

Una introducción a Parrot

BONITO LORITO

Parrot es una herramienta todo-en-uno para el desarrollo y ejecución de nuevos lenguajes de programación.

Perl 6 se ejecuta en Parrot; hay muchas probabilidades de que nuestro lenguaje de programación se ejecute en él también. **POR MARTIN STREICHER**

Está cansado de trabajar en el mismo viejo código día tras día? ¿Su lenguaje de programación es un armatoste? ¿Necesita un impulso para entrar en la autopista de la información? No sufra más. El reino de los programadores tiene soluciones para todos. Ha llegado PHP 5: práctico y estupendo para la web con aceleración de bytecode. O Ruby 1.9: Atractivo, recién salido de fábrica y un verdadero encanto para *la nube*. ¿Quiere ahorrar en costes? Llévase lenguajes de programación certificados, pre-propietarios. Perl 5.8 ya tiene precio y está listo para su venta. El reino del programador, donde cada programador es el rey.

Puede que el reino del programador no exista, pero la elección de un lenguaje de programación es notablemente similar a la compra de un coche. Igual que un coche, cada lenguaje de programación está diseñado con una intención, ya sea ésta su utilidad, velocidad o tamaño. Como un coche, un lenguaje puede ofrecer características exóticas y punteras. La elección de un lenguaje es una inversión significativa, también, con consecuencias poco deseables e incluso desastrosas en caso de hacer una elección equivocada. Además, cada lenguaje incluye partes obligatorias (literales, variables, subrutinas y control de flujos) igual que las cuatro gomas, el volante o el parabrisas de un coche.

De hecho, la mayor parte del tiempo empleado en el desarrollo de un nuevo lenguaje de programación se pasa reinventando la rueda. La fuente del idioma debe descomponerse con un parser, componer en tokens, aunar en declaraciones, abstraer en árboles de sintaxis y, finalmente, o bien interpretarlo o bien convertirlo en algo ejecutable como bytecode o binarios.

Es más, incluso una vez construido, un nuevo lenguaje de programación está limitado, a no ser que proporcione una balsa de funcionalidades supletorias, como expresiones regulares o interfaces para las llamadas al sistema. Se puede evitar parte de la sobrecarga construyendo fachadas alrededor de librerías de C, como la librería PCRE (*Perl Compatible Regular Expression*), pero eso

requiere al menos cierto esfuerzo. E irónicamente, lenguaje tras lenguaje se acaba haciendo el mismo trabajo debido a la escasez de alternativas.

Sin embargo hay excepciones: La iniciativa de Microsoft .NET permite que muchos lenguajes (*C#, Visual Basic, C++* y también lenguajes de código abierto) compartan código debido a que cada uno se compila en última instancia a un objetivo común de bajo nivel, CIL (*Common Intermediate Language*) [1]. JRuby es otra excepción: JRuby ejecuta código Ruby sobre una máquina virtual de Java (JVM).

Aún, la vasta mayoría de lenguajes de programación permanecen aislados, sería como tratar de poner un Hemi dentro de un Honda.

Un Método Unificado

Para acelerar el desarrollo de nuevos lenguajes, facilitar la compartición de código entre todos los lenguajes y agilizar la ejecución de lenguajes dinámicos como Perl, Ruby y Python, el equipo de



Dimitry Pichugin, Fotolia

desarrollo de Parrot ha creado una máquina virtual y una suite de herramientas para compilar y ejecutar (posiblemente) cualquier lenguaje de programación. Parrot [2] fue concebido inicialmente como motor para Perl 6, pero resultó ser adaptable a una gran cantidad de clases de códigos diferentes. Según el propio sitio web de Parrot, “En teoría, podremos escribir una clase en Perl, hacer una subclase de ella en Python y luego instanciar y usar la subclase en un programa Tcl”.

Parrot proporciona todas las herramientas necesarias para convertir el código fuente en programas ejecutables.

