T2T - Generate 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.

The time-saving "trick" is that *t2t* creates Javascript programs and relieves you from having to do most of the heavy lifting when building DSLs. You only need to create a grammar and a corresponding rewrite specification. Both, the grammar and the rewrite specification are contained in a single file with a '*.t2t*' extension.

T2T is a program that inhales a .t2t specification and generates a DSL transpiler program.

The generated DSL is written in Javascript and uses OhmJS.

To run the generated DSL, you can use node.js.

For example:

```
node ../t2t.mjs <${SPEC} >dsl.mjs
node dsl.mjs <${TEST}
```

The first invocation of *node.js* generates the DSL transpiler.

The second invocation of *node.js* runs the DSL transpiler against some test script written in the new DSL syntax.

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

The Holy Grail 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. Using General Purpose Programming languages 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.

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.

Example .t2t File

```
% grammar example
  Main = "a" (";" "b")+ "c" "d"+

% parameter sA
% parameter sB
% parameter sC

% rewrite
Main [_pA (_pBsemis _pBs)+ _pC _pD+]  ⎨ print ‘pre down’ ⎬ =  ⎡ sA=‘«_pA»’
                              ⎡ sB=‘«_pBs»’
                               ⎡ sC=‘«_pC»’
                                ⎡  ⎨ print ‘hello’ ⎬
                                 ‘...  ⎨ print2 ‘middle’ ‘2nd arg’ ⎬  «_pA»«_pBsemis»《sB》«_pC»«_pD»...’
                                ⎦
                               ⎦
                              ⎦
                             ⎦
```

Each section begins with a percent character (“%”[2]).

1. The *grammar* section is followed by the name of the grammar (Python, Javascript rules for forming legal names). Following this, the grammar rules are specified in OhmJS syntax. One rule per line.
2. The *parameter* section is optional. Each parameter is specified with “% parameter” followed by a name. T2t creates a reentrant variable (a “stack”) with the given name. The

[1] I call this DI, for Design Intent

[2] The “%” will probably be changed to “#” in the future to look more like markdown and to fit more easily into markdown editors.

top of the stack can be accessed in the *rewrite* section in a scoped manner using a special syntax consisting of a Unicode-bracketed name (see below, for which Unicode characters are used as parameter brackets). Parameters can be set up in a scoped manner before tree-walking is performed in that scope (see below for more details).
3. The Rewrite section begins with "% rewrite" and is followed by rewrite rules that correspond - by name - to grammar rules. See below for more details.

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[3] and to take appropriate actions by building strings. The actions are specified by programmers in the *.t2t* specification file.

Makefile

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

Examples

The *examples/* directory contains a *Makefile* for building several example SCNs.

Use these examples as guideposts for building your own SCNs.

T2T Specifcation (*.t2t files)*

A *.t2t* file contains 3 sections
1. A grammar
2. 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 this section until you need to build bigger and more complicated SCNs.
3. A *RWR,* 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

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

can create and mutate "state" by passing a blob of state as parameters, 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.

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 issues.

The idea is to simplify programming by using a code generation pipeline instead of conflating many unrelated issues in 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.

The grammar is more powerful than REGEX, since it is based on PEG technology.

Yet, the grammar is more readable than REGEX.

See the documentation for OhmJS (ohmjs.org).

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

Scoped Parameter Section

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 this section 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.

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

Reviewing the above example...

```
% grammar example
  Main = "a" (";" "b")+ "c" "d"+

% parameter sA
% parameter sB
% parameter sC

% rewrite
Main [_pA (_pBsemis _pBs)+ _pC _pD+]  ⎨ print ‘pre down’ ⎬ =  ⌈ sA=‘«_pA»’
                      ⌈ sB=‘«_pBs»’
                       ⌈ sC=‘«_pC»’
                        ⌈  ⎨ print ‘hello’ ⎬
                        ‘...  ⎨ print2 ‘middle’ ‘2nd arg’ ⎬  «_pA»«_pBsemis»《sB》«_pC»«_pD»...’
                        ⌋
                       ⌋
                      ⌋
                     ⌋
```

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

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

Simplifying, a rewrite rule consists of several parts

1. A name
2. Parameter list
3. Optional support routine call.
4. =
5. 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 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”.

Parameter List

A rule's parameter list 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[4].

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.

In the above example, the parameter list is:

```
[_pA (_pBsemis _pBs)+ _pC _pD+]
```

Which defines 5 parameters with names "_pA", "pBsemis", "_pBs", "_pC" and "_pD" and corresponds, syntactically, with the rule "Main" in the above grammar. Three of the parameters are iterations, in this case "+" iterations - "_pSemis", _pBs and "_pD".

In this example case, it was decided to prefix every parameter with the string "_p"[5], but, that was an arbitrary choice - any name would do.

Optional support routine call

The phrase

```
 ⎨ print ‘pre down’ ⎬
```

specifies a call to a routine contained in the file *support.mjs*. In this case, the routine is called *print* and takes one argument. Arguments are bracketed by Unicode single quotes ‘...’. More than one bracketed argument can be supplied. The call is transpiled into Javascript, using Javascript's *template strings* as parameters:

```
print (`pre down`);
```

Arguments to support routines are expanded in the same way that rewrite strings are expanded (see below) and can contain:

[4] 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 to do.

[5] The decision to use "_p" as a prefix was made for historical reasons during bootstrapping. It made it easier to manually inspect that parameters were being treated correctly.

1. Raw characters
2. String interpolation of rule parameters
3. String interpolation of scoped parameters.
4. String interpolation of a support routine call

=

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 possibly enveloped in one or more 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[6].
4. String interpolation of a support routine call, " ⎨ *routine-name* '...' '...' ... ⎬ ". The routine must be defined in the file *support.mjs* and must return a string.

