April 28, 2011

Key Magic Specification

A guide to the keyboard layout scripting system.

Important Note

This document does **not** represent an International Standard.

Introduction

The Key Magic layout system is used for mapping key combinations (on, for example, a QWERTY keyboard) to one or more letters in a different script. In addition, Key Magic supports several reordering rules to ease scripting of complex languages whose encoding orders differ from their common typing orders. The system was originally developed for the Myanmar language.

Scope

This document covers the format of Key Magic "layout" files (*.kms). It also covers the behavior of a keyboard with respect to its layout file. It does *not* cover the internal format of layout files, or any "compiled" format.

It is intended that this document be sufficient for enabling a developer to create his own implementation of Key Magic. Secondary concerns, such as efficiency or representation of Key Magic data structures in code, will generally be excluded or mentioned only briefly in notes.

Syntax

Note the following conventions regarding syntax:

- Non-colored (black) text refers to the current version of Key Magic.
- *Red italicized, underlined text* describes requirements which are deprecated or obsolete. At their discretion, developers may choose to implement some of these features. (Code will never be underlined.)
- *Blue italicized text* represents new (approved) requirements which will very likely appear in a later publication of the standard. Developers may choose to implement some of these features in advance of the new standard, and feedback on these features is particularly appreciated.
- **Bold text** is generally used to represent key terminology the first time it appears.

1. Source File Format

1.1 File Naming and Encoding

A Key Magic **keyboard** is represented as a **primary file**, and a series of optional **secondary files**. It is recommended that these files be given the extension "kms", and be encoded in UTF-8. However, this is not strictly required, so long as the files can be understood from a Unicode point of view.

Note: We use the term "file" to mean "readable stream of characters". Implementers may choose to represent these streams however they want for any given operating system.

When describing file elements, we will provide their PEG representations whenever possible, followed by a short description of what the syntax means. A full PEG for Key Magic source files appears in Appendix A. Note that whitespace is ignored, unless otherwise specified (e.g., inside strings).

Note: Previously, we used EBNF syntax for rule representation. However, PEG is better suited for our purposes. First, PEG is completely unambiguous, and can easily handle alternatives such as an optional first comment. Second, the major downside to PEG (non-constant space requirements) is mitigated, as Key Magic source files are usually quite small.

1.1.1 Brief PEG Primer

A Parser Expression Grammar (PEG) is an alternative to Context-Free grammars which is unambiguous and parseable in linear time. The syntax is very similar to the various extended EBNF syntaxes, with only a few key differences. Here is a (very) brief primer on PEGs; more information can be found online.

```
x ← 'a' 'b' 'c'
```

This means "x is composed of the predicates a, b, and c, in order". So, x = "abc".

```
x ← 'a'/'b'/'c'
```

This means "x is composed of *either* a, b, or c, whichever matches first".

```
x ← 'aa'/'a'/'b'
```

Note that PEGs perform greedy matching, so if the input string is "aa" then the second 'a' (pink, bold) will never match.

```
letter ← 'a'
x ← letter* letter
```

It is impossible for this pattern to match **x** since **letter*** will consume all 'a' letters, so there will be nothing left to match. This is a big difference between PEGs and CFGs.

```
nl ← '\n'
comment ← '//' (!nl .)* nl
```

This is called a "syntactic predicate", and it is a very powerful feature of PEGs. The '//' means "match two forward slashes". The final **nl** means "match a newline". In between, the pattern **(!nl .)*** matches every non-newline character before the final newline. The syntax **(!x y)** means "match *not* **x**, then rollback and match **y**". The syntactic predicate '&' does the same thing as '!', except that it requires a match to succeed, not fail. (It then backtracks to the start of the match, just like '!' does). I'm having a hard time thinking up a good example of how to use it. Consider this:

```
esc_chr ← 'n' / 'r' / 't' / '"'
esc ← '\\' (&esc_chr .)
str ← '"' (esc / .)* '"'
```

This represents a special kind of string, where a single backslash is *only* interpreted as an escape sequence if it would be valid. So, "hello\n" becomes "hello" followed by a newline, while "hello\m" is "hello" followed by a backslash, followed by the letter "m". We achieve this through the "&" syntactic predicate. In general, we rarely use "&", while "!" is much more common.

1.2 Two-Stage Parser

Key Magic uses a two-stage parser. The pre-parser performs the following tasks; the output from the pre-parser is a series of rules and variables, each on a single line.

1. Identify and save the "options" section.
2. Remove all whitespace, comments, and forced newlines.

The primary parser then operates on these rules and produces the final parse tree. Although it is possible to represent the Key Magic format using a single grammar file, a 2-stage parser was deemed appropriate for a few reasons. First, checking for comments and whitespace embedded in between each command tended to clutter the grammar and obscure what was really being checked for. Second, practically speaking many interpreted languages (including Javascript, which we used for the reference implementation) have a limit on the number of recursions that can be performed by one script. A 2-stage parser eliminates ambiguity in the first step, which greatly reduces the level of recursion in the primary parser.

We will describe the pre-parser in the remaining sections (1.2.X). All sections that refer to pre-parser elements will be prefixed with the "Pre-Parser" tag.

1.2.1 Pre-Parser: Top-Level Elements

```
file ← options? val:line+
line ← blankline sp item more_items* sp ('=>'/'=') sp item more_items* blankline
blankline ← (sp '\n')*
sp ← (skipnl / space / comment)*
skipnl ← '\\' sp_ '\n'
sp_ ← space*
space ← [ \t\r]
comment ← ('/*' (!'*/' .)* '*/') / ('//' (!'\n' .)* '\n')
```

Files consist of an optional "options" section and a series of lines. A newline can only appear at the start or end of an item. To break a long line, the "forced newline" must be used, which is a backslash (\) followed by the newline. Comments count as whitespace, and can be represented using C or C++ syntax. Comments cannot be nested. Carriage returns (\r) are ignored as whitespace; in other words, it does not matter if a carriage return appears without a newline directly after it.

1.2.2 Pre-Parser: Options

```
options ← sp_ '/*' sp_ (option / (!'*/' .))* '*/' sp
option ← '@' option_name sp_ '=' sp_ option_value sp_
option_name ← char+
option_value ← ('"' [^"]* '"') / ("'" [^']* "'")
char ← [a-zA-Z_]
```

A Key Magic "option" starts with an "@" sign, followed by at least one character, then an "=" sign, and then at least one Unicode letter. A "char" is any uppercase or lowercase letter from A to Z, or the underscore.

Older versions of Key Magic did not require options to be enclosed in quotes, and simply stopped matching after the first whitespace character:

```
option_value_old ← option_value / (sp_ (!space .)* sp_)
```

For example, the following script contains two options, colored in green bold text for the option_name and purple-bold-italic for the option_value.

```
/*
 * Our options are:
 * @track_capslocks = "false"
 * @name = "Unicode 5.1 Keyboard"
 */
'a' => 'b'
```

Note: Only the first comment contains options; any other comments with option-like syntax are ignored.

1.2.3 Allowed Options

The current list of option names and values is as follows. All options are optional; where appropriate, some are given default values.

Name	*Expected Values*	*Default Value*	*Notes*
track_capslocks	*true/false*	*false*	*Same as "track-capslock".*
track_capslock	true/false	false	If "true", then the Caps Lock key can be used in matching expressions.
eat_all_unused_keys	*true/false*	*true*	*Same as "eat-keys"*
eat-keys	true/false	true	Adds the proposition **ANY=>$1** as a default, show-stopping match for any keys that fail to match any other proposition.
us_layout_based	true/false	true	If "true", all key presses are mapped to the equivalent key at that position in an en-US keyboard. (**Question: what about keys that have no en-US equivalent?**)

smart_backspace	true/false	true	If "false", then any un-matched presses of "Backspace" will delete a single character from the end of the match string. If "true", an un-matched backspace will revert the match string to its state two keypresses back.
name	*(Anything)*	*N/A*	*Name of the keyboard layout, may be displayed to the user under implementation-defined conditions.*
description	*(Anything)*	*N/A*	*Informative description of the keyboard layout., may be displayed to the user under implementation-defined conditions.*
icon	*(Path-Name)*	*N/A*	*Absolute or relative path to a file which represents the keyboard layout.* *(***Note: Absolute paths break portability. Relative paths are fine;. Need to consider.***)*

1.2.3 Pre-Parser: Line Items

```
more_items ← sp '+' sp item sp
item ← keyword / variable / string / virtkey
keyword ← 'null' / 'NULL' / 'ANY' / vkcode / uni
vkcode ← 'VK_' [a-zA-Z0-9_]+
uni ← [uU] hex hex hex hex
hex ← [a-fA-F0-9]
variable ← '$' charnum+ var_suffix?
var_suffix ← '[' '$'? symbol+ ']'
digit ← [0-9]
charnum ← char / digit
symbol ← digit / [*^]
virtkey ← '<' sp mod* sp vkcode sp '>'
mod ← vkcode sp '&' sp
string ← '('? str_val ')'?
str_val ← (q1  str1  q1) / (q2  str2  q2)
q1 ← "'"
q2 ← '"'
```

```
str1 ← (!q1 strchar)*
str2 ← (!q2 strchar)*
strchar ← (bs esc) / .
bs ← '\\'
esc ← bs / q1 / q2 / uni
```

Each line is made up of a series of items separated by "=" or "=>". We will see later what these items mean. For now, just assume that the code snippet above tags all items in such a way that it is easy to remove whitespace. (See Appendix A for how this is done.)

Please note that the pre-parser's syntax is not strict. It will allow some invalid grammars to pass through undetected. The primary parser, however, will catch these errors.

1.3 Top-Level File Elements

```
file ← proposition+
proposition ← (variable/rule) '\n'
```

The output of the pre-parser is a formatted string of lines. There are no forced linebreaks, comments, or whitespace. Options have already been parsed and saved. Please note that, when we give examples, we will add whitespace to make these examples more readable.

A file consists of a series of "propositions", each of which is either a "rule" or a "variable". Rules are used to tell the system how to respond to key presses and what to reorder. Variables are static definitions which may be referenced by rules. Variables are of the form "x = y", while rules are of the form "x => y". The following code segment contains 2 variables (green bold) and 3 rules (purple bold italic).

```
//"Options" section is optional.
$var1    = 'abcd'
$an_elem = U1000 + U1001 + \
           U1002 + U1003
'a'         => 'b'
$var1[*]    => $an_elem[$1]
$an_elem[*] => '(' + $1 + ')'
```

At the end of each proposition is a newline character.

Future releases of Key Magic will have an additional proposition type:

```
proposition ← (variable / rule / key-rule ) '\n'
```

Here, a key rule is of the form "x |=> y"—these are used to respond to the keypress itself exactly once.

1.4 Variables

```
//General form (defined): DEFINE_VARIABLE(name, items)
//General form (used): VARIABLE(name)
variable ← var_name '=' var_item_list
var_name ← '$'  charnum+
var_item_list ← var_item var_item_opt*
var_item_opt ← '+' var_item
var_item = string / uni_letter / null / var_name / virt_key_unit / switch / var_element
charnum ← char / digit
digit ← [0-9]
char ← [a-zA-Z_]
```

A variable has a single identifier on the left of an equals sign, and one or more items (with '+' signs separating them) on the right. "Forced newlines" (a backslash followed by a literal newline character) can appear in most places to help with readability. Variables can also be used in other variables or rules; this allows developers to simplify the syntax of their Key Magic files.

Please take note of the "General form" comment. This is used much later to talk about the items that make up a Key Magic document. For example, if we have:

```
$test = 'abc' + null + U1000
$test2 = $test + 'abc'
```

...then the "General Forms" are:

```
DEFINE_VARIABLE('test', ['abc', null, U1000])
DEFINE_VARIABLE('test2', [$test, 'abc'])
```

...which will later become:

```
DEFINE_VARIABLE('test', [STRING('abc'), STRING(''), STRING('\u1000')])
DEFINE_VARIABLE('test2', [VARIABLE('test'), STRING('abc')])
```

For now, you will have to guess the general form. Later we will standardize this too, but for now I am not sure how to do this.

