Lenguajes de Programación
Nota de clase 2: Generación de Código Ejecutable

Karla Ramírez Pulido Manuel Soto Romero

7 de octubre de 2021
Facultad de Ciencias UNAM

2.1. Generación de Código Ejecutable

Antes de obtener código directamente ejecutable por una computadora el código escrito en un determinado lenguaje de programación pasa por un conjunto de fases que analizan, modifican y optimizan el código que una computadora puede entender. Algunas de estas fases son: [1]

Análisis léxico Consiste en descomponer una cadena en lexemas[1] que pueden procesarse individualmente o en conjunto con el fin de hacer un análisis más detallado. El programa encargado de realizar este análisis es llamado analizador léxico (*lexer*) y produce una estructura (generalmente una lista) con cada uno de los lexemas correspondientes.

Análisis sintáctico Se encarga de revisar que el programa escrito en un lenguaje de programación cumpla con las reglas sintácticas establecidas por la gramática del lenguaje. Si el programa es sintácticamente correcto, se construye entonces una representación intermedia que puede entender la computadora. Por lo general, esta conversión se realiza a *Árboles de Sintaxis Abstracta* (ASA). El programa encargado de realizar este análisis es llamado analizador sintáctico (*parser*).

Análisis semántico Se encarga de revisar que el programa *tenga sentido* y así darle significado a lo escrito por el programador. Este tipo de revisión o verificación suele hacerse sobre las operaciones que se aplican, el sistema de tipos, entre otras. El programa encargado de realizar este análisis es llamado analizador semántico. Una vez que estamos seguros de que el programa tenga sentido, entonces se procederá ya sea a devolver el resultado de ejecutar el programa (intérprete) o se continuará con la generación de código máquina (compilador).

Transformaciones Consiste en transformar el código escrito por el programador a otra representación o lenguaje con el fin de cambiar su estructura y facilitar alguno de los análisis anteriores y/o posteriores, por ejemplo, eliminar azúcar sintáctica, verificar tipos, etcétera.

Las optimizaciones consisten en realizar transformaciones al código en tiempo de compilación para mejorar el rendimiento de un programa, sea en tiempo o en espacio. Algunos ejemplos de este tipo de optimizaciones son: la técnica de recursión de cola, la técnica de paso de continuaciones, la técnica de memoización, entre otras.

[1] Unidad mínima con significado léxico que no presenta morfemas gramaticales.

2.2. Expresiones Aritméticas

En esta sección diseñaremos nuestro primer lenguaje de programación llamado AE (*Arithmetic Expressions*) que únicamente tendrá operaciones de suma y resta así como evaluación de números.

2.2.1. Sintaxis concreta vs. Sintaxis abstracta

Para comenzar el diseño de un lenguaje de programación, debemos definir primero que nada su sintaxis, es decir, la forma en que el programador escribirá programas. Esta sintaxis se define usualmente mediante gramáticas libres de contexto. A continuación se muestra la gramática para AE, en notación EBNF[2]: [1]

```
<expr> ::= <num>
         | {+ <expr> <expr>}
         | {- <expr> <expr>}
```

Algunos ejemplos de expresiones AE

- `1729`
- `{+ 17 29}`
- `{+ {- 18 35} {+ 17 29}}`

La gramática de un lenguaje de programación indica cómo se deben escribir las expresiones en un programa y forma la llamada ***sintaxis concreta*** que representa lo que el programador escribe antes de realizar el proceso de generación de código ejecutable. Siempre debemos de garantizar que esta expresión no sea ambigua, es decir, que dada una expresión no exista más de una derivación para la misma cadena. En este sentido, nuestra gramática no es ambigua pues nuestros operadores son prefijos y se encuentran delimitados con llaves de forma tal que siempre sabemos qué operación debe realizarse primero.

