Procesadores de Lenguaje: Práctica Anual

Ingeniería en Informática 4º C

Facultad de Informática UCM (2011-2012)

1ª Entrega

Grupo: 10

Miembros:

IVAN LAVERA ULLOA
CLAUDIA LERTORA GINÉS
MARIA TRINIDAD MARTÍN CAMPOS
CHRISTIAN GREGORIO MORALES PEÑA
DAVID RODRIGUEZ GONZALEZ
NELDA JOANNA ZEVALLOS RODRIGUEZ

Índice de Contenidos

Introducción

Muy breve descripción del contenido de la memoria y de su estructura.

Se recomienda crear un índice para acceder directamente a cada una de las secciones, numerar las páginas, etc.

0. Descripción del lenguaje fuente

En esta sección se debe describir informalmente cómo es el lenguaje de programación de alto nivel que se va a procesar. En este apartado basta con copiar las características que os proponemos a continuación (salvo las restricciones contextuales que se llevarán al punto 4.1).

Características que debe tener el lenguaje definido:

- Los programas de este lenguaje deberán comenzar por la palabra reservada "programa:" seguido del nombre del programa, finalizado por el símbolo ".".
- Los programas constarán de una *sección de declaraciones* y de una *sección de instrucciones*,
    - La sección de declaraciones comenzará por la palabra reservada "declaraciones:" y la de instrucciones por la palabra reservada "instrucciones:".
- La sección de declaraciones constará de una secuencia de una o más *declaraciones*, separadas entre sí por el símbolo "`;`"
- El lenguaje *distingue cadenas según estén escritas con letras mayúsculas o minúsculas*, tanto si son palabras reservadas (que se escriben *siempre en minúsculas*) como si son identificadores de variables
    - Es decir, aunque dos identificadores tengan las mismas letras, se consideran distintos si no coinciden en mayúsculas y minúsculas
- Cada declaración constará de la palabra reservada "variable" seguido del nombre de la *variable*, seguido de la palabra reservada "tipo" y a continuación el nombre de un *tipo*
    - Los identificadores de las variables comienzan necesariamente por una letra en minúscula, seguida de cero o más letras y dígitos
    - En la sección de declaraciones no podrá haber variables duplicadas
    - El tipo podrá ser `booleano` (valores booleanos `verdadero` y `falso`), `caracter` (caracteres alfanuméricos), `natural` (números naturales), `entero` (números enteros con signo) y `real` (números reales)
- La sección de instrucciones constará de una secuencia de una o más *instrucciones*, separadas por el símbolo "`;`"
- El lenguaje sólo tendrá tres tipos de instrucciones: *instrucciones de asignación*, *instrucciones de lectura* e *instrucciones de escritura*
- Una instrucción de asignación consta de una variable, seguida del símbolo "**=**" y a continuación una *expresión*
- Las expresiones usarán los siguientes operadores: <, >, <=, >=, ==, !=, +, -, *, /, %, y, o, no, - (unario), <<, >>, `(real)`, `(ent)`, `(nat)` y `(car)`

- El menor nivel de prioridad (nivel 0) es el de los operadores de comparación **<** (menor que), **>** (mayor que), **<=** (menor o igual que), **>=** (mayor o igual que), == (igual) y **!=** (distinto):
    - Todos estos operadores son binarios infijos y no asocian
    - Es posible comparar entre sí valores numéricos (natural con natural, natural con entero, natural con real, entero con natural, entero con entero, entero con real, real con natural, real con entero, real con real), caracteres (según la ordenación del estándar de caracteres usado en la implementación, por ejemplo UNICODE) y booleanos (`verdadero` se considera mayor que `falso`)
    - El resultado de una comparación es un valor booleano (`verdadero` cuando se cumple la comparación, `falso` cuando no se cumple)
- El siguiente nivel de prioridad (nivel 1) es el de los operadores aritméticos **+** (suma) y **-** (resta), así como el del operador lógico **o**
    - Todos ellos son operadores binarios infijos que asocian a izquierdas
    - **+** y **-** operaran sobre valores numéricos. El resultado será un valor real (siempre y cuando alguno de los operandos sea real), un valor entero (siempre y cuando no haya operandos reales y algún operando sea entero) o un valor natural (sólo cuando los dos operandos sea naturales)
    - **o** opera sobre valores booleanos. El resultado será el *o lógico* de los operandos
- El siguiente nivel de prioridad (nivel 2) es el de los operadores aritméticos ***** (multiplicación), **/** (división) y **%** (módulo), así como el operador lógico **y**
    - Todos ellos son operadores binarios infijos que asocian a izquierdas
    - ***** y **/** operan sobre valores numéricos. El resultado será un valor real (siempre y cuando alguno de los operandos sea real), un valor entero (siempre y cuando no haya operandos reales y algún operando sea entero) o un valor natural (sólo cuando los dos operandos sea naturales). En el primer caso, la división funcionará como división real. En los dos últimos casos, la división funcionará como división entera
    - En la operación *módulo* **%** el primer operando puede ser entero o natural, pero el segundo operando sólo puede ser natural. El resultado de *a* **%** *b* será el resto de la división de *a* entre *b*. El tipo del resultado será el mismo que el del primer operando
    - Por último, **y** opera únicamente sobre valores booleanos. El resultado será el *y lógico* de los operandos
- El siguiente nivel de prioridad (nivel 3) es el de los operadores de desplazamiento **<<** y **>>**
    - Ambos son operadores binarios infijos que asocian a derechas
    - Operan únicamente sobre valores naturales. Por un lado *a* **<<** *b* es el natural resultante de desplazar *b* bits hacia la izquierda en la representación binaria de *a*. Por otro lado *a* **>>** *b* es el natural resultante de desplazar *b* bit hacia la derecha en la representación binaria de *a*
- El mayor nivel de prioridad (nivel 4) es el del operador lógico **no**, el **-** unario, los operadores de *conversión* **`(real)`**, **`(ent)`**, **`(nat)`** y **`(car)`**,
    - Todos son operadores unarios prefijos, salvo el valor absoluto que sitúa al operando dentro de sus dos barras verticales

    - El operador *negación lógica* **no** asocia, y opera sobre valores booleanos, siendo el resultado la negación lógica de su operando
    - El operador **–** unario también asocia, opera sobre valores numéricos y el resultado es su operando cambiado de signo. Este resultado será real si el tipo del operando es real, y entero en otro caso
    - Los operadores de conversión no asocian
    - **`(real)`** ***a*** devuelve el propio *a* si es real. Si *a* es entero o natural el resultado es *a* convertido a real (añadiendo un único 0 en la parte decimal). Si *a* es un carácter el resultado es el código de dicho carácter convertido a real
    - **`(ent)`** ***a*** devuelve el propio *a* si es entero, Si *a* es real el resultado es la parte entera de *a*. Si *a* es natural el resultado es *a* convertido a entero (con signo positivo). Si *a* es un carácter el resultado es el código de dicho carácter convertido a entero
    - **`(nat)`** ***a*** devuelve el propio *a* si es natural. No admite operandos reales o enteros. Si *a* es un carácter el resultado es el código de dicho carácter
    - **`(car)`** ***a***.devuelve el propio *a* si es un carácter. Si *a* es natural el resultado es el carácter cuyo código es *a*. No admite operandos reales o enteros
- Como expresiones básicas, que podrán ser combinadas mediante los operadores anteriores, se consideran las siguientes:
    - Literales *naturales*. Secuencias de uno o más dígitos, no admitiéndose ceros a la izquierda. Téngase en cuenta que las expresiones enteras se pueden formar con literales naturales y el – unario
    - Literales *reales*. Secuencia formada por una parte entera, seguida de una de estas tres cosas: una parte decimal, una parte exponencial, o una parte decimal seguida de una parte exponencial. La parte entera tiene la misma estructura que un literal natural. La parte decimal está formada por el símbolo "**.**" seguido de uno o más dígitos, no admitiéndose ceros a la derecha (salvo si se trata de un único cero, como por ejemplo 2.0). La parte exponencial está formada por el símbolo **E** o el símbolo **`e`**, seguido opcionalmente del símbolo `-`, y seguido obligatoriamente de la misma estructura que tiene un literal natural
    - Literales *booleanos*. Con valores **`verdadero`** y **`falso`**
    - Literales *carácter*. Un símbolo de comilla simple **'** seguido de un carácter alfanumérico, seguido otra vez del símbolo de comilla simple **'**
    - Variables, que han debido ser convenientemente declaradas en la sección de declaraciones
- En las expresiones es posible utilizar paréntesis para alterar la forma en la que se aplican los operadores sobre los operandos
- Una instrucción de asignación debe cumplir además estas condiciones:
    - La variable en la parte izquierda debe haber sido declarada
    - A una variable de tipo real es posible asignarle un valor real, entero o natural (produciéndose automáticamente la correspondiente conversión), pero no un carácter
    - A una variable de tipo entero es posible asignarle un valor entero o natural (produciéndose automáticamente la correspondiente conversión), pero no un valor real o un carácter
    - A una variable de tipo natural únicamente es posible asignarle un valor natural

    - A una variable de tipo carácter únicamente es posible asignarle un valor de tipo carácter
    - A una variable de tipo booleano únicamente es posible asignarle un valor de tipo booleano
- Una instrucción de lectura tiene la forma **`lee(`*`v`*`)`** donde *v* es una variable. Su efecto es leer un valor del tipo de la variable *v* por la entrada estándar del usuario y almacenar en la variable *v* el valor leído
- Una instrucción de escritura tiene la forma **`escribe(`*`exp`*`)`**, donde *exp* es una expresión. Su efecto es escribir por la salida estándar del usuario el valor de *exp*
- El lenguaje admite *comentarios de línea*. Dichos comentarios comienzan con el símbolo **@** y se extienden hasta el fin de la línea

Ejemplo de programa en el lenguaje definido:

```
@ Programa de ejemplo

programa: ejemplo.

declaraciones:

variable cantidad tipo real;

variable euros tipo entero;

variable centimos tipo real;

instrucciones:

lee(cantidad);

euros = (ent)cantidad;

centimos = cantidad – euros;

escribe(euros);

escribe ('.');

escribe (centimos);

escribe ((car)10);  # Código del carácter salto de línea
```

1. Definición léxica del lenguaje

Especificación formal del léxico del lenguaje (a veces llamado microsintaxis) utilizando definiciones regulares.