1.4.1 Strings, Unicode Letters, and Text

```
//   General form: STRING(value)
//       for null: STRING('')
// for uni_letter: STRING('\uXXXX')
null ← 'null' / 'NULL'
uni_letter ← [uU] hex hex hex hex
hex ← [a-fA-F0-9]
string ← (q1 str1 q1) / (q2 str2 q2)
q1 ← "'"
q2 ← '"'
str1 ← (!q1 strchar)*
str2 ← (!q2 strchar)*
strchar ← (bs esc) / .
bs ← '\\'
esc ← bs / q1 / q2 / uni_letter
```

Strings are enclosed by either single or double quotes. To embed a quote of the same type within the string, use a backslash. Double-backslash embeds a backslash, and you can also use the syntax \u1000 to embed unicode character U+1000 within the string. For example:

```
$var2 = 'Hi there, "name"'
$var3 = "I can't use double-quote (\") unless I add a backslash (\\)."
$var4 = "Here is Myanmar letter 'ka': \u1000";
```

In addition, Unicode letters may appear as literals:

```
//These two strings are identical:
$varA = U1000 + U1001
$varB = '\u1000\u1001'
```

Finally, the null string can appear by itself as 'null':

```
'a' => null //Rule to remove the letter 'a'
```

Semanticlaly, null is treated as the empty string "". So the following two strings are identical:

```
$varA = 'a' + null + null + U102C
$varB = 'a\u102C'
```

1.4.2 Virtual Key Units

```
//General form: VIRT_KEY(modCtrl, modAlt, modShift, vkCode)
// For example, Shift+VK_KEY_A is: VIRT_KEY(false, false, true, 0x41)
//              VK_BACK is: VIRT_KEY(false, false, false, 0x08)
virt_key_unit ← vk_mod / vk_code
vk_mod ← 'VK_SHIFT' / 'VK_ALT' / 'VK_MENU' / 'VK_CTRL' / 'VK_CONTROL'
vk_code ← /** VK codes; see Appendix D */
```

Virtual key “units” represent a single virtual key code. They provide an alternate way of specifying letter or number keys. Be aware that virtual key units do ***not*** imply that a key has been pressed.

```
//These two are functionally equivalent:
$var1 = VK_KEY_A + VK_KEY_B + VK_KEY_C
$var2 = 'a' + 'b' + 'c'
```

NOTE: *Version 2.0 removes the distinction of angle brackets for key “presses”, and uses angle syntax to represent virtual keys in their entirety. So, in 2.0 syntax:*

```
//These two are functionally equivalent:
$var1 = <VK_KEY_A> + <VK_SHIFT & VK_KEY_B> + <VK_KEY_C>
$var2 = 'a' + 'B' + 'c'
```

See the section on string representation for a discussion of how VK_ALT and VK_CTRL affect this unified syntax.

1.4.3 Variable Single Elements

```
//General form: VAR_ELEMENT(name, id)
var_element ← var_name '[' (!'0' digit) digit*  ']'
```

A "variable element" refers to a variable and an index greater than zero. For example:

```
$test = 'testing'
$i    = $test[5]
$i => 'I' //Converts lower-case i to capital 'I'
```

Key Magic uses the idea of a "simple string" to represent any variable that can be trivially reduced to a single string value. "Single variable" elements can only operate on simple strings.

```
$var1 = 'testing'                  //Simple
$var2 = U1000 + U1001 + $var1      //Simple
$var3 = "Hi: \u1000" + null + "!"  //Simple
$var4 = VK_KEY_N                   //Simple
$var4 = <VK_KEY_N>                //ERROR: Not a simple string.
```

The last example is supported in 2.0 syntax.

1.4.4 Switches

```
//General form: SWITCH(name)
switch ← '(' (q1 charnum+ q1) / (q2 charnum+ q2) ')'
```

A switch is identified by a name, in quotes. Switches are used to globally modify the rule-matching process, so adding a switch to a variable definition has no direct effect on the keyboard script until that variable is used in a rule.

1.5 Rules

```
//General form: DEFINE_RULE(lhs, rhs)
rule ← rule_lhs '=>' rule_rhs
rule_lhs ← rule_lhs_item_opt* rule_lhs_item / virt_key_press
rule_lhs_item_opt ← rule_lhs_item '+'
rule_lhs_item ← wild_var / var_item / wild
virt_key_press ← '<' vk_mod_item* virt_key_unit '>'
vk_mod_item ← vk_mod '&'
rule_rhs ← rule_rhs_item rule_rhs_item_opt*
rule_rhs_item_opt ← '+' rule_rhs_item
rule_rhs_item ← backref / wild_backref / var_item
```

Rules are built of single objects chained together by plus signs, just like variables are. Syntactically, variables use "=" to separate the left and right sides of each declaration, while rules use "=>". Rules can also contain several additional items, depending on which side (left, LHS or right, RHS) one is focusing on.

In addition, rules allow a single "pressed" virtual key combination to be present at the end of the LHS. (In fact, a key press is allowed as the only item on the left.) Virtual key presses are defined as angle brackets '<' and '>' surrounding a `virt_key_unit`, with optional modifiers chained before that unit using '`&`'. These are used to signify the user holding down the modifiers and pressing a single key. For example:

```
<VK_KEY_A> => 'a key was pressed'
<VK_SHIFT & VK_KEY_B> => 'shift was held and b was pressed'
```

```
   <VK_ALT & VK_CTRL & VK_KEY_C> => 'alt and ctrl were held and c was pressed'
   <VK_ALT & VK_SFHIT> => 'alt was held and shift was pressed'
```

The purpose of the angle brackets is to specify that a single rule should match only once, the first time that key is pressed. So, of the following snippets:

```
   <VK_KEY_M> => 'mm'
   VK_KEY_M => 'mm'
   "m" => 'mm'
```

...the first will type 'm' twice for every time the 'm' key is pressed, while the second and thirdwould loop forever.

Version 2.0 unifies the syntax for virtual keys, allowing angle brackets to be used to specify a virtual key without requiring that it is pressed. This allows the VK_SHIFT modifier to be used to specify capital letters using virtual key syntax alone. A "pressed" letter is represented with a new type of rule using "|=>", and rules by default will not loop. (Will write more later.)

1.5.1 Wildcards and Back-References (Simple)

```
   //General form: WILD()
   //General form: BACKREF(id)
   wild ← 'ANY'
   backref ← '$' (!'0' digit) digit*
```

Note that older implementations allowed an asterisk (*) to represent wildcards:

```
   wild ← 'ANY' / '*'
```

A wildcard matches a single character. That character can be retrieved using a backref, numbered starting at one. For each rule item separated by a '+', the backref id is increased by one. This allows backrefs to be used to retrieve *any* matched string, not just those matched by wildcards. For example:

```
   ANY + ANY => $2 + $1                   //Switch the last two letters
   ANY + 'hello' + ANY => $3 + $1 + $2    //Would turn 'XhelloY' into 'YXhello'
```

1.5.2 Wildcards and Back-References (Complex)

```
   //General form: WILD_VAR_ALL(name) ; WILD_VAR_NONE(name)
   //General form: WILD_BACKREF(name, id)
   wild_var ← (var_name '[*]') / (var_name '[^]')
   wild_backref ← var_name '[' '$' (!'0' digit) digit* ']'
```

A *variable wildcard* will match a single character contained within that variable's string (if '*' is used) or *not* contained within that variable's string (if '^' is used). Variable wildcards use the concept of a "simple" string, defined earlier, which means that they are not restricted to single-item strings. Consider the following, which also demonstrates how regular back-references can be used in combination with variable wildcards:

```
   $var = 'abc' + VK_KEY_D + U0065 ;
   $var[*] => '(' + $1 + ')'         //Will turn 'b' into '(b)'
   $var[^] => '(' + $1 + ')'         //Will turn 'f' into '(f)'
```

A *wildcard back-reference* is similar to a back-reference, except that it operates on the relative ID of the item which is matched. It is numbered similar to a back-reference, but can only be used to refer to *variable wildcard* matches made using '*'. It returns the index in the simple string that the *variable wildcard* matched, and is intended for allowing easy parallel substitution arrays, like so:

```
   $key1 = 'abcdefg'
   $key2 = 'ABCDEFG'
   $key1[*] => $key2[$1]      //Will turn 'a' into 'A', 'b' into 'B', etc.
```

Be aware of potential confusion with similar, equally valid syntax:

```
   $key1[*] => $key2[1]       //Will turn any letter ('a' through 'g') into 'a'
   $key1[*] => $1             //Will turn any letter ('a' through 'g') into itself
```

1.6 (New Stuff)

(Add the stuff on |=> syntax, etc.)

2. Rule Matching

2.1 Matching Attempt Lifecycle

When the user presses a key, the following processing steps occur:

1. Look through each rule in **<RULES>** until you find one that "matches".
2. Execute the state machine for the rule that matched.
3. If a rule matched, go back to step 1.

There are some additional considerations and side-effects. For example, only one rule can match the actual virtual key that the user pressed, and if a rule outputs a single ASCII letter, it ceases all execution. Full pseudo-code is provided in the appendices, and the state machines in the following sections will explain any exceptional behavior as it becomes relevant to the discussion.

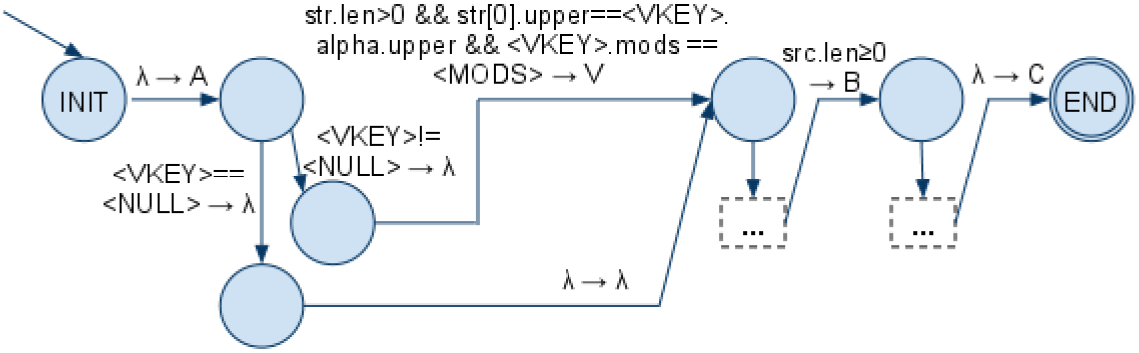
2.2 Search Order of Rules

The **<RULES>** list is actually sorted the first time it is loaded. This is to allow... (more later: Order of VK, etc.)

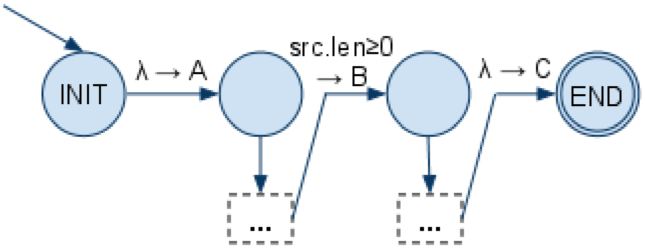
(TODO: Add pseudo-code for sorting, etc.)

