Programming languages span a wide range of human needs, beyond that of the more traditional programmer population. In fact, the non-programmer is often the more interesting subject, in examination of the use and semantics of languages meant to be written by humans for machine interpretation, if only for their exoticism in ways of thinking and organizing. Their languages are built for tasks not claimed in the territory of computer science as an academic discipline. Such languages are built for authoring stories, or animating shapes. Their domains call for creativity and problem solving unmeasured by Turing Completeness. Their logic is not (necessarily) saturated in for-loop repetition or .

Such languages strive for new logics and language fundamentals (primitives, syntax, idioms) perhaps unexplored by the programming mainstream. They are built and used and improved upon outside of computer science, heedless of its well-established paradigms. In fact, authors of such languages frequently cite in their development goals a very explicit desire for a product bearing no resemblance to traditional programming, which shoulders a great stigma in fields beyond computer science. This is to the great dismay of anyone infatuated with the magic and purity and primacy of logic in programming. For alas, computer science is neighbored with mathematics in the realm of subjects generally feared and untouched (without a ten-foot pole) by the masses.

Yet the light in this unfortunate “boring”, “bland”, “overly-complex” labeling of the act of programming lies in the emergence of languages written for non-programmers, which push the edges outward, and diversify the evolutionary pool of programmatic practice and linguistics.

As computer technology becomes paramount in daily life, programming will be an increasingly prevalent, commonplace activity, even to those not under a CS umbrella. So these “non-programmer” programming languages will multiply and gain traction. What would make this rich ecology of non-traditional and traditional languages useful and interesting is cross-pollination. Splicing attributes and considering one domain’s goals from the lens of another may inform unusual and boundary-bridging paths in the progression of modern programming languages.

Project Overview

Thus, with this enterprise, I aimed to interface languages of very different fashions: Max is a graphical, pseudo-functional language for artists, with surprising uniqueness in coding patterns and practices, Haskell is traditional and functional, and celebrated by a relatively small population of mostly academics, and Python, an object-oriented language with widespread use and popularity among traditional programmers worldwide. With considerations of each one’s idiomatic and fundamental standards-of-practice, I have written a compiler from Haskell to Max in Python. This allowed for direct and applied “cross-pollination”, as previously mentioned, such that features of one language were examined and held up to the features of another, to see how they might best meld or at least loosely translate to one another.
Translating a functional language to a graphical language by means of an object-oriented language is furthermore a non-trivial exercise in the use and balance of drastically different paradigms, and an introductory swing at parsers and compilers (of which I had no prior experience). The parsing and writing was all done “from scratch”, in that I did not use third-party parsing tools beyond the Python standard library. This was fed by the need to directly interact and manipulate the subtleties of each language, to decipher meaningful cross-language connections and differences.

The parsing algorithm itself was also largely emergent from my own interpretation of the problem and needs of a parser, as specific to a small subset of types of Haskell functions. It was, in effect, very emergent from the domain, and not informed by the dense body of work on parsing that exists in the field of computer science and programming languages.

I think this is manner of approach was beneficial, in that it allowed for more active engagement with the material, and made for a learning process uninhibited by prior knowledge.

Meet the Languages

**Max MSP:** Max was born in the 1980’s as an interactive music performance software, allowing sound artists to make virtual synthesizers and effects processors without reliance on time-intensive hardware tinkering. Since then, it has undergone significant advancement and expansion—in UI, processing power, libraries, and community, though the latter is still composed primarily of musicians and new media artists. Thus its positioning on the edge of the programming language ecosystem is solidified.

With artists as its chief users and audience, Max is implacably graphical. It was designed for easy navigation and reading by artists, and for live processing of audio. In effect, the GUI software environment for the language visualizes code in “patcher” programs, which contain function “objects” connected to one another by “patchcords”. Data and process flows through the patcher by way of these connected objects, and that data flow can be manipulated in live time by interaction with UI objects like sliders and knobs. Often times in Max, this data is a stream of stereo audio, or midi notes.

