HAWK Language Reference Manual

HTML is All We Know

Created By: Graham Gobieski, George Yu, Ethan Benjamin, Justin Chang, Jon Adelson

0. Contents

1. Introduction

HAWK is a programming language designed for web-scraping. The language takes its inspiration from AWK, using the same (pattern, action) model to manipulate HTML as opposed to text documents. The HAWK compiler compiles code to Java code, which is compiled to Java Bytecode, and then run on a Java Virtual Machine.

2. Lexical Conventions

2.1 Tokens

There are several tokens in HAWK: identifiers, keywords, constants, string literals, operators, and patterns. Whitespace and comments are ignored except as separators of tokens.

2.2 Comments

Use / *to begin a comment, and terminate it with* /. Comments can extend across multiple lines.

2.3 Identifiers

An identifier is a sequence of letters, digits, dashes, and underscores. Uppercase and lowercase letters are different. Identifiers may have any length, and are separated from other tokens by whitespace. An identifier must have at least one letter or underscore, and must begin with either of the two.

2.4 Keywords

The following identifiers are keywords reserved for particular use:

- begin
- double
- else
- end
- if
- int
- return
- string
- this
- while

2.5 Constants

There are four types of constants, described in detail below.

2.5.A Integer Constants

An integer constant consists of a sequence of digits that does not begin with 0. All integers in HAWK are taken to base 10. Negative integer constants consist of a sequence of digits prefixed by a dash.

2.5.B Double Constants

A double constant consists of an integer part, decimal point, fractional part, and optional exponential part, which consists of an integer prefixed by an e. Either the integer or fractional part, but not both, may be missing. Both integer and fractional parts are themselves integers, separated by the decimal point. The decimal point must be present.

2.5.C String Literals

A string literal is a sequence of characters surround by double quotes. HAWK contains several escape sequences which can be used as characters within string literals:

- newline “\n”
- tab “\t”
- backslash “\”
- single quote “\’”
- double quote “\””

String literals are immutable, and thus cannot be altered. Any operations performed a string literal will not affect the original literal but instead generate a new string.

2.5.D Table Literals

A table literal is a comma-separated sequence delimited by curly braces. The comma-separated sequence can take two forms.

- A sequence of values of any type — there can be different types in this sequence. This generates a table where the values are keyed in sequential order from the integer 0 to sequence length-1.
- A sequence of key-value pairs written in the form key : val. As before, values can vary within a table. Keys are restricted to integer or string types, and a table can be keyed by either integers or by strings, but not both.

2.6 Patterns

Patterns utilize valid Regex or CSS selector syntax and are defined with special offset characters within brackets before an action block in the pattern section of the program. Please see the following sections for a discussion on what a comprises a valid CSS selector and Regex expression.

2.6.A CSS Selector Patterns

HAWK implements a limited syntax of the standard W3 definition of CSS selectors. A CSS selector contains special syntax used to find elements in an HTML document which match particular criteria criteria. In HAWK, a CSS selectors patterns consists of CSS selector enclosed by @ symbols which are themselves enclosed by brackets.

```
[@ css-selector @]{
    /*action*/
}
```

2.6.A.1 CSS Selectors

A CSS selector consists of one ore more simple selector sequences chained by combinators. See below for details on simple selector sequences and combinators.

css-selector:

- *simple-selector-sequence*
- *simple-selector-sequence combinator css-selector*

2.6.A.2 Simple Selector Sequences

A simple selector sequence consists of a single type selector, or an optional type selector followed by one or more property selectors. These are the essential building blocks of CSS selectors.

