Trinity

Especificación del lenguaje

Versión 2 — 17 de septiembre de 2014

Índice general

---

1 Introducción

Esta sección es informativa.

1. Este documento define el lenguaje de programación *Trinity* y especifica requerimientos para sus implantaciones.

2. *Trinity* es un lenguaje de programación imperativo con alcance y tipos estáticos enfocado en conceptos básicos de álgebra lineal sobre los números reales, y con soporte directo para operaciones entre escalares, vectores y matrices. Su influencia principal es el lenguaje *Octave*.

3. Este documento describe el lenguaje de programación *Trinity* a través de los elementos fundamentales de todo lenguaje: sus **objetos**, sus mecanismos de **composición** para hacer objetos complejos a partir de otros simples, y sus mecanismos de **abstracción y control**. No se presenta una gramática exacta para el lenguaje ni una especificación formal de su semántica, pero sí se especifica lo necesario para construir una gramática y un intérprete.

1.1 Sinopsis

1. Un programa simple de *Trinity* podría verse así:

```
program
use
    number x;
in
    print "How old are you?\n> ";
    read x;

    if x < 18 then
```


```
        print "You can't be here.";
    else
        print "Oh, ", x, "? come right in.";
    end;
end;
```

2. Ese programa se ejecutaría así:

```
$ ./trinity bouncer.ty
How old are you?
> 22
Oh, 22? come right in.
```

3. Un ejemplo más extenso de programa en *Trinity*:

```
program
use
  matrix(2,2) m;  # se inicializa todas en '0'
  number x;       # se inicializa en '0'
  boolean b;      # se inicializa en 'false'
in
  set m = { 1, 2
          : 3, 4 };

  read x;

  for i in m do
    if i % 2 == 0 then
      # si 'i' es par
      print i;
    else
      print x;
      read x;
      set b = not b;
    end;
  end;

  if b then
    print i;
  else
    print "b is a lie";
```


```
    end;
end;
```

---

2 Especificación

Esta sección es normativa.

2.1 Estructura lexicográfica

1. El espacio en blanco en *Trinity* se utiliza para delimitar lexemas, y es ignorado. Se considera espacio en blanco a cualquier caracter espacio, al fin de línea y a los *tabs*.

2. A menos que ocurra dentro de un literal de cadena de caracteres, todos los caracteres desde un caracter numeral (`#`) hasta el final de la línea se consideran espacio en blanco.

2.2 Objetos

1. Las expresiones `false` y `true` en *Trinity* denotan los dos únicos valores con el tipo booleano, cuya especificación de tipo es `boolean`. Los dos son iguales a sí mismos y distintos entre sí. Los lexemas `boolean`, `false` y `true` son palabras reservadas.

2. Las cantidades escalares en *Trinity* corresponden al tipo numérico, cuya especificación de tipo es `number`, y se implementan como números de punto flotante según el estándar IEEE 754, incluyendo a la noción de igualdad. El lexema `number` es una palabra reservada.

3. Un literal numérico está formado por secuencias de dígitos en notación posicional decimal, con una parte fraccional opcional separada por un punto (`.`) de la parte entera. Por ejemplo, `42` y `6.2831853` son literales numéricos. Los literales numéricos son expresiones que denotan cantidades escalares y tienen el tipo numérico.

4. Las cantidades matriciales en *Trinity* son arreglos bidimensionales, rectangulares, discretos y finitos de cantidades escalares. Las cantidades matriciales de `n` filas y `m` columnas corresponden al tipo especificado por `matrix(n, m)`. Dos cantidades matriciales del mismo tipo se consideran iguales cuando las cantidades escalares correspondientes en cada posición son iguales. El lexema `matrix` es una palabra reservada.

5. Las cantidades vectoriales son cantidades matriciales de una sola fila o una sola columna. Los vectores fila de `m` dimensiones tienen el tipo especificado por `row(m)`, y los vectores columna de `n` dimensiones tienen el tipo especificado por `col(n)`. `col(n)` y `row(m)` denotan los mismos tipos que `matrix(n, 1)` y `matrix(1, m)`, respectivamente — son simples atajos en la notación, y no representan conceptos distintos. Los lexemas `row` y `col` son palabras reservadas.

6. Un literal matricial es una secuencia de filas separadas por el símbolo dos puntos (`:`) y encerrada toda entre corchetes curvos (`{` y `}`), siendo cada fila una secuencia de igual número de expresiones separadas por el símbolo coma (`,`), cada una de las cuales debe tener el tipo numérico. Los literales matriciales son expresiones que denotan cantidades matriciales y tienen el tipo especificado por `matrix(n, m)`, donde `n` es el número de filas en el literal matricial, y `m` el número de expresiones en cada una de las filas.

   Por ejemplo,

   ```
   { 1, 2, 3 }
   ```

   es un literal matricial y denota un valor con el tipo especificado por `matrix(1, 3)`, que también puede escribirse `row(3)`, y

   ```
   { 4, 5 : 6, 7 : 8, 9 }
   ```

   es otro literal matricial y denota un valor con el tipo especificado por `matrix(3, 2)`.

