ChartLan

A Language for Chart

COMS W4115 PLT project

Language Reference Manual

Professor: Stephen A. Edwards

Yibo Zhu	**(yz2486@columbia.edu)**
Xiuming Dou	**(xd2138@columbia.edu)**
Xiao Xu	**(xx2165@columbia.edu)**
Ziyue Chen	**(zc2239@columbia.edu)**
Xiang Ma	**(xm2151@columbia.edu)**

1. Introduction

ChartLan is a language specified in table creating, information storing, retrieving and modifying. When programming in ChartLan, users can easily create a Table and store various data type from integers and strings in it. ChartLan also provides easy ways to retrieve or modify stored entries in a table. Operations like addition and concatenations between data storages like arrays and tables are designed to be simple and easy in ChartLan.

Our programming language, Chartlan, is designed for users to creating table and processing data by typing a few lines of code. With the help of new data type and new defined operators, users can simplify the process of creating table, inserting and deleting data. It also includes fundamental and useful functions that meet almost every assignment of daily work.

2. Lexical conventions

(1)Comments

The characters #~ introduce a comment, which terminates with the characters ~#.

(2)Identifiers

An identifier is a sequence of letters and digits; the first character must be alphabetic. Upper and lower case letters are considered different.

(3)Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

Int	integer number
String	serial of characters
Array	list of integers or strings
Table	list of array integers or strings
If…else…	conditional control
Def	function definition
While	loop identifier
Createarray	array constructor
Createtable	table constructor
Return	return a value
Print	print values

(4)Constants

Integer constants	…-2 -1 0 1 2…
Boolean constants	integer 1 for Boolean true, 0 for Boolean false
String constants	“abcedf “ or “@$%^&”
Array constants	Createarray(some constants of integer or string

	separated by “,”) e.g. Createarray(1,2,3,”lol”,”afaf”)
Table constants	Createtable(Createarray(row1), Createarray(row2) Createarray(row3)…) where row1, row2, row3… are array constants separated by parentheses, e.g. Createtable(Createarray(1,2,3), Createarray(“a”,”b”,”c”), Createarray(1,”a”,”dd”))

3. Declarations

General rule: Type-specifier identifier = expression;
Where type-specifier can be integer, string, array, table
Eg: int x = 3; string c = “this is a string”

As for the declaration of array variable:
Array identifier = Createarray (integers or strings separated by “,”)

As for the declaration of table variable:
Table identifier = Createtable(array1 , array2 …)

4. Expression

(1) Identifier : Its type is specified by its declaration.
(2) Constants : A integer, string, array, or table is an expression.
(3) (expression): A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression.
(4) expression binary operator expression
binary operators can be: +, - , * , / , && , ||, .+ , .- , .* , ./ …
(5) Identifier [expression]: The intuitive meaning is that of a subscript, the subscript expression is int.

5. Operators

Addition operators:
The addition operator + and group left to right

Expression + expression

The binary + operator indicates addition.
Integer + integer gives integer addition value.
String + string gives the concatenation of the two strings
When two arrays perform an addition, each of the array elements will be added up in pairs. Those elements that do not have a pair to match will be appended in the back of the result array.
E.g. Createarray(1, 2, “ha”, 3) + Createarray(1, 1,”ha”) gives back (2, 3,”haha”, 3)

Finally, addition between Tables will pair wise add up the corresponding rows of the two tables and append those rows that do not have a matching pair in the end of the table.

Eg: Table1 is Createtable(Createarray(1, 2, 3), Createarray(4, 5, 6)), Table2 is Createtable(Createarray(7 ,8 ,9)) then, Table1 + Table2 will be a new Table with 2 rows: ((8,10 ,12), (4 ,5, 6)).

Subtraction operator:

The addition operator - and group left to right, and it is like the inverse operation of addition.

Expression - expression

Perform normal subtractions on two expressions.

The binary - operator indicates subtraction.

Integer - integer gives integer subtraction value.

String1 – string2 gives remained string that deletes the first matching string2 from string1's substring.

When two arrays perform a subtraction, each of the array elements will subtract in pairs, those elements that do not have a pair to match will be appended in the back of the result array.

Eg: Createarray(1, 2, "ha", 3) - Createarray(1, 1,"ha") gives back (0, 1,"ha", 3)

Finally, subtraction between Tables will pair wise subtract the corresponding rows of the two tables and append those rows that do not have a matching pair in the end of the table.

Eg: Table1 is Createtable(Createarray(1, 2, 3), Createarray(4, 5, 6)), Table2 is Createtable(Createarray(7 ,8 ,9)) then, Table1 - Table2 will be a new Table with 2 rows: ((-6,-6 ,-6), (4 ,5, 6)).

