Control implícito y explícito de secuencias.

Las estructuras de secuencias se pueden clasificar en 3 grupos:

- Estructuras que se usan en expresiones, como reglas de precedencia y paréntesis;
- Estructuras que se usan entre enunciados o grupos de enunciados, como enunciados condicionales e iterativos; y
- Estructuras que se usan entre subprogramas, como llamadas de subprogramas y corrutinas.

Las estructuras de control de secuencias pueden ser implícitas o explícitas. Las estructuras de control de secuencias implícitas (o por omisión) son las que el lenguaje define que están en operación, a menos que sea modificado por alguna estructura explícita. Dentro de estas expresiones existen una jerarquía de operación definidas por el lenguaje, que controla el orden de ejecución de las operaciones de la expresión cuando no existe paréntesis.

Las estructuras explícitas de control de secuencias son las que el programa puede optar para modificar el orden implícito definido por el lenguaje; ejemplo, usando paréntesis dentro de las expresiones y etiquetas de enunciados.

Secuenciamiento con expresiones aritméticas.

Consideremos la formula siguiente:

Raiz= −b ± " b2 − 4×a×c

2×a

Un lenguaje ensamblador o de maquina requiere al menos 15 instrucciones para obtener el resultado. Sin embargo el lenguaje FORTRAN, puede codificarlo en una sola expresión.

ROOT= (−b + SQRT (b**2−4*a*c)) / (2*a)

Estas expresiones son un dispositivo poderoso y natural para representar series de operaciones aunque se plantean nuevos problemas, el programador entiende el orden de las operaciones pero ¿qué hay con las expresiones?, en la expresión en FORTRAN es correcta, pero como saber que la sustracción tiene lugar después de la multiplicación (b**2−4*a*c), los mecanismo de control de secuencias que determinan el orden de las operaciones son bastantes complejas y sutiles

Figura 1. Representación de árbol de la formula cuadratura

Representación de estructura de árbol.

Las expresiones se han considerado como entidades únicas, sin tomar en cuenta la sintaxis de una expresión dada, al considerar las operaciones dentro de una expresión, llamaremos OPERANDOS a los argumentos de una operación.

El mecanismo del control de secuencias en expresiones es la composición funcional, se especifica una operación y sus operandos. La composición final confiere a la expresión la estructura característica de un árbol, donde la raíz es la operación principal y las hojas los referentes de datos.

La representación de árbol aclara la estructura de control de la expresión. Los niveles más bajos en el árbol sirven como operandos para operaciones a nivel mas altos en el mismo, estas referencias de datos y operaciones se deben evaluar primero.

Figura 2. Forma de árbol para la expresión simple: (a+b)×(c−a).

En la fig 1 no esta claro si −b se debe evaluar antes o después de b**2, ni tampoco si b se puede conbinarse en una sola referencia. En una definición de lenguaje; es común definir el orden de evaluación de la expresión sólo al nivel de la representación de árbol y permitir que el implementador del lenguaje decida el orden de evaluación en detalle.

Antes de examinar el problema es apropiado chequear la representación sintáctica para expresiones que están en uso.

Sintaxis para expresiones.

Si las expresiones son representadas característicamente por arboles, entonces, usar expresiones dentro de programas, sé requierelinealizar los arboles, es decir, se debe disponer de notaciones para escribir los arboles como series lineales de símbolos.

Notación prefija(polaca prefija): Se escribe primero el símbolo de la operación seguido de operandos en orden de izquierda a derecha, si un operando a su ves es una operación con operandos. Se aplica el mismo criterio, en la figura anterior, el árbol se convierte en x + a b − c a. Puesto que el signo + es un operador diádico, los argumentos para + son a y b, y similar los argumentos para x son +y−, no se necesita paréntesis para especificar como evaluar la expresión.

Una variante es el LISP(designada como polaca de Cambridge). En esta, un operador y sus argumentos están rodeados por paréntesis. Una expresión se ve, por tanto, como un conjunto anidado de lista, donde cada lista comienza con un símbolo de operandos. En polaca de Cambridge la fig. anterior se convierte en:

(x(+ab)(−ca))

Considerando la formula de la figura 1 en notación prefija (usando ! para exponenciación, " para sqrt y _ para menos unaria)

/+_b"−!b2xx4acx2a (polaca)

(/(+(_b)("(−(!b2)(x(x4a)c))))(x2a)) (polaca de Cambridge)

Notación posfija(sufija o polaca inversa): Esta notación es similar a la notación prefija excepto que el símbolo de operación sigue a la lista de operandos. Ej la figura anterior se representa como ab+ca−x.

Notación infija: esta notación es mas adecuada para operaciones binarias, en esta notación el símbolo operador se escribe entre dos operandos, la notación para estas operaciones ha sido adaptada en lenguaje de programación. La forma infija para el árbol de la figura anterior es (a+b)x(c−a).

Semántica para expresiones.

