Middle East Technical University **Department of Computer Engineering**

CENG444 - Language Processors

2023-2024 Spring

Project III

Semantic Analysis & Introductory IR

Problem

In this assignment, you are required to write a C++ program that performs syntactic and semantic conformance tests for an evaluator based on dependency graphs (DGEval). To do so, you will need to build expression trees and apply type arithmetic. You are also required to perform an analysis to determine the order of execution based on topological sort.

DGEval

A DGEval module is a list of interrelated DGEval statements. At the core of a statement lies a DGEval expression that conforms to a set of syntactic rules which are mostly similar to the rules of the conventional expressions. The type system of DGEval is based on primitive types, which are string, number, Boolean, and array. DGEval does not have flow control structures like loops, decisions, compound statements. The order of evaluations is determined by the dependency graph implied by the statements.

Types

DGEval is a statically typed, strongly typed language. There is no explicit declaration construct in DGEval to define a variable. Instead, type inference is applied when an assignment is encountered. Once a variable is declared, its type cannot be changed. DGEval supports arrays which are encoded by using expressions in brackets. An array is composed of elements which must belong to a common type.

Below is the list of the types allowed in DGEval.

Type	**Description**
`number`	Type to represent IEEE 754 double precision floating point value.
`boolean`	Type to represent boolean values.
`string`	String of characters.
`array`	An object that encapsulates values. The values are accessed by the help of an index, which must be a number. Index numbers are converted to integers by the compiler runtime on the fly and range checking is performed. DGEval uses zero based indexing.

Type conversion rule set for this language is very limited:

- A number is automatically converted to the corresponding string to implement concatenation operator only.

Expressions

The form of the expressions in DGEval is mostly conventional. This section defines the entities that can be used to construct a valid DGEval expression.

- **Operators**

  DGEval has a rich set of operators with various number of operands and associativities. You are accustomed to many of them from your earlier experience. See Appendix 1: Operators.

- **Parentheses**

  An expression can be enclosed in parentheses to change / ensure evaluation order. See the precedence and associativity of the operators to understand the default evaluation order in Appendix 1: Operators.

- **String Literals**

  A string is in the form of regular and exceptional characters enclosed in double quotes. Exceptional characters are double quote("), and back-slash(\), and new-line(character code 10).

Exceptional Character	**Encoding in string**
double quote(")	\"
back-slash(\\)	\\\\
new-line	\n

  A character, regardless of it is exceptional or regular, can be coded with help of byte coding. Any byte except zero can be encoded in a string of characters by using the hexadecimal form, which is \xhh, where h is a hexadecimal digit. Capital and small letters can be used in hexadecimal digit specification.

- **Numeric Literals**

  The numeric literals can be composed of the valid combinations of whole part (W), decimal separator (.), and fractional part (F). Whole part can be either zero, or a sequence of digits starting with a non-zero digit. Fractional part is a nonempty sequence of digits, not ending with zero.

  Following are valid examples of W,".", and F combinations:

W	**.**	**F**	**Example**
1	1	1	1.25
1	1	1	0.25432
0	1	1	.25
0	1	1	.0134
1	0	0	123
1	0	0	0

A number can have an optional exponential part prefixed by 'e' or 'E'. When supplied, an optional sign that can be either '+' or '-' may follow. Then, a sequence of digits must be given. The first digit must be non-zero.

- **Boolean Literals**

  `true` and **`false`** are the boolean literals.

- **Identifiers**

  An identifier starts with a letter or an underscore. The remaining characters in an identifier can be letters, underscores, and decimal digits. DGEval identifiers are case sensitive so `var` and `Var` are two distinct variables.

- **Array Literals**

  An expression is coded between "`[`" and "`]`" to construct an array. Each expression part becomes a member of the newly constructed array. Empty arrays are possible.

- **Function Calls**

  A function call is formed as `<identifier>` followed an expression enclosed by parentheses. The list of expressions is skipped for the functions having no formal parameters. See Appendix 2: DGEval Runtime Library for complete set of runtime library functions.

Statements

There are two types of statements in DGEval: Expression Statement, Wait Statement.

- **Expression Statements**

  An expression statement consists of an expression followed by a semicolon.

- **Wait Statements**

  A **`wait`** statement starts with the `wait` keyword and followed by the list of identifiers followed by the **`then`** keyword. The statement is finalized by an expression followed by a semicolon. The list of identifiers cannot be empty. An identifier starts the list. The list can be extended by as many comma-identifier couples as needed.

  The `wait` statement construct is useful when you need to create additional dependencies for calculation of an expression on a set of variables where the expression does not contain any references to the variables in the set.

White Spaces

The tokens found in a valid DGEval code must be separated by at least one white space character. The developer is free to use as many white spaces as required to improve readability of the code.

