Universidade Federal de Juiz de Fora

Departamento de Ciência da Computação

Quarto Trabalho
Teoria dos Compiladores (DCC045)(2021/1)

Implementação da Análise Semântica para a Linguagem Lang

Lucas Diniz da Costa - 201465524C

Agosto 2021

Implementação da Análise Semântica para a Linguagem Lang

Lucas Diniz da Costa - 201465524C

1 Introdução

Este é um trabalho elaborado pelo aluno Lucas Diniz da Costa para a disciplina de Teoria dos Compiladores(2021/1) ministrada pelo professor Leonardo. O relatório a seguir visa salientar as observações relevantes sobre o quarto trabalho que consiste na implementação da Análise Semântica para a linguagem Lang.

2 Desenvolvimento

O projeto foi desenvolvido utilizando a linguagem de programação Java, conforme as especificações do enunciado, e foi executado em uma máquina com as seguintes configurações de Hardware e Software:

- *Sistema Operacional*: Windows 10;
- *Memória RAM*: 16 Gb CORSAIR DDR4 2400 Mhz ;
- *Processador*: AMD Ryzen 5 3400G;

Contudo, na mesma máquina também foi testado o projeto no sistema operacional Linux Mint como solicitado pelo enunciado.

2.1 Implementação do Analisador Semântico

Para esta etapa de criação do Interpretador foi utilizado a base do terceiro trabalho que seria o uso da ferramenta ANTLR na versão 4.9.2.

A estratégia utilizada para criar o Analisador Semântico para a linguagem de programação foi a utilização de **Visitors**.

O Visitor criado se chama 'TypeCheck'. A ideia principal é que ele será executado antes do interpretador, assim, notificará o usuário sobre os erros semânticos antes de tentar

executá-los no interpretador. O TypeCheck caminhará na Árvore de Sintaxe Abstrata(AST) criada no terceiro trabalho da disciplina, contudo, ao identificar um erro, ele tenta prosseguir até conseguir capturar todos os erros semânticos e apresentá-los ao usuário.

Para a análise semântica foram criadas classes para trabalhar somente com o armazenamento de tipos e outras informações relevantes como o detalhamento dos tipos dos campos de retorno, parâmetros e tipo das funções, os atributos dos tipos heterogêneos(tipos data), entre outras informações. o TypeCheck verificará se durante as operações, as chamadas de funções e atribuições de variáveis, há consistência de tipo dos elementos, ou seja, se os tipos estão de acordo com o padrão da linguagem Lang, e deste modo, garantindo que quando o interpretador for executado, as operações farão sentido e terão coerência.

Vale ressaltar que mesmo que interpretador não for mais necessário, o TypeCheck ainda é necessário pois garantirá a consistência do código e adequação para possíveis fases posteriores de um compilador, como a geração de código em alto ou baixo nível, por exemplo.

O modelo de implementação utilizando visitor para a análise semântica desenvolvido neste trabalho foi inspirado no código disponibilizado pelos professores Elton e Leonardo para a disciplina de Compiladores. É composto principalmente por uma lista para armazenar os erros obtidos, um ambiente para armazenar as funções, um ambiente para executar os comandos das funções, uma pilha que armazena os tipos dos elementos, uma tabela hash que armazena os atributos dos dados heterogêneos para serem consultados e se instanciar um tipo em uma variável, um objeto que armazena o ultimo valor numérico apresentado com o intuito de verificar qual tipo será retornado na função de acordo com este índice e a lista das funções presentes no código.

2.1.1 Sobrecarga de Funções

Sobrecarga de funções é quando mais de uma função da linguagem apresenta o mesmo nome(podendo se estender até os mesmos tipos de parâmetros e tipos de retorno).

Conforme solicitado no enunciado do trabalho como requisito para esta etapa, foi implementado uma verificação e tratamento de sobrecarga de funções tanto para o interpretador quanto para o analisador semântico.

A sobrecarga foi tratada de maneira parecida em ambas as ferramentas (Interpretador, Analisador Semântico), seu funcionamento se da ao caminhar pela AST, principalmente no primeiro nó da árvore(Program), pois neste nó que serão armazenadas em uma lista as funções do código. Durante este armazenamento é verificado se o nome da função em análise já foi adicionado na lista de funções, caso tenha, significa que pode ser um caso de sobrecarga, logo deverá ter os tipos dos parâmetros verificados com todas as funções que tiverem o mesmo nome, caso não tenha, significa que a função pode ser adicionada sem problemas,

caso contrário, significa que houve sobrecarga de funções, será visto como um erro e não será adicionada na listagem de funções, e posteriormente, o problema será informado ao usuário.