7. Los tipos de las cantidades escalares y matriciales se denominan *tipos aritméticos*.

8. El tipo especificado por `matrix(1, 1)` **no** se considera equivalente al tipo numérico. Por ejemplo, las expresiones `42` y `{ 42 }` no tienen el mismo tipo, ya que la primera es del tipo numérico y la segunda es del tipo especificado por `matrix(1, 1)`. Como no son del mismo tipo, ni siquiera pueden considerarse iguales ni distintas.

2.3 Composición

1. A partir de una expresión `e` de cualquier tipo, se puede formar la expresión `(e)` del mismo tipo, cuyo valor será el valor de `e`. Por ejemplo, `42`, `(42)` y `((42))` son expresiones que valen lo mismo, al igual que `true` y `(true)`, y al igual que `{ 1, 2, 3 }` y `({ 1, (2), ((3)) })`. La evaluación de `(e)` procede con la evaluación de `e`, y al ésta producir su valor, se calcula y produce el resultado correspondiente, que en este caso es el mismo. Esta forma de evaluación será referida en el resto de este documento como «estricta».

2.3.1 Operadores lógicos

1. A partir de una expresión `e` del tipo booleano, se puede formar la expresión `not e` del tipo booleano, cuyo valor será el contrario del valor de `e`. La evaluación de `not e` es estricta. El lexema `not` es una palabra reservada.

   Por ejemplo,

   ```
   not true
   ```

   es una expresión del tipo booleano y tiene el mismo valor que la expresión `false`.

2. A partir de dos expresiones $\texttt{e}_1$ y $\texttt{e}_2$ del tipo booleano, se pueden formar las expresiones `e1 & e2` y `e1 | e2`, ambas del tipo booleano, cuyos valores serán, respectivamente, la conjunción y la disyunción de los valores de $\texttt{e}_1$ y $\texttt{e}_2$. Ambos operadores tienen asociatividad izquierda y tienen menor precedencia que el operador `not`. El operador `&` tiene mayor precedencia que el operador `|`.

   Por ejemplo, la expresión

   ```
   false & true | true
   ```

   es del tipo booleano y tiene el mismo valor que la expresión

   ```
   (false & true) | true
   ```

   debido a que la precedencia de `&` es mayor que la de `|`.

3. La evaluación de `e1 & e2` procede con la evaluación de $\texttt{e}_1$, y si ésta produce el valor `false`, la evaluación de `e1 & e2` produce `false` de inmediato, sin evaluar $\texttt{e}_2$; si, en cambio, la evaluación de $\texttt{e}_1$ produce el valor `true`, entonces se evalúa $\texttt{e}_2$ y la evaluación de `e1 & e2` produce finalmente el valor que produce la evaluación de $\texttt{e}_2$.

4. La evaluación de `e1 | e2` procede con la evaluación de $\texttt{e}_1$, y si ésta produce el valor `true`, la evaluación de `e1 | e2` produce `true` de inmediato, sin evaluar $\texttt{e}_2$; si, en cambio, la evaluación de $\texttt{e}_1$ produce el valor `false`, entonces se evalúa $\texttt{e}_2$ y la evaluación de `e1 | e2` produce finalmente el valor que produce la evaluación de $\texttt{e}_2$.

5. A partir de dos expresiones $\texttt{e}_1$ y $\texttt{e}_2$ del mismo tipo cualquiera[1], se pueden formar las expresiones `e1 == e2` y `e1 /= e2`, ambas del tipo booleano, cuyos valores serán,

[1] La igualdad en *Trinity* es homogénea. No se puede formar una expresión con los operadores `==` y `/=` entre dos expresiones con tipos diferentes. Por lo tanto, dos valores de tipos diferentes no pueden compararse entre sí ni siquiera para resultar en que son distintos. La forma `42 == { 54 }` **no** es una expresión con valor `false`, sino un fragmento de programa mal formado que debe arrojar un error estáticamente.

respectivamente, `true` y `false` si los valores de e_1 y e_2 son iguales. Ninguno de los dos operadores es asociativo y ambos tienen igual precedencia.

Por ejemplo, las siguientes expresiones tienen el mismo valor que la expresión `true`:

```
2 /= 4
```

```
true == not false
```

```
{ 1, 2 : 3, 4 } /= { 3, 4 : 1, 2 }
```

6. La evaluación de `e1 == e2` y `e1 /= e2` procede con la evaluación de e_1; al terminarla, procede a evaluar e_2; al terminarla, verifica si ambos valores producidos por las subexpresiones son iguales, y produce el resultado final. El mecanismo de evaluación análogo para expresiones formadas por otros operadores binarios será referido en el resto de este documento como estricta de izquierda a derecha.

