A REPORT ON

Designing Language for Compiler Construction

IN PARTIAL FULFILLMENT OF

COURSE PROJECT

FOR

Compiler Construction (CS F363) | Compiler Design (IS F342)

SUBMITTED BY

Group 1	UTKARSH PATHRABE 2012A7PS034P SHIKHAR VASHISHTH 2012C6PS436P
Group 2	KRISHNA KANT GARG 2012A7PS033P YASH SINHA 2012C6PS365P

STUDENTS AT

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

(8TH Feb, 2015)

Table of Contents

1. LANGUAGE FEATURES:

- Data types: Integer, Real, Boolean, String, Character, Record, Matrix (1D & 2D)
- Operations for data types:
  - Integer - addition, subtraction, division, multiplication, exponent, comparison.
  - Real - addition, subtraction, division, muliplication, exponent, comparison.
  - Boolean - equality check, logical not operation.
  - Char - addition, subtraction, comparison.
  - String - concatenation, length, character at position.
  - Matrix - addition, number of rows and columns, accessing element
  - Record - operation on elements based on data type mentioned above.
- Functions:
  - Allows more than one return values.
  - Nested functions are not allowed.
  - Basic data types - int, real, char, bool can be passed as arguments and returned from functions.
  - Arguments are passed by value.
  - Seperate section of function declaration to avoid confusion.
- Scope rules:
  - Static scoping.
- Conditional Statement:
  - If else statement with keyword: if, elif, else, endif.
- Iterative statement:
  - Loop structure: loop (intialize) (condition) (update): .... endloop.
  - Allows programmers to define condition with break and continue statements.
- I/O operations:
  - get - gets the value of variable according to its data type from stdin.
  - put - prints the value of variable, string literal, tab, newline to stdout.
- Expression:
  - Operator precedence same as C.
  - Arithmetic operators : +, -, /, *, %, **(pow)
  - Relational operators: <, >, <=, >=, ==, !=
  - Logical operators: AND, OR, NOT
- Assignment statement:
  - Allows matrix and string initialization.
  - Allows assignment of basic data types.
- Strongly typed language.
- Doesn't support type conversion.
- Same syntax for single and multi line comments:
  - Syntax : /* Comment here */
- Allows formatting code using tabs to improve readability of code.

2. LEXICAL UNITS:

Patterns	Token	Purpose
Execute	TK_EXECUTE	Execute block begin
:	TK_COLON	Colon operator
end	TK_END	Execute block end
records	TK_RECORDS	Records block begin
endrecords	TK_ENDRECORDS	Records block end
procedures	TK_REC	Procedures block begin
endprocedures	TK_ID	Procedures block end
rec	TK_ENDREC	Record declaration begin
endrec	TK_PROCS	Record declaration end
proc	TK_ENDPROCS	Procedure declaration begin
endproc	TK_PROC	Procedure declaration end
(	TK_LPAREN	Left parenthesis
)	TK_RPAREN	Right parenthesis
->	TK_ARROW	Arrow Operator
endproc	TK_ENDPROC	Ends a procedure declaration
,	TK_COMMA	Comma Operator
;	TK_SEMICOLON	Semicolon Operator
string	TK_STRING	String Data Type
mat	TK_MAT	Matrix Data Type
<	TK_LT	Less Than Operator
>	TK_GT	Greater Than Operator
[	TK_LSQ	Left square bracket
]	TK_RSQ	Right square bracket
int	TK_INT	Int Data Type
real	TK_REAL	Real Data Type
bool	TK_BOOL	Bool Data Type
char	TK_CHAR	Char Data Type
(?:\d)?\d+	TK_INT_LIT	Int Literal
if	TK_IF	If statement
elif	TK_ELIF	Else If statement
else	TK_ELSE	Else statement
endif	TK_ENDIF	EndIf statement
get	TK_GET	Read from STDIN
put	TK_PUT	Print to STDOUT
endl	TK_ENDL	New Line Symbol
tab	TK_TAB	Tab Symbol
"	TK_DQUOTE	Double Quotes
([a-zA-Z]+[a-zA-Z0-9]*)	TK_STR_LIT	String Literal
.	TK_DOT	Dot Operator
call	TK_CALL	Procedure Call Begin
loop	TK_LOOP	Loop Begin
endloop	TK_ENDLOOP	Loop end
break	TK_BREAK	Break statement
continue	TK_CONTINUE	Continue statement

