Spidr

Language Reference Manual

Alex Dong aqd2000

Katherine Haas kah2190

Matt Meisinger mrm2205

Akshata Ramesh ar3120

Contents

1. Introduction

Spidr is a programming language that allows for users to sift through a large amount of HTML data. The purpose of our language is to retrieve and parse HTML pages, as well as traverse through them. Spidr allows for the use of following all links on a page, compiling all links from a page into a list, getting a list of the URLs of all the images on a page, getting a list of all dead links within a domain, and other HTML manipulations.

2. Lexical Conventions

2.1 Tokens

The following constitute the tokens in Spidr: identifiers, reserved keywords, constants, string literals, operators, newlines, and other separators. Blanks, spaces and horizontal and vertical tabs may be used to separate tokens. Otherwise they are ignored.

2.2 Comments

The characters `(*` introduce a comment, which terminates with the characters `*)`. Comments can be nested as such `(* (* *) *)`. They do not occur within a string or character literals. Any characters within these comments are ignored.

2.3 Identifiers

An identifier is any alpha-numeric sequence. The first character of an identifier must be a letter. Upper and lower case letters in an identifier are considered to be different. Identifiers may have any length.

2.4 Keywords

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

```
url         int         main
loop        string      false
selector    element     true
if          while       return
null        void        boolean
```

2.5 Constants

In Spidr there are integer constants, and string literals.

2.5.1 Integer Constants

An integer constant may contain any numbers from 0 to 9 and is stored as a signed integer.

2.5.3 String Literals

Anything between double quotes is considered a string literal. String literals may be concatenated using the '+' sign.

3. Identifiers

Each primitive, object and function is represented by an identifier.

4. Functions

4.1 Defining Functions

Functions are defined by using the function keyword. They follow the syntax:

```
function type functionName ([parameter list]) { expression }
```

Example:

```
function int addTwo(int num1, int num2) {
	return num1 + num2
}
```

Functions can’t be overloaded.

4.2 Return Types

Functions may return any type of object. They may also be marked as void, in which case no return statement is needed in the body of the function.

4.3 Main Function

Spider looks for a main function with a return type of `void` to use as a entry point when running an application. If this function is not found, an error is thrown at compile-time.

4.4 List-Valued Function Execution

In addition to the standard mode of function execution, a function may be executed as a list-valued function by surrounding one or more of its input parameters with angle braces (‘<’ and ‘>’). If one parameter is list-valued, the function will execute using each value in that list once as the parameter.

5. Lists

Primitives and objects may be declared either as a single-value variable, or a list variable.

5.1 List Types

A list may only contain elements of a single type. If an attempt is made to concatenate two lists of different types, an error will be thrown.

5.2 Instantiating Lists

An empty list may be instantiated by following the primitive or type identifier by `[]`. For instance, the following would instantiate an empty string:

```
string[]
```

Lists may also be instantiated with initial values by listing identifiers and/or constants, separated by commas, and surrounding them with square brackets (“[“ and “]”). The following illustrates how to instantiate arrays:

```
[“Value1”]
[exampleValue]
[“Value1” , “Value2” , “Value3”]
[“Value1” , exampleValue]
[15,42,54]
```

When instantiating a new list with initial values, all of the values must be of the same type.therwise a compile-time error is thrown.

5.3 Accessing Elements

Members of a list may be accessed by placing square brackets after the list identifier. For instance:

```
int[] values = [1, 4, 6, 7]
int singleValue = values[2]
(* singleValue is 6 *)
```

5.4 List Concatenation Operator

Lists may be concatenated using the & operator, resulting in a new list. The elements from the list on the right will be at the beginning of the resulting lists.

```
string[] newList = [“val1”, “val2”] & “val3” & [“val4”, “val5”]
```

6. Expressions

6.1 Pointer Conversions

Pointer conversion only occurs in the case of objects, arrays, and strings. The value of a declared object type is a pointer to the struct that exists in memory. The value of an array is a pointer to the first element of the array, and, as strings are represented as a list of chars in memory, the value of a string is a pointer to the first char in the string.

6.2 Primary Expressions

Primary expressions are identifiers, constants, strings, or expressions in parentheses.

primary-expression
identifier
constant
string
(expression)

An identifier is a primary expression that has type pointer, object, or value. An identifier is always an lvalue as its type is always a pointer. A constant is a primary expression. A string literal is a primary expression with type pointer to char, the address to the first character in the string array. An expression surrounded by parentheses is a primary expression identical to one without them.

