INSTITUTO MAUÁ DE TECNOLOGIA

MAUÁ

ECM253 – Linguagens Formais, Autômatos e Compiladores

CUP – Gerador de analisadores sintáticos ascendentes

Marco Furlan

Setembro/2022

1 O que é CUP

CUP é o mesmo que **Constructor of Useful Parsers**. Trata-se de um programa que gera **analisadores sintáticos LALR Java** a partir de **especificações simples**. O site do projeto é: <http://www2.cs.tum.edu/projects/cup/>.

Um analisador sintático gerado pelo CUP **depende de um analisador léxico** capaz de tokenizar a entrada. Como o **CUP possui integração com o JFlex**, este último será utilizado como gerador do analisador léxico nos projetos (mas podem ser utilizados outros, inclusive, manuais).

Para **baixar o CUP**, **visitar a página** <http://www2.cs.tum.edu/projects/cup/> e então escolher uma versão de arquivo JAR para baixar (neste tutorial foi `java-cup-bin-11b-20160615.tar.gz`).

2 Especificação do arquivo de entrada

O **arquivo de entrada CUP** com a especificação do analisador sintático possui as seguintes **seções**:

- **Especificação do pacote e de importações**. É uma **seção opcional**. Aqui se define **pacote** que **conterá o código do analisador a ser gerado** e as **importações necessárias**. Define-se, também o nome da classe do analisador.
- **Componentes do código-fonte do usuário**. É uma **seção opcional**. Está dividido nas seguintes partes:
  - **`action code {: ...`*`código Java`*`...:}`**. É **opcional**. Insere-se aqui **código Java** a ser **adicionado** em uma **classe privada** que poderá ser utilizado no código embutido da gramática que será gerada. Por exemplo, poderia ser declarações e rotinas de manipulação da tabela de símbolos.
  - **`parser code {: ...`*`código Java`*`...:}`**. É opcional. Insere-se aqui **métodos e variáveis** que podem ser adicionados à classe do analisador que será gerada.
  - **`init with {: ...`*`código Java`*`...:}`**. É **opcional**. Pode-se fazer aqui a inicialização pré-análise léxica. Por exemplo, inicializar o analisador léxico.
  - **`scan with {: ...`*`código Java`*`:}`**. É opcional. Trata-se de código a ser embutido no corpo do analisador sintático que especificará como obter o próximo *token*.
- **Lista de símbolos**. É onde se **definem os símbolos da gramática** – **terminais** e **não terminais**. Os símbolos terminais e não terminais podem ser associados à classes Java. **Exemplo**:

```
// Definição dos terminais
terminal            ERROR, SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal            UMINUS, LPAREN, RPAREN;
terminal Integer    NUMBER;
// Definição dos não-terminais
non terminal            expr_list, expr_part;
non terminal Integer    expr;
```

- **Declarações de precedência e associatividade**. É opcional. Especifica a **precedência e associatividade** dos terminais. **Exemplo**:

```
// Precedência e associatividade dos oper.adores
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;
```

  A precedência cresce de cima para baixo nas declarações.

- **Gramática**. São regras na forma `X ::= α;`, **seguidas ou não** de **ações escritas em blocos** `{: ...código Java ...:}`. **Exemplo de uma regra**:

```
expr ::= expr:e1 PLUS expr:e2 {: RESULT = new Integer(e1 + e2); :}
```

3 Preparação do ambiente de desenvolvimento

Nos **projetos com CUP**, será utilizado o **Visual Studio Code**, de modo **similar** como o que foi feito nos projetos como JFlex. Então, as seguintes **extensões** (no laboratório, sempre reinstalar logo após executar o Visual Studio Code) serão necessárias:

- **Extension Pack for Java**
- **JFlex**
- **CUP**
- **Ant Target Runner**

4 Estrutura para projeto com CUP e JFlex

Propõe-se a **estrutura** a seguir para **projetos com CUP**:

```
.
├── build.xml                 # arquivo de construção para o Ant
├── src                       # pasta dos arquivos fontes
│   ├── cup                   # pasta para o arquivo fonte do CUP
│   │   └── Parser.cup        # arquivo de especificação do analisador sintático
│   ├── jflex                 # pasta para o arquivo fonte do JFlex
│   │   └── Scanner.jflex     # arquivo de especificação do analisador léxico
│   └── parser                # pacote dos arquivos fonte Java (inclui os gerados)
│       └──  Main.java        # programa principal do interpretador/compilador
├── teste.input               # aquivo texto com testes para o analisador sintático
├── tools                     # pasta para armazenar programas e bilbiotecas auxiliares
│   ├── java-cup-11b.jar      # JAR contendo programa do CUP e bibliotecas necessárias
│   └── jflex-full-1.8.2.jar  # JAR contendo o programa JFlex
└── .vscode                   # pasta oculta de configurações do projeto VSCode
    └── settings.json         # arquivo de configurações do VSCode - path de dependências
```

