Zwicky Transient Filter Language:
Specification and Explanation

Chris Cannella

October 16, 2017

Introduction

The Zwicky Transient Filter Language (ZTFL) is a Domain Specific Language (DSL) created to provide users of the GROWTH Transient Marshal to define their own programable filters to determine the relevance of astrophysical observations for their particular scientific interests. In creating ZTFL, we have attempted to balance the convenience of the language for its users with our limits of what features we can implement and maintain. These limitations have driven most of ZTFL's design. The language itself is a blend of C like syntax, for ease of parsing, and Python like syntax, where we could easily retain Python's expressive features. A ZTFL compiler is intended to be embedded within a Python program to produce a function within the program that is provided by a user created string containing a valid ZTFL program and had not been defined within the program itself. This document serves to specificy ZTFL itself, explain aspects of ZTFL's design, provide details of how we currently implement ZTFL in the GROWTH Marshal, and provide some context for how ZTFL serves the current needs of the GROWTH marshal and perhaps the needs of similar tools in the future. We hope that this guide can introduce new users of ZTFL to the language and also describe our implementation of ZTFL for others looking to tackle similar problems.

Lexical Elements of ZTFL

We will first begin with an explanation of the superficial details of what goes into a ZTFL program. In ZTFL, whitespace is ignored aside from separating distinct lexical tokens in the language. Thus, the fragments "`word`

`word`", "`word\tword`", and "`word\nword`" are identical within a ZTFL program, but are not identical to the fragment "`wordword`". This is in contrast to Python, wherein whitespace also carries semantic information about scope. In ZTFL, scope information is provided by curly brackets, "{" and "}", and individual statements are separated by semicolons, "`;`". These C like elements are used to simplify the task of parsing ZTFL programs, as copying Python's indentation dependent scoping would have required a bit more work to parse properly. ZTFL is also case insensitive for all tokens not contained within a string, so the fragments "`word`", "`WORD`", "`wOrD`", etc. are identical within a ZTFL program, though "`"word"`" and `"WORD"`" are not identical to each other.

Case insensitivity was introduced to simplify the parsing of reserved keywords in ZTFL. Currently there are 12 reserved keywords in ZTFL:

"`observation`", ""`filteron`", "`annotate`", "`test`",
"`while`", "`for`", "`if`", "`else`",
"`and`", "`or`", "`not`", "`in`"

ZTFL variable identifiers may not be chosen from the set of reserved keywords, but are otherwise determined by the regular expression "[a-zA-Z_][a-zA-Z0-9_]*", in keeping with Python's rules for identifier names. Thus, identifiers must start with either a letter or an underscore, which can then be followed by any combination of letters, numbers, and underscores.

ZTFL has three fundamental data types: booleans, numbers, and strings. The ZTFL booleans are simply "`true`" and "`false`". ZTFL numbers are determined by the regular expression "\d+\.?\d*", so all numbers must be preceded by a sequence of one or more numeric digits, followed optionally by a decimal point, which may be followed by futher numeric digits. So the fragments " `3` ", " `3.` ", and " `3.14` " represent numbers in a ZTFL program, while " `3.1.4` " and " `.14` " do not. ZTFL strings are any symbols contained within the scope of two quotation marks, as defined by the regular expression "\".*?\"". As stated above, ZTFL strings are exempted from the case insensitivity of the language as a whole and will retain their original casing.

Numerous symbolic operators are defined within ZTFL. In no particular order, these operators are:

“`==`”, “`!=`”, “`>=`”, “`<=`”, “`>`”, “`<`”,
“`+`”, “`-`”, “`*`”, “`/`”, “`**`”,
“`=`”, “`+=`”, “`-=`”, “`*=`”, “`/=`”

Finally, a few grouping and punctuating symbols are used in ZTFL:

“`{`”, “`}`”, “`(`”, “`)`”, “`[`”, “`]`”, “`:`”, “`;`”

These represent all of the rules related to the formation of valid lexical elements in ZTFL.

