Jison

Documentation

Jison takes a context-free grammar as input and outputs a JavaScript file capable of parsing the language described by that grammar. You can then use the generated script to parse inputs and accept, reject, or perform actions based on the input. If you're familiar with Bison or Yacc, or other clones, you're almost ready to roll.

- Installation
- Usage from the command line
- Usage from a CommonJS Module
- Using the Generated Parser
- Using the Parser from the Web
- The Concepts of Jison
- Specifying a Language
- Lexical Analysis
- Tracking Locations
- Custom Scanners
- Sharing Scope
- Parsing algorithms
- Projects using Jison
- Contributors
- License

Installation

Jison can be installed for Node using `npm`

Using npm:

```
npm install jison -g
```

Usage from the command line

Clone the github repository for examples:

```
git clone git://github.com/zaach/jison.git
cd jison/examples
```

Now you're ready to generate some parsers:

```
jison calculator.jison
```

This will generate `calculator.js` in your current working directory. This script can be used to parse an input file, like so:

```
echo "2^32 / 1024" > testcalc
node calculator.js testcalc
```

This will print out `4194304`.

Usage from a CommonJS Module

You can generate parsers programatically from JavaScript as well. Assuming Jison is in your commonjs environment's load path:

```
// mygenerator.js
var Parser = require("jison").Parser;

var grammar = {
    "lex": {
        "rules": [
           ["\\s+", "/* skip whitespace */"],
           ["[a-f0-9]+", "return 'HEX';"]
        ]
    },

    "bnf": {
        "hex_strings" :[ "hex_strings HEX",
                         "HEX" ]
    }
};

var parser = new Parser(grammar);

// generate source, ready to be written to disk
var parserSource = parser.generate();

// you can also use the parser directly from memory

parser.parse("adfe34bc e82a");
// returns true

parser.parse("adfe34bc zxg");
// throws lexical error
```

Alternatively, if you want to use the Jison file format but not generate a static JavaScript file for it, you could use a snippet like this:

```
// myparser.js
var fs = require("fs");
var jison = require("jison");

var bnf = fs.readFileSync("grammar.jison", "utf8");
var parser = new jison.Parser(bnf);
```

```
module.exports = parser;
```

Using the Generated Parser

Once you have generated the parser and saved it, you no longer need Jison or any other dependencies.

As demonstrated before, the parser can be used from the command line:

```
node calculator.js testcalc
```

Though, more ideally, the parser will be a dependency of another module. You can require it from another module like so:

```
// mymodule.js
var parser = require("./calculator").parser;

function exec (input) {
    return parser.parse(input);
}

var twenty = exec("4 * 5");
```

Or more succinctly:

```
// mymodule.js
function exec (input) {
    return require("./calculator").parse(input);
}

var twenty = exec("4 * 5");
```

Using the Parser from the Web

The generated parser script may be included in a web page without any need for a CommonJS loading environment. It’s as simple as pointing to it via a script tag:

```
<script src="calculator.js"></script>
```

When you generate the parser, you can specify the variable name it will be declared as:

```
// mygenerator.js
var parserSource = generator.generate({moduleName: "calc"});
// then write parserSource to a file called, say, calc.js
```

Whatever `moduleName` you specified will be the the variable you can access the parser from in your web page:

```
<script src="calc.js"></script>
<script>
  calc.parse("42 / 0");
</script>
```

The moduleName you specify can also include a namespace, e.g:

```
// mygenerator.js
```

```
var parserSource = parser.generate({moduleName: "myCalculator.parser"});
```

And could be used like so:

```
<script>
  var myCalculator = {};
</script>

<script src="calc.js"></script>

<script>
  myCalculator.parser.parse("42 / 0");
</script>
```

Or something like that – you get the picture.

A demo of the calculator script used in a web page is here.

The Concepts of Jison

Until the Bison guide is properly ported for Jison, you can refer to it for the major concepts, which are equivalent (except for the bits about static typing of semantic values, and other obvious C artifacts.)

Other helpful sections:

- Bison Grammar Files
- The Bison Parser Algorithm
- Error Recovery (alpha support, at this point)

Specifying a Language

The process of parsing a language commonly involves two phases: **lexical analysis** (tokenizing) and **parsing**, which the Lex/Yacc and Flex/Bison combinations are famous for. Jison lets you specify a parser much like you would using Bison/Flex, with separate files for tokenization rules and for the language grammar, or with the tokenization rules embedded in the main grammar.

