Universidade Federal do Rio de Janeiro
Departamento de Engenharia Eletrônica e de Computação

Relatório III

Interpretador de Lua

Alunos: Bernardo Antonio Boechat Florencio
João Pedro Soares Alves Fernandes
Professor: Miguel Elias Mitre Campista
Disciplina: Linguagens de Programação

Novembro
2019

Conteúdo

1 Introdução

Como previamente explicitado nos últimos relatórios redigidos ao longo do semestre, este projeto se baseia em um *tree walk interpreter* que é subdivido em: *lexer, parser* e *evaluator*.

Neste terceiro relatório, o enfoque será dado ao que não havia sido abordado no Relatório II: o *parser* e o *evaluator*. Não obstante, também será tratado neste relatório como o interpretador irá funcionar como um todo, sem antes lembrar que este interpretador é uma versão mais básica de um interpretador de Lua.

Inicialmente, devemos descrever o funcionamento do *parser*. Nessa etapa, o código-fonte já foi dividido em *tokens* pelo *lexer* e o próximo passo do *tree walk interpreter* é a construção de uma árvore AST [1], para que o *evaluator* possa executar os comandos solicitados pelo usuário.

Nesta árvore, temos como nós mais externos os *tokens*, sendo os demais nós conjunturas abstratas, como declarações e expressões. A figura abaixo, obtida na internet, ilustra um exemplo de implementação de uma AST para um trecho de código que corresponde a um *loop while*.

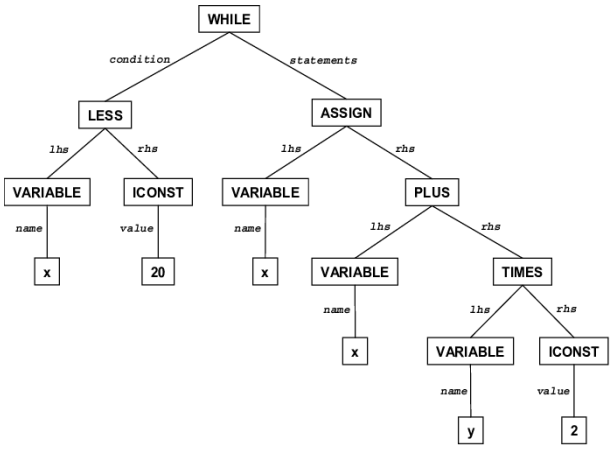

Figura 1: Exemplo de implementação de uma AST

Ao final da construção da árvore, essa é entregue ao último passo do interpretador, o *evaluator*. Então, ocorrerá a travessia dos nós de forma que o conteúdo previamente armazenado na AST seja executado de acordo com as especificações da linguagem.

Na seção 2, indicamos as modificações feitas no *lexer* em Perl, descrito no Relatório 3, e como se dá a integração deste com o programa em C++. Em seguida, a seção 3 descreve as funcionalidades implementadas. Os casos de uso são apresentados na seção 4. Por fim, a conclusão é feita na seção 5.

[1] do inglês *Abstract Syntax Tree*

2 Uso do analisador léxico

2.1 Modificações no módulo

Durante a programação do *lexer* em Perl, descrita no Relatório 2, as demais partes do interpretador ainda não tinham sido implementadas. Por isso, o módulo foi apresentado de forma que pudesse ser avaliado e testado por si só. Agora, algumas adaptações foram feitas para que seja possível integrá-lo ao programa.

Na última versão, o analisador léxico foi entregue somente realizando a impressão dos *tokens* na tela, de modo que este processo não irá se repetir ao longo do projeto, pois, a partir deste momento, os *tokens* serão retornados de forma codificada para o programa em C++ e ele os utilizará para as demais etapas da interpretação.

Além disto, outra modificação que deve ser mencionada é durante a situação de erro durante o escaneamento do código. Na versão entregue anteriormente, o *lexer* abortava a operação imediatamente. Na conjuntura atual que o programa se encontra, é inviável que o analisador léxico aborte, pois o programa em C++ está esperando alguma forma de retorno apropriado vindo do módulo.

Entretanto, ainda se faz necessário controlar a situação na qual ocorra a interpretação de *tokens* corrompidos. Em tal panorama, o que foi implementado é que o analisador léxico retornará um *token* especial de erro, que indicará ao intepretador em C++ que este deve ser abortado.

