**UNIVERSIDADE FEDERAL DE MINAS GERAIS**
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Relatório Final: POC2

# Interpretador de Tiger para uma representação tipo assembler

Área: Compiladores

| | |
|---|---|
| Projeto: | Interpretador de TIGER para uma representação tipo assembler |
| Orientadora: | Profª Mariza Andrade da Silva Bigonha |
| Aluno: | Gustavo Garcia Guerra |

Cronograma: Início: Março de 2001

| | |
|---|---|
| Término: | Junho de 2001 |
| Data: | 29 de junho de 2001 |

**Resumo:**

Este documento apresenta a linguagem tipo assembler utilizada na consutrução do interpretador. Além das etapas realizadas durante o projeto para se obter o interpretador. Por fim este documento irá apresentar os testes realizados com o interpretador.
A linguagem TIGER é uma pequena linguagem com funções aninhadas, registros com apontadores implícitos, arranjos, variáveis inteiras e *strings*, além de algumas construções de controle estruturadas.
O projeto proposto consiste no desenvolvimento de uma ferramenta para uso no curso de Compiladores, trata-se de um interpretador de TIGER para uma representação tipo assembler. Desta forma futuros projetos poderão ser realizados para obter interpretadores mais específicos, como por exemplo para a o byte code da MVJ ou para a arquitetura x86.

# Sumário

# 1 - Introdução

O projeto proposto, consiste no desenvolvimento de uma ferramenta para uso no curso de Compiladores, trata-se de um interpretador de TIGER para uma representação tipo assembler
O interpretador será implementado em Java. O seu desenvolvimento foi dividido em três etapas. A primeira consiste do estudo detalhado do Byte Code das instruções da MVJ, a segunda consiste do estudo do código intermediário para a implementação e a terceira consiste da construção do interpretador com base nos estudos realizados.
A linguagem TIGER é uma pequena linguagem com funções aninhadas, registros com apontadores implícitos, arranjos, variáveis inteiras e *strings*, além de algumas construções de controle estruturadas.

Para facilitar a leitura este texto está organizado da seguinte forma:
A Seção 2 apresenta a linguagem TIGER para a qual o interpretador será desenvolvido.

A Seção 3 apresenta uma breve descrição da Máquina Virtual JAVA.
A Seção 4 descreve as o subconjunto de instruções utilizados pela representação tipo assembler gerada pelo interpretador.
A Seção 5 apresenta as etapas da construção do interpretador, além de uma analogia entre as instruções da representação tipo assembler, e as instruções da MVJ e da arquitetura x86. Desta forma, esta seção pode ser bastante útil como sugestão para futuros projeto que podem ser implementados a partir da representação tipo assembler gerada.

A Seção 6 apresenta a forma de usar o interpretador.

A Seção 7 apresenta os testes obtidos após a execução do interpretador.
O Apêndice A, contém o código fonte das instruções utilizadas pela repesentação tipo assembler.

# 2 - A Linguagem TIGER

A linguagem TIGER é uma pequena linguagem com funções aninhadas, registros com apontadores implícitos, arranjos, variáveis inteiras e *strings*, além de algumas construções de controle estruturadas. TIGER é composta de duas seções, uma seção de declarações e uma seção de variáveis e expressões. Ela possui também uma biblioteca de funções.

## 2.1 – Declarações

Uma sequência de declarações de tipos, variáveis ou funções. Nenhuma pontuação separa ou termina declarações individuais.

decs := dec decs | ε
decs := tydec | vardec | fundec

### 2.1.1 – Tipos

Existem dois tipos pré-declarados: *int* e *string.* Tipos adicionais devem ser definidos ou redefinidos via declarações de tipos.
Registros: são definidos por meio da listagem de seus campos enclausurados entre chaves, com cada campo descrito pot *Nome-Campo : type-id*, sendo *type-id* um identificador definido por uma declaração de tipos.
Arranjos: é definido como **array of** *type-id.* O tamanho do arranjo não é especificado como parte do tipo, devendo esta tarefa ser realizada em tempo de execução.
Tipos mutuamente recursivos: uma coleção de tipos pode ser recursiva ou mutuamente recursiva. Tipos mutuamente recursivos são declarados por meio de uma sequência consecutiva de declarações de tipos, sem intervir declarações de valor ou de funções.
Reuso de nomes de campo: tipos de registros diferentes podem usar o mesmo nome para seus campos.

2.1.2 – Variáveis

Uma declaração de variável é dada por:

**var** id := exp
**var** id := type-id := exp

No primeiro caso, o tipo da variável é determinado via o tipo da expressão. No segundo, o tipo, além de ser dado, deve ser semelhante ao tipo da expressão.

2.1.3 – Funções

As funções são declaradas como:

**function** id(tyfields) = exp
**function** id(tyfields) : type-id = exp

O primeiro caso representa uma declaração de procedimento, sendo que procedimentos não retornam valores. O segundo caso representa uma declaração de função, sendo o tipo retornado especificado pelo não terminal *type-id.* O não terminal *exp* representa o corpo do procedimento ou função, e *tyfields* especifica os nomes e tipos dos parâmetros. Todos parâmetros são passados por valor.

**2.2 – Variáveis e Expressões**

2.2.1 – Variáveis

L-Value: um l-value é um local cujo valor pode ser lido ou atribuído, podendo ser representado por variáveis locais, parâmetros de procedimentos, campos de registros ou elementos de arranjos.

