Lenguaje JDC++

Procesadores de Lenguajes

Curso 2024-2025

UNIVERSIDAD
COMPLUTENSE
MADRID

Autores:

Jorge Arboleya Carrio
David Esparza Sainz
Cristina Gómez Calvo

Índice

1. Introducción **2**

2. Especificación de la sintaxis del lenguaje **2**
2.1. Identificadores y ámbitos de definición . . . 2
2.1.1. Declaración de variables simples . . . 2
2.1.2. Declaración de arrays . . . 2
2.1.3. Bloques anidados . . . 2
2.1.4. Funciones . . . 2
2.1.5. Punteros . . . 3
2.1.6. Registros . . . 3
2.1.7. Clases . . . 3
2.2. Tipos . . . 4
2.2.1. Tipos básicos predefinidos . . . 4
2.2.2. Operadores infijos . . . 4
2.2.3. Tipo array . . . 4
2.2.4. Tipos personalizados . . . 4
2.2.5. Comprobación de tipos . . . 5
2.3. Conjunto de instrucciones del lenguaje . . . 5
2.3.1. Condicionales . . . 5
2.3.2. Bucles . . . 5
2.3.3. Constantes . . . 5
2.3.4. Acceso a arrays . . . 5
2.3.5. Llamadas a función . . . 6
2.3.6. Registros . . . 6
2.3.7. Reserva de memoria dinámica . . . 6
2.3.8. Comentarios . . . 6
2.3.9. Funciones de lectura y escritura . . . 6
2.4. Gestión de errores . . . 7

3. Ejemplos de utilización del lenguaje **7**
3.1. Operaciones matemáticas elementales . . . 7
3.2. Fibonacci . . . 7
3.3. Verificar si un número es primo . . . 8
3.4. Algoritmos de ordenación . . . 8
3.5. Actualizar un valor por referencia . . . 10
3.6. Clases . . . 10

1. Introducción

En este trabajo se presenta JDC++, un lenguaje de programación desarrollado con el objetivo de ofrecer una sintaxis simplificada, pero potente, para la resolución de problemas. Este lenguaje está basado principalmente en C++ y Java, y a diferencia de otros lenguajes, en JDC++ las clases no tienen herencia, lo que facilita la estructura y comprensión del código.
Uno de los aspectos clave de JDC++ es la inclusión de mejoras en las estructuras de control. En particular, hemos realizado modificaciones en los bucles for y while, a los que denominamos cycle y during, respectivamente. Estas modificaciones amplían sus funcionalidades y permiten un control más eficiente del flujo del programa.
A lo largo del trabajo, se profundiza en las principales características de JDC++, como la declaración de variables, el uso de arrays, y la creación de funciones y estructuras de control, ilustrando su funcionamiento mediante ejemplos prácticos.

2. Especificación de la sintaxis del lenguaje

2.1. Identificadores y ámbitos de definición

2.1.1. Declaración de variables simples

Para declarar variables simples, escribiremos el nombre del tipo correspondiente seguido del nombre que le demos a la variable y, como equivalente al `;` de C++, utilizaremos el caracter punto (`.`). En caso de querer inicializar la variable en el momento de declararla, escribiremos después de su nombre una flecha hacia la izquierda (`<-`) seguida del valor que le queramos dar, teminando igualmente con punto (`.`).
Ejemplo de declaración de dos variables simples, la primera sin inicializar y la segunda inicializada:

```
t var1.
t var2 <- valor.
```

2.1.2. Declaración de arrays

Permitiremos declarar tanto arrays unidimensionales como multidimensionales. Tendrán la siguiente estructura:

```
array(t,n) a1.
array (t,n1,...,nm) a2.
```

`a1` es un array unidimensional de `n` elementos de tipo `t`.
`a2` es un array multidimensional de `m` dimensiones, cada una con `n_i` elementos, todos ellos de tipo `t`.
Veamos ahora cómo inicializar arrays en el momento de su declaración:

```
array(t,n) a1 <- [elem_1,...,elem_n].
array (t,n_1,...,n_m) a2 <- [[elem_1_1,...,elem_1_n1],...,[elem_m_1,...,elem_m_nm]].
```

2.1.3. Bloques anidados

En nuestro lenguaje, podemos anidar bloques de la misma forma que en $C++$. Más adelante, al definir la semántica de los condicionales y de los bucles, se verá algún ejemplo.

