CSE340 Fall 2022 Project 2

Due: **October 28, 2022** by 11:59pm MST on GradeScope

Weeks of programmings can save you hours of planning – Unknown[1].

A shortcut is the longest distance between two points. – Charles Issawi (Economist)

I had a running compiler and nobody would touch it. They told me computers could only do arithmetic. – Grace Murray Hopper (Inventor of the first compiler)

1 General Advice

You should read the description carefully. Multiple readings are recommended. You will not be able to understand everything on a first reading. Give yourself time by starting early and taking breaks between readings. You will digest the requirements better.

- **The answers to many of your questions can be found in this document.**
- Do not start coding until you have a complete understanding of the requirements and a clear plan for the implementation.
- Ask for help early. I said it before and I say it again: I and the TAs can save you a lot of time if you ask for help early. You can get help with how to approach the project to make the solution easier and have an easier time implementing it. When you ask for help, you should be prepared and you should have done your part.
- Have fun!

2 Overview

The goal of this project is to introduce you to code generation for a simple straight line programs (no branching). You will write a C++ program that reads an input which is a program (sequence of assignments) written in a small programming language defined for this project. You program will read the sequence of assignments and your generate an intermediate code. The specific intermediate representation that your program will generate and what it does with that intermediate representation depends on a command-line argument that is passed to your program. We provide you with code to read the command line argument into an integer variable. Depending on the value of the variable, your program will invoke the appropriate functionality. The following are the three options (total 180 points and there are 50 points bonus , so if you get 130, that counts as 100% on the project (which is 13% of the final course grade); anything above 130 is bonus):

1. Task 1 (100 points): Parse the input and print an abstract syntax tree.
2. Task 2 (50 points): Parse the input and do type checking
3. Task 3 (30): Generate an *executable* representation of the program. The representation will be break down large expressions into smaller expressions that are linked together in a linked list that is passed to the "execute" function that we provided (and which you must not modify). This part is much more involved than the first two parts to do completely. Last semester, no one was able to get full credit on it even though two of your UGTAs got very close.

[1] I was made aware of this quote by a similar statement Dean Harmon (UGTA) made on discord: "Remember, days of debugging can save you hours of planning"

Unlike the first project, this project requires you to write a parser using operator precedence parsing (precedence table provided). The rest of the document is organized as follows:

- Section 3 describes the input format
- Section 4 describes what the input represents and introduces the type system rules for the small language
- Section 5 describes what the output of your program should be for each of the three tasks.
- Section 6 Gives examples of outputs for tasks 1–3.
- Section 7 discusses command line arguments and how you should run and test your program.
- Section 8 describes the grading scheme.
- Section 9 addresses some submission concerns.

3 Input Format

The following context-free grammar specifies the input format:

program	→	decl-section block
decl-section	→	scalar-decl-section array-decl-section
scalar-decl-section	→	SCALAR id-list
array-decl-section	→	ARRAY id-list
id-list	→	ID
id-list	→	ID id-list
block	→	LBRACE stmt-list RBRACE
stmt-list	→	stmt
stmt-list	→	stmt stmt-list
stmt	→	assign-stmt
stmt	→	output-stmt
assign-stmt	→	variable-access EQUAL expr SEMICOLON
output-stmt	→	OUTPUT variable-access SEMICOLON
variable-access	→	ID
variable-access	→	ID LBRAC expr RBRAC
variable-access	→	ID LBRAC DOT RBRAC
expr	→	expr MINUS expr
expr	→	expr PLUS expr
expr	→	expr MULT expr
expr	→	expr DIV expr
expr	→	LPAREN expr RPAREN
expr	→	expr LBRAC expr RBRAC
expr	→	expr LBRAC DOT RBRAC
expr	→	primary
primary	→	ID
primary	→	NUM

A program consists of a declaration section followed by a "block" which contains the statements to be executed. The declaration section consists of a scalar declaration section for declaring scalar variables followed by an array declaration section for declaring array variables. To keep things simple, there is no size specified for arrays (this is discussed further in the *Semantics* section). The "block" consists of a sequence of statements. We only have two kinds of statements: assignment statements and output statements. An assignment statement has a lefthand side which is a `variable-access` (representing an assignable variable) followed by an `EQUAL` followed by an expression followed by a semicolon. In this simple programming language, operations on whole arrays as well as assignments to whole arrays operations are supported, so a variable access can be a simple identifier or an element of an array or a whole array. The meaning of the various parts of the input is explained in Section 4 – *Semantics*.

