TRANSPILADOR DE R PARA JULIA USANDO PLY

[1] Ageu William Morais Gomes [2] Carlos Américo Batista de Lima [3] Francisco Fernandes do Vale Segundo [4] Jorge Luiz Silva Braz [5] Pedro Eliaquim de Morais Silva [6] Thailson Emanuel de Lima Oliveira

UNIVERSIDADE DO ESTADO DO RIO GRANDE DO NORTE (UERN)

DEPARTAMENTO DE INFORMÁTICA / CIÊNCIAS DA COMPUTACÃO

DISCIPLINA: COMPILADORES E PARADIGMAS DE PROGRAMAÇÃO

PROFESSOR: SEBASTIÃO EMIDIO ALVES FILHO

MOSSORÓ, RN, BRASIL

---

1. Descrição geral e justificativa

Este projeto implementa um transpilador que converte programas escritos em uma versão simplificada de R para código equivalente em Julia. O sistema utiliza análise léxica e sintática com PLY (lex/yacc em Python) e geração de código própria, permitindo migrar scripts de R para Julia com pouca ou nenhuma intervenção manual.

A justificativa para o trabalho está na possibilidade de aproveitar scripts existentes em R em um ambiente Julia, que oferece maior desempenho em operações numéricas e um ecossistema científico moderno. Isso facilita a integração de projetos de data science, análise estatística e computação científica com bibliotecas de alto desempenho do Julia, sem a necessidade de reescrever todo o código manualmente.

---

2. Linguagem de origem e de destino

2.1 Linguagem de origem:

Uma versão simplificada de R, compatível com a maioria dos scripts R básicos. O analisador reconhece atribuições com <- e =, expressões aritméticas e lógicas, comandos condicionais (if/else), laços (for, while), declaração e

chamada de funções, além de operações com vetores, listas, data frames e acesso a campos com $ e [].

2.2 Linguagem de destino:

Julia. O transpilador gera código Julia 1.x, com estruturas como atribuições, blocos de controle, funções, vetores, dicionários, DataFrames e operações aritméticas e lógicas equivalentes às do R. O código gerado pode ser executado diretamente no REPL ou em ambientes como VSCode com o pacote Julia.

---

3. A Gramática dos Comandos

3.1 Programa e Comandos:

```
program    → statements
statements → statement
| statements statement
| statements ';' statement
| statements NEWLINE statement
statement  → ID '<-' expression
| ID '=' expression
| expression '[' expression ']' '<-' expression
| expression '$' ID '<-' expression
| expression
| ';' | NEWLINE
| 'if' '(' expression ')' block
| 'if' '(' expression ')' block 'else' block
| 'while' '(' expression ')' block
| 'for' '(' ID 'in' expression ')' block
| ID '<-' 'function' '(' param_list ')' block
| ID '<-' 'function' '(' ')' block
| 'return' expression.
```

3.2 Expressões

```
expression → expression ('+' | '-' | '*' | '/' | '^') expression
```

```
        | expression ('==' | '!=' | '<' | '<=' | '>' | '>=') expression
        | expression ('&&' | '||') expression
        | '!' expression
        | '-' expression
        | '(' expression ')'
        | INT_LITERAL
        | FLOAT_LITERAL
        | STRING_LITERAL
        | BOOL_LITERAL
        | ID
        | expression '[' expression ']'
        | expression '$' ID
        | ID '(' ')'
        | ID '(' expression ')'
        | ID '(' arg_list ')'
arg_list  → arg
        | arg_list ',' arg
arg       → ID '=' expression
        | expression
```

3.3 Blocos e Funções

```
block     → '{' statements '}'
        | statement
param_list → ID
        | param_list ',' ID
```

Essas produções cobrem declaração/atribuição de variáveis, entrada/saída padrão, expressões aritméticas e lógicas, comandos condicionais, laços e funções com parâmetros posicionais e nomeados.

