UNIVERSIDAD NACIONAL DE LA PATAGONIA
SAN JUAN BOSCO

FACULTAD DE INGENIERIA
2012

Tesis de Licenciatura

UN INTÉRPRETE MULTIPLATAFORMA PARA LA INICIACION A LA PROGRAMACION ESTRUCTURADA Y CONCURRENTE

DaVinci Concurrente

Autor:

APU Daniel E. Aguil Mallea

Director:

Lic. Beatriz Depetris

Contenido

Índice de tablas

1 - Introducción

1.1 - Objeto de estudio

Desde hace varios años me desempeño como docente en un curso CS1 (Computer Science 1), donde utilizo desde los comienzos del mismo la versión existente del lenguaje Visual DaVinci como herramienta para el desarrollo inicial de capacidades para la resolución de problemas y la expresión de las soluciones en un lenguaje de programación concreto.

Visual DaVinci es un lenguaje visual que posee un entorno de desarrollo integrado que permite tanto la codificación en forma textual como visual y tiene la particularidad de que la ejecución de un programa se puede visualizar a través de una ciudad por la que puede circular un robot interactuando con objetos (flores, papeles, etc.) distribuidos en ella.

Si bien Visual Davinci ha demostrado ser un lenguaje sólido para la iniciación del alumno en el aprendizaje de la programación, el uso intensivo de la herramienta y los avances tecnológicos me han motivado para realizar modificaciones y extensiones al mismo.

Estas modificaciones abarcan aspectos relacionados con la especificación del lenguaje, la interface, la depuración de errores y la introducción de nuevas estructuras y primitivas.

Las extensiones, en su mayoría, están relacionadas específicamente con los conceptos básicos de la programación concurrente.

1.2 - Reseña histórica

Con el surgimiento de la computadora con programa almacenado, iniciado por John Von Neumann, se hizo necesario escribir programas para que la máquina realice la acción deseada. Estos programas se escribían en lenguaje de máquina, que básicamente son códigos numéricos que representan las operaciones reales de la máquina. Debido a que la escritura de tales códigos es muy compleja y lleva mucho tiempo, esta forma de codificación pronto fue remplazada por el lenguaje ensamblador, en el cual las instrucciones y las localidades de memoria son referenciadas simbólicamente. Este avance fue el que marcó el inicio de lo que hoy conocemos como lenguajes de programación.

El lenguaje ensamblador mejoró enormemente la velocidad con la que podían escribirse los programas. Aún en nuestros días se los utiliza cuando se necesita una gran velocidad o brevedad en el código.

El siguiente avance en la evolución de la programación fue permitir escribir las operaciones de un programa de una manera parecida a la notación matemática o lenguaje natural y que fueran independientes de la máquina, pudiéndose luego traducir mediante un programa para convertirlas en código ejecutable.

En la década del 50' con la aparición del lenguaje FORTRAN (FORmula TRANslating system) se consolida el concepto de *Traductor*, que es básicamente un programa que traduce un lenguaje a otro lenguaje. A fines de dicha década surge en Europa una corriente universitaria que pretende que la definición del lenguaje sea independiente de la máquina, ya que los lenguajes, hasta ese entonces, estaban altamente ligados a la arquitectura subyacente (por ejemplo la primer versión de FORTRAN estaba ligado con la arquitectura de la máquina IBM704 (Pratt, Terrence W. y Zelkowitz, Marvin V. 1998).

ALGOL es el primer lenguaje que se independiza de la máquina e introduce muchos de los conceptos utilizados hoy en día por los lenguajes de programación. Algunos ejemplos son: la definición de la sintaxis en formato BNF, la declaración explícita del tipo de datos de los identificadores, las estructuras iterativas, la recursividad, el pasaje de parámetros por valor y por nombre y las estructuras de bloque.

A finales de la década del 60' aparece el teorema del programa estructurado y nace una nueva forma de programar, la programación estructurada, que facilita la comprensión y expresión de algoritmos, además de aumentar la fiabilidad de los mismos. Böhm y Jacopini demuestran formalmente que la sentencia GOTO no es

estrictamente necesaria en los lenguajes de programación de alto nivel y que cualquier programa que la utilice puede ser reemplazado por otro equivalente que no haga uso de la misma.

Lo demostrado por Böhm y Jacopini no fue tenido en cuenta hasta que Dijkstra probó formalmente que todo programa puede escribirse utilizando las estructuras de control *secuencia*, *selección* e *iteración*, prescindiendo de la sentencia GOTO. Además declaró que su uso generaba un mal hábito de programación.

Desde la aparición de los primeros lenguajes de programación hasta nuestros días han surgido centenares, tanto de propósito general como específico. Sin embargo se continua aún con la construcción de los mismos, debido al aporte significativo en distintas áreas y que, además contamos con métodos, estándares, técnicas y herramientas que nos facilitan su producción. Es importante destacar que solo varios (decenas) de ellos se utilizan en la actualidad. Estos lenguajes se pueden agrupar en dos grandes categorías: lenguajes declarativos y lenguajes imperativos.

La programación declarativa se refiere a que el programador le da instrucciones a la máquina de *qué* resultados quiere conseguir y en la programación imperativa se le da instrucciones de *cómo* conseguir los resultados. Dentro de estas categorías, dependiendo de cómo sea la metodología para la resolución de un problema, podemos volver a categorizarlos en *Paradigmas de Programación.* Floyd fue uno de los primeros que habló en términos de paradigma y lo definió como *un proceso de diseño que va más allá de una gramática, reglas semánticas y algoritmos, sino que es un conjunto de métodos sistemáticos aplicables en todos los niveles del diseño de programas.*

Por lo tanto, si nos referimos a los lenguajes de programación en términos de Paradigmas podemos agruparlos de la siguiente manera:

Paradigma Funcional: Considera al programa como una función, donde el dominio de tal función representa al conjunto de todas las entradas posibles y el rango sería el conjunto de todas las salidas posibles. Es importante mencionar que en este paradigma no existe el concepto de variable y que además carece de estructuras de control como la secuencia e iteración.

Paradigma Lógico: Se basa en el uso de la lógica. Se especifican hechos y reglas de inferencias y el lenguaje utiliza la deducción para responder consultas.

Paradigma Orientado a Objetos: Se basa en el diseño y construcción de objetos. El programador especifica los datos (involucrados en el problema) y se asocian a éstos operaciones necesarias para su manipulación (métodos).

Paradigma Imperativo: Describe la programación en función del estado del programa y sentencias que cambian su estado. El programador indica que debe hacer la máquina y como deberá hacerse. Este paradigma es el que agrupa a la mayoría de los lenguajes.

Dado que existen diferentes paradigmas para resolver un problema, no es una tarea trivial la elección de alguno en particular para la resolución del mismo. Debido a la filosofía de trabajo (métodos involucrados) de los distintos paradigmas con frecuencia nos veremos en la situación de elegir el que mejor se ajuste a nuestras necesidades

El presente trabajo está enfocado en brindar una herramienta más en los procesos de enseñanza y aprendizaje de los conceptos básicos de la programación estructurada y concurrente.

1.3 - Programación estructurada

Uno de los puntos más importantes a tener en cuenta cuando se programa es el control de la ejecución, es decir, el camino lógico que va a seguir el programa desde principio a fin. Tendremos una serie de instrucciones que se ejecutarán una tras otra, la posibilidad de elegir un camino u otro, dependiendo de que se de alguna condición, y la repetición de una secuencia de instrucciones una cantidad fija de veces o también dependiendo de cierta condición.

La facilidad de lectura y compresión de un algoritmo desde principio a fin es consecuencia de la utilización únicamente de las estructuras de control secuencia, repetición e iteración. Estas estructuras están caracterizadas por tener un único punto de entrada y un único punto de salida. La programación estructurada es la técnica para escribir programas utilizando únicamente las estructuras de control recién mencionadas, destacándose la no utilización de la sentencias de salto incondicional.

Generalmente, esta técnica también es acompañada con una metodología que ayuda a resolver problemas que, en general, resultan ser extensos y complejos, la programación modular. La misma consiste en es la descomposición del problema original en sub-problemas. Esta acción, reflejada en el diseño de un programa, lo veremos cómo sub-programas o módulos. Estos módulos también deben poseer un punto de entrada, el uso de las estructuras que se necesiten (solo las tres mencionadas) y un punto de retorno.

El nombre con el que conocemos esta forma de diseñar, descomponiendo el problema en sub-problemas, es técnica de *diseño descendente o top-down.* Es uno de los métodos más flexibles y potentes para mejorar la productividad de un programa. Esta descomposición, de un programa en módulos y estos módulos en módulos aún más específicos, se conoce también como el método "divide y vencerás". La idea subyacente del método es diseñar los módulos lo más independientes posibles, hasta llegar a la descomposición final de los módulos de manera jerárquica, recordando siempre que cada módulo deberá realizar una tarea específica.

Algunas de las ventajas que nos aporta este método son:

- independencia entre módulos: debido a que los módulos son independientes podríamos tener trabajando de manera simultánea a diferentes programadores en la solución de un problema.

- modificación de los módulos: al ser independientes y cada módulo tener una tarea específica se pueden modificar sin tener que preocuparse por el resto del programa.
- reusabilidad de código: dado que los módulos realizan una tarea específica, una vez pensados e implementados no es necesario volver a pensarlos y mucho menos en volver a implementarlos, pudiendo ser reutilizados.
- mantenimiento del código: ya sea por un mal funcionamiento o un cambio en la lógica del problema será una tarea más sencilla encontrar el lugar y realizar la modificación del código.

1.4 - Programación concurrente

Los conceptos de la programación concurrente surgen con la aparición de sistemas operativos multiprogramados, esto es, básicamente la gestión y ejecución simultánea de varios procesos dentro de un sistema monoprocesador.

Desde la aparición de los primeros sistemas multiprogramados hasta nuestros días, la programación concurrente se ha extendido notablemente. Hoy es un tema de estudio que involucra no solo a los sistemas operativos y a máquinas monoprocesador si no, que se ha extendido al multiprocesamiento y procesamiento distribuido.

En la actualidad la construcción de sistemas concurrentes no requiere de la habilidad que se necesitaba en sus inicios, esto es producto de la evolución de los métodos, diseños, técnicas y lenguajes de programación.

Cada día es más frecuente contar con arquitecturas de hardware que soporten el procesamiento real de procesos en paralelo, por lo que el estudio de métodos y técnicas involucradas en el área de procesamiento paralelo es de vital importancia, pudiendo de esta manera sacar el máximo beneficio en la ejecución de los algoritmos.

Como se menciona anteriormente, la programación concurrente tiene su origen en los sistemas operativos. En sus inicios la programación se realizaba a bajo nivel utilizando ensambladores, ya que no se contaba aún con lenguajes de alto nivel y se requería que el código producido fuera el más eficiente posible. Esto cambió luego de que en 1972 Brinch Hansen desarrollara Concurrent Pascal y dio comienzo a una nueva etapa al permitir el desarrollo de aplicaciones (sistemas operativos en su momento) concurrentes por medio de la utilización de lenguajes de alto nivel.

En la actualizad es común trabajar con arquitecturas que den soporte a la ejecución concurrente y/o paralela. Además los lenguajes más populares de programación nos proveen primitivas o librerías para dar soporte a estas arquitecturas.

Si observamos a la programación desde el punto de vista del paradigma imperativo, podemos ver que un programa es un conjunto de instrucciones y datos, y está escrito en algún lenguaje de programación, donde las instrucciones son ejecutadas de forma secuencial, por lo tanto el procesador ejecuta con los mismos datos el mismo camino (o hilo) lógico de ejecución el cual pertenece a un proceso. A este tipo de programas se lo conoce como programa secuencial.

En un modelo de programación concurrente existen varios hilos de ejecución, trabajando en un paralelismo aparente, ya que no es necesario utilizar un procesador físico para ejecutar cada hilo sino que se ejecutarán en el mismo procesador empleando alguna técnica de planificación de procesos.

Una propiedad fundamental de la programación concurrente es el no determinismo. Esto significa que aun cuando se inicie el proceso con los mismos datos de entrada no siempre se seguirá el mismo camino para arribar al resultado.

La ejecución de un programa que no utiliza concurrencia genera cierta traza o secuencia lógica de ejecución. Si los datos de entrada se mantienen en futuras ejecuciones, la traza será la misma. La ejecución de un programa que utiliza mecanismos concurrentes generará trazas diferentes, aun cuando se utilicen los mismos datos de entrada. Esta última situación lleva a utilizar técnicas de sincronización y comunicación para garantizar el resultado final de un programa.

Dicho esto, podemos definir a la programación concurrente como el conjunto técnicas y escritura de programas para describir el paralelismo potencial que se puede obtener de un problema, como así también para resolver los problemas de comunicación y sincronización que se presenten.

1.5 - Su enseñanza

Desde el punto de vista de la enseñanza de la programación han existido varios cambios desde sus comienzos hasta a nuestros días, surgiendo enfoques y tendencias diferentes. No obstante, en la actualidad no existe consenso sobre que método utilizar para resolver un problema. Esto se ve reflejado directamente en las materias introductorias de las carreras informáticas, existiendo una gran cantidad de enfoques didácticos y como aún no se ha demostrado un alto grado de efectividad de uno respecto de otro el problema persiste. (Szpiniak and Rojo 2006)

Existen métodos de enseñanza que se fundamentan a partir de un paradigma en particular, por ejemplo el imperativo o lógico, y a su vez estos se centran en diversos enfoques didácticos, por ejemplo existen enfoques que utilizan al lenguaje como guía y por medio de su sintaxis y semántica se enseña a programar, también están los que se abstraen del lenguaje y enseñan a programar a través algoritmos escritos en un pseudo-lenguaje más cercano al lenguaje humano y luego se traducen a un lenguaje concreto.

La tarea de enseñar a programar no es fácil, sobre todo teniendo en cuenta que los alumnos vienen por lo general con muy poca noción de lo que implica o con malos hábitos generados por la falta de conceptos de base. Además, generalmente carecen de estructuras cognitivas que permitan la abstracción necesaria para comprender algunos conceptos. Por estos factores es que la tarea de enseñar mediante un enfoque más bien práctico, apoyado en una herramienta que permita la visualización de la ejecución y/o imponer restricciones que sean visibles al alumno es clave, permitiendo así incorporar de forma sencilla abstracciones necesarias a la hora de resolver un problema.

Una herramienta, que cumple en gran medida con estas expectativas y he utilizado durante algunos años, como alumno y luego como docente, es Visual DaVinci y es justamente la herramienta que nos da el puntapié inicial para embarcarnos en el presente trabajo.

2 - Visual DaVinci

2.1 - Introducción

Visual DaVinci es la implementación de la especificación realizada por un grupo de investigadores del LIDI, llevada a cabo por el Lic. Raúl Champredonde.

Podemos resumir en que Visual DaVinci es un lenguaje visual que posee un entorno de desarrollo integrado que permite tanto la codificación de forma textual como visual y con la particularidad de que la ejecución de un programa se puede visualizar a través de una ciudad, la cual será recorrida por un robot. Este puede realizar diferentes acciones, interactuando tanto con la ciudad como con objetos distribuidos en ella.

2.2 - Características del lenguaje

Una de las características principales del lenguaje Visual DaVinci es la capacidad de representar de forma visual la ejecución de un programa y poder establecer las órdenes en el lenguaje castellano

Para poder visualizar la ejecución hace uso de una ciudad compuesta de calles y avenidas, un robot que circulará a través de ellas y objetos como flores, papeles y obstáculos con los cuales podremos interactuar.

Previa a la ejecución del programa se puede configurar la ciudad para que aparezcan flores, papeles u obstáculos.

Durante la ejecución, el robot (Lubo-I), responde a ciertas primitivas y procesos. Además se puede acceder al valor de ciertas variables que el sistema ofrece (TABLA 1- VISUAL DAVINCI - PRIMITIVAS, variables y procesos).

La especificación del lenguaje está compuesta por palabras claves, primitivas, sentencias simples o compuestas, estructuras de control, expresiones y constructores de subprogramas. Todo programa que se ajuste a esta especificación estará conformado por un área de encabezamiento, una de declaraciones y finalmente un área de cuerpo.

Primitivas	Iniciar, mover, derecha, tomarFlor, tomarPapel, depositarFlor, depositarPapel
Variables	HayFlorEnLaEsquina, HayPapelEnLaEsquina,HayObstaculo HayFlorEnLaBolsa, HayPapelEnLaBolsa, PosAv, PosCa
Procesos	Pos(Avenida, Calle), Informar(expresión)

Tabla 1- Visual DaVinci - Primitivas, variables y procesos

En el área de encabezamiento se define el nombre del programa. En la de declaraciones se definen los procesos y variables que tendrán visibilidad solo dentro del programa principal. Finalmente, en el área de cuerpo se definen las sentencias del programa principal.

Se pueden definir subprogramas por medio de la especificación de procesos, los cuales pueden utilizar parámetros para comunicarse. Estos procesos también se encuentran conformados por el área de encabezamiento, que sería el nombre del proceso con los parámetros formales que contenga, el área de declaraciones, que permite la declaración de variables locales, y el cuerpo del proceso.

