Definición de la sintaxis del lenguaje C−−

Procesadores del Lenguaje Definitivo

Óscar Calvet Sisó, Noelia Barranco Godoy

1. Introducción

El objetivo de este documento es el de especificar la sintaxis de nuestro lenguaje, C−−. Este va a ser un lenguaje de programación imperativo y fuertemente tipado. Nos vamos a basar en el padre de todos los lenguajes imperativos modernos: C. Nuestro objetico con C−− es el de crear una versión simplificada de C y añadirle algunas utilidades como la definición de clases (sin herencia y declaradas en el ámbito global de nuestro programa). Contaremos únicamente con un archivo de código.
El punto de entrada de todo programa válido de nuestro lenguaje será una función 'main', que no devolverá nada y recibirá un array de ints (potencialmente vacío).
Nuestro lenguaje implementará comentarios de una sola linea al estilo de C. Cabe destacar que la cadena // estará prohibida dentro de los comentarios. Por último, al final expondremos una lista de palabras reservadas. Estas son palabras que por su construcción bien podrían ser identificadores, pero nosotros las prohibiremos porque se usarán en otros contextos.

2. Identificadores y ámbitos de definición

El lenguaje permite declarar variables (o constantes*) simples y arrays de cualquiera de los tipos descritos más adelante (incluidos arrays de varias dimensiones). Estas variables podrán estar alojadas en memoria estática o dinámica*. Como es habitual, habrá anidamiento de bloques con sus respectivos y funciones, que podrán tener argumentos de cualquier tipo*.
Las funciones solo se podrán definir en el ámbito más general del proyecto (fuera de cualquier función), así como, los tipos de usuario, tipos estructurados y clases. Las funciones podrán recibir como argumentos parámentros de cualquier tipo, incluido punteros o referencias.
En el ámbito global, también se podrán declarar variables*y definir tipos nuevos .

3. Tipos

3.1. Tipos básicos y de usuario

En nuestro lenguaje existirán tres tipos básicos, el tipo entero (con identificador int), el tipo booleano (con identificador bool) y el tipo caracter* (con identificador char). Además, el usuario podrá declarar pseudónimos de cualquier tipo del lenguaje así como, clases y estructurados.

3.2. Arrays y punteros

Con todos estos tipos, podremos crear arrays y punteros*. La notación que identifica a los arrays será la habitual, es decir, el identificador seguido de tantos corchetes de apertura y cierre como dimensiones tenga el array. Los arrays no serán polimórficos.

*Ver seccion 8

La notación pertinente a los punteros serán el identificador precedido de la palabra reservada 'pointer' seguida de otra identificador de tipo. Añadiremos pues un operador de referencia que, dada una variable cualquiera devuelva un puntero que apunte a esa variable, que se expresará con el "&".

3.3. Funciones

A nivel sintáctico, todas las funciones estarán tipadas. Para esto exisistirá la palabra reservada "void", que indicará que una función no devuelve nada. No obstante, void no será un tipo realmente (por lo que no podrá usarse para declarar variables), void será una manera de indicar al compilador que esa función no va a devoler nada (realmente será un procedimiento).

3.4. Declaraciones

Todo tipo que no sea una función se declará de la siguiente manera: identificador de tipo seguido de un identificador (no seguido de un paréntesis para diferenciarlo de una función). Adicionalmente, podrá ir seguido del signo igual y de una expresión del tipo a declarar.
Los tipos de array se declararán de la forma una expresión de tipo existente seguida del símbolo [, una expresión* y]. Los tipos punteros se definirán con la palabra reservada 'pointer' seguida de un tipo cualquiera y terminada en un identificador seguido de un número arbitrario de corchetes cada uno conteniendo una expresión.
Las funciones se declararán de la siguiente forma: un identificador de tipo (o identificador void), seguido de un identificador y de un parentesis. Una sucesión de declaración de variables separadas por comas, terminado por un paréntesis de cierre y seguido de un bloque de código de nuestro lenguaje.
Los tipos estructurados se declarán en el ámbito general, serán inicializados por la palabra reservada "struct"seguidos de llaves que contendran declariones de variables y a continuación un identificador terminado en ';'.
Las clases se declarán en el ámbito general, serán inicializadas por la palabra reservada class seguida de llaves que contendrán instrucciones, declaraciones y declaraciones de funciones, después de las llaves aparecerá un identificador terminado en ';'.

4. Conjunto de instrucciones del lenguaje

4.1. Designadores de variable

Se considerará un designador de variable cualquier cadena alfanumérica o con el simbolo "_"que no empiece por un número y también se considerará designador de variable otro designador de variable, seguido de un corchete de apertura, un número y un corchete de cierre. Esto es para considerar los accesos a arrays como designadores de variables.