2.1.4. Funciones

Para declarar una función, primero utilizaremos la palabra reservada de nuestro lenguaje `fun` y después escribiremos el nombre que le queremos dar a nuestra función. Tras ello, escribiremos los parámetros que recibe entre paréntesis y separados por `;`, indicando para cada uno su tipo y su nombre. Por último, indicaremos el tipo de lo que devuelve mediante `dev t`. El cuerpo de la función está delimitado por {}. La última línea del mismo será de la forma `ret a.`, donde `ret` es una palabra reservada para indicar lo que devuelve la función y `a` es una del tipo indicado anteriormente en `dev`. Esto último se aplica en todos los casos salvo si tras la palabra reservada `dev` aparece la palabra `nothing`, en cuyo caso la función no devuelve nada.

```
fun nombreFuncion(t_1 arg_1;...;t_n arg_n) dev t {
    ...
    ret a.
}
```

Por defecto, supondremos que los parámetros se pasan por valor, a no ser que indiquemos explícitamente lo contrario. Para indicar el paso por referencia, escribiremos `ref` entre el tipo y el nombre del parámetro en cuestión. En el siguiente ejemplo, la función recibe un parámetro por referencia:

```
fun nombreFuncion1(t_1 ref arg1) dev t {
    ...
    ret a.
}
```

```
fun nombreFuncion2(t_1 ref arg1, t_2 arg2) dev nothing {
    ...
}
```

2.1.5. Punteros

Para declarar un puntero, escribiremos `->` antes del nombre del tipo. Con ello, indicamos que esta variable apunta a un objeto de ese tipo. Para acceder a lo apuntado, escribimos `->` antes del identificador. Si lo que queremos es acceder a la dirección de memoria a la que apunta, escribiremos `?` antes del nombre de nuestro puntero, simbolizando que le estamos preguntando a dónde apunta.

```
->t nombrePuntero.
t apuntado <- ->nombrePuntero.
?nombrePuntero
```

En la primera línea mostramos la declaración del puntero, en la segunda línea asignamos lo apuntado por el puntero a una variable de tipo `t` (tipo del puntero) y en la tercera línea obtenemos la dirección de memoria apuntada.

2.1.6. Registros

Denominaremos `reg` a los registros. Para su declaración, escribiremos `reg` seguido del nombre que le queramos dar. Luego, entre llaves, escribiremos todos los campos de nuestro registro, separados por un punto (`.`). Para cada uno, escribiremos el tipo del campo y su nombre.

```
reg nombreRegistro {
    t_1 campo_1.
    ...
    t_n campo_n.
}
```

2.1.7. Clases

En el lenguaje $JDC++$ podemos definir clases, pero no admiten herencia. La palabra reservada para denominarlas será `entity`. Todo lo relativo a la clase viene delimitado por llaves (no se permite declarar funciones de la clase fuera de la misma). Para acceder a una función de la clase utilizaremos una flecha (`-->`), a diferencia del punto utilizado en $C++$, de la siguiente forma: `nombreClase-->nombreFuncion(...)`,donde `nombreFuncion` es una función de la clase (que recibe los parámetros necesarios entre paréntesis) de la clase `NombreClase`. Para cada uno de los atributos de la clase es análogo, omitiendo los paréntesis debido a que los atributos no reciben parámetros (`NombreClase->nombreAtributo`). Dentro de la propia clase, nos referiremos a la misma mediante la palabra reservada `this`, por lo que el acceso a sus propios atributos y funciones será así: `this-->nombreFuncion(...)` y `this-->nombreAtributo`. Los constructores a grandes rasgos funcionan igual que en $C++$.

```
entity NombreClase{
    t_1 atributo_1;
    ...
    t_n atributo_n;
    NombreClase(t_1 a_1;...;t_n a_n){
        this-->atributo_1 <- a_1.
        ...
```

```
        this-->atributo <- a_n.
    }
    fun funcion1(t_1 parametro) dev t {
        t a.
        ...
        ret a.
    }
}
```

2.2. Tipos

2.2.1. Tipos básicos predefinidos

El lenguaje $JDC++$ tiene varios tipos básicos:

- `num`: números enteros
- `dec`: números que tengan parte decimal. La parte decimal se separa de la entera por el apóstrofe (').
- `tof`: booleano, puede tomar los valores `tru` o `fols`
- `array`

A continuación, mostramos un ejemplo de inicialización de estos tipos:

```
num n <- 3.
dec pi <- 3'14.
tof falacia <- fols.
array(num, 3) arrayNumerico <- [3,2,4].
```

En caso de no querer inicializarlos al momento de su declaración, simplemente omitiríamos la asignación.

