RWR Specification

Rewriting Grammar

Overview

RWR is a simple DSL for rewriting text into other text.

The intent is to couple RWR with OhmJS to create a *t2t* - a text to text rewriting tool.

RWR works only with strings. RWR provides operations for creating strings using constant characters and string interpolation. RWR provides a way to create scoped stacks of strings (called “parameters”) that can be used by rules called in a nested manner.

In essence, one uses OhmJS to perform a parse of incoming text, then uses RWR to generate the necessary Javascript code to walk the resulting AST and to pass information *down* during walking and to pass return values *up* from inner rules.

At first, it may seem to be too restrictive to deal only with strings, but this paradigm is very powerful, yet simple to manage.

This document explains the grammar for RWR and provides, in an appendix, the Javascript code, grammar and semantics that implement RWR (using OhmJS).

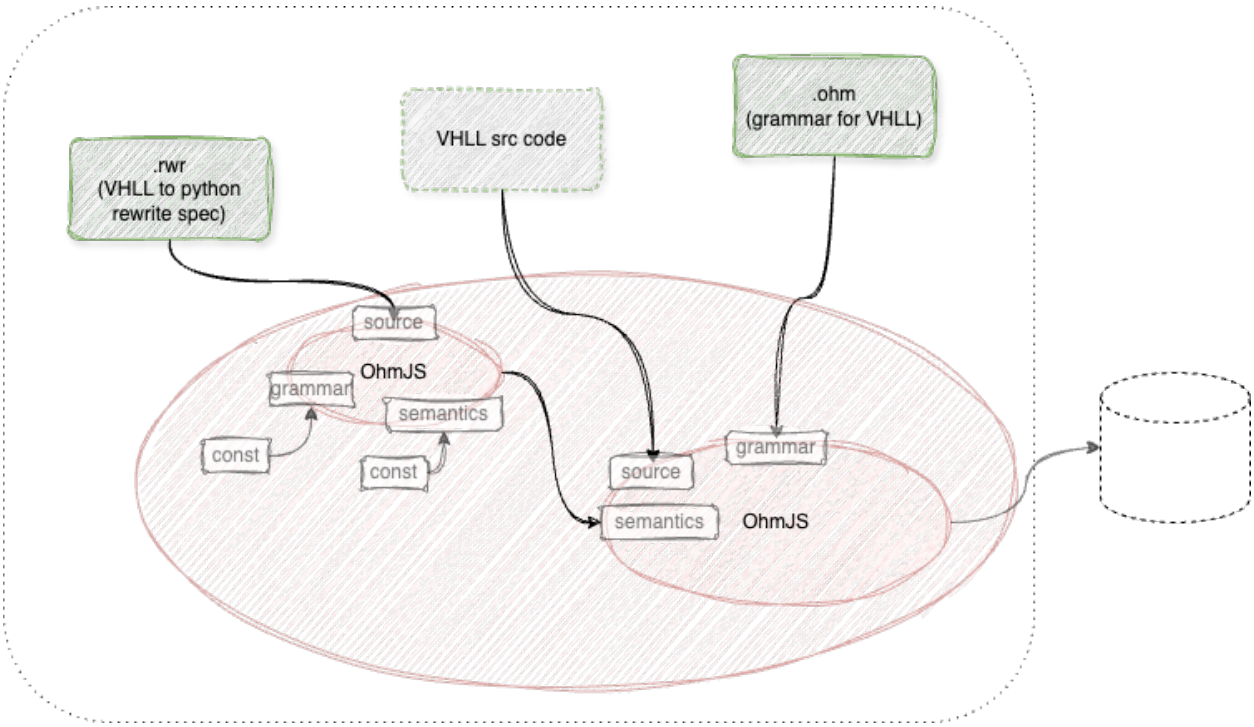
The full, working code for RWR is provided in a repository, provided in an Appendix.

[I like to see a sample first, before reading a dry specification. If that is your preference, please look at this example before reading further].

Use-Cases

A common use-case for the t2t combination - OhmJS + RWR - is to create new programming languages, convert them to existing programming languages and to run the code. It becomes unnecessary to write full-blown compilers. We already have enough compilers - the task of creating a new language boils down to transpiling code in the new language to legal code in some existing language, then performing the rest of the compilation using the existing language.