2.3 General Format of a Single Rule

Any rule in Key Magic can be represented as a state machine, of the form:



For the sake of our initial discussion, this can be further simplified into the following:



	Translation Actions

A	groups.clear()
	group_ids.clear()
	src = **<TYPED_STR>**.reverse()
	LHS = true

B	src = src.reverse()
	groups = groups.reverse()
	group_ids = group_ids.reverse()
	LHS = false

C	**<TYPED_STR>** = src

Our state machine syntax assumes the following:

- A -> B means that a state requires condition **A** before it may transition on that edge. The null condition (λ) means the state may always transition. Simple conditions like **src.len>0** may be listed; this means that the "src" state variable must have a "len" property which is greater than 0. (A "state variable" is a variable in our state machine, ***not*** a Key Magic variable.)
- In A -> B, **B** represents the state transition **action**. We list these actions in the table on the right. For example, the first transition (λ -> A) references action A, which performs five statements. The first statement, **groups.clear()**, removes all elements from the **groups** array.
- A rule may perform a null action (A -> λ), in which case it modifies nothing.
- The "..." syntax just means that these transitions will be filled in later. We will actually build them up from the "rules" themselves.
- **<TYPED_STR>** is the roman+myanmar string the user has currently typed.
- Although it is not used in the previous diagram, **<LHS>** in any action represents the matched string segment.
- The "reverse" function takes as input a string or an array and returns a copy of that string/array in reverse order.

So, each rule operates as follows:

- First, we transition and perform action "A". This resets the state of our rule matcher, and sets the "LHS" variable to "true", indicating that we are left of the "=>" symbol. We also copy the input string into the "src" variable, and reverse it, since matching is easier from the right.
- Next, we perform a series of matching transitions ("..."), corresponding to the elements left of the "=>". If any of these fail, the match fails and the rule is skipped.
  Note that these rules may modify "src", usually by trimming it.
- After all LHS elements are matched, we transition ***only*** if the source string is not in an invalid state. In this case, we un-reverse the source string and groups/group_ids arrays. We set LHS to false, inddicating that we are right of the "=>".
- Next, we match all rules to the right of the "=>". These rules tend to add on to "src".
- Finally, after all RHS rules are matched, we transition to the end state, where we copy our "src" string back into **<TYPED_STR>**. At this point, the rule has been "matched", and we go back to the search phase.

2.3.1 Data Structures Available to our State Machines

(We need to explain exactly which variables are available to our matching algorithm. For example, a "rule" is transformed into a "simple" representation, a "reverse state machine", or both. A <VKEY> VIRT_KEY() represents the current keypress. The <TYPED_STR> represents the current typed string. <SWITCHES> and <VARIABLES> arrays are available. Simple structures like an "array" (with pop/push added), etc., are available and used in the next few sections.)

2.3.2 Pseudo-Language Available to our State Machines

Our state machines are described using a kind of pseudo-code, which is completely detailed in Appendix C. However, we expect that most experienced programmers will need only a few brief examples of this language, which we will present here:

```
//Variables are either integers, strings, booleans, or user-defined types (UDTs)
myvar = 10       //Create a variable, assign it the value 10
myvar = "ABCD"   //Re-defined our variable to hold "ABCD"

second = copy(myvar) //Copy the variable and change it; the original remains unchanged.
second[0] = "Z"
test = second[0].ord // This returns 90, the Unicode value of 'Z'
second.append('EFG') //second is now "ZBCDEFG". myvar is "ABCD"
```

```
myvar = second.reverse()  //reverse() returns a reversed copy of the string or array
myvar = [1,2,3].reverse() //arrays are user-defined types.
myvar[2] = 4              //They can be indexed similarly to strings
myvar.push(5)             //They use "push" instead of "append"
"test".len == 14     //Strings and arrays both have a "len" parameter
myvar.len != 32      //  to return their length in character/elements.

myvar.clear()    //clear() empties the array/string/map
myvar['hi'] = 10 //Using a string index changes this variable from an array to a "map"
myvar.len        //Error! Maps don't have a len parameter
myvar.size       //This is ok; it returns the number of keys in the map
if (myvar['hi']) {}   //This will check if the map has a key called 'hi'.

v1 = VirtKey(false, false, true, VK_KEY_A)    //Represent "Shift+A"
v2 = VirtKey(false, false, false, VK_KEY_A)    //Represent "A" (no shift)
v1.matches(v2)  //Returns true (v2 doesn't need modShift to be true)
v2.matches(v1)  //Returns false (v1 needs modShift to be true)
```

(to-do: more, but be brief. We need to cover our own UDTs, like Variables and States)

Implementers do not need to create parsers for our pseudo-code. Rather, all they need to do is confirm that their implementation language (Java, C++, Ruby, etc.) behaves the same way as our pseudo-code. For example, if a string in some language has a "reverse()" method that operates in-place instead of returning a copy, then the Implementer should subclass **string**, override **reverse()**, and ensure that a copy is returned. Most programming languages will require Implementers to create UDTs for Variable, State, and (more...?). In an object-oriented language, UDTs can be realized using classes.

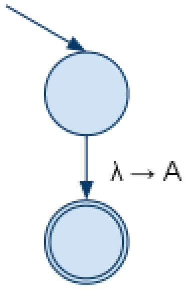
2.4 State Machine Format of Rule Primitives

Each rule implies up to two state machines, one for each side (LHS or RHS) the rule may appear on in an expression. We will use suffixes to represent these cases, so **STRING_RHS** represents a "string" rule when it appears on the RHS of the "=>" and **STRING_LHS** represents the case where it is on the left. Rules like **BACKREF_RHS** have no **_LHS** counterpart.

We will now detail each rule type by the state machine it produces. If a rule is not listed, it is assumed to produce a null-to-null transition (λ->λ), and is automatically pruned from the state machine. Comments are a good example of this. For each rule, the state machine has a single initial state and a single final state. Thus, multiple rules can be "attached" by unifying the final state of the previous rule and the initial state of the first. (Alternatively, a null-to-null transition could be performed from the previous final to the next initial state). This allows us to build up a state machine for any rule using these simple fragments.

There are two additional points to take note of, which the observant reader may have already deduced. First, all RHS elements (except variables) have state trees of the form:

(todo: Actions belong in the nodes.)



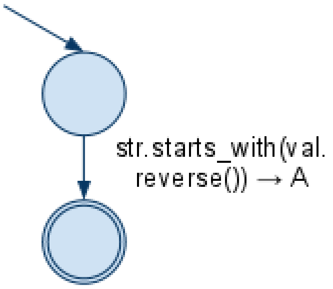
...and as such, multiple RHS elements may be combined by appending their actions.

Second, the rule has technically been "matched" after all LHS elements match. That is to say, RHS elements should never cause the matching process to fail.

2.4.1 STRING_LHS and STRING_RHS

String rules on the LHS simply match that string in reverse. Assuming STRING(val), we get:

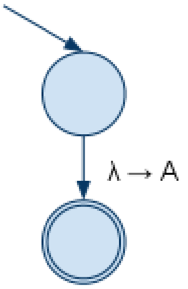
		Translation Actions

	A	groups.push(val.reverse())
		group_ids.push(-1)
		str = str.substr(val.len, str.len)

Here, **substr(X,Y)** returns a trimmed string starting at X and ending before Y.

They provide a good example of the kind of "general" update performed on "groups", "group_ids" and "src". First, the matched string is appended to "groups" so that it may be retrieved later. An invalid id (-1) is appended to "group_ids", so that ordering is preserved. Finally, "src" is trimmed by the length of the matched string, so that the next rule may continue matching on new characters.

String rules on the RHS append that string to "src":

		Translation Actions
	A	str.append(val)

Since "src" was already reversed at the "=>" transition, we don't need to perform any reversal here.

2.4.2 VARIABLE_LHS and VARIABLE_RHS

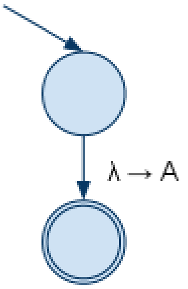
Assuming **VARIABLE(name)**, we need to introduce the syntax:

```
        str.append(<VARS>[name].simple)
```

Here, **<VARS>** is the list of declared variables, saved by name. **<VARS>[name]** retrieves the variable called **name**, which returns the list of rules making up the RHS of the variable. The property **simple** is the result of calling **simple()** on that variable's list of rules. It takes this list of rules and attempts to return the "simple string" defined earlier in Section 1.5.3. This conversion should be done once at "compile" time; otherwise, there is a risk of entering an invalid state.

That said, here is the diagram for **VARIABLE_RHS**:

		Translation Actions

	A	s = **<VARS>**[name].simple
		str.append(s)

VARIABLE_LHS is somewhat more complex. Whenever we encounter a variable on the LHS of a rule, we jump to the state machine for that variable's RHS. We represent this with the syntax:

```
        states.push(<STATE>.next())
        <STATE> = state(<VARS>[name])
```

This pushes the next state onto the stack of active "states", then set the current state equal to the first state of the variable in question. Note that ***all*** variables end with the statement:

```
        <STATE> = states.pop()
```

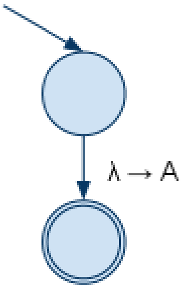
...which will return execution to where it left off before the variable occurred.

Please note that infinite variable loops are impossible, since each variable must be declared and defined before it can be used.

2.4.3 BACKREF_RHS

Back-references only occur on the RHS of an equation. They make use of the **groups** stacks that are built up on the LHS. Assuming **BACKREF(id)**, we get:

		Translation Actions

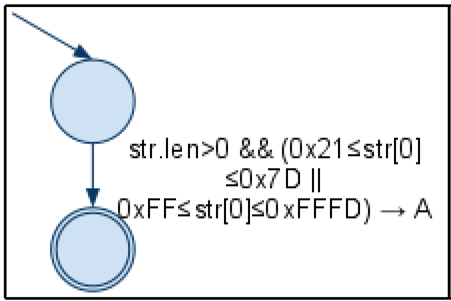
	A	s = groups[id]
		str.append(s)

Again, it is beneficial to check at compile time that the **id** of a **BACKREF** will never be larger than the groups list.

2.4.4 WILDCARD_LHS

Single-letter wildcards match a single character within the valid range:

		Translation Actions

 str.len>0 && (0x21≤str[0]
≤0x7D ||
0xFF≤str[0]≤0xFFFD) → A
	A	groups.push(str[0])
		group_ids.push(-1)
		str = str.substr(1, str.len)

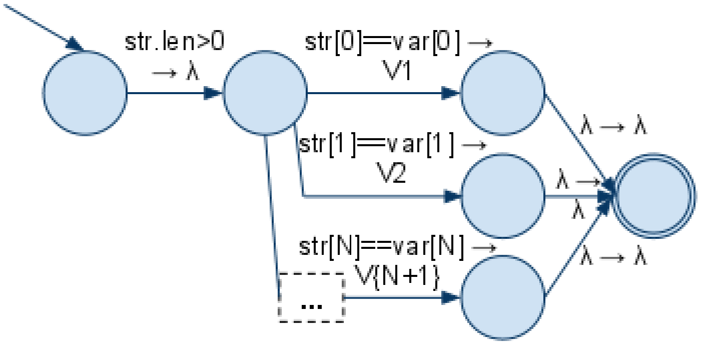
2.4.5 WILDCARD_VARIABLE_LHS

Wildcard variables can either include all characters in that variable ($var[*]) or none ($var[^]). For the case where all characters are covered, we have WILDCARD_VAR_ALL(name), and we assume:

```
    var = <VARS>[name].simple
    N = var.len
```

...for simplicity.

(note: This is wrong; it should always be str[0], not str[N])



	Translation Actions

V1	groups.push(str[0])
	group_ids.push(0)
	str = str.substr(1, str.len)

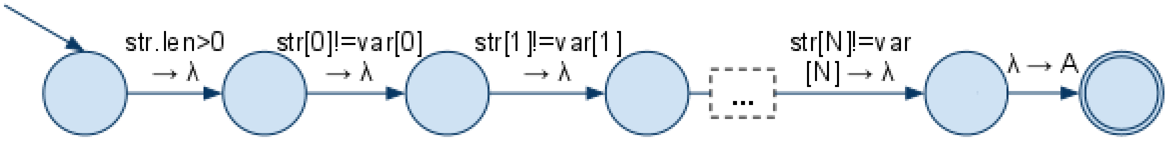
V2	groups.push(str[1])
	group_ids.push(1)
	str = str.substr(1, str.len)

...	...