2.2.2. Operadores infijos

Prioridad	Operador	Aridad	Asociatividad
0	-> (campos de registros y clases o funciones de clases)	2	Derecha
1	-> (puntero)	1	No asociativo
1	? (acceso a dirección de memoria)	1	No asociativo
2	$*, /, \%$	2	Izquierda
3	+, -	2	Izquierda
4	<=, >=, <, >	2	Izquierda
5	eq, neq	2	Izquierda
6	not	1	No asociativo
7	and, or	2	Izquierda

2.2.3. Tipo array

Anteriormente hemos visto como declarar e inicializar arrays, tanto unidimensionales como multidimensionales. Por tanto, aprovechando que acabamos de definir los tipos básicos, vamos a ver un ejemplo con tipos concretos.

```
array(num,3) a1 <- [12,2,3].
array (num,2,3) a2 <- [[1,4,14],[0, 7, 23]].
```

`a1` es un array de enteros de 3 elementos y `a2` es un array de caracteres bidimensional, cuyas dimensiones son 3 y 2. Ambos son arrays inicializados en el momento de su declaración. Por tanto, si no quisiésemos inicializarlos, simplemente deberíamos omitir la parte de la asignación.

2.2.4. Tipos personalizados

Podemos definir tipos personalizados usando `typedef`.

```
typedef matriz2por2 <- array(num,2,2).
```

2.2.5. Comprobación de tipos

Nuestro lenguaje comprobará en tiempo de ejecución los tipos. El tratamiento de errores se detalla en la sección 2.4.

2.3. Conjunto de instrucciones del lenguaje

2.3.1. Condicionales

En nuestro lenguaje, vamos a sustituir el if por la palabra reservada `check`, y el else if y else se sustituyen respectivamente por `othercheck` y `other`. Para poder utilizar estas dos últimas, al igual que en C++, es estrictamente necesario utilizar un `check` previamente. Además, podemos anidar unos dentro de otros.
El caso base es en el que tendremos un `check` de una condición:

```
check(cond){...}
```

Cuando tenemos que comprobar distintas condiciones, las comprobamos utilizando `othercheck`:

```
check(cond1){...}
othercheck(cond2){...}
...
othercheck(cond_n-1){...}
other{...}
```

2.3.2. Bucles

El bucle for en JDC++ se nombra con la palabra `cycle`, pero en este caso, tenemos dos opciones: Primero, lo podemos utilizar de la misma manera que en C++:

```
cycle(num i<-0,i<5,i <- i+1){...}
```

Sin embargo, nosotros le vamos a añadir la opción de una cuarta cláusula, donde se nos permitirá "saltarnos" algunos pasos del `cycle` sin tener que usar una cláusula `check` dentro de él. Es decir, haremos sólo las iteraciones del bucle que cumplan las condiciones especificadas en esta cuarta opción. Por ejemplo, si queremos hacer un bucle del 0 al 4 pero que se salte la iteración i=3, lo haríamos de la siguiente forma:

```
cycle(num i<-0,i<5,i <- i+1,i neq 3){...}
```

En cuanto al while, lo renombramos como `during`, y le añadimos la misma funcionalidad (opcional) que al `cycle`.

```
num i<-0.
during(i<5,i neq 3){...}
```

2.3.3. Constantes

Las constantes se crean fuera de todas las funciones como variables globales y las creamos con la palabra reservada `FINAL`, al igual que en Java. Esta palabra la tendremos que colocar antes del tipo en la declaración de la constante.

```
FINAL num cte1<-10000.
```

2.3.4. Acceso a arrays

El acceso a arrays funciona igual que en C++, es decir, si pones únicamente el nombre del array se refiere "al array completo". Sin embargo, si queremos acceder a una posición del array, digamos la i, utilizamos corchetes de la forma `nomArray[i]`.

