Apuntes de la Asignatura Procesadores de Lenguajes

Casiano R. León [1]

26 de enero de 2015

[1]DEIOC Universidad de La Laguna

Índice general

I PARTE: APUNTES DE PROCESADORES DE LENGUAJES 8

1. Expresiones Regulares y Análisis Léxico en JavaScript 9
1.1. Mozilla Developer Network: Documentación . . . 9
1.2. Práctica: Conversor de Temperaturas . . . 9
1.3. Práctica: Comma Separated Values. CSV . . . 13
1.4. Comentarios y Consejos . . . 33
1.5. Ejercicios . . . 34
1.6. Práctica: Palabras Repetidas . . . 37
1.7. Ejercicios . . . 40
1.8. Ejercicios . . . 41
1.9. Práctica: Ficheros INI . . . 41
1.10. Práctica: Analizador Léxico para Un Subconjunto de JavaScript . . . 49

2. Analizadores Descendentes Predictivos en JavaScript 51
2.1. Conceptos Básicos para el Análisis Sintáctico . . . 51
2.1.1. Ejercicio . . . 52
2.2. Análisis Sintáctico Predictivo Recursivo . . . 52
2.2.1. Introducción . . . 52
2.2.2. Ejercicio: Recorrido del árbol en un ADPR . . . 57
2.3. Recursión por la Izquierda . . . 57
2.4. Esquemas de Traducción . . . 58
2.5. Eliminación de la Recursión por la Izquierda en un Esquema de Traducción . . . 58
2.6. Práctica: Analizador Descendente Predictivo Recursivo . . . 59

3. Análisis Sintáctico Mediante Precedencia de Operadores en JavaScript 63
3.1. Ejemplo Simple de Intérprete: Una Calculadora . . . 63
3.2. Análisis Top Down Usando Precedencia de Operadores . . . 63
3.2.1. Gramática de JavaScript . . . 63

4. Análisis Descendente mediante Parsing Expresion Grammars en JavaScript 64
4.1. Introducción a los PEGs . . . 64
4.1.1. Syntax . . . 64
4.1.2. Semantics . . . 65
4.1.3. Implementing parsers from parsing expression grammars . . . 65
4.1.4. Lexical Analysis . . . 66
4.1.5. Left recursion . . . 66
4.1.6. Referencias y Documentación . . . 66
4.2. PEGJS . . . 66
4.3. Un Ejemplo Sencillo . . . 70
4.3.1. Asociación Incorrecta para la Resta y la División . . . 72
4.4. Acciones Intermedias . . . 72
4.5. PegJS en los Browser . . . 73
4.6. Eliminación de la Recursividad por la Izquierda en PEGs . . . 76

4.7. Eliminando la Recursividad por la Izquierda en la Calculadora . . . 79
4.8. Eliminación de la Recursividad por la Izquierda y Atributos Heredados . . . 80
4.8.1. Eliminación de la Recursión por la Izquierda en la Gramática . . . 80
4.8.2. Eliminación de la Recursión por la Izquierda en un Esquema de Traducción . . . 81
4.8.3. Eliminación de la Recursividad por la Izquierda en PEGJS . . . 81
4.9. Dangling else: Asociando un else con su if mas cercano . . . 82
4.10. Not Predicate: Comentarios Anidados . . . 84
4.11. Un Lenguaje Dependiente del Contexto . . . 85
4.12. Usando Pegjs con CoffeeScript . . . 86
4.13. Práctica: Analizador de PL0 Ampliado Usando PEG.js . . . 87
4.14. Práctica: Ambiguedad en C++ . . . 87
4.15. Práctica: Inventando un Lenguaje: Tortoise . . . 89

5. Análisis Sintáctico Ascendente en JavaScript 91
5.1. Conceptos Básicos para el Análisis Sintáctico . . . 91
5.1.1. Ejercicio . . . 92
5.2. Ejemplo Simple en Jison . . . 92
5.2.1. Véase También . . . 95
5.2.2. Práctica: Secuencia de Asignaciones Simples . . . 95
5.3. Ejemplo en Jison: Calculadora Simple . . . 95
5.3.1. Práctica: Calculadora con Listas de Expresiones y Variables . . . 100
5.4. Conceptos Básicos del Análisis LR . . . 100
5.5. Construcción de las Tablas para el Análisis SLR . . . 103
5.5.1. Los conjuntos de Primeros y Siguientes . . . 103
5.5.2. Construcción de las Tablas . . . 104
5.6. Práctica: Analizador de PL0 Usando Jison . . . 110
5.7. Práctica: Análisis de Ámbito en PL0 . . . 111
5.8. Práctica: Traducción de Infijo a Postfijo . . . 112
5.9. Práctica: Calculadora con Funciones . . . 112
5.10. Práctica: Calculadora con Análisis de Ámbito . . . 113
5.11. Algoritmo de Análisis LR . . . 117
5.12. El módulo Generado por `jison` . . . 118
5.12.1. Version . . . 118
5.12.2. Gramática Inicial . . . 118
5.12.3. Tablas . . . 118
5.12.4. Acciones Semánticas . . . 119
5.12.5. Tabla de Acciones y GOTOs . . . 121
5.12.6. defaultActions . . . 121
5.12.7. Reducciones . . . 122
5.12.8. Desplazamientos/Shifts . . . 123
5.12.9. Manejo de Errores . . . 124
5.12.10. Analizador Léxico . . . 125
5.12.11. Exportación . . . 127
5.13. Precedencia y Asociatividad . . . 129
5.14. Esquemas de Traducción . . . 134
5.15. Manejo en `jison` de Atributos Heredados . . . 135
5.16. Definición Dirigida por la Sintáxis . . . 138
5.17. Ejercicios: Casos de Estudio . . . 140
5.17.1. Un mal diseño . . . 140
5.17.2. Gramática no LR(1) . . . 141
5.17.3. Un Lenguaje Intrínsecamente Ambiguo . . . 141
5.17.4. Conflicto reduce-reduce . . . 142
5.18. Recuperación de Errores . . . 148

5.19. Depuración en `jison` . . . 148
5.20. Construcción del Árbol Sintáctico . . . 148
5.21. Consejos a seguir al escribir un programa `jison` . . . 149

6. Análisis Sintáctico Ascendente en Ruby **150**
6.1. La Calculadora . . . 150
6.1.1. Uso desde Línea de Comandos . . . 150
6.1.2. Análisis Léxico con `rexical` . . . 151
6.1.3. Análisis Sintáctico . . . 151
6.2. Véase También . . . 153

7. Transformaciones Árbol **154**
7.1. Árbol de Análisis Abstracto . . . 154
7.2. Selección de Código y Gramáticas Árbol . . . 157
7.3. Patrones Árbol y Transformaciones Árbol . . . 159
7.4. Ejemplo de Transformaciones Árbol: Parse::Eyapp::TreeRegexp . . . 161
7.5. Treehugger . . . 166
7.6. Práctica: Transformaciones en Los Árboles del Analizador PL0 . . . 168

Índice de figuras

1.1. Ejemplo de pantalla de La aplicación para el Análisis de Datos en Formato CSV . . . 14

4.1. pegjs en la web . . . 76

5.1. NFA que reconoce los prefijos viables . . . 103
5.2. DFA equivalente al NFA de la figura 5.1 . . . 105
5.3. DFA construido por Jison . . . 120

Índice de cuadros

2.1. Una Gramática Simple 53

A Juana

For it is in teaching that we learn
And it is in understanding that we are understood

Agradecimientos/Acknowledgments

A mis alumnos de Procesadores de Lenguajes del Grado de Informática de la Escuela Superior de Informática en la Universidad de La Laguna

Parte I

PARTE: APUNTES DE PROCESADORES DE LENGUAJES

Capítulo 1

Expresiones Regulares y Análisis Léxico en JavaScript

1.1. Mozilla Developer Network: Documentación

1. RegExp Objects
2. exec
3. search
4. match
5. replace

1.2. Práctica: Conversor de Temperaturas

Véase https://bitbucket.org/casiano/pl-grado-temperature-converter/src.

```
[~/srcPLgrado/temperature(master)]$ pwd -P
/Users/casiano/local/src/javascript/PLgrado/temperature # 27/01/2014
```

index.html

```
<html>
  <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>JavaScript Temperature Converter</title>
      <link href="global.css" rel="stylesheet" type="text/css">

     <script type="text/javascript" src="temperature.js"></script>
  </head>
  <body>
    <h1>Temperature Converter</h1>
    <table>
      <tr>
        <th>Enter  Temperature (examples: 32F, 45C, -2.5f):</th>
        <td><input id="original" onchange="calculate();"></td>
      </tr>
      <tr>
        <th>Converted Temperature:</th>
```

```
        <td><span class="output" id="converted"></span></td>
      </tr>
    </table>
  </body>
</html>
```

Escribir HTML es farragoso. Una solución es usar algún plugin para su editor favorito. En el caso de vim podemos usar Emmet-vim decargandolo desde http://www.vim.org/. Emmet existe para diversos editores, entre ellos para Atom.

global.css

```
th, td      { vertical-align: top; text-align: right; font-size:large; }
#converted  { color: red; font-weight: bold; font-size:large;          }
input       { text-align: right;  border: none; font-size:large;       }
body
{
 background-color:#b0c4de;  /* blue */
 font-size:large;
}
```

temperature.js

```
"use strict"; // Use ECMAScript 5 strict mode in browsers that support it
function calculate() {
  var result;
  var original       = document.getElementById(".......");
  var temp = original.value;
  var regexp = /............................/;

  var m = temp.match(......);

  if (m) {
    var num = ....;
    var type = ....;
    num = parseFloat(num);
    if (type == 'c' || type == 'C') {
      result = (num * 9/5)+32;
      result = ..........................
    }
    else {
      result = (num - 32)*5/9;
      result = ........................
    }
    converted.innerHTML = result;
  }
  else {
    converted.innerHTML = "ERROR! Try something like '-4.2C' instead";
```

```
  }
}
```

Despliegue

- Deberá desplegar la aplicación en GitHub Pages como página de proyecto. Vea la sección *GitHub Project Pages* **??**.

Mocha y Chai Mocha is a test framework while Chai is an expectation one.

Mocha is the simple, flexible, and fun JavaScript unit-testing framework that runs in Node.js or in the browser.

It is open source (MIT licensed), and we can learn more about it at https://github.com/mochajs/mocha

Let's say Mocha setups and describes test suites and Chai provides convenient helpers to perform all kinds of assertions against your JavaScript code.

Pruebas: Estructura

Instale `mocha`.

```
$ npm install -g mocha
```

Creamos la estructura para las pruebas:

```
$ mocha init tests
```

```
$ tree tests
tests
|-- index.html
|-- mocha.css
|-- mocha.js
`-- tests.js
```

Añadimos `chai.js` (Véase http://chaijs.com/guide/installation/) al directorio `tests`.

Chai is a platform-agnostic BDD/TDD assertion library featuring several interfaces (for example, should, expect, and assert). It is open source (MIT licensed), and we can learn more about it at http://chaijs. com/

We can also install Chai on the command line using npm, as follows:

```
npm install chai --save-dev
```

The latest tagged version will be available for hot-linking at http://chaijs.com/chai.js.

If you prefer to host yourself, use the `chai.js` file from the root of the github project.

```
[~/srcPLgrado/temperature(master)]$ tree tests/
tests/
|-- chai.js
|-- index.html
|-- mocha.css
|-- mocha.js
`-- tests.js

0 directories, 5 files
```

Podemos encontrar un ejemplo de unit testing en JavaScript en el browser con el testing framework Mocha y Chai en el repositorio https://github.com/ludovicofischer/mocha-chai-browser-demo.

Pruebas: `index.html`

Modificamos `index.html` para

- Cargar `chai.js`
- Cargar `temperature.js`
- Usar el estilo `mocha.setup('tdd')`:
- Imitar la página `index.html` con los correspondientes `input` y `span`:

```
<input id="original" placeholder="32F" size="50">
<span class="output" id="converted"></span>
```

```
[~/srcPLgrado/temperature(master)]$ cat tests/index.html
<!DOCTYPE html>
<html>
  <head>
    <title>Mocha</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="mocha.css" />
  </head>
  <body>
    <div id="mocha"></div>
    <input id="original" placeholder="32F" size="50">
    <span class="output" id="converted"></span>

    <script src="chai.js"></script>
    <script src="mocha.js"></script>
    <script src="../temperature.js"></script>
    <script>mocha.setup('tdd')</script>
    <script src="tests.js"></script>

    <script>
      mocha.run();
    </script>
  </body>
</html>
```

Pruebas: Añadir los tests

The "TDDïnterface provides

- `suite()`
- `test()`
- `setup()`
- `teardown()`.

```
[~/srcPLgrado/temperature(master)]$ cat tests/tests.js
var assert = chai.assert;

suite('temperature', function() {
    test('32F = 0C', function() {
```

```
        original.value = "32F";
        calculate();
        assert.deepEqual(converted.innerHTML, "0.0 Celsius");
    });
    test('45C = 113.0 Farenheit', function() {
        original.value = "45C";
        calculate();
        assert.deepEqual(converted.innerHTML, "113.0 Farenheit");
    });
    test('5X = error', function() {
        original.value = "5X";
        calculate();
        assert.match(converted.innerHTML, /ERROR/);
    });
});
```

Pruebas: Véase

- Testing your frontend JavaScript code using mocha, chai, and sinon by Nicolas Perriault
- An example setup for unit testing JavaScript in the browser with the Mocha testing framework and Chai a
- Testing in Browsers and Node with Mocha, Chai, Sinon, and Testem

1.3. Práctica: Comma Separated Values. CSV

Donde

```
[~/srcPLgrado/csv(master)]$ pwd -P
/Users/casiano/local/src/javascript/PLgrado/csv
```

Véase https://bitbucket.org/casiano/pl-grado-csv/src y https://github.com/crguezl/csv.

Introducción al formato CSV

Véase Comma Separated Values en la Wikipedia:

A comma-separated values (CSV) file stores tabular data (numbers and text) in plain-text form. A CSV file consists of any number of records, separated by line breaks of some kind; each record consists of fields, separated by a comma. All records have an identical sequence of fields.

Ejemplo de ejecución

Véase la página en http://crguezl.github.io/csv/. Pruebe a dar como entrada cualquiera de estas dos

```
[~/srcPLgrado/csv(gh-pages)]$ cat input.txt
"producto",           "precio"
"camisa",             "4,3"
"libro de O\"Reilly", "7,2"

[~/srcPLgrado/csv(gh-pages)]$ cat input2.txt
"producto",           "precio"  "fecha"
"camisa",             "4,3",    "14/01"
"libro de O\"Reilly", "7,2"     "13/02"
```

Pruebe también a dar alguna entrada errónea.

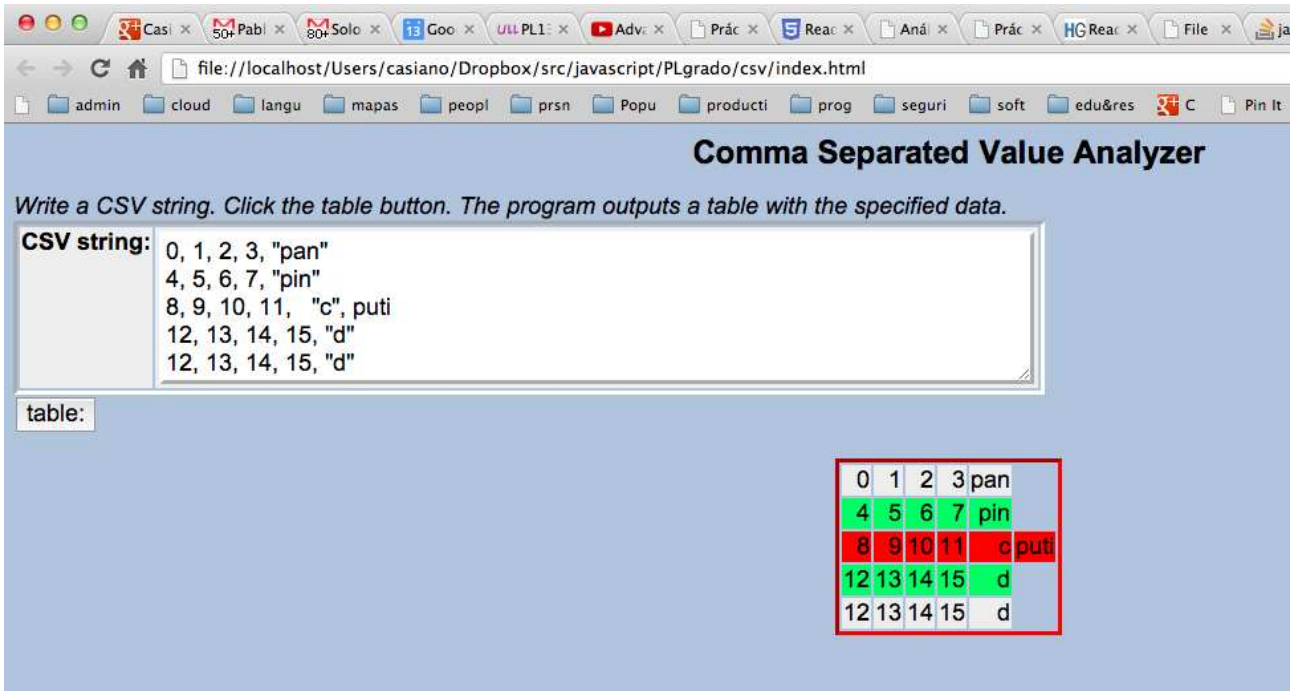


Figura 1.1: Ejemplo de pantalla de La aplicación para el Análisis de Datos en Formato CSV

Aproximación al análisis mediante expresiones regulares de CSV Una primera aproximación sería hacer `split` por las comas:

```
> x = '"earth",1,"moon",9.374'
'"earth",1,"moon",9.374'
> y = x.split(/,/)
[ '"earth"', '1', '"moon"', '9.374' ]
```

Esta solución deja las comillas dobles en los campos entrecomillados. Peor aún, los campos entrecomillados pueden contener comas, en cuyo caso la división proporcionada por `split` sería errónea:

```
> x = '"earth, mars",1,"moon, fobos",9.374'
'"earth, mars",1,"moon, fobos",9.374'
> y = x.split(/,/)
[ '"earth', ' mars"', '1', '"moon', ' fobos"', '9.374' ]
```

La siguiente expresión regular reconoce cadenas de comillas dobles con secuencias de escape seguidas opcionalmente de una coma:

```
> x = '"earth, mars",1,"moon, fobos",9.374'
'"earth, mars",1,"moon, fobos",9.374'
> r = /"((?:[^"\\]|\\.)*)"\s*,?/g
/"((?:[^"\\]|\\.)*)"\s*,?/g
> w = x.match(r)
[ '"earth, mars",', '"moon, fobos",' ]
```

If your regular expression uses the `g` flag, you can use the `exec` or `match` methods multiple times to find successive matches in the same string. When you do so, the search starts at the substring of string specified by the regular expression's `lastIndex` property.

Javascript sub-matches stop working when the `g` modifier is set:

```
> text = 'test test test test'
'test test test test'
> text.match(/t(e)(s)t/)
[ 'test', 'e', 's', index: 0, input: 'test test test test' ]
> text.match(/t(e)(s)t/g)
[ 'test', 'test', 'test', 'test' ]
```

Sin embargo el método `exec` de las expresiones regulares si que conserva las subexpresiones que casan con los paréntesis:

```
> r = /t(e)(s)t/g
/t(e)(s)t/g
> text = 'test test test test'
'test test test test'
> while (m = r.exec(text)) {
... console.log(m);
... }
[ 'test', 'e', 's', index: 0, input: 'test test test test' ]
[ 'test', 'e', 's', index: 5, input: 'test test test test' ]
[ 'test', 'e', 's', index: 10, input: 'test test test test' ]
[ 'test', 'e', 's', index: 15, input: 'test test test test' ]
undefined
```

Another catch to remember is that `exec()` doesn't return the matches in one big array: it keeps returning matches until it runs out, in which case it returns `null`.

Véase

- Javascript Regex and Submatches en StackOverflow.
- La sección *Ejercicios* 1.5

Esta otra expresión regular `/([^,]+),?|\s*,/` actúa de forma parecida al `split`. Reconoce secuencias no vacías de caracteres que no contienen comas seguidas opcionalmente de una coma o bien una sóla coma (precedida opcionalmente de blancos):

```
> x = '"earth, mars",1,"moon, fobos",9.374'
'"earth, mars",1,"moon, fobos",9.374'
> r = /([^,]+),?|\s*,/g
/([^,]+),?|\s*,/g
> w = x.match(r)
[ '"earth,', ' mars",', '1,', '"moon,', ' fobos",', '9.374' ]
```

La siguiente expresión regular es la unión de dos:

- Cadenas de dobles comillas seguidas de una coma opcional entre espacios en blanco
- Cadenas que no tienen comas

```
>  x = '"earth, mars",1,"moon, fobos",9.374'
'"earth, mars",1,"moon, fobos",9.374'
> r = /\s*"((?:[^"\\]|\\.)*)"\s*,?|\s*([^,]+),?|\s*,/g
/\s*"((?:[^"\\]|\\.)*)"\s*,?|\s*([^,]+),?|\s*,/g
> w = x.match(r)
[ '"earth, mars",', '1,', '"moon, fobos",', '9.374' ]
```

El operador `|` trabaja en circuito corto:

```
> r = /(ba?)|(b)/
/(ba?)|(b)/
> r.exec("ba")
[ 'ba', 'ba', undefined, index: 0, input: 'ba' ]
> r = /(b)|(ba?)/
/(b)|(ba?)/
> r.exec("ba")
[ 'b', 'b', undefined, index: 0, input: 'ba' ]
```

Si usamos `exec` tenemos:

```
> x = '"earth, mars",1,"moon, fobos",9.374'
'"earth, mars",1,"moon, fobos",9.374'
> r = /\s*"((?:[^"\\]|\\.)*)"\s*,?|\s*([^,]+),?|\s*,/g
/\s*"((?:[^"\\]|\\.)*)"\s*,?|\s*([^,]+),?|\s*,/g
> while (m = r.exec(x)) { console.log(m); }
[ '"earth, mars",', 'earth, mars', undefined, index: 0,
  input: '"earth, mars",1,"moon, fobos",9.374' ]
[ '1,', undefined, '1', index: 14,
  input: '"earth, mars",1,"moon, fobos",9.374' ]
[ '"moon, fobos",', 'moon, fobos', undefined, index: 16,
  input: '"earth, mars",1,"moon, fobos",9.374' ]
[ '9.374', undefined, '9.374', index: 30,
  input: '"earth, mars",1,"moon, fobos",9.374' ]
undefined
```

1. RegExp Objects

   The `RegExp` constructor creates a regular expression object for matching text with a pattern.

   Literal and constructor notations are possible:

   ```
   /pattern/flags;
   new RegExp(pattern [, flags]);
   ```

   - The literal notation provides compilation of the regular expression when the expression is evaluated.
   - Use literal notation when the regular expression will remain constant.
   - For example, if you use literal notation to construct a regular expression used in a loop, the regular expression won't be recompiled on each iteration.

   - The constructor of the regular expression object, for example, `new RegExp("ab+c")`, provides runtime compilation of the regular expression.
   - Use the constructor function when you know the regular expression pattern will be changing, or you don't know the pattern and are getting it from another source, such as user input.
   - When using the constructor function, the normal string escape rules (preceding special characters with `\` when included in a string) are necessary. For example, the following are equivalent:

     ```
     var re = /\w+/;
     var re = new RegExp("\\w+");
     ```

2. exec

3. search

   `str.search(regexp)`

   If successful, search returns the index of the regular expression inside the string. Otherwise, it returns `-1`.

   When you want to know whether a pattern is found in a string use search (similar to the regular expression `test` method); for more information (but slower execution) use `match` (similar to the regular expression `exec` method).

4. match

5. replace

   The `replace()` method returns a new string with some or all matches of a pattern replaced by a replacement. The pattern can be a string or a `RegExp`, and the replacement can be a string or a function to be called for each match.

```
> re = /apples/gi
/apples/gi
> str = "Apples are round, and apples are juicy."
'Apples are round, and apples are juicy.'
> newstr = str.replace(re, "oranges")
'oranges are round, and oranges are juicy.'
```

   The replacement string can be a function to be invoked to create the new substring (to put in place of the substring received from parameter `#1`). The arguments supplied to this function are:

Possible name	Supplied value
match	The matched substring. (Corresponds to `$&`.)
p1, p2, ...	The nth parenthesized submatch string, provided the first argument to re (Corresponds to `$1`, `$2`, etc.) For example, if `/(\a+)(\b+)/`, was given, p p2 for `\b+`.
offset	The offset of the matched substring within the total string being examine string was `abcd`, and the matched substring was `bc`, then this argument w
string	The total string being examined

```
[~/javascript/learning]$ pwd -P
/Users/casiano/local/src/javascript/learning
[~/javascript/learning]$ cat f2c.js
#!/usr/bin/env node
function f2c(x)
{
  function convert(str, p1, offset, s)
  {
    return ((p1-32) * 5/9) + "C";
  }
  var s = String(x);
  var test = /(\d+(?:\.\d*)?)F\b/g;
  return s.replace(test, convert);
}

var arg = process.argv[2] || "32F";
console.log(f2c(arg));
```

```
[~/javascript/learning]$ ./f2c.js 100F
37.77777777777778C
[~/javascript/learning]$ ./f2c.js
0C
```

index.html

```
<html>
  <head>
     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
     <title>CSV Analyzer</title>
     <link href="global.css" rel="stylesheet" type="text/css">

     <script type="text/javascript" src="../../underscore/underscore.js"></script>
     <script type="text/javascript" src="../../jquery/starterkit/jquery.js"></script>
     <script type="text/javascript" src="csv.js"></script>
  </head>
  <body>
    <h1>Comma Separated Value Analyzer</h1>
    <div>
      <i>Write a CSV string. Click the table button. The program outputs a table with the spec
    </div>
    <table>
      <tr>
        <th>CSV string:</th> <!-- autofocus attribute is HTML5 -->
        <td><textarea autofocus cols = "80" rows = "5" id="original"></textarea></td>
      </tr>
    </table>
    <button type="button">table:</button><br>
    <span class="output" id="finaltable"></span>
  </body>
</html>
```

jQuery

jQuery (Descarga la librería)
jQuery is a cross-platform JavaScript library designed to simplify the client-side scripting of HTML.

- It was released in January 2006 at BarCamp NYC by John Resig.
- It is currently developed by a team of developers led by Dave Methvin.
- jQuery is the most popular JavaScript library in use today
- jQuery's syntax is designed to make it easier to navigate a document, select DOM elements, create animations, handle events, and develop Ajax applications.
- The set of jQuery core features — DOM element selections, traversal and manipulation — enabled by its selector engine (named "Sizzle"from v1.3), created a new "programming style", fusing algorithms and DOM-data-structures; and influenced the architecture of other JavaScript frameworks like YUI v3 and Dojo.
-

How JQuery Works

- Véase How jQuery Works
- https://github.com/crguezl/how-jquery-works-tutorial en GitHub
- ```
  [~/javascript/jquery(master)]$ pwd -P
  /Users/casiano/local/src/javascript/jquery
  ```

```
~/javascript/jquery(master)]$ cat index.html
<!doctype html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Demo</title>
</head>
<body>
    <a href="http://jquery.com/">jQuery</a>
    <script src="starterkit/jquery.js"></script>
    <script>

    // Your code goes here.

    </script>
</body>
</html>
```

To ensure that their code runs after the browser finishes loading the document, many JavaScript programmers wrap their code in an onload function:

```
window.onload = function() { alert( "welcome" ); }
```

Unfortunately, the code doesn't run until all images are finished downloading, including banner ads. To run code as soon as the document is ready to be manipulated, jQuery has a statement known as the ready event:

```
$( document ).ready(function() {
    // Your code here.
});
```

For `click` and most other events, you can prevent the default behavior by calling event.preventDefault() in the event handler. If this method is called, the default action of the event will not be triggered. For example, clicked anchors will not take the browser to a new URL.

```
[~/javascript/jquery(master)]$ cat index2.html
<!doctype html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Demo</title>
</head>
<body>
    <a href="http://jquery.com/">jQuery</a>
    <script src="starterkit/jquery.js"></script>
    <script>
```
```

```
    $( document ).ready(function() {
        $( "a" ).click(function( event ) {
            alert( "The link will no longer take you to jquery.com" );
            event.preventDefault();
        });
    });

    </script>
</body>
</html>
```

Borrowing from CSS 1–3, and then adding its own, jQuery offers a powerful set of tools for matching a set of elements in a document.

See jQuery Category: Selectors.

Another common task is adding or removing a class. jQuery also provides some handy effects.

```
[~/javascript/jquery(master)]$ cat index3.html
<!doctype html>
<html>
<head>
    <meta charset="utf-8" />
    <style>
        a.test { font-weight: bold; }
    </style>
    <title>Demo</title>
</head>
<body>
    <a href="http://jquery.com/">jQuery</a>
    <script src="starterkit/jquery.js"></script>
    <script>

    $( document ).ready(function() {
        $( "a" ).click(function( event ) {
            $( "a" ).addClass( "test" );
            alert( "The link will no longer take you to jquery.com" );
            event.preventDefault();
            $( "a" ).removeClass( "test" );
            $( this ).hide( "slow" );
            $( this ).show( "slow" );
        });
    });

    </script>
</body>
</html>
```

- In JavaScript `this` always refers to the *owner* of the function we're executing, or rather, *to the object that a function is a method of.*
- When we define our function `tutu()` in a page, its owner is the page, or rather, the `window` object (or `global` object) of JavaScript.
- An `onclick` property, though, is owned by the HTML element it belongs to.

- The method .addClass(className) adds the specified class(es) to each of the set of matched elements.

  `className` is a String containing one or more space-separated classes to be added to the class attribute of each matched element.

  This method does not replace a class. It simply adds the class, appending it to any which may already be assigned to the elements.

- The method `.removeClass( [className ] )` removes a single class, multiple classes, or all classes from each element in the set of matched elements.

  If a class name is included as a parameter, then only that class will be removed from the set of matched elements. If no class names are specified in the parameter, all classes will be removed.

  This method is often used with `.addClass()` to switch elements' classes from one to another, like so:

  ```
  $( "p" ).removeClass( "myClass noClass" ).addClass( "yourClass" );
  ```

JavaScript enables you to freely pass functions around to be executed at a later time. A *callback* is a function that is passed as an argument to another function and is usually executed after its parent function has completed.

Callbacks are special because they wait to execute until their parent finishes or some event occurs. Meanwhile, the browser can be executing other functions or doing all sorts of other work.

```
[~/javascript/jquery(master)]$ cat app.rb
require 'sinatra'

set :public_folder, File.dirname(__FILE__) + '/starterkit'

get '/' do
  erb :index
end

get '/chuchu' do
  if request.xhr?
    "hello world!"
  else
    erb :tutu
  end
end

__END__

@@layout
  <!DOCTYPE html>
  <html>
    <head>
        <meta charset="utf-8" />
        <title>Demo</title>
    </head>
    <body>
        <a href="http://jquery.com/">jQuery</a>
        <div class="result"></div>
        <script src="jquery.js"></script>
        <%= yield %>
```

```
    </body>
  </html>

@@index
  <script>
  $( document ).ready(function() {
      $( "a" ).click(function( event ) {
          event.preventDefault();
          $.get( "/chuchu", function( data ) {
            $( ".result" ).html( data );
            alert( "Load was performed." );
          });
      });
  });
  </script>

@@tutu
  <h1>Not an Ajax Request!</h1>
```

- `jQuery.get( url [, data ] [, success(data, textStatus, jqXHR) ] [, dataType ] )` load data from the server using a HTTP GET request.
- `url`

  Type: String

  A string containing the URL to which the request is sent.
- `data`

  Type: PlainObject or String

  A plain object or string that is sent to the server with the request.
- `success(data, textStatus, jqXHR)`

  Type: Function()

  A callback function that is executed if the request succeeds.
- `dataType`

  Type: String

  The type of data expected from the server. Default: Intelligent Guess (`xml`, `json`, `script`, or `html`).

To use callbacks, it is important to know how to pass them into their parent function.
Executing callbacks with arguments can be tricky.
This code example will not work:

```
$.get( "myhtmlpage.html", myCallBack( param1, param2 ) );
```

The reason this fails is that the code executes

```
myCallBack( param1, param2)
```

immediately and then passes `myCallBack()`'s return value as the second parameter to `$.get()`.
We actually want to pass the function `myCallBack`, not `myCallBack( param1, param2 )`'s return value (which might or might not be a function).
So, how to pass in `myCallBack()` and include arguments?
To defer executing `myCallBack()` with its parameters, you can use an anonymous function as a wrapper.

```
[~/javascript/jquery(master)]$ cat app2.rb
require 'sinatra'

set :public_folder, File.dirname(__FILE__) + '/starterkit'

get '/' do
  erb :index
end

get '/chuchu' do
  if request.xhr? # is an ajax request
    "hello world!"
  else
    erb :tutu
  end
end

__END__

@@layout
  <!DOCTYPE html>
  <html>
    <head>
        <meta charset="utf-8" />
        <title>Demo</title>
    </head>
    <body>
        <a href="http://jquery.com/">jQuery</a>
        <div class="result"></div>
        <script src="jquery.js"></script>
        <%= yield %>
    </body>
  </html>

@@tutu
  <h1>Not an Ajax Request!</h1>

@@index
  <script>
    var param = "chuchu param";
    var handler = function( data, textStatus, jqXHR, param ) {
      $( ".result" ).html( data );
      alert( "Load was performed.\n"+
             "$data = "+data+
             "\ntextStatus = "+textStatus+
             "\njqXHR = "+JSON.stringify(jqXHR)+
             "\nparam = "+param );
    };
    $( document ).ready(function() {
        $( "a" ).click(function( event ) {
            event.preventDefault();
            $.get( "/chuchu", function(data, textStatus, jqXHR ) {
              handler( data, textStatus, jqXHR, param);
```

```
          });
       });
    });
  </script>
```

El ejemplo en `app2.rb` puede verse desplegado en Heroku: http://jquery-tutorial.herokuapp.com/

JSON.stringify() The JSON.stringify() method converts a value to `JSON` , optionally replacing values if a `replacer function` is specified, or optionally including only the specified properties if a `replacer array` is specified.

```
JSON.stringify(value[, replacer [, space]])
```

- `value`

  The value to convert to a JSON string.

- `replacer`

  - If a function, transforms values and properties encountered while stringifying;
  - if an array, specifies the set of properties included in objects in the final string.

- `space`

  Causes the resulting string to be pretty-printed.

See another example of use in http://jsfiddle.net/casiano/j7tsF/. To learn to use JSFiddle wath the YouTube video How to use JSFiddle by Jason Diamond

Underscore

Underscore: is a utility-belt library for JavaScript that provides a lot of the functional programming support that you would expect in Ruby.

- Underscore provides functions that support methods like:

  `map`, `select`, `invoke`

- as well as more specialized helpers:

  function binding, javascript templating, deep equality testing, and so on.

- Cargando la librería:

```
[~/javascript/jquery(master)]$ node
> 2+3
5
> _
5
> uu = require('underscore')
{ [Function]
  _: [Circular],
  VERSION: '1.5.2',
  forEach: [Function],
  each: [Function],
  collect: [Function],
  map: [Function],
  inject: [Function],
  reduce: [Function],
  ................
  chain: [Function] }
```

- each:

```
> uu.each([1, 2, 3], function(x) { console.log(x*x); })
1
4
9
```

- map:

```
> uu.map([1, 2, 3], function(num){ return num * 3; })
[ 3, 6, 9 ]
```

- invoke

```
> z = [[6,9,1],[7,3,9]]
[ [ 6, 9, 1 ], [ 7, 3, 9 ] ]
> uu.invoke(z, 'sort')
[ [ 1, 6, 9 ], [ 3, 7, 9 ] ]
> uu.invoke(z, 'sort', function(a, b) { return b-a; })
[ [ 9, 6, 1 ], [ 9, 7, 3 ] ]
```

- reduce:

```
> uu.reduce([1, 2, 3, 4], function(s, num){ return s + num; }, 0)
10
> uu.reduce([1, 2, 3, 4], function(s, num){ return s * num; }, 1)
24
> uu.reduce([1, 2, 3, 4], function(s, num){ return Math.max(s, num); }, -1)
4
> uu.reduce([1, 2, 3, 4], function(s, num){ return Math.min(s, num); }, 99)
1
```

- filter: (select is an alias for filter)

```
> uu.filter([1, 2, 3, 4, 5, 6], function(num){ return num % 2 == 0; })
[ 2, 4, 6 ]
```

- isEqual

```
> a = {a:[1,2,3], b: { c: 1, d: [5,6]}}
{ a: [ 1, 2, 3 ],
  b: { c: 1, d: [ 5, 6 ] } }
> b = {a:[1,2,3], b: { c: 1, d: [5,6]}}
{ a: [ 1, 2, 3 ],
  b: { c: 1, d: [ 5, 6 ] } }
> a == b
false
> uu.isEqual(a,b)
true
```

- bind

```
> func = function(greeting){ return greeting + ': ' + this.name }
[Function]
> func = uu.bind(func, {name: 'moe'})
[Function]
> func('hello')
'hello: moe'
> func = uu.bind(func, {name: 'moe'}, 'hi')
[Function]
> func()
'hi: moe'
>
```

Templates en Underscore

- Underscore: template

  `_.template(templateString, [data], [settings])`

  Compiles JavaScript templates into functions that can be evaluated for rendering. Useful for rendering complicated bits of HTML from a JavaScript object or from *JSON* data sources.

  JSON, or *JavaScript Object Notation*, is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. It is used primarily to transmit data between a server and web application, as an alternative to XML. Although originally derived from the JavaScript scripting language, JSON is a language-independent data format, and code for parsing and generating JSON data is readily available in a large variety of programming languages.

  - Template functions can both interpolate variables, using `<%= ... %>`,

    ```
    > compiled = uu.template("hello: <%= name %>")
    { [Function]
      source: 'function(obj){
        var __t,__p=\'\', __j=Array.prototype.join, i
            print=function(){__p+=__j.call(arguments,\'\');};
        with(obj||{}){
          __p+=\'hello: \'+ ((__t=( name ))==null?\'\':__t)+ \'\';
        }
        return __p;
      }'
    }
    > compiled({name: 'moe'})
    'hello: moe'
    ```

  - as well as execute arbitrary JavaScript code, with `<% ... %>`.

    ```
    > uu = require('underscore')
    > list = "\
    ... <% _.each(people, function(name) { %>\
    ..... <li><%= name %></li>\
    ... <% }); %>"
    '<% _.each(people, function(name) { %> <li><%= name %></li> <% }); %>'
    > uu.template(list, {people: ['moe', 'curly', 'larry']})
    ' <li>moe</li>  <li>curly</li>  <li>larry</li> '
    ```

  - When you evaluate a template function, pass in a data object that has properties corresponding to the template's free variables.

If you're writing a one-off, like in the example above, you can pass the data object as the second parameter to template in order to render immediately instead of returning a template function.

- If you wish to interpolate a value, and have it be HTML-escaped, use `<%- ... %>`

```
> template = uu.template("<b><%- value %></b>")
{ [Function]
  source: 'function(obj){
    var __t,__p=\'\',__j=Array.prototype.join,print=function(){__p+=__j.call(argument
    with(obj||{}){
      __p+=\'<b>\'+
      ((__t=( value ))==null?\'\':_.escape(__t))+
      \'</b>\';
    }
    return __p;
  }'
}
> template({value: '<script>'})
'<b>&lt;script&gt;</b>'
```

- The `settings` argument should be a hash containing any `_.templateSettings` that should be overridden.

```
_.template("Using 'with': <%= data.answer %>", {answer: 'no'}, {variable: 'data'});
=> "Using 'with': no"
```

Another example:

```
template = uu.template("<b>{{ value }}</b>",{value: 4 },
                       { interpolate: /\{\{(.+?)\}\}/g })
'<b>4</b>'
```

You can also use `print` from within JavaScript code. This is sometimes more convenient than using `<%= ... %>`.

