Compilador de *EPLan*

José Romildo Malaquias

Departamento de Computação
Universidade Federal de Ouro Preto

2016.2

Tópicos

Organização do compilador

- Implementado na linguagem **Java**.
- Ferramentas auxiliares:
  - **JFlex**: gerador de analisador léxico
  - **CUP**: gerador de analisador sintático
  - **LLVM**: gerador de código
  - **Maven**: ferramenta de automação de compilação de projetos Java
- Usa bibliotecas externas:
  - **commons-lang3**: complementa as classes que estão em `java.lang`.
  - **jcommander**: framework Java muito pequeno que torna trivial a análise de parâmetros de linha de comando
  - **javacpp-presets-llvm**: interface para a biblioteca LLVMC do projeto LLVM (infraestrutura de construçào de compilador escrita em C++)
  - **javaslang**: uma biblioteca funcional para Java 8+ que fornece tipos de dados persistentes e estruturas de controle funcionais
  - **javaslang-render**: biblioteca de renderização para algumas estruturas de dados fornecidas por javaslang
  - **junit**: *framework* com suporte à criação de testes automatizados em Java
  - **assertj**: fornece um rico conjunto de afirmações, com mensagens de erro úteis, melhorando a legibilidade dos testes automatizados em Java

Maven

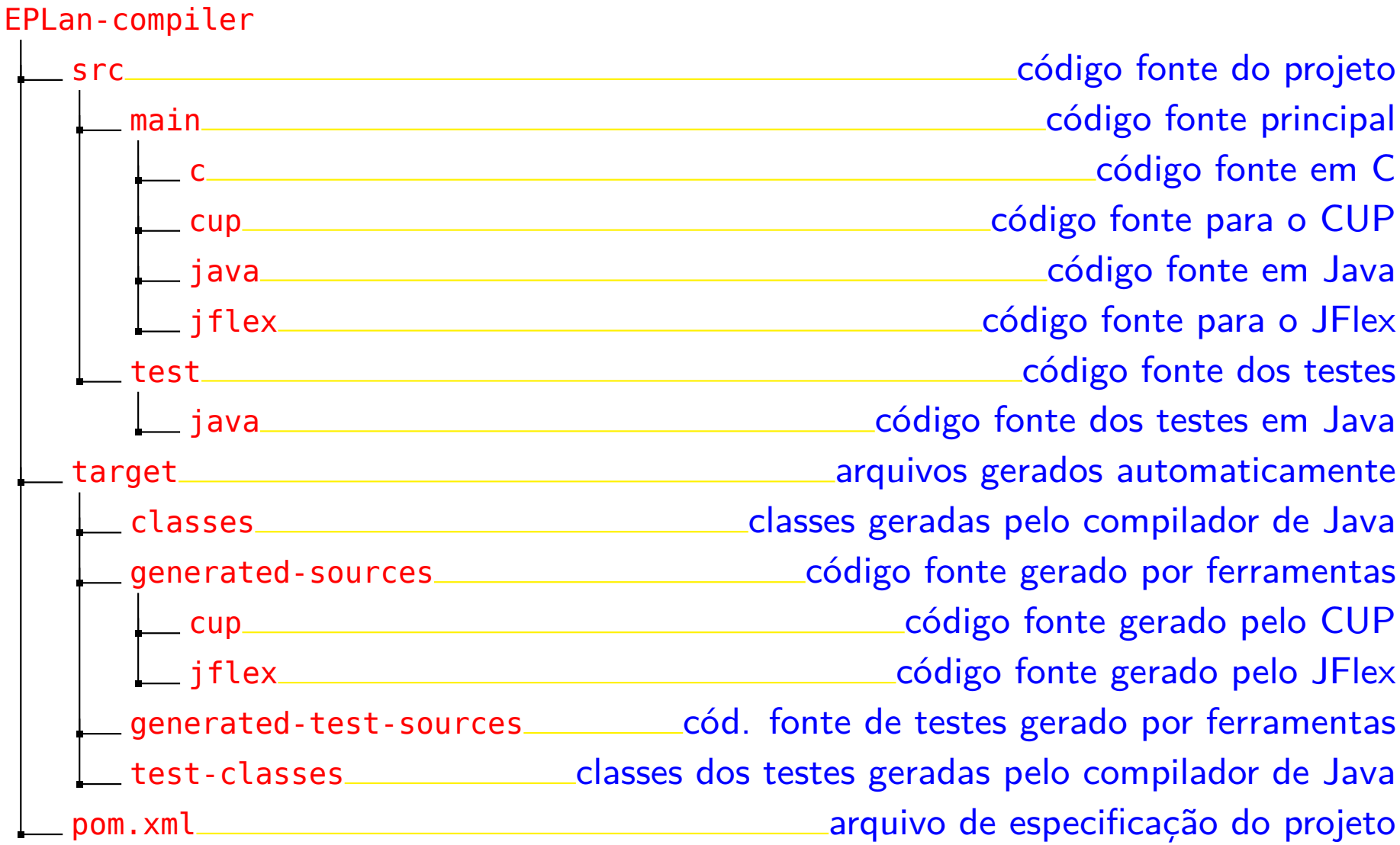
- Usado para automatizar a compilação de projetos.
- O projeto é configurado usando um **POM** (*Project Object Model*), que é armazenado em um arquivo `pom.xml`.
- O desenvolvimento pode ser feito em várias **fases**, indicadas por **objetivos**, como:

