Principles of Programming Languages (CS F301)

Assignment 1: Language design and type expression computation

Language description

The language is specifically designed to compute expressions involving arithmetic and logical operators. The operands can be of different types such as ***integer, real, boolean and arrays***. The language implements two types of arrays- ***rectangular and jagged***. The rows of rectangular arrays have same number of columns, and that of jagged arrays can have varying number of columns. A jagged array can be two or three dimensional while the rectangular arrays can be of any dimensions. The ***rectangular arrays can be static or dynamic*** in nature depending upon the range values available at compile time or run time respectively, but the ***jagged arrays are always static***.

The program uses keywords, variable identifiers, special symbols etc. The variable identifier name can be of at most 20 characters, including alphabet, digits and underscore, but it never starts with a digit. The symbols used are (,), {, }, :, ;, [,] etc. The language has only ***two types of statements – declaration and assignment statements***. A ***declaration statement*** can declare the type of one variable or type of the list of variables. The keywords used in this language are program, declare, list, of, variables, array, size, values, jagged, of, integer, real and boolean. An ***assignment statement*** has a left valued variable and a right hand side expression. The ***expression*** can be arithmetic or Boolean, and is constructed recursively. The expression uses arithmetic operators such as plus, minus, multiplication and division. The operators * and / have more precedence than the + and -. The Boolean operators &&& and ||| (used for logical ‘and’ and ‘or’ respectively) can only be applied to two variables of Boolean type. The logical ‘and’ has more precedence than logical ‘or’. The language does not have any relational operator such as less than, greater than etc. The values for variables are not initialized for integer, real and Boolean data types. However, the array variables are initialized using the description given below. All declaration statements must appear before any assignment statement in the program. This means that no assignment statement can be written before a declaration statement. The program must have at least one declaration statement and one assignment statement.

Static constants are the numbers available statically at code level. The language supports only integer numbers at text level as static constant. This means any number such as 234, 65, 53458754, 01981731, 000, 45, 16 and 7777 are valid numbers of type integers. The static real numbers are not valid at text level. This means 23.45 is not valid in the source code. Similarly, the language does not support true and false as Boolean constants.

The ***delimiter*** used in the program to separate the significant entities is the ***blank space***. This delimiter is imposed to save you from implementing the deterministic finite automaton for pattern matching by reaching the accept state, which you will implement in your Compiler Construction course. Any additional white space such as tab or an empty line is ignored while reading the source text.

***Primitive data type*:** The variables of integer, real and Boolean type are declared as follows

```
declare v1 : integer ;
```

declare list of variables a1 a2 : integer ;
declare list of variables b1 b2 b3 : boolean ;
declare list of variables c1 c2 : real ;

If the declaration statement has only one variable, then it is declared without the keywords list, of and variables as above.

Array types: There are two types of arrays in this language- *rectangular and jagged*. The array's basic element type can only be *integer*. The language does not allow array type for an element of another array. The elements of the arrays can be added, subtracted, multiplied and divided in ways similar to any variables of integer type. More details on the operations are given in a separate section later.

Rectangular array

A rectangular array is the usual multi-dimensional array with number of elements same for all rows. The user specifies the ranges in each dimension appropriately. For example, variables u and v of the two dimensional rectangular array is declared as given below

declare list of variables u v : array [2 .. 5] [3 .. 6] of integer ;

The range can also be described in rectangular arrays using variable identifier names thereby making the nature of the array dynamic. Example declaration statement is as follows

declare list of variables u v : array [low_1 .. high_1] [low_2 .. high_2] of integer ;

The names low_1, high_1, low_2 and high_2 should be declared by the programmer to be of type integer. If any one of these is not of integer type, then there is type error. The ranges can also be defined using any combination of integer variables and integer numbers. For example,

declare list of variables u v : array [126 .. high_1] [low_2 .. 653] of integer ;

A three dimensional rectangular array has ranges defined for all three dimensions. For example,

declare list of variables u v : array [126 .. high_1] [low_2 .. 653] [low_3 .. high_4] of integer ;