```
> compiled = uu.template("<% print('Hello ' + epithet); %>")
{ [Function]
  source: 'function(obj){\n
    var __t,__p=\'\',
        __j=Array.prototype.join,print=function(){
            __p+=__j.call(arguments,\'\');};\n
            with(obj||{}){\n
              __p+=\'\';\n print(\'Hello \' + epithet); \n
              __p+=\'\';\n}\n
              return __p;\n
  }'
}
> compiled({ epithet : 'stooge' })
'Hello stooge'
>
```

If ERB-style delimiters aren't your cup of tea, you can change Underscore's template settings to use different symbols to set off interpolated code:

- Define an `interpolate` regex to match expressions that should be interpolated verbatim,

- an `escape` regex to match expressions that should be inserted after being HTML escaped, and
- an `evaluate` regex to match expressions that should be evaluated without insertion into the resulting string.
- You may define or omit any combination of the three.
- For example, to perform `Mustache.js` style templating:

```
_.templateSettings = {
  interpolate: /\{\{(.+?)\}\}/g
};

var template = _.template("Hello {{ name }}!");
template({name: "Mustache"});
=> "Hello Mustache!"
```

- escape:

```
> uu.templateSettings.escape = /\{\{-(.*?)\}\}/g
/\{\{-(.*?)\}\}/g
> compiled = uu.template("Escaped: {{- value }}\nNot escaped: {{ value }}")
{ [Function]
  source: 'function(obj){\nvar __t,__p=\'\',__j=Array.prototype.join,print=function()
> compiled({value: 'Hello, <b>world!</b>'})
'Escaped: Hello, &lt;b&gt;world!&lt;/b&gt;\nNot escaped: {{ value }}'
```

- Another example:

```
> uu.templateSettings = {
.....       interpolate: /\<\@\=(.+?)\@\>/gim,
.....       evaluate: /\<\@(.+?)\@\>/gim
.....   }
{ interpolate: /\<\@\=(.+?)\@\>/gim,
  evaluate: /\<\@(.+?)\@\>/gim }
> s = " <@ _.each([0,1,2,3,4], function(i) { @>  <p><@= i @></p> <@ }); @>"
' <@ _.each([0,1,2,3,4], function(i) { @>  <p><@= i @></p> <@ }); @>'
> uu.template(s,{})
'   <p>0</p>   <p>1</p>   <p>2</p>   <p>3</p>   <p>4</p> '
```

By default, template places the values from your data in the local scope via the `with` statement. The `with` statement adds the given object to the head of this scope chain during the evaluation of its statement body:

```
> with (Math) {
... s = PI*2;
... }
6.283185307179586
> z = { x : 1, y : 2 }
{ x: 1, y: 2 }
> with (z) {
... console.log(y);
... }
2
undefined
```

However, you can specify a single variable name with the variable `setting`. This improves the speed at which a template is able to render.

```
_.template("Using 'with': <%= data.answer %>", {answer: 'no'}, {variable: 'data'});
=> "Using 'with': no"
```

- JSFIDDLE: underscore templates
- Stackoverflow::how to use Underscore template

Content delivery network or content distribution network (CDN)

Una CDN que provee `underscore` esta en http://cdnjs.com/:

```
<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.5.2
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js"></script>
```

A content delivery network or content distribution network (CDN) is a large distributed system of servers deployed in multiple data centers across the Internet. The goal of a CDN is to serve content to end-users with high availability and high performance. CDNs serve a large fraction of the Internet content today, including

- web objects (text, graphics and scripts),
- downloadable objects (media files, software, documents), applications (e-commerce, portals),
- live streaming media, on-demand streaming media, and social networks.

Google provee también un servicio CDN para los desarrolladores en https://developers.google.com/speed/librar

textarea, autofocus y button

1. textarea:

   The `<textarea>` tag defines a multi-line text input control.

   A *text area* can hold an unlimited number of characters, and the text renders in a fixed-width font (usually Courier).

   The size of a text area can be specified by the `cols` and `rows` attributes, or through CSS' `height` and `width` properties.

   `cols` and `rows` consider the font size. `height` and `width` aren't.

2. autofocus.

   The autofocus attribute is a boolean attribute.

   When present, it specifies that the text area should automatically get focus when the page loads.

   Véase también [1]

3. button:

   The `<button>` tag defines a clickable button.

   Inside a `<button>` element you can put content, like text or images.

Local Storage (HTML5 Web Storage)

Web storage and *DOM storage* (document object model) are web application software methods and protocols used for storing data in a web browser.

- Web storage supports persistent data storage, similar to cookies but with a greatly enhanced capacity and no information stored in the HTTP request header.
- Local Storage nos permite almacenar hasta 5MB del lado del cliente por dominio, esto nos permite ahora hacer aplicaciones mas robustas y con mas posibilidades. Las Cookies ofrecen algo parecido, pero con el limite de 100kb.

- There are two main web storage types: *local storage* and *session storage*, behaving similarly to persistent cookies and session cookies respectively.
- Unlike cookies, which can be accessed by both the server and client side, web storage falls exclusively under the purview of client-side scripting
- The HTML5 localStorage object is isolated per domain (the same segregation rules as the same origin policy).

  The same-origin policy permits scripts running on pages originating from the same site – a combination of scheme, hostname, and port number – to access each other's DOM with no specific restrictions, but prevents access to DOM on different sites.

Véase:

- Ejemplo en GitHub: https://github.com/crguezl/web-storage-example

  ```
  [~/javascript/local_storage(master)]$ pwd -P
  /Users/casiano/local/src/javascript/local_storage
  ```

- Como usar localstorage
- HTML5 Web Storage
- W3C Web Storage
- Using HTML5 localStorage To Store JSON Options for persistent storage of complex JavaScript objects in HTML5 by Dan Cruickshank
- HTML5 Cookbook. Christopher Schmitt, Kyle Simpson .ºReilly Media, Inc.", Nov 7, 2011 Chapter 10. Section 2: LocalStorage

While Chrome does not provide a UI for clearing localStorage, there is an API that will either clear a specific key or the entire localStorage object on a website.

```
//Clears the value of MyKey
window.localStorage.clear("MyKey");

//Clears all the local storage data
window.localStorage.clear();
```

Once done, localStorage will be cleared. Note that this affects all web pages on a single domain, so if you clear localStorage for `jsfiddle.net/index.html` (assuming that's the page you're on), then it clears it for all other pages on that site.

global.css

```
html *
{
   font-size: large;
   /* The !important ensures that nothing can override what you've set in this style (unless i
   font-family: Arial;
}

h1            { text-align: center; font-size: x-large; }
th, td        { vertical-align: top; text-align: right; }
/* #finaltable  * { color: white; background-color: black; }   */
```

```
/* #finaltable table { border-collapse:collapse; } */
/* #finaltable table, td { border:1px solid white; } */
#finaltable:hover td { background-color: blue; }
tr:nth-child(odd)    { background-color:#eee; }
tr:nth-child(even)    { background-color:#00FF66; }
input        { text-align: right;  border: none;       }     /* Align input to the right  */
textarea     { border: outset; border-color: white;       }
table        { border: inset; border-color: white; }
table.center { margin-left:auto; margin-right:auto; }
#result      { border-color: red; }
tr.error       { background-color: red; }
body
{
 background-color:#b0c4de;  /* blue */
}
```

1. Introducción a las pseudo clases de CSS3

   Una pseudo clase es un estado o uso predefinido de un elemento al que se le puede aplicar un estilo independientemente de su estado por defecto. Existen cuatro tipos diferentes de pseudo clases:

   - Links: Estas pseudo clases se usan para dar estilo al enlace tanto en su estado normal por defecto como cuando ya ha sido visitado, mientras mantenemos el cursor encima de él o cuando hacemos click en él
   - Dinamicas: Estas pseudo clases pueden ser aplicadas a cualquier elemento para definir como se muestran cuando el cursor está situado sobre ellos, o haciendo click en ellos o bien cuando son seleccionados
   - Estructurales: Permiten dar estilo a elementos basándonos en una posición numérica exacta del elemento
   - Otras: Algunos elementos pueden ser estilizados de manera diferente basándonos en el lenguaje o que tipo de etiqueta no son

2. CSS pattern matching

   In CSS, *pattern matching* rules determine which style rules apply to elements in the document tree. These patterns, called selectors, may range from simple element names to rich contextual patterns. If all conditions in the pattern are true for a certain element, the selector matches the element.

   The universal selector, written `*`, matches the name of any element type. It matches any single element in the document tree.

   For example, this rule set will be applied to every element in a document:

   ```
   * {
     margin: 0;
     padding: 0;
   }
   ```

3. CSS class selectors

   Working with HTML, authors may use the period (`.`) notation as an alternative to the `~=` notation when representing the class attribute. Thus, for HTML, `div.value` and `div[class~=value]` have the same meaning. The attribute value must immediately follow the *period* (`.`).

4. CSS3: nth-child() selector The `:nth-child(n)` selector matches every element that is the `n`th child, regardless of type, of its parent.

   `n` can be a number, a keyword, or a formula.

5. The CSS border properties allow you to specify the style and color of an element's border. The border-style property specifies what kind of border to display. For example, `inset:` Defines a 3D inset border while `:outset` defines a 3D outset border. The effect depends on the border-color value

   See CSS: border

6.

csv.js

```
// See http://en.wikipedia.org/wiki/Comma-separated_values
"use strict"; // Use ECMAScript 5 strict mode in browsers that support it

$(document).ready(function() {
   $("button").click(function() {
     calculate();
   });
 });

function calculate() {
  var result;
  var original       = document.getElementById("original");
  var temp = original.value;
  var regexp = /_____________________________________________/g;
  var lines = temp.split(/\n+\s*/);
  var commonLength = NaN;
  var r = [];
  // Template using underscore
  var row = "<%% _.each(items, function(name) { %>"     +
            "                      <td><%%= name %></td>" +
            "               <%% }); %>";

  if (window.localStorage) localStorage.original  = temp;

  for(var t in lines) {
    var temp = lines[t];
    var m = temp.match(regexp);
    var result = [];
    var error = false;

    if (m) {
      if (commonLength && (commonLength != m.length)) {
        //alert('ERROR! row <'+temp+'> has '+m.length+' items!');
        error = true;
      }
      else {
        commonLength = m.length;
        error = false;
      }
```

```
      for(var i in m) {
        var removecomma = m[i].replace(/,\s*$/,'');
        var remove1stquote = removecomma.replace(/^\s*"/,'');
        var removelastquote = remove1stquote.replace(/"\s*$/,'');
        var removeescapedquotes = removelastquote.replace(/\"/,'"');
        result.push(removeescapedquotes);
      }
      var tr = error? '<tr class="error">' : '<tr>';
      r.push(tr+_.template(row, {items : result})+"</tr>");
    }
    else {
      alert('ERROR! row '+temp+' does not look as legal CSV');
      error = true;
    }
  }
  r.unshift('<p>\n<table class="center" id="result">');
  r.push('</table>');
  //alert(r.join('\n')); // debug
  finaltable.innerHTML = r.join('\n');
}

window.onload = function() {
  // If the browser supports localStorage and we have some stored data
  if (window.localStorage && localStorage.original) {
    document.getElementById("original").value = localStorage.original;
  }
};
```

1. Tutorials:Getting Started with jQuery

Tareas

- Añada pruebas usando Mocha y Chai

1.4. Comentarios y Consejos

How can I push a local Git branch to a remote with a different name easily?

```
$ git branch -a
* gh-pages
  remotes/origin/HEAD -> origin/gh-pages
  remotes/origin/gh-pages
```

Of course a solution for this way to work is to rename your master branch:

```
$ git branch -m master gh-pages
[~/Downloads/tmp(gh-pages)]$ git branch
* gh-pages
```

Otherwise, you can do your initial push this way:

```
$ git push -u origin master:gh-pages
```

Option `-u`: for every branch that is up to date or successfully pushed, add `upstream` (tracking) reference, used by argument-less `git-pull`.

- How can I push a local Git branch to a remote with a different name easily?

favicons y shortcut icons

- A *favicon* (short for *Favorite icon*), also known as a *shortcut* icon, is a file containing one or more small icons, most commonly 16×16 pixels, associated with a particular Web site or Web page.
- A web designer can create such an icon and install it into a Web site (or Web page) by several means, and graphical web browsers will then make use of it.
- Browsers that provide favicon support typically display a page's favicon in the browser's address bar (sometimes in the history as well) and next to the page's name in a list of bookmarks.
- Browsers that support a tabbed document interface typically show a page's favicon next to the page's title on the tab
- Some services in the cloud to generate favicons:
  - Favicon Generator
  - favicon.cc
- En `index.html` poner una línea como una de estas:

```
<link rel="shortcut icon" href="etsiiull.png" type="image/x-icon">
<link rel="shortcut icon" href="logo.png" />
<link href="images/favicon.ico" rel="icon" type="image/x-icon" />
```

1.5. Ejercicios

1. Paréntesis:

```
> str = "John Smith"
'John Smith'
> newstr = str.replace(re, "$2, $1")
'Smith, John'
```

2. El método `exec`.

   If your regular expression uses the `g` flag, you can use the `exec` method multiple times to find successive matches in the same string. When you do so, the search starts at the substring of str specified by the regular expression's `lastIndex` property.

```
> re = /d(b+)(d)/ig
/d(b+)(d)/gi
> z = "dBdxdbbdzdbd"
'dBdxdbbdzdbd'
> result = re.exec(z)
[ 'dBd', 'B', 'd', index: 0, input: 'dBdxdbbdzdbd' ]
> re.lastIndex
3
> result = re.exec(z)
[ 'dbbd', 'bb', 'd', index: 4, input: 'dBdxdbbdzdbd' ]
```

```
> re.lastIndex
8
> result = re.exec(z)
[ 'dbd', 'b', 'd', index: 9, input: 'dBdxdbbdzdbd' ]
> re.lastIndex
12
> z.length
12
> result = re.exec(z)
null
```

3. JavaScript tiene lookaheads:

```
> x = "hello"
'hello'
> r = /l(?=o)/
/l(?=o)/
> z = r.exec(x)
[ 'l', index: 3, input: 'hello' ]
```

4. JavaScript no tiene lookbehinds:

```
> x = "hello"
'hello'
> r = /(?<=l)l/
SyntaxError: Invalid regular expression: /(?<=l)l/: Invalid group
> .exit

[~/Dropbox/src/javascript/PLgrado/csv(master)]$ irb
ruby-1.9.2-head :001 > x = "hello"
 => "hello"
ruby-1.9.2-head :002 > r = /(?<=l)l/
 => ll
ruby-1.9.2-head :008 > x =~ r
 => 3
ruby-1.9.2-head :009 > $&
 => "l"
```

5. El siguiente ejemplo comprueba la validez de números de teléfono:

```
[~/local/src/javascript/PLgrado/regexp]$ pwd -P
/Users/casiano/local/src/javascript/PLgrado/regexp
[~/local/src/javascript/PLgrado/regexp]$ cat phone.html
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <meta http-equiv="Content-Script-Type" content="text/javascript">
    <script type="text/javascript">
      var re = /\(?\d{3}\)?([-\/\.])\d{3}\1\d{4}/;
      function testInfo(phoneInput){
        var OK = re.exec(phoneInput.value);
        if (!OK)
```

```
          window.alert(RegExp.input + " isn't a phone number with area code!");
        else
          window.alert("Thanks, your phone number is " + OK[0]);
      }
    </script>
  </head>
  <body>
    <p>Enter your phone number (with area code) and then click "Check".
        <br>The expected format is like ###-###-####.</p>
    <form action="#">
      <input id="phone"><button onclick="testInfo(document.getElementById('phone'));">Che
    </form>
  </body>
</html>
```

6. ¿Con que cadenas casa la expresión regular `/^(11+)\1+$/`?

```
> '1111'.match(/^(11+)\1+$/) # 4 unos
[ '1111',
  '11',
  index: 0,
  input: '1111' ]
> '111'.match(/^(11+)\1+$/) # 3 unos
null
> '11111'.match(/^(11+)\1+$/) # 5 unos
null
> '111111'.match(/^(11+)\1+$/) # 6 unos
[ '111111',
  '111',
  index: 0,
  input: '111111' ]
> '11111111'.match(/^(11+)\1+$/) # 8 unos
[ '11111111',
  '1111',
  index: 0,
  input: '11111111' ]
> '1111111'.match(/^(11+)\1+$/)
null
>
```

Busque una solución al siguiente ejercicio (véase 'Regex to add space after punctuation sign' en PerlMonks) Se quiere poner un espacio en blanco después de la aparición de cada coma:

7.
```
> x = "a,b,c,1,2,d, e,f"
'a,b,c,1,2,d, e,f'
> x.replace(/,/g,", ")
'a, b, c, 1, 2, d,  e, f'
```

pero se quiere que la sustitución no tenga lugar si la coma esta incrustada entre dos dígitos. Además se pide que si hay ya un espacio después de la coma, no se duplique.

a) La siguiente solución logra el segundo objetivo, pero estropea los números:

```
> x = "a,b,c,1,2,d, e,f"
'a,b,c,1,2,d, e,f'
> x.replace(/,(\S)/g,", $1")
'a, b, c, 1, 2, d, e, f'
```

b) Esta otra funciona bien con los números pero no con los espacios ya existentes:

```
> x = "a,b,c,1,2,d, e,f"
'a,b,c,1,2,d, e,f'
> x.replace(/,(\D)/g,", $1")
'a, b, c,1,2, d,  e, f'
```

c) Explique cuando casa esta expresión regular:

```
> r = /(\d[,.]\d)|(,(?=\S))/g
/(\d[,.]\d)|(,(?=\S))/g
```

Aproveche que el método `replace` puede recibir como segundo argumento una función (vea replace):

```
> z = "a,b,1,2,d, 3,4,e"
'a,b,1,2,d, 3,4,e'
> f = function(match, p1, p2, offset, string) { return (p1 || p2 + " "); }
[Function]
> z.replace(r, f)
'a, b, 1,2, d, 3,4, e'
```

1.6. Práctica: Palabras Repetidas

Se trata de producir una salida en las que las palabras repetidas consecutivas sean reducidas a una sola aparición. Rellena las partes que faltan.

Donde

```
[~/srcPLgrado/repeatedwords(master)]$ pwd -P
/Users/casiano/local/src/javascript/PLgrado/repeatedwords
[~/srcPLgrado/repeatedwords(master)]$ git remote -v
origin  ssh://git@bitbucket.org/casiano/pl-grado-repeated-words.git (fetch)
origin  ssh://git@bitbucket.org/casiano/pl-grado-repeated-words.git (push)
```

Véase: https://bitbucket.org/casiano/pl-grado-repeated-words

Ejemplo de ejecución

Estructura

index.html

```
[~/Dropbox/src/javascript/PLgrado/repeatedwords(master)]$ cat index.html
<html>
  <head>
     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
     <title>File Input</title>
     <link href="global.css" rel="stylesheet" type="text/css">

     <script type="text/javascript" src="../../underscore/underscore.js"></script>
     <script type="text/javascript" src="../../jquery/starterkit/jquery.js"></script>
     <script type="text/javascript" src="repeated_words.js"></script>
  </head>
  <body>
    <h1>File Input</h1>
```

```
    <input type="file" id="fileinput" />
    <div id="out" class="hidden">
    <table>
      <tr><th>Original</th><th>Transformed</th></tr>
      <tr>
        <td>
          <pre class="input" id="initialinput"></pre>
        </td>
        <td>
          <pre class="output" id="finaloutput"></pre>
        </td>
      </tr>
    </table>
    </div>
  </body>
</html>
```

1. Tag input

global.css

Rellena los estilos para `hidden` y `unhidden`:

```
[~/Dropbox/src/javascript/PLgrado/repeatedwords(master)]$ cat global.css
html *
{
   font-size: large;
   /* The !important ensures that nothing can override what you've set in this style
      (unless it is also important). */
   font-family: Arial;
}

.thumb {
    height: 75px;
    border: 1px solid #000;
    margin: 10px 5px 0 0;
  }

h1            { text-align: center; font-size: x-large; }
th, td        { vertical-align: top; text-align: right; }
/* #finaltable  * { color: white; background-color: black; }   */

/* #finaltable table { border-collapse:collapse; } */
/* #finaltable table, td { border:1px solid white; } */
#finaltable:hover td { background-color: blue; }
tr:nth-child(odd)    { background-color:#eee; }
tr:nth-child(even)    { background-color:#00FF66; }
input        { text-align: right;  border: none;       }     /* Align input to the right  */
textarea     { border: outset; border-color: white;       }
table        { border: inset; border-color: white; }
.hidden      { display: ____; }
.unhidden    { display: _____; }
table.center { margin-left:auto; margin-right:auto; }
#result      { border-color: red; }
tr.error       { background-color: red; }
```

```
pre.output   { background-color: white; }
span.repeated { background-color: red }

body
{
 background-color:#b0c4de;  /* blue */
```

1. CSS display Property
2. Diferencias entre "Displayz "Visibility"

repeated_words.js

Rellena las expresiones regulares que faltan:

```
[~/srcPLgrado/repeatedwords(master)]$ cat repeated_words.js
"use strict"; // Use ECMAScript 5 strict mode in browsers that support it

$(document).ready(function() {
   $("#fileinput").change(calculate);
});

function generateOutput(contents) {
  return contents.replace(/____________________/__,'___________________________________');
}

function calculate(evt) {
  var f = evt.target.files[0];
  var contents = '';

  if (f) {
    var r = new FileReader();
    r.onload = function(e) {
      contents = e.target.result;
      var escaped  = escapeHtml(contents);
      var outdiv = document.getElementById("out");
      outdiv.className = 'unhidden';
      finaloutput.innerHTML = generateOutput(escaped);
      initialinput.innerHTML = escaped;

    }
    r.readAsText(f);
  } else {
    alert("Failed to load file");
  }
}

var entityMap = {
    "&": "&amp;",
    "<": "&lt;",
    ">": "&gt;",
    '"': '&quot;',
    "'": '&#39;',
    "/": '&#x2F;'
  };
```

```
function escapeHtml(string) {
  return String(string).replace(/_________/g, function (s) {
    return ___________;
  });
```

1. jQuery event.target
2. HTML 5 File API
3. HTML 5 File API: FileReader
4. HTML 5 File API: FileReader
5. element.className
6. HTML Entities
7. Tutorials:Getting Started with jQuery
8. Underscore: template

Ficheros de Entrada

```
[~/Dropbox/src/javascript/PLgrado/repeatedwords(master)]$ cat input2.txt
habia una vez
vez un viejo viejo
hidalgo que vivia
vivia
[~/Dropbox/src/javascript/PLgrado/repeatedwords(master)]$ cat input.txt
one one
nothing rep
is two three
three four
[~/Dropbox/src/javascript/PLgrado/repeatedwords(master)]$ cat inputhtml1.txt
habia => una vez
vez & un viejo viejo <puchum>
hidalgo  & <pacham> que vivia
vivia </que se yo>
```

1.7. Ejercicios

El formato *INI* es un formato estandar para la escritura de ficheros de configuración. Su estrucutra básica se compone de "seccionesz "propiedades". Véase la entrada de la wikipedia INI.

```
; last modified 1 April 2001 by John Doe
[owner]
name=John Doe
organization=Acme Widgets Inc.

[database]
; use IP address in case network name resolution is not working
server=192.0.2.62
port=143
file = "payroll.dat"
```

1. Escriba un programa javascript que obtenga las cabeceras de sección de un fichero INI
2. Escriba un programa javascript que case con los bloques de un fichero INI (cabecera mas lista de pares `parámetro=valor`)
3. Se quieren obtener todos los pares nombre-valor, usando paréntesis con memoria para capturar cada parte.
4. ¿Que casa con cada paréntesis en esta regexp para los pares nombre-valor?

```
> x = "h     = 4"
> r = /([^=]*)(\s*)=(\s*)(.*)/
> r.exec(x)
>
```

1.8. Ejercicios

1. Escriba una expresión regular que reconozca las cadenas de doble comillas. Debe permitir la presencia de comillas y caracteres escapados.
2. ¿Cual es la salida?

```
> "bb".match(/b|bb/)

> "bb".match(/bb|b/)
```

1.9. Práctica: Ficheros INI

Donde

```
[~/srcPLgrado/ini(develop)]$ pwd -P
/Users/casiano/local/src/javascript/PLgrado/ini
[~/srcPLgrado/ini(develop)]$ git remote -v
origin  ssh://git@bitbucket.org/casiano/pl-grado-ini-files.git (fetch)
origin  ssh://git@bitbucket.org/casiano/pl-grado-ini-files.git (push)
```

Véase

- Repositorio conteniendo el código (inicial) del analizador de ficheros `ini`: https://github.com/crguezl/pl-grad
- Despliegue en GitHub pages: http://crguezl.github.io/pl-grado-ini-files/
- Repositorio privado del profesor: https://bitbucket.org/casiano/pl-grado-ini-files/src.

index.html

```
[~/javascript/PLgrado/ini(master)]$ cat index.html
<html>
  <head>
     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
     <title>INI files</title>
     <link href="global.css" rel="stylesheet" type="text/css">
<!--
     <link rel="shortcut icon" href="logo.png" />
-->
```

```
    <link rel="shortcut icon" href="etsiiull.png" type="image/x-icon">

    <script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/underscore.js/
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js"></script>

    <script type="text/javascript" src="ini.js"></script>
  </head>
  <body>
    <h1>INI files</h1>
    <input type="file" id="fileinput" />
    <div id="out" class="hidden">
    <table>
      <tr><th>Original</th><th>Tokens</th></tr>
      <tr>
        <td>
          <pre class="input" id="initialinput"></pre>
        </td>
        <td>
          <pre class="output" id="finaloutput"></pre>
        </td>
      </tr>
    </table>
    </div>
  </body>
</html>
```

Ficheros

Vease

- Reading files in JavaScript using the File APIs by Eric Bidelman.

  Source code in files-in-javascript-tut

- W3C File API

- Ejemplo `FileList` en

  - github
  - en acción en gh-pages.
  - Tambien en jsfiddle
  - o bien

    ```
    [~/src/javascript/fileapi/html5rocks]$ pwd -P
    /Users/casiano/local/src/javascript/fileapi/html5rocks
    [~/src/javascript/fileapi/html5rocks(master)]$ ls -l filelist.html
    -rw-r--r--  1 casiano  staff  767 15 feb 17:21 filelist.html
    ```

  - The EventTarget.addEventListener() method

    ```
    target.addEventListener(type, listener[, useCapture]);
    ```

    registers the specified listener on the `EventTarget` it's called on. The event target may be an Element in a document, the Document itself, a Window, or any other object that supports events (such as `XMLHttpRequest`).

    - ```
      > date = new Date(Date.UTC(2012, 11, 12, 3, 0, 0));
      Wed Dec 12 2012 03:00:00 GMT+0000 (WET)
      > date.toLocaleDateString()
      "12/12/2012"
      ```

    - Date.prototype.toLocaleDateString()

- Ejemplo de Drag and Drop en

    - GitHub
    - gh-pages
    - jsfiddle

  o bien en:

  ```
  [~/src/javascript/fileapi/html5rocks]$ pwd -P
  /Users/casiano/local/src/javascript/fileapi/html5rocks
  [~/src/javascript/fileapi/html5rocks]$ ls -l dragandrop.html
  -rw-r--r--  1 casiano  staff  1535 15 feb 18:25 dragandrop.html
  ```

    - `stopPropagation` stops the event from bubbling up the event chain.
      Suppose you have a table and within that table you have an anchor tag. Both the table and the anchor tag have code to handle mouse clicks. When the user clicks on the anchor tag, which HTML element should process the event first? Should it be the table then the anchor tag or vice versa?

      Formally, the event path is broken into three phases.

        - In the *capture phase*, the event starts at the top of the DOM tree, and propagates through to the parent of the target.
        - In the *target phase*, the event object arrives at its target. This is generally where you will write your event-handling code.
        - In the *bubble phase*, the event will move back up through the tree until it reaches the top. Bubble phase propagation happens in reverse order to the capture phase, with an event starting at the parent of the target and ending up back at the top of the DOM tree.
        - jsfiddle

      These days, there's a choice to register an event in either the capture phase or the bubble phase. If you register an event in the capture phase, the parent element will process the event before the child element.

    - `preventDefault` prevents the default action the browser makes on that event.

- After you've obtained a `File` reference, instantiate a `FileReader` object to read its contents into memory.

  ```
  var reader = new FileReader();
  ```

  to read the file we call one of the readAs... For example readAsDataURL is used to starts reading the contents of the specified Blob or File:

  ```
  reader.readAsDataURL(f);
  ```

- Methods to remember:
  ```

- `FileReader.abort()` Aborts the read operation. Upon return, the `readyState` will be DONE.
- `FileReader.readAsArrayBuffer()` Starts reading the contents of the specified `Blob`, once finished, the `result` attribute contains an `ArrayBuffer` representing the file's data.
- `FileReader.readAsBinaryString()` Starts reading the contents of the specified `Blob`, once finished, the result attribute contains the raw binary data from the file as a string.
- `FileReader.readAsDataURL()` Starts reading the contents of the specified `Blob`.

  When the read operation is finished, the `readyState` becomes `DONE`, and the `loadend` is triggered. At that time, the result attribute contains a URL representing the file's data as base64 encoded string.
- `FileReader.readAsText()` Starts reading the contents of the specified `Blob`, once finished, the result attribute contains the contents of the file as a text string.

Once one of these read methods is called on your `FileReader` object, the `onloadstart`, `onprogress`, `onload`, `onabort`, `onerror`, and `onloadend` can be used to track its progress.

- When the load finishes, the reader's `onload` event is fired and its result attribute can be used to access the file data.

```
reader.onload = function(e) {
  var contents = e.target.result;

  ....
}
```

See

- jsfiddle
- GitHub
- gh-pages
- or

```
[~/src/javascript/fileapi/html5rocks]$ pwd -P
/Users/casiano/local/src/javascript/fileapi/html5rocks
[~/src/javascript/fileapi/html5rocks]$ ls -l readimages.html
-rw-r--r--  1 casiano  staff  1530 15 feb 21:00 readimages.html
```

- base64 testing image jsfiddle
- The `insertBefore()` method inserts a node as a child, right before an existing child, which you specify. See

```
[~/src/javascript/fileapi/html5rocks]$ ls -l readimages.html
-rw-r--r--  1 casiano  staff  1530 15 feb 21:00 readimages.html
```

global.css

```
[~/javascript/PLgrado/ini(master)]$ cat global.css
html *
{
   font-size: large;
   /* The !important ensures that nothing can override what you've set in this style (unless i
```

```
   font-family: Arial;
}

.thumb {
    height: 75px;
    border: 1px solid #000;
    margin: 10px 5px 0 0;
  }

h1            { text-align: center; font-size: x-large; }
th, td        { vertical-align: top; text-align: left; }
/* #finaltable  * { color: white; background-color: black; }   */

/* #finaltable table { border-collapse:collapse; } */
/* #finaltable table, td { border:1px solid white; } */
#finaltable:hover td { background-color: blue; }
tr:nth-child(odd)    { background-color:#eee; }
tr:nth-child(even)    { background-color:#00FF66; }
input        { text-align: right;  border: none;       }     /* Align input to the right  */
textarea     { border: outset; border-color: white;       }
table        { border: inset; border-color: white; }
.hidden      { display: none; }
.unhidden    { display: block; }
table.center { margin-left:auto; margin-right:auto; }
#result      { border-color: red; }
tr.error       { background-color: red; }
pre.output   { background-color: white; }
/*
span.repeated { background-color: red }
span.header { background-color: blue }
span.comments { background-color: orange }
span.blanks { background-color: green }
span.nameEqualValue { background-color: cyan }
span.error { background-color: red }
*/
body
{
 background-color:#b0c4de;  /* blue */
}
```

Ficheros de Prueba

```
~/Dropbox/src/javascript/PLgrado/ini(master)]$ cat input.ini
; last modified 1 April 2001 by John Doe
[owner]
name=John Doe
organization=Acme Widgets Inc.

[database]
; use IP address in case network name resolution is not working
server=192.0.2.62
port=143
file = "payroll.dat"
```

```
$ cat input2.ini
[special_fields]
required = "EmailAddr,FirstName,LastName,Mesg"
csvfile = "contacts.csv"
csvcolumns = "EmailAddr,FirstName,LastName,Mesg,Date,Time"

[email_addresses]
sales = "jack@yahoo.com,mary@my-sales-force.com,president@my-company.com"
```

```
$ cat inputerror.ini
[owner]
name=John Doe
organization$Acme Widgets Inc.

[database
; use IP address in case network name resolution is not working
server=192.0.2.62
port=143
file = "payroll.dat"
```

ini.js

```
[~/javascript/PLgrado/ini(master)]$ cat ini.js
"use strict"; // Use ECMAScript 5 strict mode in browsers that support it

$(document).ready(function() {
   $("#fileinput").change(calculate);
});

function calculate(evt) {
  var f = evt.target.files[0];

  if (f) {
    var r = new FileReader();
    r.onload = function(e) {
      var contents = e.target.result;

      var tokens = lexer(contents);
      var pretty = tokensToString(tokens);

      out.className = 'unhidden';
      initialinput.innerHTML = contents;
      finaloutput.innerHTML = pretty;
    }
    r.readAsText(f);
  } else {
    alert("Failed to load file");
  }
}

var temp = '<li> <span class = "<%= token.type %>"> <%= match %> </span>\n';

function tokensToString(tokens) {
   var r = '';
```

```
   for(var i=0; i < tokens.length; i++) {
     var t = tokens[i]
     var s = JSON.stringify(t, undefined, 2);
     s = _.template(temp, {token: t, match: s});
     r += s;
   }
   return '<ol>\n'+r+'</ol>';
}

function lexer(input) {
  var blanks         = /^\s+/;
  var iniheader      = /^\[([^\]\r\n]+)\]/;
  var comments       = /^[;#](.*)/;
  var nameEqualValue = /^([^=;\r\n]+)=([^;\r\n]*)/;
  var any            = /^(.|\n)+/;

  var out = [];
  var m = null;

  while (input != '') {
    if (m = blanks.exec(input)) {
      input = input.substr(m.index+m[0].length);
      out.push({ type : 'blanks', match: m });
    }
    else if (m = iniheader.exec(input)) {
      input = input.substr(m.index+m[0].length);
      out.push({ type: 'header', match: m });
    }
    else if (m = comments.exec(input)) {
      input = input.substr(m.index+m[0].length);
      out.push({ type: 'comments', match: m });
    }
    else if (m = nameEqualValue.exec(input)) {
      input = input.substr(m.index+m[0].length);
      out.push({ type: 'nameEqualValue', match: m });
    }
    else if (m = any.exec(input)) {
      out.push({ type: 'error', match: m });
      input = '';
    }
    else {
      alert("Fatal Error!"+substr(input,0,20));
      input = '';
    }
  }
  return out;
}
```

Véase la sección *JSON.stringify()* 1.3 para saber mas sobre `JSON.stringify`.

Dudas sobre la Sintáxis del Formato `INI` La sintáxis de `INI` no está bien definida. Se aceptan decisiones razonables para cada una de las expresiones regulares. Si quiere ver un parser en acción puede instalar la gema `inifile` (Ruby).

Una opción que no hemos contemplado en nuestro código es la posibilidad de hacer que una línea de asignación se expanda en varias líneas. En `inifile` el carácter `\` indica que la línea continúa en la siguiente:

```
[~/javascript/PLgrado/inifile(master)]$ cat test/data/good.ini
[section_one]
one = 1
two = 2

[section_two]
three =         3
multi = multiline \
support

; comments should be ignored
[section three]
four   =4
five=5
six =6

[section_four]
   [section_five]
 seven and eight= 7 & 8

[~/javascript/PLgrado/inifile(master)]$ pry
[2] pry(main)> require 'inifile'
=> true

[3] pry(main)> p = IniFile.new(:filename => 'test/data/good.ini')
=> #<IniFile:0x007fba2f41a500
 @_line=" seven and eight= 7 & 8",
 @_section={"seven and eight"=>"7 & 8"},
 @comment=";#",
 @content=
  "[section_one]\none = 1\ntwo = 2\n\n[section_two]\nthree =         3\nmulti = multiline \\\n
 @default="global",
 @encoding=nil,
 @escape=true,
 @filename="test/data/good.ini",
 @ini=
  {"section_one"=>{"one"=>"1", "two"=>"2"},
   "section_two"=>{"three"=>"3", "multi"=>"multiline support"},
   "section three"=>{"four"=>"4", "five"=>"5", "six"=>"6"},
   "section_four"=>{},
   "section_five"=>{"seven and eight"=>"7 & 8"}},
 @param="=">

[4] pry(main)> p["section_two"]
=> {"three"=>"3", "multi"=>"multiline support"}
[5] pry(main)> p[:section_two]
```

Tareas

Es conveniente que consiga estos objetivos:

- Pueden comenzar haciendo un fork del repositorio https://github.com/crguezl/pl-grado-ini-files.

- La entrada debería poder leerse desde un fichero. Añada drag and drop.
- Use Web Storage igual que en la anterior
- Escriba las pruebas
- Use templates externos `underscore` para estructurar la salida
- Añada soporte para multilíneas en las asignaciones (Véase la sección 1.9)

```
> s = 'a=b\\\nc'
'a=b\\\nc'
> n2 = /^([^=;#\r\n]+)=((?:[^;#\r\n]*\\\n)*[^;#\r\n]*)/
/^([^=;#\r\n]+)=((?:[^;#\r\n]*\\\n)*[^;#\r\n]*)/
> m = n2.exec(s)
[ 'a=b\\\nc', 'a', 'b\\\nc',
  index: 0, input: 'a=b\\\nc' ]
> d = m[2]
'b\\\nc'
> d.replace(/\\\n/g,' ')
'b c'
```

Véase

1. JSON.stringify
2. www.json.org
3. JSON in JavaScript
4. Underscore: template
5. Stackoverflow::how to use Underscore template

1.10. Práctica: Analizador Léxico para Un Subconjunto de JavaScript

TDOP, Top Down Operator Precedence Vamos a trabajar a partir de este repo de Douglas Crockford:

- https://github.com/douglascrockford/TDOP
- Autor: Douglas Crockford, douglas@crockford.com
- Fecha que figura en el repo: 2010-11-12
- Descripción:
  - tdop.html contains a description of Vaughn Pratt's Top Down Operator Precedence, and describes the parser whose lexer we are going to write in this lab. Is a simplified version of JavaScript.
  - The file `index.html` parses `parse.js` and displays its AST.
  - The page depends on on `parse.js` and `tokens.js`.
  - The file `tdop.js` contains the Simplified JavaScript parser.
  - tokens.js. produces an array of token objects from a string. This is the file we are going to work in this lab.

Objetivos de la Práctica

Douglas Crockford escribió su analizador léxico tokens.js sin usar expresiones regulares. Eso hace que sea extenso (268 líneas). Su analizador es un subconjunto de JS que no tiene - entre otras cosas - expresiones regulares ya que uno de sus objetivos era que el analizador se analizara a si mismo.

Reescriba el analizador léxico en tokens.js. usando expresiones regulares.

1. Evite que se hagan copias de la cadena siendo procesada. Muévase dentro de la misma cadena usando `lastIndex`

2. Añada botones/enlaces/menu de selección que permitan cargar un fichero específico de una lista de ficheros en la `texarea` de entrada.

   Vea el ejemplo en https://github.com/crguezl/loadfileontotexarea.