clean	remover arquivos gerados
generate-sources	gerar código automático
process-resources	processar recursos
compile	compilar
process-test-resources	processar recursos de teste
test-compile	testar compilação
test	testar
package	empacotar
install	instalar
deploy	implantar

- Exemplo: compilar o projeto na linha de comando:

```
$ mvn compile
```

Estrutura de diretórios do projeto



Tópicos

Javaslang

- Javaslang fornece várias estruturas de dados funcionais, como:
  - tuplas
  - listas
  - árvores
    - Árvores serão amplamente utilizadas para exibir as estruturas internas do compilador, incluindo as árvores sintáticas.

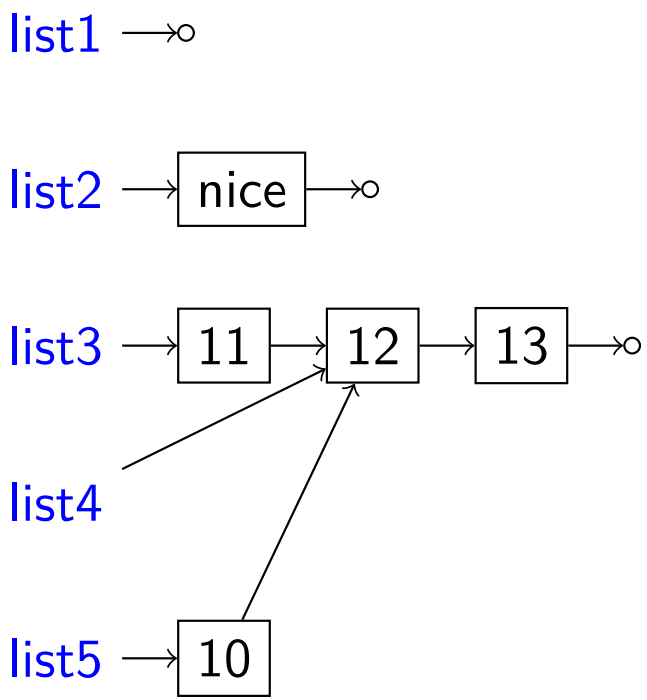
javaslang.Tuple2

```
import javaslang.Tuple;
import javaslang.Tuple2;

public class TestJavaslang {
  public static void main(String[] args) {
    Tuple2<String, Integer> person = Tuple.of("paul", 17);

    String name = person._1;
    Integer age = person._2;

    Tuple2<String, Integer> p =
            person.map(n -> n + " jones",
                       a -> a + 1);

    Tuple2<String, Integer> q =
            person.map((n, a) -> Tuple.of(n + " jones", a + 1));

    String s = person.transform((n, a) -> n + ": " + a);

    System.out.println(s);
  }
}
```

javaslang.collection.List

- Exemplos de listas (simplesmente) encadeadas:

```
List<Integer> list1 = List.empty();
List<String>  list2 = List.of("nice");
List<Integer> list3 = List.of(11, 12, 13);
List<Integer> list4 = list3.tail();
List<Integer> list5 = list4.prepend(10);
```



javaslang.collection.List (cont.)

- Operando com cada elemento de uma lista:

```
List<Integer> lst = List.of(10, 20, 30);

for (Integer x : lst)
  System.out.println(x);

lst.forEach(x -> System.out.println(x));

lst.forEach(System.out::println);
```

javaslang.collection.List (cont.)

- Aplicando uma função a cada elemento da lista e coletando os resultados em outra lista:

```
List<Double> a = List.of(4.0, 9.0, 25.0);
List<Double> c = a.map(Math::sqrt);
List<Double> b = a.map(x -> 2*x);
```