- El PGE (*Parser Grammar Engine*) define la sintaxis del lenguaje de programación a través de reglas (*rules*). Una regla podría definir la sintaxis de una declaración de asignación, mientras que otra podría definir la estructura que ha de tener una subrutina. PGE parsea la entrada, produce los tokens y trata de encontrar, en el flujo de tokens, coincidencias a partir de las reglas. Una coincidencia indica una sentencia válida para el lenguaje.
- Cada regla, a su vez, puede llamar a una o más acciones (*actions*). Al escribir un nuevo lenguaje de programación, la semántica del lenguaje se expresa como acciones. Cada acción transforma una coincidencia en una estructura de datos independiente del lenguaje llamada PAST (*Parrot Abstract Syntax Tree*). PAST representa cada declaración con un árbol: los operan-

```
Source

if (a < b) then
  print "A less than B"
else
  print "B greater than or equal to A"
end
```

Figura 1: Código fuente de una declaración *if* de tipo C.

dos son las hojas; y los operadores, incluidas las estructuras de control, son las ramas. Acumulativamente, el programa entero se representa en última instancia como un árbol PAST con el nodo especial TOP como raíz.

- El PIR (*Parrot Intermediate Representation*) se puede escribir a mano fácilmente y es la salida típica de un compilador con Parrot como objetivo. Abstrae ciertos detalles de más bajo nivel a fin de facilitarle el trabajo a los desarrolladores de programas y de compiladores.
- El PASM (*Parrot Assembly*) es más mecánico que el PIR. Se puede codificar manualmente y es generable desde un compilador.

- La representación de más bajo nivel es el PBC (*Parrot Byte Code*). Aunque se podría ver como código máquina, éste no se ejecuta en el hardware. En lugar de eso, es Parrot quien ejecuta el PBC.

Parrot puede ejecutar archivos PBC, PASM, PIR y PAST. PAST no es apto para la codificación manual, pero es ideal para integrar herramientas entre sí.

Un compilador tradicional, como el omnipresente GCC, normalmente genera un árbol de sintaxis abstracto como

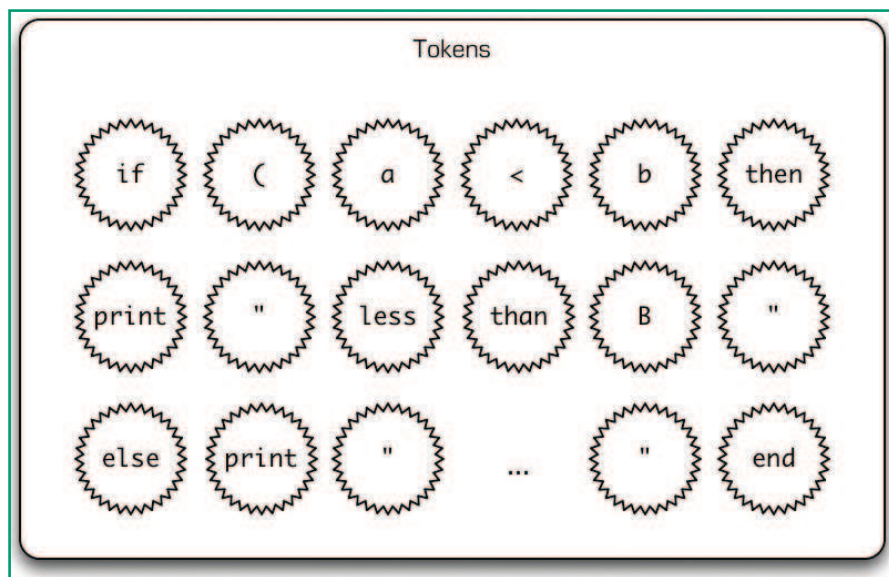


Figura 2: Código fuente agrupado en tokens.

Reglas

Una regla es como un formulario de los que se rellenan en la consulta del médico. Algunos campos ya están escritos en el formulario y sólo sirven para proporcionar algo de contexto; otros están en blanco y se deben rellenar. En la regla *if_statement*, las subreglas *expression*, *block* y (el alias para el segundo bloque) *else* son los campos vacíos. Como mencionamos en el debate acerca del objeto de Parrot para un *if-then-else*, estas subreglas proporcionan los valores usados para instanciar el objeto.

Listado 1: Una Sintaxis de Definición de Regla

```
01 rule if_statement {
02   'if' <expression> 'then'
   <block>
03   ['else' <else=block>]?
   'end'
04   {*}
05 }
```