Intuitively, a type selector is used to find HTML elements with specific tags (e.g. `<div>` or `<td>` or `<span>`) while property selectors are predicates on attributes and associated values within a tag (e.g. blah1, blah2, "something1", and "something2" in `<td blah1="something" blah2="somethingelse">`).

simple-selector-sequence:

- *type-selector*
- *type-selector property-selector-list*
- *property-selector-list*

2.6.A.3 Type Selectors

A type selector can either specify a specific tag type, or can select any tag using the universal selector. We allow matching on non-standard tag types named by any valid identifier, even those not typically supported in web browsers.

type-selector:

- *identifier*
- *universal-selector*

universal-selector:

- *

2.6.A.4 Property Selectors

Property selectors are predicates on attributes and associated values within an HTML tag.

property-selector:

- *attribute-exists-selector*
- *attribute-string-selector*
- *class-selector*
- *id-selector*

Attribute existence selectors check if a tag contains a given attribute.

attribute-exists-selector:

- [identifier]

Attribute string matching selectors check if a tag contains a given attribute, and additionally filters for attributes whose associated values have certain string properties. Most of these are self explanatory with the exception of the whitespace separated containment selector. This selects for attribute values which, when split into a list of words by whitespace, contain the chosen word (e.g. `[years ~= "1992"]` would match the element `<div years = "1488 1875 1992 1995"/>`.).

attribute-string-selector:

- *attribute-equals-selector*
- *attribute-contains-selector*
- *attribute-beginswith-selector*
- *attribute-endswith-selector*
- *attribute-whitespace-separated-contains*

attribute-equals-selector:

- [identifier = string]

attribute-contains-selector:

- [identifier *= string]

attribute-beginswith-selector:

- [identifier ^= string]

attribute-endswith-selector:

- [identifier $= string]

attribute-whitespace-separated-contains:

- [identifier ~= string]

class-selector:

- .identifier

Class selectors are used to find HTML elements whose "class" attribute contains a given value.
. id-selector*:

- #identifier

ID Selectors are used to find HTML elements whose "id" attributes equal a given value.er

2.6.A.5 Combinators

Combinators chain together multiple simple sequence selectors, and are used to find elements who have a certain hierarchical relation to other elements. For instance, the direct child combinator > can be used to find an element that is a direct child of another element in the HTML XML hierarchy (e.g. `@div[attr1=value1] > span@` will find the span in `<div attr1="value1"> Blah blah blah <span></span></div>`).

combinator:

- *direct-child-combinator*
- *descendent-combinator*
- *direct-sibling-combinator*
- *any-sibling-combinator*

direct-child-combinator:

- >

The direct child combinator finds elements matched by a simple selector sequence that are immediate children of elements matched by another simple selector sequence.

descendent-combinator:

- (empty string)

The descendent combinator finds elements matched by a simple selector sequence that are descendents of elements matched by another simple selector sequence. For instance, `@div span@` will match the nested span in `<div><span></span/></div>`.

direct-sibling-combinator:

- +

The direct sibling combinator finds elements matched by a simple selector sequence that are the next sibling of elements matched by another simple selector sequence. For instance, `@tr + span@` will match the span in `<div><tr></tr><span><span/></div>`.

2.6.A.6 Examples

- * : selects all elements
- *#id* : selects all elements with an id attribute that matches the provided string
    - **Example:** `@#first-name@` will select all elements with attribute `id="first-name"`
- *.class* : selects all elements with a class attribute that matches the provided string
    - **Example**: `@.first-name@` will select all elements with attribute `class="first-name"`
- *element* : selects all elements that have the provided tag name. Please see the HTML language reference manual for a complete list of valid HTML element tags.
    - **Example**: `@p@` will select for all paragraph elements
- *element1 element2* : law of the descendent. This pattern will select for all element2's that are child elements of element1
    - **Example**: `@div #first-name@` will select for all elements with id attribute, `id="first-name"` that are children of div elements
- *element1* > *element2* : strict law of the descendent. This pattern will select for the elements that are direction children of the parent element. In other words, these children cannot be grandchildren (elements nested within other elements)
- *element1* + *element2* : selects the immediate child (the one child that is directly below the parent), element2, of the parent element, element1.
- *element1* ~ *element2* : selects every element, element2, that is preceded by element1
- *[attribute]* : selects all elements with the given attribute. Example: [title] will select all

elements with a title attribute

- *[attribute op value]* : selects all elements that have an attribute with a value that evaluates the expression to true.
    - *[attribute = value]* : selects elements that have attribute value equal to provided value.
    - *[attribute ~= value]* : selects elements that have an attribute value that contains provided value.
    - *[attribute |= value]* : selects elements that have an attribute value that begins with provided value.
    - *[attribute $= value]* : selects elements that have an attribute value that ends with provided value.
    - *[attribute *= value]* : selects elements that have an attribute value that contains provided value.

