Compiladores
Roteiro de Laboratório 03 – Tabelas de Símbolos e de *Strings*

Parte I
Visitadores no ANTLR

1 Introdução

Durante a execução do terceiro módulo do compilador realizamos a *análise semântica*. No caso de uma linguagem natural, é neste momento em que se atribui significado às palavras a depender do contexto. No ambiente de um compilador o processo é bem semelhante: este é o momento em que há a associação de significado (semântica) à sintaxe do programa de entrada. A análise semântica envolve verificar a compatibilidade entre tipos, o uso apropriado de escopo, dentre várias outras verificações.

Nos dois módulos anteriores nós utilizamos o ANTLR para realizar a geração automática do *lexer* e do *parser*. No entanto, a partir de agora não há mais uma forma automatizada de construção dos próximos módulos. Assim, estes devem ser programados manualmente.

Neste laboratório iremos implementar e utilizar duas estruturas essenciais para o compilador: as tabelas de símbolos e de literais (*strings*). A tabela de símbolos é um repositório de dados sobre elementos do programa de entrada criados pelo usuário, por exemplo as variáveis e funções. Uma tabela de símbolos para variáveis pode armazenar, por exemplo, o nome, tipo e a linha em que a variável foi declarada, além do seu escopo, entre outros. Já a tabela de literais serve para armazenar e referenciar as *strings* da entrada, evitando que elas sejam manipuladas por outros módulos sem necessidade.

Partindo destas tabelas, vamos implementar verificações básicas de semântica muito comuns em linguagens de programação puramente estáticas, tais como C, Java, etc. Vamos garantir que as variáveis sejam declaradas antes do seu uso, e que não há colisão de nomes. No próximo laboratório, vamos expandir o analisador semântico para incluir também algumas verificações de tipos.

2 Além do *lexer* e *parser* no ANTLR

A partir deste ponto, o funcionamento do ANTLR começa a diferir bastante do `flex` e do `bison`. Em particular, no `bison` praticamente tudo fica no arquivo `.y`, principalmente a função `main()`, com as ações semânticas incluindo código no meio das regras da gramática.

Embora isso também seja possível no ANTLR, a documentação da ferramenta recomenda que todo o código fique fora do arquivo da gramática. Basicamente o analisador sintático do ANTLR entrega a *parse tree* da entrada, e a partir daí tudo é feito através de caminhamentos nesta *parse tree*. O ANTLR já provê classes padrão para ajudar no caminhamento da *parse tree*. Há duas formas gerais de implementação: *listeners* e *visitors*.

Quando se usa um *listener*, o ANTLR já caminha automaticamente em profundidade (DFS – Depth First Search) pela *parse tree* e vai chamando os métodos correspondentes ao tipo do nó atual no caminhamento. Por ser uma forma de caminhamento fixa, ela é mais limitante, e assim não vamos usar *listeners*.

A outra forma de se caminhar pela *parse tree* é com um *visitor*. Neste caso, ao chegar em um nó, o método correspondente do nó é chamado. Mas, ao contrário do *listener*, no *visitor* o programador é quem define como deve ser feito o caminhamento na subárvore subsequente, ficando livre para fazer a visitação dos filhos na ordem que quiser. Isto será explicado melhor adiante.

2.1 Atualizando o comando de execução do ANTLR

No Laboratório 02, nós usamos o seguinte comando do ANTLR:

```
$ antlr4 -no-listener -o dir_saida gramatica.g
```

A opção `-no-listener` obviamente indica que não queremos uma implementação base de um *listener*. Como vamos precisar agora de um *visitor*, o comando do ANTLR que vamos utilizar neste roteiro fica assim:

```
$ antlr4 -no-listener -visitor -o dir_saida gramatica.g
```

Lembre-se de atualizar os seus *Makefiles* para incluir a opção `-visitor` como indicado acima.

