T2T - Text to Text Transpiler

T2T creates DSLs.

T2T reduces the amount of work required to build a DSL. You should be able to create a new DSL in an afternoon.

You only need to create a grammar and a corresponding rewrite specification. You use *t2t* to translate the new DSL syntax into code for any existing programming language (like Python, Haskell, etc), then run the generated code.

This means that you write a lot less code and let the computer do the rest of the coding work for you.

The grammar is written in OhmJS format and the rewrite specification is described below.

T2t works in two ways

1. It can transpile a source program written in a newly invented DSL (*stand-alone mode*)
2. It can create a Javascript program that is the DSL. To run the DSL, you currently need to use node.js and input redirection. (*gen mode*)

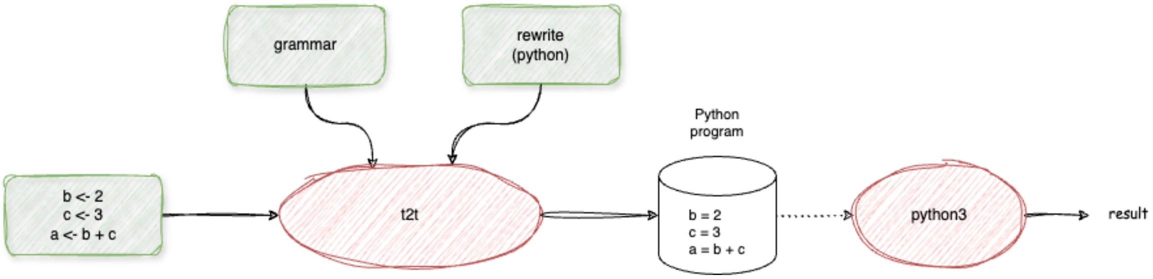

FIG. 1 STAND-ALONE MODE (DSL TRANSPILED TO PYTHON)

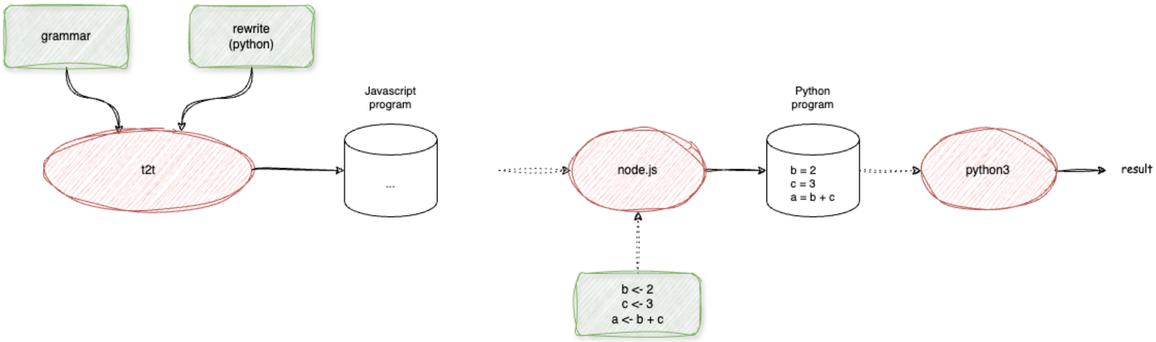

FIG. 2 GEN MODE (DSL TRANSPILED TO PYTHON)

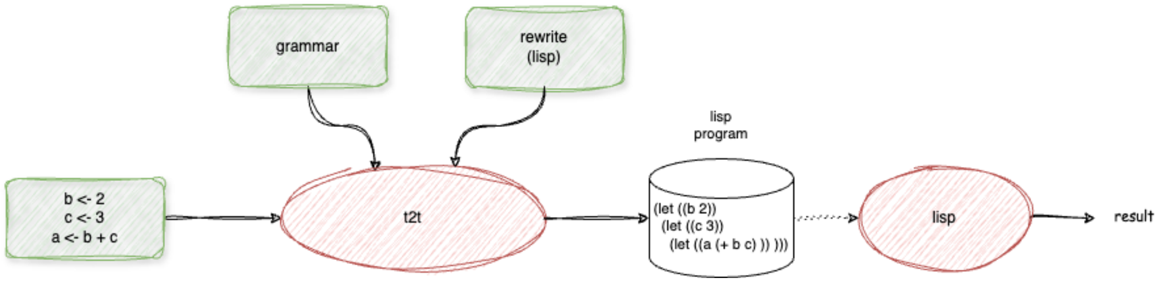

FIG. 3 STAND-ALONE MODE (DSL TRANSPILED TO LISP)

