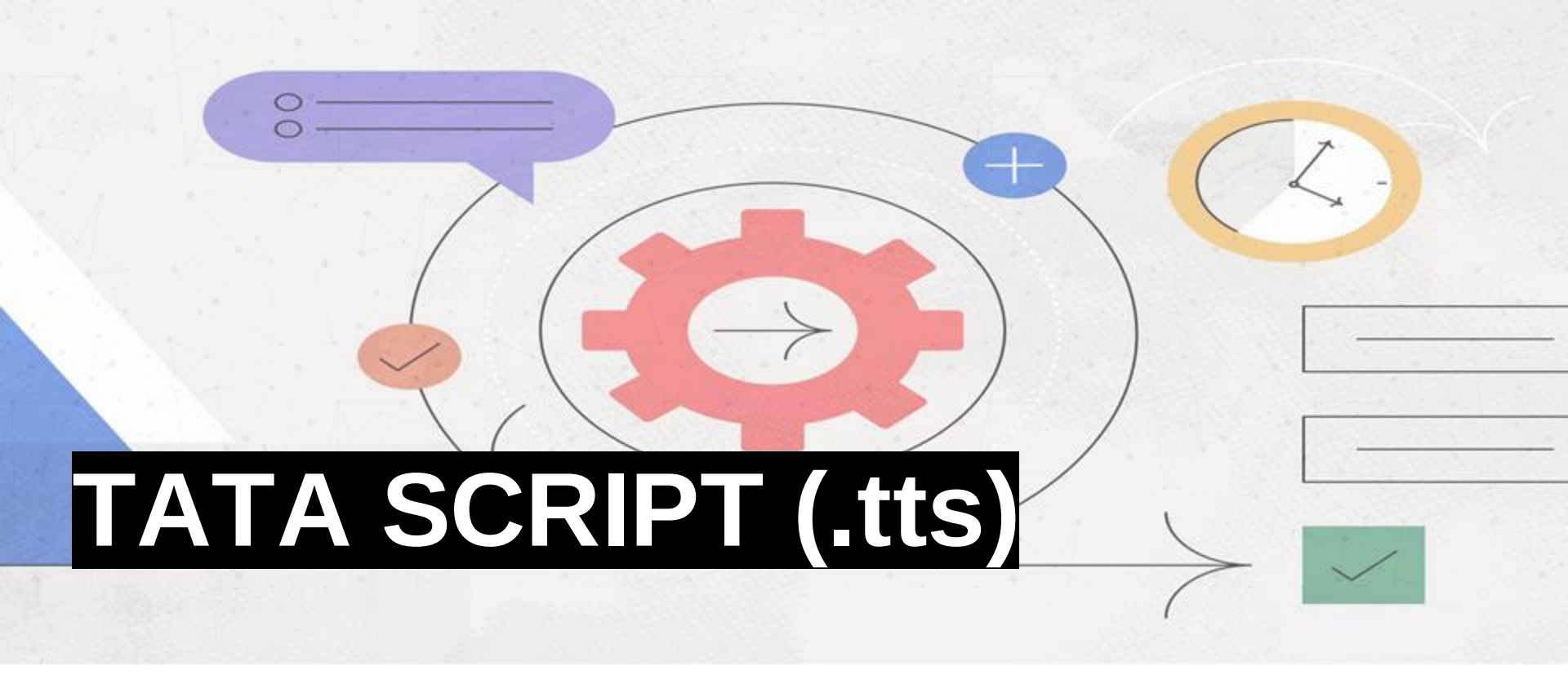

# TATA SCRIPT (.tts)

**Presenters**: Kavya, Arun, Dhruva, Poojan

**Group : 13**

Arizona State University

# Intro to TataScript

The name of our program is inspired by Sir Ratan Tata.

# Features

- Compiled and then Interpreted
- Compatible with Windows, MacOS, and Linux
- Written in Prolog
- Strongly typed
- The entire program is a block of code encapsulated in semicolon
- Delimiters: . , {}, ->, <-, ;
- Covers all basic operations
    - Assignment
    - Display
    - Conditional + Ternary
    - Loops (for & while)

# TataScript Grammar

```
% Tabling condition predicate
:- table condition/2.
:- table add_and_sub_expr/2.
:- table mul_and_div_expr/2.
% Start point of the program, program is a block encapsulated between semicolons(;)
program --> [;], main_block, [;].

% Program block is bunch of statements
main_block --> statements.

% Statements can be individual statement or a chain of statements.
statements --> statement, statements.
statements --> statement.
```

# TataScript Grammar

```
% Individual statements
statement --> assignment, [.].
statement --> display, [.].
statement --> if_statement.
statement --> for_loop.
statement --> while_loop.
statement --> increment, [.].

% Assignment can be with or without datatype
assignment --> data_type, variable, [=], expression.
assignment --> variable, [=], expression.

% this deals with the expressions
expression --> chainedassign.
expression --> ternary_statement.
expression --> boolean_op.
```

# TataScript Grammar

```
% to implement precedence in arithematic operators
chainedassign --> variable, [=], expression.
chainedassign--> add_and_sub_expr.
add_and_sub_expr--> add_and_sub_expr, [+], mul_and_div_expr.
add_and_sub_expr --> add_and_sub_expr, [-], mul_and_div_expr.
add_and_sub_expr --> mul_and_div_expr.


% multiplication and division are grouped because they have the same precedence
mul_and_div_expr --> mul_and_div_expr, [*], remaining_expression.
mul_and_div_expr --> mul_and_div_expr, [/], remaining_expression.
mul_and_div_expr --> remaining_expression.


% handling parenthesis, it has higher precedence than the operators above
remaining_expression --> ['('], expression, [')'].
```

# TataScript Grammar

% the expression can finally be an identifier and number
remaining_expression --> variable.
remaining_expression --> number.
remaining_expression --> string_literal.
%remaining_expression --> variable, [=], expression.