Jagged array

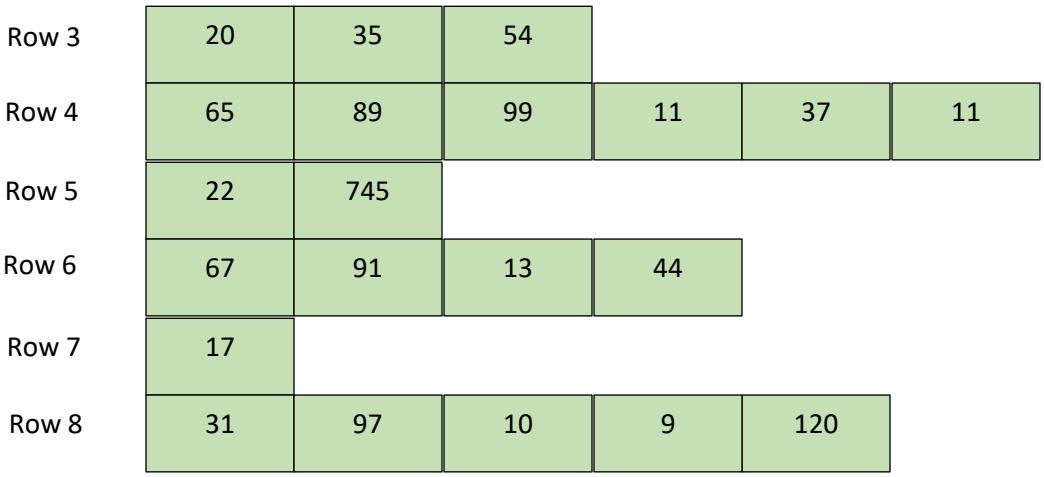
A jagged array has varying number of elements in each row. This extends to more dimensions as well. However, this language implements jagged arrays only in two and three dimensions. The ranges for second and the third dimensions for jagged array range from 1 to their sizes in R2 and R3 directions, while the indices for R1 are defined by the user appropriately.

A two dimensional jagged array is declared as given below.

declare list of variables s1 s2 : jagged array [3 .. 8] [ ] of integer ;
R1 [3] : size 3 : values { 20 ; 35 ; 54 }
R1 [4] : size 6 : values { 65 ; 89 ; 99 ; 11 ; 37 ; 11 }

R1 [5] : size 2 : values { 22 ; 745 }
R1 [6] : size 4 : values { 67 ; 91 ; 13 ; 44 }
R1 [7] : size 1 : values { 17 }
R1 [8] : size 5 : values { 31 ; 97 ; 10 ; 9 ; 120 }

The above two dimensional jagged array is displayed graphically.



A three dimensional jagged array is defined as an array with dimensions R1, R2 and R3. The array can have varying number of elements in the direction of R2 and R3.

To illustrate using an example, a three dimensional jagged array is declared as below

declare list of variables s1 s2 s3: jagged array [4 .. 7] [ ] [ ] of integer ;
R1 [4] : size 3 : values { 21 641 23 36 125; 54 221 43 ; 287 501 453 334 23 }
R1 [5] : size 2 : values { 12 10 100 ; 76 15 8 54 432 }
R1 [6] : size 3 : values { 17 61 928 785 875 ; 334 121 61 9 ; 32 465 123 }
R1 [7] : size 4 : values { 210 71 ; 90 47 32 10 93 ; 332 453 12 634 ; 44 53 55 134 }

The range of the first dimension R1 is specified in the first line as above. The user is expected to specify the sizes for the R2 dimension as shown for R1 [4] to R1 [7]. The varying sizes in R3 dimension are captured by the number of elements in each segment, separated by semicolon within curly bracket pairs. For example, in the direction of first component of of R1 [4], there are 5 numbers, hence the size is 5. Similarly, the size of the data column for second component of R [6] is 4. You will compute these only by traversing the parse tree and not by counting while creating the token stream from the dead source code. You can visualize the recursive nature of occurrences of these numbers. Design your grammar appropriately to incorporate this flexibility. If while traversing the tree later during implementation, if any such size in the third dimension is found 0, then it will be an error. For example,

