

Relatório - Analisador Semântico

25 de Junho de 2023

Nome Completo	Matrícula
Lívia Pereira Ozório	201835011
Patrick Canto de Carvalho	201935026

Sumário

1	Introdução	3
2	Analizador Semântico	3
2.1	Informações gerais	3
2.2	Estruturas de dados	3
2.3	Tipos	3
2.4	Visitor	4
3	Tratando Sobrecarga de Funções	8
4	Como compilar	8

1 Introdução

Este relatório tem como objetivo explicitar os pontos mais relevantes para a compreensão do processo de desenvolvimento de um analisador semântico para a linguagem *lang*, incluindo as decisões de projeto como as estratégias adotadas e estruturas de dados auxiliares.

2 Analisador Semântico

2.1 Informações gerais

Foi escolhido para esta parte da implementação do trabalho o padrão *visitor*. Para tal, foi utilizada uma classe abstrata *Visitor*, estendida na classe *ScopeVisitor*.

2.2 Estruturas de dados

Algumas estruturas auxiliares foram criadas para realizar a análise semântica:

- *HashMapScope*: é um *HashMap* que relaciona string e inteiro. Associa cada função ao local onde está seu escopo na lista de escopos, isto facilita a busca nas listas quando é necessário encontrar os parâmetros, os retornos ou as variáveis de uma determinada função.
- *Variables*: é uma lista de listas de *HashMap* que relaciona string e inteiro. Nela é armazenado, em cada posição, um escopo que possui um ou mais níveis. Esses níveis armazenam os nomes das variáveis e quais os tipos das variáveis de um bloco da função.
- *Params*: é uma lista de lista de string. Representa os parâmetros das funções. Cada posição da lista corresponde a uma posição na lista de escopos.
- *Returns*: é uma lista de lista de string. Armazena os tipos retornados por cada linha do código. Assim como o anterior, cada posição da lista corresponde a uma posição na lista de escopos.
- *HashMapData*: é um *HashMap* que relaciona string e um outro *HashMap* que, por sua vez, relaciona duas strings. É utilizado para associar o nome dos tipos de dados criados pelo usuário aos tipos de seus atributos.

Além dessas estruturas, existem duas variáveis inteiras que auxiliam no processo de consulta do escopo. A primeira se chama *scopeFunc*, que indica em qual função está o processo de análise em um dado momento. Já a segunda é o *level*, essa variável indica em qual nível do escopo a análise está. Isto porque cada bloco possui um escopo próprio contendo as variáveis dos escopos de menor nível e as suas variáveis locais do bloco.

2.3 Tipos

Durante a execução do analisador semântico é adicionado alguns tipos na lista de tipos, podendo estes serem:

- *Int* - que representa números do conjunto dos inteiros;
- *Float* - que representa números do conjunto dos reais;
- *Bool* - que representa valores booleanos, ou seja, verdadeiro ou falso;

- Char - que representa caracteres ;
- Error - que representa que o erro aconteceu anteriormente;
- CMD - que representa que o procedimento não possui um tipo mas está tudo certo;
- Um D, com D existente no hashmap de dados - que representa um objeto do tipo dado D;
- T[], com T pertencente a um dos outros tipos - que representa um vetor do tipo T.