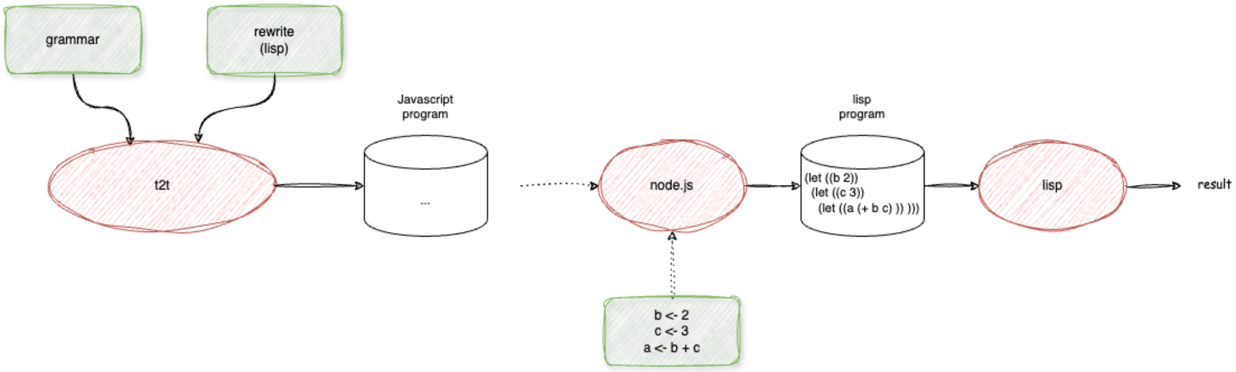

FIG. 4 GEN MODE (DSL TRANSPILED TO LISP)

Note that the same grammar is used to generate Python and Lisp.

The generated DSL program is written in Javascript and uses OhmJS. The fact that *t2t* uses Javascript does not mean you have to use, nor target, Javascript. You can write *t2t* transpilers that generate any language you want, like Python, Lisp, Go, WASM, etc.

T2t does not have builtin code generators. It relies on the .rewrite file to generate code. The programmer needs to write the .rewrite file. Examples of .rewrite files are included and can be used as sourdough starters for generating new .rewrite files.

Usage

To compile and run the DSL in stand-alone mode, you use node.js.

For example, for stand-alone mode:

```
	node t2t.mjs test.grammar test.rewrite test.txt
```

- where `t2t.mjs` is the t2t processor
- where `test.grammar` is the specification of pattern matching rules (called a "grammar")
- where `test.rewrite` is the specification of the rewriting rules
- where `test.txt` is some code written using the new DSL syntax

The output of running `text.txt` goes to `stdout`.

If the third argument is "-", then the program is read from standard input.

Power users can use gen mode

```
      node t2t.mjs test.grammar test.rewrite >newdsl.mjs
```

which generates a stand-alone DSL. To run the DSL, you invoke it with node.js (again):

```
      node newdsl.mjs test.txt
```

SCNs Are Nano-DSLs

Many DSLs are too much like full-blown compilers, i.e. mega-projects in their own right.

I use the term *SCN* to mean *light-weight DSL*. *SCN* Stands for Solution Centric Notation.

I believe that it is simpler to carve up a problem into smaller parts - i.e. divide-and-conquer. Then, address each part of the problem as a stand-alone problem with, possibly, its own nano-DSL. A DSL is actually just an API, but, with a nicer syntax.

An SCN is a nano-syntax for a nano-API for a small part of a given problem. You don't need to worry if the SCN is "Turing complete", you only need to ensure that the SCN is useful for expressing and solving a sub-portion of some larger problem.

Syntactic Composition

One goal of programming is to figure out how to compose programs out of smaller parts.

I believe that a fruitful approach is to perform composition using syntaxes (plural). Create one SCN for each sub-part of a problem, then compose all of the parts into a larger solution. Of course, this process can be recursive, i.e. divide up a problem into parts, then divide each part into parts, and so on.

This approach creates *many* nano-syntaxes within a single problem domain. Many programmers think that this creates too much thinking and too much learning. I counter that *any* interesting problem requires lots of thinking and learning, anyway. Using General Purpose Programming languages, though, tends to hide and obfuscate the fruits of such thinking. Readers need to reverse-engineer code in order to understand what the original designer was thinking and wanted to accomplish. When the original code is written using many little high-level SCNs, the design intent[1] can be captured in a rigorous manner and alleviates future readers from having to reverse-engineer the code. Capturing DI as part of the automated workflow, i.e. transpiling DI to running code, results in better capture of intent than simply using comments. Comments go out of date, whereas automated DI transpilation tracks the program more closely.

I suspect that the approach of using many little SCNs is tainted by the reality that - until recently - creating DSLs was considered to be an onerous task, on the order of being mega-projects in themselves. Reducing the effort to create new nano-DSLs (SCNs) by at least one order of magnitude (i.e. 10x), changes the way that problem solving can be approached.

[1] I call this DI, for Design Intent

T2t.mjs

The guts of the SCN-generator consist of a single Javascript program called *t2t.mjs*.

This program, *t2t.mjs*, happens to use OhmJS.

OhmJS parses the input and calls various routines (written in Javascript) to walk the parsed-tree[2] and to take appropriate actions by building strings. The actions are specified by programmers in the *.rewrite* specification file.

Makefile

The included *Makefile* is a template that you can modify and use for building your own SCNs.

