FACULDADE COTEMIG

COMPILADORES

TRABALHO PRÁTICO – SEGUNDA ETAPA

Professor
Virgilio Borges de Oliveira

Leandro Henrique Daldegam Fontes
Paulo Henrique de Almeida
COTEMIG 2016/2

O presente documento tem como objetivo apresentar a linguagem de programação “LP”, uma linguagem de paradigma procedural, fortemente tipada e declarativa, cujo seus operadores são um subconjunto da linguagem C, criada por Dennis Ritchie em 1972, com algumas adaptações.

A escolha da linguagem C como base para desenvolvimento da linguagem LP, deve-se à sua sintaxe simples, com palavras reservadas derivadas da língua inglesa.

As linhas de comandos da linguagem LP devem sempre ser finalizadas com ponto e virgula, salvo em casos específicos como os blocos de comandos. As variáveis são declaradas por letras seguidas de letras e números. Os tipos disponíveis em LP, são, char, bool, int, real e string, sendo todos esses passíveis de se tornarem arranjos, bastando colocar na frente de sua variável o valor desejado do arranjo envolvido de colchetes.

Não é permitido a declaração de funções e structs como na linguagem C.

Se trata de uma linguagem não case-sensitive (indiferente a caixa alta e baixa);

Exemplo de um código fonte:

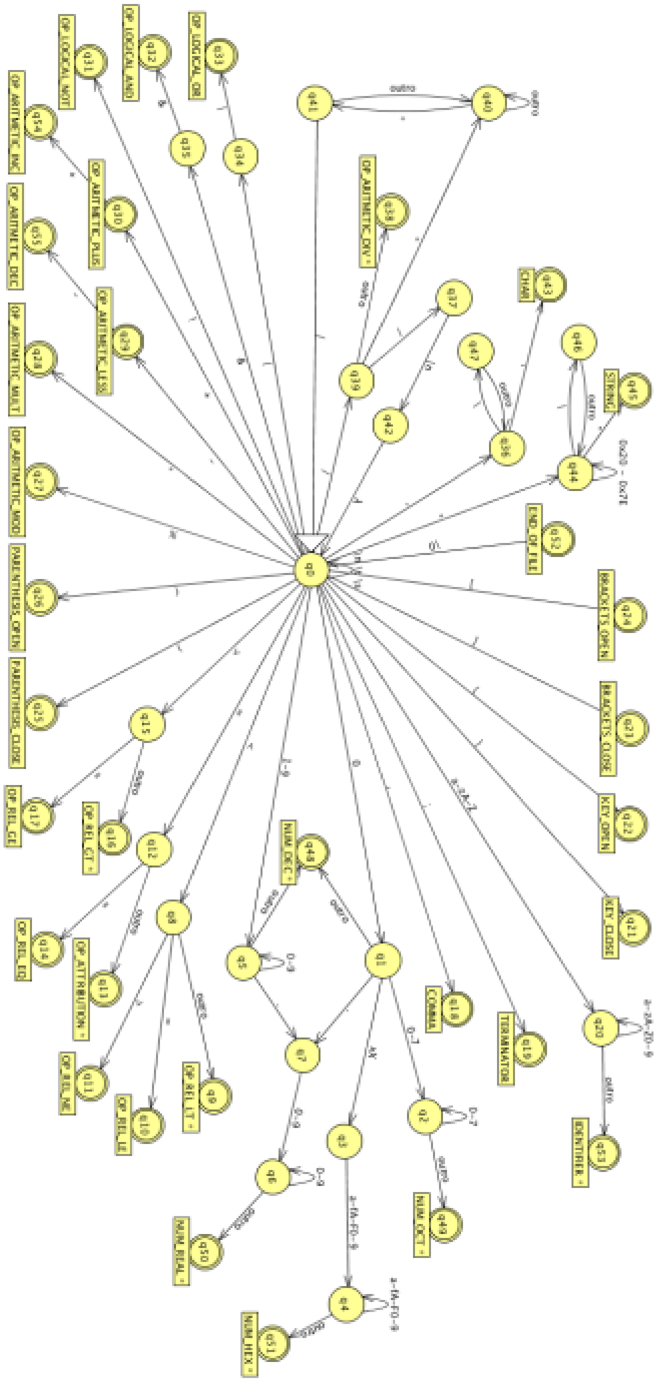
```
int variavel = 123;
if (variavel == 123) {
     print(“Valor da variável: %d”, variavel);
}
```

Definição regular:

num_dec	[0-9]+
num_hex	0x[0-9|A-Fa-f]+
num_real	[0-9]+.[0-9]+
num_octal	0[0-9]+
string	“[0x20 – 0x7E]*”
comma	,
op_rel_lt	<
op_rel_le	<=
op_rel_ne	<>
op_rel_gt	>
op_rel_ge	>=
op_rel_eq	==
op_logical_and	&&
op_logical_or	||
op_logical_not	!
op_aritmetic_plus	+
op_aritmetic_less	-
op_aritmetic_mult	*
op_aritmetic_div	/
op_aritmetic_mod	%
op_aritmetic_inc	++
op_aritmetic_dec	--
op_attribution	=
terminator	;
key_open	{
key_close	}
parenthesis_open	(
parenthesis_close	)
brackets_open	[
brackets_close	]
true	true
false	false
if	if
else	else
while	while
break	break
for	for
type_char	char
type_int	int
type_real	real
type_bool	bool
identifier	[a-zA-Z]([0-9]|[a-zA-Z])*
print	print

read	read

Autômato Finito:



Gramática Livre de Contexto:

PROG	{ CMD }*
TYPE	**type_char | type_int | type_real | type_bool**
VAR	**identifier** [**key_open (num_dec |** EXP**)** **key_close**]
DECLARATION	TYPE VAR DECLARATION_VAR | DECLARATION_ARRAY **terminator**
DECLARATION_VAR	[**op_attribution** EXP]

DECLARATION_ARRAY	**brackets_open num_dec brackets_close**
	[
	    op_attribution
	    key_open
	        [EXP] { **comma** EXP }*
	    key_close
	]

ATTRIBUTION	VAR **op_attribution** EXP **terminator**
IF	**if parenthesis_open** EXP **parenthesis_close** CMD [**else** CMD]
WHILE	**while parenthesis_open** EXP **parenthesis_close key_open** CMD [**break**] **key_close**
FOR	**for parenthesis_open** [ATTRIBUTION] **comma** [EXP } **comma** [EXP] **parenthesis_close** CMD
CMD	IF | WHILE | FOR | PRINT | READ | DECLARATION | ATTRIBUTION | BLOCK
BLOCK	**key_open** { CMD }* **key_close**
EXP	EXPS [OP_REL EXPS]
EXPS	TERM { OP_ADD TERM }*
TERM	FACTOR { OP_MUL FACTOR }*
FACTOR	**parenthesis_open** EXP **parenthesis_close** | **op_logical_not** FACTOR | VAR | **num_dec** | **num_hex** | **num real** | **num_octal** | **true** | **false** | **string**
OP_REL	**op_rel_lt** | **op_rel_le** | **op_rel_ne** | **op_rel_gt** | **op_rel_ge** | **op_rel_eq**
OP_ADD	**op_aritmetic_plus** | **op_aritmetic_less** | **op_aritmetic_inc** | **op_aritmetic_dec** | **op_logical_or**
OP_MUL	**op_aritmetic_mult** | **op_aritmetic_div** | **op_aritmetic_mod** | **op_logical_and** | **op_aritmetic_not**
PRINT	**print parenthesis_open string** [{ **comma** FACTOR }*]**parenthesis_close terminator**
READ	**read parenthesis_open string** [{ **comma** FACTOR }*] **parenthesis_close terminator**

Tabela de erros:

Léxico	Comment without end (fatal error)
Léxico	String without end (fatal error)
Léxico	Char without end (fatal error)
Léxico	Unknown symbol: X line: Y, position: T
Léxico	Unexpected symbol: X line: Y, position: T
Sintático	Unexpected symbol: X, line: Y, position: T
Sintático	Unexpected symbol: X, line: Y, position: T, Expected: P
Sintático	Source not detected!
Sintático	Source getting end of file unexpected way... your code have a problem!

Exemplos:

Programa 01:

```
/**
 * Programa com o objetivo de verificar se
 * o aluno é maior de idade
 *
 * Erro: String começa e não termina
 */

// Variável que possui a idade do aluno
int idadeAluno = 0;
// Idade mínima para ser maior de idade
int idadeMinimaRequerida = 18;

// Leia um inteiro para variável idadeAluno
read("%d", idadeAluno);

// Compara e imprime o resultado
if(idadeAluno > idadeMinimaRequerida) {
  print("O aluno é maior de idade");
} else {
  print("O aluno não é maior de idade);
}
```

Saida do compilador:

Erros:

```
Lexical Error: String without end (fatal error)
```

Análise léxica:

```
<TYPE_INT,"int">
<IDENTIFIER,"idadeAluno">
<OP_ATTRIBUTION,"=">
<NUM_DEC,"0">
<TERMINATOR,";">
<TYPE_INT,"int">
<IDENTIFIER,"idadeMinimaRequerida">
<OP_ATTRIBUTION,"=">
<NUM_DEC,"18">
<TERMINATOR,";">
<READ,"read">
<PARENTHESIS_OPEN,"(">
<STRING,""%d"">
<COMMA,",">
```

```
<IDENTIFIER,"idadeAluno">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<IF,"if">
<PARENTHESIS_OPEN,"(">
<IDENTIFIER,"idadeAluno">
<OP_REL_GT,">">
<IDENTIFIER,"idadeMinimaRequerida">
<PARENTHESIS_CLOSE,")">
<KEY_OPEN,"{">
<PRINT,"print">
<PARENTHESIS_OPEN,"(">
<STRING,""O aluno é maior de idade"">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<KEY_CLOSE,"}">
<ELSE,"else">
<KEY_OPEN,"{">
<PRINT,"print">
<PARENTHESIS_OPEN,"(">
```