A sketch of using t2t (with RWR as the first pass) to transpile some new VHLL to Python follows:



Another use-case for t2t is to build quickie *sed*-like scripts to find certain patterns in text and to replace them with other patterns. Using OhmJS - PEG - for this use-case makes it especially easy, since OhmJS makes it possible to write such quickie scripts without needing to specify full-blown parses, as required by LR and RR grammar tools, like YACC, Bison, etc. With a top-down parser, like OhmJS, it is easier to say something like “match function definitions that have parameter lists and bodies enclosed in brace brackets, but, simply collect and pass-through the stuff inside of matched parentheses and matched brace brackets”. In traditional bottom-up, YACC-like technology, one would need to fully specify what can be matched inside of parentheses and brace brackets.

Usage

RWR deals only with the string-rewriting portion of text-to-text transpilation. Parsing of the incoming text is performed by OhmJS - see the OhmJS documentation for how to write `.ohm` files. The OhmJS parser creates a CST, derived from the AST specified in OhmJS syntax. A

CST - a Concrete Syntax Tree - is just a culled version of the AST - Abstract Syntax Tree. OhmJS uses its AST as a template to pattern match the incoming text and produces a tree that represents the actual incoming text.

T2T is a shell script[1] that invokes `rwr.mjs` as part of the script. RWR is generally not meant to be invoked by itself, outside of the context of the t2t program (shell script or Javascript program).

One can invoke RWR by itself, though. One needs to clone the pbp/ toolchain (here, referred to as `${PBP}`), then

```
$ node ${PBP}/t2t/lib/rwr.mjs ${REWRITE}
```

Where `${REWRITE}` is the rewrite specification script that is to be transpiled into Javascript.

The source code for the t2t shell script is provided in an appendix below. After the rewrite specification has been transpiled into Javascript, t2t just concatenates a bunch of strings to produce a complete Javascript program that invokes OhmJS. The string concatenation includes the source code for a set of Javascript functions that support operations used in the rewrite specification. As mentioned previously, the string concatenation can easily be done in Javascript using constant strings, but, at this time, I've chosen to use the UNIX '`cat`' command instead.

RWR Specification

The grammar for RWR (in OhmJS format) is:

```
t2t {
  main = parameterDef* rewriteDef

  parameterDef = "%" s_ "parameter" s_ name s_
  rewriteDef = "%" s_ "rewrite" s_ name s_ "{" rewriteRule+ "}" s_

  rewriteRule = s_ ruleName s_ "[" s_ (argDef s_)* "]" s_ "=" s_ rewriteScope s_

  argDef =
    | "(" parenarg+ ")" ("+" | "*" | "?")  -- parenthesized
    | name ("+" | "*" | "?")               -- iter
    | name                                 -- plain

  rewriteScope =
    | "┌" s_ "┤" s_ name s_ argstring* s_ "├" s_ rewriteScope s_ "┘"      -- call
    | "┌" s_  name s_ "=" s_ rewriteFormatString  s_ rewriteScope s_ "┘"  -- parameterbinding
    | rewriteFormatString                                                -- plain

  rewriteFormatString = "`" formatItem* "'"
  formatItem =
    | "┤" s_ name s_ argstring* "├" -- supportCall
    | "《" parameterRef "》"                          -- parameter
    | "«" argRef "»"                                  -- arg
    | "\\" any                                        -- escapedCharacter
```

---

[1] It used to be a single Javascript program. The main purpose of t2t is to invoke OhmJS twice (once as RWR) then to glue the resulting text together using string concatenation. I find this easier to think about when it is detangled and used in a shell script. YMMV. The Javascript code invoked `eval()` on one of the string pieces (the RWR portion) which is the same as invoking a compiler, albeit with less attached stigma.

```
    | ~"`" ~"´" ~"⌈" ~"⌋" ~"《" ~"》" ~"«" ~"»" any  -- rawCharacter

  parenarg = name s_
  argstring =  rewriteFormatString s_
  argRef = name
  parameterRef = name
  ruleName = name

  name (a name)
    = nameFirst nameRest*
  nameFirst = ("_" | letter)
  nameRest  = ("_" | alnum)

  s_ = space*

}
```

A rewrite spec consists of two main parts:
1. parameter definitions
2. the rewrite definition: a set of rewrite rules.

Parameter Definitions

Parameter definitions begin with `%param name`, where `name` is a string. Each name is converted into a stack of strings used by the rewrite processor.

Rewrite Definition

The rewrite definition begins with `%rewrite name`, followed by rewrite rules enclosed in brace brackets.

