INGENIERIA DE SOFTWARE Y PROLOG

F. Javier Diaz

U.N.L.P.
y
E.S.L.A.I.
Cno. Gral. Belgrano Km 40
C.C. 3193
1000 Buenos Aires
Argentina

Resumen:

El presente trabajo discute las posibilidades de cumplir con los objetivos principales de la Ingenieria de Software utilizando el lenguaje Prolog. La idea central es que un buen lenguaje de programación para poder ser utilizado en toda su potencia necesita de un conjunto de herramientas que simplifiquen la tarea del diseñador.

Se describe en primer término el concepto "ambiente de programación", y posteriormente se lo analiza (con parámetros de la Ingenieria de Software) en función del lenguaje de programación Prolog.

Este trabajo se ha realizado en el marco del proyecto "Ambiente de Programación Inferencial" (API), financiado por el Programa Nacional de Informática y Electrónica de la SeCyT, Argentina (nro. 537-001).

I. Què significa "ambiente de programaciòn" ?

A medida que crece la complejidad de los problemas que se trata de resolver utilizando la computadora aparecen nuevas fuentes de dificultades para el desarrollo y la puesta a punto de los sistemas de procesamiento de datos. Usualmente se referencian estos problemas como "programming in the large" para diferenciarlos de la algoritmica tradicional, que se denomina "programming in the small" (GHE 87). Para atacar tales problemas se pensó que lo mejor era la utilizaciòn de principios de ingenieria para el dasarrollo de sistema complejos. De esta forma surgió la Ingenieria del Software. La misma trata principalmente los problemas vinculados con la disminuciòn de la complejidad de los problemas utiliando tècnicas de descomposiciòn y empleando metodologias que faciliten la elaboraciòn de las distintas partes, y su posterior integraciòn.

Resulta casi imposible pensar en atacar problemas reales de cierta magnitud sin hacer uso de ciertos principios fundamentales. Desde mediados de la dècada del 70 surgen los "ambientes para desarrollo de software" como una herramienta indispensable para englobar los distintos conceptos que conducen a la elaboraciòn y puesta a punto de sistemas (BAR 86). A continuaciòn se trata de caracterizar el concepto de "ambientes" de forma tal de tener parámetros concretos en funciòn de los cuales se pueda medir la calidad de los mismos.

I.1. Definiciòn tentativa de ambientes

Usualmente se entiende por "ambientes de programaciòn" un conjunto integrado de herramientas que posibilitan al programador cumplir con todas las etapas del ciclo de desarrollo de software utilizando un sistema de notaciòn homogèneo y sin necesidad de cambiar de contexto segùn la funciòn que desee realizar.

Usualmente en las etapas de desarrollo de software se empleaban herramientas desarrolladas ad-hoc y que, por tanto, respondian a una visiòn demasiado especifica del tema. Esto obligaba al diseñador de software a tener que manejar simultaneamente los objetivos (y lenguajes especificos) de quien desarrollò cada una de tales herramientas. De esta forma se obligaba al usuario a tener que realizar una serie de pasos, con frecuencia obvios, y aùn redundantes, los que distraian su atenciòn del objetivo final de su trabajo.

I.1.1. Definición general de "ambientes de programación".

Si bien es difícil dar una definición rigurosa que permita identificar, con total certeza, que es un ambiente de programación, existe concenso en el siguiente sentido, un sistema computarizado puede considerarse un ambiente de programación si reune las características siguientes:

I.1.1.1. Está compuesto por un conjunto de herramientas.

Esto significa que el sistema auna varias visiones distintas (cada una provista por alguna/s herramienta/s).

I.1.1.2. Se puede tener una visión integrada del mismo.

De este modo se permite una visión unificada del sistema fusionándose tareas que anteriormente eran consideradas por separado. Por otra parte se eliminan las jerarquías forzadas que dificultan la comprensión del sistema y obstaculizan su mantenimiento (por ejemplo haciendo más difícil el cambio de un cierto módulo).

I.1.1.3. Se simplifica su uso.

