===== **Turma 2 Grupo 1 - D3FDG2Dot** =====

Diogo Miguel Sousa Barroso - ei11105@fe.up.pt
João Pedro Milano da Silva Cardoso - ee09063@fe.up.pt
Kevin Carvalho de Amorim - ei12057@fe.up.pt
Luís Miguel Coelho e Magalhães - ei12054@fe.up.pt

======= **PLUGINS / FICHEIROS EXTERIORES / INSTALAÇÃO** =======

Desenvolvido em Windows 7 / 8.1
IDE: Intellij 14 Ultimate Edition com Plugin ANTLR v1.6
SDK: Java 1.8
Graphviz 2.38 está disponível em http://www.graphviz.org/Download_windows.php
Interface desenvolvido com JavaFX

Instalação IDE:

- IntelliJ 14 Ultimate Edition: https://www.jetbrains.com/idea/download/

Instalação de Plugin:

- ANTLR4 Intellij Plugin v1.6: https://plugins.jetbrains.com/plugin/7358?pr=
- No ecrã inicial “Welcome to IntelliJ IDEA”:
  Configure -> Plugin -> Install Plugin from Disk -> Selecionar o ficheiro rar.

No caso do projecto ser aberto e o plugin não tiver sido instalado, dever aparecer no canto superior direito uma mensagem com um aviso e um link que permite a sua instalação.

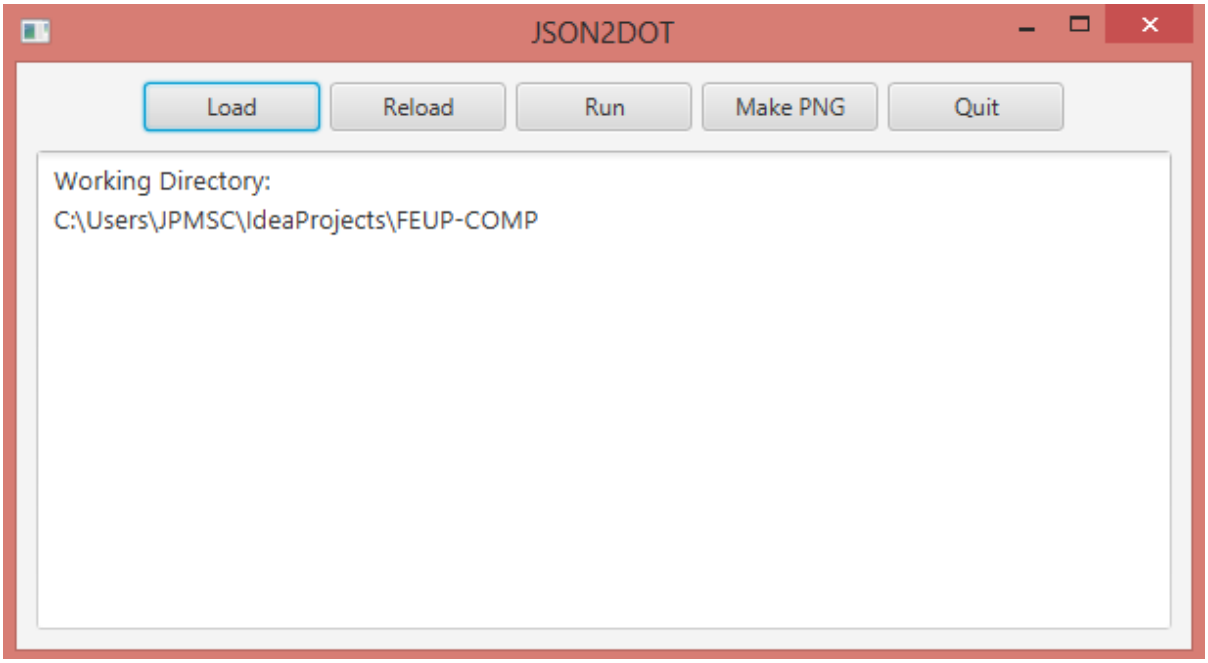
Abrir o projecto:

- No ecrã inicial “Welcome to IntelliJ IDEA”:
  Open -> Selecionar pasta do Projeto (deverá apresentar o símbolo IntelliJ)

Compilação:

- Se for alterado o ficheiro G4, fazer CTRL + SHIFT + G para gerar o parser. O código gerado encontra-se na pasta gen.
- Para compilar, Build -> Make Project (CTRL + F9)
- Para correr a partir do IDE, Run -> Run ‘JSON’ (SHIFT+F10)
- Para compilar o JAR Build -> Build Artifacts -> Build

======= **HOW TO** =======



Load - Permite carregar um ficheiro de texto.