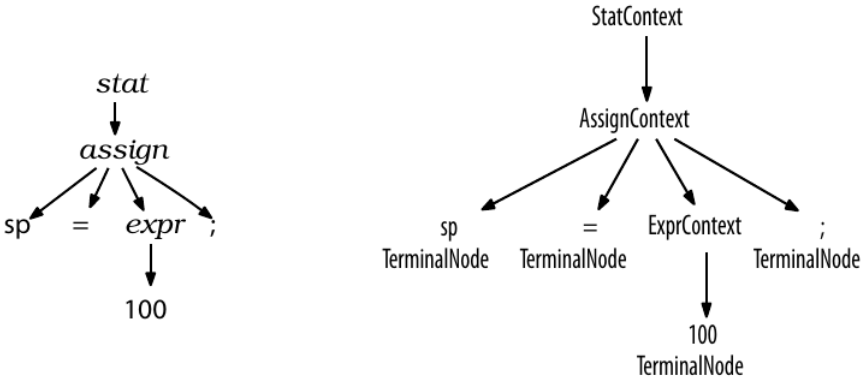
2.2 Visão geral de um *visitor* no ANTLR

Vamos agora tentar entender melhor como um *visitor* interage com a *parse tree* produzida pelo ANTLR, permitindo assim a construção de novas funcionalidades do compilador.

Considere um comando simples de atribuição em C ou Java como abaixo:

```
1 sp = 100;
```

Assuma o desenvolvimento de um *parser* com uma gramática similar a várias das que já vimos até aqui. Assim, temos uma *parse tree* representada pela árvore da esquerda na figura abaixo.



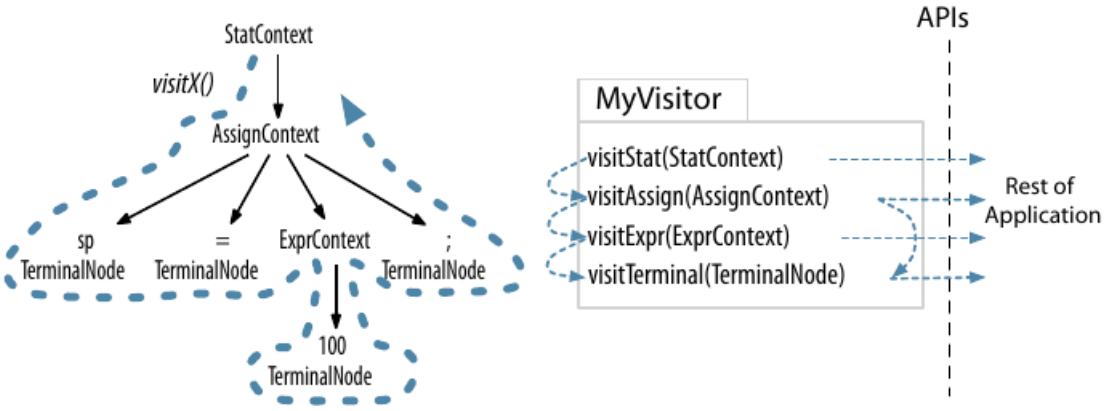
Parse tree | Parse tree node class names

Já a árvore da direita na figura indica como o ANTLR implementa propriamente esta *parse tree*. Todos os nós internos da *parse tree* recebem o sufixo `Context`. Já os nós folhas são implementados como `TerminalNodes`.

Quando executamos o ANTLR com a opção `-visitor`, ele gera dois arquivos `.java` extras. Suponha que o arquivo de entrada utilizado foi `Gram.g`. Sabemos então que o nome da gramática é `Gram`. Este nome serve de prefixo para os dois arquivos criados pelo ANTLR, como veremos a seguir.

O primeiro arquivo gerado é uma interface com o nome `GramVisitor`. Esta interface define todos os métodos para visitação dos nós da árvore. Por padrão, o ANTLR cria um método para cada não-terminal `x` da gramática, com o nome do método sendo definido automaticamente como `visitX()` – note a capitalização! Assim, para a árvore da figura acima, temos os métodos `visitStat()`, `visitAssign()`, etc.

O segundo arquivo de visitação gerado pelo ANTLR é o `GramBaseVisitor`. Esta classe provê uma funcionalidade básica de caminhamento na *parse tree*, aonde cada método `visitX()` realiza somente uma execução recursiva da visitação em todos os seus filhos, da esquerda para a direita. Isto efetivamente corresponde a um caminhamento em profundidade na árvore, como ilustra a linha tracejada na figura a seguir.



