Swil
Senior Project Report

Kris Kowal

8th December 2005

Abstract

Swil is first and foremost a document processing language, but consolidates many features from functional, imperative, aspect oriented, and markup languages. In the sense that a Swil document is a literal value with encompassing tags to denote semantics and behavior, Swil is a document language. In the sense that each line of of a Swil document is executed in order and that those lines can modify the program's state, Swil is an imperative language. In the sense that every function has a value, even control flow functions, Swil is a functional language. In the sense that a Swil document can be evaluated with different assumptions than those from which it was created, it is an aspect oriented language.

This report contains an account of some of the issues I encountered while implementing Swil, an essay about where Swil's syntax comes from, technical descriptions of Swil's syntax and semantics, and a description of functions available in a Swil document.

Contents

1 Implementation

1.1 Indentation

The first module I implemented for Swil was the indentation scanner. Indentation is a tricky problem as evidenced by the quirks of soo few languages that depend on it, not the least being Makefile. One problem with indentation is that it is rather complex to teach a computer how to read a sequence of mixed spaces, tabs, and dark characters as one would read them with the unaided eye.

It is a matter of trivia to know what column number to draw on. More interesting problems arise when you scrape a substring from a line of mixed tabs and other characters. For one, the new string carries with it the implication that if it were drawn again, it would have to be drawn starting at the same column, or an analogous column to another tab stop. Further, if you take a visual substring, there is the possibility that a final tab extends beyond the substring, in which case it really should be truncated and replaced with a number of spaces that are visually equivalent. All this is compounded by the problem that few people can agree on the width of a tab.

For this last problem, I decided to lay down a heavy hand. In Swil, you can either use tabs consistently, or use four space tab stops. This is likely to raise some ire, especially in the big UNIX camp, where many are convinced that God Herself proclaimed that tab stops are eight spaces wide.

Another interesting problem is blank lines. Nested content should implicitly end on the first line on which the first dark space comes before the nested content. However, a blank line and a line entirely containing an assortment of white spaces look exactly the same. Such lines conceptually begin at any indentation depth. So, in Swil, a blank line is treated as a line at the current indentation depth, regardless of the column number at which it ends.

I built the indentation module from the bottom up. I wrote a 'next tab' function which returns the column number that a tab would advance the cursor to from any given column. Since the 'tab length' is a power of two, I was able to use some fun bitwise arithmetic.

```
(column & ~(tab_length - 1)) + tab_length
```

From there, I wrote a function that measures what column a string ends on. The 'shift indent' function uses these functions to chew off one indentation depth of spaces and tabs from the beginning of a string to create a new string that preserves as much of the original as possible, while appearing to begin exactly four spaces to the left of its previous location.

Swil makes quite a bit of use of generator functions. The indentation module's main interface is a 'shift indent iterator', which accepts a 'peekable iteration' of 'tolerable lines', consumes, and yields as many lines from the source as have enough white space to shift a certain number of columns leftward, minding all the issues I've mentioned.

'peekable iterations' and 'tolerable lines' have become core concepts in Swil's code. A peekable iteration is an object that transparently wraps a source iteration, providing the additional ability to 'peek' or look ahead in the iteration any number of elements. 'Tolerable lines' accepts a string (possibly containing new line characters), an iteration of strings, or a peekable iteration of strings and returns a peekable iteration of strings, one line each.

1.2 Milestone Demonstration

The first milestone demonstrated Swil's potential for formatting a document in an aspect oriented fashion to multiple presentations. Using Swil's indentation module and the Reportlab[1] PDF library, three independent Python programs formatted each of two documents of varying length and diversity ("How to Write Your Very Own Bad Fiction", and "The Hobbit") into terminal-width formatted plain text, HTML, and PDF. By tracking how much of the documents' margins had been consumed, the HTML and text formatters were able to write precise and human readable documents, even intelligently determining whether to pivot an HTML tag over multiple lines or to save space by using only one.

[1] Reportlab: http://reportlab.org

The milestone programs heavily used two generator functions, 'major' and 'minor', to functionally defer processing of entire tags and each line of a given tag respectively. These functions handled many formatting cases with specificity. For example, 'major' handled formatting for the section as a whole, and deferred the processing of each line to 'minor'. 'Minor' formatted each line of immediately nested text, and deferred the formatting of each nested block to the appropriate 'major' block handler. Both 'major' and 'minor' tags heavily used indentation and line wrapping functions to format every line of text, whether it was produced by direct formatting or produced by a more specific function.

This demonstrated that simple 'for' loops and iterating on lines would not suffice to intelligently nest tags. For example, an HTML bullet-ed list production produces a 'ul' tag at the top and bottom of its content, an 'li' tag to the left and right of each line of immediate content, no markup around nested lists, and 'li' tags around any non-list nested content. However, an 'indent' tag would need to indent each line of output, regardless of whether the content were produced by a nested tag.

