LINGUAGENS E PARADIGMAS DE PROGRAMAÇÃO
TRABALHO EM GRUPO
2023/2

O trabalho se baseia na implementação de um interpretador de uma linguagem simples, chamada de *BNL – Bruno's NoSense Language*.

1. Histórico de Mudanças no Documento

Versão 4.0	• Alteração da linguagem para uso de escopo dinâmico.
Versão 3.0	• Alteração do nome da função principal do interpretador, de "process" para "execute" • Adição de "!" como terminador do "delete" • Remoção do "!" dos comandos do "if" • Indicação de que os comandos de modificação não atuarão sobre funções ativas • Correção de exemplos que estavam sem "!" no final do comando
Versão 2.0	• Alteração da data de entrega • Alteração no nome do arquivo de entrega
Versão 1.0	• Versão inicial do trabalho

2. Regras do Trabalho

- A data de entrega do trabalho será no dia 17 de dezembro de 2023, 23h59**.**
- O trabalho deve ser feito em grupo de 3 alunos.
- Os grupos deverão apresentar o trabalho para o professor, em horários agendados.
  - Caso o grupo não apresente, o trabalho terá nota zero.
- Qualquer plágio (total ou parcial) implicará em nota zero para todos os envolvidos.
- O interpretador deve ser escrito em Scheme (Racket) e será testado no ambiente Linux.
- Não é permitido o uso de bibliotecas auxiliares ou qualquer outra estrutura da linguagem que não foi vista em sala de aula.
  - A exceção é o condicional “cond”. Caso alguém queira usar, está liberado.
  - Na dúvida, consulte o professor antes de usar.
- O trabalho deve ser em um arquivo chamado “bnl.rkt” e deve ter a função principal “process”, que recebe um único parâmetro (o programa a ser interpretado). Por exemplo:

bnl.rkt

```
# lang racket

;
; Outras funções auxilizares aqui...
;

(define process (lambda (code)
   ; Seu código aqui...
))
```

- Assuma que todo o programa que você for interpretar estará correto, ou seja, não precisa ficar verificando o formato.
- Caso omissos devem ser tratados com o professor.
  - Não tente assumir qualquer coisa se estiver com dúvida.

3. Descrição da Linguagem

A linguagem é baseada na definição de funções (uma ou várias). A função “main” é o onde o programa começa. Ela é uma função que não recebe nenhum parâmetro.

3.1. Valores, Variáveis e Parâmetros

A linguagem possui dois tipos de valores: números e string. Os números podem ser inteiros ou ponto flutuante e são expressos pelos literais na base 10 (por exemplo: 5, 6.4, -42). As string são a cadeia de caracteres entre aspas duplas (Por exemplo: “hello world”).

Não há variáveis globais na linguagem, apenas variáveis locais e parâmetros. As variáveis e parâmetros de funções só podem ser do tipo número.

As strings são usadas apenas na função *print*(), para interação com o usuário.

3.2. Função

A definição de função tem a seguinte forma:

```
<func_def> → <protocol> <func_body>
           | <protocol> <var_def> <func_body>

<protocol> → function <name>  '<' '>'
           | function <name>  '<' <names> '>'

<var_def> → vars <names>

<names> → <name>
        | <name> <names>

<func_body> → begin <commands> end
```

Considere que um nome (*name*) é qualquer sequência de caracteres de “a” até “z” — isso mesmo, somente minúsculos, sem número.

O protocolo da função define seu nome e sua lista de parâmetros (se houver). Uma função pode ter ou não definição de variáveis locais. A definição de variáveis locais inicia com a palavra-chave “vars” seguida de uma lista de variáveis (inicializadas com zero). O corpo de uma função começa com a palavra-chave “begin” e termina com a palavra-chave “end”. Uma função sempre tem que terminar com um comando “return” como última linha do seu corpo (ou seja, corpo nunca será vazio).

Note que a definição utiliza parênteses angulares “< >”. Os parâmetros e as variáveis locais são uma lista de nomes separados por espaço.

Exemplos da definição de funções:

```
function process < >
vars x y
begin
  ; corpo da função omitido
end

function aux < a b >
begin
  ; corpo da função omitido
end
```

Não haverá parâmetros e variáveis com o mesmo nome na mesma função. Não haverá parâmetro ou variável com o mesmo nome da função.