T2T Specification (*.grammar and .rewrite files)*

To run the *t2t.mjs* tool, you need to supply two specification files

1. A grammar, in OhmJS format, see the OhmJS documentation[1]
2. A rewrite specification, described below.

A rewrite specification consists of two sections

1. An optional list of parameters. Parameters are reentrant variables that are created before tree-walking. Don't worry if you don't understand this bit. Parameters are needed only for complicated SCNs. Just ignore and don't use parameters until you need to build bigger and more complicated SCNs.
2. A rewrite specification that corresponds to the grammar and shows how each pattern matched by the grammar should be rewritten.

T2T deals only with strings of characters. It reads characters from input source files and produces a string of output characters. Every sub-rule (parse node) produces a single string and, maybe, creates some side-effects by calling subroutines supplied in the *support.mjs* file.

This *sounds* wildly restrictive, but, in practice, is all that you need. In fact, I would argue that this is the epitome of functional programming. All of the "rules and restrictions" in functional programming are geared towards rewriting text code (lambdas and binding and referential transparency). Functional programming forbids the use of side-effects, but, *t2t* allows side effects by insisting that the side-effects be well contained and isolated and hidden. In FP, you can create and mutate state by passing a blob of state as parameters to functions, whereas in *t2t* you don't need such complication.

The idea here, is that you use the grammar to pattern-match inhaled input source code, then rewrite the source code into some already-existing programming language (like Javascript, Python, WASM, Haskell, etc.) and use the targeted language's workflow to do the rest of the work.

[2] After parsing, OhmJS produces a tree of information and match captures. This is a CST (concrete syntax tree) which is just a culled subset of the AST (abstract syntax tree) written in OhmJS grammar syntax (akin to BNF).

When necessary, you can set up stacks of *parameter* variables that can be used by parse nodes deeper down in the parse tree. T2t creates and destroys parameter variables in a scoped manner, alleviating programmers from having to deal with heap/scoping/globals issues.

The idea is to simplify programming by using a code generation pipeline instead of conflating many unrelated issues within the same general purpose language.

T2t is but a single part of a simplified pipeline. You can even string several SCNs together one after the other to chip away at creating a problem solution.

Grammar

The grammar used for specifying new SCNs is based on OhmJS.

The grammar looks a lot like BNF[2].

The grammar is more powerful than REGEX[3], since it is based on PEG[4] technology.

Yet, the grammar is more readable than REGEX.

See the documentation for OhmJS.

The grammar provides space-skipping and grammar rule parameterization and left-recursion (as per OhmJS). Because of these features, the grammar is more powerful and cleaner than most other PEG libraries.

Caveats

The grammar used in *t2t* .grammar files is only a subset of the full-blown grammar for OhmJS.

T2T allows for only one level of bracketing of predicates, i.e. you can specify "(A B)+", but you can't specify something more complicated, like "(A B+)+". In practice, such extra complication is rarely used and this restriction is almost never a problem.

T2T grammars can contain alternation branches.

- If branches are not named, then *all* branches in a rule must not be named, and, the set of branches must be enclosed in parentheses "(... | ... | ...)"
- If branches are named, then *all* branches must be named, and, the set of branches must not be enclosed in parentheses "... -- *name1* | ... -- *name2* | ...".

This differs in syntax slightly from the possibilities allowed by OhmJS.

In *t2t,* alternation branches must be, either,

- Parenthesized "*name* = (... | ... | ...)", or,
- Named and not parenthesized and begin with "|" characters "*name* = | ... -- *name1* | ... -- *name2* | ..."

Again, this differs in syntax, slightly from what is allowed in OhmJS. The syntax used by *t2t* alternation is valid for OhmJS, but certain variants of OhmJS syntax are not valid in *t2t*.

In other words, restating the preceding restriction, in cases where branch sequences are of unequal length, each branch must be named using OhmJS "-- *case-name*" syntax. Unlike in

OhmJS, if *any* branch is named in an alternation, then *all* branches must be named. This restriction is valid OhmJS syntax. OhmJS allows one branch to remain unnamed - this is not valid *t2t* syntax, though.

In early manifestations of *t2t,* syntax validity might not be fully checked. The programmer is expected to ensure that valid syntax is used. Invalid syntax might result in undefined behaviour. I.E. don't make mistakes, for now, you are on your own without consistency-checking built into the tool.

I will document other caveats as I remember them...

Scoped Parameter Declaration

T2T specifications allow programmers to create reentrant "variables" (on stacks) that act to pass information *down* during tree-walking.

This is much like specifying parameters to *functions* in General Purpose Programming languages, like Python, Javascript, Haskell, etc.

Scoped parameters are optional and don't need to be used in most cases. You can ignore them until the need arises.

Each scoped parameter is declared using the syntax

% parameter *name*

As described below, scoped parameters come into existence at the beginning of a scope and disappear at the end of the same scope.

How Are Scoped Parameters Implemented?

Operationally, each rewrite rule uses multiple stacks - one stack for each scoped parameter - to manage scoped parameters. Rewrite rules can bind values to the top-of-stack for any parameter and can query the value during tree-walking. At the end of a rule, the stacks are popped, destroying the most recent values of all scoped parameters.