There is an issue that I did not encounter with the milestone programs but still anticipate. HTML 'pre' tags misinterpret indentation that exists for the purpose of style, so a proper HTML formatter would account for this by having any pre-formatted text nested under an indenter to somehow notify the indent tag to leave it alone. These issues remain unresolved in the fully programmatic implementation of Swil.

1.3 Enclosers

Swil uses Egenix mx.TextTools[2], a Python module that harnesses a C runtime library to perform very fast and flexible recursive descent and finite state machine parsing on character strings. TextTools allows a programmer to write parse tables as tuples of tuples in a particular format. The programmer passes their tables to the TextTools tagging engine which produces a tuple parse tree.

Starting the second third of my project, I wrote parsers for the first pass of parsing that Swil always performs. Various Swil parsers scan a string, making sure that all enclosing operators like parentheses and and quotes are 'balanced'. Balanced loosely means that every opening encloser corresponds to a closing encloser[3].

At first, I wrote these parse tables manually, to establish the patterns of the problem. Eventually I wrote a function that would build a parse table for a particular encloser, which would in tern call upon other tables, recursively, if they stumbled upon an encloser with lower precedence.

I had to revisit this module to add versions of the parenthese and square bracket parsers that would only terminate by the matching closer, rather than both accepting closing parentheses and square brackets as terminators. However, these parse tables still scanned "loosely" balanced parentheses and brackets if they were encountered inside parenthesized strings.

1.4 Delimiters

Swil employs arbitrarily low precedence delimiting operators: space, comma, semi-colon, colon, double-comma, double-semi-colon, and so on. When Swil parses a value, it makes a first pass scan for the lowest precedence, unenclosed delimiter in the string. The parser starts scanning by building a table that will search for any space, comma, semi-colon, or colon. TextTools permits you to call a Python function in a parse table if the state machine enters a 'Call' state. Each of the delimiter parsers had one such call state that would raise an exception if it found a lower precedence version of itself. For example, the comma delimiter parse table would raise a LongerDelimiter exception if it found a double-comma. This affords Swil an opportunity to rebuild its delimiter scanner tables to only search for delimiters of lower precedence than the delimiter that has already been found, incrementally. Whatever delimiter raises an exception last gets returned as the lowest delimiter.

From there, I wrote functions that performed the three forms of delimiter splitting that occur in the

[2]Egenix mx.TextTools, http://www.egenix.com

[3]see "Enclosers" on page 13

Swil value parser: left combination, right combination, or delimiting. Delimiting is a typical split, except that it doesn't split on enclosed delimiters. That is, if you split on commas, the string would remain whole across those commas contained in parentheses. The combination forms only perform one split, either on the left-most or right-most occurrence of the delimiter.

1.5 Semantics

About half way through the project, I started working on semantics. I failed several times.

The most monumental failure in developing Swil's semantics involved a rather large module that contained a class for every abstract syntax node that Swil conceptually contained: names, rows, columns, row numbers, column numbers, row ranges, number ranges, data nodes, application, invocation, functions, and the like. Each of these nodes supported an evaluate and an apply function. Each of these classes was equipped with logic to permit it to apply itself to another node in a sort of 'visitor clique'.

The approach consumed quite a bit of energy and time and never quite completely failed to work. I had to scrap the whole module when I discovered that I wouldn't be able to perform parsing independently from evaluation. Swil functions don't necessarily parse their body or arguments as if they are Swil. I got a break when I started writing a parser/evaluator generator framework.

1.6 Parser Generator

I wrote two modules, 'parser' and 'evaluator', on the premise that they would make it easier to develop compound parsers. The idea was to create a wrapper class for the mx.TextTools parser which could be 'combined' to create new parsers. A number parser could be combined with a comma delimiter parser to build a coordinate parser.

I created a 'table parser' class, and some related utilities. With the framework, you can build compound parsers with the plus, pipe, modulo, and index operators. Adding two parsers effectively concatenates them, so that if you successfully evaluate both parsers, back to back, you get a tuple of both of their contents. Using a pipe creates a choice parser; you successfully evaluate a string with the first parser that succeeds, and you get its value. Any parser modulo a delimiter parser produces a tuple of as many terms pass the left parser with the delimiter between. Indexing a parser on a string names that parser and changes the compound parser's result to a dictionary instead of a tuple.

I also provided a function decorator named 'evaluator', which would permit you to create a parser/evaluator and associate it with a function which would evaluate the resulting AST. The tuple or dictionary result of the associated parser would be passed as variadic keywords and arguments to the function. Later, I augmented the parser base class so that you could associate any number of compound parsers with the same value function by indexing them all with a function reference instead of a string.

