**Ejercicios DLP Resueltos**

# Conceptos Básicos del Diseño de Lenguajes – Tema 1

## Actividades y preguntas

1. ¿Cuál es la diferencia entre compiladores e intérpretes?

La diferencia principal es que un compilador recibe como entrada un lenguaje de alto nivel y de salida es un lenguaje de bajo nivel. En cambio un intérprete solo tendrá un lenguaje de entrada.

1. ¿Cuáles son las principales responsabilidades de las siguientes fases en un procesador de lenguaje?
   1. Análisis Léxico: separa en tokens

Comprueba que las palabras sean válidas

Asigna a cada lexema un token único

* 1. Análisis sintáctico: crea el AST

Comprueba que las palabras estén en el orden adecuado

* 1. Análisis semántico: comprueba que la sentencia se pueda realizar. Define el significado de las sentencias.
  2. Generación de Código: asigna direcciones y genera código usando plantillas
  3. Optimización de código: optimiza el código resultante

1. ¿Cuál es el patrón estructural más común usado en la implementación de un lenguaje de programación?

Visitor

1. ¿Cuál es la principal diferencia entre un árbol de análisis y un árbol de sintaxis abstracta (AST)?

Su propósito y nivel de detalle.

El árbol de análisis se usa para representar la estructura sintáctica de un programa mientras que el AST es una estructura de datos que simplifica la semántica del programa.

1. ¿Cuál es la diferencia entre un compilador y un transpilador?

El **compilador** convierte el código fuente de un lenguaje de alto nivel a código de bajo nivel, mientras que el **transpilador** convierte el código fuente de un lenguaje de alto nivel a otro de alto nivel.

1. Describe 4 ejemplos distintos de procesadores de lenguaje diferentes al compilador, intérprete y transpilador

• Ensamblador: Traduce lenguaje ensamblador a

código binario

• Desensamblador: traduce código binario/máquina

en código ensamblador

• Descompilador: Traduce código binario a alto

programas de nivel

• Compilador cruzado: Un compilador que genera

código binario para una plataforma diferente a la que está en

que está ejecutando el compilador

1. Identifica 3 usos diferentes del diseño de lenguajes de programación, distintos al diseño y la implementación de los procesadores de lenguaje

Investigación, enseñanza e implementación de dominios específicos.

1. Define front, middle y back-end de un compilador

**Front**: primera etapa del proceso de compilación, y se encarga de analizar el código fuente del programa y generar una representación intermedia o AST

**Middle**: se encarga de optimizar el código intermedio generado por el fron para mejorar su rendimiento y eficacia

**Back-end**: se encarga de generar el código ejecutable final a partir del código optimizado del middle

1. ¿Cuál es la principal diferencia entre un compilador y un traductor de lenguaje?

Un **compilador** traduce el código fuente de un lenguaje de programación de alto nivel a un código máquina que puede ser ejecutado, mientras que el **traductor de lenguaje** traduce el código fuente de un lenguaje de programación a otro lenguaje de programación.

1. ¿Qué fase del procesador del lenguaje realiza conversiones implícitas de expresiones?

Análisis semántico

1. ¿Qué fase del procesador del lenguaje comprueba que el operador – puede aplicarse a una única expresión?

Análisis sintáctico

1. Identifica las fases con las siguientes responsabilidades:
   1. Hacer que el código final se ejecute rápidamente: Optimización
   2. Transformar bucles en saltos: Generación de código
   3. Añadir información al AST: Semántico
   4. Comprobar la aridad (p.ej, el número de parámetros) de un operador: Semántico
   5. Representar el código de entrada en memoria: Sintáctico
   6. Detectar el número correcto de argumentos en la invocación a una función: Semántico
   7. Avisar al programador del uso de caracteres que no están contemplados en el lenguaje: Léxico
   8. Asignar offsets (desplazamientos) a variables en el programa de entrada: Generación de código
   9. Aprovechar al máximo el uso de los registros de la CPU: Generación de código
   10. Comprobar si se puede realizar una conversión implícita: Semántico
2. Nombra la información pasada entre:
   1. El lexer y el parser: tokens generados por el leer y parejas de lexemas
   2. El parser y el analizador semántico: árbol sintáctico generado por el parser (el AST)
   3. El analizador semántico y la generación de código: representación que contiene información sobre las estructuras de control de flujo, las operaciones aritméticas y lógicas y la asignación de valores a las variables (el AST decorado)

# Análisis Léxico – Tema 2

## Actividades y preguntas

1. Escribe una Context-Free-Grammar (CFG) para reconocer el siguiente lenguaje:

{[*n*(A|B)*m*]*n* : n,m≥0}, where VT = {A, B, [, ]}.