Multiplication operator:

The multiplication operators * group left to right

*Expression * expression*

The binary * operator indicates multiplication.

Integer * integer gives integer multiplication value.

Note when an integer or a string multiplies with an Array will insert the item into the array. 1 * Createarray(2, 3) will insert the integer 1 at the head of the array. Thus give back (1, 2, 3); Createarray(2, 3)*1 will put integer 1 at the end of the array giving back (2, 3, 1).

When two arrays perform a multiplication, the second array will be appended to the first. eg: Createarray(1 ,2, 3) *Createarray (4, 5 ,6) gives back (1 ,2 ,3 ,4, 5 ,6)

Finally, Multiplications between Tables will append the second table's rows onto the end of the first's provided: Table1 is Createtable(Createarray(1, 2, 3), Createarray(4, 5, 6)), Table2 is Createtable(Createarray(7 ,8 ,9), Createarray(10,11,12)) then, Table1 * Table2 will be a new Table with 3 rows: ((1 ,2 ,3), (4 ,5, 6), (7, 8, 9), (10,11,12)).

Division operator:

The division operators / group left to right, the division is like the inverse operation of multiplication.

Expression/expression

The binary / operator indicates division.

The integer data type can perform division as commonly use, and divisor cannot be zero.

Integer/integer gives back integer value. If the actual result is not an integer, it will return the integer part of the result.

When an array divides an integer or a string, the array will delete the first matching item from the array, e.g: Createarray(2,3,1)/1=(2,3), Createarray(3,4,4,5)/4=(3,4,5). Createarray(4,5,6)/2 will return the original array (4,5,6).

The division between table and array will delete the array from the table and return the remaining table. E.g.CreateTable (Createarray (1,2,3), Createarray(3,4,5), Createarray(4,5,6))/Createarray(4,5,6)=((1,2,3),(3,4,5)). If the array is not contained in the table, the original table will remain the same.

The division between two arrays and two tables is not allowed, e.g. Createarray(1,2,3,4)/(1,4)will throw an error, which can be replaced by Createarray(1,2,3,4)/1/4.

Arithmetic operator:

Expression .+ expression
Expression .- expression
Expression .*expression
Expression ./ expression

The arithmetic operators are used when users want to do the operation to the every element in array or table.

E.g.: Creatarray(2,3,1).+3= (5,6,4);

Creatarray (2,3,4).-1=(1,2,3);

E.g.: Createtable (Creatarray (1,2,3), Creatarray (4,5,6)).*3=((3,6,9),(12,15,18));

Equality operator:

Expression == *expression*

returns 1 (Boolean constants true)if the two expressions are identical and false otherwise.

Expression != *expression*

returns 0 (Boolean constants false) if the two expressions are identical and true

otherwise.

Expression1 >(>=) expression2

returns 1 (Boolean constants true) if expression1 is greater (not less than) expression2 and false otherwise.

Expression <(<=) expression

returns 1 (Boolean constants true) if expression 1 is less (not greater than) expression2 and false otherwise.

Assignment operator:

Identifier = expression.

Assigns an expression's value to the identifier.

Logical operator:

Expression && expression

returns 1 (Boolean constants true) if the two expressions are both true and false otherwise.

Expression || expression

returns 0 (Boolean constants true) if the one of the two expressions is true and false otherwise.

6. Statements

Expression statement:

Most statements are expression statements, which have the form *expression*;

Compound statement:

So that several statements can be used where one is expected, the compound statement is provided:
compound statement
{statement-list}

statement-list:
statement
statement statement-list

Conditional statement:

The two forms of the conditional statement are
if (expression) {statement}
if (expression) {statement1} else {statement2}

In both cases the expression is evaluated and if it is nonzero, the statement1 is executed. In the second case, the statement2 is executed if the expression is 0.

While statement:

The while statement has the form

while (*expression*) *statement*

The statement is executed repeatedly so long as the value of the expression remains nonzero.

The test takes place before each execution of the statement.

Return statement:

A function returns to its caller by means of the return statement, which has one of the forms

return ;

return (expression);

Null statement:

The null statement has the form

;

A null statement is useful to carry a label just before the "}" of a compound statement or to supply a null body to a looping statement such a while.

Function Statement:

The function-statement is a compound statement which has declarations at the start.

functionstatement:

{ variable declarationlist statementlist }

Def typespecifier functionname (typespecifer identifier1, typespecifer identifier2 …) { function body}

Where typespecifier indicates the return type of the function which could be either int, string, array or table.

Arguments consists of arguments of the function which are identifiers with their type specified in the front.

Def is the keyword that specifies a function definition.

Functionname is a combination of any letters or integers but the first must be a letter.