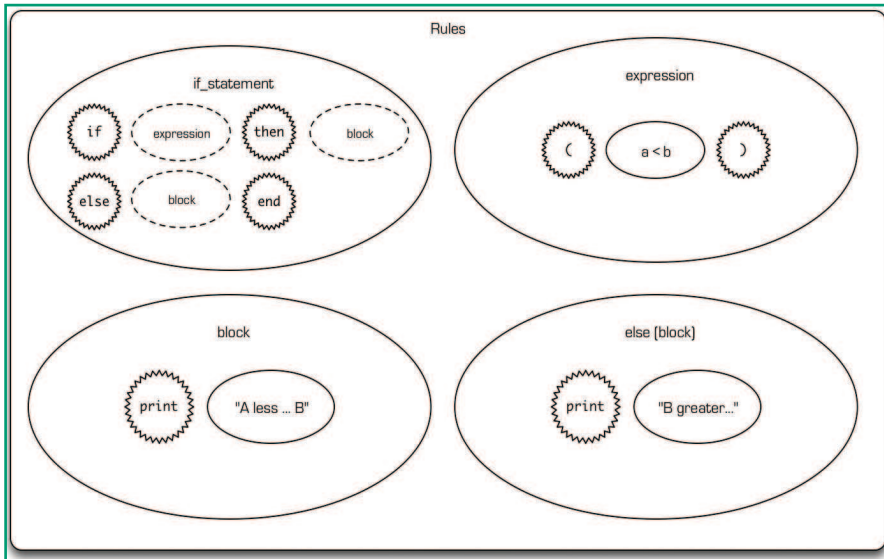


Figura 3: Los tokens de la Figura 2 formando reglas.

forma intermedia para la deducción y aplicación de optimizaciones. La adaptación de dicho árbol a PAST es sencilla y, puesto que Parrot puede leer PAST, conectar un compilador existente con Parrot es una tarea fácil.

Además de estos componentes, Parrot proporciona también varios nodos PAST (se ha de ver un nodo como una plantilla o un objeto) que representan las características fundamentales de cada lenguaje de programación.

Por ejemplo, conceptualmente Parrot incluye un nodo para *if-then-else*, construcción que se puede encontrar en todos los lenguajes. Para añadir este nodo al árbol de mayor tamaño de nuestro programa se ha de proporcionar una condición, un bloque de declaraciones a

ejecutar en caso de cumplirse la condición y, opcionalmente, un bloque de declaraciones a ejecutar en caso de no cumplirse ésta. No importa qué aspecto tenga la sintaxis original, el nodo *if* cap-

tura las semánticas de dicha declaración.

Otros nodos PAST se encargan de otras características: una variable, incluyendo el ámbito y el tipo; un bucle; y operadores, incluidos los aritméticos y las llamadas a subrutinas.

Y es que ése es el verdadero potencial de Parrot. La mayoría de las características de un lenguaje no hay por qué reinventarlas, ya que Parrot proporciona a todo el mundo unas útiles abstracciones que permiten reusarlas, independientemente de la sintaxis usada. Aún así, si un lenguaje inventase algo verdaderamente novedoso, siempre se puede implementar un nuevo nodo PAST para conseguir sus semánticas.

Las Figuras 1 a la 5 ilustran el proceso de conversión a un nodo PAST de una declaración *if* al estilo de C. La Figura 1 muestra el código fuente; de este modo, se trata de una secuencia de caracteres que podrían o no ajustarse a la sintaxis correcta.

La Figura 2 muestra el código fuente descompuesto en tokens de acuerdo a las reglas del parser. Llegados a este punto, los caracteres individuales carecen de importancia; en su lugar, los conjuntos son ahora la moneda de cambio.

La Figura 3 ilustra las reglas. Cada regla se encarga de un trozo de gramática correcta. Cuando una colección de tokens concuerda con una regla, se reúnen conforme a esa regla, otro tipo de amalgama.

Cada paso de aglomeración del proceso de compilación acerca un poco más la sintaxis a las semánticas. La Figura 4

Listado 2: Una Acción Asociada a una Regla

```
01 method if_statement($/) {
02   my $cond :=
03     $<expression>.ast;
04   my $then := $<block>.ast;
05   my $past := PAST::Op.new (
06     $cond, $then, :pasttype('if'),
07     :node($/) );
08   ## de haber un else, se
09   añadiría al nodo PAST
10   if $<else> {
11     $past.push( $<else>[0].ast
12   );
13 }
14 make $past;
15 }
```

Listado 3: Un Programa PIR

```
01 .sub main
02 # Definimos el número de
03   cuadrados a sumar
04 .local int maxnum
05 maxnum = 10
06 # Usaremos algunos registros
07   con nombre.
08 # Nótese que podemos declarar
09   muchos registros del mismo
10   tipo en una misma línea.
11 .local int i, total, temp
12 total = 0
13 # Bucle para realizar la
14   suma.
15 i = 1
16 loop:
17   temp = i * i
18   total += temp
19   inc i
20   if i <= maxnum goto loop
21 # Mostrar el resultado.
22 print "La suma de los
23   primeros "
24 print maxnum
25 print " cuadrados es "
26 print total
27 print ".\n"
28 .end
```