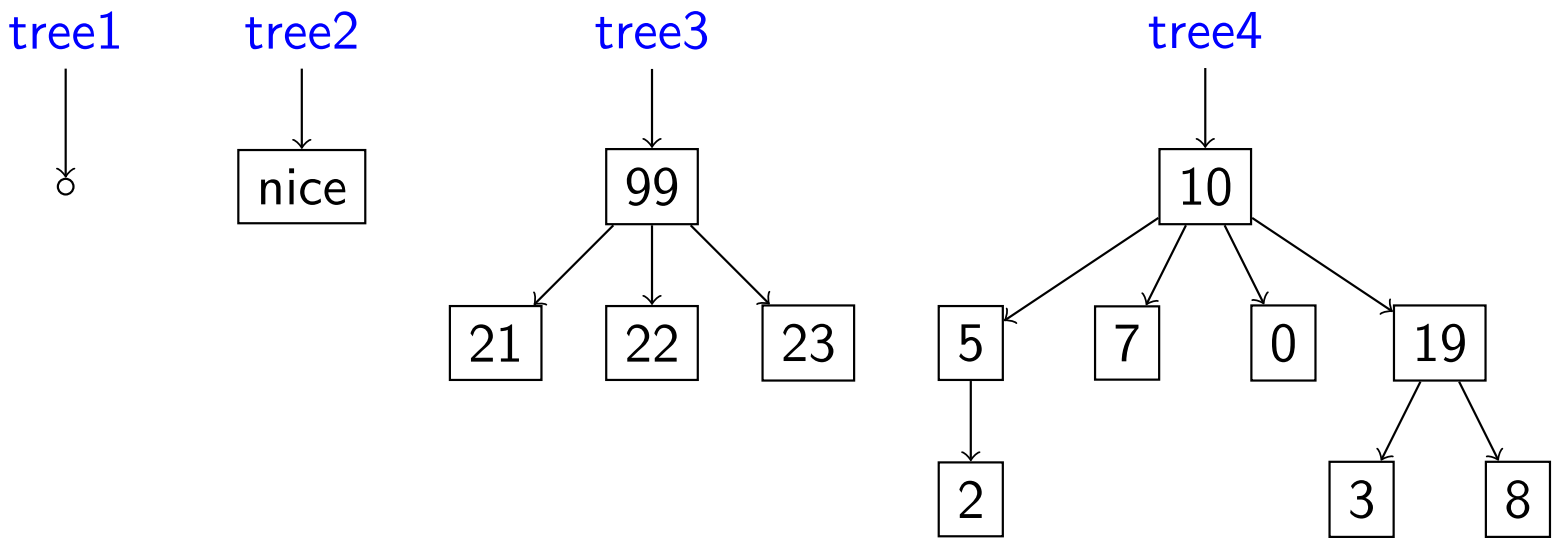
javaslang.collection.List (cont.)

- Reduzindo uma lista:

```
List<String> a = List.of("1", "2", "3");
String str = a.fold("", (a1, a2) -> a1 + a2);

List<Integer> b = List.of(1, 2, 3, 4);
Integer sum = b.fold(0, (s, x) -> s + x);
```

javaslang.collection.Tree

```
Tree<Integer> tree1 = Tree.empty();
Tree<String> tree2 = Tree.of("nice");
Tree<Integer> tree3 = Tree.of(99, 21, 22, 23);
Tree<Integer> tree4 = Tree.of(10,
                              Tree.of(5, Tree.of(2)),
                              Tree.of(7),
                              Tree.of(0),
                              Tree.of(19, Tree.of(3), Tree.of(8)));
```



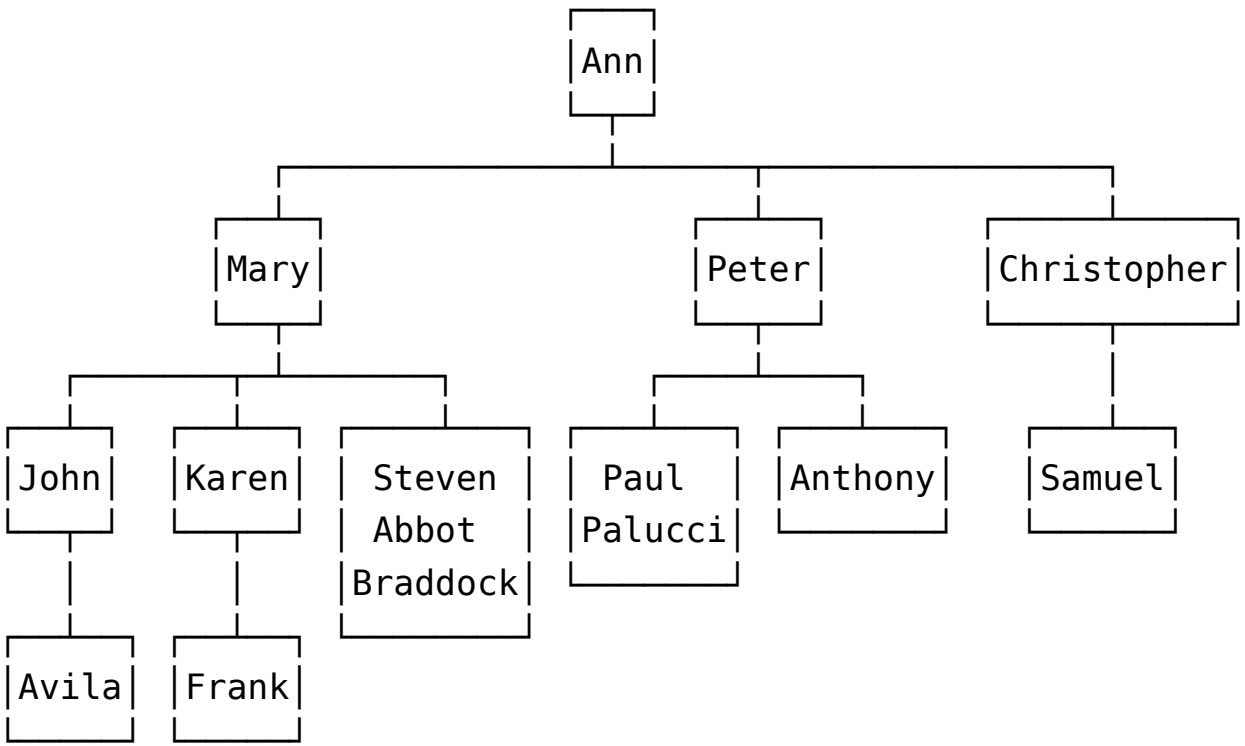
javaslang.render.text.PrettyPrinter

```
import javaslang.collection.Tree;

import javaslang.render.text.PrettyPrinter;

final Tree<String> tree =
       Tree.of("Ann",
               Tree.of("Mary",
                       Tree.of("John",
                               Tree.of("Avila")),
                       Tree.of("Karen",
                               Tree.of("Frank")),
                       Tree.of("Steven\nAbbot\nBraddock")),
               Tree.of("Peter",
                       Tree.of("Paul\nPalucci"),
                       Tree.of("Anthony")),
               Tree.of("Christopher",
                       Tree.of("Samuel")));

final String out = PrettyPrinter.pp(tree);
```

```
Ann
├──Mary
│   ├──John
│   │   └──Avila
│   ├──Karen
│   │   └──Frank
│   └──Steven
│      Abbot
│      Braddock
├──Peter
│   ├──Paul
│   │  Palucci
│   └──Anthony
└──Christopher
    └──Samuel
```

