Processamento de Linguagens - MiEI (3^{o} ano)

Desenvolvimento de um Processador de Linguagem e de um Compilador para Máquina de Stack Virtual

Relatório de Desenvolvimento

Ana Isabel Castro (a55522) | Joana Miguel (a57127) | Lúcia Abreu (a71634) | Rui Miranda (a75488)

12 de Junho de 2017

Resumo

Este documento consiste no relatório de desenvolvimento do segundo trabalho prático de Processamento de Linguagens: definição de uma linguagem de programação imperativa, e o desenvolvimento de um compilador com base na gramática definida, com recurso aos geradores `Flex` e `Yacc`.

Para a definição da linguagem de programação, uma gramática independente de contexto (GIC) vai ser especificada.

O compilador desenvolvido tem como objetivo converter o código da linguagem desenvolvida para código *Assembly* específico para uma máquina de stack virtual.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

O **segundo trabalho** da unidade curricular de Processamento de Linguagens do 3º ano do Mestrado Integrado em Engenharia Informática consiste no desenvolvimento e implementação de um **Processador de Linguagens** e um **Compilador para Máquina de Stack Virtual**.

Os **principais objetivos** com este trabalho prático são:

- aumentar a experiência em engenharia de linguagens e em programação generativa (gramatical);
- desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe, suportado numa gramática tradutora;
- desenvolver um compilador gerando código para uma máquina de stack virtual;
- utilizar geradores de compiladores baseados em gramáticas tradutoras, concretamente o `Yacc`.

E os **objectivos secundários**:

- aumentar a experiência de uso do ambiente Linux, da linguagem imperativa `C` (para codificação das estruturas de dados e respetivos algoritmos de manipulação), e de algumas ferramentas de apoio à programação;
- rever e aumentar a capacidade de escrever gramáticas independentes de contexto que satisfaçam a condição LR() usando BNF-puro.

Estrutura do Relatório

O Capítulo 2 explica a **arquitetura da solução** do projeto.

O Capítulo 3 descreve a **Linguagem Atomic** implementada neste projeto.

No Capítulo 4 é explicado o **desenho da solução**, nomeadamente a **arquitetura da solução** do problema.

No Capítulo 5 é efetuada a **especificação do projeto**, nomeadamente a GIC - Gramática Independente do Contexto, o Analisador Léxico e o Analisador Sintático.

O Capítulo 6 descreve alguns detalhes de **implementação**, como as estruturas de dados utilizadas.

No Capítulo 7 encontram-se todos os **testes e resultados** efetuados à solução obtida.

O Capítulo 8 refere-se à **Conclusão** na qual se sumariza brevemente o que se concretizou com este trabalho.

Em **Apêndice** seguem os filtros de texto implementados e os ficheiros fonte para testes.

Capítulo 2

Enunciado e Descrição do Problema

O enunciado do projeto propõe o desenvolvimento de:

- uma linguagem de programação imperativa simples;
- um processador de linguagens segundo o método da tradução dirigida pela sintaxe, suportado numa gramática tradutora;
- desenvolver um compilador gerando código para uma máquina de stack virtual;
- utilizar geradores de compiladores baseados em gramáticas tradutoras, concretamente o Yacc.

Para tal, é necessário entender o que é uma linguagem, como processar e reconhecer uma linguagem, e como proceder à sua tradução, para que se possa implementar o compilador de código para máquina de *stack* virtual.

Uma **linguagem** é um 'mecanismo' que permite a comunicação entre dois objetos distintos. As linguagens são formadas por um conjunto finito ou infinito de **frases**, construídas a partir de um conjunto finito de **símbolos** (também designado **alfabeto**).

Qualquer linguagem natural possui um número infinito de frases. Porém nem todas as combinações (de símbolos da linguagem) são frases pertencentes à linguagem. Nas linguagens as frases têm de obedecer a um **conjunto de regras**.

A forma como os diferentes símbolos do alfabeto se combinam numa frase constitui a **sintaxe da linguagem**. A sintaxe de uma linguagem pode ser especificada por um conjunto de regras, designadas **regras sintáticas**. Uma frase que obedece às regras sintáticas de uma linguagem diz-se sintaticamente válida ou bem formada. Um programa de computador que efetua o reconhecimento sintático de uma linguagem designa-se reconhecedor.

O significado de uma frase constitui a **semântica da linguagem**. No entanto, uma frase dita sintaticamente válida pode não ser semanticamente correta. Isto acontece porque embora a combinação de símbolos obedeça às regras sintáticas da linguagem, o significado da frase resultante não faz sentido.

O processamento de uma linguagem incluí além do **reconhecimento sintático** a **análise semântica** e a **tradução da linguagem** para uma outra representação (possivelmente uma outra linguagem).

As **linguagens de programação** têm a particularidade da sintaxe e a semântica serem bastante mais simples que nas linguagens naturais e, deste modo, poderem ser descritas recorrendo a um modelo formal. Por este motivo designam-se linguagens formais.

Sendo uma linguagem um conjunto de frases então um método para descrever esse conjunto será por enumeração dos seus elementos. É óbvio que este método não é viável, uma vez que as linguagens com interesse prático são constituídas por um conjunto infinito de frases. Sendo assim, são necessários outros métodos finitos que permitam definir um conjunto infinito de frases.

Iremos então utilizar uma **gramática** para descrever a linguagem, visto que permite especificar de uma forma bastante legível e sucinta uma linguagem de programação. Nomeadamente, recorrer-se-á às **Gramáticas Independentes do Contexto -** `GIC`, umas das classe de gramáticas mais usadas, nomeadamente na definição de linguagens de programação.

Uma gramática permite construir programas que verificam se uma frase pertence ou não à linguagem. É possível construir/gerar automaticamente programas que verificam se um texto está de acordo com as regras sintáticas da linguagem, definidas por uma gramática. A gramática será usada pelo Flex - uma ferramenta geradora de analisadores sintáticos - como linguagem de especificação da sintaxe da linguagem.

Porém, a simples especificação da sintaxe de uma linguagem não é suficiente para se obter um **reconhecedor dessa linguagem**. É necessário estender a gramática com **ações semânticas** de modo a ser possível efetuar a **análise semântica** e a tarefa de **tradução**.

A tarefa inicial para a implementação deste problema será definir uma linguagem de programação imperativa simples, que deverá permitir:

- declarar e manusear variáveis atómicas do tipo **inteiro**, com os quais se podem realizar as habituais **operações aritméticas, relacionais e lógicas**.
- declarar e manusear variáveis estruturadas do tipo ***array*** (a 1 ou 2 dimensões) de inteiros, em relação aos quais é apenas permitida a operação de **indexação** (índice inteiro).
- efetuar instruções algoritmicas básicas como a **atribuição de expressões a variáveis**.
- **ler** do standard *input* e **escrever** no standard *output*.
- efetuar instruções para **controlo do fluxo de execução** — condicional e cíclica — que possam ser aninhadas.
- **definir e invocar subprogramas** sem parâmetros mas que possam retornar um resultado atómico.

Como é normal em linguagens declarativas, as variáveis deverão ser declaradas no **início do programa** e não pode haver re-declarações, nem utilizações sem declaração prévia. Se nada for explicitado, o valor da variável após a declaração é zero.

De seguida, no Capítulo 3, será explicitado como se definiu esta linguagem.

Capítulo 3

Linguagem Atomic

A tarefa inicial para a implementação de um Processador de Linguagens e respetivo Compilador de código para máquina de stack virtual é a definição da linguagem a implementar.

Assim sendo, definiu-se uma linguagem de programação imperativa, cujo nome é `@tomic` e cujos ficheiros possuem a extensão `.at`. O compilador desta linguagem é designado por `atoc`.

As variáveis - neste caso inteiros - nesta linguagem são declarados com `atom`, como por exemplo: `atom nomeVariavel;` e um programa escrito na linguagem `@tomic` é iniciado com `start()`.

Figura 3.1: Logótipo da Linguagem Atomic

A Linguagem `@tomic` possui as seguintes funcionalidades, descritas no capítulo anterior:

- Declarar e manusear variáveis atómicas do tipo **inteiro**, com os quais se podem realizar operações:
  - **Aritméticas**
    - `var <- 723 + 512;`
    - `var <- 723 * 512;`
    - `var <- 723 - 512;`
    - `var <- 723 / 512;`
    - `var <- 723 % 512;`
  - **Relacionais**
    - `var >= 723`

    * `var <= 723`
    * `var == 723`
    * `var != 723`
  - **Lógicas**
    * `var <- (res1 && res2);`
    * `var <- (res1 || res2);`
    * `var <- (!res1);`

- Declarar e manusear variáveis estruturadas do tipo ***array*** (a 1 ou 2 dimensões) de inteiros, em relação aos quais é apenas permitida a operação de **indexação** (índice inteiro).
  - `atom lista [5];`
  - `lista[3] <- num;`
  - `res <- lista[3];`
  - `atom matriz [8][8];`
  - `matriz [2][7] <- res;`
  - `res <- matriz [2][7];`

- efetuar instruções algoritmicas básicas como a **atribuição de expressões a variáveis**.
  - `var <- res + 17;`
  - `var <- res * 512;`
  - `var <- res - total;`
  - `var <- res / 5;`

- **ler** do standard *input* e **escrever** no standard *output*.
  - `print("Insira um inteiro");`
  - `res <- read();`
  - `print(res);`

- efetuar instruções para **controlo do fluxo de execução** — condicional e cíclica — que possam ser aninhadas.
  - Cíclica
    * `while (res < 5){ }`
    * `while (res < 5){ if(tmp < 7) {} }`
  - Condicional
    * `if(res < 5){ } else {}`
    * `if(res < 5){ if(tmp < 7) {} } else {}`

- **definir e invocar subprogramas** sem parâmetros mas que possam retornar um resultado atómico.

- `start(){}`
- `nomeFunc();`
- `res <- nomeFunc();`

Capítulo 4

Desenho da Solução

4.1 Arquitetura da Solução

O compilador da linguagem Atomic (*atoc*) foi implementado com o recurso dos geradores de compiladores `Yacc` e `Flex`. O compilador vai ler um ficheiro de texto, verificar se a estrutura e sintaxe do programa está válida, e gerar código *Assembly* da máquina virtual.

Com o `Flex` foi possível a geração de um analisador léxico, que divide o ficheiro de texto em *tokens*. O analisador léxico é invocado a partir da função *yylex()*.

Esses *tokens* são depois lidos pelo `Yacc`, executando ações específicas para cada *token* através da função *yyparse()*.