Each rewrite rule consists of
1. a rule name,
2. bindable[2] names in square brackets,
3. an equals sign,
4. a scoped rewrite.

Bindable names can be
- names followed by Ohm iteration operators (plus, star, or question mark)
- unadorned names[3] enclosed in parentheses
- a plain name

Each argument corresponds to a parsing expression in the corresponding Ohm grammar.

A scoped rewrite can take several forms, with the basic form being a format string enclosed in unicode open and close tick marks.

---

[2] This list is usually called the *formal parameter list* in function-based languages. I use the word *parameter* to denote a global stack of strings in RWR, so I wanted to find a different word for these variables to reduce confusion. In the grammar, for historical reasons, I refer to them as *arguments*. In this document, I've begun using the word bindable.

[3] At this time only one level of parentheses is allowed inside the square brackets.

The basic rewrite format string can contain

- raw characters
- interpolated arguments surrounded by Unicode double angle brackets
- references to parameters at the top of a parameter stack using the parameter stack name inside unicode large angle brackets
- Calls to support routines enclosed in unicode brackets “ ⦃ ... ⦄ ”. A call consists of a single function name followed by zero or more basic rewrite format strings. Note that each argument to a support function is just a string, recursively created in the same way as described here. The host language for OhmJS is Javascript, so, at this time, support functions must be written in Javascript. Many use-cases of RWR (and t2t) do not require support functions, or only very simple support functions consisting of very basic Javascript operations (dictionaries, stacks, etc.). All support functions must return a string value, which in many cases is just the empty string.

Scoping is denoted by the unicode bracketing symbols “ ⌜...⌟ ”. A scoped rewrite takes one of two forms

1. A support function call, as described earlier, followed by another scoped region or a basic rewrite format string
2. The creation and pushing of a new value onto one of the declared parameter stacks, followed by another scoped region or a basic rewrite format string. The syntax for pushing a new value onto a parameter stack is
    1. a single parameter stack name
    2. An equal sign “=“
    3. A basic rewrite format string. The string is evaluated in the context that exists at the time of dynamic creation, e.g. all arguments evaluated up to this point in time can be interpolated into the pushed value. This mechanism allows a rewrite rule to push information onto a parameter stack and to pass that information “down” into other rules as they are tree-walked. For example, the name of the current function being parsed can be pushed onto a parameter stack and then incorporated into result strings produced by subordinate rules.

Scoped parameters are popped at the end of a scope, i.e at the point that the “⌟ " is encountered.

Rewrite rules are compiled (transpiled) to Javascript code and are meant to be conjoined with OhmJS parsers, as the *semantics* portions of such OhmJS parsers.

Appendix - Code for `rwr.mjs`.

The code for rwr.mjs is only part of the t2t tool. Glue text, etc., is included in the repository listed in another appendix.

The point of displaying this source code here is to show the details of the semantics of RWR to those who already understand OhmJS.

Every generated semantic function begins with a call to the function `enter_rule` and ends with a call to the function `exit_rule`. Some of the actions of these functions are governed by a global variable `verbose`, which is currently hard-coded into the generated code. The intent is to allow verbose debugging of semantic functions.

Two string stacks are used in the generated code
1. return_value_stack
2. rule_name_stack.

Every rule pushes an empty string onto the `return_value_stack`. As rules proceed, they operate on the top string of `return_value_stack`. At the end of a rule, the top string contains the string value to be returned by the rule. The function `exit_rule` pops the stack and returns the final resulting string.

The `rule_name_stack` simply contains the string name of the rewrite rule currently being processed.

Two Javascript dictionaries (namespaces) - `args` and `parameters` - are used to manipulate bindable names and parameter stacks.

Bindables contain OhmJS data structures (concrete syntax trees) that must be evaluated at runtime. The code generated by RWR evaluates the arguments using the OhmJS semantic action '`.rwr()`' and appends '`.join()`' when evaluating OhmJS parse expressions that use iteration operators.