Se trata de que el sistema reaccione más inteligentemente, a fin de disminuir los requerimientos para ser usado efectivamente por el usuario. Esta característica es fundamental. La idea que predomina es la de evitar al usuario la codificación de una serie de pasos, cuando los mismos sean previsibles. Por ejemplo es claro que si se produjo un error de sintaxis en la compilación lo siguiente que hará el usuario es editar la línea donde se detectó el inconveniente.
Otro objetivo es simplificar la tarea del usuario, empleando una cantidad mínima de comandos -con una sintaxis cómoda-, a fin de hacer más breve el período de aprendizaje, y sencilla la comprensión de las tareas que realiza.

I.1.1.4. El nivel de interacción debe ser alto.

La necesidad de que el sistema "colabore" con el usuario en las tareas que este realiza, son el mejor indicador del tipo de comunicación que se requiere. Para comprender mejor este punto basta evaluar la gran aceptación de los sistemas basados en íconos, la aplicación de "mouse" o de "joystick", etc. De esta forma las capacidades del hombre y de la máquina pueden aunarse, sin que una forma de comunicación demasiado rígida y poco flexible obstaculize el proceso.

I.1.1.5. El sistema debe se completo.

No debe ser necesario salir del sistema para realizar alguna tarea que coadyude al propósito del mismo. Es

significa que el sistema debe prever todas las posibilidades. De esta forma se logra que con la utilización de un único medio se resuelvan todos los problemas asociados con determinado fin.

I.1.2. Ejemplos de componentes en un "ambiente de programación"

Para fijar ideas es conveniente repasar las componentes básicas para desarrollar software. Debe considerarse que si bien la enumeración dada parece adecuarse más bien a un enfoque clásico del desarrollo de software las actividades que las mismas cumplen están incluidas en los objetivos para la integración de un sistema complejo.

1) Editor de texto
2) Macroprocesadores o preprocesadores
3) Intérprete / compilador
4) Sistema de manejo de archivos
5) Editor de enlaces
6) Analizadores dinámicos de frecuencias
7) Optimizadores de programas fuente
8) Sistemas para depuración de programas/sistemas

I.1.3. Objetivos de los "ambientes de programación"

Por último mencionaremos los objetivos más generales a los que tienden los diseñadores de este tipo de sistemas.

I.1.3.1. Facilitar la tarea del programador

Se trata de proveer una herramienta lo más flexible posible para cubrir una gama de aplicaciones. No sólo se trata de hacer más simple la tarea (evitando pasos innecesarios) sino de brindarle al usuario una mayor gama de opciones de las que estaban disponibles hasta ese momento.

En este punto, es fundamental definir qué se entiende por "tarea del programador". Si bien, en general se asocia tal frase a todas las etapas de desarrollo de software, las mismas están condicionadas al modelo de ciclo de vida que se utilice. A continuación se da una lista de las tareas que se consideran de fundamental importancia:

i) **edición** (del/de los programas y documentación) Etapa de entrar la información en el sistema.

ii) **demostración de corrección**, verificación formal de que el programa responde a sus especificaciones.

iii) **depuración** o puesta a punto del programa, está vinculado a la adecuación del sistema realizado respecto de la realidad (o problema real con el que se enfrenta el usuario final).

iv) **documentación**, vinculado a tener facilidades tanto para mantener simultaneamente programas y texto explicativo de los mismos, como con la facilidad de deducir alguna documentación de los programas mismos.

v) **optimización** del sistema resultante, vinculada a la posibilidad de aplicar transformaciones al código que permitan mejorar la performance del sistema

vi) **mantenimiento de los sistemas**, si bien se la puede pensar como una síntesis de las otras tareas, tiene una problemática propia asociada a la facilidad de incorporar modificaciones (para adecuar al sistema a circunstancias cambiantes).

vii) **reusabilidad de código**, está relacionado con la posibilidad de adaptar las soluciones a diversos problemas que se resolvieron con anterioridad sin necesidad de una recodificación de los correspondientes fragmentos de código. Tal reusabilidad puede verse de dos ángulos, como la posibilidad de usar facilidades que brinda el sistema de soporte básico o poder re-utilizar fácilmente (y sin conflictos) código que anteriormente especificó el mismo usuario.