% Display is used to output to the screen
display --> [display], [->], output.
output --> string_literal | variable | number.

% Condtional Statements
% If statement with else if chain followed by a final else block. If is followed by a ->
if_statement --> [if, ->], condition, block, elseif_blocks, else_block.

# TataScript Grammar

```
% Else-if and else blocks
% Optional blocks. Else-if is <--> and else is <-
elseif_blocks --> elseif_block, elseif_blocks.
elseif_blocks --> [].
elseif_block --> [<-, ->], condition, block.
else_block --> [<-], block.
else_block --> [].


% Ternary Statements
ternary_statement --> condition, [->], ternary_expression, [<-], ternary_expression.
ternary_expression --> expression | statements.
```

# TataScript Grammar

```
% For loop is the keyword for followed by ->, and init, condition and increment separated by :
for_loop --> [for, ->], init, [:], condition, [:], increment, block.


% Initialization :  for "for" loop
init --> assignment.


% Increment : for "for" loop
increment --> variable, increment_op.


% While loop is the keyword while followed by -> and a condition
while_loop --> [while, ->], condition, block.


% Code blocks are statements encapusulated within {} curly braces
block --> ['{'], statements, ['}'].
block --> ['{'],['}'].
```

# TataScript Grammar

% Expressions
%expression --> term.
%expression --> term, arithmetic_op, expression.
%expression --> arithmetic_op, parenthesis, arithmetic_op
%expression --> arithmetic_op, parenthesis, arithmetic_op
%expression --> increment.
%expression --> ternary_statement.
%expression --> boolean_op.

% Terms
%term --> variable.
%term --> number.
%term --> string_literal.

# TataScript Grammar

```
% Conditions
condition --> expression, relational_op, expression.
condition --> expression, boolean_op, expression.
condition --> condition, logical_op, condition.


% Operators
%arithmetic_op --> [+] | [-] | [*] | [/].
relational_op --> [<] | [>] | [==] | [<=] | [>=] | [!].
%parenthesis --> ['('], expression, [')'].
boolean_op --> [true] | [false].
increment_op --> [++] | [--].
logical_op --> [&] | ['|'].
```

# TataScript Grammar

```
% Data types can be int, string and bool
data_type --> [int] | [string] | [bool].


% Variable names
variable --> [Var], { atom(Var), atom_chars(Var, [FirstChar | Rest]), char_type(FirstChar, alpha),
        all_alnum_or_underscore(Rest) }.


% Numbers and string literals
number --> [Num], { number(Num) }.
string_literal --> [Str], { atom(Str), atom_concat("", , Str), atom_concat(, "", Str) }.


% Helper for checking valid variable names
all_alnum_or_underscore([]).
all_alnum_or_underscore([Char | Rest]) :-
    (char_type(Char, alnum) ; Char == '_'),
    all_alnum_or_underscore(Rest).
```

# Tokenizer

```
% Tokenize program from .tts files to a list of tokens
tokenize_file(Filename, Tokens) :-
    file_name_extension(_Base, tts, Filename),  % Check for .tts extension
    open(Filename, read, Stream),
    read_string(Stream, _, Content),          % Read File
    close(Stream),
    tokenize(Content, Tokens).


% Tokenize predicate taking program as a string giving a list of tokens
tokenize(Input, Tokens) :-
    string_chars(Input, Chars),
    tokenize_chars(Chars, Tokens).


% Base Case
tokenize_chars([], []).
```

# Tokenizer

```
% Char tokenizer to handle numbers
tokenize_chars([Char | RestChars], [Number | Tokens]) :-
    char_type(Char, digit),
    !,
    collect_number_chars([Char | RestChars], NumChars, RemainingChars),
    number_chars(Number, NumChars),
    tokenize_chars(RemainingChars, Tokens).


% Char tokenizer to handle strings
tokenize_chars(["" | RestChars], [QuotedString | Tokens]) :-
    collect_string(RestChars, StringChars, RemainingChars),
    append([""], StringChars, TempQuoted),
    append(TempQuoted, [""], QuotedStringChars),
    atom_chars(QuotedString, QuotedStringChars),
    tokenize_chars(RemainingChars, Tokens).
```

# Tokenizer

```
% Handling operators with 2 characters like -> <- -- ++
tokenize_chars([Char1, Char2 | RestChars], [Token | Tokens]) :-
    tokenize_double(Char1, Char2, RestChars, Token, RemainingChars),
    !,
    tokenize_chars(RemainingChars, Tokens).


% Hangling single character operators
tokenize_chars([Char | RestChars], [Token | Tokens]) :-
    tokenize_single(Char, RestChars, Token, RemainingChars),
    tokenize_chars(RemainingChars, Tokens).

% Whitespace and new line handling
tokenize_chars([' ' | RestChars], Tokens) :-
    tokenize_chars(RestChars, Tokens).
tokenize_chars(['\n' | RestChars], Tokens) :-
    tokenize_chars(RestChars, Tokens).
tokenize_chars(['\t' | RestChars], Tokens) :-
    tokenize_chars(RestChars, Tokens).
```