Eg: Def int max(int a, int b){ #~gives back the bigger one of the two integers~#

```
If ( a > b) {
    Return a;
}
Else{
    Return b;
}
```

Some of built-in functions:
Int fread (String filename)
Int fwrite (String filename, Table t)
Int Length(Array a)
Array SizeOfTable(Table t)

7. Scope Rules

An identifier's lexical scope, which is essentially the region of a program during which the identifier can be used directly without drawing "undefined identifier" diagnostics. There are two kinds of identifiers in ChartLan's lexical scope.
Global variable is a variable that is accessible in every scope. That is to say, in our language, a global variable can be used anywhere in the program once it is defined.
Local variable is a variable that is given local scope. Such a variable is accessible only from the function or block in which it is declared.

8. Sample programs

(1) This is a very simple program of building and keeping track of person's phone numbers.

We want a table which first column is the name of the contacts and the second column contains the corresponding phone number of the contact.

```
Table contactinfo = CreateTable( Createarray("Peter","212-505-3333"), Createarray
("Jane","374-864-5123"), Createarray ("John","974-505-2222"))

#~ we can store our table in a file ~#
Fwrite("mycontactinfo.txt", contactinfo);

#~ and when we need it, we can read the table from file ~#
contactinfo = fread("mycontactinfo.txt");

#~now say we want to add another contact ~#
Array SmithEntry = Createarray ("Smith","777-223-2222");
Contactinfo = contactinfo * SmithEntry;
#~append the smith's contact into our table~#

#~we can define functions to retrieve data from the contactinfo table ~#
Def    String    SearchPhoneNumber( Table t, String name){
       Int i = SizeOfTable(t)[0]-1;
#~built in function SizeOfTable returns array whose first element is the number of
rows table t contains and the second the number of columns. Calling SizeofTable(t)
with indexing[0] to get number of rows t contains~#.
```

```
    Array x= Createarray();

    While ( i > = 0){
        x=t[i-1];
        If (x[0] == name){ #~match the first element in the row ‘‘name’’~#

            Return x[1];
        }
        i = i – 1;
    }
}
```
#~ now we can use the function defined to look up contact information~#

```
String Johnsphonenumber= SearchPhoneNumber(contactinfo, “John”);
```
#~ which will give back John’s phone number ’’974-505-2222’’ ~#

(2) Sum of specific column of the table.

```
Array a = Createarray (“Peter”, 2000);
Array b = Createarray (“Sally” , 3000);
Array c = Createarray (“James”,1600);
Table info = CreateTable(a, b, c);
Array x= Createarray();
```

#~we can define functions to retrieve specific data from the sum table ~#

```
Def int SumOfSalary(Table t, int col)
{
        int i= SizeOfTable(t)[0]-1;
        while (i>=0)
        {
            x=t[i];
            s=s+x[col];
            i=i-1;
        }
       return s;
}
```

#~ now we can use the function defined to calculate the sum of each person’s salary~#

```
Int sum = SumofSalary(info, 1);
```

Appendix 1

Scanner.mll:

```
{ open Parser }

let letter =['a'-'z' 'A'-'Z']
let digit = ['0'-'9']

rule token = parse
  [' ' '\t' '\r' '\n']        {token lexbuf}
| '='     { ASSIGN}
| '+'       { PLUS }
| '-'     { MINUS}
| '*'     { TIMES}
| '/'     { DIVIDE}
| ".+"    { DADD}
| ".-"    { DSUB}
| ".*"    { DMULT}
| "./"    { DDIV}
| "&&"    { AND }
| "||"    { OR}
| "=="    { EQ }
| "!="    { NEQ }
| ">"     { GT }
| "<"     { LT }
| ">="    { GEQ }
| "<="    { LEQ }
| '('     { LPAREN }
| ')'     { RPAREN }
| '['     { LBRACKET}
| ']'     { RBRACKET}
| '{'     { LBRACE}
| '}'     { RBRACE}
| ';'     { SEMI}
| ','     { COMMA}
| '"'     { doublequote lexbuf}
| "if"    { IF}
| "while" { WHILE}
| "else"  { ELSE}
| "int"   { INT}
| "string"      {STR}
| "array"       {ARRAY}
| "createarray" {CREATEARRAY}
| "createtable" {CREATETABLE}
| "table"       {TABLE}
```

```
| "def"            {DEF}
| "return"      {RETURN}
| "print"       {PRINT}
| letter(letter|digit|'_')* as id {ID(id)}
| digit+ as lit  {LITERAL(int_of_string lit)}
| "#~"    {comment lexbuf}
| eof     {EOF}
| _ as invaildchar  {raise (Failure("illegal character "^ Char.escaped
invaildchar))}
and comment =
    parse "~#" {token   lexbuf}
          |_   {comment lexbuf}
and doublequote =
    parse '"'  {token lexbuf}
          |
(letter|digit|'!'|'@'|'$'|'%'|'^'|'&'|'*'|'('|')'|'-'|'_'|'+'|'='|'`'
            |'~'|','|'<'|'.'|'>'|'.'|'?'|'/'|':'|';'|'''|'{'|'}'|'['|']')*
as str {STRING(str)}
```