An example usage of scope parameters, might be to pass the name of the current routine being parsed down into the bowels of the tree walker. A parse tree is recursively walked after a successful parse, in a depth-first manner. If any child node in the tree needs to use the name of the current routine, it can query a scoped parameter to pull out the appropriate name. Since children nodes are visited first, they cannot easily access information from higher up in the tree, unless the information is placed into a scope parameter before the tree walk is executed. In this example, the name of the "current routine" is bound (pushed onto the named stack) to a specific scoped parameter and can be accessed by children lower down in the tree. Functional programming does something similar by using parameter binding and the callstack. T2T differs only in that multiple stacks are used in a more explicit manner.

Rewrite Specification for test2t2t

I will discuss .rewrite syntax by way of examples.

Simple t2t Example

The testt2t example demonstrates a very simple way of using *t2t*.

Usage:

```
make testt2t
```

The test.txt test source is:

```
a
```

The grammar test.grammar is:

```
t2t  {
  Main = "a"
}
```

This grammar specifies one rule called *Main*. It expects to match a single character *a*.

If it matches exactly one character *a*, and no other characters remain, e.g. no newlines, etc., then a parse tree - a CST (Concrete Syntax Tree) - is created and the rewrite rules are invoked using the CST nodes as parameters to the rules.

Because this example is very simple, the CST is not very complicated and contains only one node - the match capture information for the character *a*. The match capture information is stored in an internal format that is used by OhmJS. The *t2t* tool insulates programmers from needing to know what information is contained in the CST.

The rewrite rules are specified as

```
% rewrite
t2t  {
  Main [c] = ‘hello world «c»’
}
```

Again, due to the simplicity of this example, there is but one rewrite rule.

There must be one rewrite rule for every grammar rule. Each rewrite rule must have *exactly* the same name as the matching grammar rules. Grammar rules that include alternation have different requirements, described earlier.

This rewrite rule - *Main* - expects to receive one CST node and names that received node “`c`”. When the rule fires, it evaluates the rewrite string on the right hand side of the “=“. The rewrite rule creates a return string from several raw characters “hello world “ and by evaluating its argument *c* and inserting (interpolating) the resultant string into its own return string. The final result is “`hello world a`”. This result is returned from the *Main* rewrite rule. In this simple case the result is returned to the top level of the *t2t* engine.

Less-Simple t2t Example for test2

The *test2* example demonstrates the use of parameters and passing information downwards during the tree walk. This test, also, demonstrates using t2t.mjs to create a stand-alone DSL called test2.mjs.

Usage:

```
make test2t2t
```

The test2.txt test source is:

```
a;b;b;bcddddd
```

The grammar `test2.grammar` is:

```
example {
  Main = "a" (";" "b")+ "c" "d"+
}
```

This grammar says
1. First, Match the letter “a”
2. Then, match 1 or more “;b”s
3. Then, match the letter “c”
4. Then, match 1 or more letter “d”s.

Extra spaces are allowed because the rule name *Main* is capitalized.

And, the rewrite specification is:

```
% parameter paramA
% parameter paramB
% parameter paramC

% rewrite example {
Main [a (_semis b)+ c d+]=
  ⎡ ⎨print ‘pre down a=«a» _semis=«_semis» b=«b» c=«c» d=«d»’⎬
    ⎡ paramA=‘«a»’
      ⎡ paramB=‘«b»’
        ⎡ paramC=‘«c»’
          ⎡ ⎨print ‘hello’⎬
            ‘... ⎨print2 ‘middle’ ‘2nd arg’⎬
«a»«_semis»《paramB》«c»«d»...’
          ⎦
        ⎦
      ⎦
    ⎦
  ⌋
}
```

This set of rules declares 3 parameters *paramA, paramB,* and*, paramC*. The parameters are re-entrant, i.e. they are implemented as stacks instead of as single, mutable, overwriteable cells.

When the grammar successfully matches the *Main* rule, the *rewrite* rule for Main is activated.

Rewriting proceeds in the following manner...

The argument “a” is bound to the first match capture in the grammar and is tree-walked. In this simple example, the first capture is the single letter “a”.

The arguments *_semis* and *b* are bound to however many “;”s and “b”s were captured by the grammar. In the case of *test2.txt* there are 3 sequences of “;b”, so *_semis* is “;;;” and *b* is “bbb”.

The argument *c* is bound to the letter “c”.

The argument *d* is bound to “ddddd” (1 or more letter “d”s in the input file *test2.txt*).

Then, the right-hand side of the rule *Main* is activated.

The right-hand side contains five scopes, each bracketed by “ ⌈ ... ⌋ ”. A scope contains two items

1. A support function call, or, a binding.
   - A function call is bracketed by “ ⎨ ... ⎬ ”.
   - A binding is a name, an “=“ character, and, a rewrite string. The *name* must be the name of a parameter (stack). The rewrite string is evaluated then pushed as a binding to the *name* onto the stack called *name*. At the end of the scope, the binding is popped from the stack.
2. A scope can contain another scope, recursively, or, just a rewrite string.