4.2. Asignaciones

Toda asignación será una construcción de la forma o bien un identificador de variable, o bien una declaración de variable, seguida de un simbolo = y por último una expresión.

4.3. Llamada a función

Una instrucción de la forma identificador, paréntesis de apertura, una serie de identificadores de variable o literales separados por comas, y terminado por un cierre de paréntesis. En cualquier lugar donde pudiera ir un literal de cualquier tipo, también podrá aparecer una llamada de función en su lugar. En lugar de un identificador como nombre de función, podrá usarse la palabra reservada build, esta palabra reservada solo se puede utilizar para nombrar funciones, pero no tiene ningún significado especial (es legacy code, se iba a utilizar para un tipo de funciones especiales, pero al final no).

4.4. Expresiones aritméticas

Siguiendo sintaxis habitual para estos, son un conjunto formado por $+, -, *, /, \%$ como operadores binarios aplicados sobre otras expresiones aritméticas. Unos identificadores constantes (números) e identificadores de variables.

También el operador prefijo " $-$ " seguido de otra expresión aritmética, una operación aritmética rodeada de paréntesis.

4.5. Expresiones booleanas

Otra vez siguiendo el estándar, un operador booleano se definirá con los casos base de palabras reservadas "truez "false", y la aplicación de los operadores binarios "$\&\&, \|\|$" de manera infija sobre otras expresiones booleanas o el operador unario prefijo ¡". También se considerarán expresiones booleanas las que contienen dos identificadores de variable o expresiones de cualquier tipo y los siguientes símbolos " $==, != $ " separándolas como operador infijo, representando la igualdad y la desigualdad respectivamente.

Finalmente también existirán expresiones booleanas de la forma siguiente: dos expresiones aritméticas separadas por los operadores "$<=, >=, >, <$"de manera infija, representado las comparaciones aritméticas.

4.6. Expresión read

Una expresión particular es read, que tendrá como sintaxis la palabra reservada read, seguida de un paréntesis de apertura y otro de cierre. Esta devolverá siempre un int, que será leido del buffer de entrada del programa.

4.7. Expresiones de caracteres*

Estas solamente serán de la forma comillas simples, seguida de un caracter ASCII y cerrado por otra comilla.

4.8. Expresiones de arrays*

Al definir un array, siempre se hará de manera constante, osea la apertura de un corchete seguida de uan secuencia de literales separados por coma y terminado en corchete de cierre

4.9. Expresiones con punteros

Tenemos el operadores unarios "$*, \&$"precede a un identificador, para acceder al valor de un puntero o a su dirección respectivamente. También tenemos el operador binario infijo "."que se encontrará entre dos identificadores donde el de más a la izquierda hará referencia a un struct o una clase y el de la derecha a un identificador de variable o de función. La prioridad de estos operadores será la habitual de C.

4.10. Instrucción

Toda instrucción será una asignación, una de las instrucciones definidas a continuación. En el caso en el que sea una instrucción de tipo if, while o for, no terminará en punto y coma, pero si la instrucción es de tipo asignación o de cualquier otro tipo, terminará en punto y coma.

4.11. Intrucción while

Para representar bucles condicionales utilizaremos la siguiente sintaxis: la palabra reservada "while"seguida de paréntesis que contendrán una expresión (deberá ser booleana) y seguido de llaves que contendrán instrucciones.

4.12. Instrucción for

Para representar bucles con contadores utilizaremos la suguiente sintaxis: la palabra reservada "for"seguida de paréntesis que contendrán exactamente 3 instrucciones (declaración de variable, expresión booleana y una instrucción) seguido de llaves que contendrán instrucciones. Se separarán entre sí usando ';' y la última también habrá de terminar en este caracter.

4.13. Instrucción if simple

Para la instrucción condicional simple utilizaremos la sintaxis siguiente: la palabra reservada if seguida de paréntesis que contendrán una expresión (será de tipo booleana) y a continuación llaves que contendrán instrucciones.

4.14. Instrucción if compuesta

Para la instrucción condicional compuesta utilizaremos la sintaxis siguiente: la palabra reservada if seguida de paréntesis que contendrán una expresión (será de tipo booleana) y a continuación llaves que contendrán instrucciones. Seguidamente aparecerá la palabra reservada else seguida de llaves que contendrán instrucciones.

4.15. Instrucción switch*

Para la instrucción switch utilizaremos la siguiente sintaxis: la palabra reservada switch seguida de paréntesis que contendrán un identificador seguido de llaves que contendrán bloques de cases consistentes del la palabra reservada case seguida de un identicador seguido de código entre llaves. Existe un bloque case especial que en lugar de empezar por la palabra case empieza por la palabra default.