7. A partir de dos expresiones e_1 y e_2 del tipo numérico, se pueden formar las expresiones `e1 <= e2`, `e1 < e2`, `e1 >= e2` y `e1 > e2`, todas del tipo booleano, cuyos valores serán `true` únicamente si el valor de e_1 es, respectivamente, menor or igual, menor, mayor o igual o mayor que el valor de e_2. Ninguno de los cuatro operadores es asociativo y todos tienen precedencia igual a la de `==` y `/=`. La evaluación de estas expresiones es estricta de izquierda a derecha.

   Por ejemplo, la siguiente expresión tiene el mismo valor que la expresión `true`:

   ```
   4 <= 4 & 2 < 4 & 4 >= 2 & 4 > 2
   ```

2.3.2 Operadores aritméticos

1. A partir de dos expresiones e_1 y e_2 de un mismo tipo aritmético, se pueden formar las expresiones `e1 + e2` y `e1 - e2`, ambas del mismo tipo aritmético original, cuyos valores serán, respectivamente, la suma y la resta de los valores de e_1 y e_2. Ambos operadores tienen asociatividad izquierda y tienen igual precedencia. La evaluación de estas expresiones es estricta de izquierda a derecha.

   Por ejemplo, las siguientes expresiones tienen el mismo valor que la expresión `true`:

   ```
   2 + 4 == 6
   ```

   ```
   { 1, 2 } + { 2, 1 } == { 3, 3 }
   ```

2. A partir de una expresión `e` de un tipo aritmético, se puede formar la expresión `- e` del mismo tipo, cuyo valor será el inverso aditivo de `e`. Por ejemplo, `- (3 * (- 3)) == 9`. La evaluación de `- e` es estricta.

3. A partir de expresiones $\texttt{e}_1$ del tipo especificado por `matrix(n, m)` y $\texttt{e}_2$ del tipo especificado por `matrix(m, p)`, se puede formar la expresión `e1 * e2` del tipo especificado por `matrix(n, p)`, cuyo valor se calculará como el producto matricial de los valores de $\texttt{e}_1$ y $\texttt{e}_2$. Por otra parte, a partir de expresiones $\texttt{e}_1$ y $\texttt{e}_2$ del tipo numérico, se puede formar la expresión `e1 * e2` del tipo numérico, cuyo valor será el producto (escalar) de los valores de $\texttt{e}_1$ y $\texttt{e}_2$. El operador `*` tiene asociatividad izquierda y mayor precedencia que los operadores `+` y `-`. La evaluación de estas expresiones es estricta de izquierda a derecha.

   Por ejemplo, las siguientes expresiones tienen el mismo valor que la expresión `true`:

   ```
   { 1, 2, 3 } * { 1 : 2 : 3 } == { 14 }

   2 * 3 == 6
   ```

4. A partir de expresiones $\texttt{e}_1$ y $\texttt{e}_2$ del tipo numérico, se pueden formar las expresiones `e1 / e2` y `e1 % e2`, ambas del tipo numérico, cuyos valores serán, respectivamente, la división exacta y el resto exacto de la división entera entre los valores de $\texttt{e}_1$ y $\texttt{e}_2$. Los operadores `/` y `%` tienen asociatividad izquierda y ambos tienen la misma precedencia que el operador `*`. La evaluación de estas expresiones es estricta de izquierda a derecha.

   Por ejemplo, las siguientes expresiones tienen el mismo valor que la expresión `true`:

   ```
   5 / 2 == 2.5

   10.2 % 2 == 0.2
   ```

5. A partir de expresiones $\texttt{e}_1$ y $\texttt{e}_2$ del tipo numérico, se pueden formar las expresiones `e1 div e2` y `e1 mod e2`, ambas del tipo numérico, cuyos valores serán, respectivamente, la división entera y el resto entero de la división entre los valores de $\texttt{e}_1$ y $\texttt{e}_2$. Los operadores `div` y `mod` tienen asociatividad izquierda y ambos tienen la misma precedencia que el operador `*`. La evaluación de estas expresiones es estricta de izquierda a derecha. Los lexemas `div` y `mod` son palabras reservadas.

   Por ejemplo, las siguientes expresiones tienen el mismo valor que la expresión `true`:

   ```
   5 div 2 == 2

   10.2 mod 2 == 0
   ```

6. A partir de una expresión `e` del tipo especificado `matrix(n, m)`, se puede formar la expresión `e'` del tipo especificado por `matrix(m, n)`, cuyo valor será la traspuesta del valor de `e`. El operador `'` tiene mayor precedencia que el operador `*`. La evaluación de estas expresiones es estricta.

   Por ejemplo,

   ```
   { 1, 2, 3 : 4, 5, 6 }'
   ```

   es una expresión del tipo especificado por `matrix(3, 2)` y su valor es igual al de la expresión

   ```
   { 1, 4 : 2, 5 : 3, 6 }
   ```

7. Todos los operadores aritméticos tienen mayor precedencia que todos los operadores lógicos.

2.3.3 Operadores cruzados

1. *Trinity* soporta operaciones cruzadas entre cantidades escalares y matriciales. Estas operaciones aplican a cada elemento de una cantidad matricial una misma operación con una cantidad escalar.

