MAC 316 – Conceitos Fund. de Linguagens de Programação

Prof: Ana C. V. de Melo
Monitor: Cássio A. Cancio
Trabalho 2 – Interpretador: modificação do if – expressões relacionais
Número de integrantes da equipe: 5 (máximo)
Prazo de Entrega: até dia 15/12/2023

Interpretador - programação funcional

Considere uma linguagem funcional simplificada (LFSimp) definida informalmente como segue (sintaxe concreta - disponível na página da disciplina, `V2_LFSimp_trabalho_2023`) . No primeiro trabalho vocês utilizaram uma linguagem funcional simples (disponível na página da disciplina, `V1_LFSimp_trabalho_2023`) e posteriormente modificadaram a sintaxe/interpretador para só permitir identificadores com certas características e um `if` que considerava a condição com apenas os valores booleanos, `true` ou `false`. No trabalho atual a condição do `if` poderá ser um valor booleano ou uma expressão relacional, a ser inserida na nova linguagem. Para isso, teremos novas expressões na linguagem (expressões relacionais) que poderão apenas ser utilizadas como condições do `if`. A sintaxe da versão 2 (V2) da linguagem LFSimp irá incluir valores booleanos e expressões relacionais.

Aqui utilizamos uma EBNF que pode ser submetida a

`https://bnfplayground.pauliankline.com/`

para que vocês possam ver o que são sentenças bem formadas na linguagem.

```
<character>     ::= <letter> | <digit> | <symbol>
<letter>        ::= [a-z] | [A-Z]
<digit>         ::= [0-9]
<intnum>        ::= <digit>+
<number>        ::= <intnum> | <intnum> "." <intnum>
<bool>          ::= "true" | "false"
<other_symb>    ::=  "_" | "!" | "?" | "#"
<reserv_symb>   ::=   "(" | ")" | "+" | "-" | "*" | "~" | "%" |
                      "<" | ">" | "<=" | ">=" | "==" | "!="
<symbol>        ::=  <reserv_symb> | <other_symb>
<reserv_word>   ::=  "cons" | "head" | "tail" |
                     "if" | "let" | "letrec" |
                     "lambda" | "call" |
                     <bool>

<reservedW>      ::=  <reserv_symb> | <reserv_word>

/* id a ser modificado - começa com letra, seguida de letras e dígitos e não é uma palavra reservada*/
/* já realizado no primeiro trabalho */

<id>             ::=  (<character>)+


/* -- Operadores Relacionais -- */
<eq_op>  ::= " == " | " != "
<cmp_op> ::= " > " | " >= " | " < " | " <= "
<op_rel> ::= <eq_op> | <cmp_op>
```

```
/* -- Expressões Relacionais -- */
<rel_expr> ::= "(" <op_rel>  <arith_expr> " " <arith_expr> ")"


/* -- Operadores Aritméticos -- */
<op_arith_bin> ::= " + " | " - " | " * " | " % "
<op_arith_un>  ::= " ~ "

/* -- Expressões Aritméticas -- */
<arith_expr> ::= "(" <exprA_bin> ")" | "(" <exprA_un> ")" | <number>
<exprA_bin>  ::= <op_arith_bin> <code> " " <code>
<exprA_un>   ::= <op_arith_un> <code>


/* -- Operadores Listas -- */
<op_list>      ::= "head " | "tail "

/* -- Expressões sobre listas -- */
<list_expr>  ::= <cons> | "(" <exprL>  ")"
<cons>       ::= "(cons " <code> " " <code> ")"
<exprL>      ::= <op_list>  <code>


/* -- Expressões Lambda (definição e aplicação de funções) -- */
<lamb_expr> ::=  <lambda> | <call>
<lambda>    ::= "(lambda " <param> " "  <code> ")"
<param>     ::= <id>
<call>      ::= "(call " <lambda> " " <code> ")"

/* -- expressões if -- */
<if>    ::= "(if " <cond>  " " <pos>  " " <neg> ")"
<cond>  ::= <rel_expr> | <bool>
<pos>   ::= <code>
<neg>   ::= <code>

/* -- expressões let  -- */
<let_expr>  ::=  <let> | <letrec>
<let>       ::= "(let " <id> " "  <def>  " " <body> ")"
<letrec>    ::= "(letrec " <id> " "  <lambda>  " " <body> ")"
<def>       ::= <code>
<body>      ::= <code>


/* -- Código do programa -- */
<code>  ::=  <expr> | <number>

<expr>  ::=  <arith_expr> | <list_expr> | <lamb_expr> | <if> | <let_expr>
```

Essa linguagem não possui declaração de tipos e cada um dos operadores aritméticos é aplicado a expressões. A verificação da compatibilidade dos tipos é realizada mediante a aplicação dos operadores, se inteiros, listas ou outras expressões (em tempo de execução das expressões).