2.2.2 – Expressões

As expressões em TIGER podem ser:

- L-Value: quando usado como expressão, o l-value é avaliado para o conteúdo do local correspondente.
- Sequenciamento: uma sequência de duas ou mais expressões, envoltas por parênteses e separadas por ponto-e-vírgula, avaliam todas as expressões em ordem. O resultado, se existir, é o resultado da última expressão.
- Nenhum valor: um abre parênteses, seguido por um fecha parênteses, bem como a expressão *let* com nada entre *in* e *end*, são exemplos de expressões que não produzem nenhum valor.
- Literal inteiro: uma sequência de dígitos decimais é uma constante inteira que denota o valor inteiro correspondente.
- Literal string: uma constante *string* é uma sequência, entre aspas, de zero ou mais caracteres.
- Chamada de função: definida como a aplicação da função *id()* ou *id(exp{, exp})*. Se *id* representar um procedimento, uma função que não retorna resultado, então o corpo de função não deve produzir nenhum valor, bem como a aplicação da função.
- Aritmética: expressões da forma: *exp op exp,* nas quais *op* representa: +, -, *, /. Requerem argumentos inteiros e produzem resultados inteiros.
- Comparação: expressões da forma: *exp op exp*, nas quais *op* representa: =, <>, >, <, <=, >=. Estes operadores testam pela igualdade ou não de seus operandos, produzindo o valor 1 para verdadeiro e 0 para falso. Todos estes operadores podem ser aplicados a operandos inteiros. Além disso, os operadores =, <> também podem ser aplicados a registros ou arranjos do mesmo tipo.
- Comparação de strings: os operadores de comparação também se aplicam a strings, testando se o conteúdo de duas delas é ou não igual.
- Operadores booleanos: são expressões da forma: *exp op exp* com *op* sendo & ou |. Qualquer valor inteiro diferente de zero é considerado verdadeiro, apena o zero é considerado falso.
- Precedência de operadores: O menos unário(negação) possui a maior precedência. Em seguida vêm os operadors *, / , seguidos por *, -, e depois por =, <>, >, <, >=, <=, depois & e finalmente |.
- Associatividade dos operadores: os operadores *, /, +, - são todos associativos à esquerda, enquanto os operadores de comparação não se associam.
- Criação de registros: definido como *type-id id = exp, id = exp.* Cria uma nova instância de registro do tipo *type-def.*
- Criação de arranjos: definido como *type-id[exp1] of exp2.*
- Atribuição: definido como *lvalue := exp.* Avalia-se primeiramente o *lvalue*, depois *exp* e então atribui o resultado da expressão ao conteúdo de *lvalue.*
- if-then-else: a expressão *if exp1 then exp2 else exp3* avalia a expressão inteira *exp1.* Se o resultado for diferente de zero, *exp2* é avaliado, caso contrário, avalia-se *exp3.* As expressões *exp2* e *exp3* devem ser do mesmo tipo, que também é o tipo toda a expressão-if.
- if-then: a expressãp *if exp1 then exp2* avalia a expressão inteira *exp1.* Se o resultado for diferente de zero então *exp2* é avaliado.
- While: a expressão *while exp1 do exp2* avalia a expressão inteira *exp1*. Se o resultado for diferente de zero, então *exp2* é executada, e toda a expressão *while* é reavaliada.
- For: a expressão *for id := exp1 to exp2 do exp3* itera *exp3* para cada valor inteiro de *id* entre *exp1* e *exp2*. A variável *id* é uma nova variável implicitamente declarada pelo *for,* cujo escopo cobre somente *exp3.* Se o limite superior for menor que o inferior, então o corpo não é executado.
- Break: a expressão *break* termina a avaliação da expressão *while* ou *for* mais próximas. Uma expressão *break* que não esteja dentro de um *while* ou de um *for* é considerada ilgeal.
- Let: a espressão *let decs in expseq end* avalia as declarações *decs*, tipos associados, variáveis e procedimentos cujo escopo se estende sobre *expseq.* O não terminal *expseq* representa uma sequência de zero ou mais expressões, separadas por ponto e vírgula . O resultado, se existir, da última *exp* da sequência será então o resultado de toda a expressão *let.*

### 2.3 – Biblioteca de Funções

TIGER possui algumas funções em sua biblioteca; o nome destas funções são autoexplicativos, de forma que serão apenas listadas as funções:

- Function print(s : string)
- Function flush()
- Function getchar() : string
- Function ord(s : string) : int
- Function chr(i : int) : string
- Function size(s : string) : int
- Function substring(s : string, first : int, n : int) : string
- Function concat(s1 : string, s2 : string) : string
- Function exit(i : int)

## 3 – Máquina Virtual JAVA

A Máquina Virtual JAVA(MVJ)[7] é uma máquina abstrata definida por uma especificação formal. Desta forma, ela pode ser implementada de diversas maneiras, como por exemplo: por um interpretador, por um compilador ou até mesmo em *Hardware*. Esta característica da MVJ confere grande flexibilidade à arquitetura na qual ela é aplicada.
A relação existente entre a MVJ e a linguagem JAVA se resume a um formato particular de arquivo denominado arquivo *class*. Este arquivo contém o Byte Code das instruções da MVJ, a tabela de símbolos e algumas outras informações auxiliares. O formato deste arquivo é bastante rígido e algumas restrições estruturais são impostas ao seu código a fim de se obter segurança. Mesmo com estas imposições de segurança, qualquer linguagem com funcionalidades que possam ser expressas em termos de um arquivo *class* válido, pode se utilizar da MVJ. O arquivo *class* possui todas as definições necessárias para a execução do código pela MVJ, e nele pode-se encontrar uma lista ordenada, estruturada, que define variáveis, constantes, objetos com seus atributos e métodos representados na forma de tabelas e sequências de *opcodes* e operandos.
O ciclo de vida de uma classe se dá em duas etapas, a primeira na qual ela é carregada e a segunda na qual ela é executada.
Na primeira etapa, que pode ser dividida em 3 fases, é feito o carregamento e a preparação para a execução da seguinte maneira:

**Fase 1: Carregamento**

Carrega o fluxo binário contido no arquivo *class* para a estrurura interna de dados.

Fase 2: Ligação

Divide-se em 3 partes:

Verificação: assegura que o arquivo obedeça à semântica da linguagem JAVA e não viole integridade da MVJ.

Preparação: aloca memória para as variáveis de classe, iniciando-as com um valor **default**.

Resolução: efetua trocas de referência, mudando-as de simbólica para direta. É opcional, já que pode ser executada no momento de referência ao símbolo.