6.3 Postfix Expressions

The operators in postfix expressions group left to right.
postfix-expression:

```
primary-expression
postfix-expression[expression]
postfix-expression++
postfix-expression--
```

argument-expression-list:

```
assignment-expression
assignment-expression-list , assignment-expression
```

All of these expressions behave as they do in C.

```
postfix-expression[<argument-expression-list>]
```

This notation maps the function on all elements in the argument list and returns a new list that contains the new elements. The given list must be in between the angled braces; otherwise the expression will assume that the function takes a list as argument.

```
string[] food = [“cake”, “apple”, “tiger”]
string[] capFood = toUpperCase(<food>) (*Assuming toUpperCase() exists *)
```

6.4 Array References

An array expression followed by an expression inside of square brackets denotes an array reference. The first element of the array is held at index 0, and the length of the array can be obtained using calling list.length.

```
string[ ] food = [“cake”, “apple”, “tiger”]
food[0] -> returns “cake”
food.length -> returns 3
```

6.5 Equality Expressions

The notation “`==`” compares whether the values of the adjoining expressions are equal. When more than two expressions are listed in succession, the comparison is made between all expressions. Even primitive types such as `string` and `url`, the comparison made will be made between their values rather than any underlying pointers. When applied to a list, the values at each index are compared. When this notation is applied to any object types created by the user, unless there is an equal method declared for it, the “`==`” is evaluated as “`.=`” (See below). When the types of the two expressions are not the same, false is returned:

```
string urlList = [“http://www.google.com”, “http://www.microsoft.com”]
string urlList2 = [“http://www.google.com”, “http://www.columbia.edu”]
string urlList3 = [“http://www.google.com”, “http://www.columbia.edu”]
boolean test1 = urlList[0] == urlList2[0]           (* Returns True *)
boolean test2 = urlList == urlList2                 (* Returns False *)
boolean test3 = urlList2 == urlLIst3                (* Returns True *)
boolean test4 = urlList == urlList2 == urlList3     (* Returns False *)
boolean test5 = urlList[0] == urlList               (* Returns False *)
```

To compare the base pointer values of expressions, the notation “`.=`” is required. This notation can be used to compare the pointer values of urls, strings, and any other types. When more than two expressions are listed in succession, the comparison is made between all expressions.

```
string urlList = [“http://www.google.com”, “http://www.microsoft.com”]
string urlList2 = [“http://www.google.com”, “http://www.columbia.edu”]
string urlList3 = [“http://www.google.com”, “http://www.columbia.edu”]
boolean test1 = urlList .= urlList2          (* Returns False *)
boolean test2 = urlList .= urlList3          (* Returns False *)
boolean test3 = urlList .= urlList           (* Returns True *)
```

The notation “!=” and “.!=” designate the negation of the values of the “==” and “.=” operator, respectively.

7. Declarations

To declare an identifier, the one of the following syntaxes must be used:

```
datatype identifier
datatype identifier = expression
datatype identifier = null
```

If the initial value expression is not provided as part of the declaration, the identifier is initialized with a null value.

The following *datatype* tokens are allowed:

- `int`
- `string`

- `url`
- `element`
- `selector`

The following are valid declarations:

```
int a
int b = null
int c = 0
int testList = [4, 2, 5, 6, 74, -4]
string e = [“first”, “second”]
url f =: “http://www.columbia.edu”
```

8. Statements

Except as indicated, assume that all statements are executed in sequence. Each statement must be terminated by a semicolon.

8.1 Expression Statement

Most statements will be expression statements.

8.2 Conditional Statement

The two forms of conditional statements are:

```
if (expression) statement
if (expression) statement else statement
```

In both cases the expression is evaluated, and it if it is non-zero, then the first statement will be executed. In the second case, the second statement will be executed if the first expression is equal to zero.

8.3 While Statement

The while statement takes the form of:

```
while (expression) statement
```

This statement can be executed repeatedly as long as the expression never takes the value of zero.

8.4 Loop Statement

The loop statement takes on the following form:

loop (expression$_1$, expression$_2$, expression$_3$) *statement*

The first expression specifies initialization for the loop. The second expression specifies a test, made before each iteration, where the loop will exit when the expression becomes 0. The third expression specifies an increment that is performed after each iteration.

8.5 Return Statement

The return statement is used when a function returns to its caller and it takes on the following forms:

```
return
return expression
```

In the first case the value is undefined, whereas in the second case the value of the expression is returned to the caller of the function.

8.6 Null Statement

The null statement takes on the following form:

```
null
```

You can include a label before a null statement (refer to 8 for an example). You can also label a null statement and insert it immediately before the item to get the same effect.

