2009.2

Relatório de Projeto

Especificação da Linguagem de Programação HELL

Relatório do projeto da Disciplina Paradigmas de Linguagens Computacionais (if686), que consiste na implementação de um interpretador em Haskell, e na descrição semântica de uma linguagem imperativa, cuja sintaxe abstrata foi dada.

Autores:

Josiane Bezerra Ferreira – jbf2

Luís Pedro de Medeiros Filho – lpmf

Luiz Felipe da Silva Sotero – lfss

Paulo Sérgio Borges de Oliveira Filho – psbof

Centro de Informática - CIn

Universidade Federal de Pernambuco

19/11/2009

Sumário

1. Introdução ... 6
2. Descrição Informal ... 7
2.1. Estrutura do Programa ... 7
2.1.1. Programa ... 7
2.1.2. Comando ... 7
2.2. Atribuições ... 7
2.3. Expressões ... 8
2.3.1. Expressão ... 8
2.3.2. Valor ... 8
2.3.3. Expressão Unária ... 9
2.3.4. Expressão Binária ... 9
2.4. Declarações ... 10
2.4.1. Declaração ... 10
2.4.2. Declaração de Variáveis ... 10
2.4.3. Declaração de Procedimentos ... 10
2.4.4. Lista de Parâmetros ... 11
2.4.5. Comando Declaração ... 11
2.4.6. Tipos ... 11
2.5. Controle de Fluxo – Estruturas de Repetição ... 12
2.5.1. Laço While ... 12
2.5.2. Laço For ... 12
2.5.3. Laço Repeat ... 12
2.6. Controle de Fluxo – Comandos de Decisão ... 13
2.6.1. If Then Else ... 13
2.7. Procedimentos ... 13
2.7.1. Chamada de Procedimentos ... 13
2.7.2. Lista de Expressões ... 13
2.8. Comandos de Entrada e Saída ... 14
2.8.1. O Comando IO ... 14
2.9. O Comando Skip ... 14
3. Sintaxe Abstrata ... 15
3.1. Programas ... 15

3.2. Comandos 15
3.3. Atribuições 15
3.4. Expressões 16
3.5. Declarações 16
3.6. Tipos 17
3.7. Comandos de Controle de Fluxo 17
3.8. Comandos de Entrada e Saída 17
3.9. Chamadas de Procedimentos 17
4. Entidades Semânticas 18
4.1. Dados 18
4.2. Valores 18
4.3. Tipos 18
4.4. Bindings 18
4.5. Armazenamento 19
4.6. Procedimentos 19
5. Funções Semânticas 20
5.1. Programa 20
5.2. Declarações 20
5.3. Expressões 21
5.4. Comandos 22
6. Sintaxe Léxica 25
7. O Interpretador 26
7.1. Definição 26
7.2. Arquitetura 26
7.3. Comportamento 28
7.3.1. Declarações 28
7.3.2. Atribuições 28
7.3.3. Estruturas de Repetição 28
7.3.4. If-Then-Else 29
7.3.5. Chamadas de Procedimentos 29
7.3.6. Entrada e Saída 29
7.4. Implementação 29
8. Exemplos de Programas 31
8.1. Declaração e Atribuição 31

8.2. Procedimentos, Chamadas de Procedimentos, Loop While, Saida de Dados 32
8.3. Loop While 34
8.4. Loop Repeat 36
8.5. Entrada e Saída de Dados, Procedimento Recursivo, If Then Else 38
8.6. Ponteiros, Saída de Dados, Operações com Strings 41
8.7. Loop For 43
8.8. Comando If Then Else 45
8.9. Entrada de Dados 46
8.10. Saída de Dados 47
8.11. Tratamento de Erro 48
8.12. Atribuições a Conteúdos de Ponteiros 49
8.13. Loop While com Demonstração de Escopo 51
9. Conclusões 54
10. Referências 55

Índice de Ilustrações

Figura 1 - Arquitetura do Interpretador 26
Figura 2 - Camada Ambiente 27
Figura 3 – Módulos e Camadas 30
Figura 4 - Hierarquia do Código 30

1. Introdução

O uso de semântica formal para especificar linguagens de programação permite a descrição comportamental de uma linguagem em um formato bem definido, evitando erros de interpretação e tornando-se uma boa base para a implementação.

Entre os formalismos utilizados para esse fim, a semântica de ações, *framework* desenvolvido para tornar mais compreensível e acessível a formalização de linguagens de programação, define um conjunto padrão de operadores que descrevem conceitos comuns encontrados em linguagem de programação. Dessa forma a especificação da linguagem é simplificada, visto que o projetista não necessita manipular definições complexas para a descrição formal.

As especificações em semântica de ações são semelhantes à semântica denotacional. Sendo o uso de ações mais operacional, uma vez que com ações se processa a informação gradualmente. Como a semântica denotacional, a semântica de ações é composicional, permitindo que "regras" definidas em uma parte da especificação sejam utilizadas em outra parte para compor uma nova regra.

As partes principais da descrição da semântica de ações de uma linguagem são a sintaxe abstrata, as funções semânticas e as entidades semânticas.

A sintaxe abstrata descreve apenas os elementos relevantes de uma linguagem de programação. Deve fornecer a BNF (Backus-Naur Form) da mesma, mostrando as possíveis expressões e os possíveis comandos, através de uma série de produções utilizando símbolos terminais e não terminais da gramática da linguagem, similar a sintaxe concreta que é definida em termos de expressões regulares de gramáticas livres de contexto.

Funções semânticas são equações semânticas que se aplicam sobre as estruturas da linguagem e lhes dão significado. Elas mapeiam as categorias sintáticas da linguagem em domínios semânticos escolhidos. Entidades semânticas são utilizadas para representar a partes da linguagem não dependentes de implementação.

Esse trabalho define a semântica de ações para a *Highly Extravagant Logical Language* (*HELL*), uma linguagem imperativa inspirada em C e Pascal.

O documento está estruturado em seções, de forma a facilitar a compreensão por parte do leitor, da seguinte forma: Primeiro é apresentada uma descrição informal da linguagem, no intuito de oferecer uma visão geral sobre a mesma. Em seguida, é descrita a semântica formal, com base em semântica de ações, com o objetivo de permitir uma compreensão clara, sem ambigüidades, da linguagem aqui descrita, e de tornar esse um documento preciso que possa ser usado como base para a implementação da linguagem HELL. Após isso, é descrito o interpretador para a linguagem, que foi desenvolvido pelos autores desse documento. A seguir são mostrados exemplos de programas escritos em linguagem HELL, que ajudam a facilitar ainda mais a compreensão a cerca dessa linguagem.

2. Descrição Informal

2.1. Estrutura do Programa

2.1.1. Programa

- **Sintaxe:**

Programa ::= Comando

- **Semântica:**

Um programa é definido por um comando. Um comando, por sua vez, consiste no corpo do programa propriamente dito. A execução do programa consiste na execução de instruções presentes no comando.

2.1.2. Comando

- **Sintaxe:**

```
Comando ::= ComandoDeclaracao
          | Atribuicao
          | While
          | For
          | IfThenElse
          | RepeatUntil
          | IO
          | Skip
          | ChamadaProcedimento
          | Comando ";" Comando
```

- **Semântica:**

Comandos são usados para descrever as operações realizadas por um programa. A linguagem HELL define um conjunto de 10 comandos, que permitem declarações de variáveis e procedimentos, atribuições, comandos de controle de fluxo, rotinas de entrada e saída, e chamadas de procedimentos, os quais serão detalhados em seguida.

2.2. Atribuições

- **Sintaxe:**

```
Atribuicao ::= Id ":=" Expressao
             |"*"Id ":=" Expressao
```

- **Semântica:**

Atribui a uma variável previamente declarada, representada pelo seu identificador, o valor resultante de uma expressão. Isso é feito armazenando o valor da expressão no endereço de memória referenciado pelo identificador da variável.

Sendo a HELL uma linguagem fortemente tipada, verificações serão feitas de modo a não permitir que um valor de um determinado tipo, que não seja o mesmo tipo da variável, seja atribuído à mesma.

Também pode-se atribuir valores ao conteúdo de um ponteiro. Essa funcionalidade foi adicionada à sintaxe original com o objetivo de dar maior liberdade ao programador.

2.3. Expressões

2.3.1. Expressão

- **Sintaxe:**

```
Expressao ::= Valor
            | ExpUnaria
            | ExpBinaria
            | Id
            | "&"Id
            | "*"Id
```

- **Semântica:**

Uma expressão é uma sentença usada para representar desde simples valores até estruturas mais complexas, com vários termos.

A avaliação de uma expressão retorna um valor, que pode ser de qualquer um dos tipos básicos da linguagem (inteiro, booleano, string).

Seis expressões são definidas na linguagem HELL: valor, expressão unária, expressão binária e os identificadores simples, de endereço ou de conteúdo.

O valor da expressão Id – identificador – é o conteúdo da variável representada por esse identificador.

No caso de uma expressão "&"Id – identificador de endereço – o seu valor é o endereço de memória onde está armazenada a variável identificada por Id.

Já uma expressão do tipo "*"Id – identificador de conteúdo – tem como valor o conteúdo do endereço de memória representado pelo ponteiro Id.

As demais expressões serão definidas a seguir.

2.3.2. Valor

- **Sintaxe:**

```
Valor ::= ValorInteiro
        | ValorBooleano
        | ValorString
        | ValorNull
```

- **Semântica:**

Trata-se de uma expressão que representa o valor de uma determinada representação. Pode ser do tipo inteiro, booleano, string ou null. O valor null, usado quando se está trabalhando com ponteiros, indica que este não está referenciando nenhuma variável. Esse valor foi adicionado à sintaxe, justamente para permitir ao programador a inicialização de um ponteiro sem que esse esteja, necessariamente, apontando para uma variável.

