Columbia's Awk Replacement Language (CARL)

COMS 4115 Programming Languages and Translators (Spring 2017)

Darren Hakimi (dh2834)	System Architect
Keir Lauritzen (kcl2143)	Manager
Leon Song (ls3233)	Tester
Guy Yardeni (gy2241)	Language Guru

Description of Language

Columbia's Awk Replacement Language (CARL) is a record-processing, string-parsing language with syntax based on GNU Awk (Gawk) with a subset of the language features.[1] GNU Awk is an extension of the original Awk language at Bell Labs by A. Aho, P. Weinberger, and B.Kernighan (hence, AWK). Awk is an interpreted language by CARL will be compiled.

CARL programs follows the same structure as Awk:

```
pattern1 {action1}
pattern2 {action2}
```

The language reads each record (nominally each line but defined in `RS`) from the input file, checks whether it matches the pattern and if it does it takes the listed action. The pattern matching is supported using regular expressions. Each record is subsequently divided into fields and the actions can be applied to each field (nominally delimited by spaces but defined using `FS`). All records are processed against each pattern in order (for example, all records will be processed against `pattern1` then all records will be processed against `pattern2`. This behavior matches Awk's behavior as an interpreted language.

Two special-purposed patterns are BEGIN and END. All actions in a BEGIN pattern are executed at the beginning of the program before any records are processed and they are executed only once. Actions with the END pattern are the same only executing at the end of the records. The actions specified inside { } between the BEGIN and END patterns will recur per field.

The actions support the standard set of control statements, variable assignments, numerical operations, and user-defined functions. The syntax is similar to C but does not support pointers or dynamic memory (outside of the built in datatypes of string and associative array). The actions can operate on the entire record using the $0 field designator or on positional field indicators $1, $2,... Field designations are do not allow manipulation within the field, so CARL provides a set of builtin functions to manipulate records.

[1] This project is based on GNU Awk and the language documentation at https://www.gnu.org/software/gawk/manual/gawk.html was used extensively in preparation of this proposal.

Intended Uses

CARL (like Awk) is intended to be used to parse and manipulate records in text files. Common uses for complex CARL programs are performing calculations on text based data sources, such as CSV files. With CARL it is possible to calculate statistics or perform bookkeeping on complex data tables. A simple CARL usage would be to parse a log file for a specific event.

We intend to extend CARL from Awk to provide additional capability for semi-structured data. Examples of semi-structured data include XML and HTML files, where there is computer-readable information but it varies in length and form. CARL has a wide array of uses in semi-structured files. CARL can parse through files and extract portions of the file data based on tags. CARL could be used for web scraping because as it can iterate through the HTML and identify patterns in the tags. The data extracted, such as email addresses and phone numbers, can be formatted into a table for later use. An example extension identified so far is the inclusion of an `MF` reserved variable, which stores the field that matches the regular expression.

Parts of the Language

CARL, like its predecessor AWK, follows a `pattern {action}` syntax.

Patterns:

`BEGIN`	`BEGIN` is a special keyword indicating that the action is to be run before any records are processed.
`END`	`END` is a special keyword indicating that the action is to be run after all records in the file have been processed.
Expressions	Expressions can be used if they evaluate to non-zero or not null.
Character classes	Character classes are shorthands to describe sets of characters that have a specific attribute, and can only be used inside bracket expressions. They are denoted by "[:" followed by the keyword for that character class, then ":] E.g. `[:alnum:]` matches all alphanumeric characters, a convenient way to avoid typing "`/[A-Za-z0-9]/`" Character classes and their keywords are defined by the POSIX standard

Regular Expressions:

CARL will use regular expressions heavily to parse through semi-structured files.

Regex Definition and Usage	Like awk, you can define a regex by enclosing it in slashes. e.g. `/abc/`, which will attempt to iterate through every record and

	perform some action if the string “abc” is found in that record.
Dynamic Regex	CARL will allow for any expression to be used as a regex. That is, given an expression, CARL converts it to a string then uses it as a regex
Ambiguity	In cases of ambiguity, CARL takes the leftmost longest match For example, `\a*\` for the string “aaaabbaa” matches with “aaaa” instead of “aa”.

Regex Operators:

`\`	Backslash is used to suppress any special meaning associated with the character e.g. “ `\”hello\”` “ will search for the string “hello”, inclusive of quotation marks
`^`	Forces the program to look for matches at the beginning of the string for example `/^foo/` will match “football” but not “afoot”
`$`	Similar to `^`, except the program looks for matches at the end of the string instead
`.`	This matches any **single** character. For example, `/.a/` matches “ba” and even the whitespace character “ a”, but not “ab”
`[ x ]`	Called the bracket expression. Matches any single character enclosed within the square brackets, for example `[0-9]` searches for any digits in the string
`[ ^x ]`	Complement of the bracket expression. Matches any single character **not** within the square brackets e.g. `[^0-9]` will match any character that is not a digit
`*`	Repeats the preceding regular expression as many times as necessary to find a match. For example, `\ab*\` matches “ab”, “abbbbb” and “a” Whereas `\(ab*)\` matches “ab”, “abababab”, **and the empty string**

Actions:

Awk is a Turing-complete language with support for different data types and structures, control flow, I/O, and user-defined functions. CARL will implement most of the language, but will exclude many of the built-in variables and some, less common, aspects of the language.