The first rewrite rule - `main` in the `_rewrite` namespace - contains some 40 lines of preamble code in basic rewrite string format. This code might be easier to understand using an editor with syntax colouring. The code can be found in the repository listed in a subsequent appendix.

```
'use strict'

import * as ohm from 'ohm-js';

let verbose = false;

function top (stack) { let v = stack.pop (); stack.push (v); return v; }

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

let return_value_stack = [];
let rule_name_stack = [];
let depth_prefix = ' ';

function enter_rule (name) {
    if (verbose) {
        console.error (depth_prefix, ["enter", name]);
        depth_prefix += ' ';
    }
    return_value_stack.push ("");
    rule_name_stack.push (name);
```

```
}

function set_return (v) {
    set_top (return_value_stack, v);
}

function exit_rule (name) {
    if (verbose) {
        depth_prefix = depth_prefix.substr (1);
        console.error (depth_prefix, ["exit", name]);
    }
    rule_name_stack.pop ();
    return return_value_stack.pop ()
}

const grammar = String.raw`
t2t {
  main = parameterDef* rewriteDef

  parameterDef = "%" s_ "parameter" s_ name s_
  rewriteDef = "%" s_ "rewrite" s_ name s_ "{" rewriteRule+ "}" s_

  rewriteRule = s_ ruleName s_ "[" s_ (argDef s_)* "]" s_ "=" s_ rewriteScope s_

  argDef =
    | "(" parenarg+ ")" ("+" | "*" | "?")  -- parenthesized
    | name ("+" | "*" | "?")               -- iter
    | name                                 -- plain

  rewriteScope =
    | "⎡" s_ "⦃" s_ name s_ argstring* s_ "⦄" s_ rewriteScope s_ "⎦"      -- call
    | "⎡" s_  name s_ "=" s_ rewriteFormatString  s_ rewriteScope s_ "⎦"  -- parameterbinding
    | rewriteFormatString                                                 -- plain

  rewriteFormatString = "‵" formatItem* "′"
  formatItem =
    | "⦃" s_ name s_ argstring* "⦄" -- supportCall
    | "《" parameterRef "》"                          -- parameter
    | "«" argRef "»"                                  -- arg
    | "\\" any                                        -- escapedCharacter
    | ~"‵" ~"′" ~"⎡" ~"⎦" ~"《" ~"》" ~"«" ~"»" any   -- rawCharacter

  parenarg = name s_
  argstring =  rewriteFormatString s_
  argRef = name
  parameterRef = name
  ruleName = name

  name (a name)
    = nameFirst nameRest*
  nameFirst = ("_" | letter)
  nameRest  = ("_" | alnum)

  s_ = space*

}
`;

let args = {};
function resetArgs () {
    args = {};
}
function memoArg (name, accessorString) {
    args [name] = accessorString;
};
function fetchArg (name) {
    return args [name];
}

let parameters = {};
function pushParameter (name, v) {
    parameters [name].push (v);
    console.error (['pushParameter', parameters]);
}
function popParameter (name) {
    parameters [name].pop ();
    console.error (['popParameter', parameters]);
}
function getParameter (name) {
    return parameters [name];
```

```
}


let _rewrite = {

    main : function (parameterDef_i,rewriteDef,) {
        enter_rule ("main");
        set_return (`let parameters = {};
function pushParameter (name, v) {
    if (!parameters [name]) {
        parameters [name] = [];
    }
    parameters [name].push (v);
}
function popParameter (name) {
    parameters [name].pop ();
}
function getParameter (name) {
    return parameters [name];
}
${parameterDef_i.rwr ().join ('')}

let _rewrite = {
${rewriteDef.rwr ()}
_terminal: function () { return this.sourceString; },
_iter: function (...children) { return children.map(c => c.rwr ()); }
}`);
        return exit_rule ("main");
    },
    parameterDef : function (_pct,_1,_parameter,_2,name,_3,) {
        enter_rule ("parameterDef");
        set_return (`\nparameters ["${name.rwr ()}"] = [];`);
        return exit_rule ("parameterDef");
    },
    rewriteDef : function (_pct,_1,_rewrite,_2,name,_3,_lb,rule_i,rb,_6,) {
        enter_rule ("rewriteDef");
        set_return (`${rule_i.rwr ().join ('')}`);
        return exit_rule ("rewriteDef");
    },
    rewriteRule : function (_0,ruleName,_1,lb,_2,argDef_i,_3_i,rb,_4,_eq,_5,rewriteScope,_6,) {
        enter_rule ("rewriteRule");
        resetArgs ();

        set_return (`\n${ruleName.rwr ()} : function (${argDef_i.rwr ().join ('')}) {
enter_rule ("${ruleName.rwr ()}");${rewriteScope.rwr ()}
return exit_rule ("${ruleName.rwr ()}");
},`);

        return exit_rule ("rewriteRule");
    },
    argDef_parenthesized : function (lp,names_i,rp,op,) {
        enter_rule ("argDef_parenthesized");
        set_return (`${names_i.rwr ().join ('')}`);
        return exit_rule ("argDef_parenthesized");
    },
    argDef_iter : function (name,op,) {
        enter_rule ("argDef_iter");
        memoArg (`${name.rwr ()}`,`\$\{${name.rwr ()}.rwr ().join ('')\}`,);

        set_return (`${name.rwr ()},`);

        return exit_rule ("argDef_iter");
