EUROSPEECH '97
5th European Conference on Speech
Communication and Technology
Rhodes, Greece, September 22-25, 1997

MUSE: A SCRIPTING LANGUAGE FOR THE DEVELOPMENT OF INTERACTIVE SPEECH ANALYSIS AND RECOGNITION TOOLS[1]

Michael K. McCandless and James R. Glass

Spoken Language Systems Group
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139 USA
http://www.sls.lcs.mit.edu, mailto:{mike, jrg}@sls.lcs.mit.edu

ABSTRACT

Speech research is a complex endeavor, as reflected in the numerous tools and specialized languages the modern researcher needs to learn. These tools, while adequate for what they have been designed for, are difficult to customize or extend in new directions, even though this is often required. We feel this situation can be improved and propose a new scripting language, MUSE, designed explicitly for speech research, in order to facilitate exploration of new ideas. MUSE is designed to support many modes of research from interactive speech analysis through compute-intensive speech understanding systems, and has facilities for automating some of the more difficult requirements of speech tools: user interactivity, distributed computation, and caching. In this paper we describe the design of the MUSE language and our current prototype MUSE interpreter.

1. INTRODUCTION

The needs of speech understanding research place unique demands on supporting tools and systems. Interactive tools allow students and researchers to explore the properties of speech and to evaluate the strengths and weaknesses of current approaches to speech understanding. For compute-intensive training and testing we may need tools which can distribute load across many networked computers as necessary. We may use databases to represent our corpora, and statistical tools to study distributional characteristics. Often speech systems need to be flexible so as to trade off accuracy for real-time performance depending on whether one is conducting research or giving a demonstration. In all cases, these tools must be flexible and grow with our changing needs as new research breakthroughs occur.

While there exist numerous tools for certain areas of speech research [2, 3, 5, 9], we still frequently find it necessary to build our own tools [4, 10]. This might take the form of creating sets of scripts in a scripting language like Python [8], Perl, Csh or Tcl [7], using Unix tools such as Make and CVS to maintain recognizer configurations as they change over time, or resorting to a systems programming language like C or C++. Such languages are required primarily because existing systems are difficult to extend and integrate with one another. In our experience, most tools that are used within a research group are those that were developed "in-house" in systems programming languages.

In our opinion, progress in speech research is greatly limited by the difficulty of creating new tools or customizing existing ones. There are many tools we need to create, but existing languages do not address the particular needs of speech tools. We propose a new high-level scripting language, MUSE, to address these limitations. MUSE is designed specifically to make it easier to build tools which satisfy the needs of speech research, including facilities for automating user-interactivity, distributed computation, and short- and long-term caching.

MUSE is an abstract language, and we are actively designing all aspects of its syntax and semantics. At the same time, we design and test prototype MUSE interpreters which conform to the syntax and semantics of the MUSE language. Much of the MUSE language has now been designed, and we have built various prototype interpreters which are able to execute certain subsets of MUSE. Figure 1 shows a snapshot of a MUSE program running under one of our prototype interpreters. This tool, which allows the user to interactively "edit" a spectrogram, would be difficult to achieve in existing programming languages. We have built other tools using prototype MUSE interpreters, including a general transcription tool, a tool to overlay and compare spectral slices, and one to interactively examine the residual of an LPC analysis.

In this paper, we will first outline some of the unique needs of speech tools and why we feel these needs are not fulfilled by existing languages. Next we describe the semantics of the MUSE language: the rules that an interpreter will follow when executing a MUSE program. Finally, we describe our latest prototype interpreter, currently implemented in Python [8], which is able to execute a subset of MUSE.

2. YET ANOTHER LANGUAGE

One might think that with all the languages already available to the speech researcher, adding another only exacerbates the situation. We feel this is necessary because the particular needs of speech tools and systems are difficult to meet using existing systems and scripting languages. First, while interactive tools are crucial to gaining insight or brainstorming for new ideas, they are known to be difficult to build [6]. One must learn about "event-driven" programming, interface toolkits, a systems programming language such as C or C++, computation on demand or lazy evaluation, and pipelining and threads to ensure immediate response for the GUI. For example, while it should be possible to interact directly with a ten minute long utterance, many tools have difficulty with this task.