The tokens used in the above grammar description are defined by the following regular expressions (dot operator omitted in the definitions):

```
SCALAR     = (S)(C)(A)(L)(A)(R)
ARRAY      = (A)(R)(R)(A)(Y)
OUTPUT     = (O)(U)(T)(P)(U)(T)
ID         = letter (letter | digit)*
NUM        = 0 | pdigit digit*
SEMICOLON  = ';'
EQUAL      = '='
LPAREN     = '('
RPAREN     = ')'
LBRACE     = '{'
RBRACE     = '}'
LBRAC      = '['
RBRAC      = ']'
MINUS      = '-'
PLUS       = '+'
MULT       = '*'
DIV        = '/'
DOT        = '.'
```

Where `digit` is the digits from 0 through 9 , `pdigit` is the digits from 1 through 9 and `letter` is the upper and lower case letters a through z and A through Z. Tokens are case-sensitive. Tokens are space separated and there is at least one whitespace character between any two successive tokens. We provide a lexer with a getToken() function to recognize these tokens. You should use the provided lexer in you solution.

4 Semantics

A program consists of a declaration section, which introduces the variables of the program, followed by a block which contains a sequence of statements to be executed. I describe each in what follows.

4.1 Declaration Section

The declaration section lists all the variables used in the program. There are two kinds of variables: scalar variables and array variables.

Scalar Variable Variable names that appear in the scalar-decl-section subtree of the program parse tree are *scalar variables*. For our purposes, you can assume that a scalar variable can be represented as a `long` integer in the implementation.

Array Variables Variable names that appear in the array-decl-section subtree of the program parse tree are *array variables*. Array variables represent arrays of integers. To keep things simple, we do not specify the size of these arrays. You can assume that each array has size 10. Same as for scalar variables, we represent each entry in the array as a `long` integer in the implementation.

4.2 Locations and Values

Variables have memory locations associated with them. The memory location associated with a variable contains the *value* of the variable. The value of scalar variables is a scalar integer value. The value of array variables is a an array of 10 scalar integer values.

4.3 Type System

The type system specifies the rules under which assignments are valid (type compatibility rules) and the type of expressions given the types of their constituent parts (type inference rules).

The type inference rules for expressions are the following:

1. If x is a lexeme that appears in the *id-list* of the *scalar-decl-section*, then the expression x has type **scalar**.
2. The type of an expression consisting of a single *NUM* is **scalar**.
3. If x appears in the *id-list* of the *array-decl-section*, then the expression x[.] has type **array**.
4. If x appears in the *id-list* of the *array-decl-section*, and *expr* has type **scalar**, then the expression x[*expr*] has type **scalar**.
5. If $expr_1$ and $expr_2$ have type **scalar**, then expr_1 *OP* $expr_2$ (where *OP* is *PLUS*, *MINUS*, *MULT* or *DIV*) has type **scalar**.
6. If $expr_1$ and $expr_2$ have type **array**, then expr_1 *OP* $expr_2$ (where *OP* is *PLUS* or *MINUS*) has type **array**.
7. If $expr_1$ and $expr_2$ have type **array**, then expr_1 *MULT* $expr_2$ has type **scalar**.
8. If $expr_1$ has type **array** and $expr_2$ has type **scalar**, then $expr_1[expr_2]$ has type **scalar**.
9. If $expr_1$ has type **scalar**, then $expr_1$[.] has type **array**.
10. If $expr_2$ has type other than **scalar**, then $expr_1[expr_2]$ has type **error**.
11. If $expr_1$ has type **scalar** or **error**, then $expr_1[expr_2]$ has type **error**.
12. If $expr_1$ and $expr_2$ have different types, then $expr_1$ *OP* $expr_2$ (where *OP* is *PLUS* ,*MINUS MULT* or *DIV*) has type **error**.
13. If $expr_1$ and $expr_2$ have type **array**, then expr_1 *DIV* $expr_2$ has type **error**.
14. If x is a lexeme that does not appear in the *id-list* of the *scalar-decl-section* or the *id-list* of the *array-decl-section*, then the expression x has type **error**.
15. If none of the above conditions hold, the expression has type **error**.