This is an example of a number, name, and variable parser.

```
@evaluator(table_parser(
    (None, IsIn, '123456789',
     MatchFail),
    (None, AllIn, '0123456789',
     MatchOk, MatchOk),
))
def number(context, text):
    return int(text)

name = table_parser(
    (None, IsIn, 'abcdxyzw',
     MatchFail, MatchOk),
)

@evaluator(name)
def variable(context, name):
    return context[name]
```

The project was only marred when I discovered that my test cases were failing because of a bug in mx.TextTools. The problem was that the TextTools tagging engine reports success unconditionally if the engine halts on an end of file checking state. This table should fail unconditionally. The elements of

the single table entry respectively mean that: if this state succeeds, it won't create an AST node; if the string ends at the current location, fail; if the string does not end at the current location, fail.

```
table = (
    (None, EOF, Here, MatchFail, MatchFail),
)
```

I resolved the issue by writing a work-around in the 'table parser' class that would look for ParseError exception objects in the left-most column of the parse tables I defined. I also sent the TextTools developers an email with a detailed bug report.

The Swil parser generator framework has gotten a lot of mileage. It has been indispensable for any pattern where parsing drives function calls.

2 About Syntax

SGML and XML are abstract syntaxes. Both languages burn a lot of characters on meta-data. Swil divines its own abstract syntax from XML's. SGML stands for Standardized General Markup Language. XML is Extensible Markup Language.

> Conceived notionally in the 1960s - 1970s, the Standard Generalized Markup Language (SGML, ISO 8879:1986) gave birth to a profile/subset called the Extensible Markup Language (XML), published as a W3C Recommendation in 1998. Depending upon your perspective and requirements, the differences between SGML and XML are inconsequential or immense. SGML is more customizable (thus flexible and more "powerful" at the expense of being (much) more expensive to implement. In an SGML language you could say <COVER PAGES>, whereas in XML this construct could not be DTD valid. [SGML][4]

Summarily, SGML is a very general language. Other languages can express data for specific utilities within the SGML syntax and use a common SGML interpreter. HTML is a kind of SGML. XML is similar to SGML but far more specific. XHTML is a kind of XML. The distinctions of XML and SGML are unimportant in the context of this discussion.

XML, XSL (Style Sheets), XML Schema, DTD (Document Type Definitions) and a host of others are a collection of tools that simplify the task of creating quick languages and harnessing common libraries to interpret them. Effectively, this engenders decoupling between language programmers and application programmers. However, XML wastes a lot of real estate.

Consider SGML or XML. A typical XML document expresses a hierarchy.

```
<tree>
    <leaf>1</leaf>
    <leaf>2</leaf>
    <tree>
        <leaf>3</leaf>
        <leaf>4</leaf>
    </tree>
    <acorn>5</acorn>
</tree>
```

Consider a trace through this document. As the trace encounters an opening tag, like <**tree**>, put that name on top of a stack. When you encounter a closing tag, like </**acorn**>, remove a name from the top of the stack. Every name you take off the stack will be the same as the name of the closing tag. For documents where this occurs exclusively, the name of a closing tag infers nothing. That a tag closes at all implies the name of the terminated tag. Thus, a more brief form of XML would have "anonymous

[4] "What Every Web Developer Should Know" Alan Richmond, Web Developers' Virtual Library, http://wdvl.com

closing tags."

```
<tree>
    <leaf>1</>
    <leaf>2</>
    <tree>
        <leaf>3</>
        <leaf>4</>
    </>
    <acorn>5</>
</>
```

However, XML requires that closing tags have names with a good reason. An XML document is free to conceptually contain multiple parallel stacks. For instance, in HTML, there are separate stacks for implying formatting attributes: font family, bold, italic, underline, and others. Thus an HTML document can express data like:

```
Normal
<b>Bold
<i>Bold and Italic
</b>Italic
</i>
Normal
```

With such an arrangement, the name of each closing tag does not conveniently correspond to the name of the tag on the top of the stack. However, for every asymmetric XML document, there exists a symmetric equivalent.

```
Normal
<b>Bold</>
<b><i>Bold and Italic</></>
<i>Italic</>
Normal
```

Symmetric documents are easier to modify than their asymmetric equivalents. For example, in the asymmetric document, moving the line that contains the word 'Bold' one line down would cause it to inherit the italic attribute from the previous line. With the symmetric document, moving bold down a line would cause no changes in the relationship between formatting and the text.[5]

Although symmetric documents are more common and less resistant to editing, a language could arrive at a suitable compromise. This language could support both named and anonymous closing tags. Anonymous tags would pop the top element of the trace stack. Named tags would remove the topmost occurrence of their name from the trace stack.

```
Normal
<b>Bold</>
<b><i>Bold and Italic</b>
Italic</>
Normal
```