The editing mode enables program authorship: it is there that the programmer-artist may drag around or create new object boxes or patchcords to define program functionality. In presentation mode, the program’s execution may be initialized and interacted with via designated UI elements. But you may also readily switch back to edit mode to drag new connections or reroute old, while the program is running. For example, you may start a programmed chirping, and switch to edit mode with Cmd-E to route in an amplifier before the speaker output to get a louder, or otherwise synthesized chirp. Compilation is therefore somewhat moot in the eyes of the author. Programming becomes a dynamic conversation between human and code, even while it is being written, similarly to the experience of programming with interpreters of traditional languages (sbcl,
In a way more deeply integrated into the standard use of the language. See Further Work for discussion of how this might be harnessed with a Haskell-to-Max compiler.

Interestingly enough, there is significant resemblance to a strongly-typed, functional Haskell, in how Max represents and organizes programs. Due to its favorite function as a live audio manipulator, Max treats processes, and not stored data, declarations, or states, as first-class citizens. Despite its labeling of the atomic units of a program “objects”, Max is not strongly object-oriented. There is no sense of a class, or of an object in the OOP use of the term. “Object” boxes are essentially functions, that take input through one or more inlets, performs some action on that input, usually mathematical, and produces output through one or more outlet terminals. It is even strongly typed, in that the boxes producing output for other boxes are static, and inlets require a predefined type, and outlets pass along results of predefined type(s).

According to my own research, there have been no attempts made to bridge Max with more traditional languages, beyond its built-in capability for embedded Java/C/JavaScript objects. This is a useful feature, and makes its own steps toward language cross-pollination, but still explicitly isolates the different types of code, making the direct translation of one to the other a unique venture. Other translators compile between languages of the same, musical/artistic domain (e.g., SuperCollider).

Haskell: Haskell, as a strongly typed functional language with clean and legible syntax, is well-matched to Max. In determining how Haskell functions are to be represented in Max, I make heavy use of the type statements that often precede Haskell function definitions. In fact, for this limited Haskell-to-Max compiler, I require those function type statements for every function written to the Maxpatch, so that the algorithm may readily manage operand requirements for each function used. The Max inlets are precisely the Haskell function inputs, and the outlet is designated by function output.

Python: Python, learned specifically for this project, is well-suited to rapid prototyping and development intended for proof-of-concept, rather than optimized runtime or more elegant code design. Its object-oriented heritage also enabled easy parse tree structures for function definitions. Though, ideally, the parser would be written in Haskell to provide more direct and natural type translation.

Implementation

Maxpat File Output

Max 6 represents its code by compilation of simple text files, suffixed “.maxpat”, that resemble basic script or markup. Each object is defined by a unique id, a maxclass (essentially a type), a string of text that appears in the box (e.g., in Fig.1, the text attribute of the “+” object is precisely “+”), x-y coordinates of the object in the patcher window, and the visual size of the object.

```
funcdefs = {} # dictionary { name : (#args, [types]), ... }
maxboxes = {} # dictionary { id : (maxclass, text, x, y, w, h), ... }
maxcords = [] # list       [ (src-id, src-outlet, dest-id, dest-inlet), ...]
```

Fig.2 Parser’s representation of library of functions, `funcdefs`, the collection of Max “objects” waiting to be written to the `.maxpat` file, `maxboxes`, and the list of Max patchcords connecting the objects, waiting to be written, `maxcords`. 
Patchcords are defined by the source object’s id, the source object’s outlet number (since there can be multiple), the destination object’s id, and the destination object’s inlet number. Patcher files are essentially lists of these object boxes and patchcords, with a little bit of patcher overhead (file version, GUI settings, window size, etc).

As such, I represent these two collections as Python collections (see Fig.2), which are appended to after a single function definition is made into a parse tree and individual Max elements and their relations to one another are discerned from this tree. Then it is a trivial task to print these collections to a file in the proper .maxpat format.

I am thankful to the Max compiler for its accommodation of the relatively scant scripts I generate for it. In ordinary Max files, generated from the standard new-patcher option within the Max environment, much more information, like . I pushed the compiler to its limits for .maxpat input to simplify the task. Fortunately, it is able to infer much about the behavior of objects and cords, particularly by lookup of the maxclass or text attribute in its extensive built-in libraries.