Fase 3: Inicialização

Liga as variáveis de classe previamente alocadas com os valores determinados pelo programador.
Na segunda etapa, responsável pela execução, é feita a interpretação e execução do Byte Code. Para armazenar todos os dados da execução de uma aplicação é definida uma área de execução de dados da seguinte forma:

- Área de métodos;
- Heap;
- Pilhas de JAVA;
- Registradores PC;
- Pilhas de métodos nativos.

A área de métodos e o *heap* são compartilhados por todas as *threads*, sendo que na área de métodos são armazenadas as informações sobre as classes do programa e no *heap* cada objeto instanciado.
Cada *thread* possui um conjunto de registradores PC, *stack pointer*, etc e uma pilha. Esta pilha armazena *frames*, os quais são compostos por variáveis locais, parâmetros de chamada, valores de retorno e resultados intermediários.
A MVJ não possui registradores de uso geral, sendo a pilha a responsável por armazenar valores intermediários. Esta técnica foi adotada em virtude da proposta de independência de plataforma, de modo que quanto menos registradores fossem exigidos, maior seria o número de arquiteturas capazes de permitir a execução da máquina virtual.
Para realizar a execução, um conjunto de instruções é tomado como base, sendo que a maioria das instruções envolve operações de pilha, uma vez que a MVJ se basea no uso da pilha.

Os tipos de dados presentes na MVJ são:

- Tipos primitivos (*primitive types*)

Numeric Types

Floating-Point Types

Float
Double

Integral types

Byte
Short
Int
Long
Char

Return Address

- Tipos de referência(reference types)

Reference

Class types
Interface Types
Array types

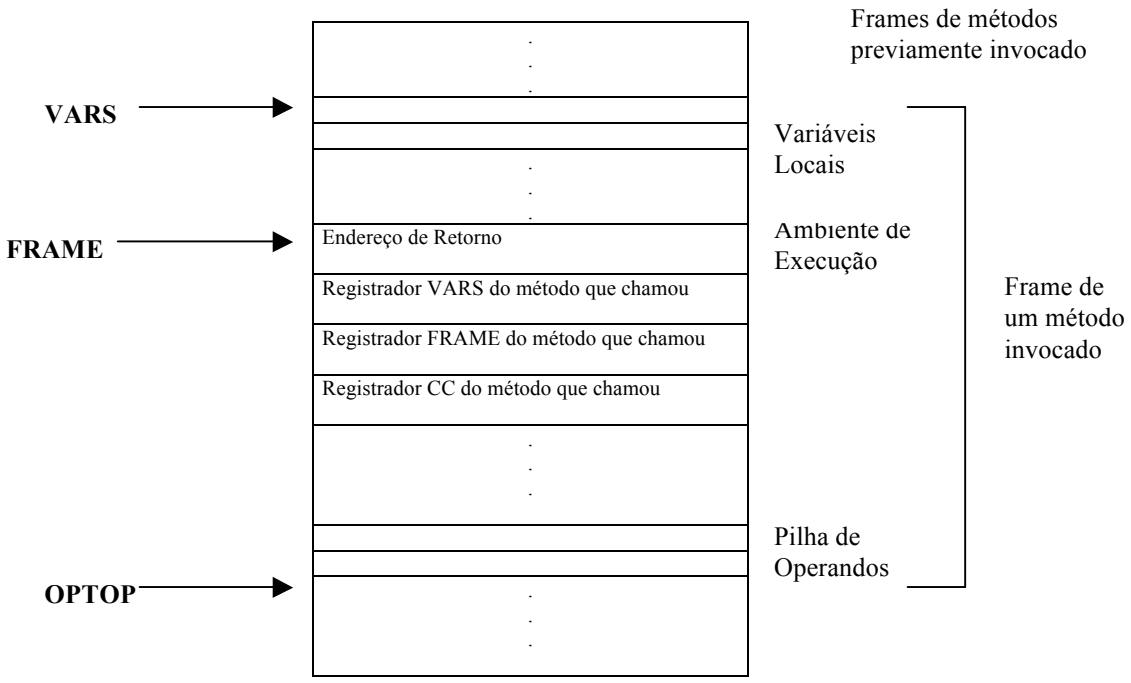
3.1 - Ambiente de Execução:

A MVJ possui 4 registradores e são eles:

PC: *program Counter*, que aponta para a posição a ser executada da próxima instrução.
VARS: todas variáveis locais são endereçadas em relação a este registrador.
OPTOP: aponta para o topo da pilha de operações.
FRAME: aponta para a primeira posição de memória do ambiente de execução.

É interessante utilizar um quinto registrador, o CC. Este registrador não é mencionado na especificação da MVJ, embora internamente algum mecanismo similar deva ser utilizado. Este registrador irá conter a Classe Corrente(CC), ou seja, a classe em que o método em execução está definido.
A Figura 1 mostra a pilha de execução da MVJ. O ambiente de execução de um método invocado contém o endereço de retorno e os valores dos registradores *VARS, FRAME e CC* do método que fez a chamada. Estes valores são necessários para restaurar os registradores do método que fez a chamada, após o retorno do método chamado.

**Figura 1**




4 – Conjunto de Instruções Utilizadas

## 4.1 – Instruções Aritméticas

As instruções aritméticas, podem ser efetuadas de duas maneiras distintas. Quando não possuem nenhum parâmetro (add), elas vão realizar a operação correspondente sobre os dois últimos valores da pilha, no entanto se possuírem um valor constante como parâmetro(add 5), elas vão realizar a operação correspondente entre o último valor da pilha e o valor constante. As operações aritméticas presentes são:

add
sub
mul
div

## 4.2 - Instruções de Comparação

Eq - testa se dois valores são iguais
Ne - testa se dois valores não são iguais
Lt - testa se um valor é menor do que o outro
Le - testa se um valor é menor ou igual ao outro
Gt - teste se um valor é maior do que o outro
Ge - testa se um valor é maior ou igual ao outro