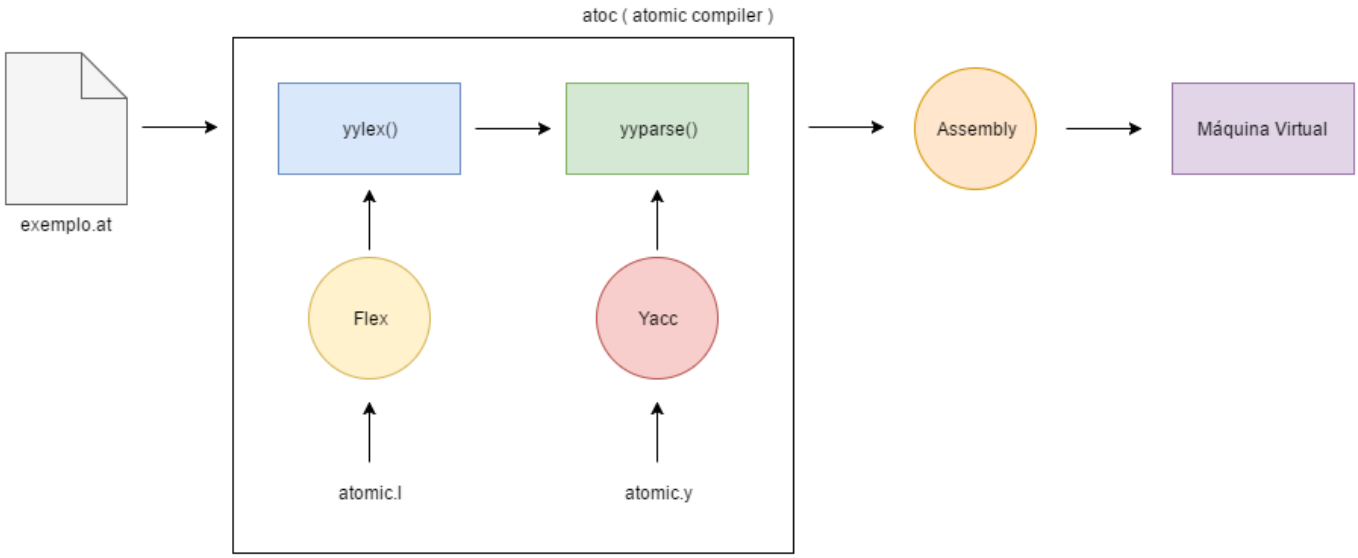

Figura 4.1: Esquema representativo da arquitetura do compilador da linguagem Atomic

Capítulo 5

Especificação

5.1 GIC - Gramática Independente de Contexto

Após a descrição da linguagem Atomic no Capítulo 3, passou-se para a especificação da sua gramática. O Símbolo Inicial é Atomic, os Símbolos Terminais são escritos só em maiúsculas ou entre apostrofes; os restantes (sempre começados por maiúsculas) serão os Símbolos Não-Terminais.

De seguida segue a especificação da gramática independente de contexto da linguagem Atomic, no formalismo designado por *Backus Naur Form* (BFN), que é universalmente aceite para escrever gramáticas independentes do contexto.

```
GIC = <
    T = {VOID, ATOM, ID, START, AT, IF, ELSE, WHILE, READ, PRINT,
            STRING, AND, OR, GEQ, LEQ, EQU, DIF, NUM, RETURN,
            '(', ')', '{', '}', '[', ']', ';', ',', '!',
            '<', '>', '+', '-', '*', '/', '%'},
    N = {Atomic, Funcoes, Funcao, Start, Body, ListaDeclars, LinhaDeclar,
            Declars, Declar, Insts, Inst, Condicional, Ciclica, Atribuicao,
            Invocacao, Leitura, Escrita, Condicao, Exp, Termo, Fator, Var, Return},
    S = Atomic,
    P = {
            p0: Atomic       --> Funcoes Start
            p1: Funcoes      --> Funcoes Funcao
            p2:              |
            p3: Funcao       --> VOID ID '('')' '{' Body '}'
            p4:              | ATOM ID '('')' '{' Body '}'
            p5: Start        --> START '('')' '{' Body '}'
            p6: Body         --> ListaDeclars Insts Return
            p7: ListaDeclars --> ListaDeclars LinhaDeclar
            p8:              |
            p9: LinhaDeclar  --> ATOM Declars ';'
            p10: Declars     --> Declars ',' Declar
            p11:             | Declar
            p12: Declar      --> ID
            p13:             | ID AT Exp
```

```
        p14:             | ID '[' Exp ']'
        p15:             | ID '[' Exp ']' '[' Exp ']'
        p16: Insts       --> Insts Inst
        p17:             |
        p18: Inst        --> Condicional
        p19:             | Ciclica
        p20:             | Atribuicao
        p21:             | Invocacao
        p22:             | Leitura
        p23:             | Escrita
        p24: Condicional --> IF '(' Condicao ')' '{' Insts '}'
        p25:             | IF '(' Condicao ')' '{' Insts '}' ELSE '{' Insts '}'
        p26: Ciclica     --> WHILE '(' Condicao ')' '{' Insts '}'
        p27: Atribuicao  --> Var AT Exp
        p28: Invocacao   --> ID '(' ')' ';'
        p29: Leitura     --> ID AT READ '(' ')' ';'
        p30: Escrita     --> PRINT '(' Exp ')' ';'
        p31:             | PRINT '(' STRING ')' ';'
        p32: Condicao    --> Condicao AND Condicao
        p33:             | Condicao OR Condicao
        p34:             | '(' Condicao ')'
        p35:             | '!' Condicao
        p36:             | Exp '<' Exp
        p37:             | Exp '>' Exp
        p38:             | Exp LEQ Exp
        p39:             | Exp GEQ Exp
        p40:             | Exp DIF Exp
        p41:             | Exp EQU Exp
        p42:             | Exp
        p43: Exp         --> Termo
        p44:             | Exp '+' Termo
        p45:             | Exp '-' Termo
        p46:             | '(' Exp ')'
        p47: Termo       --> Fator
        p48:             | Termo '*' Fator
        p49:             | Termo '/' Fator
        p50:             | Termo '%' Fator
        p51:             | '(' Termo ')'
        p52: Fator       --> NUM
        p53:             | Var
        p54:             | ID '(' ')'
        p55:             | READ '(' ')'
        p56: Var         --> ID
        p57:             | ID '[' Exp ']'
        p58:             | ID '[' Exp ']' '[' Exp ']'
        p59: Return      --> RETURN Exp ';'
        p60:             |
}
```

>

5.2 Analisador Léxico (Flex)

Definiu-se um analisador léxico em Flex que tem como input o código fonte da linguagem Atomic e devolve para o output um conjunto de Tokens, que opcionalmente podem ter valores associados. Este tokens serão mais tarde consumidos pela gramática tradutora implementada com o Yacc.

5.2.1 Especificar os padrões das Expressões Regulares

Esta etapa consistiu em especificar os padrões através de expressões regulares que permitam capturar os campos pretendidos no código fonte da linguagem Atomic.

Todas as expressões regulares que têm a forma `(?i:expressao)` são *case insensitive*, isto é, fazem *match* independentemente da palavra em questão estar escrita em maiúsculas ou minúsculas.

- `[-*/;,\[\]=><!+\(\)\{\}%]` → faz *match* com a seguinte lista de caracteres: `-`, `*`, `/`, `;`,, `[`, `]`, `=`, `>`, `!`, `+`, `(`, `)`, `{`, `}`, `%`
- `[-]?[0-9]+` → faz *match* com por números compostos por um ou mais dígitos e opcionalmente com o caracter `-` antes do número
- `(?i:void)` → faz *match* com a palavra void
- `(?i:start)` → faz *match* com a palavra start
- `(?i:if)` → faz *match* com a palavra if
- `(?i:else)` → faz *match* com a palavra else
- `(?i:while)` → faz *match* com a palavra while
- `(?i:atom)` → faz *match* com a palavra atom
- `(?i:read)` → faz *match* com a palavra read
- `(?i:print)` → faz *match* com a palavra print
- `(?i:return)` → faz *match* com a palavra return
- `\<-` → faz *match* com a seguinte expressão `<-`
- `\<=` → faz *match* com a seguinte expressão `<=`
- `\>=` → faz *match* com a seguinte expressão `>=`
- `!=` → faz *match* com a seguinte expressão `!=`
- `==` → faz *match* com a seguinte expressão `==`

- `&&` → faz *match* com a seguinte expressão `&&`
- `\|\|` → faz *match* com a seguinte expressão `||`
- `[_a-zA-Z][_A-Za-zs0-9]*` → faz *match* com uma palavra que contém letras, *underscores* (_) ou números, desde que não comece com um número e tenha pelo menos uma letra ou underscore
- `\"[^"]*\"` → faz *match* com qualquer texto que se encontra entre aspas
- `\/\/[^\n]*` → faz *match* com qualquer linha que comece com os caracteres `//`
- `([ ]|\n|\t|\r)` → faz *match* com *whitespace*
- `.` → faz *match* com qualquer caracter

5.2.2 Identificar as Ações Semânticas

Para cada uma das Expressões Regulares Especificadas, definiram-se as seguintes ações semânticas:

- `[-*/;,\[\]=><!+\(\)\{\}%]` → retorna o carácter que fez *match*
- `[-]?[0-9]+` → converte e guarda o número que fez *match* e retorna o token NUM
- `(?i:void)` → retorna o token `VOID`
- `(?i:start)` → retorna o token `START`
- `(?i:if)` → retorna o token `IF`
- `(?i:else)` → retorna o token `ELSE`
- `(?i:while)` → retorna o token `WHILE`
- `(?i:atom)` → retorna o token `ATOM`
- `(?i:read)` → retorna o token `READ`
- `(?i:print)` → retorna o token `PRINT`
- `(?i:return)` → retorna o token `RETURN`
- `\<-` → retorna o token `AT`
- `\<=` → retorna o token `LEQ`
- `\>=` → retorna o token `GEQ`
- `!=` → retorna o token `DIF`
- `==` → retorna o token `EQU`
- `&&` → retorna o token `AND`

- `\|\|` → retorna o token `OR`
- `[_a-zA-Z][_A-Za-zs0-9]*` → copia a string que fez *match* e retorna o token `ID`
- `\"[^"]*\"` → copia a string que fez *match* e retorna o token `STRING`
- `\/\/[^\n]*` → não tem ação semântica associada, ignorando qualquer linha que comece com os caracteres `//`
- `([ ]|\n|\t|\r)` → não tem ação semântica associada, ignorando todos os caracteres *whitespace* que não façam *match* com as expressões regulares anteriores
- `.` → ao fazer match com esta expressão regular, vai-se devolver um erro já que devia ter havido um match com alguma das expressões regulares anteriores

5.3 Analisador Sintático / Gramática Tradutora (Yacc)

5.3.1 Identificar as Ações Semânticas

O programa em YACC contém as regras da gramática da linguagem. Em cada uma das regras da gramática existe uma ação semântica associada. Para cada uma destas regras (produções), é de seguida analisada qual a ação semântica correspondente.

Atomic

- `Atomic : Funcoes Start`

  A ação semântica associada é a de concatenação do código em *assembly* produzido pelo termo `Funcoes` e pelo termo `Start`

Funcoes

- `Funcoes : Funcoes Funcao`

  De forma análoga à regra anterior, aqui a ação semântica também efetua uma concatenação do código em *assembly* produzido pelos termos `Funcoes` e `Funcao`.

- `Funcoes |` ϵ

  Esta produção indica que é se pode encontrar o vazio, ou seja, o caso de paragem na procura por funções. Assim sendo, é neste momento que se inicializa a string que vai conter o código *assembly* gerado por cada um dos termos `Funcao`. A partir deste ponto, por *backtracking*, será construido o código *assembly* de todas as funções.

Funcao

- `Funcao : VOID ID '('')' '{' Body '}'`

  A produção do termo `Funcao` indicada é a identificação da estrutura de uma função sem tipo de retorno. Quando se identifica uma função, garante-se que não está a ser redeclarada, isto é, se já anteriormente foi declarada uma função com o mesmo nome. Para além desta verificação é também guardado o nome da função e é colocado o endereço a 0. O nome serve para associar cada uma das variáveis que forem declaradas, e o endereço será utilizado para referenciar as variáveis utilizadas no programa.

  O termo `Funcao` produz o código *assembly* gerado pelo seu corpo, para além de conter as operações de *assembly* responsáveis por garantir a chamada e retorno da função.

- `Funcao : ATOM ID '('')' '{' Body '}'`

  Nesta produção é identificada a definição de uma função com tipo de retorno. De forma similar à produção anterior, também aqui é garantido que não se está a redeclarar a função. É também guardado o seu nome e o valor do endereço atual é colocado com o valor 0.

Start

- `Start : START`

  Apesar de esta produção identificar a função principal do programa, o seu tratamento é bastante idêntico ao tratamento dado para as duas produções anteriores. É guardado o nome da função atual e é colocado o valor do endereço atual a 0. Dado que esta é a função principal do programa, é também adicionado o código *assembly* para inicializar e finalizar o programa.

  Este termo retorna o código *assembly* de todas as suas instruções.

