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

CENG444 - Language Processors

2023-2024 Spring
Project II
Syntactic analysis with Bison

Problem

In this assignment, you are required to write a C++ program that performs syntactic conformance test for an evaluator based on dependency graphs (DGEval). This evaluator will also be the basis of the forthcoming assignments on semantic analysis and code generation. The problem particular to this assignment is implementation of a syntax checker which requires definitions for scanner and parser, and implementation of a command line tool that reports syntactic conformity of a given DGEval module.

DGEval

A DGEval module is a list of DGEval statements, which are heavily dependent on expressions. A valid DGEval expression conforms to a set of syntactic rules that are mostly similar to the rules of the conventional expressions we are familiar with. The type system of DGEval is based on primitive types, which are string, number, and boolean. 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 list of the expressions.

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 help of an index, which must be a number.

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.

In this assignment, you are required to check only for syntactic conformance. Do not get into semantic details like type checking, type evaluation, precedence, associativity, constant folding, parameter checking, declarative integrity, and similar.

- **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.

- **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 whether it is exceptional or regular, can be coded with the 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**

  Zero or more expressions are coded between "[" and "]" to construct an array. Empty arrays are possible.

- **Function Calls**

  A function call is formed as <identifier> followed an expression list enclosed by parentheses. The list of expressions is skipped for the functions having no formal parameters. See Appendix 2: Sample DGEval Program.

Expression Lists

An expression list is formed by at least one expression. Additional expressions may be coded with separating commas to form a multi-expression list.

Statements

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

Expression Statements

An expression statement consists of an expression list 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.

DGEval Program

A valid DGEval program is a sequential list of statements that satisfy following conditions.

1. Statements must be syntactically correct.

   This will be ensured by your program.

2. Statements must be free of semantic errors.

   This will be ensured by your forthcoming assignment besides other requirements. You need not to worry about semantic checks in this experiment.

Processor Output

The processor you develop (`project02syn`) will accept the name of the source file that will be syntactically analyzed from the command line. The command must contain only one parameter. No parameter or more than one parameter cases must be reported as error. The processor must create an output file (`project02syn.txt`) containing 8 lines with collected frequencies in case it recognizes the input as a syntactically valid program. The strict ordering of the lines are given below.

```
<Total number of assignments>
<Total number of evaluations>
<Total number of conditionals>
<Total number of array literals>
<Total number of numeric constants>
<Total number of string constants>
<Total number of array accesses>
<Total number of function calls>
```

In case the input file is found syntactically non-compliant, the output file (**`project02syn.txt`**) will contain a single line as follows.

```
Line <line number>: Syntax error.
```

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.
- **Evaluation:** The evaluation will be based on correct diagnosis of 5 different input files, two points for each.
- **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 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 syntactic 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-list>)	Return type of function	LR	8
[]	Array access	Binary	<array>[<expression-list>]	Element type	LR	8

Appendix 2: Sample DGEval Program

```
a=25*2/f(stddev(k)>b?sin(k*pi/180):2.5e-1);
k=[2*l, exp(cos(c*pi/180)), b, c];
c=72;
l=b+24;
b=random(10);
wait r, q then print("K is: "+k+ "\n "), print("A is: "+a+ "\n ");
q=requestint(“http://ourserver.edu/dailylimit?k=”+k);
r=requestint(“http://ourserver.edu/labhours?k=”+a);
```