O **arquivo Ant** de construção agora **além** de **invocar** o **JFlex** e construir o **analisador léxico**, invocará também o **CUP** e construirá o **analisador sintático**. O conteúdo do arquivo `build.xml` para a estrutura de projeto proposta está apresentado a seguir. Notar que o pacote JAR do CUP também tem uma `taskdef` para facilitar sua invocação pelo Ant:

```
<project name="SimpleExpr"
         default="dist" basedir=".">
    <description>
        Arquivo Ant para construir um analisador sintático simples.
    </description>
    <!-- propriedades globais -->
    <property name="mainClass" value="parser.Main" />
    <property name="src" location="src" />
    <property name="tools" location="tools" />
    <property name="jflex" location="src/jflex" />
    <property name="cup" location="src/cup" />
    <property name="build" location="build" />
    <property name="dist" location="dist" />
    <!-- Tarefa para chamar a ferramenta JFlex -->
    <taskdef name="jflex" classname="jflex.anttask.JFlexTask"
             classpath="${tools}/jflex-full-1.8.2.jar" />
    <!-- Tarefa para chamar a ferramenta CUP -->
    <taskdef name="cup" classname="java_cup.anttask.CUPTask"
             classpath="${tools}/java-cup-11b.jar" />
    <!-- Alvo: init -->
    <target name="init">
        <!-- Criar um diretório para armazenar as classes compiladas -->
        <mkdir dir="${build}" />
    </target>
    <!-- Alvo: compile -->
    <target name="compile" depends="init" description="compila os fontes">
        <!-- Executa o JFlex -->
        <jflex file="${jflex}/Scanner.jflex"
               destdir="${src}" />
        <!-- Executa o CUP -->
        <cup srcfile="${cup}/Parser.cup" destdir="${src}" parser="Parser"
             interface="true" locations="false" />
        <!-- Compila todos os códigos Java -->
        <javac srcdir="${src}" destdir="${build}"
               classpath="${tools}/java-cup-11b.jar" debug="true"/>
    </target>
    <!-- Alvo dist - cria o produto final, que pode ser redistribuído -->
    <target name="dist" depends="compile" description="cria uma distribuição">
        <!-- Cria o diretório de distribuição -->
        <mkdir dir="${dist}" />
        <!-- Empacota o programa em um arquivo JAR -->
        <jar jarfile="${dist}/simple_expr.jar"
             basedir="${build}">
            <manifest>
                <attribute name="Main-Class" value="${mainClass}" />
            </manifest>
            <!-- É necessário adicionar as bibliotecas do CUP!!! -->
            <zipgroupfileset dir="${tools}" includes="java-cup-11b.jar" />
        </jar>
    </target>
    <!-- Alvo jar - o mesmo que dist -->
    <target name="jar" depends="dist" description="cria uma distribuição">
    </target>
```

```
    <!-- Alvo run - executa o código com arquivo de teste -->
    <target name="run" depends="dist" description="executa e testa o projeto">
        <java classname="${mainClass}" classpath="${dist}/simple_expr.jar">
            <arg value="teste.input"/>
        </java>
    </target>
    <!-- Alvo clean - limpa os arquivos gerados -->
    <target name="clean" description="limpar arquivos gerados">
        <delete dir="${build}" />
        <delete dir="${dist}" />
        <delete>
            <fileset dir="${src}/parser">
                <include name="Scanner.java" />
                <include name="Scanner.java~" />
                <include name="Parser.java" />
                <include name="sym.java" />
            </fileset>
        </delete>
    </target>
</project>
```

Algumas novidades:

- A tarefa do CUP cria um analisador sintático na classe `Parser`:

```
        <!-- Executa o CUP -->
        <cup srcfile="${cup}/Parser.cup" destdir="${src}" parser="Parser"
             interface="true" locations="false" />
```

