UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

PROCESADORES DE LENGUAJES

Generación de Código

Memoria final del compilador de nADA

Arbelo Cabrera, Daniel
Mireles Suárez, Alberto Manuel
Morales Sáez, David Guillermo
Quesada Díaz, Eduardo
Sánchez Medrano, María del Carmen

Índice

Descripción del lenguaje

Introducción

A lo largo de este documento se definen las características del lenguaje de programación nADA, un subconjunto del lenguaje ADA.

Hemos decidido emplear un lenguaje basado en ADA puesto que ya hemos trabajado con él y estamos familiarizados con el mismo. A su vez tiene la parte de declaraciones bien definida y es un lenguaje fuertemente tipado, lo cual garantiza que se trabajará con tipos compatibles. Finalmente, podemos añadir que la sintaxis del lenguaje ADA es muy similar a Pascal, que es un lenguaje apropiado para escribir un compilador.

Características del lenguaje

Identificadores

Los identificadores estarán formados por los caracteres alfanuméricos del alfabeto inglés, empezando siempre por una letra y de tamaño máximo 20 caracteres.

Literales

Admitimos tres tipos de literales: números enteros, en punto flotante y caracteres. Los enteros y reales ocuparán 4 bytes, mientras que los caracteres ocuparán 1 byte.

Operadores

- **Suma**: La suma está representada con el símbolo + y consiste en combinar dos o más números del mismo tipo.

```
(Literal) + (Literal)
```

- **Resta**: La resta es la operación inversa a la suma en la que dada cierta cantidad se elimina una parte de ella, siendo el resultado de esta operación la diferencia o resta. Se usará el símbolo – para esta operación.

```
(Literal) - (Literal)
```

- **Multiplicación**: La multiplicación es una operación matemática que consiste en sumar un número tantas veces como indicas otro número. Se emplea el símbolo * para realizar esta operación.

```
(Literal) * (Literal)
```

- **División**: La división es una operación aritmética de descomposición que consiste en averiguar cuántas veces un número (divisor) está contenido en otro número (dividendo). El símbolo / representa esta operación.

```
(Literal) / (Literal)
```

- **Igualdad**: Se basa en la comprobación de que el término a la derecha del símbolo "=" tiene el mismo valor numérico del que está a la izquierda, siendo ambos términos del mismo tipo (entero o punto flotante). Así mismo el valor devuelto en dicha comprobación es de tipo booleano, siendo de valor "`true`" en caso de que la comparación sea cierta y "`false`" en caso de que sea negativa.

- **Mayor que / Menor que:** Con estos símbolos se comprueba que el valor a la derecha del comprobante condicional tiene un mayor o menor valor numérico (según el símbolo utilizado) que el elemento que esté a la izquierda del condicional, que como en el caso de la igualdad solo acepta elementos de tipo entero o punto flotante. Como ocurre en la igualdad, el valor que indica dicha comprobación es un booleano, en caso de que sea "`true`" significa que el valor a la derecha cumple la condición comparado con el valor a la izquierda, y en caso de que se devuelva "`false`" indicará que no cumple dicha condición. La operación mayor que está representada por el símbolo “>”, mientras que menor que utiliza el símbolo “<”.

- **Mayor igual que / Menor igual que:** En este caso nos encontramos ante una amalgama entre los operadores anteriores, donde comprobamos que el elemento a la izquierda del condicional cumpla dos condiciones: que sea igual o menor o mayor (según el caso que corresponda). El tipo de datos que acepta son enteros o punto flotante y el valor que devuelve esta comparación es como en los casos anteriores, de tipo booleano: "`true`" si se cumple y "`false`" si no se cumple. Para mayor o igual que se emplea “>=”, mientras que para menor o igual que se utiliza “<=”.

- **Operadores binarios:** Los operadores binarios que contemplamos son `and`, `or` y `not`. El operador `and` realiza la función booleana de producto lógico, la cual da verdadero a la salida si las dos entradas valen verdadero; el operador `or` realiza la suma lógica, ofreciendo un valor verdadero a la salida siempre que alguno de sus argumentos sea verdadero; y el operador `not` realiza la negación lógica, convirtiendo un valor de entrada en su opuesto (verdadero en falso y falso en verdadero).