3.3. Comandos

O corpo da função é um conjunto de comandos que podem ser (i) chamada de função, (ii) returno de valores, (iii) atribuição, (iv) condicional simples, ou (v) modificação de código.

```
<commands> → <command>
           | <command> <commands>

<command> → <func_call> !
          | <return> !
          | <attribution> !
          | <conditional>
          | <code_modification>
```

O símbolo ‘!’ é o terminador de comando (como o ‘;’ na linguagem C).

3.3.1. Chamada de Função

A chamada de função é feita invocando o nome da função e passando sua lista de parâmetros (que podem ser números, variáveis ou parâmetros). Note que são utilizados parênteses angulares para delimitar a lista de argumentos (“< >”).

```
<func_call> → <name> '<'  '>'
            | <name> '<' <numeric_values> '>'

<numeric_value> → <number>
                | <name>
```

Qualquer valor retornado pela função deve ser descartado. Por exemplo:

```
f < b 5.5 > !
g < > !
```

3.3.2. Retorno de Valores

Uma função pode retornar um ou mais valores numéricos (números, variáveis ou parâmetros) usando o comando “return”. Destacando que toda função deve ter como último comando um retorno de valor:

```
<return> → return <numeric_values>
```

3.3.3. Atribuição

A atribuição pode ser de três tipos:

- Atribuição simples
- Atribuição com expressão
- Atribuição com chamada de função

```
<attribution> → <attr_simple>
              | <attr_expr>
              | <attr_func>

<attr_simple> → <name> = <numeric_value>

<attr_expr> → <name> = <numeric_value> <op> <numeric_value>

<attr_func> → <names> = <func_call>

<numeric_values> → <numeric_value>
                 | <numeric_value> <numeric_values>

<op> → + | - | * | /
```

Atribuição simples: Comando onde um parâmetro ou variável recebe um número ou o valor de outro parâmetro/variável. Por exemplo:

```
a = 10 !
b = 15.5 !
a = b !
b = -53.3 !
```

Atribuição com expressão: Comando onde um parâmetro ou variável recebe o resultado de uma operação de soma, subtração, multiplicação e divisão. Os argumentos das operações podem ser números, parâmetros ou variáveis. Por exemplo:

```
a = 1 + 5.7 !
b = 2.67 - a !
c = b * a !
d = c + -6 !
```

Atribuição com chamada de função: Comando onde uma lista de parâmetros e/ou variáveis recebe o retorno de uma chamada função. Como a função pode retornar mais de um valor, a atribuição é feita da seguinte forma: o primeiro valor de retorno é atribuído ao primeiro parâmetro ou variável, o segundo valor de retorno é atribuído ao segundo parâmetro ou variável, etc. Por exemplo:

```
a = f < b 5.5 > !
a b c = g < > !
```

Deve-se aplicar um ajuste no caso da função retornar mais ou menos valores do que o esperado. Se a função retornar mais valores do que o número de receptores (variáveis ou parâmetros), os valores excedentes devem ser ignorados. Se a função retornar menor valores, os receptores (variáveis ou parâmetros) que não tiveram valores atribuídos devem receber 0 (zero).

3.3.3. Condicional

O “if” é a estrutura condicional da linguagem, mas ela não possui a parte “eles”:

```
<conditional> → if <if_test> then <if_command> fi

<if_test> → <numeric_value> <rel_op> <numeric_value>

<rel_op> → lt | le | gt | ge | eq | ne

<if_command> → <func_call>
             | <attribution>
             | <return>
```

O teste pode ser a comparação de dois valores numéricos (número, variável ou parâmetro) com os operadores:

- lt: menor
- le: menor ou igual
- gt: maior que
- ge: maior que ou igual
- eq: igual
- ne: não igual

O condicional “if” só pode um único comando em seu corpo, sendo possível: (i) chamada de função, (ii) atribuição, ou (iii) retorno de valores. Por exemplo:

```
if a gt -10.5 then a b = f < c 14 e > fi
```

3.3.4. Modificação de Código

As duas diretivas de alteração de código permitem substituir um comando por outro, ou apagar um comando do corpo de uma função.

```
<code_modification> → <code_update>
                    | <code_delete>

<code_update> → update <name> @ <integer> : <command>

<code_delete> → delete <name> @ <integer> !
```

Os comandos de modificação de código atuam de forma dinâmica, ou seja, eles precisam ser executados para afetar o código, ou seja, eles não são avaliados em tempo de compilação.