NOT	TK_NOT	Not Logical operator
[(True|False)]	TK_BOOL_LIT	Boolean constant
AND	TK_AND	And logical operator
OR	TK_OR	Or logical operator
<=	TK_LE	Less than equal rel. op.
>=	TK_GE	Greater than equal rel op.
==	TK_EQ	Equal relational operator
!=	TK_NEQ	Not equal relational operator
(?:\d*\.)?\d+	TK_REAL_LIT	Real constant
‘	TK_SQUOTE	Single quote
\p{L}	TK_CHAR_LIT	Unicode character property class that describes the Unicode characters that are letters
assign	TK_ASSIGN	Assign begin operator
=	TK_ASSIGN_OP	Assign operator
-	TK_MINUS	Subtraction operator
+	TK_PLUS	Addition operator
*	TK_MUL	Multiply operator
/	TK_DIV	Division operator
%	TK_MOD	Modulo operator
**	TK_POW	Power operator
matassign	TK_MATASSIGN	Matrix assign begin operator
{	TK_LCURL	Left Curly Bracket
}	TK_RCURL	Right Curly Bracket
@rows	TK_MAT_ROWS	Matrix row operator
@cols	TK_MAT_COLS	Matrix column operator
strassign	TK_STRASSIGN	String assign begin operator
@length	TK_STR_LENGTH	String length operator

3. LL (1) GRAMMAR:

/* MAIN PROGRAM */
<program> -> <_records> <_procedures> <execute>
<execute> -> EXECUTE COLON <stmts> END

/* RECORD DECLARATION */
<_records> -> <records>
<_records> -> EPSILON
<records> -> RECORDS COLON <recs> ENDRECORDS

<recs> -> <rec> <_recs>
<_recs> -> <recs>
<_recs> -> EPSILON
<rec> -> REC ID COLON <decl_stmts> ENDREC

<_procedures> -> <procedures>
<_procedures> -> EPSILON

/* PROCEDURE DECLARATION */
<procedures> -> PROCS COLON <procs> ENDPROCS
<procs> -> <proc> <_procs>
<_procs> -> <procs>
<_procs> -> EPSILON

<proc> -> PROC ID LPAREN <param_list> RPAREN ARROW LPAREN <param_list> RPAREN COLON <stmts> ENDPROC

<param_list> -> <basic_type> ID <_param_list>

<_param_list> -> COMMA <param_list>
<_param_list> -> EPSILON

/* STATEMENT */
<stmts> -> <stmt> <_stmts>

<_stmts> -> <stmts>
<_stmts> -> EPSILON

<stmt> -> <decl_stmt>
<stmt> -> <cond_stmt>
<stmt> -> <loop_stmt>
<stmt> -> <proc_call_stmt>
<stmt> -> <assign_stmt>
<stmt> -> <io_stmt>
<stmt> -> <mat_assign_stmt>
<stmt> -> <str_assign_stmt>

/* DECLARATION STATEMENT */
<decl_stmts> -> <decl_stmt> <_decl_stmts>
<_decl_stmts> -> <decl_stmts>
<_decl_stmts> -> EPSILON
<decl_stmt> -> <type> <id_list> SEMICOLON

<type> -> <basic_type>

```
<type> -> STRING
<type> -> MAT LT <basic_type> GT LSQ <dims> RSQ
<type> -> REC ID

<basic_type> -> INT
<basic_type> -> REAL
<basic_type> -> BOOL
<basic_type> -> CHAR

<id_list> -> ID <_id_list>

<_id_list> -> COMMA <id_list>
<_id_list> -> EPSILON

<dims> -> INT_LIT <_int_lit>

<_int_lit> -> COMMA INT_LIT
<_int_lit> -> EPSILON

/* CONDITIONAL STATEMENT*/
<cond_stmt> -> IF LPAREN <conds> RPAREN COLON <stmts> <else_if> <_cond_stmt>

<else_if> -> ELIF LPAREN <conds> RPAREN COLON <stmts> <else_if>
<else_if> -> EPSILON

<_cond_stmt> -> ELSE COLON <stmts> ENDIF
<_cond_stmt> -> ENDIF

/* IO STATEMENT */
<io_stmt> -> GET LPAREN <var> RPAREN SEMICOLON
<io_stmt> -> PUT LPAREN <put_param> RPAREN SEMICOLON

<put_param> -> <var>
<put_param> -> ENDL
<put_param> -> TAB
<put_param> -> DQUOTE STR_LIT DQUOTE

<var> -> ID <_var>
<_var> -> LSQ <dims> RSQ
<_var> -> DOT ID
<_var> -> EPSILON

/* PROCEDURE CALL STATEMENT */
<proc_call_stmt> -> CALL ID LPAREN <proc_arg_list> RPAREN ARROW LPAREN
<proc_arg_list> RPAREN SEMICOLON

<proc_arg_list> -> <arg_list>
<proc_arg_list> -> EPSILON

<arg_list> -> ID <arg_list1>
<arg_list1> -> COMMA <arg_list>
<arg_list1> -> LSQ <dims> RSQ <arg_list2>
<arg_list1> -> DOT ID <arg_list2>
<arg_list1> -> EPSILON
<arg_list2> -> COMMA <arg_list>
```

