Una experiencia en la enseñanza de teoría de lenguajes y compiladores

Jorge Aguirre
Ruta 8 Km. 603 UNRC FCEFQyN
Universidad Nacional de Río Cuarto
E-mail: jaguirre@reinfo.uba.ar

Marcelo Arroyo
Ruta 8 Km. 603 UNRC FCEFQyN
Universidad Nacional de Río Cuarto.
E-mail: marroyo@unrccc.edu.ar

Resumen: Se describe una experiencia realizada en la carrera de Licenciatura en Ciencias de Computación de la Universidad Nacional de Río Cuarto. En ella se fusionaron dos asignaturas del cuarto año: Autómatas y Lenguajes y Compiladores. Durante el primer período se impartió el contenido teórico de ambas mientras que en el segundo se realizo un de taller o laboratorio. Durante la segunda parte se realizó el diseño y la implementación de un compilador OCCAM .
Se describe el enfoque de la experiencia, los resultados obtenidos y la posición de los autores frente a la incorporación de talleres en los planes de estudio de Ciencias de la Computación.
El presente proyecto fue apoyado por la UNRC en el programa de proyectos pedagógicos innovadores.

Introducción.

El plan de estudios de la carrera, actualmente en proceso de modificación, no incluye la realización de talleres o laboratorios de diseño y desarrollo de software. No obstante, teniendo consenso en el cuerpo docente de la importancia de desarrollar la capacidad de diseño y de dotar a los alumnos de la habilidad instrumental necesaria para llevar a cabo implementaciones de cierta envergadura, se han realizado proyectos en distintas asignaturas. En estos proyectos los alumnos han tomado contacto con los lenguajes de programación y otras herramientas de desarrollo de software.

El plan de estudios incluye en cuarto año las asignaturas Autómatas y Lenguajes, en el primer cuatrimestre y Compiladores en el segundo. La experiencia que aquí se describe consistió en transformar ambas materias en una sola unidad pedagógica dictándose en la primera los contenidos teóricos de ambas materias y desarrollándose en la segunda un taller de diseño e implementación de compiladores donde se volcaron los conocimientos adquiridos.

Sobre los talleres o laboratorios

Los autores estamos convencidos de la eficiencia de los talleres para completar la formación de un profesional informático y facilitar su etapa de iniciación profesional.

Esta modalidad es utilizada tradicionalmente en otras áreas que también requieren el desarrollo de la capacidad de diseño del estudiante y de el desarrollo de su capacidad instrumental, el ejemplo paradigmático es la Arquitectura.

Estos talleres a nuestro entender deben estar destinados a resolver problemas, mediante la construcción de un adecuado producto de software. El problema debe ser seleccionado de manera tal que su solución abarque la mayor cantidad de los conocimientos adquiridos en las asignaturas ya cursadas. En este sentido se diferencian de la realización de trabajos prácticos de implementación concreta. Aquí la generalidad del problema a resolver exige la realización de un considerable trabajo de análisis y de síntesis para encontrar la modelización que permita utilizar los resultados generales que el alumno ha estudiado en el curso de su carrera sin que el contexto sugiera directamente apelar a determinado conocimiento o usar la técnica objeto de estudio en ese momento. Pensamos que esta capacidad solo puede desarrollarse en la práctica de la resolución de problemas de complejidad gradual. El docente debe guiar, criticar las soluciones y mostrar alternativas mejores, pero nunca debe desplazar al alumno del papel protagónico en el desarrollo del proyecto.

Estos talleres crean también el marco adecuado para el aprendizaje asistido del trabajo en grupo. En ellos puede abordarse la problemática que crea la modularización de un proyecto, el desarrollo paralelo por varias personas y la coordinación del trabajo del trabajo individual.

Es conveniente que las tareas realizadas durante los talleres cubran todas las fases del ciclo de desarrollo de software. Así mismo es importante que durante el taller se desarrolle la capacidad de elaborar documentos, cubriendo tanto reportes técnicos como documentación interna del sistema, destinada a su mantenimiento y manuales didácticos destinados al usuario final.

En nuestro país la ESLAI - Escuela Superior Latinoamericana de Informática - estructuró su plan de estudios incorporando un taller de las características mencionadas en cada cuatrimestre.

La Facultad de Ciencias Exactas y Naturales de la UBA, adoptó semejante temperamento incorporando ocho asignaturas, ubicadas también una en cada cuatrimestre, denominadas Laboratorios con idénticos objetivos.

