- El objetivo de la práctica es construir un traductor de un lenguaje imperativo a código P, así como un emulador de máquina P apto para ejecutar los programas resultantes.
- A continuación se describe la sintaxis concreta del lenguaje imperativo, junto con una descripción informal de sus construcciones (dicha descripción describe informalmente las restricciones contextuales, así como el significado operacional)

Sintaxis	**Descripción informal**
Programa → **program** SeccionTipos SeccionVariables SeccionSubprogramas IBloque SeccionTipos → **types** DecTipos | ε SeccionVariables → **variables** DecVariables | ε SeccionSubprogramas → **subprograms** DecSubprogramas | ε	Los programas comienzan por la palabra reservada **program**. Incluyen una sección de declaración de tipos (opcional), una sección de declaración de variables (opcional), una sección de declaración de subprogramas (también opcional), y una instrucción de tipo *bloque*, que constituye el cuerpo del programa. Si está presente, la sección de tipos comienza con la palabra reservada **types**. En el caso de la sección de variables, si está presente comienza con la palabra reservada **variables**. La sección de subprogramas, si está presente, comienza con la palabra reservada **subprograms**. Por regla general, el ámbito de los identificadores comienza inmediatamente a continuación de su declaración. Las excepciones son: (i) identificadores en declaraciones de tipo *puntero* (su ámbito comienza con la respectiva sección de tipos), y (ii) identificadores de subprogramas (su ámbito comienza con la ocurrencia del identificador). En todos los casos, el ámbito de un identificador termina con el bloque en el que está declarado. Los identificadores no pueden declararse más de una vez en el mismo bloque, aunque es posible re-declarar en bloques más internos identificadores declarados en bloque más externos (en este caso, la declaración más interna *oculta* a la más externa: la más interna tiene prioridad de cara a la determinación de vínculos). A efectos de estas reglas, los subprogramas se asumen declarados tanto en el bloque padre como en su propio bloque.
DecTipos → DecTipos ; DecTipo | DecTipo DecTipo → **id** : Tipo Tipo → **int** | **double** | **boolean** | **id** | **array** Tipo[**numeroNatural**] | **struct** {Campos} | **pointer** Tipo Campos → Campos ; Campo | Campo Campo → **id** : Tipo	Cada declaración de tipo es de la forma *identificador* : *Tipo*, donde *Tipo* es una definición de tipo arbitraria. Tales definiciones pueden ser: (i) tipos básicos, (ii) referencias a nombres de tipos, (iii) tipos *array*, (iv) tipos *registro* y (v) tipos *puntero*: • El lenguaje contempla tres tipos básicos: enteros – **int** –, reales con doble precisión – **double** - y booleanos - **boolean**. • El vínculo de los nombres de tipo deben ser una declaración de tipos. • Los tipos *array* con tipo base T y dimensión d se definen como **array** $T[d]$ (los correspondientes arrays comenzarán a indexarse desde 0, hasta d-1) • Los tipos *registro* se definen como **struct**$\{id_0{:}T_0; ..; id_n{:}T_n\}$, listando sus campos y respectivos tipos (no pueden aparecer campos duplicados) • Los tipos *puntero* a objetos de tipo T se definen como **pointer** T
DecVariables → DecVariables ; DecVariable | DecVariable DecVariable → **id** : Tipo	Cada variable se declara como *identificador*: *Tipo*.
DecSubprogramas → DecSubprogramas ; DecSubprograma | DecSubprograma DecSubprograma → **subprogram** **id** Parametros SeccionTipos SeccionVariables SeccionSubprogramas IBloque Parametros → (ListaParametros) | ( ) ListaParametros → ListaParametros , Parametro | Parametro Parametro → **id** : Tipo | & **id**: Tipo	Los subprogramas se declaran como **subprogram** *nombreSubprograma Parametros* *DeclaracionesLocales* *Cuerpo* Los parámetros por valor se especifican como *id* : *Tipo*, mientras que los parámetros por variable como *& id Tipo*. Las declaraciones locales pueden incluir declaraciones de tipos, de variables y de otros subprogramas (todas estas declaraciones son locales al subprograma). El cuerpo es una instrucción tipo *bloque*. La ejecución de los subprogramas es la habitual. En concreto, los subprogramas pueden invocarse recursivamente.
Instruccion → IAsig | IBloque | ICond | IBucle | ILlamada | IRead | IWrite | INew | IDelete	Hay nueve tipos de instrucciones: asignaciones, bloques, condicionales, bucles y llamadas a subprogramas
IAsig → Designador = Exp0	La instrucción de asignación consiste en asignar el resultado de evaluar una expresión al objeto designado por un designador. El tipo *Td* de la expresión debe ser *compatible* para asignación con el tipo T_i del designador. Las reglas de compatibilidad son las siguientes: • **int** es compatible con **int** y con **double** (es decir, un valor de tipo entero puede asignarse a un designador de tipo entero o de tipo real –pero un valor de tipo real no puede asignarse a un designador de tipo entero)