Second, we feel it is crucial that all functionality now offered piecemeal in a few dozen tools and languages should be available within a single cohesive framework. The difficulty of integrating the functionality of two or more tools, at present, greatly limits the types of experiments we are able to conduct.

Third, a common mode of research is the "tweak-and-rerun" model. For example, in order to study the impact of the search pruning threshold on recognition accuracy, we would run our recognizer several times, at different thresholds. Because this

[1]This research was supported by DARPA under contract N66001-94-C-6040, monitored though Naval Command, Control and Ocean Surveillance Center.

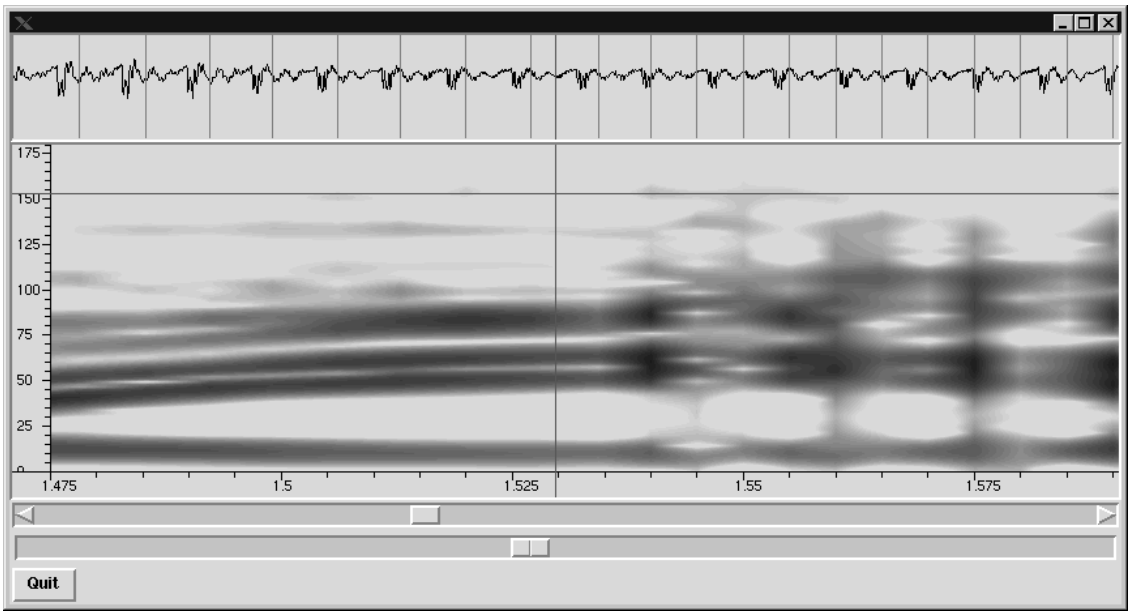

Figure 1: A MUSE tool which allows a user to edit a zoomed-in spectrogram by changing the alignment of the individual frames of the FFT used to compute the short-time Fourier transform. As the user moves a time mark overlayed into the waveform view, the spectrogram underneath is updated in real-time. In this image, the frames were originally spaced every 5 ms, and the user has edited the left-portion of the vowel so that the frames are pitch synchronous. The scroll bar allows the user to scroll through the entire utterance, while the scale just below it allows the user to vary the time scale of the display.

sequence of computations share a common front end, precaching (and saving "on disk") all computations up to the search would save a lot of time. In many systems this is a difficult process often left to the user, which can lead to disaster should the user forget what is cached and what is not. The nature of our computation is such that, from one run to the next, much recomputation is unnecessary, and our tools should help us take advantage of that property.

Finally, speech tools and systems often require massive amounts of highly parallelizable computation for training or testing across a set of utterances, yet systems for utilizing distributed resources are not generally available or easy to learn. Most research groups have developed an "in-house" model of parallel computation, or lacking such a model, they require researchers to manually divide up and statically distribute their computation.

These four areas — managing interactivity, integration, support for general caching, and distributed computation — are crucial for speech tools, yet are difficult to accommodate with existing languages and tools. With MUSE we directly address these issues.

