Relatório de atividade

Alunos: Bruno Ribeiro da Silva (12200992), Djéssica Schell Crocetta (12203762), Luan Felipe Sievers (12204515)

Atribuições

Os alunos Bruno e Luan foram responsáveis pela implementação das extensões da liguagem X+++ para atenderem os requisitos da atividade. A aluna Djéssica foi responsável por parte do relatório e exemplos sintáticos. Abaixo uma lista de contribuições ao repositório do trabalho:

- 75e794a - Bruno Ribeiro da Silva - Added print method. Resolved void to proper types in all parser methods. 7 Jul, 2019
- 2efa7da5 - Luan Felipe Sievers - Adicionados algumas coisas faltantes no relatorio. 18 Jun, 2019
- 0f0f3620 - Luan Felipe Sievers - Deletados arquivos conforme orientação do Prof. 18 Jun, 2019
- e36115f0 - Djéssica S.C - alteração no relatório e exemplo sintatico. 16 Jun, 2019
- 27bd072b - Bruno Ribeiro da Silva - Minor fix in parser to allow debugging. 15 Jun, 2019
- 61bd83da - Luan Felipe Sievers - Fix classbody. 12 Jun, 2019
- e0cf46c9 - Luan Felipe Sievers - Correção do exemplo. 12 Jun, 2019
- 1557fbb8 - Djéssica S.C - exemplos erro sintatico adicionados. 09 Jun, 2019
- 447548a5 - Djéssica S.C - mais exemplos systatic adicionados. 09 Jun, 2019
- e28928cc - Djéssica S.C - exemplos systatic sem erros adicionados. 09 Jun, 2019
- 3a7963b2 - Luan Felipe Sievers - Fix atribdecl on classbody. 09 Jun, 2019
- 722fe9f6 - Luan Felipe Sievers - Implemented atribdecl. 08 Jun, 2019
- 3d3b60fb - Luan Felipe Sievers - Added error test. 08 Jun, 2019
- eb211dac - Luan Felipe Sievers - Added ok test. 08 Jun, 2019
- 30cee0d7 - Luan Felipe Sievers - Removed unnecessary file. 08 Jun, 2019
- fa02886e - Luan Felipe Sievers - Fix lexical example - variable only final needed. 08 Jun, 2019
- 5a2bdfa9 - Luan Felipe Sievers - Removed private, public, protected from vardecl. 08 Jun, 2019
- 3c884ff8 - Bruno Ribeiro da Silva - Added options to run.sh. Updated README and assembler code. 08 Jun, 2019
- 85ad9390 - Bruno Ribeiro da Silva - Improved test routines of run.sh. 08 Jun, 2019
- 0e26b255 - Luan Felipe Sievers - Change order of copy resources - fix compile. 05 Jun, 2019
- 946566ca - Luan Felipe Sievers - fix exemplo lexico. 04 Jun, 2019
- 660d8aa1 - Luan Felipe Sievers - Fix final classdecl, missing trycatch, ifstat. 04 Jun, 2019
- 39393ddf - Bruno Ribeiro da Silva - Updated assembler code to include recovery package in the zip file. Started work report. 04 Jun, 2019
- b18650aa - Luan Felipe Sievers - final. 03 Jun, 2019
- 21092387 - Luan Felipe Sievers - Maybe fix?. 03 Jun, 2019
- e18a24d9 - Luan Felipe Sievers - Adicionado arquivo 'shortcut' para rodar testes. 03 Jun, 2019
- 201f3176 - Luan Felipe Sievers - Adicionados arquivo sintático (organização). 03 Jun, 2019
- 7ae2a662 - Luan Felipe Sievers - Adicionados códigos exemplos de semples. 03 Jun, 2019
- 69bb52c2 - Luan Felipe Sievers - Reorganizado exemplos léxicos. 03 Jun, 2019
- 6fdba899 - Luan Felipe Sievers - Adicionado tratamento para variável com underlinev. 03 Jun, 2019
- 9d674382 - Luan Felipe Sievers - Fix para compilação. 03 Jun, 2019
- c27be37c - Luan Felipe Sievers - Modified maven compile process to include source code from recovery folder. 03 Jun, 2019
- 89642291 - Bruno Ribeiro da Silva - Added semicolon to fix some JavaCC errors. 03 Jun, 2019
- 8c9659a0 - Luan Felipe Sievers - Finalizado código do analisador sintático. 02 Jun, 2019
- 9e4c48d5 - Bruno Ribeiro da Silva - Updated parser code to reflect changes from chapters 4 to 5 of Delamaro's Book. 02 Jun, 2019
- b37a6d7d - Bruno Ribeiro da Silva - Added description to task 2. 17 Mai, 2019
- a06f2ce6 - Luan Felipe Sievers - Entrega 1. 28 Abr, 2019
- 980e36bf - Bruno Ribeiro da Silva - Completed work log. Minor fix in README. 25 Abr, 2019
- fc3c86d7 - Djéssica S.C - Update relatorio.md - incompleto. 24 Abr, 2019
- 4dec0e28 - Luan Felipe Sievers - formatado. 19 Abr, 2019
- 76bc1129 - Luan Felipe Sievers - Alteração da pasta do log para o package. 18 Abr, 2019
- ceeb3d6f - Bruno Ribeiro da Silva - Updated task description. 18 Abr, 2019
- 4c359e79 - Luan Felipe Sievers - adicionado comentários e constantes byte, short, long e float *falta testes. 14 Abr, 2019
- 882e73d0 - Bruno Ribeiro da Silva - Added some constants. Implemented float. 13 Abr, 2019
- 82a8f10e - Bruno Ribeiro da Silva - Formatted parser code. Moved examples to proper folders. 13 Abr, 2019
- 6085ef06 - Bruno Ribeiro da Silva - Updated README file. Added X+++ code. Minor fixes. 12 Abr, 2019
- 12d17119 - Luan Felipe Sievers - Adicionado mais exemplos. 07 Abr, 2019
- 5c4dc3d9 - Bruno Ribeiro da Silva - POM enhanced to automatize some tasks. Instructions added to read me file. 07 Abr, 2019