VT = {A, B, [, ]}.

Vn={p,p2}

S=P

P={

X -> ‘[’ X ’]’ | X2

X2 -> ε | X2 ‘A’ | X2 ‘B’

}

1. ¿Qué reconoce el siguiente patrón léxico de ANTLR? 3.7

Un 3 cualquier caracter y un 7

REAL\_LITERAL: [0-9] + ’.’ + [0-9]+

1. Escribe un archivo de especificación léxica ANTLR para reconocer los siguientes patrones:
   1. Identificadores: var1, a, var\_2, \_\_private, \_
   2. Constantes Reales con punto flotante: 12.3, 2., .34
   3. Constantes Reales con mantisa y exponente: 34.12E-3, 3e3, .3E+3, 2.e23
   4. Descartar comentarios de línea: // This is one single-line comment

El archivo sería:

REAL\_CONSTANT: (INT\_CONSTANT\*'.'('0'|'0'[1-9][0-9]\*|[1-9][0-9]\*)?)|(([1-9][0-9]\*)[Ee][-+]?([1-9][0-9]\*))| INT\_CONSTANT\*'.'('0'|'0'[1-9][0-9]\*|[1-9][0-9]\*)?[Ee][-+]?([1-9][0-9]\*)

REAL: ([0-9]\*’.’[0-9]+) | ([0-9]+’.’[0-9]\*)

IDENT: [a-z\_]+[0-9a-z\_]\*;

COMMENT: (‘//’.\*? (‘\n’|EOF)) -> Skip;

1. En ANTLR, ¿qué sucede si un lexema coincide con dos patrones léxicos?

Siempre se elige el más largo, y si son del mismo tamaño, se selecciona el patrón que se haya definido primero

1. Define derivación en un paso

Es el resultado (otras cadenas) de aplicar cualquier transformación a una cadena

1. Define cadena de idioma

Secuencia finita de símbolos que pertenecen al alfabeto del lenguaje.

1. ¿Cuál es la principal diferencia entre las CFGs y las expresiones regulares?

La principal diferencia es que las CFGs pueden generar lenguajes más complejos y estructurados que las expresiones regulares.

1. Define gramática libre de contexto (CFG)

Metalenguaje que nos permite definir las estructuras del lenguaje.

1. En ANTLR, ¿qué sucede si un lexema no es reconocido por ningún patrón?

Se produce un error de análisis léxico y se lanza el error.

1. ¿Qué reconoce el siguiente patrón léxico de ANTLR? [3-7d-f]

Reconoce **un solo carácter** que puede ser un número del 3 al 7 o una letra de la d a la f. Ejemplos: 3, d, 4, f, 5, e…

1. Define expresión regular

Metalenguaje que permite reconocer los tokens de un lenguaje. Secuencia de caracteres que permite reconocer lexemas en un texto.

1. ¿Qué reconoce el siguiente patrón léxico de ANTLR? [A|B|C]

Reconoce A, B o C.

1. Escribe una CFG para una posible secuencia vacía de A terminales separados por el terminal coma (,)

bnf

prog -> ε | letraA

letraA-> ‘A’ | letraA ‘,’ ‘A’

ebnf

(A(‘,’A)\*)?

1. Escribe una producción léxica ANTLR para constantes enteras

LITENT: [0-9]+;

# Análisis sintáctico - Tema 3

## Actividades y preguntas

1. ¿Cuál es la principal relación entre un parse tree y un AST?

Ambos representan la estructura sintáctica de un programa.

1. Define gramática ambigua. ¿Por qué debe evitarse?

Una gramática es **ambigua** si existen dos o más parse trees posibles para una misma cadena de entrada, es decir, si es posible obtener diferentes interpretaciones para una misma cadena. Debe evitarse pues la ambigüedad puede dificultar la implementación de un analizador sintáctico o dar resultados inesperados en el procesamiento de la entrada.