La gramática, nos permite producir cadenas, sin embargo, el hecho de manejar cadenas es útil a los programadores pero no muy entendible para la computadora pues lo único que ve son un conjunto de símbolos colocados en secuencia sin saber donde inicia o termina una expresión, si tiene sub-expresiones entre muchos otros detalles relevantes. Para solucionar este problema, se proporciona una ***sintaxis abstracta*** que pueda procesar de manera más sencilla la computadora. Usualmente se define esta sintaxis mediante *Árboles de* ***Sintaxis Abstracta (ASA)***.

En este caso implementaremos un intérprete para AE usando Racket. Decimos entonces que AE es el ***lenguaje objetivo*** *y* Racket el ***lenguaje anfitrión***. Dicho esto, procedemos a definir los árboles de sintaxis abstracta en Racket, a través del dialecto `plai`, mediante la primitiva `define-type` tal y como se muestra en el Código 1. [1]

```
1 (define-type AE
2    [num (n number?)]
3    [add (izq AE?) (der AE?)]
4    [sub (izq AE?) (der AE?)])
```

Código 1: Sintaxis abstracta de AE

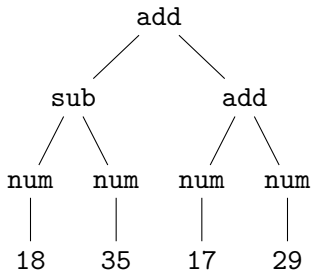
[2]Del inglés, *Extended Backus Naur Form,* el cuál es una forma de representar gramáticas de Lenguajes de Programación.

El tipo de dato **AE** del Código 1 representa la sintaxis abstracta de AE mediante un *Árbol de Sintaxis Abstracta* (ASA). Cada uno de los constructores del Código 1 representa la raíz del árbol y los parámetros representan los hijos de cada una de éstas. Al ser un tipo recursivo, cada raíz tiene a su vez subárboles de tipo **AE**.

Por ejemplo, esta es la sintaxis abstracta de las expresiones definidas al inicio de esta sección:

- `(num 1729)`
- `(add (num 17) (num 29))`
- `(add (sub (num 18) (num 35)) (add (num 17) (num 29)))`

La última expresión se vería visualmente en forma de árbol cómo:



2.2.2. Análisis Sintáctico

El programa encargado de transformar la sintaxis concreta en un ASA es el analizador sintáctico. La función **parse** del Código 2 muestra la implementación correspondiente. [1]

Observación 1. Se omite el análisis léxico pues RACKET cuenta con las primitivas **quote** y **read** que simplifican este proceso con respecto a nuestro lenguaje AE. Ambas primitivas toman una expresión y en lugar de evaluarla, devuelven un símbolo, número o lista de símbolos según corresponda. Por ejemplo, al escribir **{+ 1 2}** se obtiene la lista '**{+ 1 2}** en lugar del resultado 3.

```
1 ;; parse: s-expression -> AE
2 (define (parse sexp)
3    (cond
4        [(number? sexp) (num sexp)]
5        [(list? sexp)
6            (case (car sexp)
7                [(+) (add (parse izq) (parse der))]
8                [(-) (sub (parse izq) (parse der))])]))
```

Código 2: Análisis sintáctico de AE

La línea 4 del Código 2, verifica si el resultado obtenido del análisis léxico (con **quote** o **read**) es un número en cuyo caso se construye un árbol mediante el constructor **num**. Por otro lado, en las líneas 6 a 8 se construyen árboles que representan una suma o resta con el análisis sintáctico de sus respectivos subárboles. A continuación se muestra la ejecución de la función **parse** con algunas expresiones en sintaxis concreta.

```
;; Mediante quote
> (parse (quote 1729))
(num 1729)
> (parse (quote {+ 18 35}))
(add (num 18) (num 35))
> (parse (quote {+ {- 40 5} {- 50 2}}))
(add (sub (num 40) (num 5)) (sub (num 50) (num 2)))

;; Mediante read
> (parse (read))
1729
(num 1729)
> (parse (read))
{+ 18 35}
(add (num 18) (num 35))
> (parse (read))
{+ {- 40 5} {- 50 2}}
(add (sub (num 40) (num 5)) (sub (num 50) (num 2)))
```