- 7bca20db - Bruno Ribeiro da Silva - add delamaro book and example code. 04 Abr, 2019
- 5cea126f - Bruno Ribeiro da Silva - Initial commit. 04 Abr, 2019

Especificação léxica da linguagem X+++

Para o desenvolvimento do analisador léxico dessa atividade foi utilizado o código fonte do capítulo 3 do livro "Como Construir um Compilador Utilizando Ferramentas Java". As extensões solicitadas para essa atividade foram:

- Operadores lógicos AND, OR, XOR e NOT;
- Novos tipos de variáveis e literais: BYTE, SHORT, LONG e FLOAT, além dos já existentes;
- Qualificadores de identificadores: FINAL, PUBLIC, PRIVATE e PROTECTED, como usado em Java

O Java Compiler Compiler (JavaCC) foi utilizado através de um plugin para o automizador de compilação Maven. Conforme foram realizadas as alterações no código fonte, foram utilizados para controle os exemplos do livro para verificar se as alterações feitas estavam mantendo a consistência da linguagem, isto é, não estavam sendo detectados erros onde não haviam erros ou tokens inválidos.

Foi verificado que os exemplos error_classbody2.x e error_statement2.x estão com erro de estouro de memória (StackOverflow) e, como foi constatado que o exemplo do livro apresenta o mesmo erro, não foi feito um tratamento para a correção do mesmo.

Operadores lógicos

Os operadores lógicos AND, OR, XOR e NOT foram definidos conforme o trecho de código abaixo:

```
< AND: "&&" > |
< OR: "||" > |
< XOR: "^" > |
< NOT: "!" >
```

Eles foram definidos no mesmo grupo de tokens de operadores. Adicionamos esses operadores no exemplo de código da linguagem X+++ e utilizamos o analisador léxico gerado. O resultado da análise do código fonte X+++ estão disponíveis na pasta log, junto desse relatório.