For example, here is the grammar for the calculator parser:

```
/* description: Parses end executes mathematical expressions. */

/* lexical grammar */
%lex

%%
\s+                   /* skip whitespace */
[0-9]+("."[0-9]+)?\b  return 'NUMBER';
"*"                   return '*';
"/"                   return '/';
"-"                   return '-';
"+"                   return '+';
"^"                   return '^';
"("                   return '(';
")"                   return ')';
"PI"                  return 'PI';
```

```
"E"                   return 'E';
<<EOF>>               return 'EOF';

/lex

/* operator associations and precedence */

%left '+' '-'
%left '*' '/'
%left '^'
%left UMINUS

%start expressions

%% /* language grammar */

expressions
    : e EOF
        {print($1); return $1;}
    ;

e
    : e '+' e
        {$$ = $1+$3;}
    | e '-' e
        {$$ = $1-$3;}
    | e '*' e
        {$$ = $1*$3;}
    | e '/' e
        {$$ = $1/$3;}
    | e '^' e
        {$$ = Math.pow($1, $3);}
    | '-' e %prec UMINUS
        {$$ = -$2;}
    | '(' e ')'
        {$$ = $2;}
    | NUMBER
        {$$ = Number(yytext);}
    | E
        {$$ = Math.E;}
    | PI
        {$$ = Math.PI;}
    ;
```

which compiles down to this JSON representation used directly by Jison:

```
{
    "lex": {
        "rules": [
           ["\\s+",                    "/* skip whitespace */"],
           ["[0-9]+(?:\\.[0-9]+)?\\b", "return 'NUMBER';"],
           ["\\*",                     "return '*';"],
           ["\\/",                     "return '/';"],
           ["-",                       "return '-';"],
           ["\\+",                     "return '+';"],
           ["\\^",                     "return '^';"],
           ["\\(",                     "return '(';"],
           ["\\)",                     "return ')';"],
           ["PI\\b",                   "return 'PI';"],
           ["E\\b",                    "return 'E';"],
```

```
            ["$",                       "return 'EOF';"]
        ]
    },

    "operators": [
        ["left", "+", "-"],
        ["left", "*", "/"],
        ["left", "^"],
        ["left", "UMINUS"]
    ],

    "bnf": {
        "expressions" :[[ "e EOF",   "print($1); return $1;"  ]],

        "e" :[[ "e + e",   "$$ = $1 + $3;" ],
              [ "e - e",   "$$ = $1 - $3;" ],
              [ "e * e",   "$$ = $1 * $3;" ],
              [ "e / e",   "$$ = $1 / $3;" ],
              [ "e ^ e",   "$$ = Math.pow($1, $3);" ],
              [ "- e",     "$$ = -$2;", {"prec": "UMINUS"} ],
              [ "( e )",   "$$ = $2;" ],
              [ "NUMBER",  "$$ = Number(yytext);" ],
              [ "E",       "$$ = Math.E;" ],
              [ "PI",      "$$ = Math.PI;" ]]
    }
}
```

Jison accepts both the Bison/Flex style format, or the raw JSON format, e.g:

```
node bin/jison examples/calculator.jison
```

or

```
node bin/jison examples/calculator.json
```

When the lexical grammar resides in its own (.jisonlex) file, use that as the second argument to Jison, e.g.:

```
node bin/jison examples/classy.jison examples/classy.jisonlex
```

More examples can be found in the `examples/` and `tests/parser/` directories.

Lexical Analysis

Jison includes a rather rudimentary scanner generator, though **any module that supports the basic scanner API could be used** in its place.

The format of the input file (including macro support) and the style of the pattern matchers are modeled after Flex. Several metacharacters have been added, but there is also one minor inconvenience compared to Flex patterns, namely exact string patterns must be placed in quotes e.g.:

Bad:

```
[0-9]+zomg    print(yytext)
```

Good:

```
[0-9]+"zomg"    print(yytext);
```

Actions that span multiple lines should be surrounded by braces:

```
[0-9]+"zomg"    %{ print(yytext);
                   return 'ZOMG'; %}
```

A recently added feature are **start conditions**, which allow certain rules to only match in certain states. If the lexer is not in that state, then the rule is ignored. The lexer starts in the `INITIAL` state, but can move to new states specified by you. Read that link for the run-down. An example below shows where Jison differs, namely `this.begin('state')` instead of `BEGIN(STATE)` for changing states within an action:

```
%s expect

%%
expect-floats        this.begin('expect');

<expect>[0-9]+"."[0-9]+      {
            console.log( "found a float, = " + yytext );
            }
<expect>\n           %{
            /* that's the end of the line, so
             * we need another "expect-number"
             * before we'll recognize any more
             * numbers
             */
            this.begin('INITIAL');
            %}

[0-9]+      console.log( "found an integer, = " + yytext );

"."         console.log( "found a dot" );
```

