Documentação do interpretador para LEEBA

Daniel Sehn Colao
João Vitor Venceslau
Lucas Agnez

Julho 2022

1 Introdução

Aspectos gerais da linguagem LEEBA:

- **Público-alvo**: Alunos do ensino médio.
- **Paradigma**: Imperativo.
- **Tipos de dado primitivos**: int, real, string, char e bool.

  O tipo "real"corresponde ao tipo "Float"de linguagens de programação.
- Suporte para tipos de dados abstratos (TADs).

  Sem inclusão de métodos.
- **Listas**: suporte para listas de uma dimensão e duas dimensões (matriz).

  Podem ser declaradas com qualquer tipo, incluindo os TADs também.
- Linguagem interpretada.

  Linguagem utilizada para criar o interpretador: Haskell.
- **Alocação de variável**: Estática e dinâmica.
- **Tipagem**: Estática.
- Linguagem fracamente tipada.

  Possibilidade de coerção e conversão explícita (casting).
- **Verificação de tipos**: Estática.
- Não possui suporte para criação e importação de bibliotecas.
- Não possui sobrecarga de subprogramas, nem subprogramas genéricos.
- **Modos de passagem de parâmetros**: Valor e referência.

  Uso da palavra "ref"para passagem de referência (posicionado antes do parâmetro actual e também antes do tipo de dado do parâmetro formal).

2 Tabelas de Símbolos

2.1 Variáveis

Nesta tabela, são armazenadas as variáveis globais, stack e heap do programa.

- Escopo (Global)

  Propósito: Chave de busca nesta tabela. Trata-se do endereço da variável.

  Tipo: String.

  Formato: global + . + identificador da variável.

  Tempo de Vida: Execução do programa.

Escopo	Tipo Valor	Constante
global.0.var	int 6	false
funcSoma.2.resultado	real 4.5	false
main.0.pi	real 3.14	true
heap.0	string “Daniel”	false
main.1.teste	ref “string” “heap.0”	false

- Escopo (Stack)

  Propósito: Chave de busca nesta tabela. Trata-se do endereço da variável.

  Tipo: String.

  Formato: identificador do subprograma + . + escopo interno do subprograma + . + identificador da variável

  Tempo de Vida: a variável é removida desta tabela após o término da execução do subprograma onde foi declarada.

- Escopo (Heap)

  Propósito: Chave de busca nesta tabela. Trata-se do endereço da variável.

  Tipo: String.

  Formato: heap + . + identificador do subprograma + . + escopo interno do subprograma + . + identificador da variável

  Tempo de Vida: A variável é desalocada, removida desta tabela, mediante uso do operador “destroy”.

- Tipo Valor

  Propósito: Armazenar o tipo de dado juntamente com o valor armazenado na variável.

  Formato: Data Type + Valor armazenado

- Constante

  Propósito: Indica se o valor armazenado é constante.

  Tipo: Booleano.

2.2 Tabela de Structs

Nesta tabela são armazenados os tipos de dados abstratos definidos pelo usuário da linguagem.

Nome da Struct	Lista de Campos
Aluno	string Nome; int Idade;
intPair	int first; int second;
intQueue	int[10] queue; int first; int last;

- Nome da Struct

  Propósito: Armazenar o nome do tipo de dado abstrato definido pelo usuário.

  Tipo: String.

- Lista de Campos

  Propósito: Armazenar os atributos do tipo de dado abstrato definido pelo usuário.

  Tipo: Sequência de tokens.

2.3 Tabela de Funções e Procedimentos

Nesta tabela são armazenadas as funções e procedimentos criados pelo usuário da linguagem.

- Identificador

  Propósito: Identificar o subprograma.

  Tipo: String.

Identificador	Tipo de Retorno	Lista de Parâmetros	Lista de Instruções
funcSoma	real	real A, real B	return A + B;
algumProcedimento	∅	int ano, int idade	int nascimento = ano - idade; print(“Voce nasceu no ano: ”+ nascimento);

popQueue	int	intQueue fila	int element = fila.queue[first];
			fila.first = fila.first - 1;
			return element;

- Tipo de Retorno

  Propósito: Guardar o tipo de dado retornado pela função. Caso seja um procedimento, então este campo estará vazio.

  Tipo: String.

- Lista de Parâmetros

  Propósito: Guardar os parâmetros formais do subprograma.

  Tipo: Sequência de tokens.

- Lista de Instruções

  Propósito: Guardar as instruções do subprograma.

  Tipo: Sequência de tokens.