V{N+1}	groups.push(str[N])
	group_ids.push(N)
	str = str.substr(1, str.len)

Functionally, this code simply splits each element of the variable's "simple string" representation into a transition; this transition saves both the letter matched and the ID of the element saved.

For the case with no characters allowed, we have WILDCARD_VAR_NONE(name), and it looks like this:



	Translation Actions

A	groups.push(str[0])
	group_ids.push(-1)
	str = str.substr(1, str.len)

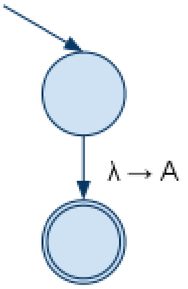
Similar to the _ALL example, this state machine splits every character of the variable's "simple string". However, instead of branching, it requires the current letter to go through a series of "not" matches on each new state. After the final "not" match, the first letter of **str** is saved and the state machine exits.

Both of these state machines may be simplified in code (e.g., using a for loop) if required.

2.4.6 BACKREF_ID_RHS

This is the only element which uses the **group_ids** stack. Given BACKREF_ID(name, id), we get:

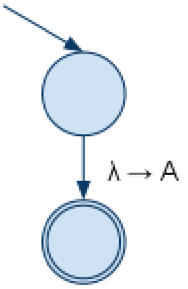
		Translation Actions

	A	s = **<VARS>**[name].simple[group_ids[id]]
		str.append(s)

As usual, it is possible to check at compile time whether **id** will refer to a group that actually generates an ID.

2.4.7 SWITCH_RHS

Switches are handled in a special case on the LHS. For the RHS, all they do is modify the global **<SWITCHES>** array. For SWITCH(name), it looks like this:

		Translation Actions
	A	**<SWITCHES>**[name] = true

2.4.8 VKEY_RHS, Special Case VKEY_LHS

Virtual Keys are also handled in a special case on the LHS. For the RHS, they are treated as STRING(make_simple(value)). In other words, only alpha-numeric virtual keys with an optional **VK_SHIFT** modifier are allowed, and these are simply treated as one-character strings.

VKEY_LHS can be treated as a string for the "bracketless" specification of virtual keys in Key Magic 1.0. So, **VK_KEY_A** can be treated as **STRING("a")**. If brackets are used (**<VK_KEY_A>**, or **<VK_SHIFT & VK_KEY_A>**), then the more complex format described later must be used.

2.4.9 UNICODE_LETTER_LHS, UNICODE_LETTER_RHS

Both are treated as STRING_LHS/RHS("\"\\"+value+"\""). In other words, U1000 is treated as STRING_LHS("\u1000") or STRING_RHS("\u1000"), depending on its context.

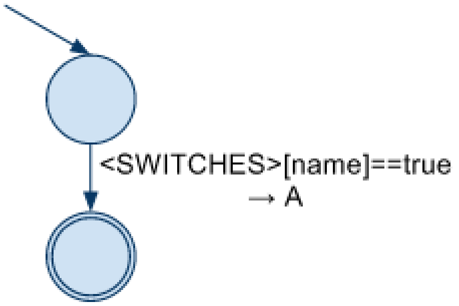
2.4.10 VARARRAY_ITEM_LHS and VARARRAY_ITEM_RHS

Given VARARRAY_ITEM_LHS/RHS(name, id), we simply convert this to STRING_LHS/RHS(**<VARS>**[name].simple[id]). In other words, we perform a match on a single-letter string after converting this variable to a simple string.

2.5 Special Case 1: SWITCH_LHS

The only problem with switches to the left of the "=>" is that they follow an "all or nothing" paradigm. In other words, if a rule requires ('switch_1') and ('switch_2') to be ON, and both are actually ON, then both should be turned OFF before transitioning to the RHS. However, if only ('switch_1') is on, that should ***not*** turn ('switch_1') OFF. More importantly, if both *are* on, but the remainder of the rule fails to match, nothing should happen.

Thus, given SWITCH(name), we represent SWITCH_LHS as follows:

		Translation Actions
	A	sw_temp.push(name)

This requires us to re-write our original "general rule" transition table to reset the temporary switch array, and to clear all switches when transitioning over to the RHS. This only affects transitions "A" and "B", and all changes involve the new "sw_temp" array.

	Translation Actions

A	groups.clear()
	group_ids.clear()
	sw_temp.clear()
	src = typed_str().reverse()
	LHS = true

B	src = src.reverse()
	groups = groups.reverse()
	group_ids = group_ids.reverse()
	for name in sw_temp {
	<SWITCHES>[name] = false
	}
	LHS = false

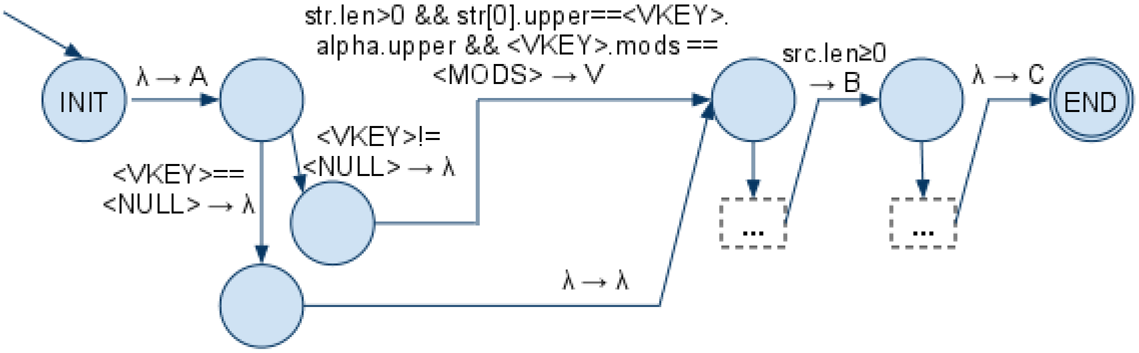
With these changes, switches now work. Implementers are actually advised to provide further optimizations by removing switches entirely from the state machine of a rule and handling them automatically. This will greatly speed up matching.

2.6 Special Case 2: VKEY_LHS

Virtual keys are, in theory, easy to handle on the LHS. However, if we wish to avoid generating invalid state machines then this becomes more difficult. (Consider, for example, a rule with several variables, each of which references a virtual key). Moreover, if a rule contains no virtual key entry, then the newest letter can still be matched using a string.

To deal with these complications, we add the following special variable **<VKEY>**, which is equal to the ***first*** non-switch rule item in a full variable expansion of the current predicate's LHS, or **<NULL>** if no such item exists. (Recall that rules and variables are expanded in reverse). Any additional virtual keys (or any virtual keys which are not the first non-switch rule item) are errors. (An algorithm for determining **<VKEY>**'s value will be detailed later).

Given **<VKEY>**, we can expand our original diagram to check if that key is down.



	Translation Actions

A	groups.clear()
	group_ids.clear()
	src = typed_str().reverse()
	LHS = true

B	src = src.reverse()
	groups = groups.reverse()
	group_ids = group_ids.reverse()
	LHS = false
	if (**<RULE_VK>**!=**<NULL>**) { **<VKEY>** = **<NULL>** }

C	typed_str = src (TO-DO: Add "single letter ascii")

V	groups.push(str[0])
	group_ids.push(-1)
	str = str.substr(1, str.len)

In addition to the **<VKEY>** special variable, we must define **<MODS>** to be equal to the current set of modifiers, and say that performing a comparison (m1 == m2) on two modifier sets checks if the first is a subset of the second. So, if m1 = {VK_CTRL} and m2 = {VK_CTRL & VK_ALT}, then m1 == m2, but m2 != m1. (We might clear this up later.) In addition, we use **<VKEY>.alphanum** to get the alphanumeric character associated with this virtual key, and use **char.upper** to refer to the upper-case representation of a letter. If a virtual key matches, that letter is removed from **str**.
Note that, after matching a rule with a **<RULE_VK>**, we set the global **<VKEY>** to **<NULL>**. This is because a virtual key is only allowed to be consumed once; after that, it no longer contributes to the matching process.

(TODO: Currently **<RULE_VK>** and **<VKEY>** are not mentioned anywhere. We need to fix this.)
(NOTE: It would make more sense for <VKEY>==<NULL> to actually add the virtual key's .alphanum value to the current string. Otherwise, we will have no way of comparing VKEYS for things like Backspace. I will have to change this diagram later.)
Again, implementers are recommended to extract virtual keys from the actual state machine and handle them independently.

3. FAQ

(Common questions)

Appendix A: Full Grammar

The following is a complete PEG representation for Key Magic. It uses the PEG.js implementation, which assumes that alternatives can be tagged with javascript code (in curly braces), which operates after a match. Strings match as strings, numbers match as numbers, and repetitions match as arrays of these values. In addition, the syntax **name:peg_syntax** signifies that the result of **peg_syntax** should be referenced by **name** in the javascript code. For example, a simplified version of our "option" code might look like this:

```
option =
   '@' name:option_name '=' value:option_value { options[name] = value; }
option_name =
   val:[a-zA-Z_]+  { return val.join(''); }
option_value =
   '"' val:[^"]* '"' { return val.join(''); }
```

The **val.join()** code in **option_name** and **option_value** take the result of the expressions **[a-zA-Z_]+** and **[^"]*** respectively, and convert them from arrays of characters to strings. These are **return**'d up to the **option** matcher, which then saves the value in an array called **options**. (Note that **options** is defined outside of the PEG grammar).

We chose PEG.js for our reference implementation because javascript is both a powerful language and a simple one to understand. However, a few more details would be pertinent. First, the following arrays (used to hold the results of parsing) are assumed to exist:

```
options = new Array();      //string->string
switches = new Array();     //string->boolean
variables = new Array();    //string->Prim[]
rules = new Array();        //int->{'lhs':Prim[],'rhs':Prim[]})
```

The **options** array is indexed by the name of that option (without the leading **@**) and stores the value of that option (without quotes). The **switches** array is similar, indexing with the name of a switch (without parentheses or quotes) and storing its current value (**false**) as the value. The **variables** array indexes by the name of each variable (without the **$**) and stores as its value another array, this one of **Prim** objects. Finally, the **rules** array employs a simple integer index, and stores a pair of **Prim** arrays, one representing the LHS of the expression and the other representing the RHS. The **Prim** class is defined like so; note that this is not the same as the **Prim** class defined in Appendix B (although there is no reason why it can't be).

```
function Prim(type, value, id) {
  this.type = type;
  this.value = value;
  this.id = id;
}
//Sample usage, $keysU[$1] becomes:
var x = new Prim('WILD_BACKREF', 'keysU', 1);
```

VKeys can be used instead of **Prim**s in any case. They contain a **type** (so that one can test whether this is a **Prim** or a **VKey** without using inheritance) and the virtual key code and modifier flags required to represent a keypress.

```
function VKey(type, mods, key) {
  this.type = type;
  this.modShift = this.find(mods, 'VK_SHIFT')
  this.modAlt = this.find(mods, 'VK_ALT')
  this.modCtrl = this.find(mods, 'VK_CTRL')
  this.vkCode = VIRT_KEY_CODES[key];

  this.find = function(arr, item) {
    for (var i=0; i<arr.length; i++) {
      if (arr[i] == item) { return true; }
    }
    return false;
  }
}
//Sample usage, <VK_ALT & VK_KEY_A> becomes:
var x = new VKey('VIRT_KEY', ['VK_ALT'], 'VK_KEY_A');
```

The variable **VIRT_KEY_CODES** is defined as:

```
{
  'key' : value,
  'key' : value,
  ...
}
```

...for each **KEY_NAME** and **KEYCODE_VALUE** in Appendix D. The canonical name is used in cases where multiple **KEY_NAME**s exist.