Reload - Permite recarregar o ficheiro mais recente.

Run - Corre as várias fases do projecto: Análise Lexical, Sintáctica, Semântica, Geração de Código.

Make PNG - Recorrendo a Graphviz cria um ficheiro PNG a partir do ficheiro de linguagem DOT e abre a imagem utilizando o programa default.

Quit - Sair da aplicação.

É também possível correr a aplicação a partir de uma consola. O ficheiro JAR para o efeito encontra-se em FEUP-COMP/out/artifacts/FEUP_COMP_jar.

java -jar FEUP-COMP.jar <inputFileName>
e.g:
java -jar FEUP-COMP.jar correct.txt

Correr o JAR sem argumentos invoca o interface.

O nome do ficheiro convertido é automaticamente definido como correct2DOT.txt.

Tanto o interface gráfico como o JAR lêem e escrevem para pasta examples.

- O interface em FEUP-COMP/examples.
- O JAR em FEUP-COMP/out/artifacts/FEUP_COMP_jar/examples.

======= **EXEMPLOS** =======

Exemplo 1 - correct.txt

Ficheiro sintática e semânticamente correto. Não deverão aparecer erros ou avisos.

Exemplo 2 - miserables.txt

Ficheiro original proveniente do enunciado do trabalho. Não deverão aparecer erros ou avisos.

Exemplo 3 - parser_errors.txt

Ficheiro com erros sintáticos. Não deverá ser possível a geração de código.

Para referência, os erros são:

Linha 4 - Falta " depois de name
Linha 5 - Vírgula em vez de ponto em margin=0,5
Linha 7 - link em vez de links
Linha 8 - Falta vírgula a separar os atributos dir e arrowtail
Linha 9 - surce em vez de source
Linha 17 - Charactér inválido em "No.deC"
Linha 17 - Falta vírgula no fim da definição do link
Linha 21 - Um grupo definido desta maneira precisa de pelo menos um atributo

Exemplo 4 - semantic_errors.txt

Ficheiro com erros semânticos e avisos. Não deverá ser possível a geração de código.

Exemplo 5 - warnings.txt

Ficheiro com avisos. A geração de código é possível.

===== **ANÁLISE LEXICAL** =====

O Lexer é criado automaticamente pelo ANTLR e encontra-se em gen/JSONLexer.
Ao ocorrer um erro, este é logado e reportado ao utilizador.

===== **ANÁLISE SINTÁTICA** =====

O Parser é criado automaticamente pelo ANTLR e encontra-se em gen/JSONParser.
Ao ocorrer um erro, este é logado e reportado ao utilizador. O ANTLR providencia recuperação automática de erros, ao consumir tokens até conseguir encontrar uma regra e continuar a análise. Isto leva a que nem todos os erros sejam reportados ao mesmo tempo e que ao ser corrigido um, outros possam desaparecer.

===== **ANÁLISE SEMÂNTICA** =====

A análise semântica está dividida em duas partes: erros e avisos.

Erros: Classe SemanticAnalyser em src.
Avisos: Classe SemanticWarnings em src/SemanticAnalysis.

Ao ocorrer um erro ou aviso o mesmo é logado e reportado ao utilizador.

Validações Semânticas

Erros

- **Nós**
  - Nomes repetidos.
  - Existência dos atributos.
  - Valores válidos para os atributos.
- **Links**
  - Existência dos nós source e target.
  - Source e target não podem ser o mesmo.
  - Existência dos atributos.
  - Valores válidos para os atributos.
- **Grupos**
  - Existência de nós no grupo. Não podem existir grupos vazios.
  - Existência dos atributos
  - Se for definido um grupo sem atributos dentro de “groups”:[...], este será inválido. É necessário pelo menos um atributo.

