MEMORIA DE LA PRÁCTICA

PROCESADORES DEL LENGUAJE

Pablo Sanz Caperote y Daniel Valverde Menasalvas

ÍNDICE

1. CARACTERÍSTICAS BÁSICAS DEL LENGUAJE

El lenguaje desarrollado es un lenguaje imperativo inspirado en C++. Sus características principales serán las siguientes:

- No será necesario indicar un punto de entrada (main) a los programas. Las instrucciones se ejecutarán en el orden que estén escritas a excepción de las funciones. Es decir, primero se comprueba de forma léxica, sintáctica y semánticamente que el código no tiene errores. Por este motivo, las funciones deben declararse antes de llamarse.
- Todas las instrucciones se escriben en una sola línea. Es decir, cada vez que existe un salto de lineal el lenguaje interpreta que es otra instrucción diferente y si no ha sido terminada producirá un error.
- No soportará *includes*, los programas se escribirán en un único fichero fuente.

2. MODIFICACIONES Y LÍNEA TEMPORAL

PRIMERA ENTREGA

En la primera entrega ideamos una primera versión de nuestro lenguaje. Especificamos la sintaxis original, las operaciones y las funcionalidades que iba a soportar. No obstante, algunas de estas primeras ideas fueron modificadas a lo largo de la creación del proyecto.

SEGUNDA ENTREGA

TAREAS REALIZADAS

En esta entrega implementamos el análisis léxico y sintáctico de nuestro lenguaje.

En el archivo *AnalizadorLexico.l* hemos incluidos todas las unidades léxicas. Estableciéndose así las palabras reservadas que va a tener. Por otro lado, mediante el análisis sintáctico, se comprueba la correcta estructura de las unidades léxicas. Para ello, en el archivo *ConstructorAST.cup* se describe una gramática que nos permite reconocer las estructuras apropiadas y avisar de que existen fallos en el caso que no reconozca un patrón. Esta gramática, formada por dos partes bien diferenciadas, reconoce tanto las expresiones como las instrucciones válidas. Además, crea una clase para cada expresión e instrucción en la que se almacenan sus datos propios.

Estos dos procesos son llamados desde el *Main.* Que, además, después de comprobar que el análisis léxico y sintáctico es correcto, imprime el árbol sintáctico obtenido, con el que se puede apreciar la estructura del código.

MODIFICACIONES RESPECTO A LA ENTREGA 1

Añadidos

- Posibilidad de usar *arrays* multidimensionales
- Instrucción de escritura.

Eliminaciones

- Se eliminó la posibilidad de utilizar punteros.

Modificaciones

- Cambios en la forma de definir *arrays*.
- Cambios en la forma de declarar variables de tipo *struct*.

TERCERA ENTREGA

TAREAS REALIZADAS

En esta tercera y última entrega se desarrolla el grueso de la práctica. Se implementa el analizador semántico y la generación del código.

El analizador semántico está formado por dos funciones recursivas independientes. Una primera función *vincula* almacena en la tabla de símbolos los identificadores usados teniendo en cuenta los ámbitos de las variables. De esta manera, comprobamos que no hayamos usado un identificador que no haya sido previamente declarado, o que estemos llamando a una variable local fuera del ámbito correspondiente. La segunda función *compruebaInstruccion* comprueba, como su nombre indica, que los tipos estén correctamente usados. Es decir que, por ejemplo; no se pueda asignar *False* a una variable de tipo *int*. Así, aseguramos que nuestro programa este correctamente tipado y que, si se dan errores de tipos inconsistentes, informemos de ello.

Por otro lado, en la generación de código también podemos identificar dos partes distintas que explicamos a continuación. Para generar código, deberemos haber superado con éxito los anteriores análisis realizados (léxico, sintáctico y semántico). Una vez superados, procedemos a generar el código.
En primer lugar, recorremos el código buscando declaraciones de variables para asignarles direcciones de memoria de manera estática. Para ello, tenemos una lista de bloques. Estos bloques contienen varias listas que almacenan la dirección de memoria correspondiente a cada variable declarada, y el tamaño de cada tipo declarado. Una vez hemos asignado direcciones de memoria a cada una de nuestras variables, estamos en disposición de generar el código propiamente dicho. De ello se encargan 3 funciones: *codeE, codeD* y *codeI*, que generan el código WebAssembly de las expresiones, identificadores e instrucciones respectivamente.

Aunque el resto de la práctica está preparado para trabajar con muchas funcionalidades, nos hemos visto obligados a reducir algunas de ellas para esta última parte de la práctica. No obstante, nuestro lenguaje es capaz de trabajar con ejecuciones de operaciones aritméticas entre *int* y entre *booleans,* bucles *while*, instrucciones condicionales *if* y *switch*, creación, acceso y uso de *strucst* y por último declaración, acceso y uso de *arrays* unidimensionales.