javaslang.render.text.Boxes

```
import javaslang.collection.Tree;
import javaslang.render.text.Boxes;

final Tree<String> tree = /* ... */

final String out = Boxes.box(tree).toString();
```

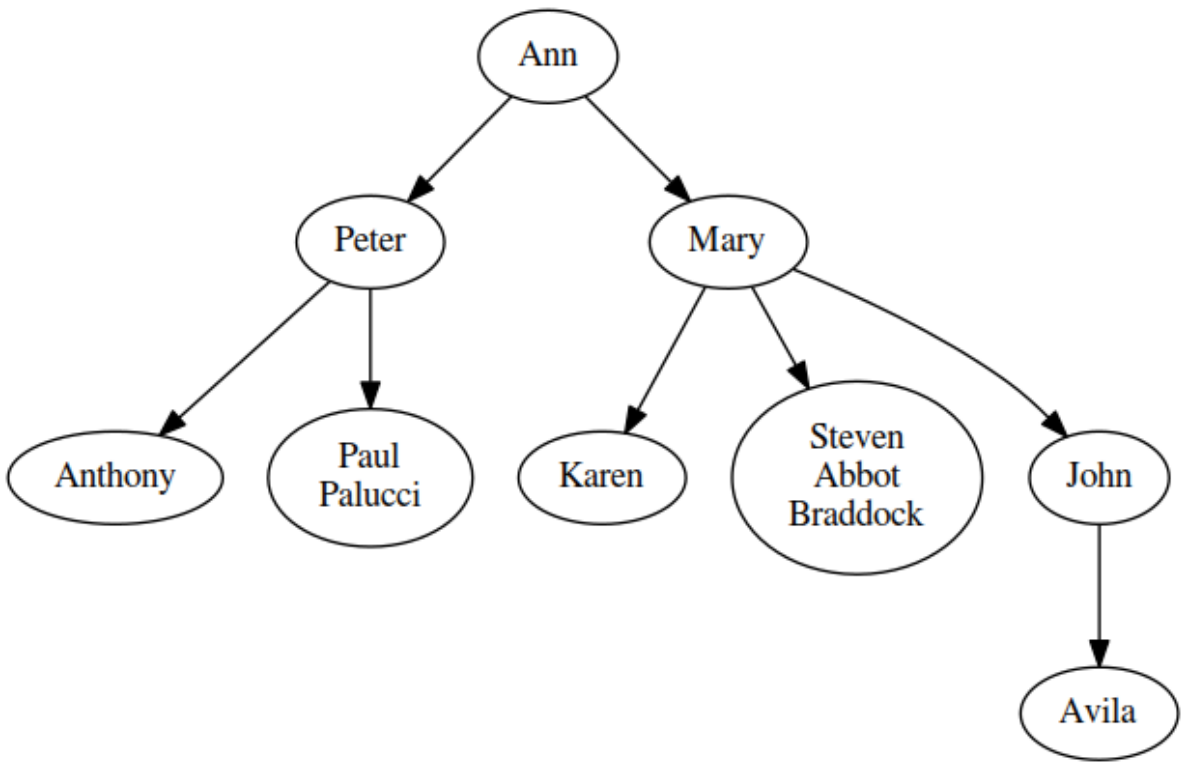


javaslang.render.text.DotFile

```
import javaslang.collection.Tree;
import javaslang.render.dot.DotFile;

final Tree<String> tree = /* ... */;

DotFile.write(tree, "tree.dot");
```