2.2.3. Análisis Semántico

Dado que nuestro lenguaje es lo suficiente simple, el análisis semántico termina siendo simplemente un evaluador. Implementaremos este evaluador mediante la función `calc` del Código 3 que realiza los cálculos correspondientes con las expresiones aritméticas dadas.

```
1 ;; calc: AE -> number
2 (define (calc expr)
3    (type-case AE calc
4       [num (n) n]
5       [add (izq der) (+ (calc izq) (calc der))]
6       [sub (izq der) (- (calc der) (calc der))]))
```

Código 3: Análisis semántico de AE

Del Código 3 destacamos que en la línea 4, se recibe un número y se regresa el número que representa. En las líneas 5 y 6 se expresan los casos cuando se recibe una suma o resta formadas por un lado izquierdo y derecho, las cuales aplican la operación correspondiente a la llamada recursiva de cada uno de los lados.

A continuación se muestra la ejecución de la función `calc` con algunas expresiones en sintaxis concreta.

```
> (interp (parse (quote 1729)))
1729
> (interp (parse (quote {+ 18 35})))
53
> (interp (parse (quote {+ {- 40 5} {- 50 2}})))
83
```

2.3. Sustitución: El Lenguaje WAE

Uno de los conceptos más importantes a lo largo de estas notas, el cual se estudia también en los cursos de Estructuras Discretas y Lógica Computacional, es el de sustitución. Este consiste en cambiar los identificadores de una expresión por un valor dado. Para estudiar este concepto y algunas definiciones relacionadas se extiende AE añadiendo la primitiva **with** que permite asignar a cada identificador un valor dentro de un alcance restringido. Similar a las expresiones **let** de HASKELL y RACKET. [1] Llamaremos a este lenguaje WAE:

```
<expr> ::= <id>
         | <num>
         | {+ <expr> <expr>}
         | {- <expr> <expr>}
         | {with {<id> <expr>} <expr>}
```

La gramática anterior, se expresa en sintaxis abstracta como se muestra en el Código 4:

```
1 (define-type WAE
2    [id   (i symbol?)]
3    [num  (n number?)]
4    [add  (izq WAE?) (der WAE?)]
5    [sub  (izq WAE?) (der WAE?)]
6    [with (id symbol?) (value WAE?) (body WAE?)])
```

Código 4: Sintaxis abstracta de AE

Por ejemplo, la forma de evaluar la siguiente expresión:

```
{with {a {- 10 2}}
   {+ a a}}
```

es:

- Asignamos el resultado de la evaluación de la expresión **{- 10 2}** al identificador **a**.
- Sumamos consigo mismo el valor de **a** su valor, devolviendo el resultado correspondiente (**16**).

De manera general, la forma de evaluar una expresión **{with {<id> <value>} <body>}** conlleva una asignación de la subexpresión **<value>** al identificador **<id>** y regresar el resultado de la evaluación de la subexpresión **<body>** con dicha asignación. Lo cual hace necesario contar con un mecanismo de sustitución.

2.3.1. Tipos de identificadores

A continuación se presentan algunas definiciones importantes relacionadas con los identificadores involucrados en una expresión que permiten llevar a cabo la evaluación de las expresiones de tipo **with**. [1] Tomaremos la siguiente expresión para analizar dichos conceptos:

```
{with {x y}
   {+ x y}}
```

Definición 1. *(Alcance) El alcance de un identificador es la región de un programa en la cual éstos alcanzan su valor. En WAE el alcance de todo identificador se da en el* **<body>** *de las expresiones* **with**.

En nuestra expresión, el alcance del identificador x definido es el cuerpo {+ x y}.

```
{with {x y}
   {+ x y}}
```

Definición 2. *(Identificador de ligado) La instancia de ligado de un identificador es la instancia de un identificador que da a éste su valor. En WAE el identificador de ligado siempre será* **<id>** *en las expresiones* **with**.

