UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO

RELATÓRIO DA IMPLEMENTAÇÃO DE COMPILADOR 1 (IC1)

João Vítor Demaria Venâncio (17104491)
Lucas Oliveira (16202595)
Lucas Varella (17100914)

Florianópolis
4 de novembro de 2019

Sumário

1. Introdução

Este trabalho consiste na formulação e implementação de um compilador de linguagem de programação, de acordo com aquilo descrito no livro Delamaro (2004), especificamente projeto de analisador léxico e sintático para uma linguagem X++, utilizando Java CC.

1.1 Integrantes da equipe e responsabilidades

Nome	Responsabilidade
João Vìtor Demaria Venâncio	- Implementação do “Possibilidade de inicializar uma variável fora de qualquer método” - Implementação do “Operadores lógicos AND, OR, XOR e NOT” - Implementação do “Novos tipos de variáveis e literais: BYTE, SHORT, LONG e FLOAT, além dos já existentes” - Implementação do “Qualificadores de identificadores: FINAL, PUBLIC, PRIVATE e PROTECTED, como usado em Java” - Implementação do “Uso de expressões lógicas nos contextos adequados” - Implementação do “Expressões lógicas com estes operadores”
Lucas Varella	- Atividades de teste da linguagem - Formulação do arquivo de teste para erro - Auxílio na implementação do reconhecimento de padrões de classe - Auxílio na implementação do reconhecimento de padrões de atributo - Auxílio na implementação do reconhecimento de padrões de variável - Auxílio na implementação do reconhecimento de padrões de método
Lucas Oliveira	- Atividades de teste da linguagem - Formulação do arquivo de teste para erro - Auxílio na implementação do reconhecimento de padrões de classe - Auxílio na implementação do reconhecimento de padrões de atributo - Auxílio na implementação do reconhecimento de padrões de variável - Auxílio na implementação do reconhecimento de padrões de método

2. Desenvolvimento

Foi desenvolvido cada parte do trabalho na

2.1 Possibilidade de inicializar uma variável fora de qualquer método

Dentro de classbody foi colocado o vardecl(), além de arrumar os lookaheads para que o analisador sintático detectasse corretamente:

```
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(3) classlist(f5)] //Colocado lookahead
      (
      LOOKAHEAD(6) methoddecl(f3) |
      LOOKAHEAD(5) varattrib(f3)  <SEMICOLON> | //Aqui
      LOOKAHEAD(4) vardecl(f2) <SEMICOLON> |
      LOOKAHEAD(3) constructdecl(f4)
  )*
   <RBRACE>
}
catch (ParseException e)
{
   consumeUntil(g, e, "classbody");
}
}
```

2.2 Operadores lógicos AND, OR, XOR e NOT

Foi adicionado no conjunto de tokens de operações os operadores AND, OR, XOR e NOT, sendo respectivamente reconhecidos através de "&&", "||", "^", "!". Pode ser conferido através do seguinte print:

```
/* Operadores */

TOKEN :
{
  < ASSIGN: "=" >
| < GT: ">" >
| < LT: "<" >
| < EQ: "==" >
| < LE: "<=" >
| < GE: ">=" >
| < NEQ: "!=" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < REM: "%" >
| < AND: "&&">
| < OR: "||">
| < XOR: "^">
| < NOT: "!">
}
```

2.3 Expressões lógicas com estes operadores

Foi criado mais um não-terminal, o lognumexpr(), e o expression foi modificado:

```
void expression(RecoverySet g) throws ParseEOFException :
{
}
{
try {
  (<NOT> <LPAREN> lognumexpr() <RPAREN> | lognumexpr() ) [ ( ( <AND> | <OR> | <XOR> ) ( <NOT> <LPAREN> lognumexpr() <RPAREN> | lognumexpr()) )+ ]
}
catch (ParseException e)
{
   consumeUntil(g, e, "expression");
}
}

void lognumexpr() throws ParseEOFException:
{
}
{
  numexpr() [( <LT> | <GT> | <LE> | <GE> | <EQ> | <NEQ>) numexpr()]
}
```

2.4 Uso de expressões lógicas nos contextos adequados

Eles podem ser usados como expressions, logo eles entram em IFs, FORs, e qualquer um outro que aceite o não-terminal “expression”:

```
void expression(RecoverySet g) throws ParseEOFException :
{
}
{
try {
  (<NOT> <LPAREN> lognumexpr() <RPAREN> | lognumexpr() ) [ ( ( <AND> | <OR> | <XOR> ) ( <NOT> <LPAREN> lognumexpr() <RPAREN> | lognumexpr()) )+ ]
}
catch (ParseException e)
{
   consumeUntil(g, e, "expression");
}
}

void lognumexpr() throws ParseEOFException:
{
}
{
  numexpr() [( <LT> | <GT> | <LE> | <GE> | <EQ> | <NEQ>) numexpr()]
}
```