3. LANGUAGE DESIGN

Figure 2 shows an example MUSE program illustrating some of the unique properties of MUSE. MUSE is a *declarative* language, allowing the programmer to specify functional *relations* that should hold among variables, without worrying about the details of how the relations will be computed. There is no temporal order in a MUSE program: variables may be used before they are created; they do not need to be previously declared; and permuting the statements in a MUSE program will not change what the program means[2]. Variables, which are used to refer to intermediate results (such as `wav` and `tscale` in Figure 2), can only be assigned one value per scope. For example, one cannot include both `a=4` and `a=10` in the same scope because MUSE would not know which value of `a` to use when it is referenced. A MUSE interpreter keeps track of the "types" of each variable at run-time and will signal an error if there is a mismatch.

MUSE syntax is intentionally kept simple so that it is easy to learn. The basic statement assigns an expression to a variable name, where the expression may be a constant value, such as the string "hello" or the float 3.14, or the result of applying a function to other expressions. Expressions may be arbitrarily embedded, such as the application of `imarks` in Figure 2. If-statements allow alternation to be included in the program, depending on the run-time boolean value of a conditional expression.

Users may define their own function abstractions, which are treated as first-class values that may be stored in variable names, passed as arguments, and returned as results. Functions are statically scoped, and each function application creates a fresh scope. These properties were inspired by the Scheme language [1].

We refer to MUSE as a scripting language because the syntax is simple, variables may be used without prior declaration, high level functions and data types are available, and most of the "real" computation will occur in built-ins written in a faster language (e.g., computing a spectrogram image from a structure representing a spectrum).

3.1. Execution

Because MUSE is a declarative language, MUSE interpreters have a high degree of freedom when executing a program. This source of freedom also means a MUSE interpreter must do much more work than interpreters for other languages. A MUSE program is first parsed, and all data dependencies are recorded. Expressions may then be executed according to the order required by dependencies. The interpreter might choose to execute computations in parallel, either on different computers, or locally on multiple processors, or serially in some order. Intermediate results may be cached, either on disk or in core memory dis-

[2]This is true as long as the permutation obeys the scoping in the program.

```
// Pixels per second.
tscale = 350

wav = waveform(file="sa1-b-faks0.wav")
wimg = iwaveform(wav, tscale)

// Overlay marks onto waveform image when
// zoomed in enough.
if tscale > 300
  wmimg = ioverlay(wimg, imarks(marks, tscale))
else
  wmimg = wimg

// The scrolled image.
simg = iextract(wmimg, s.view, 0,
                width="20c",
                height=wmimg.height)

r = root()
w = window(r, simg)
s = scrollbar(r, w, wmimg)

pack(r)
pack(w, s)
```

Figure 2: A fragment of a MUSE program to display a scrollable waveform, loaded from the file-system, with time marks overlaid. High-level functions, like `iwaveform`, `imarks` and `ioverlay`, allow the programmer to obtain images of waveforms and marks, and overlay them, without being concerned with how to allocate the space to store the images (which could be very large), or when to compute various portions of the image. When the view of the scroll bar, `s.view`, changes (because the user of the tool has scrolled), MUSE will carefully recompute only the affected objects. Because a MUSE program has no temporal order, variables may be referred to "before" they are created (for example, `s.view`).

tributed among many machines, and then later reused in order to avoid redundant computation. Large, time-consuming computations might be divided into smaller pieces, when possible.

These are examples of the details under the auspices of the MUSE interpreter, about which the programmer in general does not need to worry. In any case, the final outcome of the computation is guaranteed to be the same, although intermediate changes in reaching that outcome may differ. In general, MUSE interpreters may base their choices on tradeoffs and availability of computational and storage resources, in addition to the likelihood that a given value may be needed again in the future. For example, it may not be worth caching waveform images to disk, as they consume space and can be redrawn quickly. In contrast, it may be worth caching the accuracy a particular recognizer configuration achieves on a certain set of utterances, as this is a single floating point number which would require much computation to regenerate. Future MUSE interpreters will likely offer options to allow the programmer to control these choices to some extent.

In an interactive tool, with a scroll bar, MUSE might not cache images which can be redrawn quickly, such as a linear axis view, but might cache more costly images, such as a spectrogram view. Also, parts of an image may be cached, while other parts are computed. In all cases, MUSE will only compute those parts of a data type that are actually needed by the user at runtime. This is what makes it possible to manipulate very large, even infinite, structures.