   En este caso en vez de un fichero `index.html` arrancamos desde un programa Ruby `app.rb`. Para verlo en ejecución instale primero las dependencias:

   ```
   [~/javascript/jquery/loadfileontotexarea(master)]$ bundle install
   Using daemons (1.1.9)
   Using eventmachine (1.0.3)
   Using rack (1.5.2)
   Using rack-protection (1.5.2)
   Using tilt (1.4.1)
   Using sinatra (1.4.4)
   Using thin (1.6.1)
   Using bundler (1.3.5)
   Your bundle is complete!
   Use ‘bundle show [gemname]‘ to see where a bundled gem is installed.
   ```

   Para ejecutar puede llamar a la aplicación así:

   ```
   [~/javascript/jquery/loadfileontotexarea(master)]$ bundle exec rackup
   Thin web server (v1.6.1 codename Death Proof)
   Maximum connections set to 1024
   Listening on 0.0.0.0:9292, CTRL+C to stop
   ```

   Ahora visite en su navegador la URL `http://localhost:9292`.

   Puede ver también la aplicación corriendo en los servidores de Heroku en http://pllexer.herokuapp.com/. Visite los enlaces `withajax.html` y `withget.html`.

3. Añada pruebas

4. Haga el despliegue de su aplicación en Heroku. Para ver como hacerlo siga las indicaciones en la sección *Heroku* **??** en estos apuntes

5. Una primera solución de la que puede partir se encuentra en: https://github.com/crguezl/ull-etsii-grado-pl-mi en github. Veala en funcionamiento en GitHub Pages

6. El método `tokens` retorna el array de tokens. Puede encontrarlo en tokens.js.

7. Mejore la solución en https://github.com/crguezl/ull-etsii-grado-pl-minijavascript/tree/gh-pages

8. Para esta práctica es necesario familiarizarse con la forma en que funciona la OOP en JS. Vea este jsfiddle

Capítulo 2

Analizadores Descendentes Predictivos en JavaScript

2.1. Conceptos Básicos para el Análisis Sintáctico

Suponemos que el lector de esta sección ha realizado con éxito un curso en teoría de autómatas y lenguajes formales. Las siguientes definiciones repasan los conceptos mas importantes.

Definición 2.1.1. *Dado un conjunto A, se define A^* el cierre de Kleene de A como: $A^* = \cup_{n=0}^{\infty} A^n$*

Se admite que $A^0 = \{\epsilon\}$, donde ϵ denota la palabra vacía, esto es la palabra que tiene longitud cero, formada por cero símbolos del conjunto base A.

Definición 2.1.2. *Una gramática G es una cuaterna $G = (\Sigma, V, P, S)$. Σ es el conjunto de terminales. V es un conjunto (disjunto de Σ) que se denomina conjunto de* variables sintácticas *o* categorías gramáticales*, P es un conjunto de pares de $V \times (V \cup \Sigma)^*$. En vez de escribir un par usando la notación $(A, \alpha) \in P$ se escribe $A \to \alpha$. Un elemento de P se denomina producción. Por último, S es un símbolo del conjunto V que se denomina símbolo de arranque.*

Definición 2.1.3. *Dada una gramática $G = (\Sigma, V, P, S)$ y $\mu = \alpha A \beta \in (V \cup \Sigma)^*$ una frase formada por variables y terminales y $A \to \gamma$ una producción de P, decimos que μ deriva en un paso en $\alpha\gamma\beta$. Esto es, derivar una cadena $\alpha A \beta$ es sustituir una variable sintáctica A de V por la parte derecha γ de una de sus reglas de producción. Se dice que μ deriva en n pasos en δ si deriva en $n-1$ pasos en una cadena $\alpha A \beta$ la cual deriva en un paso en δ. Se escribe entonces que $\mu \stackrel{*}{\Longrightarrow} \delta$. Una cadena deriva en 0 pasos en si misma.*

Definición 2.1.4. *Dada una gramática $G = (\Sigma, V, P, S)$ se denota por $L(G)$ o lenguaje generado por G al lenguaje:*

$$L(G) = \{x \in \Sigma^* : S \stackrel{*}{\Longrightarrow} x\}$$

Esto es, el lenguaje generado por la gramática G esta formado por las cadenas de terminales que pueden ser derivados desde el símbolo de arranque.

Definición 2.1.5. *Una derivación que comienza en el símbolo de arranque y termina en una secuencia formada por sólo terminales de Σ se dice* completa*.*

Una derivación $\mu \stackrel{*}{\Longrightarrow} \delta$ en la cual en cada paso $\alpha A x$ la regla de producción aplicada $A \to \gamma$ se aplica en la variable sintáctica mas a la derecha se dice una derivación a derechas

Una derivación $\mu \stackrel{*}{\Longrightarrow} \delta$ en la cual en cada paso $xA\alpha$ la regla de producción aplicada $A \to \gamma$ se aplica en la variable sintáctica mas a la izquierda se dice una derivación a izquierdas

Definición 2.1.6. *Observe que una derivación puede ser representada como un árbol cuyos nodos están etiquetados en $V \cup \Sigma$. La aplicación de la regla de producción $A \to \gamma$ se traduce en asignar como hijos del nodo etiquetado con A a los nodos etiquetados con los símbolos $X_1 \ldots X_n$ que constituyen la frase $\gamma = X_1 \ldots X_n$. Este árbol se llama* árbol sintáctico concreto *asociado con la derivación.*

Definición 2.1.7. *Observe que, dada una frase $x \in L(G)$ una derivación desde el símbolo de arranque da lugar a un árbol. Ese árbol tiene como raíz el símbolo de arranque y como hojas los terminales $x_1 \dots x_n$ que forman x. Dicho árbol se denomina* árbol de análisis sintáctico concreto *de x. Una derivación determina una forma de recorrido del árbol de análisis sintáctico concreto.*

Definición 2.1.8. *Una gramática G se dice ambigua si existe alguna frase $x \in L(G)$ con al menos dos árboles sintácticos. Es claro que esta definición es equivalente a afirmar que existe alguna frase $x \in L(G)$ para la cual existen dos derivaciones a izquierda (derecha) distintas.*

2.1.1. Ejercicio

Dada la gramática con producciones:

program → declarations statements | statements
declarations → declaration ';' declarations | declaration ';'
declaration → INT idlist | STRING idlist
statements → statement ';' statements | statement
statement → ID '=' expression | P expression
expression → term '+' expression | term
term → factor '*' term | factor
factor → '(' expression ')' | ID | NUM | STR
idlist → ID ',' idlist | ID

En esta gramática, Σ esta formado por los caracteres entre comillas simples y los símbolos cuyos identificadores están en mayúsculas. Los restantes identificadores corresponden a elementos de V. El símbolo de arranque es $S =$ program.

Conteste a las siguientes cuestiones:

1. Describa con palabras el lenguaje generado.
2. Construya el árbol de análisis sintáctico concreto para cuatro frases del lenguaje.
3. Señale a que recorridos del árbol corresponden las respectivas derivaciones a izquierda y a derecha en el apartado 2.
4. ¿Es ambigua esta gramática?. Justifique su respuesta.

2.2. Análisis Sintáctico Predictivo Recursivo

La siguiente fase en la construcción del analizador es la fase de análisis sintáctico. Esta toma como entrada el flujo de terminales y construye como salida el árbol de análisis sintáctico abstracto.

El árbol de análisis sintáctico abstracto es una representación compactada del árbol de análisis sintáctico concreto que contiene la misma información que éste.

Existen diferentes métodos de análisis sintáctico. La mayoría caen en una de dos categorías: ascendentes y descendentes. Los ascendentes construyen el árbol desde las hojas hacia la raíz. Los descendentes lo hacen en modo inverso. El que describiremos aqui es uno de los mas sencillos: se denomina método de análisis predictivo descendente recursivo.

2.2.1. Introducción

En este método se asocia una subrutina con cada variable sintáctica $A \in V$. Dicha subrutina (que llamaremos `A`) reconocerá el lenguaje generado desde la variable A:

$$L_A(G) = \{x \in \Sigma^* : A \stackrel{*}{\Longrightarrow} x\}$$

statements → statement ';' statements | statement
statement → ID '=' expression | P expression
expression → term '+' expression | term
term → factor '*' term | factor
factor → '(' expression ')' | ID | NUM

Cuadro 2.1: Una Gramática Simple

En este método se escribe una rutina `A` por variable sintáctica $A \in V$. Se le da a la rutina asociada el mismo nombre que a la variable sintáctica asociada.

La función de la rutina `A` asociada con la variable $A \in V$ es reconocer el lenguaje $L(A)$ generado por A.

La estrategia general que sigue la rutina `A` para reconocer $L(A)$ es decidir en términos del terminal a en la entrada que regla de producción concreta $A \to \alpha$ se aplica para a continuación comprobar que la entrada que sigue pertenece al lenguaje generado por α.

En un analizador predictivo descendente recursivo (APDR) se asume que el símbolo que actualmente esta siendo observado (denotado habitualmente como `lookahead`) permite determinar unívocamente que producción de A hay que aplicar.

Una vez que se ha determinado que la regla por la que continuar la derivación es $A \to \alpha$ se procede a reconocer $L_\alpha(G)$, el lenguaje generado por α. Si $\alpha = X_1 \dots X_n$, las apariciones de terminales X_i en α son emparejadas con los terminales en la entrada mientras que las apariciones de variables $X_i = B$ en α se traducen en llamadas a la correspondiente subrutina asociada con `B`.

Ejemplo

Para ilustrar el método, simplificaremos la gramática presentada en el ejercicio 5.1.1 eliminando las declaraciones:

La secuencia de llamadas cuando se procesa la entrada mediante el siguiente programa construye implícitamente el *árbol de análisis sintáctico concreto*.

```
parse = (input) ->
  tokens = input.tokens()
  lookahead = tokens.shift()
  match = (t) ->
    if lookahead.type is t
      lookahead = tokens.shift()
      lookahead = null  if typeof lookahead is "undefined"
    else # Error. Throw exception
      throw "Syntax Error. Expected #{t} found '" +
            lookahead.value + "' near '" +
            input.substr(lookahead.from) + "'"
    return

  statements = ->
    result = [statement()]
    while lookahead and lookahead.type is ";"
      match ";"
      result.push statement()
    (if result.length is 1 then result[0] else result)

  statement = ->
    result = null
    if lookahead and lookahead.type is "ID"
```

```
    left =
      type: "ID"
      value: lookahead.value

    match "ID"
    match "="
    right = expression()
    result =
      type: "="
      left: left
      right: right
  else if lookahead and lookahead.type is "P"
    match "P"
    right = expression()
    result =
      type: "P"
      value: right
  else # Error!
    throw "Syntax Error. Expected identifier but found " +
      (if lookahead then lookahead.value else "end of input") +
      " near '#{input.substr(lookahead.from)}'"
  result

expression = ->
  result = term()
  if lookahead and lookahead.type is "+"
    match "+"
    right = expression()
    result =
      type: "+"
      left: result
      right: right
  result

term = ->
  result = factor()
  if lookahead and lookahead.type is "*"
    match "*"
    right = term()
    result =
      type: "*"
      left: result
      right: right
  result

factor = ->
  result = null
  if lookahead.type is "NUM"
    result =
      type: "NUM"
      value: lookahead.value

    match "NUM"
```

```
  else if lookahead.type is "ID"
    result =
      type: "ID"
      value: lookahead.value

    match "ID"
  else if lookahead.type is "("
    match "("
    result = expression()
    match ")"
  else # Throw exception
    throw "Syntax Error. Expected number or identifier or '(' but found " +
      (if lookahead then lookahead.value else "end of input") +
      " near '" + input.substr(lookahead.from) + "'"
  result

tree = statements(input)
if lookahead?
  throw "Syntax Error parsing statements. " +
    "Expected 'end of input' and found '" +
    input.substr(lookahead.from) + "'"
tree
```

```
var parse = function(input) {
  var tokens = input.tokens();
  var lookahead = tokens.shift();

  var match = function(t) {
    if (lookahead.type === t) {
      lookahead = tokens.shift();
      if (typeof lookahead === 'undefined') {
       lookahead = null; // end of input
      }
    } else { // Error. Throw exception
        throw "Syntax Error. Expected "+t+" found '"+lookahead.value+
              "' near '"+input.substr(lookahead.from)+"'";
    }
  };

  var statements = function() {
    var result = [ statement() ];
    while (lookahead && lookahead.type === ';') {
      match(';');
      result.push(statement());
    }
    return result.length === 1? result[0] : result;
  };

  var statement = function() {
    var result = null;

    if (lookahead && lookahead.type === 'ID') {
      var left = { type: 'ID', value: lookahead.value };
      match('ID');
```

```
    match('=');
    right = expression();
    result = { type: '=', left: left, right: right };
  } else if (lookahead && lookahead.type === 'P') {
    match('P');
    right = expression();
    result = { type: 'P', value: right };
  } else { // Error!
    throw "Syntax Error. Expected identifier but found "+
          (lookahead? lookahead.value : "end of input")+
          " near '"+input.substr(lookahead.from)+"'";
  }
  return result;
};

var expression = function() {
  var result = term();
  if (lookahead && lookahead.type === '+') {
    match('+');
    var right = expression();
    result = {type: '+', left: result, right: right};
  }
  return result;
};

var term = function() {
  var result = factor();
  if (lookahead && lookahead.type === '*') {
    match('*');
    var right = term();
    result = {type: '*', left: result, right: right};
  }
  return result;
};

var factor = function() {
  var result = null;

  if (lookahead.type === 'NUM') {
    result = {type: 'NUM', value: lookahead.value};
    match('NUM');
  }
  else if (lookahead.type === 'ID') {
    result = {type: 'ID', value: lookahead.value};
    match('ID');
  }
  else if (lookahead.type === '(') {
    match('(');
    result = expression();
    match(')');
  } else { // Throw exception
    throw "Syntax Error. Expected number or identifier or '(' but found "+
          (lookahead? lookahead.value : "end of input")+
```

```
            " near '"+input.substr(lookahead.from)+"'";
      }
      return result;
    };
    var tree = statements(input);
    if (lookahead != null) {
        throw "Syntax Error parsing statements. Expected end of input and found '"+
              input.substr(lookahead.from)+"'";
    }
    return tree;
  }
```

Caracterización de las Gramáticas Analizables Como vemos en el ejemplo, el análisis predictivo confía en que, si estamos ejecutando la entrada del procedimiento `A`, el cuál está asociado con la variable $A \in V$, el símbolo terminal que esta en la entrada a determine de manera unívoca la regla de producción $A \to a\alpha$ que debe ser procesada.

Si se piensa, esta condición requiere que todas las partes derechas α de las reglas $A \to \alpha$ de A comiencen por diferentes símbolos. Para formalizar esta idea, introduciremos el concepto de conjunto $FIRST(\alpha)$:

Definición 2.2.1. *Dada una gramática* $G = (\Sigma, V, P, S)$ *y un símbolo* $\alpha \in (V \cup \Sigma)^*$ *se define el conjunto* $FIRST(\alpha)$ *como:*

$$FIRST(\alpha) = \left\{ b \in \Sigma : \alpha \overset{*}{\Longrightarrow} b\beta \right\} \cup N(\alpha)$$

donde:

$$N(\alpha) = \begin{cases} \{\epsilon\} & si\ \alpha \overset{*}{\Longrightarrow} \epsilon \\ \emptyset & en\ otro\ caso \end{cases}$$

Podemos reformular ahora nuestra afirmación anterior en estos términos: Si $A \to \gamma_1 \mid \ldots \mid \gamma_n$ y los conjuntos $FIRST(\gamma_i)$ son disjuntos podemos construir el procedimiento para la variable A siguiendo este seudocódigo:

```
A = function() {
  if (lookahead in FIRST(gamma_1)) { imitar gamma_1 }
  else if (lookahead in FIRST(gamma_2)) { imitar gamma_2 }
  ...
  else (lookahead in FIRST(gamma_n)) { imitar gamma_n }
}
```

Donde si γ_j es $X_1 \ldots X_k$ el código `gamma_j` consiste en una secuencia $i = 1 \ldots k$ de llamadas de uno de estos dos tipos:

- Llamar a la subrutina `X_i` si X_i es una variable sintáctica
- Hacer una llamada a `match(X_i)` si X_i es un terminal

2.2.2. Ejercicio: Recorrido del árbol en un ADPR

¿En que forma es recorrido el árbol de análisis sintáctico concreto en un analizador descendente predictivo recursivo? ¿En que orden son visitados los nodos?

2.3. Recursión por la Izquierda

Definición 2.3.1. *Una* gramática es recursiva por la izquierda *cuando existe una derivación* $A \overset{*}{\Longrightarrow} A\alpha$.

En particular, es recursiva por la izquierda si contiene una regla de producción de la forma $A \to A\alpha$. *En este caso se dice que la recursión por la izquierda es directa.*

Cuando la gramática es *recursiva por la izquierda*, el método de análisis recursivo descendente predictivo no funciona. En ese caso, el procedimiento `A` asociado con A ciclaría para siempre sin llegar a consumir ningún terminal.

2.4. Esquemas de Traducción

Definición 2.4.1. *Un* esquema de traducción *es una gramática independiente del contexto en la cual se han insertado fragmentos de código en las partes derechas de sus reglas de producción. Los fragmentos de código asi insertados se denominan* acciones semánticas. *Dichos fragmentos actúan, calculan y modifican los atributos asociados con los nodos del árbol sintáctico. El orden en que se evalúan los fragmentos es el de un recorrido primero-profundo del árbol de análisis sintáctico.*

Obsérvese que, en general, para poder aplicar un esquema de traducción hay que construir el árbol sintáctico y después aplicar las acciones empotradas en las reglas en el orden de recorrido primero-profundo. Por supuesto, si la gramática es ambigua una frase podría tener dos árboles y la ejecución de las acciones para ellos podría dar lugar a diferentes resultados. Si se quiere evitar la multiplicidad de resultados (interpretaciones semánticas) es necesario precisar de que árbol sintáctico concreto se esta hablando.

Por ejemplo, si en la regla $A \to \alpha\beta$ insertamos un fragmento de código:

$$A \to \alpha\{action\}\beta$$

La acción $\{action\}$ se ejecutará después de todas las acciones asociadas con el recorrido del subárbol de α y antes que todas las acciones asociadas con el recorrido del subárbol β.

El siguiente esquema de traducción recibe como entrada una expresión en infijo y produce como salida su traducción a postfijo para expresiones aritmeticas con sólo restas de números:

$expr \to expr_1 - NUM$ `{ expr.TRA = expr[1].TRA+" "+NUM.VAL+" - "}`
$expr \to NUM$ `{ expr.TRA = NUM.VAL }`

Las apariciones de variables sintácticas en una regla de producción se indexan como se ve en el ejemplo, para distinguir de que nodo del árbol de análisis estamos hablando. Cuando hablemos del atributo de un nodo utilizaremos el punto (`.`). Aquí `VAL` es un atributo de los nodos de tipo NUM denotando su valor numérico y para accederlo escribiremos `NUM.VAL`. Análogamente `expr.TRA` denota el atributo traducción de los nodos de tipo *expr*.

Ejercicio 2.4.1. *Muestre la secuencia de acciones a la que da lugar el esquema de traducción anterior para la frase 7 -5 -4.*

En este ejemplo, el cómputo del atributo `expr.TRA` depende de los atributos en los nodos hijos, o lo que es lo mismo, depende de los atributos de los símbolos en la parte derecha de la regla de producción. Esto ocurre a menudo y motiva la siguiente definición:

Definición 2.4.2. *Un atributo tal que su valor en todo nodo del árbol sintáctico puede ser computado en términos de los atributos de los hijos del nodo se dice que es un* atributo sintetizado.

2.5. Eliminación de la Recursión por la Izquierda en un Esquema de Traducción

La eliminación de la recursión por la izquierda es sólo un paso: debe ser extendida a esquemas de traducción, de manera que no sólo se preserve el lenguaje sino la secuencia de acciones. Supongamos que tenemos un esquema de traducción de la forma:

$A \to A\alpha$ `{ alpha_action }`
$A \to A\beta$ `{ beta_action }`
$A \to \gamma$ `{ gamma_action }`

para una sentencia como $\gamma\beta\alpha$ la secuencia de acciones será:

```
gamma_action  beta_action alpha_action
```

¿Cómo construir un esquema de traducción para la gramática resultante de eliminar la recursión por la izquierda que ejecute las acciones asociadas en el mismo orden?. Supongamos para simplificar, que las acciones no dependen de atributos ni computan atributos, sino que actúan sobre variables globales. En tal caso, la siguiente ubicación de las acciones da lugar a que se ejecuten en el mismo orden:

$A \rightarrow \gamma$ `{ gamma_action }` R
$R \rightarrow \beta$ `{ beta_action }` R
$R \rightarrow \alpha$ `{ alpha_action }` R
$R \rightarrow \epsilon$

Si hay atributos en juego, la estrategia para construir un esquema de traducción equivalente para la gramática resultante de eliminar la recursividad por la izquierda se complica.

2.6. Práctica: Analizador Descendente Predictivo Recursivo

Partiendo del analizador sintáctico descendente predictivo recursivo para la gramática descrita en la sección 2.2.1

Donde Puede encontrar la versión de la que partir en

- Despliegue en Heroku: http://predictiveparser.herokuapp.com/
- Repositorio en GitHub: https://github.com/crguezl/prdcalc
- ```
  [~/javascript/PLgrado/predictiveRD/prdcalc(develop)]$ pwd -P
  /Users/casiano/local/src/javascript/PLgrado/predictiveRD/prdcalc
  [~/javascript/PLgrado/predictiveRD/prdcalc(develop)]$ git remote -v
  heroku  git@heroku.com:predictiveparser.git (fetch)
  heroku  git@heroku.com:predictiveparser.git (push)
  origin  git@github.com:crguezl/prdcalc.git (fetch)
  origin  git@github.com:crguezl/prdcalc.git (push)
  ```

Tareas Añada:

- Extienda y modifique el analizador para que acepte el lenguaje descrito por la gramática EBNF del lenguaje PL/0 que se describe en la entrada de la Wikipedia Recursive descent parser. Procure que el arbol generado refleje la asociatividad correcta para las diferencias y las divisiones. No es necesario que el lenguaje sea exactamente igual pero debería ser parecido. Tener los mismos constructos.
-
- Use CoffeeScript para escribir el código (fichero `views/main.coffee`)
- Use slim para las vistas
- Usa Sass para las hojas de estilo
- Despliegue la aplicación en Heroku
- Añada pruebas
  ```

Sinatra
Véase el fichero main.rb.

1. Filters
2. Helpers
   a) El helper `css` es usado en views/layout.slim
   b) El helper `current?` es usado en views/nav.slim para añadir la clase `current` a la página que esta siendo visitada.
   c) El estilo de la entrada de la página actual es modificado en el fichero de estilo views/styles.scss

   ```
   nav a.current {
     background: lighten($black, 50%);
   }
   ```

   El método lighten es proveído por Sass .
3. Views / Templates

Sass
Véase el fichero views/styles.scss.

1. Sass
2. Sass Basics
3. css2sass en GitHub (https://github.com/jpablobr/css2sass) y despliegue en Heroku (http://css2sass.heroku.c

Slim Véanse los ficheros `views/*.slim`:

- views/layout.slim
- views/home.slim
- views/nav.slim

1. slim
2. Slim docs
3. html2slim
4. 2011.12.16 Tech Talk: Slim Templates de Big Nerd Ranch (Vimeo)

CoffeeScript

1. CoffeeScript
2. CoffeeScript Cookbook
3. js2coffee.org

Construyendo Árboles con la Asociatividad Correcta Añadamos el operador `-` al código de nuestra práctica. Para ello, podemos extender nuestro gramática con una regla de producción:

$$\mid \text{expression} \rightarrow \text{term '+' expression} \mid \text{term '-' expression} \mid \text{term} \mid$$

que da lugar a un código como el que sigue:

```
expression = ->
  result = term()
  if lookahead and lookahead.type is "+"
    ....
  if lookahead and lookahead.type is "-"
    match "-"
    right = expression()
    result =
      type: "-"
      left: result
      right: right
```

Cuando le damos como entrada `a = 4-2-1` produce el siguiente AST:

```
{
  "type": "=",
  "left": {
    "type": "ID",
    "value": "a"
  },
  "right": {
    "type": "-",
    "left": {
      "type": "NUM",
      "value": 4
    },
    "right": {
      "type": "-",
      "left": {
        "type": "NUM",
        "value": 2
      },
      "right": {
        "type": "NUM",
        "value": 1
      }
    }
  }
}
```

que se corresponde con esta parentización: `a = (4 - (2 - 1))`

Este árbol no se corresponde con la asociatividad a izquierdas del operador `-`. Es un árbol que refleja una asociación a derechas (`a = 3`).

Ahora bien, el lenguaje generado por dos reglas de la forma: $A \rightarrow A\alpha$ `{ alpha_action }` / $A \rightarrow \gamma$ `{ gamma_action }` Es $\gamma\alpha*$. Por tanto el método asociado con A podría reescribirse como sigue:

```
A = () ->
  gamma() # imitar gamma
  gamma_action() # acción semántica asociada con gamma
  while lookahead and lookahead.type belongs to  FIRST(alpha)
    alpha() # imitar alpha
    alpha_action()
```

Capítulo 3

Análisis Sintáctico Mediante Precedencia de Operadores en JavaScript

3.1. Ejemplo Simple de Intérprete: Una Calculadora

1. How to write a simple interpreter in JavaScript

3.2. Análisis Top Down Usando Precedencia de Operadores

1. Véase el libro [2] Beautiful Code: Leading Programmers Explain How They Think, Capítulo 9.
2. Top Down Operator Precedence por Douglas Crockford
3. Top Down Operator Precedence demo por Douglas Crockford
4. jslint
5. David Majda - Easy parsing with PEG.js

3.2.1. Gramática de JavaScript

1. Especificación de JavaScript 1997
2. NQLL(1) grammar (Not Quite LL(1)) for JavaScrip 1997
3. Postscript con la especificación de JavaScript 1997
4. Mozilla JavaScript Language Resources
5. JavaScript 1.4 LR(1) Grammar 1999.
6. Apple JavaScript Core Specifications
7. Creating a JavaScript Parser Una implementación de ECAMScript 5.1 usando Jison disponible en GitHub en https://github.com/cjihrig/jsparser.

Capítulo 4

Análisis Descendente mediante Parsing Expresion Grammars en JavaScript

4.1. Introducción a los PEGs

In computer science, a *parsing expression grammar*, or *PEG*, is a type of analytic formal grammar, i.e. it describes a formal language in terms of a set of rules for recognizing strings in the language.

The formalism was introduced by Bryan Ford in 2004 and is closely related to the family of top-down parsing languages introduced in the early 1970s.

Syntactically, PEGs also look similar to context-free grammars (CFGs), but they have a different interpretation:

- the choice operator selects the first match in PEG, while it is ambiguous in CFG.
- This is closer to how string recognition tends to be done in practice, e.g. by a recursive descent parser.

Unlike CFGs, PEGs cannot be ambiguous; *if a string parses, it has exactly one valid parse tree.*

It is conjectured that there exist context-free languages that cannot be parsed by a PEG, but this is not yet proven.

4.1.1. Syntax

Formally, a parsing expression grammar consists of:

- A finite set N of nonterminal symbols.
- A finite set Σ of terminal symbols that is disjoint from N.
- A finite set P of parsing rules.
- An expression e_S termed the starting expression.

Each parsing rule in P has the form $A \leftarrow e$, where A is a nonterminal symbol and e is a *parsing expression.*

A parsing expression is a hierarchical expression similar to a regular expression, which is constructed in the following fashion:

1. An atomic parsing expression consists of:
   a) any terminal symbol,
   b) any nonterminal symbol, or
   c) the empty string ϵ.

2. Given any existing parsing expressions e, e_1, and e_2, a new parsing expression can be constructed using the following operators:

   a) Sequence: $e1e2$

   b) Ordered choice: $e1/e2$

   c) Zero-or-more: $e*$

   d) One-or-more: $e+$

   e) Optional: $e?$

   f) And-predicate: $\&e$

   g) Not-predicate: $!e$

4.1.2. Semantics

The fundamental difference between context-free grammars and parsing expression grammars is that the PEG's choice operator is ordered:

1. If the first alternative succeeds, the second alternative is ignored.
2. Thus ordered choice is not commutative, unlike unordered choice as in context-free grammars.
3. The consequence is that if a CFG is transliterated directly to a PEG, any ambiguity in the former is resolved by deterministically picking one parse tree from the possible parses.
4. By carefully choosing the order in which the grammar alternatives are specified, a programmer has a great deal of control over which parse tree is selected.
5. PEGs can look ahead into the input string without actually consuming it
6. The and-predicate expression $\&e$ invokes the sub-expression e, and then succeeds if e succeeds and fails if e fails, *but in either case never consumes any input.*
7. The not-predicate expression $!e$ succeeds if e fails and fails if e succeeds, *again consuming no input in either case.*

4.1.3. Implementing parsers from parsing expression grammars

Any parsing expression grammar can be converted directly into a *recursive descent parser.*

Due to the unlimited lookahead capability that the grammar formalism provides, however, the resulting parser could exhibit exponential time performance in the worst case.

It is possible to obtain better performance for any parsing expression grammar by converting its recursive descent parser into a packrat parser, which always runs in linear time, at the cost of substantially greater storage space requirements.

A packrat parser is a form of parser similar to a recursive descent parser in construction, except that during the parsing process it memoizes the intermediate results of all invocations of the mutually recursive parsing functions, ensuring that each parsing function is only invoked at most once at a given input position.

Because of this memoization, a packrat parser has the ability to parse many context-free grammars and any parsing expression grammar (including some that do not represent context-free languages) in linear time.

Examples of memoized recursive descent parsers are known from at least as early as 1993.

Note that this analysis of the performance of a packrat parser **assumes that enough memory is available to hold all of the memoized results**; in practice, if there were not enough memory, some parsing functions might have to be invoked more than once at the same input position, and consequently the parser could take more than linear time.

It is also possible to build LL parsers and LR parsers from parsing expression grammars, with better worst-case performance than a recursive descent parser, but the unlimited lookahead capability of the grammar formalism is then lost. Therefore, not all languages that can be expressed using parsing expression grammars can be parsed by LL or LR parsers.

4.1.4. Lexical Analysis

Parsers for languages expressed as a CFG, such as LR parsers, require a separate tokenization step to be done first, which breaks up the input based on the location of spaces, punctuation, etc.

The tokenization is necessary because of the way these parsers use lookahead to parse CFGs that meet certain requirements in linear time.

PEGs do not require tokenization to be a separate step, and tokenization rules can be written in the same way as any other grammar rule.

4.1.5. Left recursion

PEGs cannot express left-recursive rules where a rule refers to itself without moving forward in the string. For example, the following left-recursive CFG rule:

```
string-of-a -> string-of-a 'a' | 'a'
```

can be rewritten in a PEG using the plus operator:

```
string-of-a <- 'a'+
```

The process of rewriting indirectly left-recursive rules is complex in some packrat parsers, especially when semantic actions are involved.

4.1.6. Referencias y Documentación

- Véase Parsing Expression Grammar
- PEG.js documentation
- Testing PEG.js Online
- Michael's Blog: JavaScript Parser Generators. The PEG.js Tutorial
- The Packrat Parsing and Parsing Expression Grammars Page
- PL101: Create Your Own Programming Language. Véanse [3] y [4]
- PL101: Create Your Own Programming Language: Parsing

4.2. PEGJS

What is

PEG.js is a parser generator for JavaScript that produces parsers.

PEG.js generates a parser from a Parsing Expression Grammar describing a language.

We can specify what the parser returns (using semantic actions on matched parts of the input).

Installation

To use the pegjs command, install PEG.js globally:

```
$ npm install -g pegjs
```

To use the JavaScript API, install PEG.js locally:

```
$ npm install pegjs
```

To use it from the browser, download the PEG.js library (regular or minified version).

El compilador de línea de comandos

```
[~/srcPLgrado/pegjs/examples(master)]$ pegjs --help
Usage: pegjs [options] [--] [<input_file>] [<output_file>]

Generates a parser from the PEG grammar specified in the <input_file> and writes
it to the <output_file>.

If the <output_file> is omitted, its name is generated by changing the
<input_file> extension to ".js". If both <input_file> and <output_file> are
omitted, standard input and output are used.

Options:
  -e, --export-var <variable>        name of the variable where the parser
                                     object will be stored (default:
                                     "module.exports")
      --cache                        make generated parser cache results
      --allowed-start-rules <rules>  comma-separated list of rules the generated
                                     parser will be allowed to start parsing
                                     from (default: the first rule in the
                                     grammar)
  -o, --optimize <goal>              select optimization for speed or size
                                     (default: speed)
      --plugin <plugin>              use a specified plugin (can be specified
                                     multiple times)
      --extra-options <options>      additional options (in JSON format) to pass
                                     to PEG.buildParser
      --extra-options-file <file>    file with additional options (in JSON
                                     format) to pass to PEG.buildParser
  -v, --version                      print version information and exit
  -h, --help                         print help and exit
```

Using it

```
[~/srcPLgrado/pegjs/examples(master)]$ node
> PEG = require("pegjs")
{ VERSION: '0.8.0',
  GrammarError: [Function],
  parser:
   { SyntaxError: [Function: SyntaxError],
     parse: [Function: parse] },
  compiler:
   { passes:
      { check: [Object],
        transform: [Object],
        generate: [Object] },
     compile: [Function] },
  buildParser: [Function] }

> parser = PEG.buildParser("start = ('a' / 'b')+")
{ SyntaxError: [Function: SyntaxError],
  parse: [Function: parse] }
```

Using the generated parser is simple — just call its `parse` method and pass an input string as a parameter.

The method will return

- a parse result or
- throw an exception if the input is invalid.

You can tweak parser behavior by passing a second parameter with an `options` object to the `parse` method.

Only one option is currently supported:
`startRule` which is the name of the rule to start parsing from.

```
> parser.parse("abba");
[ 'a', 'b', 'b', 'a' ]
>
```

Opciones: allowedStartRules Specifying allowedStartRules we can set the rules the parser will be allowed to start parsing from (default: the first rule in the grammar).

```
[~/srcPLgrado/pegjs/examples(master)]$ cat allowedstartrules.js
var PEG = require("pegjs");
var grammar = "a = 'hello' b\nb = 'world'"; //"a = 'hello' b\nb='world';
console.log(grammar);

var parser = PEG.buildParser(grammar,{ allowedStartRules: ['a', 'b'] });
var r = parser.parse("helloworld", { startRule: 'a' });
console.log(r); // [ 'hello', 'world' ]
r = parser.parse("helloworld")
console.log(r); // [ 'hello', 'world' ]

r =  parser.parse("world", { startRule: 'b' })
console.log(r); // 'world'

try {
  r = parser.parse("world"); // Throws an exception
}
catch(e) {
  console.log("Error!!!!");
  console.log(e);
}

[~/srcPLgrado/pegjs/examples(master)]$ node allowedstartrules.js
a = 'hello' b
b = 'world'
[ 'hello', 'world' ]
[ 'hello', 'world' ]
world
Error!!!!
{ message: 'Expected "hello" but "w" found.',
  expected: [ { type: 'literal', value: 'hello', description: '"hello"' } ],
  found: 'w',
  offset: 0,
  line: 1,
  column: 1,
  name: 'SyntaxError' }
```

The exception contains

- message
- expected,
- found
- offset,
- line,
- column,
- name

and properties with more details about the error.

Opciones: output

When `output` is set to parser, the method will return generated parser object; if set to source, it will return parser source code as a string (default: `parser`).

```
> PEG = require("pegjs")
> grammar = "a = 'hello' b\nb='world'"
'a = \'hello\' b\nb=\'world\''
> console.log(grammar)
a = 'hello' b
b='world'
undefined
> parser = PEG.buildParser(grammar,{ output: "parse"})
undefined
> parser = PEG.buildParser(grammar,{ output: "source"})
> typeof parser
'string'
> console.log(parser.substring(0,100))
(function() {
  /*
   * Generated by PEG.js 0.8.0.
   *
   * http://pegjs.majda.cz/
   */
```

Opciones: plugin La opción plugins indica que plugin se van a usar.

```
[~/srcPLgrado/pegjs/examples(master)]$ cat plugin.coffee
#!/usr/bin/env coffee
PEG = require 'pegjs'
coffee = require 'pegjs-coffee-plugin'
grammar = """
a = 'hello' _ b { console.log 1 }
b = 'world'     { console.log 2 }
_ = [ \t]+      { console.log 3 }
"""
parser = PEG.buildParser grammar, plugins: [coffee]
r = parser.parse "hello world"
```

```
[~/srcPLgrado/pegjs/examples(master)]$ ./plugin.coffee
3
2
1
```

cache

If `true`, makes the parser cache results, avoiding exponential parsing time in pathological cases but making the parser slower (default: `false`).

optimize

Selects between optimizing the generated parser for parsing speed (`speed`) or code size (`size`) (default: `speed`).

4.3. Un Ejemplo Sencillo

Donde

```
[~/srcPLgrado/pegjs/examples(master)]$ pwd -P
/Users/casiano/local/src/javascript/PLgrado/pegjs/examples
[~/srcPLgrado/pegjs/examples(master)]$ git remote -v
dmajda  https://github.com/dmajda/pegjs.git (fetch)
dmajda  https://github.com/dmajda/pegjs.git (push)
origin  git@github.com:crguezl/pegjs.git (fetch)
origin  git@github.com:crguezl/pegjs.git (push)
```

https://github.com/crguezl/pegjs/blob/master/examples/arithmetics.pegjs

arithmetics.pegjs

```
[~/Dropbox/src/javascript/PLgrado/pegjs/examples(master)]$ cat arithmetics.pegjs
/*
 * Classic example grammar, which recognizes simple arithmetic expressions like
 * "2*(3+4)". The parser generated from this grammar then computes their value.
 */

start
  = additive

additive
  = left:multiplicative PLUS right:additive { return left + right; }
  / left:multiplicative MINUS right:additive { return left - right; }
  / multiplicative

multiplicative
  = left:primary MULT right:multiplicative { return left * right; }
  / left:primary DIV right:multiplicative { return left / right; }
  / primary

primary
  = integer
  / LEFTPAR additive:additive RIGHTPAR { return additive; }

integer "integer"
  = NUMBER

_ = $[ \t\n\r]*

PLUS = _"+"_
MINUS = _"-"_
```

```
MULT = _"*"_
DIV = _"/"_
LEFTPAR = _"("_
RIGHTPAR = _")"_
NUMBER = _ digits:$[0-9]+ _ { return parseInt(digits, 10); }
```

There are several types of parsing expressions, some of them containing subexpressions and thus forming a recursive structure:

- `expression *`

  Match zero or more repetitions of the expression and return their match results in an array. The matching is greedy, i.e. the parser tries to match the expression as many times as possible.

- `expression +`

  Match one or more repetitions of the expression and return their match results in an array. The matching is greedy, i.e. the parser tries to match the expression as many times as possible.

- `$ expression`

  Try to match the expression. If the match succeeds, return the matched string instead of the match result.

main.js

```
[~/Dropbox/src/javascript/PLgrado/pegjs/examples(master)]$ cat main.js
var PEG = require("./arithmetics.js");
var r = PEG.parse("(2+9-1)/2");
console.log(r);
```

Rakefile

```
[~/Dropbox/src/javascript/PLgrado/pegjs/examples(master)]$ cat Rakefile
PEGJS = "../bin/pegjs"
task :default => :run

desc "Compile arithmetics.pegjs"
task :compile do
  sh "#{PEGJS} arithmetics.pegjs"
end

desc "Run and use the parser generated from arithmetics.pegjs"
task :run => :compile do
  sh "node main.js"
end
```

Compilación

```
[~/Dropbox/src/javascript/PLgrado/pegjs/examples(master)]$ rake
../bin/pegjs arithmetics.pegjs
node main.js
5
```

4.3.1. Asociación Incorrecta para la Resta y la División

Definición 4.3.1. *Una* gramática es recursiva por la izquierda *cuando existe una derivación* $A \stackrel{*}{\Longrightarrow} A\alpha$.

En particular, es recursiva por la izquierda si contiene una regla de producción de la forma $A \to A\alpha$. *En este caso se dice que la recursión por la izquierda es directa.*

Cuando la gramática es *recursiva por la izquierda*, el método de análisis recursivo descendente predictivo no funciona. En ese caso, el procedimiento `A` asociado con A ciclaría para siempre sin llegar a consumir ningún terminal.

Es por eso que hemos escrito las reglas de la calculadora con recursividad a derechas,

```
additive
  = left:multiplicative PLUS right:additive { return left + right; }
  / left:multiplicative MINUS right:additive { return left - right; }
  / multiplicative


multiplicative
  = left:primary MULT right:multiplicative { return left * right; }
  / left:primary DIV right:multiplicative { return left / right; }
  / primary
```

pero eso da lugar a árboles hundidos hacia la derecha y a una aplicación de las reglas semánticas errónea:

```
[~/pegjs/examples(master)]$ cat main.js
var PEG = require("./arithmetics.js");
var r = PEG.parse("5-3-2");
console.log(r);

[~/pegjs/examples(master)]$ node main.js
4
```

4.4. Acciones Intermedias

Supongamos que queremos poner una acción semántica intermedia en un programa PEG.js :

```
[~/srcPLgrado/pegjs/examples(master)]$ cat direct_intermedia.pegjs
a = 'a'+ { console.log("acción intermedia"); } 'b'+ {
                      console.log("acción final");
                      return "hello world!";
                    }
```

Al compilar nos da un mensjae de error:

```
[~/srcPLgrado/pegjs/examples(master)]$ pegjs direct_intermedia.pegjs
1:48: Expected "/", ";", end of input or identifier but "'" found.
```

La solución consiste en introducir una variable sintáctica en medio que derive a la palabra vacía y que tenga asociada la correspondiente acción semántica:

```
[~/srcPLgrado/pegjs/examples(master)]$ cat intermedia.pegjs
a = 'a'+ temp 'b'+ {
                      console.log("acción final");
                      return "hello world!";
                    }
temp = { console.log("acción intermedia"); }
```

Este es el progrma que usa el parser generado:

```
[~/srcPLgrado/pegjs/examples(master)]$ cat main_intermedia.js
var parser = require("intermedia");
var input = process.argv[2] || 'aabb';
var result = parser.parse(input);
console.log(result);
```

al ejecutar tenemos:

```
[~/srcPLgrado/pegjs/examples(master)]$ pegjs intermedia.pegjs
[~/srcPLgrado/pegjs/examples(master)]$ node main_intermedia.js
acción intermedia
acción final
hello world!
```

4.5. PegJS en los Browser

Donde

- `~/srcPLgrado/pegjs/examples(master)]$ pwd -P`
  `/Users/casiano/local/src/javascript/PLgrado/pegjs/examples`
- `[~/srcPLgrado/pegjs/examples(master)]$ git remote -v`
  `dmajda  https://github.com/dmajda/pegjs.git (fetch)`
  `dmajda  https://github.com/dmajda/pegjs.git (push)`
  `origin  git@github.com:crguezl/pegjs.git (fetch)`
  `origin  git@github.com:crguezl/pegjs.git (push)`
- https://github.com/crguezl/pegjs/tree/master/examples
-

Versiones para Browser Podemos usar directamente las versiones para los browser:

- PEG.js — minified
- PEG.js — development

La opción -e de pegjs

```
[~/Dropbox/src/javascript/PLgrado/jison]$ pegjs --help
Usage: pegjs [options] [--] [<input_file>] [<output_file>]

Generates a parser from the PEG grammar specified in the <input_file> and
writes it to the <output_file>.

