UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Facultad de Ciencias

Licenciatura en Ciencias de la Computación

El lenguaje BAE

Lenguajes de Programación 2020-1

Alumnas:

⇒ Paredes Sánchez Jacqueline 315069473
⇒ Sanchez Benitez Maria del Pilar 315239674

19 de septiembre de 2019

Introducción

Comenzaremos por definir un lenguaje de programación.
Un **lenguaje de programación** es un lenguaje formal que proporciona una serie de instrucciones que la computadora deberá de seguir para resolver un problema.
Un programa nos sirve para poder resolver un problema de la vida real, por medio de la ayuda de la computadora, para esto lo que quisiéramos es poder de una manera sencilla y eficiente las instrucciones a la maquina y que esta nos devuelva un resultado preciso, rápido y siempre tener el mismo resultado a la misma pregunta.
Sin duda a lo largo de la historia de la programación cada vez se han ido creando más lenguajes de programación, pero debemos notar que ninguno ha sido menos importante que otro, ya que en cada momento de la historia estos les ayudaron a resolver sus problemas y fueron un pilar para la programación actual, y muchos de ellos aun los utilizamos.
Podríamos pensar que un lenguaje más actual nos serviría siempre para resolver un problema actual, pero esto no sucede, ya que ningún lenguaje es mejor o más importante que otro, ya que cada uno nos sirve de diferente manera y esto es depende al problema, por ejemplo un programa puede llevarnos en uno cientos de lineas mientras que el mismo problema lo podríamos resolver en otro con unas cuantas lineas, o incluso un lenguaje podría no servirnos para resolver cierto problema, ya que este lenguaje no fue hecho para este tipo de problemas.
Es por esto que nos interesa estudiar los lenguajes de programación, para saber la funcionalidad de cada uno y poder como programadores conocer el mejor lenguaje para un problema en especifico de manera que tengamos eficiencia, eficacia, y un facilitar el trabajo del programador.

Como vimos anteriormente queremos que un lenguaje nos ayude a resolver un problema en especifico, para esto debemos de estudiar de manera formal los lenguajes. La formalidad nos ayudara a poder probar que nuestros programas son correctos ya que nos devolverá el resultado esperado, para poder ver si es el mas eficiente, para ver si ese lenguaje nos ayuda a ese problema, para reducir errores de programación y de datos introducidos por el usuario, entre otras aplicaciones de la formalidad de un lenguaje.
Veamos un ejemplo de un programa. Un usuario "Hacherman", quiere crear un programa que el introducir un número nos devuelva el cuadrado de ese número, y este se ve de esta forma:
(Postfix 1 double)
Con pila inicial [4]
por lo que el usuario claramente desea que al ejecutar la nueva pila nos devuelva:
[16]
Notemos que la definición de un lenguaje nos dice que un lenguaje debe de ser formado por sintaxis (conjunto de cadenas y símbolos aceptados por el lenguaje) y la semántica (significado dado a esas cadenas), entonces para el lenguaje que

estamos viendo debería de ser de la siguiente manera:
La semántica deberá ser una serie de números, las palabras reservadas (en este caso postfiix y double) y la secuencia de instrucciones.Donde siempre sera un paréntesis que abre, la palabra reservada postfix, el número de elementos de la pila, la secuencia ejecutable, y el paréntesis que cierra.
La sintaxis de este programa será ejecutar el programa de izquierda a derecha, donde los elementos de la pila siempre son valores (números) o secuencias ejecutables, y es claro que queremos como resultado un valor entero. En nuestro programa no deseamos que: el número de elementos de la pila y el valor después de la palabra postfix sean distintos, al final no obtener ningún resultado (esto pasa cuando la pila final es vacía) o al final no es un valor entero.
Para demostrar esto tendríamos que ver que al hacer el mismo computo se refiere a que nos devuelve el mismo resultado, y que además realiza los mismos pasos para llegar a este resultado. Para esto deberemos ocupar nuestra pila, al meter el nuevo valor, y después observar como esta definido double y mul, para poder demostrarlo debemos ver que primero la sintaxis de esta sea correcta, y después la semántica nos diría el significado de las cadenas, es decir que nos deberá devolver ambos programas.
Es por esto que es importante definir formalmente la sintaxis y semántica de un lenguaje, para que puedas observar que pasa cuando a tu programa le cambias la sintaxis, que pasa cuando la semántica es distinta, como obtener el resultado esperado de un programa, ver si dos programa son iguales y tomar el más conveniente. Por ello nos es importante definir formalmente los lenguajes.

El lenguaje de programación BAE

El lenguaje de programación BAE se refiere al lenguaje de las expresiones aritméticas - booleanas donde nos interesa que como todo lenguaje tenga sintaxis y semántica, es por ello que en este apartado queremos definirlas, así como ver la importancia de poder definirlo formalmente, ya que esto nos servirá para ver como es que funciona cada uno de los programas de este lenguaje y así poder encontrar errores fácilmente, y poder observar el computo que este ira realizando para así obtener el resultado esperado.