4.16. Instrucción break

La instrucción break tendrá la sintaxis de siguiente: la palabra 'break'. Se utilizará para salir de los bucles.

4.17. Instrucción continue

La instrucción continue tendrá la sintaxis de siguiente: la palabra 'continue'. Se utilizará para saltar a la siguiente iteración en los bucles.

4.18. Instrucción return

La instrucción return tendrá la siguiente sintaxis: la palabra reservada 'return' seguida de una expresión. Se utilizará argumento de retorno de una función.

4.19. Instrucción new*

La instrucción new tendrá la siguiente sintaxis: la palabra reservada 'new' seguida de un identificador de tipo simple o compuesto. Podrá estar seguido de una o más aperturas de corchete con un número entre ellos, esto indicará que son arrays. Se utilizará para reservar memoria dinámica.

4.20. Instrucción delete*

La instrucción delete tendrá la siguiente sintaxis: la palabra reservada 'delete' seguida de un identificador. Se utilizará para eliminar memoria dinámica.

4.21. Instrucción print

La instrucción print tendrá como sintaxis la palabra print, seguida de un paréntesis de apertura, un identificador de tipo básico (o tipo de usuario que referencie a uno básico) y un paréntesis de cierre. Esta instrucción imprimirá por pantalla este valor.

4.22. Definición de tipos de usuario

La palabra reservada 'typedef' se usará en el ámbito global para definir alias de cualquier tipo.

4.23. Identificador this

La palabra reservada 'this' comporta como un identificador de clase y podrá aparecer allá donde aparecen estos. Luego a nivel semántico manejaremos qué pasa si la utilizamos en un contexto inadecuado.

5. Gestión de errores

La gestión de errores va a indicar el tipo de error, la ubicación en el fichero de código fuente (fila y columna) y parará la compilación. Se intentará proseguir la compilación para detectar más errores.

6. Ejemplos

```
void main(){
    int a = 5;
    int b[3] = [1, 2, 3];
    pointer int c = new int;
    *c = 5 + (7 % 2);
    b = *c + a;
    pointer int hola = new int[5];
    for(int i = 0; i < 27 + b[2]; i = i + 1;){
        if(b[i] % 2 == 0){
            continue;
        }
        else{
            break;
        }
     }
     int i = 0;
     bool finished = false;
     while(i <= 2 && !finished){
        delete hola[i];
        i = i + 1;
     }
}
```

```
//Esto es un comentario
struct{
    int a;
    int b = 2;
}tStruct;

class{
    int a;

    tClase build(int a){
        this.a = a;
    }

```

```
    void protocolo(int a){}
    int function() {
        this.a = this.a + 1;
        return this.a;
    }

}tClase;

void idonthaveatype(bool &b){
    b = false;
}

int ihaveatype(pointer int k, int j){
    *k = j + 1;
    return *k;
}

void main(){
    tClase ejemplo = tClase(5);
    ejemplo.protocolo(2);
    int a = ejemplo.funcion();
    bool c = true;
    idonthaveatype(c);
    ihaveatype(&a, 5);
}

void main(){
    tClase ejemplo = tClase(5);
    ejemplo.protocolo(2);
    int a = ejemplo.funcion();
    bool c = true;
    idonthaveatype(c);
    ihaveatype(&a, 5);
}
```

```
typedef int aux;

void main(){
    int a[2] = [5, 2];
    switch(a){
    case 1 {
        int c = 3;
        break;
    }
    case 2 {
        if(a == 2){
            aux b = 3;
        }
    }
    default {
        int d = 3;
    }
    }
}
```

```
int hf(int n){
   if(n<0){
      return 0;
   }else{
      if(n==0){
  return 0;
      }else{
  return n - hm(hf(n-1));
      }
   }
```

```
}

void main(){
int n = read();
print(hf(n));
}

int hm(int n){
    if(n<0){
        return 0;
    }else{
        if(n==0){
  return 0;
        }else{
  return n - hf(hm(n-1));
        }
    }
}
```

```
void main(){
    int n = read();
    print(fibo(n));
}

int fibo(int n){
    if(n < 2){
  return 1;
}else{
    return fibo(n-1) + fibo(n-2);
}
}
```

7. Palabras reservadas

- int
- for
- continue
- new
- delete
- char
- while
- const
- return
- typedef
- bool
- if
- void
- true
- pointer
- class
- switch
- default
- false
- read
- struct
- break
- case
- this
- print

8. Cambios y recortes

Hemos implementado todo lo descrito anteriormente menos la generación de código (osea, léxico, sintáctico, binding, tipado, gestión de errores...), no obstante, a la hora de generación de código (por las limitaciones que te comentamos en aquél correo) ha habido algunos cambios y recortes. Todo lo expuesto a continuación ha sido implementado completamente hasta la generación de código (no incluida), respecto a la generación de código, algunas cosas han sido implememntadas parcialmente o con simplificaciones, mientras que otras no han sido implementadas directamente.

