UNIVERSIDADE FEDERAL DE SANTA CATARINA

Departamento de Informática e Estatística - CTC

Curso de Ciências da Computação

Disciplina: Construção de Compiladores

Nome: Anselmo Soethe Nurnberg Junior

Deise Luise Wrasse

Suzana Pescador

**Compilação e execução no linux**

Javacc:

$ ./javacc langX++.jj

Compilar arquivos gerados:

$ javac langX.java langXConstants.java langXTokenManager.java ParseException.java SimpleCharStream.java Token.java TokenMgrError.java

Execução:

$ java parser.langX -debug\_AS samples/teste.x

**Código**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Arquivo a ser processado pelo programa JavaCC.  
Contém:  
 - descrição do analisador léxico para a linguagem X++  
  
 Autor: Marcio Delamaro, Suzana Pescador, Anselmo Soethe Nurnberg Junior, Deise Luise Wrasse  
Este programa é parte do livro "Construção de compiladores usando ferra..."  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

options {

STATIC = false;

DEBUG\_LOOKAHEAD = true;

}

PARSER\_BEGIN(langX)

package parser;

import java.io.\*;

public class langX {

final static String Version = "X++ Compiler - Version 1.0 - 2004";

int contParseError = 0; // contador de erros sintáticos

// Define o método "main" da classe langX.

public static void main(String args[]) throws ParseException

{

boolean debug = false;

String filename = ""; // nome do arquivo a ser analisado

langX parser; // analisador léxico/sintático

int i;

boolean ms = false;

System.out.println(Version);

// lê os parâmetros passados para o compilador

for (i = 0; i < args.length - 1; i++)

{

if (args[i].equals("-debug\_AS") )

debug = true;

else

{

System.out.println("Usage is: java langX [-debug\_AS] inputfile");

System.exit(0);

}

}

if (args[i].equals("-"))

{ // lê da entrada padrão

System.out.println("Reading from standard input . . .");

parser = new langX(System.in); // cria AS

}

else

{ // lê do arquivo

filename = args[args.length-1];

System.out.println("Reading from file " + filename + " . . .");

try { // cria AS

parser = new langX(new java.io.FileInputStream(filename));

}

catch (java.io.FileNotFoundException e) {

System.out.println("File " + filename + " not found.");

return;

}

}

if (! debug) parser.disable\_tracing(); // desabilita verbose do AS

try {

parser.program(); // chama o método que faz a análise

}

catch (ParseException e)

{

System.err.println(e.getMessage());

parser.contParseError = 1; // não existe recuperação de erros

}

finally {

System.out.println(parser.token\_source.foundLexError() +

" Lexical Errors found");

System.out.println(parser.contParseError + " Syntactic Errors found");

}

} // main

static public String im(int x)

{

int k;

String s;

s = tokenImage[x];

k = s.lastIndexOf("\"");

try {s = s.substring(1,k);}

catch (StringIndexOutOfBoundsException e)

{}

return s;

}

} // langX

PARSER\_END(langX)

TOKEN\_MGR\_DECLS :

{

int countLexError = 0;

public int foundLexError()

{

return countLexError;

}

}

/\* Espacos a serem desprezados no inicio de cada token \*/

SKIP :

{

" "

| "\t"

| "\n"

| "\r"

| "\f"

}

SKIP :

{

"/\*" : multilinecomment

}

SKIP :

{

"//" : singlelinecomment

}

<multilinecomment> SKIP:

{

"\*/" : DEFAULT

| <~[]>

}

<singlelinecomment> SKIP:

{

<["\n","\r"]> : DEFAULT

| <~[]>

}

/\* Palavras reservadas \*/

TOKEN :

{

< BREAK: "break" >

| < CLASS: "class" >

| < CONSTRUCTOR: "constructor" >

| < ELSE: "else" >

| < EXTENDS: "extends" >

| < FOR: "for" >

| < IF: "if" >

| < INT: "int" >

| < BOOLEAN: "boolean" >

| < DOUBLE: "double" >

| < WHILE: "while" >

| < SWITCH: "switch" >

| < CASE: "case" >

| < DEF: "default" >

| < NEW: "new" >

| < PRINT: "print" >

| < READ: "read" >

| < RETURN: "return" >

| < STRING: "string" >

| < SUPER: "super" >

}

/\* 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"])

) >

|

< string\_constant: // constante string como "abcd bcda"

"\""( ~["\"","\n","\r"])\* "\"" >

|

< null\_constant: "null" > // constante null

}

/\* Identificadores \*/

TOKEN :

{

< IDENT: <LETTER> (<LETTER>|<DIGIT>)\* >

|

< #LETTER:["A"-"Z","a"-"z"] >

|

< #DIGIT:["0"-"9"] >

}

/\* Simbolos especiais \*/

TOKEN :

{

< LPAREN: "(" >

| < RPAREN: ")" >

| < LBRACE: "{" >

| < RBRACE: "}" >

| < LBRACKET: "[" >

| < RBRACKET: "]" >

| < SEMICOLON: ";" >

| < COLON: ":" >

| < COMMA: "," >

| < DOT: "." >

}

/\* Operadores \*/

TOKEN :

{

< ASSIGN: "=" >

| < GT: ">" >

| < LT: "<" >

| < EQ: "==" >

| < LE: "<=" >

| < GE: ">=" >

| < NEQ: "!=" >

| < PLUS: "+" >

| < PLUSPLUS: "++" >

| < MINUSMINUS: "--" >

| < MINUS: "-" >

| < STAR: "\*" >

| < SLASH: "/" >

| < REM: "%" >

}

/\* Trata os erros léxicos \*/

SPECIAL\_TOKEN :