The type inference rules for variable-access are the following:

1. If x is a lexeme that appears in the *id-list* of the *scalar-decl-section*, then the variable access x has type **scalar**.
2. If x is a lexeme that appears in the *id-list* of the *array-decl-section* and *expr* has type **scalar**, then x[*expr*] has type **scalar**.
3. If x is a lexeme that appears in the *id-list* of the *array-decl-section*, then x[.] has type **array**.
4. If x is a lexeme that does not appear in the *id-list* of the *scalar-decl-section* then the variable access x has type **error**.
5. If x is a lexeme that does not appear in the *id-list* of the of the *array-decl-section*, then the variable access x[.] has type **error**.
6. If x is a lexeme that does not appear in the *id-list* of the *array-decl-section*, then the variable access x[*expr*] has type **error**.

The following is the only type compatibility rule for assignments:

1. An assignment of the form *variable-access* = *expr* is valid if the type of the *variable-access* is array or the type of *expr* is scalar.

Note that this compatibility rule allows assigning an array to an array, a scalar to a scalar as well as a scalar to an array. The meaning of assigning a scalar to an array is given in the next section.

4.4 Semantics of Expressions and Assignments

Note: You can omit this subsection on a first reading as it relates to Task 3 which you can start on after finishing with Tasks 1 and 2.

In this section, we define how expressions are evaluated and how assignments are executed. The description is independent of the specific code that your program will generate to evaluate expressions and execute assignments. The description in this section does not refer to the location of variables.

4.4.1 Variables, expressions, and values

We say that a variable x is a *scalar variable* if x appears in the *id-list* of the *scalar-decl-section* of the program. We say that a variable x is an *array variable* if x appears in the *id-list* of the *array-decl-section* of the program. Each variable has a value associated with it. We denote by S the set of scalar variables in the program and by A the set of array variables in the program. The sets S and A are disjoint and you can assume so in your implementation.

As the program executes, the values of variables can be changed by the statements being executed. The *state* of a program consists of: (1) the values of the variables of the program, (2) the program code, which, for our language, is a list statements to be executed, and (3) the next statement to be executed (in assembly, this is specified by the program counter). In this section, we are describing the execution at the statement level. In the implementation guide, we describe how a complex statement can be broken down into a sequence of statements. The description that follows will be a mix of formal and informal specification of the semantics of program execution.

The value of a scalar variable is an `integer` (`long` in the implementation). The value of an array variable is a vector of 10 integer values (array of `long` in the implementation). The content of the *memory* is specified by specifying the values of all the variables. More formally, we define the *memory* of the system to be a mapping that maps variables to values. $\mathcal{M} : S \cup A \mapsto \texttt{integer} \cup \texttt{integer}^{10}$, where $\texttt{integer}^{10}$ is the cartesian product $\texttt{integer} \times \texttt{integer} \times \ldots \times \texttt{integer}$ (10 times). If $x \in S$, $\mathcal{M}(x) \in \texttt{integer}$. If $x \in A$, $\mathcal{M}(x) \in \texttt{integer}^{10}$. The mapping $\mathcal{M}$ changes as assignment statements are executed but is not affected by output statements. In what follows, we refer to the effect of the execution of assignment statements by specifying how the values of variables change or stay the same after the execution of the assignment, but we do not explicitly give a formal definition of the new mapping or of a transition function.

The initial values of all variables are zero. Initially, $\mathcal{M}(x) = 0$ for every $x \in S$ and $\mathcal{M}(x) = 0^{10}$ for every $x \in A$ (0^{10} is a vector of 10 values, all of which are equal to 0).

4.4.2 Evaluating expressions

The value of an expression is defined recursively as follows:

- **base case**
  1. The value of the expression x, where x is a scalar or array variable, is equal to the value of x
  2. The value of the expression n, where n is NUM, is equal to the integer value corresponding to n (i.e. corresponding to the lexeme of n).
- **induction**
  1. Let x and y be two expressions that have type **scalar** and values v_x and v_y, respectively,
     - The value of the expression $x + y$ is $v_x + v_y$, where the addition is integer addition (`long` in the implementations).
     - The value of the expression $x - y$ is $v_x - v_y$, where the subtraction is integer subtraction (`long` in the implementations).
     - The value of the expression $x * y$ is $v_x * v_y$, where the multiplication is integer multiplication (`long` in the implementations).
     - The value of the expression x/y is v_x/v_y, where the division is integer division (`long` in the implementations).
  2. Let a be an expression that has type **array** and value v_a and x be an expression that has scalar type and value v_x.
     - The value of $a[x]$ is equal to $v_a[v_x]$ if $0 \leq v_x \leq 9$.
     - The value of $a[x]$ is undefined if $v_x < 0$ or $9 < v_x$.
  3. Let a and b be two expressions that have type **array** and values v_a and v_b respectively.
     - The value of the expression $a[.] + b[.]$ is an array value $c \in \texttt{integer}^{10}$ such that $c[i] = v_a[i] + v_b[i]$, $0 \leq i \leq 9$, where the addition is integer addition (`long` in the implementations).
     - The value of the expression $a[.] - b[.]$ is an array value $c \in \texttt{integer}^{10}$ such that $c[i] = v_a[i] - v_b[i]$, $0 \leq i \leq 9$, where the subtraction is integer subtraction (`long` in the implementations).
     - The value of the expression $a[.] * b[.]$ is a scalar value $c = \sum_{i=0}^{i=9} v_a[i] * v_b[i]$, where the multiplication is integer multiplication (`long` in the implementations).
  4. Let x be and expression that has type **scalar** and value v_x,
     - The value of the expression $x[.]$ is equal to $(v_x, v_x, \ldots, v_x)$, an array of size 10 in which all entries are equal to v_x.
  5. Let x be an array variable with value v_x,

– The value of the expression $x[.]$ is equal to v_x

Note that we say that x is an array variable and not "has type **array**" because when x is an array variable, the type of the expression x is not really defined according to the type inference rules!!

6. If none of the above conditions hold, the value of the expression is undefined. In your program, you do not have to handle expressions whose values are undefined.

4.4.3 Executing Assignment Statements

The values of variables are changed by the execution of assignment statements. In general assignments have the form *variable-access* = *expr*, so we define the effect of executing valid assignment (see definition in Section 4.3) depending on the types of both sides.

1. Let $x = expr$ be an assignment such that both sides have type scalar. After executing the assignment, the value of x is v_e and the values of all other variables are unchanged.

2. Let $x[expr_1] = expr_2$ be an assignment such that x is an array variable and $expr_1$ and $expr_2$ have type **scalar** with values v_1 and v_2 respectively, $0 \leq v_1 \leq 9$. After the assignment is executed, $\mathcal{M}(x)[v_1] = v_2$ and the values of all other entries of x and the values of all other variables are unchanged.

3. Let $x[.] = expr$ be an assignment such that x is an array variable and $expr$ has type **array** with value $v \in \texttt{integer}^{10}$. After the assignment is executed, $\mathcal{M}(x) = v$ and the values of all other variables are unchanged.

4. Let $x[.] = expr$ be an assignment such that x is an array variable and $expr$ has type **scalar** with value v. After the assignment is executed, each entry in the array x has value v and the values of all other variables are unchanged. In other words, $\mathcal{M}(x)[i] = v$, $0 \leq i \leq 9$ after the execution of the statement and the values of all other variables are unchanged.

5. If none of the above conditions hold, the semantics of the assignment statement are undefined. In your program, you do not have to handle assignment statements with undefined semantics.

4.4.4 Executing Output Statements

The execution of an output statement results in outputting the value of the variable access.

1. The execution of OUTPUT x, where x is a variable access that has type **scalar**, outputs the value $\mathcal{M}(x)$.

2. The execution of OUTPUT x, where x is an array variable, outputs the value $\mathcal{M}(x)$. In other words, the ten values $\mathcal{M}(x)[i]$, $0 \leq i \leq 9$, are outputted.

3. The execution of OUTPUT $x[expr]$, where x is an array variable and $expr$ is an expression that has type **scalar** and value v, $0 \leq v \leq 9$, outputs the value $\mathcal{M}(x)[v]$.

4. If none of the above conditions hold, the semantics of the output statement are undefined. In your program, you do not have to handle output statements with undefined semantics

4.4.5 Executing a List of Statements

The execution of a list of statements $s_1, s_2, \ldots, s_k$ is equivalent to the execution of s_1, then executing the list of statements $s_2, \ldots, s_k$. In other words, all the statements are executed in order one after the other.

5 Output Specifications: Tasks 1 – 3

5.1 Task 1: Parsing

For this task, you are asked to parse the input program and output an abstract syntax tree or an error message.

- If there are no syntax errors, your program should print an abstract syntax tree (defined below) of the first statement of the input. So, even if the statement list has multiple statements, only the abstract syntax tree for the first statement will be printed.