9. Scope

An object that is declared in a block has its scope restricted to that block and any sub-blocks. All functions are declared in the global scope.

10. Built-in Types

10.1 url type

The `url` type may be instantiated by placing a colon directly in front of a string literal. For instance:

```
url microsoftUrl = :”http://www.microsoft.com”
```

The same syntax can be used to convert a list of strings into a list of urls:

```
string[] sites = [“http://www.google.com”, “http://www.microsoft.com”]
url[] urlList = :sites
```

Appending the colon to the front of a parenthesized expression yields the same result as if there weren’t any parentheses:

```
url microsoftUrl = :”http://www.microsoft.com”
url micUrl = :(“http://www.microsoft.com”)
```

```
microsoftUrl == micUrl  (* returns true *)
```

10.2 element type

The `element` type represents a XML-type formatted string. It may have child elements. This type can be automatically cast into a string, or filtered by applying a `selector` to it.

10.3 selector type

A selector object is used to parse through an element tree and returns an array of either element or string objects that match the selection criteria. A selector is instantiated using the following syntax:

```
<element-selector [@ attribute-selector]>
```

A `selector` may be applied to any `element` object, `url` object, or list of either of these two types of object.

10.3.1 element-selector

This is a special selector that has its own set of token rules, separate from the rest of the language. This token may contain any combination of the following types of example token patterns:

`input` - All elements on the page of with a certain name can be selected by simply using that name. This example code returns all input controls on the page.

`div input` - If two selectors are separated by a space, it matches the first selector, then finds all of their children that match the second selector. In this example, the selector returns all inputs that are children of a `div`.

`.header-image` - A period prefixing a string indicates that all items matching that contain the class matching that string be returned.

`#example-input` - The pound sign indicates that all elements with the specified ID should be returned.

`[href]` - If a string is surrounded with square brackets, all elements that contain that attribute will be returned. In this case, all elements that contain the attribute `href` will be returned (though whether `href` has a value or not is not checked).

`[href="/images.htm"]` - An element name may be followed by an equals sign. Only elements with that attribute, and where the attribute matches the value in quotes will be returned. In this case, only elements that have an `href` element pointing to the images.htm page will be returned.

`[href="*images*"]` - In attribute selectors, the star may be used as a wildcard selector.

It matches any character(s). In this example, only elements that have an `href` attribute and the contain the word 'images' in this attribute will be returned.

`[href="{var1}"]` - Spidr variables may be included in the selection strings by adding them in curly braces. The example above indicates that all elements should be returned which have an `href` equal to the current value of `var1`.

Here is an example of how an element selector can be used to gather a list of all input html elements that exist under a
with the ID of "survey":

```
url testUrl = :"http://www.columbia.edu/"
element[] inputFields = testUrl <div#survey input>
```

10.3.2 attribute-selector (optional)

This selector is optional, and indicates whether an attribute should be read from each of the elements selected and returned. An 'at' sign (`@`) must precede the attribute selector. The selector may be the name of an attribute, or an underscore to return the contents of the attribute. For instance, the following example shows how to retrieve a string array of all hrefs from all anchors on a page:

```
url testUrl = :"http://www.columbia.edu/"
string[] links = testUrl <a @ href>
```

11. Built-in Functions

11.1 print() function

The print function converts any object to a string and displays it in the console.

For instance:

```
print(55)
```

Output:

```
55
```

If the type is a list, it uses the notation "`[ "element1", "element2", "element3" ]`" to to show the differing elements in the list.

If the object being printed has sub-lists of objects underneath it, it will print out all child objects also, up to a depth of 5. After 5, it will show all child lists as "`[ … ]`".

Example code:

```
string[][] example = [ [ 1, 2 ] , [ 30, 40 ] ]
print(example)
```

Output:

```
[ [ 1, 2 ] , [ 30, 40 ] ]
```

12. Example Code

12.1 Hello World

The following Spidr code prints `Hello world!` to the console.

```
function main() {
        print(“Hello world!”)    }
```

12.2 Full Example

The following example queries a url and returns a list of all fully qualified outgoing hyperlinks on that page, and follows the links recursively to a depth of three levels.

```
function void main() {
        url sourceUrl = :”http://www.columbia.edu”
        url[] descendants = getChildPages(sourceUrl, 3)
        loop descendants item {
                print(item)
        }
}

function url[] getChildPages(url startingPage, int depth) {
        if (depth == 0) {
                return startingPage
        }
        else {
                url[] urlsOnPage = :(startingPage <a @ href>)
                url[] descendantUrls = getChildPages(<urlsOnPage>, depth - 1)
                return startingPage ++ descendantUrls
        }
}
```