Especifiación final

Asignación

```
int x:= 10;
```

```
ASIGNACION ::= TIPO IDEN PCOMA
          |  TIPO IDEN OP_ASIGN EXPR PCOMA
```

Condicional con 1 y 2 ramas (if y if-else)

```
if cond then
codigo
else if cond then
codigo
end
```

```
INST_IF ::= IF EXPR THEN STMTs END
       |  IF EXPR THEN STMTs ELSE STMTs END
```

Bucles

```
while cond do
codigo
end
```

```
INST_WHILE ::= WHILE EXPR DO STMTs END
```

```
for k from 1 to N  do
for i from 1 to N do
  for j from 1 to N do
    codigo
  end
end
end
```

```
INST_FOR ::= FOR IDEN FROM EXPR TO EXPR DO STMTs END
        |  FOR IDEN FROM EXPR TO EXPR STEP EXPR DO STMTs END
```

Operadoes infijos.

Los mismos que C

Llamadas a función.

Se usan paréntesis

```
fun_function(a,b,10,1+3,4)
```

```
FUN_CALL ::= IDEN PAREN_AP EXPRs PAREN_CIER PCOMA;
```

Instruciones de entrada y salida.

Usamos las palabras clave `ouput` y `input`

> TODO: Preguntar sobre como implementamos esto en WASM

```
output
input
```

Opción 1: Lo implementamos como sus propias instrucciones

```
output 10;
input x;
```

```
INST_OUT ::= OUTPUT EXPR PCOMA
INST_IN  ::= INPUT IDEN PCOMA;
```

Para simplificar la declaración de variables que luego reciben valores de entrada, planteamos los siguientes azúcares sintácticos sobre

```
int x;
input x;
```

Estos son:

```
int x := input;
input int x;
```

Opción 2: `input` y `output` son funciones estándar, que no necesitan de una detección especial.

```
output(10);
x := input();
```

Expresiones con punteros y nombres cualificados (notacion `.`)

El typo de un puntero tiene el formato `ptr int`.

```
PTR_TIPO ::= PTR TIPO_PRIMITIVO;
```

Para obtener un puntero de una variable, usamos la palabra clave `ptr`

```
ptr int a := ptr x;
```

Para dereferencia un puntero, usamos el operador `@` (Pronunciado "at" en inglés)

```
@a;
```

En el caso que el puntero sea puntero a una estructura con campos, usamos el operador `->` para dereferenciar el puntero y acceder al campo de la derecha (Como C++).

```
a->next;
@a.next;
```

El operador `->` tiene mayor preferencia que `@` para permitir expresiones del tipo `@(curr->next->next)` sin usar paréntesis

> TODO: Seguro que esto es buena idea? Es más legible @curr->next->next que @(curr->next->next)? Si la dereferenciación la haremos al final, solo una no hace mucho daño. Hay que pensarlo

Instrucción case (Salto a cada rama en tiempo constante)

> TODO: Ver si hacemos esto, y como (Creo que no es opcional)

```
match a is
  case 1 do
    codigo
  case 2 do
    codigo
  else
    codigo
end
```

Instrucciones de reserva de memoria dinámica

> TODO: Ver si hacemos esto, y como (Fer vota por instrucciones especiales)

```
ptr int a = alloc 1 int;
free a;
```

Arrays

> TODO: Preguntar sobre si arrays deben ser su propio tipo, o es mejor tratarlos como punteros Si los tratamos como punteros, como reservamos memoria en la pila?

```
int a;
ptr int a = array 10; ¿?
func sort(int[] a) ->
```

Bloques anidados

> TODO: Preguntar si podemos no hacerlo En C puedes hacer cosas como
>
> ```c
> int x= 10, y = 10;
> {
>     int tmp = x;
> x = y;
>     y = tmp;
> }
> printf("La variable tmp = %i ya no existe", tmp);
> ```
>
> Queremos permitir hacer cosas de este estilo en nuestro lenguaje?

Funciones (paso parámetros por valor o referencia)

Hemos decidido pasar parámetros por valor, puesto tenemos punteros. Queda más claro.

La sintaxis es la siguiente

```
func nombre(int a, int a) -> int
 a := 10;
 halt(adsf);
 asdfa(f);
end
```

```
FUNC_DECL ::= FUNC IDEN ARGs STMTs END
          |  FUNC IDEN ARGs FLECHA TIPO STMTs END;
ARGs ::= PAREN_AP PAREN_CIERRE | PAREN_AP ARG_DECLs PAREN_CIERRE;
ARG_DECLs ::= TIPO IDEN | TIPO IDEN COMA ARG_DECLs;
```

Registros (?)

> TODO: Que son los registros específicamente? Algo de WASM que no conocemos aún?

Clases (sin herencia, más como structs)

Nuestras clases son más bien `structs`, tipos que tienen campos en sus declaraciones, que pueden ser funciones en sí mismas

> TODO: Queremos que cuando llamamos `valor.fun(...)`, se pase `valor` como primer parámetro a `fun`? Lua tiene su propio operador para esto, `:` (Así, pones `valor:fun(...)` si quieres hacer `valor.fun(valor)`)

```
struct Nombre is
int a;
int b;
func new() -> Nombre
  return Nombre{a = 1, b = 2};
end
end

Nombre a = Nombre.new();
```

```
STRUCT_INST ::= STRUCT IDEN IS DECLs END;
DECLs ::= ASIGNACION | FUNC_DECL | DECLs | ;
```

> TODO: Tenemos que ver como construir estructuras desde cero. Aquí, uso la sintaxis Nombre{} un poco arbitrariamente. Podemos asignar valores a los campos por orden en vez de por nombre.

Módulos

Los módulos sirven para declarar código en distintos espacios de nombres

```
module Std is
public int a;
int secret;
public func sort(ptr int array) -> ptr int
end
end
```

```
MODULE_INSTR := MODULE IDEN IS DECLs END
```

> TODO: Admitimos que se ejecute el código que esté en un nivel superior? O todo el código a ejecutar se debe ejecutar desde una función? Creo que mejor que todo código ejecutable esté en funciones Es similar un módulo a una clase. Podemos a lo mejor intentar desacoplar ambos conceptos

Cláusulas de importanción

> TODO: Permitimos solo que estén en el inicio de un archivo, o en cualquier parte del código? Como se ve esto desde la perspectiva del programa (Es decir, que pongamos un `import` en alguna parte hace alguna operación real, o simplemente hace que el compilador no se queje de que no conoce el símbolo?) Que funcionalidad en WASM podemos usar para implementar la funcionalidad sino? El acceso a los campos de un módulo se hace con `::`.

```
import Std;
Std::sort(a);
```

```
IMPORT_STMT ::= IMPORT IDEN;
           |  IMPORT IDEN FROM IDEN;
```

Tipos básicos predefinidos (enteros, booleanos)

> TODO: Ver como gestionamos errores

```
int
bool
```

Como implementamos los pares?

```
pair int int p = (10,1);
pair int p = (10,1);
(int, int) p = (10,1);
answer p = (10,1);
```

Definición de tipos de usuario

> TODO: A que se refiere exactamente? Es un simple alias de tipos? O con `struct` nos vale? Podemos implementar `enum`?

```
type error = int;
```