Tabela de Símbolos:

```
<IDENTIFIER, "idadeMinimaRequerida", "VARIABLE", "TYPE_INT">
<TYPE_REAL, "real", "NULL", "NULL">
<FOR, "for", "NULL", "NULL">
<TYPE_BOOL, "bool", "NULL", "NULL">
<IF, "if", "NULL", "NULL">
<WHILE, "while", "NULL", "NULL">
<FALSE, "false", "NULL", "NULL">
<BREAK, "break", "NULL", "NULL">
<READ, "read", "NULL", "NULL">
<TYPE_INT, "int", "NULL", "NULL">
<PRINT, "print", "NULL", "NULL">
<ELSE, "else", "NULL", "NULL">
<TYPE_CHAR, "char", "NULL", "NULL">
<IDENTIFIER, "idadeAluno", "VARIABLE", "TYPE_INT">
<TRUE, "true", "NULL", "NULL">
```

Programa 02:

```
/**
 * Programa com o objetivo de verificar se
 * o aluno foi aprovado, reprovado ou está
 * de recuperação
 *
 * Erro: Comentário que começa e não termina
 */

// Variável que possui a idade do aluno
int nota = 0;

// Leia um inteiro para variável nota
read("%d", nota);

/**
 * Verifica agora a nota do aluno
 * E depois imprime o resultado
if(nota > 60) {
  print("O aluno foi aprovado");
} if (nota > 40) {
  print("O aluno está de recuperação");
} else {
  print("O aluno foi reprovado");
}
```

Saida do compilador:

Erros:

```
Lexical Error: Comment without end (fatal error)
```

Análise léxica:

```
<TYPE_INT,"int">
<IDENTIFIER,"nota">
<OP_ATTRIBUTION,"=">
<NUM_DEC,"0">
<TERMINATOR,";">
<READ,"read">
<PARENTHESIS_OPEN,"(">
<STRING,""%d"">
<COMMA,",">
<IDENTIFIER,"nota">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
```

Tabela de Símbolos:

```
<FALSE, "false", "NULL", "NULL">
<TYPE_BOOL, "bool", "NULL", "NULL">
<TYPE_INT, "int", "NULL", "NULL">
<IF, "if", "NULL", "NULL">
<TYPE_CHAR, "char", "NULL", "NULL">
<TRUE, "true", "NULL", "NULL">
<BREAK, "break", "NULL", "NULL">
<FOR, "for", "NULL", "NULL">
<TYPE_REAL, "real", "NULL", "NULL">
<WHILE, "while", "NULL", "NULL">
<IDENTIFIER, "nota", "VARIABLE", "TYPE_INT">
<PRINT, "print", "NULL", "NULL">
<READ, "read", "NULL", "NULL">
<ELSE, "else", "NULL", "NULL">
```

Programa 03:

```
/**
 * Programa com o objetivo de calcular a média
 * de numeros de um vetor
 *
 * Erro: Simbolo não identificado
 */

// Notas (primeira prova, segunda prova, terceira prova, trabalho)
int nota[4];

print("Informe a nota da prova 1:\n");
read("%d", nota[0]);
print("Informe a nota da prova 2:\n");
read("%d", nota[1]);
print("Informe a nota da prova 3:\n");
read("%d", nota[2]);
print("Informe a nota do trabalho:\n");
read("%d", nota[3]);

!@#$%^

real media = nota[0] + nota[1] + nota[2] + nota[3];
```

```
media = media / 4;

print("Média calculada e arredondada: %d", media);
```

Saida do compilador:

Erros:

```
Lexical Error: Unknown symbol: @, line: 14, position: 1
Lexical Error: Unknown symbol: #, line: 14, position: 2
Lexical Error: Unknown symbol: $, line: 14, position: 3
Lexical Error: Unknown symbol: ˆ, line: 14, position: 5
Syntactic Error: Unexpected symbol: !, line: 12, position: 20
Syntactic Error: Unexpected symbol: %, line: 14, position: 1
```

Análise léxica:

```
<TYPE_INT,"int">
<IDENTIFIER,"nota">
<BRACKETS_OPEN,"[">
<NUM_DEC,"4">
<BRACKETS_CLOSE,"]">
<TERMINATOR,";">
<PRINT,"print">
<PARENTHESIS_OPEN,"(">
<STRING,""Informe a nota da prova 1:\n"">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<READ,"read">
<PARENTHESIS_OPEN,"(">
<STRING,""%d"">
<COMMA,",">
<IDENTIFIER,"nota">
<BRACKETS_OPEN,"[">
<NUM_DEC,"0">
<BRACKETS_CLOSE,"]">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<PRINT,"print">
<PARENTHESIS_OPEN,"(">
<STRING,""Informe a nota da prova 2:\n"">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<READ,"read">
<PARENTHESIS_OPEN,"(">
<STRING,""%d"">
<COMMA,",">
<IDENTIFIER,"nota">
<BRACKETS_OPEN,"[">
<NUM_DEC,"1">
<BRACKETS_CLOSE,"]">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<PRINT,"print">
<PARENTHESIS_OPEN,"(">
<STRING,""Informe a nota da prova 3:\n"">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<READ,"read">
<PARENTHESIS_OPEN,"(">
<STRING,""%d"">
<COMMA,",">
<IDENTIFIER,"nota">
<BRACKETS_OPEN,"[">
<NUM_DEC,"2">
<BRACKETS_CLOSE,"]">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<PRINT,"print">
<PARENTHESIS_OPEN,"(">
<STRING,""Informe a nota do trabalho:\n"">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<READ,"read">
<PARENTHESIS_OPEN,"(">
<STRING,""%d"">
<COMMA,",">
<IDENTIFIER,"nota">
<BRACKETS_OPEN,"[">
<NUM_DEC,"3">
<BRACKETS_CLOSE,"]">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<OP_LOGICAL_NOT,"!">
<OP_ARITMETIC_MOD,"%">
<TYPE_REAL,"real">
<IDENTIFIER,"media">
<OP_ATTRIBUTION,"=">
<IDENTIFIER,"nota">
<BRACKETS_OPEN,"[">
<NUM_DEC,"0">
<BRACKETS_CLOSE,"]">
<OP_ARITMETIC_PLUS,"+">
<IDENTIFIER,"nota">
<BRACKETS_OPEN,"[">
<NUM_DEC,"1">
<BRACKETS_CLOSE,"]">
<OP_ARITMETIC_PLUS,"+">
<IDENTIFIER,"nota">
<BRACKETS_OPEN,"[">
<NUM_DEC,"2">
<BRACKETS_CLOSE,"]">
<OP_ARITMETIC_PLUS,"+">
<IDENTIFIER,"nota">
<BRACKETS_OPEN,"[">
<NUM_DEC,"3">
<BRACKETS_CLOSE,"]">
<TERMINATOR,";">
<IDENTIFIER,"media">
<OP_ATTRIBUTION,"=">
<IDENTIFIER,"media">
<OP_ARITMETIC_DIV,"/">
<NUM_DEC,"4">
<TERMINATOR,";">
<PRINT,"print">
<PARENTHESIS_OPEN,"(">
<STRING,""Média calculada e arredondada: %d"">
<COMMA,",">
<IDENTIFIER,"media">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<END_OF_FILE,"">
```

Tabela de Símbolos:

```
<IF, "if", "NULL", "NULL">
<TYPE_BOOL, "bool", "NULL", "NULL">
<TYPE_CHAR, "char", "NULL", "NULL">
<FALSE, "false", "NULL", "NULL">
```

```
<IDENTIFIER, "media", "VARIABLE", "TYPE_REAL">
<READ, "read", "NULL", "NULL">
<ELSE, "else", "NULL", "NULL">
<TYPE_REAL, "real", "NULL", "NULL">
<TRUE, "true", "NULL", "NULL">
<TYPE_INT, "int", "NULL", "NULL">
<IDENTIFIER, "nota", "ARRAY", "TYPE_INT">
<FOR, "for", "NULL", "NULL">
<PRINT, "print", "NULL", "NULL">
<BREAK, "break", "NULL", "NULL">
<WHILE, "while", "NULL", "NULL">
```

Programa 04:

```
/**
 * Programa com o objetivo de somar a média
 * dos parametros e liberar o sistema caso a mesma
 * seja maior que: 5
 *
 * Nesse exemplo o modo pânico do analisador
 * é claramente visivel!
 *
 * Erro: Simbolo não identificado
 * Simbolo não esperado
 */

bool ativarSistema = false;
bool flagComAlgumaLogica = true;

int parametros[5];
parametros[0] = 10;
parametros[1] = 3;
parametros[2] = 5;
parametros[3] = 7;
parametros[4] = 1;

real media = ((parametros[0] + parametros[1] + parametros[2] + parametros[3] +
parametros[4]) / 5);

if(media >= 5 &SIMBOLOS_NAO_ESPERADOS& flagComAlgumaLogica == true) {
  ativarSistema = true;
}

if(ativarSistema == true) {
  print("Sistema liberado");
}
else {
  print("Sistema não liberado");
}
```

Saida do compilador:

Erros:

```
Lexical Error: Unexpected symbol: S, line: 15, position: 15
Lexical Error: Unknown symbol: _, line: 15, position: 23
Lexical Error: Unknown symbol: _, line: 15, position: 27
Lexical Error: Unexpected symbol:  , line: 15, position: 38
Syntactic Error: Unexpected symbol: SIMBOLOS, line: 15, position: 13, Expected:
PARENTHESIS_CLOSE
```

```
Syntactic Error: Unexpected symbol: NAO, line: 15, position: 23, Expected:
OP_ATTRIBUTION
Syntactic Error: Unexpected symbol: ESPERADOS, line: 15, position: 27, Expected:
TERMINATOR
Syntactic Error: Unexpected symbol: flagComAlgumaLogica, line: 15, position: 37,
Expected: OP_ATTRIBUTION
Syntactic Error: Unexpected symbol: ), line: 15, position: 66, Expected: TERMINATOR
Syntactic Error: Unexpected symbol: ), line: 15, position: 66
```

