# Tema 2:
# Descripción Formal de los Lenguajes de Programación (parte I)

Lenguajes y Paradigmas de Programación

# Sintaxis y Semántica

- Sintaxis: qué secuencia de caracteres constituyen un programa “legal”
  - elementos sintácticos del lenguaje
  - modelos de ejecución
- Semántica: qué significa (qué calcula) un programa legal dado
  - concepto y necesidad de las descripciones semánticas
  - tipos de semánticas
  - equivalencia de programas
  - corrección y completitud

# Sintaxis (1)

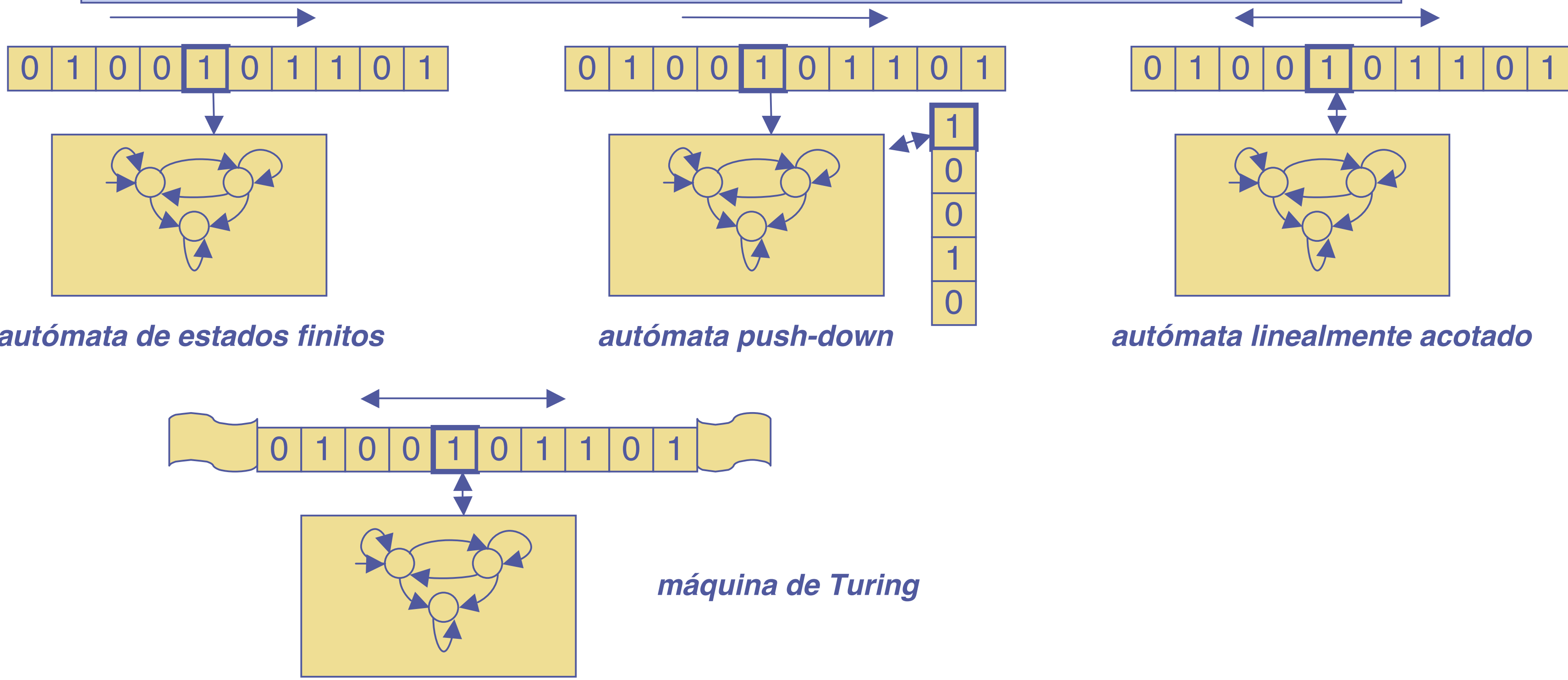
- La sintaxis define "el aspecto" de un programa

  qué programas son "legales" o conformes con la especificación de un lenguaje
- Distintos instrumentos:
  - Gramáticas generativas (Chomsky 0 ⊃ 1 ⊃ 2 ⊃ 3)
  - Autómatas (jerarquía)
  - Notación BNF – Algol
  - Representación gráfica (con diagramas) –Pascal
- La gramática describe cómo formar instrucciones (*statements*) a partir de palabras (*tokens*)
  - Una instrucción = una secuencia de palabras
  - Una palabra = una secuencia de caracteres

    Las reglas de la gramática describen tanto:
    - las instrucciones (gramática de contexto libre/autóm. *push-down*)
    - las palabras (gramática regular/autóm. de estados finitos)

# Sintaxis: fundamentos teóricos

| Nivel Chomsky | Clase gramática | Clase máquina |
| --- | --- | --- |
| 0 | sin restricciones | máquina de Turing |
| 1 | sensible al contexto | autómata linealmente acotado |
| 2 | contexto libre | autómata push-down |
| 3 | regular | autómata de estados finitos |



# Sintaxis (2)

- Una gramática de contexto libre es una 4-tupla (NT,T,S,R) donde
  - NT es un conjunto finito de símbolos no terminales
  - T es un conjunto finito de símbolos terminales

    (símbolos que no se definen; en un lenguaje de programación serán símbolos predefinidos del lenguaje, por ejemplo los números naturales 0 1 2…, los operadores aritméticos, etc)
  - $S \in NT$ es el símbolo inicial
  - R es un conjunto de reglas (producciones) del tipo

$$A \Rightarrow b$$

donde:

A es un símbolo de NT y

b es una secuencia de (cero o más) símbolos de $(NT \cup T)$

- Tipo 3: A es un NT y b cualquier cadena de T y NT sin recursión
- Tipo 2: A es un NT y b cualquier cadena de T y NT
- Tipo: A es cualquier cadena de símbolos NT y b cualquier cadena de T y NT, con A<b
- tipo 0: A es cualquier cadena de símbolos NT y b cualquier cadena de T y NT

