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.

2009.2

Centro de Informática - CIn

Universidade Federal de Pernambuco

19/11/2009



**Autores:**

Josiane Bezerra Ferreira – jbf2

Luís Pedro de Medeiros Filho – lpmf

Luiz Felipe da Silva Sotero – lfss

Paulo Sérgio Borges de Oliveira Filho – psbof

Sumário

[1. Introdução 5](#_Toc246307715)

[2. Descrição Informal 6](#_Toc246307716)

[2.1. Estrutura do Programa 6](#_Toc246307717)

[2.1.1. Programa 6](#_Toc246307718)

[2.1.2. Comando 6](#_Toc246307719)

[2.2. Atribuições 6](#_Toc246307720)

[2.3. Expressões 7](#_Toc246307721)

[2.3.1. Expressão 7](#_Toc246307722)

[2.3.2. Valor 7](#_Toc246307723)

[2.3.3. Expressão Unária 8](#_Toc246307724)

[2.3.4. Expressão Binária 8](#_Toc246307725)

[2.4. Declarações 9](#_Toc246307726)

[2.4.1. Declaração 9](#_Toc246307727)

[2.4.2. Declaração de Variáveis 9](#_Toc246307728)

[2.4.3. Declaração de Procedimentos 9](#_Toc246307729)

[2.4.4. Lista de Parâmetros 10](#_Toc246307730)

[2.4.5. Comando Declaração 10](#_Toc246307731)

[2.4.6. Tipos 10](#_Toc246307732)

[2.5. Controle de Fluxo – Estruturas de Repetição 11](#_Toc246307733)

[2.5.1. Laço While 11](#_Toc246307734)

[2.5.2. Laço For 11](#_Toc246307735)

[2.5.3. Laço Repeat 11](#_Toc246307736)

[2.6. Controle de Fluxo – Comandos de Decisão 12](#_Toc246307737)

[2.6.1. If Then Else 12](#_Toc246307738)

[2.7. Procedimentos 12](#_Toc246307739)

[2.7.1. Chamada de Procedimentos 12](#_Toc246307740)

[2.7.2. Lista de Expressões 12](#_Toc246307741)

[2.8. Comandos de Entrada e Saída 12](#_Toc246307742)

[2.8.1. O Comando IO 12](#_Toc246307743)

[2.9. O Comando Skip 13](#_Toc246307744)

[3. Sintaxe Abstrata 14](#_Toc246307745)

[3.1. Programas 14](#_Toc246307746)

[3.2. Comandos 14](#_Toc246307747)

[3.3. Atribuições 14](#_Toc246307748)

[3.4. Expressões 15](#_Toc246307749)

[3.5. Declarações 15](#_Toc246307750)

[3.6. Tipos 16](#_Toc246307751)

[3.7. Comandos de Controle de Fluxo 16](#_Toc246307752)

[3.8. Comandos de Entrada e Saída 16](#_Toc246307753)

[3.9. Chamadas de Procedimentos 16](#_Toc246307754)

[4. Entidades Semânticas 17](#_Toc246307755)

[4.1. Dados 17](#_Toc246307756)

[4.2. Valores 17](#_Toc246307757)

[4.3. Tipos 17](#_Toc246307758)

[4.4. Bindings 17](#_Toc246307759)

[4.5. Armazenamento 17](#_Toc246307760)

[4.6. Procedimentos 18](#_Toc246307761)

[5. Funções Semânticas 19](#_Toc246307762)

[5.1. Programa 19](#_Toc246307763)

[5.2. Declarações 19](#_Toc246307764)

[5.3. Expressões 20](#_Toc246307765)

[5.4. Comandos 21](#_Toc246307766)

[6. Sintaxe Léxica 24](#_Toc246307767)

[7. Exemplos de Programas 25](#_Toc246307768)

[7.1. Declaração e Atribuição 25](#_Toc246307769)

[8. O Interpretador 26](#_Toc246307770)

[8.1. Definição 26](#_Toc246307771)

[8.2. Arquitetura 26](#_Toc246307772)

[8.3. Comportamento 27](#_Toc246307773)

[8.3.1. Declarações 27](#_Toc246307774)

[8.3.2. Atribuições 27](#_Toc246307775)

[8.3.3. Estruturas de Repetição 28](#_Toc246307776)

[8.3.4. If Then Else 28](#_Toc246307777)

[8.3.5. Chamadas de Procedimentos 28](#_Toc246307778)

[8.3.6. Entrada e Saída 28](#_Toc246307779)

[8.3.7. Fluxo 28](#_Toc246307780)

[8.4. Implementação 28](#_Toc246307781)

[9. Conclusões 29](#_Toc246307782)

[10. Referências 30](#_Toc246307783)

Índice de Ilustrações

[Figura 1 - Arquitetura do Interpretador 27](#_Toc246308070)

[Figura 2 - Camada Ambiente 27](#_Toc246308071)

[Figura 3 - Fluxo de Execução do Interpretador 29](#_Toc246308072)

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 linguagem *Highly Special Lower 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.