Cada una de las 3 notaciones prefija, posfija e infija, tiene cierto atributo que son útiles en el diseño de lenguaje de programación. Difieren en la manera de calcular el valor para cada expresión. Proporcionan algoritmos para evaluar cada formato de expresión, para hacer más fácil la evaluación de expresiones.

Evaluación prefija: Con notación prefija se puede evaluar cada expresión en un solo examen, sin embargo es necesario conocer el numero de argumentos para cada operación, por esto es necesario utilizar signos especiales para sustracciones diadicas(−) y menos unarias(_), para distinguir las operaciones.

Además del ahorro de paréntesis, tiene un valor en el diseño de lenguaje de programación.

1.− La llamada de función usual ya está escrita en notación prefija.

2.− La notación prefija se puede usar para representar operaciones con cualquier numero de operandos y es, por tanto, completamente general

3.− Es relativamente fácil decodificar mecánicamente; por esto se consigue con facilidad expresiones prefijas a secuencias de código simple.

Evaluación posfija: en esta notación el operador sigue a sus operandos, cuando se examina un operador sus operandos ya fueron evaluados. La evaluación de una expresión pos fija P usando una pila, ahora tiene lugar, como sigue:

1.− Si el elemento siguiente de P es un operando, colocarlo en la pila.

2.− Si el elemento siguiente de P es un operador n−ario, entonces sus n argumentos deben ser los n elementos superiores en la pila. Reemplazar estos n elementos por el resultado de aclarar esta operación usando los n elementos como argumento.

La evaluación es directa y fácil de implementar el generador de códigos usara el algoritmo antes mencionado para determinar el orden del código de generación para contar el valor de la expresión.

- (B)

Figura 3. Orden de evaluacion de operadores.

Evaluación infija: Aunque estas notaciones común, su uso en un lenguaje de programación conduce a varios problemas siguientes.

1.− Puesto que la notación infija se adecua solo para operadores binarios, es necesario conbinarlo con otra notación ya sea prefija o posfija. La mezcla hace que la traducción sea más complejas, los operadores unarios y las funciones con argumento múltiples deben ser excepciones a la propiedad infija general.

2.− Cuando aparece mas de un operador infijo en una expresión, la notación en inherentemente ambigua a menos que se utilicen paréntesis.

En este ultimo punto se demuestra que el valor de la expresión 2x3+4,la expresión significa 10, pero con la misma facilidad puede ser 14. En la figura 3 (A) se hace la multiplicación antes que la suma y el figura 3 (B) es la suma antes que la multiplicación.

Se puede usar paréntesis para eliminar la ambigüedad de cualquier expresión indicando explícitamente el agrupamiento de operadores y operandos, como en (AxB)+C o Ax(b+C), pero la expresión complejas los nidos profundos de paréntesis que resultan se vuelven confusas.

Jerarquía de operaciones: los operadores de una expresión están puestas en una jerarquía u orden de precedencia. La jerarquía de ADA es representativa (Tabla 1). En la expresión donde intervienen operadores de mas de un nivel en la jerarquía, la regla implica que los operadores de mayor precedencia se ejecutan

primero, así, en axb+c, x esta arriba de + en la jerarquía por eso se ejecuta primero.

Nivel de precedencia Operadores Operaciones

Precedencia mas alta ** abs not Exp,valor abs., negación

*/ mod rem Multiplicación, división

+− Adición, sustracción unaria

+−& Adición, sustracción binaria

="<>" Relacionales

precedencia más baja and or xor Operaciones booleanas

Tabla 1. Jerarquía de operaciones en Ada.

Precedencia Operadores Nombres de operadores

17 componentes léxicos, a[k],f() Literales, subindizacion, llamada defunciones

.,−> Selección

16 ++,−− Incremento/decremento posfijo

15* ++,−− Incremento/decremento prefijo

~, −, sizeof Operaciones unarias, almacenadas

!, &,* Negación lógica, indireccion

14 (typename) Conversiones forzadas

13 *, /, % Operaciones multiplicativas

12 +, − Operaciones aditivos

11 <<,>> De desplazamiento

10 <, >, <=, >= Relacionales

9 ==, != De igualdad

8 & And por bits

7 ^ Xor por bits

6 ø Or por bits

5 && And logico

4 øø O logico

3* ? : Condicionalidad

2* =,+=, −=, *=, /=, %=, De asignación

<<=, >>=, &=, ^=, ø=

1 , Evaluación secuenciales

(* operaciones asociativas derechas)

Tabla 2. Niveles de precedencia en C para operaciones.

Asociatividad. Donde intervienen operaciones de igual nivel de jerarquía, se necesita otra regla implícita adicional de asociación. Por ej, en a−b−c, la asociación de izq a derecha es la regla implícita mas común, de modo que seria (a−b)−c.

Los lenguajes evolucionan con nuevas operaciones que no son de las matemáticas clásicas; C; APL;SMALTALK y FORTH son todos ej de lenguaje que manejan operadores de distintas formas:

- C: Este lenguaje utiliza un amplio conjunto de tablas, como está en la tabla 2. Todas nuestras asociatividad de izquierda a derecha menos las marcadas con estrellas. Estos niveles son razonables con PASCAL, tienen una anomalía en las expresiones booleana a=bøc=d es ilegal, puesto que pascal predetermina agrupamiento no natural de a=(bøc)=d, por eso pascal es mas seguro usar las expresiones entre paréntesis.