2.4 Visitor

- visitor(Prog p) - O visitor do Prog chama o visitor do DataList e do DataFunc. Consome os dois, caso tenha erro em algum adiciona Error na lista de tipos, caso não adiciona CMD;
- visitor(FuncList f) - O visitor do FuncList cria os escopos adicionando a função no hashmap de escopo e criando o valor de escopo de cada função, preenche a lista de parâmetros e retornos de cada uma também. Depois confere se o programa possui uma função main() caso não ele gerar uma mensagem de erro falando que o programa não possui main(). Além disto chama e processa o visitor de cada Func contida no FuncList adicionando Error na lista de tipos caso alguma função possua erro. Se nenhuma função tem erro mas o programa não possui a main() também é adicionado Error na lista de tipos, apenas é adicionado CMD caso nenhuma função tenha erro e a main() exista;
- visitor(DataList d) - O visitor do DataList chama e processa o visitor do Data para cada dado contido na lista. Caso algum Data retorne com Error é adicionado Error na lista de tipos, caso estejam todos corretos adiciona CMD;
- visitor(Data d) - O visitor do Data confere se o ID do Data que se quer inserir ainda não está no hashmap de dados, se ainda não estiver preenche o hashmap de dados e adiciona CMD na lista de tipos, mas caso ele já esteja então é colocado uma mensagem de erro na tela informando que o dado já foi inserido e adiciona Error na lista de tipos;
- visitor(Func f) - O visitor da func atualiza qual escopo estamos analisando, e passa nível do escopo para zero já que quando estamos iniciando a análise da função o nível analisado sempre será o zero e depois chama o visitor do CmdList;
- visitor(CmdList c) - O visitor do CmdList cria um novo nível de escopo na função, isto porque o CmdList indica um novo bloco, e então atualiza a variável que indica o nível. Depois chama e processa o visitor de cada comando contido na lista. Caso tenha algum comando com erro adiciona Error a lista de tipos caso contrário adiciona CMD Por ultimo regride novamente a variável do nível para o nível anterior já que sairemos do bloco;
- visitor(Print p) - O visitor do Print chama e processa visitor da expressão a ser impressa verificando se a expressão é do tipo Error, caso seja a função adiciona um Error na lista de tipos caso contrário adiciona CMD;
- visitor(Type t) - O visitor do Type confere qual o tipo adicionar na lista de tipos. Caso o tipo enviado não pertença aos tipos fundamentais da linguagem nem aos dados criados a função gera um erro na tela falando que o tipo não existe e adiciona Error à lista de tipos, caso contrario apenas adiciona o nome do Type t;

- visitor(Add a) - O visitor do Add chama e processa os tipos das duas expressões. E então conferem se pelo menos uma das duas expressões tem tipo Error caso sim a função adiciona o tipo Error na lista de tipos. Caso não, confere se as expressões são FloatAst, FloatAst ou Int, Int, caso sim adiciona o tipo da operação (FloatAst ou Int) na lista de tipos. Caso não, adiciona um tipo Error na lista de tipos e lança um erro na tela falando que os tipos das expressões dadas não podem ser somados;
- visitor(Mul a) - O visitor do Mul chama e processa os tipos das duas expressões. E então conferem se pelo menos uma das duas expressões tem tipo Error caso sim a função adiciona o tipo Error na lista de tipos. Caso não, confere se as expressões são FloatAst, FloatAst ou Int, Int, caso sim adiciona o tipo da operação (FloatAst ou Int) na lista de tipos. Caso não, adiciona um tipo Error na lista de tipos e lança um erro na tela falando que os tipos das expressões dadas não podem ser multiplicados;
- visito(Rest a) - O visitor do Rest ele chama e processa os tipos das duas expressões. E então conferem se pelo menos uma das duas expressões tem tipo Error caso sim a função adiciona o tipo Error na lista de tipos. Caso não, confere se as expressões são Int, Int, caso sim adiciona o tipo da operação (Int) na lista de tipos. Caso não, adiciona um tipo Error na lista de tipos e lança um erro na tela falando que os tipos das expressões dadas não podem ter resto;
- visitor(Div a) - O visitor do Div chama e processa os tipos das duas expressões. E então conferem se pelo menos uma das duas expressões tem tipo Error caso sim a função adiciona o tipo Error na lista de tipos. Caso não, confere se as expressões são FloatAst, FloatAst ou Int, Int, caso sim adiciona o tipo da operação (FloatAst ou Int) na lista de tipos. Caso não, adiciona um tipo Error na lista de tipos e lança um erro na tela falando que os tipos das expressões dadas não podem ser divididos;
- visitor(Sub a) - O visitor do Sub chama e processa os tipos das duas expressões. E então conferem se pelo menos uma das duas expressões tem tipo Error caso sim a função adiciona o tipo Error na lista de tipos. Caso não, confere se as expressões são FloatAst, FloatAst ou Int, Int, caso sim adiciona o tipo da operação (FloatAst ou Int) na lista de tipos. Caso não, adiciona um tipo Error na lista de tipos e lança um erro na tela falando que os tipos das expressões dadas não podem ser subtraídos;
- visitor(SubUni a) - O visitor do SubUni chama e processa o tipo da expressão. E então conferem se a expressão tem tipo Error caso sim a função adiciona o tipo Error na lista de tipos. Caso não, confere se a expressão é FloatAst ou Int, caso sim adiciona o tipo da operação (FloatAst ou Int) na lista de tipos. Caso não, adiciona um tipo erro na lista de tipos e lança um Error na tela falando que o tipo da expressão dada não podem ser operado;
- visitor(Neg a) - O visitor do Neg chama e processa o tipo da expressão. E então conferem se a expressão tem tipo Error caso sim a função adiciona o tipo Error na lista de tipos. Caso não, confere se a expressão é Bool, caso sim adiciona o tipo da operação (Bool) na lista de tipos. Caso não, adiciona um tipo Error na lista de tipos e lança um erro na tela falando que o tipo da expressão dada não podem ser operado;
- visitor(And a) - O visitor do And chama e processa os tipos das duas expressões. E então conferem se pelo menos uma das duas expressões tem tipo Error caso sim a função adiciona o tipo Error na lista de tipos. Caso não, confere se as expressões são Bool, Bool caso sim adiciona o tipo da operação (Bool) na lista de tipos. Caso não, adiciona um tipo