Context Free Grammar of ZTFL

We will now describe a context free grammar of ZTFL that defines all valid ZTFL programs. As a quick summary of how to interpret the context free grammar, each statement within the grammar outlines a possible production from an abstract non-terminal (in English, a non-terminal could be a noun, a verb, or a sentence) to one or more possible sets of abstract non-terminals and concrete terminals (in English, 'dog', 'bark', or noun verb). A program that the user writes is entirely composed of terminals and for the program to be considered valid, it must be attainable by some sequence of productions within the context free grammar. Here, non-terminals will be represented in *italic font*, while terminals will be represented by `typewriter font`. $\underline{\textit{Underlined italic font}}$ is used to represent lexical elements recognized by the regular expressions above. Variable idenitifiers, numbers, booleans, and strings will therefore be represented by $\underline{\textit{name}}$, $\underline{\textit{number}}$, $\underline{\textit{bool}}$, and $\underline{\textit{string}}$, respectively. Multiple productions that can result from the same original non-terminal will be set on new lines.

Each ZTFL program is an instance of the *statementlist*, which may decompose into a combination of the *statementlist* and the *statementblock*.

$$\begin{aligned} \textit{statementlist} \; := \; & \textit{statementlist statementblock} \\ & \textit{statementblock} \end{aligned}$$

```
statementblock := {statementlist}
                  while (expression) {statementlist}
                  if (expression) {statementlist}
                  if (expression) {statementlist} else {statementlist}
                  for (expression; expression; expression) {statementlist}
                  for name in expression {statementlist}
                  statement ;


     statement := expression
                  filteron expression
                  annotate string expression


    expression := (expression)
                  expression[expression]
                  not expression
                  -expression
                  expression + expression
                  expression - expression
                  expression * expression
                  expression / expression
                  expression ** expression
                  expression == expression
                  expression != expression
                  expression <= expression
                  expression >= expression
                  expression < expression
                  expression > expression
                  expression and expression
                  expression or expression
                  expression in expression
                  name = expression
                  name += expression
                  name -= expression
                  name *= expression
                  name /= expression
                  name(expression)
                  name
                  number
                  bool
                  string
                  observation
```

The Semantic Evaluation of a ZTFL Program

A valid ZTFL program will be transformed by the ZTFL compiler into a string of Python to be executed by the compiler's parent program at a later time. The mathematical operations desired by the user are not evaluated within ZTFL or by the ZTFL compiler and are instead translated into their Python equivalents and handed off for the parent progam to eventually sort out. The ZTFL compiler internally produces a tree representing an input program's structure as determined by the grammar described above with nodes for each non-terminal in the program. The compiler descends the tree in a depth-first manner, applying a semantic evaluation function, σ, to each node as it traverses the tree. The semantic evaluation function takes a node of the trees as an input (I'm running out of convenient bracketing symbols, let's use # node# to represent a node in the tree) and outputs a plain text string. Since Python also requires whitespace indentation, I will use the modifier * to represent a properly indented string (in this case, all lines within indented by four spaces). String concatenation will be represented by the operator $\|$. Following the format of the context free grammar above, the semantic evalutation function is defined as follows:

$$\begin{aligned}\sigma(\#\textit{statementlist}\#) :=\,& \sigma(\#\textit{statementlist}\#) \,\|\, \backslash\mathtt{n} \,\|\, \sigma(\#\textit{statementblock}\#)\\ & \sigma(\#\textit{statementblock}\#)\end{aligned}$$