2.2 Integração com o Perl

A integração com o Perl foi feita usando funções e macros da API [2] descrita na referência (3). Foram utilizados como modelos os exemplos dos Slides de Aula de Linguagems de Programação.(1)

O programa em C++, no início de sua execução, chama a função *tokenize_input* do módulo analisador léxico. Como parâmetro, é fornecido o nome do arquivo entrado pelo usuário, contendo o código fonte. Então, o módulo escaneia o arquivo, conforme descrito no Relatório 2, e retorna a lista de *tokens*, codificada como um *array* de caracteres. Assim, os *tokens* são reconstruídos e a interpretação pode seguir. Por conveniência, caso nenhum arquivo seja especificado, o programa procura por um arquivo denominado "input.lua", no mesmo diretório do executável.

[2] do inglês *Application Programming Interface*

3 Implementação

O *parser* e o *evaluator* foram implementados em C++. Foram incluídos os cabeçalhos das bibliotecas-padrão *string*, *vector*, *memory*, *cstring*, *iostream*, *unordered_map*, *exception*, *cmath*, *sstream* e *iomanip*, além dos cabeçalhos necessários para a integração com o Perl: *perl.h* e *EXTERN.h*.

A seguir, são descritas as cinco funcionalidades do interpretador, propostas no Relatório 1 e agora implementadas: avaliação de expressões, armazenamento de variáveis, executação de estruturas condicionais (*if/else*), executação de estruturas de loop (*while*) e escrita de valores na tela. Essas funções serão exemplificadas mais adiante, na seção 4.

3.1 Expressões

O interpretador lida com quatro tipos distintos de valores: *number*, *string*, *boolean* e *nil*.

As operações aritméticas da Linguagem Lua foram corretamente implementadas, sendo essas: adição, subtração, multiplicação, divisão, divisão inteira, operador módulo, exponenciação e menos unário. O intepretador realiza coerção de tipo entre strings e números, quando necessário, seguindo as especificações da Linguagem. (2)

Também foram incluídos os operadores lógicos, os operadores relacionais, a concetenação de strings (em Lua, feita com o operador "..") e o operador de comprimento (utilizado com "#").

Não foram implementados os operadores bit-a-bit. Também não estão incluídas as expressões que lidam com tipos ou funcionalidades não implementadas, como por exemplo a expressão construtora de tabelas e a expressão de chamada de funções.

3.2 Variáveis

É permitida a declaração e atribuição de variáveis, tanto globais quanto locais (utilizando-se a palavra-chave *local*), que podem armazenar quaisquer tipos de valores dentre os implementados. Uma dada variável pode assumir tipos de valores diferentes durante a execução do programa, caracterizando assim a tipagem dinâmica.

Por causa das variáveis locais, foi necessária a implementação de blocos de código. É permitida a definição de um bloco explícito com as palavras-chave *do* e *end*, e as estruturas condicionais ou de *loops* também correspondem a blocos próprios. As regras de escopo e visibilidade seguem as especificações da linguagem.

3.3 Condicionais

As estruturas condicionais *if-elsif-else* foram corretamente implementadas. Uma dada estrutura pode ser arbitrariamente longa, com tantos blocos *elsif* quanto necessários. As

expressões nas condições são avaliadas conforme especificado no manual de referência de Lua: apenas os valores *nil* e *false* são avaliados como falsos, e todos os demais correspondem a condições verdadeiras.

3.4 *Loops*

O interpretador executa corretamente estruturas de *while*, que foram as únicas estruturas de *loop* implementadas. A condição é avaliada da mesma forma que nas estruturas *if-elsif-else*.

3.5 Escrita na tela

Em Lua, *print* é uma função da biblioteca padrão. Porém, nesse interpretador não foram incluídas funções. Como a escrita na tela é uma operação fundamental que seria perdida, o *print* foi implementado como se fosse uma palavra-chave, de modo que a escrita na tela possa ser feita mesmo sem a presença da funções.

O comportamento do *statement print* aqui implementado imita a função *print* da linguagem. É possível fornecer um número arbitrário de argumentos, que são escritos separados por caracteres de tabulação ("\t"). Após o último argumento, é impresso um caracter de quebra de linha ("\n").

4 Casos de Uso

Aqui, são apresentados alguns exemplos de códigos-fonte e os resultados de suas respectivas execuções.

