MAC 316 – Conceitos Fund. de Linguagens de Programação

Prof: Ana C. V. de Melo
Monitor: Cássio A. Cancio
Trabalho 1 – Interpretador: modificação de identificadores e if
Número de integrantes da equipe: 5 (máximo)
Prazo de Entrega: até dia 15/11/2023

Interpretador - programação funcional

Considere uma linguagem funcional simplificada (LFSimp) definida informalmente como segue (sintaxe concreta). 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.

```
Simbolos/palavras da linguagem : (, ), +, -, *, ~, lambda, call,
                                 if, let, letrec, cons, head, tail

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

/* -- 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>  ::= <code>
<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. Para ver como funcionam as expressões da linguagem, leiam os comentários no `Readme.md` e no código disponível, e façam pelo menos os testes sugeridos nos arquivos: `testes_basicos_LFSimpl.txt` e `Tests.hs` (vejam como executar os testes nos próprios arquivos).

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, vocês devem ler o código fornecido e executar os testes. Os itens abaixo não vão fazer sentido para vocês antes disso. É 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:** Um dos problemas na linguagem é quando definimos identificadores nas expressões `lambda`, `let` e `letrec` que são números, ou palavras/símbolos reservados à linguagem. Nesses casos, as expressões podem ficar confusas, ou ainda gerar resultados errados ao esperado (vide os testes sugeridos). Para eliminar esses tipos de problemas, 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). Por exemplo, os seguintes identificadores **não são aceitos** pela linguagem: `let, lambda, 34, 5as, 2f, 23call`. Por outro lado, os seguintes identificadores **são aceitos** pela linguagem: `le, letr, Let, call2, LamBda, lamb, i, i2, if1, ifif, IF`.

   obs: no item 2 do trabalho, teremos as palavras reservadas `true` e `false` a serem utilizadas como valores booleanos. Além das outras palavras reservadas da linguagem, essas duas também não poderão ser usadas como identificadores.

   1. Gerar uma nova gramática para os identificadores: devem começar por uma letra do alfabeto, seguida de letras ou números em qualquer ordem, excluindo qualquer palavra reservada da linguagem. Dica: modifique a gramática fornecida para atender a esses requisitos e submeta ao `https://bnfplayground.pauliankline.com/` para ir aprimorando as suas definições.

      Vocês deverão entregar essa nova gramática porque iremos submeter ao software para verificar quais identificadores estão sendo gerados.
   2. A segunda tarefa neste item será modificar o código do interpretador fornecido para que os identificadores respeitem essas novas regras de formação. Vejam que uma vez que a gramática só admite esses novos identificadores, qualquer outro tipo de identificador deverá ser rejeitado no parser da linguagem.

2. **Condição do `if` é um valor booleano:** Na forma atual da sintaxe, o condição do `if` é um valor numérico (0 representa o falso e qq outro número representa o verdadeiro). Isso pode ser confuso para vários programadores que esperam um valor booleano na condição do `if`. Neste item, vocês irão criar os valores booleanos para a linguagem (`true` e `false`) e usá-los na condição do `if`. Dessa forma, o `if` será seguido da condição com valor booleano, seguido do código para quando a condição tem o valor `true`, e do código para quando a condição tem o valor `false`. As expressões `if` serão da forma:

   ```
   (if true (+ 1 2) (+ 10 20))
   ```

   ```
   (if false (+ 1 2) (let addnum (+ 12 13) (+ addnum 5)))
   ```

   Vejam que a condição será sempre um valor booleano fixo (`true` ou `false`), mas o restante da expressão continua como na versão atual da linguagem (qualquer código da linguagem). Vejam que esse é apenas um passo inicial para depois criarmos expressões lógicas para as decisões dos ifs (por enquanto fica bem restrito).

   1. Gerar uma nova gramática para o `if`, como descrito acima. Observem que um novo tipo `bool` deve ser criado, junto com a gramática do `if` modificada. Vocês deverão entregar essa nova gramática porque iremos submeter ao software para verificar se as expressões `if` estão sendo geradas de forma correta.
   2. A segunda tarefa neste item será modificar o código do interpretador fornecido para acomodar o novo `if`. Vejam no interpretador onde os tipos são definidos, e como certificar que a condição do `if` contém apenas um dos valores booleanos (`true` ou `false`).

O que está sendo fornecido: Um arquivo .txt com a gramática da linguagem atual, o código do interpretador que funciona para a linguagem atual, e dois conjuntos de testes.

O que vocês devem entregar:

1. Uma nova gramática contendo os dois itens do trabalho (identificadores e o novo `if`). Entreguem apenas um arquivo contendo os dois itens.
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.