## 4.3 - Instruções sobre Dados

Ld - carrega um valor da pilha de execução de acordo com o valor constante que estiver no topo da pilha
Lds - carrega um valor relativo a pilha de execuºão
St - armazena uma valor na pilha de execução de acordo com o valor constante que estiver no topo da pilha
Sts - armazena um valor relativo a pilha de execução
Pushn - empilha valores na pilha de execução
Popn - desempilha valores da pilha de execução
Xchng - troca os dois últimos valores da pilha de execução

## 4.4 - Instruções de Controle de Fluxo

Call - realiza uma chamada de uma subrotina
Ret - realiza o retorno de uma subrotina
Jmp - realiza um jump incondicional para um determinado Label
Jeq - realiza um jump para um determinado Label se o teste de igualdade entres dois valores for positivo
Jne - realiza um jump para um determinado Label se o teste de igualdade entres dois valores for negativo
Jle - realiza um jump para um determinado Label se um valor for igual ou menor ao outro
Jgt - realiza um jump para um determinado Label se um valor for maior do que o outro
Line Label, que não possui um mnemônico, é a presença de um rótulo no código que será o alvo dos jumps condicionais e incondicionais.

### 4.5 - Outras Instruções

Num - representa um valor constante
Shr - realiza um shift lógico para direita
Shl - realiza um shift lógico para esquerda
Halt - Indica o término da execução do programa
Siza - indica o tamanho de um objeto
.object - indica um objeto, utilizado para inplementar arrays
sp - retorna o valor apontado na pilha

## 5 - Etapas para Construção do Interpretador

### 5.1 – Etapas de Desenvolvimento e Futuros Projetos

As etapas realizadas para implementar um interpretador para a linguagem TIGER foram as seguintes: pegou-se o código fonte gerado pelo compilador disponível e o traduziu para uma representação de árvore abstrata. A partir daí, esta árvore teve que ser percorrida para armazenar as informações referentes aos registradores ativos a fim de se obter o código na representação tipo assembler. Cada nodo da árvore foi traduzido de acordo com as instruções do subconjunto de instruções e o código final foi gerado.
O interpretador apresenta como saída as instruções a serem executadas, e informa em forma de comentário o nodo da árvore abstrata correspondente, com a opção de debug, pode-se observar ainda qual a construção da linguagem TIGER correspondente à instrução em questão.
Por medida de tempo, nenhuma otimização em relação ao código gerado foi executada, o interpretador foi desenvolvido com o único intuito de funcionar corretamente, podendo ser facilmente expandido e adaptado para as mais diversas plataformas. Algumas alterações que podem ser feitas por projetos futuras, são a elaboração de uma política de otimização do código gerado, a construção de um garbage collector, e a adaptação do imterpretado proposto para alguma plataforma mais específica.

Analogia entres as instruções da representação tipo assembler, da MVJ e x86:

| Representação Tipo Assembler | Instruction Set x86 | Instruction Set MVJ |
|---|---|---|
| Add | Add | Iadd |
| Sub | Sub | Isub |
| Mul | Mul | Imul |
| Div | Div | Idiv |
| Num | - | Iconst,etc |
| Eq | Cmp | Ifeq |
| Ne | Cmp | Ifne |
| Lt | Cmp | Iflt |
| Le | Cmp | Ifle |

| | | |
|---|---|---|
| Gt | Cmp | Ifgt |
| Ge | Cmp | Ifge |
| Xchng | Xchg | Swap |
| Halt | - | - |
| Call | Call | Jsr |
| Jmp | Jmp | Goto |
| Jeq | JE/JZ | *if_icmpeq* |
| Jne | JNE/JNZ | *if_icmpne* |
| Jle | JBE/JNA | *if_icmple* |
| Jgt | JA/JNBE | *if_icmpgt* |
| Ld | Lods | Iload,etc |
| Lds | Lods | Aload,etc |
| St | Stos | Istore,etc |
| Sts | Stos | Astore |
| Pushn | Push | Push |
| Popn | Pop | Pop |
| Ret | Ret | Ret |
| Sar | Sar | Ishr |
| Shl | Sal | Ishl |
| Size | - | - |
| Sp | - | - |
| | | |

Desta forma, pode-se observar, que o subconjunto utilizado é bastante genérico e pode ser facilmente alterado para se obter código de acordo com um subconjunto da MVJ, ou de acordo com um subconjunto de instruções da x86. Algumas pequenas mudanças se farão necessárias, a fim de se obter estes interpretadores, por que cada arquitetura possui suas particularidades que deverão ser obedecidas, como o conjunto de registradores que varia de acordo com a arquitetura, e eventuais restrições como o fato da multiplicação e divisão só poderem ser realizadas em determinados registradores da arquitetura x86[8].

## 6 - Utilização do Programa

A utilização do programa deve ser realizada da seguinte forma:

Java Main.java opções

As opções que podem ser utilizadas são

- d ativa a opção de debug, que vai adicionar código para acompanhar qual construção da linguagem Tiger correspondente às instruções geradas
- o <Nome do Arquivo de saída> utilizado para especificar um arquivo de saída
- k para ativar a opção de Ter que pressionar alguma tecla para finalizar o programa

## 7 - Testes

### 7.1 - Teste 1

```
let
    function soma(x1: int, x2: int) : int = x1 + x2
    var resultado : int := 0
in
        resultado := soma(2, 5)
end
```

Listagem da representação tipo assembler gerada sem debug

```
num 0                          // IntExp:3.25
num 0                          // SimpleVar:5.2
num 2                          // IntExp:5.20
num 5                          // IntExp:5.23
sp                             // CallExp:5.15
add -4                         // CallExp:5.15
call _soma_1                   // CallExp:5.15
xchg                           // AssignExp:5.2
st                             // AssignExp:5.2
halt 0
_soma_1: // #Users=1           // FunctionDec:2.11
sp                             // SimpleVar:2.42
add -3                         // SimpleVar:2.42
ld                             // SimpleVar:2.42
sp                             // SimpleVar:2.47
add -3                         // SimpleVar:2.47
ld                             // SimpleVar:2.47
add                            // OpExp:2.42
sts 3                          // FunctionDec:2.11
ret 2                          // FunctionDec:2.11
```