Please see CSS and HTML language reference manuals for additional explanation of each selector pattern.

2.6.B Regex Patterns

HAWK implements a limited version of the standard regex expression syntax of the AWK language. A regex expression is used to find a particular pattern in an text document using the operations defined below (please see the POSIX and AWK language reference manuals for more). Moreover, a regex expression pattern must be offset with / symbols on both sides and be contained within a bracket before an action section.

```
[/.../]{
    /*action*/
}
```

Regex expression operations may be combined and standard regex expression operator precedence will be assumed (see POSIX reference manual and note the order of the operators below) or order may be defined using parentheses. Below are a description of the operators implemented:

- *'c'* : represents a single character.
- _ : represents any character.
- *eof* : represents the end of file character.
- *"string"* : represents a literal string of characters.
- *'a' - 'z'* : represents a range of characters. Must be contained within a pair of brackets.
- *['b' 'c']* : evaluates to true if current character matches any character provided within brackets.
- ^ : compliments a set of characters.

- *(pattern)* : evaluates pattern inside parentheses before patterns outsides. Parentheses suggest an order of evaluation.
- *pattern** : represents the kleene closure of a pattern with zero or more of the pattern present.
- *pattern+* : represents the kleene closure of a pattern with one or more of the pattern present.
- *pattern?* : represents a pattern that is optional.
- *pattern1 pattern2* : represents a pattern followed by a pattern.
- *pattern1* | *pattern2* : represents either pattern1 or pattern2

Please see the POSIX and AWK language reference manuals for additional explanation of each regex expression operator.

3. Syntax Notation

3.1 Meaning of Identifiers/Variables

Identifiers are names which can refer to functions, variables, and table fields. Each identifier is a string consisting of digits, letters, and underscores which does not begin with a digit.

Variables are storage locations that contain values. Depending on where in a program variables are initialized, they are either global or local to a particular scope. See Storage Scope for more details.

HAWK is statically typed, which means that every variable has a type. The type of a variable determines the meaning and behavior of its values, and also the nature of storage needed for those values.

3.2 Storage Scope

The visibility of an identifier and liftetime of a variable's storage depends on where a variable is initialized. If a variable is initialized within a *BEGIN* or *END* block, it is a global variable. A global variable can be accessed by any part of the program below the global variable's initialization. It's storage stays alive throughout the entire execution of the program.

If a variable is initialized within any block other than a *BEGIN* or *END* block, it is a local variable. A local variable can be accessed within the scope it is initialized, at or below its initialization. Its storage will be destroyed at the end of the scope.

3.3 Basic Types

There are four basic types in HAWK:

- integer
- double
- string

- table

Integers are 32-bit signed two's complement integer. Doubles are double-precision 64-bit IEEE 754 floating point. We will refer to doubles and integers arithmetic type.

Strings are a sequence of 0 or more unicode characters. They are guaranteed to occupy O(n) space in memory, where n is the number of characters in the string.

Both arithmetic types and strings are immutable, which means that their value cannot be changed once they are created. When a variable with an immutable type is assigned a new value, the old value and underlying storage are destroyed.

Tables, unlike the immutable types, are objects and are mutable. This means that variables do not contain tables, but rather contain references to tables. Assigning a table to a variable results in that variable storing a reference to that table. Similarly, when tables are passed as parameters to functions or returned from functions, the respective parameters and return values are references. In each of these operations, there is no copying of internal table data, only copying of references.

HAWK uses reference counting to keep track of how many variables store references to the same table. When a table no longer has any variables referencing it, the underlying storage for the table is destroyed.

3.4 Automatic Conversions

3.4.A Promotion of Integers in Mixed Arithmetic Expressions

In mathematical binary expressions where one operand is an integer and the other operand is a double, the integer will be automatically converted to a double value. The conversion will be performed using the the built-in function int_to_double.