# gramática ejemplo

- No terminales : {**frase**, **sujeto**, **artículo**, **nombre**,**predicado**}
- Terminales: {un, león, conejo, come, persigue, a}
- Símbolo Inicial: **frase**
- Reglas:

  **frase** ⇒ **sujeto verbo complemento**
  **sujeto** ⇒ **artículo nombre**
  **verbo** ⇒ come
  **verbo** ⇒ persigue
  **complemento** ⇒ **preposición artículo nombre**
  **preposición** ⇒ a
  **artículo** ⇒ un
  **nombre** ⇒ león
  **nombre** ⇒ conejo

- Ejemplo de producción:

  **frase** ⇒ **sujeto verbo complemento** (1ª regla)
  ⇒ **artículo nombre verbo complemento** (2ª regla)
  ⇒ un **nombre verbo complemento** (7ª regla) …
  ⇒ un león persigue a un conejo

# Sintaxis (3) Notación BNF

- BNF es esencialmente una notación para expresar las gramáticas de contexto libre que definen la sintaxis de un LP:

  - Se usa la notación A::=b en vez de A ⇒ b
  - Se usa <w> para indicar un símbolo no terminal (w es cualquier secuencia de caracteres)
  - En la versión "BNF extendida", que es la más popular, se usa

    A::= b I c

    para representar las dos reglas: (el símbolo I significa 'or')

    A::=b

    A::=c

Ejemplo: la siguiente regla define el símbolo no terminal <digit>

<digit>!::= 0I1I2I3I4I5I6I7I8I9

i.e., <digit> es uno de los diez terminales 0,1,2….,9

# Sintaxis (4) Notación BNF

## Otras convenciones de la “BNF extendida”:

- Paréntesis cuadrados "[..]" alrededor de los items opcionales

```
<if_statement> ::=  if <boolean_expression> then
                        <statement_sequence>
                    [ else
                        <statement_sequence> ]
                    endif “;”
```

- Llaves {} para indicar una secuencia de 0 o más items
  (a veces se usa el sufijo * de Kleene)

```
<identifier> ::= <letter> { <letter> | <digit> }

    equivalente a la regla recursiva:
<identifier> ::= <letter> |
                 <identifier> [ <letter> | <digit> ]
```

- Sufijo + para indicar una secuencia de 1 o más items

<while_statement> ::= WHILE <expression> DO
<statement> { ";" <statement>}
END

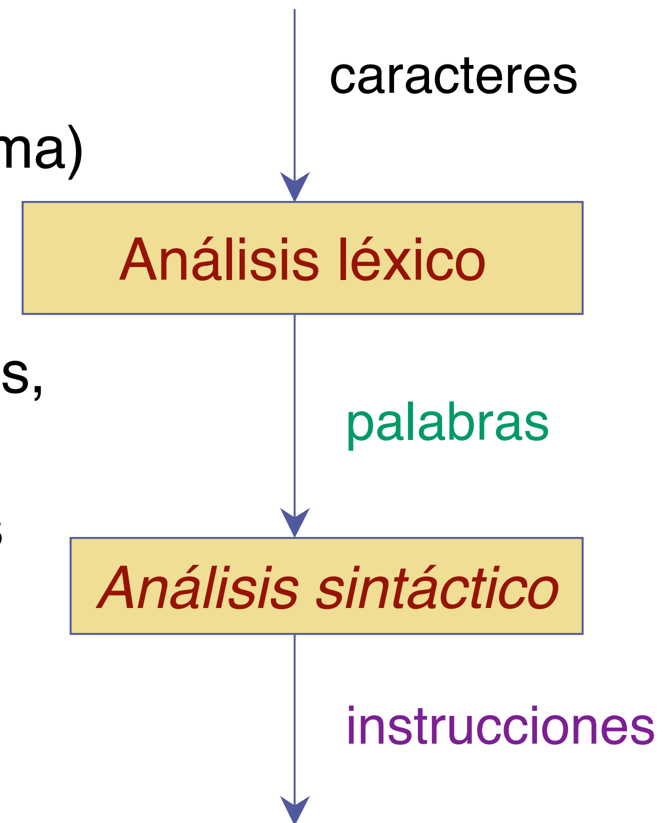
while_statement

WHILE
expression
DO
statement
;
END

# Sintaxis (5)

- Un analizador léxico (scanner) es un programa que divide una secuencia de caracteres (el programa) en una secuencia de componentes sintácticos primitivos

  o palabras (identificadores, números, palabras reservadas, etc)
- Un analizador sintáctico (parser) es un programa que reconoce una secuencia de palabras y construye una secuencia de instrucciones (en forma de árbol sintáctico)



# ejemplo

```
[f,u,n,{,F,a,c,t,' ',N,},'\n',' ',i,f,' ',N,=,=,0, '
',t,h,e,n,' ',1,'\n',' ',e,l,s,e,' ',N,*,{,F,a, c,t,' ',N,-
,1,},' ',e,n,d,i,f,'\n',e,n,d]
```
= secuencia de caracteres

```
[fun,{,Fact,N,},if,N,= =,0,then,1,else,N,*,{,Fact,N,
-,1,},endif,end] ]
```
= secuencia de palabras

```
fun {Fact N}
       if N = = 0 then 1
                   else N* {Fact N-1}
     endif
end
```
= instrucción

Si consideramos la sentencia

```
coste = precio * 0.98;
```

El analizador léxico generaría una tabla de símbolos:

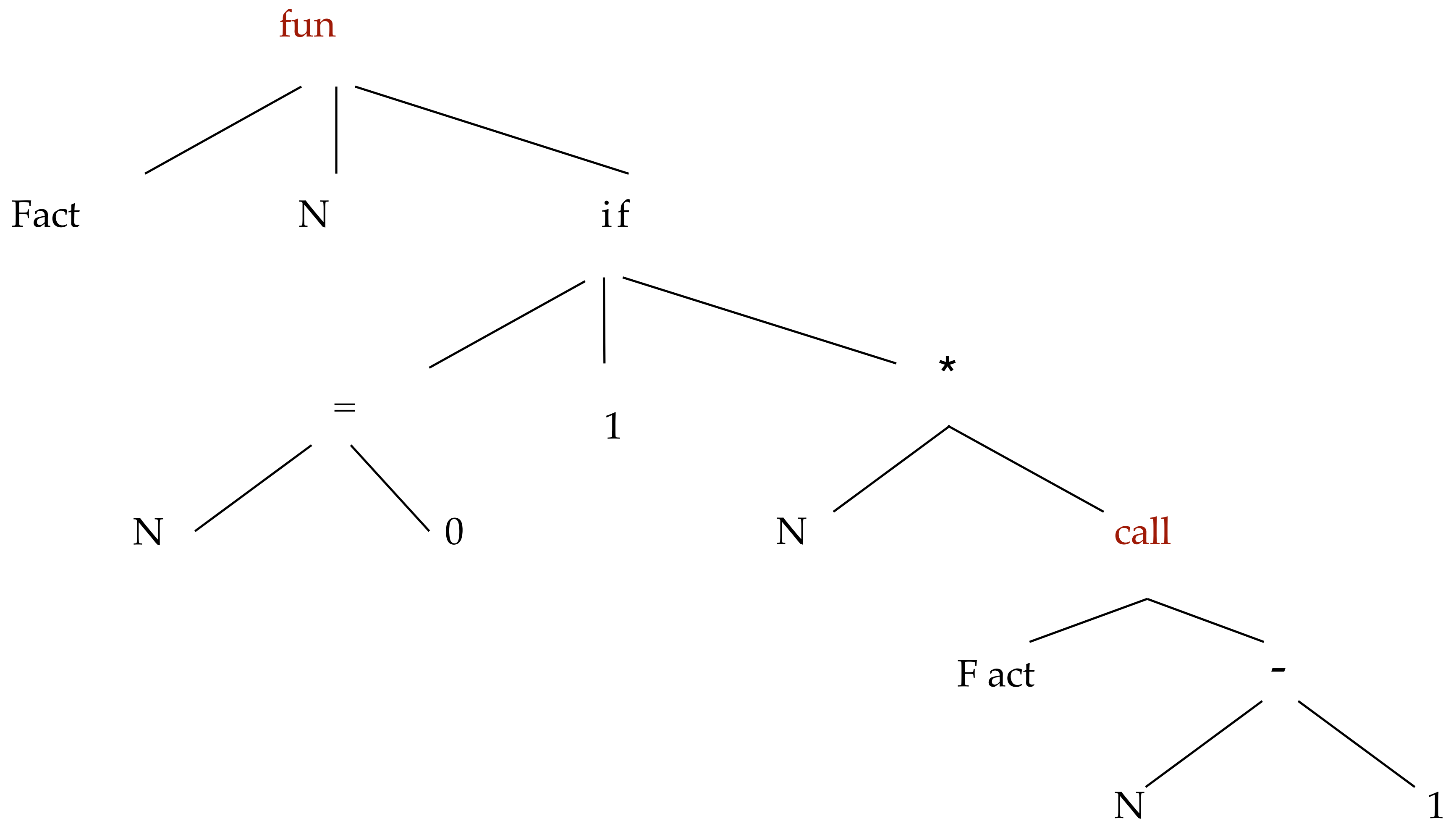
1 coste …

2 precio …

3 0.98 …

y una lista de pares

[id,1] [=, ] [id,2] [*, ] [C,3]


fun
Fact
N
if
=
1
*
N
0
N
call
Fact
-
N
1

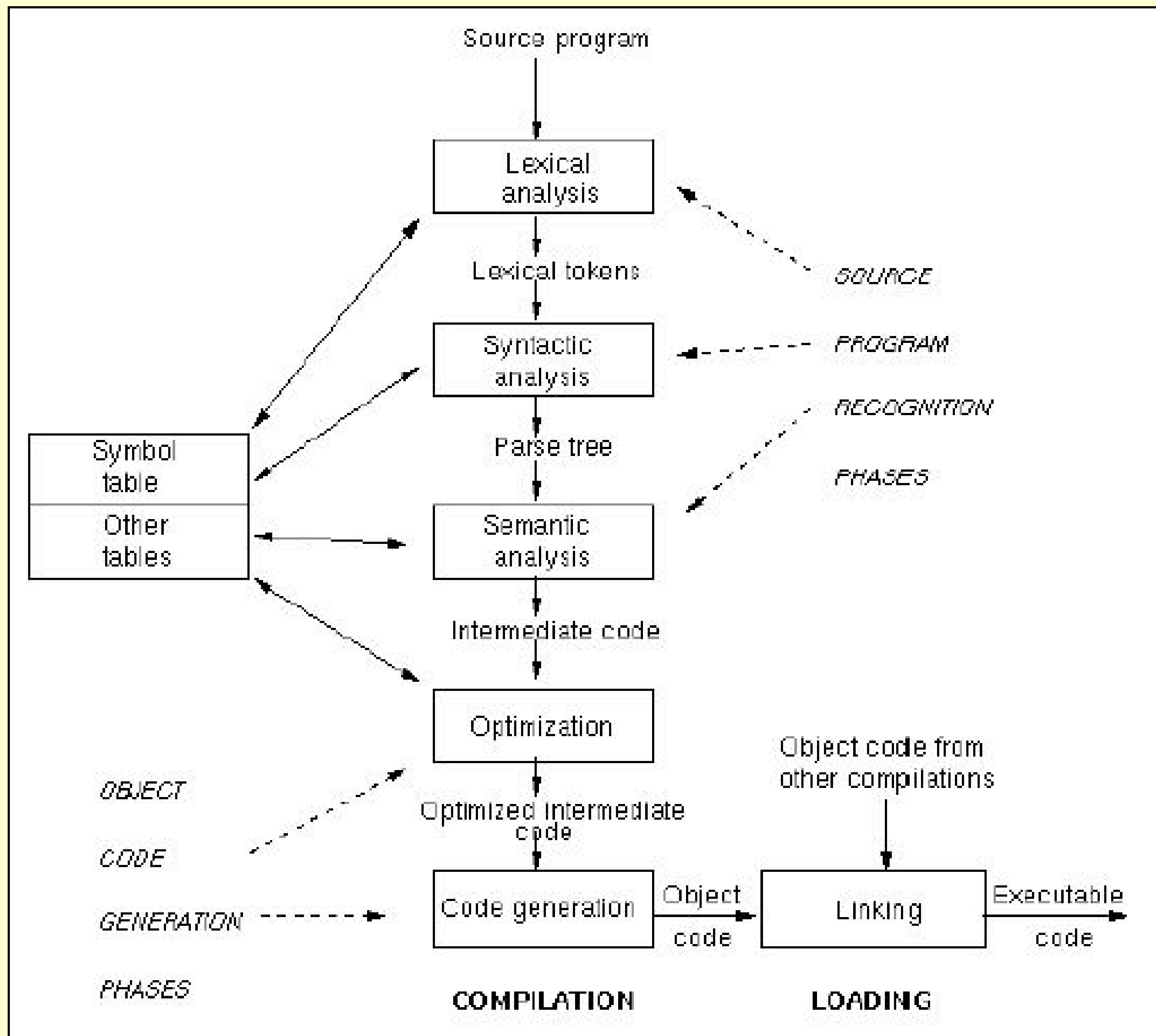
# Semántica (I)