**Funcdefs**

The other collection (dictionary) listed in Fig.2 is for the function names and types, which are the parser’s intermediary representation of the function definitions available for interpretation and use at the point of constructing the function parse tree.

For each line of Haskell file (.hs) input, a function’s type is identified, via regular expression, if there is a word between a newline and ‘::’, Haskell’s type operator. At this point, the function name is added to the funcdefs dictionary with its type, delineated into a list. A dictionary is preferable here for quick lookup, since the parser must frequently check for matching function names in the rest of the file, and since the number of functions (and the .hs file size) is assumed to be small.

```
# Take split string (list of words) of the function definition, # and dictionary of its temporary argument variables (keys) # zipped with their types (values)
def parseExp (list, args):
    exptree = Tree (list.pop(0))
    # Make boxes for each argvar in args,
    # as appropriate UI boxes
    # Patchcords added
    for arg in args:
        addmaxbox ("", translatetype (args[arg]) )
        addmaxbox ("send "+arg)
        maxcords.append (( len(maxboxes)-2, 0, len(maxboxes)-1, 0 ))
    for token in list:
        if token in funcdefs:
            exptree.addNode (token, False)
        elif token in args:
            exptree.addNode ("receive "+token) = make new "receive" box
        elif isnumber (token):
            exptree.addNode (token)
    return exptree
```

*Fig.3* The meat of the parser: Constructs a tree where operators are parents and argument variables or primitives are terminals
This is similarly true for the maxboxes dictionary structure. Maxcords are never matched or searched, so they are implemented as a list, popped until empty when printed to .maxpat.

**Function Definition Parsing**

The other kind of Haskell file line matched by the parser is for function definitions (that must follow a corresponding function type, though they need not be consecutive). These are matched by regular expression for a function name followed by any number of argument variables, followed by “=”. The name is then checked in funcdefs, ignoring the whole line if the function is not referenced in funcdefs. And the argument variable list length (argvars) is checked against the type list in funcdefs for the function at hand. If they match, they are zipped together and sent to parseExp() (See Fig.3) with the rest of the function definition (after the “=”) to be parsed into a tree, which is then sent to parseTree_tomaxboxes() to deconstruct into maxboxes and maxcords.

**Limitations**

- Only arithmetic equations — Could easily add custom names to reference Max library objects
- All functions must be typed — Could make assumptions about function based on operators.
- No pattern matching — Could be done using Max’s “select” object to conditionally route messages/data flow
- No currying — Function arguments must match what is designated in the function type; Currying would be difficult in Max because there is no passing of functions/objects or dynamic creation of objects. The best that could be done is to pass the name of the object, which would determine the path of the routed data. This would still require hard-coded patchcords enabling all possible curries of each function. Gross.

**Further Work**

Because Max is so wonderfully dynamic and immediately responsive to programmer input, a natural continuation of a static Haskell-to-Max compiler would be an in-Max
interpreter of Haskell-like function calls. This could be done using Max’s UI elements, like “textedit”. Simple Haskell function calls, in proper Haskell syntax, typed into the textedit UI element, could be parsed in Max in real time, using its library of regular expressions and pattern matching. This could be used to direct a call (a “bang”) to one of the functions constructed from the Haskell parse, such that you could send input to these functions in the Haskell mode.

Another language conceived outside the familial conventions of traditional computer science is Inform. Devoted to the authoring of interactive fiction (text adventures along the lines of Zork), it uses pseudo-natural language as its programmatic means of artistic expression.

Inform 7, the latest release of what was born in the early 1990’s, is remarkable in its use of natural language, and its smooth translation of the activity of fiction authoring into programming. Yet this is a declarative language, unlike Max. It would be an interesting task to compile or interpret a traditional language in Inform 7, to bridge an author’s language with a programmer’s language, in addition to an artist’s with a programmer’s, as with Max and Haskell.

**Downloading Max**

Max is a proprietary software developed by Cycling 74. A 12-month student authorization currently costs $60, and student and teacher licenses for a whopping $250, but there is a free application for running existing maxpatches without editing capabilities, called Max Runtime. More information on downloads can be found at cycling74.com.