<arg_list2> -> EPSILON

/* **ITERATIVE STATEMENT** */
<loop_stmt> -> LOOP LPAREN <assign_list> RPAREN LPAREN <conds> RPAREN LPAREN <assign_list> RPAREN COLON <iloop_stmts> ENDLOOP

<iloop_stmts> -> <iloop_stmt> <iloop_stmts>
<iloop_stmts> -> EPSILON
<iloop_stmt> -> <stmt>
<iloop_stmt> -> BREAK LPAREN <conds> RPAREN SEMICOLON
<iloop_stmt> -> CONTINUE LPAREN <conds> RPAREN SEMICOLON

<assign_list> -> <assign_stmt> <assign_list>
<assign_list> -> EPSILON

/* **CONDITION** */
<conds> -> LPAREN <conds> RPAREN <logical_op> LPAREN <conds> RPAREN
<conds> -> <elem> <rel_op> <elem>
<conds> -> NOT LPAREN <conds> RPAREN
<conds> -> BOOL_LIT

<logical_op> -> AND
<logical_op> -> OR

<rel_op> -> LT
<rel_op> -> GT
<rel_op> -> LE
<rel_op> -> GE
<rel_op> -> EQ
<rel_op> -> NEQ

<elem> -> <var>
<elem> -> INT_LIT
<elem> -> REAL_LIT
<elem> -> SQUOTE CHAR_LIT SQUOTE

/* **ASSIGNMENT STATEMENT** */
<assign_stmt> -> ASSIGN <var> ASSIGN_OP <expr> SEMICOLON

<expr> -> <term> <_expr>
<_expr> -> <op_-+> <term> <_expr>
<_expr> -> EPSILON

<term> -> <expo_term> <_term>
<_term> -> <op_*/%> <expo_term> <_term>
<_term> -> EPSILON

<expo_term> -> <factor> <_expo_term>
<_expo_term> -> <op_**> <factor> <_expo_term>
<_expo_term> -> EPSILON

<factor> -> LPAREN <expr> RPAREN
<factor> -> <var>
<factor> -> <const>
<factor> -> MINUS LPAREN <expr> RPAREN

<op_-+> -> PLUS
<op_-+> -> MINUS
<op_*/%> -> MUL
<op_*/%> -> DIV
<op_*/%> -> MOD
<op_**> -> POW

/* MATRIX STATEMENT */
<mat_assign_stmt> -> MATASSIGN ID ASSIGN_OP <mat_stmt>
<mat_stmt> -> LCURL <row_list> RCURL SEMICOLON
<mat_stmt> -> ID <_mat_stmt>
<_mat_stmt> -> PLUS ID SEMICOLON
<_mat_stmt> -> MAT_ROWS SEMICOLON
<_mat_stmt> -> MAT_COLS SEMICOLON

<row_list> -> <row> <_row_list>
<_row_list> -> SEMICOLON <row> <_row_list>
<_row_list> -> EPSILON

<row> -> <const> <_row>
<_row> -> COMMA <const> <_row>
<_row> -> EPSILON

<const> -> INT_LIT
<const> -> REAL_LIT
<const> -> BOOL_LIT
<const> -> SQUOTE CHAR_LIT SQUOTE