Appendix 2

Parser.mly

```
%{ open Ast %}
%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET AND OR COMMA PLUS MINUS
TIMES DIVIDE DADD DSUB DMULT DDIV
%token ASSIGN EQ NEQ LT LEQ GT GEQ RETURN IF ELSE FOR WHILE INT STR ARRAY TABLE
DEF EOF PRINT
%token CREATEARRAY CREATETABLE
%token <int> LITERAL
%token <string> ID STRING
%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left EQ NEQ
%left AND OR
%left LT GT LEQ GEQ
%left PLUS MINUS DADD DSUB
%left TIMES DIVIDE DMULT DDIV
%start program
%type <Ast.program> program
%%
program:
/* nothing */ { [], [] }
| program vdecl { ($2 :: fst $1), snd $1 }
| program fdecl { fst $1, ($2 :: snd $1) }
fdecl:
DEF types ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
{ { fname = $3;
formals = $5;
locals = List.rev $8;
body = List.rev $9
returntype = $2 } }
formals_opt:
/* nothing */ { [] }
| formal_list { List.rev $1 }
formal_list:
ID { [$1] }
| formal_list COMMA ID { $3 :: $1 }

types:
INT {}
| STR{}
| ARRAY {}
| TABLE {}
```

```
vdecl_list:
/* nothing */ { [] }
| vdecl_list vdecl { $2 :: $1 }
vdecl:
 INT ID SEMI { $2 }
|STR ID SEMI { $2 }
|ARRAY ID SEMI { $2 }
|TABLE ID SEMI { $2 }
stmt_list:
/* nothing */ { [] }
| stmt_list stmt { $2 :: $1 }

stmt:
expr SEMI { Expr($1) }
| RETURN expr SEMI { Return($2) }
| LBRACE stmt_list RBRACE { Block(List.rev $2) }
| IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
| IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) }
| WHILE LPAREN expr RPAREN stmt { While($3, $5) }
expr:
LITERAL { Literal($1) }
|STRING {String ($1)}
| ID { Id($1) }
| expr PLUS expr { Binop($1, Add, $3) }
| expr MINUS expr { Binop($1, Sub, $3) }
| expr TIMES expr { Binop($1, Mult, $3) }
| expr DIVIDE expr { Binop($1, Div, $3) }
| expr EQ expr { Binop($1, Equal, $3) }
| expr NEQ expr { Binop($1, Neq, $3) }

| expr LT expr { Binop($1, Less, $3) }
| expr LEQ expr { Binop($1, Leq, $3) }
| expr GT expr { Binop($1, Greater, $3) }
| expr GEQ expr { Binop($1, Geq, $3) }
| ID ASSIGN expr { Assign($1, $3) }
| ID LPAREN actuals_opt RPAREN { Call($1, $3) }
| LPAREN expr RPAREN { $2 }
| listvalue {}
| tablevalue {}
| ID LBRACKET indexelement RBRACKET {} /*match index*/

/*our version*/
```

```
| expr DADD expr  {Binop($1, Dadd, $3)}
| expr DSUB expr  {Binop($1, Dsub, $3)}
| expr DMULT expr {Binop($1, Dmult, $3)}
| expr DDIV  expr {Binop($1, Ddiv, $3)}
| expr AND expr {Binop($1, And, $3)}
| expr OR  expr {Binop($1, Or, $3)}

listvalue:
CREATEARRAY LPAREN listelement RPAREN {Array($3)}   /*match list */

tablevalue:
CREATETABLE LPAREN tableelement RPAREN {Table($3)} /*match table*/


listelement:
/*nothing*/ {}
|LITERAL {}
| STRING {}
| listelement COMMA STRING {}
| listelement COMMA LITERAL {}
| listelement COMMA ID {}



tableelement:
/*nothing*/ {}
| CREATEARRAY LPAREN listelement RPAREN {}
| tableelement COMMA CREATEARRAY LPAREN listelement RPAREN {}

indexelement:
LITERAL {}
| ID {}
| indexelement PLUS indexelement {}
| indexelement MINUS indexelement {}
| indexelement TIMES indexelement {}
| indexelement DIVIDE indexelement{}


actuals_opt:
/* nothing */ { [] }
| actuals_list { List.rev $1 }
actuals_list:
expr { [$1] }
| actuals_list COMMA expr { $3 :: $1 }
```