INF441 - Roteiro para Aula Prática
Análise Semântica - parte I

Faça download do arquivo usando o link a seguir, que contém um projeto Eclipse compactado:
https://drive.google.com/file/d/18Msy4HlNDgs_s_KpqyuU905H8HPPeGv7/view?usp=sharing

Em seguida, descompacte o conteúdo em uma pasta qualquer, em um computador que tenha Eclipse + ANTLR4 instalado.

Abra o IDE Eclipse e importe o projeto descompactado. Antes de iniciar este roteiro, verifique se o IDE Eclipse está devidamente configurado, seguindo as instruções em:

https://drive.google.com/file/d/1TWBsuBXlzZmSvHniZQ80POzoCVSaPbhP/view?usp=sharing

Em seguida, siga as instruções do roteiro para atividades relacionadas à implementação da análise semântica de programas MicroJava.

Instruções Iniciais

Abra o projeto Eclipse mencionado no início deste roteiro.

No pacote *semantics.interfaces*, abra o arquivo *model.jpg* e estude o diagrama de classes representando as entidades desse pacote, que estão relacionadas com os procedimentos para análise semântica de MicroJava. Esse diagrama foi explicado na aula sobre análise semântica, com resumo disponível no PVANET em:
https://www2.cead.ufv.br/sistemas/pvanet/files/conteudo/1627/analisesemantica.pdf

Neste roteiro, vamos implementar a primeira parte da análise semântica de programas MicroJava, processando as **declarações de símbolos** e sua armazenagem em uma tabela de símbolos. A segunda parte, processar o **uso dos símbolos**, será abordada posteriormente em outro roteiro.

Na ferramenta ANTLR, o produto de uma análise sintática bem-sucedida é uma árvore de sintaxe representando o programa analisado. Para realizar operações sobre essa árvore, ANTLR oferece facilidades para se trabalhar com os padrões de projeto *Visitor* e *Listener*. Neste roteiro, iremos utilizar um *Listener* para navegar na árvore de sintaxe enquanto realizamos os procedimentos de análise semântica.

O *listener SemanticListener* está parcialmente implementado e as instruções deste roteiro irão guiá-lo para completar a implementação da primeira parte da análise semântica. Abra o arquivo *SemanticListener.java* e observe os atributos que o *listener* armazena. Entre eles, pode-se ver:

```
    // Tabela de símbolos utilizada durante a análise semântica
    private SymbolTable symbolTable;

    // Pilha auxiliar para acumular informações enquanto navega na árvore
    private Stack<Object> stack = new Stack<Object>();
```

A tabela de símbolos *symbolTable* será utilizada para coletar as declarações à medida que navegamos na árvore de sintaxe. Por exemplo, no trecho de programa abaixo, após processar a declaração da classe "C", a tabela armazenará a informação de que o ambiente atual contém um símbolo de nome "C" representado por um objeto *SymbolClass*. O tipo de "C" será um objeto *TypeClass*, contendo um ambiente (*Environment*) em que os símbolos "a" e "b" estão internamente representados.

```
...

class C {
  int a, b;
}

C x;
...
```

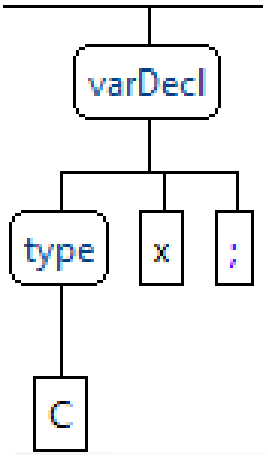
Para ilustrar como a pilha auxiliar *stack* será usada, vamos analisar o que deve acontecer durante o processamento da declaração que vem depois da declaração da classe "C" no trecho apresentado.

A gramática de MicroJava utilizada neste projeto pode ser observada em *grammars/MicroJava.g4*. As produções associadas a declarações de variáveis são:

```
varDecl : type IDENT (',' IDENT)* ';' ;

type:
    IDENT           # TypeWithoutArray
  | IDENT '[' ']'   # TypeWithArray
  ;
```

Utilizando essas produções, a declaração da variável "x" no trecho de programa será representada na árvore de sintaxe como:



O processamento de uma declaração como essa no *SemanticListener* será realizada por dois métodos:

```
    public void exitTypeWithoutArray(...) {
        ...
    }

    public void exitVarDecl(...) {
        ...
    }
```

O método *exitTypeWithoutArray* será chamado quando a navegação terminar de visitar a subárvore cuja raiz é rotulada com "type" na árvore. Este método deverá então procurar o nome "C" na tabela de símbolos e irá encontrar qual é o tipo associado a ele.

Já o método *exitVarDecl* será chamado quando a navegação terminar de visitar toda a subárvore cuja raiz é rotulada com "varDecl" na árvore (já terá visitado "type"

anteriormente). Nesse momento, deve-se obter a lista dos nomes das variáveis que estão sendo declaradas (neste caso, apenas "x") e inserir na tabela de símbolos objetos de tipo *SymbolVariable*, cujo tipo será aquele que foi definido na chamada de *exitTypeWithoutArray*.
PROBLEMA: a informação do tipo encontrado em *exitTypeWithoutArray* não está disponível quando se executa *exitVarDecl*.

A pilha auxiliar *stack* é usada exatamente para realizar a comunicação que foi necessária acima:

- *exitTypeWithoutArray* descobre o tipo associado ao nome "C" e empilha esse símbolo em *stack*;
- em um passo seguinte, *exitVarDecl* desempilha a informação e a usa para definir o tipo da variável "x" que será inserida na tabela de símbolos.

Você irá constantemente encontrar esse tipo de uso da pilha auxiliar *stack* em *SemanticListener*. Os métodos irão descobrir informações e empilhar essas informações em *stack*, para serem depois usadas por outros métodos executados posteriormente. Essa técnica é frequentemente utilizada com o padrão *Listener*.

Importante: não confunda a pilha auxiliar *stack* com a pilha de ambientes da tabela de símbolos. Uma tabela de símbolos é formada por ambientes empilhados, mapeando identificadores a símbolos. Já a pilha *stack* tem como objetivo a comunicação de valores entre os métodos do *listener*.

Retorne ao arquivo *SemanticListener* e siga as instruções que estão dentro do próprio arquivo, em forma de comentários. Em alguns pontos, está inserido um comentário

```
//TODO
```

que significa "to do" (a ser feito). Esse comentário ressalta pontos em que você deverá inserir novo código Java para implementar partes da análise semântica.

No programa piloto, você pode encontrar o comando

```
symbolTable.setDebug(true);
```

que liga o "debug" da tabela de símbolos. Quando ligado, toda inserção de símbolo ou empilhamento/desempilhamento de ambientes na tabela é reportada na saída padrão. Pode ser útil manter esse debug ligado enquanto as atividades desse roteiro são desenvolvidas, para facilitar o acompanhamento.

Instruções para entrega

Entregue o arquivo **SemanticListener.java** construído.