1. Define:
   1. Derivación en un paso: transformación de una sóla regla de producción que aplica a un símbolo no terminal en una gramática.
   2. Derivación: secuencia finita de transformaciones de una gramática hasta que se llega a una cadena compuesta de símbolos terminales.
   3. Cadena: lexema válido, conjunto de caracteres validados por parte del léxico.
   4. Programa o sentencia: secuencia de instrucciones que describen cómo realizar una tarea determinada en un lenguaje de programación específico.
2. ¿Qué recursión debe evitarse en los LL parsers? ¿Por qué?

Se debe evitar la recursión a la izquierda, pues puede causar problemas en la derivación de la gramática y en algunos casos entrar en un ciclo infinito.

1. Di si las siguientes gramáticas son ambiguas o no. ¿Por qué?
   1. statement → if-stmt  
      statement → ID = exp  
      expression → ID  
      expression → INT\_CONSTANT  
      if-stmt → IF ( exp ) statement  
      if-stmt → IF ( exp ) statement ELSE statement

Diagrama

Descripción generada automáticamente Diagrama

Descripción generada automáticamente

No es ambigua porque cada símbolo no terminal tiene una única producción.

* 1. expression → expression + expression  
     expression → ID  
     expression → INT\_CONSTANT

Sí que es ambigua, por ejemplo si se tiene la expresión a+b+c, no está claro si la derivación debería ser (a+b)+c o a +(b+c)

* 1. expression → ( expression )  
     expression → - expression  
     expression → ID  
     expression → INT\_CONSTANT

No es ambigua porque cada símbolo no terminal tiene una única producción.

1. ¿Qué hace ANTLR si la entrada de una gramática es ambigua?

Cambiar las reglas de selección. Es decir aunque en ANTlr el orden de las reglas no importe, si lo hacen en el caso de que sea una gramática ambigua, en ese caso el orden si será importante.

1. En ANTLR, ¿qué es devuelto por un método que implementa una producción cuando no hay código embebido añadido?

Se devuelve el último token de la producción.

1. Dada la siguiente gramática:

e → A e B  
e → ε

escriba el orden de creación de los nodos del árbol de análisis en un analizador de abajo hacia arriba, para la entrada program AABB; primero, escribe el árbol y luego describe el orden.

El orden es:

E => AeB => AAeBB => AABB Al revés

1. Define gramática abstracta

Metalenguaje que se usa para documentar árboles AST donde se define una regla por cada nodo. Cada regla tendrá el nombre del nodo, si pertenece a categorías sintácticas (expresión, statement…).

<nodo>:<categorías> -> <tipos hijos>

1. Dibuja el AST para el siguiente fragmento de código Java:

a = b = f(a, b);

Texto, Carta

Descripción generada automáticamente

1. Dada la siguiente gramática ANTLR, dibuja el AST creado para la entrada: - a > b + c

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

Texto, Carta

Descripción generada automáticamente

1. ¿Cuál es la principal diferencia entre los parsers LL y LR?

La principal diferencia es el orden en que realizan las operaciones. Los LL realizan primero la sustitución a la izquierda y se construye el árbol sintáctico de arriba abajo, mientras que los LR realizan la sustitución a la derecha y se construye el árbol sintáctico hacia arriba.

1. ¿Cuál es la principal diferencia entre los parsers LL(1) y LL(\*)?

Los LL(1) sólo pueden ver un símbolo de entrada adelante para decidir qué producción aplicar, mientras que el LL(\*) puede ver cualquier cantidad de símbolos de entrada en adelante.

1. ¿A qué hace referencia la primera L en los parsers LL y LR?

Significa left-to-right.

1. ¿Cómo se especifica la asociación de izquierda a derecha en ANTLR? Nombra dos operadores de asociaciones de derechas de un lenguaje conocido.