$$\begin{aligned}\sigma(\#\textit{statementblock}\#) :=\,& \sigma(\#\textit{statementlist}\#)\\ & \sigma(\#\mathtt{while(}\textit{expression}\mathtt{)}\ \{\textit{statementlist}\}\#)\\ & \sigma(\#\mathtt{if}\ \mathtt{(}\textit{expression}\mathtt{)}\ \{\textit{statementlist}\}\#)\\ & \sigma(\#\mathtt{if}\ \mathtt{(}\textit{expression}\mathtt{)}\ \{\textit{statementlist}\}\ \mathtt{else}\ \{\textit{statementlist}\}\#)\\ & \sigma(\#\mathtt{for}\ \mathtt{(}\textit{expression};\ \textit{expression};\ \textit{expression}\mathtt{)}\ \{\textit{statementlist}\}\#)\\ & \sigma(\#\mathtt{for}\ \underline{\textit{name}}\ \mathtt{in}\ \textit{expression}\ \{\textit{statementlist}\}\#)\\ & \sigma(\#\textit{statement}\#) \,\|\, \backslash\mathtt{n}\end{aligned}$$

$$\begin{aligned}\sigma(\#\mathtt{while}\ \mathtt{(}\textit{expression}\mathtt{)}\ \{\textit{statementlist}\}\#) := \mathtt{while}\ \,& \|\, \sigma(\#\textit{expression}\#) \,\|\, \mathtt{:}\ \ \backslash\mathtt{n}\\ & \|\, \sigma(\#\textit{statementlist}\#)^*\end{aligned}$$

$$\begin{aligned}\sigma(\#\mathtt{if}\ \mathtt{(}\textit{expression}\mathtt{)}\ \{\textit{statementlist}\}\#) := \mathtt{if}\ \,& \|\, \sigma(\#\textit{expression}\#) \,\|\, \mathtt{:}\ \ \backslash\mathtt{n}\\ & \|\, \sigma(\#\textit{statementlist}\#)^*\end{aligned}$$

$$\begin{aligned}\sigma(\#\mathtt{if}\ \mathtt{(}\textit{expression}\mathtt{)}\ \{\textit{statementlist}_1\}\ \mathtt{else}\ \{\textit{statementlist}_2\}\#) := \mathtt{if}\ \,& \|\, \sigma(\#\textit{expression}\#) \,\|\, \mathtt{:}\ \ \backslash\mathtt{n}\\ & \|\, \sigma(\#\textit{statementlist}_1\#)^*\\ & \|\, \mathtt{else:}\ \ \backslash\mathtt{n} \,\|\, \sigma(\#\textit{statementlist}_2\#)^*\end{aligned}$$

$$\sigma(\#\texttt{for (}expression_1\texttt{; }expression_2\texttt{; }expression_3\texttt{) }\{statementlist\}\#) := \sigma(\#expression_1\#) \parallel \texttt{\textbackslash n} \\ \parallel \texttt{while } \parallel \sigma(\#expression_2\#) \parallel \texttt{:} \quad \texttt{\textbackslash n} \\ \parallel \sigma(\#statementlist\#)^* \\ \parallel \sigma(\#expression_3\#)^* \parallel \texttt{\textbackslash n}$$

$$\sigma(\#\texttt{for } \underline{name} \texttt{ in } expression\ \{statementlist\}\#) := \texttt{for } \parallel \sigma(\#\underline{name}\#) \parallel \texttt{ in } \parallel \sigma(\#expression\#) \parallel \texttt{:} \quad \texttt{\textbackslash n} \\ \parallel \sigma(\#statementlist\#)^*$$

$$\sigma(\#statement\#) := \sigma(\#expression\#) \parallel \texttt{\textbackslash n} \\ \sigma(\#\texttt{filteron } expression\#) \\ \sigma(\#\texttt{annotate } \underline{string}\ expression\#)$$

$$\sigma(\#\texttt{filteron } expression\#) := \texttt{filteron = } \parallel \sigma(\#expression\#) \parallel \texttt{\textbackslash n}$$

$$\sigma(\#\texttt{annotate } \underline{string}\ expression\#) := \texttt{annotations[} \parallel \sigma(\#\underline{string}\#) \parallel \texttt{] = } \parallel \sigma(\#expression\#) \parallel \texttt{\textbackslash n}$$

$$\sigma(\#expression\#) := expression \\ \sigma(\#\underline{name}\texttt{(}expression\texttt{)}\#) \\ \sigma(\#\texttt{observation}\#)$$

$$\sigma(\#\underline{name}\texttt{(}expression\texttt{)}\#) := namespace(\underline{name}) \parallel \underline{name} \parallel \texttt{ (} \parallel \sigma(\#expression\#) \parallel \texttt{)}$$