3.4.B String Conversion in String Concatentation Expressions

In binary addition expressions where one operand is a string, and the other operand is not a string, the non-string will be automatically converted to a string value. Tables will be converted using table_to_string, integers using int_to_string, doubles using double_to_string.

4. Expressions

The precedence of expression operations is the same as the order of the major subjections of this section. Within each subsection, operators have the same precedence. Left or right associativity will be specified in each of the subsections for each operator.

4.1 Primary Expressions

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

primary expressions:

- *identifier*
- *constant*
- *string*
- *(expression)*

4.2 Postfix Expressions

Operators in postfix expressions group left to right.

postfix-expression:

- *postfix-expression[expression]*
- *postfix-expression(argument-expression-list)* //arglist is optional

4.2.A Table References

A postfix expression followed by an expression in square brackets is a postfix expression denoting a subscripted table reference. The expression in square brackets must be a table key of type int or string, the postfix expression must be a table. The whole expression is of type table, int, string, or double (the value of the value associated with the key).

4.2.B Function Calls

A function call is a postfix expression (function designator) followed by parentheses containing a possibly empty, comma-separated list of assignment expressions which constitute the arguments to the function. A declaration for the function must previously exist in scope. Recursive functions are permitted.

The term *argument* refers to an expression passed by a function call, the term *parameter* refers to an input object or its identifier received by the function definition.

4.3 Unary Operators

Expressions with unary operators group right to left.

unary expression:

- *unary-operator cast-expression*

unary-operator: -

4.3.A Unary Minus Operator

The operand of the unary - operator must be of type int or double, and the result is the negative of its operand. An integral operand undergoes integral promotion. The type of the result is the

type of the promoted operand.

4.4 Multiplicative Operators

The multiplicative operators *, /, and % group left-to-right.

multiplicative-expression:

- *multiplicative-expression* * *cast-expression*
- *multiplicative-expression* / *cast-expression*
- *multiplicative-expression* % *cast-expression*

The operands of *, /, and % must be of type integer or double. The usual arithmetic conversions are performed on the operands.

The binary * operator denotes multiplication.

The binary / operator yields the quotient, and the % operator the remainder, of the division of the first operand by the second; if the second operand is 0, the result is undefined. Otherwise, it is always true that (a/b)*b + a%b is equal to a. If both operands are non-negative, then the remainder is non-negative and smaller than the divisor, if not, it is guaranteed only that the absolute value of the remainder is smaller than the absolute value of the divisor.

4.5 Additive Operators

The additive operators + and - group left-to-right. If the operands have type int or double, the usual arithmetic conversions are performed.

additive-expression:

- *multiplicative-expression*
- *additive-expression* + *multiplicative-expression*
- *additive-expression* - *multiplicative-expression*

The result of the + operator is the sum of the operands. For strings, the sum is defined as the concatenation of the two strings.

4.6 Relational Operators

The relational operators group left-to-right. a<b<c is parsed as(a<b)<c, and evaluates to either 0 or 1.

relational-expression:

- *shift-expression*
- *relational-expression* < *shift-expression*
- *relational-expression* > *shift-expression*

- *relational-expression* <= *shift-expression*
- *relational-expression* >= *shift-expression*

The operators < (less), > (greater), <= (less or equal) and >= (greater or equal) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is int. The usual arithmetic conversions are performed on arithmetic operands.

4.7 Equality Operators

equality-expression:

- *relational-expression*
- *equality-expression* == *relational-expression*
- *equality-expression* != *relational-expression*

The == (equal to) and the != (not equal to) operators are analogous to the relational operators except for their lower precedence. (Thus a<b == c<d is 1 whenever a<b and c<d have the same truth-value.)

4.8 Logical AND Operators

logical-AND-expression:

- *inclusive-OR-expression*
- *logical-AND-expression* && *inclusive-OR-expression*

The && operator groups left-to-right. It returns 1 if both its operands compare unequal to zero, 0 otherwise.

&& guarantees left-to-right evaluation: the first operand is evaluated, including all side effects; if it is equal to 0, the value of the expression is 0. Otherwise, the right operand is evaluated, and if it is equal to 0, the expression's value is 0, otherwise 1.