Body

- `Body : ListaDeclars Insts Return`

  A produção de definição de um corpo, contém três constituintes: lista de declarações, instruções e retorno. A ação semântica é a concatenação do código de *assembly* produzido por cada um dos respetivos termos.

ListaDeclars

- `ListaDeclars : ListaDeclars LinhaDeclar`

  Esta regra representa a hipótese que existe de uma lista de declarações poder conter pelo menos uma linha de declaração. Cada um destes termos produz o *assembly* dos seus constituintes e portanto, uma vez mais, é responsabilidade desta ação semântica, concatenar as diferentes produções de código *assembly* e retorná-las.

- `|` ϵ

  Esta produção identifica o começo da lista de declarações e assim sendo é inicializada a estrutura que vai comportar as produções de *assembly* de cada um dos constituintes.

LinhaDeclar

- `LinhaDeclar : ATOM Declars ';'`

  Uma linha de declaração tem um conjunto de declarações. Assim, a ação semântica correspondente é apenas devolver o código *assembly* calculado para todas as declarações.

Declars

- `Declars : Declars ',' Declar`

  Um conjunto de declarações pode ser constituído por várias declarações. Assim, é feita a concatenação dos diferentes códigos *assembly* gerados para cada uma das declarações.

- `| Declar`

  Um conjunto de declarações pode ser constituído por apenas uma declaração. Quando identificada a primeira declaração, é devolvido o seu código *assembly*.

Declar

- `Declar : ID`

  Uma declaração pode ser definida por apenas um identificador de uma variável. Nesta ação semântica, garante-se que a variável não está a ser redeclarada, guarda-se a definição da variável na estrutura das variáveis com o endereço que a variável terá (valor a ser utilizado mais tarde para gerar *assembly* relacionado com a variável) e é produzido o código *assembly* respetivo.

- `| ID AT NUM`

  Uma declaração pode também conter uma atribuição de um número. Assim, quando a variável é criada, contém logo um valor. Assim, de forma análoga a todas as declarações, garante-se que a variável não foi já declarada, guarda-se a variável declarada em conjunto com o seu endereço. Produz-se o código *assembly* respetivo.

- `| ID '[' NUM ']'`

  Uma variável pode também ser um *array*. Assim, a ação semântica correspondente é garantir a não redeclaração da variável, guardar a variável declarada e os seus endereços. Os endereços aqui equivalem ao número de posições do *array*.

  É feita também uma validação ao tamanho do *array* para garantir que não é declarado um vector com tamanho negativo.

- `| ID '[' NUM ']' '[' NUM ']'`

  De forma análoga à regra anterior, também pode ser declarada uma matriz. Assim, uma vez mais, é garantido que não se redeclara a variável, guarda-se a variável e os endereços correspondentes (número de linhas $*$ o número de colunas).

  É também validado o número de linhas e de colunas para garantir que não é colocado nenhum número negativo.

Insts

- `Insts : Insts Inst`

  Um conjunto de instruções pode ser constituído por múltiplas instruções, assim, a ação semântica correspondente é a concatenação do código *assembly* produzido pelos termos `Insts` e `Inst`.

- `|`ϵ

  Quando se chega ao fim da identificação das instruções, é inicializada a estrutura onde vão ser concatenadas as operações de *assembly* das diversas instruções.

Inst

- `Inst : Condicional`

  Uma instrução pode ser uma instrução condicional e como tal, a ação semântica correspondente é produzir o código *assembly* produzido pela instrução condicional.

- `| Ciclica`

  Tal como todas as produções das instruções, a ação semântica é apenas repassar o código *assembly* produzido pela instrução correspondente, que neste caso é Cíclica.

- `| Atribuicao`

  Tal como todas as produções das instruções, a ação semântica é apenas repassar o código *assembly* produzido pela instrução correspondente, que neste caso é Atribuicao.

- `| Invocacao`

  Tal como todas as produções das instruções, a ação semântica é apenas repassar o código *assembly* produzido pela instrução correspondente, que neste caso é Invocacao.

- `| Leitura`

  Tal como todas as produções das instruções, a ação semântica é apenas repassar o código *assembly* produzido pela instrução correspondente, que neste caso é Leitura.

- `| Escrita`

  Tal como descrito a cima, e sendo esta uma instrução, a ação semântica é apenas repassar o código *assembly* produzido pela instrução Escrita

Condicional

- `Condicional : IF '(' Condicao ')' '{' Insts '}'`

  A produção desta instrução condicional contém a definição de um *if* sem *else*. Assim, a ação semântica correspondente é construir as operações *assembly* correspondentes à estrutura de controlo if e devolve-las.

- `| IF '(' Condicao ')' '{' Insts '}' ELSE '{' Insts '}'`

  A produção desta instrução condicional é em tudo semelhante à anterior, contudo, o *assembly* gerado por estra ação semântica, contém a lógica para incluir o *else*.

Ciclica

- `Ciclica : WHILE '(' Condicao ')' '{' Insts '}'`

  A regra da instrução cíclica, define como está definida um ciclo while. Assim, a ação semântica correspondente produz o código *assembly* associado a um ciclo *while*, contemplando a condição e as instruções correspondentes.

Atribuicao

- `Atribuicao : Var AT Exp ';'`

  A produção da instrução de atribuição é feita sobre uma variável ao qual é atribuída uma expressão. Assim, a ação semântica desta regra é garantir que são geradas as instruções *assembly* necessárias para guardar na variável o valor da expressão.

Invocacao

- `Invocacao : ID '(' ')' ';'`

  A regra da instrução de invocação é caracterizada pelo identificador da função a ser chamada. A ação semântica correspondente é gerar o código *assembly* responsável por chamar uma função.

Leitura

- `Leitura : Var AT READ '(' ')' ';'`

  Para a instrução de leitura, é guardada na variável um valor lido. Assim, a ação semântica calcula o código *assembly* associado ao guardar o valor na variável da leitura de um número.

Escrita

- `Escrita : PRINT '(' Exp ')' ';'`

  A instrução de escrita funciona de forma bastante direta, a ação semântica gera o código *assembly* para introduzir o valor da expressão na pilha e é gerado o código para imprimir para o ecrã o seu valor.

- `| PRINT '(' STRING ')' ';'`

  A regra da instrução de escrita contempla também a hipótese de imprimir para o ecrã string. Assim, a ação semântica correspondente é gerar o código *assembly* responsável para imprimir o valor da string para o ecrã.

Condicao

- `Condicao : Condicao AND Condicao`

  Em todas as produções da instrução de condição, a ação semântica correspondente é gerar o código *assembly* correspondente à operação condicional que especificam combinando cada um dos seus operandos.

- `| Condicao OR Condicao`

  Como em todas as produções da instrução de condição, a ação semântica produz o código *assembly* correspondente à condição associada entre os seus operandos.

- `| '(' Condicao ')'`

  Uma condição pode ter parênteses a envolve-la, e, assim, a ação semântica consiste em transmitir o *assembly* produzido pela condição.

- `| '!' Condicao`

  Como em todas as produções da instrução de condição, a ação semântica produz o código *assembly* correspondente à condição associada entre os seus operandos.

- `| Exp '<' Exp`

  Como em todas as produções da instrução de condição, a ação semântica produz o código *assembly* correspondente à condição associada entre os seus operandos.

- `| Exp '>' Exp`

  Como em todas as produções da instrução de condição, a ação semântica produz o código *assembly* correspondente à condição associada entre os seus operandos.

- `| Exp LEQ Exp`

  Como em todas as produções da instrução de condição, a ação semântica produz o código *assembly* correspondente à condição associada entre os seus operandos.

- `| Exp GEQ Exp`

  Como em todas as produções da instrução de condição, a ação semântica produz o código *assembly* correspondente à condição associada entre os seus operandos.

- `| Exp DIF Exp`

  Como em todas as produções da instrução de condição, a ação semântica produz o código *assembly* correspondente à condição associada entre os seus operandos.

- `| Exp EQU Exp`

  Como em todas as produções da instrução de condição, a ação semântica produz o código *assembly* correspondente à condição associada entre os seus operandos.

- `| Exp`

  Nesta produção, apenas uma expressão existe. A ação semântica correspondente é de devolver o código *assembly* da expressão.

Exp

- `Exp  : Termo`

  Em cada uma das produções do termo, é devolvido o código *assembly* à operação indicada. Neste caso e como é só um termo, apenas o seu código *assembly* é devolvido.

- `| Exp '+' Termo`

  Como em todas as produções do termo, a ação semântica correspondente é devolver o código *assembly* à operação indicada (mais).

- `| Exp '-' Termo`

  Como em todas as produções do termo, a ação semântica correspondente é devolver o código *assembly* à operação indicada (menos).

- `| '(' Exp ')'`

  Uma expressão pode ter parênteses a envolve-la. Assim, a ação semântica consiste apenas em transmitir o *assembly* produzido pela expressão.

Termo

- `Termo  : Fator`

  Na regra do termo onde se permite um fator, a operação semântica associada é produzir o código *assembly* do fator.

- `| Termo '*' Fator`

  Esta produção do termo é composta por dois operandos sob uma operação. Assim, com a ação semântica, gera-se o código *assembly* correspondente à operação (multiplicação) entre os dois operandos.

- `| Termo '/' Fator`

  Esta produção do termo é composta por dois operandos sob uma operação. Assim, com a ação semântica, gera-se o código *assembly* correspondente à operação (divisão) entre os dois operandos.

- `| Termo '%' Fator`

  Esta produção do termo é composta por dois operandos sob uma operação. Assim, com a ação semântica, gera-se o código *assembly* correspondente à operação (resto da divisão) entre os dois operandos.

- `| '(' Termo ')'`

  Um `Termo` pode ter parênteses a envolve-lo. Assim, a ação semântica consiste apenas em transmitir o *assembly* produzido pelo termo.

Fator

- `Fator : NUM`

  Um fator pode ser um número. Nesse caso, devolve-se a operação *assembly* correspondente à de adicionar um número à pilha.

- `| Var`

  Um fator pode ser uma variável. No caso de ser uma variável, devolve-se o código asssembly correspondente à de adicionar o valor guardado na variável (posição de memória) à pilha.

- `| ID '(' ')'`

  Um fator pode ser uma invocação de função. No caso de ser um invocação, a ação semântica produz o código asssembly correspondente à de invocar uma função.

- `| READ '(' ')'`

  Um fator pode, por fim, ser uma invocação leitura de input. No caso de ser uma leitura de input, a ação semântica produz o código asssembly correspondente à leitura de um número inteiro.

Var

- `Var : ID`

  Uma variável pode ser uma variável do tipo simples. A ação semântica garante que a variável se encontra declarada e é do tipo simples. Para além disso, são também produzidos dois tipos de código *assembly*, um para cada forma que a variável for utilizada: uma para leitura do seu valor; e outra para guardar algum valor na sua posição da memória.

- `| ID '[' Exp ']'`

  De forma semelhante à regra anterior, também quando a variável é um *array*, é feita uma verificação para garantir que ele se encontra declarado e é do tipo *array*. São também gerados os dois códigos *assembly* para ler e escrever na posição do *array*.

- `| ID '[' Exp ']' '[' Exp ']'`

  De forma semelhante às regra anterior, também quando a variável é uma matriz, é feita uma verificação para garantir que esta se encontra declarada e é do tipo matriz. São também gerados os dois códigos *assembly* para ler e escrever na posição da matriz.

Return

- `Return : RETURN Exp ';'`

  Na regra de *return* a ação semântica correspondente é garantir que para a função onde a instrução está a aparecer, é necessário *return*. Caso não seja é dado erro. É produzido o código *assembly* associado ao *return* de uma função.

- | ϵ

  Caso não exista instrução de *return*, é também feita uma verificação para caso ela fosse obrigatória (função com tipo de retorno) e ela não exista, seja dado o erro correspondente.

5.3.2 Regras de Tradução para *Assembly* da VM

Como já explicado na secção anterior, é responsabilidade de algumas ações semânticas produzir o código assembly associado às suas produções.