1. Descrição Informal

# Estrutura do Programa

## 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.

## 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.

# Atribuições

### Sintaxe:

Atribuicao ::= 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, não possa ser atribuído à mesma.

# Expressões

## 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.

## 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 ou string, ou pode-se ter ainda o valor *null* usado quando se está trabalhando com ponteiros.

## 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 foi *false*, então o resultado da expressão unária será *true*.

## 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.

# Declarações

## 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.

## 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. Porém, se o conteúdo do ponteiro for 0 (zero), isso significa que trata-se de um ponteiro nulo, ou seja, um ponteiro que não aponta para nenhum endereço de memória.

## Declaração de Procedimentos

### Sintaxe:

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

### Semântica:

Comando que define o escopo de um procedimento e o liga 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 caso de um ponteiro 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.

## 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.

## Comando Declaração

### Sintaxe:

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

### Semântica:

Estabelece a criação de um bloco entre chaves. Dentro do bloco, no início, estariam as declarações de variáveis e procedimentos, 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.

## 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 o 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).

# Controle de Fluxo – Estruturas de Repetição

## 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.

## Laço For

### Sintaxe:

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

### Semântica:

O comando *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. Os comandos são repetidos 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.

## 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 verdadeira, o conjunto de comandos é executado novamente. Esses dois últimos passos se repetem até que o valor da expressão seja falso. Quando isso ocorre o loop é encerrado e os próximos comandos são executados.

# Controle de Fluxo – Comandos de Decisão

## 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.

# Procedimentos

## 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.

## 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.

# Comandos de Entrada e Saída

## 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*.

# 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.

1. 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.

# 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.

# 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)

# Atribuições

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

# 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)

# Declarações

data ComandoDeclaracao = 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)

# Tipos

data Tipo = T\_INT | T\_BOOL | T\_STRING

| T\_PONTEIRO Tipo deriving (Show, Eq)

# 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)

# Comandos de Entrada e Saída

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

# Chamadas de Procedimentos

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

data ListaExpressao = EXPRESSAO\_UNICA Expressao | EXPRESSAO\_MULT Expressao ListaExpressao deriving (Show, Eq)

1. 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 (já mostrada), 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.

# Dados

datum[[1]](#footnote-1) = valor | tipo | procedimento | argumento | token[[2]](#footnote-2)

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

bindable[[3]](#footnote-3) = valor | cell[[4]](#footnote-4)

# Valores

Valor = boolean | int| string

string = list[char]

char = letter | digit.

# Tipos

type = boolean | int | string | ponteiro

# Bindings[[5]](#footnote-5)

Endereco = Int

Id = String

# 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)]

# Procedimentos

Procedimento = abstração[[6]](#footnote-6)

Parametros = valores | Endereco

ListaParametros = lista [Parmetros]

1. 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.

# Programa

exe \_ :: Programa → ação

1. exe [[ Comando:Comando]]

|  |
| --- |
| Executa *Comando* |

# 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 |

1. elabore [[ “pointer” p:Id “=” “^” T:Tipo ]]

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

1. 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* | |

# Expressões

avalie \_ :: Expressão → Ação

1. avalie [[ “true” ]]

|  |
| --- |
| Retorna true |

1. avalie [[ “false” ]]

|  |
| --- |
| Retorna False |

1. avalie [[ Num:Numeral ]]

|  |
| --- |
| Retorna o valor de *Num* |

1. avalie [[ String:String ]]

|  |
| --- |
| Retorna o valor de *String* |

1. avalie [[ Identificador:Id ]]

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

1. avalie [[ “-” Expressao:Expressao ]]

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

1. avalie [[ “!” Expressao:Expressao ]]

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

1. avalie [[ Expressao1:Expressao “+” Expressao2:Expressao ]]