Já no contexto de utilização de funções com nomes iguais, no momento da chamada da função será visto qual função que coincide com os tipos dos parâmetros passados na chamada, e feita a verificação, a função correspondente será executada.

2.2 Scripts para a execução do projeto

Visando facilitar a execução tanto para os testes durante o desenvolvimento do projeto quanto para a execução nos dois sistemas, foram elaborados quatro scripts para rodar o projeto tanto nos sistemas Linux quanto nos sistemas Windows:

- **`build-complete.bat`**: Script que remove todos arquivos `.class` e os gerados pelo ANTLR visando limpar o projeto. Depois o script compila todas as classes e executa o ANTLR para gerar as classes do interpretador nos sistemas Windows;
- **`build-complete.sh`**: Script que remove todos arquivos `.class` e os gerados pelo ANTLR visando limpar o projeto. Depois o script compila todas as classes e executa o ANTLR para gerar as classes do interpretador nos sistemas Linux;
- **`runSintaticVariosTestes.bat`**: Arquivo em batch. É um script utilizado para executar analisador sintático em vários arquivos de teste de acordo com o diretório escrito na classe **TestParser** em máquinas com o sistema operacional Windows;
- **`runSintaticVariosTestes.sh`**: Arquivo em sheel script. É um script utilizado para executar o analisador sintático em vários arquivos de teste de acordo com o diretório escrito na classe **TestParser** em máquinas com o sistema operacional Linux;
- **`runInterpreterVariosTestes.bat`**: Arquivo em batch. É um script utilizado para executar o interpretador em varios arquivos de teste de acordo com o diretório escrito na classe **TestVisitor** em máquinas com o sistema operacional Windows;
- **`runInterpreterVariosTestes.sh`**: Arquivo em shell script. É um script utilizado para executar o interpretador em varios arquivos de teste de acordo com o diretório escrito na classe **TestVisitor** em máquinas com o sistema operacional Linux;
- **`runInterpreterUnicoArquivo.bat`**: Arquivo em batch. É um script utilizado para executar o interpretador em um unico arquivo passado dentro do script em máquinas com o sistema operacional Windows;

- **`runInterpreterUnicoArquivo.sh`**: Arquivo em shell script. É um script utilizado para executar o interpretador em um unico arquivo passado dentro do script em máquinas com o sistema operacional Linux;

- **`runTypeCheckerComInterpreterUnicoArquivo.bat`**: Arquivo em batch. É um script utilizado para executar o interpretador(Com teste de tipos do analisador semântico) em um único arquivo passado dentro do script em máquinas com o sistema operacional Windows;

- **`runTypeCheckerComInterpreterUnicoArquivo.sh`**: Arquivo em shell script. É um script utilizado para executar o interpretador(Com teste de tipos do analisador semântico) em um único arquivo passado dentro do script em máquinas com o sistema operacional Linux;

- **`runTypeCheckerUnicoArquivo.bat`**: Arquivo em batch. É um script utilizado para executar o teste de tipos do analisador semântico em um único arquivo passado dentro do script em máquinas com o sistema operacional Windows;

- **`runTypeCheckeUnicoArquivo.sh`**: Arquivo em shell script. É um script utilizado para executar o teste de tipos do analisador semântico em um único arquivo passado dentro do script em máquinas com o sistema operacional Linux;

- **`runTypeCheckerVariosTestes.bat`**: Arquivo em batch. É um script utilizado para executar o analisador semântico de tipos em vários arquivos de teste de acordo com o diretório escrito na classe **TestSemantic** em máquinas com o sistema operacional Windows;

- **`runTypeCheckerVariosTestes.sh`**: Arquivo em shell script. É um script utilizado para executar o analisador semântico de tipos em vários arquivos de teste de acordo com o diretório escrito na classe **TestSemantic** em máquinas com o sistema operacional Linux;

Como o projeto deve rodar no Linux, conforme a especificação do trabalho, a seguir serão listadas as linhas de comando presente nos scripts e seu significado para compilar e executar o projeto:

- **`rm *.class`**: Remove os arquivos *.class* presentes no diretório do projeto, ou seja, limpa e remove os arquivos compilados das classes java visando evitar erros durante a próxima execução;