erro na lista de tipos e lança um Error na tela falando que os tipos das expressões dadas não podem ser operados;

- visitor(GreaterThan a) - O visitor do GreaterThan chama e processa os tipos das duas expressões. E então conferem se pelo menos uma das duas expressões tem tipo Error caso sim a função adiciona o tipo Error na lista de tipos. Caso não, confere se as expressões são FloatAst, FloatAst ou Int, Int, caso sim adiciona o tipo da operação (Bool) na lista de tipos. Caso não, adiciona um tipo erro na lista de tipos e lança um Error na tela falando que os tipos das expressões dadas não podem ser operados;
- visitor(LessThan a) - O visitor do LessThan chama e processa os tipos das duas expressões. E então conferem se pelo menos uma das duas expressões tem tipo Error caso sim a função adiciona o tipo Error na lista de tipos. Caso não, confere se as expressões são FloatAst, FloatAst ou Int, Int, caso sim adiciona o tipo da operação (Bool) na lista de tipos. Caso não, adiciona um tipo Error na lista de tipos e lança um erro na tela falando que os tipos das expressões dadas não podem ser operados;
- visitor(Diff a) - O visitor do Diff chama e processa os tipos das duas expressões. E então conferem se pelo menos uma das duas expressões tem tipo Error caso sim a função adiciona o tipo Error na lista de tipos. Caso não, confere se as expressões possuem tipos iguais, caso sim adiciona o tipo da operação (Bool) na lista de tipos. Caso não, adiciona um tipo Error na lista de tipos e lança um erro na tela falando que os tipos das expressões não gera uma comparação válida;
- visitor(Eq a) - O visitor do Eq chama e processa os tipos das duas expressões. E então conferem se pelo menos uma das duas expressões tem tipo Error caso sim a função adiciona o tipo Error na lista de tipos. Caso não, confere se as expressões possuem tipos iguais, caso sim adiciona o tipo da operação (Bool) na lista de tipos. Caso não, adiciona um tipo Error na lista de tipos e lança um erro na tela falando que os tipos das expressões não gera uma comparação válida;
- visitor(Int i) - O visitor do Int adiciona o valor Int na lista de tipos;
- visitor(FloatAst i) - O visitor do FloatAst adiciona o valor FloatAst na lista de tipos;
- visitor(Bool i) - O visitor do Bool adiciona o valor Bool na lista de tipos;
- visitor(iterate i) - O visitor do Iterate chama e processa o visitor da expressão e confere se a expressão é do tipo Int ou Error, caso não seja a função lança um erro na tela para o usuário falando que a expressão deve ser um Int. Então chama-se e processa o visitor do then. Com isto é avaliado o que colocar na lista de tipos, se a expressão for diferente de Int adiciona-se Error, caso seja igual a Int mas tenham algum erro no then então adiciona Error também. Apenas se não tiver nenhum erro no then e a expressão for Int que a função adiciona CMD;
- visitor(If i) - O visitor do If chama e processa o visitor da expressão e confere se a expressão é do tipo Bool ou Error, caso não seja a função lança um erro na tela para o usuário e falando que a expressão deve ser um Bool. Então chama-se e processa o visitor do then. Se o else existir chama e processa o visitor do else. Com isto é avaliado o que colocar na lista de tipos, se a expressão for diferente de Bool adiciona-se Error, caso seja igual a Bool mas tenham algum erro no then ou no else então adiciona Error também. Apenas se não tiver nenhum erro no then, no else e a expressão for Bool que a função adiciona CMD;

