FACULDADE: CENTRO UNIVERSITÁRIO DE BRASÍLIA – UniCEUB
CURSO: CIÊNCIA DA COMPUTAÇÃO
DISCIPLINA: CONSTRUÇÃO DE COMPILADORES
CARGA HORÁRIA: 75 H. A. **ANO/SEMESTRE**: 2017/01
PROFESSOR: EDUARDO FERREIRA DOS SANTOS
HORÁRIOS: Quartas às 09h40 e Sextas às 07h40

LABORATÓRIO 03 – ANALISADOR SEMÂNTICO

RESUMO

A construção de compiladores é uma disciplina da computação que tem como objetivo transformar linguagem de programação de alto nível em linguagem de máquina. O ANTLR é uma ferramenta de *parsing* construída em Java que pode ser utilizada para ler, processar, executar ou traduzir texto estruturado em arquivos binários. A utilização da ferramenta ANTLR vai facilitar a implementação de uma gramática de forma a construir um compilador.

OBJETIVOS

Objetivo Geral
Implementar a análise semântica com auxílio da ferramenta ANTLR.

Objetivos Específicos

1. Construir uma gramática eliminando a ambiguidade;
2. Aplicar a precedência na gramática;
3. Realizar a etapa de análise semântica.

EXERCÍCIO 01 – INICIALIZAÇÃO DO PROJETO

Para iniciar o laboratório baixe todos os materiais necessários diretamente da Internet no seguinte endereço: http://owncloud.eduardosan.com/index.php/s/TsqrHm2SEGKuFuG

A infra-estrutura fornecida contém os seguintes diretórios:

```
.
|-- semantics
```

Os arquivos importantes para o projeto estão no diretório semantics, que contém os testes a serem realizados nessa etapa.

Para compilar o projeto vamos utilizar a ferramenta ant[1] para compilar projetos em Java. Para iniciar o trabalho com a ferramenta, realize os seguintes passos:

1 Mais informações em http://ant.apache.org/

EXERCÍCIO 01 – INICIALIZAÇÃO DO PROJETO

1. Crie um repositório público para o projeto ou adiciona a um já existente;
2. Importe o código para dentro do Projeto.

Agora que o projeto já existe e está versionado, é possível compilar os elementos. Para fazê-lo, utilize a seguinte sequência de comandos:

```
# Baixe o repositório git
cd /home/aluno
git clone <url_do_repositorio>

# Copie o código para dentro do repositório recém criado
cp -rp skeleton/* /home/aluno/<diretorio_do_projeto>

# Adicione os arquivos ao git
cd /home/aluno/<diretorio_do_projeto>
git add .
git commit -m “Import do projeto”
git push origin master

# Agora compile o projeto
ant
```

Verifique a saída para ter certeza que o programa foi compilado com sucesso.

EXERCÍCIO 02 – EXECUTANDO O ANALISADOR SEMÂNTICO

Nas etapas anteriores foi possível implementar uma gramática que capaz de realizar a análise léxica e sintática do compilador. Contudo, para a conclusão do projeto será necessário conhecer todo o conjunto de regras necessárias ao compilador. A descrição completa da linguagem pode ser acessada no endereço http://www.eduardosan.com/wp-content/uploads/2017/06/decaf-manual-1.pdf

Essa etapa do projeto consiste nas seguintes tarefas:

1. Criar uma árvore de representação intermediária de alto nível (IR). É possível concluir a tarefa instruindo o ANTLR para criar uma árvore, adicionando algumas ações à gramática para construir uma árvore, ou navegando por uma árvore genérica e construindo você mesmo. Quando o programa estiver executando no modo *debug*, deve ser possível imprimir a árvore de maneira legível para facilitar o *debugging*.
2. Construir as tabelas de símbolos para as classes. Uma tabela de símbolos é um ambiente, ou seja, um mapeamento de identificadores para objetos semânticos tais como declarações de variáveis. Os ambientes são estruturados hierarquicamente, assim como alguns construtos de código-fonte, tais como corpo de classe, corpo de método, corpo de loop, etc.
3. Execute todas as validações semânticas atravessando a árvore e acessando a tabela de símbolos. **Obs.:**As validações em tempo de execução não são necessárias para essa etapa.

EXERCÍCIO 02 – EXECUTANDO O ANALISADOR SEMÂNTICO

Para completar as restrições, altere o arquivo provided/skeleton/src/decaf/DecafLexer.g4 para adicionar as regras que estão faltando na gramática. Ao terminar a alteração, execute novamente o comando do ant e recompile a gramática para validar o resultado.

```
# Agora compile o projeto
ant
```

Deve ser possível executar o compilador com o seguinte programa:

```
# Execute o scanner
java -jar dist/Compiler.jar -target inter ../semantics/legal01
```

A saída no terminal deve ser o relatório de todos os erros encontrados, ou nenhum saída se o programa for compilado com sucesso. Depois de implementar a etapa de análise semântica estática, seu compilador deve ser capaz de detectar e relatar todos os erros estáticos (ou seja, em tempo de compilação) em qualquer programa escrito na linguagem Decaf. Todas as validações realizadas nas etapas anteriores também devem estar contidas na saída do programa.