I.1.3.2. Incrementar la "inteligencia" del Sistema.
La tendencia actual es la programación auxiliada o asistida por computadora. Se pretende eliminar la tarea no creativa del programador. Por otra parte el sistema reacciona en forma variable según sea nivel de experiencia del usuario.

I.1.3.3. Disminuir la complejidad de los sistemas resultantes.
El objetivo es que las herramientas que estan ahora disponibles permiten intentar resolver problemas cuya complejidad los hacia intratables computacionalmente empleando los sistemas más clásicos.

I.1.3.4. Brindar un nexo más cordial con el usuario.
La característica saliente de estos sistemas es que su uso es agradable y mantienen una comunicación fluida utilisando la menor cantidad de convenciones (artificiales) que sea posible.

I.2. Objetivos del diseño de ambientes que afectan a las características de un lenguaje de programación.

La característica saliente de un ambiente de programación es que el lenguaje que debe emplear el usuario es homogéneo. Esto implica que debe incluir opciones para cubrir todas las necesidades que surjan. De este modo los lenguajes de programación más modernos incluyen opciones para la puesta punto, manejo de interrupciones, concurrencia, formas de encapsular módulos y de proveer una comunicación estandar. En particular los objetivos perseguidos pueden resumirse en los dos puntos siguientes.

I.2.1. Permitir su uso por no especialistas.
En particular si bien esto no puede cumplirse a extremo (ya que siempre es necesario algún grado de entrenamiento) la tendencia es que el lenguaje que se provee pueda definirse (y emplearse) por subconjuntos crecientes. Tales subconjuntos admiten una definición formal y cubren las necesidades de cierto tipo de aplicaciones. A medida que la complejidad de los proyectos que se pretenden resolver crece puede hacerse uso de herramientas más sofistificadas. Lo importante es que no fue necesario conocer las mismas hasta ese momento, y que al estar el lenguaje definido en forma incremental, la comprensión de la nueva estructura no afecta a lo que ya se conocia.

I.2.2. Brindar facilidades para el desarrollo de aplicaciones de gran envargadura.
Este punto es de fundamental importancia, por cuanto hace a que el lenguaje diseñado cumpla con los requerimientos de la Ingeniería de Software. Es bien conocido como cambia la forma de programar (desde el diseño al mantenimiento) de los sistemas de gran envargadura respecto de los estandares establecidos para la programación en pequeña escala (algoritmica). En este sentido pasan a ser prioritarios conceptos de seguridad. En particular surgen claramente el concepto de tipos (como forma de ocultar información y de evitar usos indebidos o no previstos de la misma) y el concepto de módulos (como forma de ocultar comportamiento y permitir la más fácil localización de los efectos laterales que ocasione el cambio de una parte del programa). Por otra parte, no es posible pensar en una ingeniería de sistemas de software sin poder reusar componentes anteriormente desarrolladas (pero

lo cual es imprescindible disponer de formas lo más general posible de implementar los conceptos arriba descriptos).

II. POSIBILIDADES DE PROLOG PARA DEFINIR UN AMBIENTE DE PROGRAMACION.

Es fundamental hacer un análisis de las características del lenguaje de soporte de un ambiente, con el fin de poder determinar hasta que punto es posible concretar los objetivos perseguidos. Con este porpòsito se enumeran a continuación los conceptos que deben estar presentes en un ambiente de desarrollo de software. En particular se analiza las posibilidades de Prolog (CLO 84, LLO 84, KOW 86), sobreentendièndose que, cuando no se lo mencione explícitamente, Prolog cumple con los objetivos enunciados.

II.1. Alto nivel de interacciòn.

Es claro que para proveer una interfase hombre/máquina razonable, es conveniente que el sistema subyacente (en el que se implementarà nuestro sistema) sea altamente interactivo. En particular esto permite que el sistema no sea estàtico y se pueda adaptar a circunstancias (y usuarios) cambiantes.