If the <output_file> is omitted, its name is generated by changing the
<input_file> extension to ".js". If both <input_file> and <output_file> are
omitted, standard input and output are used.

Options:
  -e, --export-var <variable>  name of the variable where the parser object
                               will be stored (default: "module.exports")
      --cache                  make generated parser cache results
      --track-line-and-column  make generated parser track line and column
  -v, --version                print version information and exit
  -h, --help                   print help and exit
```

Compilación

Le indicamos que el parser se guarde en `calculator`:

```
[~/Dropbox/src/javascript/PLgrado/pegjs/examples(master)]$ rake web
../bin/pegjs -e calculator arithmetics.pegjs
```

```
[~/srcPLgrado/pegjs/examples(master)]$ head -5 arithmetics.js
calculator = (function() {
  /*
   * Generated by PEG.js 0.7.0.
   *
   * http://pegjs.majda.cz/
```

calc.js Ahora, desde el JavaScript que llama al parser accedemos al objeto mediante la variable `calculator`:

```
[~/srcPLgrado/pegjs/examples(master)]$ cat calc.js
$(document).ready(function() {
  $('#eval').click(function() {
    try {
      var result = calculator.parse($('#input').val());
      $('#output').html(result);
    } catch (e) {
      $('#output').html('<div class="error"><pre>\n' + String(e) + '\n</pre></div>');
    }
  });

  $("#examples").change(function(ev) {
    var f = ev.target.files[0];
    var r = new FileReader();
    r.onload = function(e) {
      var contents = e.target.result;

      input.innerHTML = contents;
    }
    r.readAsText(f);
  });

});
```

arithmetic.pegjs El PEG describe una calculadora:

```
[~/Dropbox/src/javascript/PLgrado/pegjs/examples(master)]$ cat arithmetics.pegjs
/*
 * Classic example grammar, which recognizes simple arithmetic expressions like
 * "2*(3+4)". The parser generated from this grammar then computes their value.
 */

start
  = additive

additive
  = left:multiplicative PLUS right:additive { return left + right; }
  / left:multiplicative MINUS right:additive { return left - right; }
  / multiplicative
```

```
multiplicative
  = left:primary MULT right:multiplicative { return left * right; }
  / left:primary DIV right:multiplicative { return left / right; }
  / primary

primary
  = integer
  / LEFTPAR additive:additive RIGHTPAR { return additive; }

integer "integer"
  = NUMBER

_ = $[ \t\n\r]*

PLUS = _"+"_
MINUS = _"-"_
MULT = _"*"_
DIV = _"/"_
LEFTPAR = _"("_
RIGHTPAR = _")"_
NUMBER = _ digits:$[0-9]+ _ { return parseInt(digits, 10); }
```

calculator.html

```
[~/srcPLgrado/pegjs/examples(master)]$ cat calculator.html
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>pegjs</title>
    <link rel="stylesheet" href="global.css" type="text/css" media="screen" charset="utf-8" />
  </head>
  <body>
    <h1>pegjs</h1>
    <div id="content">
      <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js"></script
      <script src="arithmetics.js"></script>
      <script src="calc.js"></script>

      <p>
      Load an example:
      <input type="file" id="examples" />
      </p>

      <p>
      <table>
        <tr>
          <td>
            <textarea id="input" autofocus cols = "40" rows = "4">2+3*4</textarea>
          </td>
          <td class="output">
            <pre>
              <span id="output"></span> <!-- Output goes here! -->
```

```
            </pre>
          </td>
          <td><button id="eval" type="button">eval</button></td>
        </tr>
      </table>
      </p>
    </div>
  </body>
</html>
```

Figura 4.1: pegjs en la web

4.6. Eliminación de la Recursividad por la Izquierda en PEGs

Donde

- `[~/srcPLgrado/pegjs-coffee-plugin/examples(master)]$ pwd -P`
  `/Users/casiano/local/src/javascript/PLgrado/pegjs-coffee-plugin/examples`
- `[~/srcPLgrado/pegjs-coffee-plugin/examples(master)]$ git remote -v`
  `dignifiedquire  git@github.com:Dignifiedquire/pegjs-coffee-plugin.git (fetch)`
  `dignifiedquire  git@github.com:Dignifiedquire/pegjs-coffee-plugin.git (push)`
  `origin  git@github.com:crguezl/pegjs-coffee-plugin.git (fetch)`
  `origin  git@github.com:crguezl/pegjs-coffee-plugin.git (push)`
- https://github.com/crguezl/pegjs-coffee-plugin/tree/master/examples

PEGjs Coffee Plugin PEGjs Coffee Plugin is a plugin for PEG.js to use CoffeeScript in actions. Veamos un ejemplo de uso via la API:

```
[~/srcPLgrado/pegjs/examples(master)]$ cat plugin.coffee
#!/usr/bin/env coffee
PEG = require 'pegjs'
coffee = require 'pegjs-coffee-plugin'
grammar = """
a = 'hello' _ b { console.log 1; "hello world!" }
b = 'world'     { console.log 2 }
_ = [ \t]+      { console.log 3 }
```

```
"""
parser = PEG.buildParser grammar, plugins: [coffee]
r = parser.parse "hello world"
console.log(r)

[~/srcPLgrado/pegjs/examples(master)]$ coffee plugin.coffee
3
2
1
hello world!
```

Un Esquema de Traducción Recursivo por la Izquierda Consideremos el siguiente esquema de traducción implementado en Jison :

```
[~/srcPLgrado/pegjs-coffee-plugin/examples(master)]$ cat leftrec.jison
/*
Exercise: Find a PEG equivalent to the following left-recursive
grammar:
*/
%lex
%%

\s+               { /* skip whitespace */ }
y                 { return 'y';}
.                 { return 'x';}

/lex

%{
  do_y = function(y)   { console.log("A -> 'y'   do_y("+y+")"); return y; }
  do_x = function(a, x){ console.log("A -> A 'x' do_x("+a+", "+x+")"); return a+x; }
%}

%%
A : A 'x' { $$ = do_x($1, $2); }
  | 'y'   { $$ = do_y($1); }
;

[~/srcPLgrado/pegjs-coffee-plugin/examples(master)]$ jison leftrec.jison
[~/srcPLgrado/pegjs-coffee-plugin/examples(master)]$ ls -ltr leftrec.j*
-rw-r--r--  1 casiano  staff    441 18 mar 20:22 leftrec.jison
-rw-r--r--  1 casiano  staff  20464 18 mar 20:34 leftrec.js

[~/srcPLgrado/pegjs-coffee-plugin/examples(master)]$ cat main_leftrec.js
var parser = require('./leftrec');
input = "y x x x";
var r = parser.parse(input);

[~/srcPLgrado/pegjs-coffee-plugin/examples(master)]$ node main_leftrec.js
A -> 'y'   do_y(y)
A -> A 'x' do_x(y, x)
A -> A 'x' do_x(yx, x)
A -> A 'x' do_x(yxx, x)
```

Métodología

Es posible modificar la gramática para eliminar la recursión por la izquierda. En este apartado nos limitaremos al caso de recursión por la izquierda directa. La generalización al caso de recursión por la izquierda no-directa se reduce a la iteración de la solución propuesta para el caso directo.

Consideremos una variable A con dos producciones:

$$A \to A\alpha | \ \beta$$

donde $\alpha, \beta \in (V \cup \Sigma)^*$ no comienzan por A. Estas dos producciones pueden ser sustituidas por:

$$A \to \beta\alpha*$$

eliminando así la recursión por la izquierda.

Solución

```
[~/pegjs-coffee-remove-left(master)]$ cat -n remove_left_recursive.pegjs
     1  /*
     2
     3  Exercise: Find a PEG equivalent to the following left-recursive
     4  grammar:
     5
     6  A : A 'x' { $$ = do_x($1, $2); } | 'y' { $$ = do_y($1); }
     7
     8  */
     9
    10  {
    11    @do_y = (y)   -> console.log("do_y(#{y})"); y
    12    @do_x = (a, x)-> console.log("do_x(#{a}, #{x})"); a+x
    13  }
    14
    15  A = y:'y' xs:('x'*)
    16       {
    17          a = @do_y(y)
    18          for x in xs
    19            a = @do_x(a, x)
    20          a
    21       }

[~/pegjs-coffee-remove-left(master)]$ pegjs --plugin pegjs-coffee-plugin remove_left_recursive
[~/pegjs-coffee-remove-left(master)]$ ls -ltr | tail -1
-rw-rw-r--  1 casiano  staff   8919  3 jun 10:42 remove_left_recursive.js

[~/pegjs-coffee-remove-left(master)]$ cat use_remove_left.coffee
PEG = require("./remove_left_recursive.js")
inputs = [
           "yxx"
           "y"
           "yxxx"
         ]

for input in inputs
  console.log("input = #{input}")
  r = PEG.parse input
  console.log("result = #{r}\n")
```

```
[~/pegjs-coffee-remove-left(master)]$ coffee use_remove_left.coffee
input = yxx
do_y(y)
do_x(y, x)
do_x(yx, x)
result = yxx

input = y
do_y(y)
result = y

input = yxxx
do_y(y)
do_x(y, x)
do_x(yx, x)
do_x(yxx, x)
result = yxxx
```

4.7. Eliminando la Recursividad por la Izquierda en la Calculadora

```
[~/Dropbox/src/javascript/PLgrado/pegjs/examples(master)]$ cat simple.pegjs
/* From the Wikipedia
Value   ← [0-9]+ / '(' Expr ')'
Product ← Value (('*' / '/') Value)*
Sum     ← Product (('+' / '-') Product)*
Expr    ← Sum
*/
{
  function reduce(left, right) {
    var sum = left;
    // console.log("sum = "+sum);
    for(var i = 0; i < right.length;i++) {
      var t = right[i];
      var op = t[0];
      var num = t[1];
      switch(op) {
        case '+' : sum += num; break;
        case '-' : sum -= num; break;
        case '*' : sum *= num; break;
        case '/' : sum /= num; break;
        default : console.log("Error! "+op);
      }
      // console.log("sum = "+sum);
    }
    return sum;
  }
}

sum   = left:product right:($[+-] product)* { return reduce(left, right); }
product = left:value right:($[*/] value)*   { return reduce(left, right); }
value   = number:$[0-9]+                    { return parseInt(number,10); }
        / '(' sum:sum ')'                   { return sum; }
```

Es posible especificar mediante llaves un código que este disponible dentro de las acciones semánti-

cas.

Ejecución:

```
[~/pegjs/examples(master)]$ cat use_simple.js
var PEG = require("./simple.js");
var r = PEG.parse("2-3-4");
console.log(r);

[~/pegjs/examples(master)]$ node use_simple.js
-5
```

Veamos otra ejecución:

```
[~/Dropbox/src/javascript/PLgrado/pegjs/examples(master)]$ cat use_simple.js
var PEG = require("./simple.js");
var r = PEG.parse("2+3*(2+1)-10/2");
console.log(r);

[~/Dropbox/src/javascript/PLgrado/pegjs/examples(master)]$ ../bin/pegjs simple.pegjs
[~/Dropbox/src/javascript/PLgrado/pegjs/examples(master)]$ node use_simple.js
6
```

4.8. Eliminación de la Recursividad por la Izquierda y Atributos Heredados

La sección anterior da una forma sencilla de resolver el problema respetando la semántica. Si no se dispone de operadores de repetición la cosa se vuelve mas complicada. Las siguientes secciones muestran una solución para transformar un esquema de traducción recursivo por la izquierda en otro no recursivo por la izquierda respetando el orden en el que se ejecutan las acciones semánticas. Por último se ilustra como se puede aplicar esta técnica en `pegjs` (aunque obviamente es mucho mejor usar la ilustrada anteriormente).

4.8.1. Eliminación de la Recursión por la Izquierda en la Gramática

Es posible modificar la gramática para eliminar la recursión por la izquierda. En este apartado nos limitaremos al caso de recursión por la izquierda directa. La generalización al caso de recursión por la izquierda no-directa se reduce a la iteración de la solución propuesta para el caso directo.

Consideremos una variable A con dos producciones:

$$A \to A\alpha | \ \beta$$

donde $\alpha, \beta \in (V \cup \Sigma)^*$ no comienzan por A. Estas dos producciones pueden ser sustituidas por:

$$\begin{array}{l} A \to \beta R \\ R \to \alpha R \mid \epsilon \end{array}$$

eliminando así la recursión por la izquierda.

Definición 4.8.1. *La producción $R \to \alpha R$ se dice* recursiva por la derecha.

Las producciones recursivas por la derecha dan lugar a árboles que se hunden hacia la derecha. Es mas difícil traducir desde esta clase de árboles operadores como el menos, que son asociativos a izquierdas.

Ejercicio 4.8.1. *Elimine la recursión por la izquierda de la gramática*

$expr \rightarrow expr - NUM$
$expr \rightarrow NUM$

4.8.2. Eliminación de la Recursión por la Izquierda en un Esquema de Traducción

La eliminación de la recursión por la izquierda es sólo un paso: debe ser extendida a esquemas de traducción, de manera que no sólo se preserve el lenguaje sino la secuencia de acciones. Supongamos que tenemos un esquema de traducción de la forma:

$A \rightarrow A\alpha$ `{ alpha_action }`
$A \rightarrow A\beta$ `{ beta_action }`
$A \rightarrow \gamma$ `{ gamma_action }`

para una sentencia como $\gamma\beta\alpha$ la secuencia de acciones será:

```
gamma_action  beta_action alpha_action
```

¿Cómo construir un esquema de traducción para la gramática resultante de eliminar la recursión por la izquierda que ejecute las acciones asociadas en el mismo orden?. Supongamos para simplificar, que las acciones no dependen de atributos ni computan atributos, sino que actúan sobre variables globales. En tal caso, la siguiente ubicación de las acciones da lugar a que se ejecuten en el mismo orden:

$A \rightarrow \gamma$ `{ gamma_action }` R
$R \rightarrow \beta$ `{ beta_action }` R
$R \rightarrow \alpha$ `{ alpha_action }` R
$R \rightarrow \epsilon$

Si hay atributos en juego, la estrategia para construir un esquema de traducción equivalente para la gramática resultante de eliminar la recursividad por la izquierda se complica. Consideremos de nuevo el esquema de traducción de infijo a postfijo de expresiones aritméticas de restas:

$expr \rightarrow expr_1 - NUM$ `{ $expr{T} = $expr[1]{T}." ".$NUM{VAL}." - "}`
$expr \rightarrow NUM$ `{ $expr{T} = $NUM{VAL} }`

En este caso introducimos un atributo `H` para los nodos de la clase r el cuál acumula la traducción a postfijo hasta el momento. Observe como este atributo se computa en un nodo r a partir del correspondiente atributo del el padre y/o de los hermanos del nodo:

$expr \rightarrow NUM$ `{ $r{H} = $NUM{VAL} }` r `{ $expr{T} = $r{T} }`
$r \rightarrow -NUM$ `{ $r_1{H} = $r{H}." ".$NUM{VAL}." - " }` r_1 `{ $r{T} = $r_1{T} }`
$r \rightarrow \epsilon$ `{ $r{T} = $r{H} }`

El atributo `H` es un ejemplo de atributo heredado.

4.8.3. Eliminación de la Recursividad por la Izquierda en PEGJS

PegJS no permite acciones intermedias. Tampoco se puede acceder al atributo de la parte izquierda. Por eso, a la hora de implantar la solución anterior debemos introducir variables sintácticas temporales que produzcan la palabra vacía y que vayan acompañadas de la acción semántica correspondiente.

Además nos obliga a usar variables visibles por todas las reglas semánticas para emular el acceso a los atributos de la parte izquierda de una regla de proudcción.

El siguiente ejemplo ilustra como eliminar la reucrusión por la izquierda respetando la asociatividad de la oepración de diferencia:

```
[~/pegjs/examples(master)]$ cat inherited.pegjs
{
  var h = 0, number = 0;
}
e = NUMBER aux1 r          { return h; }
aux1 = /* empty */         { h = number; }

r =   '-' NUMBER  aux2 r   { return h; }
    / /* empty */
aux2 = /* empty */         { h -= number; }

NUMBER = _ digits:$[0-9]+ _  { number = parseInt(digits, 10); return number; }

_ = $[ \t\n\r]*
```

```
[~/pegjs/examples(master)]$ cat use_inherited.js
var PEG = require("./inherited.js");
var r = PEG.parse("2-1-1");
console.log(r);

var r = PEG.parse("4-2-1");
console.log(r);

var r = PEG.parse("2-3-1");
console.log(r);


[~/pegjs/examples(master)]$ pegjs inherited.pegjs
Referenced rule "$" does not exist.
[~/pegjs/examples(master)]$ ../bin/pegjs inherited.pegjs
[~/pegjs/examples(master)]$ node use_inherited.js
0
1
-2
```

4.9. Dangling else: Asociando un else con su if mas cercano

The dangling else is a problem in computer programming in which an optional `else clause` in an `If{then({else)` statement results in nested conditionals being ambiguous.

Formally, the reference context-free grammar of the language is ambiguous, meaning there is more than one correct parse tree.

In many programming languages one may write conditionally executed code in two forms:
the `if-then` form, and the `if-then-else` form – the `else` clause is optional:

```
if a then s
if a then s1 else s2
```

This gives rise to an ambiguity in interpretation when there are nested statements, specifically whenever an `if-then` form appears as `s1` in an `if-then-else` form:

```
if a then if b then s else s2
```

In this example, `s` is unambiguously executed when `a` is `true` and `b` is `true`, but one may interpret `s2` as being executed when `a` is `false`

- (thus attaching the `else` to the first `if`) or when

- `a` is `true` and `b` is `false` (thus attaching the `else` to the second `if`).

In other words, one may see the previous statement as either of the following expressions:

```
if a then (if b then s) else s2
```

or

```
if a then (if b then s else s2)
```

This is a problem that often comes up in compiler construction, especially scannerless parsing.

The convention when dealing with the dangling `else` is to attach the `else` to the nearby `if` statement.

Programming languages like Pascal and C follow this convention, so there is no ambiguity in the semantics of the language, though the use of a parser generator may lead to ambiguous grammars. In these cases alternative grouping is accomplished by explicit blocks, such as `begin...end` in Pascal and `{...}` in C.

Here follows a solution in PEG.js:

danglingelse.pegjs

```
$ cat danglingelse.pegjs
/*
S ← 'if' C 'then' S 'else' S / 'if' C 'then' S
*/

S =   if C:C then S1:S else S2:S { return [ 'ifthenelse', C, S1, S2 ]; }
    / if C:C then S:S            { return [ 'ifthen', C, S]; }
    / O                          { return 'O'; }
_ = ' '*
C = _'c'_                        { return 'c'; }
O = _'o'_                        { return 'o'; }
else = _'else'_
if = _'if'_
then = _'then'_
```

use_danglingelse.js

```
$ cat use_danglingelse.js
var PEG = require("./danglingelse.js");
var r = PEG.parse("if c then if c then o else o");
console.log(r);
```

Ejecución

```
$ ../bin/pegjs danglingelse.pegjs
$ node use_danglingelse.js
[ 'ifthen', 'c', [ 'ifthenelse', 'c', 'O', 'O' ] ]
```

Donde

- `[~/srcPLgrado/pegjs/examples(master)]$ pwd -P`
  `/Users/casiano/local/src/javascript/PLgrado/pegjs/examples`

- [~/srcPLgrado/pegjs/examples(master)]$ git remote -v
  dmajda https://github.com/dmajda/pegjs.git (fetch)
  dmajda https://github.com/dmajda/pegjs.git (push)
  origin git@github.com:crguezl/pegjs.git (fetch)
  origin git@github.com:crguezl/pegjs.git (push)
- https://github.com/crguezl/pegjs/tree/master/examples

4.10. Not Predicate: Comentarios Anidados

The following recursive PEG.js program matches Pascal-style nested comment syntax:

```
(* which can (* nest *) like this *)
```

Pascal_comments.pegjs

```
$ cat pascal_comments.pegjs
/* Pascal nested comments */

P     =   prog:N+                          { return prog; }
N     =   chars:$(!Begin ANY)+             { return chars;}
        / C
C     = Begin chars:T* End                 { return chars.join(''); }
T     =   C
        / (!Begin !End char:ANY)           { return char;}
Begin = '(*'
End   = '*)'
ANY   =   'z'    /* any character */       { return 'z';  }
        / char:[^z]                        { return char; }
```

use_pascal_comments.js

```
$ cat use_pascal_comments.js
var PEG = require("./pascal_comments.js");
var r = PEG.parse(
  "not bla bla (* pascal (* nested *) comment *)"+
  " pum pum (* another comment *)");
console.log(r);
```

Ejecución

```
$ ../bin/pegjs pascal_comments.pegjs
$ node use_pascal_comments.js
[ 'not bla bla ',
  ' pascal  nested  comment ',
  ' pum pum ',
  ' another comment ' ]
```

Donde

- [~/srcPLgrado/pegjs/examples(master)]$ pwd -P
  /Users/casiano/local/src/javascript/PLgrado/pegjs/examples

- [~/srcPLgrado/pegjs/examples(master)]$ git remote -v
  dmajda https://github.com/dmajda/pegjs.git (fetch)
  dmajda https://github.com/dmajda/pegjs.git (push)
  origin git@github.com:crguezl/pegjs.git (fetch)
  origin git@github.com:crguezl/pegjs.git (push)
- https://github.com/crguezl/pegjs/tree/master/examples

4.11. Un Lenguaje Dependiente del Contexto

El lenguaje $\{a^n b^n c^n / n \in \mathcal{N}\}$ no puede ser expresado mediante una gramática independiente del contexto.

```
[~/Dropbox/src/javascript/PLgrado/pegjs/examples(master)]$ cat anbncn.pegjs
/*
  The following parsing expression grammar describes the classic
  non-context-free language :
              { a^nb^nc^n / n >= 1 }

           S ← &(A 'c') 'a'+ B !('a'/'b'/'c')
           A ← 'a' A? 'b'
           B ← 'b' B? 'c'
*/

S = &(A 'c') 'a'+ B !('a'/'b'/'c')
A = 'a' A? 'b'
B = 'b' B? 'c'
```

Este ejemplo puede ser obtenido desde GitHub:

```
[~/Dropbox/src/javascript/PLgrado/pegjs/examples(master)]$ git remote -v
dmajda  https://github.com/dmajda/pegjs.git (fetch)
dmajda  https://github.com/dmajda/pegjs.git (push)
origin  git@github.com:crguezl/pegjs.git (fetch)
origin  git@github.com:crguezl/pegjs.git (push)
```

Veamos un ejemplo de uso:

```
[~/Dropbox/src/javascript/PLgrado/pegjs/examples(master)]$ cat use_anbncn.js
var PEG = require("./anbncn.js");
var r = PEG.parse("aabbcc");
console.log(r);

try {
  r = PEG.parse("aabbc");
  console.log(r);
}
catch (e) {
  console.log("Grr...."+e);
}
```

Ejecución:

```
[~/Dropbox/src/javascript/PLgrado/pegjs/examples(master)]$ ../bin/pegjs anbncn.pegjs

[~/Dropbox/src/javascript/PLgrado/pegjs/examples(master)]$ node use_anbncn.js
[ '', [ 'a', 'a' ], [ 'b', [ 'b', '', 'c' ], 'c' ], '' ]
Grr....SyntaxError: Expected "c" but end of input found.
```

4.12. Usando Pegjs con CoffeeScript

Instalación de pegjs-coffee-plugin

```
[~/Dropbox/src/javascript/PLgrado/pegjs/examples(master)]$ sudo  npm install -g pegjs-coffee-p
```

Ejemplo Sencillo

```
[~/Dropbox/src/javascript/PLgrado/pegjs-coffee-plugin/examples(master)]$ cat simple.pegjs
{
  @reduce = (left, right)->
    sum = left
    for t in right
      op = t[0]
      num = t[1]
      switch op
        when '+' then sum += num; break
        when '-' then sum -= num; break
        when '*' then sum *= num; break
        when '/' then sum /= num; break
        else console.log("Error! "+op)
    sum

}
sum   = left:product right:([+-] product)* { @reduce(left, right); }
product = left:value right:([*/] value)*   { @reduce(left, right); }
value   = number:[0-9]+                    { parseInt(number.join(''),10) }
        / '(' sum:sum ')'                  { sum }
```

```
[~/Dropbox/src/javascript/PLgrado/pegjs-coffee-plugin/examples(master)]$  cat use_simple.coffe
PEG = require("./simple.js")
r = PEG.parse("2+3*(2+1)-10/2")
console.log(r)
```

```
[~/Dropbox/src/javascript/PLgrado/pegjs-coffee-plugin/examples(master)]$ cat Rakefile

task :default do
  sh "pegcoffee simple.pegjs"
end

task :run do
  sh "coffee use_simple.coffee"
end
```

```
[~/Dropbox/src/javascript/PLgrado/pegjs-coffee-plugin/examples(master)]$ rake
pegcoffee simple.pegjs
[~/Dropbox/src/javascript/PLgrado/pegjs-coffee-plugin/examples(master)]$ rake run
coffee use_simple.coffee
6
```

Véase También

- pegjs-coffee-plugin en GitHub

4.13. Práctica: Analizador de PL0 Ampliado Usando PEG.js

Reescriba el analizador sintáctico del lenguaje PL0 realizado en la práctica 2.6 usando PEG.js .

Donde

- Repositorio en GitHub
- Despliegue en Heroku
- `[~/srcPLgrado/pegjscalc(master)]$ pwd -P`
  `/Users/casiano/local/src/javascript/PLgrado/pegjscalc`
- `[~/srcPLgrado/pegjscalc(master)]$ git remote -v`
  `heroku  git@heroku.com:pegjspl0.git (fetch)`
  `heroku  git@heroku.com:pegjspl0.git (push)`
  `origin  git@github.com:crguezl/pegjscalc.git (fetch)`
  `origin  git@github.com:crguezl/pegjscalc.git (push)`

Tareas

- Modifique `block` y `statement` para que los `procedure` reciban argumentos y las llamadas a procedimiento puedan pasar argumentos. Añada `if ... then ... else ...`.
- Actualice la documentación de la gramática para que refleje la gramática ampliada
- Limite el número de programas que se pueden salvar a un número prefijado, por ejemplo 10. Si se intenta salvar uno se suprime uno al azar y se guarda el nuevo.
- Las pruebas deben comprobar que la asociatividad a izquierdas funciona bien y probar todos los constructos del lenguaje así como alguna situación de error

Referencias para esta Práctica

- Véase el capítulo *Heroku* **??**
- Heroku Postgres
- Véase el capítulo *DataMapper* **??**

4.14. Práctica: Ambiguedad en C++

This lab illustrates a problem that arises in C++. The C++ syntax does not disambiguate between expression statements (`stmt`) and declaration statements (`decl`). The ambiguity arises when an expression statement has a function-style cast as its left-most subexpression. Since C does not support function-style casts, this ambiguity does not occur in C programs. For example, the phrase

```
int (x) = y+z;
```

parses as either a `decl` or a `stmt`.

The disambiguation rule used in C++ is that *if the statement can be interpreted both as a declaration and as an expression, the statement is interpreted as a declaration statement.*

The following examples disambiguate into *expression* statements when the potential *declarator* is followed by an operator different from equal or semicolon (`type_spec` stands for a type specifier):

expr	dec

`type_spec(i)++;`
`type_spec(i,3)<<d;`
`type_spec(i)->l=24;`
	`type_spec(*i)(int);`
	`type_spec(j)[5];`
	`type_spec(m) = { 1, 2 };`
	`type_spec(a);`
	`type_spec(*b)();`
	`type_spec(c)=23;`
	`type_spec(d),e,f,g=0;`
	`type_spec(h)(e,3);`

Regarding to this problem, Bjarne Stroustrup remarks:

> *Consider analyzing a statement consisting of a sequence of tokens as follows:*
>
> `type_spec (dec_or_exp) tail`
>
> *Here* `dec_or_exp` *must be a declarator, an expression, or both for the statement to be legal. This implies that* `tail` *must be a semicolon, something that can follow a parenthesized declarator or something that can follow a parenthesized expression, that is, an initializer,* `const`*,* `volatile`*,* `(`*,* `[`*, or a postfix or infix operator. The general cases cannot be resolved without backtracking, nested grammars or similar advanced parsing strategies. In particular, the lookahead needed to disambiguate this case is not limited.*

The following grammar depicts an oversimplified version of the C++ ambiguity:

```
$ cat CplusplusNested.y
%token ID INT NUM

%right '='
%left '+'

%%
prog:
    /* empty */
  | prog stmt
;

stmt:
    expr ';'
  | decl
;

expr:
    ID
  | NUM
  | INT '(' expr ')' /* typecast */
  | expr '+' expr
  | expr '=' expr
;

decl:
    INT declarator ';'
```

```
  | INT declarator '=' expr ';'
;

declarator:
    ID
  | '(' declarator ')'
;

%%
```

Escriba un programa PegJS en CoffeeScript que distinga correctamente entre declaraciones y sentencias. Este es un ejemplo de un programa que usa una solución al problema:

```
[~/Dropbox/src/javascript/PLgrado/pegjs-coffee-plugin/examples(master)]$ cat use_cplusplus.cof
PEG = require("./cplusplus.js")
input = "int (a); int c = int (b);"

r = PEG.parse(input)
console.log("input = '#{input}'\noutput="+JSON.stringify r)

input = "int b = 4+2  ;  "
r = PEG.parse(input)
console.log("input = '#{input}'\noutput="+JSON.stringify r)

input = "bum = caf = 4-1;\n"
r = PEG.parse(input)
console.log("input = '#{input}'\noutput="+JSON.stringify r)

input = "b2 = int(4);"
r = PEG.parse(input)
console.log("input = '#{input}'\noutput="+JSON.stringify r)

input = "int(4);"
r = PEG.parse(input)
console.log("input = '#{input}'\noutput="+JSON.stringify r)
```

Y este un ejemplo de salida:

```
$ pegcoffee cplusplus.pegjs
$ coffee use_cplusplus.coffee
input = 'int (a); int c = int (b);'
output=["decl","decl"]
input = 'int b = 4+2  ;  '
output=["decl"]
input = 'bum = caf = 4-1;
'
output=["stmt"]
input = 'b2 = int(4);'
output=["stmt"]
input = 'int(4);'
output=["stmt"]
```

4.15. Práctica: Inventando un Lenguaje: Tortoise

El objetivo de esta práctica es crear un lenguaje de programación imperativa sencillo de estilo LOGO. Para ello lea el capítulo Inventing a Language - Tortoise del curso PL101: Create Your Own Programming

de Nathan Whitehead. Haga todos los ejercicios e implemente el lenguaje descrito.

Puede encontrar una solución a la práctica en GitHub en el repositorio pl101 de Dave Ingram. Usela como guía cuando se sienta desorientado.

Recursos

- Inventing a Language - Tortoise por Nathan Whitehead
- Repositorio dingram / pl101 en GitHub con las soluciones a esta práctica.
  - Blog de dingram (Dave Ingram)
- Repositorio PatrixCR / PL101 en GitHub con las soluciones a esta práctica.
- Repositorio Clinton N. Dreisbach/ PL101 en GitHub con contenidos del curso PL101
- Foro
- Sobre Nathan Whitehead
  - Nathan's Lessons
  - Nathan Whitehead en GitHub
  - Nathan in YouTube

Capítulo 5

Análisis Sintáctico Ascendente en JavaScript

5.1. Conceptos Básicos para el Análisis Sintáctico

Suponemos que el lector de esta sección ha realizado con éxito un curso en teoría de autómatas y lenguajes formales. Las siguientes definiciones repasan los conceptos mas importantes.

Definición 5.1.1. *Dado un conjunto A, se define A^* el cierre de Kleene de A como: $A^* = \cup_{n=0}^{\infty} A^n$*

Se admite que $A^0 = \{\epsilon\}$, donde ϵ denota la palabra vacía, esto es la palabra que tiene longitud cero, formada por cero símbolos del conjunto base A.

Definición 5.1.2. *Una gramática G es una cuaterna $G = (\Sigma, V, P, S)$. Σ es el conjunto de terminales. V es un conjunto (disjunto de Σ) que se denomina conjunto de* variables sintácticas *o* categorías gramáticales*, P es un conjunto de pares de $V \times (V \cup \Sigma)^*$. En vez de escribir un par usando la notación $(A, \alpha) \in P$ se escribe $A \to \alpha$. Un elemento de P se denomina producción. Por último, S es un símbolo del conjunto V que se denomina símbolo de arranque.*

Definición 5.1.3. *Dada una gramática $G = (\Sigma, V, P, S)$ y $\mu = \alpha A \beta \in (V \cup \Sigma)^*$ una frase formada por variables y terminales y $A \to \gamma$ una producción de P, decimos que μ deriva en un paso en $\alpha\gamma\beta$. Esto es, derivar una cadena $\alpha A \beta$ es sustituir una variable sintáctica A de V por la parte derecha γ de una de sus reglas de producción. Se dice que μ deriva en n pasos en δ si deriva en $n-1$ pasos en una cadena $\alpha A \beta$ la cual deriva en un paso en δ. Se escribe entonces que $\mu \stackrel{*}{\Longrightarrow} \delta$. Una cadena deriva en 0 pasos en si misma.*

Definición 5.1.4. *Dada una gramática $G = (\Sigma, V, P, S)$ se denota por $L(G)$ o lenguaje generado por G al lenguaje:*

$$L(G) = \{x \in \Sigma^* : S \stackrel{*}{\Longrightarrow} x\}$$

Esto es, el lenguaje generado por la gramática G esta formado por las cadenas de terminales que pueden ser derivados desde el símbolo de arranque.

Definición 5.1.5. *Una derivación que comienza en el símbolo de arranque y termina en una secuencia formada por sólo terminales de Σ se dice* completa*.*

Una derivación $\mu \stackrel{*}{\Longrightarrow} \delta$ en la cual en cada paso $\alpha A x$ la regla de producción aplicada $A \to \gamma$ se aplica en la variable sintáctica mas a la derecha se dice una derivación a derechas

Una derivación $\mu \stackrel{*}{\Longrightarrow} \delta$ en la cual en cada paso $xA\alpha$ la regla de producción aplicada $A \to \gamma$ se aplica en la variable sintáctica mas a la izquierda se dice una derivación a izquierdas

Definición 5.1.6. *Observe que una derivación puede ser representada como un árbol cuyos nodos están etiquetados en $V \cup \Sigma$. La aplicación de la regla de producción $A \to \gamma$ se traduce en asignar como hijos del nodo etiquetado con A a los nodos etiquetados con los símbolos $X_1 \dots X_n$ que constituyen la frase $\gamma = X_1 \dots X_n$. Este árbol se llama* árbol sintáctico concreto *asociado con la derivación.*

Definición 5.1.7. *Observe que, dada una frase $x \in L(G)$ una derivación desde el símbolo de arranque da lugar a un árbol. Ese árbol tiene como raíz el símbolo de arranque y como hojas los terminales $x_1 \dots x_n$ que forman x. Dicho árbol se denomina* árbol de análisis sintáctico concreto *de x. Una derivación determina una forma de recorrido del árbol de análisis sintáctico concreto.*

Definición 5.1.8. *Una gramática G se dice ambigua si existe alguna frase $x \in L(G)$ con al menos dos árboles sintácticos. Es claro que esta definición es equivalente a afirmar que existe alguna frase $x \in L(G)$ para la cual existen dos derivaciones a izquierda (derecha) distintas.*

5.1.1. Ejercicio

Dada la gramática con producciones:

program → declarations statements | statements
declarations → declaration ';' declarations | declaration ';'
declaration → INT idlist | STRING idlist
statements → statement ';' statements | statement
statement → ID '=' expression | P expression
expression → term '+' expression | term
term → factor '*' term | factor
factor → '(' expression ')' | ID | NUM | STR
idlist → ID ',' idlist | ID

En esta gramática, Σ esta formado por los caracteres entre comillas simples y los símbolos cuyos identificadores están en mayúsculas. Los restantes identificadores corresponden a elementos de V. El símbolo de arranque es $S = program$.

Conteste a las siguientes cuestiones:

1. Describa con palabras el lenguaje generado.
2. Construya el árbol de análisis sintáctico concreto para cuatro frases del lenguaje.
3. Señale a que recorridos del árbol corresponden las respectivas derivaciones a izquierda y a derecha en el apartado 2.
4. ¿Es ambigua esta gramática?. Justifique su respuesta.

5.2. Ejemplo Simple en Jison

Jison es un generador de analizadores sintácticos LALR. Otro analizador LALR es JS/CC.

Gramática

```
%%

S   : A
    ;
A   : /* empty */
    | A x
    ;
```

basic2_lex.jison

```
[~/jison/examples/basic2_lex(develop)]$ cat basic2_lex.jison
/* description: Basic grammar that contains a nullable A nonterminal. */
```