/* STRING STATEMENT */
<str_assign_stmt> -> STRASSIGN ID ASSIGN_OP <str_stmt>
<str_stmt> -> DQUOTE STR_LIT DQUOTE SEMICOLON
<str_stmt> -> ID <_str_stmt>
<_str_stmt> -> SEMICOLON
<_str_stmt> -> PLUS ID SEMICOLON
<_str_stmt> -> LSQ <ind> RSQ SEMICOLON
<_str_stmt> -> STR_LENGTH SEMICOLON
<ind> -> ID
<ind> -> INT_LIT

4. TEST CASES:

Test Case 1: Operators

```
/* Testcase 1 - operators*/

records:
        rec student:
                int rollno;
                int marks;
        endrec
endrecords

execute:
        int a,b;
        real c,d,power;
        assign a = 10;
        assign b = 20;

        rec student st1;

        assign st1.rollno = 32;
        assign st1.marks = 56;

        put(st1.marks);
        put(endl);

        assign sum = a + b;
        put("Sum :");
        put(sum);
        put(endl);

        assign power = c ** d;
        put("Power :");
        put(power);
        put(endl);
end
```

Test Case 2: Conditionals

```
execute:
        int marks;
        get(marks);

        if(marks >= 90):
                put("Grade: A");
        elif( (marks < 90) AND (marks >= 80) ):
                put("Grade: B");
        else:
                put("Grade: C");
        endif
        put(endl);
```

```
end
```

Test Case 3: Loop

```
execute:
        mat<int>[2,2] A;
        int i,j;
        matassign A = {1, 2;
                                        3, 4};
        loop(assign i=0)(i<2)(assign i=i+1):
                loop(assign j=0)(j<2)(assign j=j+1):
                        put(A[i,j]);
                        put(tab);
                endloop
                put(endl);
        endloop
end
```

Test Case 4: Scope

```
procs:
        proc square(int num)->(int sqr):
                assign sqr = num * num;
                assign sqr = sqr * times;
        endproc
endprocs

execute:
        int num, times, n, res;
        assign num = 10;
        assign times = 2;
        assign n = 20;

        call square(n)->(res);
        put("Result: ");
        put(res);
end
```

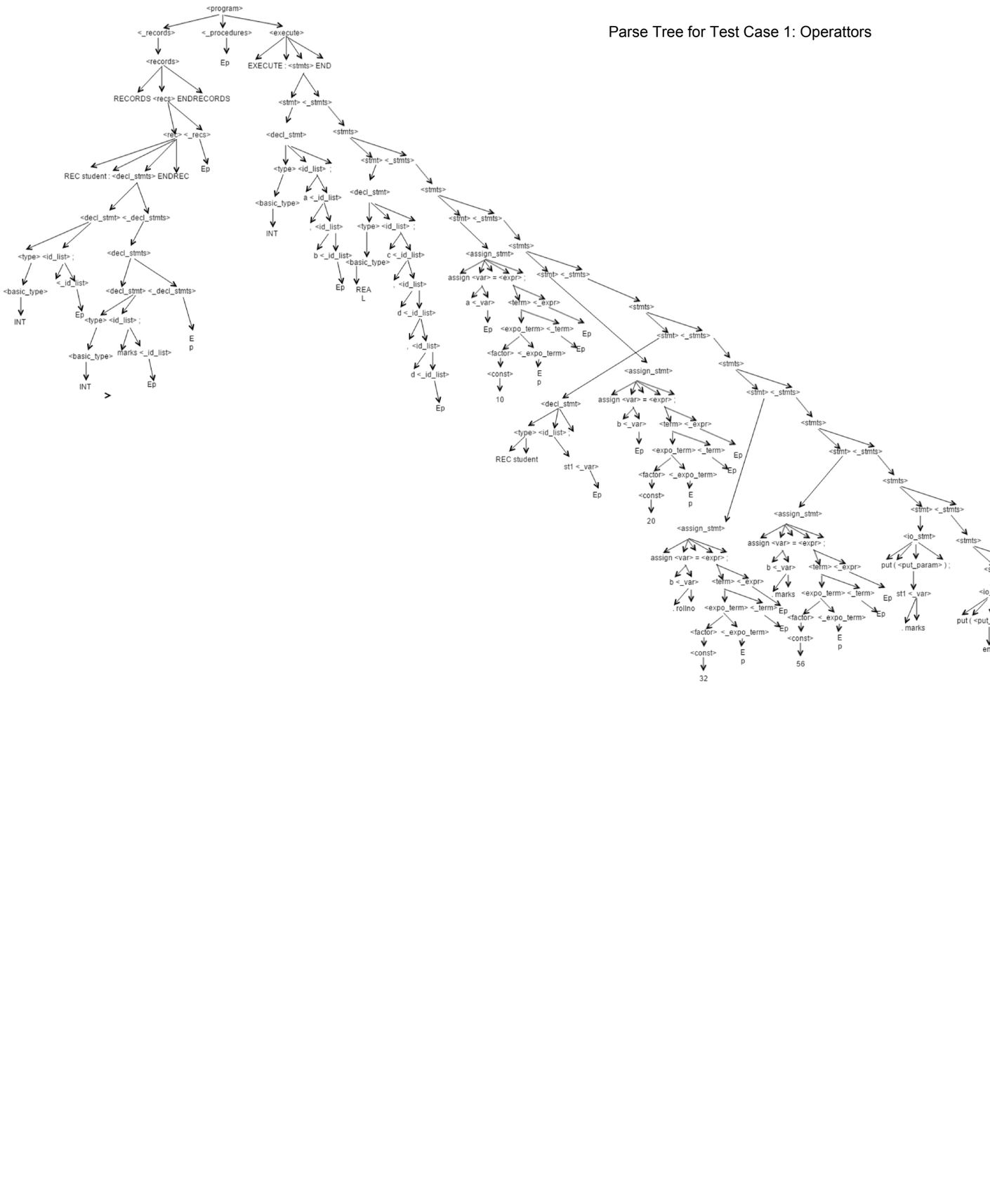
Test Case 5: Procedures

```
procs:
        proc calculateInterest(real principal, real rate, real time)->(real interest, real amount):
                assign interest = principal * (rate/100) * time ;
                assign amount = principal + interest;
        endproc
endprocs

execute:
        real P,R,T;
        assign P = 10000.0;
        assign R = 7.5;
        assign T = 4.0;

        real I, A;
```