La sintaxis informal de BAE Para definir nuestro lenguaje se necesitan los factores principales de un lenguaje: la sintaxis y la semántica, y la sintaxis se tendrá que ver de la siguiente forma:

Descripción	Cadenas
Dígitos	0,1,...,9
Booleanos	true/false
Palabras reservadas	suma, prod, let, eq

Ahora veamos que la semántica se tendría que ver de la siguiente forma:

Operación	Descripción
Suma	Realiza la suma donde tiene como parámetros dos números o dos expresiones
Prod	Realiza la multiplicación de dos expresiones o de dos números
suc	recibe un número y nos devuelve el sucesor
pred	recibe un número y nos devuelve el predecesor
if	aquí toma un boleano y dos expresiones, si el boleano es false devuelve la primera expresión, si es true devuelve la segunda expresión
iszero	recibe un numero, si es cero devuelve true, de lo contrario devuelve false
let	recibe una expresión y devuelve otra expresión

Formalización de AE

Para la formalización de AE(que se refiere solo a expresiones aritméticas), la definiremos por medio de juicios, las cuales son las reglas que nos servirán para definir nuestro lenguaje.
Tenemos para la sintaxis concreta los siguientes juicios

$\frac{}{0D}\ \frac{}{1D}\ \frac{}{2D}\ \frac{}{3D}\ \frac{}{4D}\ \frac{}{5D}\ \frac{}{6D}\ \frac{}{7D}\ \frac{}{8D}\ \frac{}{9D}$
$\frac{sD}{sN}\ \frac{s_1 N s_2 D}{s_1 s_2 N}\ \frac{sN}{sF}\ \frac{sE}{(s)F}\ \frac{sF}{sT}\ \frac{s_1 T s_2 F}{s_1 * s_2 T}\ \frac{sT}{sE}\ \frac{s_1 E s_2 T}{s_1 + s_2 E}$

Podemos ver que son definiciones inductivas, es decir que nos ayudará para hacer expresiones más elaboradas y poder demostrar que estas sean correctas.
Ahora para la sintaxis abstracta tenemos que razonar las expresiones aritméticas de forma inductiva, pero para la sintaxis concreta seria muy compleja, es por eso que utilizaremos el árbol de análisis sintáctico, pero así es como podremos eliminar la ambigüedad.Un árbol de sintaxis abstracta (asa) es un árbol ordenado cuyos nodos están etiquetados por un operador. Cada operador tiene un índice asignado que indica el número de argumentos que recibe, los cuales corresponden al numero de hijos de cualquier nodo etiquetado con él. Nuestras reglas se verán de la siguiente manera:
$\frac{nNat}{num[n]asa}\ \frac{t_1 \mathbf{asa} t_2 \mathbf{asa}}{suma(t_1,t_2)\mathbf{asa}}\ \frac{t_1 \mathbf{asa} t_2 \mathbf{asa}}{prod(t_1,t_2)\mathbf{asa}}$

La relación entre ambas sintaxis (concreta y abstracta) es que al proceso de la concreta a la abstracta se le llama analizador sintáctico. Este verifica que nuestro programa es correcto en la concreta y lo representa como un asa. Los juicios para esto se ven de la siguiente forma:

$\frac{}{0D \leftrightarrow 0Nat}$ $\frac{}{1D \leftrightarrow 1Nat}$ $\frac{}{2D \leftrightarrow 2Nat}$ $\frac{}{3D \leftrightarrow 3Nat}$ $\frac{}{4D \leftrightarrow 4Nat}$ $\frac{}{5D \leftrightarrow 5Nat}$ $\frac{}{6D \leftrightarrow 6Nat}$ $\frac{}{7D \leftrightarrow 7Nat}$
$\frac{}{8D \leftrightarrow 8Nat}$ $\frac{}{9D \leftrightarrow 9Nat}$
$\frac{sD \leftrightarrow kNat}{sN \leftrightarrow kNat}$ $\frac{s_1N \leftrightarrow k_1Nats_2D \leftrightarrow k_2Nat}{s_1s_2N \leftrightarrow 10k_1 + k_2Nat}$
$\frac{sT \leftrightarrow tasa}{sE \leftrightarrow tasa}$ $\frac{s_1E \leftrightarrow t_1asas_2T \leftrightarrow t_2asa}{s_1 + s_2E \leftrightarrow suma(t_1, t_2)asa}$
$\frac{sF \leftrightarrow tasa}{sT \leftrightarrow tasa}$ $\frac{s_1T \leftrightarrow t_1asas_2F \leftrightarrow t_2asa}{s_1 * s_2T \leftrightarrow prod(t_1, t_2)asa}$ $\frac{sN \leftrightarrow kNat}{sF \leftrightarrow num[k]asa}$ $\frac{sE \leftrightarrow tasa}{(s)F \leftrightarrow tasa}$

