Linguagens e Compiladores

Kipple

Rogério Yuuki Motisuki - 8587052
Guilherme Mamprin - 8587584

10 de dezembro de 2017

Sumário

Reconhecedor determinístico

> *“Construa um reconhecedor determinístico, baseado no autômato de pilha estruturado, que aceite como entrada válida um arquivo contendo descrições e comandos em Kipple. Não é necessário colocar toda a descrição da linguagem (incluindo bibliotecas), basta utilizar os elementos descritos explicitamente na sintaxe.”*
>
> Ricardo Rocha

Wirth

Uma maneira natural de descrever Kipple em notação de Wirth é:

```
KIPPLE = CODE {CODE}.

CODE = EXPRESSION | LOOP.

EXPRESSION = PUSH_R | PUSH_L | ADD_SUB | CLEAR .

PUSH_R = OPERAND ">" (STACK_IDENTIFIER | PUSH_L | ADD_SUB | CLEAR).

PUSH_L = STACK_IDENTIFIER "<" (OPERAND | PUSH_R | ADD_SUB | CLEAR).

ADD_SUB = STACK_IDENTIFIER ("+"|"-") (OPERAND | PUSH_R | ADD_SUB |
CLEAR) .

CLEAR = STACK_IDENTIFIER "?" .

OPERAND = STACK_IDENTIFIER | NUMBER .

LOOP = "(" (STACK_IDENTIFIER|PUSH_L|ADD_SUB|CLEAR) CODE {CODE} ")".

NUMBER = DIGIT {DIGIT}.

DIGIT = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

STACK_IDENTIFIER = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
"j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" |
"v" | "w" | "x" | "y" | "z" | "@".
```

Porém, essa definição traria problemas na hora de construir um autômato determinístico de reconhecimento. Essa mesma definição pode ser refatorada para:

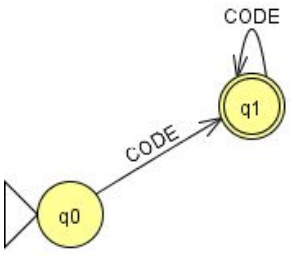
```
KIPPLE = CODE {CODE}.

CODE = EXPRESSION | LOOP.

EXPRESSION = STACK_IDENTIFIER OPERATIONS | NUMBER ">" STACK_IDENTIFIER
[OPERATIONS].

OPERATIONS = PUSH_L | PUSH_R | ADD_SUB | CLEAR.

PUSH_R = ">" STACK_IDENTIFIER [OPERATIONS] .

PUSH_L = "<" (NUMBER | STACK_IDENTIFIER PUSH_L | STACK_IDENTIFIER PUSH_R
| NUMBER PUSH_R | STACK_IDENTIFIER ADD_SUB | STACK_IDENTIFIER [CLEAR]).

ADD_SUB = ("+"|"-") (NUMBER | STACK_IDENTIFIER PUSH_L | STACK_IDENTIFIER
PUSH_R | NUMBER PUSH_R | STACK_IDENTIFIER ADD_SUB | STACK_IDENTIFIER
[CLEAR]) .

CLEAR = "?" .

LOOP = "(" STACK_IDENTIFIER [OPERATIONS] {CODE} ")".

NUMBER = DIGIT {DIGIT}.

DIGIT = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

STACK_IDENTIFIER = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
"j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" |
"v" | "w" | "x" | "y" | "z" | "@".
```

Esse nova descrição remove o prefixo comum “STACK_IDENTIFIER”, facilitando o reconhecedor sintático.