Mediante el orden de las producciones en la gramática. Ej: el = y el ^

1. Añade código embebido a la siguiente gramática ANTLR para crear el AST más simple (obvia los número de fila y columna)

Texto

Descripción generada automáticamente con confianza baja

statement returns [Statement ast]:

'write' e1=expression { $ast = new Print($e1.ast); }

(',' e2=expression { $ast = new Print($e2.ast);})\*

| ID '=' expression { $ast = new Assigment($ID.text,$expression.ast); }

;

1. Dada la asignación: a[i+1] = record.field[a][b+2] = f(3);
   1. Escribe una gramática BNF que reconozca este lenguaje
   2. Haz lo mismo con la notación EBNF
   3. Dibuja el árbol de análisis sintáctico de la gramática BNF
   4. Dibuja el árbol de análisis sintáctico de la gramática eBNF
   5. Dibuja el AST

Pizarrón blanco con letras negras

Descripción generada automáticamente con confianza media

Texto, Carta

Descripción generada automáticamente

Expression: …| IDENT ‘(‘ argOpt ‘)’

argOpt: vacio | args

args: expression |args ‘,’ expression SOLUCIÓN BNF

Un dibujo de un pizarrón blanco

Descripción generada automáticamente con confianza media

Texto, Carta

Descripción generada automáticamente

1. Los lenguajes que soportan funciones como funciones de orden superior permiten las siguientes construcciones:

myFunctionF(myFuctionG, 3) (myFunctionH)

myFunctionF recibe otra función como primer parámetro y devuelve otra función que, a su vez, recibe otra función

1. Escribe una gramática BNF que reconozca este tipo de expresiones
2. Haz lo mismo con la notación EBNF
3. Dibuja el AST de la expresión anterior

¿?

1. Dada la siguiente gramática:

a → a a  
a → ( a )  
e → ε

* 1. Describe el lenguaje que genera

Genera una serie de paréntesis balanceados (x paréntesis izquierdos y x derechos)

* 1. Demuestra que es ambigua

Es ambigua pq ese número de paréntesis no tiene pq ser igual en izquierdos que en derechos. Ejemplo:

a->(a)->(aa)->((a)a)->(((a)a)->((()a)->((()))

Esto produce dos árboles distintos.

1. Dada la siguiente gramática:

exp → exp addop term  
exp → term  
addop → +  
addop → -  
term → term mulop factor  
term → factor  
mulop → \*  
mulop → /  
factor → ( exp )  
factor → INT\_CONSTANT

Escribe las derivaciones de un paso más a la izquierda, los árboles de análisis sintáctico y los AST para los siguientes programas:

1. 3+4\*5-6

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Exp => exp addop term => exp addop term addop term => term addop term addop term …

Un dibujo de un pizarrón blanco

Descripción generada automáticamente con confianza media

1. 3\*(4-5+6)

Diagrama

Descripción generada automáticamente

1. 3-(4+5\*6)
2. ¿Es la gramática anterior ambigua? ¿Por qué? No

# Análisis semántico – Tema 4

## Actividades y preguntas

1. Describe la semántica de la siguiente declaración: fibonacci(v[i]);

Función de Fibonacci que recibe el valor que está en la posición i del array v

1. ¿Cuál es la principal diferencia entre la semántica del lenguaje y el análisis semántico?

La **semántica** es el conjunto de reglas formales que dictan el significado de los programas sintácticamente correctos.

El **análisis semántico** de un procesador de lenguaje es la fase que hace cumplir las reglas semánticas del lenguaje.

1. Nombra 5 ejemplos diferentes de reglas semánticas en Java, proporcionando un ejemplo de cada una

Para la asignación a la izquierda debe ser un lvalue.

Si hay una multiplicación ambos operadores deben poder realizer operaciones aritméticas.

Las variables deben estar definidas.

Tipos deben estar inferidos

Los operadores de una operación lógica deben eser booleanos

1. Define gramática atribuída

Las AGs añaden atributos a las construcciones sintácticas del lenguaje. Los atributos representan propiedades de las construcciones sintácticas. Las AGs especifican de forma declarativa cómo se calculan los valores de los atributos.