3 Design da implementação

3.1 BNF

```
<program> := <global_vars> <declarations>

<global_vars> :=  λ | begin global : <var_declarations> end global;

<declarations> := <declaration> | <declaration> <declarations>

<declaration> := begin <struct_declaration>
                | begin <subprograms_declaration>
                | begin <function_declaration>

<var_declarations> := <var_declaration> ; | <var_declaration> ; <var_declarations>

<struct_declaration> := struct struct_id : <field_declarations> end struct_id;

<field_declarations> := <data_type> id; | <data_type> id; <field_declarations>

<subprograms_declaration> := id ( <params> ): <statements> end id;

<function_declaration> := <data_type> <subprograms_declaration>

<params> :=  λ | <param> | <param> , <params>

<param> := <data_type> id

<return> := return <expression> | <ref_ini>

<ref_ini> := ref <maybe_var>

<maybe_var> := <function_call> | <array_acess> | id<dot_acess>

<dot_acess> := λ | .id <dot_acess>
```

```
<statements> := <statement> ; | <statement> ; <statements>

<statement> := <print>
             | newline
             | destroy id
             | <conditional>
             | <subprogram_call>
             | <array_modification>
             | <var_declaration>
             | <assignment>
             | <loop>
             | <return>

<array_modification> := id[<expression>]<subscript> = <expression>
                      | id[<expression>]<subscript> = <ref_ini>
                      | id[<expression>]<subscript> = read
                      | id[<expression>]<subscript> = create <data_type>

<array_acess> := id[<numeric_apression>]<subscript>

<subscript> := [<expression>] |  λ

<print>:= print(<expression>)

<subprogram_call> := id(<args>)

<args> := λ | <ref_ini> | <ref_ini> , <args> | <expression> | <expression> , <args>

<casting> :=  (int) | (real) | (bool) | (string) | (char)

<expression> := <numeric_expression> | <logic_expression>
              | <string_expression> | <casting> <expression>

<numeric_expression> := <term> <eval_remaining>

<eval_remaining> := + <term> <eval_remaining> | - <term> <eval_remaining>

<term> := <factor> * <term> | <factor> / <term> | <factor> % <term> | <factor>

<factor> := <literal> | <maybe_var> | - <factor> | (<numeric_expression>)

<logic_expression> := <logic_term>  or <logic_expression> | <logic_term1>

<logic_term1> := <logic_term2> and <logic_term1> | <logic_term2>

<logic_term2> := not <logic_factor> | <logic_factor>

<logic_factor> :=  <comparison>
                | bool_literal
                | <maybe_var>
                | (<logic_expression>)
                | NULL <maybe_var>

<comparison> := <expression> == <expression>
              | <expression> != <expression>
              | <numeric_expression> > <numeric_expression>
              | <numeric_expression> < <numeric_expression>
              | <numeric_expression> <= <numeric_expression>
              | <numeric_expression> >= <numeric_expression>

<string_expression> := <string_factor> + <string_expression> | <string_factor>
```

```
<string_factor> :=  string_literal | char_literal | <maybe_var>

<conditional> := begin if ( <logic_expression> ): <statements> end if
                    | begin if ( <logic_expression> ): <statements> end if
                  begin else: <statements> end else

<loop> := begin while (<logic_expression >): <statements> end while

<var_declaration> := <data_type> id
                   | <data_type> id = <expression>
                   | <data_type> id = <ref_ini>
                   | <data_type> id = read
                   | <data_type> id = create <data_type>

<assignment> := id<dot_acess> = <expression>
              | id<dot_acess> = <ref_ini>
              | id<dot_acess> = read
              | id<dot_acess> = create <data_type>

<data_type> := <type>
              | id
              | constant <type>
              | constant id
              | ref <type>
              | ref id
              | <type>[integer_literal]
              | <type>[integer_literal][integer_literal]

<type> := int | real | bool | char | string

<literal> := <numeric_literal>  | char_literal | string_literal | bool_literal

<numeric_literal> := real_literal | integer_literal
```

3.2 Transformação do código-fonte em unidades léxicas

No arquivo **lexer.x**, declaramos todos os tipos de tokens da nossa linguagem, incluindo tokens para palavras reservadas, identificadores e literais. O arquivo de código fonte, cujo formato é ".leeba", é lido caractere-a-caractere para construir tokens.

As palavras reservadas possuem um mapeamento único, por exemplo o termo "int"é mapeado unicamente para o token Int, representando o tipo de dado inteiro. Os identificadores e literais possuem uma expressão regular associada, que é utilizada para construir diferentes lexemas.

A tabela abaixo mostra a expressão regular de identificadores e cada tipo de literal da linguagem LEEBA.

Token	Expressão Regular Associada
Id	$\$alpha[\$alpha\$digit\_]^*$
Int	$\$digit+$
Real	$\$digit + .\$digit+$
Char	\'[^\']\'
String	\"[^\"]*\"

Qualquer lexema construído caractere-a-caractere que esteja de acordo com a expressão regular $alpha[\$alpha\$digit\_]^*$, sendo *alpha* uma letra do alfabeto, será mapeado para o token Id. Esse token será manipulado pelo interpretador da linguagem LEEBA.