Where $namespace(\underline{name})$ returns `math.` for all input values of $\underline{name}$ with a single exception. $namespace(\texttt{len})$ returns an empty string.

$$\sigma(\#\texttt{observation}\#) := \texttt{current\_observation}$$

After the ZTFL compiler produces a translation of the original ZTFL program, it finally appends prefixing and suffixing strings to the program to produce a Python executable string. This final operation acts as follows:

$$\sigma(program) := prefix \parallel \sigma(\#statementlist_{head}\#)^* \parallel suffix$$

The prefixing string is simply:

```
def compiledFunction(current_observation):\n ||
    filteron = False\n ||
    annotations={}\n
```

The suffixing string is simply:

```
return filteron,annotations\n
```

These relations fully specify the current behavior of the ZTFL compiler.

An Example Program

We will now provide an example ZTFL program and its resulting Python translation to illustrate the general behavior of the ZTFL compiler. The scientific content of the program itself is quite meaningless. We suppose that a completely arbitrary example is somewhat well suited for demonstrating the ability of ZTFL in handling all of the possible unknown physically meaningful calculations desired by its users ("Well, if it can work with all of that nonsense, it surely will work for my uses...").

Original ZTFL program:

```
grueBleen = False;
hasGrue = False;
hasBleen = False;
transitionDate = 2458022;
maxGrue = 0.0;
maxBleen = 0.0;
prevCandidates = observation["prv_candidates"];
m_now = observation["candidate"]["magpsf"];
t_now = observation["candidate"]["jd"];
b_now = observation["candidate"]["fid"];
if (b_now == 1){
    if (t_now < transitionDate){
        hasGrue = True;
        maxGrue = m_now;
    }
    else {
        hasBleen = True;
        maxBleen = m_now;
    }
}
for candidate in prevCandidates {
    maxBleen = candidate["magpsf"];
    if (candidate["fid"] == 1){
        if (candidate["jd"] < transitionDate) {
            if ( hasGrue ){
                if (candidate["magpsf"] < maxGrue) {
                    maxGrue = candidate["magpsf"];
                }
            }
            else {
                hasGrue = True;
                maxGrue = candidate["magpsf"];
            }
        }
        else {
            if ( hasBleen ){
                if (candidate["magpsf"] < maxBleen) {
                    maxBleen = candidate["magpsf"];
                }
            }
            else {
                hasBleen = True;
                maxBleen = candidate["magpsf"];
            }
        }
    }
}
grueBleen = hasGrue and hasBleen;
annotate "grue" maxGrue;
annotate "bleen" maxBleen;
filteron grueBleen;
```

Python translated output:

```
def compiledFunction(current_observation):
    filteron = False
    annotations={}
    gruebleen = False
    hasgrue = False
    hasbleen = False
    transitiondate = 2458022
    maxgrue = 0.0
    maxbleen = 0.0
    prevcandidates = current_observation['prv_candidates']
    m_now = current_observation['candidate']['magpsf']
    t_now = current_observation['candidate']['jd']
    b_now = current_observation['candidate']['fid']
    if (b_now == 1):
        if (t_now < transitiondate):
            hasgrue = True
            maxgrue = m_now
        else:
            hasbleen = True
            maxbleen = m_now
    for candidate in prevcandidates:
        maxbleen = candidate['magpsf']
        if (candidate['fid'] == 1):
            if (candidate['jd'] < transitiondate):
                if (hasgrue):
                    if (candidate['magpsf'] < maxgrue):
                        maxgrue = candidate['magpsf']
                else:
                    hasgrue = True
                    maxgrue = candidate['magpsf']
            else:
                if (hasbleen):
                    if (candidate['magpsf'] < maxbleen):
                        maxbleen = candidate['magpsf']
                else:
                    hasbleen = True
                    maxbleen = candidate['magpsf']
    gruebleen = hasgrue and hasbleen
    annotations['grue'] = maxgrue
    annotations['bleen'] = maxbleen
    filteron = gruebleen
    return filteron,annotations
```