Proposición 1*Se cumplen las siguientes propiedades de correctud para el analizador sintactico*

- Si $sD \leftrightarrow k$ Nat entonces s D y k Nat
- Si $sN \leftrightarrow k$ Nat entonces s N y k Nat
- Si $sE \leftrightarrow t$ asa entonces s E y t asa
- Si $sT \leftrightarrow t$ asa entonces s T y t asa
- Si $sF \leftrightarrow t$ asa entonces s F y t asa

Sintaxis de BAE

Definiremos la sintaxis concreta y la sintaxis abstracta:

- Sintaxis concreta.
  Con sintaxis concreta nos referimos a las cadenas en representación lineal, que son fáciles de entender para el ser humano .
  Definimos expresiones del tipo BAE como sigue:

$$E ::= x|n|true|false|e + e|e * e|$$

$$suc\ e|pre\ e|e \wedge e\ |e \vee e|\neg e|e < e|e > e|$$

$$e = e|\textbf{if}\ \ e\ \ \textbf{then}\ \ e\ \ \textbf{else}\ \ e\ |$$

$$\textbf{let}\ \ x{=}e\ \ \textbf{in}\ \ e\ \ \textbf{end}$$

  Notemos que la gramática anterior es ambigua, por lo siguiente:
  EJEMPLO 1:

$$true \wedge false \wedge false \vee true$$

  Para obtener el resultado de esta expresión utilizando la gramática anterior tenemos que por un lado:

  1. $true \wedge false =$ false es una expresión e
  2. $true \wedge false \wedge false = e \wedge false =$ false es una expresión *e1*
  3. $e \wedge false \vee true = e1 \vee true =$ true

  Desarrollando de otra manera tenemos que:

  1. $true \wedge false =$ false es una expresión e

2. $false \vee true$ = true es una expresión *e1*
3. $e1 \wedge e2$ = false

Por lo que para la misma expresión se tienen dos resultados distintos,lo cual no es conveniente para nuestro programa.Dado que no existe un algoritmo para eliminar la ambigüedad de una gramática por lo que definiremos la sintaxis abstracta, ya que es equivalente a la sintaxis concreta.

- Sintaxis abstracta.
  Este tipo de sintaxis nos proporciona una representación de una expresión mediante un árbol obtenido después de la fase se análisis sintáctico, el cual es más fácil de manipular.
  Un árbol de sintaxis abstracta (**asa**), es un árbol cuyos nodos están etiquetados por un operador, cada operador tiene un índice asignado con el numero de de argumentos a recibir. Para nuestro lenguaje BAE definimos la sintaxis abstracta como sigue:
  Definimos la sintaxis abstracta(usando gramática):

$$t ::= x \mid num[n] \mid bool[true] \mid bool[false] \mid$$
$$plus[t_1, t_2] \mid prod[t_1, t_2] \mid suc[t] \mid pre[t] \mid$$
$$conj[t_1, t_2] \mid disy[t_1, t_2] \mid neg[t] \mid Lt[t_1, t_2] \mid$$
$$Gt[t_1, t_2] \mid Equi[t_1, t_2] \mid Ift[t_1, t_2, t_3] \mid$$
$$LetE[x, t_1, t_2]$$

  O bien mediante juicios:

$$\frac{n \quad Nat}{num[n] \quad \mathbf{asa}} \qquad \frac{true \quad Bool}{bool[true] \quad \mathbf{asa}} \qquad \frac{t_1 \quad \mathbf{asa} \quad t_2 \quad \mathbf{asa}}{plus(t_1, t_2) \quad \mathbf{asa}}$$

$$\frac{n \quad \mathbf{asa}}{suc[n] \quad \mathbf{asa}} \qquad \frac{b_1 \quad \mathbf{asa} \quad b_2 \quad \mathbf{asa}}{conj(b_1, b_2) \quad \mathbf{asa}} \qquad \frac{b \quad \mathbf{asa}}{neg[b] \quad \mathbf{asa}}$$

$$\frac{t_1 \quad \mathbf{asa} \quad t_2 \quad \mathbf{asa}}{Equi(t_1, t_2) \quad \mathbf{asa}} \qquad \frac{t_1 \quad \mathbf{asa} \quad t_2 \quad \mathbf{asa} \quad t_3 \quad \mathbf{asa}}{Ift(t_1, t_2, t_3) \quad \mathbf{asa}}$$

$$\frac{t_1 \quad \mathbf{asa} \quad t_2 \quad \mathbf{asa}}{LetE(x, t_1, t_2) \quad \mathbf{asa}}$$

De manera análoga se pueden ver los casos restantes.
Ahora podemos notar la diferencia entre la sintaxis abstracta y la concreta con el ejemplo 1:

- S.concreta:$true \wedge false \wedge false \vee true$
- S.abstracta:conj [conj[bool[true],bool[false]], disy [bool[false], bool[true]]]

Es claro que nos evitamos el problema de la ambigüedad; pero ahora veamos que son equivalentes.
El proceso de traducción de la sintaxis concreta a la sintaxis abstracta se le conoce como *análisis sintáctico*. Un analizador sintáctico es aquel que verifica si un programa es correcto con respecto a la sintaxis concreta y lo representa con un árbol de sintaxis abstracta. Para nuestro lenguaje lo definiremos mediante juicios, que relacione ambos niveles de sintaxis.

$$e_1 \quad E \longleftrightarrow e_2 \quad \textbf{asa}$$

de tal manera que:

$$true \wedge false \wedge false \vee true \quad E \longleftrightarrow conj(conj(bool[true], bool[false]), disy(bool[false], bool[true])) \quad \textbf{asa}$$

La relación de análisis sintáctico, se define en base a las reglas de la definición inductiva de la sintaxis concreta, como sigue

$$\frac{n\ E \longleftrightarrow m\ Nat}{n\ Nat \longleftrightarrow num[m]\ \textbf{asa}} \qquad \frac{s_1\ E \longleftrightarrow t_1\ \textbf{asa} \quad s_2\ E \longleftrightarrow t_2\ \textbf{asa}}{s_1 + s_2\ E \longleftrightarrow plus(t_1, t_2)\ \textbf{asa}}$$

$$\frac{b\ E \longleftrightarrow b_1\ Bool}{b\ Bool \longleftrightarrow bool[b_1]\ \textbf{asa}} \qquad \frac{b_1\ E \longleftrightarrow b_1\ \textbf{asa} \quad b_2\ E \longleftrightarrow b_2\ \textbf{asa}}{b_1 \wedge b_2\ E \longleftrightarrow conj(b_1, b_2)\ \textbf{asa}}$$

$$\frac{e\ E \longleftrightarrow t\ \textbf{asa}}{(e) \longleftrightarrow t\ \textbf{asa}}$$

de manera análoga para los demás operadores.
El análisis sintáctico de una cadena e consiste en buscar una expresión t tal que cumple que $e\ E \longleftrightarrow t$ **asa**, En caso de que t no exista decimos que el analizador falla. Notemos que el analizador sintáctico debe cumplir que:

- Debe ser total; para cada e E debe existir un t **asa** tal que $e\ E \longleftrightarrow t$ **asa**.
- No debe ser ambiguo; es decir NO debe existir una expresión e E y dos arboles distintos t_1 y t_2 tales que $e\ E \longleftrightarrow t_1$ **asa** y $e\ E \longleftrightarrow t_2$ **asa**

Con anterioridad se definió la expresión **let**, que es un mecanismo para introducir variables con alcance restringido, es decir variables locales.

$$\frac{t_1 \quad \textbf{asa} \quad t_2 \quad \textbf{asa}}{LetE(x, t_1, t_2) \quad \textbf{asa}}$$

Pero existe un problema con nuestra representación de de sintaxis abstracta; pues no es posible distinguir entre la definición y el uso de una presencia variable.Por lo cual utilizaremos la sintaxis abstracta de orden superior.
La técnica de la sintaxis abstracta de orden superior, permite codificar información acerca del alcance y ligado de una variable de manera uniforme.
La idea general es agregar al lenguaje de sintaxis abstracta un constructor de árboles $x.t$ que denota el ligado de la variable x en el árbol t.

Esta representación también se conoce como árbol de ligado abstracto (**ala**), para el lenguaje BAE lo definimos como sigue:

$$\frac{n \quad Nat}{num[n] \quad \textbf{ala}} \qquad \frac{true \quad Bool}{bool[true] \quad \textbf{ala}} \qquad \frac{t_1 \quad \textbf{ala} \quad t_2 \quad \textbf{ala}}{plus(t_1, t_2) \quad \textbf{ala}}$$

$$\frac{n \quad \textbf{ala}}{suc[n] \quad \textbf{ala}} \qquad \frac{b_1 \quad \textbf{ala} \quad b_2 \quad \textbf{ala}}{conj(b_1, b_2) \quad \textbf{ala}} \qquad \frac{b \quad \textbf{ala}}{neg[b] \quad \textbf{ala}}$$

$$\frac{t_1 \quad \textbf{ala} \quad t_2 \quad \textbf{ala}}{Equi(t_1, t_2) \quad \textbf{ala}} \qquad \frac{t_1 \quad \textbf{aa} \quad t_2 \quad \textbf{ala} \quad t_3 \quad \textbf{ala}}{If(t_1, t_2, t_3) \quad \textbf{ala}}$$

$$\frac{}{x \ \textbf{ala}} \qquad \frac{t_1 \quad \textbf{ala} \quad t_2 \quad \textbf{ala}}{LetE(t_1, x.t_2) \quad \textbf{ala}}$$

