Construção de Compiladores 1 - 2018.1 - Profs. Mário César San Felice
(e Helena Caseli, Murilo Naldi, Daniel Lucrédio)
Trabalho 2

Olá, alun@.

Neste segundo trabalho da disciplina de Construção de Compiladores 1, você irá implementar um analisador semântico para a linguagem Lua, a mesma utilizada no trabalho 1. Assim como no primeiro trabalho, você deverá utilizar o ANTLR.

O trabalho deve ser desenvolvido em GRUPOS de até 4 pessoas.

A seguir são descritas mais informações de como você deverá fazer seu trabalho. Leia-as atentamente e busque construir uma implementação precisa, pois a avaliação será feita através de scripts automáticos.

Bom trabalho !

1. Preparando o ambiente e o projeto a ser utilizado

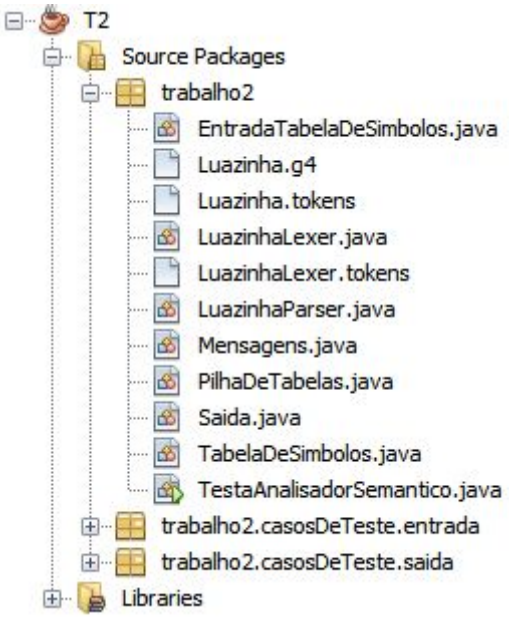
O ambiente é o mesmo do trabalho 1, ou seja, ANTLR + NetBeans.

Será fornecido como ponto de partida, em um arquivo .zip disponível na página da disciplina, já contendo:

- 1 projeto do NetBeans já com todo o código Java necessário e a gramática correta, no formato do ANTLR

Basta abrir o projeto no NetBeans, abrir a gramática no ANTLRWorks, e realizar a geração de código da mesma forma que no trabalho 1.

Os seguintes arquivos/classes já estão disponíveis no projeto:



- EntradaTabelaDeSimbolos.java: classe que representa uma entrada na tabela de símbolos. Cada entrada possui um nome e um tipo, ambos do tipo String.
- Luazinha.g4: arquivo com a gramática
- Luazinha.tokens, LuazinhaLexer.java, LuazinhaLexer.tokens e LuazinhaParser.java: código dos analisadores, a ser gerado pelo ANTLR
- Mensagens.java: classe auxiliar, para facilitar a escrita de mensagens
- PilhaDeTabelas.java: uma pilha de tabelas, para implementar múltiplos escopos
- Saida.java: classe auxiliar, para facilitar a geração da saída do analisador
- TabelaDeSimbolos.java: classe que representa a tabela de símbolos
- TestaAnalisadorSemantico.java: classe principal

Além desses, existem vários casos de teste. São fornecidas a entrada e saída esperada.

Ao abrir o projeto, pode ser necessário corrigir a referência para a biblioteca do ANTLR, para apontar para a localização do arquivo .jar em seu ambiente.

A classe principal já está preparada para executar os casos de teste (um de cada vez ou todos de uma vez), e produzir na saída uma comparação entre a saída obtida e a saída esperada.

Para utilizar, apenas especifique qual caso de teste deseja testar, na variável CASO_A_SER_TESTADO da classe TestaAnalisadorSemantico.java. Especifique 0 (zero) para testar todos, e observe na saída o resultado.

Por exemplo, especificando-se o CASO_A_SER_TESTADO = 2, se a saída produzida pelo seu analisador for idêntica à saída esperada, será exibida a mensagem:

```
OK:casoDeTeste02.txt
```

Caso contrário, será exibida uma mensagem de erro, com a saída obtida pelo seu analisador logo abaixo:

```
ERRO:casoDeTeste02.txt
================== Saída obtida ==================
Escopo: global

==================================================
```