	• **boolean** es únicamente compatible con **boolean** (es decir, los valores de tipo booleano únicamente pueden asignarse a designadores booleanos) • Si *T* es compatible con *T*', entonces: o *T*[*n*] es compatible con *T*'[*n*] (es decir, a un designador de tipo array puede asignársele arrays de la misma dimensión y de tipo base compatible con el del designador) o **pointer** *T* es compatible con **pointer** *T*' (es decir, a un designador de tipo puntero puede asignársele punteros con un tipo base compatible al del suyo) • Un tipo *struct* es compatible únicamente con otro tipo *struct* que: (i) tenga exactamente el mismo número de campos, (ii) para cada campo *i*-esimo, los tipos de dicho campo han de ser compatibles (aunque los campos en sí no tienen porque tener el mismo nombre)
`IBloque → {Instrucciones} | {}` `Instrucciones → Instrucciones ; Instruccion | Instruccion`	Los bloques pueden ser vacíos ({}) o bien contener una o más instrucciones separadas por ;

`ICond →` **`if`** `Casos` **`endif`**
`Casos → Casos [] Caso | Caso`
`Caso → Exp0 : IBloque`
	La instrucción condicional consta de una secuencia de uno o más *casos*, delimitados entre **if** … **endif**, y separados por []. Cada caso es de la forma *Exp* : *Bloque*, donde *Exp* es una expresión cuyo tipo debe ser booleano, y *Bloque* un bloque de instrucciones.
	La ejecución de esta instrucción implica evaluar en orden las expresiones de cada uno de los casos hasta que: (i) se encuentra un caso cuya expresión evalúa a cierto, o (ii) se agotan todos los casos. En el primer caso debe ejecutarse el bloque asociado. En el segundo caso no se ejecuta ningún bloque.

`IBucle →` **`do`** `Casos` **`enddo`**	La instrucción *bucle* tiene una estructura similar a la condicional: una secuencia de casos delimitados, en esta ocasión, por **do** … **endo**. La ejecución de esta instrucción supone ir evaluando en orden los casos hasta que: (i) se encuentra uno cuya expresión evalúa a *cierto*, o (ii) se agotan todos los casos. En el primer caso debe ejecutarse el correspondiente bloque, y, a continuación ejecutar de nuevo el bucle. En el segundo caso la ejecución del bucle finaliza.

`ILlamada →` **`id`** `Argumentos`
`Argumentos → ( ListaArgumentos ) | ( )`
`ListaArgumentos → ListaArgumentos , Exp0 | Exp0`
	La llamada a un subprograma se lleva a cabo utilizando el convenio habitual: *nombreSubprograma*(E_0, …, E_k).
	El nombre del subprograma debe estar efectivamente vinculado a una declaración de subprograma. Además, el número de argumentos en la llamada debe coincidir con el número de parámetros. Así mismo, el tipo de cada argumento debe ser *compatible* para asignación con el correspondiente tipo del parámetro. Por último, los argumentos pasados por variable deben ser siempre designadores.

`IRead →` **`read`** `Designador`	Lee un valor por la entrada estándar y lo almacena en el designador (el valor debe ser un valor básico: entero, booleano, real; el tipo del valor leído debe ser compatible con el tipo del designador)
`IWrite →` **`write`** `Exp0`	Escribe el valor de la expresión por la salida estándar (el tipo de la expresión debe ser básico)
`INew →` **`new`** `Designador`	El designador ha de ser de tipo puntero. Se reserva espacio para contener un objeto del tipo apuntado, y se asigna la dirección al designador.
`IDelete →` **`delete`** `Designador`	El designador ha de ser de tipo puntero. Se libera el objeto apuntado por el mismo.
`Designador →` **`id`** `| Designador[Exp0] | Designador.`**`id`** `| Designador→`	Los designadores pueden ser: (i) identificadores (deben estar vinculados, bien con variables, bien con parámetros: el tipo será el declarado en el correspondiente vínculo, adecuadamente canonimizado). (ii) designadores de la forma *D*[*E*] (*D* debe ser de tipo *array*, y *E* de tipo **int**: el tipo resultante será el tipo base del de *D*), (iii) *D.c* (*D* debe ser de tipo *struct*, y *c* debe ser un campo de dicho tipo: el tipo resultante será el tipo de *c* en el registro), (iv) *D*→ (*D* debe ser de tipo **pointer**: el tipo resultante será el tipo base del de *D*).