Listado 4: Fuentes PASM

```
01 main:
02 set I0, 10
03 set I1, 0
04 set I3, 1
05 loop:
06 mul I2, I3, I3
07 add I1, I2
08 inc I3
09 le I3, I0, loop
10 print "La suma de los
11   primeros "
12 print I0
13 print " cuadrados es "
14 print I1
15 print ".\n"
16 returncc
```

muestra la acción de la declaración *if* (*if_statement*). Cada amalgama de la Figura 3 ya ha sido reducida a su (sub)acción y está siendo amasada para el consiguiente procesamiento. A partir de aquí, cada unidad tiene un propósito, pero la lógica de la condición aún no se ha llevado a cabo.

La Figura 5 muestra el ensamblaje final después de la acción correspondiente a la declaración *if*. El nodo PAST captura el

propósito y la operación de la declaración en una estructura de datos. El paso final, que no se muestra, recorre el PAST (de arriba a abajo y de izquierda a derecha) y genera el código apropiado. El código cargaría la primera variable, luego la segunda, y realizaría una comparación. La comparación vendría seguida entonces de la continuación de la ejecución secuencial (con el siguiente código, que implementa una comparación satisfactoria, imprimiendo “A

es menor que B”) o determina que es falsa (imprimiendo “B es mayor o igual que A”).

Programando desde Parrot

El Listado 1 presenta un ejemplo concreto para los datos expuestos más arriba. Este ejemplo muestra gramática para *Squaak*, un lenguaje de programación de demostración incluido con el código fuente de Parrot. En particular, el Listado 1 es una regla que define la sintaxis de una declaración *if-then-else* en *Squaak*. Una declaración válida debe constar de: un literal inicial, el *if*; una expresión, que viene definida en su propia regla epónima (no se muestra); la palabra clave *then*; un bloque de declaraciones, definidas también en su propia regla; y la palabra clave *end*.

La frase [*else* <*else = block*>]? es opcional, tal y como indica la interrogación al final, y significa “una o ninguna” coincidencias. Si la frase no aparece, debe incluir la palabra clave *else* seguida de otro bloque. La notación <*else = block*> simplemente asigna un alias llamado *else* al bloque de la sección para diferenciarlo del primero.

En caso de coincidir esta regla, se llama a una acción al final, que es el propósito del marcador {***} situado al final de la regla.

Una acción no es más que una subrutina. Por defecto, Parrot llama a la acción que tiene el mismo nombre que la regla coincidente. Es más, llama a la acción proporcionándole argumentos, un argumento por cada subregla.

El Listado 2 muestra la acción asociada con la regla de la declaración *if*. Aún teniendo una descripción elaborada, su funcionamiento debería ser bastante obvio. Dados una condición y un bloque a ejecutar en caso de cumplirse la condición, el código creará un PAST para la declaración. La declaración final será quien haga el trabajo duro por nosotros.

Con el Listado 2 en mente, el siguiente listado es la acción para la declaración de asignación, tal como $x = 10$:

```
method assignment($/) {
    my $rhs := ⤵
    $<expression>.ast;
    my $lhs := $<primary>.ast;
    $lhs.lvalue(1);
    make PAST::Op.new ( $lhs, ⤵
    $rhs, :pasttype('bind'), ⤵
    :node($/) );
}
```

Compilar Parrot

La compilación de Parrot es sorprendentemente sencilla, suponiendo que ya se cuente con las herramientas de desarrollo habituales en un sistema Linux. Disponiendo de Perl 5, el compilador de GNU (GCC) y Subversion (o si se tiene *Xcode* instalado en Mac OS X), bastan unos pocos minutos para completar el proceso completo.