Podemos ver na figura acima que dentro do método `visitStat`, é feita uma chamada recursiva para `visitAssign`, que por sua vez visita recursivamente todos os seus filhos, e assim por diante, até que a recursão termine nas folhas, momento em que as chamadas retornam para o nó pai.

Para a nossa conveniência, o ANTLR provê um método genérico `visit`, que pode ser aplicado a qualquer nó da *parse tree*, com este método genérico ficando responsável por fazer o despacho dinâmico para o método adequado conforme o tipo do nó sendo visitado. Assim, para iniciar o caminhamento pela árvore, basta que o nosso compilador inclua um trecho de código como abaixo:

```
1 ParseTree tree = ... ; // tree is result of parsing
2 MyVisitor v = new MyVisitor();
3 v.visit(tree);
```

Os exemplos da próxima seção vão detalhar melhor essas informações sobre a implementação.

A geração automática de código do *visitor* pelo ANTLR é bem conveniente pois evita a necessidade de criação de código *boiler plate*, com implementações padrão para todos os métodos de visitação. Com isso, não é necessário fazer uma sobrescrita de todos os métodos da interface, o que nos permite concentrar somente nos métodos de nosso interesse.

3 Exemplos básicos de visitadores

Vamos agora apresentar alguns detalhes adicionais do uso de visitadores através de dois exemplos simples.

3.1 Exemplo 01 – Calculando expressões de soma e multiplicação

Neste primeiro exemplo, vamos implementar uma calculadora extremamente limitada que computa o resultado de expressões com operadores de soma e multiplicação. A gramática do ANTLR para este exemplo pode ser vista a seguir.

```
1  grammar Exemplo01;
2
3  @header {
4      package parser;
5  }
6
7  expr:
8      term PLUS term
9  ;
10 term:
11     factor TIMES factor
12 ;
13 factor:
14     NUMBER
15 ;
16
17 NUMBER: [0-9]+ ;
18 PLUS:   '+' ;
19 TIMES:  '*' ;
20 WS:     [ \t\n]+ -> skip;
```

Algumas coisas já são familiares pois foram vistas no laboratório anterior. Na linha 1 temos o nome da gramática. Note que estamos usando somente `grammar` em vez de `lexer grammar` ou `parser grammar` neste cabeçalho, pois o arquivo deste exemplo inclui tanto as regras sintáticas quanto léxicas. Falando em regras léxicas, elas estão indicadas nas linhas 17-20. Temos somente três tipos de *tokens*, que devem ser auto-explicativos a esta altura. Espaços em brancos, tabulações e quebra de linha são descartados.

Uma novidade está nas linhas 3-5, aonde utilizamos o comando `@header` para instruir o ANTLR a incluir um trecho de código no cabeçalho de todos os arquivos gerados. A partir de agora, vamos ter de escrever código Java para desenvolver o nosso compilador, então ajuda bastante na organização deixar todos os arquivos gerados automaticamente pelo ANTLR em um pacote específico. No caso, estamos chamando este pacote de `parser`.

As linhas 7-15 da gramática acima definem as regras sintáticas. Temos que uma expressão aritmética (`expr`) é formada pela soma de dois termos (`term`), que por sua vez são formados pela multiplicação de dois fatores (`factor`), que finalmente reduzem a números. Assim, vemos de imediato que a gramática admite somente expressões da forma `a*b+c*d`. Nos exemplos seguintes vamos remover essa limitação.

As regras sintáticas foram divididas em expressão, termo e fator para garantir que a precedência dos operadores seja respeitada. Basicamente, para cada nível de precedência, criamos um novo símbolo na gramática. Com isto, é possível se definir adequadamente a precedência dos operadores somente pelas regras de reescrita. (Isto foi explicado nos slides da Aula 02.) Por outro lado, em geral essa forma de resolução de precedência deixa a gramática mais complexa e vamos preferir, sempre que possível, resolver precedência pela ordem de declaração dos operadores, como visto no roteiro do Laboratório 02. A gramática deste exemplo foi criada desta

forma somente para simplificar a implementação do *visitor* depois. Uma vez criada a gramática, podemos executar o ANTLR:

```
$ antlr4 -no-listener -visitor -o parser Exemplo01.g
```

Além dos arquivos do *lexer* e *parser* já conhecidos, agora o ANTLR também criou a interface e a implementação base do *visitor*, nos respectivos arquivos `Exemplo01Visitor.java` e `Exemplo01BaseVisitor.java`. Abra estes arquivos e dê uma olhada no código gerado.