1. Explica cuál es la evaluación de una gramática atribuída

Es el cálculo de todos los atributos de un programa dado

1. Describe el algoritmo general para evaluar gramáticas atribuídas

1. Contruir el árbol de sintaxis para un programa específico

2. Crear el grafo de dependencias de atributos

3. Calcular un ordenamiento topológico del grafo dirigido

4. Ejecutar las reglas semánticas, siguiendo dicho ordenamiento topológico

1. Define una gramática de atributos bien definidos

Una AG está bien definida (no es circular) si para cada árbol sintáctico, pueden ser evaluados todos los atributos.

1. Escribe un ejemplo de una gramática de atributos mal definidos

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

1. ¿Está bien definida la siguiente gramática?

G:  
 (1) Arithmetic: expression 1 -> expression2 + expression3  
 (2) IntLiteral: expression -> INT\_CONSTANT  
R:  
 (1) expression1.value = expression2.value + expression3.value  
 (2) expression.type = new IntType()

No está bien porque en la regla (2) debería poner .value, es decir:

(2) expresión.value = INT\_CONSTANT

De la forma en la que está el enunciado, el valor del atributo expresión del IntLiteral nunca tendrá un valor.

1. ¿Se puede recorrer siempre cualquier gramática de atributos bien definidos con un solo recorrido lineal?

Existen AGs bien definidas que no pueden ser evaluadas con un solo recorrido lineal. En ese caso, se puede dividir la AG en diferentes AGs que pueden ser evaluadas recorriendo primero en profundidad.

1. Define atributos heredados y sintetizados en una gramática atribuída

**Heredados**: los atributos de los nodos hijos se calculan cuando recorremos el nodo padre. El recorrido de árbol es pre-order o in-order.

**Sintetizados**: los atributos del nodo padre dependen de los atributos de los nodos hijos. El recorrido del árbol es post-order.

1. Extiende la siguiente gramática libre de contexto para convertirla en una gramática bien definida. En la gramática resultante, incluye un atributo heredado (llamado inherited) y otro sintetizado (synthesized)

**(G):**

(1) nonTerminal1 -> nonTermial2 TERMINAL1 nonTerminal3  
(2) nonTerminal2 -> TERMINAL2  
(3) nonTerminal3 -> TERMINAL3

**(A):**

Atributo Afecta a

Synthesized nonTerminal1

Inherited nonTerminal2

(R):

(1) nonTerminal2.inherited = ‘hijo izq’

nonTerminal3.inherited = ‘hijo derecho’

nonterminal1.synthesized = nonTerminal2.a + nonTerminal3.b

(2) nonTerminal2.a = valor

(3) nonterminal3.b = valor

1. ¿Cuál es un recorrido válido para gramáticas de atributos bien definidos con todos los atributos sintetizados?

Post-Order

1. ¿Cuál es un recorrido válido para gramáticas de atributos bien definidos con todos los atributos heredados?

Pre-order o in-order

1. ¿Cuál es un recorrido válido para gramáticas de atributos bien definidos con todos los atributos sintetizados y heredados?

TODOS¿?

1. ¿Cuál es el mejor patrón de diseño para implementar la evaluación de una gramática atribuída simple?

Visitor

1. En el Visitor:
   1. ¿Cuál es el nombre del método a implementar en los nodos del AST? accept(Visitor, param)
   2. ¿Cuál es el nombre del método a implementar en el Visitor particular? visit(Class,param)
   3. ¿Cuántos métodos con respecto a este patrón deben implementarse en un determinado visitor? Los necesarios.
   4. ¿Cuántos métodos con respecto a este patrón deben implementarse en los AST nodes para n visitors? Sólo 1, el accept
   5. Dada una instancia de TypeCheckingVisitor y el nodo Expression , escribe el código para enlazar el visitor y la expresión

public interface AST{

<TP,TR> TR accept(Visitor<TP,TR> v, TP param);

}

public class Expression implements AST{

@Override

public <TP, TR> accept(Visitor<TP,TR> v, TP param){

return v.visit(this, param);

}

}

public interface Visitor<TP, TR>{

TR visit(Expression e, TP param);

}

public class TypeCheckingVisitor implements Visitor<Void,Void>{

public Void visit(Expression e, Void param){

…

}

}

* 1. ¿Cómo paso tres parámetros (integer, double and String) mientras atravieso un AST? Con un array de Objects

1. Dado el siguiente ejemplo:

double f(double r) {  
 if (r<0) return -1;  
 else return ‘0’;  
}

Para cualquier declaración de retorno, el tipo de su expresión debe ser un subtipo del tipo definido por la función. En el ejemplo anterior, int (-1) y char ('0') son subtipos de double, por lo que se cumple la condición.

