IC5701: Tarea Programada Número 1

Entregar el 5 de Octubre 2018

tecDigital 12:pm

José Castro

Contents

En esta tarea usted debe implementar varios programas:

1. un analizador léxico que reconoce hileras entre comillas, números, identificadores, y signos de puntuación.
2. un programa ensamblador `vasm`, que traduce texto escrito en el ensamblador de una máquina virtual que llamaremos la máquina de vagol VAM. su programa debe ejecuar de la siguiente manera:

   ```
   > ./vasm prueba.vasm
   ```

   el resultado de la corrida debe ser ya sea un listado de errores, o un archivo `prueba.vam` que es el código binario de su archivo `prueba.vasm`.
3. un programa que implementa la VAM llamado `vam` que ejecuta y debuguea archivos de formato y extensión `.vam`, la ejecución debe poder efectuarse paso a paso viendo el código del programa y el resultado.
4. un programa llamado `meta` que permite el reconocimiento mediante expresiones *a la* Backus Naür código de lenguajes arbitrarios simples de alto nivel.
5. una segunda versión de meta `metaII` (pronunciado: METAL) que a partir de expresiones a la Backus Naür con anotaciones permite generar código de `vasm` .
6. un compilador escrito para el lenguaje VALGOL escrito en `metaII` que genere código ensamblador de `vasm`

A continuación con más detalle cada una de estas etapas:

Problema 1

(para el 5 de Octubre) Tanto el ensamblador, el meta, y el metaII requieren de un analizador léxico (tokenizer). Dichosamente todos requieren de un analizador léxico que procese la entrada y genere tokens del mismo tipo. Su primera tarea es entonces hacer una función tokenizer que reciba como entrada una hilera (string) de caracteres y retorne una secuencia/lista de tokens. Debe considerar tanto el tab como el cambio de linea como espacios en blanco, los espacios en blanco se ignoran excepto por el hecho que le sirven para separar los tokens. Los comentarios también se deben ignorar, empiezan con el caracter – (menos) y continúan hasta el fin de la linea. Los tokens que genera deben contener la linea y columna dentro del texto en que fueron reconocidos, esto le servirá más adelante para reportar errores, así como determinar si el token/identificador que está leyendo corresponde a una etiqueta o un mnemónico de una instrucción.
Los tokens que debe reconocer son de cuatro tipos:

- *Signos de puntuación*: eston son `();.,[]`
- *Strings*: secuencias de caracteres de cualquier tipo (incuyendo espacios) que se encuentren entre comillas dobles o simples y separadas de otros elementos por espacios.
- *Números*: de punto flotante o enteros, todos se almacenan como punto flotante
- *Identificadores*: cualquier secuencia de caracteres separada por espacios o signos de puntuación. que no empiece con un número.

Debe generar por lo menos 30 casos de prueba para verificar el funcionamiento de su tokenizador.

Problema 2

(para el 5 de Octubre) Implementar el ensamblador de la máquina de VAGOL 1.0:

Problema 2 continua en próxima página...