2.3.5. Llamadas a función

Las llamadas a función funcionan al igual que en C++. Si tenemos una función de tipo `dev nothing`, no tenemos que asignarla a ninguna variable. Sin embargo, si la función devuelve algo de un tipo, se puede decidir asignarla a una variable para guardarlo o no asignarlo y "deshacernos"del valor devuelto. En el apartado 2.1.4, hemos creado una función `nombreFuncion1` que devuelve una variable de tipo `t`, y otra `nombreFuncion2`, que no devuelve nada. Estas se pueden usar de la siguiente manera:

```
t var1<-nombreFuncion1(arg1)
nombreFuncion2(arg1;arg2)
```

2.3.6. Registros

Hemos visto en la sección 2.1.6 cómo se declaran los registros y declaramos uno llamado `nombreRegistro`. Para acceder a uno de los campos del registro utilizaremos el operador $--> $. Para verlo en un ejemplo, supongamos que queremos guardar en el campo 1 del registro la suma del campo 2 y el campo 3. Se haría de la siguiente forma:

```
nombreRegistro-->campo_1 <- nombreRegistro-->campo_2 + nombreRegistro-->campo_3.
```

2.3.7. Reserva de memoria dinámica

Debido a que nuestro lenguaje admite la declaración de punteros, nos será útil tener ciertas instrucciones para reservar y liberar memoria dinámica. Estas instrucciones serán, respectivamente, `book` y `unbook`.

```
book ->num ptr.
unbook(ptr).
```

2.3.8. Comentarios

Al igual que en la mayoría de los lenguajes, vamos a tener dos tipos de comentario. Los principales serán los comentarios de una línea, que comenzarán con un #.

```
#Esto es un comentario de una linea
```

También podemos hacer comentarios multilínea utilizando $/*$ al principio del comentario. Cuando queramos cerrar el comentario, lo utilizaremos con el paréntesis al otro lado: $*/$.

```
/*Esto
es
un
comentario
multilinea */
```

2.3.9. Funciones de lectura y escritura

Para leer y escribir, utilizaremos las siguientes funciones:

- `printdec`, `printnum` y `printtof` son las funciones que sirven para mostrar por pantalla números decimales, números enteros y booleanos, respectivamente.
- `readdec`, `readnum` y `readtof` son las funciones que sirven para leer números decimales, números enteros y booleanos. Devuelven el mismo tipo que leen.

```
d <- readdec().
printdec(d).
n <- readnum().
printnum(n).
b <- readtof().
printtof(b).
```

2.4. Gestión de errores

Tanto los errores léxicos como los sintácticos serán comprobados en tiempo de compilación, mostrando un mensaje de error indicando el tipo de error y la fila y columna donde se ha producido. En cuanto se detecte un error de estos tipos, la compilación del programa será fallida pero seguiremos analizándolo en busca de más errores. De esta forma, el usuario podrá corregir una mayor cantidad de errores sin tener que compilar varias veces.
Por otro lado, los errores de tipos serán tratados en tiempo de ejecución y se mostrará un mensaje con la misma estructura que la de los otros errores. En este caso se detendrá la ejecución.
A continuación mostramos ejemplos de mensajes de los tres tipos:

```
Error lexico: fila 4, columna 20
Error sintactico: fia 10, columna 2
Error de tipos: fila 1, columna 6
```

3. Ejemplos de utilización del lenguaje

3.1. Operaciones matemáticas elementales

Ahora vamos a ver un ejemplo en el que incorporamos el main, que en nuestro lenguaje se llama `startPls`. En este caso, se encargará de calcular las operaciones matemáticas elementales (suma, resta, multiplicación y división entera).

```
fun suma ( num a ; num b ) dev num {
    num resultado.
    resultado <- a + b.
    ret resultado.
}
fun resta ( num a ; num b ) dev num {
    num resultado.
    resultado <- a - b.
    ret resultado.
}
fun multiplicacion ( num a ; num b ) dev num {
    num resultado.
    resultado <- a * b.
    ret resultado.
}
fun divEntera ( num a ; num b ) dev num {
    num resultado.
    resultado <- a / b.
    ret resultado.
}
fun startPls() dev num{
    num a <- 2.
    num b <- 3.
    num n.
    n <- suma(a,b).
    printnum(n).
    n <- resta(a,b).
    printnum(n).
    n <- multiplicacion(a,b).
    printnum(n).
    n <- divEntera(a,b).
    printnum(n).
    ret 0.
}
```

3.2. Fibonacci

Un buen ejemplo para ver como es la estructura del check y que también nos sirve para ver la recursión es Fibonacci, donde usamos los dos términos anteriores teniendo dos casos base.

```
fun fibonacci(num n) dev num {
    check(n eq 0) {
        ret 0.
    }
    othercheck(n eq 1) {
        ret 1.
    }
    other {
        ret fibonacci(n - 1) + fibonacci(n - 2).
    }
}
```