```
%lex
%%

\s+                {/* skip whitespace */}
[a-zA-Z_]\w*       {return 'x';}

/lex

%%

S   : A
            { return $1+" identifiers"; }
    ;
A   : /* empty */
            {
               console.log("starting");
               $$ = 0;
            }
    | A x   {
               $$ = $1 + 1;
               console.log($$)
            }
    ;
```

index.html

```
$ cat basic2_lex.html
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Jison</title>
    <link rel="stylesheet" href="global.css" type="text/css" media="screen" charset="utf-8" />
  </head>
  <body>
    <h1>basic2_lex demo</h1>
    <div id="content">
      <script src="jquery/jquery.js"></script>
      <script src="basic2_lex.js"></script>
      <script src="main.js"></script>
      <p>
        <input type="text" value="x x x x" /> <button>parse</button>
        <span id="output"></span> <!-- Output goes here! -->
      </p>
    </div>
  </body>
</html>
```

Rakefile

```
$ cat Rakefile
# install package:
#     sudo npm install beautifier
#
```

```
#   more about beautifier:
#       https://github.com/rickeyski/node-beautifier

dec "compile the grammar basic2_lex_ugly.jison"
task :default => %w{basic2_lex_ugly.js} do
  sh "mv basic2_lex.js basic2_lex_ugly.js"
  sh "jsbeautify basic2_lex_ugly.js > basic2_lex.js"
  sh "rm -f basic2_lex_ugly.js"
end

file "basic2_lex_ugly.js" => %w{basic2_lex.jison} do
  sh "jison basic2_lex.jison -o basic2_lex.js"
end
```

1. node-beautifier

Véase También

1. JISON
2. Try Jison Examples
3. JavaScript 1.4 LR(1) Grammar 1999.
4. Creating a JavaScript Parser Una implementación de ECMAScript 5.1 usando Jison disponible en GitHub en https://github.com/cjihrig/jsparser. Puede probarse en: http://www.cjihrig.com/development/
5. Bison on JavaScript por Rolando Perez
6. Slogo a language written using Jison
7. List of languages that compile to JS
8. Prototype of a Scannerless, Generalized Left-to-right Rightmost (SGLR) derivation parser for JavaScript

global.css

```
[~/jison/examples/basic2_lex(develop)]$ cat global.css
html *
{
   font-size: large;
   /* The !important ensures that nothing can override what you've set in this style (unless i
   font-family: Arial;
}

.thumb {
    height: 75px;
    border: 1px solid #000;
    margin: 10px 5px 0 0;
  }

h1            { text-align: center; font-size: x-large; }
th, td        { vertical-align: top; text-align: left; }
/* #finaltable  * { color: white; background-color: black; }   */

/* #finaltable table { border-collapse:collapse; } */
```

```
/* #finaltable table, td { border:1px solid white; } */
#finaltable:hover td { background-color: blue; }
tr:nth-child(odd)    { background-color:#eee; }
tr:nth-child(even)    { background-color:#00FF66; }
input        { text-align: right;  border: none;       }     /* Align input to the right  */
textarea     { border: outset; border-color: white;       }
table        { border: inset; border-color: white; }
.hidden      { display: none; }
.unhidden    { display: block; }
table.center { margin-left:auto; margin-right:auto; }
#result      { border-color: red; }
tr.error       { background-color: red; }
pre.output   { background-color: white; }
span.repeated { background-color: red }
span.header { background-color: blue }
span.comments { background-color: orange }
span.blanks { background-color: green }
span.nameEqualValue { background-color: cyan }
span.error { background-color: red }

body
{
 background-color:#b0c4de;  /* blue */
}
```

5.2.1. Véase También

1. JISON
2. Try Jison Examples
3. JavaScript 1.4 LR(1) Grammar 1999.
4. Creating a JavaScript Parser Una implementación de ECAMScript 5.1 usando Jison disponible en GitHub en https://github.com/cjihrig/jsparser. Puede probarse en: http://www.cjihrig.com/development/
5. Slogo a language written using Jison
6. List of languages that compile to JS
7. Prototype of a Scannerless, Generalized Left-to-right Rightmost (SGLR) derivation parser for JavaScript

5.2.2. Práctica: Secuencia de Asignaciones Simples

Modifique este ejemplo para que el lenguaje acepte una secuencia de sentencias de asignación de la forma `ID = NUM` separadas por puntos y comas, por ejemplo `a = 4; b = 4.56; c = -8.57e34`. El analizador retorna un hash/objeto cuyas claves son los identificadores y cuyos valores son los números. Clone el repositorio en https://github.com/crguezl/jison-basic2.

Modifique los analizadores léxico y sintáctico de forma conveniente.

Añada acciones semánticas para que el analizador devuelva una tabla de símbolos con los identificadores y sus valores.

5.3. Ejemplo en Jison: Calculadora Simple

1. Enlace al fork del proyecto jison de crguezl (GitHub)

calculator.jison

```
[~/jison/examples/html_calc_example(develop)]$ cat calculator.jison

/* 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 '%'
"("                   return '('
")"                   return ')'
"PI"                  return 'PI'
"E"                   return 'E'
<<EOF>>               return 'EOF'
.                     return 'INVALID'

/lex

/* operator associations and precedence */

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

%start expressions

%% /* language grammar */

expressions
    : e EOF
        { typeof console !== 'undefined' ? console.log($1) : 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 '!'
        {{
          $$ = (function fact (n) { return n==0 ? 1 : fact(n-1) * n })($1);
        }}
    | e '%'
        {$$ = $1/100;}
    | '-' e %prec UMINUS
        {$$ = -$2;}
    | '(' e ')'
        {$$ = $2;}
    | NUMBER
        {$$ = Number(yytext);}
    | E
        {$$ = Math.E;}
    | PI
        {$$ = Math.PI;}
    ;
```

main.js

```
[~/jison/examples/html_calc_example(develop)]$ cat main.js
$(document).ready(function () {
  $("button").click(function () {
    try {
      var result = calculator.parse($("input").val())
      $("span").html(result);
    } catch (e) {
      $("span").html(String(e));
    }
  });
});
```

calculator.html

```
[~/jison/examples/html_calc_example(develop)]$ cat calculator.html
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Calc</title>
    <link rel="stylesheet" href="global.css" type="text/css" media="screen" charset="utf-8" />
  </head>
  <body>
    <h1>Calculator demo</h1>
    <div id="content">
      <script src="jquery/jquery.js"></script>
      <script src="calculator.js"></script>
      <script src="main.js"></script>
      <p>
        <input type="text" value="PI*4^2 + 5" /> <button>equals</button>
```

```
        <span></span> <!-- Output goes here! -->
      </p>
    </div>
  </body>
</html>
```

Rakefile

```
[~/jisoncalc(clase)]$ cat Rakefile
task :default => %w{calcugly.js} do
  sh "jsbeautify calcugly.js > calculator.js"
  sh "rm -f calcugly.js"
end

file "calcugly.js" => %w{calculator.jison} do
  sh "jison calculator.jison calculator.l -o calculator.js; mv calculator.js calcugly.js"
end

task :testf do
  sh "open -a firefox test/test.html"
end

task :tests do
  sh "open -a safari test/test.html"
end
```

global.css

```
[~/jison/examples/html_calc_example(develop)]$ cat global.css
html *
{
   font-size: large;
   /* The !important ensures that nothing can override what you've set in this style (unless i
   font-family: Arial;
}

.thumb {
    height: 75px;
    border: 1px solid #000;
    margin: 10px 5px 0 0;
  }

h1            { text-align: center; font-size: x-large; }
th, td        { vertical-align: top; text-align: left; }
/* #finaltable  * { color: white; background-color: black; }   */

/* #finaltable table { border-collapse:collapse; } */
/* #finaltable table, td { border:1px solid white; } */
#finaltable:hover td { background-color: blue; }
tr:nth-child(odd)    { background-color:#eee; }
tr:nth-child(even)    { background-color:#00FF66; }
input        { text-align: right;  border: none;       }     /* Align input to the right  */
textarea     { border: outset; border-color: white;       }
table        { border: inset; border-color: white; }
```

```
.hidden      { display: none; }
.unhidden    { display: block; }
table.center { margin-left:auto; margin-right:auto; }
#result      { border-color: red; }
tr.error       { background-color: red; }
pre.output   { background-color: white; }
span.repeated { background-color: red }
span.header { background-color: blue }
span.comments { background-color: orange }
span.blanks { background-color: green }
span.nameEqualValue { background-color: cyan }
span.error { background-color: red }

body
{
 background-color:#b0c4de;  /* blue */
}
```

test/assert.html

```
$ cat test/assert.js
var output = document.getElementById('output');

function assert( outcome, description) {
  var li = document.createElement('li');
  li.className = outcome ? 'pass' : 'fail';
  li.appendChild(document.createTextNode(description));

  output.appendChild(li);
};
```

test/test.css

```
~/jisoncalc(clase)]$ cat test/test.css
.pass:before {
  content: 'PASS: ';
  color:  blue;
  font-weight: bold;
}

.fail:before {
  content: 'FAIL: ';
  color: red;
  font-weight: bold;

}
```

test/test.html

```
[~/jisoncalc(clase)]$ cat test/test.html
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="UTF-8">
```

```
    <title>Testing Our Simple Calculator</title>
    <link rel="stylesheet" href="test.css" />
    <script type="text/javascript" src="../calculator.js"></script>

  </head>
  <body>
    <h1>Testing Our Simple Calculator
    </h1>

    <ul id="output"></ul>
    <script type="text/javascript" src="______.js"></script>

    <script type="text/javascript">
      var r = __________.parse("a = 4*8");
      assert(________________, "a is 4*8");
      assert(_____________, "32 == 4*8");
      r = calculator.parse("a = 4;\nb=a+1;\nc=b*2");
      assert(____________, "4 is the first computed result ");
      assert(_____________, "a is 4");
      assert(_____________, "b is 5");
      assert(_____________, "c is 10");
    </script>
      See the NetTuts+ tutorial at <a href="http://net.tutsplus.com/tutorials/javascript-ajax/
  </body>
</html>
```

5.3.1. Práctica: Calculadora con Listas de Expresiones y Variables

Modifique la calculadora vista en la sección anterior 5.3 para que el lenguaje cumpla los siguientes requisitos:

- Extienda el lenguaje de la calculadora para que admita expresiones de asignación `a = 2*3`
- Extienda el lenguaje de la calculadora para que admita listas de sentencias `a = 2; b = a +1`
- El analizador devuelve la lista de expresiones evaluadas y la tabla de símbolos (con las parejas variable-valor).
- Emita un mensaje de error específico si se intentan modificar las constantes `PI` y `e`.
- Emita un mensaje de error específico si se intenta una división por cero
- Emita un mensaje de error específico si se intenta acceder para lectura a una variable no inicializada `a = c`
- El lenguaje debería admitir expresiones vacías, estos es secuencias consecutivas de puntos y comas sin producir error (`a = 4;;; b = 5`)
- Introduzca pruebas unitarias como las descritas en la sección **??** (*Quick Tip: Quick and Easy JavaScript Test*

5.4. Conceptos Básicos del Análisis LR

Los analizadores generados por `jison` entran en la categoría de analizadores *LR*. Estos analizadores construyen una derivación a derechas inversa (o *antiderivación*). De ahí la R en LR (del inglés *rightmost derivation*). El árbol sintáctico es construido de las hojas hacia la raíz, siendo el último paso en la antiderivación la construcción de la primera derivación desde el símbolo de arranque.

Empezaremos entonces considerando las frases que pueden aparecer en una derivación a derechas. Tales frases consituyen el *lenguaje de las formas sentenciales a rderechas* FSD:

Definición 5.4.1. *Dada una gramática* $G = (\Sigma, V, P, S)$ *no ambigua, se denota por* FSD *(lenguaje de las formas Sentenciales a Derechas) al lenguaje de las sentencias que aparecen en una derivación a derechas desde el símbolo de arranque.*

$$FSD = \left\{ \alpha \in (\Sigma \cup V)* : \exists S \underset{RM}{\overset{*}{\Longrightarrow}} \alpha \right\}$$

*Donde la notacion RM indica una derivación a derechas (*rightmost*). Los elementos de* FSD *se llaman "formas sentenciales derechas".*

Dada una gramática no ambigua $G = (\Sigma, V, P, S)$ y una frase $x \in L(G)$ el proceso de antiderivación consiste en encontrar la última derivación a derechas que dió lugar a x. Esto es, si $x \in L(G)$ es porque existe una derivación a derechas de la forma

$$S \overset{*}{\Longrightarrow} yAz \Longrightarrow ywz = x.$$

El problema es averiguar que regla $A \to w$ se aplicó y en que lugar de la cadena x se aplicó. En general, si queremos antiderivar una forma sentencial derecha $\beta\alpha w$ debemos averiguar por que regla $A \to \alpha$ seguir y en que lugar de la forma (después de β en el ejemplo) aplicarla.

$$S \overset{*}{\Longrightarrow} \beta Aw \Longrightarrow \beta\alpha w.$$

La pareja formada por la regla y la posición se denomina *handle*, *mango* o *manecilla* de la forma. Esta denominación viene de la visualización gráfica de la regla de producción como una mano que nos permite escalar hacia arriba en el árbol. Los "dedos" serían los símbolos en la parte derecha de la regla de producción.

Definición 5.4.2. *Dada una gramática* $G = (\Sigma, V, P, S)$ *no ambigua, y dada una forma sentencial derecha* $\alpha = \beta\gamma x$*, con* $x \in \Sigma^*$*, el* mango *o* handle *de* α *es la última producción/posición que dió lugar a* α*:*

$$S \underset{RM}{\overset{*}{\Longrightarrow}} \beta Bx \Longrightarrow \beta\gamma x = \alpha$$

Escribiremos: $handle(\alpha) = (B \to \gamma, \beta\gamma)$. La función *handle* tiene dos componentes: $handle_1(\alpha) = B \to \gamma$ y $handle_2(\alpha) = \beta\gamma$

Si dispusieramos de un procedimiento que fuera capaz de identificar el mango, esto es, de detectar la regla y el lugar en el que se posiciona, tendríamos un mecanismo para construir un analizador. Lo curioso es que, a menudo es posible encontrar un autómata finito que reconoce el lenguaje de los prefijos $\beta\gamma$ que terminan en el mango. Con mas precisión, del lenguaje:

Definición 5.4.3. *El conjunto de* prefijos viables *de una gramática* G *se define como el conjunto:*

$$PV = \left\{ \delta \in (\Sigma \cup V)* : \exists S \underset{RM}{\overset{*}{\Longrightarrow}} \alpha = \beta\gamma x \text{ y } \delta \text{ es un } prefijo \text{ de } handle_2(\alpha) = \beta\gamma \right\}$$

Esto es, es el lenguaje de los prefijos viables es el conjunto de frases que son prefijos de $handle_2(\alpha)) = \beta\gamma$, siendo α una forma sentencial derecha ($\alpha \in FSD$). Los elementos de PV se denominan prefijos viables.

Obsérvese que si se dispone de un autómata que reconoce PV entonces se dispone de un mecanismo para investigar el lugar y el aspecto que pueda tener el mango. Si damos como entrada la sentencia $\alpha = \beta\gamma x$ a dicho autómata, el autómata aceptará la cadena $\beta\gamma$ pero rechazará cualquier extensión del prefijo. Ahora sabemos que el mango será alguna regla de producción de G cuya parte derecha sea un sufijo de $\beta\gamma$.

Definición 5.4.4. *El siguiente autómata finito no determinista puede ser utilizado para reconocer el lenguaje de los prefijos viables PV:*

- *Alfabeto* $= V \cup \Sigma$
- *Los estados del autómata se denominan LR(0) items. Son parejas formadas por una regla de producción de la gramática y una posición en la parte derecha de la regla de producción. Por ejemplo,* $(E \to E + E, 2)$ *sería un LR(0) item para la gramática de las expresiones.*

  Conjunto de Estados:

  $Q = \{(A \to \alpha, n) : A \to \alpha \in P,\ n \leq |\alpha|\}$

  La notación $\mid \alpha \mid$ *denota la longitud de la cadena* $\mid \alpha \mid$*. En vez de la notación* $(A \to \alpha, n)$ *escribiremos:* $A \to \beta_{\uparrow}\gamma = \alpha$*, donde la flecha ocupa el lugar indicado por el número* $n =\mid \beta \mid$ *:*
- *La función de transición intenta conjeturar que partes derechas de reglas de producción son viables. El conjunto de estados actual del NFA representa el conjunto de pares (regla de producción, posición en la parte derecha) que tienen alguna posibilidad de ser aplicadas de acuerdo con la entrada procesada hasta el momento:*

$$\delta(A \to \alpha_{\uparrow} X\beta, X) = A \to \alpha X_{\uparrow}\beta \ \forall X \in V \cup \Sigma$$
$$\delta(A \to \alpha_{\uparrow} B\beta, \epsilon) = B \to_{\uparrow} \gamma \ \forall B \to \gamma \in P$$

- *Estado de arranque: Se añade la "superregla"* $S' \to S$ *a la gramática* $G = (\Sigma, V, P, S)$*. El LR(0) item* $S' \to_{\uparrow} S$ *es el estado de arranque.*
- *Todos los estados definidos (salvo el de muerte) son de aceptación.*

Denotaremos por $LR(0)$ a este autómata. Sus estados se denominan $LR(0) - items$. La idea es que este autómata nos ayuda a reconocer los prefijos viables PV.

Una vez que se tiene un autómata que reconoce los prefijos viables es posible construir un analizador sintáctico que construye una antiderivación a derechas. La estrategia consiste en "alimentar" el autómata con la forma sentencial derecha. El lugar en el que el autómata se detiene, rechazando indica el lugar exacto en el que termina el *handle* de dicha forma.

Ejemplo 5.4.1. *Consideremos la gramática:*

$$S \to a\ S\ b$$
$$S \to \epsilon$$

El lenguaje generado por esta gramática es $L(G) = \{a^n b^n\ :\ n \geq 0\}$ *Es bien sabido que el lenguaje* $L(G)$ *no es regular. La figura 5.1 muestra el* autómata finito no determinista con ϵ-transiciones *(*NFA*) que reconoce los prefijos viables de esta gramática, construido de acuerdo con el algoritmo 5.4.4.*

Véase https://github.com/crguezl/jison-aSb para una implementación en Jison de una variante de esta gramática.

Ejercicio 5.4.1. *Simule el comportamiento del autómata sobre la entrada aabb. ¿Donde rechaza? ¿En que estados está el autómata en el momento del rechazo?. ¿Qué etiquetas tienen? Haga también las trazas del autómata para las entradas aaSbb y aSb. ¿Que antiderivación ha construido el autómata con sus sucesivos rechazos? ¿Que terminales se puede esperar que hayan en la entrada cuando se produce el rechazo del autómata?*

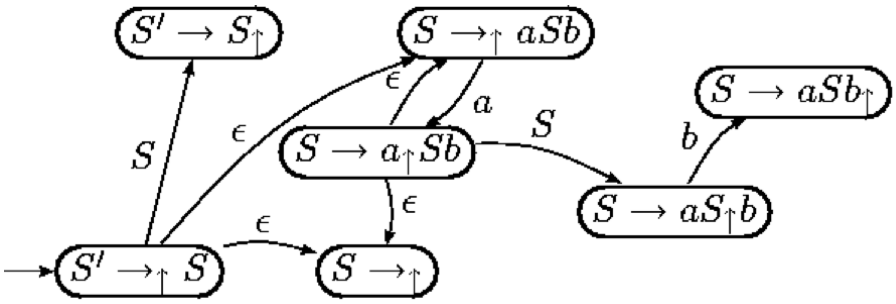


Figura 5.1: NFA que reconoce los prefijos viables

5.5. Construcción de las Tablas para el Análisis SLR

5.5.1. Los conjuntos de Primeros y Siguientes

Repasemos las nociones de conjuntos de *Primeros* y *siguientes*:

Definición 5.5.1. *Dada una gramática $G = (\Sigma, V, P, S)$ y una frase $\alpha \in (V \cup \Sigma)^*$ se define el conjunto $FIRST(\alpha)$ como:*

$$FIRST(\alpha) = \left\{ b \in \Sigma : \alpha \stackrel{*}{\Longrightarrow} b\beta \right\} \cup N(\alpha)$$

donde:

$$N(\alpha) = \begin{cases} \{\epsilon\} & si\ \alpha \stackrel{*}{\Longrightarrow} \epsilon \\ \emptyset & en\ otro\ caso \end{cases}$$

Definición 5.5.2. *Dada una gramática $G = (\Sigma, V, P, S)$ y una variable $A \in V$ se define el conjunto $FOLLOW(A)$ como:*

$$FOLLOW(A) = \left\{ b \in \Sigma : \exists\ S \stackrel{*}{\Longrightarrow} \alpha A b \beta \right\} \cup E(A)$$

donde

$$E(A) = \begin{cases} \{\$\} & si\ S \stackrel{*}{\Longrightarrow} \alpha A \\ \emptyset & en\ otro\ caso \end{cases}$$

Algoritmo 5.5.1. *Construcción de los conjuntos $FIRST(X)$*

1. *Si $X \in \Sigma$ entonces $FIRST(X) = X$*
2. *Si $X \to \epsilon$ entonces $FIRST(X) = FIRST(X) \cup \{\epsilon\}$*
3. *Si $X \in V$ y $X \to Y_1 Y_2 \cdots Y_k \in P$ entonces*

$i = 1;$
do
  $FIRST(X) = FIRST(X) \cup FIRST(Y_i) - \{\epsilon\};$
  $i++;$
mientras $(\epsilon \in FIRST(Y_i)\ and\ (i \leq k))$
si $(\epsilon \in FIRST(Y_k)\ and\ i > k)\ FIRST(X) = FIRST(X) \cup \{\epsilon\}$

Este algoritmo puede ser extendido para calcular $FIRST(\alpha)$ para $\alpha = X_1 X_2 \cdots X_n \in (V \cup \Sigma)^*$.

Algoritmo 5.5.2. *Construcción del conjunto* $FIRST(\alpha)$

$i = 1;$
$FIRST(\alpha) = \emptyset;$
do
  $FIRST(\alpha) = FIRST(\alpha) \cup FIRST(X_i) - \{\epsilon\};$
  $i++;$
mientras $(\epsilon \in FIRST(X_i)$ *and* $(i \leq n))$
si $(\epsilon \in FIRST(X_n)$ *and* $i > n)$ $FIRST(\alpha) = FIRST(X) \cup \{\epsilon\}$

Algoritmo 5.5.3. *Construcción de los conjuntos* $FOLLOW(A)$ *para las variables sintácticas* $A \in V$*:*
Repetir los siguientes pasos hasta que ninguno de los conjuntos $FOLLOW$ *cambie:*

1. $FOLLOW(S) = \{\$\}$ *(\$ representa el final de la entrada)*
2. *Si* $A \to \alpha B \beta$ *entonces*

$$FOLLOW(B) = FOLLOW(B) \cup (FIRST(\beta) - \{\epsilon\})$$

3. *Si* $A \to \alpha B$ *o bien* $A \to \alpha B \beta$ *y* $\epsilon \in FIRST(\beta)$ *entonces*

$$FOLLOW(B) = FOLLOW(B) \cup FOLLOW(A)$$

5.5.2. Construcción de las Tablas

Para la construcción de las tablas de un analizador SLR se construye el *autómata finito determinista* (*DFA*) (Q, Σ, δ, q_0) equivalente al NFA presentado en la sección 5.4 usando el *algoritmo de construcción del subconjunto*.

Como recordará, en la construcción del subconjunto, partiendo del estado de arranque q_0 del NFA con ϵ-transiciones se calcula su *clausura* $\overline{\{q_0\}}$ y las clausuras de los conjuntos de estados $\delta(\overline{\{q_0\}}, a)$ a los que transita. Se repite el proceso con los conjuntos resultantes hasta que no se introducen nuevos conjuntos-estado.

La clausura $\overline{A}$ de un subconjunto de estados del autómata A esta formada por todos los estados que pueden ser alcanzados mediante transiciones etiquetadas con la palabra vacía (denominadas ϵ transiciones) desde los estados de A. Se incluyen en $\overline{A}$, naturalmente los estados de A.

$$\overline{A} = \{q \in Q \ / \ \exists q' \in Q \ : \ \hat{\delta}(q', \epsilon) = q\}$$

Aquí $\hat{\delta}$ denota la *función de transición del autómata* extendida a cadenas de Σ^*.

$$\hat{\delta}(q, x) = \begin{cases} \delta(\hat{\delta}(q, y), a) & \text{si } x = ya \\ q & \text{si } x = \epsilon \end{cases} \tag{5.1}$$

En la práctica, y a partir de ahora así lo haremos, se prescinde de diferenciar entre δ y $\hat{\delta}$ usándose indistintamente la notación δ para ambas funciones.

La clausura puede ser computada usando una estructura de pila o aplicando la expresión recursiva dada en la ecuación 5.1.

Para el NFA mostrado en el ejemplo 5.4.1 el DFA construído mediante esta técnica es el que se muestra en la figura 5.3. Se ha utilizado el símbolo `#` como marcador. Se ha omitido el número 3 para que los estados coincidan en numeración con los generados por `jison` (véase el cuadro **??**).

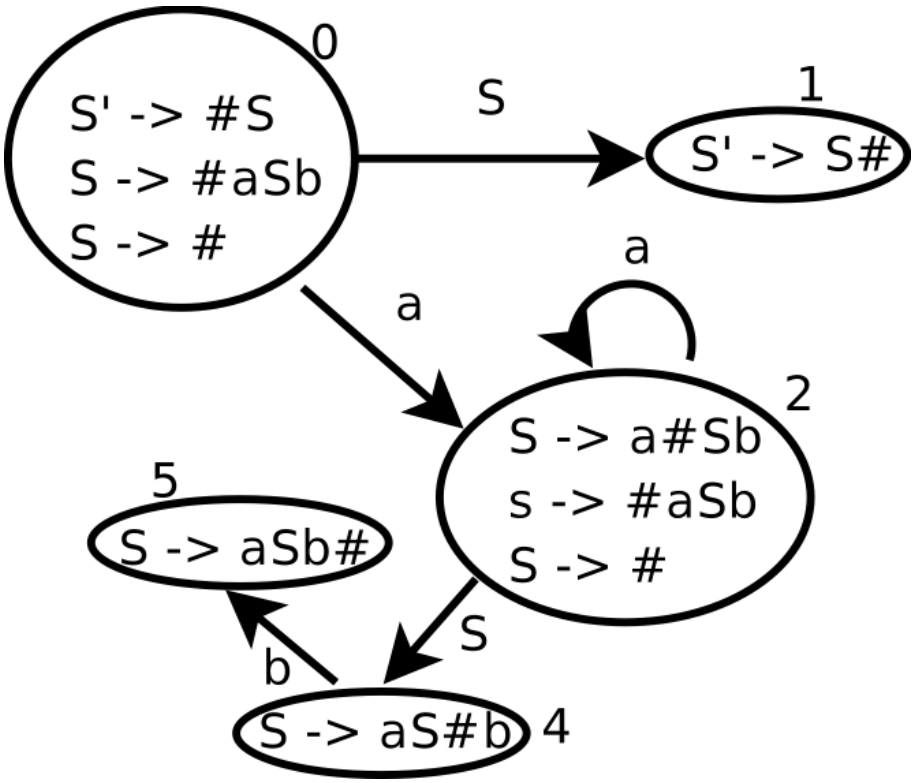


Figura 5.2: DFA equivalente al NFA de la figura 5.1

Un analizador sintáctico LR utiliza una tabla para su análisis. Esa tabla se construye a partir de la tabla de transiciones del DFA. De hecho, la tabla se divide en dos tablas, una llamada *tabla de saltos* o *tabla de gotos* y la otra *tabla de acciones*.

La tabla *goto* de un analizador *SLR* no es más que la tabla de transiciones del autómata DFA obtenido aplicando la construcción del subconjunto al NFA definido en 5.4.4. De hecho es la tabla de transiciones restringida a V (recuerde que el alfabeto del autómata es $V \cup \Sigma$, i denota al i-ésimo estado resultante de aplicar la construcción del subconjunto y que I_i denota al conjunto de LR(0) item asociado con dicho estado):

$$\delta_{|V \times Q} : V \times Q \to Q.$$
donde se define $goto(i, A) = \delta(A, I_i)$

La parte de la función de transiciones del DFA que corresponde a los terminales que no producen rechazo, esto es, $\delta_{|\Sigma \times Q} : \Sigma \times Q \to Q$ se adjunta a una tabla que se denomina *tabla de acciones*. La tabla de acciones es una tabla de doble entrada en los estados y en los símbolos de Σ. Las acciones de transición ante terminales se denominan *acciones de desplazamiento* o (*acciones shift*):

$$\delta_{|\Sigma \times Q} : \Sigma \times Q \to Q$$
donde se define $action(i, a) = shift\ \delta(a, I_i)$

Cuando un estado s contiene un LR(0)-item de la forma $A \to \alpha_{\uparrow}$, esto es, el estado corresponde a un posible rechazo, ello indica que hemos llegado a un final del prefijo viable, que hemos visto α y que, por tanto, es probable que $A \to \alpha$ sea el *handle* de la forma sentencial derecha actual. Por tanto, añadiremos en entradas de la forma (s, a) de la tabla de acciones una acción que indique que hemos encontrado el mango en la posición actual y que la regla asociada es $A \to \alpha$. A una acción de este tipo se la denomina *acción de reducción*.

La cuestión es, ¿para que valores de $a \in \Sigma$ debemos disponer que la acción para (s, a) es de reducción?

Se define $action(i, a) = reduce\ A \to \alpha$ ¿Pero, para que $a \in \Sigma$?

Podríamos decidir que ante cualquier terminal $a \in \Sigma$ que produzca un rechazo del autómata, pero podemos ser un poco mas selectivos. No cualquier terminal puede estar en la entrada en el momento en el que se produce la antiderivación o reducción. Observemos que si $A \to \alpha$ es el *handle* de γ es porque:

$$\exists S \underset{RM}{\overset{*}{\Longrightarrow}} \beta A b x \underset{RM}{\overset{*}{\Longrightarrow}} \beta \alpha b x = \gamma$$

Por tanto, cuando estamos reduciendo por $A \to \alpha$ los únicos terminales legales que cabe esperar en una reducción por $A \to \alpha$ son los terminales $b \in FOLLOW(A)$.

$$\text{Se define } action(i, b) = reduce\ A \to \alpha \text{ Para } b \in FOLLOW(A)$$

Dada una gramática $G = (\Sigma, V, P, S)$, podemos construir las tablas de acciones (*action table*) y transiciones (*gotos table*) mediante el siguiente algoritmo:

Algoritmo 5.5.4. *Construcción de Tablas* SLR

1. *Utilizando el Algoritmo de Construcción del Subconjunto, se construye el Autómata Finito Determinista (DFA)* $(Q, V \cup \Sigma, \delta, I_0, F)$ *equivalente al Autómata Finito No Determinista (NFA) definido en 5.4.4. Sea* $C = \{I_1, I_2, \cdots I_n\}$ *el conjunto de estados del DFA. Cada estado* I_i *es un conjunto de LR(0)-items o estados del NFA. Asociemos un índice i con cada conjunto* I_i*.*

2. *La tabla de* gotos *no es más que la función de transición del autómata restringida a las variables de la gramática:*

$$goto(i, A) = \delta(I_i, A) \text{ para todo } A \in V$$

3. *Las acciones para el estado* I_i *se determinan como sigue:*

   a) *Si* $A \to \alpha_{\uparrow} a \beta \in I_i$, $\delta(I_i, a) = I_j$, $a \in \Sigma$ *entonces:*

   $$action[i][a] = shift\ j$$

   b) *Si* $S' \to S_{\uparrow} \in I_i$ *entonces*

   $$action[i][\$] = accept$$

   c) *Para cualquier otro caso de la forma* $A \to \alpha_{\uparrow} \in I_i$ *distinto del anterior hacer*

   $$\forall a \in\ FOLLOW(A) :\ action[i][a] = reduce\ A \to \alpha$$

4. *Las entradas de la tabla de acción que queden indefinidas después de aplicado el proceso anterior corresponden a acciones de "error".*

Definición 5.5.3. *Si alguna de las entradas de la tabla resulta multievaluada, decimos que existe un conflicto y que la gramática no es* SLR*.*

1. *En tal caso, si una de las acciones es de 'reducción" y la otra es de 'desplazamiento", decimos que hay un* conflicto shift-reduce *o* conflicto de desplazamiento-reducción*.*

2. *Si las dos reglas indican una acción de reducción, decimos que tenemos un* conflicto reduce-reduce *o de* reducción-reducción*.*

Ejemplo 5.5.1. *Al aplicar el algoritmo 5.5.4 a la gramática 5.4.1*

1	$S \rightarrow a\ S\ b$
2	$S \rightarrow \epsilon$

partiendo del autómata finito determinista que se construyó en la figura 5.3 y calculando los conjuntos de primeros y siguientes

	FIRST	*FOLLOW*
S	*a*, ϵ	*b*, *$*

obtenemos la siguiente tabla de acciones SLR:

	a	*b*	*$*
0	*s2*	*r2*	*r2*
1			*aceptar*
2	*s2*	*r2*	*r2*
4		*s5*	
5		*r1*	*r1*

Las entradas denotadas con s n (s por shift) indican un desplazamiento al estado n, las denotadas con r n (r por reduce o reducción) indican una operación de reducción o antiderivación por la regla n. Las entradas vacías corresponden a acciones de error.

El método de análisis *LALR* usado por `jison` es una extensión del método SLR esbozado aqui. Supone un compromiso entre potencia (conjunto de gramáticas englobadas) y eficiencia (cantidad de memoria utilizada, tiempo de proceso). Veamos como `jison` aplica la construcción del subconjunto a la gramática del ejemplo 5.4.1. Para ello construimos el siguiente programa `jison`:

```
[~/srcPLgrado/aSb(develop)]$ cat -n aSb.jison
     1  %lex
     2  %%
     3  .                { return yytext; }
     4  /lex
     5  %%
     6  P: S             { return $1; }
     7  ;
     8  S: /* empty */   { console.log("empty");    $$ = ''; }
     9     | 'a' S 'b'   { console.log("S -> aSb"); $$ = $1+$2+$3; }
    10  ;
    11  %%
```

y lo compilamos con `jison`. Estas son las opciones disponibles:

```
nereida:[~/PLgradoBOOK(eps)]$ jison --help

Usage: jison [file] [lexfile] [options]

file        file containing a grammar
lexfile     file containing a lexical grammar

Options:
   -o FILE, --outfile FILE       Filename and base module name of the generated parser
   -t, --debug                   Debug mode
   -t TYPE, --module-type TYPE   The type of module to generate (commonjs, amd, js)
   -V, --version                 print version and exit
```

Desafortunadamente carece de la típica opción `-v` que permite generar las tablas de análisis. Podemos intentar usar `bison`, pero, obviamente, `bison` protesta ante la entrada:

```
[~/srcPLgrado/aSb(develop)]$ bison -v aSb.jison
aSb.jison:1.1-4: invalid directive: ‘%lex’
aSb.jison:3.1: syntax error, unexpected identifier
aSb.jison:4.1: invalid character: ‘/’
```

El error es causado por la presencia del analizador léxico empotrado en el fichero `aSb.jison`. Si suprimimos provisionalmente las líneas del analizador léxico empotrado, `bison` es capaz de analizar la gramática:

```
[~/srcPLgrado/aSb(develop)]$ bison -v aSb.jison
[~/srcPLgrado/aSb(develop)]$ ls -ltr | tail -1
-rw-rw-r--  1 casiano  staff    926 19 mar 13:29 aSb.output
```

Que tiene los siguientes contenidos:

```
[~/srcPLgrado/aSb(develop)]$ cat -n aSb.output
     1  Grammar
     2
     3      0 $accept: P $end
     4
     5      1 P: S
     6
     7      2 S: /* empty */
     8      3  | 'a' S 'b'
     9
    10
    11  Terminals, with rules where they appear
    12
    13  $end (0) 0
    14  'a' (97) 3
    15  'b' (98) 3
    16  error (256)
    17
    18
    19  Nonterminals, with rules where they appear
    20
    21  $accept (5)
    22      on left: 0
    23  P (6)
    24      on left: 1, on right: 0
    25  S (7)
    26      on left: 2 3, on right: 1 3
    27
    28
    29  state 0
    30
    31      0 $accept: . P $end
    32
    33      'a'  shift, and go to state 1
    34
    35      $default  reduce using rule 2 (S)
    36
    37      P  go to state 2
    38      S  go to state 3
    39
```

```
40
41  state 1
42
43      3 S: 'a' . S 'b'
44
45      'a'  shift, and go to state 1
46
47      $default  reduce using rule 2 (S)
48
49      S  go to state 4
50
51
52  state 2
53
54      0 $accept: P . $end
55
56      $end  shift, and go to state 5
57
58
59  state 3
60
61      1 P: S .
62
63      $default  reduce using rule 1 (P)
64
65
66  state 4
67
68      3 S: 'a' S . 'b'
69
70      'b'  shift, and go to state 6
71
72
73  state 5
74
75      0 $accept: P $end .
76
77      $default  accept
78
79
80  state 6
81
82      3 S: 'a' S 'b' .
83
84      $default  reduce using rule 3 (S)
```

Observe que el final de la entrada se denota por `$end` y el marcador en un LR-item por un punto. Fíjese en el estado 1: En ese estado están también los items

`S -> . 'a' S 'b'` y `S -> .`

sin embargo no se explicitan por que se entiende que su pertenencia es consecuencia directa de aplicar la operación de clausura. Los LR items cuyo marcador no está al principio se denominan *items núcleo*.

5.6. Práctica: Analizador de PL0 Usando Jison

Reescriba el analizador sintáctico del lenguaje PL0 realizado en las prácticas 2.6 y 4.13 usando Jison .

Donde

- Repositorio en GitHub
- Despliegue en Heroku
- `[~/jison/jisoncalc(develop)]$ pwd -P`
  `/Users/casiano/local/src/javascript/PLgrado/jison/jisoncalc`
- `[~/jison/jisoncalc(develop)]$ git remote -v`
  ```
  heroku  git@heroku.com:jisoncalc.git (fetch)
  heroku  git@heroku.com:jisoncalc.git (push)
  origin  git@github.com:crguezl/ull-etsii-grado-pl-jisoncalc.git (fetch)
  origin  git@github.com:crguezl/ull-etsii-grado-pl-jisoncalc.git (push)
  ```

Tareas

- La salida debe ser el AST del programa de entrada
- Modifique `block` y `statement` para que los `procedure` reciban argumentos y las llamadas a procedimiento puedan pasar argumentos.
- Añada `if ... then ... else ...`.
- Actualice la documentación de la gramática para que refleje la gramática ampliada
- Limite el número de programas que se pueden salvar a un número prefijado, por ejemplo 10. Si se intenta salvar uno se suprime uno al azar y se guarda el nuevo.
- Las pruebas deben comprobar que los AST generados reflejan la semántica del lenguaje así como alguna situación de error
- Sólo usuarios autenticados pueden salvar sus programas en la base de datos.
- Extienda la autenticación OAuth para que además de Google pueda hacerse con Twitter ó GitHub ó Facebook ó ... Sólo debe implementar una.
- Método de Entrega:
  - Use un repositorio privado en BitBucket o bien solicite al administrador del Centro de Cálculo un repositorio privado en GitHub.
  - Comparta dicho repositorio con sus colaboradores y con el profesor.
  - Suba la práctica al workshop/taller antes de la fecha límite
  - Cuando el taller pase a la fase de evaluación haga público su repositorio

Referencias para esta Práctica

- Véase el capítulo *Oauth: Google, Twitter, GitHub, Facebook* **??**
- Véase Intridea Omniauth y omniauth en GitHub
- La gema omniauth-google-oauth2
- Google Developers Console

- Revoking Access to an App in Google
- La gema sinatra-flash
- Véase el capítulo *Heroku* **??**
- Heroku Postgres
- Véase el capítulo *DataMapper* **??**

5.7. Práctica: Análisis de Ámbito en PL0

Objetivos

- Modifique la práctica anterior para que cada nodo del tipo `PROCEDURE` disponga de una tabla de símbolos en la que se almacenan todos las constantes, variables y procedimientos declarados en el mismo.
- Existirá ademas una tabla de símbolos asociada con el nodo raíz que representa al programa principal.
- Las declaraciones de constantes y variables no crean nodo, sino que se incorporan como información a la tabla de símbolos del procedimiento actual
- Para una entrada de la tabla de símbolos `sym["a"]` se guarda que clase de objeto es: constante, variable, procedimiento, etc.
- Si es un procedimiento se guarda el número de argumentos
- Si es una constante se guarda su valor
- Cada uso de un identificador (constante, variable, procedimiento) tiene un atributo `declared_in` que referencia en que nodo se declaró
- Si un identificador es usado y no fué declarado es un error
- Si se trata de una llamada a procedimiento (se ha usado `CALL` y el identificador corresponde a un `PROCEDURE`) se comprobará que el número de argumentos coincide con el número de parámetros declarados en su definición
- Si es un identificador de una constante, es un error que sea usado en la parte izquierda de una asignación (que no sea la de su declaración)
- Base de Datos
  1. Guarde en una tabla el nombre de usuario que guardó un programa. Provea una ruta para ver los programas de un usuario.
  2. Un programa `belongs_to` un usuario. Un usuario `has n` programas. Vea la sección DataMapper Associat
- Use la sección `issues` de su repositorio en GitHub para coordinarse así como para llevar un histórico de las incidencias y la forma en la que se resolvieron. Repase el tutorial Mastering Issues

5.8. Práctica: Traducción de Infijo a Postfijo

Modifique el programa Jison realizado en la práctica 5.3.1 para traducir de infijo a postfijo. Añada los operadores de comparación e igualdad. Por ejemplo

Infijo	Postfijo
`a = 3+2*4`	`3 2 4 * + &a =`
`b = a == 11`	`a 11 == &b =`

En estas traducciones la notación `&a` indica la dirección de la variable `a` y `a` indica el valor almacenado en la variable `a`.

Añada sentencias `if` ... `then` e `if` ... `then` ... `else`

Para realizar la traducción de estas sentencias añada instrucciones `jmp label` y `jmpz label` (por *jump if zero*) y etiquetas:

Infijo	Postfijo

`a = (2+5)*3;`
`if a == 0 then b = 5 else b = 3;`
`c = b + 1;`
	`2`
	`5`
	`+`
	`3`
	`*`
	`&a`
	`=`
	`a`
	`0`
	`==`
	`jmpz else1`
	`5`
	`&b`
	`=`
	`jmp endif0`
	`:else1`
	`3`
	`&b`
	`=`
	`:endif0`
	`b`
	`1`
	`+`
	`&c`
	`=`

Parta del repositorio https://github.com/crguezl/jison-simple-html-calc.

5.9. Práctica: Calculadora con Funciones

Añada funciones y sentencias de llamada a función a la práctica de traducción de infijo a postfijo 5.8. Sigue un ejemplo de traducción:

```
def f(x) { x + 1 }
def g(a, b) { a * f(b) }
c = 3;
f(1+c);
g(3, 4)
```

```
:f      args :x
        $x
        1
        +
        return
:g      args :a,:b
        $a
        $b
        call :f
        *
        return
:main:
        3
        &c
        =
        1
        c
        +
        call :f
        3
        4
        call :g
```

- Las funciones retornan la última expresión evaluada
- Es un error llamar a una función con un número de argumentos distinto que el número de parámetros con el que fué declarada
- En la llamada, los argumentos se empujan en la pila. Después la instrucción `call :etiqueta` llama a la función con el nombre dado por la `etiqueta`
- Dentro de la función los argumentos se sitúan por encima del puntero base. La pseudo-instrucción `args, p1, p2, ...` da nombre a los parámetros empujados. Dentro del cuerpo de la función nos referimos a ellos prefijándolos con `$`.
- La instrucción `return` limpia la pila dejándola en su estado anterior y retorna la última expresión evaluada

5.10. Práctica: Calculadora con Análisis de Ámbito

Extienda la práctica anterior para que haga un análisis completo del ámbito de las variables.

- Añada declaraciones de variable con `var x, y = 1, z`. Las variables podrán opcionalmente ser inicializadas. Se considerará un error usar una variable no declarada.
- Modifique la gramática para que permita el anidamiento de funciones: funciones dentro de funciones.

