Compiladores
Roteiro de Laboratório 02 – Construindo um Analisador Sintático

Parte I
Utilizando o *parser* do ANTLR

1 Introdução

No conteúdo teórico da disciplina, vimos que após a análise léxica, temos a segunda fase do *front-end* do compilador: a análise sintática, realizada pelo *parser*. Existem dois grupos de algoritmos para análise sintática: os *top-down* e os *bottom-up*.

O ANTLR, ferramenta que vamos usar neste laboratório, implementa um algoritmo *top-down* `ALL(*)`, abreviação de *adaptative LL*. Na prática, o que isso quer dizer é que o ANTLR é uma ferramenta muito mais flexível que o `bison` quanto ao tipo de gramática que pode ser utilizada na definição da sintaxe da linguagem de entrada. Enquanto o `bison` aceita somente gramáticas do tipo LALR(1), o ANTLR basicamente aceita quase qualquer gramática livre de contexto, ajustando o algoritmo de execução dinamicamente, conforme a entrada que deve ser analisada.

Na prática, prática mesmo, o que isso quer dizer é que aqueles conflitos infernais de *shift/reduce* e *reduce/reduce* no `bison` são uma coisa do passado, com o ANTLR tratando esses problemas automaticamente. Claro que tudo isso tem um preço; no caso aqui: desempenho. O *parser* gerado pelo `bison` é consideravelmente mais rápido que o *parser* gerado pelo ANTLR. Mas, salvo em cenários bem específicos, a conveniência do ANTLR vale muito mais do que a sua performance frente a outras ferramentas.

2 Uma visão geral do *parser* do ANTLR

2.1 Uma breve descrição do arquivo de entrada

O arquivo de entrada do *parser* é descrito por uma gramática livre de contexto, ou seja, por um conjunto de regras de reescrita. No ANTLR esta gramática pode estar em notação EBNF (*Extended Backus Naur Form*), o que significa que além das regras de reescrita usuais em BNF, podemos também usar alguns elementos de expressões regulares, como os fechos `*` e `+` para indicar repetição, e `?` para indicar elementos opcionais nas regras.

Como já dito várias vezes, sabemos que o ANTLR realiza o papel do `flex` e do `bison` em conjunto. Assim, temos que o ANTLR espera tanto as regras do *parser* quanto do *lexer* em um mesmo arquivo com a extensão `.g`. O formato deste arquivo é apresentado a seguir.

```
1 grammar NOME_DA_GRAMATICA;
2 // Regras sintáticas
3 cabeca_regra: corpo_regra { acao semantica };
4 ...
5 // Regras léxicas
6 TIPO_DO_TOKEN: expressao_regular { acao semantica };
7 ...
```

Lembrando novamente que o nome do arquivo `.g` precisa ter o mesmo nome da gramática. As regras léxicas já foram estudadas no Laboratório 01. Já as regras sintáticas correspondem a uma ou mais regras da gramática que define as possíveis sequências válidas dos *tokens*. Assim como no `bison`, sempre que uma regra é aplicada, é possível associar uma regra semântica (código Java) junto. Isso é feito constantemente no `bison`, mas aqui no ANTLR não é considerado uma boa prática por deixar a gramática mais difícil de ler.

O ANTLR permite separar as regras do *lexer* e do *parser* em arquivos diferentes. Em geral isto é útil em gramáticas maiores, pois deixa as funcionalidades de cada componente melhor separadas. Vamos aprender como fazer essa divisão mais adiante, quando a gente for realizar a tarefa deste laboratório. Por enquanto, dado que todas as gramáticas de exemplo são pequenas, vamos usar tudo em um único arquivo `.g` por ser mais conveniente.

2.2 Gerando o *parser* com o ANTLR

No laboratório anterior vimos que os comandos fundamentais da ferramenta são `antlr4`, para compilar a gramática; e `grun`, para executar o analisador gerado. Vimos também que é preferível substituir estes comandos por uma configuração de um *Makefile*, para manter a sanidade. Agora vamos apresentar algumas novas funcionalidades do ANTLR para a geração do *parser*, utilizando os comandos originais para simplificar. Depois vamos discutir o que muda no *Makefile* que vamos empregar.