O código X+++ para validação desses tokens:

```
void checkOperadoresLogicos() {
   if (1 && 1) {
      print "Operador lógico AND - OK";
   }
   if (1 || 0) {
      print "Operador lógico OR - OK";
   }
   if (1 ^ 0) {
      print "Operador lógico XOR - OK";
   }
   if (!0) {
      print "Operador lógico NOT - OK";
   }
}
```

Variáveis e literais

Diferentemente da implementação dos operadores lógicos, para a implementação das variáveis foi necessário o emprego de expressão regular que identifique os padrões de números float. Não foi necessário criar expressões regulares para os tipos short, byte e long porque esses já faziam parte da definição da linguagem do exemplo do livro.

A identificação dessas variáveis e literais foram adicionados aos tokens de palavras reservadas:

```
< BYTE: "byte" > |
< SHORT: "short" > |
< LONG: "long" > |
< FLOAT: "float" >
```

E a expressão regular para float:

```
 < float_constant: (
      <int_constant> "."( <int_constant> ( ("e" | "E")? ("-")? <int_constant> )? )?
      |
```

```
    (<int_constant>)? "." <int_constant> (("e" | "E")? ("-")? <int_constant>)? ) ("F" | "f")?
  >
```

Essa expressão regular permite que sejam identificados números de ponto conforme os padrões abaixo:

- 5.3876e4;
- 7.321E-3;
- 3.2E+4;
- 0.5e-6;
- 0.45;
- 6.e10;

Utilizamos o código abaixo para validação das variáveis e literais implementadas:

```
void checkNovasVariaveis() {
   byte testeByte = 1;
   short testeShort = 1;
   long testeShort = 1;
   float testeFloat1 = 5.3876e4;
   float testeFloat2 = 7.321E-3;
   float testeFloat3 = 3.2E+4;
   float testeFloat4 = 0.5e-6;
   float testeFloat5 = 0.45;
   float testeFloat6 = 6.e10;
}
```

Qualificadores e identificadores

Na etapa de definição do analisador léxico, para adicionar os qualificadores foi necessário apenas reservar as palavras dos mesmos:

```
< FINAL: "final" > |
< PUBLIC: "public" > |
< PRIVATE: "private" > |
< PROTECTED: "protected" >
```

O teste desses qualificadores foi feito adicionando-os aos métodos já existentes. Por exemplo:

```
protected int insert(data k) {
   int x;

   x = k.compara(key);
   if (x < 0) {
      if (left != null)
         return left.insert(k);
      left = new bintree(k);
      return 1;
   }
   if (x > 0) {
      if (right != null)
         return right.insert(k);
      right = new bintree(k);
      return 1;
   }
   return 0;
}
```

Especificação sintática

Para a especificação sintática, a equipe tomou como base os códigos fontes de exemplo para os capítulos 4 e 5 do livro do Delamaro. Incluímos nesses códigos as alterações aqui documentadas para a especificação léxica e incluímos as alterações de requisito para a segunda tarefa da atividade.

Houve uma mudança na estrutura do código do parser, que passou agora a incluir classes Java. Essa alteração refletiu na necessidade de alterar a forma que vínhamos desenvolvendo o trabalho. Passou a ser necessário incluir a pasta de código Java no processo de compilação do projeto. A alteração abaixo foi realizada no arquivo de definição do projeto:

```
<!-- copia as classes de recovery para compilação -->
     <plugin>
      <artifactId>maven-resources-plugin</artifactId>
```

```
      <version>2.6</version>
      <executions>
        <execution>
          <id>copy-resources</id>
          <phase>process-classes</phase>
          <goals>
            <goal>copy-resources</goal>
          </goals>
          <configuration>
            <outputDirectory>${basedir}/target/generated-sources/javacc/recovery/</outputDirectory>
            <encoding>UTF-8</encoding>
            <resources>
              <resource>
                <directory>${basedir}/src/main/javacc/recovery/</directory>
                <includes>
                  <include>*.java</include>
                </includes>
              </resource>
            </resources>
          </configuration>
        </execution>
      </executions>
    </plugin>
```

Além dessa alteração, o código XML que empacota o artefato para entrega da tarefa também foi alterada, para incluir na pasta java essas novas classes.