# Tokenizer

```
% Token continuous numbers
collect_number_chars([], [], []).
collect_number_chars([Char | Rest], [Char | NumChars], Remaining) :-
    char_type(Char, digit),
    !,
    collect_number_chars(Rest, NumChars, Remaining).
collect_number_chars([Char | Rest], [], [Char | Rest]) :-
    \+ char_type(Char, digit).


% String tokeninzation : Collect until double quotes uncounter
collect_string(['"' | Rest], [], Rest) :- !.
collect_string([Char | RestChars], [Char | StringChars], Remaining) :-
    collect_string(RestChars, StringChars, Remaining).
```

# Tokenizer

```
% Two character operators
tokenize_double(Char1, Char2, RestChars, Token, RestChars) :-
    member([Char1, Char2], [['<', '-'], ['-', '>'], ['+', '+'], ['-', '-'],
                            ['<','='],['>','='],['!','='],['=','=']]),
    atom_chars(Token, [Char1, Char2]).


% Single Character operators
tokenize_single(Char, RestChars, Token, RestChars) :-
    member(Char, ['=', '+', '-', '*', '/', ',', '.', ';', ':', '!', '?',
                  '(', ')', '[', ']', '{', '}', '<', '>', '"', '&', '|']),
    atom_chars(Token, [Char]).
```

# Tokenizer

```
% Identifiers and Keywords handling
tokenize_single(Char, RestChars, Word, RemainingChars) :-
    is_alnum(Char),
    collect_alnum([Char | RestChars], AlnumChars, RemainingChars),
    atom_chars(Word, AlnumChars).

collect_alnum([Char | RestChars], [Char | AlnumChars], RemainingChars) :-
    is_alnum(Char),
    !,
    collect_alnum(RestChars, AlnumChars, RemainingChars).
collect_alnum(RemainingChars, [], RemainingChars).

is_alnum(Char) :-
    char_type(Char, alnum).
```

# Tokenizer : Example

```
/Users/arun/Documents/Coding/SER502-TataScript-Team13 [branch_arun]% ./tata.sh data/fibonacci.tts
[;,int,n,=,10,.,int,prev,=,0,.,int,current,=,1,.,int,count,=,0,.,display,->,"Generating Fibonacci Sequence",.,display,->,"Number of terms: 'n'",.,if,->,n,<=,0,{,display,->,"Please enter a positive number",.,}
,<-,->,n,==,1,{,display,->,"First Fibonacci number: 'prev'",.,},<-,{,display,->,"Fibonacci Sequence:",.,display,->,"Term : 'count' : 'prev'",.,count,=,count,+,1,.,display,->,"Term : 'count' : 'current'",.,for
,->,i,=,2,:,i,<,n,:,i,++,{,int,next,=,prev,+,current,.,count,=,count,+,1,.,display,->,"Term : 'count' : 'next'",.,prev,=,current,.,current,=,next,.,},display,->,"Sequence generation complete.",.,},;]
```

# TataScript Compiler

```
% Tabling condition predicate
:- table condition/3.
:- table add_and_sub_expr/3.
:- table mul_and_div_expr/3.

% Start point of the program, program is a block encapusated between semicolons(;)
program(program(X)) --> [;], main_block(X), [;].

% Program block is bunch of statements
main_block(main_block(X)) --> statements(X).
```

# TataScript Compiler

```
% Statements can be individual statement or a chain of statements.
statements(statement_block(X, Y)) --> statement(X), statements(Y).
statements(statement(X)) --> statement(X).

% Individual statements
statement(X) --> assignment(X), [.].
statement(X) --> display(X), [.].
statement(X) --> if_statement(X).
statement(X) --> for_loop(X).
statement(X) --> while_loop(X).
statement(X) --> increment(X), [.].
```

# TataScript Compiler

% Assignment can be with or without datatype
assignment(assign(D, V, E)) --> data_type(D), variable(V), [=], expression(E).
assignment(assign(V, E)) --> variable(V), [=], expression(E).

expression(X) --> chainedassign(X).

expression(X) --> ternary_statement(X).
expression(X) --> boolean_op(X).

# TataScript Compiler

```
% to implement precedence in arithematic operators
% addition and subtraction are grouped because they have the same precedence
chainedassign(assign(X,Y)) --> variable(X), [=], expression(Y), !.
chainedassign(X)--> add_and_sub_expr(X).
add_and_sub_expr(add(X, Y)) --> add_and_sub_expr(X), [+], mul_and_div_expr(Y).
add_and_sub_expr(sub(X, Y)) --> add_and_sub_expr(X), [-], mul_and_div_expr(Y).
add_and_sub_expr(X) --> mul_and_div_expr(X).
```

# TataScript Compiler

```
% multiplication and division are grouped because they have the same precedence
mul_and_div_expr(mul(X, Y)) --> mul_and_div_expr(X), [*], remaining_expression(Y).
mul_and_div_expr(divide(X, Y)) --> mul_and_div_expr(X), [/], remaining_expression(Y).
mul_and_div_expr(X) --> remaining_expression(X).
remaining_expression(assign(X, Y)) --> variable(X), [=], expression(Y).
```

# TataScript Compiler

```
% handling parenthesis, it has higher precedence than the operators
above
remaining_expression(parenthesis(X)) --> ['('], expression(X), [')'].

remaining_expression(X) --> variable(X).
remaining_expression(X) --> number(X).
% remaining_expression(bool_val(X) -->
remaining_expression(X) --> string_literal(X).
```

# TataScript Compiler

```
% Display is used to output to the screen
output(X) --> string_literal(X).
output(X) --> number(X).
output(X) --> variable(X).
display(display(X)) --> [display], [->], output(X).
```

# TataScript Compiler

% Condtional Statements
% If statement with else if chain followed by a final else block. If is followed by a ->
if_statement(if(C, B, EI, E)) --> [if, ->], condition(C), block(B), elseif_blocks(EI), else_block(E).