2. Cada operador binario entre cantidades escalares que resulte en otra cantidad escalar tiene un operador cruzado correspondiente. Los operadores cruzados `.+.`, `.-.`, `.*.`, `./.`, `.%.`, `.div.` y `.mod.` corresponden a los operadores escalares `+`, `-`, `*`, `/`, `%`, `div` y `mod`, respectivamente.

3. A partir de expresiones $\texttt{e}_1$ del tipo numérico y $\texttt{e}_2$ de un tipo matricial, y para cada operador cruzado `.op.`, se pueden formar las expresiones `e1 .op. e2` y `e2 .op. e1` del mismo tipo de $\texttt{e}_2$, cuyos valores son la cantidad matricial obtenida al sustituir cada componente `xuv` de $\texttt{e}_2$ por `e1 op xuv` o `xuv op e1`, respectivamente, donde `op` es el operador escalar correspondiente de `.op.`. La evaluación de estas expresiones es estricta de izquierda a derecha.

4. Por ejemplo,

   ```
   { 1, 2, 3 } .*. 10
   ```

   es una expresión del tipo especificado por `matrix(2, 3)` y su valor es igual al de la expresión


```
{ 1*10, 2*10, 3*10 }
```

De igual manera,

```
12 ./. { 2, 3 : 4, 8 }
```

es una expresión del tipo especificado por `matrix(2, 2)` y su valor es igual al de la expresión

```
{ 6, 4 : 3, 1.5 }
```

2.3.4 Proyección de componentes

1. A partir de una expresión `e` del tipo especificado por `matrix(n, m)`, y dos expresiones `i` y `j` del tipo numérico, se puede formar la expresión `e[i, j]` del tipo numérico, cuyo valor será la componente del valor de `e` en la fila cuyo índice sea el valor de la expresión `i`, y en la columna cuyo índice sea el valor de la expresión `j`. Esta operación tiene la máxima precedencia.

2. La evaluación de `e[i, j]` sigue los siguientes pasos:

    1. Se procede con la evaluación de `e`.
    2. Una vez terminada y producido su valor, se procede con la evaluación de `i`.
    3. Una vez terminada y producido su valor, se verifica si ese valor es un entero entre 1 y el número de filas del tipo de la expresión `e`.
    4. Si esta condición no se cumple, la ejecución del programa abortará. Si la condición sí se cumple, se procede con la evaluación de `j`.
    5. Una vez terminada y producido su valor, se verifica si ese valor es un entero entre 1 y el número de columnas del tipo de la expresión `e`.
    6. Si esta condición no se cumple, la ejecución del programa abortará. Si la condición sí se cumple, se produce como resultado final de la evaluación de `e[i, j]` el valor en la posición del valor producido al evaluar `e` en la fila y la columna especificadas por los valores producidos al evaluar `i` y `j`.

3. Por ejemplo, la expresión

    ```
    { 1, 2 : 3, 4 }[1 + 1, 2 - 1]
    ```

    tiene el mismo valor que la expresión `3`, mientras que la expresión


```
{ 1, 2 : 3, 4 }[30, 1]
```

aborta el programa al ser evaluada.

4. A partir de una expresión `e` del tipo especificado por `row(n)` o `col(m)`, y una expresión `i` del tipo numérico, se puede formar la expresión `e[i]` del tipo numérico. Si `e` tiene el tipo especificado por `row(n)`, entonces `e[i]` tiene el mismo significado que `e[1, i]`, y si `e` tiene el tipo especificado por `col(m)`, entonces `e[i]` tiene el mismo significado que `e[i, 1]`.

   Por ejemplo, todas estas expresiones tienen el mismo significado:

```
{ 10 , 20 , 30 }[2]
```

```
{ 10 , 20 , 30 }[1, 2]
```

```
{ 10 : 20 : 30 }[2]
```

```
{ 10 : 20 : 30 }[2, 1]
```

2.4 Abstracción y control

1. *Trinity* es un lenguaje imperativo, y su mecanismo principal de operación es la ejecución de secuencias de instrucciones. Algunas instrucciones son estructuras de control que pueden contener otras secuencias de instrucciones que se ejecuten de manera condicional o reiterada.

2. A partir de una o más instrucciones $\texttt{i}_1$, $\texttt{i}_2$, …, $\texttt{i}_n$, se puede formar una secuencia de instrucciones $\texttt{i}_1\ \texttt{i}_2\ \ldots\ \texttt{i}_n$ simplemente escribiéndolas en secuencia.

3. A partir de una expresión `e` de cualquier tipo, se puede formar la instrucción `e;`. La ejecución de esta instrucción procede con la evaluación de `e` y descarta su valor.

2.4.1 Impresión

1. A partir de expresiones $\texttt{e}_1$, $\texttt{e}_2$. …, $\texttt{e}_n$ de tipos cualesquiera, o literales de cadena de caracteres, se puede formar la instrucción $\texttt{print e}_1\texttt{, e}_2\ldots\ \texttt{e}_n\texttt{;}$. La ejecución de esta instrucción procede con la evaluación de $\texttt{e}_1$, $\texttt{e}_2$ … $\texttt{e}_n$ en el orden de aparición, exceptuando aquellas que no sean expresiones sino literales de cadena de caracteres, y al todas haber producido sus valores, se escribe a la salida estándar del programa una representación textual de los valores obtenidos, o el texto representado por los literales de cadena de caracteres, en el mismo orden de aparición. El lexema `print` es una palabra reservada.