PALABRAS RESERVADAS:

ResPrograma: ”programa:”

ResDeclaraciones: “declaraciones:”

ResInstrucciones: “instrucciones:”

ResEntero: “entero”

ResNatural: “natural”

ResReal: “real”

ResBooleano: “booleano”

ResCaracter: “caracter”

ResLee: “lee”

ResEscribe: “escribe”

ResVariable:”variable”

ResTipo: “tipo”

1- CATEGORÍAS LÉXICAS AUXILIARES

Letra : ["a" - "z", "A" - "Z", "ñ", "Ñ"]

LetraMinúscula : [“a” –“z”]

Dígito : [“0”-“9”]

DígitoSinCero: [“1”-“9”]

CarácterAlfanumérico: (Dígito | Letra)

ParteEntera: LiteralEntero

Comilla : “ ‘ “

2- TOKENS DE LITERALES E IDENTIFICADORES

Iden : LetraMinúscula (Letra | Dígito) *

LiteralNatural : DígitoSinCero (Dígito)*

LiteralEntero : (- ? LiteralNatural)|0)

LiteralBooleano: (“verdadero”| “falso”)

LiteralCaracter: Comilla CaracterAlfanumérico Comilla

ParteDecimal: (“0” | Digito* DigitoSinCero)

ParteExponencial: (“E” | “e”) -? LiteralNatural

LiteralReal: ParteEntera “.”(ParteDecimal |ParteExponencial |(ParteDecimal ParteExponencial))

idenPrograma: (letra|digito)(letra|digito)*”.”

3- TOKENS DE SIMBOLOS

PyC : “;”

ParAbierto: “(”

ParCerrado: “)”

Igual: “=”

4- TOKENS DE OPERADORES

4.1 OP0

Menor: “<” MenorIgual: “<=” IgualIgual: “==”

Mayor: “>” MayorIgual: “>=” Distinto: “!=”

4.2 OP1

Mas: “+” Menos: “-“ OR: “o”

4.3 OP2

Mult: “*” Divide: “/” Modulo: “%” AND: “y”

4.4 OP3

DespIzq: “<<” DespDcha: “>>”

4.5 OP4

Negacion: “no” CambioSigno: “-“

ConvReal: ParAbierto ResReal ParCerrado

ConvNat: ParAbierto “nat” ParCerrado

ConvEnt: ParAbierto “ent” ParCerrado

ConvCar: ParAbierto “car” ParCerrado

2. Definición sintáctica del lenguaje

Especificación formal de los aspectos sintácticos del lenguaje.

2.1. Descripción de los operadores

OPERADOR	DESCRIPCIÓN	NIVEL-PRIOR.	ARIDAD	INF-PRE-POSTFIJO	ASOCIATIVIDAD
<	Menor que	0	2	INFIJO	NO
>	Mayor que	0	2	INFIJO	NO
<=	Menor o igual que	0	2	INFIJO	NO
>=	Mayor o igual que	0	2	INFIJO	NO
==	Igual a	0	2	INFIJO	NO
!=	Distinto a	0	2	INFIJO	NO
+	Suma	1	2	INFIJO	IZQUIERDA
-	Resta	1	2	INFIJO	IZQUIERDA
o	OR	1	2	INFIJO	IZQUIERDA
*	Multiplicación	2	2	INFIJO	IZQUIERDA
/	División	2	2	INFIJO	IZQUIERDA
%	Módulo	2	2	INFIJO	IZQUIERDA
Y	AND	2	2	INFIJO	IZQUIERDA
<<	Despl. Izquierda	3	2	INFIJO	DERECHA
>>	Despl. Derecha	3	2	INFIJO	DERECHA
No	Negación lógica	4	1	PREFIJO	SI
-	Cambio de signo	4	1	PREFIJO	SI
(real)	Conv.Real	4	1	PREFIJO	NO
(ent)	Conv.Entero	4	1	PREFIJO	NO

(nat)	Conv.Natural	4	1	PREFIJO	NO
(car)	Conv.Caracter	4	1	PREFIJO	NO

2.2. Formalización de la sintaxis

Formalización de la sintaxis del lenguaje utilizando una gramática incontextual. Dicha gramática debe representar de manera natural las prioridades y la asociatividad de los operadores.

Prog -> ResPrograma idenPrograma **Declaraciones Instrucciones**

Declaraciones-> ResDeclaraciones **Decs**

Instrucciones-> ResInstrucciones **Is**

Decs -> Decs PyC **Dec**

Decs -> Dec

Dec -> ResVariable **Iden** ResTipo **Tipo**

Tipo ->ResEntero **Tipo ->** ResBooleano **Tipo->** ResCaracter

Tipo -> ResReal **Tipo->** ResNatural

Is-> Is Pyc **I**

Is-> I

I-> IAsig I-> Ilectura I-> Iescritura

IAsig-> Iden Igual **Exp0**

Exp0 -> Exp1 Op0 Exp1 Exp0-> Exp1

Op0 -> Menor**|**Mayor**|**MenorIgual**|**MayorIgual**|**IgualIgual**|** Distinto

Exp1 -> Exp1 Op1 Exp2 Exp1-> Exp2

Op1-> Mas**|**Menos**|**OR

Exp2 -> Exp2 Op2 Exp3 Exp2-> Exp3

Op2-> Mult**|**Divide**|**Modulo**|**AND

Exp3 -> Exp4 Op3 Exp3 Exp3 -> Exp4

Op3 -> DespIzq**|** DespDcha

Exp4 -> Exp4Asocia | Exp4NoAsocia

Exp4Asocia -> Op4a Exp4 Exp4Asocia -> Exp5

Op4a -> CambioSigno**|** Negacion

Exp4NoAsocia -> Op4na Exp5 Exp4NoAsocia -> Exp5

Op4na -> ConvEnt**|** ConvReal**|** ConvCar**|** ConvNat

Exp5 -> Iden **|** LiteralNatural **|** LiteralReal **|** LiteralEntero **|** LiteralBooleano **|** LiteralCaracter **|** ParAbierto **Exp0** ParCerrado

Ilectura -> ResLee ParAbierto Iden ParCerrado

Iescritura ->ResEscribe ParAbierto **Exp0** ParCerrado

3. Estructura y construcción de la tabla de símbolos

Deberá contemplarse tanto los aspectos necesarios para comprobar las restricciones contextuales, como los necesarios para llevar a cabo la traducción.

3.1. Estructura de la tabla de símbolos

Descripción de las operaciones de la tabla de símbolos definiendo la cabecera de dichas operaciones, así como describiendo informalmente su cometido, incluyendo el propósito de cada uno de sus parámetros.

La tabla de símbolos que utilizaremos en nuestro compilador será una tabla con clave un String, que será el nombre de el identificador de variable, y con contenido de cada posición de la tabla dos campos(dirección de memoria en el que se aloja la variable y el tipo de esta misma.

Clave	Contenido
String	<dirección,tipo>

3.2. Construcción de la tabla de símbolos

Formalización de la construcción de la tabla de símbolos mediante una gramática de atributos.

Incluimos la construcción de la tabla de símbolos en la gramática de atributos del lenguaje que vamos a procesar en el punto 3.2.3

3.2.1 Funciones semánticas

Descripción, si procede, de las funciones semánticas adicionales utilizadas en la especificación. Para cada función debe indicarse explícitamente su cabecera, así como informalmente su cometido, incluyendo el propósito de cada uno de sus parámetros.

Esta sección puede dejarse vacía si no se van a usar funciones semánticas adicionales.

Las funciones que vamos a utilizar para mantener nuestra tabla de símbolos van a ser:

- **public TablaSimbolos creaTablaDeSimbolos()**

  Construye la tabla de símbolos y la devuelve.

- **public boolean añadirIdentificador(String nombreVariable,Tipo tipo,int direccionMemoria);**

  Inserta un nuevo identificador en la tabla de símbolos recibiendo como parámetros, el nombre del identificador, el tipo de este y la dirección de memoria en la que se alojara el identificador.

Esta función devolverá cierto si se inserto correctamente en la tabla de símbolos y falso en caso contrario.

- **public boolean estaIdentificador(String identificador)**

  Devuelve cierto si el nombre de identificador pasado como parámetro se encuentra dentro de la tabla de símbolos

- **public Tipo dameTipoIdentificador(String identificador)**

  Devuelve el tipo del nombre de identificador pasado como parámetro.

  Si el identificador no se encontrase en la tabla de símbolos se devolverá null.

- **public int dameDireccionIdentificador(String identificador)**

  Devuelve el número de la posición de memoria que ocupa el identificador pasado como nombre como parámetro.

  Si el Identificador no se encontrase en la tabla de símbolos se devolverá -1.

3.2.2 Atributos semánticos

Para cada categoría sintáctica relevante en este procesamiento deben enumerarse sus atributos semánticos, indicando si son heredados o sintetizados, y describiendo informalmente su propósito.

Los atributos semánticos que tienen relación con la tabla de símbolos son:
dir: indica la dirección a rellenar en la tabla de símbolos
tss: atributo sintetizado que contiene los símbolos declarados hasta el momento
tsh: atributo heredado para las instrucciones (Is) que viene de las declaraciones (Decs).
id: etiqueta asociada a cada variable de la tabla de símbolos.

3.2.3 Gramática de atributos

Gramática de atributos que formaliza la construcción de la tabla de símbolos.

Prog -> ResPrograma idenPrograma Declaraciones Instrucciones

Intrucciones.tsh = Declaraciones.tss

Declaraciones-> ResDeclaraciones Decs

Declaraciones.tss = Decs.tss

Instrucciones-> ResInstrucciones Is

Is.tsh=Instrucciones.tsh

Decs := Decs Pyc Dec

Decs0.tss = añadeId(Decs1.tss, id: Dec.id, tipo: Dec.tipo, dir:Decs1.dir +1)

Decs0.dir = Decs1.dir+1

Decs := Dec

Decs.tss = añadeId(creaTS(), id: Dec.id, tipo: Dec.tipo, dir: 0)

Decs.dir = 0

Is:= Is Pyc I

Is1.tsh = Is0.tsh

I.tsh = Is0.tsh

Is:= I

I.tsh = Is.tsh

I:= IAsig

IAsig.tsh = I.tsh

I:= ILectura

ILectura.tsh = I.tsh

I:= IEscritura

IEscritura.tsh = I.tsh

IAsig:= Iden Igual Exp0

Exp0.tsh = IAsig.tsh

Exp0 := Exp1 Op0 Exp1

Exp10.tsh = Exp0.tsh

Exp11.tsh = Exp0.tsh

Exp0:= Exp1

Exp1.tsh = Exp0.tsh

Exp1 := Exp1 Op1 Exp2

Exp11.tsh = Exp10.tsh

Exp2.tsh = Exp1.tsh

Exp1:= Exp2

Exp2.tsh = Exp1.tsh

Exp2 := Exp2 Op2 Exp3

Exp21.tsh = Exp20.tsh

Exp3.tsh = Exp20.tsh

Exp2:= Exp3

Exp3.tsh = Exp2.tsh

Exp3 := Exp4 Op3 Exp3

Exp31.tsh = Exp30.tsh

Exp4.tsh = Exp30.tsh

Exp3 := Exp4

Exp4.tsh = Exp3.tsh

Exp4 := Exp4Asocia | Exp4NoAsocia

Exp4Asocia.tsh = Exp4.tsh

Exp4NoAsocia.tsh = Exp4.tsh

Exp4Abs.tsh = Exp4.tsh

Exp4Asocia := Op4a Exp4

Exp4.tsh = Exp4Asocia.tsh

Exp4Asocia := Exp5

Exp5.tsh = Exp4Asocia.tsh

Exp4NoAsocia := Op4na Exp5

Exp5.tsh = Exp4NoAsocia.tsh

Exp4NoAsocia := Exp5

Exp5.tsh = Exp4NoAsocia.tsh

Exp5 -> Iden | LiteralNatural | LiteralReal | LiteralEntero | LiteralBooleano | LiteralCaracter | ParAbierto Exp0 ParCerrado

Exp0.tsh = Exp5.tsh

Iescritura ->ResEscribe ParAbierto Exp0 ParCerrado

Exp0.tsh = IEscritura.tsh

4. Especificación de las restricciones contextuales

4.1. Descripción informal de las restricciones contextuales

Enumeración y descripción informal de las restricciones contextuales del lenguaje. En este apartado basta con copiar las restricciones contextuales que os proponemos al principio de la plantilla, cuando describimos las características del lenguaje.