A more radical language might opt to abandon closing tags entirely, since the document's indentation effectively implies its nesting.

```
<tree>
    <leaf>1
    <leaf>2
    <tree>
        <leaf>3
        <leaf>4
    <acorn>5
```

XML ignores indentation. This permits XML authors to write documents that are visually difficult to understand.

```
<tree><leaf>1</><leaf>2</>
<tree><leaf>3</><leaf>4</></><acorn>5
</></>
```

XML leaves authors to battle with their own high moral standards over whether to write with "good style", where documents in "good style" have indentation that implies the container of each element. Furthermore, "good style" becomes a matter of debate among entrenched authors.

[5] Patrick Dursau and Matthew Brook O'Donnell offer an alternate opinion on overlapping hierarchies in their paper and presentation, "Just-In-Time-Trees (JITTs): Next Step in the Evolution of Markup?" at http://www.sbl-site2.org/Extreme2002/JITTs.html.

In HTML, the pre tag, meaning "pre-formatted text", derails most notions of "good style". In the pre-formatted tag, all white space suddenly bears meaning. Thus, a document with strongly meaningful indentation can be incorrect. As in this case, the formatted text has a new line and twelve spaces more text than the author likely desires.

```
<p>My List:</p>
<ul>
    <li>My Item:
        <pre>
            This is Pre-formatted
        </pre>
    </li>
</ul>
```

To correct the document, the author must break their style rule.

```
<p>My List:</p>
<ul>
    <li>My Item:
        <pre>
This is Pre-formatted</pre>
    </li>
</ul>
```

In a language that values indentation, not only do the unnecessary tags disappear, but so do problems with indentation.

```
<p>My List:
<ul>
    <li>My Item:
<pre>
    This is Pre-formatted
```

In defense of XML, using indentation poses different problems. Parsing indentation in a reasonable way is a difficult problem to solve programmatic-ally. Languages that have dared to use indentation have caused many authors agony. The Makefile format, for instance, uses a single tab to denote nesting. However, it requires exactly one tab, where many permutations of tabs and spaces are visually equivalent to a single tab. The Python language uses indentation to denote nesting in program blocks. Python's indentation interpreter can get confused when it encounters a line indented by spaces followed by a line indented by tabs. However, these are accidental problems. Given that the distinction between XML and an indentation intelligent language is difficulty, not possibility, compiler implementers should consider the weight of their own sloth against the sloth of their users. Language programmers, fortunately, are their own primary user group.

A programming language that uses indentation to imply nesting depends on line breaks. Our simplified version of XML gains two more possible simplifications if it becomes a line-by-line language. For one, the opening chevron on each line becomes meaningless.

```
tree>
    leaf>1
    leaf>2
    tree>
        leaf>3
        leaf>4
    acorn>5
```

Additionally, we can add a meaning to line breaks. Each fresh line can implicitly be a new node appropriate to the container. For our tree, each node is a leaf, tree, or acorn. For the sake of brevity, consider leaf the default.

```
tree>
    1
    2
    tree>
        3
        4
    acorn>5
```

However, the writers of XML had a good reason for ignoring new lines.

```
<p>With new lines, paragraphs
and other text data can
span multiple lines.</p>
```

In Swil, a line breaks delimit tags. However, Swil leverages indentation and permits text wrapping by

providing functions that unwrap paragraphs and one space hanging indents.

```
With new lines, paragraphs
 and other text data can
 span multiple lines.
```

3 Semantics

3.1 Contexts

A variable in Swil is a named reference to a context. A context contains any number of lines of text, an optional parent context, a dictionary of (name, context) pairs, and an optional function.

Swil does not distinguish the global context from other contexts. The global context is not necessarily the only context that has no parent, and if the user indexes a "root" context's parent, Swil creates a new root and makes the old one an anonymous child.

3.2 Evaluation

Evaluation is an operation that requires a context in which to perform the evaluation, an evaluator, and corresponding text.

Swil has evaluators for all of its syntactic variants: literals, values, value outlines, literal outlines, verbatims; but this list is not exhaustive. The user can extend Swil by providing evaluators for alternate syntaxes and Swil functions for processing them.

Swil does not distinguish execution from evaluation. The value of an expression is a sequence of lines of text. Evaluation can also affect the state of its context. That is, evaluating a Swil expression can change the values of any context accessible from the evaluation's context. Swil evaluates each expression iteratively, yielding its lines of text. Assignment and function declaration affect the context but provide no value. The value produced by an invocation is passed back through the value. Any other expression has a value and is rendered into the expression's cumulative value.