2.3.3. Expressão Unária

- **Sintaxe:**

ExpUnaria ::= "-" Expressao
| "!" Expressao

- **Semântica:**

Uma expressão unária é definida quando um operador unário ("-" ou "!") é aplicado a uma expressão.

O operador "-" deve ser aplicado somente a uma expressão inteira. O resultado será o valor simétrico ao valor da expressão à qual está sendo aplicado o operador.

Por sua vez, o operador "!" deve ser aplicada somente a uma expressão booleana. O valor da operação será o oposto do valor da expressão à qual está sendo aplicado o operador,ou seja, se o valor passado for *true*, o resultado da expressão unária será *false*; porém, se o valor passado for *false*, então o resultado da expressão unária será *true*.

2.3.4. Expressão Binária

- **Sintaxe:**

ExpBinaria ::= Expressao "+" Expressao
| Expressao "-" Expressao
| Expressao "&&" Expressao
| Expressao "||" Expressao
| Expressao "==" Expressao
| Expressao "++" Expressao

- **Semântica:**

Uma expressão binária é definida quando um operador binário (“+”, “-”, “&&”, “||”, “==” ou “++”) é aplicado a duas expressões.

Essas expressões podem ser do tipo inteiro, booleano ou string, dependendo do operador:

- o operador "+" representa a soma entre duas expressões estritamente inteiras;
- o operador "-" representa a subtração entre duas expressões estritamente inteiras;
- o operador "&&" representa a operação de conjunção (and) entre duas expressões estritamente booleanas;
- o operador "||" representa a operação de disjunção (ou) entre duas expressões estritamente booleanas;

- o operador "==" resulta em um booleano e representa a comparação entre duas expressões que retornam valores do mesmo tipo;
- por último, o operador "++" representa a concatenação de duas expressões do tipo string, estritamente.

2.4. Declarações

2.4.1. Declaração

- **Sintaxe:**

Declaracao ::= DeclaracaoVariavel
| DeclaracaoProcedimento
| Declaracao "," Declaracao

- **Semântica:**

Um comando de declaração tem como sua principal função alocar espaço na memória para variáveis ou procedimentos.

É possível a declaração de um, ou de mais de um identificador em um mesmo comando, devendo os identificadores, no segundo caso, serem separados por vírgulas.

2.4.2. Declaração de Variáveis

- **Sintaxe:**

DeclaracaoVariavel ::= "var" Id "=" Expressao
| "pointer" Id "=" "^"Tipo

- **Semântica:**

A declaração de uma variável liga o identificador dessa variável a um novo espaço alocado na memória.

O comando *"var" Id "=" Expressao* declara uma variável com identificador *Id*, cujo tipo é inferido a partir do valor de *Expressao*. Ou seja, o tipo da variável é determinado dinamicamente.

O comando *pointer" Id "=" "^"Tipo* declara um ponteiro com identificador *Id*, cujo tipo é determinado explicitamente, de forma estática . O conteúdo de uma variável do tipo ponteiro será o endereço de uma variável do mesmo tipo.

2.4.3. Declaração de Procedimentos

- **Sintaxe:**

DeclaracaoProcedimento ::= "proc" Id "(" [ListaDeclaracaoParametro] ")" "{" Comando "}"

- **Semântica:**

Comando que define o escopo de um procedimento e associa esse procedimento a um identificador. A estrutura de um procedimento é constituída por um bloco de comandos que

possui seu próprio escopo, mas que também pode acessar o escopo global. As declarações de procedimentos podem ser recursivas.

Os parâmetros para um procedimento podem ser passados por valor ou por referência. No segundo caso um ponteiro deve ser passado como parâmetro.

É importante lembrar que um procedimento não retorna nenhum valor, portanto não pode ser passado como parâmetro para outro procedimento, nem pode ser atribuído a uma variável.

2.4.4. Lista de Parâmetros

- **Sintaxe**

ListaDeclaracaoParametro ::= Id ["^"]Tipo
| ListaDeclaracaoParametro "," ListaDeclaracaoParametro

- **Semântica:**

Define como devem ser declarados os parâmetros de um procedimento. Deve ser informado o identificador e o tipo de cada parâmetro. No caso de mais de um parâmetro, esses devem ser separados por vírgula. Um procedimento não pode ser declarado sem parâmetros.

Na declaração, a quantidade de parâmetros e os seus tipos são associados ao identificador do procedimento e salvos no escopo global do programa.

2.4.5. Comando Declaração

- **Sintaxe:**

ComandoDeclaracao ::= "{" Declaracao ";" Comando "}"

- **Semântica:**

Estabelece a criação de um bloco. Dentro do bloco, no início, estará a declaração de uma variável ou procedimento, em seguida, os demais comandos. Destacando que sempre, após uma declaração, deverá haver um novo comando.

Tanto as variáveis quanto os procedimentos definidos em um bloco só poderão ser utilizados dentro do referido bloco.

2.4.6. Tipos

- **Sintaxe:**

Tipo ::= "string"
| "int"
| "boolean"
|"ponteiro" Tipo

- **Semântica:**

A linguagem HELL permite o uso de 3 tipos primitivos, mais o tipo ponteiro. Os tipos primitivos são:

- string: definido como uma cadeia de caracteres;

- int: representa um sub-conjunto dos números inteiros;
- boolean: define valores para sentenças lógicas, sendo esses valores *true* ou *false*.

O tipo ponteiro é definido para armazenar o endereço de uma determinada posição de memória, onde está armazenada uma variável. Esse endereço é um valor inteiro. Uma variável do tipo ponteiro pode receber o endereço de uma variável de qualquer um dos tipos primitivos (string, int ou boolean), ou até o endereço de um outro ponteiro. Esse tipo foi adicionado a sintaxe com o intuito de possibilitar ao programador definir ponteiros recursivamente (ponteiros para ponteiros), sem aumentar a complexidade do código do interpretador.

2.5. Controle de Fluxo – Estruturas de Repetição

2.5.1. Laço While

- **Sintaxe:**

While ::= "while" Expressao "do" Comando

- **Semântica:**

O comando *while é* composto por uma expressão booleana e uma série de comandos. Enquanto essa expressão for verificada como verdadeira, os comandos são executados. Caso seja verificado que o valor dessa expressão é falso, o programa deixa de executar o loop e passa a executar o próximo comando.

2.5.2. Laço For

- **Sintaxe:**

For ::= "for" Atribuicao "to" ValorInteiro "do" Comando

- **Semântica:**

O *for* é outro comando de repetição. Tem em sua estrutura, além de uma série de comandos, um comando de atribuição de um inteiro a uma variável, e um valor limite para essa mesma variável. A execução é repetida até que o valor dessa variável, que é incrementado a cada iteração do loop, alcance o valor limite. Ocorrendo isso, o programa deixa de executar o for e passa a executar os demais comandos.

2.5.3. Laço Repeat

- **Sintaxe:**

RepeatUntil ::= “repeat” Comando “until” Expressao

- **Semântica:**

Também trata-se de um comando de repetição. É composto por uma série de comandos e uma expressão booleana.

O conjunto de comandos é executado, em seguida a expressão é avaliada, caso seja falsa, o conjunto de comandos é executado novamente. Esses dois últimos passos se repetem até que

o valor da expressão seja verdadeiro. Quando isso ocorre o loop é encerrado e os próximos comandos são executados.

2.6. Controle de Fluxo – Comandos de Decisão

2.6.1. If Then Else

- **Sintaxe:**

IfThenElse ::= "if" Expressao "then" Comando "else" Comando

- **Semântica:**

Utilizado quando se quer que uma série de comandos ocorra somente se certa condição for satisfeita. É composto por uma expressão booleana e dois conjuntos de comandos.

Quando o valor da expressão é verdadeiro, o primeiro conjunto de comandos é executado e o segundo é ignorado, caso contrário, o segundo conjunto de comandos é executado e o primeiro é ignorado.

2.7. Procedimentos

2.7.1. Chamada de Procedimentos

- **Sintaxe:**

ChamadaProcedimento ::= "call" Id "(" ListaExpressao ")"

- **Semântica:**

Executa o procedimento associado ao identificador Id, previamente declarado. Ao menos um parâmetro deve ser passado ao procedimento, podendo ser passada uma lista de parâmetros.

2.7.2. Lista de Expressões

- **Sintaxe:**

```
ListaExpressao ::= Expressao
                 | Expressao "," ListaExpressao
```

- **Semântica:**

Define como os parâmetros são passados em uma chamada de procedimento. Em cada chamada é passada uma lista de expressões. O valor de cada uma dessas expressões é associado a um parâmetro, em sequência, da lista de parâmetros do procedimento que está sendo chamado.

2.8. Comandos de Entrada e Saída

2.8.1. O Comando IO

- **Sintaxe:**

IO ::= "write" "(" Expressao ")"
 | "read" "(" Id ")"

- **Semântica:**

A linguagem HELL compreende dois comandos de entrada e saída:

- O comando *write* recebe uma expressão e escreve o seu valor no console.
- O comando *read* lê um valor do teclado, e armazena esse valor no endereço de memória referenciado por *Id*.

Para implementação desses comandos, supõe-se que os dispositivos de entrada e saída, console e teclado, estejam mapeados na memória nas posições 0 (zero) e 1 (um), respectivamente. Dessa forma, um comando *write (Expressao)*, armazena o valor de *Expressao* no endereço 0 (zero) da memória. Já um comando *read (Id)* lê o valor contido no endereço 1 (um) da memória, e armazena esse valor na variável identificada por *Id*.