Implementation of the ZTFL Compiler

The ZTFL compiler is curretly built using the Python Lex-Yacc module (PLY). For the GROWTH marshal, we embed the compiler as a method of a FilterInterpreter object. After the FilterInterpreter compiles a ZTFL program, the Python translated output string is saved as a property of the FilterInterpreter instance. To render the Python translated output as an executable function within the parent program, another method within the FilterInterpreter instance is called. This final method simply appends the suffix `self.compiledFunction = compiledFunction\n` to the Python translated string and calls Python's built-in `exec()` function on the compiler's output. `exec` creates a local defined function called `compiledFunction` within the scope of the method, as specified by the original ZTFL program, and set's the FilterInterpreter's `compiledFunction` property to the newly defined function. This effectively sets up `compiledFunction` as a method of the FilterInterpreter object, which can then be easily called by the FilterInterpreter's parent program.

The general use is as follows. A set of user defined ZTFL programs are created to define each user's broad criteria for what they would find interesting to look into. A FilterInterpreter object is created for each ZTFL program and compiles its corresponding program. A list of candidate data is brought in with elements whose data is accessible using either Python's list or dictionary syntax (they do not need to strictly be dictionaries or lists, but simply accessible using the same syntax as with a dictionary or a list). For each candidate within the candidate list, the candidate is used as the `current_observation` argument for the newly created `compiledFunction` method of each FilterInterpreter instance. Each `compiledFunction` returns two outputs, `filteron` and `annotations`. `filteron` just gives a yea or nea response about whether the user is interested in the input candidate. `annotations` is a dictionary that can contain the user's intermediate scratch work used in deciding whether they are interested in the candidate. If a user is not interested in a given candidate, the returned `annotations` are simply discarded and the user is never notified of the existence of the rejected candidate. If the user is interested in a candidate, we then pass the candidate over to the user along with their filter's computed annotations. These annotations can then be used as the basis for sorting and prioritizing a given user's list of interesting candidates (perhaps not necessary when a user can personally review all of their interesting candidates, but certainly very useful if the influx of candidates is particularly overwhelming).

The General Problem

The iPTF, GROWTH, and (eventual) ZTF marshals exist as tools for their users to view, retreive, and contribute data on astrophysical objects according to the users' own interests and desires. With improved instruments, the rate of objects being added to these marshals has increased and is expected to continue increasing. In the iPTF marshal, all users had the ability to interact with every object added to the marshal. As the rate of object detections increases, it is expected that users would need to devote more time to simply finding their given objects of interest within a growing pile of irrelevant junk (from their perspective), leaving less time to study those objects of interest. It is clear that some method of filtering incoming objects is needed to limit the clutter of the marshal and maintain the productivity of the marshal's users.

In addition to the increased rate of object input to the marshal, the marshal's user base has expanded and diversified. The iPTF transient marshal was intended to be used by a relatively unified group of researchers and therefore has some in-built bias to be especially useful to those researchers (there is a notable emphasis on optical observations and, of course, time domain astronomy specific parameters and terminology). We expect that the user base will expand to include users who are interested in a variety of different types of objects and rely on their own unique sets of physical parameters and terminology. One set of the marshal's users might find no use or interest in the objects studied by another set of users. They might also find no reason to use the parameters or terminology of any other set of users on the marshal. So it is also clear that a method of providing user specific filtering is needed.