Swil evaluation is similar to wrapping a Python generator in a tuple.

As a negative side effect of assignment not having a value, you can not chain assignments like you can in C and C-like languages.

```
a = b = c = 10;
```

In Swil, the behavior is far more verbose.

```
c>10
b>{c}
a>{b}
```

However, using a Swil idiom, you can express the same behavior in a relatively clear form.

```
for name, ('a'; 'b'; 'c')>
    {name}>10
```

In the atypical event that you're only interested in a function's side effects, you can assign it to the current context, denoted by a single dot, '.', or a dummy variable.

```
.>
    f> argument
/>
```

Despite the similarity to a Python generator function or lazy list, Swil functions are not "lazy". That is, when a function is called, its entire body is consumed at once. Unlike a pipeline of processes, nested functions do not yield their values through a buffer and sleep until their target consumes enough buffer to warrant resuming. Both of these features may be added to the language.

Swil does not have "return", "yield", and "print" keywords, nor does it support the notion of standard input and output. Return and yield are both facilitated by the evaluation semantics. Swil facilitates print by sending a document's value to standard output. Standard input will eventually be supported by the ability to apply a Swil program on another, where the former "template" would explicate its argument list in a shebang ('#!' line in a UNIX shell script) or an XML style '<**!DOCTYPE!**>'.

3.3 Assignment

Assignment updates a variable in the current context, automatically creating a new one if it does not yet exist. Assignment has a target and a source. The target is the name that the new variable will have. The source is a (text, evaluator) pair.

Assignment initially creates a new, empty context. The new context's parent is current context. Swil then uses the source evaluator to evaluate the source text in the new context. This can modify the context's state, possibly populating nested variables, declaring a function, and having other side-effects. Ultimately, the evaluation produces lines of text. These lines become the value of the variable. Finally, Swil updates the variable in the current context to refer to the new context.

This example creates a new context named 'outer', and evaluates the content block with the literal outline evaluator in that context. The lines of 'outer' become 'A' and 'B'. Another assignment occurs in that block with the name 'inner', which produces a nested context within 'outer' that has the lines 'C' and 'D'.

```
outer>
    A
    B
    inner>
        C
        D
    />
/>
```

Swil does not distinguish name spaces and structures from variables. Any Swil context can be used as a list, set, or dictionary.

3.4 Functions

Swil functions are Python callable objects which accept a calling context and a tuple of argument (text, evaluator) pairs. A function may elect the manner in which its arguments are evaluated. They may opt to evaluate them with a parser other than the parser native to their syntax, for example, as a value instead of an interpolated literal. A 'verbatim' function elects not to parse its arguments, simply rendering their textual value. A comment function simply chooses not to evaluate its arguments.

A user defined function object contains a reference to the context in which it was created (the closure, or continuation), any number of argument names, and an evaluatable block (text, evaluator) pair.

3.5 Invocation

Invocation is the act of calling a function.

For functions defined in Swil, invocation produces a new, local context rooted in the context in which the given function was created. The invocation then applies each of its argument names in the local context to the given arguments in the context of the caller. Then, Swil evaluates the function's body text with it's corresponding evaluator.

Swil functions defined in Python, at the moment, have a great deal more flexibility than user defined functions. When invoked, they receive the caller context, and the argument (text, evaluator) pairs. The function is only required to return a tuple of strings, albeit empty.

3.6 Declaration and Combination

Combination occurs when the document applies a value to an application. Combination either declares a function or invokes a function with combined arguments.

Swil searches for the context named by the inner application. If that context has a function, the combination invokes that function with the inner arguments plus the outer argument. If the context does not contain a function, the combination declares one with the inner application's arguments as a formal parameter list, and the outer application's argument text and evaluator as the function's body.

This example of a Swil literal outline illustrates

both cases.

```
maximum x, y>
    if (x <= y)>
        y
    if (x > y)>
        x
maximum 10> 20
```

The first combination of 'maximum' creates a new function and places it in the new sub-context named 'maximum'. The second combination invokes the 'maximum' function with two arguments, 10 and 20. The value of the entire expression is 20, since it is greater or equal to than 10.

Swil does not distinguish named functions from anonymous functions (lambdas). An anonymous function is simply a function that is declared in the current context rather than another name. In this example, we assign an anonymous function that accepts an argument 'x' to the name 'adder', which, when called, introduces an anonymous function that accepts 'y' to the caller's context. We then create a variable 'add-one' which receives a function that adds two to any argument. In the last line, we invoke this function with 3. The value of the entire expression is a single line, '5'.

```
adder>
    .(x)>
        .(y)>
            {x + y}
/>
add-one>{adder 2}
add-one>3
```

3.7 Application

Application in Swil performs assignment, function declaration, function invocation, and argument combination, depending on whether the left value is an application, whether the left value's context exists, and whether there is a function associated with the left value's context.