- visitor(Attr a) - O visitor da atribuição inicialmente chama e processa o visitor da expressão. Então é visto se a variável a qual se quer atribuir valor é uma variável simples ou se é um atributo de dado ou se é um vetor. Caso seja uma variável simples confere se a variável já existe caso sim o valor atribuído a ela deve ter o mesmo tipo do valor anterior e caso descumpra essa exigência é gerado um erro na tela e adicionado Error na lista de tipos. Caso a variável não exista adiciona ela na lista de variáveis no seu devido escopo e nível e depois adiciona CMD na lista de tipos. Já no caso de ser um vetor ou um atributo de dado, é chamado e processado o visitor do LValue. Com isto se o tipo da variável e o tipo da expressão são diferentes e nenhum dos dois é do tipo Error é gerado uma mensagem na tela falando que não se pode atribuir e adicionado Error na lista de tipos. Caso a variável ou a expressão forem do tipo Error adiciona Error a lista de tipos. Caso esteja tudo certo é adicionado CMD;
- visitor(LValue l) - O visitor do LValue começa avaliando se o nome dado é um nome simples ou um nome de um acesso de atributo de dado ou nome de um acesso a uma posição de vetor. Caso seja um nome simples apenas procura o nome da variável em seu escopo e pega o tipo, se a variável existir no escopo adiciona o tipo na lista de tipos, caso não exista retorna uma mensagem na tela falando que a variável não foi inicializada, ou seja, a variável não existe. Já se for um acesso a uma posição do vetor é chamado e processado o visitor da expressão, depois é visto se a expressão é do tipo Int, se for é chamado e processado o visitor do LValue para pegar o tipo do vetor, pegando este tipo é visualizado se o tipo é um Error se for só é adicionado Error na lista de tipos, caso contrário é adicionado o tipo do vetor. No caso da expressão ser do tipo Error é adicionado Error na lista de tipos, e no caso de não ser nem Int nem Error então é gerado uma mensagem na tela dizendo que a expressão esperava tipo Int. E o último caso é o de ser um acesso a um atributo de um dado, neste caso é chamado e processado o visitor do LValue que vai me trazer o tipo do dado a qual o atributo pertence. Se o tipo for um Error é adicionado um Error na lista de tipos, caso não seja procura o tipo do atributo no hashmap de dados e então encontrando adiciona o tipo do atributo na lista de tipos, caso não encontre a função escreve na tela que o atributo procurado não existe naquele dado e adiciona Error na lista de tipos;
- visitor(ID i) - O visitor do ID adiciona na lista de tipos o tipo de uma determinada variável consultando o lista de variáveis do escopo;
- visitor(New n) - O visitor do New primeiro confere se estamos querendo criar um vetor ou apenas um tipo dado mesmo. Se for um vetor é chamado e processado o visitor da expressão, então é conferido se a expressão é do tipo Int ou Error. Se não for nenhum dos dois gera uma mensagem na tela dizendo que a expressão esperava um Int e adiciona Error na lista de tipos, caso seja um Error apenas adiciona Error na lista de tipos e se for Int é chamado e processado o visitor do Type que confere qual o tipo que desejamos criar, então adiciona este tipo na lista de tipos. Já no caso de não ser um vetor só chama e processa o visitor do Type e adiciona o valor do tipo na lista de tipos;
- visitor(Read r) - O visitor do Read confere se a variável a qual se quer armazenar um valor já existe. Se sim confere se a variável é do tipo Char ou Error. Se não for de nenhum dos dois tipos é enviado uma mensagem de erro avisando se esperava uma variável do tipo Char e então adiciona Error na lista de tipos. Se a variável for do tipo Error somente adiciona Error na lista de tipos. Se for do tipo Char adiciona CMD. Agora se a variável ainda não existe no escopo apenas adiciona a variável na lista de variáveis em seu devido escopo e nível e adiciona CMD na lista de tipos;