De seguida, explica-se de forma geral para cada um dos constituíntes do programa, como são gerados os códigos assembly.

Programa

O programa é sempre inicializado com uma primeira variável. Esta variável é global e serve para guardar o valor de retorno das funções que forem invocadas.

Funções

O código de cada uma das funções contém a *label* da função correspondente (`funcao_NOME`) que delimita o início da função. De seguida estão todas as instruções que pertencem à função e são finalizadas por uma instrução de `return`. Esta expressão pode ser precedida da salvaguarda do valor de retorno na variável global ou não caso a função assim o exija.

Valor de retorno

Nas funções que exijam valor de retorno, é criado o código de *assembly* para guardar o valor calculado na posição da variável global (`storeg 0` com posição `gp[0]`).

Invocação de funções

A invocação das funções é feita com auxilio à expressão `call`. Antes desta expressão é calculado o endereço da função através da instrução `pusha`.

Declaração de variáveis

Para cada uma das variáveis declaradas, é reservado um endereço na stack. Esta reserva é feita através da instrução `pushi VALOR`. Este valor está associado ao valor de inicialização da variável. Caso não exista será zero. Cada variável têm também associado o endereço para poder ser gerado o código correspondente numa poração de código onde esta apareça.

As variáveis podem também ser multivaloradas, isto é, podem ser *arrays* ou matrizes. Neste caso, o espaço que fica alocado é correspondente ao seu tamanho, tamanho do *array* ou número de linhas $*$ número de colunas, respetivamente.

Atribuição de variáveis

A atribuição de valores a variáveis está ineremente ligada ao tipo de variável, contudo, de forma geral é calculado o endereço da memória onde colocar a expressão da direita e é colocada a expressão da direita. No final é guardado o valor.

No caso de ser uma variável simples, o cálculo do endereço da variável é feito através da expressão `pushl` (variável local). No caso de *arrays* ou matrizes, é feita o cálculo da posição de acordo com a posição escolhida. No caso do *array* é endereço base + posição escolhida e no caso da matriz é endereço base + linha escolhida $*$ nrColunas + coluna.

Estrutura condicional *If-then-Else*

O código de *assembly* produzido para uma estrutura do tipo *If-then-Else* tem a lógica típica subjacente. Assim, primeiramente é gerado código para aferir o teste da condição lógica do if. De seguida, é feito um salto condicional caso a condição não se verifique (`jz`) para a *label* que identifica o começo do *else* (`else_NRIF`). Caso a condição se verifique vêm as instruções do *if* e de seguida um salto não condicional (`jump`) para a *label* do fim do if (`if_fim_NRIF`). Depois da *label* do *else*, seguem as instruções do *else* que são finalizadas pela *label* do fim do *if*.

É guardado o número de ifs definidos para se garantir que não são geradas *label* iguais.

Estrutura cíclica *While*

A estrutura cíclica *While*, tem o código *assembly* típico. Primeiramente é inserida uma *label* que indica o começo da condição do ciclo `while_NRWHILE`. É então gerado o código para aferir o teste da condição lógica do ciclo. De seguida, é feito um salto condicional caso a condição não se verifique (`jz`) para a *label* final do ciclo(`while_fim_NRWHILE`). Caso a condição se verifique vêm as instruções do *while* e de seguida um salto não condicional (`jump`) para o inicio do ciclo. Eventualmente a condição falhará e o clico terminará.

É também guardado o número de ciclos definidos para se garantir que não são geradas *label* iguais.

Leitura do *input*

O código de *assembly* da leitura de input é feita através das instruções `read` e `atoi`. Este valor é posteriormente guardado na variável (*array* ou matriz) correspondente.

Escrita no ecrã

O código de *assembly* gerado para imprimir no ecrã é feita de duas formas diferentes consoante o que se pretende escrever no ecrã, caso seja:

- **uma string:** primeiramente é posta a string na pilha (`pushs STRING`) e depois é feita a escrita com `writes`
- **uma expressão:** primeiramente é colocada a expressão na pilha e depois é feita a sua escrita com `writei`

Condições e expressões numéricas

As condições e expressões numéricas têm o código *assembly* associado e definido para cada uma das condições e expressões. Para a expressão de negação (`!`) inseriu-se o valor 1, fez-se a troca (`swap`) e fez-se a subtração aplicando assim a negação lógica (`negação = 1-n`). Para o operador diferente (`!=`) a lógica foi similar, aplicou-se primeiro o igual e depois negou-se o valor de verdade com a expressão da negação (troca seguida de `negação = 1-n`).

Para a implementação do operador lógico *And* utilizou-se a operação de multiplicação (`mul`) que, sendo o 0 o valor neutro da multiplicação, dará diferente de 0 se todos forem verdadeiros. Para o operador lógico *Or* utilizou-se a soma (`add`) que assim garante que caso algum seja positivo se obtém um valor positivo.

Capítulo 6

Implementação

Após a implementação do analisador léxico com recurso ao `Flex` e da gramática tradutora utilizando o `Yacc`, verificou-se a necessidade de saber se durante a execução de um programa:

- Uma variável já tinha sido declarada dentro de uma função;
- Consultar informação sobre uma variável.

Ao verificar se uma variável já se encontra declarada ou não, está-se a garantir que a linguagem Atomic não permitirá a re-declaração de variáveis. Este é um dos requisitos presente no enunciado do trabalho prático.

Para além disto, é fundamental consultar a informação de uma variável para desta forma se conseguir realizar as diversas instruções da linguagem Atomic.

Devido a estas necessidades observadas teve que se criar uma estrutura de dados auxiliar que permita verificar que uma variável já está ou não declarada e consultar a informação de uma dada variável.

6.1 Estrutura de Dados

A estrutura de dados criada trata-se de uma tabela de Hash em que a chave é uma string correspondente ao nome da função e o valor é uma estrutura que contém um booleano e uma nova tabela de Hash. O booleano permite saber se a função devolve ou não um atom (um inteiro), enquanto que a tabela de Hash guarda as variáveis declaradas nesta função e a sua informação, ou seja, cada chave é uma string correspondente ao nome da variável e o valor é a informação dessa variável.

A informação de uma variável contém 4 campos:

- **Tipo**: uma Enum que especifica qual o tipo desta variável. Pode ser Variável Simples, Array ou Matriz;
- **Endereço**: a posição desta variável na stack. No caso dos arrays e das matrizes, é a posição inicial das mesmas;
- **Número Linhas**: tamanho de uma variável array, ou número de linhas de uma matriz (não é usado para variáveis simples);

- **Número Colunas**: número de colunas de uma matriz (não é usado nos outros tipos de variáveis).

A Figura 6.1 representa a estrutura de dados acima referida.

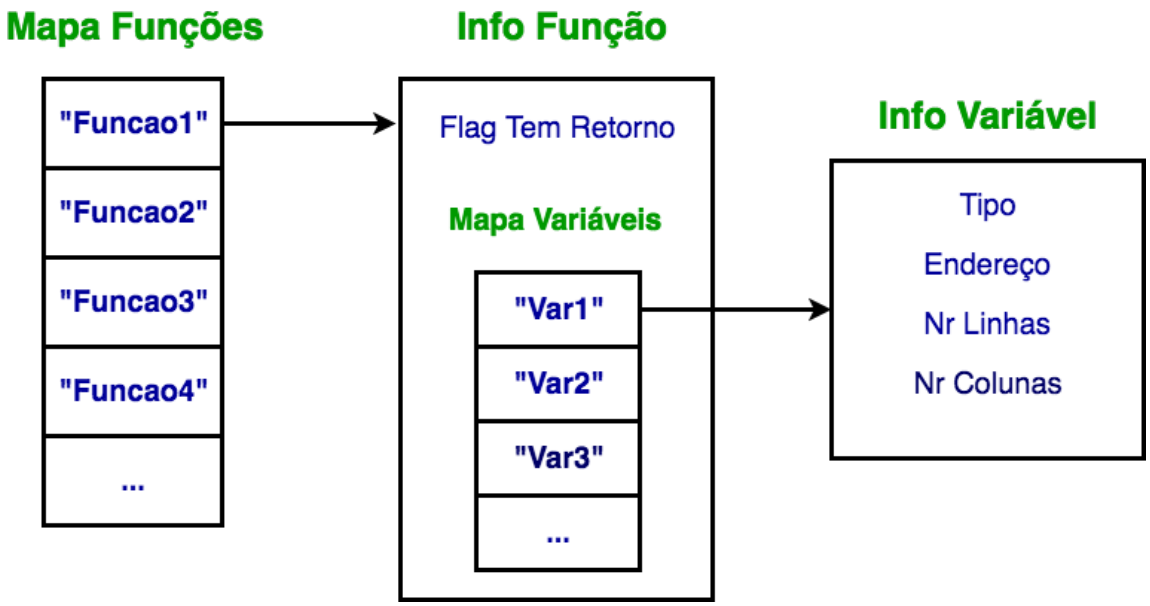

Figura 6.1: Esquema da estrutura de dados que guarda a informação referente a variáveis e funções de um programa

Conclui-se, então, que ao implementar esta estrutura de dados permite-se que diferentes funções (incluindo a função principal: Start) declarem a mesma variável, ou seja, que tenha variáveis com o mesmo nome. Adicionalmente, consegue-se validar se uma chamada a uma função pode devolver ou não um atom.

6.1.1 API

Nesta secção pode-se consultar a API que permite a manipulação da estrutura de dados descrita na secção anterior.

```
// devolve um novo mapa de funções
MapaFunVars novo_mapa_fun_vars();

// inserir função
void insere_funcao(MapaFunVars mapa, char* funcao, bool retornaAtom);

// inserir variável simples
void insere_variavel_simples(MapaFunVars mapa, char* funcao,
                char* var, int endereco);
```

```
// inserir variável array
void insere_variavel_array(MapaFunVars mapa, char* funcao,
                char* var, int endereco, int tam);

// inserir variável matriz
void insere_variavel_matriz(MapaFunVars mapa, char* funcao,
                char* var, int endereco, int linhas, int colunas);

// testa se existe função
bool existe_funcao(MapaFunVars mapa, char* funcao);

// testa se a funcao existe e retorna um atom (inteiro)
bool existe_funcao_com_retorno(MapaFunVars mapa, char* funcao);

// testa se existe variável
bool existe_variavel(MapaFunVars mapa, char* funcao, char* var);

// devolve info de variável
InfoVar info_variavel(MapaFunVars mapa, char* funcao, char* var);
```

Capítulo 7

Testes e Resultados

Para o problema abordado, realizaram-se os seguintes testes e respetivos resultados: O código Assembly gerado encontra-se em Apêndice B.

7.1 Teste 1

O programa lê 4 números e diz se podem ser os lados de um quadrado.