Tendo o arquivo da gramática que queremos processar, devemos executar o ANTLR com o comando:

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

Quando o ANTLR implementa o *parser* ele também implementa a construção da *parse tree* correspondente ao arquivo fonte de entrada. Além disto, o ANTLR também provê funções para inspeção e caminhada na árvore; com a forma padrão, mais simples, sendo um `listener`, aonde qualquer modificação na árvore pode disparar um código criado pelo usuário. Nesse momento esta funcionalidade não é necessária, então usamos a opção `-no-listener` para que o ANTLR não gere arquivos extras desnecessariamente. Este ponto será discutido detalhadamente nos próximos roteiros.

O passo seguinte é compilar os arquivos `.java` criados, igual como foi feito no laboratório anterior:

```
$ javac ./dir_saida/*.java
```

Por fim, para executar algum teste, basta rodar:

```
$ cd dir_saida
$ grun nome_da_gramatica regra_inicial
```

O nome da gramática é aquele especificado no cabeçalho do arquivo `.g` empregado. Já a regra inicial da gramática é aquela que permite derivar toda a linguagem de entrada. Por padrão, o ANTLR considera a primeira regra de reescrita que aparece no arquivo como a regra inicial. Ao executar o último comando acima, temos o terminal aberto para digitar a entrada a ser analisada, lembrando que para interromper o processamento temos que digitar `Ctrl+D`.

Assim como feito com o *lexer* do laboratório passado, também é possível analisar a entrada a partir de um arquivo:

```
$ cd dir_saida
$ grun nome_da_gramatica regra_inicial ./caminho/arquivo_de_entrada
```

Quando o processo de análise sintática é concluído com sucesso, não há nenhum retorno no terminal. Caso aconteça algum erro, uma mensagem é exibida na tela. Há também algumas opções do ANTLR para a exibição da *parse tree* que são muito úteis. Elas serão apresentadas junto com alguns exemplos adiante.

2.3 Modificando *Makefiles* anteriores para este laboratório

Assim como foi feito no Laboratório 01, vamos continuar usando *Makefiles* para a configuração, compilação e execução do ANTLR e dos analisadores sintáticos gerados com ele.

Felizmente, podemos reutilizar o formato padrão do *Makefile* do laboratório passado sem maiores alterações. Veja, para começar, o arquivo no diretório do Exemplo 01, disponível em `CC_Lab02_Exemplos_Java.zip`. Foram feitas as seguintes modificações neste arquivo:

- **Linha 19**: Alteração no diretório `GEN_PATH=parser` para onde vão os arquivos do ANTLR.
- **Linha 26**: Alteração do *target* `antlr` para incluir a opção `-no-listener`.
- **Linha 32**: Alteração do *target* `run` para deixar o comando de execução no formato explicado na seção anterior.

3 Exemplos básicos de analisadores sintáticos no ANTLR

3.1 Exemplo 01 - Expressões de comparação

Vamos começar com um exemplo bem simples que reconhece alguns operadores de comparação ($>$, $<$, e $=$) e números inteiros, como definido pela gramática abaixo:

```
1  grammar Exemplo01;
2
3  begin:
4    expr
5  ;
6
7  expr:
8    INT_VAL op INT_VAL SEMI
9  | INT_VAL SEMI
10 ;
11
12 op:
13   EQ | LT | GT
14 ;
15
16 INT_VAL : [0-9]+ ;
17 LT      : '<'    ;
18 GT      : '>'    ;
19 EQ      : '='    ;
20 SEMI    : ';'    ;
21 WS      : [ \t\n]+ -> skip ;
```

Os *tokens* reconhecidos pelo *lexer* estão definidos nas linhas 16-21. Não há nada de novo aqui em relação ao que já foi visto no Laboratório 01, com a exceção de que agora estamos definido o *parser* junto no mesmo arquivo. Não existe uma ordem predeterminada para a localização das regras do *lexer* e *parser* no arquivo, mas vamos sempre deixar as regras sintáticas primeiro e as léxicas depois.

As linhas 3-14 indicam as regras da gramática livre de contexto que define a sintaxe da linguagem de entrada. Esta linguagem só permite uma expressão matemática por vez, sendo que esta deve finalizar com um `;`, como mostram as regras das linhas 8 e 9.

Compilando e executando:

```
$ make
$ make run
2 > 4;
```

Quando a análise sintática é concluída com sucesso não há retorno no terminal.

Agora um teste com erro:

```
$ make run
2 > 4 < 3
line 1:6 mismatched input '<' expecting ';'
```

Observe que o *parser* estava esperando um `;` para finalizar a regra de comparação mas digitamos um outro comparador, o que ocasiona um erro de análise sintática. Nestes casos o ANTLR gera uma mensagem de erro informativa automaticamente.