Formación previa de los alumnos

Los alumnos que habrían de participar en esta experiencia contaban con los siguientes conocimientos y capacidades previamente adquiridos.

Conocimientos de diseño de algoritmos, tipos abstractos de datos y estructuras de información. Solvencia en el uso de PASCAL, que fue el primer lenguaje utilizado y sobre el que trabajaron los dos primeros años de la carrera. Conocimientos sobre los modelos de ejecución que requieren los distintos tipos de lenguajes de programación. Sobre alcance de variables, ligadura (binding), representación y verificación de tipos. Conocimientos básicos sobre Programación concurrente y los constructores de alto y bajo nivel usados. Somera práctica de uso de lenguaje C. Abundante práctica de uso de MS-DOS. Conocimientos básicos de los paradigmas de programación existentes.

Primera Parte

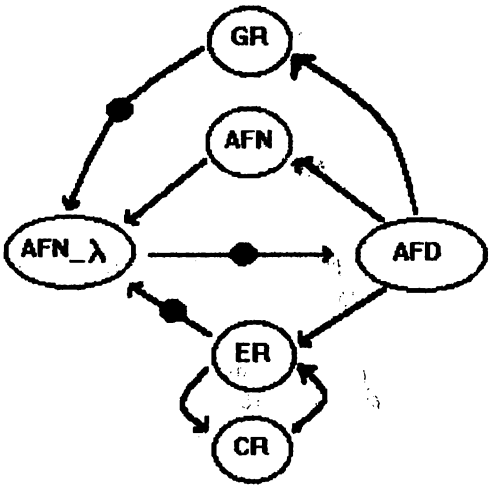
Objetivos

- Conocimientos básicos sobre la teoría de lenguajes y autómatas.
- Capacidad de trabajo con formalismos, de demostración de propiedades y obtención de conclusiones.
- Conocimientos de los algoritmos y técnicas usadas en el análisis sintáctico.
- Habilidad en la transformación de gramáticas y en la construcción de definiciones guiadas por sintaxis.
- Conocimiento de las herramientas y técnicas usadas en la construcción de compiladores.

Desarrollo

Dada la doble finalidad impuesta al curso, introducir la teoría formal por un lado, dar herramientas instrumentales por el otro, se tuvo especial cuidado en la selección de los caminos para seguir las distintas construcciones buscando que exhibieran algoritmos eficientes en los casos de utilidad práctica.

Así por ejemplo para ver la equivalencia entre las distintas definiciones de lenguajes regulares se siguió el siguiente circuito.



Para obtener las transformaciones correspondientes a los caminos marcados se dieron algoritmos eficientes. De esta manera la demostración constructiva de la equivalencia proveyó también de algoritmos utilizables en la construcción de distintas herramientas que actúan sobre lenguajes regulares.

La primera parte del curso siguió un cuidadoso desarrollo formal demostrándose, ya sea en clase o como ejercicios incluidos en los trabajos prácticos, cada uno de los resultados obtenidos. En la segunda parte se vieron las demostraciones más instructivas, aceptándose algunos resultados sin demostración, para permitir cumplir con el cronograma del curso.

Se puso especial énfasis en la construcción de soluciones usando definiciones guiadas por sintaxis, gramáticas de atributos y esquemas de traducción.

Como herramientas de construcción de analizadores léxicos y sintácticos se vieron lex y yacc y se estudió el diseño de productos similares.

El programa completo se muestra en el anexo I.

Segunda parte (taller de diseño de compiladores)

Objetivos

- Capacidad de aplicar los conocimientos adquiridos hasta el momento a la construcción de soluciones concretas.

- Capacidad autónoma de adquirir destreza en el uso de nuevos sistemas de desarrollo de software.

- Conocimiento preciso de constructores de concurrencia de alto y bajo nivel.
- Habilidad en el manejo del ambiente de desarrollo UNIX.
- Habilidad en el uso del lenguaje C.
- Capacidad de abordar el diseño de sistemas complejos.
- Capacidad de modularizar un proyecto y de trabajar en grupo.
- Habilidad en el uso de herramientas para la construcción de compiladores y trasladores.

El proyecto.

El proyecto a ser desarrollado en la etapa de taller debía presentar una problemática que permitiera cumplir con los objetivos planteados y a su vez tener un complejidad suficientemente reducida como para permitir su implementación en un cuatrimestre.

Se eligió como proyecto la implementación de un compilador OCCAM en al ambiente UNIX.
El lenguaje OCCAM también fue implementado en un cuarto taller en la ESLAI.