Quanto a essas novas classes Java, só houve necessidade de alteração do código fonte da classe First, para que fossem incluídos, por exemplo, os modificadores de acesso que implementamos:

```
// O trecho de código exemplifica nossa alteração
public class First {

    static public final RecoverySet methoddecl = new RecoverySet();
    static public final RecoverySet vardecl = new RecoverySet();
    static public final RecoverySet classlist = new RecoverySet();
    static public final RecoverySet constructdecl = new RecoverySet();
    static public final RecoverySet statlist = new RecoverySet();
    static public final RecoverySet program = classlist;

    static {
        methoddecl.add(new Integer(langXConstants.INT));
        methoddecl.add(new Integer(langXConstants.STRING));
        methoddecl.add(new Integer(langXConstants.IDENT));
        methoddecl.add(new Integer(langXConstants.BYTE));
        methoddecl.add(new Integer(langXConstants.SHORT));
        methoddecl.add(new Integer(langXConstants.LONG));
        methoddecl.add(new Integer(langXConstants.FLOAT));
        methoddecl.add(new Integer(langXConstants.PUBLIC));
        methoddecl.add(new Integer(langXConstants.PRIVATE));
        methoddecl.add(new Integer(langXConstants.PROTECTED));
        methoddecl.add(new Integer(langXConstants.FINAL));
    }
}
```

Definição sintática para operadores lógicos

O trecho de código abaixo contém a definição implementada para o uso de operadores lógicos tais como são utilizados em Java:

```
void logicexpression(RecoverySet g) throws ParseEOFException:
{
    RecoverySet f1 = new RecoverySet(XOR).union(g);
    RecoverySet f2 = new RecoverySet(OR).union(f1);
    RecoverySet f3 = new RecoverySet(AND).union(f2);
}{
    try {
        [<NOT>] expression(f3) (( <XOR> | <OR> | <AND>) [<NOT>] expression(f3))*
      } catch (ParseException e) {
        consumeUntil(g, e, "logicexpression");
    }
}
```

Abaixo um trecho de código na linguagem X+++ com o emprego dessas operações:

```
protected void testarDoisValores(float x, log y) {
    if (x && y) {
```

```
        print "AND";
    }
    if (x || y) {
        print "OR";
    }
    if (x ^ y) {
        print "XOR";
    }
    if (!x) {
        print "NOT";
    }
}
```

Definição sintática para variáveis e literais

A definição do não terminal responsável pelos novos tipos que incluímos no código do parser da linguagem X+++ segue abaixo:

```
void primitivetype(RecoverySet g) throws ParseEOFException:
{}{
    <INT> | <STRING> | <BYTE> | <SHORT> | <LONG> | <FLOAT>
}
```

E para que seja possível inicializar uma variável fora de qualquer método, os seguintes trechos foram adicionados:

Método para inicialicação do tipo int x = 1;

```
void atribdecl(RecoverySet g) throws ParseEOFException:
{
    RecoverySet f1 = new RecoverySet(IDENT).union(g);
}{
    try {
        [<FINAL>]
        primitivetype(f1)
        <IDENT>
        <ASSIGN>
        (alocexpression(g) | LOOKAHEAD(3) logicexpression(g))
    } catch (ParseException e) {
        consumeUntil(g, e, "atribdecl");
    }
}
```

Possibilidade de inicialicação dentro da classe

```
void classbody(RecoverySet g) throws ParseEOFException:
{
    RecoverySet f2 = new RecoverySet(SEMICOLON).union(g).remove(IDENT),
                f3 = First.methoddecl.union(g).remove(IDENT),
                f4 = First.constructdecl.union(f3).remove(IDENT),
                f5 = First.vardecl.union(f4).remove(IDENT);
}{
    try {
        <LBRACE>
          [LOOKAHEAD(2) classlist(f5)]
          (LOOKAHEAD(4) vardecl(f2) <SEMICOLON>
          |
          LOOKAHEAD(2) atribstat(f2) <SEMICOLON>
          |
          LOOKAHEAD(4) atribdecl(f2) <SEMICOLON>
          |
          LOOKAHEAD(2) constructdecl(f4)
          |
          methoddecl(f3)
          )*
        <RBRACE>
    }
}
```