Análise léxica:

```
<TYPE_BOOL,"bool">
<IDENTIFIER,"ativarSistema">
<OP_ATTRIBUTION,"=">
<FALSE,"false">
<TERMINATOR,";">
<TYPE_BOOL,"bool">
<IDENTIFIER,"flagComAlgumaLogica">
<OP_ATTRIBUTION,"=">
<TRUE,"true">
<TERMINATOR,";">
<TYPE_INT,"int">
<IDENTIFIER,"parametros">
<BRACKETS_OPEN,"[">
<NUM_DEC,"5">
<BRACKETS_CLOSE,"]">
<TERMINATOR,";">
<IDENTIFIER,"parametros">
<BRACKETS_OPEN,"[">
<NUM_DEC,"0">
<BRACKETS_CLOSE,"]">
<OP_ATTRIBUTION,"=">
<NUM_DEC,"10">
<TERMINATOR,";">
<IDENTIFIER,"parametros">
<BRACKETS_OPEN,"[">
<NUM_DEC,"1">
<BRACKETS_CLOSE,"]">
<OP_ATTRIBUTION,"=">
<NUM_DEC,"3">
<TERMINATOR,";">
<IDENTIFIER,"parametros">
<BRACKETS_OPEN,"[">
<NUM_DEC,"2">
<BRACKETS_CLOSE,"]">
<OP_ATTRIBUTION,"=">
<NUM_DEC,"5">
<TERMINATOR,";">
<IDENTIFIER,"parametros">
<BRACKETS_OPEN,"[">
<NUM_DEC,"3">
<BRACKETS_CLOSE,"]">
<OP_ATTRIBUTION,"=">
<NUM_DEC,"7">
<TERMINATOR,";">
<IDENTIFIER,"parametros">
<BRACKETS_OPEN,"[">
<NUM_DEC,"4">
<BRACKETS_CLOSE,"]">
<OP_ATTRIBUTION,"=">
<NUM_DEC,"1">
<TERMINATOR,";">
<TYPE_REAL,"real">
<IDENTIFIER,"media">
```

```
<OP_ATTRIBUTION,"=">
<PARENTHESIS_OPEN,"(">
<PARENTHESIS_OPEN,"(">
<IDENTIFIER,"parametros">
<BRACKETS_OPEN,"[">
<NUM_DEC,"0">
<BRACKETS_CLOSE,"]">
<OP_ARITMETIC_PLUS,"+">
<IDENTIFIER,"parametros">
<BRACKETS_OPEN,"[">
<NUM_DEC,"1">
<BRACKETS_CLOSE,"]">
<OP_ARITMETIC_PLUS,"+">
<IDENTIFIER,"parametros">
<BRACKETS_OPEN,"[">
<NUM_DEC,"2">
<BRACKETS_CLOSE,"]">
<OP_ARITMETIC_PLUS,"+">
<IDENTIFIER,"parametros">
<BRACKETS_OPEN,"[">
<NUM_DEC,"3">
<BRACKETS_CLOSE,"]">
<OP_ARITMETIC_PLUS,"+">
<IDENTIFIER,"parametros">
<BRACKETS_OPEN,"[">
<NUM_DEC,"4">
<BRACKETS_CLOSE,"]">
<PARENTHESIS_CLOSE,")">
<OP_ARITMETIC_DIV,"/">
<NUM_DEC,"5">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<IF,"if">
<PARENTHESIS_OPEN,"(">
<IDENTIFIER,"media">
<OP_REL_GE,">=">
<NUM_DEC,"5">
<IDENTIFIER,"SIMBOLOS">
<IDENTIFIER,"NAO">
<IDENTIFIER,"ESPERADOS">
<IDENTIFIER,"flagComAlgumaLogica">
<OP_REL_EQ,"==">
<TRUE,"true">
<PARENTHESIS_CLOSE,")">
<KEY_OPEN,"{">
<IDENTIFIER,"ativarSistema">
<OP_ATTRIBUTION,"=">
<TRUE,"true">
<TERMINATOR,";">
<KEY_CLOSE,"}">
<IF,"if">
<PARENTHESIS_OPEN,"(">
<IDENTIFIER,"ativarSistema">
<OP_REL_EQ,"==">
<TRUE,"true">
<PARENTHESIS_CLOSE,")">
<KEY_OPEN,"{">
<PRINT,"print">
<PARENTHESIS_OPEN,"(">
<STRING,""Sistema liberado"">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<KEY_CLOSE,"}">
```

```
<ELSE,"else">
<KEY_OPEN,"{">
<PRINT,"print">
<PARENTHESIS_OPEN,"(">
<STRING,""Sistema não liberado"">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<KEY_CLOSE,"}">
<END_OF_FILE,"">
```

Tabela de Símbolos:

```
<TYPE_CHAR, "char", "NULL", "NULL">
<READ, "read", "NULL", "NULL">
<IF, "if", "NULL", "NULL">
<IDENTIFIER, "NAO", "VARIABLE", "NULL">
<TYPE_BOOL, "bool", "NULL", "NULL">
<IDENTIFIER, "flagComAlgumaLogica", "VARIABLE", "TYPE_BOOL">
<FOR, "for", "NULL", "NULL">
<TYPE_INT, "int", "NULL", "NULL">
<TRUE, "true", "NULL", "NULL">
<BREAK, "break", "NULL", "NULL">
<IDENTIFIER, "SIMBOLOS", "VARIABLE", "NULL">
<IDENTIFIER, "ESPERADOS", "VARIABLE", "NULL">
<IDENTIFIER, "ativarSistema", "VARIABLE", "TYPE_BOOL">
<WHILE, "while", "NULL", "NULL">
<TYPE_REAL, "real", "NULL", "NULL">
<IDENTIFIER, "media", "VARIABLE", "TYPE_REAL">
<PRINT, "print", "NULL", "NULL">
<FALSE, "false", "NULL", "NULL">
<IDENTIFIER, "parametros", "ARRAY", "TYPE_INT">
<ELSE, "else", "NULL", "NULL">
```

Programa 05:

```
/**
 * Programa com o objetivo de preencher
 * uma pesquisa de pessoas e depois imprimir
 * o relatório final.
 *
 * Erro: --
 */

char nomes[10];
bool genero[10];
int idade[10];

int i = 0;
for(i = 0; i < 10; i++) {
  print("Digite o nome:\n");
  read("%s", nomes[i]);

  print("Genero (0 - Masculino / 1 - Feminino):\n");
  read("%b", genero[i]);

  print("Digite a idade:\n");
  read("%d", idade[i]);
}

for(i = 0; i < 10; i++) {
  char cGenero;
  if (genero[i] == true) {
    cGenero = "M";
  } else {
    cGenero = "F";
  }
  print("Nome: %s, Genero: %s, Idade: %d\n", nomes[i], cGenero, idade[i]);
}
```

Saida do compilador:

Erros:

--

Análise léxica:

```
<TYPE_CHAR,"char">
<IDENTIFIER,"nomes">
<BRACKETS_OPEN,"[">
<NUM_DEC,"10">
<BRACKETS_CLOSE,"]">
<TERMINATOR,";">
<TYPE_BOOL,"bool">
<IDENTIFIER,"genero">
<BRACKETS_OPEN,"[">
<NUM_DEC,"10">
<BRACKETS_CLOSE,"]">
<TERMINATOR,";">
<TYPE_INT,"int">
<IDENTIFIER,"idade">
<BRACKETS_OPEN,"[">
<NUM_DEC,"10">
<BRACKETS_CLOSE,"]">
<TERMINATOR,";">
<TYPE_INT,"int">
<IDENTIFIER,"i">
<OP_ATTRIBUTION,"=">
<NUM_DEC,"0">
<TERMINATOR,";">
<FOR,"for">
<PARENTHESIS_OPEN,"(">
<IDENTIFIER,"i">
<OP_ATTRIBUTION,"=">
<NUM_DEC,"0">
<TERMINATOR,";">
<IDENTIFIER,"i">
<OP_REL_LT,"<">
<NUM_DEC,"10">
<TERMINATOR,";">
<IDENTIFIER,"i">
<OP_ARITMETIC_INC,"++">
<PARENTHESIS_CLOSE,")">
<KEY_OPEN,"{">
<PRINT,"print">
<PARENTHESIS_OPEN,"(">
<STRING,""Digite o nome:\n"">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<READ,"read">
<PARENTHESIS_OPEN,"(">
<STRING,""%s"">
<COMMA,",">
<IDENTIFIER,"nomes">
<BRACKETS_OPEN,"[">
<IDENTIFIER,"i">
<BRACKETS_CLOSE,"]">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<PRINT,"print">
<PARENTHESIS_OPEN,"(">
<STRING,""Genero (0 - Masculino / 1 - Feminino):\n"">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<READ,"read">
<PARENTHESIS_OPEN,"(">
<STRING,""%b"">
<COMMA,",">
<IDENTIFIER,"genero">
<BRACKETS_OPEN,"[">
<IDENTIFIER,"i">
<BRACKETS_CLOSE,"]">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<PRINT,"print">
<PARENTHESIS_OPEN,"(">
<STRING,""Digite a idade:\n"">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<READ,"read">
<PARENTHESIS_OPEN,"(">
<STRING,""%d"">
<COMMA,",">
<IDENTIFIER,"idade">
<BRACKETS_OPEN,"[">
<IDENTIFIER,"i">
<BRACKETS_CLOSE,"]">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<KEY_CLOSE,"}">
<FOR,"for">
<PARENTHESIS_OPEN,"(">
<IDENTIFIER,"i">
<OP_ATTRIBUTION,"=">
<NUM_DEC,"0">
<TERMINATOR,";">
<IDENTIFIER,"i">
<OP_REL_LT,"<">
<NUM_DEC,"10">
<TERMINATOR,";">
<IDENTIFIER,"i">
<OP_ARITMETIC_INC,"++">
<PARENTHESIS_CLOSE,")">
<KEY_OPEN,"{">
<TYPE_CHAR,"char">
<IDENTIFIER,"cGenero">
<TERMINATOR,";">
<IF,"if">
<PARENTHESIS_OPEN,"(">
<IDENTIFIER,"genero">
<BRACKETS_OPEN,"[">
<IDENTIFIER,"i">
<BRACKETS_CLOSE,"]">
<OP_REL_EQ,"==">
<TRUE,"true">
<PARENTHESIS_CLOSE,")">
<KEY_OPEN,"{">
<IDENTIFIER,"cGenero">
<OP_ATTRIBUTION,"=">
```

```
<STRING,""M"">
<TERMINATOR,";">
<KEY_CLOSE,"}">
<ELSE,"else">
<KEY_OPEN,"{">
<IDENTIFIER,"cGenero">
<OP_ATTRIBUTION,"=">
<STRING,""F"">
<TERMINATOR,";">
<KEY_CLOSE,"}">
<PRINT,"print">
<PARENTHESIS_OPEN,"(">
<STRING,""Nome: %s, Genero: %s, Idade: %d\n"">
<COMMA,",">
<IDENTIFIER,"nomes">
<BRACKETS_OPEN,"[">
<IDENTIFIER,"i">
<BRACKETS_CLOSE,"]">
<COMMA,",">
<IDENTIFIER,"cGenero">
<COMMA,",">
<IDENTIFIER,"idade">
<BRACKETS_OPEN,"[">
<IDENTIFIER,"i">
<BRACKETS_CLOSE,"]">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<KEY_CLOSE,"}">
<END_OF_FILE,"">
```