As Figuras 2 e 3 apresentam, respectivamente, o código e o resultado do primeiro arquivo de caso de uso. Nesse exemplo, é interessante observar o funcionamento do escopo e sua consequência sobre a visibilidade das duas variáveis x, uma local e a outra global. Também nota-se a coerção de tipo na operação aritmética.

```
1 x = "hello"
2 do
3     local x = 9 / "2"
4     print (x)
5 end
6 print (x)
```

Figura 2: Código do arquivo *sample1.lua*

```
boechat@boechat-ubnt:~/projects/git/perlua$ ./interpreter sample/sample1.lua
4.5
hello
boechat@boechat-ubnt:~/projects/git/perlua$
```

Figura 3: Saída do arquivo *sample1.lua*

O segundo exemplo corresponde às Figuras 4 e 5. Nele, é apresentado o operador de tamanho de strings, mencionado na seção 3.1. Além disso, podem ser observadas a estrutura *if-elseif-else*, a operação lógica *and* e a impressão de múltiplos argumentos na tela com o print.

```
1 foo = "17_letters_string"
2 size = #foo
3 if size >= 20 then
4     print ("foo is large!", size)
5 elseif size >= 10 and size < 20 then
6     print ("foo is average!", size)
7 else
8     print ("foo is small!", size)
9 end
```

Figura 4: Código do arquivo *sample2.lua*

```
boechat@boechat-ubnt:~/projects/git/perlua$ ./interpreter sample/sample2.lua
foo is average! 17
boechat@boechat-ubnt:~/projects/git/perlua$
```

Figura 5: Saída do arquivo *sample2.lua*

A Figura 6 mostra um simples código de cálculo de termos da série de Fibonacci. O resultado da execução é visto na Figura 7. É interessante notar a execução correta da estrutura de *while*, assim como a atribuição múltipla de variáveis dentro do *loop*.

```
1 do
2     local it = 15 -- Define number of iterations here
3     local i = 1
4     local current, previous = 1, 0
5     while (i <= it) do
6         print (current)
7         current, previous, i =
8             current + previous, current, i + 1
9     end
10 end
```

Figura 6: Código do arquivo *sample3.lua*

```
boechat@boechat-ubnt:~/projects/git/perlua$ ./interpreter sample/sample3.lua
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
boechat@boechat-ubnt:~/projects/git/perlua$
```

Figura 7: Saída do arquivo *sample3.lua*

Por fim, são apresentados dois simples exemplos de mensagens de erro no código. O programador do código da Figura 8 esqueceu da palavra-chave *end* para encerrar a estrutura de controle *if*. Este é um erro de *parser*, ou seja, a interpretação é abortada antes do início da execução do programa. A mensagem de erro correspondente pode ser vista na Figura 9.

```
1 foo = true
2 if foo then
3     print ("Not gonna print!")
4
5 -- forgot "end" keyword
6 print ("Not gonna print either!")
```

Figura 8: Código do arquivo *parser_error.lua*

```
boechat@boechat-ubnt:~/projects/git/perlua$ ./interpreter sample/parser_error.lua
sample/parser_error.lua:7: Parser Error: Expected keyword "end" to close "if" block.
boechat@boechat-ubnt:~/projects/git/perlua$
```

Figura 9: Saída do arquivo *parser_error.lua*

No arquivo da Figura 10, há uma tentativa de soma entre uma variável com valor do tipo *number* e outra com o valor *nil*. Não há coerção entre esses dois tipos, e na realidade

o tipo *nil* não pode ser usado em nenhuma operação aritmética. Como a linguagem Lua é dinamicamente tipada, este corresponde a um erro em tempo de execução, cuja mensagem pode ser vista na Figura 11.

Nota-se que, em ambos os casos de erro, a mensagem inclui o caminho relativo para o arquivo-fonte e a linha no qual o erro foi detectado.

```
1 var = 4
2 foo = nil
3 print ("Will print!")
4 y = var + foo -- Oops!
5 print ("Will not print!")
```

Figura 10: Código do arquivo *runtime_error.lua*

```
boechat@boechat-ubnt:~/projects/git/perlua$ ./interpreter sample/runtime_error.lua
Will print!
sample/runtime_error.lua:4: Runtime Error: Attempt to perform incompatible operation on operand of type "nil".
boechat@boechat-ubnt:~/projects/git/perlua$
```

Figura 11: Saída do arquivo *runtime_error.lua*

5 Conclusão

O interpretador foi implementado com sucesso, de modo a incluir as funcionalidades descritas no Relatório 1. Para projetos futuros, seria interessante a implementação de mais estruturas e funcionalidades da linguagem, como por exemplo a declaração de funções e o uso de variáveis do tipo *table*.

6 Referências

Referências

[1] Campista, M. Slides de aula: Linguagens de programação.

[2] Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. Lua 5.3 reference manual, 2015.

[3] MacEachern, D., and Orwant, J. Perl 5 version 30.0 documentation: perlembed.

[4] Nystrom, B. *Crafting Interpreters*. 2015.