OCCAM es un lenguaje destinado a la programación concurrente en el cual los procesos solo pueden comunicarse a través de canales.

El lenguaje OCCAM presenta la ventaja de su simplicidad, la poca cantidad de constructores y la economía conceptual de su diseño. Así mismo la implementación de OCCAM supone resolver los problemas planteados por la ejecución concurrente de procesos y tratar problemas interesantes de verificación estática que permitan detectar el uso de mecanismos no permitidos de comunicación entre procesos.

En OCCAM las variables y los canales deben ser declarados explícitamente. Se pueden anidar procedimientos, el alcance es estático. El anidamiento de los distintos constructores se expresa directamente por la alineación del texto en distintos niveles de marginación (al igual que en GOFER), no existiendo paréntesis proposicionales (begin-end,{-}).

El constructor SEQ permite construir secuencias que serán ejecutadas en el orden de escritura.

El constructor PAR permite componer procedimientos que serán ejecutados concurrentemente.

El constructor ALT permite especificar una secuencia de comandos guardados, cuando alguna guarda sea satisfecha podrá ejecutarse el correspondiente proceso si más de una guarda se satisface uno solo, no determinado, de los procesos guardados se ejecuta.

Además OCCAM provee los constructores IF y WHILE.

Mediante el uso de replicadores "replicators" se puede especificar una cantidad variable de procesos.

Plan de trabajo

- Introducción al entorno de trabajo UNIX.
- Presentación del lenguaje OCCAM.
- Construcción de una gramática para OCCAM
- Construcción de un analizador lexicográfico especificado en lex.
- Construcción de un analizador sintáctico especificado en yacc.
- Estudio del modelo de tiempo de ejecución necesario para la implementación.
- Implementación de las primitivas de concurrencia necesarias.
- Implementación del analizador estático.
- Diseño del código a ser generado.
- Implementación del generador de código
- Planeamiento de la prueba del sistema (testing)
- Confección del lote de programas de pruebas.
- testeo y corrección de errores
- entrega de la documentación.
- Presentación y defensa del proyecto.

Desarrollo

La primera parte se desarrollo de acuerdo a lo previsto, pudiéndose completar el programa y realizar las guías de trabajos prácticos previstas.

Si bien la segunda parte terminó satisfactoriamente y los grupos entregaron productos funcionando, el desarrollo de la misma insumió más tiempo del previsto y debió reemplazarse el requisito original de generar código assembler por la posibilidad de generar código C.
Estos atrasos se debieron fundamentalmente a: dificultades en el tratamiento del anidamiento de constructores que complico el desarrollo del analizador léxico. Complicaciones en la eliminación de conflictos LALR, imposibilidad de conseguir la documentación del assembler de UNIX SCO, que fue el sistema operativo utilizado. Dificultades en la recuperación de errores bajo yacc.

Conclusiones

La experiencia permitió cumplir con los objetivos propuestos. Al respecto coinciden los alumnos que emitieron su opinión en una encuesta anónima. Solo uno de los encuestados opina que hubiera sido mejor mantener las dos asignaturas separadas.

Seria deseable poder llegar a generar código objeto, para ello debe reducirse la complejidad del proyecto.

Deben darse más herramientas para conseguir eliminar los errores LALR.

Debe darse una solución ya elaborada para resolver el problema de la indentación.

Consideramos que la experiencia merece repetirse.

BIBLIOGRAFÍA

- "Compiler construction". W. White - G. Goos. Springuer Verlag.
- "The Theory and Practice of Compilers Writing". J. Trembloy - P. Sorenson. Mc. Graw Hill. 1985.
- "Compiler Design in C". A. Holub. Prentice Hall. 1990.
- "Writing Compilers and Interpreters". R. Mak - John Wiley & Sons. 1991.
- "Compilers. Principles, Techniques and Tools". Aho - Sethi - Ullman. Adison Wesley. 1988.
- "The Design and Construction of Compilers". R. Hunter. John Wiley & Sons. 1981.
- "The implementation of functional programming Languages". Peynton Jones. Prentice Hall. 1987.
- "lex & yacc" J. Levine, T. Mason & D Brown. O'Reilly & Associates 1992
- "The design of the UNIX operating systems". M Bach. Prentice Hall 1986.
- "The UNIX operating systems" K. Christian. John Wiley & Sons 1983
- " The theory of Parsing Translation and Compiling". Aho, Ullman, Prentice Hall 1973.
- "The C programming language". Kernighan Ritchie. Prentice Hall 1988.
- "The UNIX programming environment". Kernighan & Pike. Prentice Hall 1984.
- "Introduction to authomata theory, languages and computation" Hopcroft Ullman . Addisson Wesley 1979.