Por ejemplo,

```
print { 1, 2, 3 }, 4.5;
```

es una instrucción de impresión.

2. La instrucción de impresión **no imprime un salto de línea** después de haber impreso la lista completa de argumentos, ni entre la impresión correspondiente a cada uno de sus argumentos.

3. En un literal de cadena de caracteres, el caracter *backslash* solo puede ocurrir como parte de una secuencia de escape. Una secuencia de escape es un caracter *backslash* seguido de una `n` (`\n`), otro *backslash* (`\\`), o una comilla doble (`\"`). Una secuencia de escape representa texto: `\n` representa un fin de línea, `\\` representa un caracter *backslash*, y `\"` representa una comilla doble.

4. Un literal de cadena de caracteres es una secuencia de caracteres encerrada entre comillas dobles (`"`) y que no contiene fines de linea ni comillas dobles, salvo como parte de una secuencia de escape. Las comillas dobles que delimitan a un literal de cadena de caracteres no pueden ser parte de una secuencia de escape. El texto representado por un literal de cadena de caracteres es el mismo texto que lo compone, sin sus delimitadores, y sustituyendo las secuencias de escape por el texto que representan.

5. Por ejemplo, al ejecutarse la instrucción de impresión

   ```
   print "Hello world!\nA string inside a string: \"this is so \\meta\""
   ```

   el programa escribirá esto a su salida estándar:

   ```
   Hello world!
   A string inside a string: "this is so \meta"
   ```

2.4.2 Variables y alcances

1. *Trinity* utiliza alcance estático y requiere que las variables sean declaradas con sus tipos antes de poder usarse o asignarse. Esta sección define con precisión el funcionamiento del alcance de las variables en el lenguaje.

2. Un identificador es una secuencia de caracteres no vacía compuesta por letras de la `A` hasta la `Z` (minúscula o mayúscula), los dígitos del `0` al `9` y el caracter *underscore* (`_`). Los identificadores deben comenzar por letras, no pueden comenzar ni por dígitos ni por `_`, y son sensibles a mayúsculas; por ejemplo, el identificador `foo` es distinto del

identificador `fOo`. Además de estas restricciones, aquellas secuencias de caracteres que sean palabras reservadas no son identificadores.

Por ejemplo, `print` y `not` no son identificadores, porque son palabras reservadas, pero `pRint` sí es un identificador.

3. No es necesario que se reconozcan letras con tildes ni la letra eñe — *Trinity* solo requiere identificadores hechos de caracteres definidos por *Unicode* en el *script* *Basic Latin* (equivalente a ASCII).

4. Una variable es una ubicación de memoria modificable que hace referencia a un valor. Pueden hacerse disponibles variables en las instrucciones contenidas directa o indirectamente en una secuencia de instrucciones particular, así como en las expresiones que éstas contengan. Para hacerlo, la secuencia de instrucciones se incluye en una instrucción de bloque que declare esa variable y la asocie a un nombre. Las instrucciones incluidas en ese bloque, y las expresiones que contengan, podrán referirse a la variable a través del nombre que se le asoció en el bloque donde fue declarada.

5. A partir de un identificador `i`, una especificación de tipo `t` y una expresión `e` del tipo especificado por `t`, se puede formar una declaración de variable `t i;` o una declaración de variable con inicialización `t i = e;`. A partir de una o más declaraciones $\texttt{d}_1$, $\texttt{d}_2$, …, $\texttt{d}_n$ donde no se declare el mismo identificador en más de una declaración, se puede formar una secuencia de declaraciones $\texttt{d}_1$ $\texttt{d}_2$ … $\texttt{d}_n$ simplemente escribiéndolas en secuencia.

   Por ejemplo,

   ```
   matrix(9, 9) foo;
   ```

   es una declaración de variable para el identificador `foo` y el tipo `matrix(9, 9)` —y también una secuencia de declaraciones con una sola declaración—, y

   ```
   number i = 42; boolean b;
   row(5) F_5 = { 1, 1, 2, 3, 5 };
   ```

   es una secuencia de declaraciones en la cual hay dos declaraciones de variable con inicialización (las de los identificadores `i` y `F_5`), y una declaración de variable regular sin inicialización (la del identificador `b`).

6. A partir de una secuencia de instrucciones `is` y una secuencia de declaraciones `ds`, se puede formar una instrucción de bloque `use ds in is end;`.