- APL: Este lenguaje esta proyectado sobre arreglos y vectores. El APL actúa de derecha a izquierda, esto es razonable en casi todos los programas en APL con excepción de ciertas expresiones "típicas" que se hacen no intuitiva, por ej, a−b−c, se evalúa como a−(b−c) que es igual a a−b+c en los lenguajes como PASCAL,FORTRAN o C.

- SMALLTALK: Emplea un modelo como el APL ya que este lenguaje desarrolla funciones que proporcionan la funcionalidad requerida, el precedente para una función no queda claro, por esto la precedencia se omite y la evaluación es de izquierda a derecha.

- FORTH: Es un lenguaje de computadoras de proceso en tiempo real, era un lenguaje que cuya estructura en tiempo era en una pila y su sintaxis era posfija,

Representación en tiempo de ejecución.

La primera etapa de esta traducción establece la estructura básica de control de árbol de la expresión utilizando las reglas implícitas de precedencia y asociatividad cuando en la expresión intervienen notación infija. En una segunda etapa optativa, se toma las decisiones en detalle respecto al orden de evaluación.

1.−secuencias de código de maquina: La expresión se traduce a código de maquina real realizando 2 etapas de traducción indicadas en un paso.

El ordenar las instrucciones refleja la estructura de control de secuencias de la expresión original, es computadores convencionales estas secuencias deben hacer uso de localidades explícitas de almacenamiento temporal, para guardar resultados temporales.

2.− Estructura de arbol: Las expresiones se pueden ejecutar directamente en su representación de árbol. Primera etapa usando un interprete de software. La ejecución (segunda etapa) se puede conseguir con un simple recorrido de árbol.

3.− Formato prefijo y posfijo: se puede ejecutar con el algoritmo que se ha dado antes. El código de maquina real se representa en formato posfijo, la representación prefija es lka forma ejecutable de programación en SNOBOL4. La ejecución se realiza de izquierda a derecha y se llama al interprete en forma recursiva.

Evaluación de representaciones de arboles de expresiones.

La traducción de expresiones de representación de árbol causa dificultad, el procedimiento de traducción es sencilla. La 2ª etapa cuando el árbol se traduce a series ejecutables de operaciones. Aquí se consideran los problemas de orden de evaluación que surge al determinar el código exacto por generar.

Problema 1: Reglas de evaluación uniforme: Al evaluar una expresión o generar un código se espera que se aplique la regla de evaluación uniforme siguiente para cada uno de sus operandos evaluados. A esto la llamaremos la regla de evaluación impaciente por que siempre se evalúan 1º los operandos. Bajo esta regla de evaluación, para la expresión (a+b)x(c−a) de la fig 2 seria correcto cualquier orden de evaluación siguiente:

Figura 4. Expresiones que contienen una condicional..

Orden 1. Calcular primero a+b

1. Obténgase el valor r de a

2. Obténgase el valor r de b

3. Súmese a y b para obtener d

4. Obténgase el valor r de c

5. Réstese a de c para obtener c

6. Multiplíquese d y e para obtener , el valor de la expresión.

Orden 2. Evalúense los operandos antes que cualquier operador.

1. Obténgase el valor r de c

2. Obténgase el valor r de b

3. Obténgase el valor r de a

4. Réstese a de c para obtener e

5. Súmese a y b para obtener d

6. Multiplíquese d y e para obtener

todo esto es muy natural, y seria deseable contar con esta regla de evaluación, por desgracia no siempre se aplica, el mejor ejemplo es que contenga condicional, por ej en C la expresión z+(y=0?x:x/y)tiene un if que calcula si Y no es cero. Seria deseable tratar una condicional con una sintaxis rara y 3 operandos, como en la

figura 4 de hecho, esto se hace en lisp , utilizando la notación polaca de Cambridge. Ahora existe un problema con la evaluación uniforme. Si se supone la regla y se evalúan los operandos del operador condicional fig 4 se produce el efecto que hace que se busca evitar con la condicional, esto es, dividir X entre Y incluso si Y es cero.

Problema 2: Efectos colaterales: Considere la expresión:

a x un (X) + a;

antes de efectuar la multiplicación se debe obtener el valor r de a y evaluar un(X). La suma requiere el valor de a y el resultado de la multiplicaron, no debería haber problema si un(X). Se evalúa antes o después de la obtención del valor de a, sin embargo, si un tiene efectos colaterales de cambiar el valor de a, entonces el orden exacto es critico. Por ej, si a tiene valor 1 y un(X) devuelve 3 y cambia el valor de A a 2, entonces los valores posibles pueden ser:

1.− Evaluar cada termino en orden: 1x3+2=5.

2.− Evaluar A sólo una vez: 1x3+1=4.