Para simplificar o trabalho, considere que os comandos de modificação de código não irão ser aplicados em funções que estão ativas. Assuma que os programas que o seu interpretador receberá já estarão seguindo essa premissa e que não há necessidade de verificar isso.

Substituição de código: O comando “update” altera o corpo da função informada substitui a linha de comando pelo conteúdo que aparece após o “:”. A função é identificada pelo seu nome e o comando por número inteiro, por exemplo, “1” significa primeiro comando do corpo, “2” o segundo comando, e assim por diante. Por exemplo:

```
function f < a b >
begin
   a = 2 !
   a = 5.0 + b !
   return 0 !
end

function main < >
begin
   f < 1 2 > !
   update f @ 2 : b = a - 5 !
   update f @ 1 : if a gt b then a = 0 fi
   f < 5 -2 > !
end
```

No exemplo acima, a função “main” executa a função “f” com o corpo original, então troca o comando “update” substitui “a = 2 !” por “b = a - 5 !”, depois o segundo comando “update” substitui “a = 5.0 + b !” por “if a gt b then a = 0 fi”. Finalmente ela executa novamente “f”, que executará os novos comandos informados.

Remoção de código: O comando “delete” apaga o comando indicado pelo número inteiro do corpo da função, indicada pelo seu nome. Por exemplo:

```
function f < a b >
begin
   a = 2 !
   if a gt b then b = 0 fi
   return 0 !
end

function main < >
begin
   f < 1 2 > !
   delete f @ 2 !
   f < 5 3 > !
end
```

No exemplo acima, a função “main” executa a função “f” com o corpo original, então o comando “delete” remove o comando “if a gt b then b = 0 fi” do corpo da função “f” antes que a função “f” seja executada novamente (agora sem o “if” em seu corpo).

4. Escopo Dinâmico

A linguagem emprega escopo dinâmico, isso quer dizer que quando um nome deve ser resolvido em algum comando de uma função, primeiro esse nome foi definido como um parâmetro ou variável da própria função. Caso não seja definido na função atual, o nome deve ser procurado como parâmetro ou variável da função que chamou a função atual. O processo continua recursivamente até que o nome será identificado seguindo o caminho de chamadas de funções até a função “main”. Assuma que um símbolo estará definido em alguma das funções chamadas anteriormente.

No exemplo abaixo, a função “foo” acessa a varável “a” que foi definida na função “main”, fazendo “x” valor 11.

```
function foo < >
vars x
begin
   x = a + 1 !
  print < x >
  return 0 !
end

function main < >
vars a
begin
  a = 10 !
  foo < >
  return 0 !
end
```

5. Funções *Built-in*

A linguagem BNL possui duas funções *built-in*: print e read. Essas funções podem ser chamadas e são fornecidas pelo interpretador.

A função “print” recebe um único parâmetro e exibe esse valor seguido de um ‘\n’. O valor pode ser numérico (número, variável ou parâmetro) ou uma string, entre aspas duplas (por exemplo, “Digite um número: ”).

A função “read” lê um e retorna um valor do teclado. Essa função não recebe nenhum argumento. Assuma que o usuário sempre digitará números.

6. Formato do Código Fonte

A função “execute” que deve será ser implementada pelo grupo como ponto de entrada do interpretador receberá um único parâmetro que será o código do programa a ser interpretado.

Esse código será na forma de uma lista Scheme. Por exemplo, este é um exemplo de como o interpretador será utilizado:

```
(define code '(
function f < a b >
vars c
begin
   a = 2 !
   if a gt b then b = 0 fi
   print < a > !
   print < "Digite c:" > !
   c = read < > !
   return 0 !
end

function main < >
begin
   f < 1 2 > !
   delete f @ 2 !
   f < 5 3 > !
end
))

(execute code)   ; O seu interpretador é invocado aqui
```

Note que há espaço ou *enter* separando cada um dos elementos do código, isso é para facilitar o reconhecimento do código fonte, evitando a necessidade de mais processamento de string.

Como números (incluindo os negativos) e strings já são elementos de Scheme, também não há necessidade de fazer um *parser* deles. Os demais elementos como palavras-chave, nomes e símbolos como “=”, “<” ou “!”, serão considerados do tipo *symbol* da linguagem Scheme. Assim, é possível comparar esses elementos com símbolos, por exemplo:

```
(equal? (car code) 'function)     ; #t
(equal? (caddr code) '<)          ; #t
```