Anexo I.

Programa de la primera parte.

Unidad I. Revisión de conceptos a ser usados. Relaciones, operaciones entre relaciones, representación mediante matrices booleanas, clausura transitiva, relaciones de equivalencia. Lenguajes, operaciónes sobre lenguajes. Gramáticas, relación de derivación, lenguaje generado. Clasificación de Chomsky, ejemplos. Lenguajes regulares, Autómatas Finitos Determinísticos y No deteminísticos, Autómatas No determinísticos con transiciones lambda, expresiones conjuntos regulares, demostración de la equivalencia de las distintas definiciones.

Unidad II. Lema de pumping para lenguajes regulares, estados accesibles, algoritmo de eliminación de estados inaccesibles. estados indistinguibles, relación de indistinguibilidad y de k'indistinguibilidad, propiedades. Autómata reducido, Construcción de un autómata reducido equivalente a uno dado, algoritmo y demostración de la equivalencia. Propiedades de clausura de los lenguajes regulares, decisión de los problemas básicos sobre lenguajes regulares. El análisis lexicográfico símbolos o tokens, patrones y lexemas. Algoritmo de generación de analizadores lexicográficos. Herramientas existentes lex, grep y awk.

Unidad III. Lenguajes independientes del contexto. Árbol de derivación derivación más hacia la derecha y más hacia la izquierda. Ambigüedad, gramáticas ambiguas y lenguajes intrínsecamente ambiguos. Lema de pumping para lenguajes independientes del contexto. Símbolos activos y alcanzables de una gramática, gramáticas reducidas. Símbolos anulables, obtención de una gramática propia equivalente a una dada. Autómatas Pila y Autómatas Pila Determiísticos. Otros modelos de autómatas, aceptación por pila vacía, equivalencia de los dos modelos. Equivalencia de los Autómatas Pila y la Gramáticas Independientes del Contexto. Propiedades de clausura de los lenguajes independientes del contexto. Análisis sintáctico ascendente y descendente. Parsing descendente recursivo. Computo de símbolos directrices, Gramáticas LL(1) y LL(k). Eliminación de recursividad a izquierda, factorización.
Método de análisis no recursivo

Unidad IV. Métodos ascendentes. Algoritmo de corrimiento y reducción, invariante del algoritmo. El problema de la determinación del pivote (handle). Somera idea de los métodos de precedencia simple y de operadores, construcción de las relaciones, algoritmos de reconocimiento, funciones de precedencia, computo y determinación de su existencia.
Gramáticas LR(k). Regularidad del conjunto de prefijos viables de un gramática independiente del contexto, ítemes válidos LR(0), construcción del autómata el lenguajes de prefijos viables. Construcción del conjunto canónico de ítemes LR(0), Gramáticas LR(0), algoritmo de reconocimiento. Gramáticas SLR(1). Gramáticas LR(1), ítem válido LR(1),Construcción del conjunto canónico de ítemes LR(1),Gramáticas LR(1), construcción del conjunto canónico de ítemes LALR(1) a partir del LR(1). Herramientas de generación de analizadores sintácticos (utilización de yacc)

Unidad V. Gramáticas de atributos. Atributos sintetizados y heredados. Árbol decorado de una sentencia, evaluación, grafo de dependencia, gramáticas evaluables. Evaluación correlativa al parsing, gramáticas de atributos S y L . Construcción de evaluadores descendente recursivos. Evaluación durante el parsing de corrimiento y reducción emulación de herencia mediante símbolos demarcadores. Definiciones guiadas por sintaxis. Esquemas de traducción. Construcción de arboles sintácticos y grafos dirigidos acíclicos.

Unidad VI. Resolución del problema del alcance de las declaraciones. Modelos del ambiente de tiempo de ejecución, revisión de los mecanismos utilizados para resolver la ligadura (binding). Generación de código intermedio, finalidad de la inclusión de esta etapa. Uso de grafos, árboles sintácticos, grafos dirigidos acíclicos. Código de tres direcciones, repertorio de instrucciones y formas de representación - cuaternas, ternas y ternas indirectas -. Generación de código intermedio para las constructores típicos de un lenguaje de programación. Generación de código objeto, criterios de optimización. Información sobre tiempo de vida, uso y ubicación de variables, construcción de un algoritmo simple de generación de código. Notación de Bratman y problemas de implementación (bootstraping).