# TataScript Compiler

```
% Else-if and else blocks
% Optional blocks. Else-if is <--> and else is <-
elseif_blocks(elseif_block(X, Y)) --> elseif_block(X), elseif_blocks(Y).
elseif_blocks([]) --> [].

elseif_block(elseIf(C, B)) --> [<-, ->], condition(C), block(B).

else_block(else(B)) --> [<-], block(B).
else_block([]) --> [].
```

# TataScript Compiler

```
% Ternary Statements
ternary_statement(ternary(C, E1, E2)) --> condition(C), [->],
ternary_expression(E1), [<-], ternary_expression(E2).
ternary_expression(X) --> expression(X).
ternary_expression(X) --> statements(X).
```

# TataScript Compiler

% For loop is the keyword for followed by ->, and init, condition and increment separated by :
for_loop(for(I, C, In, B)) --> [for, ->], init(I), [:], condition(C), [:], increment(In), block(B).

% Initialization :  for "for" loop
init(I) --> assignment(I).

% Increment : for "for" loop
increment(incr(X, Op)) --> variable(X), increment_op(Op).

# TataScript Compiler

```
% While loop is the keyword while followed by -> and a condition
while_loop(while(C, B)) --> [while, ->], condition(C), block(B).

% Code blocks are statements encapusulated within {} curly braces
block(block(B)) --> ['{'], statements(B), ['}'].
block([]) --> ['{'], ['}'].
```

# TataScript Compiler

% Expressions
%expression(X) --> term(X).
%expression(exp(X, Op, Y)) --> term(X), arithmetic_op(Op), expression(Y).
%expression(X) --> increment(X).
%expression(X) --> ternary_statement(X).
%expression(X) --> boolean_op(X).

# TataScript Compiler

```
% Terms
%term(X) --> variable(X).
%term(X) --> number(X).
%term(X) --> string_literal(X).

% Conditions
condition(cond(X, Op, Y)) --> expression(X), relational_op(Op),
expression(Y).
condition(cond(X, Op, Y)) --> condition(X), logical_op(Op), condition(Y).
```

# TataScript Compiler : Parsed Tree Example

```
program(main_block(statement_block(assign(data_type(int),var(n),num(10)),statement_block(assign(data_type(int),var(prev),num(0)),statement_block(assign(data_type(int),var(current),num(1)),statement_block(assign(data_type(int),var(count),num(0)),statement_block(display(string("Generating Fibonacci Sequence")),statement_block(display(string("Number of terms: 'n'")),statement(if(cond(var(n),<=,num(0)),block(statement(display(string("Please enter a positive number")))),elseif_block(elseIf(cond(var(n),==,num(1)),block(statement(display(string("First Fibonacci number: 'prev'"))))),[]),else(block(statement_block(display(string("Fibonacci Sequence:")),statement_block(display(string("Term : 'count' : 'prev'")),statement_block(assign(var(count),add(var(count),num(1))),statement_block(display(string("Term : 'count' : 'current'")),statement_block(for(assign(var(i),num(2)),cond(var(i),<,var(n)),incr(var(i),++),block(statement_block(assign(data_type(int),var(next),add(var(prev),var(current))),statement_block(assign(var(count),add(var(count),num(1))),statement_block(display(string("Term : 'count' : 'next'")),statement_block(assign(var(prev),var(current)),statement(assign(var(current),var(next))))))))),statement(display(string("Sequence generation complete.")))))))))))))))))))))))
```

# TataScript Compiler

```
% Operators
arithmetic_op(+) --> [+].
arithmetic_op(-) --> [-].
arithmetic_op() --> [].
arithmetic_op(/) --> [/].

relational_op(<) --> [<].
relational_op(>) --> [>].
relational_op(==) --> [==].
relational_op(<=) --> [<=].
relational_op(>=) --> [>=].
relational_op(!) --> [!].

boolean_op(true) --> [true].
boolean_op(false) --> [false].

increment_op(++) --> [++].
increment_op(--) --> [--].

logical_op(and) --> [&].
logical_op(or) --> ['|'].
```

# TataScript Compiler

```
% Numbers and string literals
number(num(Num)) --> [Num], { number(Num) }.
% Negative Number Handling
number(num(Num)) --> [-], [Num1], { number(Num1), Num is -1 * Num1 }.

string_literal(string(Str)) --> [Str], { atom(Str), atom_concat("", , Str), atom_concat(, "", Str) }.

% Helper for checking valid variable names
all_alnum_or_underscore([]).
all_alnum_or_underscore([Char | Rest]) :-
    (char_type(Char, alnum) ; Char == '_'),
    all_alnum_or_underscore(Rest).
```

# TataScript Interpreter

```
% Lookup will check the variable in the env
lookup(Var, [Var=Value | _], Value).
lookup(Var, [_ | Rest], Value) :- lookup(Var, Rest, Value).

% Update env will update the env variables with the value
update_env([], Var, Value, [Var=Value]).
update_env([Var=_ | Rest], Var, Value, [Var=Value | Rest]).
update_env([Other | Rest], Var, Value, [Other | UpdatedRest]) :-
    update_env(Rest, Var, Value, UpdatedRest).

% Program Evaluation is the evaluation of the main block which is enclosed in semicolons
program_eval(program(MainBlock), FinalEnv) :-
    main_block_eval(MainBlock, [], FinalEnv).
```