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

1. avalie [[ Expressao1:Expressao “-” Expressao2:Expressao ]]

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

1. avalie [[ Expressao1:Expressao “&&” Expressao2:Expressao ]]

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

1. avalie [[ Expressao1:Expressao “||” Expressao2:Expressao ]]

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

1. 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 | |

1. avalie [[ Expressao1:Expressao “++” Expressao2:Expressao ]]

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

# Comandos

execute\_ :: Comando → Ação

1. execute[[ “{“ Declaracao:Declaracao “;” Comando:Comando “}” ]]

|  |
| --- |
| Elabora *Declaracao* |
| Executa *Comando* |

1. execute [[ Identificador:Id “:=” Expressao:Expressao ]]

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

1. execute [[ “while” Expressao:Expressao “do” Comando:Comando]]



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

1. execute [[ “for” Atribuicao:ComandoAtribuicao “to” Inteiro:ValorInteiro “do” Comando:Comando ]]

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

1. execute [[ “if” Expressao:Expressao “then” Comando1:Comando “else” Comando2:Comando ]]

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

1. execute [[ “repeat” Comando:Comando “until” Expressao:Expressao ]]

|  |  |  |
| --- | --- | --- |
|  | Executa *Comando* | |
| Então | | |
|  | Verifica a expressão *Expressao* é verdadeira | |
|  |  | Executa *Comando* |
|  | Senão | |
|  |  | Fim |

1. execute [[ “write” “(” Expressao:Expressao “)” ]]

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

1. 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* |

1. execute [[ “skip” ]]

|  |
| --- |
| Fim |

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

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

1. execute [[ Comando1:Comando “,” Comando2:Comando ]]

|  |  |
| --- | --- |
| Executa *Comando1* | |
| Executa *Comando2* |

1. 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

1. Exemplos de Programas

# 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))))))

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

Saída:

([([("x",T\_INT,2)],[])],[(2,V\_INT 56)],[])

A saída mostra o ambiente atual, após a execução do programa. Esse ambiente é formado pelo escopo, pela memória de variáveis, e pela memória de procedimentos. No escopo tem-se uma tabela de variáveis e uma tabela de procedimentos.

Pode-se notar que:

* A tabela de variáveis possui a seguinte entrada: "x",T\_INT,2. Onde x é o identificador da variável, T\_INT e o seu tipo, e 2 é o endereço ligado ao identificador x.
* A memória de variáveis possui a entrada “2,V\_INT 56”, onde 2 é o endereço da célula, V\_INT é o tipo do valor que está armazenado nela, e 56 é o valor armazenado.

1. 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.

# 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, a medida que vai sendo 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.

# Arquitetura

(sintaxe, ambiente, variáveis e procedimentos, cálculos, execução, IO, interface)

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

Figura - 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. O código pertencente a essa camada foi mostrado nesse documento, na seção 3.

A próxima camada, denominada Ambiente, engloba todas as estruturas responsáveis por manter o estado do programa que está sendo executado, estando dividida em duas outras subcamadas:

Figura - 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 um dos escopos 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, a 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.

# 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.

## Declarações

No caso de um comando de declaração, de variável ou procedimento, entra em cena a 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.

## 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.

## Estruturas de Repetição

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, seu valor é verificado com auxílio da camada Ambiente. Esses passos se repetem até que o valor da expressão seja “False”.

## If Then Else

Assim como os demais comandos, sua execução começa na camada Execução. 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.

## 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.

## 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**,

## Fluxo

Como pode-se perceber, o fluxo de execução do interpretador segue os passos do diagrama mostrado abaixo:

Figura - Fluxo de Execução do Interpretador

# Implementação

1. 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.

1. 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]

1. Representam ítens de dados. Depende da variedade de informações processadas pelos programas de uma linguagem de programação. [↑](#footnote-ref-1)
2. Uma subcategoria de datum distintos. Depende da variedade de identificadores de uma linguagem de programação. (Geralmente, é um subconjunto de strings). [↑](#footnote-ref-2)
3. Uma subcategoria de dados. Depende da variedade de informações processadas por escopo dos programas de uma linguagem de programação. [↑](#footnote-ref-3)
4. Posição de memória. [↑](#footnote-ref-4)
5. Representam conjuntos de associações entre símbolos (tokens) e dados ligáveis (bindable). [↑](#footnote-ref-5)
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. [↑](#footnote-ref-6)