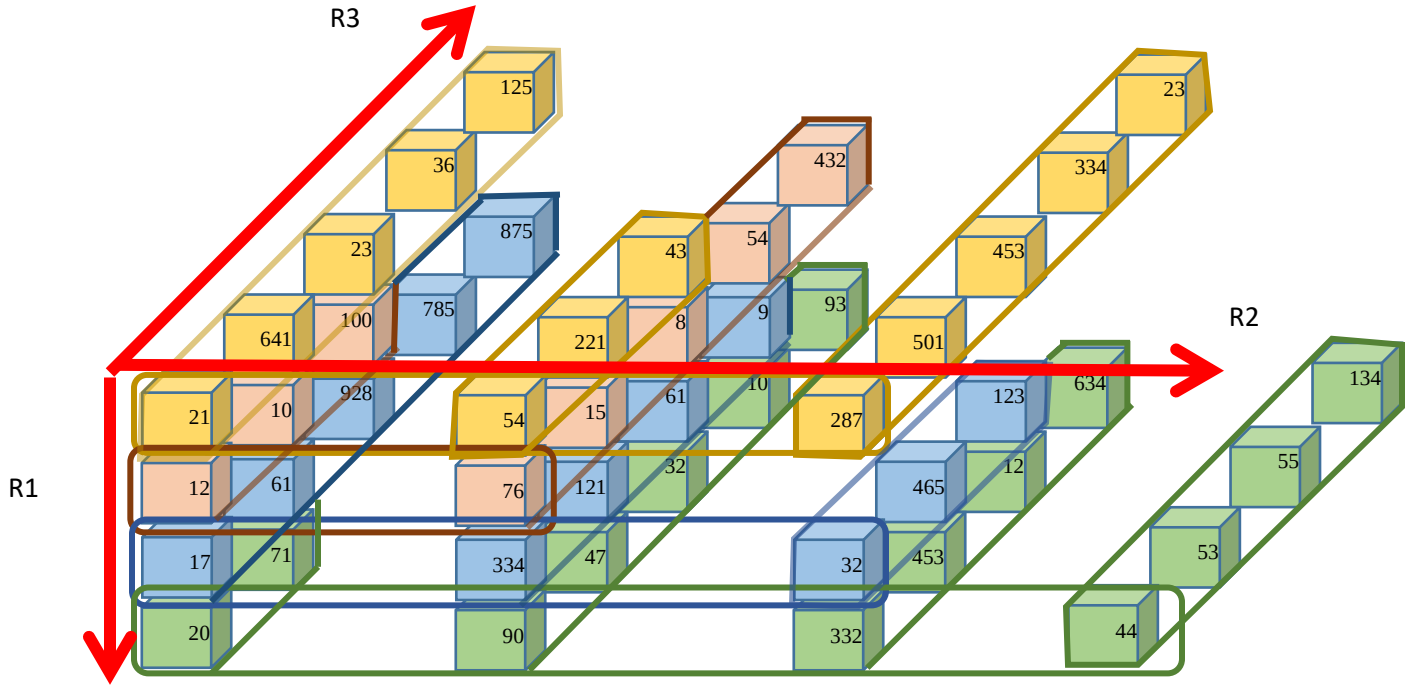
R1 [6] : size 3 : values { 17 61 928 785 875 ; ; 32 465 123 }

Here the second component is of size 0, but the total number of components is 3 matching with the specified size at the beginning, then this will be reported as an error.

Similarly, R1 [6] : size 5 : values { 17 61 928 785 875 ; 334 121 61 9 ; 32 465 123 }

With only three components computed on the fly by traversing the tree will be reported wrong as the number three does not match with size 5.

The 3-dimensional jagged array can be shown as follows



Array element construction and indices

The elements of any array are constructed in the most usual form such as A [i j k] with all names and symbols separated by the blank spaces. For example, an expression accessing (using) three dimensional array elements can be,

A [2 3 5] + B [6 9 8]

Or

abc [u v 12] - pqr [10 m 12]

The indices of array elements can be either the numbers or variable identifiers and are separated by blank spaces and enclosed within the square brackets.