A rewrite string is enclosed in Unicode quotes ‘...’ and can contain any of the following:

1. Raw characters.
2. Argument interpolations. An argument name bracketed by ‘«...»’
3. Parameter interpolations. A parameter (stack) name bracketed by “《...》”
4. Support function calls. A support function name followed by one or more comma-separate rewrite strings, all bracketed by “ ⎨ ... ⎬ ”. Note that, unlike in most popular programming languages, function calls do not use parentheses to signify argument lists. A function call consists of a function name, followed by a comma-separated list of strings, all surrounded by one set of brackets “ ⎨ ... ⎬ ”. The arguments to a function are strings that are interpreted in the same way as *rewrite strings*.

As written, this simple example, but useless, rewrite rule specification says that there is only one rewrite rule called *Main* which corresponds exactly with the grammar rule *Main*. The rewrite is nested five scopes deep. The deepest scope contains the actual rewrite.

```
‘... ⎨print2 ‘middle’ ‘2nd arg’⎬ «a»«_semis»《paramB》«c»«d»...’
```

Which is the same as several strings concatenated together, where variable values are converted to strings, and function return values are converted to strings.

```
“...” + support.print2(“middle”, “2nd arg”) + a + _semis + top_of_stack(paramB) + c + d + “...”
```

The first scope simply calls the support function *print* after evaluating (tree-walking) each of the four arguments to the rule.

```
⎨print ‘pre down a=«a» _semis=«_semis» b=«b» c=«c» d=«d»’⎬
```

The second scope evaluates the *a* argument, creates a string and pushes the string onto the parameter stack *paramA*.

Likewise, the third and fourth scopes bind strings to the parameter stacks called *paramB* and *paramC*.

The deepest scope calls the *print* support function with the constant string “hello”, and, creates a return string by evaluating the given rewrite string.

```
⌈  ⎨print ‘hello’⎬
    ‘... ⎨print2 ‘middle’ ‘2nd arg’⎬ «a»«_semis»《paramB》«c»«d»...’
⌋
```

The parameter stacks are popped and the final result string “... a;;;bbbcddddd...” Is returned from the rule.

[*Design Note: I believe that each semantic element should be made explicit. In this case I give each kind of semantic element a unique syntax. String constant, function call, argument interpolation, parameter interpolation. One can always drape a new layer of syntax onto the constructs, possibly overloading the syntax with differing underlying semantics. Understanding the underlying semantics is important, while syntax is cheap. Making everything explicit, at a low level, makes it easier to generate code automatically.*]

In this simple example, since there is only one rule, the result string is returned to the top level of the transpiler engine. In more typical cases, each rule returns a string to the rule which invoked it by tree-walking (evaluating) it. Such sub-results end up being bound to the arguments to parent rules and, usually, participate in the creation and interpolation of result strings from the parent rules.

Which rules are actually evaluated is tied to the initial parse. Evaluation is done in a depth-first manner - the deepest child rule is evaluated first. The parse sets up a tree of sub-matches, where each sub-match corresponds to an argument of a parent rule.

The control flow for tree-walking evaluation is simple, but, difficult to explain in words. Perhaps a diagram of a part of a match-tree might help understanding:

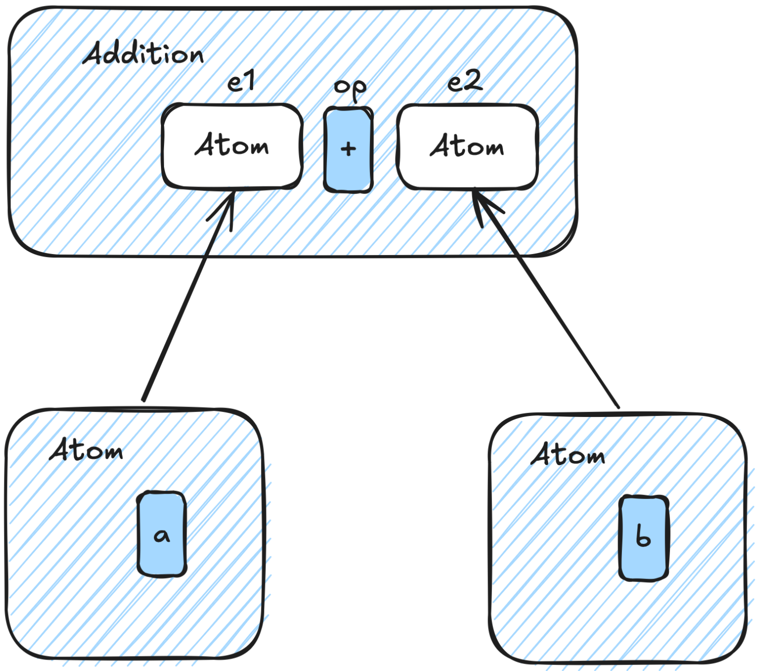

FIG. 5 SIMPLIFIED PARSE TREE

This is a stylized diagram, that does not reflect the way that actual grammars and parse trees would be written, It is only meant to illustrate how tree-walking evaluation evolves.