Finally, PEG.js also adds a C++ style commenting syntax to the grammar file.

Pre-Parser Syntax

```
/**
 * The Key Magic Pre-Parser PEG syntax file.
 *  This file knows no Key Magic semantics,
 *  except options and anything needed to remove spaces properly.
 */

file =
      options?
      val:line_or_comment+
      blanklines    { return val.join(''); }

ws =
      [ \t\r]*

options =
      blanklines
      '/*'
      ((&'@' option)/(!'*/' .))*
      '*/'

option =
      '@' name:option_name ws '=' ws value:option_value ws { options[name] = value; }

option_name =
      val:char+  { return val.join(''); }

char =
      [a-zA-Z_]

digit =
      [0-9]
```

```
charnum =
      char / digit

symbol =
      digit / [*^]

option_value =
      ('"' val:[^"]* '"') { return val.join(''); }
    / ("'" val:[^']* "'") { return val.join(''); }

skipnl =
      '\\' ws '\n'

ignore =
      (skipnl / comment / [ \t\r])*

comment =
      ('/*' (!'*/' .)* '*/')  { return ''; }
    / ('//' (!'\n' .)*)       { return ''; }

line_or_comment =
      blanklines
      val:(comment/line) { return val; }

line =
    lhs1:item  lhs2:more_items*
    ignore arrow:('=>'/'=')
    ignore rhs1:item rhs2:more_items*    { return lhs1 + lhs2.join('') + arrow + rhs1
+ rhs2.join('') + '\n'; }

blanklines =
      [ \t\r\n]*

more_items =
      ignore '+' ignore val:item ignore { return '+' + val; }

item =
      keyword
    / variable
    / string
    / virtkey

keyword =
      val:charnum+  { return val.join(''); }

variable =
      '$' v1:keyword v2:var_suffix?  { return '$' + v1 + v2; }

var_suffix =
      '[' v1:'$'? v2:symbol+ ']'   { return '[' + v1 + v2.join('') + ']'; }

uni =
      v1:[uU] v2:hex v3:hex v4:hex v5:hex  { return v1 + v2 + v3 + v4 + v5; }

hex =
      [a-fA-F0-9]
```

```
vkcode =
      pre:'VK_' post:[a-zA-Z0-9_]+   { return pre + post.join(''); }

virtkey =
      '<' ignore mods:mod* ignore code:keyword ignore '>' { return '<' + mods.join('')
+ code + '>'; }

mod =
      val:keyword ignore '&' ignore   { return val + '&'; }

q1 = "'"
q2 = '"'
bs = '\\'

//Switches are represented as strings, for now.
string =
      l:'('? val:str_val r:')'? { return l + val + r; }

str_val =
      (q1  val:str1  q1) { return "'" + val.join('') + "'"; }
    / (q2  val:str2  q2) { return '"' + val.join('') + '"'; }

str1 =
      (!q1 v:strchar { return v; })*

str2 =
      (!q2 v:strchar { return v; })*

strchar =
      (v1:bs v2:esc) { return v1 + v2; }
    / .

esc =
      bs / q1 / q2 / uni
```

Primary Parser Syntax

```
/**
 * The Key Magic PEG syntax file.
 *  This file assumes that the source has been pre-parsed to remove comments and
spaces.
 */

file =
      proposition+

digit =
      [0-9]

char =
      [a-zA-Z_]

charnum =
      char / digit

proposition =
      (variable/rule) '\n'
```

```
variable =
      key:var_name  '=' value:var_item_list   { variables[key] = value; }

var_name =
      '$'  val:charnum+   { return val.join(''); }

var_item_list =
      v1:var_item vN:var_item_opt*   { return [v1].concat(vN); }

var_item_opt =
      '+' val:var_item  { return val; }

var_item =
      val:string         { return new Prim('STRING', val, -1); }
    / val:uni_letter     { return new Prim('STRING', val, -1); }
    / null               { return new Prim('STRING', '', -1); }
    / val:var_name       { return new Prim('VAR_NAME', val, -1); }
    / val:virt_key_unit  { return new VKey('VIRT_KEY', [], val); }
    / val:switch         { switches[val]=false; return new Prim('SWITCH', val, -1); }
    / val:var_element    { return new Prim('VAR_ELEMENT', val.name, val.id); }


null =
      'null' / 'NULL'

q1 = "'"
q2 = '"'
bs = '\\'

string =
      (q1  val:str1  q1) { return "'" + val.join('') + "'"; }
    / (q2  val:str2  q2) { return '"' + val.join('') + '"'; }

str1 =
      (!q1 v:strchar { return v; })*

str2 =
      (!q2 v:strchar { return v; })*

strchar =
      (bs esc) / .

esc =
      bs / q1 / q2 / uni_letter

uni_letter =

[uU] v1:hex v2:hex v3:hex v4:hex  { return String.fromCharCode(parseInt(v1+v2+v3+v4,
16)); }

hex =
      [a-fA-F0-9]


virt_key_unit =
      vk_mod / vk_code
vk_mod =
      'VK_SHIFT'
    / 'VK_ALT'  / 'VK_MENU' { return 'VK_ALT'; }
    / 'VK_CTRL' / 'VK_CONTROL' { return 'VK_CTRL'; }


switch =
      '(' val:((q1 charnum+ q1)/val:(q2 charnum+ q2)) ')'  { return val[1].join(''); }


var_element =

key:var_name  '['  v1:(!'0' v:digit{return v;}) vN:digit*  ']'  {
value=parseInt(v1+vN.join('')); return {'name':key,'id':value} }



rule =
      lhs:rule_lhs '=>'  rhs:rule_rhs { rules.push({'lhs':lhs,'rhs':rhs}); }

virt_key_press =
      '<' mods:vk_mod_item* val:virt_key_unit '>'   { return new VKey('VIRT_KEY',
mods, val); }

vk_mod_item =
      val:vk_mod '&'  { return val; }

rule_lhs =
      lhsN:rule_lhs_item_opt* lhs1:(rule_lhs_item/virt_key_press)   { return
lhsN.concat([lhs1]); }

rule_lhs_item_opt =
      val:rule_lhs_item '+' { return val; }


rule_lhs_item =
      wild_var
    / var_item
    / wild          { return new Prim('WILD', '', -1); }

wild = 'ANY'

wild_var =
      (val:var_name '[*]')   { return new Prim('WILD_VAR_ALL', val, -1); }
    / (val:var_name '[^]')   { return new Prim('WILD_VAR_NONE', val, -1); }

rule_rhs =
      rhs1:rule_rhs_item rhsN:rule_rhs_item_opt*    { return [rhs1].concat(rhsN); }

rule_rhs_item_opt =
      '+' val:rule_rhs_item { return val; }

rule_rhs_item =
      val:backref       { return new Prim('BACKREF', '', val); }
    / val:wild_backref  { return new Prim('WILD_BACKREF', val.name, val.id); }
    / var_item
```

```
backref =
      '$' v1:(!'0' v:digit{return v;}) vN:digit*   { return parseInt(v1+vN.join('')
); }

wild_backref =

key:var_name '[' '$' v1:(!'0' v:digit{return v;}) vN:digit* ']' {
value=parseInt(v1+vN.join('')); return {'name':key,'id':value} }



//Listed last (it's long)
vk_code =
      'VK_TAB'
   / 'VK_RETURN' {return 'VK_ENTER';} / 'VK_ENTER'
   / 'VK_SHIFT'
   / 'VK_CONTROL' {return 'VK_CTRL';} / 'VK_CTRL'
   / 'VK_ALT' / 'VK_MENU' {return 'VK_ALT';}
   / 'VK_PAUSE'
   / 'VK_CAPITAL' {return 'VK_CAPSLOCK';} / 'VK_CAPSLOCK'
   / 'VK_KANJI'
   / 'VK_ESCAPE'
   / 'VK_SPACE'
   / 'VK_PRIOR'
   / 'VK_NEXT'
   / 'VK_DELETE'
   / 'VK_KEY_0' / 'VK_KEY_1' / 'VK_KEY_2' / 'VK_KEY_3' / 'VK_KEY_4' / 'VK_KEY_5'
   / 'VK_KEY_6' / 'VK_KEY_7' / 'VK_KEY_8' / 'VK_KEY_9'
   / 'VK_KEY_A' / 'VK_KEY_B' / 'VK_KEY_C' / 'VK_KEY_D' / 'VK_KEY_E' / 'VK_KEY_F'
   / 'VK_KEY_G' / 'VK_KEY_H' / 'VK_KEY_I' / 'VK_KEY_J' / 'VK_KEY_K' / 'VK_KEY_L'
   / 'VK_KEY_M' / 'VK_KEY_N' / 'VK_KEY_O' / 'VK_KEY_P' / 'VK_KEY_Q' / 'VK_KEY_R'
   / 'VK_KEY_S' / 'VK_KEY_T' / 'VK_KEY_U' / 'VK_KEY_V' / 'VK_KEY_W' / 'VK_KEY_X'
   / 'VK_KEY_Y' / 'VK_KEY_Z'
   / 'VK_NUMPAD0' / 'VK_NUMPAD1' / 'VK_NUMPAD2' / 'VK_NUMPAD3' / 'VK_NUMPAD4'
   / 'VK_NUMPAD5' / 'VK_NUMPAD6' / 'VK_NUMPAD7' / 'VK_NUMPAD8' / 'VK_NUMPAD9'
   / 'VK_MULTIPLY'
   / 'VK_ADD'
   / 'VK_SEPARATOR'
   / 'VK_SUBTRACT'
   / 'VK_DECIMAL'
   / 'VK_DIVIDE'
   / 'VK_F1' / 'VK_F2' / 'VK_F3' / 'VK_F4' / 'VK_F5' / 'VK_F6' / 'VK_F7' / 'VK_F8'
   / 'VK_F9' / 'VK_F10' / 'VK_F11' / 'VK_F12'
   / 'VK_LSHIFT'
   / 'VK_RSHIFT'
   / 'VK_LCONTROL' {return 'VK_LCTRL';} / 'VK_LCTRL'
   / 'VK_RCONTROL' {return 'VK_RCTRL';} / 'VK_RCTRL'
   / 'VK_LMENU' {return 'VK_LALT';} / 'VK_LALT'
   / 'VK_RMENU' {return 'VK_RALT';} / 'VK_RALT'
   / 'VK_OEM_1' {return 'VK_COLON';} / 'VK_COLON'
   / 'VK_OEM_PLUS'
   / 'VK_OEM_COMMA'
   / 'VK_OEM_MINUS'
   / 'VK_OEM_PERIOD'
   / 'VK_OEM_2' {return 'VK_QUESTION';} / 'VK_QUESTION'
   / 'VK_OEM_3' {return 'VK_CFLEX';} / 'VK_CFLEX'
```

```
    / 'VK_OEM_4' {return 'VK_LBRACKET';} / 'VK_LBRACKET'
    / 'VK_OEM_5' {return 'VK_BACKSLASH';} / 'VK_BACKSLASH'
    / 'VK_OEM_6' {return 'VK_RBRACKET';} / 'VK_RBRACKET'
    / 'VK_OEM_7' {return 'VK_QUOTE';} / 'VK_QUOTE'
    / 'VK_OEM_8' {return 'VK_EXCM';} / 'VK_EXCM'
    / 'VK_OEM_102' {return 'VK_LESSTHEN';} / 'VK_LESSTHEN'
    / 'VK_BACK'
```

Sample Code

The following code demonstrates how to load and run the two parsers:

```
//Substitute with code read from files, embedded manually, etc.
var preParserSrc = '/* The source to the pre-parser goes here. */'
var primaryParserSrc = '/* The source to the primary parser goes here. */'
var keymagicScriptSrc = '/* The source of the Key Magic file you're parsing */'

//Create an array to hold each option
options = new Array();

//Create the pre-parser, run it, and save the output
var preParser = PEG.buildParser(preParserSrc);
var keymagicPreparsedSrc = preParser.parse(keymagicScriptSrc);

//Create one array each for variables, rules and switches.
variables = new Array();
rules = new Array();
switches = new Array();

//Create the primary parser, run it, and discard the output
var primaryParser = PEG.buildParser(primaryParserSrc);
primaryParser.parse(keymagicPreparsedSrc);
```

Appendix B: State Machine Pseudo-Language

This section defines the pseudo-language used in the state machine diagrams. This language is intended to be as simple as possible, and to provide absolutely no surprises to programmers of imperative, object-oriented languages like C++, Java, Ruby, and Javascript. It should be relatively easy to implement this pseudo-syntax using these or any other modern programming language.