Type expressions

The number of dimensions, ranges in all dimensions and the basic element type, give the type expression of the rectangular array. For example,

declare list of variables u v : array [2 .. 5] [3 .. 6] of integer ;

The type expression of the above array is given as

<type=rectangularArray, dimensions=2, range_R1= (2, 5), range_R2 = (3, 6), basicElementType = integer>

The jagged array given below has the type expression

<type =jaggedArray, dimensions=2, range_R1=(3, 8), range_R2 = (3, 6, 2, 4, 1, 5), basicElementType = integer>

declare list of variables s4 s5 s6 : jagged array [3 .. 8] [ ] of integer ;
R1 [3] : size 3 : values { 20 ; 35 ; 54 }
R1 [4] : size 6 : values { 65 ; 89 ; 99 ; 11 ; 37 ; 11 }
R1 [5] : size 2 : values { 22 ; 745 }
R1 [6] : size 4 : values { 67 ; 91 ; 13 ; 44 }
R1 [7] : size 1 : values { 17 }
R1 [8] : size 5 : values { 31 ; 97 ; 10 ; 9 ; 120 }

The two dimensional jagged array has in its R3 dimension only one element each, i.e. there is no spread of data in R3 dimension other than the very first place.

Similarly, the type expression for the three dimensional jagged array

declare list of variables s1 s2 : jagged array [4 .. 7] [ ] [ ] of integer ;
R1 [4] : size 3 : values { 21 641 23 36 125 ; 54 221 43 ; 287 501 453 334 23 }
R1 [5] : size 2 : values { 12 10 100 ; 76 15 8 54 432 }
R1 [6] : size 3 : values { 17 61 928 785 875 ; 334 121 61 9 ; 32 465 123 }
R1 [7] : size 4 : values { 210 71 ; 90 47 32 10 93 ; 332 453 12 634 ; 44 53 55 134 }

is given as

<type =jaggedArray, dimensions=3, range_R1=(4, 7), range_R2 = (3 [5, 3, 5] , 2 [3, 5], 3 [5, 4, 3] ,4 [2, 5, 4, 4]), basicElementType = integer>

The type expressions for the primitive data types are <type= integer>, <type=real> and <type=Boolean> respectively for the corresponding types.

Implementation of type expressions

As all the constructs are derived recursively using the underlying grammar, the parse tree represents the structure of the program. The type expression is computed by *traversing the parse tree* and accumulating the relevant information from the subtree corresponding to the construct defining the type. The lexemes of the source code

are the leaves of the parse tree, tokenized appropriately and encapsulated with line number in the node. The type information is fetched from one branch of the subtree corresponding to the declaration statement and is propagated to the leaf nodes residing in the other branch comprising the list of variables. The type expression once computed by traversing the subtree is stored in the non-leaf node representing the root of the subtree appropriately. Since a tree is a hierarchical structure with its nodes linked with address fields, the information is propagated sequentially from one node to another. In this process, the partially computed types are stored in the nodes falling in the branch of traversal appropriately. The nodes can be initially labelled with the information describing its construct such as declaration statement, assignment statement, type definition, list of variables, jagged array, rectangular array, dimension, size, list of values etc. as appropriate. This can be used for designing your non-terminal names. Each non-terminal represents a construct, and can be used in defining the labels of the nodes. You can enumerate the non-terminals so as to save you from costly string comparisons throughout. Once, while traversing a type expression computation is fully computed, it is also populated for the associated variable names in the array type table. Based on the descriptions of the array examples, you can identify that the structure of the type expressions can be different for different data types and will require a *union data structure* to store these type expressions in the nodes of the parse tree.

Operations

[1] ***Arithmetic operators*** +, -, * and / are overloaded and can be applied to primitive and constructed data types in the language in many specific ways as described below.

(a) ***Plus operator***: A plus operator can be applied to two operands of the same type. The type of the resultant expression is the same as that of the two operands. An integer added with another integer type operand produces an expression of ***integer*** type. The similar is with the two operands of ***real*** type and the sum produces an expression of real type. A ***boolean*** type variable cannot be added to another *boolean* operand, and if tried, an error is reported. An ***array variable*** can be added to another array variable of the same type, *element by element*, and the resultant array is of the same type with its elements as sum of the elements of the two corresponding array variables. This applies to ***both rectangular as well as jagged arrays*****.**

(b) ***Minus operator*****:** All rules for addition apply in the same way for minus operator. The resultant type is the same as that of the two operands. Minus operator cannot be applied to operands of boolean type.