2.9. O Comando Skip

- **Sintaxe:**

Skip ::= "skip"

- **Semântica:**

O comando skip não altera o estado do programa, uma vez que não causa nenhum processamento além da sua identificação, nem altera a memória. Oferece apenas um intervalo entre as execuções.

3. Sintaxe Abstrata

Para a linguagem aqui especificada, foi desenvolvido um interpretador, que por sua vez foi escrito usando a linguagem de programação funcional *Haskell*. Por esse motivo, a sintaxe abstrata aqui mostrada está escrita nessa linguagem, e apresenta alguns lexemas da mesma. Esses lexemas serão explicados conforme aparecerem no texto.

3.1. Programas

```
data Programa = PROGRAMA Comando
                deriving (Show, Eq)
```

A palavra reservada *data* é utilizada em Haskell para a definição de tipos algébricos, que são tipos definidos pelo programador. No caso do interpretador, esse recurso foi utilizado para representar as produções da gramática da linguagem HELL.

Em Haskell, quando um tipo é definido, algumas classes podem ser instanciadas diretamente através da palavra reservada *deriving*. Uma classe é o conjunto de tipos sobre os quais uma função é definida. Os tipos instanciados na classe *Show* são todos os tipos que podem ser convertidos para strings. Já a *equality class*, ou classe *Eq*, é o conjunto de tipos em que o operador "==" (operador de igualdade) é definido.

3.2. Comandos

```
data Comando =  C_DECLARACAO DeclaracaoComando
                | C_SKIP
                | C_IO IO
                | C_ATRIBUICAO Atribuicao
                | C_WHILE While
                | C_FOR For
                | C_IF_THEN_ELSE IfThenElse
                | C_REPEAT_UNTIL RepeatUntil
                | C_CHAMADA_PROCEDIMENTO ChamadaProcedimento
                | C_COMANDO_MULT Comando Comando
                deriving (Show, Eq)
```

3.3. Atribuições

```
data Atribuicao = ATRIBUICAO Id Expressao
                | ATRIBUICAO_CONTEUDO Id Expressao
                deriving (Show, Eq)
```

3.4. Expressões

```
data Expressao = VALOR Valor
               | EXP_UNARIA ExpUnaria
               | EXP_BINARIA ExpBinaria
               | ID Id
               | ENDERECO Id
               | CONTEUDO Id
               deriving (Show, Eq)

data Valor = V_INT Int
           | V_BOOL Bool
           | V_STRING String
           | V_NULL
           deriving (Show, Eq)

data ExpUnaria = MENOS Expressao
               | NOT Expressao
               deriving (Show, Eq)

data ExpBinaria = ADICAO Expressao Expressao
               | SUBTRACAO Expressao Expressao
               | AND Expressao Expressao
               | OR Expressao Expressao
               | IGUAL Expressao Expressao
               | CONCATENA Expressao Expressao
               deriving (Show, Eq)
```

3.5. Declarações

```
data ComandoDeclaracao = COM_DECLARACAO Declaracao Comando
                       deriving (Show, Eq)

data Declaracao = DEC_VAR DeclaracaoVariavel
               | DEC_PROC DeclaracaoProcedimento
               | DEC_MULT Declaracao Declaracao
               deriving (Show, Eq)

data DeclaracaoVariavel = VAR Id Expressao
                       | PONTEIRO Id Tipo Expressao
                       deriving (Show, Eq)

data DeclaracaoProcedimento = PROC Id ListaDeclaracaoParametros Comando
                           deriving (Show, Eq)

data ListaDeclaracaoParametro = PARAM_UNICO Id Tipo
                              | PARAM_MULT Id Tipo ListaDeclaracaoParametro
                              deriving (Show, Eq)
```

3.6. Tipos

```
data Tipo = T_INT
          | T_BOOL
          | T_STRING
          | T_PONTEIRO Tipo
          deriving (Show, Eq)
```

3.7. Comandos de Controle de Fluxo

```
data While = WHILE Expressao Comando
             deriving (Show, Eq)

data For = FOR Atribuicao V_INT Comando
           deriving (Show, Eq)

data IfThenElse = IF_THEN_ELSE Expressao Comando Comando
                  deriving (Show, Eq)

data RepeatUntil = REPEAT_UNTIL Comando Expressao
                   deriving (Show, Eq)
```

3.8. Comandos de Entrada e Saída

```
data InputOutput = WRITE Expressao
                 | READ Id
                 deriving (Show, Eq)
```

3.9. Chamadas de Procedimentos

```
data ChamadaProcedimento = CALL Id ListaExpressao
                           deriving (Show, Eq)

data ListaExpressao = EXPRESSAO_UNICA Expressao
                    | EXPRESSAO_MULT Expressao ListaExpressao
                    deriving (Show, Eq)
```

4. Entidades Semânticas

Como mencionado anteriormente, a semântica de ações possui estrutura modular, semelhante à semântica denotacional. A descrição de uma linguagem em semântica de ações consiste em três módulos principais: sintaxe abstrata (apresentada anteriormente), entidades semânticas e funções semânticas. As entidades semânticas foram subdivididas em módulos, de acordo com os tipos de dados da linguagem: valores, dados, tipos, etc.

Nesta seção, serão mostradas as entidades semânticas da linguagem HELL.

4.1. Dados

datum[1] = valor | tipo | procedimento | argumento | token[2]

token = string-of (alpha, (alpha | digit)*)

bindable[3] = valor | cell[4]

4.2. Valores

Valor = boolean | int| string

string = list[char]

char = letter | digit.

4.3. Tipos

type = boolean | int | string | ponteiro

4.4. Bindings[5]

Endereco = Int

Id = String

Console = String

---

[1] Representam ítens de dados. Depende da variedade de informações processadas pelos programas de uma linguagem de programação.

[2] Uma subcategoria de datum distintos. Depende da variedade de identificadores de uma linguagem de programação. (Geralmente, é um subconjunto de strings).

[3] Uma subcategoria de dados. Depende da variedade de informações processadas por escopo dos programas de uma linguagem de programação.

[4] Posição de memória.

[5] Representam conjuntos de associações entre símbolos (tokens) e dados ligáveis (bindable).

4.5. Armazenamento

Ambiente = ([Escopo], MemVariaveis, MemProcedimentos)

Escopo = (TabVariaveis, TabProcedimentos)

TabVariaveis = [Variavel]

TabelaProcedimento = [Procedimento]

Variavel = (Id, Tipo, Endereco)

Procedimento = (Id, ListaParametros, Endereco)

ListaParametros = [(Id, Tipo)]

MemVariaveis = [(Endereco, Valor)]

MemProcedimentos = [(Endereco, Comando)]

4.6. Procedimentos

Procedimento = abstração[6]

Parametros = valores | Endereco

ListaParametros = lista [Parmetros]

[6] Seleção de informações úteis relevantes em um contexto. Consiste no processo de identificar apenas qualidades ou propriedades relevantes do fenômeno que se quer modelar. Uma abstração de procedimento contém um comando a ser executado, que quando chamado irá atualizar as variáveis envolvidas.

5. Funções Semânticas

Assim como as entidades semânticas, as funções semânticas usadas para a descrição de uma linguagem de programação, conforme citado no tópico anterior, são subdivididas em módulos, de acordo com os tipos de construções da linguagem descrita, como por exemplo, expressões, declarações, comandos, etc.

A seguir, serão descritas as principais funções semânticas da linguagem HELL.

5.1. Programa

exe _ :: Programa → ação

(1) exe [[Comando:Comando]]

| Executa *Comando*

5.2. Declarações

elabore _ :: Declaração → ação

(1) elabore [[“var” var:Idr “=” Expressao:Expressao]]

| | Avalia a expressão *Expressao*
| | Reserva um espaço de memória do mesmo tipo do valor de *Expressao*
| Entao
| | Liga o token *var* ao espaço reservado na memória

(2) elabore [[“pointer” p:Id “=” “^” T:Tipo]]

| | Reserva um espaço na memória
| Entao
| | Liga o identificador *p* a esse espaço reservado

(3) elabore [[“proc” proc:Id “(“ [Lista: ListaDeclaracaoParametro] “)” “{“ Comando:Comando “}”]]

| Faz a ligação do token *proc* ao fecho da abstração recursivamente
| | Elabora a lista de declarações de parâmetros L
| Então
| | Executa *Comando*

5.3. Expressões

avalie _ :: Expressão → Ação

(1)avalie [[“true”]]

| Retorna true

(2)avalie [[“false”]]

| Retorna False

(3)avalie [[Num:Numeral]]

| Retorna o valor de *Num*

(4)avalie [[String:String]]

| Retorna o valor de *String*

(5)avalie [[Identificador:Id]]

| Retorna o valor armazenado no endereço ligado ao token *Identificador*

(6)avalie [[“-” Expressao:Expressao]]

| Avalia a expressão *Expressao* e retorna um valor simétrico ao valor de *Expressao*

(7)avalie [[“!” Expressao:Expressao]]

| Avalia a expressão *Expressao* e retorna o seu valor negado

(8)avalie [[Expressao1:Expressao “+” Expressao2:Expressao]]

| | Avalia *Expressao1*
| | Avalia *Expressao2*
| Então
| | Retorna o resultado da soma dos valores das duas expressões

(9)avalie [[Expressao1:Expressao “-” Expressao2:Expressao]]

| | Avalia *Expressao1*

Avalia *Expressao2*
Então
Retorna o resultado da diferença dos valores das duas expressões

(10) avalie [[Expressao1:Expressao "&&" Expressao2:Expressao]]

Avalia *Expressao1*
Avalia *Expressao2*
Então
Retorna o resultado da operação *and* entre os valores das duas expressões

(11) avalie [[Expressao1:Expressao "||" Expressao2:Expressao]]

Avalia *Expressao1*
Avalia *Expressao2*
Então
Retorna o resultado da operação *ou* entre os valores das duas expressões

(12) avalie [[Expressao1:Expressao "==" Expressao2:Expressao]]

Avalia *Expressao1*
Avalia *Expressao2*
Então
Retorna *true* se os valores das duas expressões são iguais, ou *false* caso não sejam

(13) avalie [[Expressao1:Expressao "++" Expressao2:Expressao]]

Avalia *Expressao1*
Avalia *Expressao2*
Então
Retorna o resultado da concatenação dos valores das duas expressões

5.4. Comandos

execute_ :: Comando → Ação

(1) execute[["{" Declaracao:Declaracao ";" Comando:Comando "}"]]

Elabora *Declaracao*
Executa *Comando*

(2) execute [[Identificador:Id ":=" Expressao:Expressao]]

Avalia a expressão *Expressao*
Então
Armazena o valor de *Expressao* no endereço de memória ligado ao token *Identificador* .