De manera análoga para los operadores restantes.Es importante notar que que las variables se consideran como primitivas.
De igual manera podemos podemos definir el juicio de equivalencia entre la sintaxis concreta y la sintaxis abstracta de orden superior para el caso de expresiones **let**.

$$\frac{e_1 \ E \longleftrightarrow t_1 \ \textbf{ala} \quad e_2 \ E \longleftrightarrow t_2 \ \textbf{ala}}{\textbf{let} \ x = e_1 \ \textbf{in} \ e_1 \ \textbf{end} \ E \longleftrightarrow \textbf{let}(t_1, x.t_2) \ \textbf{ala}}$$

Ahora en adelante utilizaremos la sintaxis abstracta de orden superior por el conveniente que nos brinda.
Dada una expresión t definimos el conjunto de variables libres de t como:

- $FV(x) \ = \ x$
- $FV(O(t_1, ...t_n)) \ = \ FV(t_1) \cup ... \cup FV(t_n)$
- $FV(x.t) \ = \ FV(T) \setminus x$

Definimos una substitución $e[x := t]$ de manera recursiva como:

- $x[x := t] \ = \ t$
- $z[x := t] \ = \ z \ six \neq z$
- $O(t_1, ...t_n) \ = \ O(t_1[x := t], ..., t_n[x := t])$
- $(z.e)[x := t] \ = \ z.e[x := t] \ si \ z \notin x \cup FV(t)$

Una vez definida la substitución, realicemos lo siguiente:

$$(if \ (let \ x \ = \ 2 \ in \ x + y \ end) \ > \ 4 \ then \ 1 \ else \ 0 \ ) \ [x := 10, y := 3]$$

Si realizamos la substitución tenemos que

$$(if \ (let \ x \ = \ 2 \ in \ 10 + 3 \ end) \ > \ 4 \ then \ 1 \ else \ 0 \ )$$

Lo cual es incorrecto ya que cambia completamente, el significado de nuestra expresión por lo que nos es conveniente, definir para nuestro lenguaje BAE α-equivalencia; recordando la definición para lógica proposicional, tenemos que:

Decimos que dos fórmulas φ_1 y φ_2 son α-equivalentes; lo cual denotamos como $\varphi_1 \sim_\alpha \varphi_2$ si y solo si φ_1 y φ_2 difieren a lo más en los nombres de sus variables ligadas.

Para nuestro lenguaje diremos que dos expresiones e_1 y e_2 son α-equivalentes y lo representamos $e_1 \equiv_\alpha e_2$ si y solo si e_1 y e_2 difieren únicamente en los nombres de las variables ligadas
En particular para nuestro ejemplo anterior si aplicamos una α-equivalencia tenemos que:

$$(if\ (let\ x\ =\ 2\ in\ x + y\ end)\ >\ 4\ then\ 1\ else\ 0\ )\ [x := 10, y := 3]\ \equiv_\alpha$$

$$(if\ (let\ z\ =\ 2\ in\ z + y\ end)\ >\ 4\ then\ 1\ else\ 0\ )\ [x := 10, y := 3]$$

y posteriormente realizamos la substitución, obtenemos que:

$$(if\ (let\ z\ =\ 2\ in\ z + 3\ end)\ >\ 4\ then\ 1\ else\ 0\ )$$

$$(if\ (2)\ >\ 4\ then\ 1\ else\ 0\ )$$

Finalmente obtenemos 0, y no alteramos el significado del programa.
Para nuestra implementación del lenguaje BAE, no cuenta con el mecanismo de α-equivalencia ya que, la complejidad en tiempo se eleva.

Semántica de BAE

Retomando la semántica de un lenguaje de programación consiste en dar significado a diversas caracterısticas e instrucciones del lenguaje, por ejemplo, el comportamiento en tiempo de ejecución. La semántica dinámica del lenguaje BAE se dará mediante un sistema de transición particular, usando el estilo llamado semántica operacional estructural o de paso pequeño.

- Estados S = t — t asa. Es decir, los estados son árboles bien formados de la sintaxis abstracta de orden superior(ala).

$$\frac{t\ \mathbf{ala}}{t\ estado}(edo)$$

- Estados iniciales I = t — t asa, F V (t) = ∅. Es decir, los estados iniciales son expresiones cerradas,es decir, sin variables libres.

$$\frac{t\ asa\ FV(t)\ =\ \emptyset}{t\ inicial}(ein)$$

- Estados finales. Para poder modelar de manera fiel el proceso de evaluación definimos una categorıa de valores los cuales son un subconjunto de expresiones que ya se han terminado de evaluar y no pueden reducirse más.