- No JAR a ser gerado, é necessário embutir o JAR do CUP, por causa da dependência de suas classes:

```
        <!-- Empacota o programa em um arquivo JAR -->
        <jar jarfile="${dist}/simple_expr.jar"
             basedir="${build}">
            <manifest>
                <attribute name="Main-Class" value="${mainClass}" />
            </manifest>
            <!-- É necessário adicionar as bibliotecas do CUP!!! -->
            <zipgroupfileset dir="${tools}" includes="java-cup-11b.jar" />
        </jar>
```

5 Projeto desenvolvido

O objetivo deste projeto é criar um **interpretador de expressões aritméticas simples** que, além de **executar a análise** sintática sobre um **conjunto de expressões**, também irá **interpretar essas expressões**, gerando resultados numéricos na tela.

Para tanto será utilizada a **característica do CUP** de permitir **associar ações Java** a **cada regra reconhecida** durante a análise – técnica conhecida por **tradução dirigida por sintaxe**, que permite realizar **ações semânticas** durante a **execução da análise sintática**.

5.1 Gramática de expressões

Será utilizado como base a gramática de **expressões simples**, apresentada a seguir (não é necessário se preocupar com recursões à esquerda quando se utiliza o CUP):

```
expr_list ::= expr_list expr_part | expr_part
expr_part ::= expr ';'
expr      ::= expr '+' expr | expr '-' expr | expr '*' expr
            | expr '/' expr | expr '%' expr | '(' expr ')'
            | '-' expr | number
```

5.2 Analisador léxico

O arquivo de **especificação do analisador léxico** para o JFlex (`Scanner.jflex`) está apresentado a seguir:

```
// Arquivo para o scanner a ser utilizado
package parser;
// Importar classes do cup - classe Symbol
import java_cup.runtime.*;

%%

%class Scanner
%cup
%unicode
%line
%column

%{
    // type é a classe do token
    // yyline e yycolumn são variáveis reservadas
    // do JFlex para armazenar a linha e a coluna
    // de um token encontrado na entrada (precisa
    //  usar %line e %column)
    private Symbol symbol(int type) {
        return new Symbol(type, yyline, yycolumn);
    }
    private Symbol symbol(int type, Object value) {
        return new Symbol(type, yyline, yycolumn, value);
    }
%}

ws = [\ \t\f\r\n]
number = [0-9]+
%%
";"         { return symbol(sym.SEMI); }
"+"         { return symbol(sym.PLUS); }
"-"         { return symbol(sym.MINUS); }
"*"         { return symbol(sym.TIMES); }
"/"         { return symbol(sym.DIVIDE); }
"%"         { return symbol(sym.MOD); }
"("         { return symbol(sym.LPAREN); }
")"         { return symbol(sym.RPAREN); }
{number}    { return symbol(sym.NUMBER, Integer.valueOf(yytext())); }
{ws}        {/* Ignore */}
```

```
.           { return symbol(sym.ERROR, yytext()); }
```

As **novidades** aqui são:

- **Importação do pacote do CUP** para utilizar a classe `Symbol` de *token* providenciada pelo CUP (`import java_cup.runtime.*`). Não é necessário, mas é uma comodidade;
- Uso da opção `%cup`, no JFlex, para permitir a integração com o CUP;
- Adição na classe `Scanner` a ser gerada de dois **métodos sobrecarregados** de nome `symbol()`, para retornar *tokens* **simples** e com atributos a partir da classe `Symbol` provida pelo CUP;
- **Notar** as **constantes** `sym.SEMI` e outras. O próprio **CUP se encarregará** de **gerar uma classe denominada** `sym.java` com a definição dessas constantes. Essa classe será armazenada no pacote `parser`.

5.3 Analisador sintático

O arquivo de especificação CUP para o projeto está apresentado a seguir:

```
// Especificação do parser CUP
package parser;
import java_cup.runtime.*;
// Definição dos terminais
terminal            ERROR, SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal            UMINUS, LPAREN, RPAREN;
terminal Integer    NUMBER;
// Definição dos não-terminais
non terminal            expr_list, expr_part;
non terminal Integer    expr;
// Precedência e associatividade dos oper.adores
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;
// Regras da gramática
expr_list ::= expr_list expr_part
    | expr_part
    ;
expr_part ::= expr:e {:
        System.out.println("Resultado: " + e); :} SEMI
    | error SEMI
    ;
expr ::= expr:e1 PLUS expr:e2
            {: RESULT = e1 + e2; :}
    | expr:e1 MINUS expr:e2
            {: RESULT = e1 - e2; :}
    | expr:e1 TIMES expr:e2
            {: RESULT = e1 * e2; :}
    | expr:e1 DIVIDE expr:e2
            {: RESULT = e1 / e2; :}
    | expr:e1 MOD expr:e2
            {: RESULT = e1 % e2; :}
```

```
    | NUMBER:n {: RESULT = n; :}
    | MINUS expr:e {: RESULT = -e;
                    :} %prec UMINUS
    | LPAREN expr:e RPAREN {: RESULT = e; :}
    ;
```