# TataScript Interpreter

```
% Main block evaluation is the evaluation of statements
main_block_eval(main_block(Statements), Env, FinalEnv) :-
    statements_eval(Statements, Env, FinalEnv).

% Statements Evaluation
statements_eval(statement_block(Stmt, Rest), Env, FinalEnv) :-
    statement_eval(Stmt, Env, TempEnv),
    statements_eval(Rest, TempEnv, FinalEnv).
statements_eval(statement(Stmt), Env, FinalEnv) :-
    statement_eval(Stmt, Env, FinalEnv).

% Individual Statement Evaluation
% Statement evaluations for assignment
statement_eval(assign(data_type(_Type), var(Var), Expr), Env, FinalEnv) :-
    expression_eval(Expr, Env, Value),
    update_env(Env, Var, Value, FinalEnv).

statement_eval(assign(var(Var), Expr), Env, FinalEnv) :-
    expression_eval(Expr, Env, Value),
    update_env(Env, Var, Value, FinalEnv).
```

# TataScript Interpreter

```
% Statement evaluations for display statements
statement_eval(display(Output), Env, Env) :-
    output_eval(Output, Env, Str),
            strip_quotes(Str, Stripped),
    writeln(Stripped).

% If condition Evaluations
% If the first condition is true evaluate the block
statement_eval(if(Cond, ThenBlock, _ElseIf, _Else), Env, FinalEnv) :-
    condition_eval(Cond, Env, true),
    block_eval(ThenBlock, Env, FinalEnv), !.
```

# TataScript Interpreter

```
% If the first condition is false evaluate the chain of elseif and finally else statement blocks
statement_eval(if(Cond, _ThenBlock, ElseIf, Else), Env, FinalEnv) :-
    condition_eval(Cond, Env, false),
    elseif_blocks_eval(ElseIf, Env, Else, FinalEnv).

% For loop evaluation
statement_eval(for(Init, Cond, Incr, Block), Env, FinalEnv) :-
    statement_eval(Init, Env, InitEnv),
    for_loop_eval(Cond, Incr, Block, InitEnv, FinalEnv).

% While loop evaluation
statement_eval(while(Cond, Block), Env, FinalEnv) :-
    while_loop_eval(Cond, Block, Env, FinalEnv).
```

# TataScript Interpreter

```
% Increment evaluator ++
statement_eval(incr(var(Var), ++), Env, FinalEnv) :-
    lookup(Var, Env, Value),
    NewValue is Value + 1,
    update_env(Env, Var, NewValue, FinalEnv).

% Decrement evaluator --
statement_eval(incr(var(Var), --), Env, FinalEnv) :-
    lookup(Var, Env, Value),
    NewValue is Value - 1,
    update_env(Env, Var, NewValue, FinalEnv).
```

# TataScript Interpreter

```
% Ternary Condition evaluator with assignment
% When true
statement_eval(assign(data_type(_Type), var(Var), ternary(Cond, TrueExpr, _FalseExpr)),
Env, FinalEnv) :-
    condition_eval(Cond, Env, true),
    expression_eval(TrueExpr, Env, Result),
    update_env(Env, Var, Result, FinalEnv).

% When false
statement_eval(assign(data_type(_Type), var(Var), ternary(Cond, _TrueExpr, FalseExpr)),
Env, FinalEnv) :-
    condition_eval(Cond, Env, false),
    expression_eval(FalseExpr, Env, Result),
    update_env(Env, Var, Result, FinalEnv).
```

# TataScript Interpreter

```
% Ternary Assignment without Data Type
% When true
statement_eval(assign(var(Var), ternary(Cond, TrueExpr, _FalseExpr)), Env, FinalEnv) :-
    condition_eval(Cond, Env, true),
    expression_eval(TrueExpr, Env, Result),
    update_env(Env, Var, Result, FinalEnv).

% When false
statement_eval(assign(var(Var), ternary(Cond, _TrueExpr, FalseExpr)), Env, FinalEnv) :-
    condition_eval(Cond, Env, false),
    expression_eval(FalseExpr, Env, Result),
    update_env(Env, Var, Result, FinalEnv).
```

# TataScript Interpreter

```
% Display output without quotes
strip_quotes(Str, Stripped) :-
    atom_chars(Str, Chars),
    strip_quotes_helper(Chars, StrippedChars),
    atom_chars(Stripped, StrippedChars).

strip_quotes_helper(['"' | Rest], Stripped) :-  % Remove leading quote
    append(Stripped, ['"'], Rest), !.           % Remove trailing quote
strip_quotes_helper(Chars, Chars).

substitute_variables(Str, Env, FinalStr) :-
    atom_chars(Str, Chars),
    substitute_variables_in_chars(Chars, Env, SubstitutedChars),
    atom_chars(FinalStr, SubstitutedChars).
```

# TataScript Interpreter

```
% Handling vars in side display statements
substitute_variables_in_chars([], _, []).

substitute_variables_in_chars(['\\', '\'' | Rest], Env, ['\'' | SubstitutedRest]) :-
    substitute_variables_in_chars(Rest, Env, SubstitutedRest).

substitute_variables_in_chars(['\'' | Rest], Env, Result) :-
    extract_variable(Rest, VarChars, AfterVar),
    atom_chars(Var, VarChars),
    lookup(Var, Env, Value),
    atom_chars(Value, ValueChars),
    append(ValueChars, SubstitutedRest, Result),
    substitute_variables_in_chars(AfterVar, Env, SubstitutedRest).

substitute_variables_in_chars([Char | Rest], Env, [Char | SubstitutedRest]) :-
    Char \= '\\',
    substitute_variables_in_chars(Rest, Env, SubstitutedRest).

extract_variable(['\'' | Rest], [], Rest).
extract_variable([Char | Rest], [Char | VarChars], AfterVar) :-
    extract_variable(Rest, VarChars, AfterVar).
```