- Toda variable que se use en cualquier instrucción debe haber sido previamente declarada.
- No pueden declararse variables repetidas.
- Sólo es posible comparar entre sí valores numéricos, caracteres y booleanos.
- +, –, *, /, *operador unario* **-** y *valor absoluto* | | sólo operan sobre valores numéricos.
- o,y, no sólo operan sobre valores booleanos.
- En la operación *módulo* **%** el primer operando puede ser entero o natural, pero el segundo operando sólo puede ser natural.
- <<, >> sólo operan sobre valores naturales.
- (nat), (car) no admiten operandos reales, enteros o booleanos.
- (real), (ent) no admiten operandos booleanos.
- A una variable de tipo real es posible asignarle un valor real, entero o natural, pero no un carácter.
- A una variable de tipo entero es posible asignarle un valor entero o natural, pero no un valor real o un caracter.
- A una variable de tipo natural únicamente es posible asignarle un valor natural.
- A una variable de tipo caracter únicamente es posible asignarle un valor de tipo carácter.
- A una variable de tipo booleano únicamente es posible asignarle un valor de tipo booleano.

4.2. Funciones semánticas

Descripción, si procede, de las funciones semánticas adicionales utilizadas en la especificación. Para cada función debe indicarse explícitamente su cabecera, así como informalmente su cometido, incluyendo el propósito de cada uno de sus parámetros.

fun tipoAsignación(tipoVar, tipoDato)

Comprueba si puede hacerse una asignación de variable/valor comprobando los tipos. Utiliza la siguiente tabla.

tipoVar\tipoDato	Ent	Real	Nat	Booleano	Car	Error
Ent	Ent	Error	Ent	Error	Error	Error
Real	Real	Real	Real	Error	Error	Error
Nat	Error	Error	Nat	Error	Error	Error
Booleano	Error	Error	Error	Booleano	Error	Error
Car	Error	Error	Error	Error	Car	Error
Error	Error	Error	Error	Error	Error	Error

fun tipoBinario (op, tipoIz, tipoDr)

Devuelve el tipo que resulta de aplicar la operación a dos elementos de tipoIz y tipoDr respectivamente. Utiliza las siguientes tablas.

- **Comparación:**

op= {<,>,<=, >=, ==, !=}

tipoIz\tipoDr	Ent	Real	Nat	Booleano	Car	Error
Ent	Booleano			Error	Error	Error
Real				Error	Error	Error
Nat				Error	Error	Error
Booleano	Error	Error	Error	Booleano	Error	Error
Car	Error	Error	Error	Error	Booleano	Error
Error	Error	Error	Error	Error	Error	Error

- **Aritméticos:**

op= {+, - , *, /}

tipoVar\tipoDato	Ent	Real	Nat	Booleano	Car	Error
Ent	Ent	Real	Ent	Error	Error	Error
Real	Real	Real	Real	Error	Error	Error
Nat	Ent	Real	Nat	Error	Error	Error
Booleano	Error	Error	Error	Error	Error	Error
Car	Error	Error	Error	Error	Error	Error
Error	Error	Error	Error	Error	Error	Error

- **Lógicos:**

op= { y, o}

tipoVar\tipoDato	Ent	Real	Nat	Booleano	Car	Error
Ent	Error	Error	Error	Error	Error	Error
Real	Error	Error	Error	Error	Error	Error
Nat	Error	Error	Error	Error	Error	Error
Booleano	Error	Error	Error	Booleano	Error	Error
Car	Error	Error	Error	Error	Error	Error
Error	Error	Error	Error	Error	Error	Error

- **Módulo**:

op={ %}

tipoVar\tipoDato	Ent	Real	Nat	Booleano	Car	Error
Ent	Error	Error	Ent	Error	Error	Error
Real	Error	Error	Error	Error	Error	Error
Nat	Error	Error	Nat	Error	Error	Error
Booleano	Error	Error	Error	Error	Error	Error
Car	Error	Error	Error	Error	Error	Error
Error	Error	Error	Error	Error	Error	Error

- **Desplazamiento:**

op= {<<, >>}

tipoVar\tipoDato	Ent	Real	Nat	Booleano	Car	Error
Ent	Error	Error	Error	Error	Error	Error
Real	Error	Error	Error	Error	Error	Error
Nat	Error	Error	Nat	Error	Error	Error
Booleano	Error	Error	Error	Error	Error	Error
Car	Error	Error	Error	Error	Error	Error
Error	Error	Error	Error	Error	Error	Error

fun tipoUnario (op, tipo)

Devuelve el tipo que resulta de aplicar la operación al elemento tipo . Utiliza las siguientes tablas.

- **lee/escribe:**

Ent	Real	Nat	Booleano	Car	Error
Ent	Real	Nat	Booleano	Car	Error

- **Negación lógica:**

op= {no}

Ent	Real	Nat	Booleano	Car	Error
Error	Error	Error	Booleano	Error	Error

- **Cambio de signo:**

op= {-}

Ent	Real	Nat	Booleano	Car	Error
Ent	Real	Nat	Error	Error	Error

- **Conversión a entero:**

op= {(ent)}

Ent	Real	Nat	Booleano	Car	Error
Ent	Ent	Ent	Error	Ent	Error

- **Conversión a real:**

op= {(real)}

Ent	Real	Nat	Booleano	Car	Error
Real	Real	Real	Error	Real	Error

- **Conversión a natural:**

op= {(nat)}

Ent	Real	Nat	Booleano	Car	Error
Error	Error	Nat	Error	Nat	Error

- **Conversión a caracter:**

op= {(car)}

Ent	Real	Nat	Booleano	Car	Error
Error	Error	Car	Error	Car	Error

4.3. Atributos semánticos

Para cada categoría sintáctica relevante en este procesamiento deben enumerarse sus atributos semánticos, indicando si son heredados o sintetizados, y describiendo informalmente su propósito.

Err : atributo sintetizado que indica si se ha producido algún en error a lo largo del análisis sintáctico.

Tipo : atributo sintetizado que almacena la clase a la que pertenecen los datos y variables del programa.

4.4. Gramática de atributos

Gramática de atributos que formaliza la comprobación de las restricciones contextuales.

Prog -> ResPrograma idenPrograma Declaraciones Instrucciones

Prog.err= Declaraciones.err | Instrucciones.err

Declaraciones-> ResDeclaraciones Decs

Declaraciones.err= Decs.err

Instrucciones-> ResInstrucciones Is

Instrucciones.err= Is.err

Decs:= Decs Pyc Dec

Decs0.err= Decs1.err | existeID(Decs1.tss, Dec.id) | Dec.err

Decs:= Dec

Decs.err=Dec.err

Dec -> ResVariable Iden ResTipo Tipo

Dec.err= ! (Tipo.tipo==real | Tipo.tipo==ent | Tipo.tipo==nat| Tipo.tipo==car | Tipo.tipo==booleano)

Tipo:= ResReal

Tipo.tipo= real

Tipo:= ResEntero

Tipo.tipo= ent

Tipo:= ResNatural

Tipo.tipo= nat

Tipo:= ResCaracter

Tipo.tipo= car

Tipo:=Resbooleano

Tipo.tipo= booleano

Is:= Is Pyc I

Is0.err= Is1.err .| I.err

Is:= I

Is.err= I.err

I:= IAsig

I.err= IAsig.err

I:= ILectura

I.err= ILectura.err

I:= IEscritura

I.err= IEscritura.err

IAsig:= Iden Igual Exp0

IAsig.err= (!existeID(IAsig.tsh, Iden.lex) | Exp0.err |
tipoAsignacion(IAsig.tsh[Iden.lex].tipo, Exp0.tipo)==error)

Exp0:= Exp1 op0 Exp1

Exp0.err= Exp10.err | Exp11.err| tipoBinario(op0, Exp10.tipo, Exp11.tipo)==error

Exp0.tipo= tipoBinario(op0, Exp10.tipo, Exp11.tipo)

Exp0:=Exp1

Exp0.err=Exp1.err

Exp0.tipo= Exp1.tipo

Exp1:= Exp1 op1 Exp2

Exp10.err= Exp11.err | Exp2.err| tipoBinario(op1, Exp11.tipo, Exp2.tipo)==error

Exp10.tipo= tipoBinario(op1, Exp1.tipo, Exp2.tipo)

Exp1:= Exp2

Exp1.err=Exp2.err

Exp1.tipo= Exp2.tipo

Exp2:= Exp2 op2 Exp3

Exp20.err= Exp21.err | Exp3.err| tipoBinario(op2, Exp21.tipo, Exp3.tipo)==error

Exp20.tipo= tipoBinario(op2, Exp21.tipo, Exp3.tipo)

Exp2:= Exp3

Exp2.err=Exp3.err

Exp2.tipo= Exp3.tipo

Exp3:= Exp4 op3 Exp3

Exp30.err= Exp31.err | Exp4.err| tipoBinario(op3, Exp4.tipo, Exp31.tipo)==error

Exp30.tipo= tipoBinario(op3, Exp4.tipo, Exp31.tipo)

Exp3:= Exp4

Exp3.err=Exp4.err

Exp3.tipo= Exp4.tipo

Exp4 := Exp4Asocia

Exp4.err = Exp4Asocia.err

Exp4.tipo = Exp4Asocia.tipo

Exp4 := Exp4NoAsocia

Exp4.err = Exp4NoAsocia.err

Exp4.tipo = Exp4NoAsocia.tipo

Exp4Asocia := Op4a Exp4

Exp4Asocia.err = Exp4.err | tipoUnario(Op4a,Exp4.tipo) == error

Exp4Asocia.tipo = tipoUnario(Op4a,Exp4.tipo)

Exp4Asocia := Exp5

Exp4Asocia.err = Exp5.err

Exp4Asocia.tipo = Exp5.tipo

Exp4NoAsocia := Op4na Exp5

Exp4NoAsocia.err = Exp5.err | tipoUnario(Op4na,Exp5.tipo) == error

Exp4Asocia.tipo = tipoUnario(Op4na,Exp5.tipo)

Exp4NoAsocia := Exp5

Exp4NoAsocia.err = Exp5.err

Exp4NoAsocia.tipo = Exp5.tipo

Exp5 := Iden

Exp5.err = !existeID(Exp5.tsh,iden.lex)

Exp5.tipo = Exp5.tsh[iden.lex].tipo

Exp5 := LiteralNatural

Exp5.err = 0

Exp5.tipo = Nat

Exp5 := LiteralReal

Exp5.err = 0

Exp5.tipo = Real

Exp5 := LiteralEntero

Exp5.err = 0

Exp5.tipo = Ent

Exp5 := LiteralBooleano

Exp5.err = 0

Exp5.tipo = Booleano

Exp5 := LiteralCaracter

Exp5.err = 0

Exp5.tipo = Caracter

Exp5 := ParAbierto Exp0 ParCerrado

Exp5.err = Exp0.err

Exp5.tipo = Exp0.tipo

ILectura := ResLee ParAbierto Iden ParCerrado

ILectura.err =!existeID(ILectura.tsh, Iden.lex)

ILectura.tipo = ILectura.tsh[Iden.lex].tipo

IEscritura := ResEscribe ParAbierto Exp0 ParCerrado

IEscritura.err = Exp0.err

lEscritura.tipo = Exp0.tipo

5. Especificación de la traducción

5.1. Lenguaje objeto y máquina virtual

Explicar cómo es el lenguaje objeto y cómo es la arquitectura de la máquina P capaz de ejecutarlo (tipos de celdas según tipos primitivos, etc.), su comportamiento interno y todo su repertorio de instrucciones (mostrando su sintaxis y una descripción informal de su semántica).