Como mencionábamos anteriormente, los parámetros, que se utilizan para la comunicación con los procesos, los podemos definir de tres modos diferentes: entrada (simbolizado con la letra E), salida (S) y entrada-salida (ES).

Las variables y parámetros pueden ser únicamente de dos tipos de dato: enteros y lógicos. Los datos de tipo entero se especifican mediante la palabra reservada numero y los de tipo lógico mediante la palabra boolean.

Como estructuras de control, la especificación define para su utilización a la selección y repetición, tanto condicional como incondicional. La selección estará

representada por las palabras claves si y sino (esta última opcional), la repetición incondicional por la palabra repetir y la repetición condicional por la palabra mientras.

El cuerpo del programa principal y el de los procesos están demarcados por las palabras comenzar y fin.

Cabe mencionar que el lenguaje posee algunas reglas sintácticas estrictas adicionales a la especificación: la sensibilidad al tipo de caracteres que utiliza es al estilo de C (literalmente sensitivo a las mayúsculas y minúsculas) y el sangrado parecido al estilo de Haskell, Occam o Python, se utiliza para delimitar la estructura del programa (pj. desplazando dos espacios las sentencias a la derecha delimitamos un bloque o pertenencia de determinadas sentencias a una estructura de control).

A continuación veremos por medio de un sencillo ejemplo como se utiliza lo anteriormente descripto.

```
programa programaEjemplo
procesos
 proceso hacer(E entrada:numero;
        S salida:numero;
        ES entradaSalida:boolean)
 variables
  auxiliar:numero
 comenzar
  auxiliar := 0
  repetir entrada
   mientras(HayFlorEnLaEsquina)
    tomarFlor
    auxiliar := auxiliar + 1
    mover
  salida := auxiliar
  entradaSalida := ( auxiliar = entrada)
 fin

variables
 flores:numero
 cuadras:numero
 igualCantidad:boolean
comenzar
 iniciar
 Pos(15,15)
 cuadras := 10
 hacer(cuadras,flores,igualCantidad)
 si(igualCantidad)
  Informar(flores)
 sino
```

```
  Informar(f)
Fin
```

Las palabras marcadas en negrita son aquellas que el lenguaje tiene como reservadas y las que se visualizan con subrayado son variables del sistema, primitivas o procesos del sistema.

El robot posee dos bolsas, una de papeles y una de flores. Cuando el robot toma una flor o un papel de una esquina lo almacena en la bolsa adecuada, incrementando la cantidad que posee. Lo mismo sucede cuando se deposita una flor o un papel, pero esta vez disminuyendo en uno el contenido de la bolsa correspondiente.

Por último, es importante destacar que el manejo de errores que posee no es lo suficientemente específico para entender de manera rápida las causas del mismo. Los errores sintácticos y léxicos se informan por medio de mensajes al momento de compilar. Los errores en tiempo de ejecución se suelen dar en las siguientes condiciones, (nótese que cuando nos referimos a flores también se puede dar el caso con los papeles):

- Cuando se intenta tomar una flor en una esquina que no existe.
- Cuando se intenta depositar una flor y la bolsa se encuentra vacía.
- Cuando se avanza y se pasa por un obstáculo.
- Cuando se avanza más allá de los límites de la ciudad.

Además de lo descripto anteriormente hay que mencionar que Visual DaVinci cuenta con un ambiente integrado que permite la edición, depuración y visualización de la ejecución de los programas. Si bien esto es una parte importante del producto, el actual trabajo no pretende profundizar estas características dado que no es el objetivo del mismo.

Podemos concluir que Visual DaVinci nos brinda un lenguaje sencillo para la enseñanza e incorporación de los conceptos básicos de la programación estructurada y modular.

2.3 – Extensión propuesta al lenguaje

Si bien el lenguaje presenta aspectos importantes al momento de enseñar programación estructurada, estos se pueden enriquecer aún más con algunas modificaciones y agregados que sirvan también para brindar los cimientos bases en la programación concurrente.

Las extensiones y modificaciones propuestas son las siguientes:

- Que la implementación del intérprete sea multiplataforma: La actual implementación de Visual DaVinci solo funciona sobre el sistema operativo Windows, lo cual genera una dependencia no siempre deseable y una traba para su uso. En la actualidad, la mayoría de los alumnos utilizan productos libres dentro de los cuales encontramos los sistemas operativos. Al hacerlo independiente de la plataforma eliminaríamos estas restricciones y contribuiríamos en el fomento de su uso.
- Enriquecer los mensajes de error que se producen en tiempo de compilación, y en tiempo de ejecución: Si bien es cierto que el lenguaje nos aporta mensajes cuando una compilación no se pudo llevar a cabo o cuando una ejecución se vio abortada por algún motivo, también es cierto que los mensajes a veces no son precisos y el estudiante medio necesita de la ayuda de alguien con conocimientos más avanzados (generalmente el docente) para comprender lo que sucedió. Esto dificulta la práctica fuera de la institución. Mejorando los mensajes que se producen se intenta reducir la dependencia del docente. De esta forma el alumno podrá por si solo reconocer y corregir el error además de promover el uso de la herramienta (práctica) fuera del horario que se le brinda en la institución.
- Incorporación del tipo de datos para manejo de texto: El lenguaje, como mencionábamos con anterioridad, solo dispone de dos tipos de datos: el tipo de dato entero y el tipo lógico. Aportando un tercer tipo de dato, el de texto, podríamos hacerle al estudiante una introducción al manejo de cadenas y enriquecer los ejercicios, además de motivar al estudiante a través de algoritmos más reales. También, como veremos más adelante, se puede utilizar para realizar ejercicios de algoritmos concurrentes.
- Incorporar el operador módulo
- Permitir la lectura de variables: Al permitir la lectura de variables podemos aumentar de manera considerable los distintos escenarios y/o problemas

que se pueden plantear para su resolución, además de incorporar el concepto de interacción con el usuario.

- Incorporación de funciones primitivas: Incorporando al lenguaje funciones primitivas se extiende de manera considerable la resolución de problemas que se pueden llevar a cabo mediante el mismo. Estas funciones en principio serán las siguientes:
  - retornar un número de manera aleatoria.
  - retornar la cantidad de caracteres de una cadena de texto pasada como argumento.
  - retornar un número a partir de una cadena de texto pasada como argumento.
  - retornar una cadena de texto a partir de un número pasado como argumento.
  - retornar una cadena de texto a partir de un valor lógico pasado como argumento.
- Modificación del nombre de tipo de dato boolean por “logico”: El lenguaje se encuentra castellanizado, por lo que quedaría solamente traducir el símbolo empleado para definir variables o parámetros de tipo “boolean” por el símbolo “logico”.
- Cambiar los símbolos utilizados para identificar los procesos y el proceso: El uso de estos nombres genera cierta confusión en una etapa posterior al cursado de la materia, debido a que el estudiante incorpora de manera equivocada el concepto de que un módulo o subprograma es un proceso. Cambiando los símbolos utilizados para definir los procesos por "subprogramas" y al proceso por "procedimiento" quitaríamos esta confusión y la adquisición y posterior profundización de conceptos sería más natural.
- Quitar restricciones de sangrado: El criterio utilizado para imponer el sangrado estricto fue el de formar en el estudiante un estilo, y con estilo nos referimos a *mejorar en todo lo posible la legibilidad del código de forma tal que resulte sencillo entenderlo, modificarlo, adaptarlo y reusarlo, ayudando así a maximizar la productividad y minimizar el costo de desarrollo y mantenimiento* (Champredonde and De Guisti 1997). Si bien esto es válido, se considera preferible que este estilo se vaya adquiriendo paulatinamente

con la práctica de la programación (codificación) y no imponiéndolo desde los primeros momentos en que se tiene contacto con el lenguaje. Marcar bloques de código o agrupar sentencias que pertenecen a una estructura de control con los símbolos “comenzar” y “fin” (como lo hacen la mayoría de los lenguajes de programación) sería más beneficioso para el estudiante a la hora de concentrarse y determinar que sentencias se encuentran involucradas en determinadas estructuras. Otro punto en contra a la hora de evaluar el sangrado estricto es que cuando el alumno intenta dominar otro lenguaje que no posee esta restricción, vuelve al problema original de cómo separar adecuadamente las sentencias.

- Quitar restricciones del uso estricto de mayúsculas y minúsculas: Como mencionábamos en el párrafo anterior el criterio fue formar en el estudiante un estilo. Nuevamente se considera que este estilo se vaya adquiriendo con la práctica de la programación y no imponiéndolo desde los primeros momentos en que se tiene contacto con el lenguaje, permitiendo así al estudiante concentrarse en la resolución del problema.
- Flexibilizar y mejorar los mensajes que se pueden lanzar en tiempo de ejecución: el proceso primitivo "informar" tiene fuertes restricciones por lo que no se termina utilizando de la manera en que fue concebido. Al no imponer restricciones de tipo de expresión ni de cantidad se potencia el lenguaje, ampliando el tipo de problema a resolver y permitiendo obtener más claridad en los resultados dados para un problema en particular. Las salidas más documentadas ayuda a los alumnos a la resolución de sus problemas.
- Permitir la distribución de flores, papeles y obstáculos dentro de la ciudad, en lugares específicamente determinados por el alumno, y su almacenamiento para ejecuciones posteriores. Permitir la distribución de flores y papeles en la ciudad de una manera más controlada, ayuda al alumno a evaluar mejor sus algoritmos, además de enriquecer los problemas que se pueden resolver. El almacenamiento para ejecuciones posteriores es de utilidad dado que cada problema a resolver tiene necesidades intrínsecas sobre la distribución de flores, papeles y obstáculos y que estas se puedan guardar con el algoritmo resulta de gran ayuda para no tener que configurar la ciudad cada vez que se desee ejecutar. Conservar la configuración representa una ventaja adicional al facilitar ejecuciones posteriores con las

mismas condiciones iniciales, sobre todo cuando se producen errores y se debe modificar el programa y volver a ejecutar.

- Incorporar mecanismos necesarios para la concurrencia: Sería beneficioso para el alumno utilizar el lenguaje que ya ha empleado y comprendido, para la incorporación de los conocimientos básicos de la concurrencia. También que estos conocimientos se puedan visualizar a la hora de ejecutar programas concurrentes. Para llevar a cabo esto será necesario definir estructuras que permitan correr de forma concurrente ciertos bloques de código.
- Incorporar algún tipo de dato abstracto para el uso posterior en algoritmos concurrentes: Al introducir estructuras para manejar algoritmos concurrentes, será necesario definir algún tipo de dato abstracto, que permita el uso de un recurso de manera exclusiva cuando varios lo necesiten para poder sincronizar los distintos hilos de ejecución ante la petición de un determinado recurso. En primera instancia un tipo de dato que surge son los semáforos, tanto general como binario.
- Incorporar distintos tipos de planificadores de ejecución: Al introducir elementos para permitir la ejecución simultánea (aparente) de distintos hilos en un mismo programa, será necesario contar con algún planificador de ejecución. Para enriquecer y fortalecer los conocimientos del alumno se definirán los planificadores más utilizados. De esta manera se podrá comprender de manera práctica como estos influyen a la hora de la ejecución de los algoritmos.
- Manipular la secuencia lógica de ejecución: Ejecutar programas que utilizan concurrencia introduce el problema del no determinismo. Esto lleva a la necesidad de contar con algún mecanismo que nos permita, tanto la reproducción exacta de una ejecución concurrente que ha terminado, en forma correcta o incorrecta, como la posibilidad de forzar trazas de ejecución que lleven a situaciones de error y que, en función del no determinismo, pueden no producirse aun cuando se realicen un número elevado de ejecuciones del programa.

2.4 - Limitaciones encontradas

Al realizar la investigación y profundizar en la implementación de Visual DaVinci he encontrado ciertas limitaciones para poder realizar las modificaciones y agregados que he mencionado con anterioridad, al menos de manera directa, siendo las más importantes las siguientes:

- La implementación del lenguaje se encuentra realizada con Delphi. Este compilador es dependiente de la plataforma Windows, por lo cual no podríamos convertirlo en multiplataforma de manera directa ya que deberíamos migrar el código a otro lenguaje que tenga la particularidad de compilarlo o ser capaz de interpretarlo en diferentes plataformas. Esto es realmente una gran desventaja, ya que el esfuerzo de migrar es considerablemente alto.
- El lenguaje fue implementado teniendo como premisa que la secuencia lógica de ejecución es única. Esto representa otra desventaja al momento de pensar en las modificaciones necesarias a la hora de incorporar las estructuras necesarias para dar soporte a la concurrencia, ya que intentar la modificación implicaría, además de agregar, re-diseñar el intérprete para dar soporte a las nuevas características.

2.5 – ¿Por qué construir un nuevo intérprete?

Los lenguajes de programación, ya sea que posean una implementación por medio de un compilador o un intérprete, deben intentar poseer los siguientes atributos:

- claridad, sencillez y unidad
- ortogonalidad
- naturalidad para la aplicación
- apoyo para la abstracción
- facilidad para verificar programas
- entorno de programación
- portabilidad

Es cierto que existe una extensa cantidad de lenguajes para poder utilizar y ayudar al alumno en la incorporación de conocimiento inicial, pero también es cierto que no contamos con uno que cumpla los requerimientos que perseguimos en el presente trabajo:

- simbolización en español: la mayoría de los lenguajes de programación que hoy tenemos disponibles se encuentran en inglés y esto es un obstáculo más, debido a que el alumno medio no domina otro lenguaje que no sea el español.
- simbología acotada: esto es importante para delimitar los problemas posibles de resolver e ir incorporando de manera gradual algoritmos específicos.
- visualización de la ejecución: uno de los elementos más importantes para el alumno inicial es poder comprender el resultado de un algoritmo. Permitir su visualización disminuye la necesidad de abstracción para verificar resultados y llegar a la solución.
- mecanismos sencillos de concurrencia: permitir una introducción al campo de los lenguajes concurrentes de manera sencilla y práctica.
- permitir la repetición de la secuencia de ejecución de manera exacta en programas que utilicen mecanismos concurrentes.

- permitir la manipulación de la secuencia lógica de ejecución de un programa concurrente.

Ante las limitaciones expuestas y descriptas en la sección anterior es adecuado pensar en la construcción de un intérprete que incorpore las reglas gramaticales existentes de Visual DaVinci y agregue las nuevas características.

Todo esto se llevará a cabo sin dejar de lado los atributos que un intérprete debe tener, aunque no se realizará el entorno de programación debido a que no se encuentra dentro del alcance del trabajo, ya que el mismo consistirá, además de la construcción del nuevo intérprete, en la creación de las interfaces necesarias para la posterior integración con algún entorno de desarrollo y la creación de un editor para demostrar de manera práctica la ejecución de un programa como así también la sintaxis que se utilizó.

3 - DaVinci Concurrente

3.1 - Introducción

DaVinci Concurrente es un lenguaje orientado a la enseñanza de la programación estructurada para fortalecer los cimientos base de su aprendizaje, tanto en la etapa inicial de la programación secuencial como en la introducción a los conceptos básicos de la programación concurrente.

A modo de introducción podemos citar las principales características del intérprete producto de la implementación del lenguaje:

- La ejecución de un programa podrá ser visualizada a través de una ciudad, robots y otros objetos. Estos objetos son flores, papeles y obstáculos. El robot podrá interactuar con éstos depositándolos, tomándolos o esquivándolos dependiendo cual sea el caso.
- El lenguaje soporta la resolución de problemas a través de algoritmos concurrentes mediante el uso de estructuras, tipos de datos y primitivas definidas a tal fin.
- Se puede seleccionar un planificador de ejecución sobre un conjunto de planificadores previamente definidos e implementados.
- La cantidad y ubicación inicial de flores y papeles en la ciudad se pueden definir previa a la ejecución del programa, ya sea de manera aleatoria, manual o una combinación de ambas.
- Los mensajes de error. tanto sintáctico como semántico, son fáciles de interpretar.
- La cantidad de robots ejecutándose, en una “aparente” simultaneidad, estará definida por las características intrínsecas del problema a resolver.
- Un programa puede contener cero o más robots.

3.2 - Programa ejemplo

Una manera de iniciarnos con la especificación del lenguaje es a través de un ejemplo similar del famoso "Hola Mundo", en donde podemos apreciar de una manera sencilla las principales características de la sintaxis.

Es importante resaltar que una de las características del lenguaje es que se puede utilizar **un conjunto de robots y este conjunto puede encontrarse vacío**, esto es útil cuando se requiere resolver algoritmos que no necesiten del robot.

El siguiente ejemplo es justamente uno que no utiliza el robot. En secciones posteriores se verá cómo se introducen los robots a la ciudad.

```
programa ejemplo
variables
    nombre:texto
comenzar

    //pedimos el nombre por pantalla
    Pedir(nombre)

    //mostramos por pantalla el mensaje
    Informar("Hola ",nombre, " Bienvenido al lenguaje.")
fin
```

El ejemplo anterior muestra la frase de bienvenida dependiendo del nombre que se ha introducido.