Variables

Las variables se declararán después de la línea de `procedure` o `function` y antes del `begin` del programa. Una declaración de variables consta de: una lista de nombres de las variables que se declaran separadas por comas (",") y terminada con dos puntos (":"), el tipo de las variables. La asignación de un valor inicial no se permite en este momento, se debe hacer posteriormente.

En caso de que se quiera asignar el valor a una variable la forma de proceder en caso de que se haga en medio de la ejecución del código será, simplemente, el nombre de la variable seguida de los dos puntos y el igual (":=") y posteriormente el valor a asignar.

Un procedimiento será capaz de acceder a variables declaradas en su mismo nivel y también tiene posibilidad de acceder a variables de niveles superiores.

```
procedure un_procedimiento (x : in integer) is
     --declaración de variables
          x : integer;
begin
          -- código
          x := 3;
          -- código
end un_procedimiento;
```

Tipos de datos construidos

Los tipos de datos construidos que utilizaremos en nADA son dos:

- Vectores
- Enumerados

Los vectores son un conjunto de elementos ordenados. Son de tipo simple (Entero, real o carácter) y dentro de un vector, todos los elementos han de ser del mismo tipo. Se puede acceder a cada elemento en particular a través de un índice, el cual se indica poniendo entre paréntesis el mismo después del nombre de la variable, siempre que se encuentre dentro del rango de valores especificado en la declaración del vector. A la hora de la asignación no se permite la asignación entre dos vectores, sino una variable o una expresión.

Su declaración se lleva a cabo con la palabra reservada "`array`", seguido del rango de valores encapsulado por paréntesis y el tipo de dato que será cada elemento del vector, el primer valor tiene que ser obligatoriamente 1 ya que se fuerza a que vaya desde 1 hasta el valor deseado. El rango se define con el valor inicial seguido de dos puntos y el valor final del rango. Para poder usar los vectores se deberá crear por tanto el tipo del vector, y luego se podrá declarar una variable de dicho tipo. A continuación declaramos un tipo de vector de 10 elementos de tipo entero:

```
type Array_Entero is array (1..10) of Integer;
     vector : Array_Entero;
```

Los tipos enumerados son un tipo de variable que solo puede tomar uno de los valores definidos en el tipo enumerado. Es decir, al crear el tipo se deben indicar los valores que podrá tomar una variable de ese tipo. Por ejemplo, si queremos un tipo de dato enumerado con el fin de consultar el color de un semáforo, sabemos que esa variable tomará tan solo 3 valores, por lo que podemos crear el siguiente tipo enumerado:

```
type enumerado is (ROJO, AMBAR, VERDE);
     semaforo : enumerado;
```

De esta manera, la variable "`semaforo`" solo podrá tomar los tres valores establecidos, y facilita el manejo de estructuras de selección, ya que se pueden emplear valores cuyo nombre sea significativo para el programador.

Ristras

Las ristras son un tipo de dato que conforman un conjunto de caracteres encapsulados por el símbolo ' " '. Tendrán como máximo un tamaño de 255 y un último carácter que represente el fin de ristra ('`\0`'). Las ristras pueden declararse de la siguiente manera:

```
ristra : String;
```

Comentarios

Se permitirán los comentarios de línea que vendrán precedidos por "`--`". Esto indica que los caracteres que se encuentren a partir de este símbolo y hasta el final de línea son comentarios. A su vez no permitiremos los comentarios de bloque.

Procedimientos y funciones

Dentro de nuestro lenguaje nADA tenemos definidas dos tipos de estructuras, los procedimientos y las funciones. Dentro de esta primera, los procedimientos, se encuentra la estructura principal de un programa, lo que en lenguaje C conoceríamos como la función `main`.

Procedimientos

Poseen una estructura que comienza con la palabra reservada '`procedure`', seguido del `nombre del procedimiento` y entre paréntesis las variables que posea el procedimiento, separas por ",":

```
Nombre de variable : tipo de variable
```

Seguidamente, encontraremos la palabra reservada 'is', que marca el inicio de la declaración de las variables internas del proceso, así como las declaraciones de los subprogramas, procedimientos o funciones, que vayan a ser llamados.

Tras la declaración, se debe encontrar la palabra reservada 'begin', con la cual comienza el código de ejecución del procedimiento. Este mismo terminará cuando se encuentre la palabra reservada 'end', seguido del nombre del procedimiento y finalizando con un ';'.