3.2. Change

One of the unique elements of MUSE is its model of *change*. At any time while a MUSE program is running, it is possible for the value of a variable to change to a new value. Such change typically originates from a graphical widget with which the user has just interacted (clicking on a button, sliding a scroll bar). When this happens, MUSE carefully recomputes those variables whose values depend directly or indirectly on the changed variable.

This explicit model for handling change is used to implement all forms of user interactivity. For example, when the user drags a time mark, they are changing the time value associated with the mark. If in the program the time value of that particular mark was used as the input to a spectral slice computation, the computation will be redone as the mark moves, as frequently as is possible given the power of the computer and the complexity of the computation. The propagation of changed values is handled by the interpreter; the programmer does not need to do anything except express how actions by the user (dragging the mouse, clicking a button or a key) translate into changes in the MUSE program.

The granularity of a change varies with the data type. For an integer or float variable, the value either changed or did not; no further information is recorded. For images, however, change may be contained to within a rectangular region, which allows MUSE to recompute only the affected area. This is what makes it possible for the user to receive immediate feedback with the editable spectrogram tool shown in Figure 1; computing the entire spectrogram image every time a mark moves would be prohibitive for interaction.

Handling changes introduces substantial complexity into a MUSE interpreter, especially when the changes propagate through if-statements and function applications. For example, the programmer might use an if-statement to choose which set of models to use during recognition:

```
if fast
  <use fast models>
else
  <use slow accurate models>
```

If the `fast` variable changes while the tool is running, all computations which had occurred on one branch of the if will have to be revoked, and new computations from the other branch will then be performed. Similar difficulties arise with the application of user-defined functions, and with propagating change when the impact is compute-intensive.

This element of MUSE greatly simplifies the creation of interactive tools. Currently, the only source of change are graphical widgets, such as scroll bars, scales and buttons. We are actively designing more general expressions within MUSE which will allow the programmer to initiate their own changes at specified instants of time.

3.3. Data types

One of the most important aspects of a language is what built-in types are supported, and what facilities are offered for the user to create new types. While MUSE will eventually have many data types to support speech based tools, this aspect of the language has not yet been fully designed. Data types in MUSE are particularly complex as they interact heavily with MUSE's declarative style and model of change. For example, just as the programmer does not know when a particular function might be executed, the programmer will also not know whether a data object is actually stored, is being computed on demand, or something in-between. For example, a common iteration technique in the Python scripting language is:

```
for i in range(0, 1000):
  <do something>
```

The `range` function actually allocates and creates a list of 1000 integers. We would like the analogous expression in MUSE to

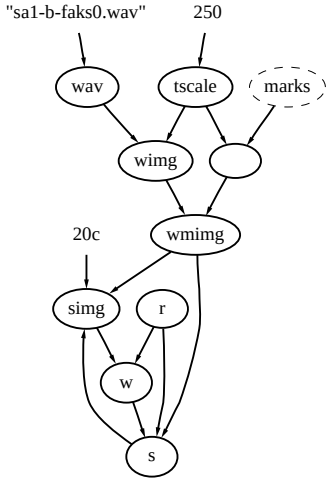

Figure 3: Graph produced when the prototype MUSE interpreter executes the program fragment in Figure 2. The `marks` node is dashed because it is referenced but not defined. Blank nodes are temporary nodes introduced by the interpreter to reduce compound expressions. Execution of the program proceeds according to the dependencies between nodes in the graph.

allow for the interpreter to choose not to actually compute and pre-cache the entire list, but rather generate its elements as they are needed.

One data type which we have thoroughly explored is the image data type. An image is a two-dimensional grid of pixels of a certain width and height (which may be infinite: a horizontal axis image extends from $-\infty$ to $+\infty$). Images are created from built-in functions: `iwaveform` produces an image from a waveform, `iaxis` produces a linear axis image, etc. Images may be overlaid, joined and spliced, and then displayed into windows. For example, one way to offer a scrollable image to the user is to splice the appropriate sub-image out of a larger image according to a "view" controlled by one or two scroll bars. MUSE will include facilities allowing the programmer to create their own data-types, such as tuples and structures.