3.− Llamar un(X) antes de evaluar A: 3x2+2=8.

Todos los resultados son correctos, de acuerdo con la sintaxis del programa.

Han surgido dos posturas en estas expresiones. Una es no permitir en absoluto funciones con efectos colaterales. Otro punto son los efectos colaterales, que se debe permitir y la definición del lenguaje es de dejar en claro el orden de evaluación. La dificultad con esta ultima postura hace imposible muchos tipos de opciones.

Problema 3: Condiciones de error: Una clase de error especial de efecto colateral es en el caso de operaciones que pueden fallar y generar una condición de error. A diferencia de otros efectos colaterales que se produce en funciones definidas por el programador, pueden surgir errores en operaciones primitivas (dividir entre cero), en tales situaciones el programador puede requerir un control preciso de orden de evaluación, pero la demanda de optimización puede impedirlo. La solución a estas diferencias tiende a ser en esencias AD HOC y varia de lenguaje a lenguaje y de implementaron a implementaron.

Problema 4: Expresiones booleanas en cortocircuito: En programación es natural usar operaciones booleanas And (o) (&& en C) y Or (øø en C) para combinar expresiones relacionadas como los enunciados en C:

IF ((A==0)øø(B/A>C)) {...}

WHILE ((I<=UB)&&(V[I]>C)) {...}

En ambas expresiones el segundo operando boolean puede llevar a error, el primer operando se incluye para asegurar que el error no ocurra. En C, si la expresión izquierda se evalúa como cierta en el primero y falsa en el segundo, entonces la segunda nunca se avalúa lógicamente, esto tiene sentido por que es claro el valor de la expresión øø es cierto si solo es falso.

El problema de evaluación antes mencionado esta presente, en muchos lenguajes ambos operandos se evalúan antes que la operación booleana, muchos errores de programación previenen del valor del operando izquierdo de la operación booleana puede poner en cortocircuito el resto de la evaluación si se puede decir cual es el valor de la expresión global a partir del valor del operando izquierdo solo. En ADA, una solución es incluir 2 operaciones boolean, And ten (y entonces) y Or Eles (si no), que proporcionan explícitamente evaluación en

cortocircuito, además de las operaciones booleanas ordinarias And (Y) y Or (O), que no lo hacen, por ej, en ADA.

IF(A=0) Or Else (B/A>C) ten...

No puede fallar debido a divisiones entre cero, puesto que si a=o la evaluación de la expresión completa concluye y el valor se toma como cierto.

Secuenciamiento con expresiones no aritméticas

En al sección anterior se realizo el orden de ejecución en la evaluación de expresiones aritméticas. Sin embargo, en los lenguaje están presentes otras formatos de expresiones. Los lenguajes como ML y PROLOG, proyectados para procesar datos de caracteres, incluyen otras formas de evaluación de expresiones.

Concordancia de patrones

Una operación crucial en lenguajes crucial en lenguajes como SNOBOL4, PROLOG y ML es la concordancia de patrones. En este caso la operación tiene éxito haciendo concordar y asignar un conjunto de variables a una plantilla predefina. El reconocimiento de árbol de analis sintáctico en gramática BNF es representativa de esta operación.

Por ej; la gramática reconoce palíndromos de longitud impar en todo el alfabeto 0 y 1:

A! 0A0 ø 1A ø0 ø1

El reconocimiento de la cadena valida 00100 tiene lugar así:

A3

0 A2 0

0 A1 0

1

Figura 5.− Concordancia de patrones en SNOBOL4.

A1 coincide con el centro 1

A2 coincide con 0 A1 0

A3 coincide con 0 A2 0

A partir de estas 3 asignaciones de A1 a 1, A2 a 0 A1 0 y A3 a 0 A2 0, se puede construir el árbol de análisis sintáctico completo de esta cadena (figura 5).

SNOBOL4 es un lenguaje para simular esta característica, como lo muestra la sinopsis de lenguaje 3. El PROLOG tiene 9 enunciados de longitud, seria difícil reproducirlo en otro lenguaje.

En tanto SNOBOL4 emplea reemplazamiento de cadena para su operación de concordancia, PROLOG utiliza una relación como un conjunto de n−tuplas como mecanismo de concordancia. Por medio de la especificación de casos conocidos de estas relaciones(llamados hechos), es posible deducir otras cosas, por ej, se puede

considerar la relación padre de con los hechos.

Padre de (Juan, María). −Juan es padre de María

Padre de (Susana, María) −Susana es padre de María

Padre de (Carlos, Juan) −Carlos es padre de Juan

Padre de (Ana, Juan) −Ana es padre de Juan

Para encontrar al padre de María se escribe la relación padre de (X, María) y PROLOG trata de asignar un valor a X a partir de conjuntos conocidos de hechos de su base de datos e inferir que X puede ser Juan o Susana, si desea a los dos padres de María que sean diferentes, por lo tanto se escribe:

Padre de (X, María) y Padre de (Y, María), not (X=Y)

La coma separa los procedimientos en un operando and (Y), todos los predicados tienen que ser ciertos para que la relación sea cierta.

