UNIVERSIDAD COMPLUTENSE DE MADRID

Doble grado en Ingeniería Informática y Matemáticas

Procesadores del Lenguaje

Memoria de la Práctica

Resumen

Este documento está dedicado a la tercera entrega de la práctica de la asignatura Procesadores del Lenguaje. En esta útlima entrega, especificamos los cambios realizados con respecto a la entrega anterior, siendo estos los definitivos. Asimismo, describimos el proceso que hemos seguido para la implementación de la práctica. Por último, incluimos la sintaxis que presentamos en la primera entrega, pero esta vez en su versión definitiva.

Juan Carlos Villanueva Quirós Miguel Franqueira Varela

15 de Junio de 2020

Índice

1. Modificaciones y Línea Temporal

1.1. Primera Entrega

En la primera entrega, pensamos en una primera versión de nuestro lenguaje. Especificamos la sintaxis original, las operaciones y las funcionalidades que iba a soportar nuestro lenguaje. En posteriores entregas, como vemos luego, tuvimos que modificar algunos aspectos de esta primera versión.

1.2. Segunda Entrega

1.2.1. Modificaciones con respecto a la Primera Entrega

En la segunda entrega, realizamos algunas modificaciones con respecto a la especificación del lenguaje de la primera entrega.

Añadimos

- Se ha añadido la posibilidad de usar structs, así como el operador . para poder acceder a sus respectivos campos.
- Se ha añadido el operador módulo % cuya funcionalidad es la convencional.

Cambiamos

- El carácter de fin de instrucción ahora es el ;
- El tamaño de los vectores ahora se declara despues del identificador.

Eliminamos

- Se omiten los caracteres iniciales de los identificadores.

1.2.2. Tareas Realizadas

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

En el archivo *ejemplo.l* hemos incluidos todas las unidades léxicas *tokens* que van a formar parte del léxico. Establecemos así las palabras reservadas que vamos a tener. Por otro lado, mediante el análisis sintáctico, comprobamos la correcta estructura de las unidades léxicas. Para ello, en el archivo *Tiny.cup* hemos descrito una gramática que nos reconozce las estructuras apropiadas. Esta gramática, formada por dos partes bien diferenciadas, reconoce tanto las expresiones como las instrucciones válidas en nuestro lenguaje. Además, crea una clase para cada expresión e instrucción en la que se almacenan los datos propios.

Por último, en el *Main*, hemos creado dos funciones que nos permiten imprimir el árbol sintáctico obtenido, pudiendo así apreciar la estructura de nuestro código.

1.3. Tercera Entrega

1.3.1. Modificaciones con respecto a la Segunda Entrega

Añadimos

- Se ha añadido la posibilidad de usar punteros, así como el operador $ para poder acceder a la variable apuntada. Podemos reservar (pero no liberar) memoria dinámica. Describimos la sintaxis para la declaración de punteros en la siguiente sección.
- Gestionamos y recuperamos los errores. De esta forma, somos capaces de detectar más de un error en una sola compilación.

Cambiamos

- Los arrays solo pueden ser declarados con dimensiones estáticas. Nuestro lenguaje no soporta arrays de dimensiones dinámicas.
- Nuestro lenguaje tampoco soporta la declaración de funciones y procedimientos anidados.

Eliminamos

- Las palabras reservadas *then* y *do* han sido suprimidas pues consideramos que no aportan nada.
- El operador exponencial ha sido eliminado al no contar con esta operación en la máquina.
- El operador menos ha sido suprimido.

1.3.2. Tareas Realizadas

Esta tercera y última entrega conforma el grueso de la práctica. Implementamos aquí el análisis semántico y la generación del código.

El analizador semántico está formado por dos funciones recursivas independientes. Una primera funcion *vincula* almacena en la tabla de símbolos los identificadores usados. 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 *compruebaTipos* comprueba, como su nombre indica, que los tipos estén correctamente usados. Esto es por ejemplo, que no podamos 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*. Cada uno de estos bloques representa un ámbito de nuestro programa. Contienen un atributo *marcoActivación* que indica si el bloque abre un nuevo marco de activación, es decir, estamos declarando una nueva función/-procedimiento, o si el bloque abre un ámbito local, es decir, estamos en el cuerpo de un if, while, switch...