4. INTERPRETER

We have a prototype MUSE interpreter implemented in Python [8], using an interface to the Tk toolkit for graphical widgets. This version implements many aspects of the MUSE language, including basic execution and user-defined abstract functions, but excludes distributed computation, if-statements and general caching. We have added numerous builtin functions appropriate for speech analysis, including functions to compute and display preemphasized waveforms, spectrograms with different analysis windows, transcriptions, spectral slices, LPC analyses, linear axes, and time marks. Built-in data-types include images, waveforms, marks and spectra, strings, integers, booleans and floats.

At runtime, the interpreter parses the program and translates it into an equivalent graph. Compound expressions are broken down into individual steps, introducing temporary nodes into the graph. In the graph, a node is created for every variable in the program, and edges link two nodes when there is a dependency between the corresponding variables. The graph is then analyzed, and nodes will be computed in order according to their dependencies. When an abstract function is applied, the effect is to duplicate the sub-graph which represents the body of the function. In this way, what begins as a small program in text can denote a large dynamic graph at run-time. Figure 3 shows the run-time graph created by the MUSE fragment in Figure 2.

5. DISCUSSION

MUSE offers unique ways to combine existing functionality. The language is simple, so that it is still approachable and easy to learn, but complex enough to support diverse functionality. The declarative model leaves many computational details to the interpreter, freeing the programmer from having to deal with them. This includes providing interactivity through a model of time-sensitive change, caching and reusing prior computations, and scheduling computation to take advantage of distributed resources. This frees the programmer from worrying about details of interaction even within a compute-intensive tool.

Of all existing tools and languages, we feel MUSE is most similar to SAPPHIRE [4]. However, unlike SAPPHIRE, which relies on Tcl as its scripting language, MUSE is a new language with very different rules of execution. MUSE can handle fine-grained changes, such as the impact of moving a single time-mark, and relies to a greater extent on lazy evaluation so that if a portion of an image is not needed, it will not be computed. Finally, SAPPHIRE is not really "gentle sloped": extending it requires writing C code and becoming quite familiar with SAPPHIRE's internal design. Instead, MUSE is designed to be expressive enough so that such functionality could generally be directly expressed within a MUSE program, and failing that, built-in functions could be added in a systems language without detailed knowledge of the interpreter's design.

MUSE is under active development and is quite far from completion. It is our top priority to first finish designing the syntax and semantics of the language, and then to build an efficient interpreter which is able to execute MUSE programs. We are actively designing data abstraction and a general model for change for the language. We plan to build MUSE interpreters which support distributed computation and short- and long-term caching, as well as supporting change in the context of if-statements and function application.

6. REFERENCES

[1] H. Abelson, G. J. Sussman, and J. Sussman. *The Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Massachusetts, 1996. Second Edition.

[2] The Hidden Markov Model Tool Kit, http://www.entropic.com/htk/htk.html.

[3] ESPS/Waves, http://www.entropic.com/products.html.

[4] L. Hetherington and M. McCandless. SAPPHIRE: an extensible speech analysis and recognition tool based on tcl/tk. In *Proceedings of the International Conference on Spoken Language Processing*, 1996.

[5] G. E. Kopec. The signal representation language SRL. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-33(4):921–932, August 1985.

[6] B. A. Myers. State of the art in user interface software tools. Technical Report CS-92-114, CMU, February 1992.

[7] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, 1994.

[8] The Python Language, http://www.python.org.

[9] S. Sutton, J. de Veilliers, J. Schalkwyk, M. Fanty, D. Novick, and R. Cole. Technical specification of the CSLU toolkit. Tech. Report No. CSLU-013-96, Center for Spoken Language Understanding, Dept. of Computer Science and Engineering, Oregon Graduate Institue of Science and Technology, Portland, OR, 1996. Also *http://www.cse.ogi.edu/CSLU/toolkit/toolkit.html*.

[10] V. W. Zue, D. S. Cyphers, R. H. Kassel, D. H. Kaufman, H. C. Leung, M. Randolph, S. Seneff, J. E. Unverferth III, and T. Wilson. The development of the MIT lisp-machine based speech research workstation. In *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, pages 329–332, Tokyo, Apr. 1986.