Tabela de Símbolos:

```
<TYPE_INT, "int", "NULL", "NULL">
<IDENTIFIER, "genero", "ARRAY", "TYPE_BOOL">
<TYPE_BOOL, "bool", "NULL", "NULL">
<TRUE, "true", "NULL", "NULL">
<IF, "if", "NULL", "NULL">
<FOR, "for", "NULL", "NULL">
<PRINT, "print", "NULL", "NULL">
<IDENTIFIER, "i", "VARIABLE", "TYPE_INT">
<IDENTIFIER, "idade", "ARRAY", "TYPE_INT">
<READ, "read", "NULL", "NULL">
<FALSE, "false", "NULL", "NULL">
<TYPE_CHAR, "char", "NULL", "NULL">
<IDENTIFIER, "nomes", "ARRAY", "TYPE_CHAR">
<BREAK, "break", "NULL", "NULL">
<ELSE, "else", "NULL", "NULL">
<WHILE, "while", "NULL", "NULL">
<IDENTIFIER, "cGenero", "VARIABLE", "TYPE_CHAR">
<TYPE_REAL, "real", "NULL", "NULL">
```

Programa 06:

```
/**
 * Programa com o objetivo de verificar se
 * o aluno foi aprovado, reprovado ou está
 * de recuperação
 *
 * Erro: Sintaxe errada
 */

// Variável que possui a idade do aluno
```

```
int nota = 0;

// Leia um inteiro para variável nota
read("%d", nota);

/**
 * Verifica agora a nota do aluno
 * E depois imprime o resultado
 */
if(nota > 60)
  print("O aluno foi aprovado");
} if (nota > 40) {
  print("O aluno está de recuperação")
} else {
  print("O aluno foi reprovado",);
}
```

Saida do compilador:

Erros:

```
Syntactic Error: Unexpected symbol: }, line: 9, position: 32
Syntactic Error: Unexpected symbol: }, line: 11, position: 38, Expected: TERMINATOR
Syntactic Error: Unexpected symbol: ), line: 13, position: 32
```

Análise léxica:

```
<TYPE_INT,"int">
<IDENTIFIER,"nota">
<OP_ATTRIBUTION,"=">
<NUM_DEC,"0">
<TERMINATOR,";">
<READ,"read">
<PARENTHESIS_OPEN,"(">
<STRING,""%d"">
<COMMA,",">
<IDENTIFIER,"nota">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<IF,"if">
<PARENTHESIS_OPEN,"(">
<IDENTIFIER,"nota">
<OP_REL_GT,">">
<NUM_DEC,"60">
<PARENTHESIS_CLOSE,")">
<PRINT,"print">
<PARENTHESIS_OPEN,"(">
<STRING,""O aluno foi aprovado"">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<KEY_CLOSE,"}">
<IF,"if">
<PARENTHESIS_OPEN,"(">
<IDENTIFIER,"nota">
<OP_REL_GT,">">
<NUM_DEC,"40">
<PARENTHESIS_CLOSE,")">
<KEY_OPEN,"{">
<PRINT,"print">
<PARENTHESIS_OPEN,"(">
<STRING,""O aluno está de recuperação"">
<PARENTHESIS_CLOSE,")">
<KEY_CLOSE,"}">
<ELSE,"else">
<KEY_OPEN,"{">
```

```
<PRINT,"print">
<PARENTHESIS_OPEN,"(">
<STRING,""O aluno foi reprovado"">
<COMMA,",">
<PARENTHESIS_CLOSE,")">
<TERMINATOR,";">
<KEY_CLOSE,"}">
<END_OF_FILE,"">
```

Tabela de Símbolos:

```
<IDENTIFIER, "nota", "VARIABLE", "TYPE_INT">
<TRUE, "true", "NULL", "NULL">
<TYPE_CHAR, "char", "NULL", "NULL">
<TYPE_BOOL, "bool", "NULL", "NULL">
<BREAK, "break", "NULL", "NULL">
<TYPE_REAL, "real", "NULL", "NULL">
<READ, "read", "NULL", "NULL">
<PRINT, "print", "NULL", "NULL">
<TYPE_INT, "int", "NULL", "NULL">
<FALSE, "false", "NULL", "NULL">
<IF, "if", "NULL", "NULL">
<FOR, "for", "NULL", "NULL">
<WHILE, "while", "NULL", "NULL">
<ELSE, "else", "NULL", "NULL">
```

Código Fonte:

File: VariableClass.java

```
public enum VariableClass {
    NULL,
    VARIABLE,
    ARRAY
}
```

File: VariableType.java

```
public enum VariableType {
    NULL,
    TYPE_CHAR,
    TYPE_INT,
    TYPE_REAL,
    TYPE_BOOL
}
```

File: LexemeType.java

```
package compiler.lexical;

import java.lang.reflect.Field;
import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

public class LexemeType {
    public static final int END_OF_FILE             = 0xFFFFFFFF;
    public static final int NUM_DEC                 = 0x00000100;
    public static final int NUM_HEX                 = 0x00000200;
    public static final int NUM_REAL                = 0x00000300;
    public static final int NUM_OCT                 = 0x00000400;
    public static final int STRING                  = 0x00000500;
    public static final int COMMA                   = 0x00000600;
    public static final int OP_REL_LT               = 0x00000700;
    public static final int OP_REL_LE               = 0x00000800;
    public static final int OP_REL_NE               = 0x00000900;
    public static final int OP_REL_GT               = 0x00000A00;
    public static final int OP_REL_GE               = 0x00000B00;
    public static final int OP_REL_EQ               = 0x00000C00;
    public static final int OP_LOGICAL_AND          = 0x00000D00;
    public static final int OP_LOGICAL_OR           = 0x00000E00;
    public static final int OP_LOGICAL_NOT          = 0x00000F00;
    public static final int OP_ARITMETIC_PLUS       = 0x00001000;
    public static final int OP_ARITMETIC_LESS       = 0x00001200;
    public static final int OP_ARITMETIC_MULT       = 0x00001300;
    public static final int OP_ARITMETIC_DIV        = 0x00001400;
    public static final int OP_ARITMETIC_MOD        = 0x00001500;
    public static final int OP_ATTRIBUTION          = 0x00001600;
    public static final int TERMINATOR              = 0x00001700;
    public static final int KEY_OPEN                = 0x00001800;
    public static final int KEY_CLOSE               = 0x00001900;
    public static final int PARENTHESIS_OPEN        = 0x00001A00;
    public static final int PARENTHESIS_CLOSE       = 0x00001B00;
    public static final int BRACKETS_OPEN           = 0x00001C00;
    public static final int BRACKETS_CLOSE          = 0x00001D00;
    public static final int TRUE                    = 0x00001E00;
    public static final int FALSE                   = 0x00001F00;
    public static final int IF                      = 0x00002000;
    public static final int ELSE                    = 0x00002100;
    public static final int WHILE                   = 0x00002300;
    public static final int BREAK                   = 0x00002400;
    public static final int FOR                     = 0x00002500;
    public static final int TYPE_CHAR               = 0x00002600;
    public static final int TYPE_INT                = 0x00002700;
    public static final int TYPE_REAL               = 0x00002800;
    public static final int TYPE_BOOL               = 0x00002900;
    public static final int IDENTIFIER              = 0x00002A00;
    public static final int PRINT                   = 0x00002B00;
    public static final int READ                    = 0x00002C00;
    public static final int CHAR                    = 0x00002D00;
    public static final int OP_ARITMETIC_INC        = 0x00002F00;
    public static final int OP_ARITMETIC_DEC        = 0x00003000;

    /**
     * Mapa dos nomes dos tipos (runtime)
     */
    private static Map<Integer, String> FriendlyTypes;

    /**
     * Obtem o nome do tipo
     * @param type Número do tipo
     * @return Nome do tipo (Nome da variavel)
     */
    public static String getTypeName(int type) {
        return FriendlyTypes.get(type);
    }

    /**
     * Executa em tempo de execução utilizando reflection;
     * Obtem as variáveis da classe LexemeType, e armazena em um HashMap
     * para que outras partes do programa obtenha o nome do tipo.
     */
    static {
        FriendlyTypes = new HashMap<Integer, String>();
```

```
        Field fieldCollection[] = LexemeType.class.getDeclaredFields();
        for (int i = 0; i < fieldCollection.length; i++) {
            Integer test = new Integer(0);
            if (fieldCollection[i].getName().equals("FriendlyTypes") == false) {
                try {
                    FriendlyTypes.put(fieldCollection[i].getInt(test),
fieldCollection[i].getName());
                } catch (IllegalArgumentException ex) {
                    Logger.getLogger(Lexeme.class.getName()).log(Level.SEVERE, null,
ex);
                } catch (IllegalAccessException ex) {
                    Logger.getLogger(Lexeme.class.getName()).log(Level.SEVERE, null,
ex);
                }
            }
        }

    }
}
```