Estos bloques contienen varias listas que almacenan la dirección de memoria correspondiente a cada variable declarada, y el tamaño de cada tipo declarado[1].

Una vez hemos asignado direcciones de memoria a cada una de nuestras variables, estamos en disposición de generar el código propiamente dicho. Para ello, usamos tres funciones mutuamente recursivas:

- generaCodigoExpresion: genera código para expresiones.
- generaCodigoLeft: genera código para la parte izquierda de una asignación.
- generaCodigoInstruccion: dada una instrucción, genera el código correspondiente de esta.

Tras ejecutar dichas funciones, habremos generado nuestro código máquina en un archivo *.txt* listo para ser ejecutado en la *máquina-P*.

[1] También almacenamos las dimensiones de los arrays y las direcciones relativas de los campos de los structs declarados.

2. Especificación del Lenguaje

2.1. Identificadores y ámbito de definición

2.1.1. Declaración de variables simples

- Las variables pueden ser declaradas con un valor inicial (usando el carácter '=' detrás del nombre). Si no son declaradas con valor inicial, se inicializarán a un **valor por defecto** dependiendo del tipo al que pertenezcan (Véase la tabla (1)).

Ejemplos:

```
int n = 5; #la variable con identificador n toma el valor 5
int n; #la variable con identificador n toma el valor por defecto 0
```

2.1.2. Declaración de arrays

- La declaración de arrays sigue una lógica similar a las variables simples. Para declarar que es un array, hay que añadir corchetes después del identificador, especificando el tamaño del array.

- Acto seguido, el array puede ser inicializado, añadiendo detrás el simbolo '=' y los valores que tomará cada posición del array entre llaves . En caso de que el array no sea declarado con un valor inicial, cada elemento será inicializado a su valor por defecto.

Ejemplos:

```
int n[4] = {0,1,2,3};
int n[4]; #array con 4 elementos inicializados al valor por defecto: 0
```

2.1.3. Declaración de estructuras

- La declaración de structs se hará declarando todas las variables simples en el cuerpo seguidas del identificador del struct.

- Las declaraciones puede ser inicializadas, añadiendo detrás el simbolo '=' y el valor de la variable simple o array. En caso de que la variable o array no sea declarado con un valor inicial, cada elemento será inicializado a su valor por defecto.

Ejemplo:

```
struct{
    int n = 3;
    boolean p;
}tEjemplo;
```

2.1.4. Declaración de Punteros

- La declaración de punteros se hará mediante el símbolo $.
- Los punteros pueden ser inicializados, añadiendo detrás el símbolo '=' y usando el operador *new*. Este operador reservará espacio en la memoria dinámica. Irá seguido del tipo del puntero y de la dimensión deseada entre corchetes. Si solo queremos un dato, podemos no poner ningún entero entre los corchetes.

Ejemplo:

```
int$ p;
int$ p = new int[];
int$ p = new int[3];
```

2.1.5. Bloques anidados

Vamos a utilizar llaves para distinguir bloques anidados. Un ejemplo con la sintaxis del while explicada en el punto 3:

```
while (condition) {
        while (condition) {
                int n = 3;
        }
}
```

2.1.6. Opcionales

Los identificadores de las funciones y procedimientos serán una cadena de enteros y letras.

Funciones

Sintaxis:

```
tipo identificadorFuncion(tipoA identificadorA, ...){

    Cuerpo

    return identificador;
}
```

Ejemplo:

```
int suma(int a, int b){
        return a + b;
}
```

Procedimientos

Sintaxis:

```
proc identificadorProcedimiento(tipoA _identificadorA , ...){
        Cuerpo
}
```

Ejemplo:

```
proc suma (int a, int b){
    int c;
        c = a + b;
}
```

2.2. Tipos

2.2.1. Tipos Básicos

Tanto los enteros como los booleanos tendrán palabras reservadas. Estas palabras son **int** y **boolean** respectivamente. Los booleanos tienen como palabras reservadas los posibles valores que pueden tomar: true y false.

- Enteros: int Identificador;
- Booleanos: boolean Identificador;

2.2.2. Operadores Infijos

- Booleanos: &, |, !, ==
- Enteros +, -, *, **, %, /, <, >, <=, >=, ==

Siendo su significado, prioridad y asociatividad el común de la mayoría de lenguajes de programación ('**' equivale a la exponenciación).