(c) ***Multiplication operator*****:** All rules for addition apply in the same way for multiplication operator. The resultant type is the same as that of the two operands. Multiplication operator cannot be applied to operands of boolean type.

(d) ***Division operator*****:** This works similar to the applicability of plus operator except that the type of the resultant expression is always real. A division operator can be applied if either both operands are integers, or both operands are reals. It cannot be applied to variables of Boolean type.

[2] ***Assignment operator*** = is used in the assignment statement. The left hand side of assignment operator is a variable identifier and the right hand side of the operator is the arithmetic or Boolean expression. The assignment of an array variable to another variable of array type is not valid, but can be applied to any other variable of primitive data type or an array element appropriately. The left hand side variable can be a simple identifier or an array element. The assignment operator is type safe if the left hand side variable is of the same type as that of the right hand side expression.

[3] ***Logical operators*** &&& and ||| can only be applied if the operands are of Boolean type only. The resultant type of the expression is Boolean. If the operands are of any other type then a type error is reported.

Sample program

A sample program of this language is as follows. Few type errors are shown with the pair of two stars for illustration only, while such ***comment is not part of the language***.

```
program ()
{
        ** All declarations appear before the assignment statements **

        declare list of variables abcd ghd2_1 ssd_2_3 : array [ 2 .. 5 ] of integer ;
        declare list of variables new d w e2 : integer ;
        declare v1 : integer ;
        declare list of variables a1 a2 : integer ;
        declare list of variables b1 b2 b3 : boolean ;
        declare list of variables c1 c2 : real ;

        declare list of variables u v : array [2 .. 5 ] [3 .. 6 ] of  integer ;

        declare list of variables s1 s2, s3 :  jagged array [ 4 .. 7 ] [ ] [ ]  of integer ;
        R1 [ 4 ]  : size 3 : values { 21 641 23 36 125 ; 54 221 43 ; 287 501 453 334 23 }
        R1 [ 5 ]  : size 2 : values { 12 10 100 ; 76 15 8 54 432 }
        R1 [ 6 ]  : size 3 : values { 17 61 928 785 875 ; 334 121 61 9 ; 32 465 123 }
        R1 [ 7 ]  : size 4 : values { 210 71 ; 90 47 32 10 93 ; 332 453  12 634 ; 44 53 55 134 }

        declare list of variables m1 m2 m3 : real ;

        declare list of variables h1 h2 h3 : array [ 12 .. 34 ] [ 11 .. 20 ] [ 18 .. 33 ] of integer ;

        declare list of variables s4 s5 s6 : jagged array [ 3 ..  8 ] [ ] of integer ;
        R1 [ 3 ]  : size 3 : values { 20 ; 35 ; 54 }
        R1 [ 4 ]  : size 6 : values { 65 ; 89 ; 99 ; 11 ; 37 ; 11 }
        R1 [ 5 ]  : size 2 : values { 22 ; 745 }
        R1 [ 6 ]  : size 4 : values { 67 ; 91 ; 13 ; 44 }
        R1 [ 7 ]  : size 1 : values { 17 }
        R1 [ 8 ]  : size 5 : values { 31 ; 97 ; 10 ; 9 ; 120 }


        declare list of variables p1 p2 p3 : jagged array [ 105 ..  107 ] [ ] of integer ;
        R1 [ 105 ]  : size 4 : values { 20  21  33 ; 102 ; 35 ; 54 }      ** type definition error for 2D jagged array**
```

R1 [106] : size 3 : values { 165 ; 809 ; 929 }
R1 [107] : size 2 : values { 22 ; 745 }

declare list of variables q1 q2, q3 : jagged array [4 .. 7] [ ] [ ] of integer ;
R1 [4] : size 3 : values { 21 641 23 36 125 ; 54 221 43 ; 287 501 453 334 23 }
R1 [5] : size 2 : values { 12 10 100 ; 76 15 8 54 432 ; **29 09 76 11; 67 27 80** } **type definition error for 3D jagged array for defined size 2**
R1 [6] : size 3 : values { 17 61 928 785 875 ; 334 121 61 9 ; 32 465 123 }
R1 [7] : size 4 : values { 210 71 ; 90 47 32 10 93 ; 332 453 12 634 ; 44 53 55 134 }