(3) execute [[“while” Expressao:Expressao “do” Comando:Comando]]

Avalia expressão *Expressao*
Então
Executa *Comando*
Senão
Fim

(4) execute [[“for” Atribuicao:ComandoAtribuicao “to” Inteiro:ValorInteiro “do” Comando:Comando]]

Executa *Atribuicao*
Verifica se *Atribuicao* é menor que *Inteiro*
Executa *Comando*
Senão
Fim

(5) execute [[“if” Expressao:Expressao “then” Comando1:Comando “else” Comando2:Comando]]

Avalia *Expressao*
Então
Executa *Comando1*
Senão
Executa *Comando2*

(6) execute [[“repeat” Comando:Comando “until” Expressao:Expressao]]

Executa *Comando*
Então
Verifica a expressão *Expressao* é falsa
Executa *Comando*
Senão
Fim

(7) execute [[“write” “(” Expressao:Expressao “)”]]

Avalia *Expressao*
Então
Escreve o valor da expressão *Expressao* no endereço 0 (zero) da memória

(8) execute [[“read” “(” entrada:Id “)”]]

Lê o valor do endereço 1 (um) da memória
Então
Armazena esse valor no endereço de memória ligado ao token *entrada*

(9) execute [[“skip”]]

Fim

(10) execute [[“call” proc:Id “(” Lista:ListaExpressao “)”]]

Avalia a lista de expressões *Lista*
Então
Executa *proc*

(11) execute [[Comando1:Comando “,” Comando2:Comando]]

Executa *Comando1*
Executa *Comando2*

6. Sintaxe Léxica

Gramática:

Letra = a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Número = Dígito | Número Dígito

Dígito = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Id = Letra | Id Letra | Id Número

Char = Letra | Dígito | ‘ ’

Int = Digito | Int Digito.

Booleano = "True" | "False"

String = Char | String Char

7. O Interpretador

Definida a linguagem de programação imperativa HELL, foi implementado pra ela um interpretador em linguagem Haskell. Essa linguagem foi escolhida pelos seguintes motivos:

- permite o rápido desenvolvimento de softwares robustos, concisos e corretos;
- possibilita o desenvolvimento de softwares flexíveis, manuteníveis e de alta qualidade;
- proporciona um aumento substancial da produtividade do programador;
- produz um código curto, claro e manutenível;
- possibilita uma maior confiabilidade uma vez que reduz a possibilidade de erros.

Detalhes de implementação e arquitetura do interpretador, serão mostrados nessa seção.

7.1. Definição

Um interpretador é um programa que lê um código-fonte escrito em uma linguagem de programação, e o converte em um código executável. Essa conversão pode ser feita de duas formas. Há interpretadores que lêem linha-a-linha do código fonte, convertendo-o em código-objeto , à medida que o programa vai sendo executado. Um segundo tipo de interpretadores, converteria o código-fonte por inteiro para depois executá-lo.

No caso do interpretador HELL, cada comando presente no código vai sendo executado, à medida que é lido. A ordem de execução é de cima para baixo, da esquerda pra direita, sendo a precedência dos comandos definida através do uso de parênteses.

7.2. Arquitetura

A arquitetura do Interpretador é composta por sete camadas, como mostrado abaixo:

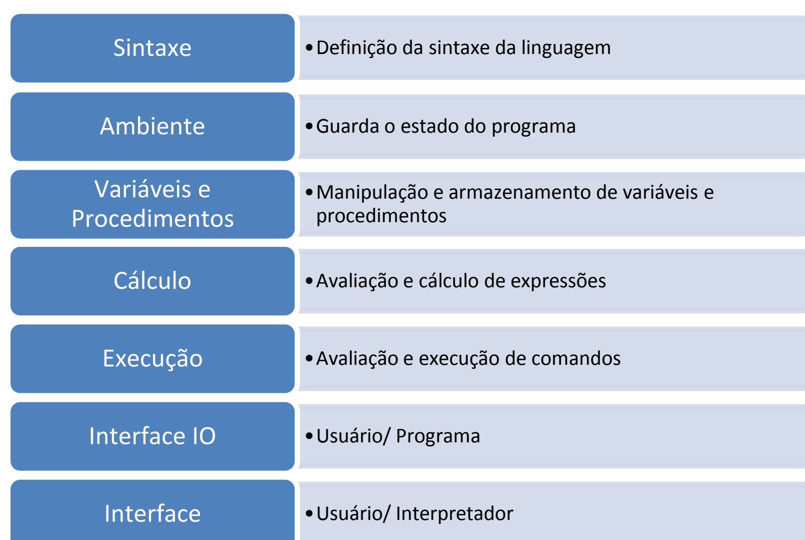

Figura 1 - Arquitetura do Interpretador

A primeira camada, denominada *Sintaxe*, é responsável exatamente por definir a sintaxe da linguagem a ser interpretada, e é a base para o funcionamento do interpretador. Parte do código pertencente a essa camada foi mostrado nesse documento, na seção 3.

A próxima camada, denominada *Ambiente*, a qual engloba todas as estruturas responsáveis por manter o estado do programa que está sendo executado, está dividida em três subcamadas:

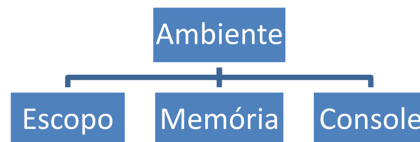

Figura 2 - Camada Ambiente

- A subcamada *Escopo* é responsável por armazenar e manipular todos os escopos do programa em execução, desde o escopo global até o escopo local. Isso é feito por meio de tabelas de variáveis e procedimentos. Cada escopo terá essas duas tabelas. A tabela de variáveis possui três entradas: o identificador da variável, o seu tipo, e o endereço no qual está armazenada. Nesta tabela estarão armazenadas todas as variáveis pertencentes ao escopo ao qual a tabela pertence. Já a tabela de procedimentos tem como entradas o identificador do procedimento, sua lista de parâmetros e o seu endereço de memória. Assim como a tabela de variáveis, na tabela de procedimentos estarão armazenados todos os procedimentos pertencentes ao escopo ao qual a tabela pertence.
- A subcamada *Memória*, por sua vez, é composta pela memória de variáveis e pela memória de procedimentos. A memória de variáveis armazena, em cada endereço, o tipo e o valor da variável cujo identificador está associado a esse endereço. A memória de procedimentos armazena, em cada endereço, o corpo do procedimento cujo identificador está ligado a esse endereço.
- Na subcamada *Console* são armazenadas as saídas dos programas, as quais serão mostradas na camada Interface.

A terceira camada, *Variáveis e Procedimentos*, é a responsável pelo tratamento de variáveis e procedimentos. No que diz respeito às variáveis, é nessa camada que elas são alocadas, que lhes são atribuídos valores, que são feitas as verificações de escopo e de tipo, e todas as manipulações de endereços.

Quanto aos procedimentos, é também nessa camada que eles são alocados, que são feitos os tratamentos de endereços e parâmetros, além das verificações de escopo.

Na quarta camada, chamada *Cálculos*, as expressões são avaliadas e seus valores são calculados, abordando todos os tipos de expressões e operações entre elas permitidos pela linguagem.

Na camada *Execução* estão as funções responsáveis por executar todos os tipos de comandos que possam estar presentes no código-fonte.

A camada *Interface IO* é responsável por auxiliar na abstração dos comandos de entrada e saída de dados. Essa camada constitui-se no que seria uma interface com o usuário do programa que está sendo executado pelo interpretador.

Por fim, é na camada *Interface* que estão os programas a serem interpretados, e é essa camada a responsável por mostrar ao usuário os resultados da execução do programa. Essa camada constitui-se em uma interface do interpretador com o usuário.

7.3. Comportamento

A execução do interpretador começa com a leitura do código-fonte, na camada *Interface*. À medida que o código vai sendo lido, os comandos presentes nele vão sendo executados, a partir da camada *Execução*. A primeira linha do código sempre conterá a instrução PROGRAMA, a qual é composta por um comando. Os comandos podem ser: declaração, skip, entrada e saída, atribuição, loops (while, for ou repeat), comandos de decisão (if-then-else) ou chamadas de procedimentos.

7.3.1. Declarações

No caso de um comando de declaração, de variável ou procedimento, a execução é passada à camada *Variáveis e Procedimentos*, que irá alocar o espaço necessário, fazendo todos os tratamentos de erros cabíveis, e em seguida passará a execução para a camada Ambiente, que armazenará as informações necessárias no escopo e na memória.

7.3.2. Atribuições

No caso de um comando de atribuição, após ser identificado na camada *Execução*, o processamento, incluindo verificações de tipos, será realizado na camada *Variáveis e Procedimentos*, e as alterações no escopo e na memória serão feitas pela camada *Ambiente*.