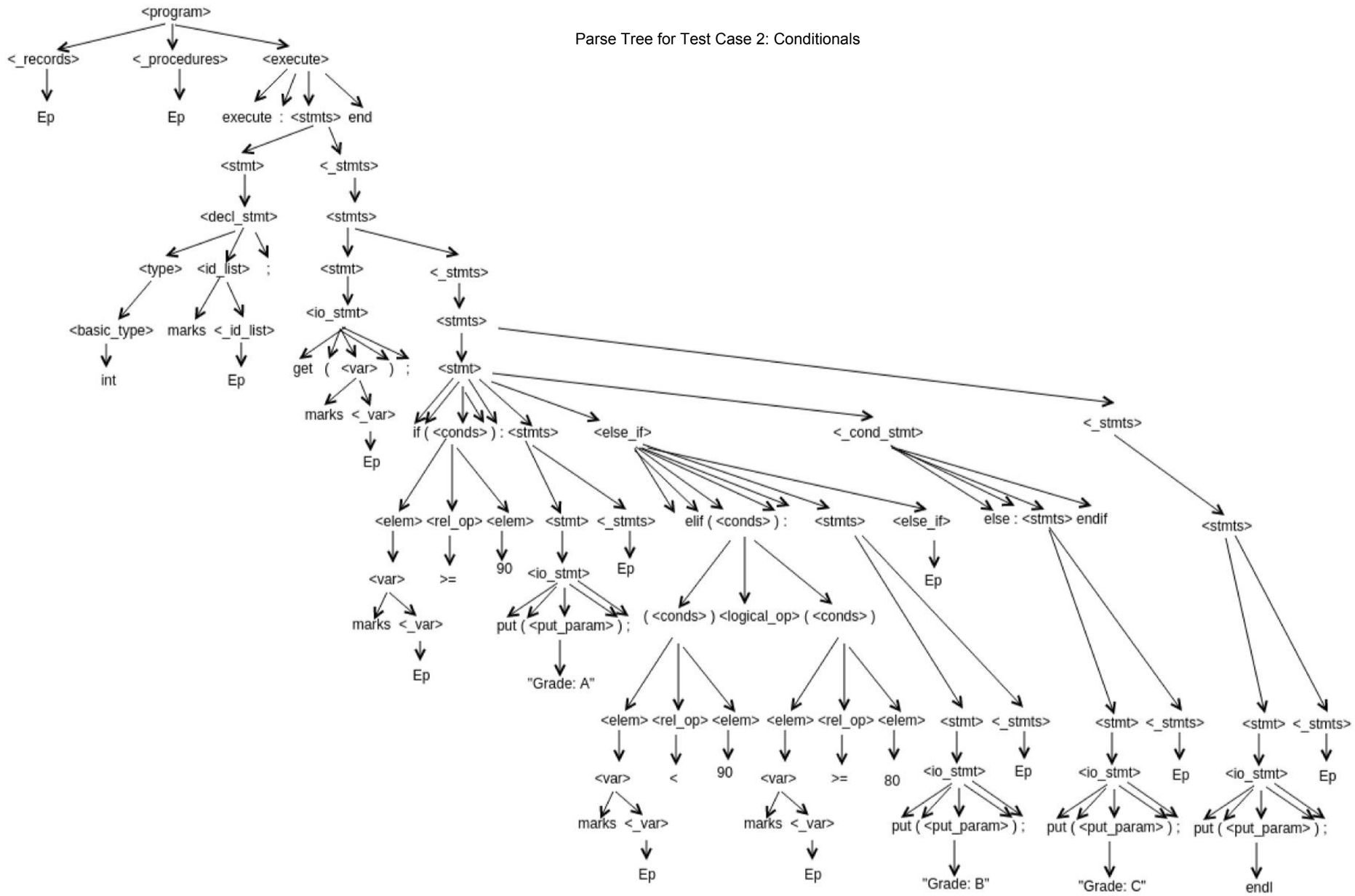
```
        call calculateInterest(P, R, T)->(I, A);

        put("Interest :");
        put(I);
        put(endl);
        put("Amount :");
        put(A);
end
```