**** Assignment statements start here ****

abcd = ghd2 + ssd_2_3 ; ** no error **

new = d * e2 + w – 76875; ** no error **

v1 = a1 – a2 * d ; ** no error **

b1 = b2 &&& b3 ||| b1 ; ** no error **

c1 = c2 + c1 ; ** no error **

c2 = a1 / a2 ; ** No error as a1 and a2 are of integer and produce a resultant value of real type**

c1 = a1 * a2 ; **type error **

h1 = s2 + s3 * u – h2 * h3 ; ** type error**

u [2 5] = v [3 4] + v1 ; ** no type error **

u [3 7] = v [3 5] – new ; ** type error as 7 is greater than 6, the high range of dimension 2**

u = v + u ; ** no error **

h1 = s4 – s5 * s6 ; ** type error as h1 and RHS expression are of 3D rectangular array type and 2D jagged array type respectively**

s1 = s2 + s3 * s1 ; ** no error **

h1 = h2 + h3 * h1 – h2 * h3 ; ** no error **

s1 = s2 + s4 * s6 ; ** type error as s1 and s2 are of jagged arrays of different sizes **

q1 = q2 + q3 ; ** type error as their type definition carries an error **

}

The teams are expected to create many test cases with all possible variations and expression sizes to test their code. Some test cases will also be provided towards the end, about three days prior to submission of this assignment code, to test final working of team's code on my test cases.

Problem description

This assignment involves the following aspects regarding the language design and testing.

Grammar design

The teams are expected to design the grammar correctly incorporating all features and constructs defined in the language specification. The grammar should be unambiguous in such a way that it produces only one parse tree for the given input source code. Also, the care should be taken in making the rules in such a way that the selection of rules for creating the parse tree for the given input is not ambiguous. Once the rules are designed, the students should verify the construction of parse tree manually on paper. You may require a chart paper to accommodate the complete parse tree for input source code. Once the rules are verified, then type all the rules in a file named as "grammar.txt". Write grammar in this file with first word as the name of the nonterminal in the left hand side of the production rule while all other terminals and non-terminals appearing in the right hand side of the grammar rule will start from the second place and continue until end of line.

Data structures design

- ***grammar:*** This is an array of linked lists, where each cell node of the array contains the name of the nonterminal at the left hand side of the grammar rule and the link to the first node of the linked list containing the right hand side symbols. This data structure is populated using the grammar file grammar.txt. This array of linked list capturing the grammar designed by your team is later used for parse tree creation for the given input source code.
- ***tokenStream***: This is the linked list containing the token names, lexemes and **line numbers**. The node of this linked list is created while reading the lexemes separated by blank spaces in the input source code. As the lines in the source text are separated by '\n', the line number count is maintained by incrementing that at each line change. The token names are associated with each lexeme appropriately and information is populated in the node. This list is used for creating the parse tree.
- ***parseTree:*** This is an n-ary tree with nodes containing nonterminal symbol, link to children, the type of the subexpression or variable corresponding to the subtree of the node. The parse tree is created using the grammar and input token stream. Once the parse is created, it is traversed to collect type expression information and populating its non-leaf nodes with accumulated information appropriately. Parse tree nodes are implemented using variant records, which has its variant part implemented using union data type for storing the type expressions.

- ***typeExpressionTable***: The table is a two dimensional array implemented for storing the types for each array. This is implemented using variant record with three of the following fields being fixed and the fourth for type expression being the variant one. The four fields are as follows.
    - *Field 1*: The name of variable extracted from the declaration statement and to be used in the assignment statement.
    - *Field 2:* The information about whether it is an array or not, if yes, then whether it is rectangular array or a jagged array. Use numbers enumerated values of 0, 1 and 2 for primitive data type, rectangular array and jagged array respectively. The value 0 corresponds to integer, real and Boolean types. However, these primitive type details are filled in the fourth field explicitly defining the integer, real or Boolean specifications appropriately.
    - *Field 3:* If the type is a rectangular array, then whether it is static or dynamic based on the type of range parameters used in defining the array type. If it is not a rectangular array this value is stored as "not_applicable".
    - *Field 4*: type expression, which is implemented using the union data type.

