ThunderViper

Specifications for a lightening – inspired language

# What is it?

ThunderViper is actually a Python source compiler that turns Python (or, rather, ThunderPython) into corresponding Assembly code files. This allows Python to be used for system programming, or in my case, designing the Dinosaur kernel. I think of it as injecting “lightening” into an already powerful language concept.

# Is Python Changing?

Due to special requirements of system programming, ThunderViper will change Python in very few ways. However, for the majority of programming, ThunderViper will use syntax and semantics relatable to Python 3.0. For example, for one to print to screen, they would use print() instead of print “”. In essence, this makes all methods follow the same grammatical convention, making parsing easier.

However, all built-in functions in Python have to be implemented in a “Thunderized” library first before they can be used. This means that there may not be standard functions for the first stages of ThunderViper because I haven’t had the time to code them yet.

Since ThunderViper will be used in system programming, it needs some facilities to allow inline assembly. What an inline assembly block looks like and how each section is used is in the section **Inline Assembly: The Works**.

# Built-In Python Functions

There are certain functions in Python that are so built-in to the language that it would be intrusive to have to force the user to import a Thunderized library that handles standard libraries. The solution: we’ll import it for them.

Not all standard functions will be implemented, for I am one person and that seems like a lot of work. However, for standard functions I use, I will make sure to implement Thunder-style. There are other standard functions that due to the structure of Dinosaur would be obsolete or non-applicable.

# Inline Assembly: The Works

ThunderViper is intended for system programming, and as such, will need to give the programmer the ability to write fine-tuned assembly for critical portions that need optimization or need to have unambiguous behavior for things like device drivers. To write this kind of code, there needs to be an assembly block that looks like the following:

%Assembly {

Start:

# Saves Python variables to registers

Code:

# Actual assembly code

End:

# Saves contents of registers to Python variables

}

If, for example, somebody wanted to add two numbers together in assembly, here is what it would look like:

x = 2

y = 3

%Assembly {

Start:

# These can only be assignment operators, strictly from Python to

# register

EAX = x

EBX = y

Code:

add EAX, EBX

End:

# These also only can be assignment operators, from registers

# to Python

result = EAX

}

print(result) # Output would be “5”

You may notice in the above example that I didn’t have to initialize result in order to use it. When I store EAX into result, the result object was automatically created and now can be used in subsequent calculations.

What if you have strings? Or arrays? The principle of loading them in is the same, however you must make sure that you load them into segment registers. However, in order to store the result, you have to do the following:

End:

result = EDI, <start>, <count>

Because you are dealing with assembly, you are not protected by out-of-bound exceptions. Therefore, you are free to use whatever values you may desire. To specify these values using registers, you can use something like this:

result = EDI, ESI, ECX

Or, you can simply use literals like this:

result = EDI, 0, 5

The resulting type using this operation is always a Python tuple.

# The Viper Parser

The only reason why I would do this is if I had an explicit reason not to use the provided Python parser. When I tried using it, it produced a nested listing that had many pointless layers and a whole lot of garbage tokens that added any additional meaning to it besides for special cases for when those tokens may apply. Also, the representation wasn’t clear because it used enumerations from an AST class that for the short time looking at it did not have any sort of method to get a string representation of what the enumeration stood for. I was frustrated so much by this that I thought it would be better use of my time to make a no-nonsense Python parser.

Besides that, I also wanted to add a few features to the language to better support system programming. Everybody knows that you can’t do that very easily with someone else’s code, so here I am concocting one of my own.

Python has a very no-nonsense style to the language where you do not need to read ahead any additional lines to deduce the makeup of the program. This makes creating a parser for Python easier than most other alternative languages.

## Making the Tokens

The Viper parser uses a top-down approach to parsing, which has an iterative behavior rather than a recursive one. The benefits to this are that use of memory is much lower because the stack is not involved. Also, iterative approaches are always just as fast as or faster than recursive methods. The only problem with iteration is that the design may be more complex and less intuitive. However, as you will see, this shouldn’t be that much of a problem.

When the Viper parser finds and stores tokens, it stores the following information:

[<token type as string>, <occurrence #>]

So what tokens is it looking for? Well, here are the different types:

* Left Parenthesis (
* Right Parenthesis )
* Left Bracket [
* Right Bracket ]
* Left Curly Brace {
* Right Curly Brace }
* Comma ,
* Period .
* Word (series of characters that begin with [a-zA-Z\_] with the rest as [a-zA-Z\_0-9])
* Number (series of characters of [-]?[0-9])
* Assignment (simply [=])
* Boolean Operator ([==], [<], [>], [>=], [<=], [!], [||], [and], [or], [not])
* Math Operator ([+], [-], [\*], [/], [<<], [>>])

## Determining Semantics

Notice how only a few of the tokens have explicit semantics associated with them. To figure out the rest of the semantics, or rather, what the rest of it means, we find combinations of tokens that correspond to a program idea. Here is another list of what we can guess from a series of tokens:

* [Word][Left Paren.] = Method or Class Constructor
* [Word][Period][Left Paren.] = Method from Package or Object, or Class Constructor from Package
* [Left Bracket]<anything>[Right Bracket] = List
* [Left Curly Brace]<anything>[Right Curly Brace] = Dictionary
* [Word][Left Bracket]<value>[Right Bracket] = Value from a List

The thing about Python is that everything is an object. This has problems when one tries to create optimized assembly programs, which depends on certain things being consistent to work, like data types.