En nuestra expresión, el único identificador de ligado es x.

```
{with {x y}
   {+ x y}}
```

Definición 3. *(Identificador ligado) Un identificador está ligado si se encuentra contenido en el alcance un identificador de ligado por su nombre. En WAE todos los identificadores que sean iguales a* **<id>** *son de ligado.*

En nuestra expresión, el único identificador ligado es la x que se encuentra en el alcance de nuestro identificador de ligado.

```
{with {x y}
   {+ x y}}
```

Definición 4. *(Identificador libre) Un identificador está libre si no se encuentra contenido en el alcance de un identificador de ligado por su nombre. En WAE todos los identificadores que sean distintos a* **<id>** *son libres.*

En nuestra expresión, el único identificador libre es y pues no se encuentra ligado a ningún identificador de ligado.

```
{with {x y}
   {+ x y}}
```

2.3.2. Algoritmo de sustitución

La operación de sustitución indica que se debe reemplazar toda presencia del identificador **x** por **t** una expresión, en la expresión **expr** denotada por: [2]

expr[x:=t]

Usando el concepto anterior, se establecen las siguientes reglas de sustitución:

```
;; si x igual a <id>
(1) <id>[x:=t]     = t

;; si x distinto de <id>
(2) <id>[x:=t]     = <id>

(3) <num>[x:=t]    = <num>

(4) {+ a b}[x:=t] = {+ a[x:=t] b[x:=t]}

(5) {- a b}[x:=t] = {- a[x:=t] b[x:=t]}

;; si x igual a <id>
(6) {with {<id> <value>} <body>}[x:=t] =
        {with {<id> <value>[x:=t]} <body>}

;; si x distinto de <id>
(7) {with {<id> <value>} <body>}[x:=t] =
        {with {<id> <value>[x:=t]} <body>[x:=t]}
```

Para realizar la operación de sustitución en expresiones de tipo **with** se deben separar en dos casos, pues si se trata de sustituir una variable de ligado, podría cambiarse la semántica de la expresión, cosa que debe evitarse al tratarse de una operación complemente sintáctica, por ello es importante tomar el concepto de variable libre.

Por ejemplo, dada la siguiente expresión:

```
{with {a {+ a a}}
   {+ a 2}}
```

si se realiza la sustitución de **[a:=2]**, se obtiene:

```
{with {a 4}
   {+ 2 2}}
```

Lo anterior genera un error pues la semántica de la expresión fue cambiada. Con lo cual, si se trata de sustituir un identificador de ligado, sólo debe cambiarse la parte **<value>** que lo permita, obteniendo como resultado:

```
{with {a 4}
   {+ a 2}}
```

La implementación en Racket se define en el Código 5 mediante la función **subst**. Esta función aplica las 7 reglas de sustitución revisadas anteriormente. Las líneas 9 a 11 hacen el análisis de casos para las expresiones de tipo **with**.

```
1  ;; subst: WAE symbol WAE -> WAE
2  (define (subst expr sub-id val)
3     (type-case WAE expr
4        [id (i) (if (symbol=? i sub-id) val expr]
5        [num (n) expr]
6        [add (izq der) (add (subst izq sub-id val) (subst der sub-id val))]
7        [sub (izq der) (sub (subst izq sub-id val) (subst der sub-id val))]
8        [with (id value body)
9           (if (symbol=? id sub-id)
10              (with id (subst value sub-id val) body)
11              (with id (subst value sub-id val) (subst body sub-id val)))]))
```

Código 5: Sustitución en WAE

Con lo cual la función **calc** se modifica como en el Código 6. Para evaluar una expresión **with** se debe interpretar el cuerpo del mismo una vez realizada la sustitución del **<id>** por **<value>** en **<body>**. En el caso de las variables, al no tener un valor asociado por sí mismas, se lanza un error.

```
1  ;; calc: WAE -> number
2  (define (calc expr)
3     ...
4        [id (i) (error 'calc "Identificador Libre")]
5        ...
6        [with (id value body)
7           (calc (sust body id value))])
```