   Por ejemplo, esta es una instrucción de bloque:


```
use
  number tau = 6.2831853;
  matrix(2, 2) m = { 4, 2 : 1, 1/2 };
in
  print tau, m[2, 1]
end;
```

7. La secuencia de declaraciones en cada instrucción de bloque produce un contexto en el cual cada declaración hace disponible una variable del tipo de la declaración que se asocia al identificador de la declaración. Ese contexto se extiende sobre todas las instrucciones y expresiones contenidas directa o indirectamente en la secuencia de instrucciones de la instrucción de bloque. Si el alcance de un contexto de declaración ocurre dentro del alcance de otro, entonces en el alcance del contexto interior, la asociación de la declaración del contexto interior sustituye a la asociación de la declaración del contexto exterior si usa el mismo identificador.

   Por ejemplo, en la secuencia de instrucciones

```
print 1;
print 2;
use
  number foo = 42;
  matrix(1, 1) bar;
in
  print 3;
  print 4;
  use
    boolean foo;
  in
    print 5;
    print 6;
  end;
  print 7;
end;
print 8;
print 9;
```

   Las tres declaraciones introducen regiones del programa donde hay identificadores asociados a tipos:

   - Las instrucciones de impresión que imprimen 1, 2, 8 y 9 no están dentro de la extensión del contexto de ninguna declaración.

- Las instrucciones que imprimen 3, 4 y 7 están dentro de la extensión del contexto en el cual los identificadores `foo` y `bar` están asociados a un número y a una matriz, respectivamente. :w
- Las instrucciones que imprimen 5 y 6 están dentro de la extensión del contexto antes mencionado, y también dentro de la extensión del contexto establecido por la declaración `boolean foo;`.

Debido a la regla de sustitución, en el alcance del contexto interior (establecido por la declaración `boolean foo;`), la asociación del identificador `foo` con una variable del tipo booleano *sustituye* a la asociación del contexto exterior de `foo` con una variable del tipo numérico. Esa sustitución *únicamente* aplica para las instrucciones que imprimen 5 y 6.

8. A partir de un identificador `i`, puede utilizarse `i` como una expresión de uso de variable, que es del tipo `t`, en la extensión de un contexto producido por una declaración que asocie al identificador `i` con el tipo `t`.

   Por ejemplo, el programa

```
use
  number foo = 10 + 5;
in
  print foo + 27;
end;
```

   está bien formado, porque el uso del identificador `foo` como una expresión del tipo numérico ocurre en la extensión del contexto establecido por la declaración `number foo = 10 + 5;`, que asocia el identificador `foo` a una variable del tipo numérico (y además le asocia el valor de la expresión `10 + 5`).

9. Un programa en *Trinity* está mal formado si un identificador `i` se utiliza como expresión fuera de la extensión de un contexto de alguna declaración que asocie ese identificador con algún tipo. En otras palabras: un identificador solo puede usarse como expresión en la extensión de su declaración.

   Por ejemplo,

```
print 1;
use
  number foor = 42;
in
  print 2;
end;
```


```
print 3;
print foor;
print 5;
```

es un programa mal formado, porque el uso del identificador `foor` como expresión ocurre fuera de la extensión del contexto establecido por la declaración.

10. Un programa en *Trinity* también está mal formado si un identificador `i` se utiliza como expresión en la extensión de un contexto de una declaración asocie ese identificador con algún tipo, pero que se use como expresión de un tipo distinto al asociado a `i` por la declaración con el contexto más interior que asocia a ese identificador con algún tipo en ese punto. En otras palabras: un identificador solo puede usarse como expresión de un tipo en aquellas partes del programa donde la declaración más cercana de ese identificador lo asocie a ese mismo tipo.

    Por ejemplo,

```
use
  boolean foo = true;
in
  print 0 == foo;
end;
```

    es un programa mal formado, porque aunque `foo` se usa como identificador dentro de la extensión del contexto establecido por una declaración que lo asocia a un tipo, se usa `foo` con el tipo numérico, pero está asociado al tipo booleano en la declaración relevante.

2.4.3 Asignación

1. *Trinity* es un lenguaje imperativo con variables con el modelo de valor. Esta sección define con precisión el funcionamiento de la asignación en el lenguaje.

    El siguiente ejemplo es válido en *Trinity* y pone en evidencia las reglas de alcance:

```
program
  use
    number x, y; # inicializados en 0
  in
    use
      row(3) x;
      col(3) y;
```


```
    in
      set x = { 1 , 2 , 3 };
      set y = { 1 : 2 : 3 };
      print "Print 1", x; # x es del tipo especificado por matrix(1, 3)
    end;

    use
      boolean x, y;
    in
      set x = true;
      print "Print 2", x; # x es del tipo booleano
    end;

    print "Print 3", x; # x es del tipo numérico

    for x in
      { 1, 2, 3 : 4, 5, 6 }
    do
      use
      bool x; # esconde la x declarada en el for y se inicializa en 'false'
      in
        set x = not x;
        print "Print 4", x; # x es del tipo booleano
      end;
    end;
  end;
end;
```

2. A partir de un identificador `i` y una expresión `e` del tipo `t`, puede utilizarse la instrucción de asignación `set i = e;` como instrucción en la extensión de un contexto producido por una declaración que asocie al identificador `i` con el tipo `t`. La ejecución de esta instrucción procede con la evaluación de `e`, y al producir su valor, lo asocia al identificador `i` en toda la extensión de la asociación de la declaración correspondiente. El lexema `set` es una palabra reservada.