```
var c = 4, d = 1, e;
def g(a, b) {
  var d, e;
  def f(u, v) { a + u + v + d }
  a * f(b, 2) + d + c
}
```

- Una declaración de variable en un ámbito anidado tapa a una declaración con el mismo nombre en el ámbito exterior.

```
var c = 4, d = 1, e;
def g(a, b) {
  var d, e; # esta "d" tapa la d anterior
  def f(u, v) { a + u + v + d }
  a * f(b, 2) + d + c
}
```

```
        # global:       var c,d,e
:g.f
        $a, 1
        $u, 0
        +
        $v, 0
        +
        d, 1
        +
        return
:g
        $a, 0
        $b, 0
        2
        call :g.f
        *
        d, 0     # acceder a la d en el ámbito actual
        +
        c, 1
        +
        return
```

- Los nombres de funciones se traducen por una secuencia anidada de nombres que indican su ámbito. Así la función `f` anidada en `g` es traducida a la función con nombre `g.f`. Una función `h` anidada en una función `f` anidada en `g` es traducida a la función con nombre `g.f.h`

- Las variables ademas de su nombre (dirección/offset) reciben un entero adicional 0,1,2, ... que indica su nivel de anidamiento. El número de stack frames que hay que recorrer para llegar a la variable

```
$a, 1
$u, 0
+
$v, 0
+
d, 1
+
```

  Asi `$a, 1` significa acceder al parámetro `a` que está a distancia `1` del stack frame/ámbito actual y `$v, 0` es el parámetro `v` en el ámbito/stack frame actual

- El frame pointer o base pointer BP indica el nivel de anidamiento estático (en el fuente) de la rutina. Así cuando se va a buscar una variable local declarada en la rutina que anida la actual se recorre la lista de frames via BP o frame pointer tantas veces como el nivel de anidamiento indique.

- 1. Esto es lo que dice la Wikipedia sobre la implementación de llamadas a subrutinas anidadas:

  > *Programming languages that support nested subroutines also have a field in the call frame that points to the stack frame of the latest activation of the procedure that most closely encapsulates the callee, i.e. the immediate scope of the callee. This is*

called an access link *or* static link *(as it keeps track of static nesting during dynamic and recursive calls) and provides the routine (as well as any other routines it may invoke) access to the local data of its encapsulating routines at every nesting level.*

2. Esto es lo que dice sobre las ventajas de tener una pila y de almacenar la dirección de retorno y las variables locales:

   When a subroutine is called, the location (address) of the instruction at which it can later resume needs to be saved somewhere. Using a stack to save the return address has important advantages over alternatives. One is that each task has its own stack, and thus the subroutine can be *reentrant*, that is, can be active simultaneously for different tasks doing different things. Another benefit is that *recursion* is automatically supported. When a function calls itself recursively, a return address needs to be stored for each activation of the function so that it can later be used to return from the function activation. This capability is automatic with a stack.

3. Almacenamiento local:

   A subroutine frequently needs memory space for storing the values of local variables, the variables that are known only within the active subroutine and do not retain values after it returns. It is often convenient to allocate space for this use by simply moving the top of the stack by enough to provide the space. This is very fast compared to heap allocation. Note that each separate activation of a subroutine gets its own separate space in the stack for locals.

4. Parámetros:

   Subroutines often require that values for parameters be supplied to them by the code which calls them, and it is not uncommon that space for these parameters may be laid out in the call stack.
   The call stack works well as a place for these parameters, especially since each call to a subroutine, which will have differing values for parameters, will be given separate space on the call stack for those values.

5. Pila de Evaluación

   Operands for arithmetic or logical operations are most often placed into registers and operated on there. However, in some situations the operands may be stacked up to an arbitrary depth, which means something more than registers must be used (this is the case of *register spilling*). The stack of such operands, rather like that in an RPN calculator, is called an *evaluation stack*, and may occupy space in the call stack.

6. Puntero a la instancia actual

   Some object-oriented languages (e.g., C++), store the `this` pointer along with function arguments in the call stack when invoking methods. The `this pointer` points to the object instance associated with the method to be invoked.

- Los parámetros se siguen prefijando de `$` como en la práctica anterior

- Sigue un ejemplo de traducción:

```
var c = 4, d = 1, e;
def f(x) {
  var y = 1;
  x + y
}
def g(a, b) {
  var d, e;
  def f(u, v) { a + u + v + d }
  a * f(b, 2) + d + c
}
c = 3;
f(1+c);
g(3, 4)
```

```
        # global:       var c,
        # f: args x
        # f:    var y
:f
        1
        &y, 0
        =
        $x, 0
        y, 0
        +
        return
        # g: args a,b
        # g:    var d,e
        # g.f: args u,v
:g.f
        $a, 1
        $u, 0
        +
        $v, 0
        +
        d, 1
        +
        return
:g
        $a, 0
        $b, 0
        2
        call :g.f
        *
        d, 0
        +
        c, 1
        +
        return
:main:
        4
        &c, 0
        =
        1
        &d, 0
        =
        3
        &c, 0
        =
        1
        c, 0
        +
        call :f
        3
        4
        call :g
```

- Puede comenzar haciendo un fork del proyecto ull-etsii-grado-pl-infix2postfix en GitHub. Esta incompleto. Rellene las acciones semánticas que faltan; la mayoría relacionadas con el análisis de ámbito.
-
  - Una solución completa se encuentra en el proyecto crguezl/jisoninfix2postfix.
  - `[~/jison/jisoninfix2postfix(gh-pages)]$ pwd -P`
    `/Users/casiano/local/src/javascript/PLgrado/jison/jisoninfix2postfix`
  - `[~/jison/jisoninfix2postfix(gh-pages)]$ git remote -v`
    ```
    bitbucket       ssh://git@bitbucket.org/casiano/jisoninfix2postfix.git (fetch)
    bitbucket       ssh://git@bitbucket.org/casiano/jisoninfix2postfix.git (push)
    origin  git@github.com:crguezl/jisoninfix2postfix.git (fetch)
    origin  git@github.com:crguezl/jisoninfix2postfix.git (push)
    ```
- Veanse:
  - Véase COMP 3290 Compiler Construction Fall 2008 Notes/Symbol Tables
  - El capítulo Symbol Table Structure del libro de Muchnick Advanced Compiler Design Implementation [5]
  - El capítulo Symbol Table Structure del libro de Basics of Compiler Design de Torben Ægidius Mogensen [6]

5.11. Algoritmo de Análisis LR

Asi pues la tabla de transiciones del autómata nos genera dos tablas: la tabla de acciones y la de saltos. El algoritmo de análisis sintáctico *LR* en el que se basa *jison* utiliza una pila y dos tablas para analizar la entrada. Como se ha visto, la tabla de acciones contiene cuatro tipo de acciones:

1. Desplazar (*shift*)
2. Reducir (*reduce*)
3. Aceptar
4. Error

El algoritmo utiliza una pila en la que se guardan los estados del autómata. De este modo se evita tener que "comenzar" el procesado de la forma sentencial derecha resultante después de una reducción (antiderivación).

Algoritmo 5.11.1. *Análizador LR*

```
push(s0);
b = yylex();
for( ; ; ;) {
  s = top(0); a = b;
  switch (action[s][a]) {
    case "shift t" :
      t.attr = a.attr;
      push(t);
      b = yylex();
      break;
    case "reduce A ->alpha" :
      eval(Sem{A -> alpha}(top(|alpha|-1).attr, ... , top(0).attr));
      pop(|alpha|);
      push(goto[top(0)][A]);
      break;
```

```
    case "accept" : return (1);
    default : yyerror("syntax error");
  }
}
```

- Como es habitual, $|x|$ denota la longitud de la cadena x.
- La función `top(k)` devuelve el elemento que ocupa la posición `k` desde el *top* de la pila (esto es, está a profundidad `k`).
- La función `pop(k)` extrae `k` elementos de la pila.
- La notación `state.attr` hace referencia al atributo asociado con cada estado, el cual desde el punto de vista del programador esta asociado con el correspondiente símbolo de la parte derecha de la regla. Nótese que cada estado que está en la pila es el resultado de una transición con un símbolo. El atributo de ese símbolo es guardado en el objeto estado cada vez que ocurre una transición.
- Denotamos por `Sem {reduce A -> alpha}` el código de la acción semántica asociada con la regla $A \to \alpha$.

Todos los analizadores LR comparten, salvo pequeñas excepciones, el mismo algoritmo de análisis. Lo que más los diferencia es la forma en la que construyen las tablas. En `jison` la construcción de las tablas de *acciones* y *gotos* se realiza por defecto mediante el algoritmo *LALR*.

5.12. El módulo Generado por `jison`

5.12.1. Version

En esta sección estudiamos el analizador generado por Jison:

```
[~/Dropbox/src/javascript/PLgrado/jison-aSb(develop)]$ jison --version
0.4.2
```

5.12.2. Gramática Inicial

Veamos el módulo generado por jison para esta gramática:

```
[~/srcPLgrado/aSb(develop)]$ cat aSb.jison
%lex
%%
.               { return yytext; }
/lex
%%
S: /* empty */  { console.log("empty"); }
   | 'a' S 'b'  { console.log("S -> aSb"); }
;
%%
```

5.12.3. Tablas

Esta es la primera parte del parser generado:

```
/* parser generated by jison 0.4.2 */
var aSb = (function() {
    var parser = {
        trace: function trace() {},
```

```
        yy: {},
        symbols_: {
            "$accept": 0, /* super-arranque $accept -> S */
            "$end": 1     /* end of input */
            "error": 2, /* numero para el símbolo 'error' */
            "S": 3,     /* numero para el símbolo 'S' */
            "a": 4,
            "b": 5,
        },
        /* array inverso de terminales */
        terminals_: {   /* numero -> terminal */
            2: "error",
            4: "a",
            5: "b"
        },
        productions_:
        [0,
/* 1 */     [3, 0], /* S : vacio        simbolo,longitud de la parte derecha */
/* 2 */     [3, 3]  /* S : a S b        simbolo,longitud */
        ],
```

5.12.4. Acciones Semánticas

Cada vez que se produce una acción de reducción esta función es llamada:

```
performAction: function anonymous(yytext, yyleng, yylineno, yy, yystate, $$, _$) {

    var $0 = $$.length - 1;
    switch (yystate) { /* yystate: numero de regla de producción */
        case 1:
            console.log("empty");
            break;
        case 2:
            console.log("S -> aSb");
            break;
    }
},
```

- Parece que cuando se llama a este método `this` refiere a un objeto `yyval`. Este es el punto de llamada a la acción semántica dentro del parser generado por Jison. Puede encontrarse dentro del parser en el caso de un `switch` que corresponde a la acción de reducción:

  ```
  r = this.performAction.call(yyval, yytext, yyleng, yylineno, this.yy, action[1], vstack,
  ```

  El método `call` nos permite invocar una función como si fuera un método de algún otro objeto. Véase la sección **??**.

  Este objeto `yyval` tiene dos atributos: `$` y `_$`.

  - El atributo `$` se corresponde con `$$` de la gramática (atributo de la variable sintactica en la parte izquierda)
  - El atributo `_$` guarda información sobre la posición del último token leído.

- `yytext` parece contener el texto asociado con el token actual

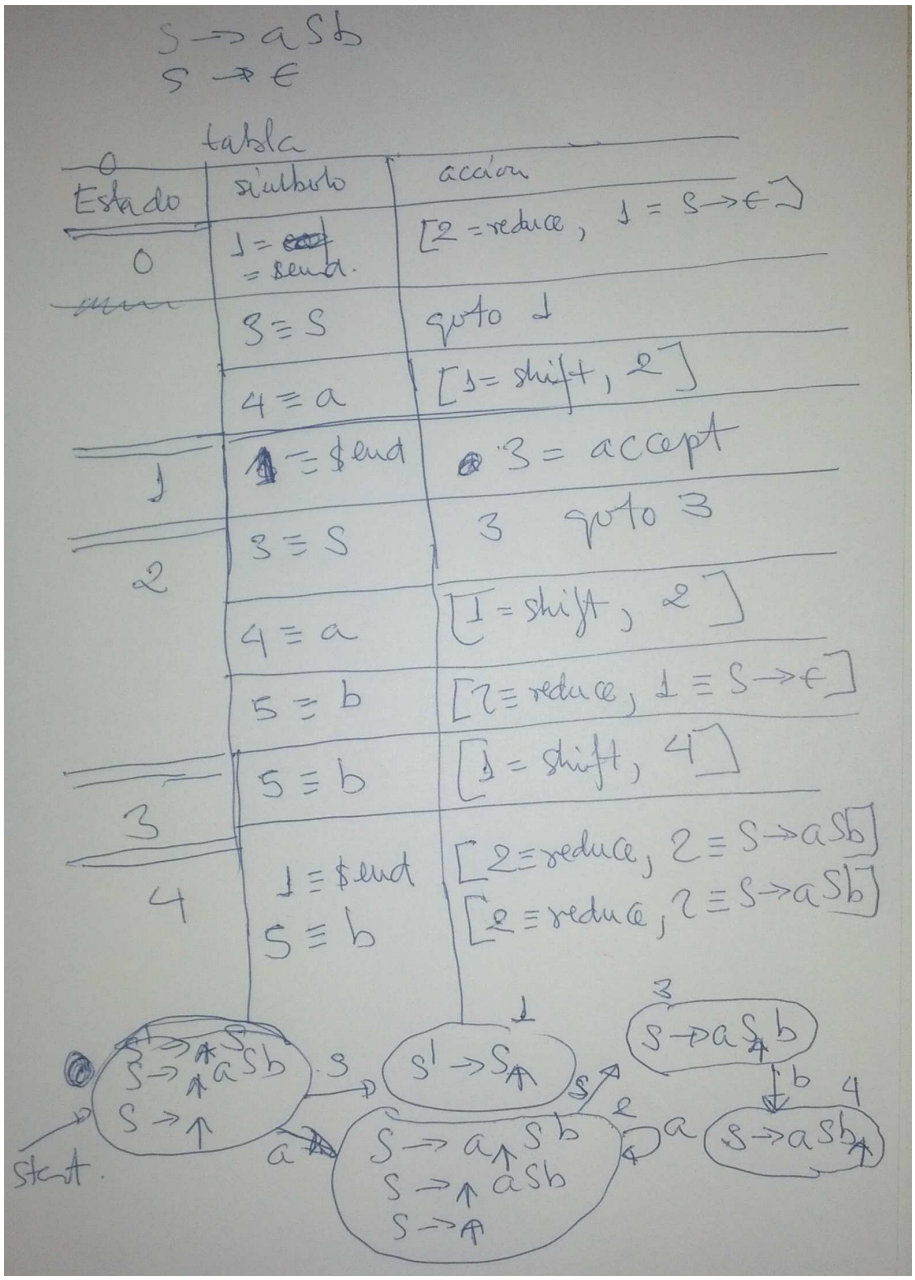


Figura 5.3: DFA construido por Jison

- `yyleng` es la longitud del token actual
- `yylineno` es la línea actual (empezando en 0)
- `yy` es un objeto con dos atributos `lexer` y `parser`
- `yystate` es el estado actual
- `$$` parece ser un array/pila conteniendo los valores de los atributos asociados con los estados de la pila (`vstack` ¿Por value stack?)
- Asi pues `$0` es el índice en `$0` del último elemento de `$$`. Por ejemplo, una acción semántica asociada con una regla `A : B C D` con tres elementos como:

  ```
  $$ = $1 + $2 + $3;
  ```

  Se traduce por:

```
this.$ = $$[$0 - 2] + $$[$0 - 1] + $$[$0];
```

- `_$` Es un array con la información sobre la localización de los simbolos (`lstack` ¿Por location stack?)

5.12.5. Tabla de Acciones y GOTOs

```
        table: [{
/* 0 */     1: [2, 1],    /* En estado 0 viendo $end(1) reducir por S : vacio */
            3: 1,         /* En el estado 0 viendo S(3) ir al estado 1 */
            4: [1, 2]     /* Estado 0 viendo a(4) shift(1) al estado 2 */
        }, {
/* 1 */     1: [3]        /* En 1 viendo $end(1) aceptar */
        }, {
/* 2 */     3: 3,         /* En 2 viendo S ir a 3 */
            4: [1, 2],    /* En 2 viendo a(4) shift a 2 */
            5: [2, 1]     /* En 2 viendo b(5) reducir por regla 1: S -> vacio */
        }, {
/* 3 */     5: [1, 4]     /* En 3 viendo b(5) shift a 4 */
        }, {
/* 4 */     1: [2, 2],    /* En 4  viendo $end(1) reducir(2) por la 2: S -> aSb */
            5: [2, 2]     /* En 4 viendo b(5) reducir por la 2: S-> aSb */
        }],
```

- La tabla es un array de objetos
- El índice de la tabla es el estado. En el ejemplo tenemos 5 estados
- El objeto/hash que es el valor contiene las acciones ante los símbolos.
  1. Los atributos/claves son los símbolos, los valores las acciones
  2. Las acciones son de dos tipos:
     a) El número del estado al que se transita mediante la tabla `goto` cuando el símbolo es una variable sintactica
     b) Un par `[tipo de acción, estado o regla]`. Si el `tipo de acción` es 1 indica un shift al `estado` con ese número. Si el `tipo de acción` es 2 indica una reducción por la `regla` con ese número.
  3. Por ejemplo `table[0]` es

```
{
    1: [2, 1],    /* En estado 0 viendo $end(1) reducir(2) por S : vacio */
    3: 1,         /* En el estado 0 viendo S(3) ir (goto) al estado 1 */
    4: [1, 2]     /* Estado 0 viendo a(4) shift(1) al estado 2 */
}
```

5.12.6. defaultActions

```
defaultActions: {},
```

- `defaultActions` contiene las acciones por defecto.
- Después de la construcción de la tabla, Jison identifica para cada estado la reducción que tiene el conjunto de lookaheads mas grande. Para reducir el tamaño del parser, Jison puede decidir suprimir dicho conjunto y asiganr esa reducción como acción del parser por defecto. Tal reducción se conoce como *reducción por defecto*.

- Esto puede verse en este segmento del código del parser:

```
while (true) {
      state = stack[stack.length - 1];
      if (this.defaultActions[state]) {
          action = this.defaultActions[state];
      } else {
          if (symbol === null || typeof symbol == "undefined") {
              symbol = lex();
          }
          action = table[state] && table[state][symbol];
      }
      ...
}
```

5.12.7. Reducciones

```
parse: function parse(input) {
    ...
    while (true) {
        state = stack[stack.length - 1];
        if (this.defaultActions[state]) {
            action = this.defaultActions[state];
        } else {
            if (symbol === null || typeof symbol == "undefined") {
                symbol = lex(); /* obtener siguiente token */
            }
            action = table[state] && table[state][symbol];
        }
        if (typeof action === "undefined" || !action.length || !action[0]) {
          ... // error
        }
        if (action[0] instanceof Array && action.length > 1) {
            throw new Error("Parse Error: multiple actions possible at state: ..."
        }
        switch (action[0]) {
            case 1:                                     // shift
                ...
                break;
            case 2:                                     // reduce
                len = this.productions_[action[1]][1]; // longitud de la producción
                yyval.$ = vstack[vstack.length - len];
                yyval._$ = {                            // datos de la posición
                    first_line: lstack[lstack.length - (len || 1)].first_line,
                    last_line: lstack[lstack.length - 1].last_line,
                    first_column: lstack[lstack.length - (len || 1)].first_column,
                    last_column: lstack[lstack.length - 1].last_column
                };
                ...
                r = this.performAction.call(yyval, yytext, yyleng, yylineno, this.yy, action[1
                if (typeof r !== "undefined") {
                    return r; /* un return de algo distinto de undefined nos saca del parser *
                }
                if (len) {                                  /* retirar de las pilas */
```

```
                stack = stack.slice(0, - 1 * len * 2);  /* simbolo, estado, simbolo, estad
                vstack = vstack.slice(0, - 1 * len);    /* retirar atributos */
                lstack = lstack.slice(0, - 1 * len);    /* retirar localizaciones */
            }
            stack.push(this.productions_[action[1]][0]); /* empujemos el símbolo */
            vstack.push(yyval.$);                        /* empujemos valor semantico */
            lstack.push(yyval._$);                       /* empujemos localización */
            newState = table[stack[stack.length - 2]][stack[stack.length - 1]];
            stack.push(newState);                        /* empujemos goto[top][A]*/
            break;
        case 3: // accept
            return true;
    }
  }
  return true;
}
```

5.12.8. Desplazamientos/Shifts

```
parse: function parse(input) {
    ...
    while (true) {
        state = stack[stack.length - 1];     /* estado en el top de la pila */
        if (this.defaultActions[state]) {    /* definida la acción por defecto? */
            action = this.defaultActions[state];
        } else {
            if (symbol === null || typeof symbol == "undefined") {
                symbol = lex();              /* obtener token */
            }
            action = table[state] && table[state][symbol]; /* obtener la acción para el estado
        }
        if (typeof action === "undefined" || !action.length || !action[0]) {
            ... /* error */
        }
        if (action[0] instanceof Array && action.length > 1) {
            throw new Error("Parse Error: multiple actions possible at state: " + state + ", t
        }
        switch (action[0]) {
            case 1:
                stack.push(symbol);                /* empujamos token */
                vstack.push(this.lexer.yytext);    /* empujamos el atributo del token */
                lstack.push(this.lexer.yylloc);    /* salvamos la localización del token */
                stack.push(action[1]);             /* salvamos el estado */
                symbol = null;
                if (!preErrorSymbol) {             /* si no hay errores ... */
                    yyleng = this.lexer.yyleng;    /* actualizamos los atributos */
                    yytext = this.lexer.yytext;    /* del objeto */
                    yylineno = this.lexer.yylineno;
                    yyloc = this.lexer.yylloc;
                    if (recovering > 0) recovering--; /* las cosas van mejor si hubieron error
                } else {
                    symbol = preErrorSymbol;
                    preErrorSymbol = null;
                }
```

```
                break;
            case 2:
                ...
                break;
            case 3:
                return true;
        }
    }
    return true;
}
```

5.12.9. Manejo de Errores

```
while (true) {
    state = stack[stack.length - 1];
    if (this.defaultActions[state]) { action = this.defaultActions[state]; }
    else {
        if (symbol === null || typeof symbol == "undefined") { symbol = lex(); }
        action = table[state] && table[state][symbol];
    }
    if (typeof action === "undefined" || !action.length || !action[0]) {
        var errStr = "";
        if (!recovering) { /* recovering = en estado de recuperación de un error */
            expected = [];                      /* computemos los tokens esperados */
            for (p in table[state])             /* si el estado "state" transita con p */
              if (this.terminals_[p] && p > 2) { /* y "p" es un terminal no especial */
                  expected.push("'" + this.terminals_[p] + "'"); /* entonces es esperado */
              }
            if (this.lexer.showPosition) { /* si esta definida la función showPosition */
                errStr = "Parse error on line " + (yylineno + 1) +
                         ":\n" + this.lexer.showPosition() +
                         "\nExpecting " + expected.join(", ") +
                         ", got '" +
                         (this.terminals_[symbol] || symbol) + /* terminals_ es el array inver
                         "'";                                   /* numero -> terminal
            } else { /* ¡monta la cadena como puedas! */
                errStr = "Parse error on line " + (yylineno + 1) +
                         ": Unexpected " +
                         (symbol == 1 ? "end of input" : "'" +
                         (this.terminals_[symbol] || symbol) + "'");
            }
            this.parseError(errStr, {    /* genera la excepción */
                text: this.lexer.match,  /* hash/objeto conteniendo los detalles del */
                token: this.terminals_[symbol] || symbol,                   /* error */
                line: this.lexer.yylineno,
                loc: yyloc,
                expected: expected
            });
        }
    }
    if (action[0] instanceof Array && action.length > 1) {
        throw new Error("Parse Error: multiple actions possible at state: " + state + ", token
    }
    ...
```

```
}
```

La función `parseError` genera una excepción:

```
parseError: function parseError(str, hash) {
    throw new Error(str); /* El hash contiene info sobre el error: token, linea, etc.
},
```

- `parseError` es llamada cada vez que ocurre un error sintáctico. `str` contiene la cadena con el mensaje de error del tipo: `Expecting something, got other thing'`. `hash` contiene atributos como `expected`: el array de tokens esperados; `line` la línea implicada, `loc` una descripción de la localización detallada del punto/terminal en el que ocurre el error; etc.

5.12.10. Analizador Léxico

El analizador léxico:

```
/* generated by jison-lex 0.1.0 */
var lexer = (function() {
    var lexer = {
        EOF: 1,
        parseError: function parseError(str, hash) { /* manejo de errores léxicos */ },
        setInput: function(input) { /* inicializar la entrada para el analizadorléxico */},
        input: function() { /* ... */ },
        unput: function(ch) { /* devolver al flujo de entrada */ },
        more: function() { /* ... */ },
        less: function(n) { /* ... */ },
        pastInput: function() { /* ... */ },
        upcomingInput: function() { /* ... */ },
        showPosition: function() { /* ... */ },
        next: function() {
                if (this.done) { return this.EOF; }
                if (!this._input) this.done = true;

                var token, match, tempMatch, index, col, lines;
                if (!this._more) { this.yytext = ''; this.match = ''; }
                var rules = this._currentRules();
                for (var i = 0; i < rules.length; i++) {
                    tempMatch = this._input.match(this.rules[rules[i]]);
                    if (tempMatch && (!match || tempMatch[0].length > match[0].length)) {
                        match = tempMatch;
                        index = i;
                        if (!this.options.flex) break;
                    }
                }
                if (match) {
                    lines = match[0].match(/(?:\r\n?|\n).*/g);
                    if (lines) this.yylineno += lines.length;
                    this.yylloc = {
                        first_line: this.yylloc.last_line,
                        last_line: this.yylineno + 1,
                        first_column: this.yylloc.last_column,
                        last_column:
                          lines ? lines[lines.length - 1].length -
                                  lines[lines.length - 1].match(/\r?\n?/)[0].length
```

```
                        :
                        this.yylloc.last_column + match[0].length
            };
            this.yytext += match[0];
            this.match += match[0];
            this.matches = match;
            this.yyleng = this.yytext.length;
            if (this.options.ranges) {
                this.yylloc.range = [this.offset, this.offset += this.yyleng];
            }
            this._more = false;
            this._input = this._input.slice(match[0].length);
            this.matched += match[0];
            token = this.performAction.call(
                        this,
                        this.yy,
                        this,
                        rules[index],
                        this.conditionStack[this.conditionStack.length - 1]
                    );
            if (this.done && this._input) this.done = false;
            if (token) return token;
            else return;
        }
        if (this._input === "") { return this.EOF; }
        else {
            return this.parseError(
                     'Lexical error on line ' + (this.yylineno + 1) +
                      '. Unrecognized text.\n' + this.showPosition(),
                      { text: "", token: null, line: this.yylineno }
                   );
        }
    },
lex: function lex() {
    var r = this.next();
    if (typeof r !== 'undefined') {
        return r;
    } else {
        return this.lex();
    }
},
begin: function begin(condition) { },
popState: function popState() { },
_currentRules: function _currentRules() { },
topState: function() { },
pushState: function begin(condition) { },
options: {},
performAction: function anonymous(yy, yy_, $avoiding_name_collisions, YY_START)
{
    var YYSTATE = YY_START;
    switch ($avoiding_name_collisions) {
        case 0:
            return yy_.yytext;
```

```
                    break;
            }
        },
        rules: [/^(?:.)/], /* lista de expresiones regulares */
        conditions: { /* ... */ }
  }
};
```

5.12.11. Exportación

Si no ha sido exportado ya ...

```
if (typeof require !== 'undefined' && typeof exports !== 'undefined') {
    exports.parser = aSb;         /* hacemos accesible el objeto aSb  */
    exports.Parser = aSb.Parser;
```

El objeto `aSb.Parser` representa al parser. Este es el código que lo crea.

```
      function Parser() {
          this.yy = {};
      }
      Parser.prototype = parser;
      parser.Parser = Parser;
      return new Parser;
  })();
```

También se exporta una función `parse`:

```
    exports.parse = function() {
        return aSb.parse.apply(aSb, arguments);
    };
```

y una función `main`:

```
    exports.main = function commonjsMain(args) {
        if (!args[1]) {
            console.log('Usage: ' + args[0] + ' FILE');
            process.exit(1);
        }
        var source = require('fs').readFileSync(require('path').normalize(args[1]), "utf8");
        return exports.parser.parse(source);
    };
    if (typeof module !== 'undefined' && require.main === module) {
        exports.main(process.argv.slice(1));
    }
}
```

Esto permite ejecutar el módulo directamente:

```
[~/Dropbox/src/javascript/PLgrado/jison-aSb(develop)]$ node aSb.js input.ab
empty
S -> aSb
S -> aSb
[~/Dropbox/src/javascript/PLgrado/jison-aSb(develop)]$ cat input.ab
aabb
```

```
~/Dropbox/src/javascript/PLgrado/jison-aSb(develop)]$ node debug aSb.js input.ab
< debugger listening on port 5858
connecting... ok
break in aSb.js:2
  1 /* parser generated by jison 0.4.2 */
  2 var aSb = (function() {
  3     var parser = {
  4         trace: function trace() {},
debug> n
break in aSb.js:390
 388     return new Parser;
 389 })();
 390 if (typeof require !== 'undefined' && typeof exports !== 'undefined') {
 391     exports.parser = aSb;
 392     exports.Parser = aSb.Parser;

debug> repl
Press Ctrl + C to leave debug repl
>
> typeof require
'function'
> typeof exports
'object'
> aSb
{ yy: {} }
> aSb.Parser
[Function]
^C
debug> sb(396)
 395     };
debug> c
break in aSb.js:396
 394         return aSb.parse.apply(aSb, arguments);
 395     };
*396     exports.main = function commonjsMain(args) {
 397         if (!args[1]) {
 398             console.log('Usage: ' + args[0] + ' FILE');
debug> n
break in aSb.js:404
 402         return exports.parser.parse(source);
 403     };
 404     if (typeof module !== 'undefined' && require.main === module) {
 405         exports.main(process.argv.slice(1));
 406     }
debug> repl
Press Ctrl + C to leave debug repl
> process.argv.slice(1)
[ '/Users/casiano/Dropbox/src/javascript/PLgrado/jison-aSb/aSb.js',
  'input.ab' ]
> typeof module
'object'
> require.main
{ id: '.',
  exports:
```

```
   { parser: { yy: {} },
     Parser: [Function],
     parse: [Function],
     main: [Function] },
  parent: null,
  filename: '/Users/casiano/Dropbox/src/javascript/PLgrado/jison-aSb/aSb.js',
  loaded: false,
  children: [],
  paths:
   [ '/Users/casiano/Dropbox/src/javascript/PLgrado/jison-aSb/node_modules',
     '/Users/casiano/Dropbox/src/javascript/PLgrado/node_modules',
     '/Users/casiano/Dropbox/src/javascript/node_modules',
     '/Users/casiano/Dropbox/src/node_modules',
     '/Users/casiano/Dropbox/node_modules',
     '/Users/casiano/node_modules',
     '/Users/node_modules',
     '/node_modules' ] }
^C
debug> n
break in aSb.js:405
 403     };
 404     if (typeof module !== 'undefined' && require.main === module) {
 405         exports.main(process.argv.slice(1));
 406     }
 407 }
debug> n
< empty
< S -> aSb
< S -> aSb
break in aSb.js:409
 407 }
 408
 409 });
debug> c
program terminated
debug>
```

5.13. Precedencia y Asociatividad

Recordemos que si al construir la tabla LALR, alguna de las entradas de la tabla resulta multievaluada, decimos que existe un conflicto. Si una de las acciones es de 'reducción" y la otra es de 'desplazamiento", se dice que hay un *conflicto shift-reduce* o *conflicto de desplazamiento-reducción*. Si las dos reglas indican una acción de reducción, decimos que tenemos un *conflicto reduce-reduce* o de *reducción-reducción*. En caso de que no existan indicaciones específicas *jison* resuelve los conflictos que aparecen en la construcción de la tabla utilizando las siguientes reglas:

1. Un conflicto *reduce-reduce* se resuelve eligiendo la producción que se listó primero en la especificación de la gramática.

2. Un conflicto *shift-reduce* se resuelve siempre en favor del *shift*

Las declaraciones de precedencia y asociatividad mediante las palabras reservadas `%left` , `%right` , `%nonassoc` se utilizan para modificar estos criterios por defecto. La declaración de `token` s mediante

la palabra reservada `%token` no modifica la precedencia. Si lo hacen las declaraciones realizadas usando las palabras `left` , `right` y `nonassoc` .

1. Los *tokens* declarados en la misma línea tienen igual precedencia e igual asociatividad. La precedencia es mayor cuanto mas abajo su posición en el texto. Así, en el ejemplo de la calculadora en la sección **??**, el *token* `*` tiene mayor precedencia que `+` pero la misma que `/`.

2. La precedencia de una regla $A \rightarrow \alpha$ se define como la del terminal mas a la derecha que aparece en α. En el ejemplo, la producción

   ```
   expr : expr '+' expr
   ```

   tiene la precedencia del *token* `+`.

3. Para decidir en un conflicto *shift-reduce* se comparan la precedencia de la regla con la del terminal que va a ser desplazado. Si la de la regla es mayor se reduce si la del *token* es mayor, se desplaza.

4. Si en un conflicto *shift-reduce* ambos la regla y el terminal que va a ser desplazado tiene la misma precedencia *jison* considera la asociatividad, si es asociativa a izquierdas, reduce y si es asociativa a derechas desplaza. Si no es asociativa, genera un mensaje de error.
   Obsérvese que, en esta situación, la asociatividad de la regla y la del *token* han de ser por fuerza, las mismas. Ello es así, porque en *jison* los *tokens* con la misma precedencia se declaran en la misma línea y sólo se permite una declaración por línea.

5. *Por tanto es imposible declarar dos* tokens *con diferente asociatividad y la misma precedencia.*

6. Es posible modificar la precedencia "natural" de una regla, calificándola con un *token* específico. para ello se escribe a la derecha de la regla `prec token`, donde `token` es un *token* con la precedencia que deseamos. Vea el uso del *token* `dummy` en el siguiente ejercicio.

Para ilustrar las reglas anteriores usaremos el siguiente programa `jison`:

```
[~/jison/jison-prec(ast)]$ cat -n precedencia.jison
 1  %token NUMBER
 2  %left '@'
 3  %right '&'  dummy
 4  %%
 5  s
 6      : list      { console.log($list); }
 7      ;
 8
 9  list
10      :
11                  {
12                    $$ = [];
13                  }
14      | list '\n'
15                  {
16                    $$ = $1;
17                  }
18      | list e
19                  {
20                    $$ = $1;
21                    $$.push($e);
22                  }
```

```
23      ;
24
25  e : NUMBER
26                  {
27                    $$ = "NUMBER ("+yytext+")";
28                  }
29    | e '&' e
30                  {
31                    $$ = [ "&", $e1, $e2];
32                  }
33    | e '@' e %prec dummy
34                  {
35                    $$ = ["@", $e1, $e2];
36                  }
37    ;
38
39  %%
```

Obsérvese la siguiente ejecución:

```
[~/jison/jison-prec(ast)]$ cat input.txt
2@3@4
2&3&4
[~/jison/jison-prec(ast)]$ node precedencia.js input.txt
[ [ '@', [ '@', 'NUMBER (2)', 'NUMBER (3)' ], 'NUMBER (4)' ],
  [ '&', 'NUMBER (2)', [ '&', 'NUMBER (3)', 'NUMBER (4)' ] ] ]
```

Compilamos a continuación con `bison` usando la opción `-v` para producir información sobre los conflictos y las tablas de salto y de acciones:

```
[~/jison/jison-prec(ast)]$ bison -v precedencia.jison
precedencia.jison:6.31: warning: stray ‘$’
precedencia.jison:21.27: warning: stray ‘$’
precedencia.jison:31.31: warning: stray ‘$’
precedencia.jison:31.36: warning: stray ‘$’
precedencia.jison:35.30: warning: stray ‘$’
precedencia.jison:35.35: warning: stray ‘$’
```

La opción `-v` genera el fichero `Precedencia.output` el cual contiene información detallada sobre el autómata:

```
[~/jison/jison-prec(ast)]$ cat precedencia.output
Grammar

    0 $accept: s $end

    1 s: list

    2 list: /* empty */
    3     | list '\n'
    4     | list e

    5 e: NUMBER
    6  | e '&' e
    7  | e '@' e
```

```
Terminals, with rules where they appear

$end (0) 0
'\n' (10) 3
'&' (38) 6
'@' (64) 7
error (256)
NUMBER (258) 5
dummy (259)


Nonterminals, with rules where they appear

$accept (8)
    on left: 0
s (9)
    on left: 1, on right: 0
list (10)
    on left: 2 3 4, on right: 1 3 4
e (11)
    on left: 5 6 7, on right: 4 6 7


state 0

    0 $accept: . s $end

    $default  reduce using rule 2 (list)

    s     go to state 1
    list  go to state 2


state 1

    0 $accept: s . $end

    $end  shift, and go to state 3


state 2

    1 s: list .
    3 list: list . '\n'
    4     | list . e

    NUMBER  shift, and go to state 4
    '\n'    shift, and go to state 5

    $default  reduce using rule 1 (s)
```

```
    e  go to state 6


state 3

    0 $accept: s $end .