A note on naming: UDTs named in lowercase (e.g., **array**) are considered generic enough to be re-used in other specifications. Those written with the first letter of each word capitalized (e.g., **VirtKey**) are specific to Key Magic. Those surrounded by angle brackets and completely capitalized (e.g., **<VKEY>**) denote run-time variables used by the state machines to transform user input.

Variables, Integers, Booleans, and Strings

```
var1 = 123
var2 = 0x80
var2 = "abc"
var3 = copy(var2)
var3 = true       //var2 is still "abc"
```

Variables are made of letters and numbers, and do not need to be declared before they are first used. They are either integers, strings, or user-defined types; this tying information is assigned dynamically (at run-time). Integers are defined using decimal or hexadecimal syntax. Strings are enclosed in single or double quotes. Various escape sequences are allowed. A variable can change its type if its value is re-assigned. Variables are generally assigned by reference;

the function "copy" performs a deep copy of one variable and returns that. The boolean values **true** and **false** are sometimes returned by functions. There is no implicit casting.

The null type

```
var1 = null
var1 + 2     //Error: can't use null
var2 = var1  //Ok; var2 == null
var1 == null //Ok to check against null
var3 = "123".append('456') //Ok; "append" implicitly returns null
```

There is a null type, called **null**. If a function does not explicitly return anything, the return type is null. Any operation on the null type (except setting a variable equal to null, or checking for equality to null) raises an error.

Equality and Looping

```
x = 'hi'
y = 'there'
x == y     //Returns false
x = 2
x == y     //Error
while (x<4) { x++ }  //x will be 4 after this
for (i=0; i<x; i++) { y += '1' }  //Loop from 0 to 4, y will contain 'there1111'
```

The equality/inequality operators **==** and **!=** can be used between two variables both of type string or integer, or between one variable and **null** to check if that variable's value is null. There is no way to compare heterogeneous types or user-defined types.

Looping can be performed with **while** loops, which execute the loop body as long as the conditional remains true. Only boolean values and **null** are allowed in the conditional; **null** returns false, obviously. The second loop construct is the **for** loop, which contains additional initialization and iteration clauses. Semicolons are used to separate these clauses. Note that, stylistically, the bodies of single-line loops are moved up to the same line as the loop declaration.

Normal logical operators like **&&**, **||**, and **!** are supported. The **++** operator is never used ambiguously.

String Functions

```
x = "123"
x.len     //Returns 3
z = x + '4' //z == "1234"
z = x + 5   //z == "1235"
x.len = 6
x         //Returns "123\0\0\0"
x[0]      //Returns '1'
x[0].ord  //Returns 49
49.chr    //Returns '1'
```

Strings have a built-in **len** parameter which returns the length of the string. Setting **len** will expand the string using the default character of '\0'. Array-access syntax for strings returns the character at that index (numbering from zero) in a single-character string. Strings of **len==1** have access to the "ord" read-only property, which returns the Unicode value of that letter. The inverse function to **ord** is **chr**, which returns a single-letter string with the first letter equal (in Unicode value) to the integer **chr** is applied to. Adding two strings will produce a new string with their combined value. Adding a number to a string will append the Unicode value for that number.

User Defined Types (UDTs)

```
person(n) {
  name = n
  age=0
  height=0
  weight=0
  str() {
    return name + ': ' + age
  }
}
j = person('John')
j.age = 20
j.str()  //Returns "John: 20"
```

User-defined types function much like classes in object-oriented languages. Their declaration may use "initializer" values, much like a single constructor. Each variable declared in the top-level scope of a UDT is accessible using the "dot" syntax. Functions are also listed, using parentheses to enclose parameters. A variable is initialized to a UDT by specifying the name of that UDT along with any initializer values. Properties and functions of that UDT are accessed using a dot syntax.

Scope

```
x = 'hi'
for (x=0; x<10; x++) {
  y = 'Hello ' + x   //"Hello 0", "Hello 1", etc.
}
x   //x is still 'hi'
y   //Error; y doesn't exist anymore
```

Brackets denote scope. The definition of a "for" loop is actually part of the scope inside the loop, so temporary variables declared for the purpose of looping are not saved. Temporary variables obscure (but do not overwrite) higher-level-scoped variables until they leave scope. Variables in the global scope are never removed.

Lambda Functions

```
myvar = 'hi'
x = lambda(x,y) {
   z = myvar    //Just an example; the encapsulating scope is visible.
   return x + y
}
x(1,2)   //Returns 3
x(15)    //Error, function requires exactly 2 variables
x(1,2,3) //Also an error.
myfunc(x, y, z) {   //Syntactic sugar for myfunc = lambda(x, y, z) { ... }
  x = y + z
}
```

A lambda function is simply a reference to an un-named function. Lambda functions are only used to represent state machines (the alternative was to use inheritance), and their use should generally be avoided except in these cases. Assigning a lambda function to a variable and then calling that variable will execute that function's body on any given arguments. It is assumed that all variables visible to the declaration of the lambda function are visible to its body.

Note that all functions, whether inside UDTs or visible within the global scope, are actually declared as lambda functions. This allows us to avoid name conflicts; so in the example above, setting "myfunc" to an integer value like "3" would remove the ability to call it as a function.

Some programming languages may not allow you to reference variables from outside a lambda function's body. In that case, you can mimic closure support by making a UDT, like so:

```
//Defining a custom closure for a 3-argument function
Closure(fnc, ctxt) {
  function = fnc
  context = ctxt

  call(x, y) {
    context2 = copy(context)
    context2['x'] = x
    context2['y'] = y
    function(context2)
  }
}

//Sample data
my_var = 'hello'
x = 'testing'
z = ', hello!'

//Make a lambda function; simulate closures
ctxt = []
ctxt['my_var'] = my_var
ctxt['x'] = x
ctxt['z'] = z
func = lambda(context) {
  return context['x'] + context['z']
}
my_lambda = Closure(func, ctxt)

//Call it
my_lambda.call('John', 10)  //Returns "John, hello!"
```

Of course, this is a bit messy. A cleaner way to handle closures if your language doesn't provide built-in support is to simply modify each lambda function in the sample source code to pass in ***all*** the variables required for a computation.

String UDT Functions

```
string(v) {
  val = v

  //Does a string contain a given substring at its beginning?
  starts_with(str) {
    if (str.len>val.len) { return false }
    for (i=0; i<str.len; i++) {
      if (str[i]!=val[i]) { return false }
    }
    return true
  }

  //Add a second string to the end of this string
  append(str) {
    val.len = val.len + str.len
    for (i=0; i<str.len; i++) {
      val[val.len-str.len+i] = str[i]
    }
  }

  //Reverse this string and return a copy
  reverse() {
    res = copy(val)
    for (i=0; i<val.len; i++) {
      res[res.len-i-1] = val[i]
    }
    return res
  }

  //Return a substring, from start to end-1
```

```
    substr(start, end) {
      ensure(start>=0 && end<=val.len && end>start)
      res = copy(val)
      for (i=start; i<end; i++) {
        res[i-start] = val[i]
      }
      return res
    }

    //Return the upper-case value of a single-letter string, or the letter itself
    upper() {
      ensure(val.len==1)
      if (val[0].ord<'a'.ord || val[0].ord>'z'.ord) { return val[0] }
      return ((val[0].ord-'a'.ord) + 'A'.ord).chr
    }
  }
```

Besides their intrinsic functions, strings can be thought of as having several UDT-like functions. These include functions which append data to this string, reverse it, check if it starts with a given sequence, and so forth. The function **ensure()** will raise an error if its condition is false; it is used for quality control an might be implemented using exceptions or error codes. The function **upper** is used to return the upper-case equivalent of a single-letter string. It only affects the letters 'a' through 'z'; more complex Unicode letters are ignored (for now).

Arrays

```
  x = [1,2,3]
  x.len     //Returns 3
  x = ["hi", "hello", "hi"]
  x.len    //Returns 3
  x.len = 5
  x         //Returns ["hi", "hello", "hi", null, null]
  [1,2,3].len = 4    //[1,2,3,null]
  x[1]      //Returns "hello"
```

Arrays contain strings, integers, or UDTs. They may contain other arrays (although this never happens in Key Magic). Arrays have a "len" property, and can be indexed. In this way, they are exactly like strings. Increasing an array's length will fill the remaining entries with **null**.

Array UDT Functions

```
  array (v) {
    val = v,

    //Add an element to an array
    push(item) {
      val.len = val.len+1
      val[val.len-1] = item
    }

    //Remove the last element of the array
    pop() {
      item = val[val.len-1]
      val.len = val.len-1
      return item
    }

    //Reverse this array and return a copy
    reverse() {
      res = copy(val)
      for (i=0; i<val.len; i++) {
        res[res.len-i-1] = val[i]
      }
      return res
    }
  }
```

Arrays contain many similar functions to strings. For example, **push** will add an item, similar to a string's **append** method. However, **push** will only add a single item.

Maps

```
  x = []
  x['key'] = 'value'
  x['key2'] = 'value2'
  x['key3'] = 'value3'
  x.size      //Returns 3
  x.size = 5  //Error, can't set size
  x['key']    //Returns 'value', or true if used in a conditional
  x['key4']   //Returns null, or false if used in a conditional
  if (x['some_key']) { return 10 }  //Example of checking a key in a conditional
```

Maps, which are called **hashes** or **dictionaries** (or even **objects**) in other languages, are simply containers indexed by a string value. They have no **len** property; instead, they have a **size** which returns the number of pairs in the map. Array syntax is used for retrieving a key, and this syntax (wrapped with an "if") can also be used for checking if a key exists in the map.

Maps are extremely simple data containers, and contain no UDT-like methods.

Looping with "in"

```
  x = []
  x.push(1)
  x.push(2)
  x.push(3)
  for (y in x) { //Will change 'x' to [2,3,4]
    y++
  }
  x = []
  x['hi'] = 1
  x['there'] = 2
  for (y in x) { //Will change 'x' to ["hi":2, "there":3]
    y++
  }
```

In order to keep our syntax under control, we introduce the keyword "in", which allows us to iterate (by reference) over elements in an array or map.