{

<INVALID\_LEXICAL:

(~ ["a"-"z", "A"-"Z",

"0"-"9",

"\"",

"(",

")",

"[",

"]",

"{",

"}",

";",

",",

".",

"=",

">",

"<",

"!",

"+",

"-",

"\*",

"/",

"%",

" ",

"\t",

"\n",

"\r",

"\f"

])+>

{

System.err.println("Line " + input\_stream.getEndLine() +

" - Invalid string found: " + image);

countLexError++;

}

|

<INVALID\_CONST:

"\"" (~ ["\n","\r","\""])\* ["\n","\r"]>

{

System.err.println("Line " + input\_stream.getEndLine() +

" - String constant has a \\n: " + image);

countLexError++;

}

}

void program() :

{

}

{

[ classlist() ] <EOF>

}

void classlist():

{

}

{

classdecl() [ classlist() ]

}

void classdecl():

{

}

{

<CLASS> <IDENT> [ <EXTENDS> <IDENT> ] classbody()

}

void classbody():

{

}

{

<LBRACE>

[classlist()]

(LOOKAHEAD(3) vardecl() <SEMICOLON>)\*

(constructdecl())\*

(methoddecl())\*

<RBRACE>

}

void vardecl():

{

}

{

(<INT> | <STRING> | <IDENT> | <BOOLEAN> | <DOUBLE>)

<IDENT> ( <LBRACKET> <RBRACKET>)\*

(<COMMA> <IDENT> ( <LBRACKET> <RBRACKET>)\* )\*

}

void constructdecl():

{

}

{

<CONSTRUCTOR> methodbody()

}

void methoddecl():

{

}

{

(<INT> | <STRING> | <IDENT> | <BOOLEAN> | <DOUBLE>) (<LBRACKET> <RBRACKET>)\*

<IDENT> methodbody()

}

void methodbody():

{

}

{

<LPAREN> paramlist() <RPAREN> statement()

}

void paramlist():

{

}

{

[

(<INT> | <STRING> | <IDENT> | <BOOLEAN> | <DOUBLE>) <IDENT> (<LBRACKET> <RBRACKET>)\*

(<COMMA> (<INT> | <STRING> | <IDENT> | <BOOLEAN> | <DOUBLE>) <IDENT> (<LBRACKET> <RBRACKET>)\* )\*

]

}

void statement():

{

}

{

LOOKAHEAD(2)

vardecl() <SEMICOLON>

|

atribstat() <SEMICOLON>

|

printstat() <SEMICOLON>

|

readstat() <SEMICOLON>

|

returnstat() <SEMICOLON>

|

superstat() <SEMICOLON>

|

ifstat()

|

forstat()

|

whilestat()

|

switchstat()

|

<LBRACE> statlist() <RBRACE>

|

<BREAK> <SEMICOLON>

|

<SEMICOLON>

}

void atribstat():

{

}

{

LOOKAHEAD(2)

lvalue() <ASSIGN> ( alocexpression() | expression())

|

lvalue() [<PLUSPLUS> | <MINUSMINUS> ]

}

void printstat():

{

}

{

<PRINT> expression()

}

void readstat():

{

}

{

<READ> lvalue()

}

void returnstat():

{

}

{

<RETURN> [expression()]

}

void superstat():

{

}

{

<SUPER> <LPAREN> arglist() <RPAREN>

}

void ifstat():

{

}

{

<IF> <LPAREN> expression() <RPAREN> statement()

[ <ELSE> statement()]

}

void forstat():

{

}

{

<FOR> <LPAREN> [atribstat()] <SEMICOLON>

[expression()] <SEMICOLON>

[atribstat()] <RPAREN>

statement()

}

void whilestat():

{

}

{

<WHILE> <LPAREN> [expression()] <RPAREN>

statement()

}

void switchstat():

{

}

{

<SWITCH> <LPAREN> lvalue() <RPAREN> <LBRACE>

(<CASE> factor() <COLON> statement() (<BREAK> <SEMICOLON>))\*

<DEF> <COLON> statement()

<RBRACE>

}

void statlist() :

{

}

{

statement() [statlist()]

}

void lvalue() :

{

}

{

<IDENT> (

<LBRACKET> expression() <RBRACKET> |

<DOT> <IDENT> [<LPAREN> arglist() <RPAREN>]

)\*

}

void alocexpression() :

{

}

{

<NEW> (

LOOKAHEAD(2) <IDENT> <LPAREN> arglist() <RPAREN> |

( <INT> | <STRING> | <IDENT> )

(<LBRACKET> expression() <RBRACKET>)+

)

}

void expression() :

{

}

{

numexpr() [( <LT> | <GT> | <LE> | <GE> | <EQ> | <NEQ>) numexpr()]

}

void numexpr():

{

}

{

term() ((<PLUS> | <MINUS>) term())\*

}

void term():

{

}

{

unaryexpr() ((<STAR> | <SLASH>| <REM>) unaryexpr())\*

}

void unaryexpr() :

{

}

{

[(<PLUS> | <MINUS>)] factor()

}

void factor():

{

}

{

(

<int\_constant> |

<string\_constant> |

<null\_constant> |

lvalue() |

<LPAREN> expression() <RPAREN>)

}

void arglist():

{

}

{

[expression() (<COMMA> expression())\*]

}