El PROLOG infiere de manera lógica las relaciones a partir de un conjunto de hechos. Esto se pude construir con base en otras relaciones, como en Abuelo de:

Abuelo de (X,Y):− padre de (X, Z), padre de (Z, Y)

Significa que la relación abuelo de se define como (se escribe :−) 2 relaciones abuelo de tales que Z es el padre de Y, X es el padre de Z. Por lo tanto, abuelo de (X, María) dará el resultado X que es Carlos o Ana, pero abuelo de (Y, John) es incorrecto, puesto que no existe precedente en la base de datos.

Reescritura de términos

La reescritura de términos es una forma restringida de concordancia que tiene numerosas aplicaciones. Dada la cadena **a**1 **a**2...**a**n y la regla se reescribe ! , si =**a** i, se dice que **a**1... **a** i−1 ... **a**n.

La concordancia de una consulta a una base de datos es una forma de reescritura.

La generación de una derivación de una cadena en un lenguaje, es simplemente una forma de proceso de Reescritura, por ej:

A!0B ø1

B!0A

Se puede generar la cadena 001 con la derivación siguiente:

A!0B usando la regla A! 0B

0B!00A usando la regla B! 0A

00A!001 usando la regla A! 1

el ML emplea el termino Reescritura como una forma de definición de funciones, por ej la función factorial se puede especificar de manera sencilla en ML:

fun factorial(n:int)= **if** n=1 **else** n * factorial (n−1);

el dominio para factorial se compone de 2 conjuntos para n=1 el valor devuelto es 1, para los demás enteros positivos, el valor devuelto es la expresión n, factorial (n−1).

El enunciado **if** divide los dos casos , estos se puede ver como 2 funciones separadas, una constante 1 para el conjunto {1} y el valor n* factorial (n−1), para los enteros restantes (es decir {n øn>1}).

En vez de usar el **if** par separar los subdominios se puede emplear Reescritura de términos para indicar cada caso por separado.

Fun longitud(nada) = o

ø Longitud(a::y)= 1+longitud(y);

cada subdominio separado porø. El ML reemplazara la llamada de función por la definición apropiada.

Retroceso.

Al describir el comportamiento de ejecución de prolog para add, todas las submetas concordaron en el proceso del cómputo. sin embargo, ¿qué ocurre si esto no es cierto?, ¿Qué pasa si alguna submeta no se puede unificar con una regla de datos?. En este caso, se dice que regla fracasa.

En todo momento se hace concordar una submeta , si la concordancia ha de tener éxito , entonces una de las metas posibles tendrá éxito, solo que no se sabe necesariamente cuál. Si se intenta primero la meta incorrecta , entonces esa meta fracasará. En esta caso, simplemente se intenta otra meta posible.

Para poder describir el *algoritmo general de retroceso* se debe hacer:

1.− Para la submeta fijar k=1 como una nueva meta.

2.− Intentar concordar en forma sucesiva goal, para k=1..M y devolverse ya sea éxito o fracaso, según si alguna meta tiene éxito o todas fracasan.

3.− Si la meta tiene éxito, la submeta también tiene éxito.

4.− Si la meta fracasa para todas las k, entonces . Se tendría que devolver a la submeta e intentar la meta siguiente posible k+1 para esa submeta.

5.− Si la meta tiene éxito, entonces devolver éxito como resultado de la meta padre que se estaba buscando.

6.− Si la submeta fracasa , entonces la meta fracasa; se deberá devolver el fracaso como resultado de la búsqueda de la meta por el padre.

Lo que intenta hacer el algoritmo es tratar de concordar en forma exitosa, apilando y desapilando sucesivamente resultados parciales.

Control de secuencias entre enunciados.

Esto trata los mecanismos básicos que se usan para controlar el orden de ejecución de los enunciados dentro de un programa.

ENUNCIADOS BÁSICOS

Los resultados de cualquier programa están determinados para sus enunciados básicos que aplican operaciones a objetos de datos. Son ejemplos de estos enunciados básicos los enunciados de asignación, llamadas de subprogramas, y enunciados de entrada y salida. Dentro de un enunciado básico se puede invocar una serie de operaciones usando expresiones, como se dijo en el caso anterior, pero para un fin actual, cada enunciado básico se puede considerar como una unidad que representa un solo paso en el cómputo.

Asignaciones a objetos de datos

Los cambios al estado del cómputo por asignación de valores a objetos de datos es el mecanismo más importante que afecta el estado de un cómputo que realiza un programa, para esto existen varias formas de estos enunciados:

Enunciado de asignación: El propósito de una asignación es atribuir al valor L de un objeto de dato (es decir, la posición de memoria) el valor R (es decir, el valor del objeto de dato) de cierta expresión. La asignación es una operación fundamental definida por todos los tipos elementales de datos. La sintaxis es diferente para cada tipo de lenguaje, por ejemplo:

A:=B	PASCAL, ADA.
A=B	C, FORTRAN, PL/I, PROLOG, ML, SNOBOL4.
MOVE B TO A	COBOL.
A B	APL.
(SET Q A B)	LISP.