Avisos

Atributos Repetidos - O formato Dot aceita que cada grupo, nó ou link tenha atributos repetidos, dos quais será utilizado o último.

Compatibilidade dos Atributos - O formato Dot aceita que cada grupo, nó ou link tenha atributos que possam entrar em conflito uns com os outros:

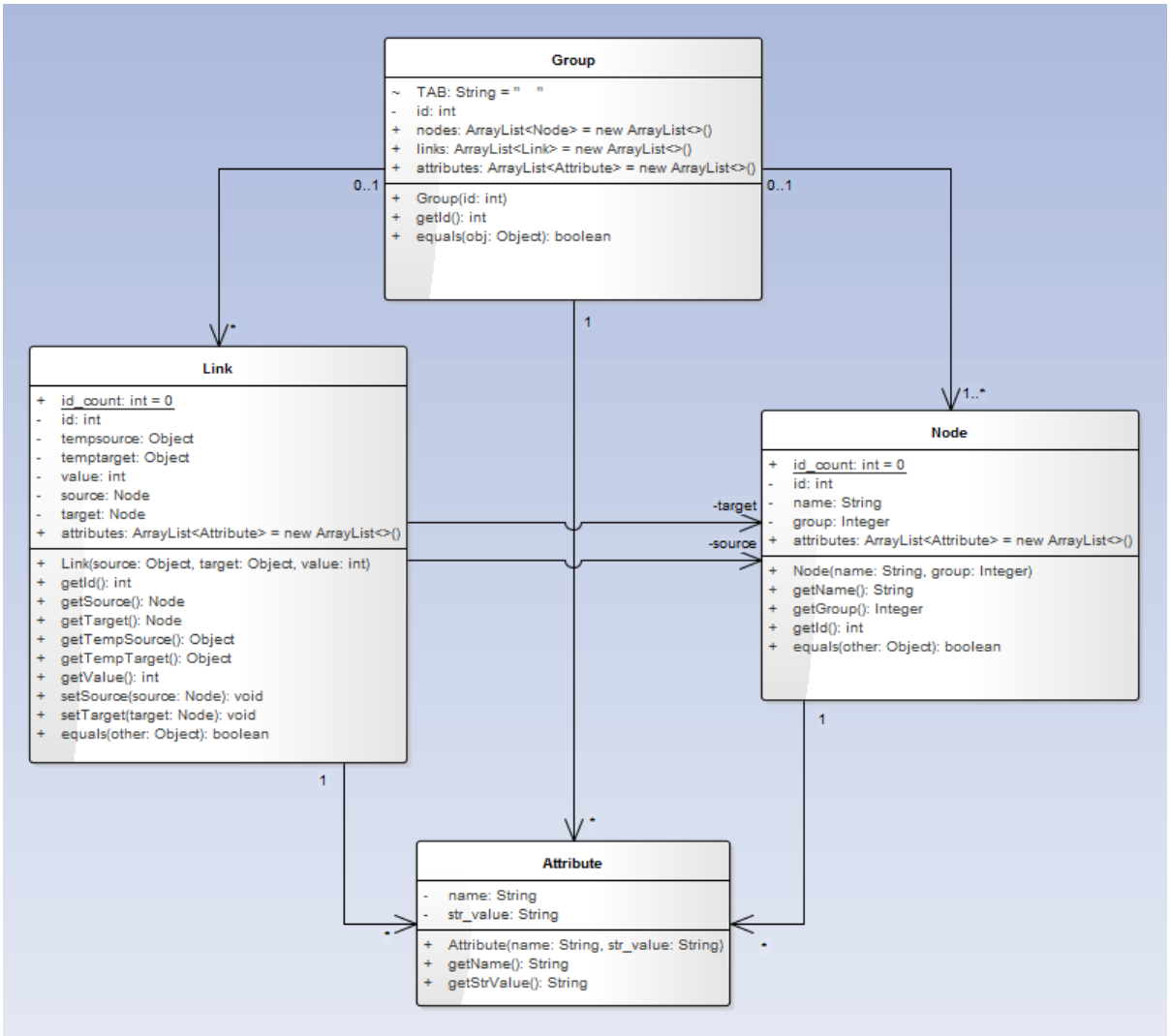
- Atributo Comum:
  - Style:
    - Se style = filled e nem fillcolor nem color estiverem presentes, será utilizador o valor por default.
    - Se style = filled e fillcolor não estive presente, será utilizado color.
    - Se fillcolor estiver presente mas style != filled, o atributo será inútil.