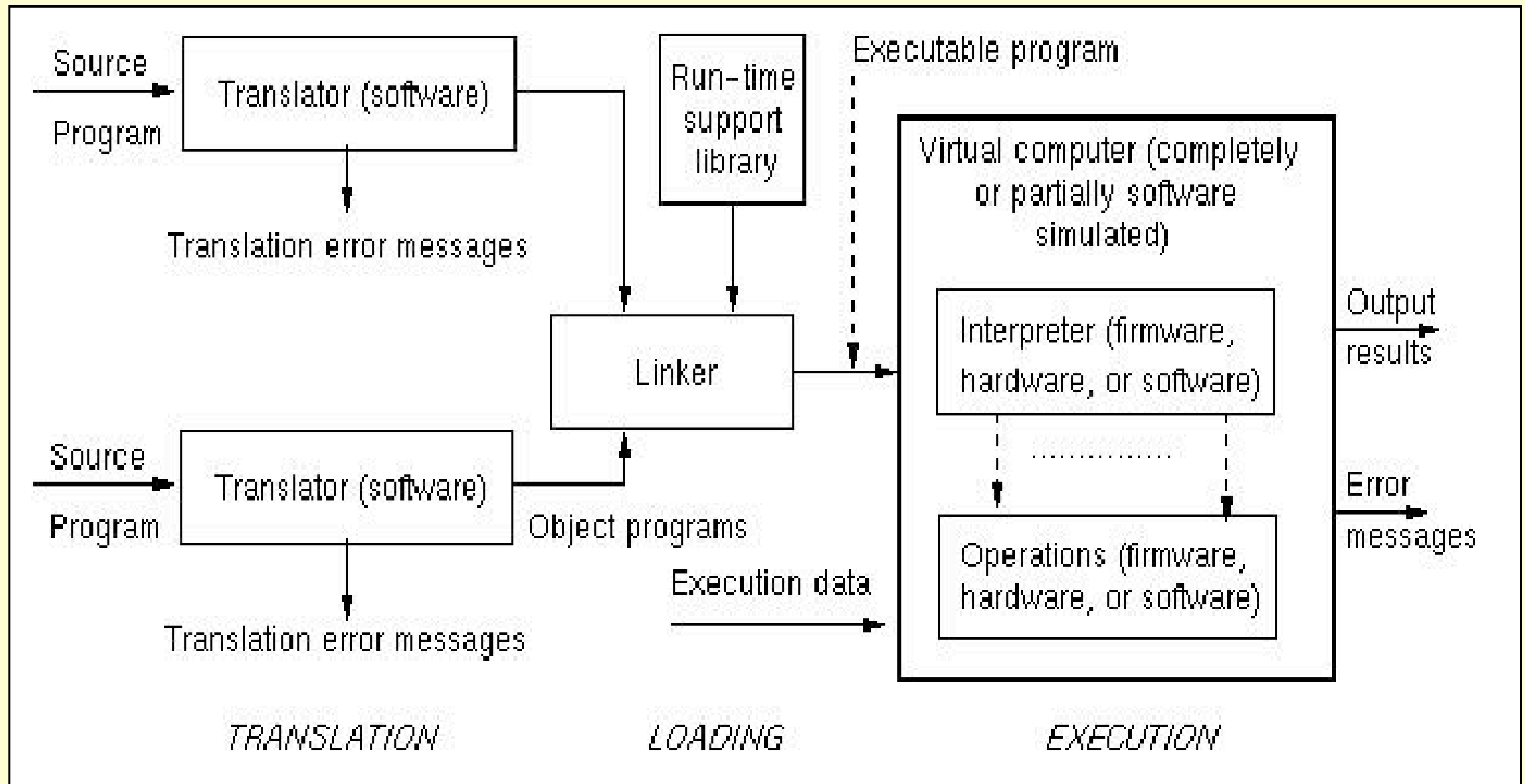
## Concepto y necesidad de las descripciones semánticas

- Las sintaxis de contexto libre (BNF) no son suficientes para describir ciertas reglas:
  - compatibilidad de tipos,
  - alcance de las variables
  - signatura de las funciones (coincidencia del número de parámetros en una llamada con los parámetros formales)