MODIFICACIONES RESPECTO A LA ENTREGA 2

Añadidos

- Posibilidad de acceso a variables de *arrays* multidimensionales

Eliminaciones

- Para la parte de generación de código se han suprimido las funciones y los *arrays* multidimensionales.
- Eliminación *arrays* de *struct.*

Modificaciones

- Forma en la asignación de *arrays* multidimensionales.
- Los campos del *struct* no admiten *arrays.*
- No se admiten funciones que devuelvan arrays.

3. ESPECIFICACIÓN DEL LENGUAJE

TIPOS

Para declarar una variable de un cierto tipo es necesario escribir antes del nombre de la variable la palabra reservada que define a cada tipo. Es posible declarar variables con valor inicial (mediante el uso del carácter "=" detrás del nombre). Si no se asigna un valor se asigna uno por defecto dependiendo del tipo de la variable. Nuestro lenguaje cuenta con los siguientes tipos:

VARIABLES SIMPLES:

- **Enteros:** Se identifican con la palabra reservada *int*. El valor por defecto es 0.
- **Booleanos:** Se identifican con la palabra reservada *boolean*. Únicamente existirán los valores *True* y *False* que son palabras reservadas. El valor por defecto es *False*.

```
int n = 5;
int n;
boolean b = True;
boolean x;
```

ARRAYS

Los *arrays* seguirán una lógica similar a las variables simples.

- Se pueden crear *arrays* de todos los tipos soportados por el lenguaje.
- La creación de *arrays* multidimensionales se hará escribiendo secuencias de corchetes con el tamaño de cada dimensión. Los *arrays* multidimensionales solo pueden ser de un tipo.
- Se declaran escribiendo la palabra reservada del tipo de la variable seguida por su tamaño entre corchetes y posteriormente el identificador. La longitud de este será el número indicado entre corchetes.

A su vez, el *array* puede ser inicializado o no. Para hacerlo, hay que añadir tras el carácter "=" los valores que tomará cada posición del *array* entre corchetes. En caso de que no sea inicializado, cada posición del *array* será inicializada por defecto. La inicialización de los *arrays* multidimensionales no se hace igual que los unidimensionales ya que no se pueden encadenar secuencia de llaves. Para hacerlo correctamente es necesario definir previamente los *arrays* que van a conformar el *arrays* multidimensional.

Por otra parte, para acceder a una posición del *array* se seleccionará la posición entre corchetes. Si el *array* es multidimensional, se pondrán una lista de posiciones entre corchetes una detrás de otra. De la misma forma, se puede modificar el valor de una cierta posición del *array*.

```
int[3] n = {1, 4, 5};
int[4] n;
Boolean[2][2] b;
Boolean[2] b1 = {True, False};
Boolean[2][3] b2 = {True, True};
b = {b1, b2};
Boolean b3 = b[1][1];
n[2] = 10;
```

REGISTROS

Para trabajar con *struct* hay que seguir dos pasos. El primero de ellos es la creación del tipo concreto del *struct* que es una instrucción y se verá más adelante. Posteriormente hay que declarar una variable de un tipo *struct* seguido de un identificador.

Para declarar un *struct* de un tipo previamente creado se hace de la siguiente forma:

```
struct myStruct nombreStruc;
```

Para acceder a los campos del struct se utiliza el operador "**.**".

```
nombreStruct.n = 10;
nombreStruct.b = True;
```

Operadores

Existirán operadores que nos permitirán operar con los tipos *int* y *boolean*.

Booleanos: "&", "|", "!", "==".

Enteros: "+", "-", "*", "%", "/", "<", ">", "<=", ">=", "==".

Estos operandos tienen la precedencia normal. Para que el compilador reconozca las operaciones tiene que existir un carácter en blanco entre el operador y el segundo operando.

INSTRUCCIONES

Como se ha explicado antes, las instrucciones del lenguaje solo pueden estar formadas por una única línea y además deben terminar por el carácter *;*.

INSTRUCCIÓN DECLARACIÓN

Las declaraciones de los tipos de variables se han expresado en la subsección anterior a la vez que se iban definiendo los distintos tipos.

INSTRUCCIÓN ASIGNACIÓN

La sintaxis de la asignación para variables simples será la siguiente:

Identificador = Valor;

En el caso de los *array* se añade un corchete con el número del elemento del *array* al que queremos acceder. Si queremos acceder al elemento *i* del *array* Identificador se hará:

Identificador[i] = Valor;

El caso multidimensional es análogo. Para acceder al elemento *i* del *array j* del *array* multidimensional Identificador se hará:

Identificador[N][M] = Valor;

En el caso de una estructura, para acceder a un campo del *struct* Identificador se utilizará el operador .:

Identificador.n = Valor;

INSTRUCCIÓN STRUCT

La creación de los *struct* se realiza poniendo la palabra reservada *struct* seguida del nombre del tipo del *struct* y luego entre llaves todos los campos que lo van a componer que pueden ser de cualquiera de los demás tipos permitidos, incluidos otros registros.