Para a implementar a nossa calculadora, vamos estender as funcionalidades do visitador base, como ilustra o código abaixo.

```
1  package calc;
2
3  import parser.Exemplo01Parser;
4  import parser.Exemplo01BaseVisitor;
5
6  public class Calculator extends Exemplo01BaseVisitor<Integer> {
7
8      // Visita a regra expr: term PLUS term
9      public Integer visitExpr(Exemplo01Parser.ExprContext ctx) {
10         // Visita recursivamente a expressao da esquerda
11         int l = visit(ctx.term(0));
12         // Visita recursivamente a expressao da direita
13         int r = visit(ctx.term(1));
14         return l + r;
15     }
16
17     // Visita a regra term: factor TIMES factor
18     @Override
19     public Integer visitTerm(Exemplo01Parser.TermContext ctx) {
20         // Visita recursivamente a expressao da esquerda
21         int l = visit(ctx.factor(0));
22         // Visita recursivamente a expressao da direita
23         int r = visit(ctx.factor(1));
24         return l * r;
25     }
26
27     // Visita a regra factor: NUMBER
28     @Override
29     public Integer visitFactor(Exemplo01Parser.FactorContext ctx) {
30         return Integer.parseInt(ctx.NUMBER().getText());
31     }
32
33 }
```

Cada um dos três métodos da classe `Calculator` serve para visitar uma das regras da gramática. No caso do método `visitExpr` (linha 9), precisamos caminhar recursivamente nas duas expressões dos termos (linhas 11 e 13). Note como todo o acesso aos elementos do corpo da regra (`expr`) é feito através do objeto de contexto (`ctx`) passado para o visitador. Quando temos vários símbolos iguais na regra (`term`), basta acessá-los individualmente por

índices. Uma vez que a visitação recursiva dos termos termina, obtemos como retorno o resultado das sub-expressões como dois inteiros `l` e `r`. (Isto funciona porque instanciamos a classe genérica do visitador base com o tipo de retorno `Integer`.) Por fim, para calcular o valor da expressão, basta realizar a soma dos termos e retornar o resultado (linha 14).

O funcionamento da regra de visitação dos termos funciona de forma análoga. Por fim, na linha 30 temos a conversão do lexema de um número inteiro no seu valor correspondente. Esta é a base da recursão do caminhamento pela *parse tree*.

Até o laboratório passado, estávamos usando o `TestRig` do ANTLR para executar o *lexer* e o *parser*. No entanto, a partir de agora vamos precisar criar uma função principal que execute a nossa calculadora depois da análise sintática. Temos então a classe `Main` como a seguir.

```
1  import java.io.IOException;
2
3  import org.antlr.v4.runtime.CharStream;
4  import org.antlr.v4.runtime.CharStreams;
5  import org.antlr.v4.runtime.CommonTokenStream;
6  import org.antlr.v4.runtime.tree.ParseTree;
7
8  import calc.Calculator;
9  import parser.Exemplo01Lexer;
10 import parser.Exemplo01Parser;
11
12 public class Main {
13
14     public static void main(String[] args) throws IOException {
15         // Cria um CharStream que le os caracteres de stdin.
16         CharStream input = CharStreams.fromStream(System.in);
17         // Cria um lexer que consome a entrada do CharStream.
18         Exemplo01Lexer lexer = new Exemplo01Lexer(input);
19         // Cria um buffer de tokens vindos do lexer.
20         CommonTokenStream tokens = new CommonTokenStream(lexer);
21         // Cria um parser que consome os tokens do buffer.
22         Exemplo01Parser parser = new Exemplo01Parser(tokens);
23         // Inicia o processo de parsing na regra 'expr'.
24         ParseTree tree = parser.expr();
25         if (parser.getNumberOfSyntaxErrors() != 0) {
26             // Houve um erro sintatico. Termina a compilacao aqui.
27             return;
28         }
29         // Cria a calculadora e visita a ParseTree para computar.
30         Calculator calc = new Calculator();
31         int result = calc.visit(tree);
32         // Saida final.
33         System.out.println("Result = " + result);
34     }
35
36 }
```

Os comentários no código acima devem torná-lo auto-explicativo. É interessante notar como o processo de compilação no ANTLR é totalmente sequencial, isto é, cada etapa de compilação

é feita uma após a outra. Isto contrasta totalmente quanto ao comportamento do `bison`, que realiza todas as análises "ao mesmo tempo".