  This table is populated on the fly as the parse tree is traversed and the types are computed fully.

Functions prototypes

- ***readGrammar("grammar.txt", grammar G):*** This function reads the grammar rules line by line from the file grammar.txt and populates the array of linked list G. Each cell of the array holds the LHS nonterminal of the production rule, and the linked list contains the remaining symbols of terminals and nonterminals.
- ***tokeniseSourcecode("sourcecode.txt", tokenStream *s):*** The source code text is read from the file "sourcecode.txt" and processed for extracting the names and symbols, all separated by blank spaces. It captures the line numbers, lexemes and the appropriate token names to populate the nodes of the linked list s.
- ***createParseTree(parseTree *t, tokenStream *s, grammar G):*** This function uses the start symbol for deriving the complete program, and looks at the tokens in the tokenStream sequentially. Based on the next token, it selects the rule to expand the nonterminals maintained in the stack. The function of stack is same that of push down automata. The stack symbols are the symbols of right hand sides of the grammar rules, which can be both terminals and non-terminals. The derivation to be implemented is left to right derivation, which expands the nonterminal at the left first and maintains remaining nonterminals in the stack. The flow of terminals and nonterminals is last in first out, therefore, the right hand side symbols of the selected grammar rule are pushed onto the stack in such a way that the first symbol on the RHS of the rule goes to the top of the stack to ensure left most derivation. The right hand side of a rule has number of terminals and nonterminals, therefore the stack node should be capable of storing the information whether the symbol stored is a nonterminal or a terminal. This will help in matching the top of the stack with the next token. If the top of the stack is a non-terminal, then it is expanded further without consuming the input token. If the top of the stack is a terminal, then, it is checked whether it matches the next token (called lookahead). If the top of the stack terminal symbol is epsilon, then the rule is selected appropriately. The epsilon is popped out in such a way that the next symbol on the stack, if a terminal matches with the input token, or if a non-terminal is expanded. Each nonterminal A, if expanded using rule A→ α, where α is the stream of terminals and nonterminals on the RHS of the rule, then becomes the parent node for all its children in α.

For all test cases, it is assumed that the code is syntactically correct; this means there is no grammatical mistake in the code. Therefore, there is no possibility of encountering any error while creating the parse tree. Of course, when the next token does not match with the top of the stack terminal symbol, then you should modify your technique for rule selection and select the rule appropriately. At the end, print a message that "parse tree is created successfully".

- ***traverseParseTree(parseTree *t, typeExpressionTable T):*** The parse tree is traversed from its root 't', and its children are traversed from left to right. The sub-trees corresponding to the non-leaf nodes labelled as ***declaration statements*** are traversed for type expression computation via one branch of it. In this process, the information will propagate from children nodes to parent nodes. The other branch comprising the list of variables is traversed to pick up the variable names (tokenized as ID) and the computed type is associated appropriately. In this process, the type information residing at the root of the subtree, is propagated down to the leaf nodes. Each variable at the leaf node in this subtree in the corresponding declaration thus gets a type. Before entering into the subtree corresponding to next statement, the function also populates the *typeExpressionTable* for each variable name found in the list of variables along with its associated type. The typeExpressionTable is fully populated before it enters the other branch comprising all assignment statements. It is assumed that the variables used in any expressions should have been declared in the program. This is to minimize your work on checking whether a variable is declared or not, which will fall in the scope of compiler construction course later. Hence all test cases created by you, must have all the variables used in the expressions declared previously.

  If the sub-tree corresponding to the ***assignment statement*** is traversed, then the types of the variable identifier at the left hand side of the assignment statement, and the types of variables used in the expression are obtained from the *typeExpressionTable*. For each occurrence of any variable, the typeExpressionTable is searched to obtain type of the variable; therefore, the typeExpressionTable should be implemented efficiently. Consider an expression a+b-c*d, then if the types of the operands of the sub-expression (say c*d) follow the permissible rules for types defined above for the used operator, then the type of the non terminal in the parse tree corresponding to the expression c*d is populated in the root of the corresponding sub-tree. Next the type of the variable identifier b is obtained from the *typeExpressionTable*, and the type expression stored in the root for c*d is used for type checking the feasibility of – (minus operation) in the sub-expression b-c*d. If this follows the type rules, then the type of this sub-expression is stored in the root node (the non-leaf node) corresponding to the sub-tree for this sub-expression. This continues until the traversal reaches back to the non-leaf node corresponding to the RHS expression of the assignment statement. If the type of the expression and the type of left hand side variable identifier follow the type rules for assignment operator, then the next assignment statement is picked for traversal, else a type error is indicated with line number (of source code).