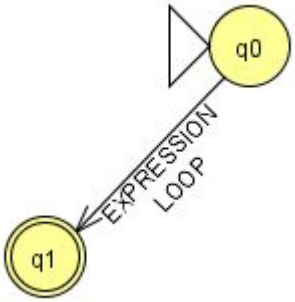
Autômatos

Utilizamos a ferramenta do Mc Barau (http://mc-barau.herokuapp.com/) para a gerar os autômatos correspondentes, e geramos as seguintes representações via JFLAP:

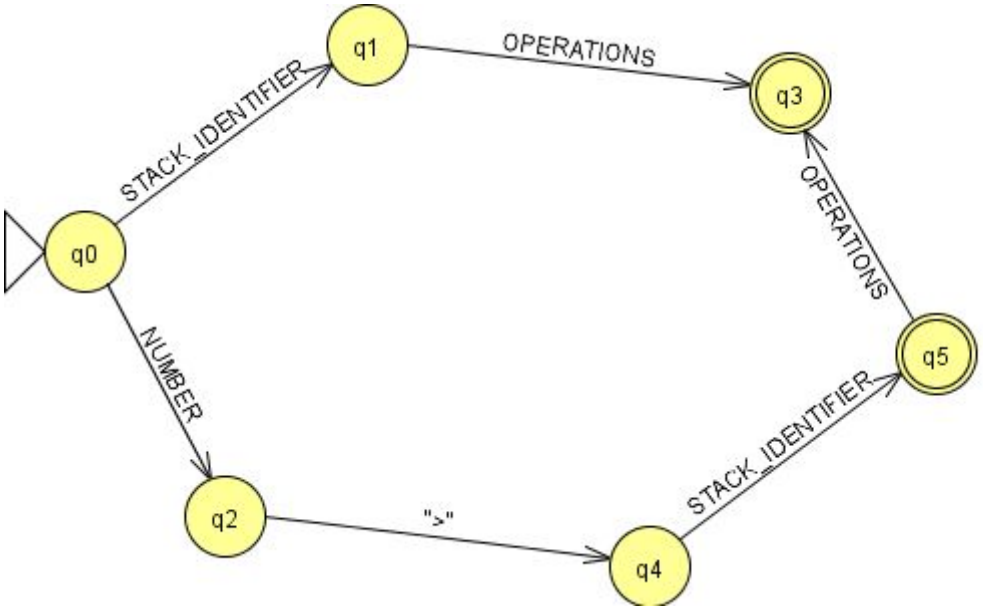
KIPPLE



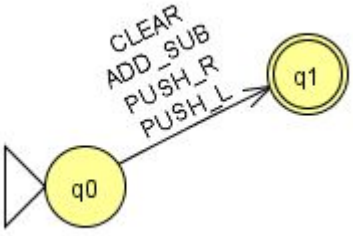
CODE



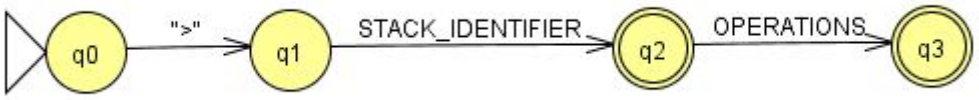
EXPRESSION



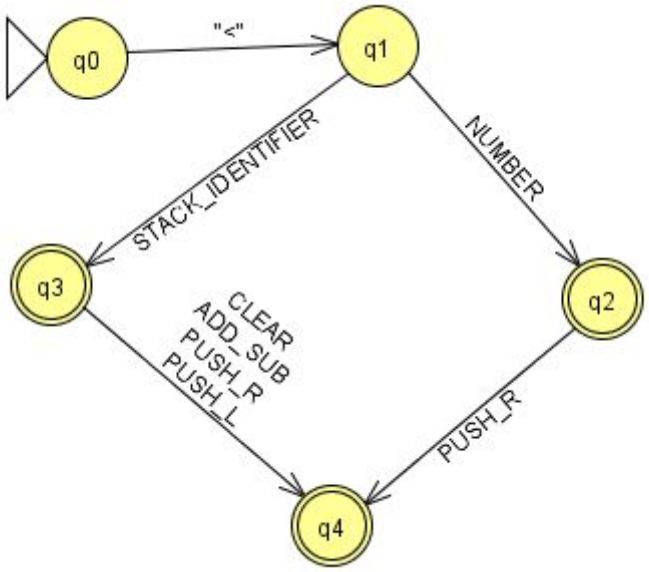
OPERATIONS



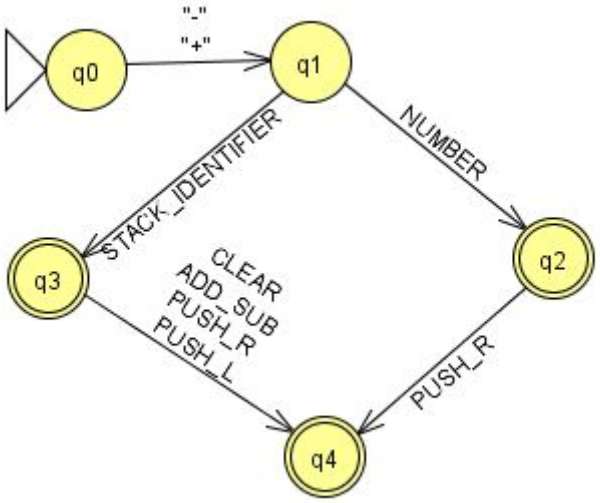
PUSH_R



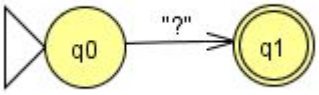
PUSH_L



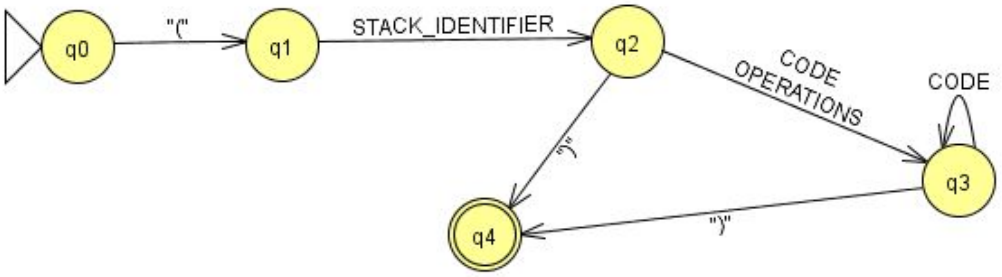
ADD_SUB



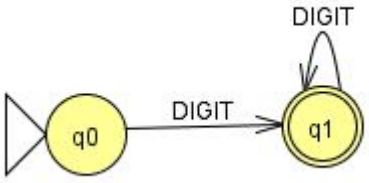
CLEAR



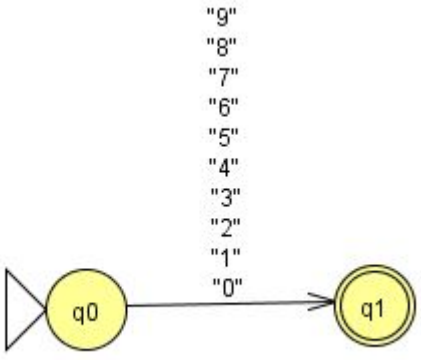
LOOP



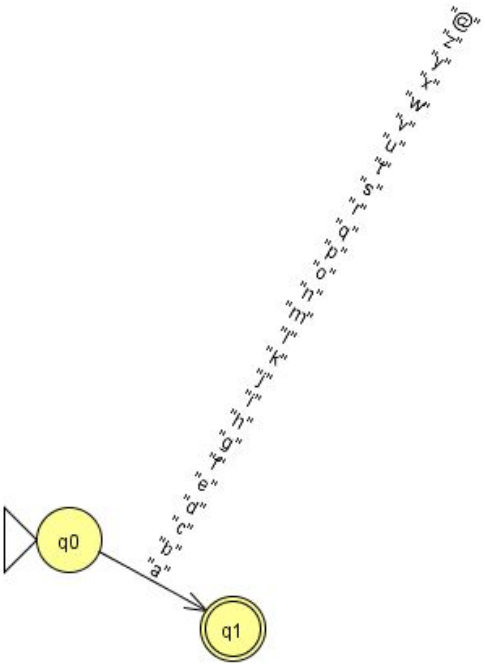
NUMBER



DIGIT



STACK_IDENTIFIER



Compilador para JVM

> *"Construa o sistema de programação para a linguagem Kipple que terá um compilador para a linguagem Java, um ambiente de execução que contará com bibliotecas da linguagem. Assim, mapeia a biblioteca de execução de Kipple para a JVM. Não é necessário produzir E/S na linguagem."*
>
> Ricardo Rocha