2.2.3. Tipo Array

Sintaxis:

```
tipo identificador[tam];
```

El tipo del array a declarar deberá ser de un tipo básico un tipo de usuario definido previamente. Para declarar un array multidimensional, la sintaxis será la siguiente:

```
tipo identificador[tamA][tamB] = { ... };
```

Tipo	Valor Por Defecto
int	0
boolean	false

(1)

2.3. Conjuntos de instrucciones del lenguaje

2.3.1. Fin de Instruccion

El carácter de fin de instrucción elegido es el punto y coma ;.

2.3.2. Asignación

La sintáxis de la asignación para variables simples será la siguiente:
Identificador = Valor;

En el caso de los array añadiremos un corchete con el número del elemento del array al que queremos acceder. Si queremos acceder al elemento N del array Identificador se hará:
Identificador[N] = Valor;

El caso multidimensional es análogo. Para acceder al elemento M del array N 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;

2.3.3. Condicional

Puede ser formado con **else** (caso en el que no se cumple la condición) o prescindiendo de esta. La condición deberá ser especificada entre paréntesis.

Sin rama else:

```
if (condition) {
        Cuerpo
}
```

Con rama else:

```
if (condition) {
    Cuerpo
}
else {
    Cuerpo
}
```

2.3.4. Bucles

Utilizaremos la siguiente sintaxis del while:

```
while(condition) {
    Cuerpo
}
```

2.3.5. Constantes

Usaremos el rango de números enteros de 32 bits como posibles constantes para inicializar valores enteros. Asimismo añadiremos la opción de declarar variables que tomen un valor constante, siendo sustituídas por su valor en tiempo de compilación.

Sintaxis:

```
const tipo identificador = Valor;
```

Ejemplo:

```
const int constante1 = 3;
int n = constante1;
```

2.3.6. Opcionales

Case

Salto a cada rama en tiempo constante, pudiendo usar **default** en caso de que no se alcance ninguna de las ramas declaradas. Se utilizará la siguiente sintaxis:

```
switch (Identificador) {
    case valorA {

    }

    case valorB {

    }

    .
    .
    .

    default {

    }
}
```

Llamada a funciones

Llamaremos a las funciones de la siguiente manera:
identificador = identificadorFuncion(parámetros);

Llamada a procedimientos

Llamaremos a los procedimientos de la siguiente forma:
identificadorProcedimiento(parámetros);

2.4. Gestión de errores

Indicamos el tipo de error, la fila y la columna.

2.4.1. Opcionales

Implementamos la recuperación de errores, consiguiendo de proseguir la compilación tras un error con el fin de detectar más errores.

2.5. Ejemplos

A continuación veamos ejemplos de código siguiendo la sintáxis especificada en los puntos anteriores.

2.5.1. Número de primos de un array

Programa para calcular el número de primos de una lista:

[CÓDIGO EN NUESTRO LENGUAJE]

```
const int N = 10;
int numeroPrimos(int v[N]){
    int resultado = 0;
    int i = 0;

    while (i < N) {
        if(esPrimo(v[i])) {
            resultado = resultado + 1;
        }
        i = i + 1;
    }
    return resultado;
}

boolean esPrimo(int n){
    int i = 2;
    boolean resultado = true;
    while((i < n) & resultado) {
        #en el caso de que n%i sea 0, entonces el numero no es primo
        resultado = resultado & (n%i != 0);
        i= i + 1;
    }
    return resultado;
}
```

2.5.2. Controlador

Póngamonos ahora en el caso de que queremos decidir con que valores llamamos a esa función o a otra dependiendo de un parámetro de entrada. Por ejemplo, por facilitar la implementación, pongamos que nos pasan un número del 1 al 3 representando las diferentes posibilidades.

[CÓDIGO EN NUESTRO LENGUAJE]

```
proc controlador(int decision){
    int vect[10] = {2, 7, 13, 4, 6, 23, 7, 8, 20, 50};
    switch(decision){
        case 1 {
            vect[1] = 6;
        }
        case 2 {
            vect[2] = 14;
        }
        case 3 {
            vect[3] = 5;
        }
        default {
            #El valor decision no deberia valer algo que no sea 1, 2 o 3.
        }
    }
    int resultado = numeroPrimos(vect);
}
```