# TataScript Interpreter

```
% Block Evaluation
block_eval([], Env, Env).
block_eval(block(B), Env, FinalEnv) :-
    statements_eval(B, Env, FinalEnv).
check_non_zero(RightVal) :-
    RightVal \= 0.
check_non_zero(_) :-
    write("ERROR : Divide by Zero Exception"),
    halt.
```

```
% Expression Evaluation
expression_eval(add(L, R), Env, Value) :-
    expression_eval(L, Env, LeftVal),
    expression_eval(R, Env, RightVal),
    Value is LeftVal + RightVal.
expression_eval(sub(L, R), Env, Value) :-
    expression_eval(L, Env, LeftVal),
    expression_eval(R, Env, RightVal),
    Value is LeftVal - RightVal.
expression_eval(mul(L, R), Env, Value) :-
    expression_eval(L, Env, LeftVal),
    expression_eval(R, Env, RightVal),
    Value is LeftVal * RightVal.
expression_eval(divide(L, R), Env, Value) :-
    expression_eval(L, Env, LeftVal),
    expression_eval(R, Env, RightVal),
    check_non_zero(RightVal),
    Value is LeftVal // RightVal.
expression_eval(parenthesis(Expr), Env, Value) :-
    expression_eval(Expr, Env, Value).
expression_eval(num(Value), _, Value).
expression_eval(var(Var), Env, Value) :-
    lookup(Var, Env, Value).
expression_eval(string(Str), _, Str).
```

# TataScript Interpreter

```
% Ternary Expression Evaluation
expression_eval(ternary(Cond, TrueExpr, _FalseExpr), Env, Value) :-
    condition_eval(Cond, Env, true), % If condition evaluates to true
    expression_eval(TrueExpr, Env, Value).

expression_eval(ternary(Cond, _TrueExpr, FalseExpr), Env, Value) :-
    condition_eval(Cond, Env, false), % If condition evaluates to false
    expression_eval(FalseExpr, Env, Value).

% Output Evaluation
output_eval(string(Str), Env, FinalStr) :-
            substitute_variables(Str, Env, FinalStr), !.
output_eval(var(Var), Env, Value) :-
    lookup(Var, Env, Value).
output_eval(num(Num), _, Num).
```

# TataScript Interpreter

```
% Condition Evaluation
condition_eval(cond(L, Op, R), Env, true) :-
    expression_eval(L, Env, LeftVal),
    expression_eval(R, Env, RightVal),
    relational_eval(Op, LeftVal, RightVal, true).
condition_eval(cond(L, Op, R), Env, false) :-
    expression_eval(L, Env, LeftVal),
    expression_eval(R, Env, RightVal),
    relational_eval(Op, LeftVal, RightVal, false).
condition_eval(cond(X, and, Y), Env, true) :-
    condition_eval(X, Env, true),
            condition_eval(Y, Env, true).

condition_eval(cond(X, and, Y), Env, false) :-
    condition_eval(X, Env, false),
            condition_eval(Y, Env, true).

condition_eval(cond(X, and, Y), Env, false) :-
    condition_eval(X, Env, true),
            condition_eval(Y, Env, false).

condition_eval(cond(X, and, Y), Env, false) :-
    condition_eval(X, Env, false),
            condition_eval(Y, Env, false).

condition_eval(cond(X, or, Y), Env, true) :-
    condition_eval(X, Env, true),
            condition_eval(Y, Env, true).

condition_eval(cond(X, or, Y), Env, true) :-
    condition_eval(X, Env, false),
            condition_eval(Y, Env, true).

condition_eval(cond(X, or, Y), Env, true) :-
    condition_eval(X, Env, true),
            condition_eval(Y, Env, false).

condition_eval(cond(X, or, Y), Env, false) :-
    condition_eval(X, Env, false),
            condition_eval(Y, Env, false).
```

# TataScript Interpreter

```
% Relational Operators
relational_eval(<, L, R, true) :- L < R.
relational_eval(>, L, R, true) :- L > R.
relational_eval(==, L, R, true) :- L =:= R.
relational_eval(<=, L, R, true) :- L =< R.
relational_eval(>=, L, R, true) :- L >= R.
relational_eval(!, L, R, true) :- L \= R.
relational_eval(_, _, _, false).

% Loops Evaluation
% For loop evaluations
for_loop_eval(Cond, Incr, Block, Env, FinalEnv) :-
    (condition_eval(Cond, Env, true) ->
        block_eval(Block, Env, TempEnv),
        statement_eval(Incr, TempEnv, NextEnv),
        for_loop_eval(Cond, Incr, Block, NextEnv, FinalEnv)
    ; FinalEnv = Env).
```

```
% while loop evaluations
while_loop_eval(Cond, Block, Env, FinalEnv) :-
    condition_eval(Cond, Env, true),
    block_eval(Block, Env, TempEnv),
    while_loop_eval(Cond, Block, TempEnv, FinalEnv).

while_loop_eval(Cond, _Block, Env, Env) :-
    condition_eval(Cond, Env, false).
```