El pedido de datos para cargar el valor a la variable "nombre", como así también la frase que se visualizará, quedará en función de la ciudad que estemos utilizando. Este concepto se ampliará en próximas secciones.

A medida que avancemos en la especificación del lenguaje veremos cómo determinadas acciones quedarán en función de la implementación que se realice de la ciudad.

3.3 – Características del lenguaje

DaVinci Concurrente fue desarrollado con la premisa de brindarle al alumno una herramienta más a la hora de incorporar los conocimientos de base en la resolución de problemas a través de una computadora.

Podemos citar las características más significativas que nos aporta el lenguaje:

- La ejecución de un programa, que se ajuste a la especificación, podrá ser **visualizado a través de una ciudad** y objetos que interactúan en ella.
- La ciudad podrá contener cero o más **robots**, siendo el límite de estos la necesidad a cubrir en cada problema.
- Cada robot podrá ejecutar un conjunto de instrucciones, las que podrán ser diferentes o iguales.
- El lenguaje contiene estructuras, tipos de datos y primitivas para **resolver problemas de manera concurrente**.
- El intérprete podrá ser configurado con diferentes opciones, por ejemplo se podrá configurar la cantidad de calles y avenidas, se podrá **seleccionar el planificador** de ejecución.
- La cantidad inicial de **flores y papeles** en la ciudad se pueden definir previo a la ejecución del programa. Sus ubicaciones **podrán ser definidas aleatoriamente o ser ubicadas manualmente**.
- La cantidad de flores y de papeles que el robot puede llevar en sus bolsas puede ser inicializada antes de la ejecución del programa.
- **Mensajes de error sintácticos bien definidos** cuando se analiza el código para su ejecución, aportando clara evidencia de porque se produjo.
- Los mensajes de error durante la ejecución se encuentran bien definidos para que el alumno pueda intentar resolverlos con sus conocimientos.
- Sintaxis **sin restricciones de sangrado ni sensibilidad** a mayúsculas y minúsculas.
- Tipo de datos para manejo de **cadenas y funciones primitivas** para su manipulación.

- Posibilidad de **pedir datos al usuario** para luego ser almacenados en variables.

3.3.1 – Concurrencia

Como mencionamos anteriormente, la especificación del nuevo lenguaje aporta estructuras y mecanismos para resolver problemas de manera concurrente. El lenguaje soporta la definición de hilos, que son básicamente bloques de código que pueden ejecutarse de manera concurrente.

Un hilo podrá ser puesto en la cola de hilos listos para ejecutarse con la palabra reservada **arrancar**. Los hilos podrán ser arrancados la cantidad de veces que se necesite, teniendo siempre en cuenta la limitación que si en su interior se encuentra la inicialización de un robot, la palabra clave iniciar, este hilo podrá ser arrancado solamente una única vez.

La ejecución del programa terminará una vez que todos sus hilos hayan finalizado. Al programa principal lo podemos considerar como un caso especial de hilo.

Para la sincronización y exclusión mutua de los diferentes hilos que componen un programa, se proveen estructuras llamadas semáforos (binarios y generales).

Al igual que la definición brindada por Dijkstra, sobre la manera en que los semáforos pueden ser manipulados, el lenguaje permite únicamente las primitivas **iniciar, esperar y avisar** para su manipulación.

El lenguaje provee una sección de definición global para definir los semáforos. Estos serán visibles y utilizables en cualquier parte del programa, incluidos los bloques correspondientes a los hilos.

Como veremos en el siguiente apartado el lenguaje provee mecanismos sencillos para resolver los clásicos problemas que introduce la concurrencia.

3.3.1.1 – Problemas clásicos resueltos con DaVinci Concurrente:

Para incorporar conceptos claves en la resolución de problemas de manera concurrente, como son la sincronización y protección de recursos compartidos, se suele acudir a interpretar, entender y resolver los problemas clásicos del mundo de la concurrencia.

A continuación se presentan soluciones (utilizando el lenguaje) a los problemas clásicos de la concurrencia, luego de haber adaptados los mismos a las posibilidades de resolución que ofrece DaVinci Concurrente.

Productor/Consumidor con buffer circular (o buffer acotado)

El problema Productor/Consumidor consiste en el acceso concurrente por parte de procesos productores y procesos consumidores sobre un recurso común que resulta ser un buffer de elementos. Los productores tratan de introducir elementos en el buffer de uno en uno, y los consumidores tratan de extraer elementos de uno en uno. Para asegurar la consistencia de la información almacenada en el buffer, el acceso de los productores y consumidores debe hacerse de manera sincronizada y exclusiva. Adicionalmente, el buffer es de capacidad limitada, de modo que el acceso por parte de un productor para introducir un elemento en el buffer lleno debe provocar la detención del proceso productor. Lo mismo sucede para un consumidor que intente extraer un elemento del buffer vacío.

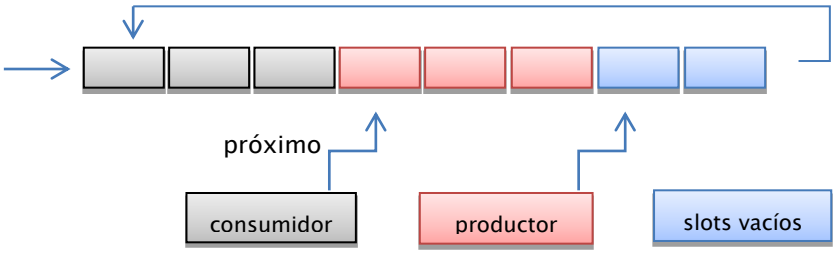


Figura 1 – Esquema Productor/Consumidor

Una posible analogía al problema del productor/consumidor con buffer acotado podría ser:

- El buffer será una calle de la ciudad (slots de consumidores, Fig. 2).
- Los elementos serán flores.
- En el buffer se podrán depositar los elementos que se produzcan y luego podrán ser consumidos. El elemento inicial será depositado en la primera avenida de la calle, los elementos posteriores se depositarán en las avenidas siguientes. Al llegar al final de buffer se procederá nuevamente con la primera avenida, solo se podrá depositar si la esquina se encuentra vacía.

- El productor para producir elementos deberá recorrer un área de la ciudad en busca de elementos (cuadrante de recursos para productores, Fig. 2). Al encontrar uno la tomará y luego lo depositará en el buffer.
- El consumidor irá tomando Los elementos del buffer y luego las depositará en el cuadrante del productor.

Para garantizar que el algoritmo no se detenga por escasez de recursos y pueda ser explotado como tal será necesario contar con las siguientes condiciones:

- Se dispondrá de al menos una flor.
- Inicialmente la/las flores deberán ser ubicadas en el interior del cuadrante del productor.
- Cuando un consumidor tome una flor de la calle, la depositará de manera aleatoria en el cuadrante del productor.
- Si el buffer se encuentra lleno y el productor intentase depositar una flor, deberá esperar.
- Si el buffer se encuentra vacío y el consumidor intentase tomar algún elemento, deberá esperar.

En la sección Anexos A se deja el ejemplo (productor-consumidor) de una implementación que resuelve el problema anterior, utilizando un consumidor y un productor. A continuación se muestra una instantánea del programa en ejecución.

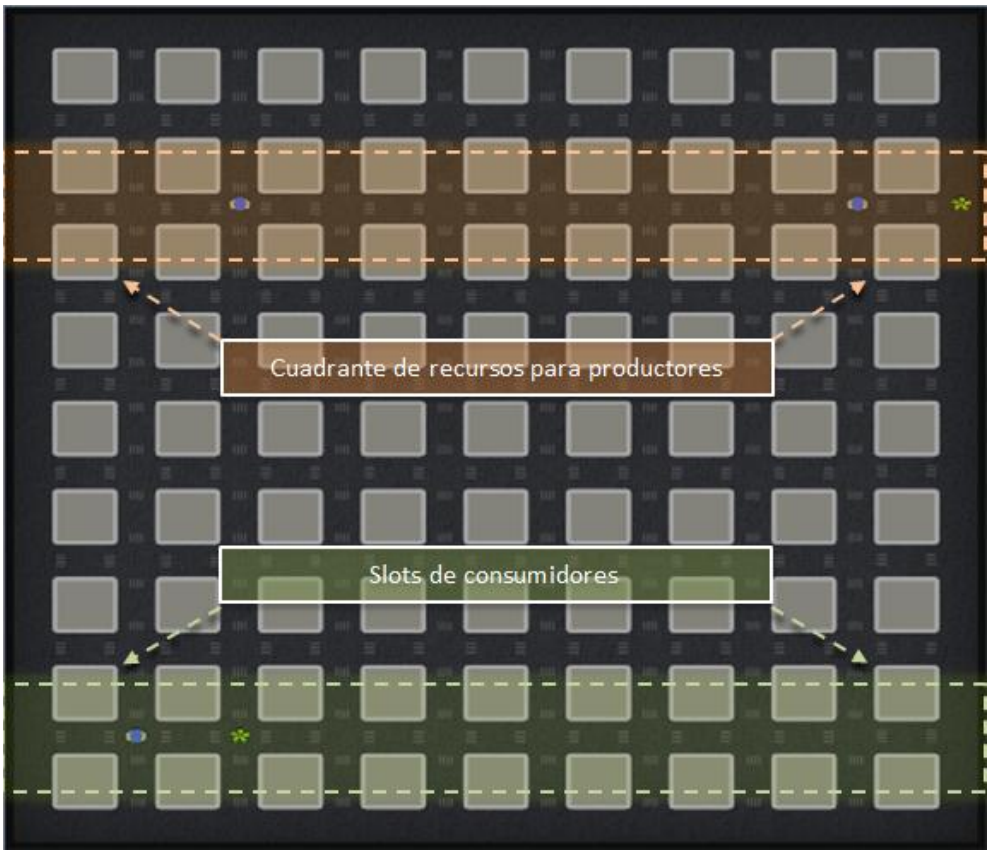


Figura 2 – Instantánea Productor / Consumidor

Filósofos

Otro problema clásico, pensando en sincronización, es el de los filósofos, planteado por Dijkstra. Sintéticamente podemos decir que consiste en cinco filósofos que se sientan alrededor de una mesa redonda y pasan su vida alternando entre cenar y pensar. Cada filósofo tiene un plato de fideos enfrente de él y dos tenedores, uno a cada lado (un tenedor a la izquierda de cada plato, esto se puede observar en la Fig. 3). Para comer los fideos son necesarios dos tenedores y cada filósofo deberá tomar el tenedor de su izquierda y el de su derecha. Si cualquier filósofo levanta un tenedor y el otro está ocupado, se quedará esperando, con el tenedor en la mano, hasta que pueda levantar el otro tenedor, para luego empezar a comer.

Si dos filósofos adyacentes intentan tomar el mismo tenedor a una vez, se produce una condición de carrera: ambos compiten por tomar el mismo tenedor, y uno de ellos se queda sin comer.

Si todos los filósofos levantan el tenedor que está a su derecha al mismo tiempo, entonces todos se quedarán esperando eternamente, porque alguien debe liberar el tenedor que les falta. Nadie lo hará, porque todos se encuentran en la misma situación (esperando que alguno deje sus tenedores). Entonces los filósofos se morirán de hambre. Este bloqueo mutuo se denomina interbloqueo o deadlock.

El problema consiste en encontrar un algoritmo que permita que los filósofos nunca se mueran de hambre. (Dijkstra 1965)

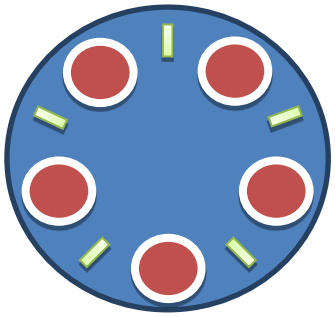


Figura 3 –Esquema Filósofos

Una posible analogía para su resolución a través del lenguaje sería utilizar flores en lugar de tenedores y robots en lugar de los filósofos, a continuación se describe con mayor detalle el algoritmo y sus restricciones:

- Se dispondrán cinco robots, en una calle de la ciudad, posicionados en las avenidas pares (Fig. 4).
- Se posicionarán cinco flores, cada una se ubicará en la calle siguiente a la que se encuentra el robot y en las avenidas impares.
- Cuando a un robot le toque el turno de comer deberá tomar, de la calle siguiente, la flor que se encuentra a su izquierda y a su derecha.
- Cuando le toque el turno de tomar las flores al robot, y no exista de su lado derecho por las restricciones propias de la ciudad (limite), tomará la flor del extremo izquierdo.
- Luego de comer deberá depositar nuevamente las flores en el lugar de donde las tomo.

En la sección Anexos A se deja el ejemplo (filósofos) de una implementación que resuelve el problema anterior. A continuación se muestra una instantánea del programa en ejecución.

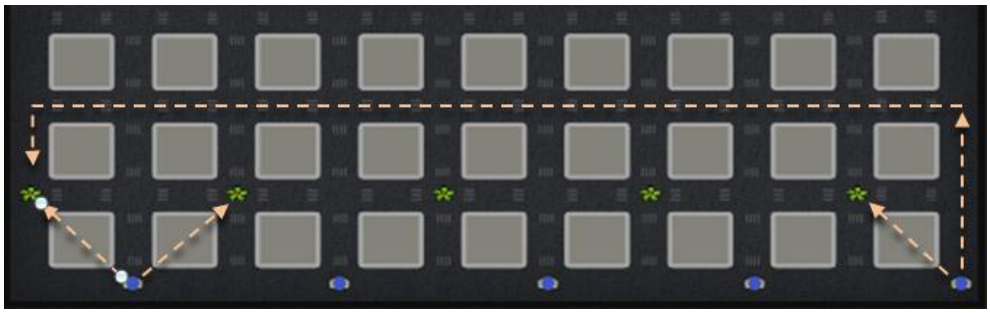


Figura 4 – Instantánea Filósofos

3.4 – Especificación del lenguaje

3.4.1 – Ámbito de aplicación

DaVinci Concurrente está orientado a la resolución de problemas que sirvan para enseñar de manera análoga los que suelen presentarse en la programación estructurada secuencial y concurrente en su etapa inicial.

3.4.2 – Gramática

En las siguientes secciones se describirán todos las aspectos que tienen que ver con la gramática, libre de contexto, utilizada para definir la estructura léxica y sintáctica de los programas.

Una gramática libre de contexto *es una especificación para la estructura sintáctica de un lenguaje de programación, es muy parecida a la especificación de la estructura léxica de un lenguaje (tokens) utilizando expresiones regulares, excepto que la gramática* ***involucra reglas de recursividad****.* (Louden 2005)

3.4.3 – Estructura Léxica

3.4.3.1 – Codificación:

Como punto de partida en la especificación léxica es conveniente mencionar que los programas se escriben en ASCII.

3.4.3.2 – Sensibilidad a mayúsculas:

El lenguaje no es sensible a mayúsculas y minúsculas, por lo tanto para el lenguaje el símbolo "COMENZAR" es tratado exactamente igual que el símbolo "comenzar".

3.4.3.3 – Terminadores de línea y espacios:

Los terminadores de líneas utilizados por el lenguaje son nueva línea, retorno de carro, avance de página o una combinación de las anteriores y de espacio. Se reconocen también a los tabuladores. Si bien se reconocen los terminadores de líneas, estos no hacen falta para separar sentencias o instrucciones pudiendo escribir todo el programa en una sola línea. Un programa se interpretará hasta encontrar el fin de archivo.

3.4.3.4 – Comentarios:

Los comentarios sobre el código fuente se pueden realizar de dos maneras, si lo que se quiere comentar es una única línea se pueden utilizar los caracteres "//" lo cual inicia el comentario hasta encontrar una nueva línea. Si se quiere comentar varias líneas se pueden usar el símbolos "{" para iniciar el comentario y "}" para finalizarlo, todo lo que quede dentro de estos símbolos se tomará como un comentario aun cuando se encuentren avances de líneas, retornos de carro y demás. Es importante destacar que todo lo que esté dentro de un comentario el intérprete lo ignorará.

3.4.3.5 – Identificadores:

Se podrán definir como un conjunto de letras, números y símbolos especiales. Todo identificador deberá comenzar con una letra o un símbolo especial y puede continuar con una repetición de letras, números o símbolos especiales. Las letras permitidas son las del alfabeto español y los dígitos los comprendidos del "0" al "9". Como símbolo especial se puede usar el guión bajo "_". Los identificadores se usan para definir variables, parámetros, procedimientos e hilos. El nombre del programa no cuenta como un identificador válido pero se encuentra definido bajo las mismas reglas. Los identificadores no pueden ser palabras reservadas.

3.4.3.6 – Palabras reservadas:

La lista de palabras que se detallan a continuación son las palabras que utiliza el lenguaje por lo que su uso como un identificador no es posible. De hacerlo el intérprete retornará un error.