- Dot é uma linguagem para descrever grafos.
- É necessário o pacote **graphviz**.

```
$ dot -Tpng -O tree.dot
```



tree.dot.png

Tópicos

Localização no código fonte

- A localização é usada para reportar erros.
- Indica onde uma frase do programa começa e termina no código fonte.
- Representada pela classe `parse.Loc`

```
Loc(Location left, Location right)
```

- Usa o tipo

```
java_cup.runtime.ComplexSymbolFactory.Location
```

contendo informações como:

  - nome da unidade de compilação (arquivo fonte)
  - número da linha
  - número da coluna

Construção da localização

- Alguns métodos de fábrica:

```
import java_cup.runtime.Symbol;
import java_cup.runtime.ComplexSymbolFactory.ComplexSymbol;
import java_cup.runtime.ComplexSymbolFactory.Location;

/* ... */

public static Loc loc()
public static Loc loc(Location left)
public static Loc loc(Location left, Location right)
public static Loc loc(Symbol symbol)
public static Loc loc(Symbol a, Symbol b)
public static Loc loc(ComplexSymbol symbol)
public static Loc loc(ComplexSymbol a, ComplexSymbol b)
```

- `Symbol` e `ComplexSymbol` representam símbolos terminais

Tópicos

Reportagem de erro

- A classe `error.ErrorManager` é usada para reportar erros.
- Todos os erros reportados são coletados internamente em uma lista.
- Por padrão os erros reportados são exibidos na saída padrão.
- Use a variável de classe `error.ErrorManager.em` para reportar erros.
- Há vários métodos para reportar erros:

```
public void error(String format, Object... args)
public void error(Loc loc, String format, Object... args)

public void warning(String format, Object... args)
public void warning(Loc loc, String format, Object... args)

public void fatal(String format, Object... args)
public void fatal(Loc loc, String format, Object... args)

public void summary()
```

Tópicos

Símbolos terminais

- Os **símbolos terminais** são definidos na gramática livre de contexto do CUP (arquivo `src/main/cup/parser.cup`).
- Exemplos:

```
terminal Double LITNUM;
terminal        PLUS, MINUS, TIMES, DIV;
terminal        LPAREN, RPAREN;
```

- O CUP gera uma interface `parse.SymbolConstants` contendo a definição de constantes correspondentes aos terminais que foram declarados.
- Quando relevante os terminais podem ter um valor semântico associado.
- A classe `ComplexSymbol` é usada para representar os símbolos terminais, contendo as seguintes informações:
  - classificação (como declarado na gramática)
  - lexema
  - localização (começo e fim) no código fonte
  - valor semântico

Regras léxicas

- O analisador léxico é gerado pelo **JFlex**.
- As regras léxicas são especificadas no arquivo `src/main/jflex/lexer.jflex`.
- O JFlex cria a classe `parse.Lexer` compatível com o CUP.
- Os **lexemas** (palavras que formam os símbolos léxicos) são descritos por **expressões regulares**.
- A classificação so símbolo terminal é feita na **ação semântica** (código em Java que faz parte da regra léxica).
- Para facilitar a criação dos diversos símbolos léxicos recomenda-se o uso dos métodos auxiliares (definidos no próprio arquivo de especificação):

```
private Symbol tok(int type, String lexeme, Object value)
private Symbol tok(int type, Object value)
private Symbol tok(int type)
```

Exemplo de regras léxicas

```
[ \t\f\n\r]+              { /* skip */ }

[0-9]+ ("." [0-9]+)?      { return tok(LITNUM, new Double(yytext())); }

"+"                       { return tok(PLUS); }
"-"                       { return tok(MINUS); }
"*"                       { return tok(TIMES); }
"/"                       { return tok(DIV); }
"("                       { return tok(LPAREN); }
")"                       { return tok(RPAREN); }

.                         { em.error(Loc.loc(locLeft()),
                                     "unexpected char '%s'",
                                     yytext()); }
```

Tópicos

Análise sintática

- A especificação sintática (**gramática livre de contexto**) é feita no arquivo `src/main/cup/parser.cup`.
- Algumas opções de execução do CUP são indicadas no arquivo `pom.xml`.
- O analisador sintático é gerado pelo **CUP**, que cria a classe `parse.Parser` e a interface `parse.SymbolConstants`.
- Na gramática livre de contexto são especificados:
  - os símbolos terminais
  - os símbols não terminais
  - o símbolo inicial
  - as regras de produção

Ações semânticas

- Quando um símbolo (terminal ou não terminal) tem um **valor semântico**, o tipo do valor semântico é informado na declaração do símbolo.
- O cálculo do valor semântico do símbolo no lado esquerdo de uma regra de produção é feito usando os valores semênticas dos símbolos que aparecem no lado direiro da regra através de uma **ação semântica** (código em Java).
- Quando um nome é associado a um símbolo no lado direito de uma regra (por exemplo `exp:nome`), o CUP cria três variáveis (no exemplo `nome`, `nomexleft` e `nomexright`) no codigo gerado, contendo o valor semântico, a localização esquerda (onde começa no código fonte), e a localização direita (onde termina no código fonte) do símbolo, respectivamente.