File: Lexeme.java

```
package compiler.lexical;

import java.lang.reflect.Field;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Lexeme {

    private int type;
    private String lexeme;
    private int sourceLine;
    private int sourceColumn;

    private VariableClass variableClass;
    private VariableType variableType;

    public Lexeme() {
        this.lexeme = "";
        this.variableClass = VariableClass.NULL;
        this.variableType = VariableType.NULL;
    }

    public Lexeme(String lexeme, int type) {
        this.lexeme = lexeme;
        this.type = type;
        this.variableClass = VariableClass.NULL;
        this.variableType = VariableType.NULL;
    }

    public Lexeme setType(int type) {
        this.type = type;
        return this;
    }

    public int getType() {
        return this.type;
    }

    public String getLexeme() {
        return this.lexeme;
    }

    public void appendLexeme(char character) {
        this.lexeme += character;
    }

    public Lexeme removeLastChar() {
        this.lexeme = this.lexeme.substring(0, this.lexeme.length() - 1);
        return this;
    }

    public String getTypeString() {
        return LexemeType.getTypeName(this.type);
    }

    public int getSourceLine() {
        return sourceLine;
    }

    public void setSourceLine(int sourceLine) {
        this.sourceLine = sourceLine;
    }

    public int getSourceColumn() {
        return sourceColumn;
    }

    public void setSourceColumn(int sourceColumn) {
        this.sourceColumn = sourceColumn;
    }

    public VariableClass getVariableClass() {
        return variableClass;
    }

    public void setVariableClass(VariableClass variableClass) {
        this.variableClass = variableClass;
    }

    public VariableType getVariableType() {
        return variableType;
    }

    public void setVariableType(VariableType variableType) {
        this.variableType = variableType;
    }
}
```

File: Lexical.java

```
package compiler.lexical;

import compiler.SymbolTable;
import java.util.ArrayList;
import java.util.List;

public class Lexical {

    /**
     * Lista de erros
     */
    private List<String> errors;
```

```java
public List<String> getErrors() {
    return this.errors;
}

private List<String> output;

public List<String> getOutput() {
    return this.output;
}

/**
 * Tabela de simbolos
 */
private SymbolTable symbolTable;

/**
 * Código fonte
 */
private String sourceCode;

/**
 * Posição da leitura no código fonte
 */
private int sourceOffsetPointer;

/**
 * Posição de qual linha está atualmente
 */
private int sourceOffsetLinePointer;

/**
 * Posição do cursor na linha
 */
private int sourceOffsetLinePositionPointer;

/**
 * Estado do automato finito
 */
private int finiteState = 0;

/**
 * Construtor
 *
 * @param sourceCode Codigo fonte
 */
public Lexical(SymbolTable symbolTable, String sourceCode) {
    this.symbolTable = symbolTable;
    this.sourceCode = sourceCode;
    this.sourceCode += '\0';
    this.sourceOffsetPointer = 0;
    this.sourceOffsetLinePointer = 0;
    this.sourceOffsetLinePositionPointer = 0;
    this.errors = new ArrayList<String>();
    this.output = new ArrayList<String>();
}

/**
 * Levanta um erro de declaração de comentário sem termino
 *
 * @param c Simbolo
 */
private void raiseErrorCommentWithoutEnd() {
        this.errors.add("Comment without end (fatal error)\n");
    }

    /**
     * Levanta um erro de declaração de string sem termino
     *
     * @param c Simbolo
     */
    private void raiseErrorStringWithoutEnd() {
        this.errors.add("String without end (fatal error)\n");
    }

    /**
     * Levanta um erro de declaração de char sem termino
     *
     * @param c Simbolo
     */
    private void raiseErrorCharWithoutEnd() {
        this.errors.add("Char without end (fatal error)\n");
    }

    /**
     * Levanta um erro de simbolo não reconhecido
     *
     * @param c Simbolo
     */
    private void raiseErrorUnknownSymbol(char c) {
        this.errors.add("Unknown symbol: " + c + ", line: "
                + (this.sourceOffsetLinePointer + 1) + ", position: " +
(this.sourceOffsetLinePositionPointer - 1) + "\n");
    }

    /**
     * Levanta um erro de simbolo não esperado
     *
     * @param c Simbolo
     */
    private void raiseErrorUnexpectedSymbol(char c) {
        this.errors.add("Unexpected symbol: " + c + ", line: "
                + (this.sourceOffsetLinePointer + 1) + ", position: " +
(this.sourceOffsetLinePositionPointer - 1) + "\n");
    }

    /**
     * Adiciona uma mensagem no output do analisador
     *
     * @param message Mensagem
     */
    private void addOutput(String message) {
        this.output.add(message);
    }

    /**
     * Retorna um proximo token do codigo fonte
     *
     * @return Lexema
     */
    public Lexeme getToken() {
        Lexeme lexeme = this.getInternalToken();
        if (lexeme != null) {
            this.addOutput(String.format("<%s,\"%s\">\n", lexeme.getTypeString(),
lexeme.getLexeme()));
        }
        return lexeme;
    }

    /**
     * Incrementa uma posição do carro no codigo fonte
     *
     * @return Posição global no código fonte
     */
    private int nextSourceOffsetPointer() {
        this.sourceOffsetLinePositionPointer++;
        return this.sourceOffsetPointer++;
    }

    /**
     * Decrementa uma posição do carro no código fonte
     *
     * @return Posição global no código fonte
     */
    private int backSourceOffsetPointer() {
        this.sourceOffsetLinePositionPointer--;
        return this.sourceOffsetPointer--;
    }

    /**
     * Obtem o proximo char do codigo fonte
     *
     * @return caractere a ser tratado
     */
    private char getNextChar() {
        return this.sourceCode.charAt(this.nextSourceOffsetPointer());
    }

    /**
     * Retorna um proximo token do codigo fonte
     *
     * @return Lexema
     */
    private Lexeme getInternalToken() {
        this.finiteState = 0;
        Lexeme lexeme = new Lexeme();
        lexeme.setSourceLine(this.sourceOffsetLinePointer);
        lexeme.setSourceColumn(this.sourceOffsetLinePositionPointer);
        while (this.sourceOffsetPointer < this.sourceCode.length()) {
            char currentChar = this.getNextChar();
            lexeme.appendLexeme(currentChar);
            //System.out.print(currentChar);

            switch (this.finiteState) {
                case 0:
                    if (currentChar == '\0') {
                        return
lexeme.removeLastChar().setType(LexemeType.END_OF_FILE);
                    }
                    if (currentChar == ' ' || currentChar == '\t') {
                        lexeme.removeLastChar();
                        this.finiteState = 0;
                    } else if (currentChar == '\n') {
                        this.sourceOffsetLinePointer++;
                        this.sourceOffsetLinePositionPointer = 0;
                        lexeme.removeLastChar();
                        this.finiteState = 0;
                    } else if (currentChar == '0') {
                        this.finiteState = LexemeType.NUM_DEC;
                    } else if (currentChar >= '1' && currentChar <= '9') {
                        this.finiteState = LexemeType.NUM_DEC + 1;
                    } else if ((currentChar >= 'a' && currentChar <= 'z') ||
currentChar >= 'A' && currentChar <= 'Z') {
                        this.finiteState = LexemeType.IDENTIFIER;
                    } else if (currentChar == ';') {
                        return lexeme.setType(LexemeType.TERMINATOR);
                    } else if (currentChar == ',') {
                        return lexeme.setType(LexemeType.COMMA);
                    } else if (currentChar == '{') {
                        return lexeme.setType(LexemeType.KEY_OPEN);
                    } else if (currentChar == '}') {
                        return lexeme.setType(LexemeType.KEY_CLOSE);
                    } else if (currentChar == '[') {
                        return lexeme.setType(LexemeType.BRACKETS_OPEN);
                    } else if (currentChar == ']') {
                        return lexeme.setType(LexemeType.BRACKETS_CLOSE);
                    } else if (currentChar == '(') {
                        return lexeme.setType(LexemeType.PARENTHESIS_OPEN);
                    } else if (currentChar == ')') {
                        return lexeme.setType(LexemeType.PARENTHESIS_CLOSE);
                    } else if (currentChar == '+') {
                        this.finiteState = LexemeType.OP_ARITMETIC_PLUS;
                    } else if (currentChar == '-') {
                        this.finiteState = LexemeType.OP_ARITMETIC_LESS;
                    } else if (currentChar == '*') {
                        return lexeme.setType(LexemeType.OP_ARITMETIC_MULT);
                    } else if (currentChar == '%') {
                        return lexeme.setType(LexemeType.OP_ARITMETIC_MOD);
                    } else if (currentChar == '/') {
                        this.finiteState = LexemeType.OP_ARITMETIC_DIV;
                    } else if (currentChar == '|') {
                        this.finiteState = LexemeType.OP_LOGICAL_OR;
                    } else if (currentChar == '&') {
                        this.finiteState = LexemeType.OP_LOGICAL_AND;
                    } else if (currentChar == '!') {
                        return lexeme.setType(LexemeType.OP_LOGICAL_NOT);
                    } else if (currentChar == '<') {
                        this.finiteState = LexemeType.OP_REL_LT;
                    } else if (currentChar == '=') {
                        this.finiteState = LexemeType.OP_ATTRIBUTION;
                    } else if (currentChar == '>') {
                        this.finiteState = LexemeType.OP_REL_GT;
                    } else if (currentChar == '"') {
                        this.finiteState = LexemeType.STRING;
                    } else if (currentChar == '\'') {
                        this.finiteState = LexemeType.CHAR;
                    } else {
                        lexeme.removeLastChar();
                        this.raiseErrorUnknownSymbol(currentChar);
                    }
                    break;
                case LexemeType.NUM_DEC:
                    if (currentChar >= '0' && currentChar <= '7') {
                        this.finiteState = LexemeType.NUM_OCT;
                    } else if (currentChar == '.') {
                        this.finiteState = LexemeType.NUM_REAL;
                    } else if (currentChar == 'X' || currentChar == 'x') {
                        this.finiteState = LexemeType.NUM_HEX;
                    } else {
                        this.backSourceOffsetPointer();
                        return lexeme.removeLastChar().setType(LexemeType.NUM_DEC);
                    }
                    break;
                case LexemeType.NUM_DEC + 1:
                    if (currentChar >= '0' && currentChar <= '9') {
                        this.finiteState = LexemeType.NUM_DEC + 1; // do not change
state
                    } else if (currentChar == '.') {
                        this.finiteState = LexemeType.NUM_REAL;
                    } else {
                        this.backSourceOffsetPointer();
                        return lexeme.removeLastChar().setType(LexemeType.NUM_DEC);
                    }
                    break;
                case LexemeType.NUM_OCT:
                    if (currentChar >= '0' && currentChar <= '7') {
                        this.finiteState = LexemeType.NUM_OCT; // do not change state
                    } else {
                        this.backSourceOffsetPointer();
                        return lexeme.removeLastChar().setType(LexemeType.NUM_OCT);
                    }
                    break;
                case LexemeType.NUM_REAL:
                    if (currentChar >= '0' && currentChar <= '9') {
                        this.finiteState = LexemeType.NUM_REAL; // do not change state
                    } else {
                        this.backSourceOffsetPointer();
                        return lexeme.removeLastChar().setType(LexemeType.NUM_REAL);
                    }
                    break;
                case LexemeType.NUM_HEX:
                    if (currentChar >= '0' && currentChar <= '9') {
                        this.finiteState = LexemeType.NUM_HEX; // do not change state
                    } else if ((currentChar >= 'A' && currentChar <= 'F') ||
(currentChar >= 'a' && currentChar <= 'f')) {
                        this.finiteState = LexemeType.NUM_HEX; // do not change state
                    } else {
                        this.backSourceOffsetPointer();
                        return lexeme.removeLastChar().setType(LexemeType.NUM_HEX);
                    }
                    break;
                case LexemeType.IDENTIFIER:
                    if ((currentChar >= 'a' && currentChar <= 'z') || currentChar >=
'A' && currentChar <= 'Z') {
                        this.finiteState = LexemeType.IDENTIFIER;
                    } else if (currentChar >= '0' && currentChar <= '9') {
                        this.finiteState = LexemeType.IDENTIFIER;
                    } else {
                        this.backSourceOffsetPointer();
                        lexeme.removeLastChar();
                        int tempLexemeType =
this.symbolTable.resolveLexemeType(lexeme);
                        return lexeme.setType(tempLexemeType);
                    }
                    break;
                case LexemeType.OP_LOGICAL_OR:
                    if (currentChar == '|') {
                        return lexeme.setType(LexemeType.OP_LOGICAL_OR);
                    } else {
                        lexeme.removeLastChar(); // first |
                        lexeme.removeLastChar(); // actual symbol

                        this.raiseErrorUnexpectedSymbol(currentChar);

                        this.backSourceOffsetPointer();
                        this.finiteState = 0; // back to initial state
                    }
                    break;
                case LexemeType.OP_LOGICAL_AND:
                    if (currentChar == '&') {
                        return lexeme.setType(LexemeType.OP_LOGICAL_AND);
                    } else {
                        lexeme.removeLastChar(); // first &
                        lexeme.removeLastChar(); // actual symbol

                        this.raiseErrorUnexpectedSymbol(currentChar);

                        this.backSourceOffsetPointer();
                        this.finiteState = 0; // back to initial state
                    }
                    break;
                case LexemeType.OP_REL_LT:
                    if (currentChar == '=') {
                        return lexeme.setType(LexemeType.OP_REL_LE);
                    } else if (currentChar == '>') {
                        return lexeme.setType(LexemeType.OP_REL_NE);
                    } else {
                        this.backSourceOffsetPointer();
                        return lexeme.removeLastChar().setType(LexemeType.OP_REL_LT);
                    }
                case LexemeType.OP_ARITMETIC_PLUS:
                    if (currentChar == '+') {
                        return lexeme.setType(LexemeType.OP_ARITMETIC_INC);
                    } else {
                        this.backSourceOffsetPointer();
                        return
lexeme.removeLastChar().setType(LexemeType.OP_ARITMETIC_PLUS);
                    }
                case LexemeType.OP_ARITMETIC_LESS:
                    if (currentChar == '+') {
                        return lexeme.setType(LexemeType.OP_ARITMETIC_DEC);
                    } else {
                        this.backSourceOffsetPointer();
                        return
lexeme.removeLastChar().setType(LexemeType.OP_ARITMETIC_LESS);
                    }
                case LexemeType.OP_ATTRIBUTION:
                    if (currentChar == '=') {
                        return lexeme.setType(LexemeType.OP_REL_EQ);
                    } else {
                        this.backSourceOffsetPointer();
                        return
lexeme.removeLastChar().setType(LexemeType.OP_ATTRIBUTION);
                    }
                case LexemeType.OP_REL_GT:
                    if (currentChar == '=') {
                        return lexeme.setType(LexemeType.OP_REL_GE);
                    } else {
                        this.backSourceOffsetPointer();
                        return lexeme.removeLastChar().setType(LexemeType.OP_REL_GT);
                    }
                case LexemeType.OP_ARITMETIC_DIV: // possible comment
                    lexeme.removeLastChar();
                    if (currentChar == '*') {
                        lexeme.removeLastChar();
                        this.finiteState = LexemeType.OP_ARITMETIC_DIV + 1; // comment
block
                    } else if (currentChar == '/') {
                        lexeme.removeLastChar();
                        this.finiteState = LexemeType.OP_ARITMETIC_DIV + 3; // comment
line
                    } else {
                        this.backSourceOffsetPointer();
                        return lexeme.setType(LexemeType.OP_ARITMETIC_DIV); //
operator div
                    }
                    break;
                case LexemeType.OP_ARITMETIC_DIV + 1: // comment block
                    lexeme.removeLastChar();
                    if (currentChar == '*') {
                        this.finiteState = LexemeType.OP_ARITMETIC_DIV + 2;
                    } else if (currentChar == '\0') {
                        this.raiseErrorCommentWithoutEnd();
                    }
                    break;
                case LexemeType.OP_ARITMETIC_DIV + 2: // comment block
                    lexeme.removeLastChar();
                    if (currentChar == '/') {
                        this.finiteState = 0; // begin to initial state
                    } else {
                        this.finiteState = LexemeType.OP_ARITMETIC_DIV + 1;
                    }
                    break;
                case LexemeType.OP_ARITMETIC_DIV + 3: // comment
                    lexeme.removeLastChar();
                    if (currentChar == '\n') {
                        this.finiteState = 0; // begin to initial state
                    }
                    break;
                case LexemeType.STRING:
                    if (currentChar == '\"') {
                        return lexeme.setType(LexemeType.STRING);
                    } else if (currentChar == '\\') {
                        this.finiteState = LexemeType.STRING + 1;
                    } else if (currentChar == '\0') {
                        this.raiseErrorStringWithoutEnd();
                    }
                    break;
                case LexemeType.STRING + 1:
                    this.finiteState = LexemeType.STRING; // just consume char and
back to string state
                    break;
                case LexemeType.CHAR:
                    if (currentChar == '\'') {
                        return lexeme.setType(LexemeType.CHAR);
                    } else if (currentChar == '\\') {
                        this.finiteState = LexemeType.CHAR + 1;
                    } else if (currentChar == '\0') {
                        this.raiseErrorCharWithoutEnd();
                    }
                    break;
                case LexemeType.CHAR + 1:
                    this.finiteState = LexemeType.CHAR; // just consume char and back
to char state
                    break;
                default:
                    this.raiseErrorUnknownSymbol(currentChar);
            }
        }
        return null;
    }
```

```
}
```