De esta manera los valores representan a los posibles resultados finales de un proceso de evaluación. Su definición es la esperada, dada mediante un juicio v valor definido como:

$$\frac{}{bool[true]\ valor}(vtrue) \qquad \frac{}{bool[false]\ valor}(vfalse) \qquad \frac{}{num[n]\ valor}(vnum)$$

Por lo cual definimos la semántica por medio de juicios como sigue:

$$\frac{}{num[n] final}(fnum) \qquad \frac{}{bool[true] valor}(ftrue) \qquad \frac{}{bool[false] valor}(ffalse)$$

$$\frac{}{plus(num[n], num[m]) \longrightarrow num[n+m]}(plusf) \qquad \frac{t_1 \longrightarrow t_1'}{plus(t_1, t_2) \longrightarrow plus(t_1', t_2)}(plusi)$$

$$\frac{t_2 \longrightarrow t_2'}{plus(num[n], t_2) \longrightarrow plus(num[n], t_2')}(plusd) \qquad \frac{}{suc(num[n]) \longrightarrow num[n+1]}(sucNum)$$

$$\frac{t \longrightarrow t'}{suc(t) \longrightarrow suct'}(sucT) \qquad \frac{}{pred(num[0]) \longrightarrow num[0]}(pre0) \qquad \frac{}{pre(num[n+1]) \longrightarrow num[n]}(preNum)$$

$$\frac{t \longrightarrow t'}{pre(t) \longrightarrow pret'}(preT) \qquad \frac{}{prod(num[n], num[m]) \longrightarrow num[n*m]}(prodf)$$

$$\frac{t_1 \longrightarrow t_1'}{prod(t_1, t_2) \longrightarrow prod(t_1', t_2)}(prodi) \qquad \frac{t_2 \longrightarrow t_2'}{prod(num[n], t_2) \longrightarrow prod(num[n], t_2')}(prodd)$$

$$\frac{}{neg(bool[true]) \longrightarrow bool[false]}(negtrue) \qquad \frac{}{neg(bool[false]) \longrightarrow bool[true]}(negfalse)$$

$$\frac{t \longrightarrow t'}{neg(t) \longrightarrow neg(t')}(negT) \qquad \frac{}{conj(b_1, b_2) \longrightarrow bool[b1 \wedge b2]}(conjf) \qquad \frac{t_1 \longrightarrow t_1'}{conj(t_1, t_2) conj(t_1', t_2)}(conji)$$

$$\frac{t_2 \longrightarrow t_2'}{conj(b, t_2) \longrightarrow conj(b, t_2')}(conjd) \qquad \frac{}{disy(b_1, b_2) \longrightarrow bool[b1 \vee b2]}(disyf) \qquad \frac{t_1 \longrightarrow t_1'}{disy(t_1, t_2) disy(t_1', t_2)}(disyi)$$

$$\frac{t_2 \longrightarrow t_2'}{disy(b, t_2) \longrightarrow disy(b, t_2')}(disyd) \qquad \frac{}{if(bool[true], t_2, t_3) \longrightarrow t_2}(iftrue)$$

$$\frac{}{if(bool[false], t_2, t_3) \longrightarrow t_3}(iffalse) \qquad \frac{t_1 \longrightarrow t_1'}{if(t_1, t_2, t_3) \longrightarrow if(t_1', t_2, t_3)}(ifT)$$

$$\frac{v\ valor}{let(v, x.e_2) \longrightarrow e_2[x := v]}(letf) \qquad \frac{t_1 \longrightarrow t_1'}{let(t_1, x.t_2) \longrightarrow let(t_1', x.t_2)}(letT)$$

Es importante notar que: Las variables son estados bloqueados, pero no son finales.

La relación $\longrightarrow$ de nuestro lenguaje BAE cumple que:

- Si v es valor entonces $\not\longrightarrow$,*i.e* v esta bloqueado.
- **Determinismo de** $\longrightarrow$; si $e \longrightarrow e_1$ y $e \longrightarrow e_2$ entonces $e_1 = e_2$

Dada una relación de transición $\longrightarrow$, se definen inductivamente las siguientes relaciones derivadas de importancia:

cerradura reflexiva-transitiva

Definimos la *Cerradura reflexiva-transitiva* $\leftrightarrow^\star$ como:

$$\frac{}{S \longrightarrow S'}(crt1) \qquad \frac{S \longrightarrow S' \quad S' \longrightarrow^\star S''}{S \longrightarrow^\star S''}(crt2)$$

Para *crt1*, nos referimos a que de una expresión S en un paso, o bien en una evaluación llegamos a una expresión S'.
De manera inductiva, para *crt2*, si en una evaluación de una expresión S llegamos a una expresión S', y de Sén más de una evaluación llegamos S", entonces podemos concluir que que de S en 0 o más pasos llegamos a S"