3.3 Tratamento de estruturas condicionais e de repetição

Para o tratamento de laços de repetição, implementamos um interpretador recursivo, isto é, após interpretar todos os tokens relacionados ao comando de repetição while, é utilizada a função getInput para obter os próximos

tokens a serem lidos, o resultado é armazenado em uma variável temporária, em seguida utilizamos da função setInput para colocar como próximos tokens as serem interpretados os tokens que acabaram de ser obtidos ao fim do parser do while. Realizamos outra chamada ao parser do while e ao fim dela utilizamos novamente o setInput, porém com os tokens salvos inicialmente com o getInput para continuar o fluxo de execução do programa. Porém esse procedimento só é realizado caso a condição do while seja satisfeita, caso contrário, os tokens são lidos, porém nenhuma interpretação ocorre, finalizando o parser normalmente.

Enquanto que para o tratamento dos condicionais, apenas é verificada se a condição é satisfeita ou não, caso seja, os tokens dos comandos internos são interpretados, caso contrário, apenas lidos. Se for o caso do IF ter uma cláusula ELSE, a mesma é interpretada caso a condição não seja satisfeita, e caso contrário, os tokens são apenas lidos.

3.4 Tratamento de subprogramas

O tratamento de subprogramas é dividido entre dois parsers, um para armazenar o subprograma, incluindo seu identificador, tipo de retorno, parâmetros e instruções, e outro responsável pela execução. O primeiro parser além de armazenar o subprograma, também realiza validações relacionadas ao tipo de retorno e tipos dos parâmetros, assim como compatibilização desses.

O parser responsável pela execução do subprograma, ao identificar a chamada do mesmo com argumentos válidos, utiliza a função getInput, da linguagem Haskell, para armazenar os tokens após a chamada do subprograma e, em seguida, a função setInput para preparar os tokens das instruções do subprograma. Após isso, é chamado o parser de execução de instruções. Quando a execução das instruções do subprograma terminar, é utilizada novamente a função setInput para voltar ao fluxo de tokens do momento em que a chamada de função foi identificada.

3.5 Verificações realizadas

- Todas as variáveis precisam ser declaradas antes de usadas. Caso não sejam, um erro será emitido
- Ocorre a verificação para que não hajam duas variáveis declaradas com o mesmo nome no mesmo escopo
- É permitido que variáveis de escopos distintos não aninhados tenham o mesmo identificador
- Não é permitido que variáveis de escopos distintos tenham mesmo identificador, caso os escopos estejam aninhados
- As coerções permitidas são testadas e aplicadas se necessário. São elas:
  - real para int: realizando truncamento
  - int para real: realizando alargamento
  - char para string: encapsulando em uma lista
- É apontado um erro caso uma função seja chamada com mais/menos parâmetros que o declarado
- É apontado um erro caso os tipos dos argumentos não sejam compatíveis com os tipos dos parâmetros informados em chamadas de subprogramas.
- É apontado um erro caso não exista um comando de retorno na declaração de uma função.

4 Instruções de uso do interpretador

Após realizar o download do código-fonte, e instalar o haskell e o cabal, basta executar o seguinte comando:
cabal v2-run LPCP-LanguageProject – pathToFile.leeba

4.1 Instruções - LEEBA

Abaixo serão listados os comandos disponíveis para uso na linguagem LEEBA.

4.1.1 Definição de Subprogramas

```
begin <type> func_name(parametros): //funcao
    statements;
    return expression;
end func_name;

begin proc_name(parametros): //procedimento - nao ha tipo de retorno
    statements;
end func_name;
```

4.1.2 Chamadas de Procedimento e de Função

```
procedimento(params); //chamada de procedimento
<type> a = funcao(params); //chamada de funcao
```

4.1.3 Declaração de Structs (TADs)

```
begin struct struct_name:
    <type> id_1;
    ...
    <type> id_n;
end struct_name;
```

4.1.4 Declaração de Variáveis

```
<type> var_id;
<type> var_id = <expression>; //inicializacao
ref <type> var_id = create <type>; //alocacao na heap - inicializacao com valor padrao
ref <type> var_id = ref var_id;
constant <type> var_id = <literal>; //constante
```

4.1.5 Declaração de Arrays e Matrizes

```
<type>[integer_literal] var_id; //array 1D
<type>[integer_literal][integer_literal] var_id; //matriz
```

4.1.6 Desalocação de memória heap

```
destroy var_id;
```

4.1.7 Leitura e Escrita - IO

```
print(expression); //impressao na tela
var_id = read; //leitura de entrada
newline; // quebra de linha
```

4.1.8 Laços de Repetição

```
begin while(expression):
    statements;
end while;
```

4.1.9 Condicionais

```
begin if(expression):
    statements;
end if;
```

```
begin if(expression):
    statements;
end if;
begin else:
    statements;
end else;
```