Hay que matizar, que si estamos en el procedimiento del programa principal este no posee variables de entrada, salida o entrada-salida, simplemente después del nombre del procedimiento le sigue la palabra reservada is. También hay que decir que si un programa quiere utilizar una variable o llamar a un subprograma, es estrictamente necesario que se encuentre declarado en entre el is-begin. Un ejemplo del uso de procedimientos sería:

```
procedure main is
-- Declaración de variables
    A: integer;
    B: integer;
    C: integer;
-- Procedimiento suma
   procedure suma (A1: in integer, B1: in integer, C1: out
integer) is
   begin
     C1 := A1+B1;
   end suma;
begin
     A := 1;
     B := 3;
     suma(A,B,C);
end
```

Donde tenemos el programa principal main, en el que después del is se realiza una declaración de variables, A, B, C de tipo entero, y luego se crea un subprocedimiento suma, el cual inserta en la variable de salida C1, el resultado de la suma de A1+B1. Este subprocedimiento es llamado por el programa principal, suma (A, B, C), donde C obtendrá el resultado de sumar, A+B es decir 1+3.

Hay que tener en cuenta, que cualquier procedimiento, sea el principal o no, puede tener definido en el bloque `is-begin` tantas funciones y procedimiento como sean necesarios, para la ejecución de la tarea a realizar.

Funciones

Una función consiste en un conjunto de instrucciones con un objetivo concreto que puede ser ejecutada desde otra función o procedimiento. Se diferencian de los procedimientos porque, a diferencia de éstos, las funciones han de devolver un único resultado, y sus parámetros (en caso de tenerlos) serán solo de entrada. Además, una función puede llamarse a sí misma dentro de su código, generando recursividad.

En nuestro lenguaje, la declaración de una función comenzará por la palabra reservada `function`, tras lo cual se indicará el nombre de la función y, entre paréntesis, los parámetros con su tipo de datos asociado y separados por comas (como las funciones sólo admiten parámetros de entrada, no se permite añadir la palabra reservada in junto a éstos). Luego se especifica el tipo de datos que devolverá la función, escribiendo `return [tipo de dato]`, y la palabra reservada `is`. Dentro de un bloque `begin … end [nombre de la función]` se incluirá el código de la misma. Para devolver el valor desde la misma función, se utilizará la sentencia `return`. Como una función prototipo puede considerarse la siguiente:

```
function [nombre de la función] (parámetro : tipo,
parámetro : tipo , …) return [tipo de dato] is
begin
      instrucción;
      instrucción;
      …
      instrucción;
      return [dato];
end [nombre de la función];
```

Un ejemplo para una función en nADA puede ser el siguiente:

```
function Minimo (A : Integer, B : Integer) return Integer
is
begin
      if A <= B then
            return A;
      else
            return B;
      end if;
end Minimo;
```

Paso de parámetros

El paso de parámetros se ha limitado a que solamente puedan ser de tipo de entrada, "in":

- `in`: Si la variable o expresión en cuestión es solo de entrada se hará una copia de dicha variable que existirá mientras dicha función todavía exista, y una vez acabe, la copia será destruida.

A la hora de declarar una variable dentro de un procedimiento ésta puede ser de 3 tipos:

- `in`: la variable o expresión es de entrada y su valor es usado en el procedimiento, puede ser actualizado dentro del mismo pero no es devuelto, por tanto en el procedimiento que lo llamó mantendrá el mismo valor con el cual fue pasado.

Estructuras de control

Este lenguaje dispondrá de las siguientes estructuras de control:

- Estructura condicional
- Estructura de selección
- Estructura de repetición

Como estructura condicional utilizaremos el bloque `if – then – else - end if`, estableciendo una condición y ejecutándose las instrucciones pertinentes en función de la misma. Por ejemplo:

```
if(condición) then
          instrucciones
else
          instrucciones
end if;
```

Para la estructura de selección, utilizaremos la instrucción `case – is – when - end case`, estableciendo la variable a evaluar y escogiendo la entrada correspondiente. Por ejemplo:

```
case variable is
     when alternativa => instrucciones;
end case;
```

Para la estructura de repetición, se empleará la instrucción `while – loop – end loop`, definiendo la condición de entrada del bucle justo después de la palabra reservada "`while`". Por ejemplo:

```
while (condición) loop
     instrucciones
end loop;
```

Cambios realizados

Opciones del lenguaje modificadas

Por falta de tiempo y debido a la complejidad de las siguientes opciones se ha decidido no incluirlas en la implementación final del lenguaje nADA:

- Niveles de anidamiento. Sólo permitiremos un procedimiento principal con subprocedimientos y funciones dentro del mismo, que podrán ser llamadas desde el procedimiento principal. Si una función ha sido implementada anteriormente a otra, la segunda podrá llamar a la primera dentro de su código. Sin embargo, no se podrá definir una función dentro de otra función. No se manejan declaraciones prototipo, por lo que sólo podrán ser llamadas subprocedimientos o funciónes que hayan sido declaradas anteriormente.
- Parámetros de entrada y salida. En procedimientos, habíamos determinado que se implementarían parámetros para pasar valores de entrada, de salida y de entrada/salida. Debido a complejidad, hemos decidido no implementar los parámetros de salida y entrada/salida.
- Manejo de ficheros. Habíamos desarrollado una librería prototipo para el manejo de ficheros como entrada y salida. Finalmente no se ha implementado esta característica.

Modificaciones tabla de símbolos

Debido a diversas necesidades a la hora de implementar la generación de código, hemos realizados diversas modificaciones a la tabla de símbolos:

- Creadas funciones para casos específicos de variable tipo (`insertaVariableTipo`).
- Creadas funciones para la carga y lectura de ristras, debido a nuestra forma de almacenarlas.
- Añadidas variables a nuestra tabla de símbolos. Por ejemplo, `rcima_aux` almacena el valor de `rcima` a la hora de pasar parámetros.
- `dirvar` se modificó para el caso en el que encontremos una variable de ámbito 1 y nos encontremos en el ámbito 2, se devuelva la dirección de la variable en base a Z, que es la dirección de los datos estáticos que se corresponde con el ámbito superior.

Adiciones a la gramática

Se han añadido una serie de funciones para la muestra de valores por pantalla:

- `printd(Integer)`. Permite la impresión de valores enteros por pantalla.
- `printf(Float)`. Permite la impresión de valores en coma flotante por pantalla.
- `prints(String)`. Permite la impresión de ristras por pantalla.
- `printc(Char)`. Permite la impresión de caracteres.

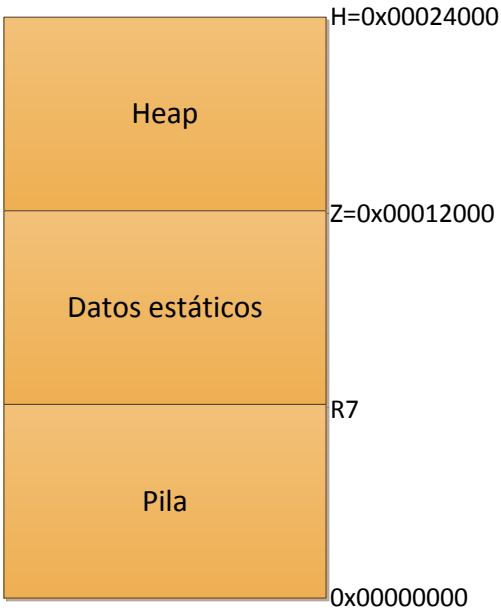
Modificaciones a la máquina Q

La llamada `putf_` ha sido modificada con el fin de que imprima según el tipo de datos que se le pasa, correspondiendo a las tres funciones que se han definido anteriormente.

Generación de código

Gestión de la memoria de Q

La memoria en Q se divide en tres segmentos de memoria: el segmento de heap, el segmento de datos estáticos y el segmento de pila (stack).



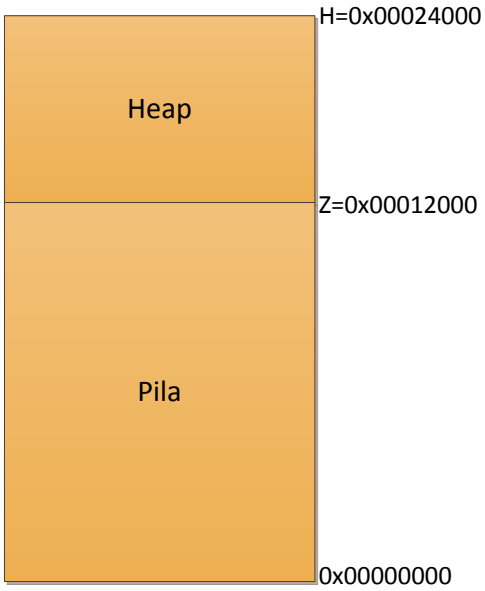
Los datos estáticos son los definidos en segmentos `STAT` del programa Q. El segmento de datos estáticos ocupa desde la dirección `Z=0x00012000` hasta la dirección de memoria que deje un tamaño de memoria tal que permita almacenar todos los datos estáticos. Dicha dirección de memoria se almacenará en el registro `R7` inicialmente. Por tanto, los bytes utilizables del segmento de datos estáticos son los bytes del `Z-1` hasta el valor inicial de `R7` (el valor de `R7` tras la carga de datos estáticos).