II.2. Sintaxis comùn de las primitivas del sistema y los predicados definidos por el usuario.

Esta caracterìstica, que fue mencionada anteriormente cuando se habló de que el lenguaje fuese incremental, es fundamental si se trata de que el mismo sea homogéneo y simple.

II.3. Posibilidad de amalgamar datos y programas.

Esto significa que es posible regenerar el kernel de intèrprete, modificar programas y ampliar al sistema dinàmicamente. De esta forma el sistema podrà incorporar aquella informaciòn que considere necesaria y usarla como una extensiòn del ambiente predefinido. De esta forma, los programa entrados por el usuario, pueden ser transformados antes de ser manipulados por el sistema.

II.4. La existencia de una màquina de inferencias.

Prolog provee en su nùcleo una implementaciòn del mètodo de resoluciòn linear. De esta forma, el sistema siempre responderà segùn los axiomas lògicos que constituyen el programa. Es decir, no es necesario demostrar que el comportamiento del programa es el indicado en las especificaciones, sino que esto serà una consecuencia directa de trabajar en Prolog. (Sin embargo debe tenerse en cuenta que el sistema no valida la consistencia de los axiomas cuando se los

odifica, es decir pueden existir clausulas redundantes, y aun contradictorias).

II.5. Síntesis de programas.
Existen varios trabajos relativos a la posibilidad de sintetizar programas a partir de ejemplos y de derivar programas a partir de especificaciones en lógica.

II.6. Poseer una sintaxis natural.
La sintaxis de Prolog se asemeja a la de la lógica clausal de primer orden. Por su parte, la notación de la lógica matemática fue creada para poder modelizar con propiedad (eliminando ambiguedades) el lenguaje natural. De modo, que si bien difiere de este último, parece ser el formalismo que mejor se adapta a su representación. Por otra parte, es posible agregar módulos que realizan la conversión a Prolog de subconjuntos del lenguaje natural.

III. DESVENTAJAS DE PROLOG PURO desde el punto de vista de la Ingenieria de Software.

Ya se destacó anteriormente que un lenguaje de programación debe incorporar ciertos conceptos a fin de poder ser empleado con éxito en el desarrollo de sistemas de gran envergadura (BOB 85). Mas aún cuando la aplicación es crítica (es decir cuando las fallas no son tolerables) o el usuario final es quien usa libremente el sistema desarrollado, se hace imprescindible la incorporación al lenguaje de construcciones que permitan controlar tales situaciones (FRA 86).

III.1. Lenguaje chato.
No existen jerarquías en la definición de un programa. Si uno considera los lenguajes tradicionales, es común la existencia de bloques (o unidades de programa) que pueden anidarse y que proveen un medio para determinar el alcance de las variables (ya sean datos o nombres de rutinas).
Si bien puede considerarse que todas las claúsulas que definen determinado predicado constituyen la definición de un procedimiento (al cual se lo puede considerar como no-determinístico respecto de la acción que efectivamente realiza) se debe destacar que se puede referenciar a cualquier claúsula del programa. De esta forma no es posible, por ejemplo, definir un predicado padre que sólo sea accesible (o utilizable) desde el predicado hermano.

III.2. La estructura de datos si bien tienen un gran poder de representación no provee selectores built-in.

A pesar de que la existencia de tales selectores causaría que coexistan functores y funciones (ya que los selectores actuarían como tales) en la definición de los términos, existirían ventajas de expresividad (pues nos evitaríamos codificaciones engorrosas y aún la definición de predicados definidos ad-hoc)
Por otra parte también sería afectada la eficiencia en la manipulación de las estructuras de datos.