### 7.2 - Teste 2

```
let
    var a := 3
    var b := 4
    var c := 0
in
        c = a + b
end
```

Listagem da representação tipo assembler gerada com debug

```
// LET                                  // LetExp:1.1
// Var Declaration of a                 // VarDec:2.5
num 3                                   // IntExp:2.10
// Var Declaration of b                 // VarDec:3.5
num 4                                   // IntExp:3.10
// Var Declaration of c                 // VarDec:4.5
num 0                                   // IntExp:4.10
// IN                                   // LetExp:1.1
// Var Reference: c Offs=2              // SimpleVar:6.1
num 2                                   // SimpleVar:6.1
ld                                      // SimpleVar:6.1
// Var Reference: a Offs=0              // SimpleVar:6.5
num 0                                   // SimpleVar:6.5
ld                                      // SimpleVar:6.5
// Var Reference: b Offs=1              // SimpleVar:6.9
num 1                                   // SimpleVar:6.9
ld                                      // SimpleVar:6.9
add                                     // OpExp:6.5
eq                                      // OpExp:6.1
sts 2                                   // LetExp:1.1
popn  2                                 // LetExp:1.1
// END                                  // LetExp:1.1
popn  0
halt
```

7.3 - Teste 3

---

```
/* valid for and let */
let
in
        for i:=0 to 100 do ()
end
```

Listagem da representação tipo assembler gerada com debug

```
// LET                                  // LetExp:3.1
// IN                                   // LetExp:3.1
// Var Declaration of i
num 0                                   // IntExp:5.16
num 100                                 // IntExp:5.21
```

```
jmp ForLoopEnterLabel_1          // ForExp:5.9
ForLoopLabel_0:                  // ForExp:5.9
lds 1                            // ForExp:5.9
add 1                            // ForExp:5.9
sts 1                            // ForExp:5.9
ForLoopEnterLabel_1:             // ForExp:5.9
lds 1                            // ForExp:5.9
lds 1                            // ForExp:5.9
jle ForLoopLabel_0:              // ForExp:5.9
popn  1                          // ForExp:5.9
ForLoopExitLabel_2:              // ForExp:5.9
// END                           // LetExp:3.1
halt
```

## 9. Conclusão

Este relatorório apresentou o projeto e desenvolvimento de uma ferramenta para uso no curso de Compiladores, trata-se de um interpretador de TIGER para uma representação tipo assembler.
O interpretador foi implementado em Java, está operacional e pode ser testado como indicou a Seção 6. Ele apresenta como saída as instruções a serem executadas, e informa em forma de comentário o nodo da árvore abstrata correspondente. Por questão de tempo, nenhuma otimização em relação ao código gerado foi executada, o interpretador foi desenvolvido com o único intuito de funcionar corretamente

*Apêndice A*

*Código fonte das instruções*

- Add Sub

```
// AddSub class, implementa as instruções “add” e “sub”
package SamCode;
import Absyn.Absyn;

public class AddSub extends OptionalConstant {
public  static final boolean ADD = false;
public  static final boolean SUB  = true;
boolean Dir;

public AddSub(Absyn FromSym, boolean dir, int Amount) {
    super(FromSym, Amount);
    Dir = dir;
```

```
    if (Dir == SUB) { // Converte SUB em ADD
    Dir = ADD;
    Constant = -Constant;
   }
  }

public AddSub(Absyn FromSym, boolean dir) {
    super(FromSym);
    Dir = dir;
  }

public String getInstruction() {
    if (Dir == ADD)
    return "add"+super.getInstruction();
    else
    return "sub"+super.getInstruction();
  }
}
```

- Comentários

```
// Comment class, Guarda os comentários
//
package SamCode;
import Absyn.Absyn;

public class Comment extends SamCode {
String Remark;

public Comment(Absyn FromSym, String Rem) {
    super(FromSym);
    Remark = "// " + Rem;
  }

boolean Optimize() {
    RemoveThisInstruction();
    return true;
  }

String getInstruction() {
        return Remark;
  }
}
```

- Constante Numérica

```
// Constant class, implementa a constante "num"
//
```

```
package SamCode;
import Absyn.Absyn;

public class Constant extends SamCode {
int Value;

public Constant(Absyn FromSym, int Val) {
    super(FromSym);
    Value = Val;
 }

String getInstruction() {
        return "num " + Value;
 }

boolean Optimize() {
    if (Next != null && Next instanceof OptionalConstant) {
    OptionalConstant T = (OptionalConstant)Next;
    if (!T.ConstantValid) {
        T.ConstantValid = true;
        T.Constant = Value;
        FoldInto(T);
        return true;
    }
   }

        return false;
 }
}
```

- Equality

```
// Class Equality - Instruçoes de teste de igualdade
//
package SamCode;
import Absyn.Absyn;
import Absyn.OpExp;

public class Equality extends OptionalConstant {
public final static int
    EQ=OpExp.EQ, NE=OpExp.NE, LT=OpExp.LT,
    LE=OpExp.LE, GT=OpExp.GT, GE=OpExp.GE;

private int Type;
public Equality(Absyn FromSym, int type, int Const) {
        super(FromSym, Const); Type = type;
 }

public Equality(Absyn FromSym, int type) {
        super(FromSym); Type = type;
```

```
 }

String getInstruction() {
    switch (Type) {
    case EQ:
    return “eq” + super.getInstruction();
    case NE:
    return “ne” + super.getInstruction();
    case LT:
    return “lt” + super.getInstruction();
    case LE:
    return “le” + super.getInstruction();
    case GT:
    return “gt” + super.getInstruction();
    case GE:
    return “ge” + super.getInstruction();
    default:
    throw new Error(“Unrecognized type of Equality operator!”);
   }
  }
}
```

- Exchange

```
// Class Exchange - Implementa a instrução ‘xchg’
//
package SamCode;
import Absyn.Absyn;

public class Exchange extends SamCode {
public Exchange(Absyn FromSym) {
        super(FromSym);
  }

String getInstruction() {
        return “xchg”;
  }
}
```