As listas são construídas com o construtor `cons`, e as operações `head` e `tail` só podem ser aplicadas a listas construídas na linguagem. Cada lista é construída com pelo menos 2 elementos (vejam os vários testes sugeridos,

junto ao código do interpretador).

Além das listas, a LFSimp contém expressões aritméticas, expressões lambda, expressões `if` e expressões `let` e `letrec`. Estes últimos definem nomes que podem ser associados a quaisquer valores de expressões disponíveis na linguagem de forma simplificada ou recursiva, respectivamente.

A execução de um programa é dada pela avaliação da expressão. No interpretador, essa avaliação é precedida por outras tarefas, como descrito no código correspondente (vejam os comentários no código do interpretador):

```
interp = eval . desugar . analyze . parse . tokenize
```

Para a sintaxe aqui definida, alguns elementos são permitidos na escrita das expressões mas podem ser problemáticos na execução (erro na avaliação da expressão). Por exemplo, sob o ponto-de-vista sintático, uma expressão aritmética admite que cada um dos operandos seja uma expressão qualquer da linguagem, mas na avaliação (execução) a operação só pode ser realizada sobre números. Nesses casos, o erro só será apontado na avaliação da expressão. Isso também acontece com outras operações sobre listas... (vejam nos testes sugeridos). Isso significa que parte da verificação de tipos é realizada em tempo de execução das expressões.

As Suas Tarefas:

Antes de iniciar as tarefas aqui definidas, será preciso entender no código como cada uma das tarefas é realizada. Com isso vocês vão entender onde é realizada cada tarefa do interpretador, inclusive em que passo são dadas as mensagens de erro quando determinados programas são executados.

1. **Formação dos identificadores - já realizada no Trabalho 1:** No primeiro trabalho vocês definiram uma nova gramática para os identicadores. Nela, um identificador deve sempre iniciar por uma letra, seguida de letras e números, e **não pode ser** uma palavra reservada da linguagem (ex.: `let, if,...` – as palavras reservadas estão todas em letras minúsculas).

    1. Gerar uma nova gramática: a atual no arquivo `V2_LFSimp_trabalho_2023` junto com as modificações que vocês já fizeram no Trabalho 1 para os identificadores.
    2. Rever o interpretador que vocês implementaram no primeiro trabalho e corrigir caso seja necessário.

2. **Condição do `if` é um valor booleano ou uma expressão relacional:** Na nova gramática, observem que a condição do `if` pode ser um valor booleano ou uma expressão relacional. Dessa forma, o `if` será seguido da condição (valor booleano ou expressão relacional), seguido do código para quando a condição tem o valor verdadeiro, e do código para quando a condição tem o valor falso. As expressões `if` poderiam ser por exemplo:

    ```
    (if true (+ 1 2) (+ 10 20))
    (if (> 9 8) (+ 1 2) (+ 10 20))
    (if false (+ 1 2) (let addnum (+ 12 13) (+ addnum 5)))
    (if (!= 5 12) (+ 1 2) (let addnum (+ 12 13) (+ addnum 5)))
    (if (<= ( + 92.7 07) (call (lambda z (+ 0.8 z)) 5)) 5.2 49.19)
    ```

    1. Modificar o código do interpretador (o código já modificado por vocês no primeiro trabalho) para acomodar o novo `if`. Vejam no interpretador onde os tipos são definidos, e como certificar que a condição do `if` é um valor booleano ou uma expressão relacional. Vocês precisarão modificar praticamente todos os passos do interpretador.

O que está sendo fornecido: Um arquivo .txt com a gramática da linguagem atual e o código do interpretador original. Vocês devem usar o código do interpretador já modificado por vocês no primeiro trabalho.

O que vocês devem entregar:

1. Uma nova gramática usando a gramática atual e as modificações que vocês já fizeram no primeiro trabalho.
2. O código do interpretador modificado com os itens das tarefas a serem realizadas. **Não mude os nomes dos arquivos e nem das funções já implementadas** pq vamos fazer os testes com esses nomes. Se achar pertinente, acrescente novas funções, tipos...

Como o seu EP será avaliado:

1. Vamos submeter a nova gramática ao software sugerido para verificar as expressões geradas/aceitas.
2. Vamos submeter o seu código a um conjunto de testes, tanto para o que já funciona quanto para os novos itens solicitados no trabalho atual. Claro que vamos olhar o código modificado também.