O compilador de Kipple foi implementado produzindo código para JVM. No fim, é produzido um arquivo *.jar* pronto para execução.

Analisador léxico

O mesmo analisador léxico do compilador BIRL++ foi utilizado, mas com outras definições de *tokens*:

Token	**Expressão Regular**
STACK_IDENTIFIER	`[a-zA-Z@]`
NUMBER	`[0-9]+`
SYMBOL	`\(|\)|\+|-|\?|>|<`
Comentário	`\/\/.*|\/\*(.|[\n\r])*\*\/`
Espaçadores	`[ \t\n\r]*`

Analisador sintático

O mesmo analisador sintático do compilador BIRL++ foi utilizado, mas com a definição de sintaxe do Kipple, dado que há um gerador de autômatos automático a partir de Wirth.

Ambiente de execução

O ambiente de execução é a JVM. A linguagem Kipple tem 27 pilhas, sendo elas 1 pilha de entrada, 1 pilha de saída e 1 pilha de conversão ASCII.

Para reproduzir esses recursos da linguagem na MVN, foi criado um ambiente de execução, denominado KRE (Kipple Runtime Engine).

O KRE foi implementado em Java, contendo 4 classes:

- **BaseStack**: Classe base para os dois tipos de pilha.
- **AsciiStack**: Classe que sobrescreve o método de *push*, convertendo números em valores ASCII
- **KippleStack**: Classe para pilha padrão do Kipple.
- **KippleRuntime**: Classe estática que fornece métodos de manipulação das pilhas para o código Kipple.

Todas as funções disponíveis do ambiente de execução são fornecidas pela **KippleRuntime** para serem utilizadas pelo compilador.

BaseStack

```
package com.kipple.runtime;

import java.util.Stack;

abstract class BaseStack {
   private Stack<Integer> stack = new Stack<>();

   public int pop() {
       return stack.empty() ? 0 : stack.pop();
   }

   public int peek() {
       return stack.empty() ? 0 : stack.peek();
   }

   public void push(int val) {
       stack.push(val);
   }

   public int isEmpty() {
       return stack.empty() ? 1 : 0;
   }

   public void add(int val) {
       stack.push(peek() + val);
   }

   public void add(BaseStack s) {
       stack.push(peek() + s.pop());
   }

   public void sub(int val) {
       stack.push(peek() - val);
   }

   public void sub(BaseStack s) {
       stack.push(peek() - s.pop());
   }

   public void clear() {
       if (pop() == 0) stack.clear();
   }
}
```