ANEXO II

EL LENGUAJE OCCAM

Procesos Primitivos: Existen tres procesos primitivos o básicos que son:

1. Proceso de asignación: Transfiere el valor de su expresión a la variable referenciada.

   variable := expresión
   variable[expresión]=expresión

2. Proceso de entrada: Copia el valor desde un canal en una variable.

   channel ? variable
   channel ? ANY

3. Proceso de salida: Copia el valor de la expresión al canal.

   channel ! variable
   channel ! ANY

Estas tres primitivas se pueden combinar secuencialmente o concurrentemente para crear procesos más complejos y ellas forman bloques básicos para los programas. Además se dispone de las siguientes primitivas auxiliares.

1. SKIP : Termina sin otro efecto.

2. WAIT NOW AFTER time: Suspende la ejecución hasta que el período de tiempo haya pasado. NOW retorna el valor del reloj de la máquina local.

Declaraciones: Las declaraciones de variables , constantes y canales preceden al proceso al que pertenecen (estos identificadores son locales al proceso que continúa inmediatamente). Cada declaración es introducida por una palabra clave (como VAR), el tipo de dato ,una lista de identificadores (conteniendo el número de elementos en caso de ser un arreglo) y finaliza con un dos puntos (cuyo significado es la asignación al siguiente proceso) como en el siguiente ejemplo:

VAR INT A,B:	*(Declara las variables enteras A y B)*
VAR INT R[5]:	*(Declara un arreglo de 5 elementos enteros)*
CHAN C:	*(Declara el canal C)*
DEF max=10, min=1:	*(Declara dos constantes max y min)*
DEF s="abcd":	*(Arreglo constante de 4 elementos de tipo CHAR)*
DEF table=[1,2,3,4,5,6,7,8,9,0]	*(Vector constante de 10 elementos enteros)*

Las declaraciones de constantes no necesitan del tipo de dato ya que lo determina el valor asignado. Los canales son "sin tipo", quedando a cargo del programador la consistencia de valores.

Tipos de datos: Las variables se deben declarar con un tipo determinado . Los tipos de datos que tenemos en Occam son:

INT : Tipo entero.
CHAR : Tipo caracter.
BOOL : Tipo lógico (toma valores TRUE o FALSE)

Constructores: Un constructor permite definir un proceso a partir de otros procesos más simples. En Occam tenemos constructores que nos permiten construir una secuencia de procesos (SEQ), procesos que se ejecutan concurrentemente (PAR), lecturas alternativas, es decir, lista de lecturas, cada una de las cuales tiene asociada una expresión o guarda que habilita su ejecución (ALT) y otros constructores que nos permiten definir iteraciones (WHILE) y selecciones de procesos en base a condiciones (IF).

SEQ: El constructor SEQ permite definir una secuencia de procesos. Los procesos componentes se ejecutan secuencialmente. Se utiliza la indentación para definir el alcance de un constructor. Los procesos componentes de un constructor se encuentran generalmente indentados dos espacios).

Ejemplo:

```
VAR INT x:
CHAN c:
PAR
  c ? x
  x=x+1
```

WHILE: El constructor WHILE permite elaborar procesos repetitivos. Los procesos componentes se ejecutan mientras la condición del cnstructor WHILE evalúe a TRUE.

Ejemplo:

```
VAR INT x, y:
CHAN c:
SEQ
  x=1
  WHILE x>0
    SEQ
      c ? y
      x=y-1
```

Replicadores: Un replicador se utiliza con un constructor para replicar los procesos componentes un determinado número de veces.

Ejemplo:
```
DEF a="ABCDE"
SEQ i= [ 1 FOR 5 ]
   out1 ! a[i]
```

IF: Este constructor permite seleccionar uno de los procesos componentes condicionales. Cada proceso componente está precedido por una condición o guarda. Un proceso condicional ejecuta el primer proceso componente (textualmente) si su expresión correspondiente evalúa a TRUE. Un proceso condicional el cual contiene otros procesos condicionales componentes se ejecuta si uno de sus procesos condicionales componentes se puede ejecutar. En el caso que ninguno de sus procesos componentes se pueda ejecutar el constructor IF termina sin ningún efecto (actúa como en el caso que no tenga procesos componentes).

