Luis Alonso (lra2103)   COMS W4115-001
Hila Becker (hb2143)
Kate McCarthy (km2302)
Isa Muqattash (imm2104)

The CEAS Language

Language Reference Manual

1. Lexical Conventions

1.1   Comments

CEAS supports common single and multi-line comments. The character combination "//" denotes single-line comments. Multi-line comments begin with "/*" and end with "*/". It is illegal to embed multi-line comments within either style of comment. However, single-line comments may be embedded within a multi-line comment block.

1.2   Identifiers

Identifiers may be of arbitrary length and must consist of a letter or an underscore followed by any combination of letters, digits, and underscores. CEAS is case sensitive, so the identifiers my_var and My_Var are considered to be distinct identifiers.

1.3   Keywords

The following identifiers are reserved as keywords within the CEAS language:

for	while	if	else	do
true	false	int	boolean	string
Page	break	continue	include	return
char	function	void		

1.4   Literals

Literals in the language are integers, characters, strings, booleans, or lists.

1.4.1   Integers

An integer consists of a sequence of one or more consecutive digits.

1.4.2   Characters

Characters represent individual letters and symbols from the ASCII standard. They must be written within single-quotes ('). The single-quote character itself is represented using two single quotes.

1.4.3   Strings

Strings are represented as any sequence of acceptable characters found between double-quotes ("). Double-quotes can be embedded within strings by placing two double-quotes next to each other (""). For example, the string """" is a one-character string consisting of the double-quote character.

1.4.4   Boolean

The two boolean constants, *true* and *false* are keywords in the language and represent the logical values.

1.4.5   Lists

A list literal is created by enumerating all of the elements in a list. Each list element is separated from the following element by a comma and the whole expression is enclosed in '[' and ']'. All list elements must be of the same type or be expressions that evaluate to the same type.

1.5   Other tokens

The following characters and sequences of characters all have meaning:

{	}	(	)	[	]
.	,	=	<	<=	>
>=	==	!=	!	+	-
%	/	*	;	+=	-=
*=	/=	&	|	:	

2. Types

The language is strongly typed. Variables are bound to a type at declaration. Constants are bound to a type based on their format as described above. The following types are supported:

- *int*   : 32-bit integers
- *char*   : single character from the ASCII character set
- *string*   : list of characters
- *boolean*   : either true or false (both keywords)
- *Page*   : logical representation of a web address
- *List*   : collection of elements of a single type
- *void*   : used for functions that do not return values

3. Expressions

Expressions are listed below in order of precedence.

3.1   Primary Expressions

Primary expressions are literals, identifiers, list access, function calls, and expressions contained within "(" and ")".

3.1.1   Literals

Literals are integers, characters, strings, or booleans as defined above. They evaluate to the expressed value and their type is determined by the interpreter.

3.1.2   Identifiers

Identifiers evaluate to the value they were bound to last. The type is determined by the type assigned to the identifier at declaration.

3.2   List Access

Access to the nth element of list *A* is written as:

     A[*<expression>*]

where *<expression>* is an integer expression that evaluates to *n*. Bounds checking is performed by the interpreter. The type of this expression is equivalent to the type of the elements contained within the list.

3.3   Function Calls

Functions are invoked by naming the function followed by a comma-separated list of parameters contained within "(" and ")". The parameter list is optional but the parentheses are not. The type of the function is the type returned by the function.

Functions may have return types, in which case, the expression is of the same type as the function.

3.4   Parenthesized Expressions

Parentheses are used to ensure proper precedence. The value of a parenthesized expression is the equal to the result of the expressions contained within the parenthesis.

3.5   Arithmetic Expressions

CEAS supports multiplication, integer division, modulus, addition, and subtraction. These expressions take primary expressions of type int as operands.

Unary Operators

'+' and '–' may be used as prefix unary operators on integer expressions. A '+' before an expression returns the value of that expression. A '–' returns the negative of the expression.

Binary Operators

Multiplication (*), division (/), and modulus (%) have the highest precedence and associate from left to right.

Addition (+) and subtraction (-) are at the next level of precedence and also associate from left to right.