7.1.1 Programa em Atomic

```
start(){
        // ler 4 numeros e dizer se podem ser os lados de um quadrado

        atom l1;
        atom l2;
        atom l3;
        atom l4;

        print("Insira primeiro numero:");
        l1 <- read();

        print("Insira segundo numero:");
        l2 <- read();

        print("Insira terceiro numero:");
        l3 <- read();

        print("Insira quarto numero:");
        l4 <- read();

        if (l1==l2) {
                if(l1==l3){
```

```
                        if(l1==l4){
                                print("Pode ser um quadrado!");
                        }
                        else {print("Não pode ser um quadrado!");}
                }
                else {print("Não pode ser um quadrado!");}
        }
        else {
                print("Não pode ser um quadrado!");
        }

}
```

7.1.2 Resultado

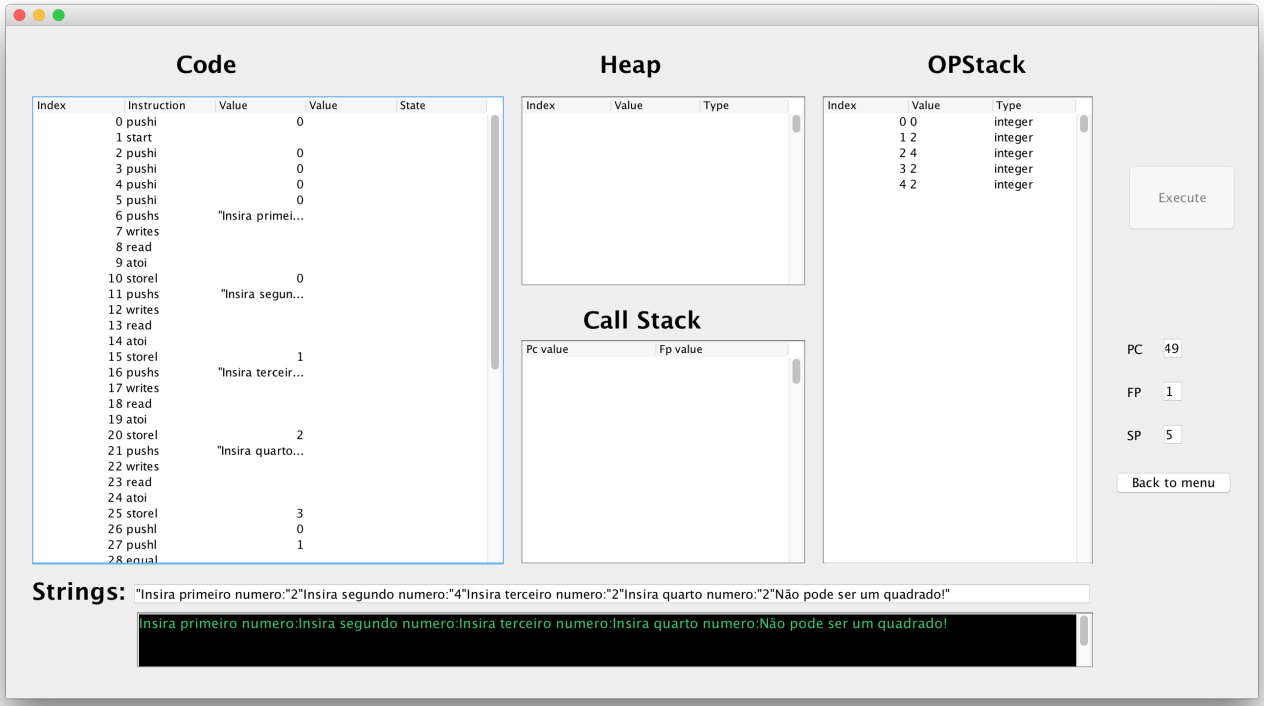

Figura 7.1: Resultado do Teste 1

7.2 Teste 2

O programa lê um inteiro N, depois lê N números e escreve o menor deles.

7.2.1 Programa em Atomic

```
start() {
    // ler um inteiro N, depois ler N números e escrever o menor deles

    atom N;
    atom min;
    atom lidos <- 1;
    atom lido;

    print("Introduza o numero de inteiros a ler: ");
    N <- read();

    print("Insira um inteiro: ");
    min <- read();

    while (lidos < N){
        print("Insira um inteiro: ");
        lido <- read();
        if (lido < min) {
            min <- lido;
        }
        lidos <- lidos + 1;
    }

    print("O valor minimo -> ");
    print(min);
}
```

7.2.2 Resultado

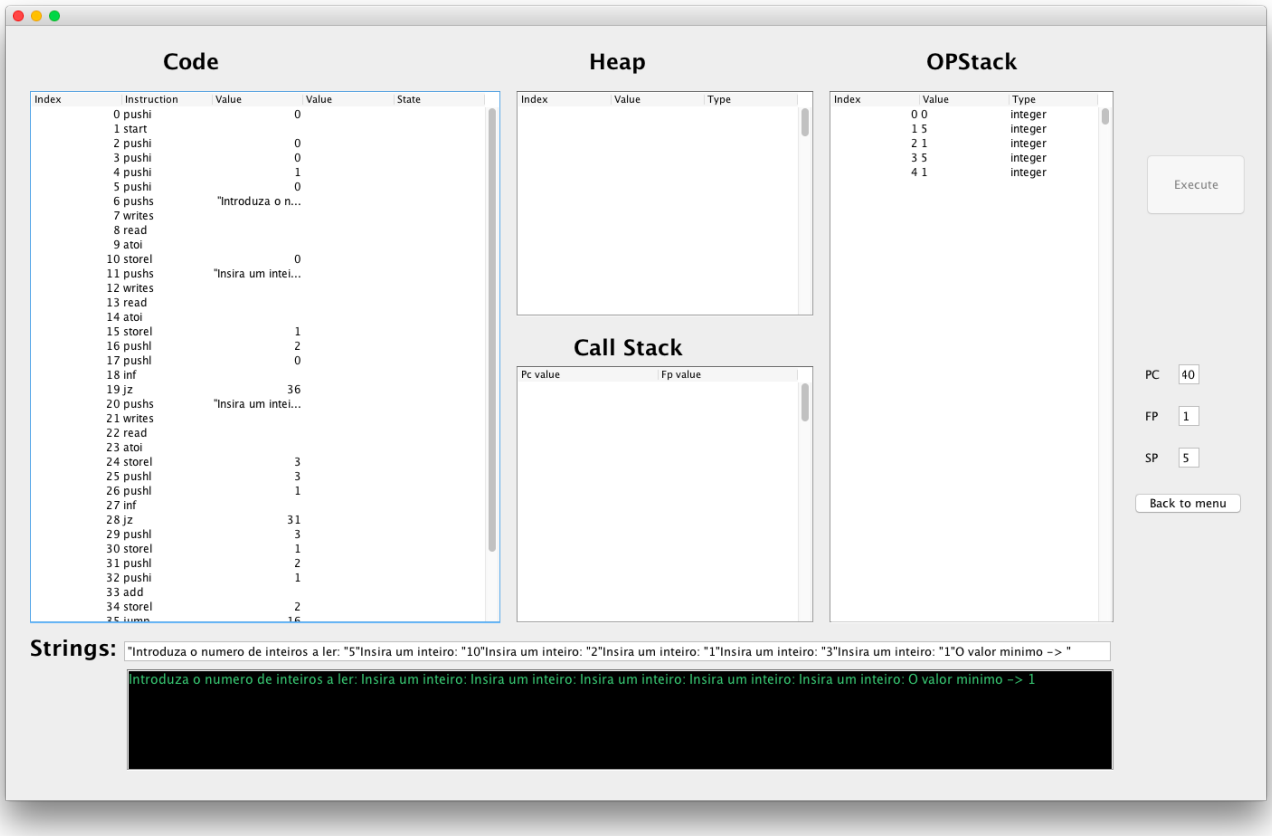

Figura 7.2: Resultado do Teste 2

7.3 Teste 3

O programa lê N (constante do programa) números e calcula e imprime o seu produtório.

7.3.1 Programa em Atomic

```
start(){
    // ler N (constante do programa) números e calcular e imprimir o seu produtorio

    atom N <- 5;
    atom produtorio <- 1;
    atom lidos <- 0;
    atom lido;

    while (lidos < N){
```

```
        print("Insira um inteiro:");
        lido <- read();
        produtorio <- produtorio * lido;
        lidos <- lidos + 1;
    }

    print ("Produtorio: ");
    print (produtorio);
}
```

7.3.2 Resultado

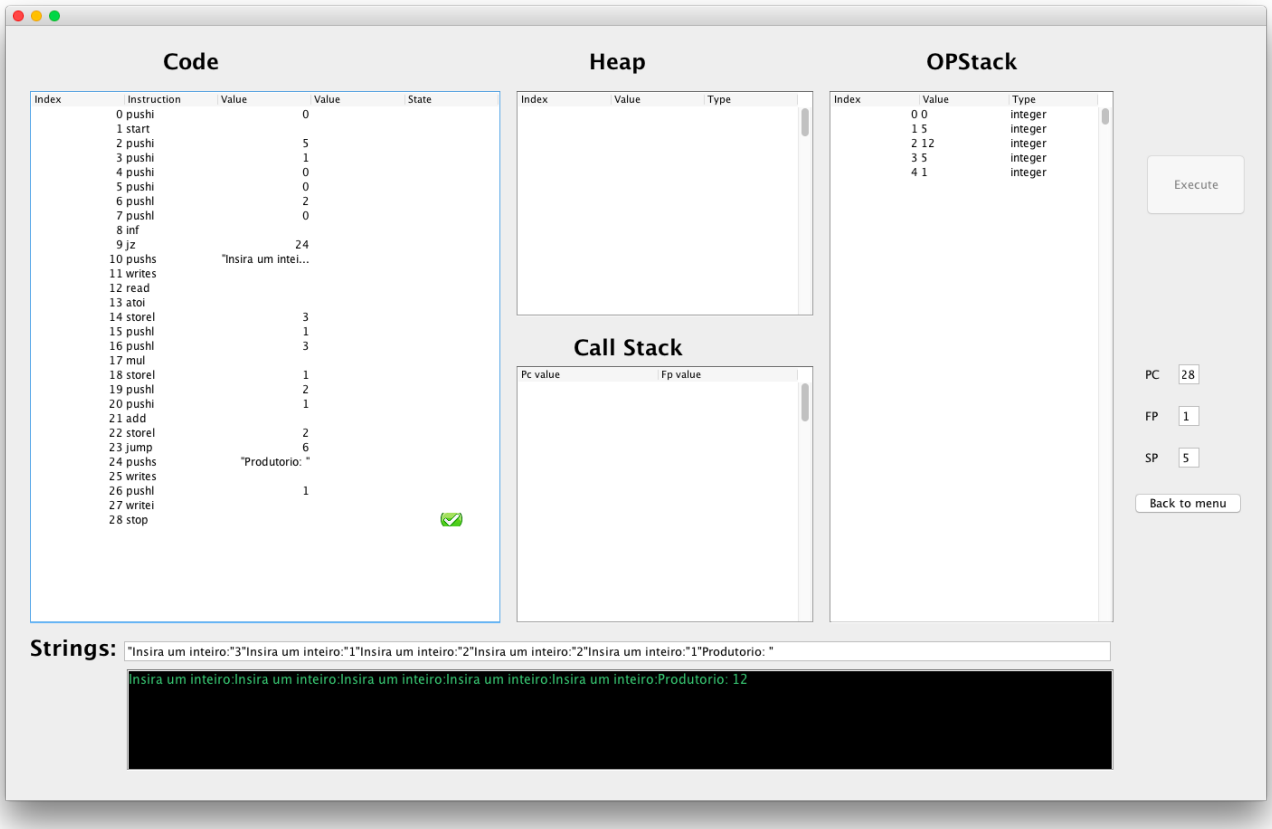

Figura 7.3: Resultado do Teste 3

7.4 Teste 4

O programa conta e imprime os números ímpares de uma sequência de números naturais.