- Function Call

```
// Class FuncCall - Implementa uma Function call “call”...
//
package SamCode;
import Absyn.Absyn;

public class FuncCall extends LabelUser {
public FuncCall(Absyn FromSym, Label Target) {
        super(FromSym, Target);
```

```
 }
public String getInstruction() {
        return "call "+Target.NiceName;
 }
}
```

- Func call predefined

```
    // Class FuncCallPredefined - Uma function call para uma biblioteca predefinida
    package SamCode;
    import Absyn.Absyn;

    public class FuncCallPredefined extends SamCode {
    String TargetName;

public FuncCallPredefined(Absyn FromSym, String targetName) {
    super(FromSym);
    TargetName = targetName;
 }

public String getInstruction() {
        return "call "+TargetName;
 }
}
```

- Halt

```
// Halt class, implementa a instrução "halt"
//
package SamCode;
import Absyn.Absyn;

public class Halt extends OptionalConstant {
public Halt(Absyn FromSym, int RetCode) {
        super(FromSym, RetCode);
 }

public Halt(Absyn FromSym) {
        super(FromSym);
 }

boolean IsSequentialInstruction() { return false; }
String getInstruction() {
        return "halt" + super.getInstruction();
 }
}
```

- Jump

```
// Class Jump - Implementa um jump
//
package SamCode;
import Absyn.Absyn;
import TreeDisplay.TreeDisplayable;

public class Jump extends LabelUser {
public static final int

JMP = 0,
    EQ = 2, NE = 3,
    LE = 4, GT = 5;
int JumpType;
public Jump(Absyn FromSym, Label Target, int jumpType) {
    super(FromSym, Target);
    JumpType = jumpType;
 }

public Jump(Absyn FromSym, Label Target) {
    super(FromSym, Target);
    JumpType = JMP;
 }

boolean IsSequentialInstruction() { return JumpType != JMP; }

String getInstruction() {
        String Result;
    switch (JumpType) {
    case JMP:Result = "jmp"; break;
    case EQ: Result = "jeq"; break;
    case NE: Result = "jne"; break;
    case LE: Result = "jle"; break;
    case GT: Result = "jgt"; break;
    default:
        throw new Error("Unknown conditional jump type in SamCode.CondJump:" +
        JumpType);
   }

        return Result + " " + Target.NiceName;
 }

public boolean Optimize() {
        if (Next == null) return false;
    if (Next instanceof Label) {
    Label L = (Label)Next;
    if (Target == L) {
```

```
        RemoveThisInstruction();
        return true;
     }
   }

    if (JumpType == JMP)
    return OptimizeUnconditionalJump();
    else
    return OptimizeConditionalJump();
 }

public final boolean OptimizeUnconditionalJump() {
        boolean DidStuff = false;
    while (!(Next instanceof Label)) {
    Next.RemoveThisInstruction();
    DidStuff = true;
   }

        if (DidStuff) return true;
    if (!Target.Prev.IsSequentialInstruction()) {
    Next.Prev = CurStream.Tail;
    CurStream.Tail.Next = Next;
    Target.Prev.Next = null;
    CurStream.Tail = Target.Prev;
    Next = Target;
    Target.Prev = this;
    return true;
   }

        return false;
 }

public final boolean OptimizeConditionalJump() {
    if (Next instanceof Jump) {
    Jump J = (Jump)Next;
    if (J.JumpType == JMP) {
        if (J.Next == Target) {
        J.JumpType = JumpType ^ 1;
        RemoveThisInstruction();
        return true;
          }
     }
   }

        return false;
 }
}
```

- Label

```
// Class Label - Implementa umaa linha de label
//
package SamCode;
import Absyn.Absyn;
import java.util.Vector;

public class Label extends SamCode {
private static int LabelNum = 0;
private Vector LabelUsers = new Vector();
public String NiceName;

public Label(Absyn FromSym, String niceName) {
    super(FromSym);
    NiceName = niceName+LabelNum++;
 }

public Label(Absyn FromSym) {
        this(FromSym, "_");
 }

boolean IsSequentialInstruction() { return true; }
public String getInstruction() {
        return NiceName+":"+" // #Users="+getNumUsers();
 }

public final int getNumUsers() {
        return LabelUsers.size();
 }

public boolean Optimize() {
    if (LabelUsers.size() == 0) {
    RemoveThisInstruction();
    return true;
   }

    if (Next != null && Next instanceof Label) {
    Label L = (Label)Next;
    L.NiceName = NiceName + "_" + L.NiceName;

    int NumUsers = LabelUsers.size();
    while (--NumUsers >= 0) {
        LabelUser LU = (LabelUser)LabelUsers.elementAt(NumUsers);
        LU.Target = L;
        L.addUser(LU);
    }

    FoldInto(L);
    return true;
   }

        return false;
```

```
 }

public void addUser(LabelUser NewUser) {
        LabelUsers.addElement(NewUser);
 }

public void removeUser(LabelUser User) {
        LabelUsers.removeElement(User);
 }
}
```

- Label User

```
// Class LabelUser - Superclasse de todos jumps ou calls para um label
//
package SamCode;
import Absyn.Absyn;
import TreeDisplay.TreeDisplayable;

public abstract class LabelUser extends SamCode {
Label Target;

public LabelUser(Absyn FromSym, Label target) {
    super(FromSym);
    Target = target;
    Target.addUser(this);
 }

public void RemovingThisInstruction() {
    Target.removeUser(this);
    super.RemovingThisInstruction();
 }

public TreeDisplayable[] getDrawTreeLinks() {
        return new TreeDisplayable[] { Target };
 }
}
```

- Load

```
// Load class, implementa as instruções ld e lds
//
package SamCode;
import Absyn.Absyn;

public class Load extends OptionalConstant {
boolean StackRelative;

public Load(Absyn FromSym, boolean SR, int Off) {
```

```
    super(FromSym, Off);
    StackRelative = SR;
  }

public Load(Absyn FromSym, boolean SR) {
    super(FromSym);
    StackRelative = SR;
  }

String getInstruction() {
    if (StackRelative)
    return "lds" + super.getInstruction();
    else
    return "ld " + super.getInstruction();
  }
}
```