3.6   Boolean Expressions

Boolean expressions always return boolean values and consist of relational expressions and logical expressions

3.7   Relational Expression

All relational expressions evaluate to the boolean values *true* or *false*. All are binary operators and grouped left to right. The operators accept arithmetic expressions and primary expressions that evaluate to numeric values as operands. The operators are:

- >   : Greater than
- <   : Less than
- >=   : Greater than or equal to
- <=   : Less than or equal to
- !=   : Not equal
- ==   : Equal

3.8   Logical Expressions

Logical expressions consist of an operator that accepts boolean expressions as operands and produces a boolean value. They are listed below from highest to lowest precedence:

- !*bool*   : Not operator
- *bool* & *bool*   : And operator
- *bool* | *bool*   : Or operator

4. Statements

A statement represents a command in the language and is always terminated with a semi-colon (;). Statements are either an expression or one or more statements separated by semi-colons. In general, program flow proceeds from the first statement in a file to the last. Program flow can be modified with conditional and iterative statements or with function calls. The different types of statements are described in detail below.

4.1   Statement Blocks

Statements can be grouped within '{' and '}'. Statements contained between these characters are treated as a single statement. Blocks may be legally nested within other blocks.

4.2   Identifier Declaration

All variables must be declared before first use. Declaration statements are as follows:

> *<type> <identifier>*;

where *identifier* is any legal identifier that has not yet been declared and *<type>* is either int, char, boolean, string, or Page. Multiple variables of the same type may be declared on the same line by separating the variable identifiers with commas. For example:

> int var2, var3, var4;
> char c1, c2, c3;

are valid declarations.

4.2.1   List Declarations

List declarations are a special case of the general declaration rules. Lists are declared as follows:

> *<type> <varname>[<size>]*;

Where *<type>* is one of the types, *<varname>* is a legal identifier, and *<size>* is an integer expression. When the interpreter encounters a list declaration it will create a space in memory to hold *size* values.

As with other declarations, multiple lists of the same type and size may be defined on the same line by separating identifiers with commas. The following are all legal declarations:

> int bigList[10];
> char smallCharList[2], anotherSmallList[3];
> Page pages[10];

List declarations may be mixed with non-list declarations of the same type:

> int x, small[4], y; // declares two integers and an integer list

4.3   Assignment

Identifiers can be assigned a value using the assignment operator, '='. An identifier must be declared in a separate statement before it can be used in an assignment statement. The identifier appears on the left-hand side and an expression appears on the right-hand side. The type of the expression must match the type assigned to the identifier at declaration.

> *<identifier> = <expression>*;

The semi-colon is required as a statement terminator.

4.3.1   Assignment at Declaration

As a shortcut, variables can be assigned a value at declaration. This takes the form of:

>   *<type> <identifier> = <expression>*;

<expression> may be any valid expression that evaluates to a value of the same type as *<identifier>*.

When declaring and initializing lists it is not necessary to declare the size of the list since the size is implied by the right side of the assignment. The following is legal:

>   int numList[] = [0, 1, 2, 3, 4];

4.4   If Statements

If statements allow the programmer to choose, at runtime, which commands in a program will be executed. They have the following format:

>   if (*<boolean_expression>*) *<statement>*
>   if (*<boolean_expression>*) *<statement>* else *<statement>*

In both versions of the if statement the boolean expression is evaluated to either true or false. If it evaluates to true the first substatement is evaluated. In the second version of the if statement, the second substatement will be evaluated when the boolean expression evaluates to false. An else is always connected to the most recent else-less if in the current block of statements.

4.5   Iterative Statements

The following iterative statements are defined:

>   while (*<boolean_expression>*) *<statement>*
>   do *<statement>* while (*<boolean_expression>*)
>   for (*<identifier>* = *<integer_expression>* : *<integer_expression>*) *statement*

The while statement will execute its substatement as long as the boolean expression evaluates to true. Note that if the boolean expression evaluates to false at the first iteration, the substatement will never be executed. The do statement is similar, except that it guarantees that the substatement will be executed at least once. It will continue to execute the statement as long as the boolean expression evaluates to true.

The for statement allows for iteration over a range of numbers. The identifier in the for statement must be an integer. The integer values following the identifier form a range of values. The substatement will be executed for each value in the range, inclusively. For example, the range 0:5 will have six iterations.

4.5.1　　　　　Break Statement

When the keyword **break** is encountered in a statement block, the program continues at the end of the encapsulating iterative statement. A break statement encountered outside of an iterative block is not a legal statement.

4.5.2　　　　　Continue Statement

When encountered within an iterative block, the **continue** statement will cause the program to immediately jump to the next iteration. A continue statement encountered outside of an iterative block is not a legal statement.