If you use `%x` instead of `%s` to declare your start condtion then *only* rules that match the current start condition will be considered

Consider the following example of a scanner that simply scans all double-quote delimited strings in a text file but disallows newlines inside quotations:

```
%x string

%%
["]               this.begin("string");
<string>[^"\n]*   return "STRING";
<string>[\n]      return "NEWLINE_IN_STRING";
<string><<EOF>>   return "EOF_IN_STRING";
<string>["]       this.popState();

[.\n]+      /* skip over text not in quotes */
<<EOF>>     return "EOF";
```

Additionaly, use `this.popState()` within an action to revert to the previous state.

Using the JSON format, start conditions are defined with an array before the rule's matcher:

```
{
  rules: [
    [['expect'], '[0-9]+"."[0-9]+', 'console.log( "found a float, = " + yytext );']
```

```
  ]
}
```

The array contains the list of start conditions for the rule.

Tracking Locations

Jison's lexical analyzer will track line number and column number information for each token and make them available within parser actions. The API is identical to Bison's.

Custom Scanners

You don't have to use the builtin Jison lexical scanner. An object with a `lex` and a `setInput` function would suffice, e.g.:

```
parser.lexer = {
    lex: function () {
        return 'NIL';
    },
    setInput: function (str) {
    }
};
```

This lexer would simply return `NIL` tokens *ad infinitum*.

The following example demonstrates a scanner that looks for upper and lower case letters, ignoring all whitespace

```
// myscanner.js
function AlphabetScanner() {
    var text = "";
    this.yytext = "";
    this.yyloc = {
        first_column: 0,
        first_line: 1,
        last_line: 1,
        last_column: 0
    };
    this.yylloc = this.yyloc;
    this.setInput = function(text_) {
        text = text_;
    };
    this.lex = function() {
        // Return the EOF token when we run out of text.
        if (text === "") {
            return "EOF";
        }

        // Consume a single character and increment our column numbers.
        var c = text.charAt(0);
        text = text.substring(1);
        this.yytext = c;
        this.yyloc.first_column++;
        this.yyloc.last_column++;

        if (c === "\n") {
```

```
            // Increment our line number when we hit newlines.
            this.yyloc.first_line++;
            this.yyloc.last_line++;
            // Try to keep lexing because we aren't interested
            // in newlines.
            return this.lex();
        } else if (/[a-z]/.test(c)) {
            return "LOWER_CASE";
        } else if (/[A-Z]/.test(c)) {
            return "UPPER_CASE";
        } else if (/\s/.test(c)) {
            // Try to keep lexing because we aren't interested
            // in whitespace.
            return this.lex();
        } else {
            return "INVALID";
        }
    };
}
parser.lexer = new AlphabetScanner();
```

Sharing Scope

In Bison, code is expected to be lexically defined within the scope of the semantic actions. E.g., chunks of code may be included in the generated parser source, which are available from semantic actions.

Jison supports inline code blocks like Bison, but also exposes state that can be accessed from other modules. Instead of pulling code into the generated module, the generated module can be required and used by other modules.The parser has a `yy` property which is exposed to actions as the `yy` free variable. Any functionality attached to this property is available in both lexical and semantic actions through the `yy` free variable.

An example from orderly.js:

```
var parser = require("./orderly/parse").parser;

// set parser's shared scope
parser.yy = require("./orderly/scope");

// returns the JSON object
var parse = exports.parse = function (input) {
    return parser.parse(input);
};
...
```

The `scope` module contains logic for building data structures, which is used within the semantic actions.

Parsing algorithms

Like Bison, Jison can recognize languages described by LALR(1) grammars, though it also has modes for LR(0), SLR(1), and LR(1). It also has a special mode for generating LL(1) parse tables (requested by my professor,) and could be extended to generate a recursive descent parser for LL(k) languages in the future. But, for now, Jison is geared toward bottom-up parsing.

**LR(1) mode is currently not practical for use with anything other than toy grammars, but that is entirely*

a consequence of the algorithm used, and may change in the future.

Projects using Jison

View them on the wiki, or add your own.

Contributors

via github

License