Table 1: Instrucciones de la máquina de VAGOL

Instrucción	Nombre	Explicacion
`load DIR`	Cargar 01 AB CD EF	Ponga los contenidos de la dirección `DIR` en la pila. Código de instrucción 0x01, 1 Byte para el código, 3 bytes para `DIR`
`loadl N`	Cargar Literal 02 01 23 45 67 89 AB CD EF	Ponga el valor `N` en la pila. Código de instrucción 0x02, 1 byte para el código, 8 bytes para `N`
`store DIR`	Guarde en `DIR` 03 AB CD EF	Saque el Top de la pila y guardelo en `DIR`. Código de instrucción 0x03, 1 byte para el código, 3 bytes para `DIR`
`add`	Sume 04	Remplace los dos elementos en el Top de la pila por su suma. Código 0x04, sin parámetros, instrucción de 1 byte.
`sub`	Reste 05	Remplace los dos elementos en el Top de la pila por su resta Código 0x05, sin parámetros, instrucción de 1 byte.
`mult`	Multiplique 06	Remplace los dos elementos en el Top de la pila por su multiplicación Código 0x06, sin parámetros, instrucción de 1 byte.
`equal`	¿Iguales? 07	Remplace los dos elementos en el Top de la pila por su comparación Código 0x07, sin parámetros, instrucción de 1 byte.
`jmp DIR`	Jump no condicional 08 AB CD EF	Brinque a la dirección `DIR` y ejecute desde ahí CÓDIGO 0x07, 3 bytes para la dirección
`jmpz DIR`	Brinque si 0 09 AB CD EF	Haga Pop el Top de la pila y si es igual a 0 brinque a `DIR` Código 0x09, 3 bytes para la dirección
`jmpnz DIR`	Brinque si no es 0 0A	Haga Pop al Top de la pila y brinque a `DIR` si es distinto de 0 Código 0x0A, 3 bytes para la dirección
`edit STR`	Edite `STR` 1N STR	N = ROUND(pop(pila)); ponga `STR` en la columna N del output. Código 0x1N, N es el largo de la hilera 1 byte para cada caracter Si `STR` no cabe (excede 80 columnas la impresión) entonces no efectúe la acción.
`print`	Imprima 0C	Mande lo que esta en el buffer de impresión al standard output, limpie el buffer de output. Código 0x0C
`halt`	Pare	Pare la ejecución del programa, Código 0x0D
`space N`	`N` espacios	Agregue `N` espacios al buffer de salida. Código 0x2N donde N corresponde a la cantidad de espacios a imprimir
`block N`	Bloque	Declara un bloque de `N` Palabras (enteros en nuestro caso), no requiere instrucción
`end`	Fin	Indica el final del texto del código ensamblador, no requiere instrucción

Problema 2 continua en próxima página...

La máquina abstracta de VAGOL 1.0, llamada VAM (VAGOL Abstract Machine) tiene una memoria, una pila, un program counter, y un conjunto de 12 instrucciones. El ensamblador de la VAM reconoce 14 instrucciones, de las cuales dos de ellas (`block` y `end`) no generan código, y sirven nada más para dar indicaciones al ensamblador. La máquina VAM, y todos los lenguajes vistos en esta tarea, solo reconocen números de punto flotante que toman 8 bytes en la memoria del VAM.
El formato del texto del código en ensamblador `vasm` de la máquina VAM es el siguiente:

- Los comentarios en el VALGOL empiezan con el caracter – (menos) y continuan hasta el fin de linea. Deben ser eliminados previo a la etapa de ensamblar (sugerencia: resuelva esto en el analizador léxico).
- El código del ensamblador se reconoce por lineas, cada linea puede ser una, y solo una, de tres opciones: (1) una etiqueta, (2) o una instrucción, (3) o una linea en blanco. Las etiquetas entonces se encontrarán en su propia linea y no estarán asociadas con ninguna instrucción, aunque una vez ensambladas, probablemente apunten a una dirección donde resida una instrucción. Una linea que tenga solo comentarios es para todos fines prácticos una linea en blanco.
- Una etiqueta es cualquier texto contiguo (sin espacios en blanco) escrito en la primera columna, siempre y cuando no empiece con el caracter – (menos) en cuyo caso la linea es un comentario. Una etiqueta puede, aunque no se recomienda, extenderse por toda la linea.
- Las instrucciones deben empezar en una columna distinta a la primera y siempre empiezan con el mnemónico de la instrucción seguida por los parámetros de la instrucción, si es que los tiene.

La máquina VAM está diseñada para poder compilar lenguajes de alto nivel simples, en particular en ejercicios siguientes compilaremos VAGOL 1.0, un ejemplo de código de VAGOL 1.0 es el siguiente (no lo debe implementar todavía, estamos haciendo la máquina virtual y su ensamblador):

```
begin
     real x
     0 -> x
     until x == 3 do
        begin
             edit(x*x*10+1,'*')
             print()
             x + 0.1 -> x
        end
end
```

El programa anterior, se puede compilar al siguiente código de `vasm`, al lado del código `vasm` se ilustra la dirección en hexadecimal de cada instrucción, y el código en hexadecimal que cada instrucción genera, exceptuando la representación en 8 bytes de los números de punto flotante.