    $default  accept


state 4

    5 e: NUMBER .

    $default  reduce using rule 5 (e)


state 5

    3 list: list '\n' .

    $default  reduce using rule 3 (list)


state 6

    4 list: list e .
    6 e: e . '&' e
    7  | e . '@' e

    '@'  shift, and go to state 7
    '&'  shift, and go to state 8

    $default  reduce using rule 4 (list)


state 7

    7 e: e '@' . e

    NUMBER  shift, and go to state 4

    e  go to state 9


state 8

    6 e: e '&' . e

    NUMBER  shift, and go to state 4

    e  go to state 10
```

```
state 9

    6 e: e . '&' e
    7  | e . '@' e
    7  | e '@' e .

    '&'  shift, and go to state 8

    $default  reduce using rule 7 (e)


state 10

    6 e: e . '&' e
    6  | e '&' e .
    7  | e . '@' e

    '&'  shift, and go to state 8

    $default  reduce using rule 6 (e)
```

La presencia de conflictos, aunque no siempre, en muchos casos es debida a la introducción de ambiguedad en la gramática. Si el conflicto es de desplazamiento-reducción se puede resolver explicitando alguna regla que rompa la ambiguedad. Los conflictos de reducción-reducción suelen producirse por un diseño erróneo de la gramática. En tales casos, suele ser mas adecuado modificar la gramática.

5.14. Esquemas de Traducción

Un *esquema de traducción* es una gramática independiente del contexto en la cuál se han asociado atributos a los símbolos de la gramática. Un atributo queda caracterizado por un identificador o nombre y un tipo o clase. Además se han insertado acciones, esto es, código JavaScript/Perl/Python/C, ... en medio de las partes derechas. En ese código es posible referenciar los atributos de los símbolos de la gramática como variables del lenguaje subyacente.

Recuerde que el orden en que se evalúan los fragmentos de código es el de un recorrido primero-profundo del árbol de análisis sintáctico. Mas específicamente, considerando a las acciones como hijos-hoja del nodo, el recorrido que realiza un esquema de traducción es:

```
 1     function esquema_de_traduccion(node) {
 2
 3       for(c in node.children) { # de izquierda a derecha
 4         child = node.children[i];
 5         if (child.instanceof('SemanticAction') { # si es una acción semántica
 6           child.execute;
 7         }
 8         else { esquema_de_traduccion(child) }
 9       }
10     }
```

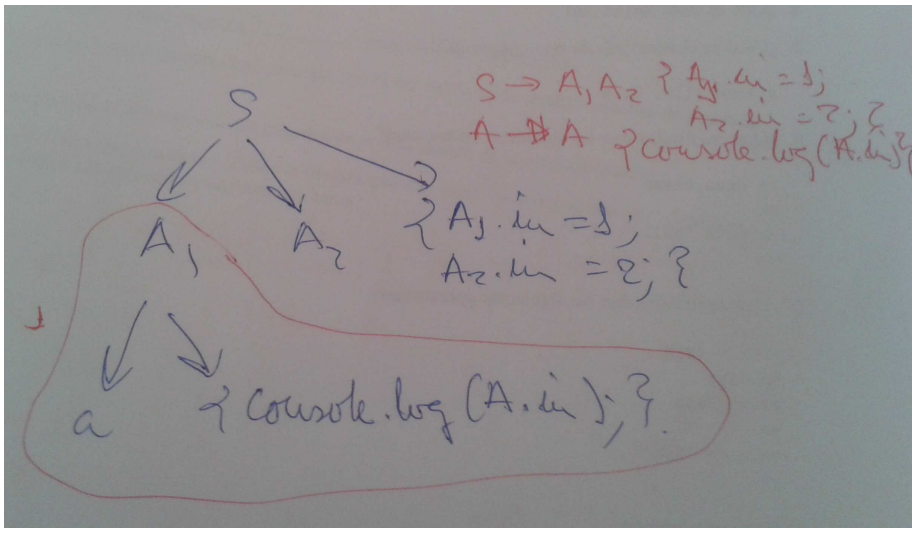
Obsérvese que, como el bucle recorre a los hijos de izquierda a derecha, se debe dar la siguiente condición para que un esquema de traducción funcione:

Para cualquier regla de producción aumentada con acciones, de la forma

$$A \to X_1 \dots X_j \{\ \texttt{action(A\{b\}, X}_1\texttt{\{c\}... X}_n\texttt{\{d\})}\} X_{j+1} \dots X_n$$

debe ocurrir que los atributos evaluados en la acción insertada después de X_j dependan de atributos y variables que fueron computadas durante la visita de los hermanos izquierdos o de sus ancestros. En particular no deberían depender de atributos asociados con las variables $X_{j+1} \dots X_n$. Ello no significa que no sea correcto evaluar atributos de $X_{j+1} \dots X_n$ en esa acción.

Por ejemplo, el siguiente esquema no satisface el requisito:



porque cuando vas a ejecutar la acción `{ console.log(A.in) }` el atributo `A.in` no ha sido computado.

Los atributos de cada símbolo de la gramática $X \in V \cup \Sigma$ se dividen en dos grupos disjuntos: *atributos sintetizados* y *atributos heredados*:

- Un atributo de X es un *atributo heredado* si depende de atributos de su padre y hermános en el árbol.
- Un *atributo sintetizado* es aquél tal que el valor del atributo depende de los valores de los atributos de los hijos, es decir en tal caso X ha de ser una variable sintáctica y los atributos en la parte derecha de la regla semántica deben ser atributos de símbolos en la parte derecha de la regla de producción asociada.

5.15. Manejo en `jison` de Atributos Heredados

Supongamos que `jison` esta inmerso en la construcción de la antiderivación a derechas y que la forma sentencial derecha en ese momento es:

$$X_m \dots X_1 X_0 Y_1 \dots Y_n a_1 \dots a_0$$

y que el mango es $B \to Y_1 \dots Y_n$ y en la entrada quedan por procesar $a_1 \dots a_0$.

No es posible acceder en `jison` a los valores de los atributos de los estados en la pila del analizador que se encuentran "por debajo" o si se quiere "a la izquierda" de los estados asociados con la regla por la que se reduce.

Vamos a usar un pequeño hack para acceder a los atributos asociados con símbolos vistos en el pasado remoto":

```
[~/jison/jison-inherited(grammar)]$ cat inherited.jison
%lex
%%
```

```
\s+                           {}
(global|local|integer|float)  { return yytext; }
[a-zA-Z_]\w*                  { return 'id'; }
.                             { return yytext; }
/lex
%%
D
  : C T L
  ;

C
  : global
  | local
  ;

T
  : integer
  | float
;

L
  : L ',' id    {
                  console.log("L -> L ',' id ("+yytext+")");
                  var s =  eval('$$');
                  console.log(s);
                }
  | id          {
                  console.log("L -> id ("+yytext+")");
                  var s =  eval('$$');
                  console.log(s);
                }
  ;
%%
```

Veamos un ejemplo de ejecución:

```
[~/jison/jison-inherited(grammar)]$ cat input.txt
global integer a, b, c
[~/jison/jison-inherited(grammar)]$ node inherited.js input.txt
L -> id (a)
[ null, 'global', 'integer', 'a' ]
L -> L ',' id (b)
[ null, 'global', 'integer', 'a', ',', 'b' ]
L -> L ',' id (c)
[ null, 'global', 'integer', 'a', ',', 'c' ]
```

Esta forma de acceder a los atributos es especialmente útil cuando se trabaja con *atributos heredados*. Esto es, cuando un atributo de un nodo del árbol sintáctico se computa en términos de valores de atributos de su padre y/o sus hermanos. Ejemplos de atributos heredados son la clase y tipo en la declaración de variables.

Es importante darse cuenta que en cualquier derivación a derechas desde`D`, cuando se reduce por una de las reglas

$$\mathrm{L} \rightarrow \mathrm{id} \mid \mathrm{L}_1 \text{ ',' } \mathrm{id}$$

el símbolo a la izquierda de L es T y el que esta a la izquierda de T es C. Considere, por ejemplo la derivación a derechas:

$$\begin{aligned} \text{D} \Longrightarrow \text{C T L} \Longrightarrow \text{C T L, id} \Longrightarrow \text{C T L, id, id} \Longrightarrow \text{C T id, id, id} \Longrightarrow \\ \Longrightarrow \text{C float id, id, id} \Longrightarrow \text{local float id, id, id} \end{aligned}$$

Observe que el orden de recorrido de `jison` es:

$$\begin{aligned} \text{local float id, id, id} \Longleftarrow \text{C float id, id} \Longleftarrow \text{C T id, id, id} \Longleftarrow \\ \Longleftarrow \text{C T L, id, id} \Longleftarrow \text{C T L, id} \Longleftarrow \text{C T L} \Longleftarrow \text{D} \end{aligned}$$

en la antiderivación, cuando el mango es una de las dos reglas para listas de identificadores, L $\rightarrow$ id y L $\rightarrow$ L, id es decir durante las tres ultimas antiderivaciones:

$$\text{C T L, id, id} \Longleftarrow \text{C T L, id} \Longleftarrow \text{C T L} \Longleftarrow \text{D}$$

las variables a la izquierda del mango son T y C. Esto ocurre siempre. Estas observaciones nos conducen al siguiente programa `jison`:

```
[~/jison/jison-inherited(deepstack)]$ cat inherited.jison
%lex
%%
\s+                             {}
(global|local|integer|float)  { return yytext; }
[a-zA-Z_]\w*                    { return 'id'; }
.                               { return yytext; }
/lex
%%
D
  : C T L
  ;

C
  : global
  | local
  ;

T
  : integer
  | float
  ;

L
  : L ',' id    {
                  var s =  eval('$$');
                  var b0 = s.length - 3;

                  console.log("L -> L ',' id ("+yytext+")");
                  console.log($id + ' is of type ' + s[b0-1]);
                  console.log(s[b0] + ' is of class ' + s[b0-2]);
                }
  | id          {
                  var s =  eval('$$');
                  var b0 = s.length - 1;
```

```
                    console.log("L -> id ("+yytext+")");
                    console.log($id + ' is of type ' + s[b0-1]);
                    console.log(s[b0] + ' is of class ' + s[b0-2]);
                  }
  ;
%%
```

A continuación sigue un ejemplo de ejecución:

```
[~/jison/jison-inherited(deepstack)]$ node inherited.js input.txt
L -> id (a)
a is of type integer
a is of class global
L -> L ',' id (b)
b is of type integer
a is of class global
L -> L ',' id (c)
c is of type integer
a is of class global
```

En este caso, existen varias alternativas simples a esta solución:

- Montar la lista de identificadores en un array y ponerles el tipo y la clase de "golpe.[en] la regla de producción superior `D : C T L ;`
- usar variables visibles (globales o atributos del objeto parser, por ejemplo) como `current_type`, `current_class`

  ```
  C
    : global { current_class = 'global'; }
    | local  { current_class = 'local'; }
  ```

  y depués accederlas en las reglas de `L`
- La que posiblemente sea la mejor estrategia general: construir el árbol de análisis sintáctico. Posteriormente podemos recorrer el árbol como queramos y tantas veces como queramos.

5.16. Definición Dirigida por la Sintáxis

Una *definición dirigida por la sintáxis* es un pariente cercano de los esquemas de traducción. En una definición dirigida por la sintáxis una gramática $G = (V, \Sigma, P, S)$ se aumenta con nuevas características:

- A cada símbolo $S \in V \cup \Sigma$ de la gramática se le asocian cero o mas atributos. Un atributo queda caracterizado por un identificador o nombre y un tipo o clase. A este nivel son *atributos formales*, como los parámetros formales, en el sentido de que su realización se produce cuando el nodo del árbol es creado.
- A cada regla de producción $A \rightarrow X_1 X_2 \ldots X_n \in P$ se le asocian un conjunto de *reglas de evaluación de los atributos* o *reglas semánticas* que indican que el atributo en la parte izquierda de la regla semántica depende de los atributos que aparecen en la parte derecha de la regla. El atributo que aparece en la parte izquierda de la regla semántica puede estar asociado con un símbolo en la parte derecha de la regla de producción.

- Los atributos de cada símbolo de la gramática $X \in V \cup \Sigma$ se dividen en dos grupos disjuntos: *atributos sintetizados* y *atributos heredados*. Un atributo de X es un *atributo heredado* si depende de atributos de su padre y hermános en el árbol. Un *atributo sintetizado* es aquél tal que el valor del atributo depende de los valores de los atributos de los hijos, es decir en tal caso X ha de ser una variable sintáctica y los atributos en la parte derecha de la regla semántica deben ser atributos de símbolos en la parte derecha de la regla de producción asociada.

- Los atributos predefinidos se denominán *atributos intrínsecos*. Ejemplos de atributos intrínsecos son los atributos sintetizados de los terminales, los cuáles se han computado durante la fase de análisis léxico. También son atributos intrínsecos los atributos heredados del símbolo de arranque, los cuales son pasados como parámetros al comienzo de la computación.

La diferencia principal con un esquema de traducción está en que no se especifica el orden de ejecución de las reglas semánticas. Se asume que, bien de forma manual o automática, se resolverán las dependencias existentes entre los atributos determinadas por la aplicación de las reglas semánticas, de manera que serán evaluados primero aquellos atributos que no dependen de ningún otro, despues los que dependen de estos, etc. siguiendo un esquema de ejecución que viene guiado por las dependencias existentes entre los datos.

Aunque hay muchas formas de realizar un evaluador de una definición dirigida por la sintáxis, conceptualmente, tal evaluador debe:

1. Construir el árbol de análisis sintáctico para la gramática y la entrada dadas.

2. Analizar las reglas semánticas para determinar los atributos, su clase y las dependencias entre los mismos.

3. Construir el *grafo de dependencias* de los atributos, el cual tiene un nodo para cada ocurrencia de un atributo en el árbol de análisis sintáctico etiquetado con dicho atributo. El grafo tiene una arista entre dos nodos si existe una dependencia entre los dos atributos a través de alguna regla semántica.

4. Supuesto que el grafo de dependencias determina un *orden parcial* (esto es cumple las propiedades reflexiva, antisimétrica y transitiva) construir un *orden topológico* compatible con el orden parcial.

5. Evaluar las reglas semánticas de acuerdo con el orden topológico.

Una definición dirigida por la sintáxis en la que las reglas semánticas no tienen efectos laterales se denomina una *gramática atribuída*.

Si la definición dirigida por la sintáxis puede ser realizada mediante un esquema de traducción se dice que es *L-atribuída*. Para que una definición dirigida por la sintáxis sea L-atribuída deben cumplirse que cualquiera que sea la regla de producción $A \to X_1 \dots X_n$, los atributos heredados de X_j pueden depender únicamente de:

1. Los atributos de los símbolos a la izquierda de X_j

2. Los atributos heredados de A

Nótese que las restricciones se refieren a los atributos heredados. El cálculo de los atributos sintetizados no supone problema para un esquema de traducción. Si la gramática es LL(1), resulta fácil realizar una definición L-atribuída en un analizador descendente recursivo predictivo.

Si la definición dirigida por la sintáxis sólo utiliza atributos sintetizados se denomina *S-atribuída*. Una definición S-atribuída puede ser fácilmente trasladada a un programa `jison`.

Ejercicio 5.16.1. *El siguiente es un ejemplo de definición dirigida por la sintáxis:*

$S \rightarrow a\ A\ C$	`$C{i} = $A{s}`
$S \rightarrow b\ A\ B\ C$	`$C{i} = $A{s}`
$C \rightarrow c$	`$C{s} = $C{i}`
$A \rightarrow a$	`$A{s} = "a"`
$B \rightarrow b$	`$B{s} = "b"`

Determine un orden correcto de evaluación de la anterior definición dirigida por la sintáxis para la entrada `b a b c`.

Ejercicio 5.16.2.

Lea el artículo Are Attribute Grammars used in Industry? por Josef Grosch

> *I am observing a lack of education and knowledge about compiler construction in industry. When I am asking the participants of our trainings or the employees we met in our projects then only few people have learned about compiler construction during their education. For many of them compiler construction has a bad reputation because of what and how they have learned about this topic. Even fewer people have a usable knowledge about compilers. Even fewer people know about the theory of attribute grammars. And even fewer people know how to use attribute grammars for solving practical problems. Nevertheless, attribute grammars are used in industry. However, in many cases the people in industry do not know about this fact. They are running prefabricated subsystems constructed by external companies such as ours. These subsystems are for example parsers which use attribute grammar technology.*

5.17. Ejercicios: Casos de Estudio

Véase nuestro proyecto Grammar Repository en GoogleCode.

5.17.1. Un mal diseño

Ejercicio 5.17.1. *This grammar*

```
%token D S

%%
p: ds ';' ss  | ss ;
ds: D ';' ds
  | D
;
ss: S ';' ss  | S  ;
%%
```

illustrates a typical LALR conflict due to a bad grammar design.

- *Reescriba la gramática para que no existan conflictos*
- *Escriba las acciones semánticas necesarias para imprimir la lista de* `D`*s seguida de la lista de* `S`*s*

Donde

- `[~/jison/jison-decs-sts(master)]$ pwd -P`
  `/Users/casiano/local/src/javascript/PLgrado/jison/jison-decs-sts`
- `[~/jison/jison-decs-sts(master)]$ git remote -v`
  `origin  git@github.com:crguezl/jison-decs-sts.git (fetch)`
  `origin  git@github.com:crguezl/jison-decs-sts.git (push)`
- https://github.com/crguezl/jison-decs-sts

5.17.2. Gramática no LR(1)

La siguiente gramática no es LR(1).

```
[~/srcPLgrado/jison/jison-nolr]$ cat confusingsolvedppcr.y
%%
A:
    B 'c' 'd'
  | E 'c' 'f'
;
B:
    'x' 'y'
;
E:
    'x' 'y'
;

%%
```

Encuentre una gramática sin conflictos equivalente a la anterior.

Donde

- `[~/jison/jison-nolr(master)]$ pwd -P`
  `/Users/casiano/local/src/javascript/PLgrado/jison/jison-nolr`
- `[~/jison/jison-nolr(master)]$ git remote -v`
  `origin  git@github.com:crguezl/jison-nolr.git (fetch)`
  `origin  git@github.com:crguezl/jison-nolr.git (push)`
- https://github.com/crguezl/jison-nolr

5.17.3. Un Lenguaje Intrínsecamente Ambiguo

Ejercicio 5.17.2. *A context-free language is inherently ambiguous if all context-free grammars generating that language are ambiguous. While some context-free languages have both ambiguous and unambiguous grammars, there are context-free languages for which no unambiguous context-free grammar can exist. An example of an inherently ambiguous language is the set*

`{ a^n b^n c^m : n >= 0, m >= 0 } U { a^n b^m c^m : n >= 0, m >= 0 }`

Esto es: Concatenaciones de repeticiones de `a`*s seguidas de repeticiones de* `b`*s y seguidas de repeticiones de* `c`*s donde el número de* `a`*s es igual al número de* `b`*s o bien el número de* `b`*s es igual al número de* `c`*s.*

- *Escriba una gramática que genere dicho lenguaje*

  ```
  s: aeqb | beqc ;
  aeqb: ab cs ;
  ab: /* empty */ | 'a' ab 'b' ;
  cs: /* empty */ | cs 'c' ;
  beqc: as bc ;
  bc: /* empty */ | 'b' bc 'c' ;
  as: /* empty */ | as 'a' ;
  ```

  The symbol `aeqb` *correspond to guess that there are the same number of* `a`*s than* `b`*s. In the same way,* `beqc` *starts the subgrammar for those phrases where the number of* `b`*s is equal to the number of* `c`*s. The usual approach to eliminate the ambiguity by changing the grammar to an unambiguous one does not work.*

- *Escriba un programa Jison que reconozca este lenguaje.*

Donde

- `[~/jison/jison-ambiguouslanguage(master)]$ pwd -P`
  `/Users/casiano/local/src/javascript/PLgrado/jison/jison-ambiguouslanguage`
- `[~/jison/jison-ambiguouslanguage(master)]$ git remote -v`
  `origin  git@github.com:crguezl/jison-ambiguouslanguage.git (fetch)`
  `origin  git@github.com:crguezl/jison-ambiguouslanguage.git (push)`
- https://github.com/crguezl/jison-ambiguouslanguage

5.17.4. Conflicto reduce-reduce

La siguiente gramática presenta conflictos reduce-reduce:

Ejercicio 5.17.3. `[~/srcPLgrado/jison/jison-reducereduceconflict]$ cat reducereduceconflictPPCR2`

```
%token ID

%%

def:    param_spec return_spec ','
        ;
param_spec:
             type
        |    name_list ':' type
        ;
return_spec:
             type
        |    name ':' type
        ;
type:
             ID
        ;
name:
             ID
        ;
name_list:
             name
        |    name ',' name_list
        ;
%%
```

Este es el diagnóstico de Jison:

```
~/srcPLgrado/jison/jison-reducereduceconflict]$ jison reducereduceconflictPPCR2.y
Conflict in grammar: multiple actions possible when lookahead token is ID in state 5
- reduce by rule: name -> ID
- reduce by rule: type -> ID
Conflict in grammar: multiple actions possible when lookahead token is : in state 5
- reduce by rule: name -> ID
- reduce by rule: type -> ID
Conflict in grammar: multiple actions possible when lookahead token is , in state 5
- reduce by rule: name -> ID
- reduce by rule: type -> ID
```

```
States with conflicts:
State 5
  type -> ID . #lookaheads= ID : ,
  name -> ID . #lookaheads= ID : ,
```

Este es el resultado de compilar con `bison -v`*:*

```
[~/jison/jison-reducereduceconflict(master)]$ bison -v reducereduceconflictPPCR2.y
reducereduceconflictPPCR2.y: conflicts: 1 reduce/reduce
```

El fichero `reducereduceconflictPPCR2.output` *queda así:*

```
[~/jison/jison-reducereduceconflict(master)]$ cat reducereduceconflictPPCR2.output
State 1 conflicts: 1 reduce/reduce


Grammar

    0 $accept: def $end

    1 def: param_spec return_spec ','

    2 param_spec: type
    3           | name_list ':' type

    4 return_spec: type
    5            | name ':' type

    6 type: ID

    7 name: ID

    8 name_list: name
    9          | name ',' name_list


Terminals, with rules where they appear

$end (0) 0
',' (44) 1 9
':' (58) 3 5
error (256)
ID (258) 6 7


Nonterminals, with rules where they appear

$accept (6)
    on left: 0
def (7)
    on left: 1, on right: 0
param_spec (8)
    on left: 2 3, on right: 1
return_spec (9)
```

```
    on left: 4 5, on right: 1
type (10)
    on left: 6, on right: 2 3 4 5
name (11)
    on left: 7, on right: 5 8 9
name_list (12)
    on left: 8 9, on right: 3 9


state 0

    0 $accept: . def $end

    ID  shift, and go to state 1

    def         go to state 2
    param_spec  go to state 3
    type        go to state 4
    name        go to state 5
    name_list   go to state 6


state 1

    6 type: ID .
    7 name: ID .

    ','       reduce using rule 6 (type)
    ','       [reduce using rule 7 (name)]
    ':'       reduce using rule 7 (name)
    $default  reduce using rule 6 (type)


state 2

    0 $accept: def . $end

    $end  shift, and go to state 7


state 3

    1 def: param_spec . return_spec ','

    ID  shift, and go to state 1

    return_spec  go to state 8
    type         go to state 9
    name         go to state 10


state 4
```

```
    2 param_spec: type .

    $default  reduce using rule 2 (param_spec)


state 5

    8 name_list: name .
    9          | name . ',' name_list

    ','  shift, and go to state 11

    $default  reduce using rule 8 (name_list)


state 6

    3 param_spec: name_list . ':' type

    ':'  shift, and go to state 12


state 7

    0 $accept: def $end .

    $default  accept


state 8

    1 def: param_spec return_spec . ','

    ','  shift, and go to state 13


state 9

    4 return_spec: type .

    $default  reduce using rule 4 (return_spec)


state 10

    5 return_spec: name . ':' type

    ':'  shift, and go to state 14


state 11

    9 name_list: name ',' . name_list
```

```
    ID  shift, and go to state 15

    name       go to state 5
    name_list  go to state 16


state 12

    3 param_spec: name_list ':' . type

    ID  shift, and go to state 17

    type  go to state 18


state 13

    1 def: param_spec return_spec ',' .

    $default  reduce using rule 1 (def)


state 14

    5 return_spec: name ':' . type

    ID  shift, and go to state 17

    type  go to state 19


state 15

    7 name: ID .

    $default  reduce using rule 7 (name)


state 16

    9 name_list: name ',' name_list .

    $default  reduce using rule 9 (name_list)


state 17

    6 type: ID .

    $default  reduce using rule 6 (type)
```

```
state 18

    3 param_spec: name_list ':' type .

    $default  reduce using rule 3 (param_spec)


state 19

    5 return_spec: name ':' type .

    $default  reduce using rule 5 (return_spec)
```

Encuentre una gramática equivalente a la anterior sin conflictos.

Solución When the problem arises, you can often fix it by identifying the two parser states that are being confused, and adding something to make them look distinct. In the above example, adding one rule to `return_spec` as follows makes the problem go away:

```
%token BOGUS
...
%%
...
return_spec:
             type
        |    name ':' type
        /* This rule is never used.  */
        |    ID BOGUS
        ;
```

This corrects the problem because it introduces the possibility of an additional active rule in the context after the ID at the beginning of `return_spec`. This rule is not active in the corresponding context in a `param_spec`, so the two contexts receive distinct parser states. As long as the token BOGUS is never generated by yylex, the added rule cannot alter the way actual input is parsed.

In this particular example, there is another way to solve the problem: rewrite the rule for `return_spec` to use ID directly instead of via name. This also causes the two confusing contexts to have different sets of active rules, because the one for `return_spec` activates the altered rule for `return_spec` rather than the one for name.

```
param_spec:
             type
        |    name_list ':' type
        ;
return_spec:
             type
        |    ID ':' type
        ;
```

Donde

- `[~/jison/jison-reducereduceconflict(master)]$ pwd -P`
  `/Users/casiano/local/src/javascript/PLgrado/jison/jison-reducereduceconflict`
- `[~/jison/jison-reducereduceconflict(master)]$ git remote -v`
  `origin  git@github.com:crguezl/ull-etsii-grado-PL-reducereduce.git (fetch)`
  `origin  git@github.com:crguezl/ull-etsii-grado-PL-reducereduce.git (push)`

- https://github.com/crguezl/ull-etsii-grado-PL-reducereduce

Véase Véase Mysterious Reduce/Reduce Conflicts

5.18. Recuperación de Errores

La recuperación de errores no parece estar implementada en Jison. véase

- la sección Error Recovery de la documentación
- Pullreq 5 - parser built-in grammar error recovery was completely broken

```
[~/srcPLgrado/jison/jison-aSb(error)]$ cat aSb.jison
%lex
%%
\s+             {}
[ab]            { return yytext; }
.               { return "INVALID"; }
/lex
%%
S: /* empty */  { $$ = ''; console.log("empty"); }
   | 'a' S 'b'  { $$ = $1 + $2 + $3; console.log("S -> aSb"); }
   | 'a' S error
;
%%

        parse: function parse(input) {
              var self = this,
                  stack = [0],
                  vstack = [null], // semantic value stack
                  lstack = [], // location stack
                  ...
                  recovering = 0,
                  TERROR = 2,
                  EOF = 1;
              while (true) {
                  // retreive state number from top of stack
                  state = stack[stack.length - 1];

                  ...

                  // handle parse error
>>                _handle_error: if (typeof action === 'undefined' || !action.length || !  ...

                      var errStr = '';
                      if (!recovering) {
```

5.19. Depuración en `jison`

5.20. Construcción del Árbol Sintáctico

El siguiente ejemplo usa `jison` para construir el árbol sintáctico de una expresión en infijo:

5.21. Consejos a seguir al escribir un programa `jison`

Cuando escriba un programa `jison` asegurese de seguir los siguientes consejos:

1. Coloque el punto y coma de separación de reglas en una línea aparte. Un punto y coma "pegado" al final de una regla puede confundirse con un terminal de la regla.

2. Si hay una regla que produce vacío, coloquela en primer lugar y acompáñela de un comentario resaltando ese hecho.

3. Nunca escriba dos reglas de producción en la misma línea.

4. Sangre convenientemente todas las partes derechas de las reglas de producción de una variable, de modo que queden alineadas.

5. Ponga nombres representativos a sus variables sintácticas. No llame `Z` a una variable que representa el concepto "lista de parámetros", llámela `ListaDeParametros`.

6. Es conveniente que declare los terminales simbólicos, esto es, aquellos que llevan un identificador asociado. Si no llevan prioridad asociada o no es necesaria, use una declaración `%token`. De esta manera el lector de su programa se dará cuenta rápidamente que dichos identificadores no se corresponden con variables sintácticas. Por la misma razón, si se trata de terminales asociados con caracteres o cadenas no es tan necesario que los declare, a menos que, como en el ejemplo de la calculadora para `'+'` y `'*'`, sea necesario asociarles una precedencia.

7. Es importante que use la opción `-v` para producir el fichero `.output` conteniendo información detallada sobre los conflictos y el autómata. Cuando haya un conflicto shift-reduce no resuelto busque en el fichero el estado implicado y vea que LR(0) items $A \to \alpha_\uparrow$ y $B \to \beta_\uparrow\gamma$ entran en conflicto.

8. Si según el informe de `jison` el conflicto se produce ante un terminal a, es porque $a \in FOLLOW(A)$ y $a \in FIRST(\gamma)$. Busque las causas por las que esto ocurre y modifique su gramática con vistas a eliminar la presencia del terminal a en uno de los dos conjuntos implicados o bien establezca reglas de prioridad entre los terminales implicados que resuelvan el conflicto.

9. Nótese que cuando existe un conflicto de desplazamiento reducción entre $A \to \alpha_\uparrow$ y $B \to \beta_\uparrow\gamma$, el programa `jison` contabiliza un error por cada terminal $a \in FOLLOW(A) \cap FIRST(\gamma)$. Por esta razón, si hay 16 elementos en $FOLLOW(A) \cap FIRST(\gamma)$, el analizador `jison` informará de la existencia de 16 conflictos *shift-reduce*, cuando en realidad se trata de uno sólo. No desespere, los conflictos "auténticos" suelen ser menos de los que `jison` anuncia.

10. Si necesita declarar variables globales, inicializaciones, etc. que afectan la conducta global del analizador, escriba el código correspondiente en la cabecera del analizador, protegido por los delimitadores `%{` y `%}`. Estos delimitadores deberán aparecer en una línea aparte. Por ejemplo:

```
%{
our contador = 0;
%}

%token NUM
...
%%
```

Capítulo 6

Análisis Sintáctico Ascendente en Ruby

6.1. La Calculadora

6.1.1. Uso desde Línea de Comandos

```
[~/src/PL/rexical/sample(master)]$ racc --help
Usage: racc [options] <input>
    -o, --output-file=PATH           output file name [<input>.tab.rb]
    -t, --debug                      Outputs debugging parser.
    -g                               Equivalent to -t (obsolete).
    -v, --verbose                    Creates <filename>.output log file.
    -O, --log-file=PATH              Log file name [<input>.output]
    -e, --executable [RUBYPATH]      Makes executable parser.
    -E, --embedded                   Embeds Racc runtime in output.
        --line-convert-all           Converts line numbers of user codes.
    -l, --no-line-convert            Never convert line numbers.
    -a, --no-omit-actions            Never omit actions.
        --superclass=CLASSNAME       Uses CLASSNAME instead of Racc::Parser.
        --runtime=FEATURE            Uses FEATURE instead of 'racc/parser'
    -C, --check-only                 Checks syntax and quit immediately.
    -S, --output-status              Outputs internal status time to time.
    -P                               Enables generator profile
    -D flags                         Flags for Racc debugging (do not use).
        --version                    Prints version and quit.
        --runtime-version            Prints runtime version and quit.
        --copyright                  Prints copyright and quit.
        --help                       Prints this message and quit.
```

```
[~/Dropbox/src/PL/rexical/sample(master)]$ cat -n Rakefile
     1  task :default => %W{racc rex} do
     2    sh "ruby calc3.tab.rb"
     3  end
     4
     5  task :racc do
     6    sh "racc calc3.racc"
     7  end
     8
     9  task :rex do
    10    sh "rex calc3.rex"
    11  end
```

6.1.2. Análisis Léxico con `rexical`

```
[~/Dropbox/src/PL/rexical/sample(master)]$ cat -n calc3.rex
     1  #
     2  # calc3.rex
     3  # lexical scanner definition for rex
     4  #
     5
     6  class Calculator3
     7  macro
     8    BLANK         \s+
     9    DIGIT         \d+
    10  rule
    11    {BLANK}
    12    {DIGIT}       { [:NUMBER, text.to_i] }
    13    .|\n          { [text, text] }
    14  inner
    15  end
```

6.1.3. Análisis Sintáctico

```
[~/Dropbox/src/PL/rexical/sample(master)]$ cat -n calc3.racc
     1  #
     2  # A simple calculator, version 3.
     3  #
     4
     5  class Calculator3
     6    prechigh
     7      nonassoc UMINUS
     8      left '*' '/'
     9      left '+' '-'
    10    preclow
    11    options no_result_var
    12  rule
    13    target  : exp
    14            | /* none */ { 0 }
    15
    16    exp     : exp '+' exp { val[0] + val[2] }
    17            | exp '-' exp { val[0] - val[2] }
    18            | exp '*' exp { val[0] * val[2] }
    19            | exp '/' exp { val[0] / val[2] }
    20            | '(' exp ')' { val[1] }
    21            | '-' NUMBER  =UMINUS { -(val[1]) }
    22            | NUMBER
    23  end
    24
    25  ---- header ----
    26  #
    27  # generated by racc
    28  #
    29  require 'calc3.rex'
    30
    31  ---- inner ----
    32
```

```
33  ---- footer ----
34
35  puts 'sample calc'
36  puts '"q" to quit.'
37  calc = Calculator3.new
38  while true
39    print '>>> '; $stdout.flush
40    str = $stdin.gets.strip
41    break if /q/i === str
42    begin
43      p calc.scan_str(str)
44    rescue ParseError
45      puts 'parse error'
46    end
47  end
```

Precedencias `right` is yacc's `%right`, `left` is yacc's `%left`.
`= SYMBOL` means yacc's `%prec SYMBOL`:

```
prechigh
  nonassoc '++'
  left     '*' '/'
  left     '+' '-'
  right    '='
preclow

rule
  exp: exp '*' exp
     | exp '-' exp
     | '-' exp       =UMINUS   # equals to "%prec UMINUS"
         :
         :
```

Atributos You can use following special local variables in action.

1. `result ($$)`

   The value of left-hand side (lhs). A default value is `val[0]`.

2. `val ($1,$2,$3...)`

   An array of value of right-hand side (rhs).

3. `_values (...$-2,$-1,$0)`

   A stack of values. DO NOT MODIFY this stack unless you know what you are doing.

Declaring Tokens By declaring tokens, you can avoid bugs.

```
token NUM ID IF
```

Opciones
You can write options for racc command in your racc file.