En C, la asignación es simplemente un operador de modo que se puede escribir: c=b=1, que significa (de acuerdo con los valores de precedencia de la tabla).

1.− Se asigna b al valor 1.

2.− La expresión (b=1) devuelve el valor 1.

3.− SE da c el valor 1.

Es mas frecuente, sin embargo, que la asignación se considere como un enunciado por separado. Puesto que la operación de asignación de pascal no devuelve un resultado explícito, se usa solo en el nivel de enunciados, en un enunciado de asignación explícito:

X::=B + 2*C

Y:= A + X;

La mayoría de los lenguajes contienen un solo operador de asignación, los cuales son:

A = B Asignar el valor R de B a valor L de A , devolver valor R.

A+= B(−=) disminuir A en B (A=A+B o A = A − B), devolver valor nuevo.

++a (−−A) disminuir A, luego devolver valor nuevo.

A++ (A−−) devolver valor de A , luego disminuir A.

Si bien todos de estos son similares, cada uno tiene una semántica ligeramente distinta y puede tener un efecto diferente en diversas situaciones. Puesto que la operación básica de asignación consiste en asignar el valor R de una expresión al valor 1 de otra expresión, se puede conseguir mayor flexibilidad incluyendo operadores que afectan el valor 1 y el valor R de las variables. En C, el operador unario * es un operador de indirección que hace que el valor R de una variable se comporte como si fuera el valor 1, y el operador unario & es un operador de dirección que convierte el valor 1 en un valor R, como por

ejemplo.

Int i, *q

P = &i;

*p = 7

Esto se puede explicar de esta forma:

1.− i se declara como un entero;

2.− p se declara como un apuntador a un entero.

3.− p se fija de modo que apunte a i; es decir, el valor 1 de i |se convierte en un valor R (&i) y este valor R se guarda como el valor R de p.

Enunciado de entrada

Casi todos los lenguajes de programación incluyen una forma de enunciados para leer datos dl usuario en una terminal de archivos o de una línea de comunicaciones.

Otras operaciones

Los parámetros se suelen definir como una asignación del valor de argumento al parámetro formal. Hay diversas formas de asignación implícita; en SNOBOL4, cada referencia a la variable implícita hace que se asigne un nuevo valor a ella.

*Formas de control de secuencias al nivel de enunciados***.**

A nivel de enunciados hay tres formas principales de control de secuencia:

1.− C**omposición**: Los enunciados pueden disponer de una serie textual de modo que deben ejecutarse en orden y siempre la ejecución de la estructura mayor del programa que contenga la serie.

2.− A**lternancia**: Dos series de enunciados pueden formar alternativas de modo que se ejecute una u otra serie, pero no las dos siempre que se ejecute la estructura mayor del programa que contenga la serie.

3.− I**teración:** Cuando hay una serie de enunciados pueden ejecutar en forma repetitiva, cero o más veces; cuando es cero significa que se puede omitir del todo la ejecución. Para esto se debe ejecutar la estructura mayor del programa que contenga la serie.

*Control explícito de secuencia***.**

Las maquinas se componían de posiciones de memoria, los primeros lenguajes por ejemplo (FORTRAM,

angol) las modelaban con tipos de datos simples traducibles directamente a objetos de maquina (FLOAT DE C) Y (REAL DE FORTRAM) a punto flotante de hardware.

Enunciado goto: En muchos lenguajes están presentes dos forma de enunciado goto (ir a), uno es el enunciado goto incondicional y el otro es el enunciado goto condicional.

Goto incondicional: Dentro de una seré de enunciados, un goto incondicional, como:

Goto PROXIMO

GOTO CONDICIONAL: Dentro de una serie de enunciados, un goto condicional, como:

If A = 0 then goto próximo

En tanto los lenguajes contengan estructuras de control anidables , como las intrucciones **while e if,** las cuales se analizan en breve, el goto es un artefacto que se puede pasar en alto sin dificultad. Desdichadamente, en ciertos lenguajes, como las primeras versiones de FORTRAM y APL, la transferencia explícita del control es necesaria, puesto que faltan las estructuras compuestas apropiadas.

Enunciado break: Ciertos lenguajes, como C, incluyen un enunciado break (escapar) como una forma de control explícito estructurado. Por lo común, el break causa que el control pase más adelante en el programa hasta un punto explícito al final de una estructura de control dada. Este break en C hace que el control salga del enunciado **while**, **for** o **switch** que lo encierra directamente.

En resumen el enunciado break cumple la función de que cauda la salida de una iteración, mientras que el continúe ocasiona una nueva iteración.

Diseño de programación estructurada

Algunas de las ventajas aparentes de los goto son:

1.− Manejo directo por el hardware para una ejecución eficiente si las etiquetas son simplemente marcas sintácticas locales en los enunciados.

2.− Sencillos y fáciles de usar en programas pequeños.