An innocuous application in Swil can be either an assignment or a function call, depending on the context.

```
author>Ursula K. LeGuin
```

If there is no context called 'author' or there is no function in that context, this line is an assignment. It would create a new context and set it's lines to the single line of text, "Ursula K. LeGuin". If there is a function in the 'author' context, this line would be an invocation of that function with a single argument, a single line, and in turn the verbatim value "Ursula K. LeGuin".

While this scheme makes Swil's maintainability sketchy, it renders the potential of very powerful, aspect oriented functionality. The author can confidently state her name with the 'author' attribute of the document. She can assume that the interpreter will simply create an author variable by that name and either ignore it, place it beneath the title, format it to her liking. This leaves the template maker the option of using 'author' as a variable throughout the presentation of the document, or creating an author function to format the text in place.

In a literal outline[6], application can take the following forms, all using the angle bracket, '>', operator. Application of a literal argument on a value performs either assignment or invocation.

```
value>Argument
```

Application of a literal argument on an application of arguments on a value. This performs either declaration or combination.

```
value(arguments)>Argument
```

Parentheses are optional on application in value space.

```
value arguments>Argument
```

In a value outline[7], application takes on the typical form of a function call. This performs either assign-

[6] see "Syntax: Literal Outline" on the next page

[7] see "Value Outline" on page 12.

ment or invocation. Parentheses are optional unless there are no arguments.

```
value(arguments)
```

```
value arguments
```

Application of an argument on a value is analogous to the first literal outline notation, except using the colon, ':', operator.

```
value: argument
```

Analogous to the second and third literal examples, application of an argument on another application of arguments on a value in value space performs either declaration or combination.

```
value(arguments): argument
```

```
value arguments: argument
```

4 Syntax

Swil's syntax starts as a literal outline, so a line of plain text is just a literal, and gets yielded as such. Since values yielded from the global context are simply yielded to standard output, the classic "Hello, World!" program in Swil, is simply "Hello, World!"

```
Hello, World!
```

Literal outlines can be "tagged" in the sense of a markup language.

```
h1>Hello, World!
```

The markup to the left of the angle bracket is a name (traditionally called an L-value), and markup to the right is a literal. Together, they perform an application. If a line contains no markup, it's interpreted as a literal.

Literals can defer to a value to fill in text. Any "balanced"[8] expression bounded by curly braces will be replaced by its value.

```
h1>Hello, {you}!
```

Values can defer to a literal. Any "balanced" expression bounded by double quotes will be replaced by its evaluation as a literal outline.

```
h1>Hello, {"World"}!
```

Swil's syntax affords the ability to easily nest literal space and value space within each other indefinitely.

```
h1>Hello, {"{"{"World"}"}"}!
```

Swil can perform the same operation can entirely in value space, as a typical programing language would.

```
{h1: "Hello, World!"}
```

4.1 Literal Outline

The simplest form of markup in the literal outline space is a one line application. Any line that contains a balanced expression up to the first unenclosed closed angle bracket and some non-white-space content is an application of the right literal on the left target.

```
value>Literal
```

Markup may span multiple lines. Swil uses indentation to infer where to end such nested content.

```
value>
    Literal
    Literal
```

Unlike value space where blank lines are ignored, in literal space a blank line has the value of an empty string, effectively yielding a blank line. So, its often

[8] see "Enclosers" on page 13.

desirable to explicitly close a section with slash and an closed angle bracket.

```
value>
    Literal
    Literal
/>
```

While you can not nest multiple markups on one line, you can nest them inside the content of the tag.

```
outer>
    inner>
        Literal
        Literal
/>
```

4.2 Literal

The value of a literal is its text. However, any part of that text that contains a balanced expression bounded by curly braces is replaced with the evaluation of its content as a value outline (interpolation).

In this example, a literal contains a variable 'you'. If the value of 'you' is "World", the value of the literal is "Hello, World!".

```
Hello, {you}!
```

If an embedded value has multiple lines, the literal spans to include the following lines. Swil prefixes the new lines with a number of spaces equivalent to the number of characters that had preceded the value.

```
bah>
    Bah
    Bah
ram>
    Ram
    Ram
ewe>
    Ewe
    Ewe
{bah} {ram} {ewe}
```

This example has the following output.

```
Bah
Bah Ram
    Ram Ewe
        Ewe
```

4.3 Value Outline

A value outline is analogous to a literal outline. It differs in several major ways.

- The combination operator is a colon instead of an closing angle bracket. Therefore, there is no need to bind relational expressions with parentheses.
- Many combinations can be nested on one line.
- Blank lines have no value.
- There is no terminator for a combination block.