Data types and structures

Integers	Supports a single type of integer.

Floating point	Supports a single type of floating-point number.
String	Standard string data type.
Associative arrays	One-dimensional associative array. There is language support for for-loops through the array and adding and deleting elements. Indices of arrays can be either numbers or strings.

There is automatic type conversion between integers and floating point operators, with escalation from integers to floating-point as needed (irreversible).

Control flow

`If` statement	C-style if-then-else controls. `if (expr) { body } else {body}`
`For` statement	C-style for loops. Python-style loops `(element in array)` for associative arrays.
`Break` statement	C-style breaks to exit loops.
`Return` statement	Return from user-defined functions.
User-defined functions	The ability to implement functions

CARL enforces the need for {} around the body of control switches. Only for loops are supported. While and do-while are not supported.

Reserved variables:

`RS`	Variable that stores the literal for separating the input file into multiple records.
`FS`	Variable that stores the literal for separating a record into multiple fields.
`NR`	Number of records processed.
`NF`	Number of fields in a record.
`RSTART`	Index value that matches the beginning of a regex expression.
`RLENGTH`	Length of the matching regex expression.
`MF`	Matching value of the regex pattern to eliminate more complex expressions.
`$0`	Variable storing the entire record.

`$1, $2, ...`	Positional variables for the parsed fields.

Builtin Functions

`match(`	Given a *string*, return the character index of the longest, leftmost substring matched by *regex*. Return 0 otherwise (note that we return 1 if the match occurs at the beginning of the string).
`substr(`	Looks at *string* and returns a *length* character-long substring beginning from *start*. For example, `substr("hello", 2, 3)` returns "`llo`".
`sub( _`	Searches the *target* string for the given *regex*, and replaces it with *new*. We always find the longest, leftmost substring and replace only once. For example, `str = "columbia college"` `sub("col", "example", str)` will set `str` to be "`exampleumbia college`"
`gsub( )`	Like `sub()`, except we replace every occurrence in *target*. Note that the *target* argument is optional, in which case we search the entire input record. Return the number of substitutions made.
`length( )`	Returns the number of characters in .

`split(`	Splits the given into pieces using , and stores each piece in .
	For example, `str = "example@email@fake.com"`
	`split(str, array, "@")` sets the contents of *array* to be:
	`array[0] = "example"`
	`array[1] = "email"`
	`array[2] = "fake.com"`
	
	remains unchanged. `split()` returns the number of elements created, so in this case `split() = 3`.

Example programs

Email parsing example:

```
carl '$0, /.*@.*\..*/ { print MF}'
```
replaces the longer
```
awk 'match($0, /.*@.*\..*/) { print substr( $0, RSTART, RLENGTH )}'
```

Phone number parsing example:

CARL example:
```
carl'$0, /\([0-9]{3}\)[[:space:]][0-9]{3}\-[0-9]{4}/ { print MF}' filename
```
replaces the longer awk version:
```
awk 'match($0, /\([0-9]{3}\)[[:space:]][0-9]{3}\-[0-9]{4}/) { print substr( $0,
RSTART, RLENGTH )}'
```

Text database parsing example:

Input file:

```
1.    (212)-123-1011     $120,000    Andrew        M
2.    (222)-298-8494     $50,000     Bob           M
3.    (114)-124-1234     $170,000    Katy          F
4.    (123)-222-1111     $90,000     Lisa          F
5.    (123)-132-4666     $80,000     Billy         M
6.    (222)-115-1515     $130,000    Alexa         F
7.    (718)-123-4556     $242,000    Mia           F
```

Code:

```
BEGIN {
      total=0;
       highEarners=0;
      lowEarners=0;
      females=0;
      males=0;
}
{
      gsub(/\$|,/,"",$3);
      gsub(/\(|\)|-/,"",$2);
      personID=$1;
      phoneNum=$2;
      salary=$3;
      name=$4;
      gender=$5;
      total=total+salary;
      if (salary>=100000) {
            highEarners=highEarners+1;
      } else {
            lowEarners=lowEarners+1;
      }
      if (gender=="F") {
            females=females+1;
      } else if (gender=="M") {
            males=males+1;
      }
      areaCode = substr(phoneNum, 0, 3);
      if (dict[areaCode]) {
            dict[areaCode] = dict[areaCode]+1;
      } else {
            dict[areaCode] = 1;
      }
}
END   {
		print "Total Amount: $"total;
	average=total/NR;
	print "Average Amount: $"average;
	print "High Earners: "highEarners;
	print "Low Earners: "lowEarners;
	print "Male to Female Ratio: "males":"females;
	for (i in dict) {
		print "Area code: "i" has "dict[i]" people.";
	}
}
```

Output (by running by “carl -f [code-file] [data-file]”:

```
Total Amount: $882000
Average Amount: $126000
High Earners: 4
Low Earners: 3
Male to Female Ratio: 3:4
Area code: 222 has 2 people.
Area code: 114 has 1 people.
Area code: 212 has 1 people.
Area code: 718 has 1 people.
Area code: 123 has 2 people.
```