- If there is syntax error in the input program, your program should output the following syntax error message:

	+	-	*	/	(	)	[	.	]	num	id	$
+	>	>	<	<	<	>	<	err	>	<	<	>
-	>	>	<	<	<	>	<	err	>	<	<	>
*	>	>	>	>	<	>	<	err	>	<	<	>
/	>	>	>	>	<	>	<	err	>	<	<	>
(	<	<	<	<	<	=	<	err	<	<	<	err
)	>	>	>	>	err	>	>	err	>	err	err	>
[	<	<	<	<	<	<	<	=	=	<	<	err
.	err	err	err	err	err	err	err	err	=	err	err	err
]	>	>	>	>	err	>	>	err	>	err	err	>
num	>	>	>	>	err	>	>	err	>	err	err	>
id	>	>	>	>	err	>	>	err	>	err	err	>
$	<	<	<	<	<	err	<	err	err	<	<	acc

Table 1: Operator Precedence Parser Table

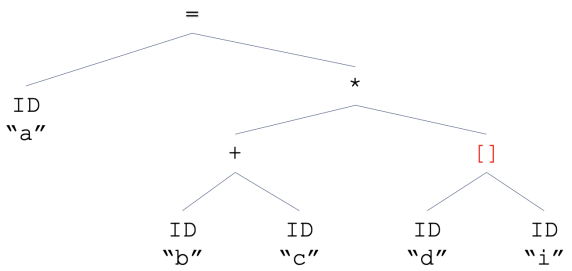

Figure 1: Abstract syntax tree for `a = (b+c)*d[i];`

```
SNYATX EORRR !!!
```

and exit.

For parsing, you need to combine predictive recursive descent parsing with operator precedence parsing. The operator precedence parser is invoked for parsing expr. The operator precedence parsing table is given in Table 1.

Next, I define what an abstract syntax tree is and give examples input programs with corresponding outputs.

5.1.1 Abstract Syntax Trees

An abstract syntax tree is a tree that captures the essential syntax of the input. For example, a program block has curly braces around a stmt-list. The curly braces are part of the block's *concrete syntax* but are not really needed to understand the structure of a block when that structure is represented as a tree.

The abstract syntax tree for the assignment statement `a = (b+c)*d[i];` is shown in Figure 1.

Notice how the parentheses are not needed in the abstract syntax tree and the semicolon is also omitted. Also, notice how the assignment operator `=` and the array access operator `[]` are included as labels of tree nodes.

For this project, you are only asked to print the abstract syntax tree for the first assignment statement of a test case. Figures 2 and 3 give recursive definitions of abstract syntax trees for assignment statements and expressions. In the figure, the tree of a construct is defined in terms of the tree(s) of its constituent parts. For example, the tree for expr_1 PLUS expr_2 is defined in terms of the abstract syntax trees $\texttt{T}_{expr1}$ and $\texttt{T}_{expr1}$ of expr_2 and expr_2 respectively.

5.1.2 Output Format

The output for this task should be the abstract syntax tree of the first assignment statement in the input program printed according to a breadth first traversal of the tree (you need to brush up on breadth first traversal from CSE310).

5.1.3 Examples

The following two examples show the output for three input programs. In the output, each level of the tree is printed on a separate line. That is not required and you can print the whole tree on one line, but I think it would be useful to keep the