Palabras Reservadas del Lenguaje		
aleatorio	hilo	programa
arrancar	hilos	repetir
avisar	hayRobot	procedimiento
comenzar	informar	sa

depositarFlor	iniciar	semaforoBinario
depositarPapel	iniciarSemaforo	semaforoGeneral
derecha	logico	Si
en	logicoATexto	Sino
es	longitud	Subprogramas
esperar	mientras	Sustraer
f	mover	Texto
fin	numero	textoANumero
hayFlorEnLaBolsa	numeroATexto	tomarFlor
hayFlorEnLaEsquina	pedir	tomarPapel
hayObstaculo	pos	v
hayPapelEnLaBolsa	posAv	variables
hayPapelEnLaEsquina	posCa	

Tabla 2 – DaVinci Concurrente – Palabras reservadas

3.4.3.7 – Literales:

En la especificación se definen literales enteros, de texto y lógicos.

Los literales enteros se encuentran representados por una secuencia no vacia de dígitos decimales (0 al 9).

Los literales de texto están compuestos por un carácter de comienzo simbolizado con " (doble comillas) y el mismo carácter de finalización. En el interior se puede encontrar cualquier conjunto de símbolos exceptuando a la doble comilla.

Los literales lógicos se encuentran definidos por las letras v y f siendo los valores verdadero y falso respectivamente.

3.4.3.8 – Separadores:

La separación de sentencia se realiza internamente de acuerdo a las reglas gramaticales del lenguaje. Se utilizan los terminadores de líneas y espacios definidos anteriormente para el reconocimiento de las instrucciones.

3.4.4 – Tipos de datos, valores y operaciones

El lenguaje soporta cinco tipos de datos (ver Tabla 3 – DaVinci Concurrente – Tipos de datos): entero, lógico, texto, semáforo binario y semáforo general. A continuación se detallarán las operaciones y el conjunto de valores posibles de cada tipo.

3.4.4.1 – Tipo de dato entero:

El rango de valores posibles es de -2^{31} a $2^{31}-1$ (32 bits con signo). Los operadores aritméticos son la suma (simbolizado con +), resta (–), multiplicación (*), módulo (%) y división (/). Cabe destacar que este último realiza la división entera, entregando como resultado otro entero. Cuando el resultado de la división no es entero se toma el entero más cercano y en el caso de que este se encuentre a la misma distancia se retorna la parte entera.

El operador módulo (%) realiza la división entera y retorna el resto de la operación.

Los operadores de relación o comparación son mayor (>), mayor o igual (>=), menor (<), menor o igual (<=), igual (=) y distinto (<>)

3.4.4.2 – Tipo de dato texto:

Se puede representar cualquier secuencia de caracteres que no supere la longitud de 2^{31}. El tipo de datos texto tiene sobrecargado el operador suma (+) que realiza la concatenación de los dos operandos. Se pueden hacer comparaciones al igual de los números enteros teniendo presente que estas se hacen independientemente si están en minúsculas o mayúsculas, y se utiliza el orden alfabético.

3.4.4.3 – Tipo de dato lógico:

Los valores permitidos serán verdadero y falso representados con los símbolos "v" y "f" respectivamente, las operaciones que se pueden utilizar con estos tipos de datos son la conjunción (&), disyunción (|) y negación (!). Las comparaciones permitidas son la igualdad (=) y la diferencia (<>) aunque estas últimas no son deseables de utilizar.

3.4.4.4 – Tipo de dato semáforo:

Es un tipo de dato abstracto para trabajar con problemas resolubles mediante algoritmos concurrentes.

Este tipo de datos semáforo pude ser definido de dos maneras, de tipo binario o de tipo general.

- semáforo binario: Los semáforos de este tipo pueden tomar solamente los valores 0 (cero) o 1 (uno).
- semáforos generales: Los semáforos de este tipo pueden tomar cualquier valor entero positivo incluido el cero.

Las únicas operaciones con las cuales se los pude manipular son: "iniciarSemaforo", "avisar" y "esperar".

En la sección correspondiente a los hilos se amplía la descripción y funcionamiento de estas primitivas.

Tipo	Valores	Aritméticos	Comparación
entero	32 bits con signo	+ - * / %	> >= < <= = <>
texto	Secuencia de caracteres	+(concatena)	> >= < <= = <>
logico	V / F	& | !	= <>
Tipo	**Valores**	**Operaciones**	
semaforoBinario	0 / 1	iniciarSemaforo, avisar, esperar	
semaforoGeneral	$0..2^{31}-1$	iniciarSemaforo, avisar, esperar	

Tabla 3 – DaVinci Concurrente – Tipos de datos

3.4.5 - Variables

El lenguaje provee la utilización de variables, que son sintéticamente espacios de memoria utilizados para albergar datos. Estas pueden ser manipuladas a través de las operaciones definidas para cada tipo de dato.

Las variables deberán estar definidas de un tipo de dato específico y asociadas a un identificador.

Existen sólo tres maneras de poder modificar el contenido de una variable. La primera por medio de una asignación, la segunda a través del uso de parámetros de entrada y salida de un subprograma, por último, mediante la utilización del procedimiento primitivo "pedir".

La inicialización del valor de las variables se deja como responsabilidad al programador, aunque internamente esta se realice de la siguiente manera:

- tipo numero: inicializa el valor de la variable con cero (0).
- tipo texto: inicializa el valor de la variable con la cadena de texto vacía.
- tipo logico: inicializa el valor de la variable falso.
- tipo semaforoBinario: inicializa el valor de la variable con uno (1).
- tipo semaforoGeneral: inicializa el valor de la variable con uno (1).

Según el ámbito donde se encuentren definidas las variables podrán ser accedidas de manera local al programa principal, local al subprograma, local al hilo o accedidas de manera global a todo el programa.

Se muestra a continuación mediante un ejemplo las distintas áreas donde pueden estar definidas y el ámbito donde pueden ser utilizadas.

```
programa ejemplo
variables
      global:semaforoBinario

subprogramas
      procedimiento modulo(es pEntSal:numero)
      variables
            local:logico
      comenzar
            local := v
      fin

hilos
      hilo carlitos(en pEnt:numero)
      variables
            local:texto
      comenzar
            local := "algo"
      fin

variables
      local:numero
comenzar
  local := 10
  iniciarSemaforo(global,local)
fin
```

Si bien el uso de variables globales se encuentra permitido por el lenguaje, será responsabilidad del docente instruir al alumno sobre su uso y dejar este tipo de variables para problemas que involucren a la concurrencia, por lo que se verán, por lo general, como variables de tipo semáforo o como abstracciones de recursos compartidos.

Cabe destacar que el lenguaje es fuertemente tipado, lo cual implica que las comprobaciones se realizan al momento previo de su ejecución. Esto es así porque las variables y las expresiones tienen un tipo conocido al momento de la compilación.

3.4.6 – Operador de asignación

El operador de asignación que se utiliza es al estilo Pascal, siendo los símbolos para representarlo el dos puntos seguido de igual (:=)

La asignación involucra tres secciones: la primera será el identificador de la variable que se quiera modificar, la segunda constará de los símbolos " := " y la tercer sección será una expresión que retorne un resultado acorde al tipo de datos definido al identificador.

ej: miVariableNumerica := (2 + variableNumerica)

3.4.7 – Manejo de errores

Los errores que produce el lenguaje los podemos agrupar en las siguientes categorías:

3.4.7.1 – errores en tiempo de compilación (léxicos y sintácticos):

Son errores que se producen porque el código fuente no se corresponde con la gramática asociada al lenguaje. Por ejemplo que aparezcan tokens que no se reconocen, identificadores no declarados, llamada a subprogramas con distinto número o clase de parámetros o que se haya utilizado diferentes tipos de datos en una expresión, etc.

Estos errores son los más fáciles de identificar y resolver. El intérprete informará estos errores de una manera comprensiva e intentará señalar exactamente donde se produjo, identificando el número de línea.

3.4.7.2 – errores en tiempo de ejecución (semánticos o de lógica):

Encontraremos dentro de esta categoría errores que se producen al aplicar de manera equivocada ciertas reglas o conceptos. Algunos de los problemas más comunes son: levantar una flor donde no se encuentra ninguna, que se intente depositar un papel y la bolsa se encuentre vacía, también pueden darse los casos de que algún de robot supere los límites de la ciudad, que se intente atravesar un obstáculo, que se realice una división por cero, etc. Estos errores producen que la ejecución se aborte informando claramente la razón del mismo.

Existen otros errores que se producen cuando el algoritmo utiliza mecanismos de concurrencia, son errores más difíciles de detectar y solo se ha puesto énfasis en la detección de hilos suspendidos informando al usuario que no es posible continuar con la ejecución debido a que todos los hilos se encuentran en estado bloqueado.

3.4.8 – Expresiones

Toda expresión que se ajuste a la especificación del lenguaje deberá retornar un valor asociado a un tipo de dato válido, pudiendo mezclar valores y variables siempre y cuando se realicen las operaciones de manera correcta.

Los valores literales, las variables, las funciones primitivas y los parámetros también podrán ser reconocidos como una expresión o parte de una.

Existen tres tipos de expresiones: aquellas que retornan un valor lógico, las que retornan un valor numérico y las que retornan un valor de tipo textual. Podrán encontrarse en cualquier lugar del programa que se requiera un valor.

Los siguientes ejemplos muestran diferentes tipos de expresiones válidas en asignaciones, procedimientos e hilos:

- variable := **expresion**
- varLogico := **! (HayFlorEnLaBolsa & (cantCuadras <= 10))**
- varNumerico := **2 * (longitud("cadena") + 5)**
- varTextual := **sustraer("hola luna",1,5) + "mundo"**
- cuadrado(**avenida + 1, calle**)
- informar(**"El procedimiento terminó en la av "+avenidaTexto**)
- informar(**"El procedimiento arranco en la calle ", calleNumero**)
- informar(**"El procedimiento arranco en la calle ", calleTexto**)
- arrancar juan(**avenida,1**)

3.4.9 - Subprogramas

El lenguaje soporta la utilización de procedimientos en un área especial identificada por la palabra reservada "subprogramas". Para poder definirlos se utilizará la palabra reservada "procedimiento" seguida de un identificador, que es el que se utilizará en su invocación, y luego los parámetros que se necesiten.

Una vez definidos pueden utilizarse la cantidad de veces que se requieran. Solo se debe tener en cuenta que la búsqueda de la definición será siempre hacia arriba (debe haber sido definida antes de utilizarla), como lo realizan otros lenguajes (ej. Pascal).

3.4.9.1 - Parámetros

El programa principal, los procedimientos e hilos utilizan los parámetros para comunicarse entre sí.

No existe límite en la cantidad de parámetros que se puedan definir, siendo tarea del docente instruir al alumno en su utilización.

A continuación veremos un ejemplo de la definición de un procedimiento con parámetros.

```
programa ejemplo

subprogramas

     procedimiento rectangulo(en alto:numero; en largo:numero)
     comenzar
      repetir 2
           comenzar
           repetir alto
            mover
           derecha
           repetir largo
            mover
           derecha
           fin
     fin

comenzar
 iniciar
 rectangulo(3,5)
fin
```

Los parámetros se definen indicando primero el modo de comunicación luego el nombre del identificador y por último su tipo.

Si se necesitan definir más de un parámetro se utiliza como separador el carácter coma (,).

El lenguaje define tres modos de comunicación para los parámetros:

- **en**: modo entrada, este modo de comunicación permite pasar valores al procedimiento como copia del argumento utilizado en la llamada.

  Si el argumento utilizado en la llamada es una variable el valor permanecerá inalterado luego de la ejecución del procedimiento.

- **sa**: modo salida, este modo permite comunicar el valor generado dentro del procedimiento a quien lo llamó, se debe utilizar como argumento en la llamada una variable puesto que se utiliza su referencia.

  Cuando el procedimiento es iniciado el valor es inicializado de acuerdo a las reglas descriptas para las variables.

- **es**: modo entrada y salida, este modo permite comunicar un valor hacia el procedimiento y retornar a quien lo llamó el valor alterado durante la ejecución del procedimiento.

  Al igual que en el modo salida se debe utilizar una variable ya que se utiliza la referencia de la misma con la salvedad de que en este caso no se inicializa su valor.

3.4.10 – Funciones primitivas

El lenguaje incorpora diferentes funciones para la manipulación de cadenas, números y valores lógicos. A continuación se describe cada una de ellas y se incorpora un ejemplo sobre su posible uso:

primitiva	descripción
aleatorio	Retorna un valor aleatorio entre 0 y el número anterior definido en su parámetro de entrada. Sintaxis: **aleatorio**(en limite:numero) **retorna numero** Ejemplos: 1) n := aleatorio(10)+1 2) repetir aleatorio(5) informar("…")
longitud	Retorna la longitud de un texto pasado como argumento a la función. Sintaxis: **longitud**(en cadena:texto) **retorna numero** Ejemplo: cantidad := longitud("Hola mundo…")

sustraer	Retorna una sub-cadena de la cadena original pasada como argumento. La sub-cadena será la que surja de la extracción según las posiciones definidas en los argumentos. Sintaxis: **sustraer**(en cadena:texto; en desde:numero; en hasta:numero) **retorna texto** Ejemplos: 1) subcadena := sustraer("original", 2, 3) 2) informar(sustraer("hola mundo",1,4))
numeroATexto	Retorna en forma de texto el número pasado como argumento, si número no es válido se produce un error. Sintaxis: **numeroATexto** (en valor:numero) **retorna texto** Ejemplos: 1) cadena := numeroATexto(2+5) 2) cadena := sustraer(numeroATexto(10*10+1),1,2)
textoANumero	Retorna en forma de número el texto pasado como argumento, si no se puede convertir se produce un error. Sintaxis: **textoANumero**(en cadena:texto) **retorna numero** Ejemplo: n := textoANumero("100")
logicoATexto	Retorna en forma de texto el valor lógico pasado como argumento.

	Sintaxis: **logicoATexto** (en val:logico) **retorna texto** Ejemplo: cadena := logicoATexto(v)
hayRobot	Retorna verdadero si se encuentra un robot en la avenida y calle pasadas como argumentos. Si se pasan valores que superen los límites de la ciudad retorna falso. Sintaxis: **hayRobot** (en avenida:numero; en calle:numero) **retorna logico** Ejemplo: puedoAvanzar := hayRobot (posAv+1,posCa) si(puedoAvanzar) mover

Tabla 4 – DaVinci Concurrente – primitivas

3.4.11 – Hilos

El lenguaje brinda mecanismos para permitir la ejecución de diferentes bloques de código en una "aparente" simultaneidad. Estos bloques de código deberán ser declarados en un área especial denominada "hilos".

Cada bloque de código será definido mediante la palabra reservada "hilo", y como describimos en los procedimientos, serán nombrados mediante un identificador y parámetros. Se destaca que solo se permitirán parámetros de entrada.

Cada hilo podrá colocarse en la cola de ejecución mediante la palabra reservada "arrancar".

El siguiente ejemplo muestra la utilización de dos hilos para imprimir por pantalla diferentes valores y se muestra una posible salida producto de su ejecución:

```
programa ejemploHilo


//definición de hilos
hilos

    hilo calcular(en num:numero; en incremento:numero)
    comenzar
     repetir 10
      comenzar
           informar(num)
           num := num + incremento
       fin
    fin

//programa principal
comenzar

 //imprime los primeros 10 pares
 arrancar calcular(2,2)

 //imprime los primeros 10 impares
 arrancar calcular(1,2)

fin
```

Salida: 2, 4, 1, 3, 6, 5, 8, 10, 7, 9, 11,13, 12, 14, 16, 18, 15, 17, 19, 20

El código anterior al ser ejecutado puede imprimir la salida que se expuso, aunque también podría ser otro totalmente diferente debido al planificador que se esté utilizando. En secciones posteriores se profundizará esta característica.

Los hilos serán ejecutados de manera simultánea no pudiendo saber quién tomará el control para interpretar la próxima instrucción.

Se puede decir que la secuencia lógica obtenida de la ejecución de un programa con hilos no podrá ser determinada con antelación por lo tanto se proveen primitivas para permitir al alumno poder sincronizar los distintos hilos y/o proteger determinados recursos.

3.4.11.1 – Primitivas aplicables a semáforos

Las primitivas que se proveen y se describen a continuación serán aplicables a las variables definidas, del tipo semáforo binario o del tipo semáforo general y serán las únicas con las cuales podrán ser manipuladas estas variables:

- iniciarSemaforo: inicializa el valor del semáforo en uno (1), también es posible declarar el valor con el cual se quiere iniciar el semáforo.

  Si se inicia un semáforo binario con un valor mayor a uno se producirá un error en tiempo de ejecución indicando esta circunstancia.

- esperar: es la primitiva encargada de disminuir el valor del semáforo.

  Si el valor se hace negativo el hilo que ejecuta la primitiva se bloquea.

- avisar: es la primitiva encargada de incrementar el valor del semáforo.