Cerradura transitiva

Para el caso de la cerradura transitiva ($\longrightarrow^+$). Para el caso base solo se necesita que de una expresión S en una evaluación llegamos a una expresión S',entonces en más de 0 pasos podemos llegar de S a S'.
Para el paso inductivo suponemos que si de S en una evaluación llegamos a S'y además si de Sén más de 0 paso llegamos a una expresión S" entonces podemos inferir que de una expresión S en más de 0 evaluaciones llegamos a un expresión S". **Determinismo de** $\longrightarrow^\star$; si $e \longrightarrow^\star e_1$ y $e \longrightarrow^\star e_2$ con $e_1 \not\longrightarrow$ y $e_2 \not\longrightarrow$ entonces $e_1 = e_2$

Semántica estática de BAE

La semántica estática determina qué estructuras de la sintaxis abstracta (asa o ala) están bien formadas de acuerdo a ciertos criterios sensibles al contexto como la resolución del alcance, al requerir que cada variable sea declarada antes de usarse. Por lo general la semántica estática consiste de dos fases:

- *La resolución del alcance de variables, el cual en nuestro caso está codificado directamente en la sintaxis abstracta.*
- *La verificación de correctud estática de un programa mediante la interacción con el sistema de tipos.*

Para BAE nos es conveniente verificar los tipos, ya que:

- *Permite descubrir errores de programación tempranamente.*
- *Un programa correctamente tipado no puede funcionar mal.*
- *Los tipos documentan un programa de manera mas simple y manejable.*
- *Los lenguajes tipados pueden implementarse de manera más clara y eficiente.*

Para la semantica estatica de BAE, la definiremos mediante juicios entre expresiones t, tipos $\boldsymbol{T}$ *y contextos de declaraciones de variables tipadas* Γ *; de la siguiente manera "la expresión t tiene tipo* $\boldsymbol{T}$ *bajo el contexto* Γ*.º bien* $\Gamma \vdash t : \boldsymbol{T}$
Definimos el sistema de tipos como para BAE con solo dos tipos;

$$\boldsymbol{T} ::= \ Nat | Bool$$

Y la relación de tipado $\Gamma \vdash t : \boldsymbol{T}$*, se define de la siguiente manera:*

- *Una variable pude tener cualquier tipo asociado.*

$$\frac{}{\Gamma, x \ : \ \boldsymbol{T} \vdash \ x : \boldsymbol{T}}$$

- *Un numero es de tipo natural.*

$$\frac{}{\Gamma \vdash \ num[n] \ : \ \boldsymbol{Nat}}$$

- *Para valores booleanos:*

$$\frac{}{\Gamma \vdash \ bool[true] \ : \ \boldsymbol{Bool}} \qquad \frac{}{\Gamma \vdash \ bool[false] \ : \ \boldsymbol{Bool}}$$

- *Para expresiones aritmeticas:*

$$\frac{\Gamma \vdash t_1 \ : \ \boldsymbol{Nat} \quad \Gamma \vdash t_2 \ : \ \boldsymbol{Nat}}{\Gamma \vdash \boldsymbol{plus}(t_1, t_2) \ : \ \boldsymbol{Nat}} \qquad \frac{\Gamma \vdash t_1 \ : \ \boldsymbol{Nat} \quad \Gamma \vdash t_2 \ : \ \boldsymbol{Nat}}{\Gamma \vdash \boldsymbol{prod}(t_1, t_2) \ : \ \boldsymbol{Nat}}$$

$$\frac{\Gamma \vdash t \ : \ \boldsymbol{Nat}}{\Gamma \vdash \boldsymbol{suc}\, t \ : \ \boldsymbol{Nat}} \qquad \frac{\Gamma \vdash t \ : \ \boldsymbol{Nat}}{\Gamma \vdash \boldsymbol{pre}\, t \ : \ \boldsymbol{Nat}}$$

- *Para expresiones booleanas.*

$$\frac{\Gamma \vdash b \ : \ \boldsymbol{Bool}}{\Gamma \vdash \boldsymbol{neg}\, b \ : \ \boldsymbol{Bool}} \qquad \frac{\Gamma \vdash t_1 \ : \ \boldsymbol{Nat} \quad \Gamma \vdash t_2 \ : \ \boldsymbol{Nat}}{\Gamma \vdash \boldsymbol{Lt}(t_1, t_2) \ : \ \boldsymbol{Bool}}$$

$$\frac{\Gamma \vdash t_1 \ : \ \boldsymbol{Nat} \quad \Gamma \vdash t_2 \ : \ \boldsymbol{Nat}}{\Gamma \vdash \boldsymbol{Gt}(t_1, t_2) \ : \ \boldsymbol{Bool}} \qquad \frac{\Gamma \vdash t_1 \ : \ \boldsymbol{Nat} \quad \Gamma \vdash t_2 \ : \ \boldsymbol{Nat}}{\Gamma \vdash \boldsymbol{Equi}(t_1, t_2) \ : \ \boldsymbol{Bool}}$$

$$\frac{\Gamma \vdash t_1 \ : \ \boldsymbol{Bool} \quad \Gamma \vdash t_2 \ : \ \boldsymbol{T} \quad \Gamma \vdash t_3 \ : \ \boldsymbol{T}}{\Gamma \vdash \boldsymbol{if}(t_1, t_2, t_3) \ : \ \boldsymbol{T}}$$