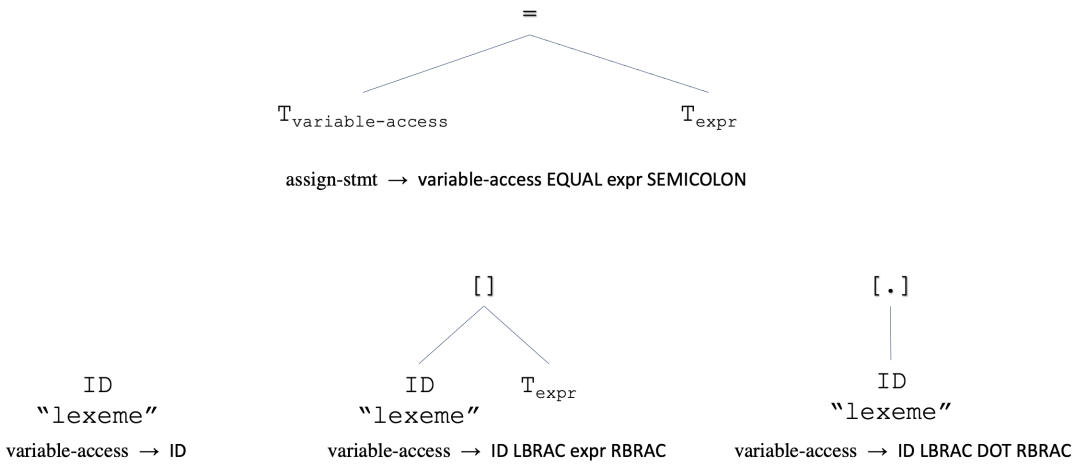

Figure 2: Abstract syntax trees definitions for assignment and variable access

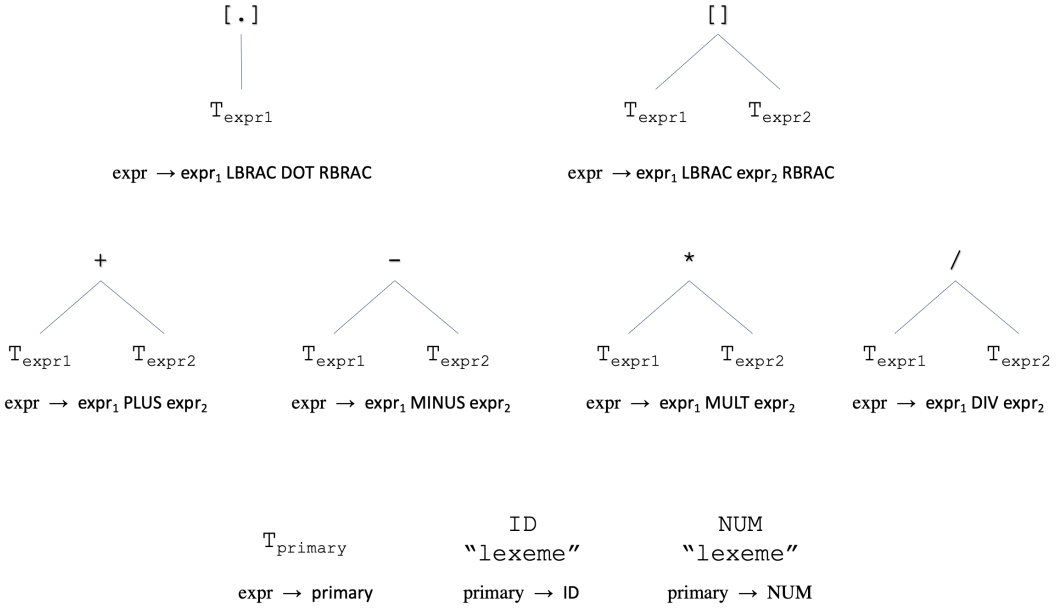

Figure 3: Abstract syntax trees definitions for expressions

separation so that you can more easily inspect the output that your program generates.

Example 1

```
SCALAR x y z
ARRAY a b c
{
    y = 1+x-z[2];
    y = 1;
    a[x] = z[y];
}
```

```
          =
ID "y"                    -
                  +                 []
            NUM "1"    ID "x"  ID "z"   NUM "2"
```

First, note how the output is only shown for the first assignment statement, as required, and not for the whole program. The output is shown here with spacing added to show you how it really corresponds to an abstract syntax tree. In your output, you do not need to add space and you do not even need to have each level on a separate line, but that is advisable so that you can better understand the output that your program is generating and it will help with debugging. Finally, Notice how `z` is used as an array in this example even though it is not declared as an array. The fact that there is a problem with the way `z` is accessed has no bearing on this task.

Example 2

```
SCALAR x y z
ARRAY a b c
{
    a[x+y] = 1+x*a[z]-a[a[z]];
    y = 1;
    a[x] = c[y];
}
```

```
                  =
       []                                    -
ID "a"        +                    +                          []
      ID "x"    ID "y"   NUM "1"          *            ID "a"     []
                                    ID "x"      []            ID "a"  ID "z"
                                           ID "a" ID "z"
```

Again, the added spacing is not required and it is there to make it easier for you to identify the various part of the assignment statement.

Example 3

```
SCALAR x y z
ARRAY a b c
{
    a[.] = 1[.]+b[.]-c[.];
    y = 1;
    a[x] = b[y];
}
```

```
              =
    [.]                         -
   ID "a"            +                 [.]
               [.]      [.]          ID "c"
             NUM "1"   ID "b"
```

This last example illustrates operations and assignment on whole arrays.