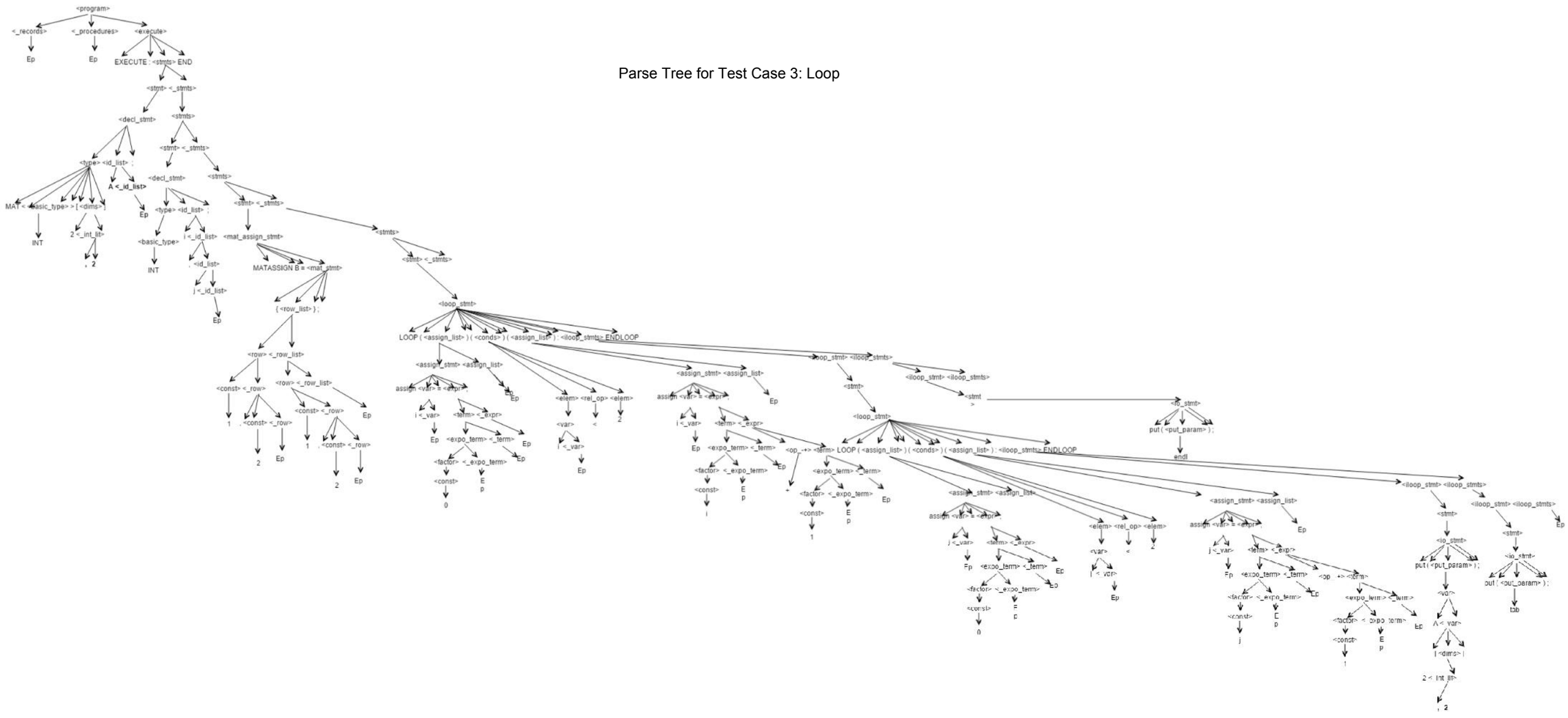
5. DERIVATION OF TEST CASES:

Parse Tree for Test Case 1: Operattors



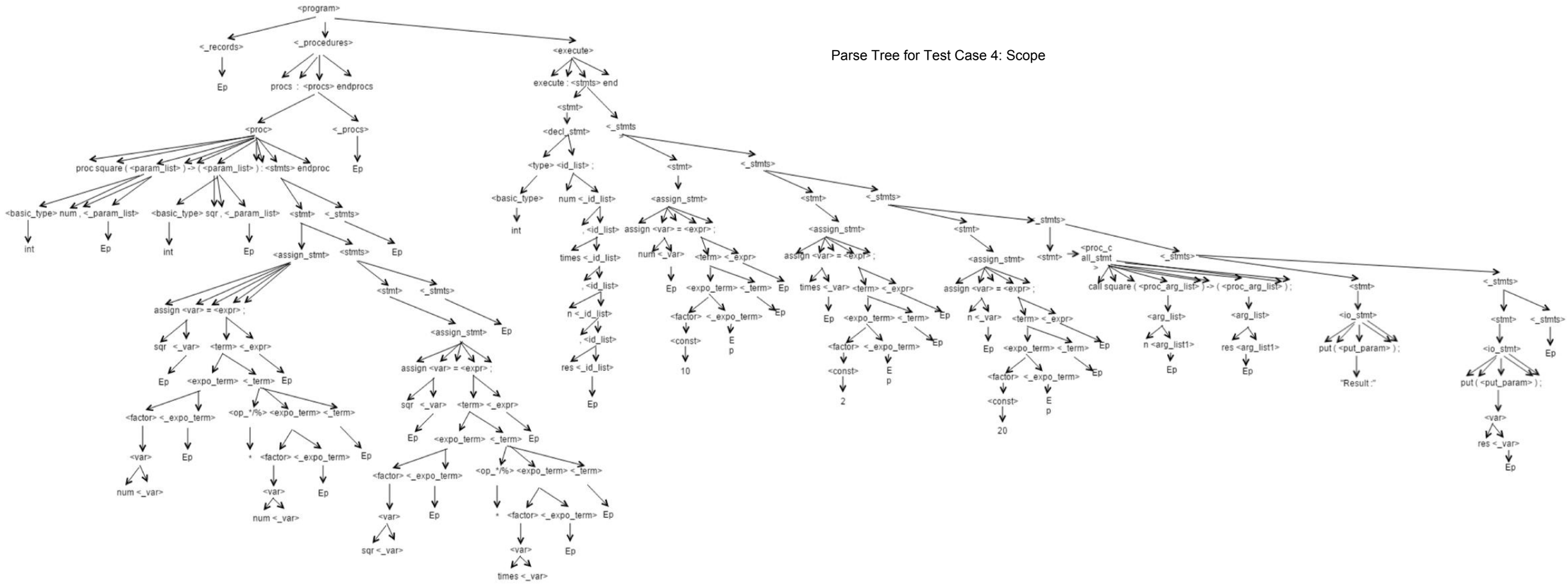
Parse Tree for Test Case 2: Conditionals





Parse Tree for Test Case 3: Loop

Parse Tree for Test Case 4: Scope



Parse Tree for Test Case 5: Procedures