A Valid DGEval Program

A valid DGEval program is a sequential list of statements that satisfy following conditions, which will be ensured by your program:

1. Statements are syntactically correct.
2. Statements are free of semantic errors.

Your Assignment

You will develop a C++ program that accepts one command line argument as the text file to be compiled to achieve semantic representation, which implements partially the intermediate representation that creates the basis for final translation. Your program will generate an output file having JSON format. The name of the output file will be based on the name of the input file. The extension of the input is assumed as `.txt`, so the input file name will be specified without extension. The output file will have `.json` extension.

Example run:

When your program is compiled and run as "`project3 test1`", your program will read the input file "`test1.txt`", and generate the output as "`test1.json`". In case your program fails to open the input file, your program will print "File not found!" message and terminate.

Processor Output

The processor you develop (`project3`) will accept the name of the source file that will be compiled. The program must contain only one parameter. No parameter or more than one parameter cases will be reported as errors. The processor must create an output file that conveys the symbol table, the list of the statements with augmented expression trees, and the messages, which are mainly the result of the syntactic and semantic checks.

The results generated by the processor you are required to develop must fully conform to the format that can be observed by the supplied samples.

Detected Symbols

The language processor is required to generate the symbol table of the identifiers defined by the statements. The symbol table will have entries with symbol name and type.

Expression Trees with Types

The language processor that you will develop is required to generate tree of expressions. Each node in the expression tree must have a type calculated during compile time. The type of the root node of the expression becomes the type of the expression.

Semantic Properties and Constraints

1. A variable in a DGEval program cannot be redefined.
2. Any runtime library function name cannot be redefined as a variable name.
3. An index in array access operator must be a number.
4. The first operand of a ternary operator can be a Boolean value only.
5. The comparison operators except "equal" and "not equal" operators cannot be applied on arrays and Booleans.
6. The boolean operators (and, or) are applied to booleans only.
7. Multiplication, subtraction, and division operators are applied to numbers only.
8. The + operator can be used in various configurations.
   a. `<number> + <number>` generates a `<number>`.
   b. `<string> + <string>` generates a `<string>`.
   c. `<string> + <number>` generates a `<string>`.
   d. `<number> + <string>` generates a `<string>`.
   e. When the first operand is array, the second operand is appended if its type is compliant with the type of array.

9. The not (`!`) operator is applied on a boolean value only.
10. The unary minus (`-`) operator is applied on a numeric value only.
11. A parameter list (actual parameters) passed to a function must conform to the formal parameters. This covers the number of parameters and the type equality of respective parameters.
12. The LHS of an assignment operator must be an identifier.
13. The first operand of a call operator (callee) can be an identifier only.
14. Array access operator can be applied to an array only.
15. The expression inside an array access operator may have one expression part only. Each operand of a comma operator is an expression part.
16. A function call without any parameter is possible.
17. Circular reference of calculations is not acceptable and must be reported. A statement affected by a circular reference will have an abstract syntax tree but will not be analyzed further semantically.
18. Any operator that is not applicable to the operand(s) found is an error in the code.
19. For the operators having two operands of type1 and type2, type1 must be identical to the type2.
20. Any array declared by an expression must produce elements of identical types.

Regulations & Hints

- **Implementation:** You should use Flex, Bison with C++ to develop your program. Make sure that your program will accept the name of the input file. In case the program is run without a parameter or with more than one parameter your program must terminate immediately with appropriate error message sent to the standard output.
- **Supplemental Material:** You will be provided with a supplemental code and an example set for support and standardization.
  - A standardized grammar for DGEval
  - A small library for fundamental data structures, runtime functions, and the language runtime.

  Note that you may be asked to develop your own extensions to any of these components.
- **Evaluation:** The evaluation will be based on correctness of each component (the symbol table, the expression tree, and the semantic check list).
- **Submission:** You need to submit all relevant files you have implemented (`.cpp, .h, .l, .y`, etc.) as well as a `README` file with instructions on how to build and run, in a single `.zip` file named `<your studentID>`.

Following notes are for more clarity to address possible concerns and/or questions:

- Use Flex and Bison to generate C++ not C file. C implementation will not be accepted.
- Never modify the automatically generated files generated by Flex and Bison. You may prefer consulting the recitation material and the sample project on semantic analysis.
- Describe precisely the steps to build and run your program in `README.txt` file, which is essential part of your submission. Provide any configuration items such as scripts, configuration files that will be necessary.

Appendix 1: Operators

Below are the operators that can be used in DGEval expressions. For each operator, form (coding syntax), possible applications on types, type of the result, associativity, and precedence rules are given.