- **Para limpar os arquivos gerados pelo analisador sintático**: Serão utilizados os seguintes comandos:

- **rm *.interp**
- **rm *.tokens**
- **rm *.class**
- **rm LangLexer.***
- **rm LangParser.***
- **rm LangBaseListener.***
- **rm LangListener.***

- **java -cp lib/ANTLR.jar:. lang/LangCompiler -bs**: Inicializa o LangCompiler, colocando a dependência do ANTLR, e a opção **-bs**. A partir do analisador sintático montado, será feita a execução dos testes presente na diretório `./testes/sintaxe/certo` ou no diretório que estiver na escrito na classe **TestParser**, e será impresso na tela quais arquivos passaram no teste de análise sintática, quais não passaram e os erros encontrados.

  Diretório de execução do comando: : **./** => Raiz do projeto

- **java -cp lib/ANTLR.jar:. lang/LangCompiler -bsm**: Inicializa o LangCompiler, colocando a dependência do ANTLR, e a opção **-bsm**. A partir do analisador sintático e interpretador montados, será feita a execução dos testes presente na diretório `./testes/semantica/certo` ou no diretório que estiver na escrito na classe **TestVisitor** e será impresso na tela quais arquivos passaram no teste do interpretador, o resultado do interpretador e quais não passaram e os erros encontrados.

  Diretório de execução do comando: : **./** => Raiz do projeto

- **java -cp lib/ANTLR.jar:. lang/LangCompiler -byt**: Inicializa o LangCompiler, colocando a dependência do ANTLR, e a opção **-byt**. A partir do analisador semântico montado, será feita a execução dos testes presente na diretório `./testes/semantica/certo` ou no diretório que estiver na escrito na classe **TestSemantic**, e será impresso na tela quais arquivos passaram no teste de análise semântica, quais não passaram e os erros encontrados.

- **java -jar ../../lib/ANTLR.jar -visitor Lang.g4**: Inicializa a ferramenta ANTLR com o arquivo estrutural da linguagem Lang, em seguida, gera as classes do analisado léxico, sintático da linguagem e o **Visitor**.

  Diretório de execução do comando: : **./lang/parser/**

- **java -cp lib/ANTLR.jar:. lang/LangCompiler -i NomeDoArquivo**: Inicializa o LangCompiler, colocando a dependência do ANTLR, e a opção **-i**. A partir do analisador sintático e interpretador montados, será feita a execução do teste do

interpretador no arquivo com correspondente ao caminho passado e será impresso na tela o resultado do interpretador, impressão do ambiente de desenvolvimento e erros que ocorreram.

Exemplo de nome de arquivo para teste:

`./testes/semantica/certo/teste0.lan`

Diretório de execução do comando: : `./` => Raiz do projeto

- **`java -cp lib/ANTLR.jar:. lang/LangCompiler -ti NomeDoArquivo`**: Inicializa o LangCompiler, colocando a dependência do ANTLR, e a opção **`-ti`**. A partir do analisador semântico e interpretador montados, será feita a execução do teste de tipos e do interpretador no arquivo correspondente ao caminho passado e será impresso na tela o resultado do teste tipos(impressão de erros semânticos, se houver) e o resultado do interpretador, impressão do ambiente de desenvolvimento e erros que ocorreram.

  Exemplo de nome de arquivo para teste:

  `./testes/semantica/certo/teste0.lan`

  Diretório de execução do comando: : `./` => Raiz do projeto

- **`java -cp lib/ANTLR.jar:. lang/LangCompiler -tp NomeDoArquivo`**: Inicializa o LangCompiler, colocando a dependência do ANTLR, e a opção **`-tp`**. A partir do analisador semântico montado, será feita a execução do teste de tipos no arquivo correspondente ao caminho passado e será impresso na tela o resultado do teste tipos(impressão de erros semânticos, se houver).

  Exemplo de nome de arquivo para teste:

  `./testes/semantica/certo/teste0.lan`

  Diretório de execução do comando: : `./` => Raiz do projeto

- **`javac -cp lib/ANTLR.jar:. -d . lang/ast/*.java`**: Compila todas as classes presentes na pasta **ast**, ou seja, as classes base da árvore AST que serão utilizadas no interpretador através do Visitor.

  Diretório de execução do comando: : `./` => Raiz do projeto

- **`javac -cp lib/ANTLR.jar:. -d . lang/parser/*.java`**: Compila todas as classes presentes na pasta **parser**, ou seja, as classes geradas pelo ANTLR, as recebidas na especificação do trabalho e as implementadas.