Exemplo de gramática para o CUP

```
terminal Double LITNUM;
terminal        PLUS, MINUS, TIMES, DIV;
terminal        LPAREN, RPAREN;

non terminal Exp exp;
non terminal Exp term;
non terminal Exp factor;

start with exp;

exp ::=
  exp:x PLUS term:y      {: RESULT = new ExpBinOp(ExpBinOp.Op.PLUS, x, y);  :}
| exp:x MINUS term:y     {: RESULT = new ExpBinOp(ExpBinOp.Op.MINUS, x, y); :}
| term:x                 {: RESULT = x;                                      :}
;

term ::=
| term:x TIMES factor:y  {: RESULT = new ExpBinOp(ExpBinOp.Op.TIMES, x, y); :}
| term:x DIV factor:y    {: RESULT = new ExpBinOp(ExpBinOp.Op.DIV, x, y);   :}
| factor:x               {: RESULT = x;                                      :}
;

factor ::=
  LITNUM:x               {: RESULT = new ExpNum(x);                          :}
| LPAREN exp:x RPAREN    {: RESULT = x;                                      :}
;
```

Tópicos

Árvores sintáticas

- **Árvore sintática** é uma estrutura de dados hierárquica que representa a estrutura sintática do programa fonte.
- A **raiz** da árvore é o símbolo inicial da gramática.
- As **folhas** são os símbolos terminais que, lidos da esquerda para a direita, correspondem ao programa fonte.
- Pode ser:
  - **concreta**: todos os símbolos na sequência de derivação são colocados na árvore
  - **abstrata**: apenas as informações relevantes para o entendimento da estrutura do programa são mantidos na árvore
- A saída do **analisador sintático** é uma árvore sintática abstrata (AST).

Representação

- A representação das árvores sintáticas é feita através de classes no pacote `absyn`.
- A classe abstrata `absyn.AST` é superclasse de todas as árvores sintáticas abstratas.
- Esta classe implementa a interface `ToTree`:

```
package javaslang.render;

import javaslang.collection.Tree;

public interface ToTree<E> {
    public abstract Tree.Node<E> toTree();
}
```

- O método `toTree` converte a árvore abstrata em uma estrutura de dados geral para árvores cujos nós armazenam strings, útil na apresentação visual da árvore sintática.

Definindo as árvores abstratas

- Para cada categoria sintática (como expressões, comandos, ou declarações) reprentada por um **símbolo não terminal** definimos uma subclasse abstrata de `absyn.AST`.
- Para cada forma da categoria sintática para um não terminal (como as formas de expressão: expressão constante, operação binária, chamada de função, etc.), representada por uma **regra de produção**, definamos uma subclasse da classe que representa aquela forma específica da categoria sintática.
- Nesta classe deve-se:
  - definir os campos necessários para os componentes (sub-árvores) da árvore sintática,
  - definir construtores que inicializam estes campos com valores passados como argumentos,
  - definir o método `toTree`.

A classe AST

```
package absyn;

import javaslang.render.ToTree;
import org.apache.commons.lang3.builder.ToStringBuilder;
import org.apache.commons.lang3.builder.ToStringStyle;

public abstract class AST implements ToTree<String> {

   @Override
   public String toString() {
      return ToStringBuilder.reflectionToString(
                  this,
                  ToStringStyle.SHORT_PREFIX_STYLE);
   }

}
```

A classe Exp

```
package absyn;

public abstract class Exp extends AST {
}
```

A classe ExpNum

```java
package absyn;

import javaslang.collection.Tree;

public class ExpNum extends Exp {

   public final Double value;

   public ExpNum(Double value) {
      this.value = value;
   }

   @Override
   public Tree.Node<String> toTree() {
      return Tree.of("ExpNum: " + value);
   }
}
```

A classe ExpNum

```java
package absyn;

import javaslang.collection.Tree;

public class ExpBinOp extends Exp {

   public enum Op {PLUS, MINUS, TIMES, DIV}

   public final Op op;
   public final Exp left;
   public final Exp right;

   public ExpBinOp(Op op, Exp left, Exp right) {
      this.op = op;
      this.left = left;
      this.right = right;
   }

   @Override
   public Tree.Node<String> toTree() {
      return Tree.of("ExpBinOp: " + op, left.toTree(), right.toTree());
   }
}
```

Tópicos

Representação intermediária

- A árvore sintática do programa fonte é traduzida para uma representação intermediária do código fonte.
- Usaremos a representação intermediária do framework **LLVM**.
- A biblioteca Java `javacpp-presets-llvm` será usada para criar a representação intermediária.
- A classe `absyn.Exp` deve ter um método abstrato para converter uma expressão para a sua representação intermediária.

```
import static org.bytedeco.javacpp.LLVM.*;
/* ... */
public abstract LLVMValueRef codegen(LLVMModuleRef module,
                                     LLVMBuilderRef builder);
```

- Suas subclasses devem implementar este método.