Dada la siguiente gramática

(1) FuncDefinition: funcdefinition -> type ID vardefinition\* statement\*  
(2) FunctionType: type -> type vardefinition\*  
(3) CharType: type -> ε  
(4) IntType: type -> ε  
(5) DoubleType: type -> ε  
(6) If: statement 1 -> expression statement 2 + statement 3 \*  
(7) While: statement 1 -> expression statement 2 \*  
(8) Return: statement -> expression

Escriba una gramática de atributos para verificar que cualquier declaración de retorno realmente devuelva una expresión cuyo tipo es un subtipo del tipo declarado en la definición de la función. Puede asumir el diseño correcto del sistema de tipos y utilizar sus servicios (por ejemplo, métodos públicos).

1. = {statement.returnType} Dominio={Type}

(R):

(1) statement\*.foreach(stmt -> stm.returnType = funcdefinition.type.returnType)

(8) expression.type.mustPromotesTo(statement.returnType)

Este método comprueba que un argumento promociona al tipo de un parámetro

1. ¿Cuál es el principal objetivo de la fase de Identificación?

El objetivo es comprobar que todos los símbolos que se utilicen hayan sido previamente declarados.

1. ¿Cuál es el nombre de la estructura de datos usada en la fase de Identificación? ¿Cuál es su propósito?

Tabla de símbolos. Tiene como propósito almacenar los nodos VarDefinition para permitir su posterior recuperación. Una vez que los nodos Variable están vinculados a las VarDefinition, la tabla se puede eliminar.

1. ¿Cuáles son las dos fases del análisis semántico?

Identificación y Comprobación de tipos

1. ¿Cuáles son las principales tareas en que consiste la Comprobación de Tipos?

Debe inferirse el tipo de cada expresión y debe verificarse que las operaciones aplicadas a cada expresión sean válidas.

1. Da 3 definiciones de tipo desde los puntos de vista denotacional, basado en la abstracción y constructivo.

**Denotacional**: un tipo es un conjunto de valores.

**Basado en abstracción**: un tipo es una interfaz que consta de un conjunto de operaciones.

**Constructivo**: un tipo es un tipo built-in (primitivo o predefinido) o un tipo compuesto creado a partir de un constructor de tipos (array, struct…)

1. Define:
   1. Type expression: representación de un tipo
   2. Type system: colección de reglas para asignar expresiones de tipo a las diferentes construcciones sintácticas del lenguaje
   3. Type checker: implementa un type system y es parte del analizador semántico
2. ¿Cuál es el propósito del tipo Visitor en el patrón de diseño Visitor?

Permite separar el algoritmo de recorrido de la estructura de datos sobre la que opera. Permite que se puedan hacer diferentes algoritmos de recorridos del AST. Los nuevos recorridos pueden ser añadidos sin modificar el AST.

1. Dada la expresión f(v[i+1]), traza todas las posibles operaciones que se pueden realizar en el analizador semántico

**Fase de Identificación**: comprueba que la i está definida, luego v y por último f.

**Fase de Comprobación de tipos**: comprueba que i sea de tipo entero para realizar la operación aritmética. Infiere el tipo de la suma. Comprueba si v es de tipo Array. Luego comprueba que el resultado de la suma sea de tipo int. Comprueba si f es una función y que tiene un parámetro. Comprueba si el primer argumento sea del mismo tipo que el del parámetro de la definición de f. Infiere el tipo resultante de la expresión.

1. ¿Cuáles son los principales beneficios que aporta el patrón Composite para implementar un tipo system?

* Modelado de estructuras de árbol recursivas para representar jerarquías de parte-todo
* Tratamiento uniforme de objetos simples y objetos compuestos sin importar su estructura

1. ¿Realiza la máquina virtual de Java algún análisis semántico cuando ejecuta los .class?

¿?