2.5 Novos tipos de variáveis e literais: BYTE, SHORT, LONG e FLOAT, além dos já existentes

No analisador léxico:

```
/* Palavras reservadas */

TOKEN :
{
  < BREAK: "break" >
| < CLASS: "class" >
| < CONSTRUCTOR: "constructor" >
| < ELSE: "else" >
| < EXTENDS: "extends" >
| < FOR: "for" >
| < IF: "if" >
| < INT: "int" >
| < BYTE: "byte" > //Byte
| < SHORT: "short" > //Short
| < LONG: "long" > //Long
| < FLOAT: "float" > //Float
| < NEW: "new" >
| < PRINT: "print" >
| < READ: "read" >
| < RETURN: "return" >
| < STRING: "string" >
| < SUPER: "super" >
| < FINAL: "final" >
| < PUBLIC: "public" >
| < PRIVATE: "private" >
| < PROTECTED: "protected" >
}
```

```
/* constantes */

TOKEN :
{
  < int_constant:(  // números decimais, octais, hexadecimais ou binários
                  (["0"-"9"] (["0"-"9"])* ) |
                  (["0"-"7"] (["0"-"7"])* ["o", "O"] ) |
                  (["0"-"9"] (["0"-"7","A"-"F","a"-"f"])* ["h", "H"] ) |
                  (["0"-"1"] (["0"-"1"])* ["b", "B"])
                 ) >
|
< byte_constant:( // números do tipo byte
                 (["0"-"9"] (["0"-"9"])* )

  ) >
|
//BYTE, SHORT, LONG e FLOAT
< short_constant:(// números do tipo short
                  (["0"-"9"] (["0"-"9"])* )
  ) >
|
< long_constant:(// números do tipo long
                 (["0"-"9"] (["0"-"9"])* )

  ) >
|
< float_constant:(// números do tipo float
                  ( ["0"-"9"] (["0"-"9"])* "." ["0"-"9"] (["0"-"9"])* ["f"] )
  ) >
|
  < string_constant:    // constante string como "abcd bcda"
      "\""( ~["\"","\n","\r"])* "\"" >
|
  < null_constant: "null" > // constante null
}
```

No analisador sintático foi adicionado o <BYTE> | <SHORT> | <LONG> | <FLOAT>:

```
void varattrib(RecoverySet g) throws ParseEOFException :
{
}
{
try {
   [ <PUBLIC> | <PRIVATE> | <PROTECTED> ] [ <FINAL> ] ( <BYTE> | <SHORT> | <LONG> | <FLOAT> | <INT> |  <STRING> | <IDENT> )
   <IDENT> ( <LBRACKET> <RBRACKET>)*
  (<COMMA> <IDENT> ( <LBRACKET> <RBRACKET>)* )* <ASSIGN> ( alocexpression(g) | expression(g))
}
catch (ParseException e)
{
   consumeUntil(g, e, "vardecl");
}
}
```

```
void vardecl(RecoverySet g) throws ParseEOFException :
{
}
{
try {
   [ <PUBLIC> | <PRIVATE> | <PROTECTED> ] [ <FINAL> ] ( <BYTE> | <SHORT> | <LONG> | <FLOAT> | <INT> |  <STRING> | <IDENT> )
   <IDENT> ( <LBRACKET> <RBRACKET>)*
  (<COMMA> <IDENT> ( <LBRACKET> <RBRACKET>)* )*
}
catch (ParseException e)
{
   consumeUntil(g, e, "vardecl");
}
}
```

```
void methoddecl(RecoverySet g) throws ParseEOFException :
{
}
{
try {
   [ <PUBLIC> | <PRIVATE> | <PROTECTED> ] [ <FINAL> ] ( <BYTE> | <SHORT> | <LONG> | <FLOAT> | <INT> | <STRING> | <IDENT> ) (<LBRACKET> <RBRACKET>)*
   <IDENT> methodbody(g)
}
catch (ParseException e)
{
   consumeUntil(g, e, "methoddecl");
}
}
```

```
void paramlist(RecoverySet g) throws ParseEOFException :
{
}
{
try {
   [
    ( <BYTE> | <SHORT> | <LONG> | <FLOAT> | <INT> |  <STRING> | <IDENT>) <IDENT> (<LBRACKET> <RBRACKET>)*
    (<COMMA> (<BYTE> | <SHORT> | <LONG> | <FLOAT> | <INT> | <STRING> | <IDENT>) <IDENT> (<LBRACKET> <RBRACKET>)* )*
   ]
}
catch (ParseException e)
{
   consumeUntil(g, e, "paramlist");
}
}
```

```
void alocexpression(RecoverySet g) throws ParseEOFException :
{
RecoverySet f1 = new RecoverySet(RPAREN).union(g),
            f2 = new RecoverySet(RBRACKET).union(g);
}
{
    <NEW> (
             LOOKAHEAD(2) <IDENT> <LPAREN> arglist(f1) <RPAREN> |
             ( <BYTE> | <SHORT> | <LONG> | <FLOAT> | <INT> | <STRING> | <IDENT> )
             (<LBRACKET> expression(f2) <RBRACKET>)+
          )
}
```