5.1.1. Arquitectura

La máquina virtual está compuesta por:

Pila --> Pila

Cima de pila --> CPila

Memoria de datos --> Mem

Memoria de programa --> Prog

Contador de programa -->PC

Pila

La pila de ejecución funciona como una estructura de datos tipo pila. Trabaja tomando los datos de la cima y la subcima y aplicando la operación correspondiente. El resultado se deja en la pila para posteriores cálculos o se guarda en memoria.

Puntero de pila

Es un valor natural que indica el número de elementos que hay en la pila y permite acceder al dato de la cima.

Memoria de datos

La función de la memoria de datos es almacenar el valor de las variables que se van a utilizar a lo largo de la ejecución del programa.

Esta se implementa con una estructura vectorial en la que cada posición de la misma almacenará únicamente un valor. (Cada posición puede ser de un tipo).

Memoria de programa

Almacena las instrucciones obtenidas (según el repertorio de instrucciones que se haya definido) tras el procesamiento del código fuente.

Contador de programa

Valor natural que indica la instrucción que se está ejecutando actualmente.

5.1.2. Comportamiento interno

1. PC toma la siguiente instrucción a ejecutar.
2. Se cargan en la pila los datos necesarios para la operación: apilando tanto direcciones de memoria como datos de los tipos definidos.
3. Se calcula el resultado.
4. Se almacena en memoria o se deja en la pila para posteriores operaciones.
5. Se aumenta PC.

5.1.3. Repertorio de instrucciones

ApilaNat (dato): Carga en la cima de la pila el valor de dato de tipo Natural

ApilaEntero (dato): Carga en la cima de la pila el valor de dato de tipo Entero

ApilaReal (dato): Carga en la cima de la pila el valor de dato de tipo Real

ApilaBool (dato): Carga en la cima de la pila el valor de dato de tipo Booleano

ApilaCar (dato): Carga en la cima de la pila el valor de dato de tipo Caracter

ApilaDir(dir): Obtiene el valor de memoria de la posición dir y lo carga en la cima.

DesapilaDir(dir): Almacena en la posición de memoria dir el valor de la cima y lo elimina de la pila.

Stdin (var): Obtiene el valor n de la entrada estándar , lo guarda en la dirección de la variable var.

Stdout (dato): Escribe por la salida estándar *dato*.

Menor, Mayor, MenorIgual ,MayorIgual,Igual,Distinto : Si la subcima es < | > | <= | >= | == | ¡= que la cima apila verdadero, falso en otro caso.

Or: Apila el resultado de realizar la o lógica de los valores booleanos de la cima y de la subcima.

And: Apila el resultado de realizar la and lógico de los valores booleanos de la subcima y la cima.

Suma: Apila la suma de la cima y de la subcima.

Resta: Apila la diferencia entre la subcima y la cima.

Multiplicación: Apila el producto de la cima y la subcima.

División: Apila el cociente entre la subcima y la cima.

Modulo: Apila el resto de dividir la subcima entre la cima.

CambioSigno: Invierte el signo de la cima de la pila y apila el resultado.

Not : apila el resultado de realizar el not lógico del valor binario de la cima.

CastEnt: Transforma la cima de la pila en un valor de tipo *entero*.

CastReal: Transforma la cima de la pila en un valor de tipo *real*.

CastNat: Transforma la cima de la pila en un valor de tipo *natural*.

CastCar: Transforma la cima de la pila en un valor de tipo *caracter*.

DesplazaDrcha : Apila el resultado de realizar el desplazamiento a derechas de los bytes indicados por la cima en la subcima de la pila.

DesplazaIzda: Apila el resultado de realizar el desplazamiento a izquierdas de los bytes indicados por la cima en la subcima de la pila.

5.2. Funciones semánticas

Descripción, si procede, de las funciones semánticas adicionales utilizadas en la especificación. Para cada función debe indicarse explícitamente su cabecera, así como informalmente su cometido, incluyendo el propósito de cada uno de sus parámetros.

Cadena1 '++' Cadena2 : cadena

Devuelve en cadena la concatenación de cadena1 y cadena2.

5.3. Atributos semánticos

Para cada categoría sintáctica relevante en este procesamiento deben enumerarse sus atributos semánticos, indicando si son heredados o sintetizados, y describiendo informalmente su propósito.

Cod : Atributo sintetizado que acumula el código generado

5.4. Gramática de atributos

Gramática de atributos que formaliza la traducción.

Prog:= ResPrograma idenPrograma Declaraciones Instrucciones

Prog.cod = Instrucciones.cod ++ stop

Instrucciones:= ResInstrucciones Is

Instrucciones.cod= Is.cod

Is:= Is PyC I

Is0.cod = Is1.cod ++ I.cod

Is:= I

Is.cod = I.cod

I:= IAsig

I.cod = IAsig.cod

IAsig:= Iden Igual Exp0

IAsig.cod = Exp0.cod ++ desapilaDir(tsh.dameDirIden(Iden.lex))

Exp0:= Exp1 op0 Exp1

Exp0.cod = Exp10.cod ++ Exp11.cod ++ op0

Exp0:=Exp1

Exp0 = Exp1.cod

Exp1:= Exp1 op1 Exp2

Exp10.cod = Exp11.cod ++ Exp2.cod ++ op1

Exp1:= Exp2

Exp1.cod = Exp2.cod

Exp2:= Exp2 op2 Exp3

Exp20.cod = Exp21.cod ++ Exp3.cod ++ op2

Exp2:= Exp3

Exp2.cod = Exp3.cod

Exp3:= Exp4 op3 Exp3

Exp30.cod = Exp4.cod ++ Exp31.cod ++ op3

Exp3:= Exp4

Exp3.cod = Exp4.cod

Exp4 := Exp4Asocia

Exp4.cod = Exp4Asocia.cod

Exp4 := Exp4NoAsocia

Exp4.cod = Exp4NoAsocia.cod

Exp4Asocia := Op4a Exp4

Exp4Asocia.cod = Exp4.cod ++ Op4a

Exp4Asocia := Exp5

Exp4Asocia.cod = Exp5.cod

Exp4NoAsocia := Op4na Exp5

Exp4NoAsocia.cod = Exp5.cod ++ Op4na

Exp4NoAsocia := Exp5

Exp4NoAsocia.cod = Exp5.cod

Exp5 := Iden

Exp5.cod = apilaDir(tsh.dameDirIden(Iden.lex))

Exp5 := LiteralNatural

Exp5.cod = apilaNat(NumNatural)

Exp5 := LiteralReal

Exp5.cod = apilaReal(NumReal)

Exp5 := LiteralEntero

Exp5.cod = apilaEntero(NumEntero)

Exp5 := LiteralBooleano

Exp5.cod = apilaBool(bool)

Exp5 := LiteralCaracter

Exp5.cod = apilaCar(car)

Exp5 := ParAbierto Exp0 ParCerrado

Exp5.cod = Exp0.cod

ILectura := ResLee ParAbierto Iden ParCerrado

Ilectura.cod = stdin(tsh.dameDirIden(Iden.lex)) ++ desapilaDir(tsh.dameDirIden(Iden.lex))

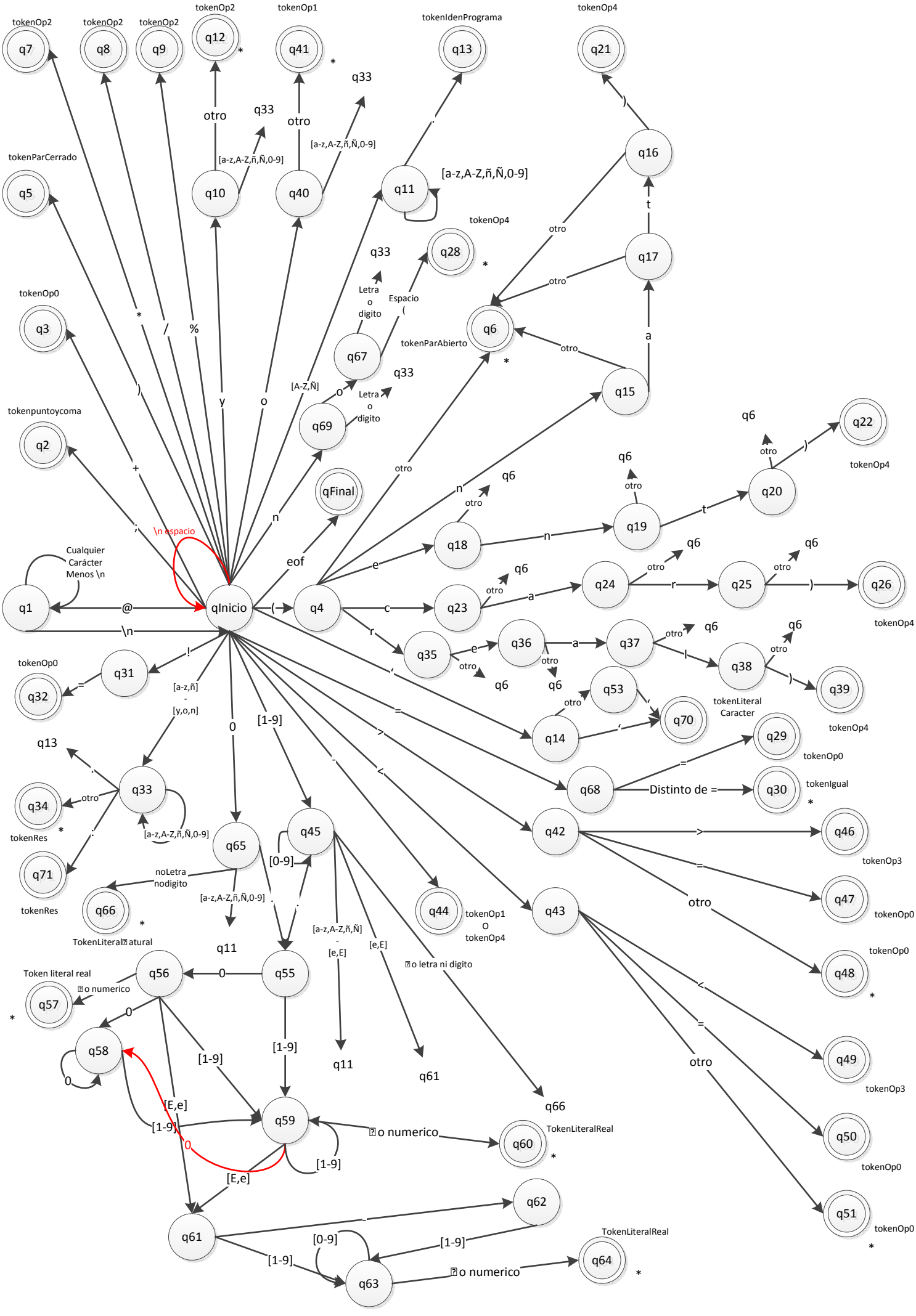
IEscritura := ResEscribe ParAbierto Exp0 ParCerrado

lEscritura.cod = Exp0.cod ++ stdout

6. Diseño del analizador léxico

Diagrama de transición que caracterice el diseño del analizador léxico. La implementación del analizador léxico debe estar guiada por este diseño.