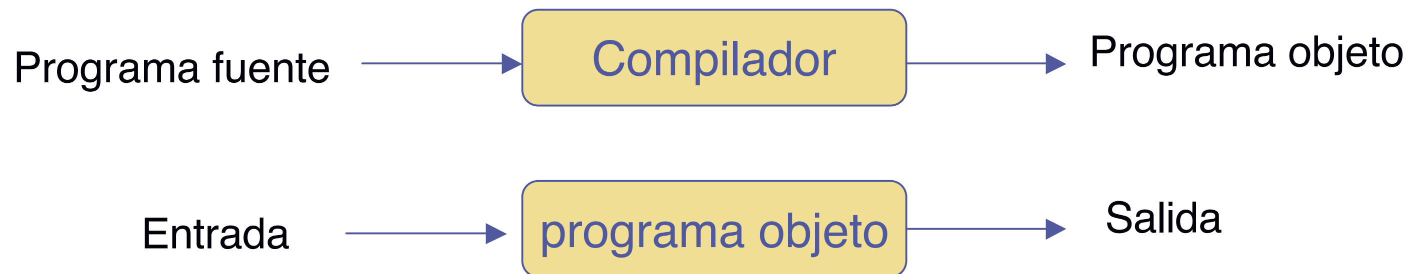
  ⇒ estas reglas forman parte de la semántica "estática" del lenguaje (*la que se puede determinar en tiempo de compilación*)
- Por ejemplo A := B + C

  podría no ser legal si A, B o C no han sido declaradas previamente
- Comprobaciones que lleva a cabo el análisis semántico
  1. Comprobación y conversión de tipos (coerciones)
  2. Declaración de variables previa a su uso, etc


Source program
Lexical analysis
Lexical tokens
Syntactic analysis
Parse tree
Semantic analysis
Intermediate code
Optimization
Optimized intermediate code
Code generation
Object code
Linking
Executable code
Object code from other compilations
Symbol table
Other tables
SOURCE
PROGRAM
RECOGNITION
PHASES
OBJECT
CODE
GENERATION
PHASES
COMPILATION
LOADING


Source
Program
Translator (software)
Translation error messages
Run-time
support
library
Executable program
Virtual computer (completely
or partially software
simulated)
Linker
Interpreter (firmware,
hardware, or software)
Output
results
Source
Program
Translator (software)
Object programs
Error
messages
Operations (firmware,
hardware, or software)
Execution data
Translation error messages
TRANSLATION
LOADING
EXECUTION

- Lenguajes compilados




- Lenguajes interpretados

# Semántica (2)

## Porqué la semántica no es siempre “estática”?

¿Por qué no es suficiente considerar sólo la semántica estática, es decir, por qué no es posible que el compilador detecte todos los errores posibles?:

Algunos errores sólo se manifiestan durante la ejecución:

Z=X/Y error si se ejecuta con un valor de Y = 0
Z=V[Y] error si Y tiene un valor que cae fuera del rango del vector V

De hecho, muchas propiedades interesantes de un programa (escrito en un lenguaje Turing-completo) no son decidibles. Por ejemplo , no se puede decidir:

1. la terminación (semidecidible… basta ejecutar el programa para “semi-decidirlo”)
2. si dos programas cualesquiera computan la misma función
3. si dos gramáticas de contexto libre generan el mismo lenguaje

# Semántica (3)

- La semántica sirve para definir exactamente “qué hace” un programa (qué “computa” )
- La definición (semi-)formal de un lenguaje es parte del estándar de definición de un LP desde los años 80

- Además de ayudar al programador a “razonar” sobre el programa (recursos usados, corrección, …), es necesaria para implementar correctamente el lenguaje, y sirve para desarrollar técnicas y herramientas de:
  - Análisis y Optimización
  - Depuración
  - Verificación
  - Transformación
- No existe una metodología estándar aceptada universalmente, existen distintos “estilos”

# Semántica (4)

- Estilos de definición semántica
  - Operacional
  - Axiomática
  - Declarativa
    - Algebraica
    - Teoría de Modelos
    - Punto fijo
    - Denotacional

# Semántica operacional (i)

- es el enfoque más antiguo
  (con este estilo se definió la semántica de ALGOL'60)

- primero se define una máquina abstracta M, y el significado de cada construcción se expresa en términos de las acciones a realizar por la máquina

- la forma má simple de definirla es proporcionar un intérprete para el lenguaje L sobre la máquina M cuyas componentes se describen de modo matemático

- la definición semántica operacional de un lenguaje lo hace ejecutable

- proporciona un modelo para la implementación

# Semántica operacional (ii)

- ejemplo: *Structural Operacional Semantics* (SOS) (o sistemas de transición de Plotkin)

- se de finen reglas de transición que especifican los pasos de computación para una construcción compuesta A op B en términos de la semántica de las componentes

- las reglas se suelen escribir en el estilo de los sistemas de *deducción natural*

$$\frac{\textit{premisa}}{\textit{conclusión}}$$

# Semántica operacional (iii)

- Dado que las construcciones en general modifican una cierta noción de estado, las reglas de transición se suelen definir sobre configuraciones del tipo

  <Instrucción, Estado>

- las reglas de transición tienen entonces la forma:

$$\frac{\text{premisa}}{\langle i,e\rangle \rightarrow \langle i',e'\rangle}$$

  indicando que una configuración en otra, cuando se satisface cierta premisa

- el conjunto de estas reglas define una relación de transición (que llamaremos $\rightarrow$) sobre el conjunto de las configuraciones

# Semántica operacional (iv)

- Formalmente, un ST es una 4-tupla $(C,I,F,\rightarrow)$, donde
  - C es el conjunto de las configuraciones c, que son pares de la forma <i,e>
  - $I \subseteq C$ es el conjunto de las configuraciones iniciales
  - $F \subseteq C$ es el conjunto de las configuraciones finales
  - $\rightarrow \subseteq C \times C$ es la relación de transición (escribiremos $c \rightarrow c'$ para indicar que el par $(c,c') \in \rightarrow$