El segmento de pila comenzará en la dirección `R7` (justo después del segmento de datos estáticos) y se extenderá hasta la dirección `0x00000000` de memoria. Por tanto, los bytes utilizables del segmento de datos serán los bytes desde el `0x00000000` hasta el valor inicial de `R7-1` (el valor de `R7` tras la carga de datos estáticos).

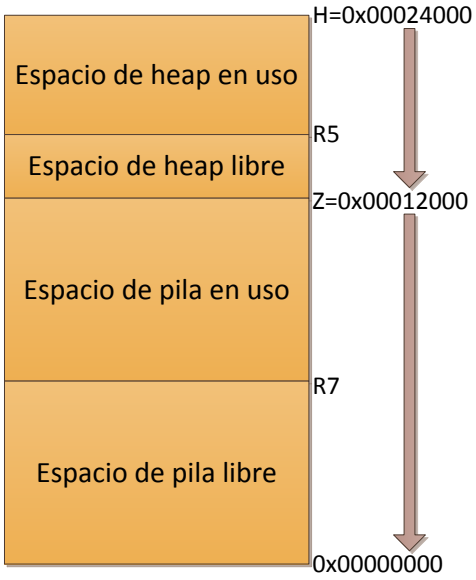
Sin embargo, la generación de código no produce datos estáticos, sino que todas las variables se almacenan en la pila. Por tanto, realmente la pila ocupará el espacio comprendido entre la dirección `0x00000000` y la dirección `Z=0x00012000`, dejando así utilizables los bytes desde el `0x00000000` hasta el `Z-1=0x00011FFF`.

El segmento de heap es el espacio comprendido desde la dirección `H=0x00024000` hasta la dirección `Z=0x00012000`. Por tanto, los bytes utilizables del heap son los bytes del `Z=0x00012000` al `H-1=0x00023FFF`. Puesto que no se permitirá el manejo de memoria dinámica, este segmento queda inutilizado en principio. Sin embargo, conviene utilizarlo para el almacenamiento de las cadenas de caracteres (strings) literales. De esta manera, se evita tener que hacer dos pasadas del código fuente que se está compilando, o tener que generar una representación del programa en un lenguaje intermedio, y toda la compilación se puede realizar en una única pasada y sin una representación intermedia.

La distribución de la memoria en la práctica queda de esta forma:



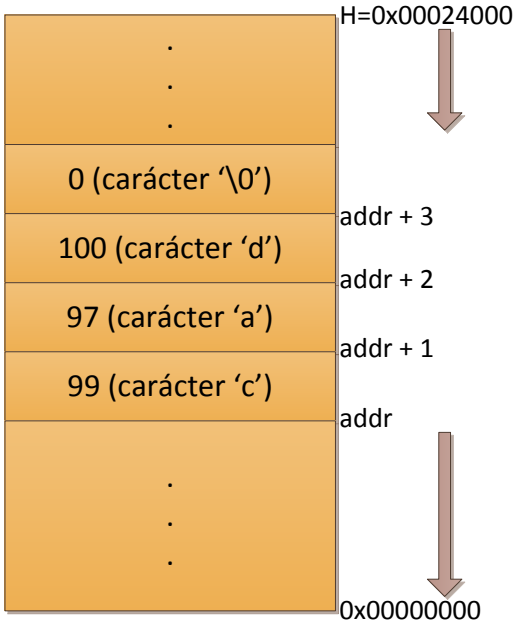
Durante la ejecución ambos segmentos están inicialmente vacíos y, a medida que se van reservando, van creciendo desde direcciones más altas de memoria hacia direcciones más bajas. Además, en ejecución se mantienen dos punteros apuntando al final de cada segmento: el registro `R5` apunta al final del segmento de heap, y el registro `R7` apunta a la cima de pila. A medida que el espacio utilizado de cada segmento crece, se actualiza el puntero (puesto que los segmentos crecen hacia direcciones más bajas, el puntero se decrementa). Cuando se libera una parte del segmento (del heap no se libera espacio, sólo de la pila), se incrementa el puntero de manera que apunta a la nueva cima.