Código 6: Análisis semántico de expresiones with en WAE

A continuación se muestra la ejecución de la función **calc** con algunas expresiones en sintaxis concreta.

```
> (calc (parse (quote foo)))
error: 'calc Identificador Libre
> (calc (parse (quote {with {a 2} {+ a a}})))
4
> (calc (parse (quote {with {b 3} {with {a 2} {+ a b}}})))
5
```

2.4. Regímenes de evaluación

Dada la siguiente expresión, se tienen dos formas de evaluarla: [9]

```
{with {x {+ 5 5}}
   {with {y {- x 3}}
      {+ x y}}}
```

La primera consiste en evaluar el valor asociado a cada identificador y luego sustituir en el cuerpo de la expresión correspondiente:

```
{with {x {+ 5 5}} {with {y {- x 3}} {+ x y}}}
= {with {x 10} {with {y {- x 3}} {+ x y}}}
= {with {y {- 10 3}} {+ 10 y}}
= {with {y 7} {+ 10 y}}
= {+ 10 7}
= 17
```

La segunda consiste en sustituir en el cuerpo de la expresión y luego evaluar cada identificador asociado a la expresión correspondiente:

```
{with {x {+ 5 5}} {with {y {- x 3}} {+ x y}}}
= {with {y {- {+ 5 5} 3}} {+ {+ 5 5} y}}
= {+ {+ 5 5} {- {+ 5 5} 3}}
= {+ 10 {- {+ 5 5} 3}}
= {+ 10 {- 10 3}}
= {+ 10 7}
= 17
```

Observación 2. La evaluación de expresiones se realiza de lo más interno a lo más externo y de izquierda a derecha.

La primera forma es conocido como régimen de ***evaluación glotona*** o ansiosa y consiste en evaluar cada valor conforme aparece, mientras que la segunda forma, conocida como régimen de ***evaluación perezosa*** evalúa cada valor hasta que sea necesario. El Código 6 utiliza régimen de evaluación perezosa, ya que realiza la sustitución correspondiente sin evaluar la subexpresión **<value>** contenida en las expresiones **with**.

El Código 7 muestra la modificación a la función **calc** para que use un régimen de evaluación glotón. Basta con evaluar la subexpresión **<value>** correspondiente (línea 5).

```
1 ;; calc: WAE -> number
2 (define (calc expr)
3    ...
4       [with (id value body)
5          (interp (sust body id (num (interp value))))])
```

Código 7: Análisis semántico de expresiones with en WAE glotón

Más adelante, en otra nota, se profundiza en ambos regímenes de evaluación.

2.5. Índices de Bruijn

Los índices de Bruijn son una notación creada por el matemático Nicolaas Govert de Bruijn que permite reemplazar el nombre de los identificadores por números. Este número indica la profundidad del alcance a la que se encuentra cada uno de los identificadores a partir del cuerpo de una expresión. Por ejemplo, dada la siguiente expresión: [9]

```
{with {x 5}
   {+ x x}}
```

puede reescribirse usando índices de Bruijn como sigue:

```
{with 5
   {+ <:0> <:0>}}
```

Los identificadores ligados son reemplazadoss por índices que indican su profundidad de alcance. Al tener un único identificador en esta expresión **with**, la profundidad de éste es cero. De la misma forma, el nombre de los identificadores de ligado se elimina y únicamente se indica el valor asociado dentro del alcance de su definición. Otro ejemplo se tiene en la expresión: [9]

```
{with {x 5}
   {with {y 6}
      {+ x y}}}
```

que se reescribe usando índices de Bruijn como sigue:

```
{with 5
   {with 6
      {+ <:1> <:0>}}}
```