Eis outro exemplo de erro:

```
$ make run
2 > 4
line 2:0 missing ';' at '<EOF>'
```

E agora um código que parece correto:

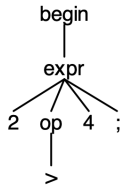
```
$ make run
2 > 4; 2 < 3;
```

Nesse caso o ANTLR não retornou nada, por conta disso poderíamos acreditar que a análise sintática foi concluída com sucesso, dado que as duas regras de comparação são seguidas de seus respectivos finalizadores. Porém se observarmos direito nosso *parser*, vamos perceber que ele só aceita uma única regra de comparação! Mas como podemos detectar esse erro com a saída do ANTLR? Para ajudar nesse caso, o ANTLR possui duas opções adicionais: a `-tree`, que imprime no terminal a *parse tree*; e a `-gui`, que cria um representação gráfica da árvore em uma janela.

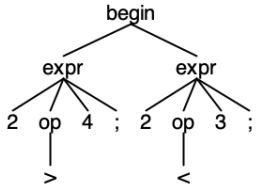
Usando essas opções no exemplo acima, temos os resultados a seguir. (Os exemplos abaixo mostram os comandos do ANTLR sendo executados diretamente para que o uso das opções extras fiquem evidentes, mas lembre-se que você deve editar o *Makefile* na linha que especifica o alvo `make run` para incluir a opção que você quer.)

```
$ grun Exemplo01 begin -tree
2 > 4; 2 < 3;
(begin (expr 2 (op >) 4 ;))
```

Com a saída do teste acima é possível ver que só foi analisada uma única expressão. Com a opção `-gui` no lugar da `-tree` a imagem da árvore fica como na figura abaixo. Observe que a árvore gerada é a mesma nos dois casos, com a única diferença sendo a representação.



Para reconhecer toda a entrada, vamos usar uma opção da notação EBNF. Basta modificar a regra inicial para `begin: expr+ ;`. Agora, compilando de novo e rodando mais uma vez, a saída da árvore gerada fica:



3.2 Exemplo 02 - Prioridade entre regras

Vamos modificar o exemplo anterior para torná-lo mais interessante. Queremos agora misturar operadores de comparação com o operador de soma para poder fazer a análise sintática de expressões como `2 + 3 = 3 + 2`. Em um caso assim, sabemos imediatamente que é necessário calcular os dois lados da igualdade antes de fazer a comparação. Em outras palavras, temos que o operador de soma tem prioridade (precedência) maior do que os de comparação.

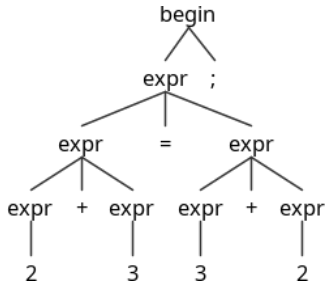
Para poder começar a descrever as regras, devemos aprender como o ANTLR resolve a prioridade das regras de acordo com as suas ocorrências. A precedência de uma regra é inversa ao número da linha de declaração. Isto é, as regras com maior precedência devem ser declaradas primeiro. (Note que no `bison` é o contrário: operadores que são declarados mais para baixo têm prioridade maior!) Se dois ou mais operadores tiverem a mesma precedência, eles devem ser declarados na mesma linha.

A gramática deste exemplo fica como abaixo.

```
1  grammar Exemplo02;
2
3  begin:
4    (expr SEMI)+
5  ;
6
7  expr:
8    expr PLUS expr
9  | expr (LT | GT | EQ) expr
10 | INT_VAL
11 ;
12
13 INT_VAL : [0-9]+ ;
14 PLUS    : '+'    ;
15 LT      : '<'    ;
```

```
16 GT       : '>'    ;
17 EQ       : '='    ;
18 SEMI     : ';'    ;
19 WS       : [ \t\n]+ -> skip ;
```

Vamos agora destacar as alterações. No *lexer* bastou incluir um novo tipo de *token* para o operador de soma. Já no *parser*, alteramos a linha 4, para não precisar ficar declarando o separador `;` em todas as regras de `expr`. Além disso, esta regra `expr` ficou recursiva para permitir a criação de sub-expressões. Olhando as linhas 8 e 9, vemos que o operador de soma ficou com maior prioridade que os operadores de comparação, e que estes todos possuem a mesma precedência. Executando o *parser* para o caso de teste que queríamos analisar, o resultado fica como na figura abaixo.