Almacenamiento de ristras en el heap

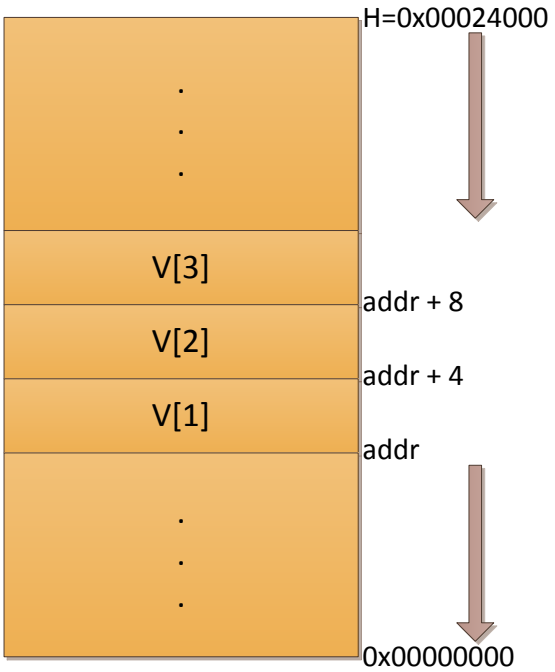
Cada vez que se encuentra un literal string al analizar el código fuente, se comprueba si está ya en la tabla de símbolos. Si no está en la tabla de símbolos, se inserta, y se calcula la dirección de memoria en el segmento de heap donde se empezará a almacenar la string. Si la string ya se encuentra en la tabla de símbolos, simplemente se recupera su dirección de memoria.

A la hora de almacenar los literales string en el heap, se almacenan en formato big endian, con el byte más significativo en direcciones menores de memorias, y se marca el final de la string con un byte nulo (el carácter de fin de cadena, tal como se hace en C estándar). Por tanto, el literal string `"cad"` se almacenará el heap a partir de la dirección `addr` de la siguiente forma:



Almacenamiento de variables locales

El almacenamiento de variables locales se hace en la pila. Para almacenar las variables en pila, simplemente se reserva el espacio del tamaño de la variable, teniendo en cuenta que un carácter ocupará 4 bytes en pila debido a la alineación de datos, y se almacena su valor en la posición reservada. Para las variables de tipo array se reservan tantas posiciones de memoria como el tamaño del mismo, y se almacena su posición de memoria inicial en la tabla de símbolos. Un array de tres elementos en pila se almacenaría de esta manera:



Almacenamiento de tipos enumerados

Cuando declaramos un tipo enumerado, almacenamos en la tabla de símbolos tanto el nombre del tipo como todos los posibles valores que pueda tener, cada uno de ellos con su nombre un una codificación mediante números naturales. A la hora de realizar una asignación de ese tipo enumerado, se le asigna dicha codificación al valor de la variable en cuestión. Se ha escogido este mecanismo para el tratamiento de enumerados porque resulta más sencillo de utilizar a la hora de programar este compilador.

Uso de registros

El compilador está implementado como una máquina de pila. Por ello, siempre que se realice un cálculo se cargan los operadores en `R0` y `R1` (ó `RR0` y `RR1`), almacenando el resultado en el propio `R0` y volcándolo inmediatamente a la pila. Por ello, cuando el cálculo se completa consideramos automáticamente que `R0` y `R1` (ó `RR0` y `RR1`) vuelven a estar disponibles y el derramado de registros es implícito.