`Exp0 → Exp1 OpComp Exp1 | Exp1`
`Exp1 → Exp1 OpAditivo Exp2 | Exp2`
`Exp2 → Exp2 OpMultiplicativo Exp3 | Exp3`
`Exp3 → OpUnario Exp3 | Exp4`
`Exp4 →` **`true`** `|` **`false`** `|` **`numeroNatural`** `|` **`numeroReal`** `| Designador| (Exp0)`
	Las expresiones básicas pueden ser los literales booleanos *true* y *false*, números naturales, números reales o designadores (en el caso de los literales, el tipo será el del correspondiente literal)
	Es posible formar expresiones más complejas mediante operadores de comparación (los menos prioritarios, y no asociativos), operadores aditivos (asociativos a izquierdas, siguiente nivel de prioridad), operadores multiplicativos (asociativos a izquierdas, siguiente nivel), o operadores unarios (asociativos, los más prioritarios). También es posible usar paréntesis para alterar prioridades y asociatividades.

`OpComp → == | != | < | > | <= | >=`	Los operandos de los operadores de comparación deben ser: • Ambos de tipo booleano (se asume que *false* es menor que *true*, a efectos de comparación)

	• Ambos de tipo numérico (**int** o **double**, aunque se permite comparar enteros con reales) En ambos casos, el tipo resultante es booleano.
OpAditivo → + | - | **or**	Los tipos de los operandos de + y – han de ser numéricos. Si alguno de ellos es de tipo **double**, el resultado es de tipo **double**. Si ambos son **in**t, el resultado es de tipo **int**. Los operandos de **or** han de ser de tipo booleano
OpMultiplicativo → * | / | % | **and**	Los tipos de los operandos de * y / han de ser numéricos. Si el tipo de alguno de ellos es **double**, el resultado es de tipo **double** (en este caso, la división se comporta como división real). Si ambos son de tipo **int**, el resultado será de tipo **int** (en este caso, la división se comporta como división entera). Los operandos de % deben ser ambos enteros, y el resultado es entero (la operación es la operación *módulo*). Los operandos de **and** deben ser ambos de tipo booleano, y el tipo del resultado es booleano.
OpUnario → - | **not** | **toint** | **todouble**	El tipo del operando de – ha de ser numérico (el tipo del resultado es el del operando). El tipo del operando de **not** ha de ser booleano (el tipo del resultado es booleano). El tipo del operando de **toint** ha de ser *double* (el resultado es el resultado de truncar el argumento, y el tipo es **int**). El tipo del operando de **todouble** debe ser *int* (el resultado es el del valor doublé equivalente al del operando)

- Para facilitar el desarrollo de esta práctica se proporciona una implementación JLex + CUP del analizador sintáctico para este lenguaje.
- Tareas a realizar en esta práctica:
  - Diseñar una sintaxis abstracta adecuada para dicho lenguaje.
  - Especificar, mediante una gramática de atributos, el constructor de árboles de sintaxis abstracta.
  - Ampliar la implementación JLex + CUP proporcionada para implementar el constructor de árboles.
  - Implementar los procesamientos necesarios sobre los árboles de sintaxis abstracta para garantizar el cumplimiento de las reglas de la semántica estática por parte de los programas procesados, así como para traducir dichos programas a código P.
  - Adaptar la implementación de la máquina P para permitir ejecutar el código P generado.
  - Probar las implementaciones realizadas con una batería suficientemente representativa de programas de prueba (en particular, se deberá probar sobre una versión del programa ejemplo utilizado en la práctica 3, implementado en el lenguaje propuesto en esta práctica; la batería deberá incluir tanto programas correctos como programas con errores de compilación)
  - (**Opcional**) Acondicionar la especificación del constructor de árboles para permitir su implementación descendente, e implementar la especificación acondicionada mediante JavaCC.