- *Para la expresión Let*

$$\frac{\Gamma \vdash e_1 \ : \ \boldsymbol{T} \quad \Gamma, \ x : \boldsymbol{T} \vdash e_2 \ : \ \boldsymbol{S}}{\Gamma \vdash \boldsymbol{let}(e_1, x.e_2) \ : \ \boldsymbol{S}}$$

Tomemos el siguiente programa y verifiquemos que es de tipo **Nat***:*

*(let y = x*3 in (let z = y*3 in x*y)*2+x end)[x := 6*y+z]*

1. *Como en la sustitución hay variables que estan ligadas en nuestra expresión, aplicamos α-equivalencia.*
   - *(let u = x*3 in (let z = u*3 in x*u)*2+x end)[x := 6*y+z].*
   - *(let u = x*3 in (let v = u*3 in x*u)*2+x end)[x := 6*y+z]*
2. *Aplicamos la sustitución*
   - *(let u = (6*y+z)*3 in (let v = u*3 in (6*y+z)*u)*2+(6*y+z)end)*
3. *Utilizaremos razonamiento hacia atrás, para verificar que es de tipo* ***Nat***

a) $\emptyset \vdash$*(let u = (6*y+z)*3 in (let v = u*3 in (6*y+z)*u)*2+(6*y+z)end):* ***Nat***

1. *u:* ***Nat*** $\vdash$ *z :* ***Nat***
2. *u:* ***Nat*** $\vdash$ *y :* ***Nat***
3. *u:* ***Nat*** $\vdash$ *y + z :* ***Nat***
4. *u:* ***Nat*** $\vdash$ *6 :* ***Nat***
5. *u:* ***Nat*** $\vdash$ *(6*y+z) :* ***Nat***
6. *u:* ***Nat*** $\vdash$ *3 :* ***Nat***
7. *u:* ***Nat*** $\vdash$ *(6*y+z)*3 :* ***Nat***
8. *u:* ***Nat****,v:* ***Nat*** $\vdash$ *3:* ***Nat***
9. *u:* ***Nat****,v:* ***Nat*** $\vdash$ *u * 3:* ***Nat***
10. *u:* ***Nat*** $\vdash$ *(let v = u*3 in (6*y+z)*u) :* ***Nat***
11. *u:* ***Nat*** $\vdash$ *2:* ***Nat***
12. *u:* ***Nat*** $\vdash$ *2+(6*y+z):* ***Nat***
13. *u:* ***Nat*** $\vdash$ *(let v = u*3 in (6*y+z)*u)*2+(6*y+z):* ***Nat***
14. $\emptyset \vdash$*(let u = (6*y+z)*3 in (let v = u*3 in (6*y+z)*u)*2+(6*y+z)end):* ***Nat***

Por tanto la expresión si es de tipo ***Nat***

Programa

Nuestro programa fue realizado en haskell. Para poder desarrollar toda la teoría anterior realizaremos las siguientes funciones, las cuales nos ayudaran a primero verificar que las expresiones sean sintácticamente y semánticamente correctas, y ya que verifiquemos que estén correctamente escritas se hará la evaluación, aquí mencionaremos las funciones principales.

- *Primero se definen los tipos que utilizaremos y las palabras reservadas*

- *la función transform utiliza una función y nos devuelve su expresión aritmetica-booleana, usa como función auxiliar a transform_bae donde hacemos los casos para todos los operadores que tenemos declarados, tanto los tipos como las operaciones.*
- *La función fv nos dice cuales son las variables que no estan ligadas, como vimos en la teoria necesitamos la funcion de variables libres para usarlas en diversos casos, nos devolvera la lista de las variables libres, podemos notar que las operaciones son las que se deben verificar, ya que en lis nnúmerosy en los booleanos no hay variables ligadas.*
- *Las funciones que haran toda la evaluación sera evals y eval 1, donde evals será el que nos devuelva la evaluacion pero evals 1 la utilizaremos como funcion auxiliar para ver cada caso de la evaluacion, ya que como evaluacion solo queremos devolver un entero o un booleano. Y en eval es donde mandara a llamar a evals y verificara que el tipo sea el correcto.*
- *la funcion vt es la que nos ayudara a verificar que los tipos sean correctos, verificara lo que vimos en el ultimo capitulo para cada caso de cada operador.*

Conclusión

Una vez definido y comprendido la teoría antes mencionada, podemos notar la importancia de cada uno de los componentes de un lenguaje de programación. Es importante definir claramente la sintaxis y semántica de un lenguaje de programación, ya que de ello depende la manera que se quiera definir distintos programas.