En la siguiente página podemos observar el diagrama de transiciones del analizador léxico y los Token que devuelve en los estados de aceptación.


q1
Cualquier Carácter Menos \n
@
\n
qInicio
\n espacio
(
q4
eof
qFinal
;
+
)
*
/
%
y
o
n
q2
tokenpuntoycomma
q3
tokenOp0
q5
tokenParCerrado
q7
tokenOp2
q8
tokenOp2
q9
tokenOp2
q10
q12
tokenOp2
otro
[a-z,A-Z,ñ,Ñ,0-9]
q40
q41
tokenOp1
q33
[A-Z,Ñ]
q69
Letra o digito
q67
Espacio (
q28
tokenOp4
q11
tokenIdenPrograma
q13
q6
tokenParAbierto
q15
q16
q17
q21
tokenOp4
a
t
q18
n
q19
q20
q22
tokenOp4
q23
c
q24
r
q25
q26
tokenOp4
q35
q36
q37
q38
q39
tokenOp4
tokenLiteral Caracter
q53
q70
q14
q68
Distinto de =
q30
tokenIgual
q29
tokenOp0
q42
q46
tokenOp3
q47
tokenOp0
q48
tokenOp0
q49
tokenOp3
q50
tokenOp0
q51
tokenOp0
q43
q44
tokenOp1 O tokenOp4
q45
q31
!
=
q32
tokenOp0
[a-z,ñ] - [y,o,n]
0
[1-9]
q33
q34
tokenRes
q71
tokenRes
q65
noLetra nodigito
q66
TokenLiteral atural
[a-z,A-Z,ñ,Ñ,0-9]
[0-9]
[a-z,A-Z,ñ,Ñ] - [e,E]
[e,E]
o letra ni digito
q11
q56
q55
q57
Token literal real
o numerico
q58
q59
q60
TokenLiteralReal
q61
q62
q63
q64
TokenLiteralReal
[E,e]
[1-9]
0
otro

7. Acondicionamiento de las gramáticas de atributos

Transformaciones realizadas sobre las gramáticas de atributos para permitir la traducción predictivo-recursiva.

Únicamente deben incluirse la transformación de las producciones que se ven afectadas. Si alguna de las gramáticas no necesitan acondicionamiento, dejar el correspondiente subapartado en blanco.

7.1. Acondicionamiento de la gramática para la construcción de la tabla de símbolos

Decs := Dec RDecs

RDecs.tsh = añadeId(creaTS(),id: Dec.id,dir:0)
RDecs.dirh = 0 /*ultima posición ocupada*/
Decs.tss=RDecs.tss;
Decs.dirs = RDecs.dirs;

RDecs := PyC Dec RDecs

RDecs0.tss=RDecs1.tss;
RDecs0.dirs = RDecs1.dirs;
RDecs1.tsh = añadeID(RDecs0.tsh,Dec.id,RDecs0.dirh+1);
RDecs.dirh = RDecs0.dirh + 1;

RDecs := lambda

Rdecs.tss = RDecs.tsh

Is := I RIs

I.tsh = Is.tsh;
RIs.tsh = Is.tsh;

RIs := Pyc I RIs

I.tsh = RIs0.tsh;
RIs1.tsh = RIs0.tsh;

RIs := lambda //nada sobre la tabla de símbolos

Exp0 := Exp1 FExp1

Exp1.tsh = Exp0.tsh
FExp1.tsh = Exp0.tsh

FExp1 := op0 Exp1

Exp1.tsh = FExp1.tsh

FExp1 := lambda

Exp1 := Exp2 RExp1
Exp2.tsh = Exp1.tsh
RExp1.tsh = Exp1.tsh
RExp1 := op1 Exp1 RExp1
Exp1.tsh = RExp1.tsh
RExp11.tsh = RExp10.tsh
RExp1 := lambda

Exp3 := Exp4 Fexp3
Exp4.tsh = Exp3.tsh
FExp3.tsh = Exp3.tsh
FExp3 := op3 Exp3
Exp3.tsh = FExp3.tsh
FExp3 := lambda

7.2. Acondicionamiento de la gramática para la comprobación de las Restricciones Contextuales

Decs := Dec RDecs
Decs.err = Dec.err | RDecs.err
RDecs := Pyc Dec RDecs
RDecs0.err = Dec.err | RDecs1.err | existeID(RDecs0.tsh,Dec.id)
RDecs := lambda

Is := I RIs
Is.err = I.err | RIs.err
RIs := Pyc I RIs
RIs0.err = I.err | RIs.err
RIs := lambda

Exp0 := Exp1 FExp1
Exp0.err = Exp1.err | FExp1.err
Exp0.tipos = FExp1.tipos
FExp1.tipoh = Exp1.tipos
FExp1 := op0 Exp1
FExp1.err = Exp1.err | tipoBinario(op0,FExp1.tipoh,Exp1.tipo)== Error
FExp1.tipos = tipoBinario(op0,FExp1.tipoh,Exp1.tipo)

FExp1 := lambda

FExp1.err = false

Exp1 := Exp2 RExp1

Exp1.err = Exp2.err | RExp1.err

Exp1.tipos = RExp1.tipos

RExp1.tipoh = Exp2.tipos

RExp1 := op1 Exp1 RExp1

RExp11.tipoh = tipoBinario(op1,RExp0.tipoh,Exp1.tipo)

RExp10.tipos = RExp11.tipos

RExp10.err = Exp1.err | RExp11.err | tipoBinario(op1,RExp0.tipoh,Exp1.tipo) == error

RExp1 := lambda

RExp1.err = false

Exp3 := Exp4 FExp3

Exp3.err = Exp4.err | FExp3.err

Exp0.tipos = FExp3.tipos

FExp3.tipoh = Exp3.tipos

FExp3 := op3 Exp3

FExp3.err = Exp3.err | tipoBinario(op3,FExp3.tipoh,Exp3.tipo)== Error

FExp3.tipos = tipoBinario(op3,FExp3.tipoh,Exp3.tipo)

FExp3 := lambda

FExp3.err = false

7.3. Acondicionamiento de la gramática para la traducción

Is := I RIs

RIs.codh = I.cod

Is.cod = RIs.cod

RIs := Pyc I RIs

RIs1.codh = RIs0.codh ++ I.cod

RIs0.cod = RIs1.cod

RIs := lambda

RIs.cod = RIs.codh

Exp0 := Exp1 FExp1

FExp1.codh = Exp1.cod

Exp0.cod = FExp1.cod

FExp1 := op0 Exp1

FExp1.cod = FExp1.codh ++ Exp1.cod ++ op0

FExp1 := lambda

FExp1.cod = FExp1.codh

Exp1 := Exp2 RExp1

RExp1.codh = Exp2.cod

Exp1.cod = RExp1.cod

RExp1 := op1 Exp1 RExp1

RExp11.codh = RExp10.codh ++ Exp2.cod ++ op1

RExp0.cod = RExp1.cod

RExp1 := lambda

RExp1.cod = RExp1.codh

Exp2 := Exp3 RExp2

RExp2.codh = Exp3.cod

Exp2.cod = RExp2.cod

RExp2 := op2 Exp3 RExp2

RExp21.codh = RExp20.codh ++ Exp3.cod ++ op2

RExp20.cod = RExp21.cod

RExp2 := lambda

RExp2.cod = RExp2.codh

Exp3 := Exp4 FExp3

FExp3.codh = Exp4.cod

Exp3.cod = FExp3.cod

FExp3 := op3 Exp3

FExp3.cod = FExp3.codh ++ Exp3.cod ++ op3

FExp3 := lambda

FExp3.cod = FExp3.codh

8. Esquema de traducción orientado a las gramáticas de atributos

Esquema de traducción en el que se muestre, mediante acciones semánticas, los lugares en los que deben evaluarse las distintas ecuaciones contempladas en todas las gramáticas de atributos: la de construcción de la tabla de símbolos, la de comprobación de las restricciones contextuales y la de especificación de la traducción.

El recorrido del árbol será el realizado por un analizador predictivo-recursivo.