No pueden existir *struct* vacíos, es decir; sin ningún campo en su interior. Y tampoco pueden existir variables constantes en su interior.

```
struct myStruct{
   int n = 7;
   boolean b;
   int[2] n;
};
```

BUCLES

Se incluyen el bucle *while* que tendrá la siguiente sintaxis:

```
while (condición) {
   cuerpo
}
```

INSTRUCCIONES CONDICIONALES

El caso básico es la sentencia *if(*condición), pero también puede estar formado por *if*(condición) - *else* que es opcional.

```
if (condición){
   cuerpo_if
}

if (condición){
   cuerpo_if
}
else {
   cuerpo_else
}
```

Por otra parte, también existen instrucciones de *Switch*. Existe una expresión que se evalúa y varios casos con expresiones del mismo tipo que la expresión del switch. Se va comparando el valor de la expresión principal con la de cada caso y se ejecutan las instrucciones del cuerpo de ese caso. Los valores comparativos tienen que ir desde el valor 0 hasta el valor *n.* Existe una última expresión que se ejecuta si ninguna de las anteriores lo ha hecho. Su esquema general es el siguiente:

```
switch (expresion){
   case 0 {
         cuerpo_case_0
   }
   case 1 {
         cuerpo_case_1
   }
   .
   .
   .
    case n {
         cuerpo_case_n
   }

   default {
         cuerpo_default
   }
}
```

BLOQUES ANIDADOS

Usaremos llaves para distinguir los bloques anidados.

La indentación no será obligatoria, pero sí recomendable.

Usaremos un ejemplo con sintaxis de *if* y *while*:

```
while (i < 4) {
   if (i > 2){
   }
}
```

DECLARACIÓN FUNCIONES

Las funciones se declaran indicando el tipo que devuelven (*void* si no devuelven nada y serán conocidas como procedimientos).

A continuación, un identificador del nombre de la función y seguido de la declaración de los parámetros que recibirá con sus respectivos tipos separados por comas. Los parámetros siempre se pasarán por copia.

Las instrucciones que componen la función estarán encerradas en llaves. Si la función no es declarada como *void*, la última instrucción de esta deberá ser un *return* que devuelva una variable del tipo indicado previamente.

Las funciones pueden recibir y devolver parámetros de cualquier tipo existentes en el lenguaje.

```
int resta(int x, int b){
   return a+b;
}
void cuadrado (int x, int y){
   int z = x*y;
}
```

LLAMADAS A FUNCIONES

Aquí podemos establecer una división ya que no es lo mismo llamar a funciones que a procedimientos:

- Llamadas a procedimientos: Las llamadas a procedimientos sí que son entendidas como instrucciones como tal. Se hacen de la siguiente forma:

  identificador = identificadorFuncion(parámetros);

- Llamadas a funciones: Las llamadas a funciones que devuelven una variable son entendidas como expresiones ya que cumplen la misma función.

INSTRUCCIÓN DE ESCRITURA

Esta instrucción permite escribir por pantalla la variable deseada mediante la palabra reservada "*print*(variable)". Se permite ejecutar dicha instrucción únicamente con expresiones y tipos básicos. Es decir, no se puede imprimir por pantalla directamente un array, habría que hacerlo valor por valor; o en el caso de los estruct, campo a campo.

```
print(4+3);
print(x);
```

GESTIÓN DE ERRORES

Se indica el tipo de error mediante un mensaje explicativo. Además, se indica también el número de error que es contando todos los que aparecen en el código, señalando la fila y la columna en la que se encuentran.

4. EJEMPLOS

La ejecución de los archivos de prueba se realiza pasando al programa el nombre del fichero y su ruta por parámetro.

Los ejemplos de prueba se encuentran en una carpeta llamada *test* que a su vez contiene varias subcarpetas. La subcarpeta *final* alberga varios archivos de texto que simulan programas que generan códigos. Estos archivos son los más importantes y en los que se refleja el resultado final de la práctica.

Por otra parte, existen otras subcarpetas, cada una de las cuales contiene varios archivos que prueban diferentes partes de la práctica como por ejemplo los errores semánticos o sintácticos. Estos archivos son simplemente ejemplos de código bien y mal escrito con su correspondiente salida de errores, en los que queda reflejado que el lenguaje es consistente, está a prueba de errores y los reconoce cuándo los hay.