The operands must be of type int or double, but don't have to be of the same type. The result is int.

4.9 Logical OR Operators

logical-OR-expression:

- *logical-AND-expression*
- *logical-OR-expression* || *logical-AND-expression*

The || operator groups left-to-right. It returns 1 if either of its operands compare unequal to zero, and 0 otherwise.

|| guarantees left-to-right evaluation: the first operand is evaluated, including all side effects; if it

is unequal to 0, the value of the expression is 1. Otherwise, the right operand is evaluated, and if it is unequal to 0, the expression's value is 1, otherwise 0.

The operands must be of type int or double, but don't have to be of the same type. The result is int.

4.10 Constant Expressions

A constant expression is an expression that is just a constant. The type of the constant expression is the type of the constant, the value of the constant expression is the value of the constant.

4.11 Built-In Functions

HAWK includes several built-in functions that are reserved and are specially-interpreted by the compiler. These include:

- `open("...")` : must be placed before the BEGIN section of the program and sets the context of the program. In other words, the function determines which file the pattern-actions will operate on. Files are automatically closed at the end of program execution.
- `print("...")` : prints a strings to standard output.
- `exists(table[i])` : checks to see if a table element exists. Returns 1 if so, otherwise 0.
- `length(table)` : returns the length of a table
- `keys(table)` : returns a table containing every key of the passed table. The returned table uses the index as a the key and the return key value as the value.
- `children()` : takes the `this` data structure and returns a table populated with the children of the found element. Only defined in CSS selector patterns.
- `inner_html()`: returns the inner_html corresponding to the element defined by `this`. Only defined in CSS selector patterns.

5. Declarators

Declarations give a specific meaning to each identifier.

declaration:

- declarator

declarator:

- identifier = expression
- identifier(arg-list) compound-statement

For identifier = expression, the type of identifier is the result of the expression on the RHS. If this identifier is later used in an expression, it yields a type that is the same as when it was created.

You must initialize a variable when you create it.

5.1 Function Declarators

identifier(arg1, arg2, ..., argn) compound-statement

The type of identifier(arg1, arg2, ..., argn) is the type of the expression in the return statement. The types of arg1, arg2, ..., argn are inferred within the compound-statement. The names of arg1, arg2, ..., argn must be valid HAWK identifiers.

All arguments to a function are call-by-value. When tables are passed as arguments, the reference to the table is passed by value. See the Syntax Section for more information.

6. Statements

Statements are executed in sequence.

Statement:

- expression-statement
- assignment-statement
- compound-statement
- conditional-statement
- while-statement
- return-statement
- pattern-statement
- BEGIN statement
- END statement

6.1 Expression Statements

expression-statement:

- expression

Expression statements will typically be a function call but can be any arbitrary expression.

6.2 Assignment Statement

assignment-statement:

- *identifier = expression*

Assignment statements are used to declare a variable or they can be used to update the value of a variable that already exists.

6.3 Compound Statements

compound-statement:

- *statement-list*

statement-list:

- *{statement}*
- *{statement; statement-list}*

Compound statements are used to write several statements when one statement is expected.

6.4 Conditional Statement

conditional-statement:

- *if (expression) compound-statement*
- *if (expression) compound-statement else compound-statement*

The first substatement is executed if the expression is not equal to zero, otherwise the second statement is executed. The second statement can either be another Conditional Statement or a Compound Statement.

6.5 while Statement

while-statement:

- *while (expression) compound-statement*

The compound-statement is executed until the expression is not equal to 0.

6.6 return Statement

return-statement:

- *return expression*

Return statements are only used within function bodies.

6.7 Pattern Statement

pattern-statement:

- *pattern-expression compound-statement*

Pattern is a pattern to match against and the compound-statement is executed each time that

the patern is matched. Refer to Expressions and Program Structure sections for additional information.

6.8 BEGIN Statement

BEGIN compound-statement

Refer to Expressions section for additional information.

6.9 END Statement

END compound-statement

Refer to Program Structure section for additional information.