  An array element used in the expression such as u [2 5] will be required to be verified for its type from the table entry for u. If the index while accessing an element is not a variable identifier (as in u [k m]), the type of u is not a dynamic array, if the range values preserved in the type expression for *u* allow usage of indices 2 and 5 in the respective dimensions, and are with in permissible ranges (bound), then the element access is type error free, else an error will be reported.

All type errors are printed in the following details in the same order in each line on the console

- Line number (This refers to the line number in the leaf node taken from the token stream at the time of creating parse tree. **The errors printed without this line number will not get any credit**.)
- Statement type (declaration or assignment)
- operator
- lexeme of first operand
- type of first operand
- lexeme of second operand
- type of second operand
- position in the parse tree in terms of its depth from parse tree's root node (take root node depth as 0)
- short message (in maximum 30 characters length)

The statement type is either declaration statement or an assignment statement. If the error was in size mismatch of three dimensional jagged arrays, then print a message "3D JA size mismatch" while marking *** for operator and other fields which are not applicable in declaration statement. If the error was in any assignment statement then report that appropriately. Make sure that your error messages are column justified (you can use %12d, %20s etc. in your printf statement to justify the details). All details above, corresponding to an error must be printed in single line on the console. The type expressions, even if enumerated at the internal level, should be printed in user readable form as has been mentioned above.

- ***printParseTree(parseTree *t)***: prints the parse tree nodes in the preorder traversal in the following format
  - Symbol name
  - Whether terminal or non terminal
  - Type expression stored in the corresponding node (if non-leaf)
  - Name of lexeme (if leaf node)
  - Line number (if leaf node)
  - Grammar rule applied for expansion of this node while parsing (if non leaf)
  - Depth of node (root of the parse tree at depth 0)

All of these for one will have to be justified in pretty columns and will be printed only in a line. Similarly keep printing the nodes information line after line.

- ***printTypeExpressionTable (typeExpressionTable T)***: This function prints all four field details stored in the type expression table as described above. The printing format is as follows

  Field 1 Field 2 Field 3 Field 4

  Print the details line by line for each variable. All field details should be justified to give clear view of details. The type expression should also be printed in human readable form irrespective of how you enumerated the types.

- ***Driver function:*** You should create the driver (main) function that uses options in loops and continues being in loop until receives a 0 as an option.

  Option 0: exit

Option 1: Create parse tree

Option 2: Traverse the parse tree to construct typeExpressionTable. Also print the type errors while traversing the parse tree and accessing the typeExpressionTable.

Option 3: Print parse tree in the specified format

Option 4: Print typeExpressionTable in the specified format.

Also, ensure that all options are independent of each other. For example, option 2 must first create the parse tree and then traverse appropriately. Similarly the Option 1 created earlier should be independent from other options.

Implementation platform and compiler: All implementation will be required to be done in C programming language. You should use the **Ubuntu (version 20.04.1 LTS) and GCC (version 9.3.0)** strictly for all your implementations. If you have only windows operating system, then install Ubuntu on the virtual box (free from oracle) and install the given gcc version for compiling your code. You can also get your hard disk partitioned for the Ubuntu operating system. If you use any other platform then your code will not be evaluated.

Evaluation: As has been mentioned earlier, that the evaluation will be test case based. There will be no marks for your code if it does not compile and execute. Plagiarized code will not be evaluated and penalty will be imposed as announced earlier.

Errata: Any updates regarding the assignment language or problem description will be posted on Nalanda. Students are advised to visit Nalanda regularly.

The due date for submission of this assignment is **October 27, 2020, 7:00 p.m.** Students are advised to read all details carefully and inform me if there exists any discrepancy in the description above. Please feel free to write to me in case of any doubts.

Vandana
October 2, 2020