7.4.1 Programa em Atomic

```
start(){
    // Contar e imprimir os numeros impares de uma sequencia de naturais
    atom lista[5];
    atom lidos <- 0;
    atom impares <- 0;

    lista[0] <- 1;
    lista[1] <- 2;
    lista[2] <- 3;
    lista[3] <- 4;
    lista[4] <- 5;

    while (lidos < 5){
        if(lista[lidos] % 2 == 1) {
            impares <- impares + 1;
            print(lista[lidos]);
        }
        lidos <- lidos + 1;
    }

    print("Total impares: ");
    print(impares);
}
```

7.4.2 Resultado

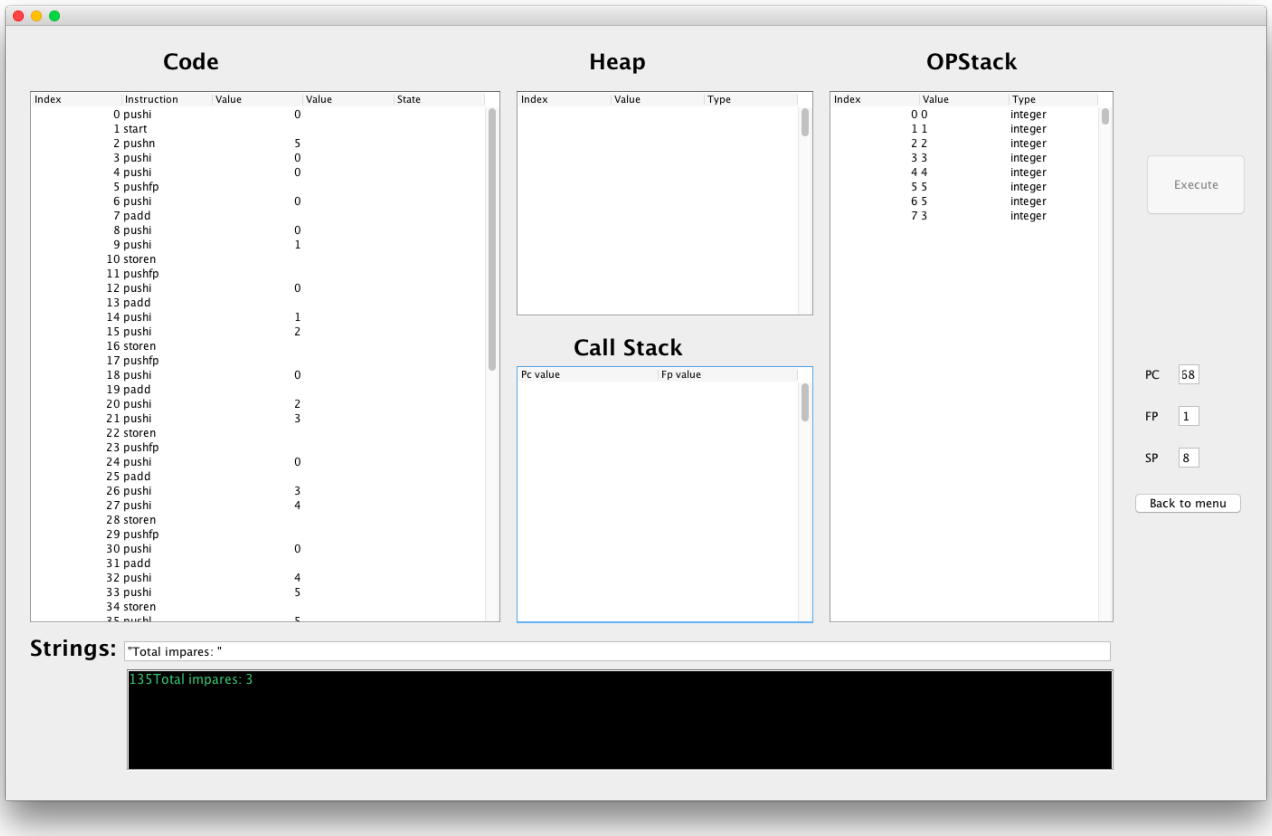

Figura 7.4: Resultado do Teste 4

7.5 Teste 5

O programa lê e armazena os elementos de um vetor de comprimento N; imprime os valores por ordem decrescente após fazer a ordenação do array por trocas diretas.

7.5.1 Programa em Atomic

```
start(){
    //  ler e armazenar os elementos de um vetor de comprimento N; imprimir os valores por

    atom lista[5];
    atom num;
    atom c, d, cond;
    atom lidos <- 0;
    atom total <- 5;
```

```
    atom tmp;

    while (lidos < total){
        print("Insira um inteiro: ");
        num <- read();
        lista[lidos] <- num;
        lidos <- lidos + 1;
    }

    lidos <- 0;

    while (lidos < total){
        cond <- total - lidos;
        cond <- cond - 1;
        d <- 0;

        while (d < cond) {
            if (lista[d] > lista[d+1]){
                tmp <- lista[d];
                lista[d] <- lista[d+1];
                lista[d+1] <- tmp;
            }
            d <- d + 1;
        }
        lidos <- lidos + 1;
    }

    lidos <- 0;
    print("| ");
    while (lidos < 5){
        print(lista[lidos]);
        lidos <- lidos + 1;
        print(" | ");
    }
}
```

7.5.2 Resultado

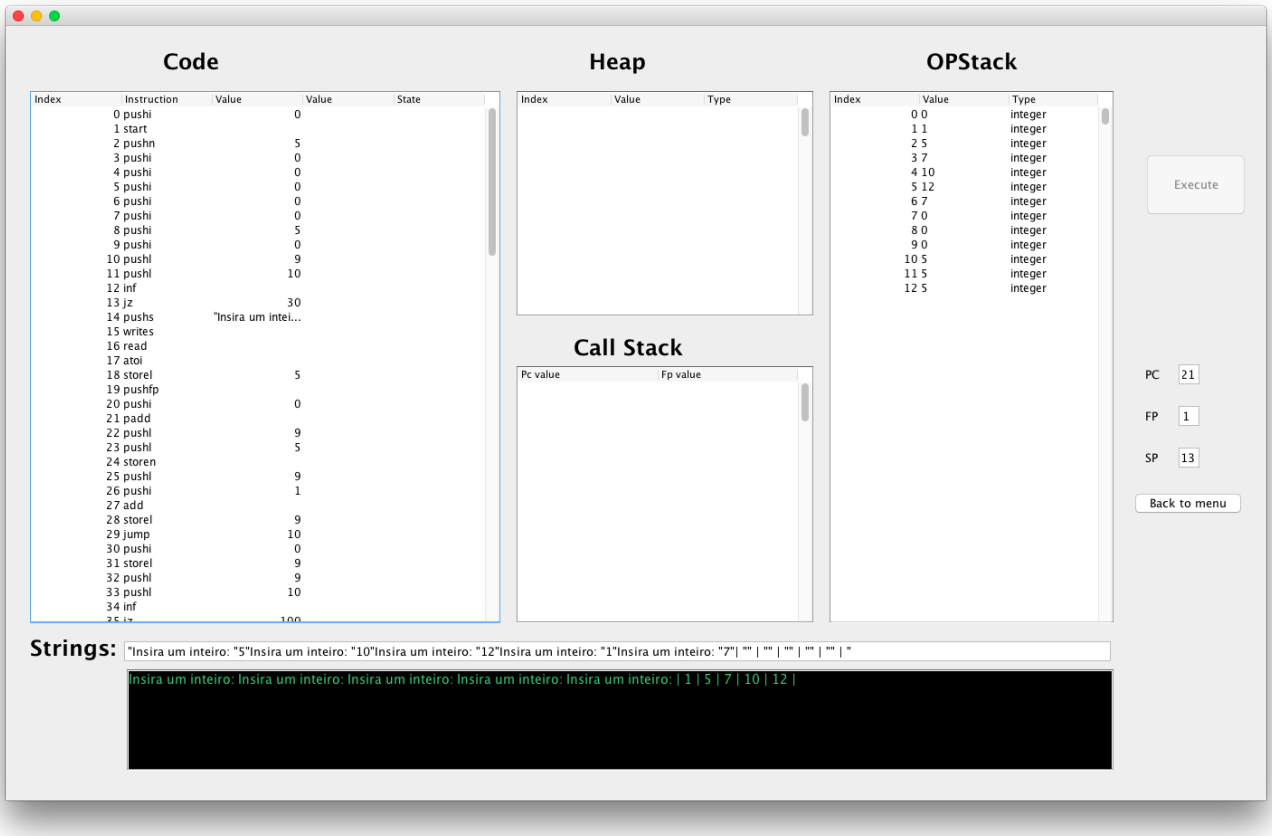

Figura 7.5: Resultado do Teste 5

7.6 Teste 6

O programa lê e armazena N números num array; imprime os valores por ordem inversa.

7.6.1 Programa em Atomic

```
start(){
    // ler e armazenar N numeros num array; imprimir os valores por ordem inversa

    atom lista[5];
    atom num;
    atom lidos <- 0;

    while (lidos < 5){
        print("Insira um inteiro: ");
```

```
        num <- read();
        lista[lidos] <- num;
        lidos <- lidos + 1;
    }

    lidos <- lidos - 1;
    print("| ");
    while (lidos >= 0){
        print(lista[lidos]);
        lidos <- lidos - 1;
    print(" | ");
    }
}
```

7.6.2 Resultado

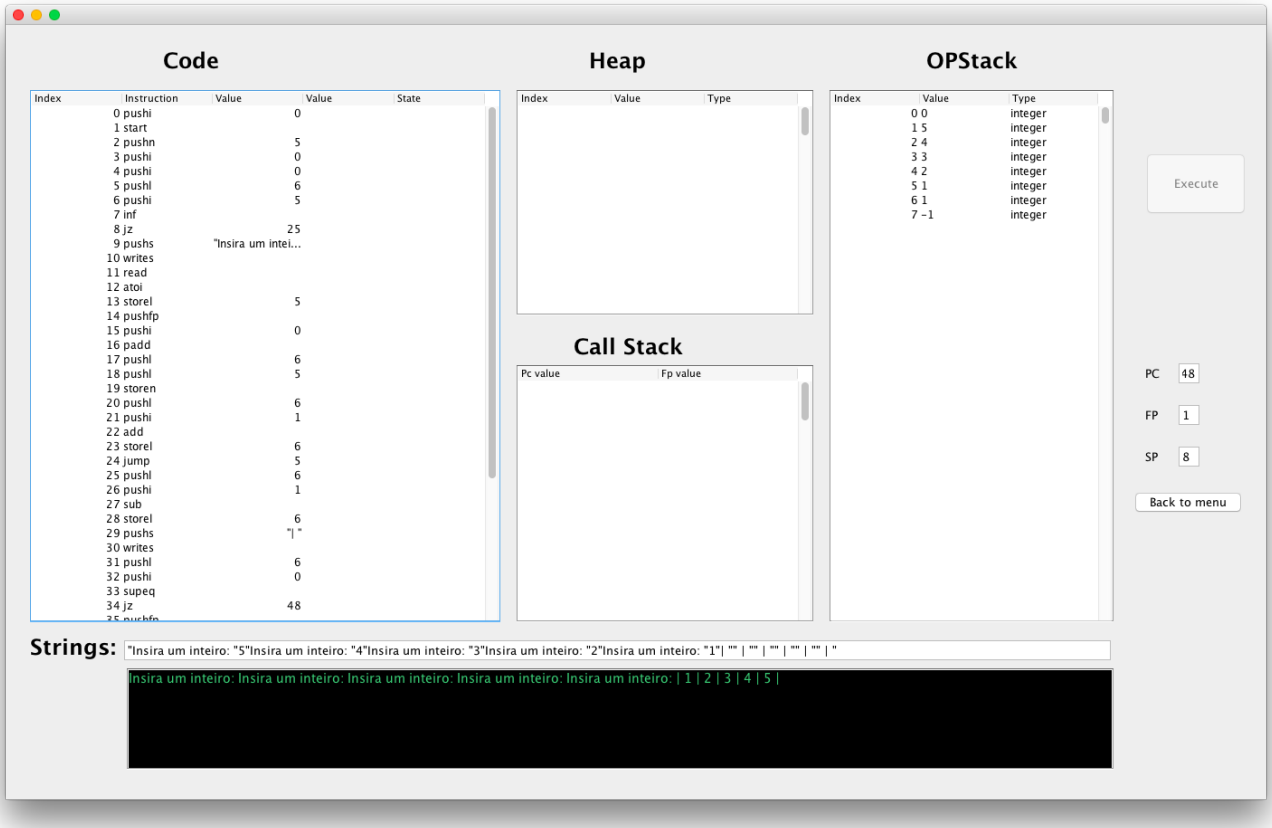

Figura 7.6: Resultado do Teste 6

7.7 Teste 7

Igual ao teste 1, mas chama duas funções auxiliares.