5.2 Task 2: Type Checking

For this task, you are asked to do basic type checking on assignment statements. There are three possible types for expressions: **scalar**, **array** and **error**. The type system defines *type inference* rules to determine the type of expressions and *type compatibility* rules to determine when an assignment is valid. For this task the output is defined as follows:

- If no expression has type error and all assignments are valid, the output should be:

```
Amazing! No type errors here :)
```

- If some expression has type error, the output should be

```
Disappointing expression type error :(

Line #1
Line #2
...
Line #k
```

  Where the line numbers are the numbers of the lines in which an expression type error occurs. For this part, you can assume that the righthand sides of assignments will not be broken up into multiple lines.

- If no expression has type error, but some assignment is invalid, the output should be

```
The following assignment(s) is/are invalid :(

Line #1
Line #2
...
Line #k
```

  Where the line numbers are the numbers of the lines with invalid assignments. For this part, you can assume that an invalid assignment will not be broken up into multiple lines.

5.3 Task 3: Code Generation and Program Execution

This task is involved and the points assigned to it are not proportional to the effort that is required to finish it. The points assigned to tasks 1 and 2 will get you to above A. The points assigned to this task will get you to A+. You should not work on this task before finishing tasks 1 and 2.

For this task, you can assume that the program has no type errors and that all expressions have well-defined values and that all assignments have well-defined semantics (see section above on semantics of expressions and assignments).

For this task, the output of your program should be a pointer to a data structure that will be *executed* by the execute function that we provided. The execution will produce the output that the assignment statements produce.

The data structure that your program generates will break down large expressions into a sequence of assignments. For example, `a = b+c+d+e` can be broken down into

```
t1 = b + c;
t2 = t1 + d;
t3 = t2 + e;
a = t3;
```

Your program will not be generating a new program like the one above, instead, it will generate a linked list represents the code above. More details on the linked list representation is given in the implementation guide.

6 Examples

6.1 Task 1

I already gave three examples for this task. Here is one more example that shows the syntax error message.

Example

```
SCALAR x y z
ARRAY a b c
{
    x = a[.]*b[.] - b[.]*c[.]
}
```

```
SNYATX EORRR !!!
```

6.2 Task 2

Example 1

```
SCALAR x y z
ARRAY a b c
{
    x = a[.]*b[.] - b[.]*c[.];
}
```

```
No type errors here :)
```

Example 2

```
1:    SCALAR x y z
2:    ARRAY a b c
3:    {
4:        x = a[.]*b[.] - b[.];
5:        x = a[.]*b[.] - b[.]*c[.];
6:        x = a[.]*b[.] - b[.];
7:   }
```

Note that in this example, line number are added to the program to be able to refer to the lines. The actual program will not have the line numbers.

```
Expression type error :(

Line 4
Line 6
```

Example 3

```
1:    SCALAR x y z
2:    ARRAY a b c
3:    {
4:        x = (a[.]*b[.] - b[.]*c[.])[.];
5:        a[.] = (a[.]*b[.] - b[.]*c[.])[.];
6:        x = (a[.]*b[.] - b[.]*c[.])[.];
7:   }
```

Note that in this example, line number are added to the program to be able to refer to the lines. The actual program will not have the line numbers.

```
Invalid assignment :(

Line 4
Line 6
```

6.3 Task 3

I give two examples for Task 3. One example shows whole array operations and one example shows individual array operations.

Example 1

```
1:     SCALAR x y z
2:     ARRAY a b c
3:     {
4:         x = 1;
5:         y = 2;
6:         z = 3;
7:         a[x] = x;
8:         a[y] = a[a[a[x]]] + 2;
9:         b[x] = a[x]+a[a[x]+a[a[y]]-1];
10:        OUTPUT a[1];
11:        OUTPUT a[2];
12:        OUTPUT b[x];
12:    }
```

In this example, line 7, sets `a[1] = 1`. Line 8 sets `a[2] = 3` because `x = 1` , so `a[x] = 1`, `a[a[x]] = 1` and `a[a[a[x]]] = 1`. Line 9 sets `b[1] = a[x]+a[a[x]+a[a[y]]-1] = 1 + a[1+a[3]-1] = 1 + a[0] = 1`.

The output of the execution (not of your program)

```
1
3
1
```

Note that in this example, line number are added to the program to be able to refer to the lines. The actual program will not have the line numbers.