1. Escribe una gramática abstracta que convierta expresiones infijas a notación postfija

A + b

A b +

+ a b

G =

1. Arithmetic: expression -> expression + expression
2. Identifier: expression -> ID
3. IntLiteral: expression -> INT\_CONSTANT

(A)={expression.postfix, expression.prefix}

(R):

(1) expression.postfix = expression2.postfix expression3.postfix ‘+’

expression.prefix = ‘+’ expression2.postfix expression3.postfix

(2) expression.postfix = ID

expression.prefix = ID

(3) expression.postfix = INT\_CONSTANT

expression.prefix = INT\_CONSTANT

1. Dada la siguiente producción:

Invocation: expression1 -> expression2 expression3\*

* 1. Escribe una gramática atribuída para inferir el tipo de invocación
  2. Implementa un método del tipo sistema para inferir el tipo de invocación

a)

(A)={expression.type} Dominio={Type}

(R):

(1) expression1.type -> expression2.type.parenthesis(expression3\*.stream.map((par) => par.type).toList())

b)

public Type parenthesis(List<Type> list, ASTNode a){

if(list.size() != parameters.size())

return new ErrorType(a.getColumn(), a.getLine(), “Error: número de parámetros incorrecto”)

for(int i = 0; i < parameters.size(); i++){

Type inicial = parameters.get(i).getType();

Type t = inicial.promotesTo(list.get(i), a);

If(t instanceof ErrorType){

return t;

}

}

Return returnType;

}

# Lenguajes Intermedios y Representaciones

## Actividades y preguntas

1. ¿Por qué se usaron primero máquinas abstractas en la construcción de compiladores?

Para reducir la cantidad de traducciones de código realizadas en un compilador redirigible

1. ¿Cuál es la principal diferencia entre una máquina abstracta y una virtual?

Una máquina abstracta es un modelo teórico que describe cómo se debe ejecutar un programa, mientras que una máquina virtual es un programa que emula una máquina física real y puede ejecutar programas de la misma manera que lo hace una máquina física.

1. ¿En qué fases se usan comúnmente las representaciones de alto nivel? ¿Para qué se usan? Nombra 3 de ellas

Se usan en las fases de análisis (Comprobación de tipos, generación de código y optimizaciones de código). Son usadas como código intermedio.

1. ¿Qué es el middle-end de un compilador?

Es el enlace entre su front y su backend, donde se realizan muchas optimizaciones.

1. Nombra dos representaciones intermedias de medio nivel

Máquinas de pila, notación polaca inversa.

1. Escribe en tres direcciones de código la siguiente expresión: v[a+3\*g].field

NO ENTRA

1. Escribe el código MAPL para la siguiente expresión: v[a+3\*g].second .Asume que todo son variables globales y:

* La dirección de memoria de v es 0 y es un array de 10 elementos
* Los elementos de v tienen dos campos: first (real con offset 0) y second (carácter con offset 4)
* La dirección de memoria de a es 50 y es un carácter
* La dirección de memoria de g es 51 y es un real

¿?

1. Escribe un programa de alto nivel donde la instrucción dup en MAPL es usada

¿?

1. Escribe un programa en MAPL equivalente al siguiente:

int i, j, n;  
void p() {}  
char c;  
int f(int n, double r) {  
 double real;  
 char c1, c2;  
 c1 = (char)n;  
 real = (double)c1 + (double)n + r;  
 p();  
 return (int)c1;  
}

void main() {  
 struct {  
 int integer;  
 char character;  
 } pair;  
 int[10][5] vector;  
 i=0;  
 j=0;  
 while(i<10){  
 while(j<5) {  
 vector[i][j] = i + j;  
 j=j+1;  
 }  
 i=i+1;  
 }  
 i=0;  
 pair.character = '0';  
 pair.integer = 48;  
 while (pair.integer >= i) {  
 if (pair.integer == vector[0][0] || !(int)pair.character || i%2==0)  
 write 't', 'r', 'u', 'e', '\n';  
 else  
 write 'f', 'a', 'l', 's', 'e', '\n';  
 i=i+1;  
 }  
 write f(i, (double)i);  
 vector[9][(int)4.3]=5;  
 f(1, 2.2);  
}

¿?