El primer paso consiste en obtener el último código mediante Subversion:

```
$ svn co https://svn.parrot.org/parrot/trunk parrot
$ cd parrot
```

Luego, configuramos la compilación con el script especial de Parrot *Configure.pl*. Normalmente será suficiente con la configuración predeterminada del script. Sin embargo, si se quiere personalizar la compilación, al ejecutar *Configure.pl --help* podremos ver el listado completo de opciones. La mayoría de éstas son generales, como la opción de instalación *--prefix*, pero hay un montón de ellas que afectan a las características de Parrot, como por ejemplo la estrategia a usar en la recolección de basura.

```
$ perl Configure.pl
Parrot Version 1.2.0 Configure 2.0
Copyright (C) 2001-2009, Parrot Foundation.
Hello, I'm Configure. My job is to poke and prod your system to figure
out how to build Parrot. The process is completely automated, unless you
passed in the <C>=ask<C> flag on the command line, in which case I'll
prompt you for a few pieces of info.
```

```
Since you're running this program, you obviously have Perl
5—I'll be pulling some defaults from its configuration.
```

```
...
```

```
Now you can run make to build your Parrot.
```

```
After that, you can use make test to run the test suite
```

Entonces ya podremos usar *make* para compilar Parrot. Después de eso, con *make test* es posible ejecutar la suite de prueba. De este modo nos aseguramos de que la compilación es estable:

```
$ make
$ make test
```

Suponiendo que la prueba tenga éxito, podemos proceder a instalar las herramientas de Parrot en el sistema:

```
$ sudo make install
```

Por defecto, el software de Parrot se instala en */usr/local*. Para instalarlo en el directorio de inicio, hay que especificarlo mediante *perl Configure.pl --prefix=\$HOME/parrot* en el paso correspondiente a la configuración. La instalación añade ocho utilidades al sistema. La más importante es *parrot*, la máquina virtual de parrot. Para probarla usaremos los Listados 3 y 4, así como los comandos mostrados más arriba.

Una vez estemos en posesión del código fuente de Parrot, podemos mantenerlo al día mediante *svn update*, eso sí, con una advertencia: se debe hacer *make realclean* antes de cualquier actualización.

```
$ make realclean
$ svn update
```

De no usar *realclean*, Parrot se podría romper o sufrir extraños bugs poco reproducibles.

El Listado 3 muestra un ejemplo de programa PIR tomado del sitio web de Parrot. La aplicación suma el cuadrado de los números del 1 al 10. En el código, el sangrado se usa para una mayor legibilidad, pero no las semánticas (dicho de otro modo, el espacio en blanco no está activo).

El Listado 3 parece lenguaje ensamblador, pero con ciertas excepciones convenientes para el PIR. Específicamente, `temp = i * i` es una abreviatura para `mul temp, i, i`. El código `.local int i, total, temp` hace uso de nombres amigables para las tres variables, en vez de registros, y de `.local` para especificar un ámbito. El operador de asignación (`=`) es un alias para `set`. Y `goto` es otra de esas conveniencias cuyo equivalente sería `le i, maxnum, loop`.

Para ejecutar el código mostrado en el Listado 3, se puede interpretar el código PIR o bien compilarlo a PBC y ejecutar el PBC en su lugar (consultar el cuadro "Compilando Parrot" para saber cómo se ha de compilar Parrot).

```
$ parrot sum.pir
La suma de los primeros 10 ➤
cuadrados es 385.
$ parrot -o sum.pbc sum.pir
$ parrot sum.pbc
La suma de los primeros 10 ➤
cuadrados es 385.
```

Parrot puede emitir fuentes PASM, si se tiene curiosidad por saber cómo traduce

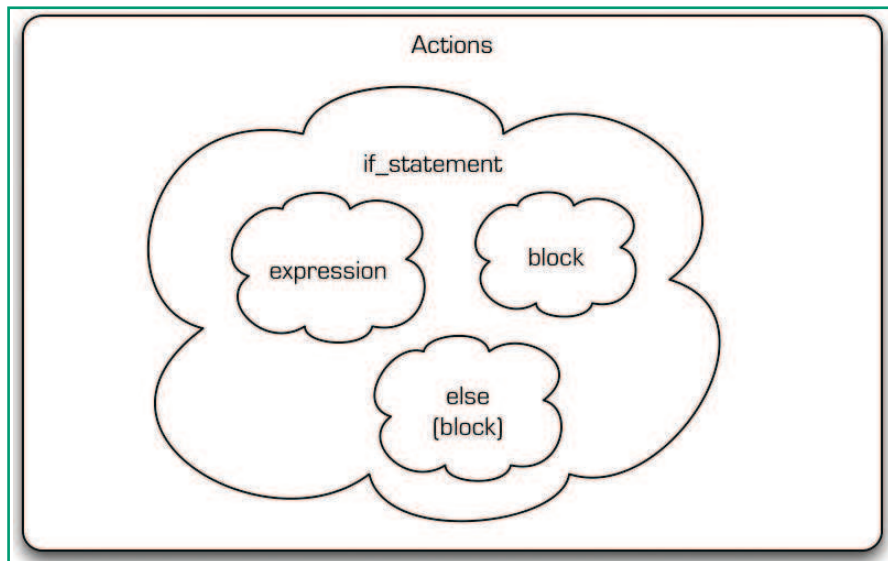


Figura 4: Una acción prepara los datos asociados a una regla coincidente para transformarlos en un PAST. En esta imagen se amasa la declaración `if_statement`.