2.6 Qualificadores de identificadores: FINAL, PUBLIC, PRIVATE e PROTECTED, como usado em Java

No conjunto de tokens de palavras reservadas do analisador léxico foi adicionado no final da lista o FINAL, PUBLIC, PRIVATE e PROTECTED, o qual pode ser observado através do print a seguir:

```
/* Palavras reservadas */

TOKEN :
{
  < BREAK: "break" >
| < CLASS: "class" >
| < CONSTRUCTOR: "constructor" >
| < ELSE: "else" >
| < EXTENDS: "extends" >
| < FOR: "for" >
| < IF: "if" >
| < INT: "int" >
| < BYTE: "byte" > //Byte
| < SHORT: "short" > //Short
| < LONG: "long" > //Long
| < FLOAT: "float" > //Float
| < NEW: "new" >
| < PRINT: "print" >
| < READ: "read" >
| < RETURN: "return" >
| < STRING: "string" >
| < SUPER: "super" >
| < FINAL: "final" > //Final
| < PUBLIC: "public" > //Public
| < PRIVATE: "private" > //Private
| < PROTECTED: "protected" > //Protected
}
```

No analisador sintático foi adicionado [<PUBLIC> | <PRIVATE> | <PROTECTED>] [<FINAL>]:

```
void classdecl(RecoverySet g) throws ParseEOFException :
{
}
{
try {
   [ <PUBLIC> | <PRIVATE> | <PROTECTED> ] [ <FINAL> ] <CLASS>  <IDENT> [ <EXTENDS> <IDENT> ] classbody(g)
}
catch (ParseException e)
{
    consumeUntil(g, e, "classdecl");
}
}
```

```
void varattrib(RecoverySet g) throws ParseEOFException :
{
}
{
try {
   [ <PUBLIC> | <PRIVATE> | <PROTECTED> ] [ <FINAL> ] ( <BYTE> | <SHORT> | <LONG> | <FLOAT> | <INT> |  <STRING> | <IDENT> )
   <IDENT> ( <LBRACKET> <RBRACKET>)*
  (<COMMA> <IDENT> ( <LBRACKET> <RBRACKET>)* )* <ASSIGN> ( alocexpression(g) | expression(g))
}
catch (ParseException e)
{
   consumeUntil(g, e, "vardecl");
}
}
```

```
void vardecl(RecoverySet g) throws ParseEOFException :
{
}
{
try {
   [ <PUBLIC> | <PRIVATE> | <PROTECTED> ] [ <FINAL> ] ( <BYTE> | <SHORT> | <LONG> | <FLOAT> | <INT> |  <STRING> | <IDENT> )
   <IDENT> ( <LBRACKET> <RBRACKET>)*
  (<COMMA> <IDENT> ( <LBRACKET> <RBRACKET>)* )*
}
catch (ParseException e)
{
   consumeUntil(g, e, "vardecl");
}
}
```

```
void constructdecl(RecoverySet g) throws ParseEOFException :
{
}
{
try {
    [ <PUBLIC> | <PRIVATE> | <PROTECTED> ] [ <FINAL> ] <CONSTRUCTOR> methodbody(g)
}
catch (ParseException e)
{
   consumeUntil(g, e, "constructdecl");
}
}
```

```
void methoddecl(RecoverySet g) throws ParseEOFException :
{
}
{
try {
   [ <PUBLIC> | <PRIVATE> | <PROTECTED> ] [ <FINAL> ] ( <BYTE> | <SHORT> | <LONG> | <FLOAT> | <INT> | <STRING> | <IDENT> ) (<LBRACKET> <RBRACKET>)*
   <IDENT> methodbody(g)
}
catch (ParseException e)
{
   consumeUntil(g, e, "methoddecl");
}
}
```

3. Testes

Para testar o compilador, foi utilizado na pasta “\01-analisador_lexico_e_sintatico\” os seguintes comandos

```
cd parser
javacc "langX++.jj"
cd ..
javac parser\langX.java
java parser.langX -debug_recovery testes/documentoTeste.x
```

Foi testado usando os arquivos “documentoTeste.x” e “documentoTesteErro.x”

Referências

DELAMARO, M. E. Como Construir um Compilador: utilizando ferramenta Java. São Paulo, Brazil: Novatec Editora Ltda, 2004.