- Una secuencia de ejecución es una secuencia de configuraciones $c_1\ c_2\ \ldots\ c_n$ tal que
  - $c_1 \in I$
  - $c_n \in F$
  - $c_i \rightarrow c_{i+1}$, para cada i en [0..n[

- Un ST se llama determinista si para cada $c \in C$ existe como mucho un c' tal que $c \rightarrow c'$

# Semántica operacional (v)

- La relación de transición $\rightarrow$ se define recursivamente como la mínima relación que satisface que:

**Si** $c_1 \rightarrow c'_1, \ldots.\ c_n \rightarrow c'_n$

**entonces**

$$op(c_1,\ldots, c_n) \rightarrow op(c'_1,\ldots, c'_n)$$

# ejemplo:
# SOS de un minilenguaje imperativo

## Expresiones aritméticas, booleanas e instrucciones

- **Arit-expr:**

  $a ::= n \mid X \mid a_0 \mathbf{+} a_1 \mid a_0 \mathbf{-} a_1 \mid a_0 \mathbf{*} a_1$

- **Bool-expr:**

  $b ::= \text{true} \mid \text{false} \mid a_0 \mathbf{=} a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \vee b_1$

- **Command:**

  $i ::= \textbf{skip} \mid X \mathbf{:=} a \mid i_0 \mathbf{;} i_1 \mid \textbf{if } b \textbf{ then } i_0 \textbf{ else } i_1 \mid \textbf{while } b \textbf{ do } i$

# ejemplo

Semántica de las expresiones aritméticas:

- Evaluación de constantes $\langle n,e\rangle \rightarrow n$
- Evaluación de variables

$$\langle X,e\rangle \rightarrow e(X)$$

asumir que el estado está representado como una función $e=\{X_1 \rightarrow n_1 \ldots X_k \rightarrow n_k\}$ que asigna a cada variable $X$ su valor $n$ en dicho estado (o error si $X$ no está inicializada)

- Evaluación de sumas

$$\frac{\langle a_0,e\rangle \rightarrow n_0 \qquad \langle a_1,e\rangle \rightarrow n_1}{\langle a_0 + a_1, e\rangle \rightarrow n_0 + n_1}$$

- Evaluación de restas y productos.... similar

# ejemplo

Semántica de las expresiones booleanas:

- Evaluación de las constantes

$$< true, e > \rightarrow true$$

$$< false, e > \rightarrow false$$

- Evaluación de la igualdad

$$\frac{<a_0, e> \rightarrow n_0 \qquad <a_1, e> \rightarrow n_1}{< a_0 = a_1, e> \rightarrow true}$$

si $n_0$ y $n_1$ son iguales

$$\frac{<a_0, e> \rightarrow n_0 \qquad <a_1, e> \rightarrow n_1}{< a_0 = a_1, e> \rightarrow false}$$

si $n_0$ y $n_1$ son distintos

# ejemplo

Semántica de las expresiones booleanas:

- Evaluación de la comparación

$$\frac{\langle a_0,e\rangle \rightarrow n_0 \qquad \langle a_1,e\rangle \rightarrow n_1}{\langle a_0 \leq a_1, e\rangle \rightarrow \text{true}}$$

$n_0$ es menor o igual que $n_1$

$$\frac{\langle a_0,e\rangle \rightarrow n_0 \qquad \langle a_1,e\rangle \rightarrow n_1}{\langle a_0 \leq a_1, e\rangle \rightarrow \text{false}}$$

$n_0$ no es menor o igual que $n_1$

- Evaluación de la negación

$$\frac{\langle b,e\rangle \rightarrow \text{true}}{\langle \neg b,e\rangle \rightarrow \text{false}} \qquad \frac{\langle b,e\rangle \rightarrow \text{false}}{\langle \neg b,e\rangle \rightarrow \text{true}}$$

- *ejercicio*: disyunción

# ejemplo

Semántica de las instrucciones:

- Evaluación de las instrucciones simples

$$\langle \mathbf{skip}, e\rangle \rightarrow e$$

$$\frac{\langle a,e\rangle \rightarrow n}{\langle X{:=}a\ , e\rangle \rightarrow e\ \ \{X \rightarrow n\}}$$

- Evaluación del condicional

$$\frac{\langle b,e\rangle \rightarrow \text{true} \qquad \langle i_0,e\rangle \rightarrow e'}{\langle \mathbf{if}\ b\ \mathbf{then}\ i_0\ \mathbf{else}\ i_1\ , e\rangle \rightarrow e'}$$

$$\frac{\langle b,e\rangle \rightarrow \text{false} \qquad \langle i_1,e\rangle \rightarrow e'}{\langle \mathbf{if}\ b\ \mathbf{then}\ i_0\ \mathbf{else}\ i_1\ , e\rangle \rightarrow e'}$$

# ejemplo

Semántica de las instrucciones:

- Evaluación de la iteración

$$\frac{\langle b,e\rangle \rightarrow \text{false}}{\langle\ \mathbf{while}\ b\ \mathbf{do}\ i,\ e\rangle \rightarrow e}$$

$$\frac{\langle b,e\rangle \rightarrow \text{true} \qquad \langle\ i,e\rangle \rightarrow e'' \ \langle\ \mathbf{while}\ b\ \mathbf{do}\ i,\ e''\rangle \rightarrow e'}{\langle\ \mathbf{while}\ b\ \mathbf{do}\ i,\ e\rangle \rightarrow e'}$$

- Evaluación de la secuencia

$$\frac{\langle\ i_0,e\rangle \rightarrow e'' \ \langle\ i_1,e''\rangle \rightarrow e'}{\langle\ i_0;\ i_1,\ e\rangle \rightarrow e'}$$

# Semántica axiomática (i)

- enfoque típico de los trabajos sobre verificación fomal de programas

  (con este estilo se definió la semántica de Pascal)

- el significado de cada construcción ***i*** del lenguaje se expresa en términos de una transformación que establece qué se puede afirmar sobre el estado de la máquina tras la ejecución de ***i*** en términos de lo que era cierto antes

  o viceversa, qué debe cumplirse antes para llegar al estado que se quiere obtener tras la ejecución