- Mul Div

```
// Class MultDiv - Implementa as instruções 'mul' e 'div'
//
package SamCode;
import Absyn.Absyn;

public class MultDiv extends OptionalConstant {
public static final int MUL = 0;
public static final int DIV = 1;
private int Function;

public MultDiv(Absyn FromSym, int Func, int Const) {
    super(FromSym, Const);
    Function = Func;
  }

public MultDiv(Absyn FromSym, int Func) {
    super(FromSym);
    Function = Func;
  }

boolean Optimize() {
        if (!ConstantValid) return false;
    int FirstBitSet = CalcFirstBitSet(Constant);
    if (FirstBitSet != -1) {
    if (FirstBitSet == 0)
        RemoveThisInstruction();
        else {
    if (Function == MUL)
    ReplaceThisInstruction(new Shift(this, Shift.LEFT, FirstBitSet));
    else
    ReplaceThisInstruction(new Shift(this, Shift.RIGHT, FirstBitSet));
```

```
    }
        return true;
   }

        return false;
 }

static final int CalcFirstBitSet(int Value) {
    int BitCount = 0;
    while ((Value & 1) == 0) {
    Value >>= 1;
    BitCount++;
   }
    if ((Value & ~1) != 0) return -1;
    return BitCount;
 }

String getInstruction() {
    if (Function == MUL)
    return "mul" + super.getInstruction();
    else
    return "div" + super.getInstruction();
 }
}
```

- Obj

```
// Class Object - Implementa a pseudo Op '.object'
//
package SamCode;
import Absyn.Absyn;

public class Obj extends SamCode {
String Value;
String Name;
static int StringCount = 0;

public Obj(Absyn FromSym, String value) {
    super(FromSym);
    Value = value;
    Name = "ObjectExp_" + (StringCount++);
 }

public String getInstruction() {
        return ".object " + Name + " " + Value;
 }
}
```

- Object Load

```
// Class ObjectLoad - Implementa um ld de um objeto
//
package SamCode;
import Absyn.Absyn;

public class ObjectLoad extends SamCode {
Obj ObjRef;

public ObjectLoad(Absyn FromSym, Obj Obj) {
    super(FromSym);
    ObjRef = Obj;
 }

String getInstruction() {
        return "ld " + ObjRef.Name;
 }
}
```

- Optional Constant

```
    // OptionalConstant class - usada por todas instruções que podem ter um valor
    //opcional constante
    package SamCode;
    import Absyn.Absyn;

    public abstract class OptionalConstant extends SamCode {
    boolean ConstantValid;
    int Constant;

public OptionalConstant(Absyn FromSym, int constant) {
    super(FromSym);
    Constant = constant; ConstantValid = true;
 }

public OptionalConstant(Absyn FromSym) {
        super(FromSym); ConstantValid = false;
 }

public OptionalConstant(SamCode FromSym, int constant) {
    super(FromSym);
    Constant = constant; ConstantValid = true;
 }

public OptionalConstant(SamCode FromSym) {
        super(FromSym); ConstantValid = false;
 }
```

```
String getInstruction() {
    if (ConstantValid)
    return " "+Constant;
    else
    return "";
  }
}
```

- Push pop

```
// PushPop class, implementa as instruções pushn and popn
//
package SamCode;
import Absyn.Absyn;

public class PushPop extends OptionalConstant {
public  static final boolean PUSH = false;
public  static final boolean POP  = true;
boolean Dir;

public PushPop(Absyn FromSym, boolean dir, int Amt) {
    super(FromSym, Amt);
    Dir = dir;
  }

public PushPop(Absyn FromSym, boolean Dir) {
    super(FromSym);
    this.Dir = Dir;
  }

String getInstruction() {
    if (Dir == PUSH)
    return "pushn" + super.getInstruction();
    else
    return "popn " + super.getInstruction();
  }

boolean Optimize() {
    if (ConstantValid) {
    if (Constant == 0) {
        RemoveThisInstruction();
        return true;
    }

        if (Next != null && Next instanceof PushPop) {
    PushPop N = (PushPop)Next;
    if (N.ConstantValid) {
    boolean Add = Dir == PUSH;
    if (N.Dir != PUSH) Add = !Add;
    if (Add)
                                                N.Constant += Constant;
```

```
else
                                                    N.Constant -= Constant;
    FoldInto(N);
    return true;
          }
    }
   }

        return false;
 }
}
```

- Return

```
// Class Return - Implementa a instrução 'ret'
//
package SamCode;
import Absyn.Absyn;

public class Return extends OptionalConstant {
public Return(Absyn FromSym, int Const) {
        super(FromSym, Const);
 }

public Return(Absyn FromSym) {
        super(FromSym);
 }

boolean IsSequentialInstruction() { return false; }
String getInstruction() {
        return "ret" + super.getInstruction();
 }
}
```

- Sam Code

```
// SamCode Class - Classe base de todas as instruções
//
package SamCode;
import Absyn.Absyn;
import FrontEnd.ErrorMsg;
import TreeDisplay.TreeDisplayableAdapter;

public abstract class SamCode extends TreeDisplayableAdapter {
public SamCode Prev, Next;
public String CreateID = "";
static Stream CurStream;

public SamCode(Absyn FromSym) {
    if (FromSym != null) {
```

```
    CreateID = "// " + FromSym.shortClassName()+":"
        +Absyn.getErrorObj().lineAndChar(FromSym.pos);
  }
 }

public SamCode(SamCode FromSym) {
    if (FromSym != null) {
    CreateID = "// " + FromSym.shortClassName() +
        "—"+FromSym.CreateID;
  }
 }

abstract String getInstruction();
boolean IsSequentialInstruction() { return true; }
void EmitCode(Stream CS) {
    String Padding = "";
    String Code = getInstruction();
    CS.print(Code);

    if (Code.length() < 40)
    Padding = getSpaces(40-Code.length());

        CS.println(Padding + CreateID);
 }

boolean Optimize() {
        return false;
 }

public void RemovingThisInstruction() {}
public void RemoveThisInstruction() {
        RemovingThisInstruction();
    if (Prev != null)
    Prev.Next = Next;
    else
    CurStream.Head = Next;

    if (Next != null)
    Next.Prev = Prev;
    else
    CurStream.Tail = Next;
 }

public void ReplaceThisInstruction(SamCode NewInst) {
    NewInst.Prev = Prev; NewInst.Next = Next;
    if (Prev != null)
    Prev.Next = NewInst;
    else
    CurStream.Head = NewInst;
```

```
    if (Next != null)
    Next.Prev = NewInst;
    else
    CurStream.Tail = NewInst;
  }

public void FoldInto(SamCode Inst) {
    if (!Inst.CreateID.equals(“”))
    Inst.CreateID = CreateID + “, “ + Inst.CreateID;
    else
    Inst.CreateID = CreateID;
    RemoveThisInstruction();
  }

public String getTreeDesc(Object obj) {
    String Name = getInstruction();
    Name += getSpaces(20-Name.length())+” “+shortClassName();
    return Name+” “+getSpaces(32-Name.length())+CreateID;
  }
}
```