É importante entender a **estrutura do analisador**:

- No início do arquivo, tem-se a **inclusão do pacote do CUP** e também a definição do **pacote** onde a **classe** `Parser` será gerada:

```
package parser;
import java_cup.runtime.*;
```

- Na **definição dos terminais**, utilizam-se os **símbolos retornados** pelo **analisador léxico**. Para **símbolos simples** (exemplo: `SEMI`), nãoé necessário definir um tipo de classe, mas para **símbolos que foram associados a uma classe** é importante, pois será possível realizar operações semânticas com eles (exemplo: `NUMBER`);

- O nome dos **símbolos não terminais** foram tirados da **definição da gramática** apresentada na seção 5.1;

- Podem existir **não terminais** sem associação a um tipo classe, assim como não terminais que **tenham tipo classe**. O **não terminal** `expr` foi declarado com tipo de classe `Integer`, pois **quando ele for reconhecido**, o valor carregado por ele **será o resultado numérico de uma expressão**;

- Na definição de **precedência**, todos os operadores são classificados. A precedência aumenta de cima para baixo. Além da precedência, define-se também a associatividade: todos os operadores possuem associatividade à esquerda;

- Para **diferenciar o operador binário** "menos" do **operador de negação unário**, foi criado um simbolo denominado `UMINUS`, que possui a maior precedência de todos – ele será utilizado depois na regra que reconhece uma expressão negada;

- A **sintaxe das regras gramaticais** é quase idêntica a das notações mais comuns sobre gramáticas. No entanto, toda regra tem que terminar por ponto e vírgula e pode, opcionalmente, ter uma **ação associada**, para permitir a execução de código durante a análise sintática. **Exemplo** de uma regra que não possui ação associada, pois ela apenas serve para permitir a repetição de expressões:

```
expr_list ::= expr_list expr_part
    | expr_part
    ;
```

Já a regra a seguir, que resulta uma expressão, possui uma ação definida:

```
expr_part ::= expr:e {:
        System.out.println("Resultado: " + e); :} SEMI
    | error SEMI
    ;
```

Notar a **simbologia** `expr:e`: `expr` é um **não terminal** da gramática que se utiliza aqui pela própria definição da gramática, mas o **símbolo** `e` é um **símbolo definido pelo programador**

apenas para **capturar** o **valor numérico** da expressão, quando uma expressão for corretamente reconhecida. Assim que isso ocorrer, a ação proposta é exibir esse valor.

Mas, de onde vem o valor da expressão? O reconhecedor gerado é do tipo LALR, ascendente. Então, o valor da expressão (é um inteiro) é calculado nas regras inferiores e carregado para cima, como, por exemplo, na regra que soma duas expressões:

```
expr ::= expr:e1 PLUS expr:e2
            {: RESULT = e1 + e2; :}
```

Observar que, do lado direito, as expressões possuem dois símbolos associados, `e1` e `e2` que, por sua vez, carregam os valores de expressões reconhecidas na soma. Basta então obter os valor numérico delas e somá-las. Porém, para que o resultado da soma seja propagado de "baixo para cima", utiliza-se um **símbolo especial do CUP** denominado `RESULT`, que **automaticamente** armazenará o resultado do lado direito em algum atributo da cabeça da regra do lado esquerdo.

Por fim, a fonte dos valores das expressões vem da regra:

```
expr ::= NUMBER:n {: RESULT = n; :}
```

Então é importante que os tipos da expressão e do número utilizado sejam os mesmos.

6 Testes

O arquivo de testes fornecido possui o seguinte conteúdo (testar com outras expressões):

```
11;
5+-3;
3+4;
7-6 % 4 - 2;
(2+4) * 6 / 10;
```

A cada expressão correta, separada por ponto e vírgula, um resultado será exibido:

```
Resultado: 11
Resultado: 2
Resultado: 7
Resultado: 3
Resultado: 3
```