8.1. Chars

El tipo char directamente no existe. Aunque la sintaxis y el tipado funcionan bien para este tipo, no se llega a generar código que lo maneje, por esto no se puede usar este tipo en los programas (aunque la sintaxis lo permita).

8.2. Switch

El switch funciona correctamente, pero con una limitación. Los valores de las distintas clausulas çase"tienen que ser enteros positivos.

8.3. Tipos de las funciones

Aunque la sintaxis lo permita, la generación de código no permite que las funciones devuelvan tipos no básicos (los alias de tipo de usuario que referencien a tipos básicos, se consideran básicos).

8.4. Continue

La instrucción de control continue no genera código. Si se hace la comprobación de que cuando aparece esté en un sitio permitido, pero no se llega a generar el código que la implementa.

8.5. Constantes

Si bien se pueden declarar variables como constantes, en ningún momento se considera un error de compilación el intentar cambiar el valor a una de estas, así que aunque se declaren como constantes, funcionan como variables normales.

8.6. Dimensión de los arrays

Al declarar un array, la dimensión tiene que tener un valor constante, no puede declararse con una variable.

8.7. Asignación de listas

Las listas no se pueden asignar de forma directa. La instrucción "int a[2] = [1,2]" no genera correctamente el código. Para trabajar con listas, tendríamos que declararlas primero y luego poner los valores uno a uno: "int a[2]; a[0] = 1; a[1] = 2;".

8.8. Paso de parámetros a función

En los parámetros de una función, no se pueden pasar listas de manera literal, solo mediante identificadores. Para hacer "funcion([1,2]);" habría que hacer "int aux[2]; aux[0] = 1; aux[1] = 2; funcion(aux);".

Además, tampoco se pueden pasar clases por referencia.

8.9. Declaraciones globales

No existen las declaraciones globales.

8.10. Memoria dinámica

No existe la memoria dinámica. Es decir, no podemos declarar variables tipo puntero, no podemos pedir espacio en el heap con la instrucción "new" y, por supuesto, no podemos borrar memoria dinámica con la instrucción "delete". Por tanto, los punteros realmente no existen.

9. Compilación de un programa

En esta sección describiremos como compilar y ejecutar código nuevo, así como ejecutar el código que damos de ejemplo.

9.1. Expresiones sueltas

Nuestro lenguaje no permite poner en mitad del código una expresión. Esto es aceptado por el compilador, pero a la hora de generar el código, dará un error (se quedará basura en la pila de wasm).

9.2. Ejecutar código de ejemplo

En el directorio del proyecto existe una carpeta llamada "codigo_fuente". En esta carpeta figuran todos los ejemplos de nuestro lenguaje (tanto los compilables, como los que tienen errores en compilación). Los llamados "ejemplo*n*.txt" son los compilables, los otros tienen errores de compilación.

Si se desea hacer un código nuevo, el código fuente (formato txt) tendrá que estar en la carpeta mencionada anteriormente.

Para los códigos que ofrecemos como ejemplo, ya están compilados en la carpeta "compilados". En esta carpeta hay, por cada código fuente, un archivo wat (que se genera de manera intermedia al compilar, pero que hemos decidido no borrar con propósito de poder leer el código generado), un archivo wasm (que es el archivo binario que se ejecuta) y un archivo js, todos con el mismo nombre que el código fuente.

Para ejecutar cualquiera de estos códigos, simplemente abrir una consola en esta carpeta y ejecutar el comando "node ejemplo*n*.js". Seguidamente habrá que introducir tantos números como requiera la entrada del programa en particular.

9.3. Crear código nuevo

Para crear un código nuevo, o modificar uno existente, el procedimiento es el siguiente. Primero creamos nuestro código, un fichero de texto plano (formato .txt), y lo colocamos en la carpeta "codigo_fuente". Ejecutamos el main de nuestro compilador (el programa de java), lo que creará en el directorio del proyecto dos archivos nuevos.

El primero es simplemente el archivo wat, que podemos borrar si queremos (no lo borramos automáticamente para poder leer el código ensamblador generado). El segundo es el archivo wasm, que será necesario para ejecutar.

Para poder ejecutar el programa, simplemente abrir el archivo .exe.jsz modificar la línea 50 (para que el nombre del archivo a ejecutar sea el correcto) y la línea 57 (Si nuestro programa no va a tener entrada, comentar la línea, si nuestro programa va a tener entrada por consola, descomentarla y poner en la función cuántas entradas tiene que leer).