- `visitor(CallFunction c)` - O visitor do `CallFunction` primeiro avalia se a função existe no hashmap de funções, com isto já é possível saber se os parâmetros foram passados corretamente, já que a chave do hashmap é a combinação do ID dele e dos tipos dos parâmetro. Se a função não existe é retornado um erro na tela informando que a função não foi definida e adiciona `Error` na lista de tipos. Caso exista a função, confere se o número de retornos esperado é igual ao recebido, se for confere para cada retorno se a variável a qual será salvo já existe se já confere se os tipos batem, caso batam adiciona `CMD` na lista de tipos, no caso de não bater é colocado o tipo `Error` e enviado uma mensagem avisando. Se a variável ainda não existe no escopo adiciona na lista de variáveis no escopo e nível correto e coloca `CMD` na lista de tipos também. Já caso o número de retornos seja diferente do esperado então gera uma mensagem avisando e coloca-se `Error` na lista de tipos;
- `visitor(CallFunctionVet c)` - O visitor do `CallFunction` primeiro avalia se a função existe no hashmap de funções, com isto já é possível saber se os parâmetros foram passados corretamente, já que a chave do hashmap é a combinação do ID dele e dos tipos dos parâmetro. Se a função não existe é retornado um erro na tela informando que a função não foi definida e adiciona `Error` na lista de tipos. Caso a função exista é chamado e processado o tipo da expressão e confere se a expressão é do tipo `Int` ou `Error` caso não seja, é adicionado uma mensagem na tela informando que se espera um `Int` e coloca `Error` na lista de tipos, se for um `Error` somente adiciona `Error` na lista de tipos, já se for um `Int` confere se o valor do retorno pedido faz parte dos retornos da função caso sim adiciona o tipo do retorno corresponde a posição passada pela expressão na lista de tipos, caso não adiciona `Error` na lista de tipos e gera uma mensagem informando;

3 Tratando Sobrecarga de Funções

Para o tratamento de sobrecarga, o hashmap que contem os escopos possui como chave o nome da função concatenada aos tipos dos parâmetros dela. Dessa forma, quando se busca no `CallFunction` ou no `CallFuncVet`, o visitor deles cria uma string que contem o nome da função chamada e os tipos dos parâmetros passados. Caso essa string seja igual a alguma chave do hashmap, a função existe e podemos conferir o resto do programa.

Para tratar este problema no interpretador, a solução foi passar o `ScopeVisitor` criado na análise semântica para a função `interpret`. Assim o `interpret` tem todos os escopos da análise semântica para consultar. Ao entrar no `interpret` de uma função, o escopo anterior e o nível anterior são salvos e então as duas variáveis são atualizadas. A variável `escopoFunc` do `scopeVisitor` é passado para o valor que representa a função a qual entramos e o nível para 0. Ao final do `interpret` da função os valores de escopo e nível anteriores são restaurados. Já ao entramos no `interpret` do `CmdList`, da mesma forma que o visitor dele atualiza a variável `level`, o `interpret` também o fará. E ao final vai decrescer em 1 o valor da variável `level` para voltar ao bloco anterior. Dessa forma ao entrarmos no `interpret` do `CallFunction` ou do `CallFuncVet` é possível avaliar os tipos dos parâmetros passados para ver qual função é chamada já que possuímos o escopo e o nível certo.

4 Como compilar

Para compilar e executar usando o programa `Make`, basta executar o seguinte comando no diretório do projeto:


```
make run <nome do arquivo lang>
```

Alternativamente, executar os seguintes comandos para compilar :

```
make compile
```

E o seguinte para executar:

```
java -cp .:antlr-4.8-complete.jar Interpretador <nome do arquivo lang>
```

Para realizar todos os testes presentes na pasta testes/semantica, executar:

```
./teste.sh
```

O programa deverá ter as suas saídas apresentadas no terminal. A chamada do compilador no arquivo teste.sh possui uma *flag* -v, que faz com que seja impresso o nome do arquivo na saída, antes do resultado da execução, para facilitar a verificação.