Podemos perceber imediatamente que a precedência dos operadores está correta.

3.3 Exemplo 03 - Um *parser* que reconhece comandos de `switch-case`

Neste exemplo vamos construir um *parser* para reconhecer uma versão simplificada do comando de `switch-case`, comum em muitas linguagens. Para ilustrar com um exemplo, queremos tratar entradas como abaixo.

```
1 switch id {
2     case 1:
3         break;
4     case 2:
5         break;
6     default:
7 }
```

É claro que cada `case` precisaria de mais comandos, mas no momento estamos interessados somente em reconhecer a estrutura básica de um `switch`, aonde temos um ou mais `cases` terminados por um `break`, e por fim uma entrada `default` que é opcional.

Utilizando as opções estendidas de BNF providas pelo ANTLR, a gramática fica simples de ser criada, como mostra o código abaixo.

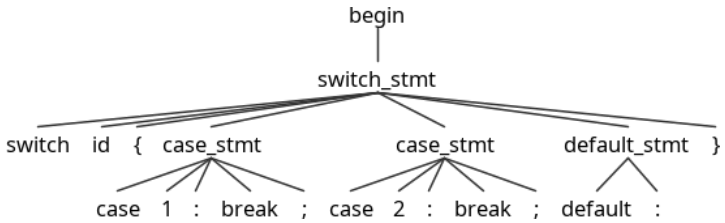
```
1 grammar Exemplo03;
2
3 begin:
4   switch_stmt
5 ;
6 switch_stmt:
7   SWITCH ID LBRACE case_stmt+ default_stmt? RBRACE
8 ;
```

```
case_stmt:
  CASE INT_VAL COLON BREAK SEMI
;
default_stmt:
  DEFAULT COLON
;

SWITCH  : 'switch';
CASE    : 'case';
DEFAULT : 'default';
BREAK   : 'break';
COLON   : ':' ;
SEMI    : ';' ;
LBRACE  : '{' ;
RBRACE  : '}' ;
INT_VAL : [0-9]+ ;
ID      : [a-zA-Z]+ ;
WS      : [ \t\n]+ -> skip ;
```

Esse exemplo é útil para se observar as vantagens da utilização de EBNF. Temos o fecho positivo na regra `switch_stmt`, permitindo uma ou mais repetições de `case_stmt`. Além disso, temos o comando `?` para dizer que o `default_stmt` é opcional, isto é, pode acontecer zero ou uma vez. Agora pense como ficaria a gramática usando somente a notação BNF, como no `bison`.

Para a entrada indicada no início deste exemplo, a *parser tree* construída fica como abaixo:



3.4 Exercícios de aquecimento

0. Faça o *download* dos arquivos de exemplo. Compile-os e execute-os como explicado acima. Observe os arquivos gerados pelo ANTLR, abra-os e veja se reconhece funções e como ele organiza as estruturas de dados.
1. Implemente um *parser* para o reconhecimento de parênteses pareados segundo a gramática abaixo (apresentada nos *slides* da Aula 02).

$$E \rightarrow (E) \mid a$$

2. Modifique a gramática abaixo para reconhecer as quatro operações aritméticas básicas e expressões com parênteses, respeitando adequadamente a precedência dos operadores.

$$\begin{aligned} E &\rightarrow E\ O\ E \mid (E) \mid \text{num} \\ O &\rightarrow + \mid - \mid * \mid / \end{aligned}$$

3. Implemente um *parser* que reconhece comandos *if-then-else* segundo a gramática a seguir (apresentada nos *slides* da Aula 02).

statement → *if-stmt* | **other**
if-stmt → **if** (*exp*) *statement*
| **if** (*exp*) *statement* **else** *statement*
exp → **0** | **1**

O seu *parser* deve aceitar entradas como abaixo.

```
other
if (0) other
if (1) other
if (0) other else other
if (1) other else other
if (0) if (1) other else other
```

E rejeitar entradas como:

```
if (0) if (1) other if
```

4. Crie um *parser* que reconhece e realiza uma sequência de operações de soma indicadas pela expressão de entrada. Ao final, imprima o resultado da soma caso a entrada seja sintaticamente válida. Por exemplo:

```
$ make run <<< "2 + 3 + 42"
Result = 47
```

Dica: Use uma variável para um acumulador colocando o trecho de código abaixo no seu arquivo `.g`

```
@members {
    int acc = 0;
}
```

e utilize ações semânticas nas suas regras sintáticas para acumular os *tokens* dos números lidos.