3.3. Verificar si un número es primo

Ahora veamos una función booleana que se encarga de verificar si un número pasado por parámetro es primo o no.

```
fun esPrimocv( num n ) dev tof {
    check ( n <= 1 ){
        ret fols.
    }
    cycle ( num i < -2 , i < n , i ++ ) {
        check ( n % i eq 0 ){
            ret fols.
        }
    }
    ret tru.
}
```

3.4. Algoritmos de ordenación

Veremos como se implementan los algoritmos principales de ordenación de un array: bubblesort, mergesort y quicksort. Empezamos con bubblesort, que es el más sencillo de implementar ya que no utiliza funciones auxiliares.

```
fun bubbleSort(array(num, n) arr) dev nothing {
    num i <- 0.
    num j <- 0.
    num temp.

    cycle(i <- 0, i < n - 1, i <- i+1) {
        cycle(j <- 0, j < n - i - 1, j<- j+1) {
            check(arr[j] > arr[j + 1]) {
                temp <- arr[j].
                arr[j] <- arr[j + 1].
                arr[j + 1] <- temp.
            }
        }
    }
}
```

Ahora veamos como se implementan el mergesort y quicksort, ambos con funciones auxiliares merge y partition respectivamente. Son funciones que se llaman recursivamente hasta que está finalizado el algoritmo:

```
fun merge(array(num, n) arr, num l, num m, num r) dev nothing {
    num i <- 0.
    num j <- 0.
    num k <- l.

    num n1 <- m - l + 1.
    num n2 <- r - m.
```

```
    array(num, n1) left.
    array(num, n2) right.

    cycle(i <- 0, i < n1, i <- i+1) {
        left[i] <- arr[l + i].
    }
    cycle(j <- 0, j < n2, j<- j+1) {
        right[j] <- arr[m + 1 + j].
    }

    i <- 0.
    j <- 0.

    cycle(k <- l, k <= r, k++) {
        check(i < n1 and (j >= n2 or left[i] <= right[j])) {
            arr[k] <- left[i].
            i <- i+1.
        }
        other {
            arr[k] <- right[j].
            j<- j+1.
        }
    }
}

fun mergeSort(array(num, n) arr, num l, num r) dev nothing {
    check(l < r) {
        num m <- (l + r) / 2.
        mergeSort(arr, l, m).
        mergeSort(arr, m + 1, r).
        merge(arr, l, m, r).
    }
}
```

```
fun partition(array(num, n) arr, num low, num high) dev num {
    num pivot <- arr[high].
    num i <- low - 1.
    num temp.

    cycle(num j <- low, j < high, j<- j+1) {
        check(arr[j] <= pivot) {
            i <- i+1.
            temp <- arr[i].
            arr[i] <- arr[j].
            arr[j] <- temp.
        }
    }

    temp <- arr[i + 1].
    arr[i + 1] <- arr[high].
    arr[high] <- temp.

    ret i + 1.
}

fun quickSort(array(num, n) arr, num low, num high) dev nothing {
    check(low < high) {
        num pi <- partition(arr, low, high).
        quickSort(arr, low, pi - 1).
        quickSort(arr, pi + 1, high).
    }
}
```

3.5. Actualizar un valor por referencia

Para el siguiente ejemplo vamos a crear un registro Persona que tenga como campos la identificación y la edad. Además, creamos una función que actualiza la edad de la persona por referencia.

```
reg Persona {
    num id.
    num edad.
}

fun actualizarEdad ( Persona ref p ; num nuevaEdad ) dev num {
    p->edad <- nuevaEdad.
    ret p->edad.
}
```

3.6. Clases

En este lenguaje vamos a poder utilizar clases sin herencia, y un ejemplo de ello es el siguiente, donde creamos una clase llamada CuentaBancaria, que tendrá un constructor y 3 funciones dentro de la clase.

```
entity CuentaBancaria {
    num idCuenta.
    num saldo.

    CuentaBancaria(num id ; num saldoInicial) {
        this->idCuenta <- id.
        this->saldo <- saldoInicial.
    }

    fun depositar(num cantidad) dev nothing {
        check(cantidad > 0) {
            this->saldo <- this->saldo + cantidad.
            printnum(this-->saldo).
        }
    }

    fun retirar(num cantidad) dev nothing {
        check(cantidad > 0 and cantidad <= this->saldo) {
            this->saldo <- this->saldo - cantidad.
            printnum(this-->saldo).
        }
    }

    fun consultarSaldo() dev nothing {
        printnum(this-->saldo).
    }
}
```