In Fig. 5, the arguments to *Addition, e1* and *e2* are holes that are not filled until the children *Atom* rules have been evaluated.

Once the holes are filled - by evaluating the atom sub-trees - the Addition rewrite rule can be fired.

The two *Atom* rules don't know that they are being evaluating by the *Addition* parent rule. The children rules cannot include information from "above", since the information has not yet been finalized. If context information is required by the children rules (*Atom*), the parent rule (*Addition*) must set up some context information before evaluating the children rules. In the case of *t2t,* this is done by pushing values onto *parameter* stacks.

The *t2t* tool provides a syntax - the *scope* syntax described above - to allow setting up context information before tree-walking occurs.

Rewrite Rule Details

The *rewrite* section begins with “% rewrite”.

There must be one rewrite rule for every grammar rule. (See below for rewriting OhmJS grammars rules that involve alternation that contain *case labels*).

Simplifying, a rewrite rule consists of several parts

1. A name
2. An argument list
3. =
4. A rewrite string specification.
   - Rewrite strings can, optionally, be enclosed in bracketed *scopes* that are preceded by scoped parameter definitions or support calls

Rule Name

The name must be exactly the same as the corresponding grammar rule.

Case is significant.

Note that OhmJS uses the first letter of a rule name to signify space-skipping (upper-case letter) vs. raw PEG behaviour (lower-case letter), called *syntactic* and *lexical* rules, respectively. See the OhmJS documentation for further details.

In the above example, the rewrite rule name is “Main”.

Argument List

A rule’s argument list (more often called a *parameter list* in function-based programming*,* but, I use the word *parameter* for something else in rewrite specifications ; anyways, the *values* of arguments come from “below” instead of from “above”, which implies that these arguments are different from regular function parameters) is bracketed by square brackets “[...]” and contains arbitrary names for each predicate matched by the corresponding grammar rule.

Unlike other PEG libraries, programmers are not given the choice of which matches to tag with names - *all* matches must be named[3].

Each parameter must be adorned with corresponding OhmJS grammar syntax, like iteration operators (+ / * / ?) and parentheses.

Parameter names adhere to Javascript standards, i.e. the first character must be a letter or an underscore, following characters may be letters, underscores or digits. Spaces and other characters are not allowed in names.

[3] This is a good thing. Regularity and normalization make for easier automation and for less distraction for programmers. Programmers are allowed to concentrate on devising grammars, instead of dealing with minutia like which parts of the rules will be captured and tagged with names. Ignoring unwanted captures is easy enough to do.

In the above example, the parameter list is:

```
[a (_semis b)+ c d+]
```

Which defines 5 parameters with names “a”, “_semis”, “b”, “c” and “d” and corresponds, syntactically, with the rule “Main” in the above grammar. Three of the parameters are iterations, in this case “+” iterations - “_semis”, “b” and “d”.

=

The left-hand side of a rewrite rule is separated from the right-hand side by a single “=“ character.

The left-hand side contains the rule name, the rule parameter list and the optional support routine call.

The right-hand side contains the rewrite string optionally nested in a hierarchy of scopes.

Rewrite string specification

A rewrite string is bracketed by Unicode quotes ‘...’.

A rewrite string is composed of

1. Raw characters
2. String interpolation of rule parameters, denoted by unicode brackets around a rule parameter name “«...»”
3. String interpolation of scoped parameters, denoted by unicode brackets around a scoped parameter name “《...》”. Note that these unicode brackets are different from the brackets used for interpolation of rule parameters[4].
4. String interpolation of a support routine call, “ ⎨ *routine-name* ‘...’ ‘...’ ... ⎬ ”. The routine must be defined two times in the file *support.mjs* and must return a string. The two variants of the function have two different prefixes - “*pre_”, “post_”*[5]*.* In this example, we need to define *pre_print(...),* and *post_print(...).* The *pre_* version of the function is called on entry to a scope, whereas the *post_* version is called on exit from the scope.

The above example contains examples of all four components (raw characters “...”, interpolation of rule parameters, e.g. “«c»”, interpolation of scoped parameters, e.g. “《paramB》” and a support call to a routine named “print2” that takes two arguments, respectively.

[4] The choice of brackets is not arbitrary. “«...»” must be used for interpolation of rule parameters, whereas “《...》” must be used for interpolation of scoped parameters.

[5] This will likely be changed in the future to supply an extra argument to the support function instead of mangling the name of the function.

The left-hand side of a rewrite rule is separated from the right-hand side by a single "=" character. The left-hand side contains the rule name, and, the rule argument list. The right-hand side contains the rewrite string optionally enveloped in nested scopes.

Scopes

Scopes are bracketed by the Unicode brackets " ⎡... ⎦ ".