  Diretório de execução do comando: : `./` => Raiz do projeto

- **`javac -cp lib/ANTLR.jar:. -d . lang/interpreter/*.java`**: Compila todas as classes presentes na pasta **interpreter**, ou seja, basicamente as principais classes são o Visitor, interpretador e o adaptador da padrão parseTree para Node.

  Diretório de execução do comando: : **`./`** => Raiz do projeto

- **`javac -cp lib/ANTLR.jar:. -d . lang/semantic/*.java`**: Compila todas as classes presentes na pasta **semantic**, ou seja, todas as classes dos tipos do analisador semântico e o visitor TypeCheck.

  Diretório de execução do comando: : **`./`** => Raiz do projeto

- **`javac -cp lib/ANTLR.jar:. -d . lang/LangCompiler.java`**: Compila a classe principal do projeto responsável por inicializar o interpretador e o analisador sintático da linguagem Lang.

  Diretório de execução do comando: : **`./`** => Raiz do projeto

  Observação: : **Como o projeto trabalha com dependências de outras bibliotecas ou ferramentas, neste caso o ANTLR.jar, deve ser informado por parâmetro ao java que o usuário deseja compilar ou executar o projeto utilizando aquela dependência. Assim, o diretório de execução do comando é extremamente importante, pois se não for respeitado, pode acontecer de não reconhecer os atributos e classes internas do ANTLR e impossibilitar a compilação e execução do projeto.**

Portanto, para executar o script no Windows, basta apenas executar o arquivo **buildcomplete.bat** para construir o projeto e o arquivo **runTypeCheckerComInterpreterUnicoArquivo.bat** ou **runTypeCheckerComInterpreterUnicoArquivo.sh** para rodar o projeto. Contudo, para executar no Linux é necessário utilizar o terminal, se atentando de o terminal estar no diretório correto, e também usar o reconhecedor de arquivos *Shell Script* com o comando:

- **`bash build-complete.sh`** ou **`sh build-complete.sh`** $\Rightarrow$ Para construir o projeto;
- **`bash run.sh`** ou **`sh run.sh`** $\Rightarrow$ Para executar o projeto;

2.3 Estrutura dos Arquivos

Dentro do arquivo enviado estão presentes todos os arquivos do projeto. Segue abaixo uma descrição de cada arquivo e sua funcionalidade no projeto:

- **`./compileOnly.bat`**: Arquivo em batch. É utilizado apenas para compilar as classes existentes do projeto nos sistemas operacionais Windows;

OBS: Ideal para uso, desde que tenha executado a build-complete do projeto primeiro;

- **`./compileOnly.sh`**: Arquivo em shell script. utilizado apenas para compilar as classes existentes do projeto nos sistemas operacionais Linux;

  OBS: Ideal para uso, desde que tenha executado a build-complete do projeto primeiro;

- **`./build-complete.bat`**: Arquivo em batch. É um script utilizado para limpar a pasta do projeto, compilar, construir o projeto do interpretador em máquinas com o sistema operacional Windows;

- **`./build-complete.sh`**: Arquivo em shell script. É utilizado para limpar a pasta do projeto, compilar, construir e executar o projeto do interpretador em máquinas com o sistema operacional Linux;

- **`./runSintaticVariosTestes.bat`**: Arquivo em batch. É um script utilizado para executar analisador sintático em vários arquivos de teste de acordo com o diretório escrito na classe **TestParser** em máquinas com o sistema operacional Windows;

- **`./runSintaticVariosTestes.sh`**: Arquivo em sheel script. É um script utilizado para executar o analisador sintático em vários arquivos de teste de acordo com o diretório escrito na classe **TestParser** em máquinas com o sistema operacional Linux;

- **`./runInterpreterVariosTestes.bat`**: Arquivo em batch. É um script utilizado para executar o interpretador em varios arquivos de teste de acordo com o diretório escrito na classe **TestVisitor** em máquinas com o sistema operacional Windows;

- **`./runInterpreterVariosTestes.sh`**: Arquivo em sheel script. É um script utilizado para executar o interpretador em varios arquivos de teste de acordo com o diretório escrito na classe **TestVisitor** em máquinas com o sistema operacional Linux;

- **`./runInterpreterUnicoArquivo.bat`**: Arquivo em batch. É um script utilizado para executar o interpretador em um unico arquivo passado dentro do script máquinas com o sistema operacional Windows;