1. Pros y contras de las representaciones intermedias de bajo nivel

Pros:

* Permiten la máxima optimización para un microprocesador específico
* La semántica de sus instrucciones está cerca del microprocesador de destino
* Optimizaciones a nivel de registro

Contras

* Sólo sirven para un conjunto de microprocesadores de la misma arquitectura
* La mayoría de ellos proporcionan registros simbólicos

1. Escribe un diagrama de arquitectura de un procesador de lenguaje que use representaciones de leguaje de los 3 niveles de abstracción

Diagrama

Descripción generada automáticamente

# Code Generation – Tema 6

## Actividades y preguntas

1. ¿Cuál es el principal objetivo de la fase de generación de código?

Generar código destino con la misma semántica que el programa fuente.

1. ¿Cúales son los principales subproblemas que se deben abordar en la fase de generación de código?

Asignación del almacenamiento

Selección de instrucciones

Asignación de registros

1. ¿La generación de código es parte del front-end o del back-end de un compilador?

Back-end

1. En MAPL, ¿cuál es la ecuación que define el offset de las variables locales?

Desde el registro BP, si quieres coger la variable local 1, le sumas su tamaño.

1. En MAPL, ¿cuál es el principal propósito de BP?

Se usa para establecer un punto de interrupción en el código fuente. Es útil para la depuración.

1. En MAPL, ¿dónde apunta el BP?

Siempre apunta a la información de control del marco de pila

1. ¿Qué funciones de generación de código deben definirse para la invocación de funciones?

Address, value y execute

1. Escribe dos ejemplos de código Java donde las asignaciones se usen como expresiones

v[0] = a+b

z = 7+a

1. Dada la siguiente función: double f(char c, double d){ int[10] v; int w;};

En MAPL, ¿cuáles son las direcciones de memoria de c, d, v y w?

c 12

d 14

v 0

w 10

¿?

1. Detalla las partes de un marco de pila de funciones (registro de activación)

**Argumentos** (parámetros reales): los valores de cada parámetro de función

**Información de control**: información necesaria para restaurar el estado anterior de ejecución

**Variables locales (de la función llamada)**: ser liberarán cuando finalize la ejecución de la función

**Ocasionalmente, los valores temporales necesarios en la ejecución**

1. En MAPL, ¿cuál es la ecuación que define el offset de los parámetros de una función?

¿?

1. En Java, qué funciones de generación de código se deben definir para la operación de asignación

¿?

1. Dado Type[size1][size2] w; ¿Cuál es la dirección de memoria de w[i][j]?

address(w) + i\*Size2\*numberOfBytes(Type) + j\*numberOfBytes(Type)

1. Escribe la plantilla de código para acceder al campo de registro

¿?

1. Dado int i; int j; double[10][20] w; Escribe el código MAPL para la siguiente expresión: w[i][j]

PUSHA 0

PUSHA 2

PUSHA 4

¿?

1. Define la plantilla de código “execute” para la invocación de funciones para generar código MAPL

¿?

1. Dada la siguiente producción:

Assignment: expression 1 → expression2 expression3

Define la plantilla apropiada para generar su código MAPL

[[Assignment: expression1 -> expression2 expression3]]

1. Escribe las plantillas necesarias para generar el código MAPL de un Switch
2. Escribe las plantillas necesarias para generar el código x86 de las siguientes instrucciones MAPL: push, add, load y store.
3. Considera el operador ‘+=’ como un nuevo tipo de statement (no expresión) para un lenguaje determinado. Con tal operador, las siguientes declaraciones aceptadas por el compilador son:

myInteger += 1;  
myDouble += myDouble \* 2.1;  
myIntVector[myInteger+1] += myInteger\*2;  
myRecord.intField += function\_returning\_int(myInteger)

Los tipos de los dos operandos deben ser iguales y deben ser enteros o dobles. No se permiten caracteres.

1. Escribe la producción abstracta (no concreta) que especifica el nuevo += statement
2. Identifica las funciones de código necesarias para la producción previa y escribe las correspondientes plantillas de código para la producción anterior
3. Escribe las plantillas de código necesarias para generar código MAPL para las operaciones aritmética(+, -, \* y /) y comparación (>, <, >=, <=, == y !=) para cualquier combinación de expresiones char, int y double. Ambas operaciones permiten operandos de cualquiera de esos 3 tipos