7.7.1 Programa em Atomic

```
void mensagemEntrada() {
    print("Vamos testar lados de um quadrado!");
}

atom lerNumero() {
    atom num;

    print("Insira numero:");
    num <- read();

    return num;
}

start(){
    // ler 4 numeros e dizer se podem ser os lados de um quadrado

    atom l1;
    atom l2;
    atom l3;
    atom l4;

    mensagemEntrada();

    l1 <- lerNumero();
    l2 <- lerNumero();
    l3 <- lerNumero();
    l4 <- lerNumero();

    if (l1 == l2) {
        if (l1 == l3) {
            if (l1 == l4) {
                print("Pode ser um quadrado!");
            }
            else {
                print("Não pode ser um quadrado!");
            }
        }
        else {
            print("Não pode ser um quadrado!");
```

```
        }
    }
    else {
        print("Não pode ser um quadrado!");
    }
}
```

7.7.2 Resultado

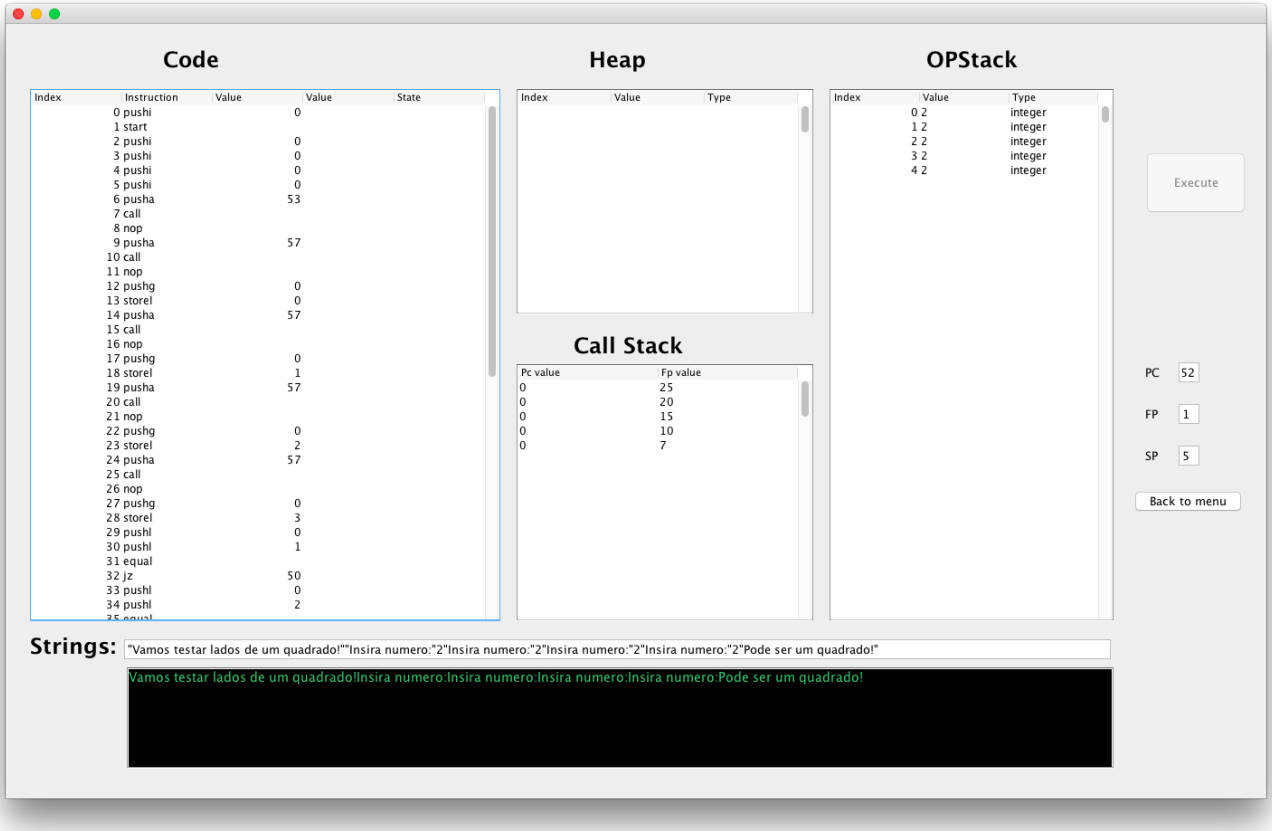

Figura 7.7: Resultado do Teste 7

7.8 Teste 8

Programa que apenas testa os operadores lógicos `&&` e `||` com variáveis e resultados de operações aritméticas.

7.8.1 Programa em Atomic

```
start() {
    atom x <- 1;
    atom y <- 2;
    atom z <- 2;

    if (x==1 && y==(z/x)) {
        print("| X é igual a 1 e Y é igual a Z a dividir por X |");
        if( x == y*z || y == x*z) {
            print("| X é igual à multiplicação de Y por Z, ou, Y é igual à multiplicação a
        }
    }
}
```

7.8.2 Resultado

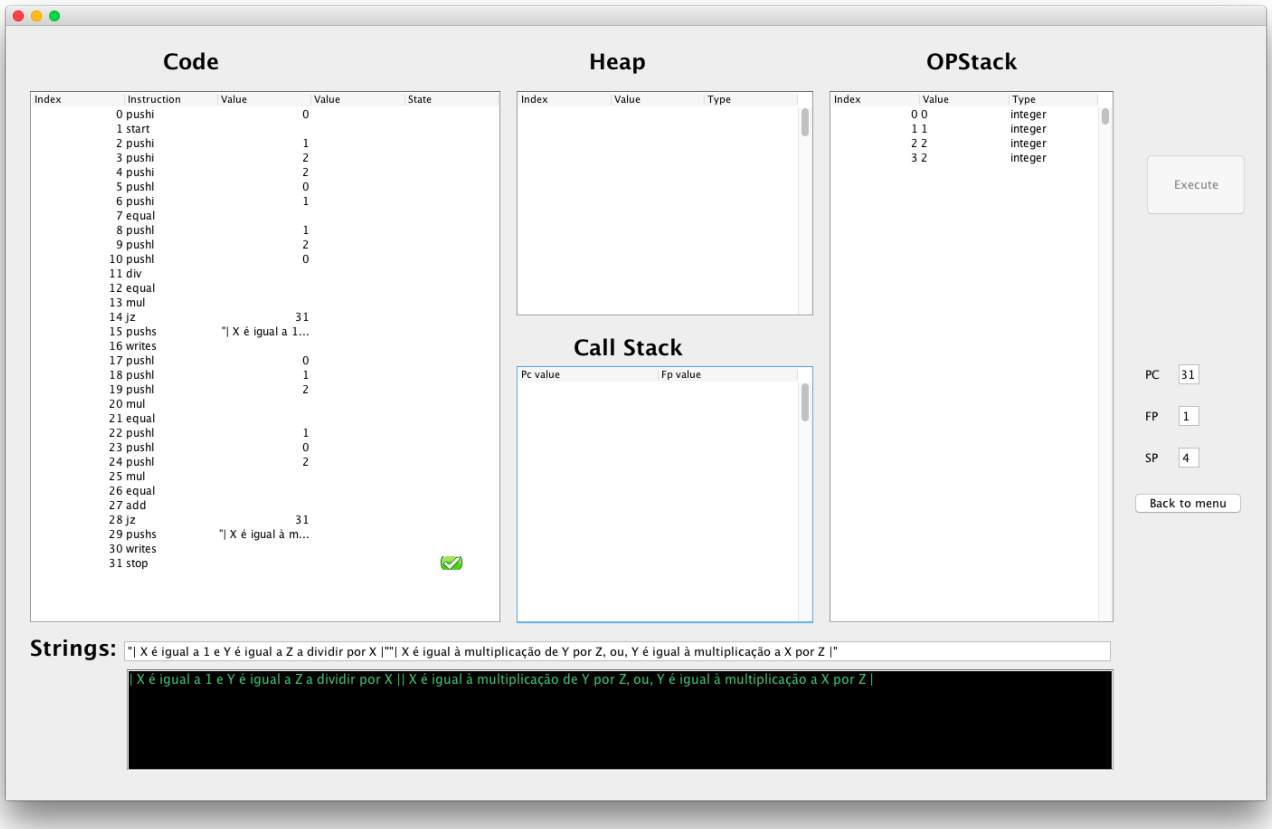

Figura 7.8: Resultado do Teste 8

7.9 Teste 9

Programa que apenas testa se o operador de negação e o operador de diferença funcionam.

7.9.1 Programa em Atomic

```
start() {
    if(! 5!=5 ) {
        print("5 é igual a 5");
    }
    else {
        print("5 não é igual a 5");
    }
}
```

7.9.2 Resultado

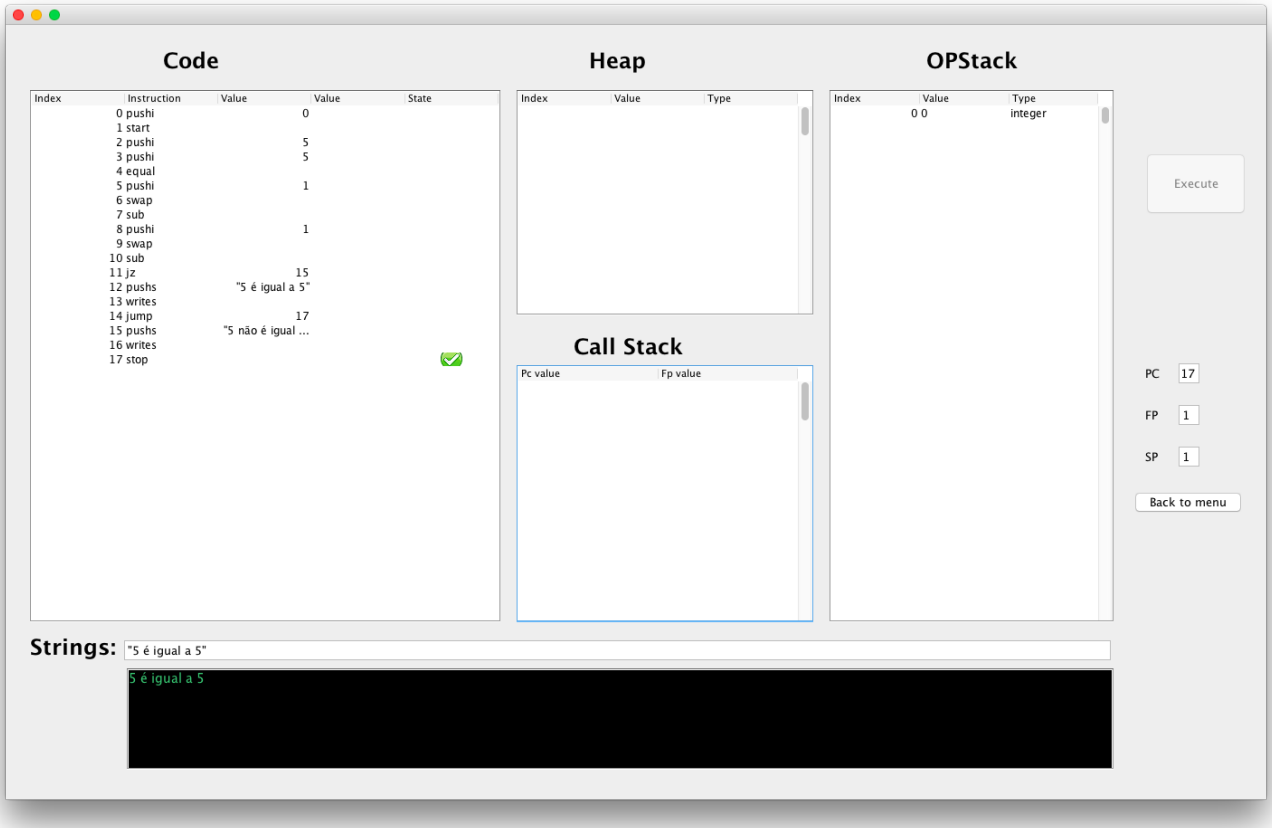

Figura 7.9: Resultado do Teste 9

Capítulo 8

Conclusão

A elaboração do trabalho sobre `Flex` e `Yacc` permitiu aplicar os conhecimentos lecionados nas aulas teóricas e práticas desta Unidade Curricular, culminando na especificação de uma gramática independente de contexto e gramática tradutora, e na implementação de um compilador para a linguagem Atomic.