File: Syntactic.java

```
package compiler.syntactic;

import compiler.SymbolTable;
import compiler.lexical.Lexeme;
import compiler.lexical.LexemeType;
import compiler.lexical.Lexical;
import compiler.lexical.VariableClass;
import compiler.lexical.VariableType;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JOptionPane;

public class Syntactic {

    /**
     * Lista de erros
     */
    private List<String> errors;

    public List<String> getErrors() {
        return this.errors;
    }

    /**
     * Levanta um erro de simbolo não esperado
     *
     * @param lexeme Lemexa recebido
     */
    private void raiseErrorUnexpectedSymbol(Lexeme lexeme) {
        this.errors.add("Unexpected symbol: " + lexeme.getLexeme() + ", line: "
                + (lexeme.getSourceLine() + 1) + ", position: " +
lexeme.getSourceColumn() + "\n");
    }

    /**
     * Levanta um erro de simbolo não esperado
     *
     * @param lexeme Lexema recebido
     * @param expected Tipo esperado
     */
    private void raiseErrorUnexpectedSymbol(Lexeme lexeme, int expected) {
        this.errors.add("Unexpected symbol: " + lexeme.getLexeme() + ", line: "
                + (lexeme.getSourceLine() + 1) + ", position: " +
lexeme.getSourceColumn()
                + ", Expected: " + LexemeType.getTypeName(expected) + "\n");
    }

    /**
     * Instância do analisador lexico
     */
    private Lexical lexical;

    /**
     * Tabela de simbolos
     */
    private SymbolTable symbolTable;
    /**
     * Construtor
     *
     * @param lexical
     * @param symbolTable
     */
    public Syntactic(Lexical lexical, SymbolTable symbolTable) {
        this.lexical = lexical;
        this.symbolTable = symbolTable;
        this.errors = new ArrayList<String>();
    }

    public void run() {
        this.lexeme = this.lexical.getToken();

        if (this.lexeme.getType() == LexemeType.END_OF_FILE) {
            JOptionPane.showMessageDialog(null,
                    "Source not detected!",
                    "Oops!",
                    JOptionPane.ERROR_MESSAGE);
        }

        try {
            // Start
            PROG();
        } catch (SourceErrorException ex) {
            System.out.println("Source getting end of file unexpected way... your code
have a problem!");
        }
    }

    private Lexeme lexeme;

    private boolean matchToken(int expected) throws SourceErrorException {
        if (lexeme == null) {
            throw new SourceErrorException();
        } else if (lexeme.getType() == expected) {

            // if current lexeme is a Identifier, check and install into symboltable
            if (lexeme.getType() == LexemeType.IDENTIFIER) {
                this.symbolTable.checkAndInstall(lexeme);
            }

            // Get next lexeme
            this.lexeme = this.lexical.getToken();
            return true;
        } else {
            System.out.println(">> Unexpected token...");
            this.raiseErrorUnexpectedSymbol(lexeme, expected);
        }
        return false;
    }

    private void PROG() throws SourceErrorException {
        System.out.println("Called PROG();");

        while (this.lexeme != null
                && this.lexeme.getType() != LexemeType.END_OF_FILE) {
            this.CMD();
        }
    }

    private VariableType TYPE() throws SourceErrorException {
```

```
    System.out.println("Called TYPE();");
    switch (this.lexeme.getType()) {
        case LexemeType.TYPE_CHAR:
            this.matchToken(LexemeType.TYPE_CHAR);
            return VariableType.TYPE_CHAR;
        case LexemeType.TYPE_INT:
            this.matchToken(LexemeType.TYPE_INT);
            return VariableType.TYPE_INT;
        case LexemeType.TYPE_REAL:
            this.matchToken(LexemeType.TYPE_REAL);
            return VariableType.TYPE_REAL;
        case LexemeType.TYPE_BOOL:
            this.matchToken(LexemeType.TYPE_BOOL);
            return VariableType.TYPE_BOOL;
        default:
            this.raiseErrorUnexpectedSymbol(lexeme);
    }
    return VariableType.NULL;
}

private VariableClass VAR(VariableType varType) throws SourceErrorException {
    System.out.println("Called VAR();");
    Lexeme variableReference = this.lexeme;
    this.matchToken(LexemeType.IDENTIFIER);

    if(varType != null) {
        variableReference.setVariableType(varType);
    }

    if (this.lexeme.getType() == LexemeType.BRACKETS_OPEN) { // Array

        this.matchToken(LexemeType.BRACKETS_OPEN);

        switch (this.lexeme.getType()) {
            case LexemeType.NUM_DEC:
                this.matchToken(LexemeType.NUM_DEC);
                break;
            case LexemeType.IDENTIFIER:
                this.EXP();
                break;
            default:
                this.raiseErrorUnexpectedSymbol(lexeme);
                break;
        }

        this.matchToken(LexemeType.BRACKETS_CLOSE);

        variableReference.setVariableClass(VariableClass.ARRAY);
        return VariableClass.ARRAY; // Array
    }

    variableReference.setVariableClass(VariableClass.VARIABLE);
    return VariableClass.VARIABLE; // Variable
}

private void DECLARATION() throws SourceErrorException {
    System.out.println("Called DECLARATION();");
    VariableType varType = this.TYPE();
    VariableClass type = this.VAR(varType);
    if (type == VariableClass.ARRAY) { // array
        this.DECLARATION_ARRAY();
    } else if (this.lexeme.getType() == LexemeType.OP_ATTRIBUTION) {
        this.DECLARATION_VAR();
```

```java
        }
        this.matchToken(LexemeType.TERMINATOR);
    }

    private void DECLARATION_VAR() throws SourceErrorException {
        System.out.println("Called DECLARATION_VAR();");
        this.matchToken(LexemeType.OP_ATTRIBUTION);
        this.EXP();
    }

    private void DECLARATION_ARRAY() throws SourceErrorException {
        System.out.println("Called DECLARATION_ARRAY();");
        if (this.lexeme.getType() == LexemeType.OP_ATTRIBUTION) {
            this.matchToken(LexemeType.OP_ATTRIBUTION);
            this.matchToken(LexemeType.KEY_OPEN);

            if (this.lexeme.getType() == LexemeType.PARENTHESIS_OPEN
                    || this.lexeme.getType() == LexemeType.OP_LOGICAL_NOT
                    || this.lexeme.getType() == LexemeType.IDENTIFIER
                    || this.lexeme.getType() == LexemeType.NUM_DEC
                    || this.lexeme.getType() == LexemeType.NUM_HEX
                    || this.lexeme.getType() == LexemeType.NUM_REAL
                    || this.lexeme.getType() == LexemeType.NUM_OCT
                    || this.lexeme.getType() == LexemeType.TRUE
                    || this.lexeme.getType() == LexemeType.FALSE
                    || this.lexeme.getType() == LexemeType.STRING) {
                this.EXP();
            }

            while (this.lexeme.getType() == LexemeType.COMMA) {
                this.matchToken(LexemeType.COMMA);
                this.EXP();
            }

            this.matchToken(LexemeType.KEY_CLOSE);
        }
    }

    private void ATTRIBUTION() throws SourceErrorException {
        System.out.println("Called ATTRIBUTION();");
        this.VAR(null);
        this.matchToken(LexemeType.OP_ATTRIBUTION);
        this.EXP();
        this.matchToken(LexemeType.TERMINATOR);

    }

    private void IF() throws SourceErrorException {
        System.out.println("Called IF();");
        this.matchToken(LexemeType.IF);
        this.matchToken(LexemeType.PARENTHESIS_OPEN);
        this.EXP();
        this.matchToken(LexemeType.PARENTHESIS_CLOSE);

        this.CMD();

        if (this.lexeme.getType() == LexemeType.ELSE) {
            this.matchToken(LexemeType.ELSE);
            this.CMD();
        }
    }

    private void WHILE() throws SourceErrorException {
        System.out.println("Called WHILE();");
        this.matchToken(LexemeType.WHILE);
        this.matchToken(LexemeType.PARENTHESIS_OPEN);
        this.EXP();
        this.matchToken(LexemeType.PARENTHESIS_CLOSE);

        this.matchToken(LexemeType.KEY_OPEN);

        this.CMD();

        if (this.lexeme.getType() == LexemeType.BREAK) {
            this.matchToken(LexemeType.BREAK);
        }

        this.matchToken(LexemeType.KEY_CLOSE);
    }

    private void FOR() throws SourceErrorException {
        System.out.println("Called READ();");
        this.matchToken(LexemeType.FOR);
        this.matchToken(LexemeType.PARENTHESIS_OPEN);

        // warning! attribution has expected a terminator on end!
        if (this.lexeme.getType() == LexemeType.IDENTIFIER) {
            this.ATTRIBUTION();
        } else {
            this.matchToken(LexemeType.TERMINATOR);
        }

        if (this.lexeme.getType() == LexemeType.PARENTHESIS_OPEN
                || this.lexeme.getType() == LexemeType.OP_LOGICAL_NOT
                || this.lexeme.getType() == LexemeType.IDENTIFIER
                || this.lexeme.getType() == LexemeType.NUM_DEC
                || this.lexeme.getType() == LexemeType.NUM_HEX
                || this.lexeme.getType() == LexemeType.NUM_REAL
                || this.lexeme.getType() == LexemeType.NUM_OCT
                || this.lexeme.getType() == LexemeType.TRUE
                || this.lexeme.getType() == LexemeType.FALSE
                || this.lexeme.getType() == LexemeType.STRING) {
            this.EXP();
        }
        this.matchToken(LexemeType.TERMINATOR);

        if (this.lexeme.getType() == LexemeType.PARENTHESIS_OPEN
                || this.lexeme.getType() == LexemeType.OP_LOGICAL_NOT
                || this.lexeme.getType() == LexemeType.IDENTIFIER
                || this.lexeme.getType() == LexemeType.NUM_DEC