   Por ejemplo, en

```
use
  boolean go = true;
in
  while
    go
```


```
  do
    print 1;
    set go = false & true;
    print 2;
  end;
end;
```

a pesar de que la condición del ciclo no está sintacticamente después de la asignación, sigue estando la condición dentro de la extensión del contexto establecido por la declaración, así que la asociación del valor computado de `false & true` (que es igual al de `false`) aplica para la condición del ciclo, por lo cual el cuerpo del ciclo solo es ejecutado una vez.

3. Las declaraciones de variable con inicialización se comportan como asignaciones: ejecutar la instrucción donde ocurren ejecuta una asignación implícita al identificador de la declaración con la expresión dada en la inicialización. Si hay varias declaraciones en una secuencia, las asignaciones de inicialización se ejecutan en el mismo orden en que ocurren en la secuencia.

   Por ejemplo, en

```
use
  number p = 42;
  row(2) q = { 6, 7 };
in
  use
    boolean r = q[p] == 0;
  in
    print "inalcanzable!";
  end;
end;
```

   el programa aborta antes de imprimir, ya que se ejecuta la asignación `set r = q[p] == 0`, lo cual evalúa la expresión `q[p] == 0`, que comienza por evaluar `q[r]`, pero `p` está asociada al valor de la expresión `42`, que sobrepasa los límites de la cantidad matricial de la cual se desea obtener un elemento..

4. Las declaraciones de variable sin inicialización se comportan como declaraciones de variable con inicialización donde la expresión de inicialización es un literal del tipo apropiado con valores por defecto. La inicialización por defecto para el tipo booleano es con el literal `false`, para el tipo numérico es con el literal `0`, y para tipos matriciales, es un literal matricial de las dimensiones adecuadas al tipo y con `0` en todas las componentes.

Por ejemplo, en

```
use
  boolean b;
  number n;
  matrix(3, 2) m;
in
  print b, n, m;
end;
```

el programa imprime `false`, `0`, y un representación textual de una matriz 3×2 llena de ceros.

5. La ejecución de una instrucción de uso de variable con el identificador `i` produce como valor el último valor que se asociara a `i` en una instrucción de asignación, sea implícita o explícita, y no tiene otros efectos.

   Por ejemplo, en

   ```
   use
     number n;
   in
     print n;
     set n = 42;
     print n;
   end;
   ```

   el programa imprime `0` (porque al momento de la primera impresión, la última instrucción de asignación ejecutada fue la asignación implícita `set n = 0;` en la declaración sin inicialización `number n;`, y *luego* de la asignación explícita `set n = 42;`, el programa imprime `42`.

6. A partir de un identificador `m`, dos expresiones `i` y `j` del tipo numérico, y una expresión `e` del tipo `t`, puede utilizarse la instrucción de asignación matricial `set m[i, j] = e;` como instrucción en la extensión de un contexto producido por una declaración que asocie al identificador `m` con el tipo especificado por `matrix(n, m)`.

   Por ejemplo,

   ```
   use
     number n;
   in
     set n[1] = 2;
   end;
   ```

es un programa mal formado, porque el identificador `n` se asocia con el tipo numérico en el contexto establecido por la declaración `number n;`, pero el tipo numérico no es especificado por una especificación de tipo de la forma `matrix(n, m)` —es decir, no es una variable matricial— así que no puede utilizarse en una instrucción de asignación matricial en la extensión de ese contexto.

En cambio,

```
use
  matrix(2, 3) n;
in
  set n[1] = 2;
end;
```

sí es un programa bien formado.

7. La ejecución de `set m[i, j] = e` sigue los siguientes pasos:

   2. Se procede con la evaluación de `i`.
   3. Una vez terminada y producido su valor, se verifica si ese valor es un entero entre 1 y el número de filas del tipo asociado al identificador `m`.
   4. Si esta condición no se cumple, la ejecución del programa abortará. Si la condición sí se cumple, se procede con la evaluación de `j`.
   5. Una vez terminada y producido su valor, se verifica si ese valor es un entero entre 1 y el número de columnas del tipo asociado al identificador `m`.
   6. Si esta condición no se cumple, la ejecución del programa abortará. Si la condición sí se cumple, se procede con la evaluación de `e`.
   7. Una vez terminada y producido su valor, se asocia al identificador `m` un valor igual al que tuviera asociado en el momento de la asignación, salvo por el valor en la componente en la fila y columna especificadas por los valores producidos en la evaluación de las expresiones `i` y `j`. El valor de esa componente del valor nuevo asociado a `m` será el valor producido por la evaluación de la expresión `e`.