# Semántica axiomática (ii)

- En general se define representando los estados mediante predicados (en vez de como funciones) y asociando a cada instrucción ***i*** del lenguaje un *transformador de predicados* (o transformador de estados) que funciona en "sentido inverso" al programa

es decir, a partir del estado "de llegada", representado por el predicado ***p***, y dada una instrucción ***i*** del lenguaje considerado, el transformador de predicados

***pmd:*** *Instrucciones x LógicaPred. -> LógicaPred*

proporciona el "predicado más debil" ***pmd(i, p)*** que expresa *lo que debe cumplirse en el estado anterior a la ejecución de **i** para que, después de dicha ejecución, se haya alcanzado el estado **p***

# Semántica axiomática (iii)

Por ejemplo, si ***i*** es una instruccion de asignación del tipo ***”X:=e“*** definimos:

$$pcm(\text{”}X{:=}e\text{“},p) = p[X \rightarrow e]$$

donde ***p[X → e]*** es el predicado que resulta de “deshacer el efecto” de haber sustituido ***X*** por ***e*** en ***p***, es decir, donde está ***e*** poner ***X*** otra vez

$$pcm(\text{“}X{:=}a\text{“},\ Y{=}[a,b,Z]) \equiv Y{=}[X,b,Z]$$

# Semántica declarativa (i)

- el significado de cada construcción se define en términos de elementos y estructuras de un dominio matemático conocido.

- este enfoque ha dado lugar a diferentes aproximaciones:

| TEORÍA MATEMÁTICA | | ESTILO SEMÁNTICO |
|---|---|---|
| 1. Tª Modelos Lógica | -> | Tª DE MODELOS |
| 2. Tª Categorías | -> | ALGEBRAICA |
| 3. Tª Funciones Recursivas | -> | PUNTO FIJO |
| 4. Tª Dominios | -> | DENOTACIONAL |

# Semántica declarativa (ii)

1. Semántica por TEORÍA DE MODELOS $S_{TM}$
   Con este estilo se ha definido la semántica de los lenguajes lógicos como Prolog

- Intuitivamente, un programa lógico P es un conjunto de *fórmulas lógicas* que definen *relaciones*

- ejemplo P= par(0).
  par(s(s(X))) ← par(X).

- el significado del programa lógico P dado por una semántica $S_{TM}$ es el conjunto de átomos (sin variables) que son consecuencia lógica de P:

$$S_{TM}(P) = \{par(0),par(s(s(0))),par(s(s(s(s(0))))),....\}$$

# Semántica declarativa (iii)

2. Semántica ALGEBRAICA $S_{ALG}$

   Con este estilo se ha definido la semántica de algunos lenguajes funcionales como OBJ

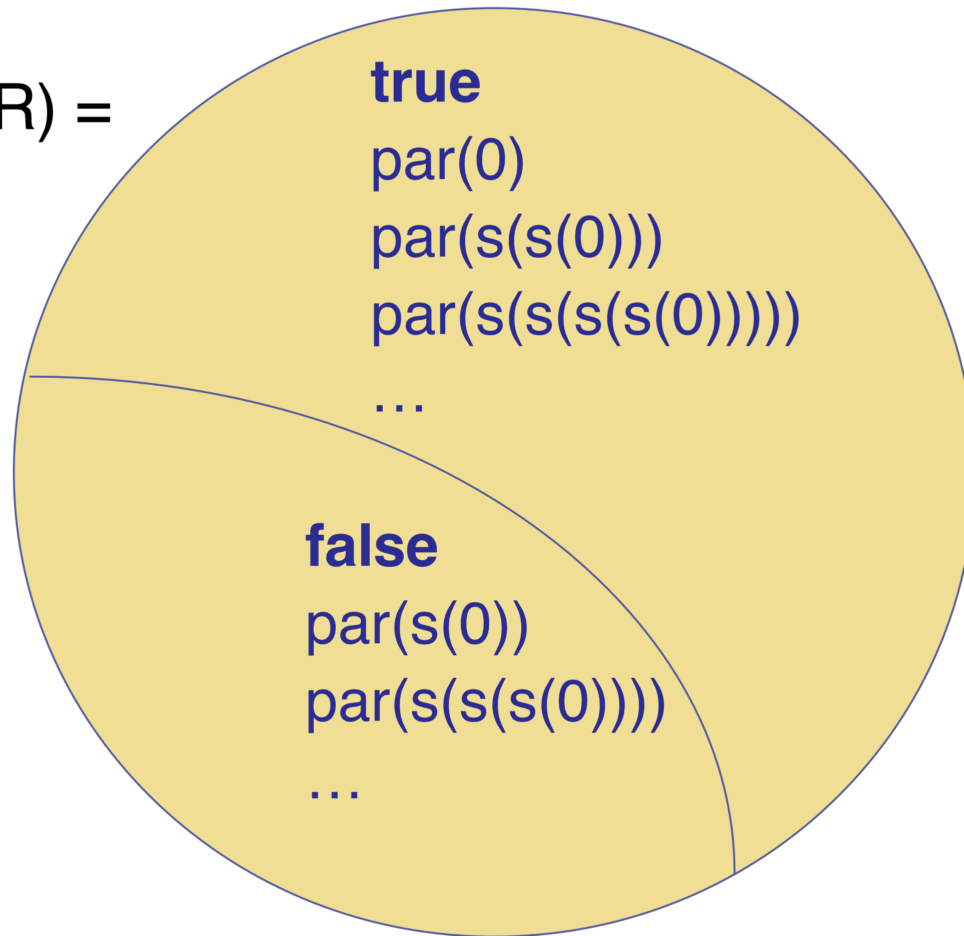
- Intuitivamente, un programa funcional R es un conjunto de ecuaciones que definen funciones

- ejemplo R = par(0)=true
  par(s(0))=false
  par(s(s(X)) = par(X)

  el significado $S_{ALG}$(R) del programa funcional R es el Tipo Abstracto de Datos asociado, definido formamente como la menor congruencia inducida en el dominio del programa (el conjunto de datos que éste manipula) por las ecuaciones del mismo *(álgebra inicial)*

# Semántica declarativa (iv)

$S_{ALG}(R)$ =



# Semántica declarativa (v)

## 3. Semántica de PUNTO FIJO $S_{PF}$

Se usa para todo tipo de lenguajes (imperativo, lógico, funcional, etc)

Se utiliza como enlace para demostrar la equivalencia entre diferentes caracterizaciones de un mismo lenguaje.