Problema 2 continua en próxima página...

```
-- begin
      jmp A01              0x0000 - 07 00 00 0C
-- real x
x                          0x0004
      block 1              0x0004
-- 0 -> x
A01                        0x000C
      loadl 0              0x000C - 02 ????????  ????????
      store x              0x0015 - 03 00 00 04
-- until x == 3 do begin
A02                        0x0019
      load x               0x0019 - 01 00 00 04
      loadl 3              0x001D - 02 ????????  ????????
      equal                0x0026 - 07
      jmpz A03             0x0027 - 04 00 00 61
-- edit( x*x*10+1, '*')
      load x               0x002B - 01 00 00 04
      load x               0x002F - 01 00 00 04
      mult                 0x0033 - 06
      loadl 10             0x0034 - 02 ????????  ????????
      mult                 0x003D - 06
      loadl 1              0x003E - 02 ????????  ????????
      add                  0x0047 - 04
      edit '*'             0x0048 - 11 2A
-- print()
      print                0x004A - 0C
-- x + 0.1 -> x
      load x               0x004B - 01 00 00 04
      loadl 0.1            0x004F - 02 ????????  ????????
      add                  0x0058 - 04
      store x              0x0059 - 03 00 00 04
   -- end
      jmp A02              0x005D - 07 00 00 19
A03                        0x0061
-- end
      halt                 0x0061 - 0D
      space 1              0x0062 - 21
      end                  0x0063
--end
```

Su primer programa consiste en hacer el ensamblador de vasm, debe generar un archivo de código binario. Su ensamblador de vasm debe tomar el nombre de un archivo de la linea de comandos, digamos que `prueba.vasm` ya partir de éste generar el binario `prueba.vbin`

Problema 3

(para el 5 de Octubre) Su segunda tarea es implementar la máquina virtual de VAM, junto con un debugger de vasm. El debugger debe desplegar el estado de la memoria, así como cargar un archivo binario a memoria, y ejecutarlo. Debe desplegar el estado del program counter y de la pila, así como desplegar la salida del

buffer en una ventana independiente. Para correrlo debe leer desde la linea de comandos un archivo con extensión `.vbin`, por ejemplo `prueba.vam`, posicionar el program counter en la posición 0 de memoria, y ejecutar el programa. El programa debe tener las opciones de (S)tep, (Run) y rese(T) o reloa(D) como mínimo.

```
> vam prueba.vam
pc = 0
stack = []
program =
 0000 --> ...
 0004     ...
  .
  .
 Accion: (S)tep | (R)un | reloa(D)
 -- digite su comando:
```

Problema 4

(de aquí en adelante fecha y problemas por definir) Su siguiente ejercicio es implementar el metacompilador de meta (versión 0.0 de metaII). Escribir código de metaII es similar a escribir expresiones de Backus Naur. Aquí un ejemplo de código en meta:

```
Expr    =  Term ( '+' Term | null )
Term    =  Factor ( '*' Expr | null )
Factor  =  .id | '(' Expr ')'
```

Este sería un poco de código escrito en metaII para reconocer expresiones con suma, multiplicación, variables y paréntesis, tal como las siguientes:

```
x + y * (z + w)
uno + dos * tres * _CUATRO
```

Esto indica varias cosas:

- el metaII ya tiene una primitiva para reconocer identificadores, y se invoca utilizando la palabra id.
- de hecho el meta ya reconoce cuatro cosas básicas: identificadores, números (reales o enteros, ambos representados como núemros reales), e hileras.
- no se aprecia en los ejemplos, pero un identificador en metaII tiene las mismas reglas que en la mayoría de los lenguajes de programación, un caracter alfabéticoo o el *undesrcore*, seguido de cero o mas caracteres ya sea alfanuméricos o el *underscore*.
- otros tokens que se quieran incluir en el input deben mencionados explícitamente como hileras entre comillas, tal como se hace en este ejemplo con '+', '*', '(', y ')'.

El metaII es un metacompilador porque para reconocer un archivo le prove dos archivos en la linea de comandos

```
> ./metaII valgol.mtl prueba.val
... ok
```

En esta versión del metal sólo debe reconocer expresiones del lenguaje correctamente estructuradas. El primer parámetros