Aparte de estos dos registros, usamos otros con propósitos específicos:

- `R7`: Señala a la dirección más baja disponible de la pila. Cuando reservamos nuevo espacio en la pila, se decrementa `R7`.
- `R6`: Señala a la base de la pila local (comienzo de los parámetros y variables locales) en la función actual. De esta forma, todos los parámetros y las variables locales tienen direcciones relativas a `R6`, y cuando se sale del ámbito de la función podemos liberar la memoria de todos los parámetros y variables locales igualando `R7` a `R6`.
- `R5`: Es equivalente en su funcionamiento a `R7`, con la diferencia de que gestiona la dirección disponible más baja del heap, donde se almacenan los literales string.
- `R4`: Almacena temporalmente el valor de `R7` durante el paso de parámetros en la llamada de una función. Tomamos el valor de `R7` antes de que se copien los parámetros a la pila, de manera que cuando asignemos el valor de `R4` a `R6` éstos formaran parte de las variables locales de la función llamada.
- `R3`: Al realizar un acceso a un vector, en `R3` se calcula la dirección del elemento a acceder.
- `R2`: Se emplea en la estructura `case`, para controlar la variable que estamos comparando inicialmente y a la hora de realizar una operación de salida por pantalla.

Evaluación de expresiones

En nuestro compilador a la hora de trabajar con expresiones, por ejemplo `a = a + b`, simulamos el funcionamiento de una máquina de pila, es decir, vamos a la pila a recoger los operandos al igual que dejamos el resultado en la misma para que lo recoja quien vaya a usarlo. Para ello usamos siempre dos registros `R0` y `R1` (ó `RR0` y `RR1`). De esta forma los pasos a seguir cuando nos encontramos con una expresión del tipo `a = b+c*d` son (en este ejemplo partimos de la base de que todos los valores ya están en la pila y los recogemos para evaluar la expresión):

- Por precedencia primero la multiplicación:
  - `R0 = Cima;  // Cargamos d`
  - `R1 = Cima + 4;  // Cargamos c`
  - `R0 = R0*R1;  // Realizamos la operación`
  - `Cima + 4 = R0;  // Almacenamos en pila el resultado`
  - `Cima = Cima + 4;`
- Finalmente la suma:
  - `R0 = Cima;  // Cargamos c*d`
  - `R1 = Cima + 4;  // Cargamos a`
  - `R0 = R0+R1;  // Realizamos la operación`
  - `Cima + 4 = R0;  // Almacenamos el resultado final`
  - `Cima = Cima + 4;`

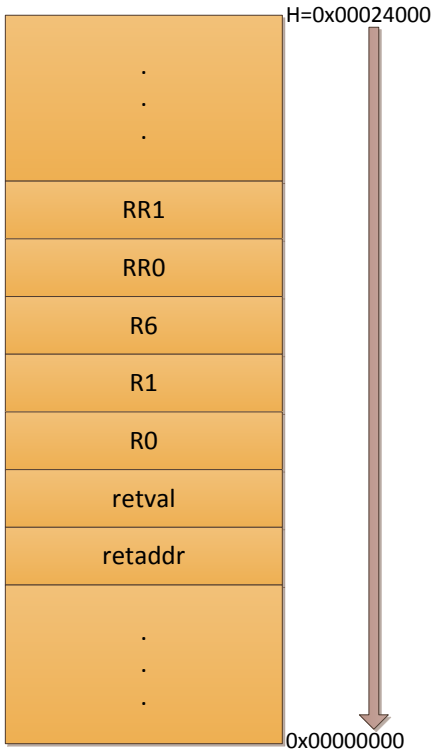
Invocación de funciones

Llamada a funciones y paso de parámetros

Al invocar una función, se debe realizar el paso de parámetros en pila y el salto a la posición de memoria donde comienza el código de la función. Para poder realizar la invocación, se realizan las siguientes operaciones:

- Salvado de registros. Se guardan los siguientes registros:
  - `R0`
  - `R1`
  - `R6`
  - `RR0`
  - `RR1`
- Reserva de espacio para el valor de retorno (`retval`)
- Reserva de espacio para la dirección de retorno (`retaddr`) y almacenamiento de la dirección de retorno.
- Almacenamiento temporal de `R7` en `R3` (se guarda la dirección de la cima de la pila).
- Reserva de espacio y copia de parámetros a la pila.
- Almacenamiento de `R3` en `R6` (se asigna a `R6` la dirección de la cima de la pila, que es donde comienzan los parámetros de la función).

De esta forma, la pila queda así:



Para el caso de los procedimientos con sólo parámetros de entrada, el procedimiento es similar, exceptuando el valor `retval`, ya que al ser un procedimiento no tiene ningún valor de retorno.

Retorno de funciones y devolución de valor

Al retornar de una función, es necesario almacenar el valor de retorno en la posición reservada para `retval`. Tras almacenar el valor de retorno, entonces se efectuará un salto a la dirección de retorno, para así volver al llamador. El llamador será entonces el encargado de tomar el valor de retorno y restaurar los registros.