Compare o conteúdo exibido com a saída esperada para descobrir onde está errado. Seu objetivo é, ao especificar CASO_A_SER_TESTADO = 0, obter a saída:

```
OK:casoDeTeste01.txt
OK:casoDeTeste02.txt
OK:casoDeTeste03.txt
OK:casoDeTeste04.txt
OK:casoDeTeste05.txt
OK:casoDeTeste06.txt
OK:casoDeTeste07.txt
OK:casoDeTeste08.txt
OK:casoDeTeste09.txt
OK:casoDeTeste10.txt
OK:casoDeTeste11.txt
OK:casoDeTeste12.txt
OK:casoDeTeste13.txt
OK:casoDeTeste14.txt
OK:casoDeTeste15.txt
OK:casoDeTeste16.txt
OK:casoDeTeste17.txt
OK:casoDeTeste18.txt
OK:casoDeTeste19.txt
OK:casoDeTeste20.txt
OK:casoDeTeste21.txt
OK:casoDeTeste22.txt
```

Note, no entanto, que isso não significa que a nota do trabalho será 10! Ainda há outros critérios

para correção, descritos mais adiante.

2. Semântica da linguagem Lua

Neste trabalho, você irá implementar um analisador semântico, que irá verificar algumas regras da linguagem Lua:

- Aninhamento de escopos
- Amarração de variáveis a um determinado escopo (globais, locais e parâmetros)
- Uso de variáveis não amarradas
- Parâmetro “self” implícito (regra semântica opcional)

Claro, existem muitas outras regras semânticas da linguagem Lua, mas não serão consideradas.

A seguir são descritas essas regras. Consulte o manual da linguagem para mais informações.

2.1. Aninhamento de escopos

Em Lua, existe um escopo global, sem delimitação léxica nenhuma, isto é, um arquivo pode conter comandos diretamente, como no exemplo abaixo:

```
-- uma expressão simples, com variáveis globais
x = 10
y = 10
z,h,a = x + y, y*10, 22
print(z)
```

Além disso, é possível declarar funções, como no exemplo abaixo:

```
-- funcao que calcula o fatorial
function fact(n)
   if n == 0 then
      return 1
   else
      return n * fact(n - 1)
   end
end;

print(fact(5))
```

As funções podem ser aninhadas, como no exemplo abaixo:

```
-- funcoes aninhadas, sem variáveis nem parâmetros
function a()
    function b()
    end
    function c()
        function d()
```

```
        end
        function e()
        end
    end
    function f()
    end
end;

function g()
    function h()
        function i()
        end
    end
end
```

Cada nova função cria um novo escopo, aninhado ao escopo que o contém. No exemplo acima, existe o escopo global, dentro do qual existe o escopo de “a”, dentro do qual existem os escopos de “b” e “c”, e assim por diante.

Além disso, é também possível criar um novo escopo através do comando “for”, como no exemplo abaixo:

```
-- funcoes aninhadas e for, com variáveis sem sobreposição
x1 = 0
function a()
    x2 = 0
    x3 = 0
    for i=0,10 do
        i = i+2
    end
    function b()
        x4 = 0
    end
...
```

Nesse exemplo, o escopo global contém o escopo de “a”. Dentro do escopo de “a”, existe o escopo de “b”, mas também existe um novo escopo, criado a partir da declaração “for” que faz a iteração de i de 0 a 10.

Existem outras formas de criação de escopo, mas não serão tratadas neste trabalho.

2.2. Amarração de variáveis a um determinado escopo (globais, locais e parâmetros)

O aninhamento de escopos descrito acima serve basicamente para fazer a amarração de variáveis. Em Lua, não existe declaração de variáveis. Uma variável é amarrada a um escopo quando ela é alvo de um comando de atribuição (para os escopos globais e funções), quando ela é declarada como parâmetro de função (para funções somente), ou quando é definida em um laço “for” (outras formas de amarração não serão consideradas).

Considere o exemplo abaixo

```
x1 = 0
function a()
    x2 = 0
    x3 = 0
    x1 = 10
    for i=0,10 do
        i = i+2
    end
    function b(p3, x2)
        x4 = 0
        local x3 = 10
        x4 = x5
        x6,x7 = 100,200
    end
...
```

Nesse exemplo:
- A variável x1 é alvo de atribuição no escopo global, portanto está amarrada ao escopo global.

- A variável x2 é alvo de atribuição no escopo de “a”, portanto está amarrada ao escopo de “a”.

- A variável x3 é alvo de atribuição no escopo de “a”, portanto está amarrada ao escopo de “a”.

- A variável x1 é alvo de atribuição dentro do escopo de “a”, porém, essa nova atribuição (x1 = 10) NÃO CRIA uma nova variável, pois x1 já existe em um escopo visível (o escopo global nesse caso). Trata-se da mesma variável.

- A variável i é definida no comando “for”, portanto está amarrada ao escopo desse “for”.