- Shift

```
// Class Shift - implementa as instruções de shift
//
package SamCode;
import Absyn.Absyn;

public class Shift extends OptionalConstant {
public static final int LEFT = 0;
public static final int RIGHT = 1;
private int Type;

public Shift(Absyn FromSym, int type, int Const) {
          super(FromSym, Const); Type = type; }
public Shift(Absyn FromSym, int type) {
          super(FromSym); Type = type; }
public Shift(SamCode FromSym, int type, int Const) {
          super(FromSym, Const); Type = type; }
public Shift(SamCode FromSym, int type) {
          super(FromSym); Type = type; }
String getInstruction() {
    if (Type == LEFT)
    return “shl” + super.getInstruction();
    else
    return “sar” + super.getInstruction();
  }
}
```

- Size

```
// Class Size - Implementa a instrução ‘size’
//
package SamCode;
import Absyn.Absyn;

public class Size extends OptionalConstant {
public Size(Absyn FromSym, int Const) {
        super(FromSym, Const);
  }

public Size(Absyn FromSym) {
        super(FromSym);
  }

String getInstruction() {
        return “size” + super.getInstruction();
  }
}
```

- SpValue

```
// SPValue class, implementa a instrução “sp”
//
package SamCode;
import Absyn.Absyn;

public class SPValue extends SamCode {
public SPValue(Absyn FromSym) {
        super(FromSym);
  }

String getInstruction() {
        return “sp”;
  }
}
```

- Store

```
// Store class, implementa as instruções st and sts
//
package SamCode;
import Absyn.Absyn;

public class Store extends OptionalConstant {
boolean StackRelative;
```

```
public Store(Absyn FromSym, boolean SR, int Off) {
    super(FromSym, Off);
    StackRelative = SR;
 }

public Store(Absyn FromSym, boolean SR) {
    super(FromSym);
    StackRelative = SR;
 }

String getInstruction() {
    if (StackRelative)
    return "sts" + super.getInstruction();
    else
    return "st " + super.getInstruction();
 }
}
```

- Stream

```
// Stream class, define o recipiente de uma stream
//
package SamCode;
import java.io.PrintWriter;
import TreeDisplay.*;

public class Stream extends TreeDisplayableAdapter {
PrintWriter OutStream;
SamCode     Head, Tail;

public Stream() {
    Head = Tail = null;
    OutStream = null;
    SamCode.CurStream = this;
 }

public void add(SamCode Inst) {
    if (Head == null) {
    Head = Tail = Inst;
    Inst.Prev = Inst.Next = null;
    } else {
    Tail.Next = Inst;
    Inst.Prev = Tail;
    Inst.Next = null;
    Tail = Inst;
   }
 }

public void print(String S) {
       OutStream.print(S);
```

```
 }

public void println(String S) {
        print(S+"\n");
 }

public void Optimize() {
    SamCode Cur = Head, Prev;
    boolean Changed = true;

    while (Changed) {
    Changed = false;

        while (Cur != null) {
    Prev = Cur.Prev;
    if (Cur.Optimize()) {
    Cur = (Prev != null) ? Prev : Head;
    Changed = true;
    } else {
    Cur = Cur.Next;
          }
    }
   }
 }

public void EmitCode(PrintWriter OS) {
    OutStream = OS;
    SamCode Cur = Head;

    while (Cur != null) {
    Cur.EmitCode(this);
    Cur = Cur.Next;
   }

        OutStream = null;
 }


public TreeDisplayable[] getDrawTreeSubobj() {
    int InstCount = 0;
    SamCode Cur = Head;
    while (Cur != null) {
    InstCount++;
    Cur = Cur.Next;
   }

    TreeDisplayable[] RetList = new TreeDisplayable[InstCount];
    int i = 0;

    for (Cur = Head; Cur != null; Cur = Cur.Next, i++)
    RetList[i] = Cur;
```

```
        return RetList;
  }

public String getTreeDesc(Object auxData) {
        return “SAM Instruction Stream”;
  }
}
```

- 8 – Bibliografia

[1] Appel, Andrew W., *Modern Compiler Implementation in Java,* Cambridge University Press, 1997.
[2] Aho, A. V.; Sethi, R.; Ullman, J. D., *Compilers Principles, Tschniques, and Tools*, Addison Wesley, 1986.
[3] Meyer, Bertrant, *Object-Oriented Software Construvtion*, Prentice Hall, c.ªR. Hoare, 1998.
[4] Flanagan, D., *Java in a Nutshell*, O, Reilly, 1996.

[5] http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html
[6] http://www.cs.princeton.edu/~appel/modern
[7] http://www.cs.msu.edu/~cse450/Labs/Lab1
[8] http://www7.informatik.uni-erlangen.de/~msdoerfe/embedded/386html/toc.htm

*Gustavo Garcia Guerra*

*Mariza Andrade da Silva Bigonha*

Belo Horizonte, 25 de agosto de 2000