# TataScript Interpreter

```
% elseif chain evaluations
elseif_blocks_eval(elseif_block(X, _Y), Env, _Else,
FinalEnv) :-
        elseif_block_eval(X, Env, FinalEnv, true).

elseif_blocks_eval(elseif_block(X, Y), Env, Else,
FinalEnv) :-
        elseif_block_eval(X, Env, _Env2, false),
        elseif_blocks_eval(Y, Env, Else, FinalEnv).

elseif_blocks_eval([], Env, Else, FinalEnv) :-
        else_block_eval(Else, Env, FinalEnv).

elseif_block_eval(elseIf(C, B), Env, FinalEnv, true) :-
        condition_eval(C, Env, true),
        block_eval(B, Env, FinalEnv).

elseif_block_eval(elseIf(C, _B), Env, Env, false) :-
        condition_eval(C, Env, false).

else_block_eval(else(B), Env, FinalEnv) :-
    block_eval(B, Env, FinalEnv).

else_block_eval([], Env, Env).
```

# Steps to run the script

- For Mac and Linux:
  - bash tata.sh <path_to_.tts file> OR
  - ./tata.sh <path_to_.tts file>
  - Example : bash tata.sh assignment.tts OR ./tata.sh assignment.tts
- For Windows
  - tata.bat <path_to_.tts file>
  - Example: tata.bat assignment.tts

# Assignment & Display

- Assignment Syntax : <datatype> variable_name = expression.
- Examples
  - int x = 42 + 3.
  - int a = 10.
  - string msg = "Hello World"
- Display Syntax: display -> statements_to_display.
- Examples:
  - display -> “Hello World”.
  - display -> 42.
  - display -> x.
  - display -> “Variable is ‘x’”

# Screenshots

```
;
    int a = 42 + 4 / 2.
    int b = (a - 5) + a.
    int c = ( 50 / 10 / 5 + 5) - ( a + b ).
    string msg = "This is so awesome".
    string mango = "tasty".
    display -> "a : 'a'".
    display -> "b : 'b'".
    display -> "c : 'c'".
    display -> "msg : 'msg'".
    display -> "mango : 'mango'".
;       You, 7 days ago • Minor fixes
```

```
/Users/arun/Documents/Coding/SER502-TataScript-Team13 [branch_arun]% ./tata.sh data/assign_complex.tts
a : 44
b : 83
c : -121
msg : "This is so awesome"
mango : "tasty"
```

# Conditional Statement

- Syntax for if:

  if -> <conditions>

  { conditional_statements.}

  Eg) if -> x > 3

  { display -> "x is greater".}

- Syntax for if-else:

  if -> <conditions>

  { conditional_statements.}

  <- { conditional_statements.}

  Eg) int x = 9.

  if -> x > 3 {

  display -> "x is greater".

  } <- {

  display -> "x is less".

  }

# Conditional Statement

- Syntax for if else-if else:

  if -> <conditions>

  { conditional_statements.}

  <- -> <conditions> { conditional_statements.}

  <- { conditional_statements.}

Eg. int x = 1.

```
if -> x > 10 {
  display -> "x is greater than 10".
} <- -> x > 5 {
  display -> "x is greater than 5".
} <- {
  display -> "x is less than or equal to 5". }
```

# Screenshots

```
;
    int x = 9.
    if -> x > 3 {
        display -> "x is greater".
    } <- {
        display -> "x is less".
    }
;       You, 7 days ago • Fixed some examples, removed u
```

```
/Users/arun/Documents/Coding/SER502-TataScript-Team13 [branch_arun]% ./tata.sh data/if_else_stmt.tts
x is greater
```

# Screenshots

```
;
    int x = 1.
    if -> x > 10 {
        display -> "x is greater than 10".
    } <- -> x > 5 {
        display -> "x is greater than 5".
    } <- {
        display -> "x is less than or equal to 5".
    }
;       You, 7 days ago • Fixing Merge Conflicts
```

```
/Users/arun/Documents/Coding/SER502-TataScript-Team13 [branch_arun]% ./tata.sh data/if_elif_stmt.tts
x is less than or equal to 5
```

# Screenshots

```
;
    int x = 1.
    int y = 2.
    if -> x > 0 {
        if -> y > 0 {
            display -> "both positive".
        }
    }
;       You, 7 days ago • Fixing Merge Conflicts
```

```
/Users/arun/Documents/Coding/SER502-TataScript-Team13 [branch_arun]% ./tata.sh data/nested_if.tts
both positive
```

# Conditonal with Compound

- Syntax with ‘&’:

```
if -> <condition> & <condition>{
        Conditional_statements.
        }
```

```
Eg: int x = 1.
   int y = 5.
   if -> x > 0 & y < 10 {
       display -> "condition met".
   }
```

- Syntax with ‘|’:

```
if -> <condition> | <condition>{
        Conditional_statements.
        }
```

```
Eg: int x = 1.
   int y = 9.
   if -> x > 0 | y < 10 {
       display -> "condition met".
   }
```

# Screenshots

```
;
    int x = 1.
    int y = 5.
    if -> x > 0 & y < 10 {
        display -> "condition met".
    }
;
```

```
/Users/arun/Documents/Coding/SER502-TataScript-Team13 [branch_arun]% ./tata.sh data/if_and_stmt.tts
condition met
```

# Screenshots

```
;
    int x = 1.
    int y = 9.
    if -> x > 0 | y < 10 {
        display -> "condition met".
    }
;       You, 7 days ago • Fixing Merge Conflicts
```

```
/Users/arun/Documents/Coding/SER502-TataScript-Team13 [branch_arun]% ./tata.sh data/if_or_stmt.tts
condition met
```

# Ternary Operations

- Syntax:
  var_name = <conditional> -> <True_expression> <- <False_expression>.