The problem faced by the current marshal is straightforward. We have a single input stream of objects and some set of users who have different needs and we need to hand off objects to the users who will find them interesting. It appears that ZTFL can effectively resolve this problem.

Alternative Solutions

There are a few possible alternatives to using ZTFL. I will briefly run through these and (perhaps a bit unfairly) state why the ZTFL set up is superior. All of these alternatives can, in principle, pass off the right objects to the right people, but their implementations carry some undesirable risks or obligations for the maintenance of the marshal as a whole.

A very simple alternative is to create a pre-defined set of filters and allow our users to use some combination of those in-built filters. A weak objection to this set up is that the pre-defined filter set will dictate some aspect of the user's behavior (in the extreme, if we only give a filter on object brightness, our users will be forced into inheriting some emphasis on object brightness). A stronger objection to this set up is that the marshal would be obligated to maintain a useful filter set. If the user's needs change or if a new observational parameter becomes included with our input data, we would need to update the pre-defined filters. ZTFL leaves the filter design entirely up to the users (with some limit on filter complexity). With ZTFL, the responsibility of useful filter design falls entirely on our users. If our input stream has additional parameters added later, our users will simply need to add references to the added parameters to their filters, if they want to check that parameter. While we would be obligated to update our users about changes to the structure of our input data, we would not be obligated to modify the marshal's code base. With ZTFL, the code we use to implement filtering on the day we roll out a marshal can be identical to the code used on the day the marshal is retired.

Another alternative would be to allow user's to define their filters using raw Python. This would allow users to build whatever filter they could possibly need. This would also allow our users to build filters that have free reign within the marshal's backend. It's simply too risky to implement. With ZTFL, the language is intentionally limited to prevent a user defined filter from affecting anything other than the decision to show an object to the user.

We could also attempt to render raw Python safe to execute. Perhaps we would allow our users to submit Python code, and then attempt to parse the submitted code to detect and/or replace risky fragments. This is similar to the approach taken with ZTFL, though from the opposite direction. This alternative begins with the set of all valid Python programs and then begins to carve out all of the programs that we would not want executed. ZTFL essentially starts from the bottom and defines all of the programs that we would allow to be executed. Using a small DSL like ZTFL allows us to define a language wherein the only valid programs that will be passed through the ZTFL compiler without error are those that we would be comfortable executing for our users. If we were to attempt to parse or modify arbitrary Python code, we would need to be sure that no risky program could be passed on undetected. With ZTFL, it can be more easily proven that only acceptable programs are passed through the compiler. ZTFL's approach allows us to make sure that we, in some sense, understand everything written within a filter and only execute the filters that we understand.

The Drawbacks of ZTFL

ZTFL does come with a potential downside that I feel should be highlighted. Because the use of this sort of DSL is not particularly common for tools in astrophysics and is also not a standard part of astrophysics education, it is potentially less maintainable than other aspects of the marshal codebase. ZTFL could become a black box system. This guide is perhaps a bit too specific for the users of ZTFL. Hopefully it provides enough explanation and details about how ZTFL is built to allow future maintenance and modification of ZTFL for use on the marshal. The reading list below might also help where this guide fails.

Reading List

Theories of Programming Languages, John Reynolds. This book was particularly helpful and provides some excellent examples of specifying and defining aspects of programming languages. ZTFL is an extension of the imperative language described in Chapter 2.

Compilers Principles, Techniques, and Tools, Aho et al.. This book contains some handy explanations about compilers in general. Although it more describes how PLY works, rather than ZTFL, it is less formal and more approachable than Reynolds.

The PLY documentation, `http://www.dabeaz.com/ply/ply.html`. The PLY documentation is, obviously, quite handy when starting to use PLY. ZTFL also started out as one of the calculator examples and gradually expanded to become a ZTFL to Python transpiler.