Ainda na parte de declaração de variável foi definido também a parte de inicialização dentro do método no trecho a seguir no métogo `statement`

Possibilidade de inicialicação dentro de métodos

```
void statement(RecoverySet g) throws ParseEOFException:
.
```

```
.
.

      <SEMICOLON>
   |
      LOOKAHEAD(4)
      atribdecl(f1) <SEMICOLON>
   } catch (ParseException e) {
      consumeUntil(g, e, "statement");
   }
}
```

Na sequência seu uso na linguagem X+++ num código de controle:

```
class exemplo_syntatic_ok {

   byte exemploByte = 2;
   short exemploShort = 1;
   long exemploLong = 2.3456;
   float exemploFloat = 1.2;

   byte exemploByte2;
   short exemploShort2;
   long exemploLong2;
   float exemploFloat2;
   exemploByte2 = 3;
   exemploShort2 = 4;
   exemploLong2 = 3.456;
   exemploFloat2 = 5.1;

   byte exemploByte3;
   short exemploShort3;
   long exemploLong3;
   float exemploFloat3;

   short comparaSeShortIgualAoDefinido(short x) {
      if (x == exemploShort) {
         print "Short igual!";
         return 1;
      }
      return -1;
   }
}
```

Definição sintática de qualificadores e identificadores

A implementação sintática dos qualificadores foram realizadas em mais partes do código porque estas ocorrem tanto em definições de métodos quanto de variáveis e outros pontos, tais como temos na linguagem Java. O trecho abaixo exemplifica como foram feitas essas alterações:

```
void methoddecl(RecoverySet g) throws ParseEOFException:
{
   RecoverySet f1 = new RecoverySet(LBRACKET).union(g);
   RecoverySet f2 = new RecoverySet(IDENT).union(f1);
}{
   [<PUBLIC> | <PRIVATE> | <PROTECTED>]
   [<FINAL>]
   (primitivetype(f2) | <IDENT> ) (<LBRACKET> <RBRACKET>)*
   <IDENT> methodbody(g)
}
```

Com essa alteração o trecho de código abaixo é identificado corretamente pelo compilador:

```
protected long comparaSeLongMaiorQueDefinido(long x) {
   if (x > exemploByte) {
      print "Long maior!";
      return 1;
   }
   return -1;
}

private float comparaSeFloatIgualAoDefinido(float x) {
   if(x == exemploFloat){
      print "float igual!";
      return 1;
   }
```

```
    return -1;
}

public byte comparaSeByteMenorQueDefinido(byte x) {
    if (x < exemploByte) {
        print "Byte menor!";
        return 1;
    }
    return -1;
}

public final void comparaDoisFloat(float x) {
    if(x < exemploFloat) { print "Valor informado Menor que o definido!"; }
    if(x > exemploFloat) { print "Valor informado Maior que o definido!"; }
    if(x == exemploFloat) { print "Valor informado Igual ao definido!"; }
}
```

Impressão da árvore sintática

Para a impressão da árvore sintática, foi necessária a inclusão de 34 classes Java do pacote syntacticTree, que totalizam aproximadamente 1400 linhas de código. Além disso, toda a definição do parser precisou ser revisada porque todos os métodos retornavam nulo e nesse ponto precisam retornar objetos do pacote syntacticTree correspondentes ao método em avaliação. Além dos retornos, o método principal do parser também foi alterado para a adição da opção de impressão da árvore sintática. Infelizmente após as alterações que permearam por toda a base de código não foi possível realizar a impressão da árvore sintática. O retorno da aplicação informa que está pendente um retorno de algum método, mas não fica claro a razão pela qual o retorno não é produzido. Em face do problema no estado atual da construção da árvore sintática, não foram produzidas classes para representar os novos tipos de nodos possíveis para a árvore com base nas adições da linguagem X+++ em relação a X++.