VirtKey

```
  //Helper class & array
  Sym(v,s,r) = {
    vk     = v
    shift  = s
    result = r
  }
 symbols = []
```

```
//Shifted numbers
symbols.push(Sym(VK_KEY_1, true, '!'))
symbols.push(Sym(VK_KEY_2, true, '@'))
symbols.push(Sym(VK_KEY_3, true, '#'))
symbols.push(Sym(VK_KEY_4, true, '$'))
symbols.push(Sym(VK_KEY_5, true, '%'))
symbols.push(Sym(VK_KEY_6, true, '^'))
symbols.push(Sym(VK_KEY_7, true, '&'))
symbols.push(Sym(VK_KEY_8, true, '*'))
symbols.push(Sym(VK_KEY_9, true, '('))
symbols.push(Sym(VK_KEY_0, true, ')'))

//Misc. Symbols (by pairs)
symbols.push(Sym(VK_CFLEX, false, '`'))
symbols.push(Sym(VK_CFLEX, true, '~'))
symbols.push(Sym(VK_OEM_MINUS, false, '-'))
symbols.push(Sym(VK_OEM_MINUS, true, '_'))
symbols.push(Sym(VK_OEM_PLUS, false, '='))
symbols.push(Sym(VK_OEM_PLUS, true, '+'))
symbols.push(Sym(VK_LBRACKET, false, '['))
symbols.push(Sym(VK_LBRACKET, true, '{'))
symbols.push(Sym(VK_RBRACKET, false, ']'))
symbols.push(Sym(VK_RBRACKET, true, '}'))
symbols.push(Sym(VK_BACKSLASH, false, '\\'))
symbols.push(Sym(VK_BACKSLASH, true, '|'))
symbols.push(Sym(VK_COLON, false, ';'))
symbols.push(Sym(VK_COLON, true, ':'))
symbols.push(Sym(VK_OEM_COMMA, false, ','))
symbols.push(Sym(VK_OEM_COMMA, true, '<'))
symbols.push(Sym(VK_OEM_PERIOD, false, '.'))
symbols.push(Sym(VK_OEM_PERIOD, true, '>'))
symbols.push(Sym(VK_QUESTION, false, '/'))
symbols.push(Sym(VK_QUESTION, true, '?'))

VirtKey (mCtrl, mAlt, mShift, vK) {
  modCtrl  = mCtrl
  modAlt   = mAlt
  modShift = mShift
  alphanum = getANum(vk, mShift)
  vkCode   = vk

  //Compare 2 key presses. Note that modifiers are only compared if ON in the
  //  "stronger" (v2) key.
  matches(v2) {
    mods_match = (!v2.modCtrl||modCtrl) &&
                 (!v2.modAlt||modShift) && (!v2.modAlt||modShift)
    return mods_match && (vkCode==v2.vkCode)
  }

  //Get the "alphanum" value given a vkCode
  //This function is equivalent to "just look at your keyboard" for en_US users.
  getANum(code, shift) {
    //Letters, [a-zA-Z]
    if (code>='a'.ord && code<='z'.ord)
      return code.chr
    if (code>='A'.ord && code<='Z'.ord)
      return ((code-'A'.ord)+'a'.ord).chr
    //Numbers, [0-9,VK_NUMPAD0-VK_NUMPAD9]
    if (!shift && code>='0'.ord && code<='9'.ord)
      return code.chr
    if (code>=VK_NUMPAD0 && code<=VK_NUMPAD9
      return ((code-VK_NUMPAD0)+'0'.ord).chr

    //Various other symbols
    for (i=0; i<symbols .len; i++) {
      if (symbols[i].shift==shift && symbols[i].vk==code) { return symbols[i].result }
    }
    return '\0'
  }
}
```

VirtKeys represent the virtual keypresses of a computer keyboard. The main purpose of this UDT is to provide a function for comparing 2 UDTs. A secondary purpose is to provide an easy-to-read "alphanum" property, which represents the visual appearance of this virtual key. This property is '\0' for any keys with no visual representation (like Backspace). Note that an alphanum is never capitalized, but symbols will be interpreted by their shifted counterparts; the point is to allow easy comparison of what the user actually typed.

The VirtKey UDT uses a helper UDT called "Sym". This is not used anywhere else, and can be marked "private" in languages which allow restricted control. Similarly, the "symbols" array may be marked private. (In general, deciding what to make private versus public will be up to the implementer to decide.)

Key Magic Primitives

```
  //A primitive takes our "General Form" and converts it to a simple UDT.
  Prim(token) = {
    type  = get_type(token)
    value = get_value(token)  //"Value" may be a string or a VirtKey
    id    = get_id(token)

    get_type(token) {
      if (token==STRING(value))
        return "STRING"
      if (token==VIRT_KEY(modCtrl, modAlt, modShift, vkCode))
        return "VIRT_KEY"
      if (token==VAR_ELEMENT(name, id))
        return "VAR_ELEMENT"
      if (token==VARIABLE(name))
        return "VARIABLE"
      if (token==SWITCH(name))
        return "SWITCH"
      if (token==WILD())
        return "WILD"
      if (token==BACKREF(id))
        return "BACKREF"
      if (token==WILD_VAR_ALL(name))
        return "WILD_VAR_ALL"
      if (token==WILD_VAR_NONE(name))
        return "WILD_VAR_NONE"
      if (token==WILD_BACKREF(name, id))
        return "WILD_BACKREF"

      ensure(false) //Fail immediately: Unknown token type
    }
    get_value(token) {
      mtyp = get_type(token)
      if (mtyp=="STRING")
        return token.value
      if (mtyp=="VIRT_KEY")
        return VirtKey(token.modCtrl, token.modAlt, token.modShift, token.vkCode)
      if (mtyp=="VAR_ELEMENT" || mtyp=="SWITCH" || mtyp=="WILD_VAR_ALL" ||
          mtyp=="WILD_VAR_NONE" || mtyp=="WILD_BACKREF" || mtyp=="VARIABLE")
        return token.name

      return null //Silently fail
    }

    get_id(token) {
      mtyp = get_type(token)
      if (mtyp=="VAR_ELEMENT" || mtyp=="BACKREF" || mtyp=="WILD_BACKREF")
        return token.id

      return null //Silently fail
    }
  }
```

The Prim UDT serves as the link between the "General Form" that the parser provides and the advanced functionality of Key Magic. Each "General Form" token is represented in green bold text in the previous code segment. The only other place this green bold text appears is in the constructors for Rules and Vars.

Prims contain a type, id, and value. The type is a string describing what type of primitive is being represented. ID and value are used differently depending on the type. ID is always an integer, and can represent the **1** in **$1** or the **2** in **$myvar[$2]**. Value is always a string (except for type **VIRT_KEY**, when it represents a VirtKey) and can represent the **my_sw** in **('my_sw')** or the **myvar** in **$myvar[$2]**. ID and value are not always used.

Key Magic Rules

```
  //Commonly used lambda functions
  always = lambda() { return true }
  nothing = lambda() {}

  make_state(prim, side) {
    //Depending on the primitive type and side, either return null or
    // create a series of states with actions and return the root node
    ensure(side=='lhs' || side=='rhs')
    id = prim.type + ':' + side
    ret = null

    //String, LHS
    if (id=='STRING:lhs') {
      ret = State(nothing)
      ret.add_transition(lambda(){
        return str.starts_with(prim.val.reverse())
      }, State(lambda() {
        groups.push(prim.value.reverse())
        group_ids.push(-1)
        str = str.substr(prim.value.len, str.len)
      }))
    }

    //String, RHS
    if (id=='STRING:rhs') {
```

```
  ret = State(nothing)
  ret.add_transition(always, State(lambda() {
    str.append(prim.value)
  }))
}

//Virtual Key, LHS
if (id=='VIRT_KEY:lhs') {
  ret = State(nothing)
  ret.add_transition(lambda(){
    if (str.len==0 || <VKEY> == null) { return false }
    if (!<VKEY>.matches(prim.val)) { return false }
    return true
  }, State(lambda() {
    groups.push(str[0])
    group_ids.push(-1)
    str = str.substr(1, str.len)
  }))
}

//Variable, RHS
if (id=='VARIABLE:rhs') {
  ret = State(nothing)
  ret.add_transition(always, State(lambda() {
    s = <VARS>[prim.value].simple
    str.append(s)
  }))
}

//Backref, RHS
if (id=='BACKREF:rhs') {
  ret = State(nothing)
  ret.add_transition(always, State(lambda() {
    s = groups[prim.id]
    str.append(s)
  }))
}

//Wildcard, LHS
if (id=='WILDCARD:lhs') {
  ret = State(nothing)
  ret.add_transition(lambda() {
    if (str.len==0) { return false }
    if (str[0]>=0x21 && str[0]<=0x7D)   { return true }
    if (str[0]>=0xFF && str[0]<=0xFFFD) { return true }
    return false
  }, State(lambda() {
    groups.push(str[0])
    group_ids.push(-1)
    str = str.substr(1, str.len)
  }))
}

//Wildcard Variable All, LHS
if (id=='WILDCARD_VAR_ALL:lhs') {
  ret = State(nothing)
  branch = State(nothing)
  ret.add_transition(lambda() {
    return str.len > 0
  }, branch))
  final = State(nothing)
  for (i=0; i<prim.value.len; i++) {
    next = branch.add_transition(lambda() {
      str[0] == prim.value[i]
    }, State(lambda() {
      groups.push(str[0])
      group_ids.push(i)
      str = str.substr(1, str.len)
    })
    next.add_transition(always, final)
  }
}

//Wildcard Variable None, LHS
if (id=='WILDCARD_VAR_NONE:lhs') {
  ret = State(nothing)
  next = State(nothing)
  ret.add_transition(lambda() {
    return str.len > 0
  }, next))
  for (i=0; i<prim.value.len; i++) {
    next = next.add_transition(lambda() {
      str[0] != prim.value[i]
    }, State(nothing)
  }
  next.action =lambda() {
    groups.push(str[0])
    group_ids.push(-1)
    str = str.substr(1, str.len)
  }
}

//Backref ID, RHS
if (id=='BACKREF_ID:rhs') {
  ret = State(nothing)
  ret.add_transition(always, State(lambda() {
    s = <VARS>[prim.value].simple[group_ids[prim.id]]
    str.append(s)
  }))
}

//Switch, LHS
if (id=='SWITCH:lhs') {
  ret = State(nothing)
  ret.add_transition(lambda(){
    return <SWITCHES>[prim.value]
  }, State(lambda() {
    sw_temp.push(prim.value)
  }))
}

//Switch, RHS
if (id=='SWITCH:rhs') {
  ret = State(nothing)
```

```
    ret.add_transition(always, State(lambda() {
      <SWITCHES>[prim.value] = true
    }))
  }

  //Variable, LHS
  if (id=='VARIABLE:lhs') {
    ret = State(nothing)
    ret.add_transition(always, State(lambda() {
      //Change the next transition
      old_transition = next_transition
      plus_one_state = next_transition()
      next_transition = lambda() {
        next_state = <VARS>[prim.value].enter_context(plus_one_state)
        next_transition = old_transition
        return next_state   //Jump into the variable's state machine
      }
    }))
  }

  //Return the item we created, or null if nothing matched
  return ret

}
build_state_tree(token_list, side, reverse) {
  ensure(side=='lhs' || side=='rhs')
  ensure(token_list.len > 0)  //Should already be true from our grammar

  elems = []
  elems.len = token_list.len
  for (i=0; i<token_list.len; i++) {
    t_id = i
    if (reverse) { t_id = token_list.len-i-1 }

    //Translate this token into a primitive and then a state
    elem = make_state((Prim(token_list[t_i])), side)
    if (elem == null) { return null }  //Silently fail
    elems[i] = elem

    //Hook the previous state into this one
    if (i>0) {
      prev = find_final_state(elems[i-1])
      prev.transitions = elem.transitions
    }
  }

  return elems[0]
}
str_diff(old, new) {
  //Find the point at which the new differs from the old
  i = 0
  while (i<old.len && i<new.len && old[i]==new[i]) { i++ }

  //Build up the new string
  res = ''
  for (i=0; i<new.len; i++) {
    res += new[i]
  }
  return res
}
Rule(token) = {
  ensure(token==DEFINE_RULE(lhs, rhs))
  start = combine_halves(
    build_state_tree(token.lhs, 'lhs', true),
    build_state_tree(token.rhs, 'rhs', false)
  )
  VkPress = get_primary_vkey(token.lhs, null, true)

  combine_halves(lhs, rhs) {
    ret = State(nothing)
    ret = ret.add_transition(always, State(lambda() {
      groups.len = 0
      group_ids.len = 0
      sw_temp.len = 0
      src = <TYPED_STR>.reverse()
      isLHS = true
    }))

    ret.add_transition(always, lhs)
    ret = find_final_state(ret)

    ret = ret.add_transition(lambda(){
      return !singleASCII && src.len>=0
    }, State(lambda() {
      src = src.reverse()
      groups = groups.reverse()
      group_ids = group_ids.reverse()
      for (name in sw_temp) {
        <SWITCHES>[name] = false
      }
      isLHS = false
      if (<RULE_VK>!=null) { <VKEY> = null }
    }))

    ret.add_transition(always, rhs)
    ret = find_final_state(ret)

    ret = ret.add_transition(always, State(lambda() {
      diff = str_diff(<TYPED_STR>, str)
      if (diff.len==0 || (diff.len==1 && diff[0].ord>=0x20 && diff[0].ord<=0x7F) {
        singleASCII = true
      }
      <TYPED_STR> = src
    }))
  }

  get_primary_vkey(prims, res, allowed) {
    //Check each primitive in the list
    for (prim in prims) {
      if (prim.type=='VIRT_KEY') {
        ensure(res==null && allowed)  //Only 1 result, and VirtKeys are still allowed
        res = prim.value
      }
      if (prim.type=='VARIABLE') {
        //Needs to be handled recursively
        res = get_primary_vkey(<VARS>[prim.value].rev_states, res, allowed)
      }

      if (prim.type!='VIRT_KEY' && prim.type!='SWITCH' && prim.type!='VARIABLE') {
        //Can't match a VirtKey unless it's at the end of the string
        allowed = false
      }
    }
    return res
  }
}
```