Parte II
Construindo um *parser* para EZLang

4 Implementado um *parser* para a linguagem EZLang

Nesse laboratório você terá que construir um *parser* para EZLang, a linguagem apresentada no laboratório passado. Para isso, utilize a gramática da linguagem em notação EBNF como abaixo, onde os *tokens* são escritos em CAIXA ALTA, e as regras de reescrita aparecem em caixa baixa:

```
program -> PROGRAM ID SEMI vars-sect stmt-sect
vars-sect -> VAR var-decl*
var-decl -> type-spec ID SEMI
type-spec -> BOOL | INT | REAL | STRING
stmt-sect -> BEGIN stmt+ END
```

```
stmt -> assign-stmt | if-stmt | read-stmt | repeat-stmt | write-stmt
assign-stmt -> ID ASSIGN expr SEMI
if-stmt -> IF expr THEN stmt+ (ELSE stmt+)? END
read-stmt -> READ ID SEMI
repeat-stmt -> REPEAT stmt+ UNTIL expr
write-stmt -> WRITE expr SEMI
expr -> expr op expr | LPAR expr RPAR | TRUE | FALSE |
        INT_VAL | REAL_VAL | STR_VAL | ID
op -> LT | EQ | PLUS | MINUS | TIMES | OVER
```

Lembrando que o ANTLR resolve precedência pela ordem de declaração das regras. Observe que nem todas as regras de reescrita da gramática acima estão com a precedência correta. Isto precisa ser consertado.

A partir da gramática você terá que escrever um *parser* que analisa arquivos de entrada escritos em EZLang e retorna a resposta padrão do ANTLR. Utilize os mesmos arquivos de teste do laboratório anterior (`in.zip`), com as saídas esperadas em `out02_java.zip`, ambos disponíveis no Classroom. Antes de começar a programar certifique-se que entendeu todas as regras da gramática.

4.1 Separando o *lexer* e o *parser* em arquivos distintos

Em todos os exemplos visto neste roteiro, um único arquivo `.g` do ANTLR continha a especificação tanto do *lexer* quanto do *parser*. Embora isto tenha sido feito por motivos didáticos, para simplificar os exemplos, na prática é bastante comum que o *lexer* e o *parser* de uma linguagem de entrada sejam especificados em arquivos distintos. Vamos agora indicar as alterações necessárias para realizar esta separação.

Quando usamos dois arquivos `.g`, é comum, para facilitar a organização, que ambos tenham um mesmo prefixo no nome. Para esta tarefa, vamos usar `EZ` como este prefixo. Com isso, os dois arquivos de entrada do ANTLR ficam nomeados como `EZLexer.g` e `EZParser.g`. O arquivo do *lexer* é exatamente o mesmo que foi disponibilizado na solução do Laboratório 01, e você pode continuar utilizando-o para desenvolver esta tarefa. Já o arquivo `EZParser.g` deve começar com os seguintes comandos:

```
1 parser grammar EZParser;
2 options {
3     tokenVocab = EZLexer;
4 }
```

A linha 1 do código acima indica que o arquivo contém somente a gramática do analisador sintático. Sem esta informação o ANTLR espera as regras da análise léxica no mesmo arquivo. Já a opção da linha 3 serve para fazer a conexão do *lexer* com o *parser*, indicando em qual arquivo estão as definições dos *tokens*.

Para compilar as gramáticas, basta informar ambos os arquivos ao chamar o ANTLR:

```
$ antlr4 -no-listener -o parser EZLexer.g EZParser.g
```

Na hora de executar o programa gerado, devemos lembrar que o nome da gramática é o prefixo de ambos os arquivos (`EZ`). Além disso, observe que a regra inicial da gramática EBNF acima é `program`. Então, na hora de executar o `grun` o comando fica:

```
$ grun EZ program
```

A *parse tree* gerada com a opção `-gui` é bem útil para auxiliar na inspeção visual do que o *parser* está reconhecendo com as regras da nossa linguagem. Para testar a saída do seu *parser* contra a fornecida pelo professor, modifique o *script* do laboratório anterior.

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.