- **`./runInterpreterUnicoArquivo.sh`**: Arquivo em sheel script. É um script utilizado para executar o interpretador em um unico arquivo passado dentro do script máquinas com o sistema operacional Linux;

- **`./lib/ANTLR.jar`**: Arquivo executável em Java da ferramenta ANTLR. Através dele serão geradas as classes do analisador sintático por meio do arquivo *./lib/parser/Lang.g4* que apresenta uma estrutura que irá comportar a linguagem Lang. Além disso, esta biblioteca é essencial para a execução, compilação e construção do projeto, pois internamente há classes dentro do .jar que são usadas no projeto. Portanto, deve-se salientar

que ao compilar o projeto, este arquivo **deve** ser passado como dependência, caso contrário, haverá erros;

- **`./lang/LangCompiler.java`**: Classe auxiliar do projeto enviada pelo professor com objeto de facilitar a realização da bateria de testes sintáticos do analisador sintático da linguagem Lang e também chama o interpretador para os testes semânticos;
- **`./lang/parser/Lang.g4`**: Arquivo com as especificações da estrutura léxica e sintática da linguagem Lang. É utilizado pelo ANTLR para gerar um analisador sintático descendente da linguagem;
- **diretório `./testes`**: Contém códigos e expressões que serão verificadas pelo analisador sintático e códigos para teste no interpretador voltado para a parte semântica, ou seja, se o código passado pertence a estrutura da linguagem Lang. É uma bateria de testes sintáticos e semânticos conforme enviado pelo professor junto ao enunciado do trabalho;
- **`./lang/ast/`**: Diretório que armazena todas as classes dos nós da AST que serão utilizadas no Visitor pára o interpretador.
- **`./lang/interpreter/Visitor.java`**: Classe Abstrata que declara todas as funções de visita do Visitor para cada elemento presente na AST.
- **`./lang/interpreter/Visitable.java`**: Inteface que implementa em cada node o método de aceitação do Visitor, e quando o Visitor passar por ele, será utiliza o método de visita e assim, executará a funcionalidade do nó.
- **`./lang/interpreter/VisitorAdapter.java`**: É uma classe de transformação, também pode ser chamada de "AntLRToNode“, de modo que ela implementa a mudança no padrão de parseTree, presente na ferramenta, para comportar o tipo Node implementado no trabalho, assim, observa o contexto de cada regra na gramática e instância um elemento do tipo Node com o objetivo de criar uma AST de Nodes.
- **`./lang/interpreter/InterpretVisitor.java`**: Dado que a montagem dos nós foi realizada com sucesso, nesta classe será implementado os métodos de visita em cada nó por parte do Visitor, ela é a base de todo o ambiente de execução do interpretador com hashs armazenando as variáveis instanciadas e até as funções presentes no código fonte da linguagem Lang.
- **`./lang/interpreter/InterpreterAdaptor.java`**: Interface auxiliar do projeto parecida com `ParserAdaptor.java` enviada pelo professor com objeto de facilitar a realização da bateria de testes sintáticos do analisador sintático da linguagem Lang. Contudo, implementei esta interface para ser a mesma ideia do ParseAdaptor só que voltado para o método do interpretador;

- **`./lang/interpreter/InterpreterAdaptorImplementation.java`**:

  Classe auxiliar do projeto que implementa o método **interpretFile** da interface **InterpretAdaptor** e torna possível a realização da bateria de testes sintáticos do analisador sintático e a execução do interpretador da linguagem Lang.

- **`./lang/interpreter/TestVisitor.java`**: Classe auxiliar do projeto parecida com a TestParser.java enviada pelo professor com objeto de facilitar a realização da bateria de testes sintáticos do analisador sintático da linguagem Lang, contudo, elaborei esta classe para interpretar vários testes para todos arquivos da diretório especificada na classe.

- **`./lang/ast/Node.java`**: Classe auxiliar do projeto criada para que receba por herança os métodos e atributos da classe abstrata **SuperNode** e finalmente seja possível utilizar o método **parseFile** no Analisador Sintático. Vale ressaltar que o tipo Node é a estrutura base para todos os elementos do Visitor do Intrepretador.

- **`./lang/ast/SuperNode.java`**: Classe abstrata auxiliar do projeto enviada pelo professor com objeto de facilitar a realização da bateria de testes sintáticos do analisador sintático da linguagem Lang. É o tipo de dado retornado pelo método parseFile.