    },
    argDef_plain : function (name,) {
        enter_rule ("argDef_plain");
        memoArg (`${name.rwr ()}`,`\$\{${name.rwr ()}.rwr ()\}`,);

        set_return (`${name.rwr ()},`);

        return exit_rule ("argDef_plain");
    },
    rewriteScope_call : function (lb,_1,lb2,_a,fname,_b,arg_i,_c,rb2,_2,rewriteScope,_3,rb,) {
        enter_rule ("rewriteScope_call");
        set_return (`
    ${fname.rwr ()} (${arg_i.rwr ().join ('')});
    ${rewriteScope.rwr ()}
`);
        return exit_rule ("rewriteScope_call");
    },
    rewriteScope_parameterbinding : function (lb,_1,pname,_2,_eq,_3,s,_4,scope,_5,rb,) {
        enter_rule ("rewriteScope_parameterbinding");
```

```
    set_return (`
pushParameter ("${pname.rwr ()}", \`${s.rwr ()}\`);${scope.rwr ()}\npopParameter ("${pname.rwr ()}");`);
    return exit_rule ("rewriteScope_parameterbinding");
},
rewriteScope_plain : function (s,) {
    enter_rule ("rewriteScope_plain");
    set_return (`
set_return (\`${s.rwr ()}\`);`);
    return exit_rule ("rewriteScope_plain");
},
rewriteFormatString : function (lq,formatItems_i,rq,) {
    enter_rule ("rewriteFormatString");
    set_return (`${formatItems_i.rwr ().join ('')}`);
    return exit_rule ("rewriteFormatString");
},
formatItem_supportCall : function (lb,_1,name,_2,argString_i,rb,) {
    enter_rule ("formatItem_supportCall");
    set_return (`\$\{${name.rwr ()} (${argString_i.rwr ().join ('')})\}`);
    return exit_rule ("formatItem_supportCall");
},
formatItem_parameter : function (lb,parameterRef,rb,) {
    enter_rule ("formatItem_parameter");
    set_return (`${parameterRef.rwr ()}`);
    return exit_rule ("formatItem_parameter");
},
formatItem_arg : function (lb,argRef,rb,) {
    enter_rule ("formatItem_arg");
    set_return (`${argRef.rwr ()}`);
    return exit_rule ("formatItem_arg");
},
formatItem_escapedCharacter : function (bslash,any,) {
    enter_rule ("formatItem_escapedCharacter");
    set_return (`${bslash.rwr ()}${any.rwr ()}`);
    return exit_rule ("formatItem_escapedCharacter");
},
formatItem_rawCharacter : function (c,) {
    enter_rule ("formatItem_rawCharacter");
    set_return (`${c.rwr ()}`);
    return exit_rule ("formatItem_rawCharacter");
},
parenarg : function (name,ws,) {
    enter_rule ("parenarg");
    memoArg (`${name.rwr ()}`,`\$\{${name.rwr ()}.rwr ().join ('')\}`,);

    set_return (`${name.rwr ()},`);

    return exit_rule ("parenarg");
},
argstring : function (str,ws,) {
    enter_rule ("argstring");
    set_return (`\`${str.rwr ()}\`,`);
    return exit_rule ("argstring");
},
argRef : function (name,) {
    enter_rule ("argRef");
    set_return (`${fetchArg (`${name.rwr ()}`,)}`);
    return exit_rule ("argRef");
},
parameterRef : function (name,) {
    enter_rule ("parameterRef");
    set_return (`\$\{getParameter ("${name.rwr ()}")\}`);
    return exit_rule ("parameterRef");
},
ruleName : function (name,) {
    enter_rule ("ruleName");
    set_return (`${name.rwr ()}`);
    return exit_rule ("ruleName");
},
name : function (nameFirst,nameRest_i,) {
    enter_rule ("name");
    set_return (`${nameFirst.rwr ()}${nameRest_i.rwr ().join ('')}`);
    return exit_rule ("name");
},
nameFirst : function (c,) {
    enter_rule ("nameFirst");
    set_return (`${c.rwr ()}`);
    return exit_rule ("nameFirst");
},
nameRest : function (c,) {
    enter_rule ("nameRest");
    set_return (`${c.rwr ()}`);
```

```
        return exit_rule ("nameRest");
    },
    s_ : function (space_i,) {
        enter_rule ("s_");
        set_return (`${space_i.rwr ().join ('')}`);
        return exit_rule ("s_");
    },
    _terminal: function () { return this.sourceString; },
    _iter: function (...children) { return children.map(c => c.rwr ()); }
}
import * as fs from 'fs';

function grammarname (s) {
    let n = s.search (/{/);
    return s.substr (0, n).replaceAll (/\n/g,'').trim ();
}

try {
    const argv = process.argv.slice(2);
    let srcFilename = argv[0];
    if ('-' == srcFilename) { srcFilename = 0 }
    let src = fs.readFileSync(srcFilename, 'utf-8');
    try {
        let parser = ohm.grammar (grammar);
        let cst = parser.match (src);
        if (cst.failed ()) {
            //throw Error (`${cst.message}\ngrammar=${grammarname (grammar)}\nsrc=\n${src}`);
            throw Error (cst.message);
        }
        let sem = parser.createSemantics ();
        sem.addOperation ('rwr', _rewrite);
        console.log (sem (cst).rwr ());
        process.exit (0);
    } catch (e) {
        //console.error (`${e}\nargv=${argv}\ngrammar=${grammarname (grammar)}\src=\n${src}`);
        console.error (`${e}\n\ngrammar = "${grammarname (grammar)}"`);
        process.exit (1);
    }
} catch (e) {
    console.error (`${e}\n\ngrammar = "${grammarname (grammar)}`);
    process.exit (1);
}
```