4.6　　　　　Function Definition

Functions are defined with this syntax:

```
<type> function <func_name> ( <variable_list> ) {
    <statement>
}
```

<func_name> is a legal identifier used to refer to the function. *<variable_list>* is an optional list of variable declarations separated by commas. *<statement>* is zero or more statements that make up the body of the function. *<type>* is the optional return type of the function. It may be omitted if the function does not return a result.

Function definitions may not appear within other functions.

Functions are uniquely identified by their function signature which includes the name of the function and the list of types the function accepts as parameters.

Once a function has been defined, it is available to any statement that follows it.

4.7　　　　　Return Statement

When encountered in the body of a function, the **return** statement will cause program control to return to the calling statement. An optional expression following the **return** keyword will cause the function to return the result of the expression to the calling statement. The interpreter will exit the program if a **return** statement is encountered at the top most level of the program.

A return statement may look like:

```
        return;
```
or
```
        return <expression>;
```

4.8　　　　　Include Statement

The **include** statement is used to include previously written programs in the current program. It uses this syntax:

```
        include <file_name>;
```

<file_name> is a string referring to a physical file on the drive. It is assumed to be relative to the current working directory unless an absolute path is provided.

Using the include statement has the effect of inserting the text of the included file into the current file. Include statements must precede all other statements in a file.

5. Internal Functions

5.1   Output

The built-in functions, print() and println(), print the passed parameter to standard output. print() is used to print without a new line and is useful for constructing error messages. println() appends a system-dependent new line to the end of the passed parameter.

Both print() and println() take exactly one parameter, which can be an expression that evaluates to any suitable type. It will print the value of the expression. When a Page object is passed as a parameter, the URL of the page will be printed.

5.2   Page Creation Functions

A new Page type is created using the createPage() function. In order to create a Page type, the URL of the web page must be specified in one of the following ways:

-        Page createPage(*<URL>*)
-        Page createPage(*<page>*)
-        Page createPage(*<host_URL>, <relative_URL>*)

<URL>, *<host_URL>*, and *<relative_URL>* are string arguments for the complete URL, the host, and the relative path, respectively. Page is a Page type whose URL is used to create the new Page. All of the above function calls return a Page type. In the special case that the specified URL is not found, the HTML document associated with the page is an empty document.

5.3   Page Manipulation Functions

Page Manipulation functions are applied to a Page type in order to customize the appearance of a web-page in our tabbed browsing environment.

Extract – *boolean extract(Page p, int extraction_constant1, int extraction_constant2, …, int extraction_constantn)*

The function extract() takes one or more extraction constant arguments that correspond to the different sections and tags of the HTML document that need to be removed. This function does not return any value.

The following identifiers are reserved constants that are used as arguments to the extract function.

- **IMAGES** – All images are to be extracted from the HTML document.

- **ADS** - All advertisements are to be extracted from the HTML document.

- **FLASH** - All flash animation is to be extracted from the HTML document.

- **SCRIPTS** - All scripts are to be extracted from the HTML document.

- **TXTLINKS** - All textual links are to be extracted from the HTML document.

- **IMGLINKS** - All images with links are to be extracted from the HTML document.

- **XTRNSTYLE** - All external stylesheets are to be ignored.

- **STYLES** - All style tags are to be extracted from the HTML document.

- **FORMS** - All forms are to be removed from the HTML document.

- **LINKLISTS** - All lists of consecutive links are to be removed from the HTML document.

- **EMPTYTBLS** – All empty tables are to be removed from the HTML document.

- **INPUT** - All input tags are to be removed from the HTML document.

- **META** - All meta tags are to be removed from the HTML document.

- **BUTTON** - All button tags are to be removed from the HTML document.

- **IFRAME** - All iframe tags are to be removed from the HTML document.

Append – *boolean append(Page p, string keyword)*

The function append() takes a string that specifies a keyword. If the keyword appears as part of a link on the web document (between the <a> and tags in the HTML), the URL that the link points to is fetched and its contents are appended to the end of the Page object and the function returns true. If the keyword is not found, the Page remains unchanged and the function returns false.

Title – *string title(Page p)*

The function title() takes a string argument and displays it as the title of the Page when opened in a tab.

Rank – *rank(Page p, int rank)*

The function rank() takes an integer that represents the priority assigned to the Page when opened in a tabbed browser. A larger integer corresponds to a higher priority. When the Page is added to a list and opened in the browser, its location will be determined by its rank relative to the other Page types in the collection. The display order is not defined when two pages have the same rank.