Seu compilador deve possuir um modo *debug* onde deverá imprimir a árvore Ir e as estruturas de dados da tabela de símbolos. Para mostrar os dados deve ser executado o seguinte comando:

```
# Execute o scanner
java -jar dist/Compiler.jar -target inter -debug ../semantics/legal01
```

SUGESTÕES DE IMPLEMENTAÇÃO

- Pode ser necessário declarar classes para cada um dos nós na sua IR. Na maior parte dos casos, a hierarquia das classes dos nós será bastante similar ao definido na gramática. Uma parte de sua árvore pode ser apresentada da seguinte forma (identação representa herança):

```
abstract class Ir
abstract class   IrExpression
abstract class   IrLiteral
         class     IrIntLiteral
         class     IrBooleanLiteral
         class   IrCallExpr
         class     IrMethodCallExpr
         class     IrCalloutExpr
         class   IrBinopExpr
abstract class IrStatement
         class   IrAssignStmt
         class   IrPlusAssignStmt
         .
         .
         .
         class IrClassDecl
```

SUGESTÕES DE IMPLEMENTAÇÃO

Classes definidas nesse formato implementam a *abstract syntax tree* – AST – do programa de entrada. Na sua forma mais simples, cada classe é apenas uma tupla de suas subclasses.

```
public class IrBinopExpr extends IrExpression
{
    private final int          operator;                    |
    private final IrExpression lhs;                         +
    private final IrExpression rhs;                        / \
}                                                        lhs   rhs

or:

public class IrAssignStmt extends IrStatement
{                                                           :=
    private final IrLocation   lhs;                        /  \
    private final IrExpression rhs;                      lhs    rhs
}
```

Figura 1: Exemplo de classes e suas subárvores

Será ainda necessário definir classes para as entidades semânticas do programa que representam as propriedades abstratas (tipos de expressão, assinaturas de método, descritores, etc.) e estabelecer correspondências entre elas. Alguns exemplos: Cada expressão tem um tipo; cada declaração de variável introduz uma variável; cada bloco define um escopo. Muitas das propriedades são derivadas da árvore recursivamente.

Na medida do possível, as instâncias das classes de símbolo devem ser "canônicas", ou seja, devem ser comparadas utilizando igualdade de referência. Desenhe as classes com carinho, valide o código e, principalmente, escreva a documentação!

- Todas as mensagens de erro devem ser acompanhadas do nome do arquivo, linha, e coluna do token relevante para a mensagem de erro. Significa que, quando estiver construindo sua AST, você deve tentar garantir que cada nó contenha informação suficiente para que você determine o número da linha mais tarde.

  Não é conveniente disparar uma exceção quando encontramos um erro na cadeia de entrada: se você tomar essa decisão somente uma mensagem de erro será apresentada a cada execução do compilador. Uma boa escolha para o front-end é empilhar os erros e mostrar todos os que foram encontrados ao final da execução.

- A análise semântica deve ser feita utilizando a abordagem **top-down**. Enquanto o componente de verificação de tipos pode ser realizado utilizando a abordagem *bottom-up*, outros tipos não podem. Ex.: detecção de variáveis não utilizadas.

SUGESTÕES DE IMPLEMENTAÇÃO

Existem duas formas de atingir esse objetivo. A primeira é utilizar o parser no meio das produções. Essa abordagem pode requerer menos código mas pode ser mais complexa, já que mias trabalho precisa ser feito diretamente na estrutura do ANTLR.

Uma abordagem mais "limpa" é chamar o seu analisador semântico numa AST completa depois de executar o *parsing*. O pseudocódigo para a *block* seria aproximadamente:

```
void checkBlock(EnvStack envs, Block b) {
  envs.push(new Env());
  foreach s in b.statements
    checkStatement(envs, s);
  envs.pop();
}
```

No pseudocódigo apresentado um novo ambiente é criado e enviado à pilha do ambiente; o corpo de *block* é avaliado no contexto da pilha do ambiente e o novo ambiente é descartado ao atingir o fim da execução.

A análise semântica é então expressada como um *visitor* no AST. Já que a geração de código, algumas otimizações e impressão dos resultados podem ser implementados utilizando *visitors*, é a abordagem recomendada para uma implementação mais limpa.

- O tratamento de inteiros literais negativos requer algum cuidado. É possível perceber do laboratório anterior que inteiros literais negativos são, na verdade, dois tokens: o inteiro positivo literal precedido por um símbolo '-'. Sempre que o seu analisador top-down encontrar um operador unário negativo, deve verificar se o seu operando é um inteiro literal positivo; se for, deve substituir sua árvore por um único inteiro literal negativo.

BIBLIOGRAFIA

PARR, Terence. **The definitive ANTLR 4 reference**. Pragmatic Bookshelf, 2013. Disponível em http://www4.di.uminho.pt/~gepl/GQE/documents/books/Pragmatic.The.Definitive.ANTLR.4.Reference.Jan.2013.pdf

AHO, Alfred V. E Outros. **Compiladores: princípios, técnicas e ferramentas**. PEARSON, 2007.

PRICE, Ana Maria de Alencar. **Implementação de linguagens de programação: compiladores**. SAGRA-LUZZATTO, 2005.

Material inspirado no disponível em https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-035-computer-language-engineering-spring-2010/