3.− Familiares para los programadores capacitados en lenguaje ensamblador o lenguajes más antiguos.

4.− De uso completamente general como bloque de construcción para representar (simular) cualquiera de otras formas de control que se analizan mas adelante.

Es más importante que el programa se ejecute correctamente en ves de tener la máxima eficiencia. El diseño de lenguajes de programación debe reflejar estas necesidades.

El concepto de la estructura de control de un punto de entrada y uno de salida contribuye a un diseño más inteligible. Un programa de mas de unos pocos enunciados es difícil de entender a menos que los enunciados estén organizados jerárquicamente en grupos, donde cada grupo represente una unidad conceptual del cómputo subyacente.

No es necesario que el orden de los enunciados en el programa corresponda al orden de ejecución. Mediante el uso de enunciado goto, es fácil escribir en los programas en los cuales el control salta entre diferentes series de enunciados siguiendo patrones irregulares. En este caso, el orden en que los enunciados aparecen en el

programa tienen poca relación con el orden de ejecución de los mismos.

Programación estructurada

Este término se usa para el diseño de programas que hace énfasis en:

1.− El diseño jerárquico de estructuras de programas usando solo las formas simples de control, de composición, alternancia e iteración.

2.− La representación directa del diseño jerárquico en el texto del programa, usando los enunciados de control estructurados.

3.− El texto de programa en el cual orden textual de los enunciados corresponden al orden de ejecución.

4.− El uso de grupos de enunciados de propósito único, aun cuando sea necesario copiar los enunciados.

Control de secuencia estructurado

Casi todos los lenguajes suministran un conjunto de enunciados de control para expresar las formas básicas de control: composición, alternancia e iteración. Un aspecto importante de los enunciados que se van a analizar es que cada uno es un enunciado de entrada por salida, lo que significa de que cada enunciado hay un solo punto de entrada al enunciado y un solo punto de salida al mismo.

Los lenguajes más antiguos, como COBOL y FORTRAM, contienen algunos enunciados de control de entrada por salida, pero todavía se apoyan fuertemente en enunciados goto y etiquetas de enunciados. Ha sido difícil adaptar ambos lenguajes a conceptos modernos de lenguaje.

Enunciados compuestos

Ejemplos:

Begin

.. − serie de enunciados (uno o más)

end

En C se escriben simplemente como {}

Enunciados condicionales

Ejemplos:

Enunciados if. La ejecución opcional de un enunciado se expresa como un **if** de una bifurcación.

Ejemplo.

If condición **then** enunciado **endif**

Enunciados de iteración

La iteración suministra el mecanismo básico para los cálculos repetidos en casi todos los programas. La

estructura básica de un enunciado de iteración consiste en una cabeza y un cuerpo. La cabeza controla el número de veces que el cuerpo se va a ejecutar, en tanto que el cuerpo es ordinariamente de enunciado (compuesto) que proporciona la acción del enunciado.

Repetición simple

El tipo mas sencillo de cabeza de enunciado de iteración especifica que el cuerpo se debe ejecutar cierto número fijo de veces. El PERFORM de COBOL es representativo de esta construcción:

Perform cuerpio K **time**

Hace que se evalúe K y luego se ejecute el cuerpo del enunciado ese número de veces.

Repetición mientras se cumple la condición. Se puede construir una iteración algo mas compleja usando una cabeza de repetir mientras. Una forma típica es:

While prueba **do** cuerpo

En esta forma de enunciado de iteración, la expresión de prueba se evalúa después de cada vez que se ha ejecutado el cuerpo.

Repetición mientras se incrementa un controlador. La tercera forma alternativa de enunciado de iteración es el enunciado cuya cabeza especifica una variable que sirve como un contador o índice durante la iteración. En la cabeza se especifica un valor inicial, un valor final y un incremento, y el cuerpo se ejecuta repetidamente usando primero el valor inicial como valor de la variable índice, luego el valor inicial mas el incremento, después el valor inicial mas dos veces el incremento, y así sucesivamente, hasta que se alcanza el valor final. En FORTRAM−77 ésta es la única forma de enunciado de iteración disponible. El enunciado for en ANGOL ilustra la estructura típica.

For l := l step 2 **until** 30 **do** cuerpo

En su forma general:

For k := N−1 step 2 * (W−1) **until** M * N **do** cuerpo

Repetición indefinida

Cuando hay problemas en la salida de la iteración, se suele emplear una iteración sin prueba explícita de terminación en la cabeza.

Ejemplo en ADA :

loop

..

exit when condición:

end loop:

En pascal:

While cierta do begin. End;

El enunciado for en C permite todos estos conceptos en una sola construcción:

For (opción; opción; opción) { cuerpo }

Implementación en enunciados iterativos

Para implementar una iteración for, las expresiones de la cabeza de la iteración que definen el valor final y el incremento se deben evaluar a la entrada final de la iteración y guardar en áreas especiales de almacenamiento temporal.