---

4. Tokens suportados

O analisador léxico reconhece os seguintes tokens:

Token	Exemplo(s)	Descrição
INT_LITERAL	42, 42L	Inteiros, com ou sem sufixo L.
FLOAT_LITERAL	3.14, 1.2e-3	Números de ponto flutuante.
STRING_LITERAL	"texto", 'texto'	Strings entre aspas simples ou duplas.
BOOL_LITERAL	TRUE, FALSE	Literais booleanos.
ID	x, funcao, is.double	Identificadores (nomes de variáveis, funções, etc.).
PLUS	+	Operador de adição.
MINUS	-	Operador de subtração.
MUL	*	Operador de multiplicação.
DIV	/	Operador de divisão.
POW	^	Operador de potência.
EQ	==	Operador de igualdade.
NE	!=	Operador de diferença.
LT	<	Operador de menor que.

LE	<=	Operador de menor ou igual.
GT	>	Operador de maior que.
GE	>=	Operador de maior ou igual.
ASSIGN_ARROW	<-	Operador de atribuição.
ASSIGN_EQ	=	Operador de atribuição.
AND	&&	Operador lógico E.
OR	`	
NOT	!	Operador lógico NÃO.
LPAREN	(	Parêntese esquerdo.
RPAREN	)	Parêntese direito.
LBRACE	{	Chave esquerda.
RBRACE	}	Chave direita.
LBRACK	[	Colchete esquerdo.
RBRACK	]	Colchete direito.
COMMA	,	Vírgula.
SEMICOLON	;	Ponto e vírgula.

COLON	:	Dois pontos.
DOLLAR	$	Acesso a campo.
BACKTICK	`	Crase (usada em nomes especiais de função).
Palavras reservadas	if, else, for, in, while, function, return	Palavras reservadas.

5. AST

Nó (classe)	Campos principais	Representa
Program	stmts	Programa completo (lista de comandos).
Block	stmts	Bloco { ... } ou corpo de controle/função.
Assign	name, expr	Atribuição de variável (x <- expr ou x = expr).
ExprStmt	expr	Expressão usada como comando (ex.: print(x)). a
If	cond, then_block, else_block	Comando condicional (if/if-else).
For	var, start_expr, end_expr, body	Laço for (var in expr) ou for (var in a:b).
While	cond, body	Laço while.
FunctionDecl	name, params, body	Declaração de função (f <- function(...) { ... }).
Return	expr	Comando de retorno (return expr).

Nó (classe)	Campos principais	Representa
Call	name, args	Chamada de função (com argumentos posicionais ou nomeados).
NamedArg	name, value	Argumento nomeado (nome = valor).
BinaryOp	op, left, right	Expressão binária (+, -, *, /, &&, :, etc.).
UnaryOp	op, expr	Expressão unária (!expr, -expr).
Var	name	Uso de variável ou identificador.
IntLiteral	value	Literal inteiro.
FloatLiteral	value	Literal de ponto flutuante.
StringLiteral	value	Literal string.
BoolLiteral	value	Literal booleano (TRUE/FALSE).
IndexOp	target, index	Acesso a índice (x[i]).
AssignIndex	target, index, expr	Atribuição a índice (x[i] <- v ou x$campo <- v).
DollarAccess	target, field	Acesso a campo (x$campo).
IsDouble	expr	Verificação de tipo (is.double(expr)).
IsInteger	expr	Verificação de tipo (is.integer(expr)).
S3FunctionDecl	operator_name, class_name, params, body	Declaração de método S3 (ex: `+.classe`).

6. Instruções de execução

- **Pré-requisitos:**
    - Python 3.x instalado.
    - Biblioteca ply instalada (pip install ply).
    - Estrutura de diretórios: pasta RProjectExamples para os arquivos .R de entrada e pasta juliaExamples para os arquivos .jl de saída.
- **Modo interativo:**
    - Coloque os arquivos .R na pasta RProjectExamples.
    - Execute python src/transpile.py.
    - O script listará os arquivos .R disponíveis; escolha um índice para transpilar.
    - O código Julia gerado será salvo na pasta juliaExamples.
- **Modo via linha de comando:**
    - Execute python src/transpile.py caminho/para/arquivo.R [saida.jl].
    - Se não for fornecido o nome de saída, o arquivo será salvo como juliaExamples/<nome>.jl.
- **Execução do código Julia:**
    - Abra o arquivo .jl gerado no REPL do Julia ou em um editor compatível e execute o código.

---

7. Link do Repositório

Repositório no GitHub: https://github.com/valesecond/transpilador-r-julia