractSyntaxTree.IntegerType@73c6c3b2 -> 8
    num2 -> AbstractSyntaxTree.IntegerType@48533e64 -> 12
  Locais:
    aux02 -> AbstractSyntaxTree.IntegerType@64a294a6 -> 8
    retval -> AbstractSyntaxTree.BooleanType@7e0b37bc -> 4

```

```

.text
.global asm_main

```

```

asm_main:
push 28
call mallocEquiv
add esp, 4
lea edx, BS$$
mov [eax], edx
mov ecx, eax
push ecx
call BS$BS
pop eax
mov [ebp+4], eax
mov eax, 20
push eax
mov ecx, [ebp+4]
mov eax, [ecx]
call dword ptr [eax+8]
push eax
call put
add esp, 4
mov ecx, [esp]
ret
.data

```


cmd Prompt de Comando - java -jar ProjetoFinalMiniJavaKaioMateus.jar

```
.data
BS$$      dd      0
          dd      BS$BS
          dd      BS$Div
          dd      BS$Print
          dd      BS$Init
          dd      BS$Start
          dd      BS$Search
          dd      BS$Compare
          .data
glob_BS:  dd      number
          dd      size
          .code
push ebp
mov  ebp,esp
sub  esp,8
mov  ecx,BS$
mov  eax,ebp+8
push eax
mov  ecx,BS$
mov  eax,[ecx]
call dword ptr [eax+48]
mov  eax,[ebp-4]
mov  ecx,BS$
mov  ecx,BS$
mov  eax,[ecx]
call dword ptr [eax+40]
mov  eax,[ebp-8]
mov  ecx,BS$
mov  eax,8
push eax
mov  ecx,BS$
mov  eax,[ecx]
call dword ptr [eax+16]
mov  eax,1
push eax
call put
add  esp,4
mov  ecx,[esp]
jmp done1
else1: mov  eax,0
push eax
call put
```

cmd Prompt de Comando - java -jar ProjetoFinalMiniJavaKaioMateus.jar

```
ebp-4
mul  eax,edx
mov  eax,[ebp-16]
mov  eax,ebp-8
mov  edx,3
sub  eax,edx
mov  eax,[ebp-12]
mov  eax,ebp-16
mov  edx,ebp-12
add  eax,edx
push eax
ebp-4
push eax
pop  edx
pop  eax
mov  ecx,[ecx+4]
shl  edx,2
add  edx,ecx
mov  [edx],eax
mov  ecx,[esp]
mov  eax,ebp-4
mov  edx,1
add  eax,edx
mov  eax,[ebp-4]
mov  eax,ebp-8
mov  edx,1
sub  eax,edx
mov  eax,[ebp-8]
jmp  test18
done18:
pop  ebp
ret
```

```
-----
(1) - BinarySearch
(2) - BubbleSort
(3) - LinearSearch
(4) - QuickSort
(5) - Final
(0) - EXIT
-----
```

6. CONCLUSÃO

O trabalho foi parte fundamental para o aprofundamento no conhecimento sobre o funcionamento de um compilador por meio da prática. Os passos nesse trabalho representam cada etapa de um processo de compilação e os seus resultados.