Appendix - Repository for `rwr.mjs`

The code for RWR can be found in the pbp-dev/t2t repository.

Various other tools can, also, be found in the parent pbp-dev/ repository. Notably, the t2t lib/ repository contains code snippets that are used to compose generated code into valid OhmJS programs.

The composition consists of gluing strings (of code) together to form valid Javascript programs that invoke the OhmJS parser. At present, this gluing is done using UNIX command line tools, but can be easily performed wholly in Javascript, since the main operation is just string concatenation.

Appendix - T2T Shell Script

```
#!/bin/bash
set -e
wd=$1
pbp=$2
grammar=$3
rewrite=$4
support=$5
src=$6
lib=${pbp}/t2t/lib

node ${lib}/t2t.mjs ${rewrite} >${wd}/temp.rewrite.mjs
cat ${lib}/front.part.js ${grammar} ${lib}/middle.part.js ${lib}/
args.part.js ${support} ${wd}/temp.rewrite.mjs ${lib}/tail.part.js >$
{wd}/temp.nanodsl.mjs
node ${wd}/temp.nanodsl.mjs ${src}
```

The command line parameters to t2t are

1. The working directory of the project (usually '.' is sufficient).
2. The directory of the PBP toolchain (usually './pbp', with no trailing slash)
3. The `.ohm` grammar for some new SCN/DSL/programming-language.
4. The `.rwr` rewrite specification that corresponds to the grammar.
5. Javascript source code for any support functions called by the `.rwr` specification.
6. The source code the program written in the new SCN/DSL/programming-language.

The `pbp/` directory contains:

- A shell script called `t2t.bash`.
- Various other shell scripts, like `tas.bash`, `das.bash`, etc.
- A subdirectory called `t2t/` which contains
  - A subdirectory called `lib/`, which contains `rwr.mjs` along with front, middle and back end strings for forming the final t2t program (along with, maybe, some experimental and bootstrap files which have yet to be removed).
- Various other subdirectories for other tools in the pbp toolchain, like `kernel/`, `das/`, `tas/`.

This script can be found in the repository given in another appendix.

Appendix - Sample Test Program and Generated Code

This test does not produce any useful results, but, it does show and test the various features of RWR.