La expresión anterior muestra que la suma, definida en la última línea, i.e. **{+ <:1> <:0>}** contiene en el lado izquierdo una referencia al identificador con profundidad 1 (comenzando desde cero) de alcance y el lado derecho una referencia al identificador con profundidad 0 de alcance. La profundidad se toma a partir del cuerpo actual y hacia arriba. De la misma forma, si el valor asociado a un identificador de ligado incluye identificadores de un **with** anterior, este también debe reescribirse mediante índices de Bruijn tomando la posición actual como el índice de profundidad 0. Por ejemplo,

```
{with {x 5}
   {with {y {+ x 7}}
      {+ x y}}}
```

se reescribe usando índices de Bruijn como sigue:

```
{with 5
   {with {+ <:0> 7}
      {+ <:1> <:0>}}}
```

En esta versión de WAE únicamente toma la profundidad de alcance de las variable pues sólo hay un identificador de ligado en las expresiones **`with`** (el parámetro **`<id>`**), sin embargo, si se tuviera una especie de *multi-with*, debe tomarse también la posición de cada identificador, especificando cada identificador ligado mediante coordenadas. Por ejemplo,

```
{with {{x 5} {y 6} {z 7}}
      {+ x {- y z}}}
```

se reescribe usando índices de Bruijn como sigue, el primer número indica la profundidad y el segundo la posición:

```
{with {5 6 7}
   {+ <:0 0> {- <:0 1> <:0 2>}}}
```

Estas coordenadas reciben el nombre de *direcciones léxicas* o *direcciones estáticas*. Son útiles en muchos contextos, en particular son ampliamente usados por algunos compiladores para indicar qué tan lejos se encuentra un identificador ligado.

2.6. Ejercicios

2.6.1. Sintaxis Concreta

1. Dada la sintaxis concreta del lenguaje WAE, muestra una derivación de las siguientes expresiones o justifica por qué éstas no son parte de la gramática:

   (a) `{+ {- {+ 2 3} {- 4 5}} {+ a {- 18 3}}}`

   (b) `{+ 1 2 3 {- 18 2 4}}`

   (c) `{with {a 2} {with {b 3} {+ a {- b 3}}}}`

   (d) `{with c 3 {with d 8 {+ c d}}}`

2.6.2. Análisis Léxico

1. En la *Observación 1* se menciona el uso de la función `quote` como un mecanismo para realizar el análisis léxico correspondiente por medio de las llamadas *s-expressions*. En el caso de WAE, una *s-expression* es alguna de las siguientes:

   - Un símbolo
   - Un número
   - Una lista de *s-expressions*

   Usando la función `quote` muestra el análisis léxico de las siguientes expresiones, es decir, da la *s-expression* asociada. Recuerda que las expresiones deben formar parte de la gramática de WAE.

   (a) `1729`

   (b) `foo`

   (c) `{+ {- 1835 1729} 405}`

   (d) `{with {a 2} {+ a 3}}`

2.6.3. Análisis Sintáctico

1. Dada la definición para los árboles de sintaxis abstracta de WAE del Código 4, muestra la sintaxis abstracta de cada una de las siguientes expresiones o justifica por qué no existe:

   (a) `{+ {+ 4 0} {- {- 5 5} {+ 0 2}}}`

   (b) `{with {a {+ 18 35}} {+ {- a a} a}}`

   (c) `{with {{a 2} {b 3}} {+ a b}}`

   (d) `{with {a {+ 2 3}} {with {b {- 4 5}} {- a b}}}`

Sustitución

1. Realiza las siguientes sustituciones indicando en cada paso cuáles son los identificadores de ligado, ligados y libres:

   (a) `{+ {- a 2} {+ {- a a} b}} [a:={+ 5 3}]`

   (b) `{with {b {+ a a}} {+ a 2}} [a:={- 17 29}]`

   (c) `{with {a {+ a a}} {+ a 2}} [a:={- 17 29}]`

   (d) `{with {a {+ a a}} {with {b {+ a a}} {+ a b}}} [a:={- 17 b}]`

2. Define las siguientes funciones en Racket:

   (a) Una función **`de-ligado`** que dada una expresión de WAE en sintaxis abstracta, obtiene una lista con todos los identificadores de ligado correspondientes.