7. Program Structure

7.1 General Structure

All programs must contain three sections: a begin section, a pattern section, and an end section. Programs must contain these in the order given and may not have more than one begin or end section. Following are a syntactic outline of such a structure and descriptions of each section.

```
BEGIN{
    /*action*/
}
[/*pattern*/]{
    /*action*/
}
END{
    /*action*/
}
```

7.1.A Begin Section

This section is executed first. Functions and variables may be declared in this section and defined and made available to other sections. In other words, variables and functions defined in this section are visible in the pattern and end sections. The section may be empty and all normal syntax previously defined should be followed.

7.1.B Pattern Section

This section is executed in the order and has one or more pattern blocks that begin with a CSS or Regex expression in brackets and are followed by an optional-empty action. The patterns

must align with the syntax provided in the following pattern sections. The action will be executed after each instance of such pattern has been found in the given file. The action block should contain normal syntax previously defined. Functions may not be defined in this section nor can patterns be nested. Relevant information returned by the pattern match can be accessed using the `this` keyword data structure. See below for further information on access to this structure.

***this*:**

A table of relevant information returned by the pattern. In the case of a regex expression this is simply a table containing a single element of type string. For a CSS selector, on the other hand, this table is populated with the following keys and values:

- `this["id"]`: return id of the found element
- `this["class"]`: return the class(es) of the found element
- `this["handle"]`: return a handle that references the found element. This is read-only.
- `this[/*custom attribute*/]`: return a custom attribute of the found element. This may not exist.

7.1.C End Section

This section is executed last after all pattern sections have been executed, this section may have include new functions and variables as well as references to previously defined (in begin or pattern sections) functions and variables. New functions and variable only are visible in the end section and normal syntax previously defined should be implemented.

8. Sample Programs

8.1 Sample 1: General Syntax

Creates a table, adds elements to this table each time a new line is encountered and prints this table. Notice that tables are mutable and how values of a table do not have to be of the same type.

```
BEGIN{
    table = {1, 2, "hello", 70, 90, 100, 100.1}
}
[ ]{
    table[length(table)] = table[length(table)-1]+10;
}
END{
    print(table);
}
```

8.2 Sample 2: CSS Selectors

Looks for a certain CSS pattern and populates a global table with a count derived from the pattern. Then it prints this table.

```
BEGIN{
    counts = { }
}
[@ table . wikitable > tr @]{
    state = this["title"];
    if(exists(counts[state])){
        counts[state] = counts[state]+1;
    }else{
        counts[state] = 0;
    }
}
END{
    len = length(counts) - 1;
    while(len >= 0){
        print(”STATE: ” + state + ”, COUNT: ” + counts[state]);
        len = len - 1;
    }
}
```

8.3 Sample 3: Regex

Looks for a regex expression, converts the output of the pattern to an integer and prints the integer if it satisifies a condition.

```
BEGIN{ }
[/ ['0'-'9']+ ft /]{
    height = int_of_string(this);
    if( height >= 14000 && height <=15000){
        print("found");
    }
}
END{ }
```

9. References

The following are helpful references that may have been previously refered to in above sections.

9.1 AWK Language Reference Manual

HAWK was inspired by AWK and, as such, the AWK LRM is a good way to start to understand the structure and syntax of HAWK programs.

http://www.gnu.org/software/gawk/manual/gawk.html

9.2 C Language Reference Manual

HAWK grammar and syntax is very similar to C. See the C LRM as further reference material.

See *The C Programming Language 2nd Edition* by Brian Kernighan and Dennis Ritchie

9.3 CSS Reference Manual

HAWK implements a limited set of CSS selectors. See the CSS reference manual for a fuller explanation of the implemented selectors.

https://developer.mozilla.org/en-US/docs/Web/CSS/Reference

9.4 LUA Language Reference Manual

HAWK tables take inspiration from Lua tables. See the Lua LRM as further reference material.

http://www.lua.org/manual/5.3/

9.5 POSIX Reference Manual

HAWK implements a limited set of Regex expressions. Regex operators are further defined in the POSIX reference manual.

https://www.gnu.org/software/guile/manual/html_node/POSIX.html