KippleRuntime

```
package com.kipple.runtime;

import java.util.HashMap;

public final class KippleRuntime {
   private static HashMap<Integer, BaseStack> stacks = new HashMap<>(28,
1);

   private static BaseStack getStack(String idStack) {
       return stacks.get((int)idStack.charAt(0));
   }

   public static void init(String[] args) {
       stacks.put((int) "@".charAt(0), new AsciiStack());

       for (char idStack = 'a'; idStack <= 'z'; idStack++) {
           stacks.put((int) idStack, new KippleStack());
       }

       BaseStack input = getStack("i");
       for (int i = 0; i < args.length; i++) {
           String arg = args[i];
           for (int j = 0; j < arg.length(); j++) {
               input.push((int) arg.charAt(j));
           }
       }
   }

   public static void push(int idStack, int val) {
       stacks.get(idStack).push(val);
   }

   public static int pop(int idStack) {
       return stacks.get(idStack).pop();
   }

   public static void clear(int idStack) {
       stacks.get(idStack).clear();
   }

   public static int isEmpty(int idStack) {
       return stacks.get(idStack).isEmpty();
   }

   public static void add(int idStack, int val) {
       stacks.get(idStack).add(val);
```

```
    }

    public static void addStacks(int idStackLeft, int idStackRight) {
        stacks.get(idStackLeft).add(stacks.get(idStackRight));
    }

    public static void sub(int idStack, int val) {
        stacks.get(idStack).sub(val);
    }

    public static void subStacks(int idStackLeft, int idStackRight) {
        stacks.get(idStackLeft).sub(stacks.get(idStackRight));
    }

    public static void printOutput() {
        BaseStack stack = getStack("o");
        while (stack.isEmpty() != 1) {
            System.out.print((char)stack.pop());
        }
    }
}
```

AsciiStack

```
package com.kipple.runtime;

public final class AsciiStack extends BaseStack {
    @Override
    public void push(int val) {
        String s = Integer.toString(val);
        for (int i = 0; i < s.length(); i++) {
            super.push((int) s.charAt(i));
        }
    }
}
```

KippleStack

```
package com.kipple.runtime;

public final class KippleStack extends BaseStack {

}
```

Rotinas semânticas

As rotinas semânticas implementadas foram as descritas na prova. O compilador utiliza as seguintes variáveis para armazenar estado entre uma rotina e outra:

```
int L_counter = 0;
Token *operandoEsquerdo = NULL, *operador = NULL, *operandoDireito = NULL;
int Loop_counter = 0;
UT_array *loopLabels = NULL;
```

- **inicialização**

Roda antes do programa, gerando como código toda a inicialização e a preface contendo a classe *Main*.

```
void inicializacao() {
   addCode(".version 52 0 ");
   addCode(".class public super Main ");
   addCode(".super java/lang/Object ");
   addCode("");
   addCode(".method public <init> : ()V ");
   addCode("    .code stack 1 locals 1 ");
   addCode("L0:     aload_0 ");
   addCode("L1:     invokespecial Method java/lang/Object <init> ()V ");
   addCode("L4:     return ");
   addCode("L5:     ");
   addCode("        .linenumbertable ");
   addCode("            L0 3 ");
   addCode("        .end linenumbertable ");
   addCode("        .localvariabletable ");
   addCode("            0 is this LMain; from L0 to L5 ");
   addCode("        .end localvariabletable ");
   addCode("    .end code ");
   addCode(".end method ");
   addCode("");
   addCode(".method public static main : ([Ljava/lang/String;)V ");
   addCode("    .code stack 2 locals 1 ");
   addMainCode("aload_0");
   addMainCode("invokestatic Method com/kipple/runtime/KippleRuntime init
([Ljava/lang/String;)V");
}
```

- **finalização**

Roda no fim do programa, emitindo como código a chamada da função *printOutput* e o fim do arquivo da classe *Main*.

```
void finalizacao() {
   addMainCode("invokestatic Method com/kipple/runtime/KippleRuntime printOutput
()V");
   addMainCode("return");
   addCode("    .end code");
   addCode(".end method");
   addCode(".sourcefile 'Main.java'");
   addCode(".end class");
}
```

- **começa_loop**

Roda no começo do loop, gerando uma label para pular no fim do loop. O código sob essa label verifica se a pilha está vazia, utilizando a função *isEmpty*.

```
void comeca_loop(SemanticoTrigger *trigger) {
   if (loopLabels == NULL) utarray_new(loopLabels, &ut_int_icd);

   int thisLoop = Loop_counter;
   utarray_push_back(loopLabels, &thisLoop);

   char *buf = (char*)smart_malloc((50) * sizeof(char));
   sprintf(buf, "LOOP%d_START:    bipush %d", thisLoop,
getStackId(trigger->token));
   addCode(buf);
   free(buf);

   addMainCode("invokestatic Method com/kipple/runtime/KippleRuntime isEmpty
(I)I");
   buf = (char*)smart_malloc((50) * sizeof(char));
   sprintf(buf, "ifgt LOOP%d_END", thisLoop);
   addMainCode(buf);
   free(buf);

   Loop_counter++;
}
```