- **`./lang/parser/TestParser.java`**: Classe auxiliar do projeto enviada pelo professor com objeto de facilitar a realização da bateria de testes sintáticos do analisador sintático da linguagem Lang. A ideia é utilizar a classe que implementou o método *parseFile*, e retornar o resultado da bateria de testes do analisador sintático para todos arquivos da diretório `./testes/sintaxe/certo/`;

- **`./lang/parser/ParseAdaptor.java`**: Interface auxiliar do projeto enviada pelo professor com objeto de facilitar a realização da bateria de testes sintáticos do analisador sintático da linguagem Lang. Foi implementado uma classe auxiliar (**ParseAdaptorImplementation.java**) para implementar o método **parseFile** e possibilitar que os testes sintáticos fossem executados;

- **`./lang/parser/ParseAdaptorImplementation.java`**: Classe auxiliar do projeto que implementa o método **parseFile** da interface **ParseAdaptor** e torna possível a realização da bateria de testes sintáticos do analisador sintático da linguagem Lang.

- **`./lang/semantic/TypeCheckVisitor.java`**: Classe principal do analisador semântico, pois a partir dela que será montado ambiente para capturar os tipos dos elementos da linguagem e, em seguida, apresentar os erros de inconsistência.

- **`./lang/semantic/SemanticAdaptor.java`**: Interface auxiliar do projeto parecida com `ParserAdaptor.java` enviada pelo professor com objeto de facilitar

a realização da bateria de testes sintáticos do analisador sintático da linguagem Lang. Contudo, implementei esta interface para ser a mesma ideia do ParseAdaptor só que voltado para o método do analisador semântico;

- **`./lang/semantic/SemanticAdaptorImplementation.java`**:

  Classe auxiliar do projeto que implementa o método **parseFile** da interface **SemanticAdaptor** e torna possível a realização da bateria de testes de tipos do analisador semântico.

- **`./lang/semantic/TestSemantic.java`**: Classe auxiliar do projeto parecida com a TestParser.java enviada pelo professor com objetivo de facilitar a realização da bateria de testes de sintaxe do analisador sintatico da linguagem Lang, contudo, elaborei esta classe para o teste de tipos do analisador semântico para todos arquivos da diretório especificada na classe.

2.4 Decisões de Projeto

- 01: Foi necessário realizar correções no interpretador, pois no terceiro trabalho, o interpretador foi entregue com uma falha para executar matrizes e funções recursivas, contudo, foi resolvido o problema;

- 02: A classe **ObjectDefault.java** que foi criada para trabalhar com objeto heterogêneo(Tipo data) e seu tipo, teve seu uso expandido para tipos data, matriz, arrays e tipos comuns. Ela é fundamental para o interpretador, pois com ela, é mais fácil saber o tipo do objeto que está armazenado, e portanto, passar em uma chamada de função se torna mais fácil e com menos propensão a erros. Inclusive, através deste tipo, foi possível criar o controle de sobrecarga de funções no interpretador;

- 03: O interpretador inicialmente trabalhava somente com uma tabela hash para armazenar as funções, contudo, para trabalhar com sobrecarga de funções e possibilitar que mais uma função tenham o mesmo nome, foi necessário trabalhar com uma tabela hash de listas de funções, e quando for consultada a função, será feita a verificação dos tipos do parâmetro e passada a função correta para ser executada.

- 04: Conforme foi solicitado que todos os programas devem apresentar uma função 'main' sem parâmetros, durante a resolução do problema de sobrecarga de funções definiu-se que só poderia haver uma única função 'main' no arquivo, deste modo, a única função que não aceita sobrecarga e outras funções com o mesmo nome é a 'main'.

- 05: Como na descrição do enunciado foi explicitado que não poderiam ter operações aritméticas com tipos diferentes, logo, não seria possível operações entre Int e Float

e assim, foi retirado do projeto essa possibilidade, portanto, agora somente cada tipo pode operar com ele próprio. Contudo, está gerando inconsistências em algumas operações como **"`num = 50 + 0.5 + 3 + 9.5`"** pois **3** e **50** são reconhecidos no analisador sintático como Números Inteiros, e portanto a operação é inválida, mas naquele contexto os elementos poderiam ser convertidos para Float e fiquei na dúvida se poderia ter deixado essas conversões possíveis ou ter retirado mesmo e como no enunciado pediu para manter somente Float com Float e Int com Int, foi optador por manter esse tipo de operação como não sendo possível.