Prog -> ResPrograma idenPrograma **Declaraciones**

{Intrucciones.tsh = Declaraciones.tss}
Instrucciones
{Prog.err= Declaraciones.err | Instrucciones.err
Prog.cod = Instrucciones.cod ++ stop}

Declaraciones-> ResDeclaraciones **Decs**

{Declaraciones.tss = Decs.tss
Declaraciones.err= Decs.err
Declaraciones.dirs= Decs.dirs}

Instrucciones-> ResInstrucciones

{Is.tsh=Instrucciones.tsh}
Is
{Instrucciones.err= Is.err
Instrucciones.c
od= Is.cod}

Decs -> **Dec**

{RDecs.tsh = añadeId(creaTS(),id: Dec.id,dir:0)
RDecs.dirh = 0 /*ultima posicion ocupada*/
}
RDecs
{ Decs.tss=RDecs.tss;
Decs.dirs = RDecs.dirs;
Decs.err = Dec.err | RDecs.err
}

RDecs -> PyC **Dec**

{ RDecs1.tsh = añadeID(RDecs0.tsh,Dec.id,RDecs0.dirh+1);
RDecs1.dirh = RDecs0.dirh + 1;
}
RDecs
{RDecs0.tss=RDecs1.tss;
RDecs0.dirs = RDecs1.dirs;
RDecs0.err = Dec.err | RDecs1.err | existeID(RDecs0.tsh,Dec.id)
}

RDecs -> **lambda**

{Rdecs.tss = RDecs.tsh
}

Dec -> ResVariable **Iden** ResTipo **Tipo**

{Dec.err= ! (Tipo.tipo==real | Tipo.tipo==ent | Tipo.tipo==nat| Tipo.tipo==car | Tipo.tipo==booleano)
Dec.id= Iden.lex}

Tipo -> ResEntero

{Tipo.tipo= ent}

Tipo -> ResBooleano

{Tipo.tipo<- boleano}

Tipo-> ResCaracter

{Tipo.tipo<- car}

Tipo -> ResReal

{Tipo.tipo= real}

Tipo-> ResNatural

{Tipo.tipo= nat}

Is->

{I.tsh = Is.tsh;}
I
{ RIs.tsh = Is.tsh;}
{RIs.codh = I.cod }
RIs
{Is.err = I.err | RIs.err
Is.cod = RIs.cod}

RIs->

{ I.tsh = RIs0.tsh;}
PyC **I**
{ RIs1.tsh = RIs0.tsh;
RIs1.codh = RIs0.codh ++ I.cod }
RIs
{RIs0.err = I.err | RIs1.err
RIs0.cod = RIs1.cod}

RIs-> **lambda**

{RIs.cod = RIs.codh
RIs.err= 0}

I->

{IAsig.tsh = I.tsh**}**
IAsig
{I.err= IAsig.err
I.cod = IAsig.cod}

I->

{ILectura.tsh = I.tsh}
ILectura
{I.err= ILectura.err
I.cod = ILectura.cod}

I->

{IEscritura.tsh = I.tsh}
IEscritura
{I.err= IEscritura.err
I.cod = IEscritura.cod}

IAsig->

Iden Igual

{Exp0.tsh = IAsig.tsh}
Exp0
{IAsig.err= (!existeID(IAsig.tsh, Iden.lex) | Exp0.err |
tipoAsignacion(IAsig.tsh[Iden.lex].tipo, Exp0.tipo)==error)
IAsig.cod = Exp0.cod ++ desapilaDir(tsh.dameDirIden(Iden.lex))}

Exp0 ->

{Exp1.tsh = Exp0.tsh}
Exp1

{FExp1.tsh = Exp0.tsh
FExp1.tipoh = Exp1.tipos
FExp1.codh = Exp1.cod}
FExp1
{Exp0.err = Exp1.err | FExp1.err
Exp0.tipos = FExp1.tipos
Exp0.cod = FExp1.cod}

FExp1 := op0

{Exp1.tsh = FExp1.tsh}
Exp1
{FExp1.err = Exp1.err | tipoBinario(op0,FExp1.tipoh,Exp1.tipo)== Error
FExp1.tipos = tipoBinario(op0,FExp1.tipoh,Exp1.tipo)
FExp1.cod = FExp1.codh ++ Exp1.cod ++ op0.cod}

FExp1 := **lambda**

{FExp1.err = 0
FExp1.cod = FExp1.codh
FExp1.tipos=FExp1.tipoh}

Op0 -> Menor**|**Mayor**|**MenorIgual**|**MayorIgual**|**IgualIgual**|** Distinto

{Op0.cod= < | > | <= | >= | == | != (así se llaman las operaciones en la pila)}

Exp1 :=

{Exp2.tsh = Exp1.tsh}
Exp2
{RExp1.tsh = Exp1.tsh
RExp1.tipoh = Exp2.tipos
RExp1.codh = Exp2.cod}
RExp1
{Exp1.err = Exp2.err | RExp1.err
Exp1.tipos = RExp1.tipos
Exp1.cod = RExp1.cod}

RExp1 := **op1**

{Exp1.tsh = RExp10.tsh}

Exp1
{RExp11.tsh = RExp10.tsh
RExp11.tipoh = tipoBinario(op1,RExp10.tipoh,Exp1.tipo)
RExp11.codh = RExp10.codh ++ Exp1.cod ++ op1.cod}
RExp1

{RExp10.tipos = RExp11.tipos
RExp10.err = Exp1.err | RExp11.err |
tipoBinario(op1,RExp0.tipoh,Exp1.tipo) == error
RExp0.cod = RExp1.cod}

RExp1 := **lambda**

{RExp1.err = false

RExp1.cod = RExp1.codh
FExp1.tipos=FExp1.tipoh

Op1-> Mas|Menos|OR

{Op1.cod= + | - | Or}

Exp2 :=

{Exp3.tsh = Exp2.tsh}
Exp3
{RExp2.tsh = Exp2.tsh
RExp2.tipoh = Exp3.tipos
RExp2.codh = Exp3.cod}
RExp2
{Exp2.err = Exp3.err | RExp2.err
Exp2.tipos = RExp2.tipos
Exp2.cod = RExp2.cod}

RExp2 := **op2**

{Exp2.tsh = RExp20.tsh}
Exp2
{RExp21.tsh = RExp20.tsh
RExp21.tipoh = tipoBinario(op2,RExp20.tipoh,Exp2.tipo)
RExp21.codh = RExp20.codh ++ Exp2.cod ++ op2.cod}
RExp2
{RExp20.tipos = RExp21.tipos
RExp20.err = Exp2.err | RExp21.err | tipoBinario(op2,RExp20.tipoh,Exp2.tipo)
== error
RExp20.cod = RExp21.cod}

RExp2 := **lambda**

{RExp2.cod = RExp2.codh

RExp2.err=false

FExp1.tipos=FExp1.tipoh }

Op2-> Mult|Divide|Modulo|AND

{Op2.cod= * | / | % | And}

Exp3 :=

{Exp4.tsh = Exp3.tsh}
Exp4
{FExp3.tsh = Exp3.tsh
FExp3.tipoh = Exp4.tipos
FExp3.codh = Exp4.cod}
FExp3
{Exp3.err = Exp4.err | FExp3.err
Exp3.tipos = FExp3.tipos
Exp3.cod = FExp3.cod}

FExp3 :=

op3
{Exp3.tsh = FExp3.tsh}
Exp3
{FExp3.err = Exp3.err | tipoBinario(op3.cod,FExp3.tipoh,Exp3.tipo)== Error
FExp3.tipos = tipoBinario(op3.cod,FExp3.tipoh,Exp3.tipo)
FExp3.cod = FExp3.codh ++ Exp3.cod ++ op3.cod}

FExp3 := **lambda**

{FExp3.err = false
FExp3.cod = FExp3.codh
FExp1.tipos=FExp1.tipoh }

Op3 -> DespIzq**|** DespDcha

{Op3.cod= << | >>}

Exp4 ->

{Exp4Asocia.tsh = Exp4.tsh}
Exp4Asocia
{Exp4.err = Exp4Asocia.err
Exp4.tipo = Exp4Asocia.tipo
Exp4.cod = Exp4Asocia.cod}

Exp4 ->

{Exp4NoAsocia.tsh = Exp4.tsh
Exp4NoAsocia
{Exp4.err = Exp4NoAsocia.err
Exp4.tipo = Exp4NoAsocia.tipo
Exp4.cod = Exp4NoAsocia.cod}

Exp4Asocia ->

Op4a
{Exp4.tsh = Exp4Asocia.tsh}
Exp4
{Exp4Asocia.err = Exp4.err | tipoUnario(Op4a.cpd,Exp4.tipo) == error
Exp4Asocia.tipo = tipoUnario(Op4a.cod,Exp4.tipo)
Exp4Asocia.cod = Exp4.cod ++ Op4a.cod}

Exp4Asocia ->

{Exp5.tsh = Exp4Asocia.tsh}
Exp5
{Exp4Asocia.err = Exp5.err
Exp4Asocia.tipo = Exp5.tipo
Exp4Asocia.cod = Exp5.cod}

Op4a -> CambioSigno**|** Negacion

{Op4a.cod= cambio | not}

Exp4NoAsocia -> Op4na

{Exp5.tsh = Exp4NoAsocia.tsh}
Exp5
{Exp4NoAsocia.err = Exp5.err | tipoUnario(Op4na,Exp5.tipo) == error
Exp4NoAsocia.tipo = tipoUnario(Op4na,Exp5.tipo)
Exp4NoAsocia.cod = Exp5.cod ++ Op4na.cod}

Exp4NoAsocia ->

{Exp5.tsh = Exp4NoAsocia.tsh}
Exp5
{Exp4NoAsocia.err = Exp5.err
Exp4NoAsocia.tipo = Exp5.tipo
Exp4NoAsocia.cod = Exp5.cod}

Op4na -> ConvEnt**|** ConvReal**|** ConvCar**|** ConvNat

{Op4na.cod= (int) | (real) | (char) | (nat)}

Exp5 ->

Iden

{Exp5.err = !existeID(Exp5.tsh,iden.lex)
Exp5.tipo = Exp5.tsh[iden.lex].tipo
Exp5.cod = apilaDir(tsh.dameDirIden(Iden.lex))}

Exp5 -> LiteralNatural

{Exp5.err = 0
Exp5.tipo = Nat
Exp5.cod = apilaNat(LiteralNatural)}

Exp5 ->

ParAbierto
{Exp0.tsh = Exp5.tsh}
Exp0
ParCerrado
{Exp5.err = Exp0.err
Exp5.tipo = Exp0.tipo
Exp5.cod = Exp0.cod}

Ilectura -> ResLee ParAbierto Iden ParCerrado

{ILectura.err =!existeID(ILectura.tsh, Iden.lex)
ILectura.tipo = ILectura.tsh[Iden.lex].tipo
Ilectura.cod = stdin ++ desapilaDir(tsh.dameDirIden(Iden.lex))

Iescritura ->

ResEscribe ParAbierto
{Exp0.tsh = IEscritura.tsh}
Exp0
ParCerrado
{IEscritura.err = Exp0.err
lEscritura.tipo = Exp0.tipo
lEscritura.cod = Exp0.cod ++ stdout}

9. Esquema de traducción orientado al traductor predictivo – recursivo

Esquema de traducción en el que se haga explícito los parámetros utilizados para representar los atributos, así como en los que se muestre la implementación de las ecuaciones semánticas como asignaciones a dichos parámetros.

9.1. Variables globales

Se usan como variables globales el código y la tabla de símbolos, puesto que son los mismos para todas las producciones y se van modificando dentro de sus procedimientos asociados.

9.2. Nuevas operaciones y transformación de ecuaciones semánticas

Se ha introducido una nueva operación "emite", que añade a la variable global del programa que actúa como código, el código de la última instrucción generada.