Much like a literal outline, any line that does not have a combination is simply evaluated as a value.

4.4 Value

Values can be compound values, composed of semicolon delimited evaluations. Each part gets yielded as a line. Values can alternately be expressions or names. An expression can be a binary operation of values or parenthesized expressions. Values can be numbers. Any value bounded in double quotes gets

evaluated as a literal. Any value bounded in single quotes gets evaluated as a verbatim.

Application in value context occurs when you supply a name and arguments. As long as there is at least a single argument, the parentheses are optional.

4.5 Verbatim

There is nothing special about a verbatim. It's value is the lines of its text.

4.6 Names and Targets

Names in Swil are similar to UNIX file system paths. Dot, '.', refers to the current context; Dot-dot, '..', refers to the parent context. Names can contain any characters other than delimiters, enclosers, white space, dot, slash, and tilde[9]. Names can not begin with a dash.

Slashes delimit context names and indices, and thus can be used to traverse the context name space.

```
root>
    child>
        leaf>
            Hi!
{root/child/leaf}
```

Names can contain expressions in curly braces, to indicate that the value of the contained expression is the name sought, effectively dereferencing any number of levels of indirection.

```
variable>Hi!
variable-name>variable
{{variable-name}}
```

Names can be numeric indices to get a single line from a context.

```
list>
    A
    B
    C
{list/2 = 'B'}
```

[9]You actually can use tilde, but it wont be forward compatible. Dot will eventually be permitted to specify a type in much the same sense that you would append a file extension.

4.7 Enclosers

In Swil, enclosing operators include parentheses, single and double quotes, curly braces, angle brackets, and square brackets. To minimize the necessity of "escaping" text in particular values, Swil parses its text in progressively more specific passes, mostly content to verify that arguments are "balanced" before evaluating them. This permits the text of the given arguments to be in any language where all enclosers are loosely balanced. Opening parentheses have matching closing parentheses. Open curly braces have matching closing curly braces.

However, to effectively scan most languages, Swil employs a broad notion of balanced enclosers. For example, angle brackets can be used as either enclosers or relation operators. To avoid ambiguity of expressions containing angle brackets, free angle brackets are considered balanced as long as they're surrounded by parentheses.

```
if (x < y): x
```

In most languages, enclosing characters need not be balanced inside quotes of any kind. Swil provides an exemption for curly braces so that its own literal syntax can nest values. It is conceivable that a syntax exists where parentheses and square brackets are used in "interval notation", so Swil considers an string balanced even if a square bracket closes an open parenthese.

```
range: [0, 10) # all x where
               # x >= 0 and
               # x < 10
```

To facilitate scanning of balanced text, Swil has a table of precedence for enclosing operators. These are the enclosers from lowest to highest precedence.

- single quotes and back ticks
- curly braces
- double quotes
- parentheses and square brackets, interchangeably
- angle brackets (chevrons)

4.8 Delimiters

In Swil whitespace, comma, colon, and semi-colon are delimiters. Semi-colons delimit rows, and commas delimit columns. Colon delimits combinations. While one would typically write each row of their program on a separate line and use indentation to connote the structure of the document, Swil provides semi-colons and arbitrarily low precedence delimiters so that one can write an arbitrarily complex, multiline value on a single line. Consider this example.

```
evaluate>
    value:
        if n % 2:
            n - 1
        if n !% 2:
             1 - n
```

Comma precedes semi-colon. Semi-colon precedes colon. However, if you need a colon to bind less tightly, you can use a double-colon, '::'. Likewise, any operator can be repeated an arbitrary number of times to achieve lower precedence. All operators of the same length have analogous precedence relationships.

```
value:: if n % 2: n - 1;; if n !% 2: 1 - n
```

A delimiter's associativity depends on whether you place it next to the left or right operand.

If you place a colon immediately after the left operand, it has right to left associativity, as all preceding examples have. However, if you place it just before the right operand, it becomes left to right associative. This is handy for expressing function calls as a pipeline.

```
{lines :nonblank :where n, n !% 2}
```

This is semantically equivalent.

```
{where(n, (n !%2), nonblank(lines))}
```

Similarly, commas and semi-colons are normally left to right associative. Placing them against the right operand makes them right to left. This example demonstrates the list (1, 2, 3) expressed with prefix semi-colons.

```
{3 ;2 ;1}
```

5 Tools

5.1 Operators

Swil supports some arithmetic, logic and relation operators. All operators must have space on both sides. This permits variable names to contain dashes and slashes without being confused for arithmetic. Most of these operators simply invoke a function by the same name in the 'number' context, and thus can be overridden. Swil does not support an 'order of operations' (precedence and associativity). All compound terms must explicate the order in which its terms should be evaluated with parentheses. Swil evaluates expressions from left to right.

```
{(4 * 4) + 1}
```