Status - *boolean status(Page p)*

The function status() returns a boolean value that indicates the status of the Page. A false return value indicates that the page cannot be displayed in the way that was specified due to a failure in either the URL fetching or content extraction process.

5.4   List Functions

Length – *int length(type[] list)*

The length() function takes a list type argument and returns the number of elements in the given list as an integer.

5.5   Output Functions

Show – *show(Page[] pageList)*

The show() function takes a list of Page types as an argument and displays each page as a tab in a built-in browser. This function uses the page values for rank and title to determine the position and the title of each tab, respectively.

Save Page – *boolean savePage(Page p, string filename)*

The save() function is applied to a Page type in order to save its HTML contents in a file for off-line browsing. This function takes a string argument indicating the name of the file and returns a boolean indicating success or failure.

6. Namespaces

The namespaces exposed to the user are divided into variables, functions, and types. This implies that variables can have the same names as types and functions.

Within the function namespace, names of functions can be reused as long as the type and order of the parameters can uniquely identify the function.

Files that are included in a source file may affect the namespace of the programmer's source file. As an example, assume a programmer has two files, toInclude.ceas and main.ceas where toInclude.ceas is included in main.ceas. The names of any functions declared in toInclude.ceas will become part of the function namespace of main.ceas. Likewise, any variables declared at the top level of toInclude.ceas will also exist in the top-level namespace of main.ceas.

7. Scoping Rules

The language is statically scoped. In general, the user will encounter three levels of scopes. The first and most important is the top level. All functions are defined at this level and are therefore available to any statement that follows the function declaration. Variables declared at the top level are also available to any statement that follows the declaration.

The second most common scoping level is the function-level scoping. All identifiers declared within a function are only available to that function. A function may shadow a top-level identifier.

Finally, the user can start a new scope any time they create a statement block with '{' and '}'. This often occurs in iterative loops or selection statements. As with functions, identifiers declared within curly brackets are not accessible outside of the brackets.

8. Example Program

The following is a sample program that can be written in the CEAS Language. The sample program is inefficient because it is intended to show the features of the language. An astute programmer will easily see ways to improve the performance of the code.

The first file contains functions that will be used to extract parts of documents. The file is then included in another file, which creates Page objects, calls the functions, and then displays the results in a tabbed window.

Contents of favoriteFilters.ceas:

```
/* Begin favoriteFilters.ceas */

/* First define a function used to remove flash
    Returns true upon success, false otherwise */

function boolean removeFlash(Page[] plist) {
    int i;
    int lastElement = length(plist) − 1;

    //Error checking
    if (lastElement < 0)
        return false;
    else {
        for (i = 0 : lastElement)
            plist[i].extract(FLASH);
    }

    return true;
}

/* This function removes advertising from a list of pages
    Returns true if it completes without error and false otherwise */
function boolean removeAdsAndScripts(Page[] plist) {
    int i = 0;
    int length = length(plist);
    if (length == 0)
        return false;

    while (i < length) {
        plist[i].extract(SCRIPTS);
        plist[i].extract(ADS);
        i = i + 1;
    }
    return true;
}

/* End of favoriteFilters.ceas file */
```

Contents of displayNews.ceas

```
/* Begin file displayNews.ceas */

include("favoriteFilters.ceas");

/* A helper function that calls both of the functions in
    the other file */
function boolean remFlashAdsScripts(Page[] plist) {
    if (removeFlash(plist))
        if (removeAdsAndScripts(plist))
            return true;
    return false;
}

/* Demonstrates different ways to produce page objects */
Page firstPage = createPage("http://www.cnn.com/");
Page secPage = createPage("http://www.msnbc.com/");

Page pages[4] = {firstPage, secPage,
                createPage("http://www.foxnews.com/"),
                createPage("http://www.nytimes.com/", "pages/world/index.html")};

// Calls a function that will attempt to remove flash, ads, and scripts
if (!remFlashAdsScripts(pages)) {
    println("Error detected while applying filters");
    return;
}

// Assigns rankings so that the last element in the last has the highest
// rating
int numPages = length(pages);
int j = numPages − 1;
do {
    pages[j].rank(numPages − j);
    j = j − 1;
} while (j >= 0);

show(pages);

/* End file displayNews.ceas */
```