  Si el valor no es positivo, se desbloquea un hilo bloqueado por una llamada a la primitiva esperar. (Stallings 1997)

primitiva	descripción
iniciarSemaforo(semaforoB)	Inicializa el semáforo binario identificado con el nombre "semaforoB" con el valor uno (1).
iniciarSemaforo(semaforoG, 10)	Inicializa al semáforo general identificado con el nombre "semaforoG" con el valor diez (10), este argumento es opcional.
esperar(semáforo)	Realiza la operación esperar sobre el semáforo pasado como argumento. Si el valor se hace negativo se suspende el hilo que lo ejecuta.
avisar(semáforo)	Realiza la operación avisar sobre el semáforo pasado como argumento. Si el valor no es positivo, se desbloquea un hilo bloqueado.
arrancar identHilo(params...)	Inserta al hilo en la cola de ejecución procesando los parámetros pasados como argumento, cuando el planificador lo indique será ejecutado de acuerdo a las reglas que este imponga.

Tabla 5 – DaVinci Concurrente – Primitivas de semáforos

Es importante resaltar que se pueden realizar tantas llamadas como se requieran a la primitiva "arrancar" con el mismo identificador de hilo siempre y cuando en su interior no contengan primitivas asociadas al robot.

Como se menciona en párrafos anteriores, un programa en ejecución terminará solo si todos sus hilos han terminado o si se produce alguna situación de error. Los hilos pueden ser lanzados con la primitiva arrancar y terminarán cuando finalice la ejecución de las sentencias de su interior. Al programa principal se lo puede considerar un caso especial de hilo, si bien tiene las mismas características que lo hilos, este será lanzado al comenzar la ejecución del programa y no podrá volver a ejecutarse una vez terminado.

3.4.12 – Robots

Los robots, que son una parte esencial del lenguaje ya que con ellos se podrá visualizar la salida que generarán los códigos al ser ejecutados, se pueden incorporar a la ciudad a través de la primitiva "iniciar".

La primitiva iniciar no es obligatoria pero sí será necesaria para incorporar al robot a la ciudad.

Si un robot no ha sido incorporado y se realiza una operación que está asociada a él, como por ejemplo el caso de las primitivas para depositar o levantar flores, se producirá un error en tiempo de ejecución abortando la ejecución del programa e indicando cual fue el motivo.

En el siguiente ejemplo se pueden visualizar dos programas casi idénticos que no advierten problemas previos a su ejecución (problemas de sintaxis) pero uno de ellos sí produce un error en tiempo de ejecución ya que se procede a ejecutar una primitiva que responde al robot y este no ha sido insertado:

```
programa produceError

//programa principal
comenzar



 //avanza una cuadra al robot
 mover

 //informa la posición
 informar(posAv," ",posCa)

fin
```

```
programa funcionaBien

//programa principal
comenzar

 //inserta al robot en la ciudad
 iniciar

 //avanza una cuadra al robot
 mover

 //informa la posición
 informar(posAv," ",posCa)

fin
```

3.4.12.1 – Primitivas y variables aplicables al robot

Cuando un robot es insertado en la ciudad puede ser controlado sólo a través de las primitivas que se detallan a continuación. En secciones posteriores (múltiples robots) se detalla como impactan las primitivas al robot (la primitiva aplica al robot bajo su ámbito).

primitiva	descripción
iniciar	Introduce un robot en la ciudad en la avenida uno y calle uno (1,1) orientándolo hacia el norte. Esta primitiva solo puede ser invocada una única vez en cada hilo de ejecución incluido el programa principal.
pos	Posiciona al robot en la avenida y calle pasados como argumentos. Sintaxis: pos (en av:numero; en ca:numero) Ejemplo: pos(10,20)
mover	Avanza una cuadra al robot en la dirección que este se encuentre. Sintaxis: mover Ejemplo: repetir 5 mover
derecha	Gira 90° al robot de acuerdo su orientación. El giro se produce en el mismo sentido que las agujas

	del reloj, las posibles orientaciones son norte, sur, este y oeste. Sintaxis: derecha Ejemplo: //orientamos al robot a la izquierda repetir 3 derecha
tomarFlor	Toma una flor de la esquina en la que se encuentra el robot. Si se intentase tomar una flor que no existe se producirá un error que abortará la ejecución del programa. Sintaxis: tomarFlor Ejemplo: repetir 4 mover tomarFlor
tomarPapel	Toma un papel de la esquina en la que se encuentra el robot. Si se intentase tomar un papel que no existe se producirá un error que abortará la ejecución del programa. Sintaxis: tomarPapel Ejemplo: repetir 9 mover tomarPapel

depositarFlor	Deposita una flor en la esquina en la que se encuentra el robot. Si se intentase depositar una flor en una esquina y el robot no tuviera flores en la bolsa se producirá un error que abortará la ejecución del programa. Sintaxis: depositarFlor Ejemplo: si(hayFlorEnLaBolsa) depositarFlor
depositarPapel	Deposita un papel en la esquina en la que se encuentra el robot. Si se intentase depositar un papel en una esquina y el robot no tuviera papeles en la bolsa se producirá un error que abortará la ejecución del programa. Sintaxis: depositarPapel Ejemplo: si(hayPapelEnLaBolsa) depositarPapel
posAv	Retorna la avenida en la que se encuentra el robot. Sintaxis: posAv Ejemplo: avenida := posAv
posCa	Retorna la calle en la que se encuentra el robot. Sintaxis:

	posCa Ejemplo: calle := posCa
hayFlorEnLaEsquina	Retorna verdadero en caso de existir al menos una flor en la esquina en la que se encuentra el robot; en caso contrario retorna falso. Sintaxis: hayFlorEnLaEsquina Ejemplo: mientras hayFlorEnLaEsquina tomarFlor
hayPapelEnLaEsquina	Retorna verdadero en caso de existir al menos papel en la esquina en la que se encuentra el robot, caso contrario retorna falso. Sintaxis: hayPapelEnLaEsquina Ejemplo: mientras hayPapelEnLaEsquina tomarPapel
hayFlorEnLaBolsa	Retorna verdadero en caso de existir en la bolsa del robot al menos una flor. Sintaxis: hayFlorEnLaBolsa Ejemplo: si hayFlorEnLaBolsa depositarFlor

hayPapelEnLaBolsa	Retorna verdadero en caso de existir en la bolsa del robot al menos un papel. Sintaxis: hayPapelEnLaBolsa Ejemplo: si hayPapelEnLaBolsa depositarPapel
hayObstaculo	Retorna verdadero en caso de que en la siguiente cuadra, hacia donde se encuentra orientado el robot, exista un obstaculo. En cualquier otro caso retorna falso. Sintaxis: hayObstaculo Ejemplo: si(!hayObstaculo) mover

Tabla 6 – DaVinci Concurrente – primitivas de robot

3.4.12.2 – Cómo incorporar robots adicionales

Una característica que posee el lenguaje y que ha sido mencionada es que se pueden insertar varios robots en la ciudad.

El programa principal es considerado un hilo y cuando se ejecuta la primitiva **iniciar** esta incorpora a la ciudad un robot relacionándolo directamente a este hilo. Lo mismo sucede con las primitivas asociadas al robot.

Para incorporar otro robot a la ciudad, entonces, será necesario definirlo bajo un nuevo hilo que los contenga.

A continuación se muestra un ejemplo que incorpora dos robots a la ciudad y estos realizan las mismas operaciones diferenciándose únicamente con su posición:

```
programa ejemploRobots2

//definición de hilos
hilos

 hilo robot2
 comenzar

  //inserta en la ciudad otro robot
  iniciar

  //posiciona al robot
  Pos(20,20)

  //avanza una cuadra
  mover

  //informa la posición
  Informar(posAv," ",posCa)

 fin

//programa principal
comenzar

 //inserta al robot en la ciudad
 iniciar

 //inicia el hilo del segundo robot
 arrancar robot2

 //avanza una cuadra
 mover

 //informa la posición
 Informar(posAv," ",posCa)

Fin
```

El programa anterior inserta dos robots en la ciudad y se realizan las mismas acciones diferenciándose entre ellos únicamente en la posición que cada uno se encuentra. De esta manera se pude observar como cada primitiva asociada al robot

responde según el hilo donde se produce, afectando cada una al robot contenido en él. Al ejecutar el código se podrá observar que se informan dos posiciones diferentes correspondiéndose cada una a cada robot.

Otro manera de escribir el mismo programa es haciendo uso de los subprogramas. Cuando un subprograma es llamado desde un hilo se transfiere la referencia del hilo para poder diferenciar a que robot aplicar las diferentes primitivas.

```
programa ejemploRobots2

//definición de subprogramas
subprogramas

 procedimiento hacerAlgo
 comenzar

  //avanza una cuadra
  mover

  //informa la posición
  Informar(posAv," ",posCa)

 fin

//definición de hilos
hilos

 hilo robot2
 comenzar

  //inserta en la ciudad otro robot
  iniciar

  //posiciona al robot
  Pos(20,20)

  //llama al subprograma
  hacerAlgo

 fin

//programa principal
comenzar

 //inserta al robot en la ciudad
 iniciar
```

```
    //inicia el hilo del segundo robot
    arrancar robot2

    //llama al subprograma
    hacerAlgo

fin
```

Se puede observar en este ejemplo que cada primitiva, aplicable al robot, del interior del subprograma responde según el hilo donde se ejecute. Cada primitiva afectará al robot contenido en el hilo que se hizo la llamada.

Existe una función primitiva llamada **hayRobot** que tendrá un valor agregado a la hora de utilizarse en un escenario con múltiples robots en ejecución y una ciudad configurada para que no puedan superponerse.

De acuerdo a la configuración que se establezca, los robots pueden o no superponerse. Si se configura para que puedan superponerse los robots que se encuentren en la misma esquina en un instante dado no tendrán ningún inconveniente, si la ciudad se encuentra configurada para que no se superpongan los robots no podrán ocupar la misma más de un robot, salvo la esquina inicial avenida 1, calle 1 que sí la podrán ocupar en cualquier momento.

3.4.13 – Planificador

El lenguaje se encuentra diseñado con la característica especial de que se puede cambiar el planificador de ejecución. Se ha puesto especial atención en esta parte del diseño para permitir al alumno estudiar diferentes aspectos que intervienen en la resolución de algoritmos de manera concurrente y como estos se ven afectados por las distintas implementaciones de los planificadores.

3.4.13.1 – Planificadores implementados

En la presente publicación se definieron para su utilización cuatro planificadores que se pueden seleccionar en el momento previo a una ejecución:

- Planificador esencial: es el planificador de ejecución elemental y se aconseja para algoritmos que no hagan uso de características concurrentes debido a que utiliza una especie de cola de tareas con política FIFO (también conocido como FCFS). La utilidad adicional de este planificador es que el resto de los planificadores podrán heredar su comportamiento y solo tendrán que redefinir la política de ejecución de las tareas. Esto último trae como ventaja la facilidad con la se puede implementar otros planificadores.
- Planificador aleatorio: Se selecciona de manera aleatoria la siguiente tarea a ejecutar de la cola de tareas. Si bien puede parecer en principio un algoritmo poco sofisticado también se puede decir que el costo de procesamiento para la planificación de las tareas es muy bajo.
- Planificador cíclico (round-robin): Este planificador se encuentra diseñado para hacer un reparto equitativo del tiempo que tienen las tareas para utilizar el procesador (ejecutarse). Es útil para algoritmos que utilicen concurrencia debido al equilibrio que presenta en la selección de la tarea y el tiempo que esta se ejecutará. Básicamente el algoritmo agrega las tareas en una cola de tareas listas para ejecutarse y selecciona la que se encuentra en la cabecera para su ejecución. Si se acaba el tiempo (quantum) otorgado, la tarea se retorna a la cola de listas. Si una tarea, que se encuentra en ejecución, solicita un recurso no disponible se bloquea y se pasa a la cola de tareas bloqueadas. Las tareas bloqueadas irán saliendo de la cola en función de la liberación del recurso que causó su bloqueo.
- Planificador repetitivo: Todos los planificadores anteriormente descriptos tienen la capacidad de retornar la secuencia lógica de ejecución que se generó al ejecutar un programa. El planificador repetitivo recibirá como entrada la secuencia lógica de ejecución generada a partir de la ejecución de algún programa o generada manualmente y la ejecutará en orden hasta terminarla. Este planificador es útil en dos sentidos: en primer lugar ejecutar un programa sabiendo de antemano el camino lógico no recarga al procesador en la elección de la tarea a ejecutar y en segundo lugar al obtener la entrada ésta podría ser modificada previa a la inserción dentro del planificador y de esta manera permitir la “manipulación lógica de la

ejecución". Es responsabilidad del programador asegurar que la secuencia lógica propuesta es coherente con el código escrito.

3.5 – Sintaxis

3.5.1 – Diagramas de ferrocarril

Se deja en el Anexo B, los diagramas de sintaxis del lenguaje, también conocidos como diagramas de ferrocarril. Estos diagramas son realmente útiles para tener el primer contacto con el lenguaje, son una alternativa gráfica para la Forma de Backus-Naur (BNF) o la forma Extendida (EBNF).

3.5.2 – Forma extendida de Backus-Naur (EBNF)

A continuación mostraremos mediante la notación extendida de bnf la sintaxis del lenguaje, posterior a la tabla se dejan los terminales y su sustitución:

ProgramaDavinciConcurrente	**::=**	**<PROGRAM> <IDENTIFIER> Declaraciones Cuerpo <EOF>**
Declaraciones	::=	(Variables)? (Subprogramas)? (Hilos)? (Variables)?
Cuerpo	::=	<BEGIN> (Sentencia)* <END>
Hilos	::=	<THREADS> (DefinicionHilo)+
DefinicionHilo	::=	(<THREAD> <IDENTIFIER> (<LPAREN> ParametrosFormalesHilo <RPAREN>)? (Variables)? Cuerpo)
ParametrosFormalesHilo	::=	(ParametroFormalEntradaHilo (<SEMICOLON> ParametroFormalEntradaHilo)*)
ParametroFormalEntradaHilo	::=	<INPUT> <IDENTIFIER> <COLON> TipoPrimitivo
Subprogramas	::=	<SUBPROGRAM> (DefinicionProcedimiento)+
DefinicionProcedimiento	::=	(<PROCEDURE> <IDENTIFIER> (<LPAREN> ParametrosFormales <RPAREN>)? (Variables)? Cuerpo)
ParametrosFormales	::=	(ParametroFormal (<SEMICOLON> ParametroFormal)*)
ParametroFormal	::=	ParametroFormalEntrada | ParametroFormalSalida | ParametroFormalEntradaSalida
ParametroFormalEntrada	::=	<INPUT> <IDENTIFIER> <COLON> TipoPrimitivo

ParametroFormalSalida	::=	<OUTPUT> <IDENTIFIER> <COLON> TipoPrimitivo
ParametroFormalEntradaSalida	::=	<INPUTOUTPUT> <IDENTIFIER> <COLON> TipoPrimitivo
Variables	::=	<VAR> (DefinicionVariable)+
DefinicionVariable	::=	<IDENTIFIER> <COLON> TipoPrimitivo
TipoPrimitivo	::=	<INTEGER> | <BOOLEAN> | <CHAR> | <STRING> | <SEMAPHORE_BINARY> | <SEMAPHORE_GENERAL>
Sentencia	::=	(Primitiva | SentenciaSimple | SentenciaCompuesta)
Primitiva	::=	<START>| <MOVE>| <LEFT>|<TAKEFLOWER>|<TAKEPAPER>| <PUTFLOWER>|<PUTPAPER>
SentenciaSimple	::=	Asignacion | Invocacion
Asignacion	::=	Identificador <ASSIGN> Expresion
Identificador	::=	<IDENTIFIER>
Expresion	::=	(ExpresionSimple (OperadorRelacional ExpresionSimple)?)
ExpresionSimple	::=	((SignoTermino | Termino) (OperadorAditivo Termino)*)
Termino	::=	Factor (OperadorMultiplicativo Factor)*
Factor	::=	(Identificador | Constante | VariablePrimitiva | FuncionPrimitiva | <LPAREN> Expresion<RPAREN> | (<NOT> Factor))
FuncionPrimitiva	::=	(Aleatorio | Longitud | Sustraer | NumeroATexto | TextoANumero | LogicoATexto)
Aleatorio	::=	<RANDOM> <LPAREN> Expresion<RPAREN>
Longitud	::=	<LENGTH> <LPAREN> Expresion<RPAREN>
Sustraer	::=	<SUBSTRING> <LPAREN> Expresion<COMMA> Expresion<COMMA> Expresion<RPAREN>
NumeroATexto	::=	<NUMBERTOTEXT> <LPAREN> Expresion<RPAREN>
TextoANumero	::=	<TEXTTONUMBER> <LPAREN> Expresion<RPAREN>
LogicoATexto	::=	<LOGICTOTEXT> <LPAREN> Expresion<RPAREN>
VariablePrimitiva	::=	(<POSAV> | <POSCA> | <HAYFLORENLAESQUINA> | <HAYFLORENLABOLSA> | <HAYPAPELENLAESQUINA> | <HAYPAPELENLABOLSA> | <HAYOBSTACULO>)