Example 2

```
1:     SCALAR x y z
2:     ARRAY a b c
3:     {
4:         x = 1;
5:         y = 2;
6:         z = 3;
7:         a[x] = z;
8:         a[y] = z+1;
9:         b[.] = a[.];
10:        c[.] = a[.]+1[.];
11:        x = a[.]*b[.] - b[.]*c[.];
12:        OUTPUT x;
13:        OUTPUT y;
14:        OUTPUT z;
15:     }
```

This example shows whole-array operations in the two assignment statements: `b[.] = a[.]` and `c[.] = a[.]+1[.]`. The value of the array `a` before these assignments is `(0,3,4,0,0,0,0,0,0,0)`. So, after the assignment on line 9, the value of `b =`

`(0,3,4,0,0,0,0,0,0,0)`. The assignment on line 10 adds `(1,1,1,1,1,1,1,1,1,1)` to the array `a` and assign the result to the array `c`. So, after adding `(1,1,1,1,1,1,1,1,1,1)` to `a`, the value of `c` becomes `(1,4,5,1,1,1,1,1,1,1)`. The assignment on line 11 is calculated as follows:

```
a[.]*b[.] = (0,3,4,0,0,0,0,0,0,0)*(0,3,4,0,0,0,0,0,0,0) = 25
b[.]*c[.] = (0,3,4,0,0,0,0,0,0,0)*(1,4,5,1,1,1,1,1,1,1) = 32
a[.]*b[.] - b[.]*c[.] = 25 - 32 = -7
```

The output is

```
-7
2
3
```

7 Provided Code

7.1 Lexer

A lexer that can recognize the tokens is provided for this project. You are required to use it and you should not modify it.

7.2 Reading command-line argument

As mentioned in the introduction, your program must read the grammar from stdin and the task number from command line arguments. The following piece of code shows how to read the first command line argument and perform a task based on the value of that argument. Use this code as a starting point for your main function.

```
/* NOTE: You should get the full version of this code as part of the project
   material, do not copy/paste from this document. */

#include <stdio.h>
#include <stdlib.h>

int main (int argc, char* argv[])
{
    int task;

    if (argc < 2) {
        printf("Error: missing argument\n");
        return 1;
    }

    task = atoi(argv[1]);

    switch (task) {
        case 1:
            // TODO: perform task 1.
            break;

        // ...

        default:
            printf("Error: unrecognized task number %d\n", task);
            break;
    }
    return 0;
}
```

7.3 Testing

You are provided with a script to run your program on all tasks for each of the test cases. The test cases that we provided for this project are not extensive. **They are meant to serve as example cases and are not meant to test all functionality**. The test cases on the submission site will be extensive. **You are expected to develop your own additional test cases based on the project specification**.

To run your program for this project, you need to specify the task number through command line arguments. For example, to run task 3:

```
$ ./a.out 3
```

Your program should read the input grammar from standard input. To read the input grammar from a text file, you can redirect standard input:

```
$ ./a.out 3 < test.txt
```

For this project we use 3 expected files per each test case input. For an input file named test.txt , the expected files are test.txt.expected1, test.txt.expected2 and test.txt.expected3, corresponding to tasks 1 through 3. The test script test_p3.sh , provided with the project material, takes one command line argument indicating the task number to use. So for example to test your program against all test cases for task 2, use the following command:

```
$ ./test_p2.sh 2
```

To test your program against all test cases for all tasks, you need to run the test script 5 times:

```
$ ./test_p3.sh 1
$ ./test_p3.sh 2
$ ./test_p3.sh 3
```

8 Evaluation

Your submission will be graded on passing the automated test cases. The test cases (there will be multiple test cases in each category, each with equal weight) will be broken down in the following way (total 115 points, so 15 points bonus):

- Task 1: 100 points
- Task 2: 50 points
- Task 3: 30 points
  - 15 without supporting whole-array operations
  - 10 points for supporting whole-array operations

For each category, the grade will be proportional to the number of test cases for which your program produces correct output on canvas.

9 Submission

Submit your individual code files on GradeScope.

- **Do not submit .zip files even if Gradescope says it is ok.**
- **Do not modify and do not submit the provided inputbuf.h, inputbuf.cc, lexer.h and lexer.cc files.**

Important Note. For this project, there is a timeout that we enforce when testing submissions. Programs that are functionally correct but that take an inordinate amount of time can be timed out before finishing execution. This is typically not an issue because the timeout period is generous, but if **your implementation is very inefficient, it risks being timed out and you will not get credit for test cases for which the program is timed out**.