9.3. Esquema de traducción

Prog (out err, out cod)::=

Reconoce(ResPrograma)
Reconoce(idenPrograma)
Declaraciones (out Declaraciones.tss,out Declaraciones.err, out Declaraciones.cod)
Instrucciones(in Declaraciones.tss,out Instrucciones.err, out Instrucciones.cod)
{Prog.err <- Declaraciones.err | Instrucciones.err
Prog.cod <- Instrucciones.cod ++ stop}

Declaraciones (out Declaraciones.tss, out Declaraciones.err, out Declaraciones.cod, out Declaraciones.dirs)::=

Reconoce (ResDeclaraciones)
Decs (out Decs.tss, out Decs.err, out Decs.cod,out Decs.dirs)
{Declaraciones.tss <- Decs.tss
Declaraciones.err<-Decs.err
Declaraciones.dirs= Decs.dirs }

Instrucciones (in Instrucciones.tsh,out Instrucciones.err, out Instrucciones.cod)::=

Reconoce(ResInstrucciones)
Is (in Instrucciones.tsh, out Is.err, out Is.cod)
{Instrucciones.err <- Is.err
Instrucciones.cod<- Is.cod}

Decs (out Decs.tss, out Decs.dirs, out Decs.err, out Decs.cod) ::=

Dec(out Dec.cod, out Dec.id, out Dec.error);
{RDecs.tsh <- añadeId(creaTS(),id: Dec.id,dir:0)
RDecs.dirh <- 0 /*ultima posicion ocupada*/
}
RDecs(in RDecs.tsh, In RDecs.dirh, in RDecs.codh,out RDecs.tss,out RDecs.dirs,out RDecs.err, out RDecs.cod);
{ Decs.tss<-RDecs.tss;
Decs.dirs <- RDecs.dirs;
Decs.err <- Dec.err | RDecs.err
}

RDecs(in RDecs0.tsh, In RDecs0.dirh, in RDecs0.codh,out RDecs0.tss,out RDecs0.dirs,out RDecs0.err, out RDec0s.cod)

Reconoce (PyC);

Dec(out Dec.cod, out Dec.id, out Dec.error);

{ RDecs1.tsh <- añadeID(RDecs0.tsh,Dec.id,RDecs0.dirh+1);
RDecs1.dirh <- RDecs0.dirh + 1;}
RDecs(in RDecs1.tsh, In RDecs1.dirh, in RDecs1.codh,out RDecs1.tss,out RDecs1.dirs,out RDecs1.err, out RDecs1.cod);
{RDecs0.tss<-RDecs1.tss;
RDecs0.dirs <-RDecs1.dirs;
RDecs0.err <-Dec.err | RDecs1.err | existeID(RDecs0.tsh,Dec.id)
}

RDecs (**in RDecs.tsh, In RDecs.dirh, in RDecs.codh,out RDecs.tss,out RDecs.dirs,out RDecs.err, out RDecs.cod)**

{Rdecs.tss <- RDecs.tsh
}

Dec (out Dec.err,out Dec.id, out Dec.cod)::=

Reconoce (ResVariable)
Reconoce (Iden)
Reconoce(ResTipo)
Tipo(out Tipo.tipo)
{Dec.err<- ! (Tipo.tipo==real | Tipo.tipo==ent | Tipo.tipo==nat|
Tipo.tipo==car | Tipo.tipo==booleano)
Dec.id= Iden.lex)}

Tipo(out Tipo.tipo) ::=

Reconoce(ResEntero)
{Tipo.tipo=<-ent}

Tipo(out Tipo.tipo) ::=

Reconoce(ResBooleano)
{Tipo.tipo<- booleano}

Tipo(out Tipo.tipo) ::=

Reconoce(ResCaracter)
{Tipo.tipo<- car}

Tipo(out Tipo.tipo) ::=

Reconoce(ResReal)
{Tipo.tipo<- real }

Tipo(out Tipo.tipo) ::=

Reconoce(ResNatural)
{Tipo.tipo<- nat }

Is (in Is.tsh, out Is.err, out Is.cod)::=

I (in Is.tsh, out I.err, out I.cod);
{ RIs.tsh <- Is.tsh;
RIs.codh <- I.cod }
RIs (in RIs.tsh, in Ris codh, out RIs.err, out RIs.cod)
{Is.err = I.err | RIs.err
Is.cod = RIs.cod}

RIs (in RIs0.tsh, in Ris0.codh, out RIs0.err, out RIs0.cod)::=

Reconoce (PyC);
I (in RIs0.tsh, out I.err, out I.cod);

{ RIs1.tsh = RIs0.tsh;
RIs1.codh = RIs0.codh ++ I.cod }
RIs (in RIs1.tsh, in RIs1. codh, out RIs1.err, out RIs1.cod)
{RIs0.err = I.err | RIs1.err
RIs0.cod = RIs1.cod}

RIs (in RIs.tsh, in Ris.codh, out RIs.err, out RIs.cod)::=

{RIs.cod = RIs.codh
RIs.err= 0}

I (in I.tsh, out I.err, out I.cod)::=

IAsig (in I.tsh, out IAsig.err, out IAsig.cod)

{I.err= IAsig.err
I.cod = IAsig.cod}

I (in I.tsh, out I.err, out I.cod)::=

ILectura (in I.tsh, out ILectura.err, out ILectura.cod)

{I.err= ILectura.err
I.cod = ILectura.cod}

I (in I.tsh, out I.err, out I.cod)::=

IEscritura (in I.tsh, out IEscritura.err, out IEscritura.cod)

{I.err= IEscritura.err
I.cod = IEscritura.cod}

IAsig (in IAsig.tsh, out IAsig.err, out IAsig.cod)::=

Reconoce(Iden)
Reconoce(Igual)
Exp0(in IAsig.tsh, out Exp0.tipo, out Exp0.err, out Exp0.cod)

{IAsig.err= (!existeID(IAsig.tsh, Iden.lex) | Exp0.err | tipoAsignacion(IAsig.tsh[Iden.lex].tipo, Exp0.tipo)==error)
IAsig.cod = Exp0.cod ++ desapilaDir(tsh.dameDirIden(Iden.lex))}

Exp0(in Exp0.tsh, out Exp0.tipos, out Exp0.err, out Exp0.cod)::=
Exp1(in Exp0.tsh, out Exp1.err, out Exp1.tipos, out Exp1.cod)
{FExp1.tsh = Exp0.tsh
FExp1.tipoh = Exp1.tipos
FExp1.codh = Exp1.cod}
FExp1(in FExp1.tsh,in FExp1.tipoh, in FExp1.codh, out FExp1.err, out FExp1.tipos, out FExp1.cod)
{Exp0.err = Exp1.err | FExp1.err
Exp0.tipos = FExp1.tipos
Exp0.cod = FExp1.cod}

FExp1(in FExp1.tsh,in FExp1.tipoh, in FExp1.codh, out FExp1.err, out FExp1.tipos, out FExp1.cod)::=

Op0(out op0.cod)
Exp1(in FExp1.tsh,out Exp1.tipo, out Exp1.err, out Exp1.cod)
{FExp1.err = Exp1.err | tipoBinario(op0.cod,FExp1.tipoh,Exp1.tipo)== Error
FExp1.tipos = tipoBinario(op0.cod,FExp1.tipoh,Exp1.tipo)
FExp1.cod = FExp1.codh ++ Exp1.cod ++ op0.cod}

FExp1(in FExp1.tsh,in FExp1.tipoh, in FExp1.codh, out FExp1.err, out FExp1.tipos, out FExp1.cod)::=
{FExp1.err = 0
FExp1.cod = FExp1.codh
FExp1.tipos=FExp1.tipoh }

Op0 (out op0.cod) ::=
{Op0.cod= < | > | <= | >= | == | !=}

Exp1(in Exp1.tsh,out Exp1.tipos, out Exp1.err, out Exp1.cod)::=

Exp2(in Exp1.tsh,out Exp2.tipos, out Exp2.err, out Exp2.cod)
{RExp1.tsh = Exp1.tsh
RExp1.tipoh = Exp2.tipos
RExp1.codh = Exp2.cod}
RExp1(in RExp1.tsh,in RExp1.tipoh, in RExp1.codh, out RExp1.err, out RExp1.tipos, out RExp1.cod)
{Exp1.err = Exp2.err | RExp1.err
Exp1.tipos = RExp1.tipos
Exp1.cod = RExp1.cod}

RExp1(in RExp10.tsh,in RExp10.tipoh, in RExp10.codh, out RExp10.err, out RExp10.tipos, out RExp10.cod)::=
Op1(out op1.cod)
Exp1(in RExp10.tsh,out Exp1.tipo, out Exp1.err, out Exp1.cod)
{RExp11.tsh = RExp10.tsh
RExp11.tipoh = tipoBinario(op1,RExp10.tipoh,Exp1.tipo)
RExp11.codh = RExp10.codh ++ Exp1.cod ++ op1.cod}
RExp1(in RExp11.tsh,in RExp11.tipoh, in RExp11.codh, out RExp11.err, out RExp11.tipos, out RExp11.cod)
{RExp10.tipos = RExp11.tipos
RExp10.err = Exp1.err | RExp11.err |
tipoBinario(op1.cod,RExp10.tipoh,Exp1.tipo) == error
RExp10.cod = RExp11.cod}

RExp1(in RExp1.tsh,in RExp1.tipoh, in RExp1.codh, out RExp1.err, out RExp1.tipos, out RExp1.cod)::=
{RExp1.err = false

RExp1.cod = RExp1.codh
FExp1.tipos=FExp1.tipoh }

Op1 (out Op1.cod)::=

{Op1.cod= + | - | Or}

Exp2(in Exp2.tsh,out Exp2.tipos, out Exp2.err, out Exp2.cod)::=

Exp3(in Exp2.tsh,out Exp3.tipos, out Exp3.err, out Exp3.cod)
{RExp2.tsh = Exp2.tsh
RExp2.tipoh = Exp3.tipos
RExp2.codh = Exp3.cod}
RExp2(in RExp2.tsh,in RExp2.tipoh, in RExp2.codh, out RExp2.err, out RExp2.tipos, out RExp2.cod)
{Exp2.err = Exp3.err | RExp2.err
Exp2.tipos = RExp2.tipos
Exp2.cod = RExp2.cod}

RExp2(in RExp20.tsh,in RExp20.tipoh, in RExp20.codh, out RExp20.err, out RExp20.tipos, out RExp20.cod)::=

Op2(out op2.cod)
Exp2(in RExp20.tsh,out Exp2.tipo, out Exp2.err, out Exp2.cod)
{RExp21.tsh = RExp20.tsh
RExp21.tipoh = tipoBinario(op2,RExp20.tipoh,Exp2.tipo)
RExp21.codh = RExp20.codh ++ Exp2.cod ++ op2.cod}
RExp2(in RExp21.tsh,in RExp21.tipoh, in RExp21.codh, out RExp21.err, out RExp21.tipos, out RExp21.cod)

RExp2(in RExp2.tsh,in RExp2.tipoh, in RExp2.codh, out RExp2.err, out RExp2.tipos, out RExp2.cod)::=

{RExp2.err = 0

RExp2.cod = RExp2.codh

FExp1.tipos=FExp1.tipoh }

Op2(out Op2.cod)::=

{Op2.cod= * | / | % | And}

Exp3(in Exp3.tsh,out Exp3.tipos, out Exp3.err, out Exp3.cod)::=

Exp4(in Exp3.tsh,out Exp4.tipos, out Exp4.err, out Exp4.cod)
{FExp3.tsh = Exp3.tsh
FExp3.tipoh = Exp4.tipos
FExp3.codh = Exp4.cod}
FExp3 (in FExp3.tsh,in FExp3.tipoh, in FExp3.codh, out FExp3.err, out FExp3.tipos, out FExp3.cod)
{Exp3.err = Exp4.err | FExp3.err
Exp3.tipos = FExp3.tipos
Exp3.cod = FExp3.cod}