The *first* item after the opening bracket must be either

- A binding to a scoped variable, e.g. "paramB='«b»'", or
- A call to a support routine " ⎨ print 'hello' ⎬ "

The *second* item after the opening bracket must be either

- a rewrite string, or,
- another bracketed scope

A scope consists of exactly two items. Following the items, a close-scope bracket must appear " ⎦ ".

The first item after the opening bracket is treated in a scoped manner:

- In the case of scoped variables, the binding remains in effect throughout the whole scope, but is undone when the scope is exited (operationally, the binding is *pushed* onto a distinct stack on entry to the scope and *popped* on exit from the scope).
- In the case of a call to a support routine, the routine must be defined twice, with prefixes "pre_" and "post_"[6]. The "pre_" version of the routine is called on entry to the scope and the "post_" version is called on exit from the scope.

Scopes are optional and used only in more complicated rewrites. In many cases, no scopes are needed and the right-hand-side of a rewrite rule consists solely of a single rewrite string.

The idea of *scopes* is to allow rules to pass context information *down* to sub-rules during tree-walking. The information is created (and stored in scoped-variable stacks) before recursive tree-walking is performed within a given rule. This is similar to the idea of passing parameters to functions in General Purpose Languages, but, uses a different syntax. The scoping syntax of *t2t* encourages distinct stacks to be used. A similar effect can be created using nested *Let* statements in functional languages like Lisp, but, has traditionally been implemented using only a single stack - the callstack - whereas the current *t2t* implementation uses one distinct stack for each *parameter*.

Rule Names for Grammar Alternation Cases

In some OhmJS grammars, rules can be specified as a set of alternations.

In simple cases, the alternation branches need not be named. This is possible when each branch matches predicate sequences that are all exactly the same length (often of length 1, in practice).

[6] It is likely that this detail will change to that of defining the routine once, but with an extra parameter specifying whether entry or exit is being performed.

In cases where branch sequences are of unequal length, each branch must be named using OhmJS “-- *case-name*“ syntax. Unlike in OhmJS, if any branch is named in an alternation, then *all* branches must be named. This restriction is valid OhmJS syntax. OhmJS, though, allows one branch to remain unnamed - this is not valid *t2t* syntax.

See the OhmJS syntax documentation for further elucidation.

The rewrite section must contain one explicit rule for each named branch. The branch rule name is a concatenation of the rule name, an underscore, and, the branch name.

For example, the OhmJS grammar snippet

```
    AddExp
        = AddExp "+" MulExp  -- plus
        | AddExp "-" MulExp  -- minus
        | MulExp
```

Must be written in *t2t* as

```
    AddExp =
        | AddExp "+" MulExp  -- plus
        | AddExp "-" MulExp  -- minus
        | MulExp             -- default
```

and the rewrite section must contain three different rules named

```
    AddExp_plus [...] ...
    AddExp_minus [...] ...
    AddExpr_default [...] ...
```

When a named branch is used in the grammar, the rewrite section must not contain a rule that is not suffixed with only the branch name, e.g. `AddExp [...] ...`

Support.mjs

The Javascript file *support.mjs* is imported by the generated transpiler *.mjs* code.

This file must create a Javascript namespace and export it. In the example code, the namespace is called “_”. The actual name used is arbitrary, but, the use of longer names will uglify the rewrite specification.

This file is simply standard Javascript and is usually very small in length. The example code exports only 12 functions. If you need to create many more exported functions, then it is likely that you are doing something in an overly-compilcated manner, or, need to use divide-and-conquer to an even greater extent.

Maintaining and Upgrading t2t

You don’t need to futz with the stuff described below, in order to simply use *t2t* to build new SCNs. Don’t read this section unless you intend to extend, maintain and rebuild *t2t*.

On the other hand, the self-compilation workflow demonstrates the use of a pipeline of *t2t* passes. The second pass is fairly trivial and simply replaces one chunk of the resulting file with another chunk.

T2t can rebuild itself by invoking `make self`.

The tail end of the generated *t2t.mjs* file needs to be replaced by a different tail. This is done by using yet another *t2t* post-pass that involves `self_boilerplate.grammar` and `self_boilerplate.rewrite`. In this case, the post-pass snips off the stock mainline code and replaces it with different mainline code. This is much like the Unix *sed* tool, except that Unicode characters are endured.

[Collaborators are welcome, contact me if interested].