Rules are defined as a sequence of states. A single primitive may generate more than one state, and several states are added as scaffolding to drive the engine. The function **combine_halves** is used to create the rule's finished state machine while **get_primary_vkey** ensures that virtual keys are not used improperly.

The **<RULES>** array is created by calling **Rule(token)** on every **DEFINE_RULE** token produced by the parser. This must be done after creating the **<VARIABLES>** array, described in the next section.

Key Magic Variables

```
make_simple(token_list) = {
  ensure(token_list.len > 0)

  res = ''
  for (i=0; i<token_list.len; i++) {
    p = Prim(token_list[i])

    //Invalid types cause the entire function to silently fail
    if (p.type=='SWITCH' || p.type=='WILD' || p.type=='BACKREF' ||
        p.type=='WILD_VAR_ALL' || p.type=='WILD_VAR_NONE' || p.type=='WILD_BACKREF')
      return null

    //Valid types
    if (p.type=='STRING')
      res += p.value
    if (p.type=='VIRT_KEY') {
      val = p.value.getANum()
      if (val == '\0')
        return null
      res += val
    }
    if (p.type=='VAR_ELEMENT') {
      val = make_simple(<VARS>[p.value])
      if (val == null)
        return null
      res += val[p.id]
    }

    ensure(false) //Shouldn't ever reach this
  }
}
Var(token) = {
  ensure(token==DEFINE_VARIABLE(name, items))
  name = token.name
  rev_states = build_state_tree(token.items, 'lhs', true)
  simple = make_simple(token.items)
  ret_stack = []

  //Need to add a special "end" state for variables
  last = find_final_state(rev_states)
  last.add_transition(always, State(lambda() {
      <STATE> = ret_stack.pop()
      <STATE>.action() //Perform this state's action for it.
    }))

  //Allow context switching
  enter_context(next_state) {
    ret_stack.push(next_state)
    return rev_states
  }
  reset() {
    ret_stack.len = 0   //In case a previous match aborted halfway through
  }
}
```

Much of the complexity of variables is already handled in the section on Rules. The ability to make a "simple string" out of a variable's token list is required for when that variable is used on the RHS of a Rule.

The **<VARS>** array is created by calling **Var(token)** on every **DEFINE_VARIABLE** token produced by the parser. The order that Vars are defined in matters; if **$test** is defiend before **$test2**, then **$test** cannot reference **$test2**.

State Machine States

```
find_final_state(s) {
  //Any transition should eventually lead to the final state regardless of the option
  if (s.transitions.len==0)
    return s
  return find_final_state(s.transitions[0].s)
}
Trans(t, s) {
  test  = t
  state = s
}
State(toPerform) = {
  action = toPerform
  transitions = []

  add_transition(test, state) {
    transitions.push(Trans(test, state))
    return state
  }

  //Attempt to move to the next state
  next_transition() {
    for (i=0; i<transitions.len; i++) {
      if (transitions[i].t()) {
        return transitions[i].s
      }
    }
    return null
  }

  is_end_state() {
    return transitions.len == 0
    }
  }
```

A State contains two variables. The first is a list of allowed transitions. If the **test** condition of a Trans UDT returns true, the **state** can be transitioned to. The second is a lambda function to perform when that State is transitioned to.

Rule Matching Algorithm

```
  //Required initialization:
  <TYPED_STR> = /* The string that the user has typed so far */
  <VKEY> = /* The VirtKey that the user just pressed */
  <MATCH_MAX> = /* Config option: maximum number of matches allowed */

  //Helper function
  do_matches(rules_list, match_amount) {
    matched = true
    singleASCII = false
    while (matched && !singleASCII) {
      matched = false
      for (i=0; i<rules_list.len && !matched; i++) {
        for (v in <VARS>) { v.reset() }
        candidate = rules_list[i]
        if (perform(candidate)) { matched=true }
      }
    }
  }

  //Helper function
  perform(rule) {
    <STATE> = rule.start
    <RULE_VK> = rule.VkPress
    while(<STATE> != null) {
      <STATE>.action()
      if (<STATE>.is_end_state()) { return true }
      <STATE> = <STATE>.transition()
    }
    return false
  }

  //Here is our matching logic:
  do_matches(<RULES>, <MATCH_MAX>)

  //Matching logic for 2.0 syntax:
  do_matches(<PRESSES>, 1)
  do_matches(<RULES>, <MATCH_MAX>)
```

The matching algorithm takes as input some of the bracketed variables (like **<VKEY>**), generates others (like **<RULE_VK>**), and performs a series of Rule matches.

Appendix C: Complete Sate Machine Syntax

(Complete syntax for the state machines. Will probably use Javascript for this.)

Appendix D: Complete List of Virtual Key Codes

The complete list of virtual key codes is as follows. Any **KEYCODE_VALUE** with multiple **KEY_NAME**s has exactly ***one*** "canonical" name; this name will be listed first in regular type, while any non-canonical (but equivalent) name is listed after in italic type. Be aware that some of these keys may not actually be type-able on an en_US keyboard. For example: *VK_OEM_8* and *VK_OEM_102*.

NOTE: The ***order*** of virtual key definitions matters, since a PEG will stop matching at the first valid alternative. Hence, *VK_BACKSLASH* should be listed before *VK_BACK* in the PEG. Please see Appendix A for a detailed ordering.

Virtual Key Codes and their Names

KEY_NAME	**KEYCODE_VALUE**	**Description**
VK_SHIFT	0x0010	SHIFT key *May also function as a modifier.*
VK_CTRL *VK_CONTROL*	0x0011	CTRL key *May also function as a modifier.*
VK_ALT *VK_MENU*	0x0012	ALT key *May also function as a modifier.*
VK_BACK	0x0008	BACKSPACE key
VK_TAB	0x0009	TAB key
VK_ENTER *VK_RETURN*	0x000D	ENTER key
VK_PAUSE	0x0013	PAUSE key
VK_CAPSLOCK *VK_CAPITAL*	0x0014	Caps Lock key
VK_KANJI	0x0019	Japanese kanji key
VK_ESCAPE	0x001B	Esc key
VK_SPACE	0x0020	The spacebar
VK_PRIOR	0x0021	The Pg Up key
VK_NEXT	0x0022	The Pg Down key
VK_DELETE	0x002E	The Del/Delete key
VK_KEY_0	0x0030	The number keys, 0-9
VK_KEY_1	0x0031	
VK_KEY_2	0x0032	
VK_KEY_3	0x0033	
VK_KEY_4	0x0034	
VK_KEY_5	0x0035	
VK_KEY_6	0x0036	
VK_KEY_7	0x0037	
VK_KEY_8	0x0038	
VK_KEY_9	0x0039	
VK_KEY_A	0x0041	The letter keys, A-Z
VK_KEY_B	0x0042	
VK_KEY_C	0x0043	
VK_KEY_D	0x0044	
VK_KEY_E	0x0045	
VK_KEY_F	0x0046	

VK_KEY_G	0x0047	
VK_KEY_H	0x0048	
VK_KEY_I	0x0049	
VK_KEY_J	0x004A	
VK_KEY_K	0x004B	
VK_KEY_L	0x004C	
VK_KEY_M	0x004D	
VK_KEY_N	0x004E	
VK_KEY_O	0x004F	
VK_KEY_P	0x0050	
VK_KEY_Q	0x0051	
VK_KEY_R	0x0052	
VK_KEY_S	0x0053	
VK_KEY_T	0x0054	
VK_KEY_U	0x0055	
VK_KEY_V	0x0056	
VK_KEY_W	0x0057	
VK_KEY_X	0x0058	
VK_KEY_Y	0x0059	
VK_KEY_Z	0x005A	
VK_NUMPAD0	0x0060	Numeric keypad, 0-9
VK_NUMPAD1	0x0061	
VK_NUMPAD2	0x0062	
VK_NUMPAD3	0x0063	
VK_NUMPAD4	0x0064	
VK_NUMPAD5	0x0065	
VK_NUMPAD6	0x0066	
VK_NUMPAD7	0x0067	
VK_NUMPAD8	0x0068	
VK_NUMPAD9	0x0069	
VK_MULTIPLY	0x006A	The multiply key, *
VK_ADD	0x006B	The add key, +
VK_SEPARATOR	0x006C	The separator key, (*"Enter" on the NumPad?*)
VK_SUBTRACT	0x006D	The subtact key, -
VK_DECIMAL	0x006E	The deicmal key, (*"." on the NumPad?*)
VK_DIVIDE	0x006F	The divide key, \
VK_F1	0x0070	The function keys, F1-F12
VK_F2	0x0071	
VK_F3	0x0072	

VK_F4	0x0073	
VK_F5	0x0074	
VK_F6	0x0075	
VK_F7	0x0076	
VK_F8	0x0077	
VK_F9	0x0078	
VK_F10	0x0079	
VK_F11	0x007A	
VK_F12	0x007B	
VK_LSHIFT	0x00A0	The left Shift key
VK_RSHIFT	0x00A1	The right Shift key
VK_LCTRL *VK_LCONTROL*	0x00A2	The left Ctrl key
VK_RCTRL *VK_RCONTROL*	0x00A3	The right Ctrl key
VK_LALT *VK_LMENU*	0x00A4	The left Alt key
VK_RALT *VK_RMENU*	0x00A5	The right Alt key
VK_COLON *VK_OEM_1*	0x00BA	The colon key, ; :
VK_OEM_PLUS	0x00BB	The plus key, = +
VK_OEM_COMMA	0x00BC	The comma key, , <
VK_OEM_MINUS	0x00BD	The minus key, - _
VK_OEM_PERIOD	0x00BE	The period key, . >
VK_QUESTION *VK_OEM_2*	0x00BF	The question key, / ?
VK_CFLEX *VK_OEM_3*	0x00C0	The tilde key, ` ~
VK_LBRACKET *VK_OEM_4*	0x00DB	The left bracket key, [{
VK_BACKSLASH *VK_OEM_5*	0x00DC	The backslash key, \ |
VK_RBRACKET *VK_OEM_6*	0x00DD	The right bracket key,] }
VK_QUOTE *VK_OEM_7*	0x00DE	The quote key, ' "
VK_EXCM *VK_OEM_8*	0x00DF	An alternative exclamation mark key, § !
VK_LESSTHEN *VK_OEM_102*	0x00E2	An alternative less-than key, < >