7.3.3. Estruturas de Repetição

Para todos os comandos de repetição são criados escopos locais, de maneira que o usuário possa fazer declarações de variáveis dentro do loop, que possuam o mesmo identificador de variáveis pertencentes a escopos externos. Assim, a cada iteração do loop um novo escopo é criado. Ao final da execução, todos os escopos que foram abertos são fechados. Isso acontece devido à interpretação recursiva do programa, característica da linguagem usada para implementação do interpretador.

A execução do comando **while** começa na camada *Execução*, onde é identificado. Em seguida, na camada *Cálculos*, a expressão booleana do loop é avaliada, e o seu valor é retornado. O valor da expressão é verificado com o auxílio da camada *Ambiente*. Os comandos pertencentes ao loop são executados a partir da camada *Execução*. Essas ações se repetem até que o valor da expressão booleana seja “False”.

A execução do comando **for** também começa na camada *Execução*, com a sua identificação. A partir daí, é tratado o comando de atribuição, conforme descrito anteriormente. Em seguida, são executados, a cada iteração, o incremento da variável de controle pela camada *Cálculos*, e os comandos pertencentes ao loop, a partir da camada *Execução*. Essas ações se repetem até que o valor da variável de controle seja igual ao valor limite definido.

O comando **repeat**, assim como os demais, começa a partir da sua identificação na camada *Execução*. Em seguida os comandos pertencentes ao loop são executados, ainda a partir da camada *Execução*. O valor da expressão booleana é obtido na camada *Cálculos* e o seu valor é verificado com auxílio da camada *Ambiente*. Esses passos se repetem até que o valor da expressão seja "True".

7.3.4. If-Then-Else

Assim como os demais comandos, sua execução começa na camada *Execução*. Em seguida é passada para a camada *Ambiente*, onde um novo escopo é criado. O comando expressão é executado pela camada *Cálculos*, e testado com auxílio da camada *Ambiente*. De acordo com o valor da expressão, é executado um dos blocos de comandos definidos, a partir da camada *Execução*. Executados os comandos, o escopo é encerrado pela camada *Ambiente*.

7.3.5. Chamadas de Procedimentos

A chamada de um procedimento é identificada na camada *Execução*. Após isso, a existência do procedimento é verificada, a partir da camada *Variáveis e Procedimentos*, com auxílio da camada *Ambiente*. Em seguida, a lista de parâmetros é verificada, os valores das expressões são obtidos na camada *Cálculos*, e a checagem de tipos é feita com auxílio da camada *Variáveis e Procedimentos*.

Após a verificação dos parâmetros, é criado um escopo para o procedimento, que estará localizado na camada *Ambiente*, e os parâmetros são alocados nesse escopo, a partir da camada *Execução*, com auxílio das camadas *Cálculos* e *Variáveis e Procedimentos*.

Os comandos pertencentes ao procedimento serão executados a partir da camada *Execução*.

7.3.6. Entrada e Saída

A execução dos comandos de entrada e saída começam na camada *Execução*, onde são identificados. Após isso, a execução é passada à camada *Interface IO*.

No caso do comando **READ**, deve ser lido o valor no endereço 1 (um) da memória e esse valor deve ser atribuído à variável passada como parâmetro ao comando. Na camada *Interface IO* o usuário pode entrar com o valor que será armazenado nessa posição de memória. Em seguida a execução é passada para a camada *Variáveis e Procedimentos*, que fará a atribuição.

No caso do comando **WRITE**, que deve armazenar no endereço 0 (zero) da memória o valor da expressão passada como parâmetro ao comando, a execução é passada da camada *Interface IO* para a camada *Cálculos*, que fará o cálculo do valor da expressão. Em seguida, na camada *Sintaxe*, esse valor é convertido para uma String, então a execução é passada à camada *Ambiente*, que anexa essa String ao *Console* (subcamada da camada *Ambiente*). Por fim, na camada *Interface*, o *Console* é mostrado ao usuário, junto com as demais saídas do interpretador.

7.4. Implementação

O código do Interpretador foi dividido em módulos, de forma a implementar as camadas descritas acima. A relação entre os módulos e as camadas é mostrada no diagrama abaixo:

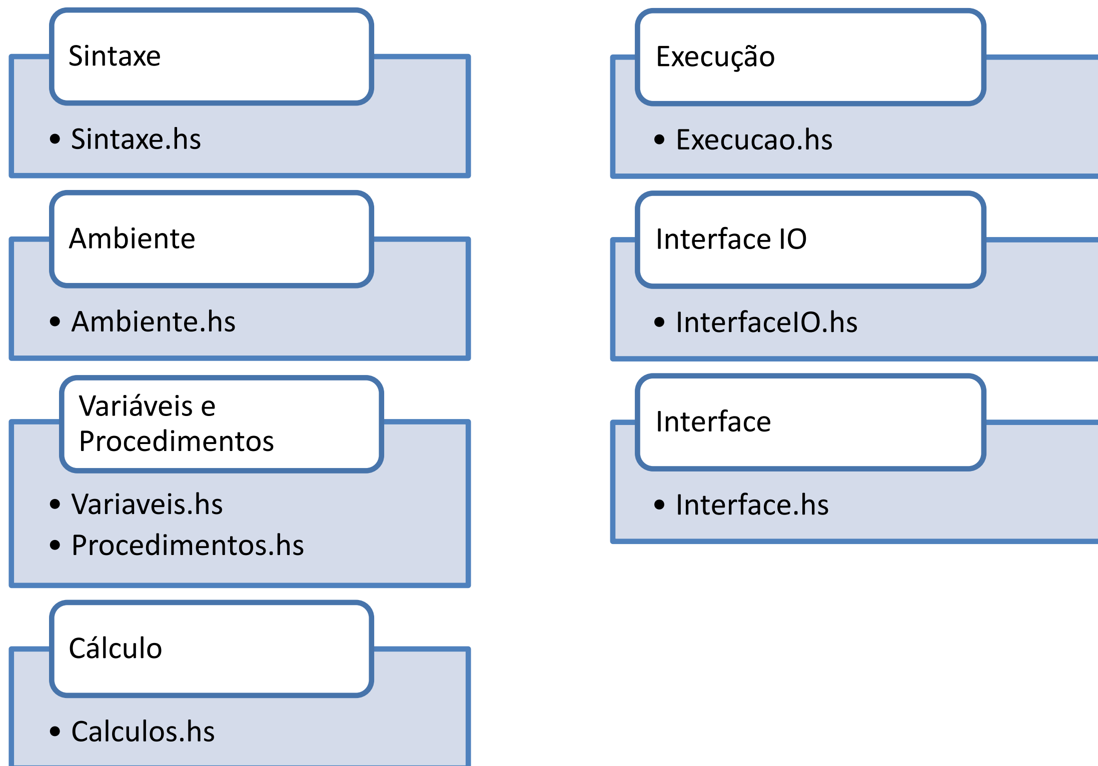

Figura 3 – Módulos e Camadas

O próximo diagrama mostra a hierarquia entre os módulos do programa:

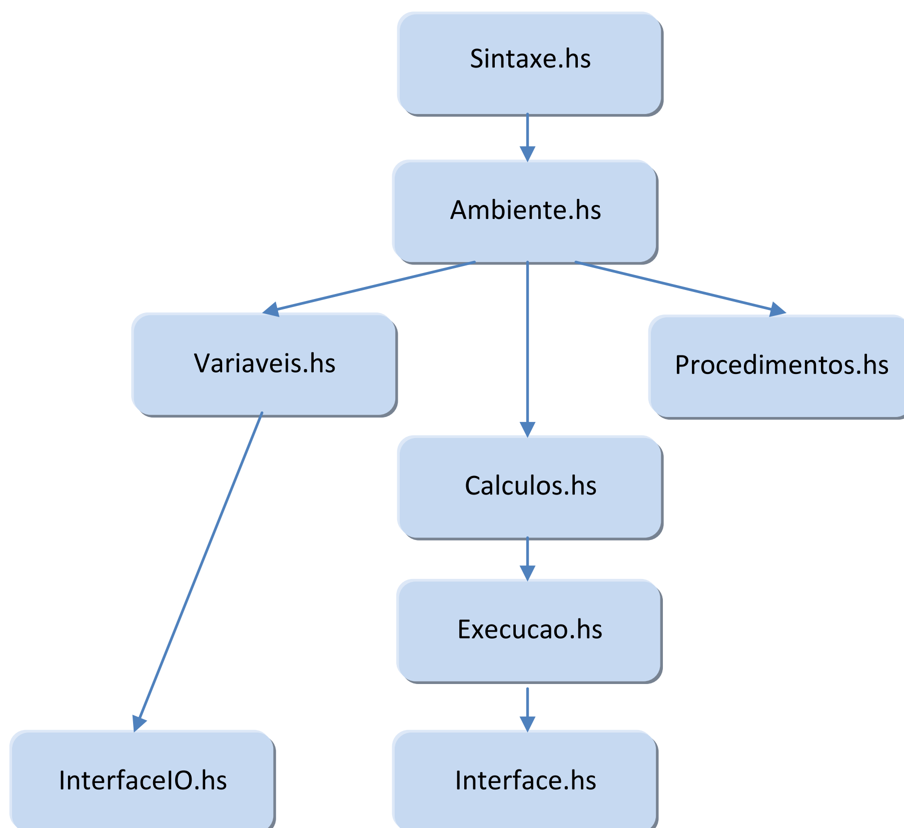

Figura 4 - Hierarquia do Código

8. Exemplos de Programas

Nesta seção serão mostrados os programas que foram usados para validar o interpretador. Para cada programa serão mostrados os códigos em linguagem HELL, e a sua versão em linguagem C. Esses códigos estão presentes no arquivo Interface.hs, do interpretador.