Ejemplo:
```
IF
  i=1
     out1 ! x
  i=2
      out2 ! x
```

PAR: Es un constructor para definir procesos que se ejecutan en paralelo. La palabra clave PAR es seguida por un número de procesos componentes. El efecto es que se ejecutan todos los procesos componentes en concurrentemente y el constructor termina cuando todos los procesos componentes terminan. Si no existen procesos componentes el constructor termina inmediatamente.

Dos procesos componentes de un constructor PAR se pueden comunicar enviándose valores usando canales pero no a través de variables. Para eso establecen una comunicación, un proceso realiza salidas por un canal mientras otro realiza entradas por el mismo canal. Ningún otro proceso puede usar ese canal. La comunicación es sincrónica, o sea, el proceso que escribe espera que otro haya leído el valor emitido.

Una variable visible por todos los procesos componentes, puede ser usada por los procesos componentes pero no la pueden modificar (por input o asignación). Las reglas reguladoras del uso de variables no siempre pueden ser chequeados, particularmente en el caso de operaciones con índices en arreglos. El criterio adoptado fue realizar todas las verificaciones estáticas posibles y emitir advertencias cuando es menester una verificación en tiempo de ejecución.

Ejemplo:
```
CHAN comms:
PAR
  WHILE TRUE
    VAR x:
    SEQ
      c1 ? x
      comms ! x
  WHILE TRUE
    VAR x:
    SEQ
      comms ? x
      c2 ! x
```

ALT: Un proceso alternativo espera hasta que al menos una de las guardas que preceden a cada proceso componente esté lista para ejecutar. Un proceso (componente) con guarda que comienza con un input desde un canal está listo si otro proceso está esperando por una salida en el canal. Si el proceso con guarda es el seleccionado, se realiza el input y el proceso componente se ejecuta.

Un proceso con guarda que comienza con un WAIT está listo para ser ejecutado si el WAIT está listo. Un proceso con guarda que comienza con un SKIP está siempre listo. Si una guarda comienza con una expresión seguida por un input o WAIT está lista si la expresión arroja valor TRUE y el input o WAIT está listo.

Si un preoceso con guarda es un constructor ALT, entonces está listo si uno o más de sus procesos con guarda componentes está listo.

Si más de un proceso componente está listo, se selecciona arbitrariamente uno de ellos y se ejecuta.

Ejemplo:
```
WHILE TRUE
  VAR x:
  ALT
    c1 ? x
      c3 ! x
    c2 ? x
      c3 ! x
```

PROCESOS CON NOMBRE: Se puede asignar un nombre a una porción de texto de un proceso. Se puede pensar que un proceso con nombre es una definición de una macro expansión. El texto se sustituye (expande) por todas las ocurrencias del nombre en cada referencia. Las variables y canales se pueden usar definendo parámetros que se substituyen textualmente.

Los parámetros formales se declaran de la misma manera que en una declaración de variables, canales o constantes (VALUE). El el caso que un parámetro formal corresponda a un vector no se especifica el tamaño entre los corchetes.

En el caso de un parámetro formal VALUE (constante), el parámetro actual debe corresponder a una expresión, la cual debe ser evaluada entes de hacerse la substitución.

```
Ejemplo:PROC buffer(CHAN in, out) =
          WHILE TRUE
             VAR x:
             SEQ
                in ? x
                out ! x :

       CHAN c:
       PAR
          buffer(c1,c)
          buffer(c,c2)
```

ESTRUCTURA DE UN PROGRAMA OCCAM

Occam utiliza la indentación para indicar la estructura del programa. Cada proceso comienza en una línea al nivel de indentación determinada por las siguientes reglas:

Constructores: La palabra clave del constructor y un replicador opcional ocupa la primer línea. Cada uno de los procesos componentes comienzan en una nueva línea y están identados por dos espacios más hacia la derecha.

Procesos con guardas: La expresión y/o input o wait ocupa la primer línea. Cada uno de los procesos componentes comienza en la línea siguiente indentado dos o más espacios.

Procesos condicionales: La expresión ocupa la primer línea. Los procesos componentes comienzan en la línea siguiente, indentado dos o más espacios.

Declaraciones: Cada declaración comienza en una nueva línea, al mismo nivel de indentación del proceso que le sigue inmediatamente (el cual es afectado por la declaración). La línea final de una declaración termina con dos puntos (:).

Comentarios: Los comentarios son comenzados por un doble guión (--) y termina en el final de la línea.