   (b) Una función **`ligado`** que dada una expresión de WAE en sintaxis abstracta, obtiene una lista con todos los identificadores ligados correspondientes.

   (c) Una función **`libres`** que dada una expresión de WAE en sintaxis abstracta, obtiene una lista con todos los identificadores libres correspondientes.

2.6.4. Índices de Bruijn

1. Dadas las siguientes expresiones de WAE, obtén su respectiva representación usando índices de Bruijn.

   (a) `{with {x {+ 2 2}} {+ x x}}`

   (b) `{with {x 2} {with {y 3} {+ x y}}}`

2. Dadas las siguientes expresiones de WAE con multi-`with`, obtén su respectiva representación usando direcciones léxicas.

   (a) `{with {{x 2} {y 3} {z 4}} {+ x y z}}`

   (b) `{with {{x 2} {y 3}} {with {{z 3} {w 2}} {+ x {+ y {+ z w}}}}}`

3. Dadas las siguientes expresiones de WAE representadas por medio de índices de Bruijn, obtén su respectiva representación sin usar índices. Asigna los nombres de identificadores que consideres más adecuados.

(a) `{with 4 {+ <0> <0>}}`

(b) `{with 2 {with 3 {+ <0> <1>}}}`

4. Dadas las siguientes expresiones de WAE con multi-`with` representadas por medio de direcciones léxicas, obtén su respectiva representación sin usar direcciones léxicas. Asigna los nombres de identificadores que consideres más adecuados.

   (a) `{with {2 3 4} {+ <:0,2> <:0,0> <:0,1>}}`

   (b) `{with {2 3} {with {3 2} {+ <:0,0> {+ <:0,1> {+ <:1,0> <:1,1>}}}}}`

Referencias

[1] Samuel A. Rebelsky, *Programming Languages,* Notas de clase, Grinell College revisión 99S. Consultado el 20 de noviembre de 2018 [**http://www.math.grin.edu/~rebelsky/Courses/CS302/99S/Outlines/outline.02.html**]

[2] Favio E. Miranda, Elisa Viso, *Matemáticas Discretas*, Las Prensas de Ciencias, Segunda Edición, 2016.

[3] Raúl Rojas, *A Tutorial Introduction to the Lambda Calculus*, ArXiv, 2015.

[4] Bobby Kleinberg, Notas de clase, Cornell Universitu, revisión 2012sp. Consultado el 7 de enero de 2019. [**http://www.cs.cornell.edu/courses/cs4820/2012sp/handouts/turingm.pdf**]

[5] History of Computers, *The Analitical Engige of Charles Babbage,* Consultado el 7 de enero de 2019. [**https://history-computer.com/Babbage/AnalyticalEngine.html**]

[6] José Galaviz, *Organización y Arquitectura de Computadoras,* Notas de clase, Facultad de Ciencias UNAM, revisión 2015-2.

[7] Imelda Avalos, Introducción a la programación, Universidad Autónoma de Nayarit, Consultado el 7 de enero de 2019. [**http://correo.uan.edu.mx/~iavalos/introprog.htm#Lenguajes**]

[8] EcuRed, *Enciclopedia Cubana*. Consultado el 7 de enero de 2019. [**https://www.ecured.cu/**]

[9] Shriram Krishnamurthi, *Programming Languages: Application and Interpretation*, Primera Edición, 2007.

[10] Sitio oficial de significados. [**https://www.significados.com/sintaxis/**]

[11] Codeforwin, *Classification of programming languages*, Consultado el 10 de julio de 2019. [**https://codeforwin.org/2017/05/programming-languages-classification.html**]

[12] Keith, London, *4, Programming Introduction to Computers. 24 Russell Square London WC1: Faber and Faber Limited.* The 'high' level programming languages are often called autocodes and the processor program, a compiler.

[13] Stephen, Levy. *Hackers: Heroes of the Computer Revolution*, Penguin Books. 1994.