Serão mostrados também os resultados da execução de cada código, que correspondem à saída do interpretador. Essa saída mostra o ambiente atual, após a execução do programa. Esse ambiente é formado pelo escopo, pela memória de variáveis, pela memória de procedimentos, e pelo console. No escopo tem-se uma tabela de variáveis e uma tabela de procedimentos.

8.1. Declaração e Atribuição

programa1 = PROGRAMA (C_DECLARACAO ((COM_DECLARACAO (DEC_VAR (VAR "x" (VALOR (V_INT 40))))) (C_ATRIBUICAO (ATRIBUICAO "x" (VALOR (V_INT 56))))))

Versão em C:

```
void main ()
 {
        int x = 40;
        x = 56;
}
```

O programa mostrado declara uma variável do tipo inteiro com identificador *x*, e em seguida atribui a essa variável o valor 56.

Saída:

Tabela de Variaveis:
ID: x Tipo: T_INT Endereco: 2

Memoria de Variaveis:
Output: V_NULL
Input: V_NULL
Endereco: 2 Valor: V_INT 56

Tabela de Procedimentos:

Memoria de Procedimentos:

Console:

8.2. Procedimentos, Chamadas de Procedimentos, Loop While, Saida de Dados

```
programa2 = PROGRAMA (
          C_DECLARACAO (COM_DECLARACAO
                    (DEC_VAR (VAR "X" (VALOR (V_INT 15))))
                              (C_DECLARACAO (COM_DECLARACAO
                                        (DEC_VAR (VAR "Y" (VALOR (V_INT 25))))
                                                  (C_DECLARACAO
                                                            (COM_DECLARACAO
                                                                      (DEC_VAR (VAR "Mult" (VALOR (V_INT
0))))
                                                                                (C_DECLARACAO
(COM_DECLARACAO
                                                                                          (DEC_PROC
                                                                                                    (PROC
"multiplica" (PARAM_MULT "arg0" T_INT (PARAM_UNICO "arg1" T_INT))

          (C_DECLARACAO

          (COM_DECLARACAO (DEC_VAR (VAR "Z" (VALOR (V_INT 0))))

          (C_COMANDO_MULT (C_ATRIBUICAO (ATRIBUICAO "Mult" (VALOR (V_INT 0))))

                    (C_WHILE (WHILE (EXP_UNARIA (NOT (EXP_BINARIA (IGUAL (ID "Z") (ID
"arg1")))))

                                        (C_COMANDO_MULT (C_ATRIBUICAO (ATRIBUICAO "Mult"
(EXP_BINARIA (ADICAO (ID "Mult") (ID "arg0")))))

                                                  (C_ATRIBUICAO (ATRIBUICAO "Z" (EXP_BINARIA
(ADICAO (ID "Z") (VALOR (V_INT 1))))))

                                        )

                              )

                    )

          )

          )
                                                                                                              )
                                                                                                    )
                                                                                          )
```

```
                                                    (C_COMANDO_MULT
(C_CHAMADA_PROC (CHAMAR "multiplica" (EXPRESSAO_MULT (ID "X") (EXPRESSAO_UNICA
(ID "Y")))))
                                                    (C_IO (WRITE (ID
"Mult")))))
                                              )
                                    )
                               )
                          )
                     )
                )
           )
```

Versão em C:

```
void main ()
{
	int x = 15;
	int y = 25;
	int mult = 0;

	void multiplica(int arg0, int arg1)
	{
		int z = 0;
		mult = 0;

		while(z != arg1)
		{
			mult += arg0;
			z++;
		}
	}

	multiplica(x, y);

	printf("%d", mult);

}
```

O programa declara três variáveis do tipo inteiro, com identificadores *x*, *y*, e *mult*, e atribui a essas variáveis os valores 15, 25 e 0, respectivamente. A seguir é declarado o procedimento *multiplica*, com 2 argumentos na lista de parâmetros, do tipo inteiro, *arg0* e *arg1*. No corpo do procedimento é declarada mais uma variável, também do tipo inteiro, com identificador z, a qual é inicializada com o valor 0. Após a declaração existe uma atribuição, e em seguida um

loop while, que é executado enquanto o valor da variável *z* é diferente do valor do argumento *arg1*. Dentro do loop são feitas duas atribuições.

Encerrado o corpo do procedimento, este é chamado, passando como argumentos as variáveis as variáveis *x* e *y*.

Após isso é impresso no console o valor da variável *mult*, que deve ser o resultado do produto x * y.

Saída:

Tabela de Variaveis:
ID: Mult Tipo: T_INT Endereco: 4
ID: Y Tipo: T_INT Endereco: 3
ID: X Tipo: T_INT Endereco: 2

Memoria de Variaveis:
Output: V_INT 375
Input: V_NULL
Endereco: 2 Valor: V_INT 15
Endereco: 3 Valor: V_INT 25
Endereco: 4 Valor: V_INT 375

Tabela de Procedimentos:
ID: multiplica Endereco: 0

Memoria de Procedimentos:
Endereco: 0 Valor: C_DECLARACAO (COM_DECLARACAO (DEC_VAR (VAR "Z" (VALOR (V_INT 0)))) (C_COMANDO_MULT (C_ATRIBUICAO (ATRIBUICAO "Mult" (VALOR (V_INT 0)))) (C_WHILE (WHILE (EXP_UNARIA (NOT (EXP_BINARIA (IGUAL (ID "Z") (ID "arg1"))))) (C_COMANDO_MULT (C_ATRIBUICAO (ATRIBUICAO "Mult" (EXP_BINARIA (ADICAO (ID "Mult") (ID "arg0"))))) (C_ATRIBUICAO (ATRIBUICAO "Z" (EXP_BINARIA (ADICAO (ID "Z") (VALOR (V_INT 1))))))))))

Console:
375

8.3. Loop While

```
programa3 = PROGRAMA (
        C_DECLARACAO (COM_DECLARACAO
                                (DEC_VAR
                                        (VAR "X"
```

```
                                                            (VALOR (V_INT 0))
                                                  )
                                        )
                              (C_DECLARACAO (COM_DECLARACAO
                                                                      (DEC_VAR
                                                                                (VAR "Y"
                                                                                          (VALOR (V_INT 5))
                                                                                )
                                                                      )
                                                            (C_WHILE (WHILE (EXP_UNARIA (NOT
(EXP_BINARIA
                                                                                          (IGUAL
                                                                                                    (ID "X") (ID "Y")
                                                                                          )
                                                                                )))

                                                                                (C_ATRIBUICAO
                                                                                          (ATRIBUICAO "X"
                                                                                                    (EXP_BINARIA

          (ADICAO

          (ID "X") (VALOR (V_INT 1))
                                                                                                              )
                                                                                                    )
                                                                                          )
                                                                                )
                                                                      )
                                                            )
                                                  )
                                        )
                    )
          )
```

Versão em C:

```
void main ()
{
	int X = 0;
	int Y = 5;

	while(X != Y)
	{
		X++;
	}
}
```

O programa é composto por duas declarações e um loop while, onde existe um comando de atribuição.

Saída:

Tabela de Variaveis:
ID: Y   Tipo: T_INT   Endereco: 3
ID: X   Tipo: T_INT   Endereco: 2

Memoria de Variaveis:
Output: V_NULL
Input: V_NULL
Endereco: 2   Valor: V_INT 5
Endereco: 3   Valor: V_INT 5

Tabela de Procedimentos:

Memoria de Procedimentos:

Console:

8.4. Loop Repeat

```
programa4 = PROGRAMA (
        C_DECLARACAO (COM_DECLARACAO
                                (DEC_VAR
                                        (VAR "X"
                                                (VALOR (V_INT 0))
                                        )
                                )
                        (C_DECLARACAO (COM_DECLARACAO
                                                        (DEC_VAR
                                                                (VAR "Y"
                                                                        (VALOR (V_INT 5))
                                                                )
                                                        )
                                                (C_REPEAT_UNTIL (REPEAT_UNTIL
                                                                (C_ATRIBUICAO
                                                                        (ATRIBUICAO "X"
                                                                                (EXP_BINARIA

        (ADICAO

        (ID "X") (VALOR (V_INT 1))
```

```
                                                                        )
                                                                )
                                                        )
                                                )
                                                (EXP_BINARIA
                                                        (IGUAL
                                                                (ID "X") (ID "Y")
                                                        )
                                                )
                                        )
                                )
                        )
                )
        )
)
```

Versão em C:

```
void main ()
{
	int X = 0;
	int Y = 5;

	do {
		X++;
	while(!(X == Y));
}
```

O programa começa com duas declarações de variáveis do tipo inteiro. Em seguida, é executado o loop repeat, que contém um comando de atribuição.