Representação intermediária de constantes

```
package absyn;

// ...
import static org.bytedeco.javacpp.LLVM.*;

public class ExpNum extends Exp {
   // ...

   @Override
   public LLVMValueRef codegen(LLVMModuleRef module,
                               LLVMBuilderRef builder) {
      return LLVMConstReal(LLVMDoubleType(), value);
   }
}
```

Representação intermediária de operação binária

```java
package absyn;

// ...
import static org.bytedeco.javacpp.LLVM.*;
import static error.ErrorManager.em;

public class ExpBinOp extends Exp {
   // ...

   @Override
   public LLVMValueRef codegen(LLVMModuleRef module, LLVMBuilderRef builder) {
      final LLVMValueRef v_left = left.codegen(module, builder);
      final LLVMValueRef v_right = right.codegen(module, builder);
      switch (op) {
         case PLUS:  return LLVMBuildFAdd(builder, v_left, v_right, "addtmp");
         case MINUS: return LLVMBuildFSub(builder, v_left, v_right, "subtmp");
         case TIMES: return LLVMBuildFMul(builder, v_left, v_right, "multmp");
         case DIV:   return LLVMBuildFDiv(builder, v_left, v_right, "divtmp");
         default:    em.fatal("unknown operator %s in binary operation", op);
                     return LLVMConstReal(LLVMDoubleType(), 0);
      }
   }
}
```

Atividade 1

Atividade 1 (cont.)

Inverso aditivo

Implementar a operação que calcula o inverso aditivo (ou negação) à linguagem *EPLan*. Será usado o operador unário prefixo `-`, com precedência maior que dos operadores aritméticos binários.

1. Definir uma nova classe `absyn.ExpNegate`
   - subclasse de `absyn.Exp`
   - contendo um campo correspondente ao operando da negação
   - implementar o método `toTree`
   - implementar o método `codegen`
2. Acrescentar uma regra de produção adequada na gramática da linguagem.

Dicas:

* Na geração de código use o método `LLVMBuildFNeg`

`LLVMBuildFNeg`

Atividade 1 (cont.)

Comandos úteis:

```
$ cd <working directory>/eplan
$ git checkout master
$ git pull upstream master
$ git checkout -b atividade1
$ # desenvolva sua atividade
$ # faça testes
$ git status
$ git add <arquivos modificados>
$ git commit -m <mensagem>
$ git push origin atividade1
$ # faça um pull request no github
```

Tópicos

Anotando a localização na árvore sintática

- Na análise semêntica os erros encontrados devem ser reportados.
- Deve-se indicar a localização do erro no código fonte.
- Por isto a árvore sintática deve conter a **localização**.
- O atributo `loc` foi adicionado à classe abstrata `absyn.AST`.

```
import parse.Loc;

public abstract class AST implements ToTree<String> {

   // Location where the phrase was found in the source code
   protected final Loc loc;

   public AST(Loc loc) {
      this.loc = loc;
   }

   // ...
}
```

Anotando a localização na árvore sintática (cont.)

- Um argumento correspondente foi adicionado aos construtores das subclasses de `absyn.AST`. Por exemplo:

```
public class ExpReal extends Exp {

   public final Double value;

   public ExpReal(Loc loc, Double value) {
      super(loc);
      this.value = value;
   }

   // ...
}
```

Anotando a localização na árvore sintática (cont.)

- A criação das árvores sintáticas nas ações semânticas do analisador sintático deve informar a localização. Exemplo:

```
factor ::=
  LITNUM:x {: RESULT = new ExpReal(loc(xxleft,xxright), x); :}
;
```

- Na regra de produção, ao indicarmos o nome do valor semântico de um símbolo, são criadas três variáveis no analisador gerado pelo CUP, contendo as seguintes informações:
  - valor semântico do símbolo
  - posição onde o símbolo começou no código fonte
  - posição onde o símbolo terminou no código fonte

  No exemplo estas variáveis são respectivamente `x`, `xxleft`, e `xxright`.