- A variável p3 é parâmetro de “b”, e portanto está amarrada ao escopo de “b”.

- A função “b” também define um parâmetro chamado x2. Já existe uma variável x2 amarrada a um escopo mais amplo e visível (escopo de “a”). Porém, como se trata de um parâmetro, É CRIADA UMA NOVA VARIÁVEL, que fica amarrada ao escopo de “b”. Nesse escopo, portanto, existem DUAS variáveis x2, mas a segunda está se sobrepondo à primeira.

- A variável x4 é alvo de atribuição no escopo de “b”, e portanto está amarrada ao escopo de “b”.

- A variável x3 é alvo de atribuição dentro do escopo de “b”. Já existe uma variável x3 amarrada a um escopo mais amplo e visível (escopo de “a”). Porém, a chamada local x3 = 10 CRIA UMA NOVA VARIÁVEL, devido ao prefixo “local”, que força a amarração de uma nova variável mesmo que já exista uma disponível em um escopo mais amplo.

- A atribuição x4 = x5 NÃO CRIA UMA VARIÁVEL x5, pois somente são criadas (e amarradas) as variáveis do lado esquerdo da atribuição. Nesse exemplo, espera-se que a variável x5 já exista e

esteja amarrada apropriadamente a um escopo visível.

- As variáveis x6 e x7 são alvos de atribuição em uma única instrução, e portanto são amarradas ao mesmo tempo ao escopo onde aparecem (“b”).

Em resumo:
- Um comando de atribuição cria uma nova variável, amarrada ao escopo onde a atribuição acontece.
- Somente são criadas (e amarradas) as variáveis que aparecem no lado esquerdo da atribuição.
- Se a variável envolvida na atribuição já existir em um escopo visível, não é criada uma nova variável. Trata-se da mesma variável.
- Se a variável envolvida na atribuição já existir em um escopo visível, mas existir o prefixo “local”, será criada uma nova variável, no escopo atual. (Não será considerado o caso da variável ser declarada duas vezes com “local” em um mesmo escopo)
- Uma declaração de função com parâmetros cria novas variáveis para os parâmetros, amarradas ao escopo da função.
- Um comando “for” cria novas variáveis que porventura apareçam em sua estrutura, amarradas ao escopo do “for”. (Neste trabalho, mesmo que uma variável definida em um “for” já exista em um escopo visível, será criada uma nova variável).

2.3. Uso de variáveis não amarradas

Variáveis que aparecem em expressões, lado direito de atribuição, dentro de um print, dentro de um if, dentro de um while, etc, devem estar amarradas a um escopo visível. Caso contrário, trata-se de um erro semântico. No exemplo abaixo:

```
-- funcoes simples, apenas com variaveis, e globais
-- usando variáveis não amarradas
x1 = x

function a()
    x2 = x11 + 2
    if x3 > 4 then x3 = 5 end
end;

x1 = x2
```

Existem os seguintes erros semânticos:

- A variável x é usada no lado direito na atribuição x1 = x, no entanto não está amarrada ao escopo global.

- A variável x11 é usada no lado direito da atribuição x2 = x11 + 2, no entanto não está amarrada a nenhum escopo visível (nem “a” e nem o global).

- A variável x3 é usada dentro de um comando “if”, no entanto não está amarrada a nenhum escopo visível (nem “a” e nem o global)

- A variável x2 é usada no lado direito da atribuição x1 = x2, no entanto está amarrada a um escopo não visível (“a”), e não está amarrada no escopo global.

2.4. Parâmetro “self” implícito (regra semântica opcional)

Como já visto, uma função gera um novo escopo, e seus parâmetros são variáveis amarradas a ele. No exemplo:

```
function math.soma(a,b)
    return a + b
end;
```

É criado um escopo “math.soma”, e duas variáveis a e b são amarradas a ele.

Em Lua, funções podem ser definidas como métodos, através do caractere “:”. Segundo a documentação da linguagem Lua, uma definição:

```
function x.y.f:g(a,b)
end
```

é equivalente a:

```
function x.y.f.g(self,a,b)
end
```

Ou seja, existe um parâmetro implícito, chamado “self”, que é utilizado para que a função seja acessada de uma forma “orientada a objetos” (ou seja, um método de uma classe).

Portanto, no exemplo abaixo:

```
function math:multiplica(e,f)
    return e * f
end;
```

É criado um escopo “math.multiplica”, e três variáveis (self, e, f) são amarradas a ele.

Obs: essa regra é opcional, mas poderá ajudar na nota, como descrito mais adiante.