- **termina_loop**

Roda no fim do loop, gerando um goto pra label do início do loop.

```
void termina_loop(SemanticoTrigger *trigger) {
   int thisLoop = (int) *utarray_back(loopLabels);
   utarray_pop_back(loopLabels);

   char *buf = (char*)smart_malloc((50) * sizeof(char));
   sprintf(buf, "goto LOOP%d_START", thisLoop);
   addMainCode(buf);
   free(buf);

   buf = (char*)smart_malloc((25) * sizeof(char));
   sprintf(buf, "LOOP%d_END:    nop", thisLoop);
   addCode(buf);
   free(buf);
}
```

- **operador_binario**

Roda sempre que encontra um operador com dois operandos: +, -, > e <.

Armazena o ***operador*** em uma variável no compilador, para que uma rotina semântica futura emita o código apropriado.

```
void operador_binario(SemanticoTrigger *trigger) {
   operador = (Token*) smart_malloc(sizeof(Token));
   operador->type = trigger->token->type;
   operador->value = trigger->token->value;
}
```

- **operando**

Roda sempre que encontra um operando. Sempre que essa rotina executa, o ***operando direito*** é copiado pro ***operando esquerdo***, e o novo operando encontrado é armazenado no ***operando direito***.

Caso tenha um ***operador*** pendente, executa uma nova rotina semântica, dependendo do operador.

```
void operando(SemanticoTrigger *trigger) {
   free(operandoEsquerdo);
   operandoEsquerdo = operandoDireito;
   operandoDireito = (Token*) smart_malloc(sizeof(Token));
   operandoDireito->type = trigger->token->type;
   operandoDireito->value = trigger->token->value;

   if (operador != NULL && strcmp(operador->value, "<") == 0) {
       push_l(trigger);
   }
   else if (operador != NULL && strcmp(operador->value, ">") == 0) {
       push_r(trigger);
   }
   else if (operador != NULL && strcmp(operador->value, "+") == 0) {
       add_sub(trigger);
   }
   else if (operador != NULL && strcmp(operador->value, "-") == 0) {
       add_sub(trigger);
   }
}
```

- **push_l**

Emite um código utilizando *push* para adicionar um valor na pilha do operando esquerdo. O valor é obtido a partir do operando direito. Caso seja uma outra pilha, a função *pop* é utilizada.

A variável **operador** é limpada no final.

```
void push_l(SemanticoTrigger *trigger) {
   addMainCode( formatStackInstruction("bipush", operandoEsquerdo->value) );

   switch (operandoDireito->type) {
       case NUMBER:
           addMainCode( formatNumberInstruction("bipush",
operandoDireito->value) );
           addMainCode("invokestatic Method com/kipple/runtime/KippleRuntime
push (II)V");
           break;
       case STACK_IDENTIFIER:
```

```
            addMainCode( formatStackInstruction("bipush", operandoDireito->value)
);
            addMainCode("invokestatic Method com/kipple/runtime/KippleRuntime pop
(I)I");
            addMainCode("invokestatic Method com/kipple/runtime/KippleRuntime
push (II)V");
            break;
    }
    free(operador); operador = NULL;
}
```

- **push_r**

Emite um código utilizando *push* para adicionar um valor na pilha do operando direito. O valor é obtido a partir do operando esquerdo. Caso seja uma outra pilha, a função *pop* é utilizada.