                || this.lexeme.getType() == LexemeType.NUM_HEX
                || this.lexeme.getType() == LexemeType.NUM_REAL
                || this.lexeme.getType() == LexemeType.NUM_OCT
                || this.lexeme.getType() == LexemeType.TRUE
                || this.lexeme.getType() == LexemeType.FALSE
                || this.lexeme.getType() == LexemeType.STRING) {
            this.EXP();
        }

        this.matchToken(LexemeType.PARENTHESIS_CLOSE);
        this.CMD();

    }

    private void CMD() throws SourceErrorException {
        System.out.println("Called CMD();");
        switch (this.lexeme.getType()) {
            case LexemeType.IF:
                this.IF();
                break;
            case LexemeType.WHILE:
                this.WHILE();
                break;
            case LexemeType.FOR:
                this.FOR();
                break;
            case LexemeType.PRINT:
                this.PRINT();
                break;
            case LexemeType.READ:
                this.READ();
                break;
            case LexemeType.TYPE_CHAR:
            case LexemeType.TYPE_INT:
            case LexemeType.TYPE_REAL:
            case LexemeType.TYPE_BOOL:
                this.DECLARATION();
                break;
            case LexemeType.IDENTIFIER:
                this.ATTRIBUTION();
                break;
            case LexemeType.KEY_OPEN:
                this.BLOCK();
                break;
            case LexemeType.END_OF_FILE:
                break;
            default:
                this.raiseErrorUnexpectedSymbol(lexeme);
                this.lexeme = this.lexical.getToken(); // get next token
        }
    }

    private void BLOCK() throws SourceErrorException {
        System.out.println("Called BLOCK();");
        this.matchToken(LexemeType.KEY_OPEN);

        while (this.lexeme.getType() == LexemeType.IF
                || this.lexeme.getType() == LexemeType.WHILE
                || this.lexeme.getType() == LexemeType.FOR
                || this.lexeme.getType() == LexemeType.PRINT
                || this.lexeme.getType() == LexemeType.READ
                || this.lexeme.getType() == LexemeType.TYPE_CHAR
                || this.lexeme.getType() == LexemeType.TYPE_INT
                || this.lexeme.getType() == LexemeType.TYPE_REAL
                || this.lexeme.getType() == LexemeType.TYPE_BOOL
                || this.lexeme.getType() == LexemeType.IDENTIFIER
                || this.lexeme.getType() == LexemeType.KEY_OPEN) {
            this.CMD();
        }

        this.matchToken(LexemeType.KEY_CLOSE);
    }

    private void EXP() throws SourceErrorException {
        System.out.println("Called EXP();");
        this.EXPS();

        if (this.lexeme.getType() == LexemeType.OP_REL_EQ
                || this.lexeme.getType() == LexemeType.OP_REL_GE
                || this.lexeme.getType() == LexemeType.OP_REL_GT
                || this.lexeme.getType() == LexemeType.OP_REL_LE
                || this.lexeme.getType() == LexemeType.OP_REL_LT
                || this.lexeme.getType() == LexemeType.OP_REL_NE) {
            this.OP_REL();
            this.EXPS();
        }
    }

    private void EXPS() throws SourceErrorException {
        System.out.println("Called EXPS();");
        this.TERM();

        while (this.lexeme.getType() == LexemeType.OP_ARITMETIC_PLUS
                || this.lexeme.getType() == LexemeType.OP_ARITMETIC_LESS
                || this.lexeme.getType() == LexemeType.OP_ARITMETIC_INC
                || this.lexeme.getType() == LexemeType.OP_ARITMETIC_DEC
                || this.lexeme.getType() == LexemeType.OP_LOGICAL_OR) {

            if (this.lexeme.getType() == LexemeType.OP_ARITMETIC_INC
                    || this.lexeme.getType() == LexemeType.OP_ARITMETIC_DEC) {
                this.OP_ADD(); // if ++ or -- not call TERM
            } else {
                this.OP_ADD();
                this.TERM();
            }
        }
    }

    private void TERM() throws SourceErrorException {
        System.out.println("Called TERM();");
        this.FACTOR();
        while (this.lexeme.getType() == LexemeType.OP_ARITMETIC_MULT
                || this.lexeme.getType() == LexemeType.OP_ARITMETIC_DIV
                || this.lexeme.getType() == LexemeType.OP_ARITMETIC_MOD
                || this.lexeme.getType() == LexemeType.OP_LOGICAL_AND
                || this.lexeme.getType() == LexemeType.OP_LOGICAL_NOT) {
            this.OP_MUL();
            this.FACTOR();
        }
    }

    private void FACTOR() throws SourceErrorException {
        System.out.println("Called FACTOR();");
        switch (this.lexeme.getType()) {
            case LexemeType.PARENTHESIS_OPEN:
                this.matchToken(LexemeType.PARENTHESIS_OPEN);
                this.EXP();
                this.matchToken(LexemeType.PARENTHESIS_CLOSE);
                break;
            case LexemeType.OP_LOGICAL_NOT:
                this.matchToken(LexemeType.OP_LOGICAL_NOT);
                this.FACTOR();
                break;
            case LexemeType.IDENTIFIER:
                this.VAR(null);
                break;
            case LexemeType.NUM_DEC:
                this.matchToken(LexemeType.NUM_DEC);
                break;
            case LexemeType.NUM_HEX:
                this.matchToken(LexemeType.NUM_HEX);
                break;
```

```java
            case LexemeType.NUM_REAL:
                this.matchToken(LexemeType.NUM_REAL);
                break;
            case LexemeType.NUM_OCT:
                this.matchToken(LexemeType.NUM_OCT);
                break;
            case LexemeType.TRUE:
                this.matchToken(LexemeType.TRUE);
                break;
            case LexemeType.FALSE:
                this.matchToken(LexemeType.FALSE);
                break;
            case LexemeType.STRING:
                this.matchToken(LexemeType.STRING);
                break;
            default:
                this.raiseErrorUnexpectedSymbol(lexeme);
        }
    }

    private void OP_REL() throws SourceErrorException {
        System.out.println("Called OP_REL();");
        switch (this.lexeme.getType()) {
            case LexemeType.OP_REL_EQ:
                matchToken(LexemeType.OP_REL_EQ);
                break;
            case LexemeType.OP_REL_GE:
                matchToken(LexemeType.OP_REL_GE);
                break;
            case LexemeType.OP_REL_GT:
                matchToken(LexemeType.OP_REL_GT);
                break;
            case LexemeType.OP_REL_LE:
                matchToken(LexemeType.OP_REL_LE);
                break;
            case LexemeType.OP_REL_LT:
                matchToken(LexemeType.OP_REL_LT);
                break;
            case LexemeType.OP_REL_NE:
                matchToken(LexemeType.OP_REL_NE);
                break;
            default:
                this.raiseErrorUnexpectedSymbol(lexeme);
        }
    }

    private void OP_ADD() throws SourceErrorException {
        System.out.println("Called OP_ADD();");
        switch (this.lexeme.getType()) {
            case LexemeType.OP_ARITMETIC_PLUS:
                matchToken(LexemeType.OP_ARITMETIC_PLUS);
                break;
            case LexemeType.OP_ARITMETIC_LESS:
                matchToken(LexemeType.OP_ARITMETIC_LESS);
                break;
            case LexemeType.OP_ARITMETIC_INC:
                matchToken(LexemeType.OP_ARITMETIC_INC);
                break;
            case LexemeType.OP_ARITMETIC_DEC:
                matchToken(LexemeType.OP_ARITMETIC_DEC);
                break;
            case LexemeType.OP_LOGICAL_OR:
                matchToken(LexemeType.OP_LOGICAL_OR);
                break;
            default:
                this.raiseErrorUnexpectedSymbol(lexeme);
        }
    }

    private void OP_MUL() throws SourceErrorException {
        System.out.println("Called OP_MUL();");
        switch (this.lexeme.getType()) {
            case LexemeType.OP_ARITMETIC_MULT:
                matchToken(LexemeType.OP_ARITMETIC_MULT);
                break;
            case LexemeType.OP_ARITMETIC_DIV:
                matchToken(LexemeType.OP_ARITMETIC_DIV);
                break;
            case LexemeType.OP_ARITMETIC_MOD:
                matchToken(LexemeType.OP_ARITMETIC_MOD);
                break;
            case LexemeType.OP_LOGICAL_AND:
                matchToken(LexemeType.OP_LOGICAL_AND);
                break;
            case LexemeType.OP_LOGICAL_NOT:
                matchToken(LexemeType.OP_LOGICAL_NOT);
                break;
            default:
                this.raiseErrorUnexpectedSymbol(lexeme);
        }
    }

    private void PRINT() throws SourceErrorException {
        System.out.println("Called PRINT();");
        this.matchToken(LexemeType.PRINT);
        this.matchToken(LexemeType.PARENTHESIS_OPEN);
        this.matchToken(LexemeType.STRING);

        while (this.lexeme.getType() == LexemeType.COMMA) {
            this.matchToken(LexemeType.COMMA);
            this.FACTOR();
        }

        this.matchToken(LexemeType.PARENTHESIS_CLOSE);
        this.matchToken(LexemeType.TERMINATOR);
    }

    private void READ() throws SourceErrorException {
        System.out.println("Called READ();");
        this.matchToken(LexemeType.READ);
        this.matchToken(LexemeType.PARENTHESIS_OPEN);
        this.matchToken(LexemeType.STRING);
        while (this.lexeme.getType() == LexemeType.COMMA) {
            this.matchToken(LexemeType.COMMA);
            this.FACTOR();
        }

        this.matchToken(LexemeType.PARENTHESIS_CLOSE);
        this.matchToken(LexemeType.TERMINATOR);
    }

}
```