The above example contains examples of all four components (raw characters "...", interpolation of rule parameters, e.g. "_pC", interpolation of scoped parameters, e.g. "sB" and a support call to a routine named "print2" that takes two arguments, respectively:

```
  ...
 «_pA» and «_pSemis» and «_pC» and «_pD»
 《sB》
  ⎨ print2 'middle' '2nd arg' ⎬
```

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 possible enveloped in one or more scopes.

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

Scopes

Scopes are bracketed by the Unicode brackets “ ⎡... ⎦ ”.

The first item after the opening bracket must be
• A binding to a scoped variable, “sB=‛«_pBs»’”, or
• A call to a support routine “ ⎨ print ‛hello’ ⎬ ”

The second item after the opening bracket must be
• a rewrite string
• 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_”[7]. 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 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.

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).

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.

[7] 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.

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 rewritten 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 named branches exist in the grammar, the rewrite section must not contain a rule not suffixed with a 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 6 functions. If you need to create many more exported functions, then it is likely that you are doing something wrong, 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.

You only need to use the *build-t2t* repository if you want to upgrade / maintain *t2t.mjs*.

The main program, *t2t.mjs*, is created automatically using the repository *build-t2t*.

In fact, *t2t.mjs* is created using a bootstrap version of itself.

The Repository for upgrading, modifying, fixing *t2t.mjs* is elsewhere, in https://github.com/guitarvydas/build-t2t.

Build-t2t can be considered to be an example of how to use Drawware and 0D technology to generate code generators.

[Collaborators are welcome, contact me if interested].

Appendix 1

T2T inhales the following (do-nothing) test code and produces the following Javascript program, using the following *support.mjs* file ...

~~~~~~

```
    % grammar example
      Main = "a" (";" "b")+ "c" "d"+

    % parameter sA
    % parameter sB
    % parameter sC

    % rewrite
    Main [_pA (_pBsemis _pBs)+ _pC _pD+] ⎨print ‘pre down’⎬= ⎡ sA=‘«_pA»’
                                    ⎡ sB=‘«_pBs»’
                                      ⎡ sC=‘«_pC»’
                                        ⎡ ⎨print ‘hello’⎬
                                          ‘... ⎨print2 ‘middle’ ‘2nd arg’⎬
«_pA»«_pBsemis»《sB》«_pC»«_pD»...’
                                        ⎦
                                      ⎦
                                    ⎦
                                  ⎦
```

~~~~~~

```
   'use strict'

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

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

    let sA_stack = [];
    let sB_stack = [];
    let sC_stack = [];

    const grammar = String.raw`
    example {

      Main = "a" (";" "b")+ "c" "d"+


    }
    `;

    const rewrite_code = {
        Main : function (_pA, _pBsemis, _pBs, _pC, _pD, ) {
```

```
        let _pre = _.print (`pre down`);
        return_value_stack.push ("");
        rule_name_stack.push ("");
        _.set_top (rule_name_stack, "Main");
        sA_stack.push ('');
        sB_stack.push ('');
        sC_stack.push ('');

        _pA = _pA.rwr ()
        _pBsemis = _pBsemis.rwr ().join ('')
        _pBs = _pBs.rwr ().join ('')
        _pC = _pC.rwr ()
        _pD = _pD.rwr ().join ('')

        _.set_top (sA_stack, `${_pA}`);
        _.set_top (sB_stack, `${_pBs}`);
        _.set_top (sC_stack, `${_pC}`);

        _.pre_print (`hello`);
        _.set_top (return_value_stack, `... ${_.print2 (`middle`, `2nd arg`)} ${_pA}$
{_pBsemis}${_.top (sB_stack)}${_pC}${_pD}...`);

        _.post_print (`hello`);
        sA_stack.pop ('');
        sB_stack.pop ('');
        sC_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 ()); }
    };


    function main (src) {
        let parser = ohm.grammar (grammar);
        let cst = parser.match (src);
        if (cst.succeeded ()) {
       let cstSemantics = parser.createSemantics ();
       cstSemantics.addOperation ('rwr', rewrite_code);
       var generated_code = cstSemantics (cst).rwr ();
       return generated_code;
        } else {
       return parser.trace (src).toString ();
        }
    }

    import * as fs from 'fs';
    let src = fs.readFileSync(0, 'utf-8');
    var result = main (src);
    console.log (result);

~~~~~~
    let _ = {
        top : function (stack) { let v = stack.pop (); stack.push (v); return v; },

        set_top : function (stack, v) { stack.pop (); stack.push (v); return v; },


        // for rewriter
        parameter_names : [],
        argnames : [],
        evaled_args : [],
```

```
        reset_stacks : function () {
        _.argnames = [];
        _.evaled_args = [];
        },

        memo_parameter : function (str) {
        _.parameter_names.push (str);
        return "";
        },
        foreach_parameter : function (str) {
        let s = [];
        _.parameter_names.forEach (p => s.push (str.replaceAll ("□", `${p}`) + "\n"));
        return s.join ('');
        },

        foreach_arg : function (str) {
        let s = [`//foreach_arg (${str})\n`];
        _.evaled_args.forEach (p => s.push (str.replaceAll ("□", `${p}`) + "\n"));
        return s.join ('');
        },

        memo_arg : function (name, s) { _.argnames.push (name); _.evaled_args.push
(s.replaceAll ("□", `${name}`)); return ""; },
        args_as_string : function () { return _.evaled_args.join (''); },

        // for examples
        pre_print : function (s) {console.log (`pre: ${s}`);},
        print : function (s) {console.log (`mid: ${s}`); return "";},
        post_print : function (s) {console.log (`post: ${s}`);},

        print2 : function (s1, s2) {console.log (`print2: ${s1} ${s2}`); return "";},



    };

    export {_};
```