Saída:

Tabela de Variaveis:

ID: Y	Tipo: T_INT	Endereco: 3
ID: X	Tipo: T_INT	Endereco: 2

Memoria de Variaveis:

Output:	V_NULL
Input:	V_NULL
Endereco: 2	Valor: V_INT 5
Endereco: 3	Valor: V_INT 5

Tabela de Procedimentos:

Memoria de Procedimentos:

Console:

8.5. Entrada e Saída de Dados, Procedimento Recursivo, If Then Else

```
programa5 = PROGRAMA (C_DECLARACAO
                                        (COM_DECLARACAO
                                                (DEC_VAR
                                                        (VAR "x" (VALOR (V_INT 0)))
                                                )
                                                (C_COMANDO_MULT
                                                        (C_IO
                                                                (READ "x")
                                                        )

                                                        (C_DECLARACAO
                                                                (COM_DECLARACAO
                                                                        (DEC_PROC
                                                                                (PROC
"imprimeAteDiferenteDe20" (PARAM_UNICO "n" (T_INT))

        (C_COMANDO_MULT

        (C_IO

                (WRITE (ID "n"))

        )

        (C_COMANDO_MULT

                (C_DECLARACAO

                        (COM_DECLARACAO

                                (DEC_VAR

                                        (VAR "espaco" (VALOR (V_STRING " ")))

                                )

                                (C_IO
```

```
                                        (WRITE (ID "espaco"))
                                )
                        )
                )

                (C_IF_THEN_ELSE
                        (IF_THEN_ELSE (EXP_UNARIA (NOT (EXP_BINARIA (IGUAL (ID "n")
(VALOR (V_INT 20))))))
                                (C_CHAMADA_PROC
                                        (CHAMAR "imprimeAteDiferenteDe20"
(EXPRESSAO_UNICA (EXP_BINARIA (ADICAO (ID "n") (VALOR (V_INT 1))))))
                                )
                                (C_DECLARACAO
                                        (COM_DECLARACAO
                                                (DEC_VAR
                                                                        (VAR "msgSaida"
(VALOR (V_STRING "FIM")))
                                                )
                                                (C_IO
                                                        (READ "msgSaida")
                                                )
                                        )
                                )
                        )
```

```
                )

        )

        )
                                                                                )
                                                                        )

        (C_CHAMADA_PROC

        (CHAMAR "imprimeAteDiferenteDe20" (EXPRESSAO_UNICA (ID "x")))
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                )
```

Versão em C:

```
void main ()
{
	int x = 0;

	scanf("%d", &x);

	void imprimeAteDiferenteDe20(int n)
	{
		printf("%d ", n);

		if(n != 20)
		{
			imprimeAteDiferenteDe20(n + 1);
		}
		else
		{
			printf("FIM");
		}
	}

	imprimeAteDiferenteDe20(x);
}
```

O programa começa com a declaração de uma variável do tipo inteiro. Em seguida é lido um valor do teclado e armazenado nessa variável.

Após isso, é declarado o procedimento recursivo *imprimeAteDiferenteDe20*, que recebe um parâmetro do tipo inteiro. Esse procedimento imprime uma sequência de números começando do número informado até o número 20. Para isso existe no corpo do procedimento um comando If-Then-Else.

Encerrada a declaração do procedimento, este é chamado, passando como parâmetro o valor lido do teclado.

Saída:

Tabela de Variaveis:
ID: x Tipo: T_INT Endereco: 2

Memoria de Variaveis:
Output: V_STRING "FIM"
Input: V_INT 0
Endereco: 2 Valor: V_INT 0

Tabela de Procedimentos:
ID: imprimeAteDiferenteDe20 Endereco: 0

Memoria de Procedimentos:
Endereco: 0 Valor: C_COMANDO_MULT (C_IO (WRITE (ID "n"))) (C_COMANDO_MULT (C_DECLARACAO (COM_DECLARACAO (DEC_VAR (VAR "espaco" (VALOR (V_STRING " ")))) (C_IO (WRITE (ID "espaco"))))) (C_IF_THEN_ELSE (IF_THEN_ELSE (EXP_UNARIA (NOT (EXP_BINARIA (IGUAL (ID "n") (VALOR (V_INT 20)))))) (C_CHAMADA_PROC (CHAMAR "imprimeAteDiferenteDe20" (EXPRESSAO_UNICA (EXP_BINARIA (ADICAO (ID "n") (VALOR (V_INT 1))))))) (C_DECLARACAO (COM_DECLARACAO (DEC_VAR (VAR "msgSaida" (VALOR (V_STRING "FIM")))) (C_IO (WRITE (ID "msgSaida")))))))

Console:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 FIM

8.6. Ponteiros, Saída de Dados, Operações com Strings

```
programa6 =    PROGRAMA (C_DECLARACAO
          (COM_DECLARACAO
            (DEC_VAR
              (PONTEIRO "p" (T_STRING))
            )
            (C_DECLARACAO
              (COM_DECLARACAO
                (DEC_VAR
                  (VAR "conteudo"
```

```
                        (VALOR (V_STRING "NOVO CONTEUDO"))
                      )
                    )
                    (C_DECLARACAO
                          (COM_DECLARACAO
                            (DEC_VAR
                              (VAR "saida"
                                (VALOR (V_STRING ""))
                              )
                            )
                            (C_COMANDO_MULT
                              (C_ATRIBUICAO
                                (ATRIBUICAO "p"
                                  (ENDERECO ("conteudo"))
                                )
                          )

                              (C_ATRIBUICAO
                                (ATRIBUICAO "saida"
                                  (EXP_BINARIA
                                    (CONCATENA
                                      (CONTEUDO ("p")) (VALOR (V_STRING "eh o conteudo do
ponteiro"))
                                )
                              )
                            )
                          )
                        )
                          )
                        )
                    )
                  )
                )
              )
```

Versão em C++:

```
void main ()
{
        String *p;
        String conteudo = "NOVOCONTEUDO";
        String saida = "";

        p = &conteudo;
        saida = *p ++ "eh o conteudo do ponteiro";
}
```

O programa começa com a declaração de uma variável do tipo ponteiro pra String, e mais duas variáveis do tipo String, as quais são inicializadas com strings constantes. Em seguida, faz-se o ponteiro apontar para uma das strings. Então o conteúdo do ponteiro é concatenado a uma string constante, e o resultado é atribuído à segunda string.

Saída:

```
Tabela de Variaveis:
ID: saida           Tipo: T_STRING Endereco: 4
ID: conteudo        Tipo: T_STRING Endereco: 3
ID: p         Tipo: T_PONTEIRO T_STRING     Endereco: 2

Memoria de Variaveis:
Output:             V_NULL
Input:              V_NULL
Endereco: 2         Valor: V_INT 3
Endereco: 3         Valor: V_STRING "NOVO CONTEUDO"
Endereco: 4         Valor: V_STRING "NOVO CONTEUDOeh o conteudo do ponteiro"


Tabela de Procedimentos:

Memoria de Procedimentos:

Console:
```

8.7. Loop For

```
programa7 = PROGRAMA (C_DECLARACAO
        (COM_DECLARACAO
          (DEC_VAR
            (VAR "a"
              (VALOR (V_INT 1))
            )
          )
          (C_DECLARACAO
            (COM_DECLARACAO
              (DEC_VAR
                (VAR "i"
                  (VALOR (V_INT 10))
                )
              )
              (C_FOR
                (FOR
```

```
(ATRIBUICAO "i"
   (VALOR (V_INT 0))
)
(4)
(C_ATRIBUICAO
   (ATRIBUICAO "a"
      (EXP_BINARIA
         (ADICAO
            (ID ("a")) (VALOR (V_INT 1))
         )
      )
   )
)
                              )
                        )
                     )
                  )
               )
            )
```

Versão em C:

```
void main ()
{
	int a = 1;
	int i = 10;

	for (i = 0; i < 4; i++)
	{
		a++;
	}
}
```

O programa começa com a declaração de duas variáveis do tipo inteiro, que são inicializadas com os valores 1 e 10. O próximo comando é um loop for, que tem a segunda variável declarada como variável de controle. O loop é executado a partir de i = 0, até que o valor dessa variável atinja o valor 4. Dentro do loop a primeira variável vai sendo incrementada.

Saída:

```
Tabela de Variaveis:
ID: i		Tipo: T_INT	Endereco: 3
ID: a		Tipo: T_INT	Endereco: 2

Memoria de Variaveis:
Output:			V_NULL
```

Input: V_NULL
Endereco: 2 Valor: V_INT 6
Endereco: 3 Valor: V_INT 5

Tabela de Procedimentos:

Memoria de Procedimentos:

Console:

8.8. Comando If Then Else

```
programa8 = PROGRAMA (C_DECLARACAO
                        (COM_DECLARACAO
                            (DEC_VAR
                                (VAR "a"
                                    (VALOR (V_INT 1))
                                )
                            )
                            (C_IF_THEN_ELSE
                                (IF_THEN_ELSE
                                    (VALOR(V_BOOL False))

    (C_ATRIBUICAO(ATRIBUICAO "a"(VALOR(V_INT 2))))

    (C_ATRIBUICAO(ATRIBUICAO "a"(VALOR(V_INT 3))))
                                )
                            )
                        )
                    )
```

Versão em C:

```
void main ()
{
    int a = 1;

    if (0)
        a = 2;
    else
        a = 3;
}
```

O programa começa com a declaração de uma variável do tipo inteiro, que é inicializada com o valor 1. O próximo comando é um If-Then-Else, que recebe o valor booleano "False", portanto executa o segundo comando, o qual atribui à variável declarada o valor 3.

Saída:

Tabela de Variaveis:
ID: a Tipo: T_INT Endereco: 2

Memoria de Variaveis:
Output: V_NULL
Input: V_NULL
Endereco: 2 Valor: V_INT 3

Tabela de Procedimentos:

Memoria de Procedimentos:

Console:

8.9. Entrada de Dados

```
programa9 = PROGRAMA (C_DECLARACAO
                        (COM_DECLARACAO
                            (DEC_VAR
                                (VAR "a"
                                    (VALOR (V_INT 1))
                                )
                            )
                            (C_IO
                                (READ "a")
                            )
                        )
                    )
```

Versão em C:

```
void main ()
{
    int a = 1;
    scanf("%d", &a);
}
```

O programa começa com a declaração de uma variável do tipo inteiro, que é inicializada com o valor 1. O próximo comando lê um valor do teclado e atribui a essa variável.