OperadorAditivo	::=	(<PLUS> | <MINUS> | <OR>)
OperadorMultiplicativo	::=	(<PROD>| <MOD> | <DIV> | <AND>)
OperadorRelacional	::=	(<EQ> | <LE> | <GT> | <LT> | <GE> | <NE>)
SignoTermino	::=	((<PLUS> | <MINUS>) Termino)
Constante	::=	(<INTEGER_LITERAL>) | (<STRING_LITERAL>) | (<TRUE>) | (<FALSE>)
SentenciaCompuesta	::=	Seleccion | IteracionCondicional | IteracionIncondicional
Seleccion	::=	<IF> Expresion (Bloque) (<ELSE> (Bloque))?
Bloque	::=	(Sentencia | Cuerpo)
IteracionCondicional	::=	<WHILE> Expresion (Bloque)
IteracionIncondicional	::=	<REPEAT> Expresion (Bloque)
Invocacion	::=	(ProcedimientoUsuarioHilo | ProcedimientoUsuario | ProcedimientoPrimitivo)
ProcedimientoUsuario	::=	<IDENTIFIER> (<LPAREN> (Identificador | Expresion) (<COMMA> (Identificador | Expresion))* <RPAREN>)?
ProcedimientoUsuarioHilo	::=	<THREAD_START> <IDENTIFIER> (<LPAREN> Expresion (<COMMA> Expresion)* <RPAREN>)?
ProcedimientoPrimitivo	::=	Informar | Pos |Leer| IniciarSemaforo | Senal | Esperar
Informar	::=	<MESSAGE> <LPAREN> Expresion (<COMMA> Expresion)* <RPAREN>
Pos	::=	<POSITION> <LPAREN> Expresion<COMMA> Expresion<RPAREN>
Leer	::=	<READ><LPAREN> Identificador <RPAREN>
IniciarSemaforo	::=	<SEMAPHORE_INIT> <LPAREN> Identificador (<COMMA> Expresion)? <RPAREN>
Senal	::=	<SIGNAL> <LPAREN> Identificador <RPAREN>
Esperar	::=	<WAIT> <LPAREN> Identificador <RPAREN>

Tabla 7 – DaVinci Concurrente – EBNF

TOKEN (TERMINAL)	VALOR
BOOLEAN	logico
STRING	texto
INTEGER	numero
TRUE	v
FALSE	f
PROGRAM	programa
SUBPROGRAM	subprograma
PROCEDURE	procedimiento
INPUT	en
OUTPUT	sa
INPUTOUTPUT	es
VAR	variables
BEGIN	comenzar
IF	si
ELSE	sino
WHILE	mientras
REPEAT	repetir
END	fin
START	iniciar
MOVE	mover
LEFT	derecha
TAKEFLOWER	tomarFlor
PUTFLOWER	depositarFlor
TAKEPAPER	tomarPapel
PUTPAPER	depositarPapel
POSAV	posAv
POSCA	posCa
HAYFLORENLAESQUINA	hayFlorEnLaEsquina
HAYFLORENLABOLSA	hayFlorEnLaBolsa
HAYPAPELENLAESQUINA	hayPapelEnLaEsquina
HAYPAPELENLABOLSA	hayPapelEnLaEsquina
HAYOBSTACULO	hayObstaculo
MESSAGE	informar
POSITION	pos
READ	pedir
THREADS	hilos
THREAD	hilo
THREAD_START	arrancar
SEMAPHORE_BINARY	semaforoBinario

SEMAPHORE_GENERAL	semaforoGeneral
SEMAPHORE_INIT	iniciarSemaforo
SIGNAL	avisar
WAIT	esperar
RANDOM	aleatorio
LENGTH	longitud
SUBSTRING	sustraer
NUMBERTOTEXT	numeroATexto
TEXTTONUMBER	textoANumero
LOGICTOTEXT	logicoATexto
INTEGER_LITERAL	["0"-"9"] (["0"-"9"])*
STRING_LITERAL	"\"" (~["\"","\\","\n","\r"] "\\" (["n","t","b","r","f","\\","\'","\""] ["0"-"7"] (["0"-"7"])? ["0"-"3"] ["0"-"7"] ["0"-"7"]))* "\""
IDENTIFIER	["a"-"z","A"-"Z","_","ñ","Ñ"] (["a"-"z","A"-"Z","0"-"9","_","ñ","Ñ"])*
LPAREN	(
RPAREN	)
LBRACE	{
RBRACE	}
SEMICOLON	;
COLON	:
COMMA	,
ASSIGN	:=
EQ	=
LE	<=
GT	>
LT	<
GE	>=
NE	<>
OR	|
AND	&
NOT	!
PLUS	+
MINUS	-
PROD	*
DIV	/
MOD	%

Tabla 8 – DaVinci Concurrente – EBNF – Terminales

3.6 – Semántica

A continuación se describe cual es el comportamiento de cada palabra reservada del lenguaje.

declaraciones	programa	Indica el comienzo de un programa. Debe continuar con un identificador.
	subprogramas	Indica el inicio del área de declaraciones de los procedimientos.
	procedimiento	Indica el comienzo de la declaración de un procedimiento. Debe continuar con un identificador y opcionalmente los parámetros formales.
	en	Establece la clase del parámetro como entrada.
	sa	Establece la clase del parámetro como salida.
	es	Establece la clase del parámetro como entrada y salida.
	hilos	Indica el inicio del área de declaraciones de los hilos.
	hilo	Indica el comienzo de la declaración de un hilo. Debe continuar con un identificador y opcionalmente los parámetros formales.
	variables	Indica el inicio del área de declaraciones de las

		variables.
	numero	Establece que el tipo de dato asociado al identificador será numérico.
	logico	Establece que el tipo de dato asociado al identificador será lógico.
	texto	Establece que el tipo de dato asociado al identificador será un texto.
	carácter	Establece que el tipo de dato asociado al identificador será un carácter.
	semaforoBinario	Establece que el tipo de dato asociado al identificador será un semáforo de tipo binario.
	semaforoGeneral	Establece que el tipo de dato asociado al identificador será un semáforo de tipo general.
	comenzar	Indica el comienzo de un bloque de programa.
	fin	Indica la finalización de un bloque de programa.
estructuras de control	repetir	Indica el comienzo de una estructura de control repetitiva. Debe continuar con una expresión numérica la cual indicará la cantidad de veces que se repetirá la sentencia o bloque definidos a continuación.

	mientras	Indica el comienzo de una estructura de control iterativa. Debe continuar con una expresión lógica la cual indicará que mientras se evalúe y de cómo resultado verdadero la sentencia o bloque definido a continuación seguirá iterando.
	si	Indica el comienzo de una estructura de control de selección. Debe continuar con una expresión lógica la cual al momento de evaluar da como resultado verdadero se procesa la sentencia o bloque definido a continuación.
	sino	Forma parte de la estructura de selección. Se utiliza para indicar un camino alternativo cuando se evalúa la expresión y da como resulta falso. Se procesa la sentencia o bloque definido a continuación.
primitivas	iniciar *	Agrega el robot a la ciudad situándolo en la avenida 1 calle 1 Si la primitiva iniciar se encuentra en el cuerpo de un hilo generará que se agregue un robot a la ciudad con el mismo nombre que el hilo posee.
	mover *	Desplaza al robot una cuadra en el sentido que se encuentra orientado.
	derecha *	Orienta al robot haciendo que gire 90 grados en sentido de las agujas del reloj.

	tomarFlor *	Toma una flor en la posición que se encuentra incrementando en 1 la cantidad que lleva en su bolsa. De no existir una flor la posición que se encuentra se produce un error.
	depositarFlor *	Deposita una flor en la posición que se encuentra disminuyendo en 1 la cantidad que lleva en su bolsa. De no existir una flor en su bolsa se produce un error.
	tomarPapel *	Toma un papel en la posición que se encuentra incrementando en 1 la cantidad que lleva en su bolsa. De no existir un papel en la posición que se encuentra se produce un error.
	depositarPapel *	Deposita un papel en la posición que se encuentra disminuyendo en 1 la cantidad que lleva en su bolsa. De no existir una flor en su bolsa se produce un error.
funciones	posAv *	Retorna el número de avenida en la que se encuentra posicionado.
	posCa *	Retorna el número de calle en la que se encuentra posicionado.
	hayFlorEnLaEsquina *	Retorna verdadero si existe al menos una flor en la esquina que se encuentra posicionado.
	hayFlorEnLaBolsa *	Retorna verdadero si existe al menos una flor en su bolsa.

	hayPapelEnLaEsquina *	Retorna verdadero si existe al menos un papel en la esquina que se encuentra posicionado.
	hayPapelEnLaEsquina *	Retorna verdadero si existe al menos un papel en su bolsa.
	hayObstaculo *	Retorna verdadero si en la cuadra siguiente en su orientación existe un obstáculo.
	hayRobot	Retorna verdadero si en la avenida y calle que se especifican se encuentra al menos un robot.
	longitud	Retorna la cantidad de caracteres que posee una expresión de tipo texto.
	sustraer	Retorna el texto definido por las posiciones desde y hasta sobre una expresión de tipo texto.
	numeroATexto	Convierte una expresión de tipo numérica a tipo texto.
	textoANumero	Convierte una expresión de tipo texto a una numérica.
	logicoATexto	Convierte una expresión de tipo lógica a una de tipo texto.
procedimientos	aleatorio	Retorna un numero de forma aleatoria.
	informar	Informa al usuario el resultado de una expresión.

	pos *	Posiciona al robot en la avenida y calle que se define.
	pedir	Pide al usuario un valor para el identificador pasado como argumento.
	iniciarSemaforo	Inicia los semáforos de tipo binario y general.
	arrancar	Arranca con la ejecución de un hilo.
	avisar	Realiza sobre un semáforo la operación wait.
	esperar	Realiza sobre un semáforo la operación signal.

Tabla 9 – DaVinci Concurrente – significado semántico

* Si se encuentran definidos bajo el ámbito de un hilo y este contiene en su interior la primitiva iniciar entonces responderán al robot del hilo.

3.6.1 – Semántica guiada por ejemplos

A continuación se muestra mediante ejemplos como se utilizan las palabras reservadas del lenguaje

<table>
<tr><td>declaraciones / tipos de datos</td><td>programa
subprogramas
procedimiento
en
sa
es
hilos
hilo
variables
comenzar
fin

numero
logico
texto
carácter
semaforoBinario
semaforoGeneral</td><td colspan="2"><pre>programa ejemplo
variables
 semaforo : semaforoBinario

subprogramas
 procedimiento saltar(en pasos:numero)
 comenzar
  Pos(posAv + pasos, posCa)
  …
 fin

hilos
 hilo juan(en av:numero en ca:numero)
 comenzar
  iniciar
  saltar(2)
  …
 fin
variables
 num:numero
 tex:texto
 log:logico
comenzar
 …
fin</pre></td></tr>
<tr><td rowspan="3">estructuras de control</td><td>repetir</td><td><pre>repetir 4
 comenzar
 mover
 derecha
 fin</pre></td><td><pre>repetir cantidad
 mover</pre></td></tr>
<tr><td>mientras</td><td><pre>mientras cond
 comenzar
 mover
 derecha
 cond := …
 fin</pre></td><td><pre>mientras(cant<3)
 cant := cant + 1</pre></td></tr>
<tr><td>si / sino</td><td><pre>si cantidad=10
 Informar("SI!!")
sino
 Informar("No!!")</pre></td><td><pre>si (hayCantidad)
 comenzar
 informar("hay!")
 hayCantidad := f
 fin</pre></td></tr>
</table>

primitivas	iniciar mover derecha tomarFlor depositarFlor tomarPapel depositarPapel	programa ejemplo comenzar iniciar repetir 4 mientras(hayFlorEnLaBolsa) depositarFlor mover derecha si(HayPapelEnLaEsquina) tomarPapel fin
funciones	posAv posCa hayFlorEnLaEsquina hayFlorEnLaBolsa hayPapelEnLaEsquina hayPapelEnLaEsquina hayObstaculo hayRobot longitud sustraer numeroATexto textoANumero logicoATexto aleatorio	Informar("estamos en la avenida:",posAv) posicionCalle := posCa si(HayFlorEnLaEsquina) si(HayPapelEnLaBolsa) mientras !HayObstaculo mover sino si(hayRobot(posAv+1,posCa+1)) tomarFlor num0 := longitud("cuantas letras?") num1 := numeroATexto("25") num2 := aleatorio(100) cantidad := longitud(sustraer("abcde",1,2)) frase := "Hay flores en la bolsa?: " prep := logicoATexto(HayFlorEnLaBolsa) Informar(frase + prep)

procedimientos	pos pedir informar iniciarSemaforo arrancar esperar avisar	programa ejemplo variables S1:semaforoBinario Hilos hilo juan comenzar repetir 50 mover avisar(S1) fin variables avenida:numero comenzar informar("comienza el programa") iniciar pedir(avenida) pos(avenida,5) iniciarSemaforo(S1) arrancar juan esperar(S1) informar("Esquina:",posAv," ",posCa) fin

Tabla 10 – DaVinci Concurrente – Semántica guiada por ejemplos

3.7 - Implementación

La implementación del intérprete se realizó utilizando el lenguaje JAVA. La elección se debe a los conocimientos adquiridos a lo largo de la carrera y la profundización de los mismos en el campo laboral. Por otra parte se destaca la facilidad que provee de obtener un producto multiplataforma.

Si bien el trabajo consistía solamente en la especificación e implementación del intérprete, he desarrollado un entorno EDIS (**Entorno de Desarrollo Integrado Simple**), el cual se ha utilizado para probar de manera visual el intérprete. EDIS consiste básicamente en un panel de edición del código fuente, un panel de resultados donde se podrá visualizar los mensajes, un panel de estado del contexto de ejecución y finalmente un panel de la ciudad y los objetos que la componen. En el panel de la ciudad se podrá visualizar la ejecución de los algoritmos que utilicen al menos un robot.

La ciudad tendrá una implementación grafica sencilla y estará acotada a diez avenidas y diez calles. A continuación se presenta una instantánea del entorno (Fig. 5).

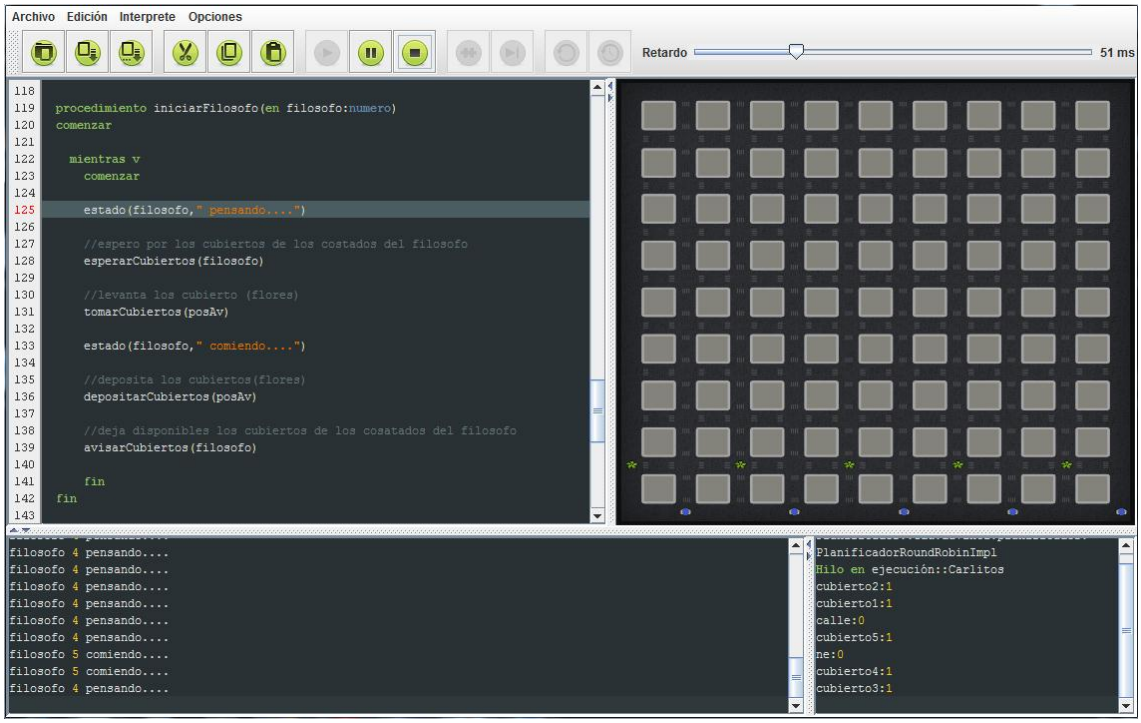


Figura 5 - EDIS(**Entorno de Desarrollo Integrado Simple**)

3.7.1 - Herramientas involucradas en el desarrollo

Si bien el intérprete pudo ser escrito en un editor de texto y luego compilado a mano, utilicé para el desarrollo el entorno integrado de desarrollo Eclipse, sobre todo por la facilidad que aporta en la escritura, lectura y depuración del código.