Appendix - Generated Code For test2

The generated DSL `test2.mjs` is:

```
'use strict'

import {_} from './support.mjs';
import * as ohm from 'ohm-js';

let return_value_stack = [];
let rule_name_stack = [];

const grammar = String.raw`
    example {
  Main = "a" (";" "b")+ "c" "d"+
}

`;

let paramA_stack = [];
let paramB_stack = [];
let paramC_stack = [];
const rewrite_js = {
    Main : function (_a, __semis, _b, _c, _d, ) {
        let a = undefined;
        let _semis = undefined;
        let b = undefined;
        let c = undefined;
        let d = undefined;
        return_value_stack.push ("");
        rule_name_stack.push ("");
        _.set_top (rule_name_stack, "Main");
        paramA_stack.push (paramA_stack [paramA_stack.length-1]);
        paramB_stack.push (paramB_stack [paramB_stack.length-1]);
        paramC_stack.push (paramC_stack [paramC_stack.length-1]);
        a = _a.rwr ()
        _semis = __semis.rwr ().join ('')
        b = _b.rwr ().join ('')
        c = _c.rwr ()
        d = _d.rwr ().join ('')


        _.pre_print (`pre down a=${a} _semis=${_semis} b=${b} c=${c} 
d=${d}`);
        _.set_top (paramA_stack, `${a}`);
        _.set_top (paramB_stack, `${b}`);
        _.set_top (paramC_stack, `${c}`);

        _.pre_print (`hello`);

        _.set_top (return_value_stack, `... ${_.print2 (`middle`, `2nd 
arg`)} ${a}${_semis}${_.top (paramB_stack)}${c}${d}...`);

        _.post_print (`hello`);
        _.post_print (`pre down a=${a} _semis=${_semis} b=${b} c=${c} 
d=${d}`);
        paramA_stack.pop ();
        paramB_stack.pop ();
        paramC_stack.pop ();

        rule_name_stack.pop ();
        return return_value_stack.pop ();
    },
    _terminal: function () { return this.sourceString; },
    _iter: function (...children) { return children.map(c => c.rwr 
()); }
};
```

```
// ~~~~~~ stock main ~~~~~~
function main (src) {
    let parser = ohm.grammar (grammar);
    let cst = parser.match (src);
    if (cst.succeeded ()) {
        let cstSemantics = parser.createSemantics ();
        cstSemantics.addOperation ('rwr', rewrite_js);
        var generated_code = cstSemantics (cst).rwr ();
        return generated_code;
    } else {
        return cst.message;
    }
}

import * as fs from 'fs';
let src = fs.readFileSync(0, 'utf-8');
```

```
var result = main (src);
console.log (result);
```

The odd-looking comment

```
// ~~~~~~ stock main ~~~~~~
```

is inserted to allow self-compilation of the *t2t* tool. Ignore this line unless you intend to rebuild the *t2t* tool. See the maintenance section only if interested in the details of rebuilding.

Appendix - Why Javascript?

Why is *t2t* implemented in Javascript? Why is the *t2t* rewriter written in Javascript?

There is a difference between machine-readable code and human-readable code.

Most existing programming languages are geared towards human-readability.

When we want to generate code, though, we want to emphasize machine-readability, and, machine-writability.

For example, we want to target a language that has very little syntax, or has a very regular, normalized, syntax, to make code generation easier.

For generating code, formatting doesn't matter.

For generating code, repetition and DRY don't matter. The machine doesn't care if you ask it to do repetitive tasks. The machine doesn't make errors even when asked to duplicate code.

We want a language that contains a bag of functionality that will let us do anything we want.

In this particular case, the most helpful library is OhmJS. We use OhmJS to cut development time down.

OhmJS is written in Javascript, so it makes sense to use Javascript as the target language for the rewrite DSL, too. We need a parser and we need a rewriter. Development of the *t2t* is easier if, both, the parser and the rewriter are written in the same language. In the end, it doesn't matter which language is used for implementation of the *t2t* tool. The tool can inhale any language and exhale any language. Typically, the input language is some newly invented DSL, and, the output language is any language that is convenient for a given project, be it Python, or Javascript, or WASM, or Haskell, etc.

Most programmers think that Javascript has too many shortcomings compared to other languages, but, that perspective is based on human readability and error checking. In building the *t2t* rewrite DSL, we are interested, primarily in machine readability and the ability to generate code in the target language.

As far as error checking, we want very little automated error checking. Ideally, the newly invented DSL built using *t2t*, will provide all necessary human-oriented error checking. We don't want automated error checking to get in the way when developing the code generator.

Javascript is not the ideal target language for machine readability, though. Common Lisp and assembler are better target languages. Yet, OhmJS is written in Javascript and that fact swamps out other factors in choosing which language to target for the rewriter DSL.

I could have used any other language and relied on VSH (Visual Shell) and 0D (zero dependencies, drawware) to bolt languages to OhmJS, but, we didn't. In fact, it is already possible to build 0D programs in Python that use OhmJS, so that was a definite possibility.

I wanted to create a simple, stand-alone blob of source code that could be easily included in projects built by anyone. This implies that the generated code be written in Javascript, which most people can understand and debug and which can natively import OhmJS.

This project is an off-shoot of another project - a project to create a Larson scanner from Drawware that runs in a browser. Again, this consideration points at generating Javascript code. At present, we use node.js to run the generated code, but, in the future I will tweak the generator to create browser-friendly code, beginning with node.js-oriented Javascript.

Bibliography

[1] OhmJS from https://ohmjs.org
[2] Backus-Naur Form from https://en.wikipedia.org/wiki/Backus–Naur_form
[3] Regular Expression from https://en.wikipedia.org/wiki/Regular_expression
[4] Parsing Expression Grammar from https://en.wikipedia.org/wiki/Parsing_expression_grammar