```
options OPTION OPTION ...
```

Options are:

1. `omit_action_call`

   omit empty action call or not.

2. `result_var`

   use/does not use local variable result"

You can use `no_` prefix to invert its meanings.

User Code Block

Üser Code Blockïs a Ruby source code which is copied to output. There are three user code block, "headerinner."and "footer".

Format of user code is like this:

```
---- header
  ruby statement
  ruby statement
  ruby statement

---- inner
  ruby statement
     :
     :
```

If four - exist on line head, racc treat it as beginning of user code block. A name of user code must be one word.

6.2. Véase También

- Racc en GitHub
-
- Racc User's Manual
- Martin Fowler Hello Racc
- Rexical en GitHub

Capítulo 7

Transformaciones Árbol

7.1. Árbol de Análisis Abstracto

Un *árbol de análisis abstracto* (denotado AAA, en inglés *abstract syntax tree* o AST) porta la misma información que el árbol de análisis sintáctico pero de forma mas condensada, eliminándose terminales y producciones que no aportan información.

Alfabeto con Aridad o Alfabeto Árbol

Definición 7.1.1. *Un* alfabeto con función de aridad *es un par* (Σ, ρ) *donde* Σ *es un conjunto finito y* ρ *es una función* $\rho : \Sigma \to \mathbb{N}_0$*, denominada* función de aridad. *Denotamos por* $\Sigma_k = \{a \in \Sigma : \ \rho(a) = k\}$.

Lenguaje de los Arboles *Definimos el* lenguaje árbol homogéneo $B(\Sigma)$ *sobre* Σ *inductivamente:*

- *Todos los elementos de aridad 0 están en* $B(\Sigma)$*:* $a \in \Sigma_0$ *implica* $a \in B(\Sigma)$
- *Si* $b_1, \ldots, b_k \in B(\Sigma)$ *y* $f \in \Sigma_k$ *es un elemento k-ario, entonces* $f(b_1, \ldots, b_k) \in B(\Sigma)$

Los elementos de $B(\Sigma)$ *se llaman* árboles *o* términos.

Ejemplo 7.1.1. *Sea* $\Sigma = \{A, CONS, NIL\}$ *con* $\rho(A) = \rho(NIL) = 0, \ \rho(CONS) = 2$. *Entonces* $B(\Sigma) = \{A, NIL, CONS(A, NIL), CONS(NIL, A), CONS(A, A), CONS(NIL, NIL), \ldots\}$

Ejemplo 7.1.2. *Una versión simplificada del alfabeto con aridad en el que estan basados los árboles construidos por el compilador de Tutu es:*
$\Sigma = \{ID, NUM, LEFTVALUE, STR, PLUS, TIMES, ASSIGN, PRINT\}$
$\rho(ID) = \rho(NUM) = \rho(LEFTVALUE) = \rho(STR) = 0$
$\rho(PRINT) = 1$
$\rho(PLUS) = \rho(TIMES) = \rho(ASSIGN) = 2$.
Observe que los elementos en $B(\Sigma)$ *no necesariamente son árboles "correctos". Por ejemplo, el árbol* $ASSIGN(NUM, PRINT(ID))$ *es un elemento de* $B(\Sigma)$.

Gramática Árbol

Definición 7.1.2. *Una* gramática árbol regular *es una cuadrupla* $((\Sigma, \rho), N, P, S)$*, donde:*

- (Σ, ρ) *es un alfabeto con aricidad* $\rho : \Sigma \to \mathbb{N}$
- N *es un conjunto finito de variables sintácticas o no terminales*
- P *es un conjunto finito de reglas de producción de la forma* $X \to s$ *con* $X \in N$ *y* $s \in B(\Sigma \cup N)$
- $S \in N$ *es la variable o símbolo de arranque*

Lenguaje Generado por una Gramática Árbol

Definición 7.1.3. *Dada una gramática (Σ, N, P, S), se dice que un árbol $t \in B(\Sigma \cup N)$ es del tipo $(X_1, \dots X_k)$ si el j-ésimo noterminal, contando desde la izquierda, que aparece en t es $X_j \in N$.*

Si $p = X \to s$ es una producción y s es de tipo $(X_1, \dots X_n)$, diremos que la producción p es de tipo $(X_1, \dots X_n) \to X$.

Definición 7.1.4. *Consideremos un árbol $t \in B(\Sigma \cup N)$ que sea del tipo $(X_1, \dots X_n)$, esto es las variables sintácticas en el árbol leídas de izquierda a derecha son $(X_1, \dots X_n)$.*

- *Si $X_i \to s_i \in P$ para algún i, entonces decimos que el árbol t* deriva en un paso en el árbol *t' resultante de sustituir el nodo X_i por el árbol s_i y escribiremos $t \Longrightarrow t'$. Esto es, $t' = t\{X_i/s_i\}$*
- *Todo árbol deriva en cero pasos en si mismo $t \overset{0}{\Longrightarrow} t$.*
- *Decimos que un árbol t deriva en n pasos en el árbol t' y escribimos $t \overset{n}{\Longrightarrow} t'$ si t deriva en un paso en un árbol t'' el cuál deriva en $n-1$ pasos en t'. En general, si t deriva en un cierto número de pasos en t' escribiremos $t \overset{*}{\Longrightarrow} t'$.*

Definición 7.1.5. *Se define el* lenguaje árbol generado por una gramática *$G = (\Sigma, N, P, S)$ como el lenguaje $L(G) = \{t \in B(\Sigma) : \exists S \overset{*}{\Longrightarrow} t\}$.*

Ejemplo 7.1.3. *Sea $G = (\Sigma, V, P, S)$ con $\Sigma = \{A, CONS, NIL\}$ y $\rho(A) = \rho(NIL) = 0$, $\rho(CONS) = 2$ y sea $V = \{exp, list\}$. El conjunto de producciones P es:*

$$P_1 = \{list \to NIL,\ list \to CONS(exp, list),\ exp \to A\}$$

La producción $list \to CONS(exp, list)$ es del tipo $(exp, list) \to list$.

El lenguaje generado por G se obtiene realizando sustituciones sucesivas (derivando) desde el símbolo de arranque hasta producir un árbol cuyos nodos estén etiquetados con elementos de Σ. En este ejemplo, $L(G)$ es el conjunto de arboles de la forma:

$$L(G) = \{NIL, CONS(A, NIL), CONS(A, CONS(A, NIL)), \dots\}$$

Ejercicio 7.1.1. *Construya una derivación para el árbol $CONS(A, CONS(A, NIL))$. ¿De que tipo es el árbol $CONS(exp, CONS(A, CONS(exp, L)))$?.*

Cuando hablamos del AAA producido por un analizador sintáctico, estamos en realidad hablando de un lenguaje árbol cuya definición precisa debe hacerse a través de una gramática árbol regular. Mediante las gramáticas árbol regulares disponemos de un mecanismo para describir formalmente el lenguaje de los AAA que producirá el analizador sintáctico para las sentencias Tutu.

Ejemplo 7.1.4. *Sea $G = (\Sigma, V, P, S)$ con*

$$\begin{aligned}
&\Sigma = \{ID, NUM, LEFTVALUE, STR, PLUS, TIMES, ASSIGN, PRINT\}\\
&\rho(ID) = \rho(NUM) = \rho(LEFTVALUE) = \rho(STR) = 0\\
&\rho(PRINT) = 1\\
&\rho(PLUS) = \rho(TIMES) = \rho(ASSIGN) = 2\\
&V = \{st, expr\}
\end{aligned}$$

y las producciones:

$$\begin{aligned}
P = \ & \{\\
& st \ \to ASSIGN(LEFTVALUE, expr)\\
& st \ \to PRINT(expr)\\
& expr \ \to PLUS(expr, expr)\\
& expr \ \to TIMES(expr, expr)\\
& expr \ \to NUM\\
& expr \ \to ID\\
& expr \ \to STR\\
& \}
\end{aligned}$$

Entonces el lenguaje $L(G)$ contiene árboles como el siguiente:

$$
\begin{array}{lll}
ASSIGN & (& \\
 & LEFTVALUE, & \\
 & PLUS & (\\
 & & ID, \\
 & & TIMES \quad (\\
 & & \qquad NUM, \\
 & & \qquad ID \\
 & & \qquad) \\
 & &) \\
 &) &
\end{array}
$$

El cual podría corresponderse con una sentencia como `a = b + 4 * c`.

El lenguaje de árboles descrito por esta gramática árbol es el lenguaje de los AAA de las sentencias de Tutu.

Ejercicio 7.1.2. *Redefina el concepto de árbol de análisis concreto dado en la definición 5.1.7 utilizando el concepto de gramática árbol. Con mas precisión, dada una gramática $G = (\Sigma, V, P, S)$ defina una gramática árbol $T = (\Omega, N, R, U)$ tal que $L(T)$ sea el lenguaje de los árboles concretos de G. Puesto que las partes derechas de las reglas de producción de P pueden ser de distinta longitud, existe un problema con la aricidad de los elementos de Ω. Discuta posibles soluciones.*

Ejercicio 7.1.3. *¿Cómo son los árboles sintácticos en las derivaciones árbol? Dibuje varios árboles sintácticos para las gramáticas introducidas en los ejemplos* **??** *y* **??**.

Intente dar una definición formal del concepto de árbol de análisis sintáctico asociado con una derivación en una gramática árbol

Notación de Dewey o Coordenadas de un Árbol

Definición 7.1.6. *La notación de Dewey es una forma de especificar los subárboles de un árbol $t \in B(\Sigma)$. La notación sigue el mismo esquema que la numeración de secciones en un texto: es una palabra formada por números separados por puntos. Así $t/2.1.3$ denota al tercer hijo del primer hijo del segundo hijo del árbol t. La definición formal sería:*

- $t/\epsilon = t$
- *Si $t = a(t_1, \ldots t_k)$ y $j \in \{1 \ldots k\}$ y n es una cadena de números y puntos, se define inductivamente el subárbol $t/j.n$ como el subárbol n-ésimo del j-ésimo subárbol de t. Esto es: $t/j.n = t_j/n$*

Ejercicio 7.1.4. *Sea el árbol:*

$$
\begin{array}{lll}
t = ASSIGN & (& \\
 & LEFTVALUE, & \\
 & PLUS & (\\
 & & ID, \\
 & & TIMES \quad (\\
 & & \qquad NUM, \\
 & & \qquad ID \\
 & & \qquad) \\
 & &) \\
 &) &
\end{array}
$$

Calcule los subárboles t/ϵ, $t/2.2.1$, $t/2.1$ y $t/2.1.2$.

7.2. Selección de Código y Gramáticas Árbol

La generación de código es la fase en la que a partir de la *Representación intermedia* o *IR* se genera una secuencia de instrucciones para la máquina objeto. Esta tarea conlleva diversas subtareas, entre ellas destacan tres:

- La selección de instrucciones o selección de código,
- La asignación de registros y
- La planificación de las instrucciones.

El problema de la *selección de código* surge de que la mayoría de las máquinas suelen tener una gran variedad de instrucciones, habitualmente cientos y muchas instrucciones admiten mas de una decena de modos de direccionamiento. En consecuencia,

There Is More Than One Way To Do It (The Translation)

Es posible asociar una gramática árbol con el juego de instrucciones de una máquina. Las partes derechas de las reglas de producción de esta gramática vienen determinadas por el conjunto de árboles sintácticos de las instrucciones. La gramática tiene dos variables sintácticas que denotan dos tipos de recursos de la máquina: los registros representados por la variable sintáctica R y las direcciones de memoria representadas por M. Una instrucción deja su resultado en cierto lugar, normalmente un registro o memoria. La idea es que las variables sintácticas en los lados izquierdos de las reglas representan los lugares en los cuales las instrucciones dejan sus resultados.

Ademas, a cada instrucción le asociamos un coste:

Gramática Arbol Para un Juego de Instrucciones Simple		
Producción	Instrucción	Coste
R → NUM	`LOADC R, NUM`	1
R → M	`LOADM R, M`	3
M → R	`STOREM M, R`	3
R → PLUS(R,M)	`PLUSM R, M`	3
R → PLUS(R,R)	`PLUSR R, R`	1
R → TIMES(R,M)	`TIMESM R, M`	6
R → TIMES(R,R)	`TIMESR R, R`	4
R → PLUS(R,TIMES(NUM,R))	`PLUSCR R, NUM, R`	4
R → TIMES(R,TIMES(NUM,R))	`TIMESCR R, NUM, R`	5

Consideremos la IR consistente en el AST generado por el front-end del compilador para la expresión `x+3*(7*y)`:

```
PLUS(M[x],TIMES(N[3],TIMES(N[7],M[y])
```

Construyamos una derivación a izquierdas para el árbol anterior:

Una derivación árbol a izquierdas para $P(M, T(N, T(N, M)))$			
Derivación	Producción	Instrucción	Coste
$R \Longrightarrow$	R → PLUS(R,R)	`PLUSR R, R`	1
$P(R, R) \Longrightarrow$	R → M	`LOADM R, M`	3
$P(M, R) \Longrightarrow$	R → TIMES(R,R)	`TIMESR R, R`	4
$P(M, T(R, R)) \Longrightarrow$	R → NUM	`LOADC R, NUM`	1
$P(M, T(N, R)) \Longrightarrow$	R → TIMES(R,R)	`TIMESR R, R`	4
$P(M, T(N, T(R, R))) \Longrightarrow$	R → NUM	`LOADC R, NUM`	1
$P(M, T(N, T(N, R))) \Longrightarrow$	R → M	`LOADM R, M`	3
$P(M, T(N, T(N, M)))$			Total: 17

Obsérvese que, si asumimos por ahora que hay suficientes registros, la secuencia de instrucciones resultante en la tercera columna de la tabla si se lee en orden inverso (esto es, si se sigue el orden de instrucciones asociadas a las reglas de producción en orden de anti-derivación) y se hace una asignación correcta de registros nos da una traducción correcta de la expresión `x+3*(7*y)`:

```
LOADM R, M          # y
LOADC R, NUM        # 7
TIMESR R, R         # 7*y
LOADC R, NUM        # 3
TIMESR R, R         # 3*(7*y)
LOADM R, M          # x
PLUSR R, R          # x+3*(7*y)
```

La gramática anterior es ambigua. El árbol de `x+3*(7*y)` puede ser generado también mediante la siguiente derivación a izquierdas:

Otra derivación árbol a izquierdas para $P(M,T(N,T(N,M)))$			
Derivación	Producción	Instrucción	Coste
$R \Longrightarrow$	R → PLUS(R,TIMES(NUM,R))	`PLUSCR R, NUM, R`	4
$P(R,T(N,R)) \Longrightarrow$	R → M	`LOADM R, M`	3
$P(M,T(N,R)) \Longrightarrow$	R → TIMES(R,M)	`TIMESM R, M`	6
$P(M,T(N,T(R,M)))$	R → NUM	`LOADC R, NUM`	1
$P(M,T(N,T(N,M)))$			Total: 14

La nueva secuencia de instrucciones para `x+3*(7*y)` es:

```
LOADC R, NUM        # 7
TIMESM R, M         # 7*y
LOADM R, M          # x
PLUSCR R, NUM, R    # x+3*(7*y)
```

Cada antiderivación a izquierdas produce una secuencia de instrucciones que es una traducción legal del AST de `x+3*(7*y)`.

El problema de la selección de código óptima puede aproximarse resolviendo el problema de encontrar la derivación árbol óptima que produce el árbol de entrada (en representación intermedia IR)

Definición 7.2.1. *Un* generador de generadores de código *es una componente software que toma como entrada una especificación de la plataforma objeto -por ejemplo mediante una gramática árbol- y genera un módulo que es utilizado por el compilador. Este módulo lee la representación intermedia (habitualmente un árbol) y retorna código máquina como resultado.*

Un ejemplo de generador de generadores de código es iburg [7].

Véase también el libro Automatic Code Generation Using Dynamic Programming Techniques y la página http://www.bytelabs.org/hburg.html

Ejercicio 7.2.1. *Responda a las siguientes preguntas:*

- *Sea G_M la gramática árbol asociada segun la descripción anterior con el juego de instrucciones de la máquina M. Especifique formalmente las cuatro componentes de la gramática $G_M = (\Sigma_M, V_M, P_M, S_M)$*
- *¿Cual es el lenguaje árbol generado por G_M?*
- *¿A que lenguaje debe pertenecer la representación intermedia IR para que se pueda aplicar la aproximación presentada en esta sección?*

7.3. Patrones Árbol y Transformaciones Árbol

Una transformación de un programa puede ser descrita como un conjunto de *reglas de transformación* o *esquema de traducción árbol* sobre el árbol abstracto que representa el programa.

En su forma mas sencilla, estas reglas de transformación vienen definidas por ternas $(p, e, action)$, donde la primera componente de la terna p es un *patrón árbol* que dice que árboles deben ser seleccionados. La segunda componente e dice cómo debe transformarse el árbol que casa con el patrón p. La acción *action* indica como deben computarse los atributos del árbol transformado a partir de los atributos del árbol que casa con el patrón p. Una forma de representar este esquema sería:

$$p \Longrightarrow e \texttt{ \{ action \}}$$

Por ejemplo:

$PLUS(NUM_1, NUM_2) \Longrightarrow NUM_3$ `{ $NUM_3{VAL} = $NUM_1{VAL} + $NUM_2{VAL} }`

cuyo significado es que dondequiera que haya un nódo del AAA que case con el *patrón de entrada* $PLUS(NUM, NUM)$ deberá sustituirse el subárbol $PLUS(NUM, NUM)$ por el subárbol NUM. Al igual que en los esquemas de traducción, enumeramos las apariciones de los símbolos, para distinguirlos en la parte semántica. La acción indica como deben recomputarse los atributos para el nuevo árbol: El atributo `VAL` del árbol resultante es la suma de los atributos `VAL` de los operandos en el árbol que ha casado. La transformación se repite hasta que se produce la *normalización del árbol.*

Las reglas de "casamiento" de árboles pueden ser mas complejas, haciendo alusión a propiedades de los atributos, por ejemplo

$$ASSIGN(LEFTVALUE, x) \textit{ and } \texttt{\{ notlive(\$LEFTVALUE\{VAL\}) \}} \Longrightarrow NIL$$

indica que se pueden eliminar aquellos árboles de tipo asignación en los cuáles la variable asociada con el nodo $LEFTVALUE$ no se usa posteriormente.

Otros ejemplos con variables S_1 y S_2:

$IFELSE(NUM, S_1, S_2)$ and `{ $NUM{VAL} != 0 }` $\Longrightarrow S_1$
$IFELSE(NUM, S_1, S_2)$ and `{ $NUM{VAL} == 0 }` $\Longrightarrow S_2$

Observe que en el patrón de entrada $ASSIGN(LEFTVALUE, x)$ aparece un "comodín": la variable-árbol x, que hace que el árbol patrón $ASSIGN(LEFTVALUE, x)$ case con cualquier árbol de asignación, independientemente de la forma que tenga su subárbol derecho.

Las siguientes definiciones formalizan una aproximación simplificada al significado de los conceptos *patrones árbol* y *casamiento de árboles.*

Patrón Árbol

Definición 7.3.1. *Sea* (Σ, ρ) *un alfabeto con función de aridad y un conjunto (puede ser infinito) de variables* $V = \{x_1, x_2, \ldots\}$. *Las variables tienen aridad cero:*

$$\rho(x) = 0 \ \forall x \in V.$$

Un elemento de $B(V \cup \Sigma)$ *se denomina* patrón *sobre* Σ.

Patrón Lineal

Definición 7.3.2. *Se dice que un patrón es un* patrón lineal *si ninguna variable se repite.*

Definición 7.3.3. *Se dice que un patrón es de tipo* $(x_1, \ldots x_k)$ *si las variables que aparecen en el patrón leidas de izquierda a derecha en el árbol son* $x_1, \ldots x_k$.

Ejemplo 7.3.1. *Sea* $\Sigma = \{A, CONS, NIL\}$ *con* $\rho(A) = \rho(NIL) = 0, \rho(CONS) = 2$ *y sea* $V = \{x\}$. *Los siguientes árboles son ejemplos de patrones sobre* Σ*:*

$$\{\ x, CONS(A, x), CONS(A, CONS(x, NIL)), \ldots\}$$

El patrón $CONS(x, CONS(x, NIL))$ *es un ejemplo de patrón no lineal. La idea es que un patrón lineal como éste "fuerza" a que los árboles* t *que casen con el patrón deben tener iguales los dos correspondientes subárboles* $t/1$ *y* $t/2$.[1] *situados en las posiciones de las variables* [1]

Ejercicio 7.3.1. *Dado la gramática árbol:*

$$S \to S_1(a, S, b)$$
$$S \to S_2(NIL)$$

la cuál genera los árboles concretos para la gramática

$$S \to aSb \mid \epsilon$$

¿Es $S_1(a, X(NIL), b)$ *un patrón árbol sobre el conjunto de variables* $\{X, Y\}$*? ¿Lo es* $S_1(X, Y, a)$*? ¿Es* $S_1(X, Y, Y)$ *un patrón árbol?*

Ejemplo 7.3.2. *Ejemplos de patrones para el AAA definido en el ejemplo* **??** *para el lenguaje Tutu son:*

$$x, y, PLUS(x, y), ASSIGN(x, TIMES(y, ID)), PRINT(y) \ldots$$

considerando el conjunto de variables $V = \{x, y\}$. *El patrón* $ASSIGN(x, TIMES(y, ID))$ *es del tipo* (x, y).

Sustitución

Definición 7.3.4. *Una* sustitución árbol *es una aplicación* θ *que asigna variables a patrones* $\theta : V \to B(V \cup \Sigma)$.

Tal función puede ser naturalmente extendida de las variables a los árboles: los nodos (hoja) etiquetados con dichas variables son sustituidos por los correspondientes subárboles.

$$\theta : B(V \cup \Sigma) \to B(V \cup \Sigma)$$
$$t\theta = \begin{cases} x\theta & si\ t = x \in V \\ a(t_1\theta, \ldots, t_k\theta) & si\ t = a(t_1, \ldots, t_k) \end{cases}$$

Obsérvese que, al revés de lo que es costumbre, la aplicación de la sustitución θ *al patrón se escribe por detrás:* $t\theta$.

También se escribe $t\theta = t\{x_1/x_1\theta, \ldots x_k/x_k\theta\}$ *si las variables que aparecen en* t *de izquierda a derecha son* $x_1, \ldots x_k$.

Ejemplo 7.3.3. *Si aplicamos la sustitución* $\theta = \{x/A, y/CONS(A, NIL)\}$ *al patrón* $CONS(x, y)$ *obtenemos el árbol* $CONS(A, CONS(A, NIL))$. *En efecto:*

$$CONS(x, y)\theta = CONS(x\theta, y\theta) = CONS(A, CONS(A, NIL))$$

Ejemplo 7.3.4. *Si aplicamos la sustitución* $\theta = \{x/PLUS(NUM, x), y/TIMES(ID, NUM)\}$ *al patrón* $PLUS(x, y)$ *obtenemos el árbol* $PLUS(PLUS(NUM, x), TIMES(ID, NUM))$*:*

$$PLUS(x, y)\theta = PLUS(x\theta, y\theta) = PLUS(PLUS(NUM, x), TIMES(ID, NUM))$$

[1] Repase la notación de Dewey introducida en la definición **??**

Casamiento Árbol

Definición 7.3.5. *Se dice que un patrón $\tau \in B(V \cup \Sigma)$ con variables $x_1, \ldots x_k$* casa con un árbol *$t \in B(\Sigma)$ si existe una sustitución de τ que produce t, esto es, si existen $t_1, \ldots t_k \in B(\Sigma)$ tales que $t = \tau\{x_1/t_1, \ldots x_k/t_k\}$. También se dice que τ* casa con la sustitución *$\{x_1/t_1, \ldots x_k/t_k\}$.*

Ejemplo 7.3.5. *El patrón $\tau = CONS(x, NIL)$ casa con el árbol $t = CONS(CONS(A, NIL), NIL)$ y con el subárbol $t.1$. Las respectivas sustituciones son $t\{x/CONS(A, NIL)\}$ y $t.1\{x/A\}$.*

$$t = \tau\{x/CONS(A, NIL)\}$$
$$t.1 = \tau\{x/A\}$$

Ejercicio 7.3.2. *Sea $\tau = PLUS(x, y)$ y $t = TIMES(PLUS(NUM, NUM), TIMES(ID, ID))$. Calcule los subárboles t' de t y las sustituciones $\{x/t_1, y/t_2\}$ que hacen que τ case con t'.*

Por ejemplo es obvio que para el árbol raíz t/ϵ no existe sustitución posible:
$t = TIMES(PLUS(NUM, NUM), TIMES(ID, ID)) = \tau\{x/t_1, y/t_2\} = PLUS(x, y)\{x/t_1, y/t_2\}$
ya que un término con raíz $TIMES$ nunca podrá ser igual a un término con raíz $PLUS$.

El problema aquí es equivalente al de las expresiones regulares en el caso de los lenguajes lineales. En aquellos, los autómatas finitos nos proveen con un mecanismo para reconocer si una determinada cadena "casa"' o no con la expresión regular. Existe un concepto análogo, el de *autómata árbol* que resuelve el problema del "casamiento" de patrones árbol. Al igual que el concepto de autómata permite la construcción de software para la búsqueda de cadenas y su posterior modificación, el concepto de autómata árbol permite la construcción de software para la búsqueda de los subárboles que casan con un patrón árbol dado.

7.4. Ejemplo de Transformaciones Árbol: Parse::Eyapp::TreeRegexp

Instalación

```
[~/jison/jison-aSb(master)]$ sudo cpan Parse::Eyapp
```

Donde

- `[~/src/perl/parse-eyapp/examples/MatchingTrees]$ pwd -P`
  `/Users/casiano/local/src/perl/parse-eyapp/examples/MatchingTrees`
- Parse::Eyapp
- Ejemplo de uso de Parse::Eyapp::Treeregexp
- Tree Matching and Tree Substitution
- Node.pm (Véase el método `s`)

La gramática: Expresiones

```
my $grammar = q{
  %lexer {
      m{\G\s+}gc;
      m{\G([0-9]+(?:\.[0-9]+)?)}gc and return('NUM',$1);
      m{\G([A-Za-z][A-Za-z0-9_]*)}gc and return('VAR',$1);
      m{\G(.)}gcs and return($1,$1);
  }

  %right  '='     # Lowest precedence
```

```
  %left   '-' '+' # + and - have more precedence than = Disambiguate a-b-c as (a-b)-c
  %left   '*' '/' # * and / have more precedence than + Disambiguate a/b/c as (a/b)/c
  %left   NEG     # Disambiguate -a-b as (-a)-b and not as -(a-b)
  %tree           # Let us build an abstract syntax tree ...

  %%
  line:
      exp <%name EXPRESSION_LIST + ';'>
        { $_[1] } /* list of expressions separated by ';' */
  ;

  /* The %name directive defines the name of the
     class to which the node being built belongs */
  exp:
      %name NUM
      NUM
    | %name VAR
      VAR
    | %name ASSIGN
      VAR '=' exp
    | %name PLUS
      exp '+' exp
    | %name MINUS
      exp '-' exp
    | %name TIMES
      exp '*' exp
    | %name DIV
      exp '/' exp
    | %name UMINUS
      '-' exp %prec NEG
    | '(' exp ')'
        { $_[2] }  /* Let us simplify a bit the tree */
  ;

  %%
}; # end grammar
```

Ejecución

El trozo de código:

```
\begin{verbatim}
$parser->input(\"2*-3+b*0;--2\n");       # Set the input
my $t = $parser->YYParse;
```

da lugar a este árbol:

```
[~/src/perl/parse-eyapp/examples/MatchingTrees]$ ./synopsis.pl
Syntax Tree:
EXPRESSION_LIST(
  PLUS(
    TIMES(
      NUM( TERMINAL[2]),
      UMINUS( NUM( TERMINAL[3])) # UMINUS
    ) # TIMES,
    TIMES( VAR( TERMINAL[b]), NUM( TERMINAL[0])) # TIMES
```

```
  ) # PLUS,
  UMINUS(
    UMINUS( NUM( TERMINAL[2])) # UMINUS
  ) # UMINUS
) # EXPRESSION_LIST
```

Al aplicar las transformaciones:

```
# Let us transform the tree. Define the tree-regular expressions ..
my $p = Parse::Eyapp::Treeregexp->new( STRING => q{
    { #  Example of support code
      my %Op = (PLUS=>'+', MINUS => '-', TIMES=>'*', DIV => '/');
    }
    constantfold: /TIMES|PLUS|DIV|MINUS/:bin(NUM($x), NUM($y))
      => {
        my $op = $Op{ref($bin)};
        $x->{attr} = eval  "$x->{attr} $op $y->{attr}";
        $_[0] = $NUM[0];
      }
    uminus: UMINUS(NUM($x)) => { $x->{attr} = -$x->{attr}; $_[0] = $NUM }
    zero_times_whatever: TIMES(NUM($x), .) and { $x->{attr} == 0 } => { $_[0] = $NUM }
    whatever_times_zero: TIMES(., NUM($x)) and { $x->{attr} == 0 } => { $_[0] = $NUM }
  },
  OUTPUTFILE=> 'main.pm'
);
$p->generate(); # Create the tranformations

$t->s($uminus); # Transform UMINUS nodes
$t->s(@all);    # constant folding and mult. by zero
```

Obtenemos el árbol:

```
Syntax Tree after transformations:
EXPRESSION_LIST(NUM(TERMINAL[-6]),NUM(TERMINAL[2]))
```

synopsis.pl

```
[~/src/perl/parse-eyapp/examples/MatchingTrees]$ cat synopsis.pl
#!/usr/bin/perl -w
use strict;
use Parse::Eyapp;
use Parse::Eyapp::Treeregexp;

sub TERMINAL::info {
  $_[0]{attr}
}

my $grammar = q{
  %lexer {
      m{\G\s+}gc;
      m{\G([0-9]+(?:\.[0-9]+)?)}gc and return('NUM',$1);
      m{\G([A-Za-z][A-Za-z0-9_]*)}gc and return('VAR',$1);
      m{\G(.)}gcs and return($1,$1);
  }
```

```
  %right  '='     # Lowest precedence
  %left   '-' '+' # + and - have more precedence than = Disambiguate a-b-c as (a-b)-c
  %left   '*' '/' # * and / have more precedence than + Disambiguate a/b/c as (a/b)/c
  %left   NEG     # Disambiguate -a-b as (-a)-b and not as -(a-b)
  %tree           # Let us build an abstract syntax tree ...

  %%
  line:
      exp <%name EXPRESSION_LIST + ';'>
        { $_[1] } /* list of expressions separated by ';' */
  ;

  /* The %name directive defines the name of the
     class to which the node being built belongs */
  exp:
      %name NUM
      NUM
    | %name VAR
      VAR
    | %name ASSIGN
      VAR '=' exp
    | %name PLUS
      exp '+' exp
    | %name MINUS
      exp '-' exp
    | %name TIMES
      exp '*' exp
    | %name DIV
      exp '/' exp
    | %name UMINUS
      '-' exp %prec NEG
    | '(' exp ')'
        { $_[2] }  /* Let us simplify a bit the tree */
  ;

  %%
}; # end grammar

our (@all, $uminus);

Parse::Eyapp->new_grammar( # Create the parser package/class
  input=>$grammar,
  classname=>'Calc', # The name of the package containing the parser
);
my $parser = Calc->new();                # Create a parser
$parser->input(\"2*-3+b*0;--2\n");       # Set the input
my $t = $parser->YYParse;                # Parse it!
local $Parse::Eyapp::Node::INDENT=2;
print "Syntax Tree:",$t->str;

# Let us transform the tree. Define the tree-regular expressions ..
my $p = Parse::Eyapp::Treeregexp->new( STRING => q{
    { #  Example of support code
```

```
      my %Op = (PLUS=>'+', MINUS => '-', TIMES=>'*', DIV => '/');
    }
    constantfold: /TIMES|PLUS|DIV|MINUS/:bin(NUM($x), NUM($y))
      => {
        my $op = $Op{ref($bin)};
        $x->{attr} = eval  "$x->{attr} $op $y->{attr}";
        $_[0] = $NUM[0];
      }
    uminus: UMINUS(NUM($x)) => { $x->{attr} = -$x->{attr}; $_[0] = $NUM }
    zero_times_whatever: TIMES(NUM($x), .) and { $x->{attr} == 0 } => { $_[0] = $NUM }
    whatever_times_zero: TIMES(., NUM($x)) and { $x->{attr} == 0 } => { $_[0] = $NUM }
  },
  OUTPUTFILE=> 'main.pm'
);
$p->generate(); # Create the tranformations

$t->s($uminus); # Transform UMINUS nodes
$t->s(@all);    # constant folding and mult. by zero

local $Parse::Eyapp::Node::INDENT=0;
print "\nSyntax Tree after transformations:\n",$t->str,"\n";
```

El método s

El código de s está en lib/Parse/Eyapp/Node.pm:

```
sub s {
  my @patterns = @_[1..$#_];

  # Make them Parse::Eyapp:YATW objects if they are CODE references
  @patterns = map { ref($_) eq 'CODE'?
                      Parse::Eyapp::YATW->new(
                        PATTERN => $_,
                        #PATTERN_ARGS => [],
                      )
                      :
                      $_
                  }
                  @patterns;
  my $changes;
  do {
    $changes = 0;
    foreach (@patterns) {
      $_->{CHANGES} = 0;
      $_->s($_[0]);
      $changes += $_->{CHANGES};
    }
  } while ($changes);
}
```

Véase

- Parse::Eyapp
- Ejemplo de uso de Parse::Eyapp::Treeregexp

- Tree Matching and Tree Substitution
- Node.pm (Véase el método `s`)

7.5. Treehugger

Donde

- `[~/srcPLgrado/treehugger(master)]$ pwd -P`
  `/Users/casiano/local/src/javascript/PLgrado/treehugger`
- `[~/srcPLgrado/treehugger(master)]$ git remote -v`
  `origin  git@github.com:crguezl/treehugger.git (fetch)`
  `origin  git@github.com:crguezl/treehugger.git (push)`
- https://github.com/crguezl/treehugger

learning.html

```
[~/srcPLgrado/treehugger(master)]$ cat learning.html
<!DOCTYPE html>
<html>
  <head>
    <title>treehugger.js demo</title>
    <script data-main="lib/demo" src="lib/require.js"></script>
    <link rel="stylesheet" href="examples/style.css" type="text/css" />
  </head>
  <body>
  <h1>Treehugger.js playground</h1>
    <table>
    <tr>
      <th>Javascript</th>
      <th>AST</th>
    </tr>
    <tr>
      <td><textarea id="code" rows="15" cols="42">var a = 10, b;
console.log(a, b, c);</textarea></td>
      <td><textarea id="ast" rows="15" cols="42" readonly style="background-color: #eee;"></te
    </tr>
    <tr>
      <th>Analysis code <button id="runbutton">Run</button></th>
      <th>Output</th>
    </tr>
    <tr>
      <td><textarea id="analysis" rows="15" cols="42">var declared = {console: true};

ast.traverseTopDown(
   'VarDecl(x)', function(b) {
      declared[b.x.value] = true;
   },
   'VarDeclInit(x, _)', function(b) {
      declared[b.x.value] = true;
   },
   'Var(x)', function(b) {
      if(!declared[b.x.value])
```

```
        log("Variable " + b.x.value + " is not declared.");
   }
);
</textarea></td>
      <td><textarea id="output" rows="15" cols="42" readonly style="background-color: #eee;"><
    </tr>
    </table>
  </body>
</html>
```

lib/demo.js

```
[~/srcPLgrado/treehugger(master)]$ cat lib/demo.js
require({ baseUrl: "lib" },
    ["treehugger/tree",
     "treehugger/traverse",
     "treehugger/js/parse",
     "jquery",
     "treehugger/js/acorn", // Acorn is a JavaScript parser
     "treehugger/js/acorn_loose" // This module provides an alternative
                                 // parser with the same interface as
                                 // ‘parse‘, but will try to parse
                                 // anything as JavaScript, repairing
                                 // syntax error the best it can.
    ], function(tree, traverse, parsejs, jq, acorn, acorn_loose) {
          window.acorn_loose = acorn_loose

          function log(message) {
            $("#output").val($("#output").val() + message + "\n");
          }

          function exec() {
            var js = $("#code").val();
            var analysisJs = $("#analysis").val();
            $("#output").val("");

            // https://developer.mozilla.org/en-US/docs/Web/API/Performance.now()
            var t = performance.now();
            var ast = parsejs.parse(js);
            t -= performance.now();
            $("#ast").val(t + "\n" + ast.toPrettyString());
            try {
              eval(analysisJs);
            } catch(e) {
              $("#output").val("JS Error");
              console.log(e.message)
            }
          }

          tree.Node.prototype.log = function() {
            $("#output").val(this.toPrettyString());
          }

          require.ready(function() {
```

```
            $("#code").keyup(exec);
            $("#runbutton").click(exec);
            exec();
        });
    }
);
```

Véase

- treehugger.js is a Javascript library for program processing. It has generic means to represent and manipula
- You can see treehugger.js in action in this simple demo.
- Avoiding JavaScript Pitfalls Through Tree Hugging YouTube. Slides.
- AST traversal javascript libraries
- RequireJS

7.6. Práctica: Transformaciones en Los Árboles del Analizador PL0

Partimos del código realizado en la práctica *Análisis de Ámbito en PL0* 5.7.

Modifique el árbol generado por el código de esa práctica usando las transformaciones de *constant folding* o *plegado de las constantes*:

$PLUS(NUM_1, NUM_2) \Longrightarrow NUM_3$ `{ $NUM_3{VAL} = $NUM_1{VAL} + $NUM_2{VAL} }` $MINUS(NUM_1, NU$

NUM_3 `{ $NUM_3{VAL} = $NUM_1{VAL} - $NUM_2{VAL} }` $TIMES(NUM_1, NUM_2) \Longrightarrow NUM_3$ `{ $NUM_3{VAL}`

$DIV(NUM_1, NUM_2) \Longrightarrow NUM_3$ `{ $NUM_3{VAL} = $NUM_1{VAL} / $NUM_2{VAL} }`

etc.

Opcionalmente si lo desea puede considerar otras transformaciones: $TIMES(X, NUM_2)$ and

`{ $NUM_2{VAL} =` 2^s para algún s `}` $\Longrightarrow SHIFTLEFT(X; NUM_3)$ `{ $NUM_3{VAL} = s }`

Índice general

Índice de figuras

Índice de cuadros

Índice alfabético

árbol de análisis abstracto, 154
árbol de análisis sintáctico concreto, 53
árbol sintáctico concreto, 51, 91
árboles, 154

AAA, 154
abstract syntax tree, 154
access link, 115
acción de reducción, 105
acciones de desplazamiento, 105
acciones semánticas, 58
acciones shift, 105
alfabeto con función de aridad, 154
algoritmo de construcción del subconjunto, 104
antiderivación, 100
AST, 154
atributo heredado, 135, 139
atributo sintetizado, 58, 135, 139
atributos formales, 138
atributos heredados, 135, 136, 139
atributos intrínsecos, 139
atributos sintetizados, 135, 139
autómata árbol, 161
autómata finito determinista, 104
autómata finito no determinista con ϵ-transiciones, 102

bubble phase, 43

callback, 21
capture phase, 43
casa con la sustitución, 161
casa con un árbol, 161
casamiento de árboles, 159
clausura, 104
conflicto de desplazamiento-reducción, 106, 129
conflicto reduce-reduce, 106, 129
conflicto shift-reduce, 106, 129
constant folding, 168

definición dirigida por la sintáxis, 138
deriva en un paso en el árbol, 155
DFA, 104
DOM storage, 29

Ejercicio
    Recorrido del árbol en un ADPR, 57
esquema de traducción, 58, 134
esquema de traducción árbol, 159
evaluation stack, 115

favicon, 34
Favorite icon, 34
función de aridad, 154
función de transición del autómata, 104

generador de generadores de código, 158
goto, 105
grafo de dependencias, 139
gramática árbol regular, 154
gramática atribuída, 139
gramática es recursiva por la izquierda, 57, 72

handle, 101

INI, 40
IR, 157
items núcleo, 109

JavaScript Object Notation, 26
jQuery, 18
JSON, 26

L-atribuída, 139
LALR, 107
lenguaje árbol generado por una gramática, 155
lenguaje árbol homogéneo, 154
lenguaje de las formas sentenciales a rderechas, 101
local storage, 30
LR, 100

manecilla, 101
mango, 101

NFA, 102
normalización del árbol, 159

orden parcial, 139
orden topológico, 139

parsing expression, 64
parsing expression grammar, 64
patrón, 159
patrón árbol, 159

patrón de entrada, 159
patrón lineal, 159
patrones árbol, 159
pattern matching, 31
PEG, 64
plegado de las constantes, 168
Práctica
    Ambiguedad en C++, 87
    Análisis de Ámbito en PL0, 111
    Analizador de PL0 Ampliado Usando PEG.js, 87
    Analizador de PL0 Usando Jison, 110
    Analizador Descendente Predictivo Recursivo, 59
    Analizador Léxico para Un Subconjunto de JavaScript, 49
    Calculadora con Análisis de Ámbito, 113
    Calculadora con Funciones, 112
    Calculadora con Listas de Expresiones y Variables, 100
    Comma Separated Values. CSV, 13
    Conversor de Temperaturas, 9
    Ficheros INI, 41
    Inventando un Lenguaje: Tortoise, 89
    Palabras Repetidas, 37
    Secuencia de Asignaciones Simples, 95
    Traducción de Infijo a Postfijo, 112
    Transformaciones en Los Árboles del Analizador PL0, 168
Primeros, 103

recursion, 115
recursiva por la derecha, 80
recursiva por la izquierda, 58, 72
recursive descent parser, 65
reducción por defecto, 121
reducción-reducción, 106, 129
reentrant, 115
register spilling, 115
reglas de evaluación de los atributos, 138
reglas de transformación, 159
reglas semánticas, 138
Representación intermedia, 157
rightmost derivation, 100

S-atribuída, 139
selección de código, 157
session storage, 30
shortcut, 34
siguientes, 103
SLR, 105, 106
static link, 115
sustitución árbol, 160

términos, 154
tabla de acciones, 105
tabla de gotos, 105
tabla de saltos, 105
target phase, 43
text area, 29

Web storage, 29

Bibliografía

[1] Mark Pilgrim. *Dive into HTML5*. http://diveinto.html5doctor.com/index.html, 2013.

[2] G. Wilson and A. Oram. *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly Media, 2008.

[3] Nathan Whitehead. *Create Your Own Programming Language. http://nathansuniversity.com/*. 2012.

[4] Nathan Whitehead. *What's a Closure?. http://nathansjslessons.appspot.com/*. 2012.

[5] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[6] T. Mogensen and T.A. Mogensen. *Introduction to compiler design*. Undergraduate topics in computer science. Springer London, Limited, 2011.

[7] Todd A. Proebsting. Burg, iburg, wburg, gburg: so many trees to rewrite, so little time *(invited talk)*. In *ACM SIGPLAN Workshop on Rule-Based Programming*, pages 53–54, 2002.