Uma vez que o nosso código Java estiver completo, basta compilar:

```
$ javac -d bin */*.java Main.java
```

Lembre novamente que os comandos do `antlr4` e `javac` na verdade são executados pelo `Makefile` com o *target* `make`. Dê uma olhada no código deste arquivo caso tenha alguma dúvida.

Por fim, podemos executar a calculadora para testar.

```
$ make run <<< "2*3+4*5"
Result = 26
```

3.2 Exemplo 02a – Calculando as quatro operações aritméticas

Vamos agora modificar o exemplo anterior para que a nossa calculadora aceite as quatro operações aritméticas básicas. Começamos modificando a gramática para usar o formato usual de prioridade de operadores do ANTLR. As regras sintáticas da nova gramática ficam como abaixo.

```
1 line:
2   expr
3 ;
4
5 expr:
6   expr (TIMES | OVER) expr
7 | expr (PLUS | MINUS) expr
8 | NUMBER
9 ;
```

A regra `line` foi adicionada somente porque o ANTLR não aceita que a regra inicial da gramática seja recursiva. As demais regras de `expr` seguem o mesmo formato já visto no laboratório passado. (Lembrando mais uma vez que os operadores definidos primeiro têm mais prioridade.)

Após executar o ANTLR para esta nova gramática, chegamos em um problema fundamental: o visitador gerado possui somente um método `visitExpr`! Na verdade, sabemos que temos de fato três regras (uma para cada linha em 6-8), mas como todas possuem a mesma cabeça (`expr`), o ANTLR tenta tratar todas com um único método. Vamos corrigir esse problema na nova versão deste exemplo a seguir.

3.3 Exemplo 02b – Visitando múltiplas regras com a mesma cabeça

A solução para o problema que acabamos de encontrar é nomear individualmente cada uma das regras de `expr`.

```
1 expr:
2   expr (TIMES | OVER) expr  # timesOver
3 | expr (PLUS | MINUS) expr  # plusMinus
4 | NUMBER                    # number
5 ;
```

Usando o comando de #, criamos um nome para cada regra. Execute o ANTLR e veja o visitador gerado. Agora temos os métodos `visitTimesOver`, `visitPlusMinus` e `visitNumber`.

Resolvemos um problema mas ainda temos outro para superar. Veja um trecho do código da calculadora agora.

```
1  // Visita a regra expr: expr (TIMES | OVER) expr
2  public Integer visitTimesOver(Exemplo02Parser.TimesOverContext ctx){
3      // Visita recursivamente a expressao da esquerda
4      int l = visit(ctx.expr(0));
5      // Visita recursivamente a expressao da direita
6      int r = visit(ctx.expr(1));
7      // Como saber qual dos dois operadores foi usado?
8      return 0;
9   }
```

Como indicado na linha 7, não temos como saber qual dos operadores foi utilizado na expressão, e com isso não conseguimos avaliar a expressão. Infelizmente não podemos colocar os operadores em linhas separadas porque isto criaria níveis de precedência distintos entre eles. A solução é mostrada na versão final deste exemplo a seguir.

3.4 Exemplo 02c – Distinguindo operadores com a mesma precedência

É possível resolver o problema anterior introduzindo um nome adicional para os operadores na gramática.

```
1  expr:
2    expr op=(TIMES | OVER) expr  # timesOver
3  | expr op=(PLUS | MINUS) expr  # plusMinus
4  | NUMBER                       # number
5  ;
```

Com isto podemos acessar o *token* do operador pelo nome `op`, como ilustra o código abaixo.

```
1  // Visita a regra expr: expr op=(TIMES | OVER) expr
2  public Integer visitTimesOver(Exemplo02Parser.TimesOverContext ctx){
3      // Visita recursivamente a expressao da esquerda
4      int l = visit(ctx.expr(0));
5      // Visita recursivamente a expressao da direita
6      int r = visit(ctx.expr(1));
7      // Olha qual eh o operador e computa a expressao
8      if (ctx.op.getType() == Exemplo02Parser.TIMES) {
9          return l * r;
10     } else { // OVER -- Programacao defensiva aqui seria bom...
11         return l / r;
12     }
13 }
```

Executando o programa, vemos que o resultado está correto.

```
$ make run <<< "2+3*4"
Result = 14
```

Podemos também utilizar o *target* `make debug` para gerar a figura da *parse tree* construída. Isto ajuda bastante a analisar o comportamento do *visitor*.