3. Descrição do trabalho

Você precisará fazer um analisador semântico que faça a verificação das regras semânticas descritas na seção 2. Para isso, pode inserir regras semânticas na própria gramática (arquivo Luazinha.g), ou criar um Visitante/Listener, conforme visto em aula.

Não é necessário alterar as regras sintáticas, basta adicionar regras semânticas. No entanto, se achar que seu trabalho será mais simples se as regras sintáticas forem alteradas, fique à vontade para mexer, desde que TODAS AS MUDANÇAS estejam devidamente documentadas em forma de comentários.

Algumas regras semânticas já estão implementadas, para retornar valores úteis, como nomes compostos, e linha/coluna de variáveis (para mensagem de erro, conforme abaixo).

As regras poderão utilizar as classes já prontas no projeto do NetBeans:

- O objeto “pilhaDeTabelas”, da classe PilhaDeTabelas, está disponível para ser utilizado. Essa classe disponibiliza métodos como empilhar(), desempilhar() e topo(), para a regra dos escopos aninhados.
- Cada novo escopo (global, função e for) deve gerar o empilhamento e desempilhamento da tabela. Para isso, insira as ações necessárias nos locais apropriados da gramática, como visto em aula (já está feito para o escopo global).
- A cada término de escopo deve haver o desempilhamento da respectiva tabela de símbolos. Nesse momento, o conteúdo daquele escopo será impresso. Ou seja, a saída dos casos de teste irá mostrar os escopos não na ordem em que aparecem, mas na ordem em que são desempilhados. Observe as saídas esperadas dos casos de teste, em especial os caso de teste 02, para compreender a lógica de impressão.
- A classe TabelaDeSimbolos deve ser utilizada para amarrar as variáveis e parâmetros. Cada escopo terá sua própria tabela (empilhada apropriadamente), cujo nome deve ser definido na sua criação. O nome do escopo global é “global”, o nome do escopo de uma função é o nome da função (inclusive nomes compostos), e o nome de um escopo “for” é simplesmente “for”.
- Para cada tabela, estão disponíveis métodos para adicionar símbolos e verificar se existe um símbolo. Para adicionar um símbolo, deve ser especificado um de dois tipos possíveis: “parametro” ou “variavel” (assim mesmo, sem acentos).
- Veja que existe um médodo TabelaDeSimbolos.existeSimbolo(String), e um método PilhaDeTabelas.existeSimbolo(String). O primeiro verifica somente em uma tabela. O segundo verifica em toda a pilha. Analise bem onde usar cada um.
- Veja que existem dois métodos para inserir símbolos na tabela de símbolos. Um deles adiciona um único símbolo, de um tipo. O outro adiciona uma lista de símbolos do mesmo tipo de uma única vez, e é útil para declarações múltiplas de variáveis. Analise bem onde usar cada um.
- Para exibir os erros semânticos de variável não amarrada utilize a chamada: `Mensagens.erroVariavelNaoExiste(int,int,String)`. (obs: veja que a regra “var” já está implementada para retornar linha e coluna)
- Tome sempre como base os casos de teste, pois eles servem de exemplo do que deve ser feito, e também servirão de critério de avaliação.

4. Saída do analisador semântico

Se fizer as regras corretamente, a saída do analisador semântico será automaticamente gerada com as chamadas aos métodos “desempilhar” e “Mensagens.erroVariavelNaoExiste”. Não é necessário inserir nenhum outro código de impressão.

5. Entrega do trabalho e avaliação

Você deverá entregar todo o projeto do Netbeans, compactado em um arquivo .zip ou .rar. No início do arquivo da gramática, coloque o RA dos membros do grupo no local indicado (dentro da variável string "grupo" na área @members). A entrega será feita exclusivamente via Moodle, até a data indicada no ambiente.

A nota do seu trabalho será composta de 4 parcelas:

- 70% da nota será calculada automaticamente com base nos casos de teste de 1 a 15. Estes testam as regras semânticas 2.1 e 2.2.
- 30% da nota será calculada automaticamente com base nos casos de teste de 16 a 21. Estes testam a regra semântica 2.3.
- 10% da nota será calculada automaticamente com base no caso de teste 22. Este testa a regra semântica 2.4.

Portanto, será possível tirar até uma nota 11 no trabalho, se a regra opcional for implementada corretamente.

Há ainda um critério de documentação. Serão considerados aspectos como indentação, nome das regras, e o principal - COMENTÁRIOS EXPLICATIVOS em todas as regras semânticas adicionadas. A ausência destes irá resultar em decréscimo da nota, em até 30%. Portanto, capriche na gramática e no código.