*Problemas en el control de secuencia estructurada***.**

El enunciado goto se usa como último recurso cuando los enunciados de control son inadecuados, hay áreas que hacen innecesario el enunciado goto las cuales son:

ITERACIONES DE SALIDAS MULTIPLES

EJEMPLO DE ESTA ES :

For 1 en 1..K iteración

Exit when vect(1) = 0

End loop;

OTRO EJEMPLO ES EL DE DO−WHILE−DO:

Loop

Read (x)

If fin−de−archivo then goto & { afuera de la iteración }

Process (x)

End loop;

A do−while−do se le conoce como (HACER MIENTRAS HACER), ya que while intermedio maneja la secuencia.

OTRO EJEMPLO ES:

Dowhiledo

Read (x)

While (no fin_de_archivo)

Process (x)

Endowhiledo;

Desafortunadamente ningún lenguaje implementa esta estructura, aunque en C, if (condición) BREAK se acerca y exit when de ADA es similar.

Programas primos

Estos fueron desarrollados por MADDUX (MADDUX 1975) como una generalización de la programación estructurada para definir la disposición jerárquica peculiar de un diagrama jerárquico.

Los nodos de función representan cómputos que hace un programa y muestra como cuadros con un solo arco que entra a este tipo de nodo y un solo arco que sale del mismo.

/

*

+

2

A

SQRT

–

B

–

*

**

2

B

C

*

A

4

×

–

+

b

c

a

a

+

x

3

2

4

x

+

4

3

2

Características: todas las operaciones aritméticas son aplicables a vectores y arreglos; y existen operaciones adicionales para crear vectores especiales.puesto que muchas de estas operaciones vectoriales no son intuitivas, no tienen presedente en APL y los enunciados se ejecutan de izquierda a derecha.

Sinopsis de lenguaje 1.: APL.

Ejemplo: Sumar elementos a un arreglo.

1 k! Leer tamaño del arreglo en k.

2 A!› Leer el primer elemento de un arreglo.

3 A!A, › Leer próximo elemento del arreglo. Expandir arreglo.

- !3x i k > A Tamaño de A(A)comparado con k dando 0o1. Generar vector de

0o1(operador i) y goto (!) enunciado 3 si el resultado es 1 (es

Decir, no hecho todavía)

5 !+/A Sumar elementos del arreglo (+/) y producir suma.

Sinopsis de lenguaje 2 : FORTH

Características: El sistema se ejecuta en dos pilas: una pila de retorno de subrutinas y una pila de evaluación de expresiones. Modelo en tiempo de ejecución muy pequeño, lo cual lo hace útil en computadoras pequeñas.

Ejemplo: programa para el computo de: 1 + 2 + 3+.....+ 9 + 10

(notación a,b,c es pila de expresiones, c es pila (tope))

:SQRT DUP*; (define cuadrado por: n ! n, n ! (n*n))

:DOSUM SWAP 1 + SWAP OVER SQRT +; (N,S ! N + 1, S + (N + 1))

(N,S ! S, N ! S,(N+1) ! (N+1), S !

(N+1), S, (N+1) !(N+1),S ,(N+1) !

(N+1), S + (N+1))

3 6 DOSUM. . 22 4 ok (Punto(.)imprima pila (tope). Salida es 22=4 +6)

0 0 10 0 DOSUM LOOP .385 ok (Aplicar DOSUM de 0 a 9 (Alto en 10))

+

Z

IF

=

Y

/

0

X

Y

X

Sinopsis de lenguaje 3: SNOBOL4

Características : Concordancia de patrones con base en gramáticas BNF; totalmente dinámico, la implementaron emplea macros virtuales de procesamiento de cadenas.

Ejemplo: encontrar palíndromos mas largo de longitud non sobre 0 y 1 en cadenas de entrada.

START GRAMMAR= 0ø1ø0 *GRAMMAR 0ø1 * GRAMMAR 1

* Fijar patrón como gramática BNF

```
LOOP NEWLINE= TRIM(INPUT) :F(END)

* Obtener próxima línea sin espacios colgantes. Si fracaso, ir a END

NEWLINE(POS(0)SPAN("01") RPOS(0)) :F(BAD)

* Hacer coincidir reglon con 0s y 1s.

* SPAN es cadena de 0s y 1s

* POS(0) es primer pos, RPOS(0) es el ultimo

SN= SIZE(NEWLINE)

NEXT NEWLINE POS(0) GRAMMAR, PALINDROME POS(SN)

− :S (OK) F(NOTOK)

* Reglon coincide con gramática a través de POS(SN)

* Si fracaso, mover ultimo pos. Si éxito, imprimir respuesta.

* Parte igualada asignada a PALINDROME

OK OUTPUT= "CONCORDANCIA:" PALINDROMO :LOOP

NOTOK SN= SN−1 : (NEXT)

BAD OUTPUT = " ENTRADA INCORRECTA: "NEWLINE :LOOP

END
```

Ejecución de muestra: Entrada Salid

11011 CONCORDANCIA: 11011

1101101 CONCORDANCIA: 11011

11211 ENTRADA INCORRCTA: 11211