Parte II
Construindo um analisador semântico para EZLang

4 Semântica de EZLang para declaração e uso de variáveis

As regras semânticas de EZLang para declaração e uso de variáveis são simples: todas as variáveis do programa de entrada devem ser declaradas antes de serem utilizadas no corpo do programa. Além disso, não é correto redeclarar variáveis, isto é, um identificador (lexema) não pode aparecer mais de uma vez na seção de declaração de variáveis. Sabendo dessas regras, podemos começar a construção do nosso analisador semântico.

4.1 Utilizando as tabelas de símbolos e de *strings*

Realize as atividades abaixo para concluir a tarefa deste laboratório.

ATIVIDADE 0: Baixe o arquivo de código disponibilizados pelo professor no Classroom (arquivo `CC_Lab03_src_java.zip`). Entenda a implementação miserável das tabelas de símbolos e de *strings* fornecida. A ideia aqui é focar na *utilização* das tabelas e não na sua implementação.

ATIVIDADE 1: Utilizando o código disponibilizado, crie um visitador da *parse tree* que inclui todas as *strings* do código fonte na tabela de *strings*.

ATIVIDADE 2: Utilizando o código disponibilizado, modifique o seu visitador do item anterior para incluir as variáveis declaradas na tabela de símbolos. Ao reconhecer uma nova variável, o seu analisador deve verificar se ela já foi declarada consultando a tabela de símbolos. Passando nesse teste, inclua a variável (identificador) na tabela, juntamente com as seguintes informações: linha de declaração e tipo da variável (inteiro, real, etc).

Após preencher a tabela de símbolos com as declarações de variáveis, inclua novos métodos de visitação, agora para verificar todas as ocorrências de variáveis no corpo do programa. Em todas as regras sintáticas aonde aparece um *token* `ID`, teste se a variável foi previamente declarada (basta consultar a tabela de símbolos). Caso o teste falhe, seu compilador deve exibir uma mensagem de erro, como descrito abaixo.

4.2 Mensagens do analisador semântico

O seu analisador semântico deve exibir as seguintes mensagens.

Utilização de variáveis. Se uma variável for utilizada sem ser declarada, imprima no terminal:

```
SEMANTIC ERROR (XX): variable 'VV' was not declared.
```

Onde `XX` é o número da linha do programa onde o erro foi detectado e `VV` é o nome da variável.

Redeclarações de variáveis. Se uma variável for redeclarada no programa de entrada, imprima no terminal:

```
SEMANTIC ERROR (XX): variable 'VV' already declared at line YY.
```

Onde `XX` é o número da linha do programa onde o erro foi detectado, `VV` é o nome da variável e `YY` é o número da linha aonde a variável foi originalmente declarada.

Impressão das tabelas. Ao final do processo de análise, imprima as tabelas de símbolos e de *strings* no terminal. Veja os arquivos de saída disponibilizados para testes.

Demais mensagens. As mensagens de erros léxicos e sintáticos são as mesmas do Laboratório 02 e devem continuar sendo exibidas como antes.

4.3 Implementado as Tabelas de Símbolos e de Literais

ATIVIDADE 3 (opcional): Implemente a sua versão da tabela de símbolos e de *strings*, substituindo a versão disponibilizada pelo professor. Idealmente, você deve implementar uma tabela *hash*.

Algumas observações importantes:

- O seu compilador pode terminar a execução ao encontrar o primeiro erro no programa de entrada.
- Os programas de entrada para teste são os mesmos de sempre (`in.zip`). As saídas esperadas desta tarefa estão no arquivo `out03_java.zip`.
- Uma implementação de referência para este laboratório será disponibilizada pelo professor em um futuro próximo. No entanto, você é *fortemente* encorajado a realizar a sua implementação completa antes de ver uma solução em outro lugar.