Saída:

Tabela de Variaveis:
ID: a Tipo: T_INT Endereco: 2

Memoria de Variaveis:
Output: V_NULL
Input: V_INT 0
Endereco: 2 Valor: V_INT 0

Tabela de Procedimentos:

Memoria de Procedimentos:

Console:

8.10. Saída de Dados

```
programa10 = PROGRAMA (C_DECLARACAO
                        (COM_DECLARACAO
                            (DEC_VAR
                                (VAR "a"
                                    (VALOR (V_INT 1))
                                )
                            )
                            (C_IO
                                (WRITE (VALOR (V_STRING
"Isto Funciona!")))
                            )
                        )
                    )
```

Versão em C:

```
void main ()
{
    int a = 1;

    printf ("Isto Funciona!");
}
```

O programa começa com a declaração de uma variável do tipo inteiro, que é inicializada com o valor 1. O próximo comando imprime uma string no console.

Saída:

Tabela de Variaveis:
ID: a		Tipo: T_INT	Endereco: 2

Memoria de Variaveis:
Output:		V_STRING "Isto Funciona!"
Input:		V_NULL
Endereco: 2		Valor: V_INT 1

Tabela de Procedimentos:

Memoria de Procedimentos:

Console:
Isto Funciona!

8.11. Tratamento de Erro

```
programa11 = PROGRAMA (C_DECLARACAO
                          (COM_DECLARACAO
                              (DEC_VAR
                                  (VAR "inteiro" (VALOR (V_INT
10)))
                              )

                              (C_ATRIBUICAO
                                  (ATRIBUICAO "inteiro" (VALOR
(V_STRING "string")))
                              )
                          )
                      )
```

Versão em C:

```
void main ()
{
	int inteiro = 10;

	inteiro = "string" //Error!
}
```

O programa começa com a declaração de uma variável do tipo inteiro, que é inicializada com o valor 10. O próximo comando tenta atribuir a essa variável um valor do tipo String.

Saída:

Interface> run programa11

Program error: Associacao invalida de tipos: T_INT, T_STRING - "setValorVariavel"

8.12. Atribuições a Conteúdos de Ponteiros

```
programa12 = PROGRAMA (C_DECLARACAO
            (COM_DECLARACAO
              (DEC_VAR
                (VAR "x" (VALOR (V_INT 1)))
              )

              (C_DECLARACAO
                (COM_DECLARACAO
                  (DEC_VAR
                    (VAR "y" (VALOR (V_INT 9)))
                  )

                  (C_DECLARACAO
                    (COM_DECLARACAO
                      (DEC_PROC
                        (PROC "swap" (PARAM_MULT "p1" (T_PONTEIRO T_INT)
(PARAM_UNICO "p2" (T_PONTEIRO T_INT)))
                        (C_DECLARACAO
                          (COM_DECLARACAO
                            (DEC_VAR
                              (VAR "temp" (CONTEUDO ("p1")))
                            )
                            (C_COMANDO_MULT
                              (C_ATRIBUICAO
                                (ATRIBUICAO_CONTEUDO "p1" (CONTEUDO ("p2")))
                              )

                              (C_ATRIBUICAO
                                (ATRIBUICAO_CONTEUDO "p2" (ID ("temp")))
                              )
                            )
                          )
                        )
                      )
                    )

                    (C_CHAMADA_PROC
```

```
                            (CHAMAR "swap" (EXPRESSAO_MULT (ENDERECO "x")
(EXPRESSAO_UNICA (ENDERECO "y"))))
                          )
                        )
                      )
                    )
                  )
                )
              )
```

Versão em C:

```
void main ()
{
   int x = 1;
   int y = 9;

   void swap(int *p1, int *p2)
   {
      int temp = *p1;
      *p1 = *p2;
      *p2 = temp;
   }

   swap(&x, &y);
}
```

O programa começa com a declaração de duas variáveis do tipo inteiro, as quais são inicializadas com os valores 1 e 9. Em seguida é declarado o procedimento swap, que recebe dois parâmetros do tipo ponteiro pra inteiro. No corpo do procedimento, os conteúdos dos parâmetros são trocados, utilizando operações com ponteiros.

O próximo comando é uma chamada ao procedimento swap, passando como argumentos os endereços das duas variáveis declaradas no início.

Saída:

```
Tabela de Variaveis:
ID: y      Tipo: T_INT    Endereco: 3
ID: x      Tipo: T_INT    Endereco: 2

Memoria de Variaveis:
Output:         V_NULL
Input:        V_NULL
Endereco: 2      Valor: V_INT 9
Endereco: 3      Valor: V_INT 1
```

Tabela de Procedimentos:
ID: swap Endereco: 0

Memoria de Procedimentos:
Endereco: 0 Valor: C_DECLARACAO (COM_DECLARACAO (DEC_VAR (VAR "temp" (CONTEUDO "p1"))) (C_COMANDO_MULT (C_ATRIBUICAO (ATRIBUICAO_CONTEUDO "p1" (CONTEUDO "p2"))) (C_ATRIBUICAO (ATRIBUICAO_CONTEUDO "p2" (ID "temp")))))

Console:

8.13. Loop While com Demonstração de Escopo

```
programa13 = PROGRAMA (

   C_DECLARACAO
      (COM_DECLARACAO
         (DEC_VAR
            (VAR "x"
               (VALOR (V_INT 0))

            )
         )
         (C_DECLARACAO
            (COM_DECLARACAO
               (DEC_VAR
                  (VAR "y" (VALOR (V_STRING "escopo global\n")))
               )
               (C_COMANDO_MULT
                  (C_WHILE (WHILE (EXP_UNARIA (NOT (EXP_BINARIA
                           (IGUAL (ID "x") (VALOR (V_INT 5)))
                        )))


                        (C_DECLARACAO
                           (COM_DECLARACAO
                              (DEC_VAR
                                 (VAR "y" (VALOR (V_STRING "escopo local\n")))
                              )
                              (C_COMANDO_MULT
                                 (C_IO
                                    (WRITE (ID "y"))
                                 )

                                 (C_ATRIBUICAO
                                    (ATRIBUICAO "x"
                                       (EXP_BINARIA
```

```
                                    (ADICAO (ID "x") (VALOR (V_INT 1)))
                                  )
                                )
                              )
                            )
                          )
                        )
                      )
                    )

                    (C_IO
                      (WRITE (ID "y"))
                    )
                  )
                )
              )
            )
          )
```

Versão em C++:

```
void main ()
{
        int x = 0;
        String y = "escopo global\n";

        while(x != 5)
        {
                String y = "escopo local\n";
                printf("%s", y);
                x++;
        }

        printf("%s", y);
}
```

O programa começa com a declaração de duas variáveis, uma do tipo inteiro, e outra do tipo String, as quais são inicializadas com os valores 0 e "escopo global", respectivamente. O próximo comando é um loop while, que é executado enquanto a variável inteira declarada no início for diferente de 5. Dentro do loop é declarada uma variável do tipo String, inicializada com a string constante "escopo local", e com o mesmo identificador da string declarada fora do loop. Isso foi feito com o intuito de demonstrar que um novo escopo é criado, a cada iteração do loop, de forma a permitir as 6 (seis) declarações de variáveis com o mesmo identificador, porém em escopos distintos. Dentro do loop a string "escopo local" é impressa. Encerrado o loop, é impressa a string inicialmente definida.

Saída:

Tabela de Variaveis:
ID: y Tipo: T_STRING Endereco: 3

ID: x Tipo: T_INT Endereco: 2

Memoria de Variaveis:
Output: V_STRING "escopo global\n"
Input: V_NULL
Endereco: 2 Valor: V_INT 5
Endereco: 3 Valor: V_STRING "escopo global\n"

Tabela de Procedimentos:

Memoria de Procedimentos:

Console:
escopo local
escopo local
escopo local
escopo local
escopo local
escopo global

9. Conclusões

Foi apresentada, nesse trabalho, a especificação da linguagem HELL por meio de semântica de ações. Essa linguagem foi desenvolvida pelos autores desse documento, os quais implementaram também um interpretador, escrito em Haskell, para a referida linguagem. A validação desse interpretador foi feita usando os códigos aqui mostrados como exemplos.

A principal contribuição desse trabalho foi, sem dúvida, o ganho em conhecimento, por parte dos autores, no que diz respeito à descrição formal de linguagens de programação, tanto a nível sintático quanto a nível semântico, com ênfase na descrição formal da semântica, através do *framework* semântica de ações. Além disso, vale ressaltar também o ganho em experiência com o uso do paradigma funcional, proporcionado pelo uso da linguagem Haskell para implementação do interpretador.

10. Referências

1. **Santos, Ana Carla e Yi, Jin Jin.** http://www.cin.ufpe.br/~rat/action-notation/directive.html. *Informal Summary of Action Notation.* [Online]

2. **Wildt, Daniel de Freitas.** http://pessoal.facensa.com.br/daniel/files/pl/ConceitosAbstracaoAmarracao.pdf. *Paradigmas de Linguagens de Programação - Conceitos, Abstração e Amarração.* [Online]

3. **Manssour, Isabel Harb.** http://www.inf.pucrs.br/~gustavo/disciplinas/pli/material/paradigmas-aula03.pdf. [Online]

4. **Carlos, Luis e Moura, Hermano.** Semântica de Linguagens de Programação.

5. **Soares, Sérgio C. B.** Semântica de Ações de MiniJava. 1999.

6. **Mosses, Peter D.** System Demonstration. *Action Semantics and ASF+SDF.* s.l. : Elsevier Science B. V., 2002.

7. **Sarinho, Victor Travassos.** Uma biblioteca de componentes semânticos para especificação de linguagens de programação. 2009.

8. **Sebesta, Robert W.** *Concepts of Programming Languages.* s.l. : Adison Wesley.

9. **Scott, Michael Lee.** *Programming language pragmatics.* s.l. : Academic Press, 2000.

10. **Heitor, et al.** http://www.slideshare.net/raquelcarsi/haskell-presentation-910788. *Haskell - Linguagem de Programação Funcional.* [Online]