3.7.2 - Meta-compilador

Para la construcción del analizador léxico-sintáctico se utilizó una herramienta llamada javacc (Java Compiler Compiler) que se encuentra dentro de la categoría de meta-compiladores y de código abierto. Genera un parser a partir de una gramática especificada en notación ebnf.

3.7.3 - Javacc (Java Compiler Compiler)

En esencia javacc es una herramienta para generar analizadores léxicos y sintácticos. La herramienta acepta como entrada un determinado conjunto de especificación para un lenguaje específico y produce como salida el analizador de dicho lenguaje.

Características principales:

- Top-Down. Recursividad descendente.
- Especificación léxica y gramática en un solo archivo.
- Soporta especificaciones LL(k).
- Incluye la herramienta JJTree.

JJTree es un preprocesador para el desarrollo de árboles sintácticos. Por defecto genera código para construir cada uno de los nodos del árbol del parser correspondientes a los no terminales del lenguaje. Este comportamiento se puede modificar para que se generen los nodos que uno requiera y no todos, de hecho se pueden crear nodos para expandir solo una parte de la producción.

JJTree define una interface de Nodo que todos los nodos de los árboles del parser deben implementar. La interface provee métodos para realizar operaciones con los nodos padres e hijos.

Se puede pensar de manera sencilla que en los nodos radicará el código que representará la semántica del lenguaje.

3.8 – Trabajos Futuros

Si bien el trabajo se encuentra terminado, y se piensa que ha cumplido con todas las expectativas, es oportuno aclarar que en el desarrollo del mismo han surgido varios temas de interés que se han dejado para futuras etapas.

Dentro de estos temas, se destaca el relacionado con la evaluación en el aula. Su importancia radica en que es el último paso para poder contar con el producto pensado desde su concepción, verificando realmente si fueron de ayuda las modificaciones y agregados realizados al lenguaje.

El intérprete también se lo brindará a expertos en el área para que realicen sus aportes y evalúen si necesita cambios sustanciales.

A continuación se detallan otros temas de interés:

3.8.1 – Desacoplar los robots de los hilos:

Como se ha descripto, en varias partes del documento, el lenguaje soporta la ejecución de varios robots; éstos fueron concebidos originalmente para que existieran bajo la órbita o ambiente de un hilo contenedor. Esta concepción fue con el único objetivo de facilitar las primitivas que afectan a los robots, ya que al ejecutarlas la primitiva afectará al robot del hilo que lo contiene.

Permitir que los robots puedan ser independientes de los hilos traería ciertas ventajas adicionales. Se podría pensar en primera instancia que un mismo hilo podría ser arrancado una cantidad arbitraria de veces dotando de mayor flexibilidad al lenguaje a la hora de resolver problemas que involucren concurrencia.

A continuación se presenta a modo de ejemplo como sería un posible uso de robots desacoplados de los hilos:

```
programa RobotsEHilos

//definición de hilos
hilos

 hilo robot(en nombre:texto)
 comenzar

  //inserta en la ciudad un robot
```

```
  iniciar(nombre)

  //posiciona al robot
  Pos(aleatorio(2)+1,20,nombre)

  //hace algo
  …

 fin

//programa principal
comenzar

 //inicia el hilo
 arrancar robot("juan")

 //inicia el hilo
 arrancar robot("pepe")

fin
```

En este ejemplo se puede notar como agregando la posibilidad de hacer explicita a que robot aplica cada primitiva se pueden arrancar tantas instancias del hilo como se desee.

También se podría pensar que los argumentos de las primitivas puedan ser opcionales para facilitar el desarrollo de algoritmos que sólo requieran un único robot, ya que en un escenario en donde hay sólo un robot podría usarse el nombre del robot por defecto.

Un posible cambio de estas primitivas se detallan a continuación:

primitiva actual	primitiva propuesta
iniciar	iniciar [(nombreRobot)] **retorna texto**
mover	mover [(nombreRobot)]
izquierda	izquierda [(nombreRobot)]
pos	pos(expresion, expresion[,nombreRobot])
tomarFlor	tomarFlor [(nombreRobot)]

tomarPapel	tomarPapel [(nombreRobot)]
depositarFlor	depositarFlor [(nombreRobot)]
depositarPapel	depositarPapel [(nombreRobot)]
posAv	posAv[(nombreRobot)]
posCa	posCa[(nombreRobot)]
HayFlorEnLaEsquina	HayFlorEnLaEsquina[(nombreRobot)]
HayFlorEnLaBolsa	HayFlorEnLaBolsa[(nombreRobot)]
HayPapelEnLaEsquina	HayPapelEnLaEsquina[(nombreRobot)]
HayPapelEnLaBolsa	HayPapelEnLaBolsa[(nombreRobot)]
HayObstaculo	HayObstaculo[(nombreRobot)]

Tabla 11 – DaVinci Concurrente – primitivas de robots futuras.

3.8.2 – Incorporar funciones:

Los cambios y extensiones propuestas al lenguaje surgieron, en principio del actual Visual DaVinci, el cual no incorpora el concepto de declaración de función para su posterior uso. Al haber profundizado en los aportes significativos de las extensiones y modificaciones realizadas en DaVinci Concurrente ha surgido la posibilidad de que el lenguaje provea a futuro la declaración y el uso de funciones por parte del usuario.

Las funciones son estructuras que pueden ser remplazadas por procedimientos, y si bien el lenguaje posee esta alternativa, la utilización de funciones otorgaría más claridad, sencillez y legibilidad a la hora de realizar los algoritmos, además de acercar al alumno a lenguajes procedimentales que se utilizan en la actualidad.

Otro punto que promueve la utilización de funciones es la reutilización de código, permitiendo comprender la necesidad de bajo acoplamiento de las unidades de software como también entender la necesidad de que la función deberá realizar una tarea específica.

3.8.3 – Incorporar arreglos

Para extender la capacidad del lenguaje de resolver problemas más cercanos a la realidad se deberá permitir al alumno definir estructuras tipo arreglo para la resolución de dichos problemas.

Este tipo de estructuras, si bien en principio serán estáticas, se encuentran actualmente en uso por la mayoría de los lenguajes de programación y la utilización temprana por parte del alumno traería como ventaja la rápida internalización de este tipo de datos.

Otra ventaja de incorporar arreglos es la posibilidad de extender de manera sencilla los algoritmos de resolución concurrente posibles de realizar con el lenguaje.

4 – Anexo A – Implementaciones

4.1 – Productor-consumidor

El siguiente código resuelve el clásico problema del productor-consumidor utilizando un productor y un consumidor, los dos robots trabajaran en conjunto sobre una calle que actuará de buffer circular.

4.1.1 código

```
programa ProductorConsumidor

variables
 //globales
 lleno:semaforoGeneral
 vacio:semaforoGeneral
 turno:semaforoBinario
 turno_cuadrante:semaforoBinario
 avenida_prod:numero
 avenida_cons:numero

 //Constantes
 AV_FINAL : numero
 CALLE_CONS : numero
 CALLE_PROD : numero

subprogramas

 procedimiento tomarFlorDeCuadrante(en calle:numero;
                                    en av_fin:numero)
 comenzar
       Pos(1,calle)
       mientras ! HayFlorEnLaEsquina
             Pos((posAv % av_fin)+1,posCa)
       tomarFlor
```

```
 fin

hilos
 hilo productor (en ca_pro:numero;
                 en ca_con:numero;
                 en av_fin:numero)
 comenzar
       iniciar
       mientras v
             comenzar
             esperar(turno_cuadrante)
                   tomarFlorDeCuadrante(ca_pro, av_fin)
             avisar(turno_cuadrante)
             esperar(vacio)
                   esperar(turno)
                         Pos(avenida_prod,ca_con)
                         depositarFlor
                         avenida_prod := (avenida_prod % av_fin)+ 1

                   avisar(turno)
             avisar(lleno)
             fin
 fin

 hilo consumidor (en ca_pro:numero;
                  en ca_con:numero;
                  en av_fin:numero)
 comenzar
       iniciar
       mientras v
             comenzar
             esperar(lleno)
                   esperar(turno)
                         Pos(avenida_cons,ca_con)
```

```
                        tomarFlor
                        avenida_cons := (avenida_cons % av_fin)+ 1

                    avisar(turno)
                avisar(vacio)
                Pos(aleatorio(av_fin)+1,ca_pro)
                depositarFlor
                fin
 fin

comenzar
 //constantes
 AV_FINAL := 10
 CALLE_CONS := 2
 CALLE_PROD := 8

 //recursos compartidos
 avenida_prod := 1
 avenida_cons := 1

 //inicilizacion semaforos
 iniciarSemaforo(lleno,0)
 iniciarSemaforo(vacio,AV_FINAL)
 iniciarSemaforo(turno,1)
 iniciarSemaforo(turno_cuadrante,1)

 //arrancamos los hilos
 arrancar productor(CALLE_PROD, CALLE_CONS, AV_FINAL)
 arrancar consumidor(CALLE_PROD, CALLE_CONS, AV_FINAL)
fin
```

4.2 – Filósofos

El siguiente código resuelve el clásico problema de los filósofos, los robots utilizaran flores en vez de cubiertos.

4.2.1 – código

```
programa filosofos_

variables
 CALLE:numero

 cubierto1:semaforoBinario
 cubierto2:semaforoBinario
 cubierto3:semaforoBinario
 cubierto4:semaforoBinario
 cubierto5:semaforoBinario

subprogramas

 procedimiento esperarCubiertos(en filosofo:numero)
 comenzar
       si(filosofo=1)
             comenzar
             esperar(cubierto1)
             esperar(cubierto5)
             fin
       sino si (filosofo = 2)
             comenzar
             esperar(cubierto2)
             esperar(cubierto1)
             fin
       sino si (filosofo = 3)
             comenzar
             esperar(cubierto3)
```

```
            esperar(cubierto2)
            fin
      sino si (filosofo = 4)
            comenzar
            esperar(cubierto4)
            esperar(cubierto3)
            fin
      sino
            comenzar
            esperar(cubierto4)
            esperar(cubierto5)
            fin
fin

procedimiento avisarCubiertos(en filosofo:numero)
comenzar
      si(filosofo=1)
            comenzar
            avisar(cubierto1)
            avisar(cubierto5)
            fin
      sino si (filosofo = 2)
            comenzar
            avisar(cubierto2)
            avisar(cubierto1)
            fin
      sino si (filosofo = 3)
            comenzar
            avisar(cubierto3)
            avisar(cubierto2)
            fin
      sino si (filosofo = 4)
            comenzar
            avisar(cubierto4)
```

```
            avisar(cubierto3)
            fin
      sino
            comenzar
            avisar(cubierto4)
            avisar(cubierto5)

            fin
fin

procedimiento estado(en filosofo:numero; en mensaje:texto)
comenzar
      repetir aleatorio(10)
            informar("filosofo ",filosofo, mensaje)
fin

procedimiento tomarCubiertos(en avenida_:numero; en calle:numero)
variables
      av:numero
comenzar
      av := avenida_ - 1

      //1 3 5 7 9
      Pos(av,calle)
      tomarFlor

      // 3 5 7 9 1
      av := avenida_ + 1
      si(av=11)
            av := 1
      Pos(av,calle)
      tomarFlor

      Pos(avenida_,CALLE + 1)
```

```
    fin

    procedimiento depositarCubiertos(en avenida_:numero; en
calle:numero)
    variables
          av:numero
    comenzar
          av := avenida_ - 1

          //1 3 5 7 9
          Pos(av,calle)
          depositarFlor

          // 3 5 7 9 1
          av := avenida_ + 1
          si(av=11)
                av := 1
          Pos(av,calle)
          depositarFlor

          Pos(avenida_,calle - 1)
    fin

    procedimiento iniciarFilosofo(en filosofo:numero; en calle:numero)
    comenzar

          mientras v
                comenzar

                estado(filosofo," pensando....")

                //espero por los cubiertos de los costados del filosofo
                esperarCubiertos(filosofo)

                //levanta los cubierto (flores)
```

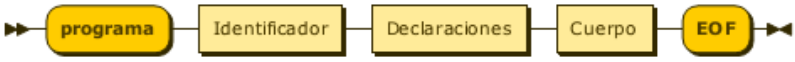
```
                tomarCubiertos(posAv,calle)

                estado(filosofo," comiendo....")

                //deposita los cubiertos(flores)
                depositarCubiertos(posAv,calle)

                //deja disponibles los cubiertos del filosofo
                avisarCubiertos(filosofo)

                fin
 fin

hilos
 hilo filosofo1(en calle:numero)
 comenzar
                iniciar
                pos(2,1)
                iniciarFilosofo(1,calle)
 fin

 hilo filosofo2(en calle:numero)
 comenzar
       iniciar
       pos(4,1)
       iniciarFilosofo(2,calle)
 fin

 hilo filosofo3(en calle:numero)
 comenzar
       iniciar
       pos(6,1)
       iniciarFilosofo(3,calle)
 fin
```

```
      hilo filosofo4(en calle:numero)
      comenzar
            iniciar
            pos(8,1)
            iniciarFilosofo(4,calle)
      fin

      hilo filosofo5(en calle:numero)
      comenzar
            iniciar
            pos(10,1)
            iniciarFilosofo(5,calle)
      fin

    comenzar

      {se debe distribuir una flor en las siguientes intersecciones:
            (av,ca): (1,2)-(3,2)-(5,2)-(7,2)-(9,2)}
      //nro de calle en la cual estarán los recursos
      CALLE := 2

      iniciarSemaforo(cubierto1,1)
      iniciarSemaforo(cubierto2,1)
      iniciarSemaforo(cubierto3,1)
      iniciarSemaforo(cubierto4,1)
      iniciarSemaforo(cubierto5,1)

      arrancar filosofo1(CALLE)
      arrancar filosofo2(CALLE)
      arrancar filosofo3(CALLE)
      arrancar filosofo4(CALLE)
      arrancar filosofo5(CALLE)
fin
```

5 – ANEXO B – Diagramas

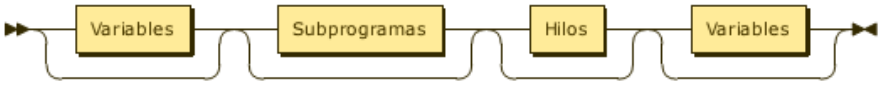
5.1 – Diagrama de sintaxis:

DavinciConcurrente:



sin referencias

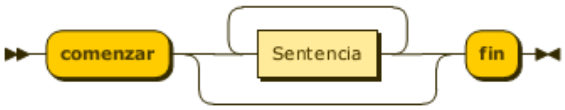
Declaraciones:



referenciado por:

- DavinciConcurrente

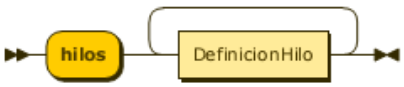
Cuerpo:



referenciado por:

- Bloque
- DavinciConcurrente
- DefinicionHilo
- DefinicionProcedimiento

Hilos:



referenciado por:

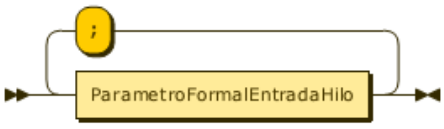
- Declaraciones

DefinicionHilo:

referenciado por:

- Hilos

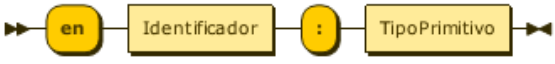
ParametrosFormalesHilo:



referenciado por:

- DefinicionHilo

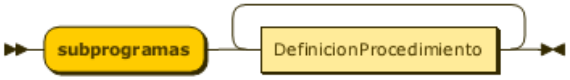
ParametroFormalEntradaHilo:



referenciado por:

- ParametrosFormalesHilo

Subprogramas:



referenciado por:

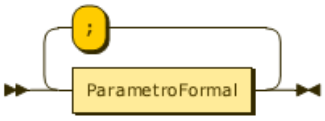
- Declaraciones

DefinicionProcedimiento:

referenciado por:

- Subprogramas

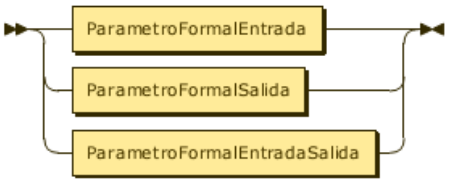
ParametrosFormales:



referenciado por:

- DefinicionProcedimiento

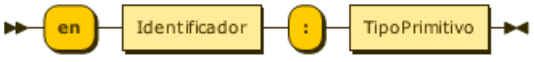
ParametroFormal:



referenciado por:

- ParametrosFormales

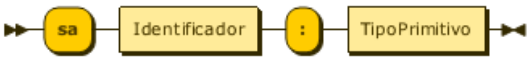
ParametroFormalEntrada:



referenciado por:

- ParametroFormal

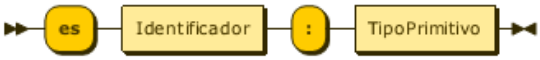
ParametroFormalSalida:



referenciado por:

- ParametroFormal

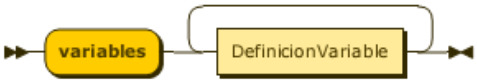
ParametroFormalEntradaSalida:



referenciado por:

- ParametroFormal

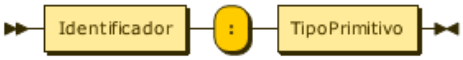
Variables:



referenciado por:

- Declaraciones
- DefinicionHilo
- DefinicionProcedimiento

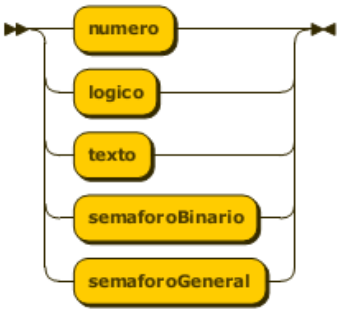
DefinicionVariable:



referenciado por:

- Variables

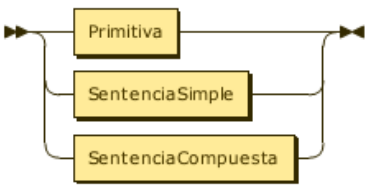
TipoPrimitivo:



referenciado por:

- DefinicionVariable
- ParametroFormalEntrada
- ParametroFormalEntradaHilo
- ParametroFormalEntradaSalida
- ParametroFormalSalida

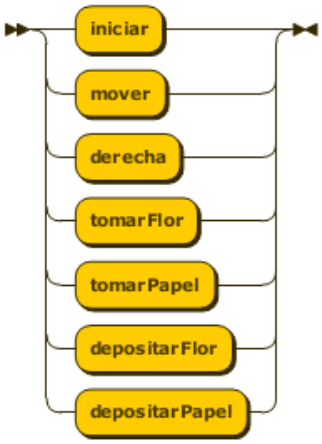
Sentencia:



referenciado por:

- Bloque
- Cuerpo

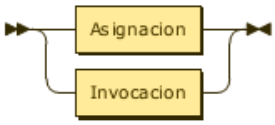
Primitiva:



referenciado por:

- Sentencia

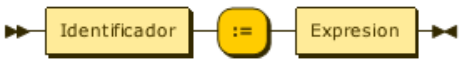
SentenciaSimple:



referenciado por:

- Sentencia

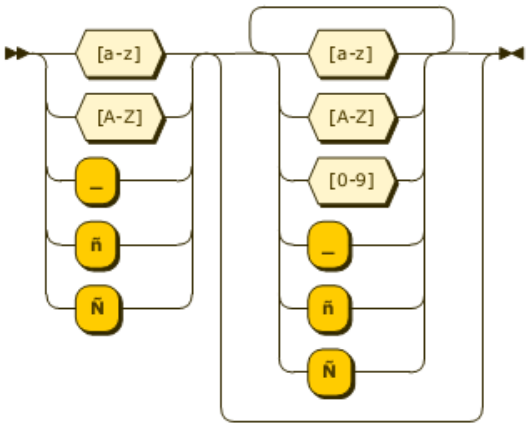
Asignacion:



referenciado por:

- SentenciaSimple

Identificador:



referenciado por:

- Asignacion
- DavinciConcurrente
- DefinicionHilo
- DefinicionProcedimiento

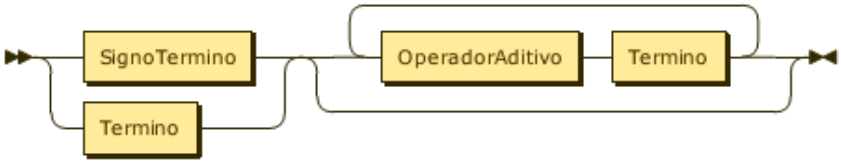
- DefinicionVariable
- Esperar
- Factor
- IniciarSemaforo
- Leer
- ParametroFormalEntrada
- ParametroFormalEntradaHilo
- ParametroFormalEntradaSalida
- ParametroFormalSalida
- ProcedimientoUsuario
- ProcedimientoUsuarioHilo
- Senal

Expresion:

referenciado por:

- Aleatorio
- Asignacion
- Factor
- Informar
- IniciarSemaforo
- IteracionCondicional
- IteracionIncondicional
- LogicoATexto
- Longitud
- NumeroATexto
- Pos
- ProcedimientoUsuario
- ProcedimientoUsuarioHilo
- Seleccion
- Sustraer
- TextoANumero

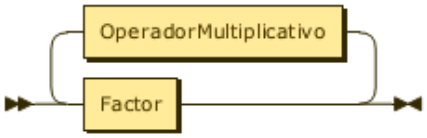
ExpresionSimple:



referenciado por:

- Expresion

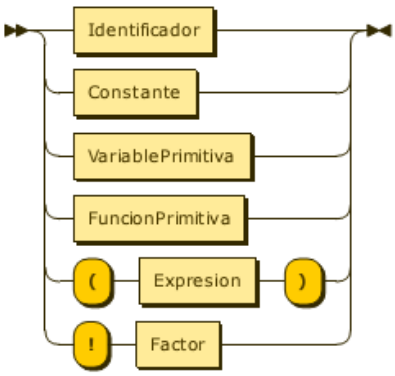
Termino:



referenciado por:

- ExpresionSimple
- SignoTermino

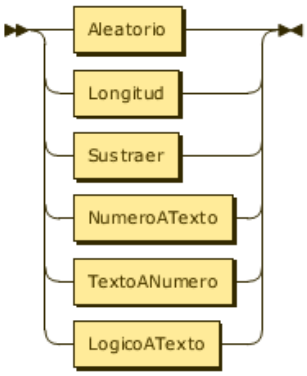
Factor:



referenciado por:

- Factor
- Termino

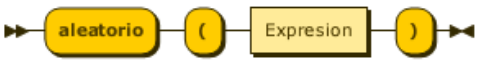
FuncionPrimitiva:



referenciado por:

- Factor

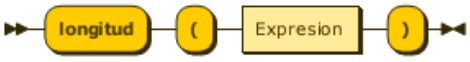
Aleatorio:



referenciado por:

- FuncionPrimitiva

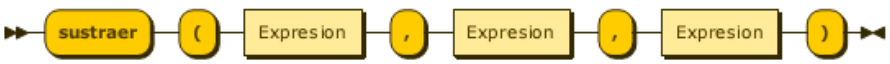
Longitud:



referenciado por:

- FuncionPrimitiva

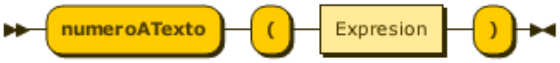
Sustraer:



referenciado por:

- FuncionPrimitiva

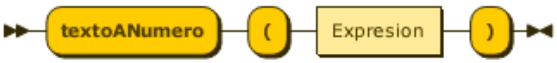
NumeroATexto:



referenciado por:

- FuncionPrimitiva

TextoANumero:



referenciado por:

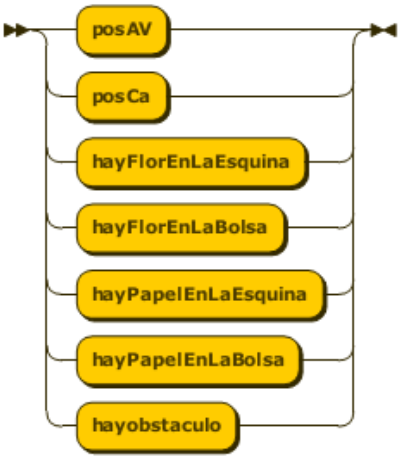
- FuncionPrimitiva

LogicoATexto:

referenciado por:

- FuncionPrimitiva

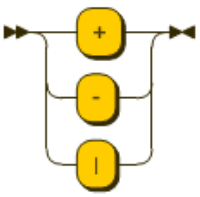
VariablePrimitiva:



referenciado por:

- Factor

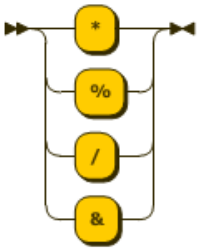
OperadorAditivo:



referenciado por:

- ExpresionSimple

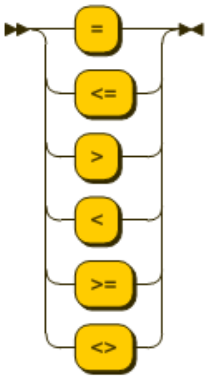
OperadorMultiplicativo:



referenciado por:

- Termino

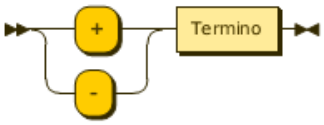
OperadorRelacional:



referenciado por:

- Expresion

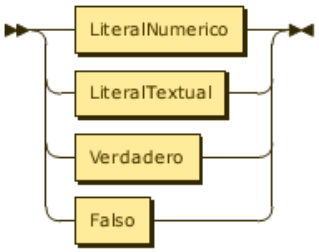
SignoTermino:



referenciado por:

- ExpresionSimple

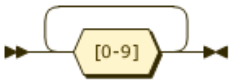
Constante:



referenciado por:

- Factor

LiteralNumerico:



referenciado por:

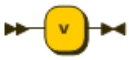
- Constante

LiteralTextual:

referenciado por:

- Constante

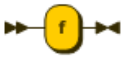
Verdadero:



referenciado por:

- Constante

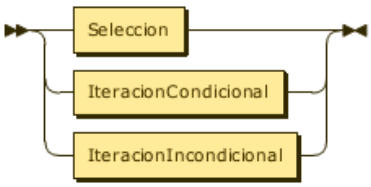
Falso:



referenciado por:

- Constante

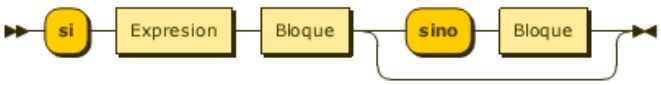
SentenciaCompuesta:



referenciado por:

- Sentencia

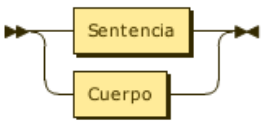
Seleccion:



referenciado por:

- SentenciaCompuesta

Bloque:



referenciado por:

- IteracionCondicional
- IteracionIncondicional
- Seleccion

IteracionCondicional:

referenciado por:

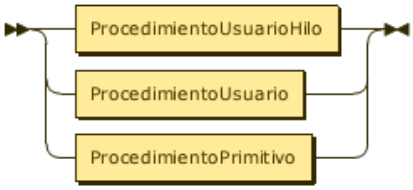
- SentenciaCompuesta

IteracionIncondicional:

referenciado por:

- SentenciaCompuesta

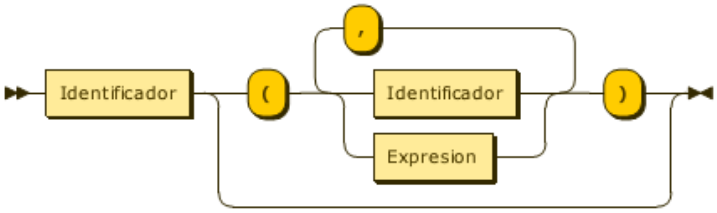
Invocacion:



referenciado por:

- SentenciaSimple

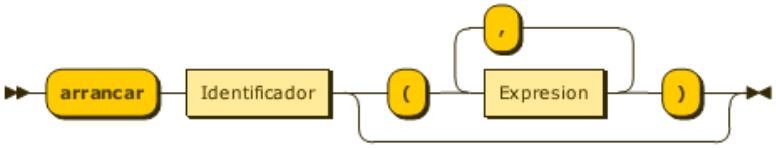
ProcedimientoUsuario:



referenciado por:

- Invocacion

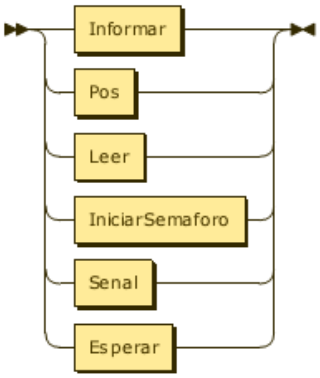
ProcedimientoUsuarioHilo:



referenciado por:

- Invocacion

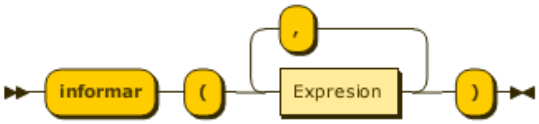
ProcedimientoPrimitivo:



referenciado por:

- Invocacion

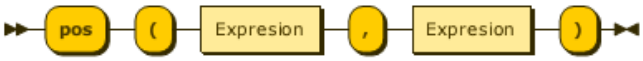
Informar:



referenciado por:

- ProcedimientoPrimitivo

Pos:



referenciado por:

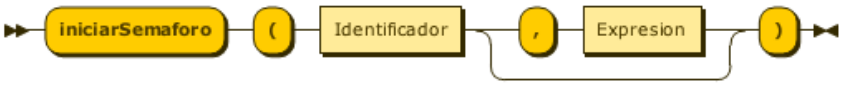
- ProcedimientoPrimitivo

Leer:

referenciado por:

- ProcedimientoPrimitivo

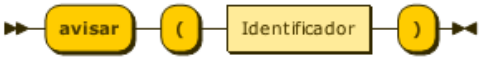
IniciarSemaforo:



referenciado por:

- ProcedimientoPrimitivo

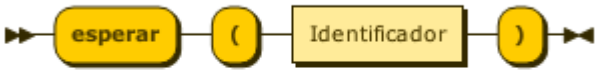
Senal:



referenciado por:

- ProcedimientoPrimitivo

Esperar:



referenciado por:

- ProcedimientoPrimitivo

6 – Bibliografía

Abraham Silberschatz, Peter B. Galvin, Greg Gagne. *Operating System Concepts.* Wiley, 2009.

Aho, A. V.; Sethi, R.; Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* 2da. Pearson/Addison Wesley, 2007.

Andrews, G. R. *Foundations of Multithreaded, Parallel and Distributed Programming.* Addison Wesley, 2000.

Appel, A. W., y J. Palseberg. *Modern Compiler Implementation in Java.* Nueva York, Estados Unidos: Cambridge University Press, 2004.

Champredonde, Raul, y Armando De Guisti. *Design and Implementation of Visual Da Vinci.* La Plata, Argentina: III Congreso Argentino en Ciencias de la Computación, 1997.

De Giusti, Armando y otros. *Algoritmos, datos y programas.* Prentice Hall, 2001.

Devlin, K. *Why universities require computer science students to take math, Communications of ACM.* 2003.

Dijkstra, Wikipedia. *Dijkstra.* 1965.

Feierherd, Guillermo; Depetris, Beatriz; Jerez, Marcela. *Una Evaluación sobre la incorporación temprana de algorítmica y programación en el ingreso a Informática.* El Calafate, Santa Cruz: VII Congreso Argentino de Ciencias de la Computación, 2001.

Gálvez Rojas, S., y M. A. Mora Mata. *Java a tope: Traductores y compiladores con Lex/Yacc, JFlex/Cup y JavaCC.* Malaga, 2005.

Grune, D., y J.H. Ceriel. *Parsing Techniques, A Practical Guide.* Ithaca, Nueva York: Springer, 2008.

Iñesta Quereda, J. M. *Introducción a la programación con Pascal.* Univ. Jaume I, 2000.

Jeff Magee, Jeff Kramer. *Concurrency: state models & Java programs.* Wiley, 2006.

Joyanes Aguilar, L. *Fundamentos de la programación.* McGraw Hill, 2008.

Keith D. Cooper, Linda Torczon. *Engineering a Compiler.* Elsevier, 2003.

Lea, Douglas. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley Professional, 2000.

Louden, Kenneth C. *Construcción de compiladores. Principios y práctica.* Mexico: Thomsom, 2005.

—. *Lenguajes de programación: Principios y práctica.* Thomson, 2004.

Metsker, S. J. *Building Parsers with Java.* Addison Wesley, 2001.

Molina López, José Manuel. *Programación en lenguajes estructurados. Grado Superior.* España: McGraw Hill, 2006.

Pratt, Terrence W. y Zelkowitz, Marvin V. *Lenguajes de programación: Diseño e Implementación.* Mexico: Prentice Hall - 3ra Ed., 1998.

Schneider, Fred B. *On Concurrent Programming.* Springer, 1997.

Stallings, William. *Operating Systems: Internals and Design Principles.* Pearson College Division, 2011.

Steven John Metsker, William C. Wake. *Design Patterns in Java.* Addison-Wesley Professional, 2006.

Szpiniak, Ariel, y Guillermo Rojo. *Enseñanza de la programación.* TE & ET, 2006.

Tanenbaum, Andrew S. y Woodhoull AlbertS. *Operating Systems Design and Implementation.* 3ra. Mexico: Prentice Hall, 2009.

FACULTAD DE INGENIERIA

2012