- Intuitivamente, se asocia al programa P una *transformación* (generalmente una función continua) $T_P$ definida sobre conjuntos de átomos
- el significado del programa se define como el menor punto fijo *(least fixpoint lfp)* de dicha *transformación lfp*($T_P$)

# Semántica declarativa (vi)

- dado un conjunto C de átomos, definimos la transformación $T_P$ (C) así:

  $T_P$ (C) = { A | hay una regla $H \leftarrow B \in P$ tal que en C hay una instancia del átomo B que está en la premisa de la regla y A se calcula como la correspondiente instancia de la conclusión H}
- el menor punto fijo de una función T es el menor valor C de su argumento t.q. T(C)=C
  y se obtiene aplicándolo infinitas veces empezando desde el conjunto vacío
- ejemplo P= par(0).
  par(s(s(X)) ← par(X).

  $S_{PF}$(P) = *lfp*($T_P$)= {par(0),par(s(s(0))),par(s(s(s(s(0))))),….}

# Semántica declarativa (vii)

4. Semántica DENOTACIONAL $S_{DEN}$

Con este estilo se ha la semántica de lenguajes funcionales e imperativos, como ML y ADA

- Técnicamente, es la más compleja, pero también muy rica; permite dar cuenta de computaciones que no terminan, orden superior …
- Se requiere definir:
  - los dominios sintácticos (construcciones sintác. correctas)
  - los dominios semánticos (valores asociados a cada const. sintác. correcta)
  - las funciones de evaluación semántica (de los dominios sintácticos a los semánticos)
  - las ecuaciones semánticas
- Operadores estándar sobre dominios $+$ $(\cup)$, $\times$, $\rightarrow$

# ejemplo:
# $S_{DEN}$ de un minilenguaje imperativo

```
<programa> ::= PROGRAM READ <id>;
                    BEGIN <instruccion>
                    END;
                    WRITE <expresion>
               END

<instruccion> ::=     <id> := <expresion>  |
                 <instruccion>; <instruccion> |
                 WHILE <expresion> DO <instruccion> END

<expresion> ::= <id> | <cte> |  (<expresion>) |
                <expresion> <op> <expresion>

<id>  ::= ……
<cte>  ::= ……..
<op>  ::= ……..
```

# ejemplo

- dominios sintácticos: (conjuntos)

  Id (identificadores) - *predefinido*

  - Cte (Constantes) - *idem*
  - Op (Operadores) - *idem*
  - Exp (Expresiones) - *definido como:*

    Exp= Id + Cte + (Exp x Op x Exp)
  - Inst (Instrucciones) - *definido como:*

    Inst= (Id x Exp) + (Inst x Inst) + (Exp x Inst)
  - Prog (Programas)

    Prog = Id x Inst x Exp

# ejemplo

- dominios semánticos: (funciones)
  - E (Estados)
  - V (Valores) - *predefinido*
  - Sop (Dominio semántico asociado a los operadores)
  - Sexp (Dominio semántico asociado a las expresiones)
  - Sinst (Dominio semántico asociado a las instrucciones)
  - Sprog (Dominio semántico asociado a los Programas)

con las siguientes definiciones: (como funciones)

$$E = Id \rightarrow V$$
$$Sop = V \times V \rightarrow V$$
$$Sexp = E \rightarrow V$$
$$Sinst = E \rightarrow E$$
$$Sprog = V \rightarrow V$$

# ejemplo

- valuaciones :
  - Vconst: Const $\rightarrow$ V  - *predefinida*
  - Vop: Op $\rightarrow$ Sop  - *predefinida*

  - Vexp: Exp $\rightarrow$ Sexp
  - Vins: Inst $\rightarrow$ Sinst
  - Vprog: Prog $\rightarrow$ Sprog

- ecuaciones semánticas

  ejemplo:

$$Vinst[i_1;i_2](e) = Vinst[i_2]\ (Vinst[i_1]\ (e))$$

# Semántica declarativa (viii)

- La **elección** de la semántica depende de:
  * el uso que se dará a la definición semántica
    - ayuda a la implementación del lenguaje
    - ayuda al programador
    - diseño del lenguaje
    - …
  * el tipo de lenguaje
    - LOGICO
    - FUNCIONAL
    - IMPERATIVO
    - …
  * la riqueza pretendida para las descripciones

# Equivalencia de programas. Corrección y Completitud.

- La semántica de un lenguaje nos permite razonar sobre la equivalencia de programas
  - $P \equiv_{OB} P'$ si y solo si $S_{OB}(P)=S_{OB}(P')$
  - P es completo respecto a P' si $S_{OB}(P) \supseteq S_{OB}(P')$
  - P es correcto respecto a P' si $S_{OB}(P) \subseteq S_{OB}(P')$

  donde OB= cualquiera de las semánticas que hemos visto

- EJEMPLO: $S_{OP}$(while false do Q) = $S_{OP}$(skip)

- En particular, si P' es una especificación formal (presentada también como un programa, probablemente escrito en otro lenguaje, que resuelve el mismo problema que P de manera más simple aunque menos eficiente).

  la semántica ayuda a verificar si P es una implementación correcta y completa de la especificación P': es decir, si computa todo lo que debe y sólo eso.

- EJEMPLO: P'=programa funcional "par"

  P={par(0)=true, par(s(s(X)))=true}

  $S_{ALG}(P) \neq S_{ALG}(P')$

  de hecho, P no sería una implementació correcta ni completa de P'