A variável **operador** é limpada no final.

```
void push_r(SemanticoTrigger *trigger) {
    addMainCode( formatStackInstruction("bipush", operandoDireito->value) );

    switch (operandoEsquerdo->type) {
        case NUMBER:
            addMainCode( formatNumberInstruction("bipush",
operandoEsquerdo->value) );
            addMainCode("invokestatic Method com/kipple/runtime/KippleRuntime
push (II)V");
            break;
        case STACK_IDENTIFIER:
            addMainCode( formatStackInstruction("bipush",
operandoEsquerdo->value) );
            addMainCode("invokestatic Method com/kipple/runtime/KippleRuntime pop
(I)I");
            addMainCode("invokestatic Method com/kipple/runtime/KippleRuntime
push (II)V");
            break;
    }
    free(operador); operador = NULL;
}
```

- **add_sub**

Caso o ***operador*** seja de adição, emite um código utilizando *add* ou *addStack*, dependendo se o ***operando direito*** é um número ou outra pilha.

Caso o ***operador*** seja de subtração, emite um código utilizando *sub* ou *subStack*, dependendo se o ***operando direito*** é um número ou outra pilha.

A variável **operador** é limpada no final.

```
void add_sub(SemanticoTrigger *trigger) {
   addMainCode( formatStackInstruction("bipush", operandoEsquerdo->value) );

   if (strcmp(operador->value, "+") == 0) {
       switch (operandoDireito->type) {
           case NUMBER:
               addMainCode( formatNumberInstruction("bipush",
operandoDireito->value) );
               addMainCode("invokestatic Method com/kipple/runtime/KippleRuntime
add (II)V");
               break;
           case STACK_IDENTIFIER:
               addMainCode( formatStackInstruction("bipush",
operandoDireito->value) );
               addMainCode("invokestatic Method com/kipple/runtime/KippleRuntime
addStacks (II)V");
               break;
       }
   }
   else {
       switch (operandoDireito->type) {
           case NUMBER:
               addMainCode( formatNumberInstruction("bipush",
operandoDireito->value) );
               addMainCode("invokestatic Method com/kipple/runtime/KippleRuntime
sub (II)V");
               break;
           case STACK_IDENTIFIER:
               addMainCode( formatStackInstruction("bipush",
operandoDireito->value) );
               addMainCode("invokestatic Method com/kipple/runtime/KippleRuntime
subStacks (II)V");
```

```
                break;
        }
    }
    free(operador); operador = NULL;
}
```

- **clear**

Emite um código utilizando o método *clear* do ambiente de execução. No caso, utiliza-se o **operando direito**, pois foi o último a ser lido.

```
void clear(SemanticoTrigger *trigger) {
    addMainCode( formatStackInstruction("bipush", operandoDireito->value) );
    addMainCode("invokestatic Method com/kipple/runtime/KippleRuntime clear
(I)V");
}
```

Montagem e Linkagem

O código gerado corresponde ao mnemônico do *bytecode* correspondente, e é uma classe *Main* que precisa ser executada junto com as classes do ambiente de execução. Para isso, após a compilação, o resultado precisa passar pelos seguintes processos:

1. **Montagem para *bytecode* binário;**

   Utilizamos o *assembler* Krakatau.

2. **Arquivamento em um *JAR*.**

Para auxiliar esses passos, criamos um script:

```
#!/bin/sh

cp $1 jvm/kre/out/production/kre/Main.j
python jvm/krakatau/assemble.py -out jvm/kre/out/production/kre
jvm/kre/out/production/kre/Main.j
jar cfve $2 Main -C jvm/kre/out/production/kre .
rm jvm/kre/out/production/kre/Main.*
```

Execução

Para executar, o arquivo *JAR* gerado, utiliza-se o comando:

```
java -noverify -jar SAIDA.jar [args...]
```

Para testar o funcionamento, utilizamos os seguintes programas de entrada:

Fibonacci

```
24>n 0>t 1>a
(n-1
  a+0
  t<a>b+a
  c<b>a<c
  n?
)
(t>@
  (@>o)
  32>o
)
```

KippleCat

```
(i>o)
```