Eg:
```
int apple = 3.
int mango = 10.
int difference = 0.
difference = apple < 5 -> apple - mango  <- apple + mango.
display -> difference.
```

# Screenshots

```
;
    int apple = 3.
    int mango = 10.
    int difference = 0.
    difference = apple < 5 -> apple - mango  <- apple + mango.
    display -> difference.
;       Kavya Parekh, 7 days ago • ternary example updated
```

```
/Users/arun/Documents/Coding/SER502-TataScript-Team13 [branch_arun]% ./tata.sh data/ternary.tts
-7
```

# Loops (for and while)

- Syntax for For-loops:

```
for -> <initiator>:<condition>:<iterator>
        { statements.}

Eg) for -> i = 0 : i < 3 : i++
                { display -> i.
                display -> "it". }
```

- Syntax for while-loops:

```
while -> <condition>
        { statements.}

Eg) int count = 10.
    while -> count > 0 {
        display -> "Count \'count\'".
        if -> count == 5 {
            display -> "Count is 5".
          }
    }
```

# Screenshots

```
;
    for -> i = 1 : i <= 10 : i++ {
        if -> i > 0 & i < 3 {
            display -> "Small Number".
        } <- -> i >=3 & i < 7 {
            display -> "Medium number".
        } <- {
            display -> "Large Number".
        }
    }
;       You, 7 days ago • Fixing Merge Conflicts
```

```
/Users/arun/Documents/Coding/SER502-TataScript-Team13 [branch_arun]% ./tata.sh data/for_multiple.tts
Small Number
Small Number
Medium number
Medium number
Medium number
Medium number
Large Number
Large Number
Large Number
Large Number
```

# Screenshots

```
;
    int count = 10.
    while -> count > 0 {
        display -> "Count 'count'".
        if -> count == 5 {
            display -> "Count is 5".
            count-- .
        } <- -> count == 10 {
            display -> "Count is 10".
            count-- .
        } <- {
            count-- .
        }
    }
;       You, 7 days ago • Fixing Merge Conflicts
```

```
/Users/arun/Documents/Coding/SER502-TataScript-Team13 [branch_arun]% ./tata.sh data/while.tts
Count 10
Count is 10
Count 9
Count 8
Count 7
Count 6
Count 5
Count is 5
Count 4
Count 3
Count 2
Count 1
```

# Miscellaneous Examples - Factorial

```
;
    int n = 5.
    int result = 1.
    display -> "Calculating factorial for: 'n'".

    if -> n < 0 {
        display -> "Factorial not defined for negative numbers".
    }
    <- -> n == 0 {
        display -> "Factorial of 0 is 1".
    } <- {
        display -> "Intermediate results:".
        for -> i = 1 : i <= n : i++ {
            result = result * i.      Arun, 4 weeks ago • tokenizer, gra
            display -> "Factorial of 'i' is 'result'".
        }

    display -> "Final result: 'result'".
} ;
```

```
/Users/arun/Documents/Coding/SER502-TataScript-Team13 [branch_arun]% ./tata.sh data/factorial.tts
Calculating factorial for: 5
Intermediate results:
Factorial of 1 is 1
Factorial of 2 is 2
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
Final result: 120
```

# Miscellaneous Examples - Fibonacci

```
;
    int n = 10.
    int prev = 0.
    int current = 1.
    int count = 0.

    display -> "Generating Fibonacci Sequence".
    display -> "Number of terms: 'n'".

    if -> n <= 0 {
        display -> "Please enter a positive number".
    } <- -> n == 1 {
        display -> "First Fibonacci number: 'prev'".
    }   <- {
        display -> "Fibonacci Sequence:".

        display -> "Term : 'count' : 'prev'".

        count = count + 1.      Arun, 4 weeks ago • tokenizer, g
        display -> "Term : 'count' : 'current'".

        for -> i = 2 : i < n : i++ {
            int next = prev + current.
            count = count + 1.

            display -> "Term : 'count' : 'next'".

            prev = current.
            current = next.
        }

        display -> "Sequence generation complete.".
    }
;
```

```
/Users/arun/Documents/Coding/SER502-TataScript-Team13 [branch_arun]% ./tata.sh data/fibonacci.tts
Generating Fibonacci Sequence
Number of terms: 10
Fibonacci Sequence:
Term : 0 : 0
Term : 1 : 1
Term : 2 : 1
Term : 3 : 2
Term : 4 : 3
Term : 5 : 5
Term : 6 : 8
Term : 7 : 13
Term : 8 : 21
Term : 9 : 34
Sequence generation complete.
```

# Miscellaneous Examples - Palindrome

```
;
int n = 10201.
int original = n.
int reverse = 0.
int remainder = 0.
display -> "Checking if number is a palindrome: 'n'".
int temp = n.
if -> n < 0 {
    display -> "Error: Number cannot be negative to calculate palindrome".
} <- {
    int temp = n.
    while -> temp > 0 {
        remainder = temp - (temp / 10) * 10.
        reverse = reverse * 10 + remainder.
        temp = temp / 10.
    }
    if -> original == reverse {
        display -> "The number is a palindrome".
    } <- {
        display -> "The number is not a palindrome".
    }
}
;      dhruvamalik123, 7 days ago • new files added for example programs
```

```
/Users/arun/Documents/Coding/SER502-TataScript-Team13 [branch_arun]% ./tata.sh data/palindrome.tts
Checking if number is a palindrome: 10201
The number is a palindrome
```

# THANK YOU