PIR. Para generar PASM, sustituimos `-o algunarchivo.pbc` por `-o algunarchivo.pasm:`

```
$ parrot -o sum.pasm sum.pir
```

El Listado 4 muestra el equivalente PASM del Listado 3.

Parrot puede también ejecutar código PASM directamente:

```
$ parrot sum.pasm
```

La suma de los primeros 10 cuadrados es 385.

Un Pájaro Charlatán

Haciendo honor a su nombre, Parrot habla multitud de lenguajes [3].

Rakudo Perl 6 [4] es una implementación de Perl 6 sobre Parrot. Cardinal [5] es una versión de Ruby 1.0, y Pipp [6] es una versión de PHP.

Hay proyectos llevando Java, Lua y SmallTalk al motor de Parrot, así como otros 40 proyectos listados en el índice.

En el momento de enviar este número de Linux Magazine a la imprenta, la última versión de Parrot es la 1.9.0. El software es estable y apto tanto para su uso como lenguaje de programación como para investigación y desarrollo sobre máquinas virtuales. Si necesita un lenguaje para un dominio específico o si desea ampliar un lenguaje de programación existente, conviene probar Perl 6 desde ya.

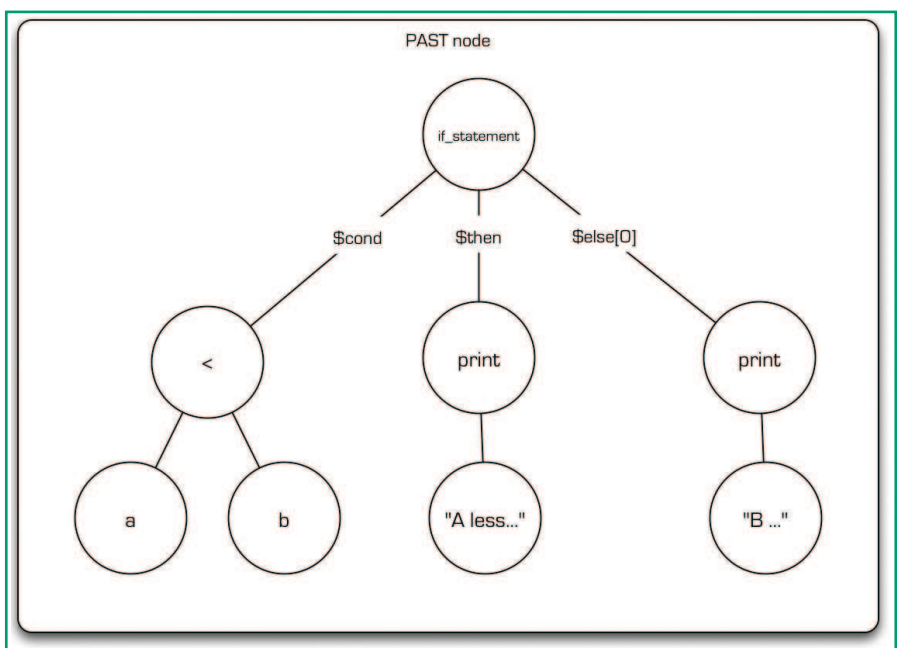


Figura 5: En última instancia, la declaración `if` produce un PAST, el cual es apto para optimización y generación de código.

RECURSOS

- [1] Lenguaje Común Intermedio: http://es.wikipedia.org/wiki/Common_Intermediate_Language
- [2] Página de inicio de Parrot: <http://www.parrot.org/>
- [3] Implementaciones de lenguajes basadas en Parrot: <http://www.parrot.org/languages>
- [4] Rakudo Perl 6 en Parrot: <http://www.rakudo.org>
- [5] Cardinal, Ruby para Parrot: <http://github.com/cardinal/cardinal/tree/master>
- [6] Pipp, PHP para Parrot: <http://wiki.github.com/bschmalhofer/pipp/>