III.3. La no existencia de tipos.
Si bien la genericidad de los argumentos de los predicados provee una gran flexibilidad en el uso de los mismos. La falta de seguridades respecto del sistema de tipos hace que se puedan producir errores muy difíciles de detectar. En particular, tales seguridades son obvias si el usuario final puede ser una persona sin experiencia.
Algunos chequeos de tipos pueden realizarse si el programador tiene la disciplina de asociar distintos functores a datos de distinta naturaleza, y si la cabeza de cada predicado tiene previsto con qué functores hace matching. Sin embargo no parece aconsejable que todo el peso de utilizar tipos recaiga en el programador quien ve complicada su forma de codificación.
Por último, es factible pensar en extensiones del proceso de unificación, provisto por Prolog, que podrían incluir un chequeo de tipos.

III.4. No existen facilidades de debugging.
Considerado como un sistema de demostración de teoremas que simplemente busca inconsistencias, parece razonable que no incluya tales facilidades. Sin embargo, si consideramos Prolog como un Lenguaje de Programación (es decir un lenguaje en el cual se expresan modelos que se esperan reflejen el comportamiento de algún sistema dado) es claro que interesa saber el porque de su comportamiento, es decir saber si algún predicado esta mal definido, o si la secuencia de matchings que realiza no corresponde con lo que se deseaba expresar.

III.5. No existen facilidades para la modificación de los programas.
Por ejemplo cada vez que se corrige un programa de algún modo se está modificando la definición de un procedimiento. De este modo debería poder verse qué otras definiciones existen (de forma de estar seguro que la modificación introducida es compatible con las demás definiciones que forman un procedimiento). Tampoco existe la posibilidad de ver que cláusulas son las que lo invocan, a fin de ver si la modificación que se introduce puede tener algún efecto no deseado.

III.6. No existen variables globales
Las variables que se utilizan en la definición de cada cláusula son de efecto local a la misma. Esto se debe a que puede considerarse que cada predicado está afectado de tantos cuantificadores universales como variables libres posee

De esta forma si bien se favorece el principio de "localidad de efectos" y de "granularidad", es imposible usar variables no locales a una claùsula. Esto es particularmente molesto cuando en realidad un procedimiento està formado por todas las claùsulas que definen un predicado. De esta forma no se puede tener una variable que tenga alcance en todo un procedimiento. Por otra parte para simular el efecto de una variable global se deberia implementar un predicado cuyo argumento represente el valor corriente de esa variable (lo que implica que, para reflejar el comportaiento tradicional de una variable, se deberian usar caracteristicas extralògicas como agregar y borrar claùsulas)

III.7. No es posible especificar estrategias de busqueda alternativas

Si bien la programaciòn en lògica posee la caracteristica de evitar que el programador tenga que preocuparse del control, las implementaciones por una cuestiòn de eficiencia, implementan un mètodo de resoluciòn que no es completo. Es decir, el sistema puede no responder aùn cuando existan soluciones encontrables. De esta forma, parece ser conveniente permitirle que el programador especifique alguna forma de control. Una posibilidad seria indicar que para determinadas deducciones usarà una estrategia de control (distinta de depth-first-search) provista ex-profeso por el usuario.

IV. CONCLUSIONES

El presente trabajo muestra que existen posibilidades ciertas de definir extensiones al lenguaje de Programaciòn Prolog de forma tal de poder satisfacer los requerimientos de la Ingenieria de Software. Sin embargo las mismas deben respetar las caracteristicas de diseño del lenguaje, es decir, la cantidad de primitivas que se agregen debe ser minima y su introducciòn estar destinada a la solución de alguno de los problemas planteados en el punto III.

V. BIBLIOGRAFIA

BAR 86: BARSTOW, SHROBE & SANDEWALL: Interactive Programming Environments. Morgan Kaufman 86.

BRA 86: BRATKO: "Artificial Intelligence Programming in Prolog" Addison Wesley 86.

BOB 85: BOBROW: "If Prolog is the answer, what is the question" IEEE Computer, Sept 85.

CLO 84: CLOCKSIN & MELISH: Programming in Prolog. Springer Verlag 84.

GHE 86: GHEZZI & YAZAJERI: Programming Language Concepts John Wiley, 2nd ed, 1986.

KOW 86: KOWALSKI: "Logic Programming" Imperial College, marzo 1986.

LLO 84: LLOYD: "Foundations of Logic Programming" Springer Verlag 1984.