Swil, as yet, supports the following operators:

addition	x + y
subtraction	x - y
negation	-y
multiplication	x * y
division	x / y
reciprocation	/y
modulo	x % y
divisibility	x !% y
exponentiation	x ** y
root	x // y
equality	x = y
inequality	x != y
less	x < y
less or equal	x <= y
greater	x > y
greater or equal	x >= y
logical and	x & y
logical or	x | y
logical not	!y
range	x..y

Addition, subtraction, multiplication, division, and modulo need no introduction. These all perform integer arithmetic and round toward negative infinity. Much as unary negation is equivalent to subtraction from the arithmetic identity (zero), the unary reciprocation operator is equivalent to division under the

multiplicative identity (one). The divisibility operator is equivalent to the logical negation of a modulo operation, which is handy for testing whether a number is evenly divisible by another, or just plain even if the right operand is two.

```
a-is-even>{a !% 2}
```

Doubling multiplication and division operators ascends to the next order of arithmetic, providing exponentiation and root[10].

The relation and logic operators need little introduction, save that 'equals', 'and', and 'or' differ from those in C. Since the combination operators, colon and right angle bracket depending on the evaluation context, perform assignment, there is no collision between assignment and the equality test operators, so a single equal sign will suffice. I've also opted not to continue C's legacy of doubling ampersand and pipe for logical intersection and union.

The range operator is comparable to the Perl range. The bounds are both inclusive and cardinal, unlike iterators in C++ and ranges in Python where they are ordinal and the upper bound is exclusive.

```
{1..3}
```

is equivalent to

```
1
2
3
```

5.2 Built-in Functions

Swil, as yet, supports the following built-in functions.

5.2.1 Flow Control

- `if(condition, block)`

  evaluates its block conditionally.

- `while(condition, block)`

  evaluates its block as long as the condition passes.

[10] Pending a difficult choice, slash-slash, '//', may become the logarithm operator.

5.2.2 Abstract Level Flow Control

- `for(iterand, lines, block)`

  evaluates a given block iteratively, assigning each line to a variable named by iterand. Swil's for(iterand, lines, block) is comparable to for(iterand in lines): block in Python, for or foreach $iterand (@lines) {block} in Perl, or foreach ($lines as $iterand) {block} in PHP.

- `each(iterand, block, lines)`

  evaluates a given block iteratively, assigning each line to a variable named by iterand. each is simply a permutation of the later arguments of for. each is comparable to map in functional and some imperative languages, except that the code block must explicate its parameters rather than simply pass a function.

- `times(quantity, block)`

  evaluates a given block an number of times.

- `where(iterand, condition, lines)`

  iterates each line as the name iterand, yielding only those lines that meet the given condition block. Comparable to SQL.

- `reverse(lines)`

  lines in reverse order.

- `count(value)`

  how many lines a value contains.

5.2.3 Numeric Manipulation

- `number`

  a name space for numeric functions

  - `+(terms)`

    addition of all arguments (variadic).

  - `*(terms)`

    product of all arguments (variadic).

5.2.4 Text Formatting

- `unparagraph(lines)`

  merges adjacent lines, as separated by any number of blank lines.

- `unhang(lines)`

  merges all lines that are indented (hanging) one space deeper than each "first" line.

- `wrap(lines)`

  wraps lines of text at column 80.

- `wrap(width, lines)`

  wraps lines of text at a given column number.

- `indent(lines)`

  some text, indented by four spaces.

- `nonblank(lines)`

  yields only those lines that have no dark text.

5.2.5 Text Manipulation

- `split(lines)`

  yields the letters of each line of text.

- `split(delimiter, lines)`

  yields substrings of each line that straddle the given delimiter.

- `join(lines)`

  merges all lines onto a single line with no intervening text.

- `join(delimiter, lines)`

  merges all lines, placing delimiter between each.

- `begins(value, beginner)`

  whether lines are equivalent up to the length of some beginning text.

- `ends(value, ends)`

  whether lines are equivalent from the length of some ending text.

- `length(lines)`

  the total number of characters in all given lines.

5.2.6 System

- `file(name)`

  reads the lines of the named file.

- `file(name, value)`

  writes value to the named file.

5.2.7 Parsing

- `evaluate(block)`

  evaluates the result of evaluating block as a Swil value.

- `verbatim(*lines)`

  returns the text of lines without evaluating them.

- `literal(*lines)`

  returns the literal value of the lines. This includes evaluating the contents of curly braces as values.

- `python-execute(python)`

  executes a Python program in the current context.

- `python-evaluate(python)`

  evaluates a Python expression and returns the lines of its result.

- `#(comment)`

  does not evaluate its arguments.