OP	**Short Name**	**Form**	**Application**	**Result Type**	**Assoc.**	**Prec.**
`,`	Comma	Binary		Multi	LR	0
`=`	Assign	Binary, infix	`id=<expression>`	`Type of <expression>`	RL	1
`?:`	Conditional	Ternary	`<boolean>?<type1>:<type2>`	`<type1>`		2
`&&`	Boolean and	Binary, infix	`<boolean> && <bolean>`	`<boolean>`	LR	3
`||`	Boolean or	Binary, infix	`<boolean> || <bolean>`	`<boolean>`	LR	3
`==`	Equal	Binary, infix	`<type1> == <type2>`	`<boolean>`	LR	4
`!=`	Not equal	Binary, infix	`<type1> != <type2>`	`<boolean>`	LR	4
`<`	Less than	Binary, infix	`<type1> > <type2>`	`<boolean>`	LR	4
`>`	Greater than	Binary, infix	`<type1> < <type2>`	`<boolean>`	LR	4
`<=`	Less than or equal	Binary, infix	`<type1> <= <type2>`	`<boolean>`	LR	4
`>=`	Greater than or equal	Binary, infix	`<type1> >= <type2>`	`<boolean>`	LR	4
`+`	Add	Binary, infix	`<number> + <number>`	`<number>`	LR	5
`+`	Concatenate	Binary, infix	`<string> + <string>`	`<string>`	LR	5
`+`	Concatenate	Binary, infix	`<string> + <number>`	`<string>`	LR	5
`+`	Concatenate	Binary, infix	`<number> + <string>`	`<string>`	LR	5
`+`	Append	Binary, infix	`<array> + <type1>`	`<array>`	LR	5
`-`	Subtract	Binary, infix	`<number> - <number>`	`<number>`	LR	5
`*`	Multiply	Binary, infix	`<number> * <number>`	`<number>`	LR	6
`/`	Divide	Binary, infix	`<number> / <number>`	`<number>`	LR	6
`-`	Negate	Unary, prefix	`-<number>`	`<number>`	RL	7
`!`	Boolean negate	Unary, prefix	`!<boolean>`	`<boolean>`	RL	7
`()`	Call function	Binary	`id(<expression >)`	Return type of function	LR	8
`[]`	Array access	Binary	`<array>[<expression>]`	Element type	LR	8

Appendix 2: DGEval Runtime Library

DGEval runtime library includes a limited set of functions that can be accessed by using function call operator.

Index	**Prototype**	**Description**
0	number stddev(array)	Calculates standard deviation of the values in an array of numbers.
1	number mean(array)	Calculates the mean of the values in an array of numbers.
2	number count(array)	Calculates the count of the values in an array.
3	number min(array)	Calculates the minimum value found in an array.
4	number max(array)	Calculates the minimum value found in an array.
5	number sin(number)	Calculates the sine of a given number.
6	number cos(number)	Calculates the cosine of a given number.
7	number tan(number)	Calculates the tangent of a given number.
8	number pi()	Returns pi.
9	number atan(number)	Calculates the invers tangent of a given number.
10	number asin(number)	Calculates the inverse sine of a given number.
11	number acos(number)	Calculates the inverse cosine of a given number.
12	number exp(number)	Evaluates e powered to the given parameter.
13	number ln(number)	Calculates natural logarithm of the given parameter.
14	number print(string)	Prints the passed string on the console. Returns 1.
15	number random(number)	Returns a random number x for a given number y with the following condition. 0 <= x < y.
16	number len(string)	Returns the length of a given string.
17	number right(string, n)	Returns a new string containing rightmost n characters of the first parameter.
18	number left(string, n)	Returns a new string containing leftmost n characters of the first parameter.

Appendix 3: Language Runtime

The functions in the language runtime are not directly called by the DGEval call operator. These are the functions are called implicitly to implement related operators.

Index	Function	Description
0	array appendarray(value)	Appends two arrays returning a new array.
1	string appendstring(string)	Concatenates two strings
2	string allocatestring(string-data)	Generates a string.
3	string number2str(number)	Converts a number to a string.
4	string strcmp(string, string)	Compares two strings. Returns 1 if the first parameter is greater -1 if the second parameter is greater 0 if both strings are equal
5	Array allocatearray(type signature)	Allocates an empty array.
6	boolean arrcmp(array1, array2)	Evaluates equality of two arrays. Evaluates true if two are equal, false otherwise.

Appendix 4: Sample DGEval Program

Below is an error free sample DGEval program. Note that the first set of the supplemental material conveys the result JSON file that is expected after processing this sample.

```
a=25*2/sin(stddev([2,4,6])>b?c>t?sin(k*pi()/180):-0.5:2.5e-1);

k=24+b;

b=5+c/t;

wait a, d then print("Calculation complete");

c=4;

t=1;

d=k+t;
```