Representando os tipos da linguagem

- O pacote `type` contém classes usadas para representar os tipos da linguagem sendo compilada.
- A representação geral é `Type`.
- `Type` é uma classe abstrata, com subclasses concretas para representar cada uma das possibiliades:

descrição	**classe**	**objeto**
`bool`	`BOOL`	`BOOL.T`
`int`	`INT`	`INT.T`
`real`	`REAL`	`REAL.T`
`char`	`CHAR`	`CHAR.T`
`string`	`STRING`	`STRING.T`
array	`ARRAY`	`new Array(Type element)`
registro	`RECORD`	`new RECORD(List<Tuple2<Symbol,Type>> fields)`
registro nulo	`NIL`	`NIL.T`
`unit`	`UNIT`	`UNIT.T`
nome	`NAME`	`new Name(Symbol name, Type binding)`

Algumas operações com tipos

`Type actual()`

- Retorna a representação que é de fato usada para o tipo.
- É o próprio objeto, exceto para a classe `NAME`, onde é retornado `binding.actual()`.

Algumas operações com tipos (cont.)

```
boolean canBe(Type type)
```

- Verifica se o tipo que recebe a mensagem é compatível (é igual ou pode ser convertido para) o tipo dado como argumento.
- Retorna `true` somente se os dois tipos forem idênticos, exceto nos seguintes casos:
  - o tipo do registo nulo (`nil`), `NIL`, é compatível com `NIL` e com qualquer tipo registro,
  - um tipo `NAME` é compatível com algum tipo, se e somente se o seu atributo `binding` for compatível com o tipo.

Algumas operações com tipos (cont.)

```
boolean canBe(Type... types)
```

- Verifica a compatibilidade com algum dos tipos dados como argumentos.

Reportando erros comuns

- A interface `error.CommonErrors` define alguns métodos para reportar erros comumente encontrados pelo compilador.
- Um dos erros mais comuns é a inconsistência de tipos.

```
public interface CommonErrors {

   default void typeMismatch(Loc loc, Type found, Type... expected) {
      // ...
   }

   // ...
}
```

Análise semântica

- O analisador semântico
  - verifica a consistência do programa
  - calcula o tipo das expressões
- Expressões tem um atributo `type` cujo valor é a representação do tipo obtido para a expressão.
- `type` é calculado pelo analisador semântico.

Análise semântica (cont.)

```
import types.Type;

public abstract class Exp extends AST {

   // Type of the expression, calculated by the semantic analyser
   public Type type;

   // Obtain the type of the expression as a string prefixed by the gi
ven text.
   protected String annotateType(String text) {
      final String theType = type == null ? "" : "\n<" + type + ">";
      return text + theType;
   }

   // ...
}
```

Análise semântica (cont.)

- O método `semantic` da classe `absyn.Exp` faz análise semântica da expressão.
  - Usa o método auxiliar `semantic_` para fazer a análise semântica de fato, obtendo o tipo da expressão.
  - Coloca o tipo encontrado no atributo `type`.

```
public abstract class Exp extends AST {
   // ...

   // Do semantic analysis of the expression
   public Type semantic() {
      type = semantic_();
      return type;
   }

   // Type check the expression. Should be defined in the concrete sub
classes.
   protected abstract Type semantic_();
}
```

Análise semântica (cont.)

- Subclasses de `absyn.Exp` devem definir o método `semantic_` apropriadamente.
- Por exemplo:

```
public class ExpReal extends Exp {
   // ...

   @Override
   protected Type semantic_() {
      return REAL.T;
   }
}
```

Atividade 2: literal inteiro

Implementar **literais inteiros** no compilador de *EPLan*.

1. Para desenvolver esta atividade faça o *checkout* da versão 0.16 do projeto.
2. Definir uma nova classe `types.INT` para representar o tipo `int` de *EPLan*.
3. Definir uma nova classe `absyn.ExpInt` para representar as árvores sintáticas das constantes inteiras:
   - subclasse de `absyn.Exp`
   - contendo um campo correspondente ao valor da constante
   - implementar o método `toTree`
   - implementar o método `semantic_`
   - implementar o método `codegen`
4. Modifique a classe `absyn.ExpBinOp` de forma que os operadores aritméticos aceitem operandos inteiros e reais. Neste momento não é necessário fazer conversão implícita dos operandos de inteiro para real. Por hora os operandos devem ser do mesmo tipo numérico.
5. Acrescentar na gramática da linguagem:
   - o símbolo terminal `LITINT`
   - uma regra de produção adequada

Atividade 2: literal inteiro (cont.)

6. Na especificação léxica da linguagem
   - acrescentar uma regra para o literal inteiro:
     Um literal inteiro é uma sequência não vazia de dígitos decimais.
   - modificar a regra do literal real para não casar com literais inteiros
7. Definir a função `__eplan_print_int` (em C) na biblioteca padrão de *EPLan* para exibir um valor inteiro.
8. Modificar o método `addPrintResult` na classe `codegen.Generator` para considerar o caso de valores inteiros. Este método gera código para exibir o valor de uma expressão na saída padrão.

Dicas:

* Na geração de código use os métodos
  `LLVMConstInt`
  `LLVMBuildAdd`
  `LLVMBuildSub`
  `LLVMBuildMul`
  `LLVMBuildSDiv`

Fim