- Nós:
  - Shape:
    - Se shape = polygon é necessária a existência do atributo sides é também necessária. Sides precisa de ser igual ou maior a 3.
    - Se sides existir mas shape != polygon o atributo será inútil.

- Links:
  - Arrows:
    - Se existir arrowtail, dir deve ser both ou back para o efeito aparecer.
    - Se existir arrowhead, dir deve ser both ou front para o efeito aparecer.

===== **REPRESENTAÇÃO INTERMÉDIA** =====

A representação intermédia é feita através de quatro classes: Attribute, Group, Link e Node em src/InterRep.



É de notar que o valor de um link (value) cria para todos os links um atributo minlength, de modo a que o valor seja reflectido no grafo final.

===== **GERAÇÃO DE CÓDIGO** =====

A geração de código é feita simplesmente através de um PrintWriter e da conversão de cada classe da representação intermédia para string.

Independentemente do ficheiro de origem, todos os ficheiros Dot necessitam têm a estrutura:

```
digraph G{
        compound = true;
        ...
}
```

Digraph permite ter links direcionados e compound = true permite que links liguem nós de diferentes grupos. O nome do grafo (G) é irrelavante.

Cada grupo será um cluster, e.g:

```
subgraph cluster1{
    penwidth=5.0; color=red;
    NodeA[color=blue];
    NodeB[];
    NodeA -> NodeB[minlen=1.0, color=violet, dir=both, arrowtail=crow];
  }
```

Neste caso o grupo 1 será o cluster1 com 2 atributos, 2 nós e 1 link.

Os links que não pertencem a nenhum grupo (source e target pertencem a grupos diferentes) são escritos de seguida:

```
NodeB -> NodeC[minlen=2.0, arrowhead=vee, style=dashed];
NodeC -> NodeE[minlen=4.0, arrowsize=2, dir=both, arrowtail=crow];
NodeA -> NodeE[minlen=4.0];
```

O formato é o mesmo.

===== TESTES =====

Os teste efetuados recorreram aos ficheiros utilizados na secção de exemplos, os quais foram evoluindo à medida que a linguagem original foi sendo modificada.

===== ARQUITETURA =====

Não existe estruturação de alto nível do projecto mencionável.

===== PONTOS POSITIVOS =====

Adições à linguagem original:

- Links podem referenciar Nós por id ou por nome.
- Nós, Links e Grupos podem ter atributos.
- As secções de Nós, Links e Grupos podem ser definidas por qualquer ordem.

Extras:

- Interface.
- Utilizando Graphviz é possível a criação de um ficheiro PNG com o grafo descrito no ficheiro de texto criado para verificar o resultado.

===== PONTOS NEGATIVOS =====

O lexer e parser criados automaticamente pelo ANTLR4 têm os seus próprios métodos de recuperação de erros, constituindo uma “caixa negra” pelo que, para além da criação da Parse Tree, a nossa interação com eles limita-se a reportar os erros ao utilizador. É possível que uma melhoria na linguagem permita uma melhor análise por parte do lexer e parser.

Não existem bugs conhecidos.

O projecto escreve e lê da pasta FOLDER, definida em JSONRun, no directório de trabalho. Perde assim flexibilidade.

Melhorias:

A linguagem DOT contém dois níveis de complexidade: um mais simples, utilizado neste projecto e um mais complexo, onde as coordenadas dos nós e a curvatura dos links pode ser manualmente definidos.

Uma possível melhoria seria a aplicação de algoritmos de distribuição espacial dos nós tendo em conta os seus atributos.