---

tests/test2.txt

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

---

tests/test2.grammar

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

---

tests/test2.rewrite

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

% rewrite example {
Main [a (_semi b)+ c d+]=
  ⌈ {print 'pre down a=«a» _semi=«_semi» b=«b» c=«c» d=«d»'}
    ⌈ paramA='«a»'
      ⌈ paramB='«b»'
        ⌈ paramC='«c»'
          ⌈ {print 'hello'}
            '... {print2 'middle' '2nd arg'}
«a»«_semi»《paramB》«c»«d»...'
          ⌋
        ⌋
      ⌋
    ⌋
  ⌋
}
```

Generated Code

This is all of the code generated by t2t. The portion generated by RWR begins with the line "`let _rewrite = {`".

The generated code, below, can also be seen in the development repository as example-dsl2.mjs.

```
'use strict'

import * as ohm from 'ohm-js';
```

```
let verbose = false;

function top (stack) { let v = stack.pop (); stack.push
(v); return v; }

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

let return_value_stack = [];
let rule_name_stack = [];
let depth_prefix = ' ';

function enter_rule (name) {
    if (verbose) {
        console.error (depth_prefix, ["enter", name]);
        depth_prefix += ' ';
    }
    return_value_stack.push ("");
    rule_name_stack.push (name);
}

function set_return (v) {
    set_top (return_value_stack, v);
}

function exit_rule (name) {
    if (verbose) {
        depth_prefix = depth_prefix.substr (1);
        console.error (depth_prefix, ["exit", name]);
    }
    rule_name_stack.pop ();
    return return_value_stack.pop ()
}

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

```
`;

let args = {};
function resetArgs () {
    args = {};
}
function memoArg (name, accessorString) {
    args [name] = accessorString;
};
function fetchArg (name) {
    return args [name];
}

function print (s) {
    console.log (`PRINT: ${s}`);
    return "";
}

function print2 (s1, s2) {
    console.log (`PRINT2: ${s1} ${s2}`);
    return "";
}

let parameters = {};
function pushParameter (name, v) {
    if (!parameters [name]) {
        parameters [name] = [];
    }
    parameters [name].push (v);
}
function popParameter (name) {
    parameters [name].pop ();
}
function getParameter (name) {
    return parameters [name];
}

parameters ["paramA"] = [];
parameters ["paramB"] = [];
parameters ["paramC"] = [];
```

```
let _rewrite = {

    Main : function (a,_semi,b,c,d,) {
        enter_rule ("Main");
        print (`pre down a=${a.rwr ()} _semi=${_semi.rwr
().join ('')} b=${b.rwr ().join ('')} c=${c.rwr ()} d=$
{d.rwr ().join ('')}`,);

        pushParameter ("paramA", `${a.rwr ()}`);
        pushParameter ("paramB", `${b.rwr ().join ('')}`);
        pushParameter ("paramC", `${c.rwr ()}`);
        print (`hello`,);

        set_return (`... ${print2 (`middle`,`2nd arg`,)} $
{a.rwr ()}${_semi.rwr ().join ('')}${getParameter
("paramB")}${c.rwr ()}${d.rwr ().join ('')}...`);

        popParameter ("paramC");
        popParameter ("paramB");
        popParameter ("paramA");

        return exit_rule ("Main");
    },
    _terminal: function () { return this.sourceString; },
    _iter: function (...children) { return children.map(c
=> c.rwr ()); }
}
import * as fs from 'fs';

function grammarname (s) {
    let n = s.search (/{/);
    return s.substr (0, n).replaceAll (/\n/g,'').trim ();
}

try {
    const argv = process.argv.slice(2);
    let srcFilename = argv[0];
    if ('-' == srcFilename) { srcFilename = 0 }
    let src = fs.readFileSync(srcFilename, 'utf-8');
```

```
    try {
        let parser = ohm.grammar (grammar);
        let cst = parser.match (src);
        if (cst.failed ()) {
            //throw Error (`${cst.message}\ngrammar=$
{grammarname (grammar)}\nsrc=\n${src}`);
            throw Error (cst.message);
        }
        let sem = parser.createSemantics ();
        sem.addOperation ('rwr', _rewrite);
        console.log (sem (cst).rwr ());
        process.exit (0);
    } catch (e) {
        //console.error (`${e}\nargv=${argv}\ngrammar=$
{grammarname (grammar)}\src=\n${src}`);
        console.error (`${e}\n\ngrammar = "${grammarname
(grammar)}"`);
        process.exit (1);
    }
} catch (e) {
    console.error (`${e}\n\ngrammar = "${grammarname
(grammar)}`);
    process.exit (1);
}
```