8. Por ejemplo, en

```
use
  matrix(2, 3) m = { 1, 2, 3 : 4, 5, 6 };
in
  print m;
  set m[2, 3] = 42;
  print m;
end;
```

la primera impresión escribe

```
|  1  2  3 |
|  4  5  6 |
```

a la salida estándar del programa, mientras que la segunda escribe

```
|  1  2  3 |
|  4  5 42 |
```

9. Esta definición de asignación tiene como consecuencia que *Trinity* utiliza el modelo de valor. Por ejemplo, en

```
use
  row(1) box1;
  row(1) box2;
in
  set box1[1] = 27;
  set box2 = box1;
  set box1[1] = 42;
  print box2;
end;
```

la instrucción `set box2 = box1` evalúa la expresión `box1` y asocia el **valor** de esa expresión a `box2`. Por lo tanto, `box2` tendrá una **copia** del estado de `box1` al momento de realizar la asignación, que será igual al de la expresión `{ 27 }`.

Luego, la instrucción `set box1[1] = 42;` no tiene efecto alguno sobre el valor asociado al identificador `box2`, así que el programa imprime

```
| 27 |
```

2.4.4 Lectura de entrada

1. A partir de un identificador `i`, puede utilizarse la instrucción de lectura de entrada `read i;` como instrucción en la extensión de un contexto producido por una declaración que asocie al identificador `i` con el tipo numérico o booleano. La ejecución de esta instrucción procede con la lectura de una línea de la entrada estándar del programa y la conversión de los datos leídos en un valor del tipo asociado al identificador `i`; el valor convertido se asocia con el identificador `i` en toda la extensión de la asociación de la declaración correspondiente.

2. La conversión de datos de entrada a un valor del tipo booleano solo es exitosa si los datos de entrada son exactamente `false` o `true`, y el valor convertido es el de la expresión `false` o `true`, respectivamente. La conversión de datos de entrada a un valor del tipo numérico solo es exitosa si los datos de entrada tienen el formato exacto de un literal numérico de *Trinity*, y el valor convertido es el que ese literal numérico denota en *Trinity*.

3. Si la conversión de los datos de entrada no es exitosa, la instrucción de lectura de entrada produce un mensaje de error en la salida estándar del programa y aborta su ejecución.

2.4.5 Instrucciones condicionales

```
if <condición> then <instrucciones> else <instrucciones> end;

if <condición> then <instrucciones> end;
```

La condición debe ser una expresión de tipo booleano, de lo contrario debe arrojarse un error.

Ejecutar está instrucción tiene el efecto de evaluar la condición y si su valor es `true` se ejecuta la `<instrucción 1>`; si su valor es `false` se ejecuta la `<instrucción 2>`. Es posible omitir la palabra clave `else` y la `<instrucción 2>` asociada, de manera que si la expresión vale `false` no se ejecuta ninguna instrucción.

2.4.6 Iteración determinada

```
for <identificador> in <vector/matrix> do <instrucciones> end;
```

Para ejecutar esta instrucción se evalúa la expresión `<vector/matrix>` que puede ser tanto una vector como una matriz, su tipo debe ser matricial, y a la variable `<identificador>` se le asigna cada valor dentro de la matriz, avanzando de izquierda a derecha y de arriba hacia abajo. Para cada valor de `<identificador>` se ejecuta `<instrucción>`. Note que cambiar el valor de la variable `<identificador>` **no cambiará el valor** de la posición representada por éste de la matriz.

La instrucción declara automáticamente una variable llamada `<identificador>` de tipo numérico y local al cuerpo de la iteración.

2.4.7 Iteración indeterminada

```
while <condición> do <instrucciones> end;
```

La condición debe ser una expresión de tipo booleano. Para ejecutar la `<instrucción>` se evalúa la `<condición>`, si es igual a `false` termina la iteración; si es `true` se ejecuta la `<instrucción>` del cuerpo y se repite el proceso.

2.4.8 Funciones

`function <identificador>(<parámetros>) return <tipo> begin <instrucciones> end;`

Las funciones en *Trinity* tienen un nombre, una lista de parámetros, un tipo de retorno y un cuerpo con una lista de instrucciones. Los parámetros pueden usarse en el cuerpo como variables. El pasaje de parámetros es por valor. La instrucción de retorno `return e` puede usarse en el cuerpo de la función para retornar el valor de la expresión `e`. La lista de parámetros es una secuencia de declaraciones.

Las funciones pueden llamarse en una expresión de la forma `<identificador>(<argumentos>)`, donde `<argumentos>` es una lista de expresiones separadas por comas que sean de igual número y con los mismos tipos en el mismo orden que las declaraciones en la lista de parámetros de la función del mismo nombre. Evaluar la expresión implica evaluar todos los argumentos en orden, asignar sus valores a los parámetros correspondientes, y ejecutar el cuerpo. La última instrucción ejecutada en el cuerpo debe ser una instrucción de retorno, y el valor de la llamada a la función será el valor de la primera expresión de retorno que se ejecute del cuerpo de la función llamada. La ejecución de una función no continúa luego de ejecutar una instrucción de retorno.

2.4.9 Estructura de un programa

Un programa de *Trinity* tiene la siguiente estructura:

```
<funciones>
program
  <instrucciones>;
end;
```