Operaciones de salida

El código de implementación está localizado en `Qlib.c`. Se utiliza la etiqueta `put_f` para imprimir diferentes tipos de datos, y ésta es llamada en las tres operaciones que se mostraron anteriormente.

Los operandos que necesita para su correcto funcionamiento son: la etiqueta de retorno, la dirección de la variable que se imprimirá y el formato de impresión. Este paso de parámetros se realiza con los siguientes registros:

- `R0`: etiqueta de retorno.
- `R1`: valor a mostrar o dirección del dato si es una ristra.
- `R2`: tipo del dato a visualizar.

Estructura del programa generado

La estructura del programa Q generado por nuestro compilador de nADA es la que sigue:

```
BEGIN
      [<declaraciones de funciones>]
L 1:   <declaración de la función main>
      [<declaraciones de funciones>]
L 0:  [<carga de literales string en heap>]
     GT(1);
END
```

En este código no se realiza la declaración de datos estáticos, puesto que todas las variables se almacenan en la pila. Un tratamiento aparte reciben los literales string, que no se almacenarán en la pila, sino en el heap. Por ello, el programa Q debe comenzar con la carga de literales string en el heap (a partir de la etiqueta 0) y posteriormente debe saltar al procedimiento principal (etiqueta 1).

Por supuesto, en el caso de que no se usen literales string en el programa, entonces no habrá carga de literales string en el heap (es opcional), sino que sólo se producirá el salto a la etiqueta del procedimiento principal.

Dentro de la parte de declaración de variables del procedimiento principal puede haber declaraciones de otras funciones. Las haya o no, el compilador de nADA es capaz de reconocer dónde se define del procedimiento principal global y le asignará siempre la etiqueta 1. Asimismo, nuestro compilador utilizará las declaraciones para asignar una etiqueta mayor que 1 a cada función o procedimiento distinto del principal, y almacenará dicha etiqueta en la tabla de símbolos, de manera que dichas funciones puedan ser invocadas desde cualquier función que se defina, siempre y cuando ya estén definidas antes de ejecutarla.

Manual de uso del compilador

Compilación del compilador

Junto con esta memoria se proporciona el código fuente del compilador. Con el código se incluye un fichero compilar.sh que permite compilar el compilador y produce el ejecutable compilador. Este fichero, como su nombre indica, es el compilador.

Compilación de un programa en nADA con nuestro compilador

Para compilar un fichero fuente en nADA a código Q se puede ejecutar el compilador de dos formas diferentes. Puesto que Bison analiza el código fuente leyéndolo desde la entrada estándar, la ejecución del programa compilador sin parámetros producirá el compilador se quede esperando la introducción del código fuente por teclado, y dejará de esperar si se pulsa Ctrl-C, lo cual interrumpirá el programa y no se compilará, o si se pulsa Ctrl-D, lo cual introducirá el EOF y el compilador sí compilará el código.

Dado que esta forma de uso no es suficientemente útil e intuitiva, la solución más directa es redirigir la entrada de un fichero a la entrada estándar del compilador. Por ejemplo, si se desease compilar el fichero fuente `ejemplo.ada`, se debería ejecutar en la consola la orden

```
./compilador < ejemplo.ada
```

obtendremos, en un fichero llamado `código_generado.txt`, el código Q listo para ser ejecutado por el intérprete de Q.

Conclusiones y opinión personal

Gracias a esta práctica, hemos conseguido comprender el comportamiento interno de un compilador, y hemos visto los grandes problemas a los que se someten los programadores de este tipo de aplicaciones. Estos esfuerzos se han visto recompensados al ver, finalmente, nuestro compilador funcionando y ver el código generado siendo interpretado por la máquina Q.

Sin embargo, y debido a los nuevos ajustes en las fechas de entrega y del calendario académico en general, no hemos tenido tiempo para implementar todas las funcionalidades que nos gustaría a nuestro compilador. Como la entrega final se encontraba en la última semana de clase en Diciembre, nos hemos visto con el tiempo más apretado para poder completar la implementación, y a toda prisa hemos podido terminar a tiempo.

A pesar de estos contratiempos, terminamos este trabajo con un buen sabor de boca, habiendo creado algo que al principio veíamos difícil, y proporcionándonos una gran dosis de conocimientos y de satisfacción personal.