FExp3 (in FExp3.tsh,in FExp3.tipoh, in FExp3.codh, out FExp3.err, out FExp3.tipos, out FExp3.cod)::=

Op3(out op3.cod)
Exp3(in FExp3.tsh,out Exp3.tipo, out Exp3.err, out Exp3.cod)
{FExp3.err = Exp3.err |
tipoBinario(op3.cod,FExp3.tipoh,Exp3.tipo)== Error
FExp3.tipos = tipoBinario(op3.cod,FExp3.tipoh,Exp3.tipo)
FExp3.cod = FExp3.codh ++ Exp3.cod ++ op3.cod}

FExp3 (in FExp3.tsh,in FExp3.tipoh, in FExp3.codh, out FExp3.err, out FExp3.tipos, out FExp3.cod)::=

{FExp3.err = false
FExp3.cod = FExp3.codh
FExp1.tipos=FExp1.tipoh }

Op3(out op3.cod)::=

{Op3.cod= << | >>}

Exp4 (in Exp4.tsh, out Exp4.err, out Exp4.tipo, out Exp4.cod)::=

Exp4Asocia (in Exp4.tsh, out Exp4Asocia.err, out Exp4 Asocia.tipo, out Exp4Asocia.cod)

{Exp4.err = Exp4Asocia.err
Exp4.tipo = Exp4Asocia.tipo
Exp4.cod = Exp4Asocia.cod}

Exp4 (in Exp4.tsh, out Exp4.err, out Exp4.tipo, out Exp4.cod)::=

Exp4NoAsocia (in Exp4.tsh, out Exp4NoAsocia.err, out Exp4NoAsocia.tipo, out Exp4NoAsocia.cod)

{Exp4.err = Exp4NoAsocia.err
Exp4.tipo = Exp4NoAsocia.tipo
Exp4.cod = Exp4NoAsocia.cod}

Exp4Asocia (in Exp4Asocia.tsh, out Exp4Asocia.err, out Exp4 Asocia.tipo, out Exp4Asocia.cod)::=

Op4a(out op4a.cod);
Exp4 (in Exp4Asocia.tsh, out Exp4.err, out Exp4.tipo, out Exp4.cod)
{Exp4Asocia.err = Exp4.err | tipoUnario(Op4a,Exp4.tipo) == error
Exp4Asocia.tipo = tipoUnario(Op4a,Exp4.tipo)
Exp4Asocia.cod = Exp4.cod ++ Op4a.cod}

Exp4Asocia (in Exp4Asocia.tsh, out Exp4Asocia.err, out Exp4 Asocia.tipo, out Exp4Asocia.cod)::=

Exp5 (in Exp4Asocia.tsh, out Exp5.err, out Exp5.tipo, out Exp5.cod)
{Exp4Asocia.err = Exp5.err
Exp4Asocia.tipo = Exp5.tipo
Exp4Asocia.cod = Exp5.cod}

Op4a(out op4a.cod)::=

{Op4a.cod= cambio | not}

Exp4NoAsocia (in Exp4NoAsocia.tsh, out Exp4NoAsocia.err, out Exp4NoAsocia.tipo, out Exp4NoAsocia.cod)::=

Op4na(out op4na.cod)

Exp5 (in Exp4NoAsocia.tsh, out Exp5.err, out Exp5.tipo, out Exp5.cod)
{Exp4NoAsocia.err = Exp5.err | tipoUnario(Op4na,Exp5.tipo) == error
Exp4NoAsocia.tipo = tipoUnario(Op4na,Exp5.tipo)
Exp4NoAsocia.cod = Exp5.cod ++ Op4na.cod}

Exp4NoAsocia (in Exp4NoAsocia.tsh, out Exp4NoAsocia.err, out Exp4NoAsocia.tipo, out Exp4NoAsocia.cod)::=

Exp5 (in Exp4NoAsocia.tsh, out Exp5.err, out Exp5.tipo, out Exp5.cod)
{Exp4NoAsocia.err = Exp5.err
Exp4NoAsocia.tipo = Exp5.tipo
Exp4NoAsocia.cod = Exp5.cod}

Op4na(out op4na.cod)::=

{Op4na.cod= (int) | (real) | (char) | (nat)}

Exp5 (in Exp5.tsh, out Exp5.err, out Exp5.tipo, out Exp5.cod)::=

Reconoce(Iden **)**
{Exp5.err = !existeID(Exp5.tsh,iden.lex)
Exp5.tipo = Exp5.tsh[iden.lex].tipo
Exp5.cod = apilaDir(tsh.dameDirIden(Iden.lex))}

Exp5 (in Exp5.tsh, out Exp5.err, out Exp5.tipo, out Exp5.cod)::=

Reconoce(LiteralNatural)
{Exp5.err = 0
Exp5.tipo = Nat
Exp5.cod = apilaNat(LiteralNatural)}

Exp5 (in Exp5.tsh, out Exp5.err, out Exp5.tipo, out Exp5.cod)::=

Reconoce(ParAbierto);

Exp0(in Exp5.tsh, out Exp0.err, out Exp0.tipo,out Exp0.cod)

Reconoce(ParCerrado);

{Exp5.err = Exp0.err
Exp5.tipo = Exp0.tipo
Exp5.cod = Exp0.cod}

ILectura(in ILectura.tsh, out ILectura.err, out ILectura.tipo, out ILectura.cod)::=

Reconoce(ResLee);
Reconoce(ParAbierto);
Reconoce(Iden);
Reconoce(ResParCerrado);

{ILectura.err =!existeID(ILectura.tsh, Iden.lex)
ILectura.tipo = ILectura.tsh[Iden.lex].tipo
Ilectura.cod stdin ++ desapilaDir(tsh.dameDirIden(Iden.lex))

IEscritura(in IEscritura.tsh, out IEscritura.err, out IEscritura.tipo, out IEscritura.cod)::=

Reconoce(ResEscribe);
Reconoce(ParAbierto);
Exp0(IEscritura.tsh, out Exp0.err, out Exp0.tipo, out Exp0.cod);
Reconoce(ResParCerrado);

{IEscritura.err = Exp0.err
lEscritura.tipo = Exp0.tipo
lEscritura.cod = Exp0.cod ++ stdout}

10. Formato de representación del código P

Deberá describirse el formato de archivo aceptado por el intérprete de la máquina P, llamado código a pila (código P). Se valorará la eficiencia de dicho formato (por ejemplo: uso de *bytecode* binario en lugar de texto).

La máquinaP necesita para comenzar la ejecución una lista de instrucciones (en la implementación ArrayList<Instrucción>)y una memoria de datos con las variables que fueron declaradas (se añadieron con éxito en la tabla de símbolos en la etapa de traducción de las declaraciones y es un ArrayList<Dato<T>>). La etapa de traducción genera ambas cosas y las almacena en un fichero binario. Para ello, todas las instrucciones y datos implementan la interfaz de Java Serializable, que facilita la escritura y lectura de cualquier objeto que la haya implementado.

11 Notas sobre la implementación

Descripción de la implementación realizada.

Deberán implementarse dos programas separados:

1. Traductor del lenguaje fuente a código de una máquina P. Este programa funcionará en modo *línea de comandos*, tomando dos argumentos: el nombre del archivo que contiene el programa fuente y el nombre del archivo donde se va a generar el código objeto. El resultado dependerá de si el programa fuente es o no correcto:
   - Si es correcto, se generará el archivo con el código objeto
   - Si no es correcto, se mostrará por pantalla un listado de errores y no se generará ningún archivo
     - Cada error estará precedido por los números de fila y de columna que identifican el punto exacto en el programa fuente en el que se ha producido el error
     - La ejecución del programa se interrumpe en el momento en que se encuentra el primer error sintáctico. El resto de los errores (léxicos y de violación de restricciones contextuales) no interrumpen la ejecución del traductor
2. Intérprete trivial capaz de simular el comportamiento de la máquina P. Este programa también funcionará en modo *línea de comandos*, tomando como argumento el archivo donde está el código P a ejecutar, así como un valor que indicará si la ejecución debe ser en *modo normal* o en *modo traza*. El resultado será la ejecución de dicho código:
   - En *modo normal* el único resultado visible será el producido por las instrucciones específicas de lectura/escritura
   - En *modo traza* el intérprete mostrará una traza por la consola de comandos con los distintos pasos de ejecución. En cada paso el intérprete debe:
     1. Mostrar el contenido de la pila de ejecución y de las celdas relevantes en la memoria de datos
     2. Esperar a que el usuario pulse una tecla para proseguir la ejecución
     3. Mostrar la instrucción a ejecutar y ejecutarla, avanzando al siguiente paso

Deberán entregarse un conjunto de archivos conteniendo los casos de prueba (programas de ejemplo) utilizados en la validación de la implementación, incluyendo tanto casos correctos como incorrectos, para demostrar que los programas implementados son capaces de detectar y tratar todos los tipos de errores posibles de una forma adecuada a los requisitos mencionados anteriormente.

11.1. Descripción de archivos

Enumeración de los archivos con el código fuente de la implementación, y descripción de lo que contiene cada archivo.

Nuestra aplicación consta de seis paquetes:

1. Analizador Léxico:

   Contiene la implementación del autómata reconocedor de Tokens, con todos y cada uno de los estados, y todos y cada uno de los Tokens que puede generar.

2. Compilador:

   Contiene el programa principal que realiza el analizador sintáctico y el parser, y muestra por pantalla el código generado.

3. Común:

   Conjunto de clases que se utilizan en diversos apartados, ya sea para el reader, el parser, o la máquina virtual.

4. Máquina:

   Programa principal que ejecuta el código recibido por parámetro. Contiene además la implementación de la pila de trabajo y la definición de los datos que se puede encontrar en ella.

5. Parser:

   Realiza el análisis sintáctico de la gramática.

6. Tabla Símbolos:

   Contiene la implementación de la tabla de símbolos y su contenido.

11.2. Otras notas

Otras notas sobre la implementación que se consideren pertinentes (por ejemplo: diagramas de clase UML describiendo la arquitectura del sistema).

Conclusiones

Qué se ha conseguido y qué se ha dejado pendiente para más adelante.

En la implementación de esta primera parte de la práctica, hemos conseguido extraer los tokens correspondientes dado un fichero de texto, analizando además si se encontrase algún error léxico en él. Además, también hemos parseado sintácticamente los tokens recibidos de tal manera que realiza un recorrido por toda la gramática, totalmente acondicionada, analizando en el mismo, los errores debidos a restricciones contextuales y errores debidos al análisis sintáctico.

Una vez hemos generado el código objeto, contruyendo una máquina P, hemos ejecutado consecuentemente el código del mismo, obteniendo el resultado esperado después de cada una de las ejecuciones.

En definitiva, se han conseguido todos los objetivos propuestos para dicha práctica.