A realização deste trabalho também permitiu:

- aumentar a experiência em engenharia de linguagens e em programação generativa (gramatical);
- desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe, suportado numa gramática tradutora;
- desenvolver um compilador gerando código para uma máquina de stack virtual;
- utilizar geradores de compiladores baseados em gramáticas tradutoras, concretamente o `Yacc`.

A linguagem de programação Atomic foi especificada e documentada, bem como o desenvolvimento de um processador de linguagens e um compilador, gerando código para uma máquina de stack virtual.

O foco central da resolução deste trabalho prático foi a consolidação dos conhecimentos adquiridos sobre `Flex` e `Yacc`, logo conclui-se que os objetivos foram plenamente cumpridos, com a implementação de todas as funcionalidades propostas na linguagem Atomic, bem como casos de demonstração.

Bibliografia

[1] Saraiva, J. A. 1995. *Especificação e Processamento de Linguagens* Universidade do Minho

[2] Ramalho, J. C. 2001. *Lex: Um Gerador de analisadores léxicos para linguagens regulares* Universidade do Minho

[3] Lesk, M. E. and Schmidt, E. 1975. *Lex - A Lexical Analyzer Generator*

[4] 1995. *Flex Documentation, version 2.5*

[5] *GLib Reference Manual* Open Source Software Foundation

Appendices

Apêndice A

Filtros de Texto

A.1 Analisador Sintático FLex

```
%{
        #include <string.h>
%}

%option noyywrap
%option yylineno

%%

[-*/;,\[\]=><!+\(\)\{\}%]        { return yytext[0]; }
[-]?[0-9]+                                          {
                                                              yylval.valor = atoi(yytext
                                                              return NUM;
                                                    }

(?i:void)                                          { return VOID; }
(?i:start)                                          { return START; }
(?i:if)                                                  { return IF; }
(?i:else)                                          { return ELSE; }
(?i:while)                                          { return WHILE; }
(?i:atom)                                          { return ATOM; }
(?i:read)                                          { return READ; }
(?i:print)                                          { return PRINT; }
(?i:return)                                          { return RETURN; }
\<-                                                          { return AT; }
\<=                                                          { return LEQ; }
\>=                                                          { return GEQ; }
!=                                                          { return DIF; }
==                                                          { return EQU; }
```

```
&&                                                  { return AND; }
\|\|                                              { return OR; }

[_a-zA-Z][_A-Za-zs0-9]*                 {
                                                            yylval.str = strdup(yytext
                                                            return ID;
                                                    }

\"[^"]*\"                                       {
                                                            yylval.str = strdup(yytext
                                                            return STRING;
                                                    }

\/\/[^\n]*                              { }
([ ]|\n|\t|\r)                                  { }
.                                                   { fprintf(stderr, "Erro char invalido

%%
```

Apêndice B

Código gerado Assembly

B.1 Teste 1 - Código gerado Assembly

```
pushi 0
start
pushi 0
pushi 0
pushi 0
pushi 0
pushs "Insira␣primeiro␣numero:"
writes
read
atoi
storel 0
pushs "Insira␣segundo␣numero:"
writes
read
atoi
storel 1
pushs "Insira␣terceiro␣numero:"
writes
read
atoi
storel 2
pushs "Insira␣quarto␣numero:"
writes
read
atoi
storel 3
pushl 0
pushl 1
equal
```

```
jz else_0
pushl 0
pushl 2
equal
jz else_0
pushl 0
pushl 3
equal
jz else_0
pushs "Pode␣ser␣um␣quadrado!"
writes
jump if_fim_0
else_0:
pushs "Nao␣pode␣ser␣um␣quadrado!"
writes
if_fim_0:
jump if_fim_0
else_0:
pushs "Nao␣pode␣ser␣um␣quadrado!"
writes
if_fim_0:
jump if_fim_0
else_0:
pushs "Nao␣pode␣ser␣um␣quadrado!"
writes
if_fim_0:
stop
```

B.2 Teste 2 - Código gerado Assembly

```
pushi 0
start
pushi 0
pushi 0
pushi 1
pushi 0
pushs "Introduza␣o␣numero␣de␣inteiros␣a␣ler:␣"
writes
read
atoi
storel 0
pushs "Insira␣um␣inteiro:␣"
writes
```

```
read
atoi
storel 1
while_0:
pushl 2
pushl 0
inf
jz while_fim_0
pushs "Insira␣um␣inteiro:␣"
writes
read
atoi
storel 3
pushl 3
pushl 1
inf
jz if_fim_0

pushl 3
storel 1
if_fim_0:

pushl 2
pushi 1
add

storel 2
jump while_0
while_fim_0:
pushs "O␣valor␣minimo␣->␣"
writes
pushl 1
writei
stop
```

B.3 Teste 3 - Código gerado Assembly

```
pushi 0
start
pushi 5
pushi 1
pushi 0
pushi 0
```

```
while_0:
pushl 2
pushl 0
inf
jz while_fim_0
pushs "Insira␣um␣inteiro:"
writes
read
atoi
storel 3

pushl 1
pushl 3
mul

storel 1

pushl 2
pushi 1
add

storel 2
jump while_0
while_fim_0:
pushs "Produtorio:␣"
writes
pushl 1
writei
stop
```

B.4 Teste 4 - Código gerado Assembly

```
pushi 0
start
pushn 5
pushi 0
pushi 0
pushfp
pushi 0
padd
pushi 0
pushi 1
storen
```

```
pushfp
pushi 0
padd
pushi 1
pushi 2
storen
pushfp
pushi 0
padd
pushi 2
pushi 3
storen
pushfp
pushi 0
padd
pushi 3
pushi 4
storen
pushfp
pushi 0
padd
pushi 4
pushi 5
storen
while_0:
pushl 5
pushi 5
inf
jz while_fim_0
pushfp
pushi 0
padd
pushl 5loadn
pushi 2
mod

pushi 1
equal
jz if_fim_0

pushl 6
pushi 1
add

storel 6
```

```
pushfp
pushi 0
padd
pushl 5loadn
writei
if_fim_0:

pushl 5
pushi 1
add

storel 5
jump while_0
while_fim_0:
pushs "Total␣impares:␣"
writes
pushl 6
writei
stop
```

B.5 Teste 5 - Código gerado Assembly

```
pushi 0
start
pushn 5
pushi 0
pushi 0
pushi 0
pushi 0
pushi 0
pushi 5
pushi 0
while_0:
pushl 9
pushl 10
inf
jz while_fim_0
pushs "Insira␣um␣inteiro:␣"
writes
read
atoi
storel 5
pushfp
```

```
pushi 0
padd
pushl 9
pushl 5
storen

pushl 9
pushi 1
add

storel 9
jump while_0
while_fim_0:

pushi 0
storel 9
while_2:
pushl 9
pushl 10
inf
jz while_fim_2

pushl 10
pushl 9
sub

storel 8

pushl 8
pushi 1
sub

storel 8

pushi 0
storel 7
while_1:
pushl 7
pushl 8
inf
jz while_fim_1
pushfp
pushi 0
padd
pushl 7loadn
```

```
pushfp
pushi 0
padd
pushl 7
pushi 1
add
loadn
sup
jz if_fim_0

pushfp
pushi 0
padd
pushl 7loadn
storel 11
pushfp
pushi 0
padd
pushl 7
pushfp
pushi 0
padd
pushl 7
pushi 1
add
loadn
storen
pushfp
pushi 0
padd
pushl 7
pushi 1
add

pushl 11
storen
if_fim_0:

pushl 7
pushi 1
add

storel 7
jump while_1
while_fim_1:
```

```
pushl 9
pushi 1
add

storel 9
jump while_2
while_fim_2:

pushi 0
storel 9
pushs "|␣"
writes
while_3:
pushl 9
pushi 5
inf
jz while_fim_3
pushfp
pushi 0
padd
pushl 9loadn
writei

pushl 9
pushi 1
add

storel 9
pushs "␣|␣"
writes
jump while_3
while_fim_3:
stop
```

B.6 Teste 6 - Código gerado Assembly

```
pushi 0
start
pushn 5
pushi 0
pushi 0
while_0:
```

```
pushl 6
pushi 5
inf
jz while_fim_0
pushs "Insira␣um␣inteiro:␣"
writes
read
atoi
storel 5
pushfp
pushi 0
padd
pushl 6
pushl 5
storen

pushl 6
pushi 1
add

storel 6
jump while_0
while_fim_0:

pushl 6
pushi 1
sub

storel 6
pushs "|␣"
writes
while_1:
pushl 6
pushi 0
supeq
jz while_fim_1
pushfp
pushi 0
padd
pushl 6loadn
writei

pushl 6
pushi 1
sub
```

```
storel 6
pushs "␣|␣"
writes
jump while_1
while_fim_1:
stop
```

B.7 Teste 7 - Código gerado Assembly

```
pushi 0
start
pushi 0
pushi 0
pushi 0
pushi 0
pusha funcao_mensagemEntrada
call
nop

pusha funcao_lerNumero
call
nop
pushg 0

storel 0

pusha funcao_lerNumero
call
nop
pushg 0

storel 1

pusha funcao_lerNumero
call
nop
pushg 0

storel 2

pusha funcao_lerNumero
call
```

```
nop
pushg 0

storel 3
pushl 0
pushl 1
equal
jz else_0
pushl 0
pushl 2
equal
jz else_0
pushl 0
pushl 3
equal
jz else_0
pushs "Pode␣ser␣um␣quadrado!"
writes
jump if_fim_0
else_0:
pushs "Nao␣pode␣ser␣um␣quadrado!"
writes
if_fim_0:
jump if_fim_0
else_0:
pushs "Nao␣pode␣ser␣um␣quadrado!"
writes
if_fim_0:
jump if_fim_0
else_0:
pushs "Nao␣pode␣ser␣um␣quadrado!"
writes
if_fim_0:
stop
funcao_mensagemEntrada:
nop
pushs "Vamos␣testar␣lados␣de␣um␣quadrado!"
writes
return
funcao_lerNumero:
nop
pushi 0
pushs "Insira␣numero:"
writes
read
```

```
atoi
storel 0
pushl 0
storeg 0
return
```

B.8 Teste 8 - Código gerado Assembly

```
pushi 0
start
pushi 1
pushi 2
pushi 2
pushl 0
pushi 1
equal

pushl 1
pushl 2
pushl 0
div

equal

mul
jz if_fim_1
pushs "|␣X␣e'␣igual␣a␣1␣e␣Y␣e'␣igual␣a␣Z␣a␣dividir␣por␣X␣|"
writes
pushl 0
pushl 1
pushl 2
mul

equal

pushl 1
pushl 0
pushl 2
mul

equal

add
```

```
jz if_fim_0
pushs "|␣X␣e'␣igual␣'a␣multiplicacao␣de␣Y␣por␣Z,␣ou,␣Y␣e'␣igual␣'a␣multipli
writes
if_fim_0:
if_fim_1:
stop
```

B.9 Teste 9 - Código gerado Assembly

```
pushi 0
start
pushi 5
pushi 5
equal
pushi 1
swap
sub

pushi 1
swap
sub
jz else_0
pushs "5␣e'␣igual␣a␣5"
writes
jump if_fim_0
else_0:
pushs "5␣nao␣e'␣igual␣a␣5"
writes
if_fim_0:
stop
```