File: SymbolTable.java

```
package compiler;
```

```java
import compiler.lexical.Lexeme;
import compiler.lexical.LexemeType;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;

public class SymbolTable {

    private HashSet<Lexeme> symbolTable;

    public SymbolTable() {
        this.symbolTable = new HashSet<Lexeme>();
        this.installDefaultTokens();
    }

    private void installDefaultTokens() {
        this.add(new Lexeme("true", LexemeType.TRUE));
        this.add(new Lexeme("false", LexemeType.FALSE));
        this.add(new Lexeme("if", LexemeType.IF));
        this.add(new Lexeme("else", LexemeType.ELSE));
        this.add(new Lexeme("while", LexemeType.WHILE));
        this.add(new Lexeme("break", LexemeType.BREAK));
        this.add(new Lexeme("for", LexemeType.FOR));
        this.add(new Lexeme("char", LexemeType.TYPE_CHAR));
        this.add(new Lexeme("int", LexemeType.TYPE_INT));
        this.add(new Lexeme("real", LexemeType.TYPE_REAL));
        this.add(new Lexeme("bool", LexemeType.TYPE_BOOL));
        this.add(new Lexeme("print", LexemeType.PRINT));
        this.add(new Lexeme("read", LexemeType.READ));
    }

    public void add(Lexeme lexeme) {
        this.symbolTable.add(lexeme);
    }

    public void checkAndInstall(Lexeme lexeme) {
        if (lexeme.getType() == LexemeType.IDENTIFIER) {
            for (Lexeme symbol : this.symbolTable) {
                if
(symbol.getLexeme().toLowerCase().equals(lexeme.getLexeme().toLowerCase())) {
                    return; // has exists into table
                }
            }
            this.add(lexeme);
        }
    }

    public int resolveLexemeType(Lexeme lexeme) {
        for (Lexeme symbol : this.symbolTable) {
            if (symbol.getLexeme().equals(lexeme.getLexeme().toLowerCase())) {
                return symbol.getType();
            }
        }
        return LexemeType.IDENTIFIER;
    }

    public List<String> getTableString() {
        List<String> output = new ArrayList<String>();
        for (Lexeme symbol : this.symbolTable) {
            output.add(String.format("<%s, \"%s\", \"%s\", \"%s\">\n",
                    symbol.getTypeString(),
                    symbol.getLexeme(),
                        symbol.getVariableClass(),
                        symbol.getVariableType()));
        }
        return output;
    }
}
```

File: Compiler.java

```
package compiler;

import compiler.lexical.Lexeme;
import javax.swing.JEditorPane;

import compiler.lexical.Lexical;
import compiler.syntactic.Syntactic;
import java.util.List;

public class Compiler {

    private JEditorPane sourceEditor;

    private SymbolTable symbolTable;

    private Lexical lexical;

    private Syntactic syntactic;

    public List<String> getLexicalErrors() {
        return this.lexical.getErrors();
    }

    public List<String> getLexicalOutput() {
        return this.lexical.getOutput();
    }

    public List<String> getSymbolTable() {
        return this.symbolTable.getTableString();
    }

    public List<String> getSyntacticErrors() {
        return this.syntactic.getErrors();
    }

    public Compiler(JEditorPane sourceEditor) {
        this.sourceEditor = sourceEditor;
        this.symbolTable = new SymbolTable();
    }

    public void run() {
        this.lexical = new Lexical(this.symbolTable, this.sourceEditor.getText());
        this.syntactic = new Syntactic(this.lexical, this.symbolTable);

        this.syntactic.run();
    }
}
```

File: SourceErrorException.java

```
public class SourceErrorException extends Exception {

}
```