IC5701: Tarea Programada Número 2

Entregar el 5 de Octubre 2018

tecDigital 12:pm

José Castro

Contents

Problema 1 **3**

Problema 2 **3**

Problema 3 **6**

Problema 4 **7**

Problema 5 **9**

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

Analyzador Lexico (TOKENIZER). 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

ENSAMBLADOR. 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.

```
-- 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

Debugger y Máquina Virtual. 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

Análisis Sintáctico – Generalidades.
Su siguiente ejercicio es implementar la versión 0.0 de metaII. Para hacer este punto de la tarea su programa debe SOLAMENTE reconocer la gramática y desplegar el arbol sintáctico.
El metaII es un metacompilador, esto es: un compilador de compiladores. En metaII para reconocer la gramática escrita en un archivo de texto se necesita leer primero las reglas de la gramática que se encuentran en otro archivo. Escribir código de metaII es similar a escribir expresiones de Backus Naur.

Ejemplo.
Para empezar con un ejemplo, supongamos que se tiene un archivo de texto llamado `expresiones.mtl` que contiene:

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

Esta gramática esperamos que le sea facil de leer. Sin embargo vale la pena notar que tanto sintáctica como semánticamente es un poco distinta a las vistas en el libro de texto (¿Cuáles son las diferencias?). El propósito de estas diferencias es facilitarle al metaII la construcción de un parser por descenso recursivo, que como ustedes saben, es el más simple de todos.
¿Qué diferencias tiene con las gramáticas que se ven en el texto? Primero en el ejemplo se puede apreciar que se permite poner paréntesis en las reglas como parte de la regla gramatical, y no solo como parte de los tokens/terminales que la gramática debe reconocer (las reglas anteriores tienen ambos, paréntesis para expresar parte de la regla, y paréntesis para expresar lo que debe reconocer, los cuales se ponen entre comillas). Los paréntesis en la regla son utilizadas igual a como se usan en las expresiones regulares (las gramáticas definidas en el libro no las tienen). Su interpretación es simple de deducir: los paréntesis se usan para indicar una disyunción de casos, igual a como se interpretan en la expresiones regulares. Otra diferencia mínima se encuentra en utilizar = en vez de $\rightarrow$ para denotar la derivación de la regla gramatical, y `.null` en vez de ϵ para representar a la hilera nula. Todos los terminales en las expresiones en metaII se ponen dentro de comillas simples.
Es facil transformar la gramática anterior a otra equivalente que tenga el formato de las vistas en el libro sustituyendo la expresión en paréntesis por un no–terminal nuevo, y poniendo la expresión en paréntesis al lado derecho de la definición del nuevo no–terminal. Si hacemos esto la gramática equivalente a la anterior queda:

$$
\begin{array}{lcl}
\text{Expr} & \rightarrow & \text{Term Expr'} \\
\text{Expr'} & \rightarrow & \text{`+' Term} \mid \epsilon \\
\text{Term} & \rightarrow & \text{Factor Term'} \\
\text{Term'} & \rightarrow & \text{`*' Expr} \mid \epsilon \\
\text{Factor} & \rightarrow & \text{.id} \mid \text{`(' Expr `)'}
\end{array}
$$

Es ser ventajoso expresar las gramáticas en este sentido, ya que se requieren menos símbolos, y promueve que se factoricen a la izquierda las reglas. Sin embargo queda a su criterio como implementar las reglas, y es válido transformar la gramática y luego aplicar alguno de los algoritmos de parsing del libro de texto (más adelante explicamos como puede parsear este tipo de gramáticas directamente)
Dado el archivo `expresiones.mtl` visto mas arriva y un archivo (llamemosle `expresion.txt`) que contenga:

```
x + y * (z + w)
```

Su programa debería ser capaz de reconocer que se encuentra correctamente construido, y deplegar su arbol sintáctico. La forma en que el programa se llama es en la linea de comandos del shell de la siguiente manera:

```
> ./metaII expresiones.mtl expresion.txt
parsing:
x + y * (z + w)
... ok
Expr -+- Term --- Factor --- .id    --- x
      +- +
      +- Term -+- Factor --- .id    --- y
               +- *
               +- Expr   --- Factor -+- (
                                     +- Expr --- Term -+- Factor --- .id    --- z
                                     +- )              +- *
                                                       +- Expr   --- Factor --- .id --- w
```

(Este ejemplo despliega el arbol de la gramática original, y no la modificada para que sea igual a las del libro, de nuevo, queda a su criterio como la despliega).
Vale la pena mencionar varias cosas:

- el metaII que usted programe debe contar con una primitiva para reconocer identificadores, y en las expresiones de la gramática se invocan utilizando la palabra ".id"
- también el parser debe reconocer la palabra reservada `.num` para la cual el tokenizador debe leer un número.
- más generalmente, su implementación de meta debe reconcer cuatro cosas básicas: identificadores, números (reales o enteros, ambos representados como núemros reales), e hileras, y operadores y signos de puntuación.
- para lograr esto, modifique (si es que no lo hizo ya) el tokenizador que creo para el ensamblador del punto anterior, tal que los tokens que genera vengan clasificados en éstas cuatro categorías.
- utilice las reglas clásicas para reconocer un identificador: un identificador es un caracter alfabéticoo o el *undesrcore*, seguido de cero o mas caracteres ya sea alfanuméricos o el *underscore*.

Especificación de las gramáticas de metaII
Las gramáticas de metal tienen las siguientes reglas:

- Las reglas gramaticales en metaII se ponen de las mas específicas a la más general, tal que la regla principal de la gramática es la última.

- Cada no–terminal se define en una única regla gramatical, y cada regla gramatical define un único no terminal, hay una relación de uno a uno entre reglas y no terminales.
- Cada regla gramatical solo puede referirse a reglas antes definidas o a sí misma. Esta propiedad junto con las dos anteriores facilita la generación del parser.
- Las reglas gramaticales terminan con punto y coma `;`
- Los no–terminales empiezan con mayúscula (i.e. `Expr`, `Program`)
- Los terminales se encuentran entre comillas simples (i.e. '+', 'program' ...).
- Puede utilizar paréntesis para acomodar opciones, al igual que se hace en expresiones regulares.
- Los identificadores con significado especial empiezan con punto `.`, como por ejemplo `.id` o `.null` (los únicos hasta el momento, más de ellos más adelante: `.out`).
- El caracter `$` denota repetición de 0 o más veces, y permite simplificar las reglas y evitar recursión. Así la siguiente gramática en metaII:

```
Expr3  =  .id | '(' Expr1 ')';
Expr2  =  Expr3 $ ('*' Expr3) ;
Expr1  =  Expr2 $ ('+' Expr2);
```

  Es prácticamente (no completamente) equivalente a la gramática:

$$\begin{array}{lcl} \text{Expr1} & \rightarrow & \text{Expr2} \mid \text{Expr1 '+' Expr2} \\ \text{Expr2} & \rightarrow & \text{Expr3} \mid \text{Expr2 '*' Expr3} \\ \text{Expr3} & \rightarrow & \text{.id} \mid \text{`(' Expr `)'} \end{array}$$

Reconocimiento directo de las gramáticas de metaII Cada regla de una gramática de metaII se traduce directamente a una función recursiva utilizando el algoritmo de parsing por descenso recursivo en la figura 4.13 en la página 219 del libro de texto. El mapeo se puede hacer directamente porque la forma en que están acomodadas las reglas permite 1. Solo una regla por no–terminal, 2. procurar que las reglas ya estén factorizadas por la izquierda y sin recursión por la izquierda. La única salvedad (cambio al algoritmo) que ustedes deben hacer es la incorporación de los paréntesis y de las opciones (no están explícitamente en el algoritmo, pero bueno, algo tienen que hacer :))
El parser de metaII funciona por sustitución, esto significa que no construye el arbol de parsing, sino que cada vez que el compilador de metaII reconoce algo sencillamente lo elimina del input y continúa con el resto.

Problema 5

Generación de Código
En esta parte de la tarea es donde se empieza a combinar el ensamblador con el reconocimiento de las gramáticas. ya que vamos a agregar instrucciones a la gramática que nos permiten generar código del `vam`. Ahora bien, generar código para vam significa generar texto en el formato del ensamblador de vam, las reglas de la gramática entonces, deben generar texto. Esto se logra mediante la incorporación de los comandos `.out` y `.label` entre sus reglas, además de los parámetros especiales `*`, `*1`, y `*2`.
Al inorporar estos elementos en la gramática de metaII convertimos sus reglas en un esquema STL (Syntax Directed Translation).
Primero veamos el `.out`: el propósito del `.out` es generar instrucciones del ensamblador de la `vam`. Retomemos el ejemplo de la expresión, pero ahora le agregamos generación de código:

```
Expr3  =  .num .out('loadl ' *) | '(' Expr1 ')';
Expr2  =  Expr3 $ ('*' Expr3 .out('mult')) ;
Expr1  =  Expr2 $ ('+' Expr2 .out('add'));
```

La instrucción `.out` tiene la siguiente especificación:

- `.out` debe ser seguido de paréntesis y una lista de parámetros que son los valores que el metaII escribe en la salida estándar.
- cada invocación del `.out(...)` genera una nueva linea de texto en el archivo de salida.
- dentro de los paréntesis se encuentra la lista de parámetros del comando `.out`.
- estos parámetros pueden ser:
  - un asterísco *, que corresponde al valor del último token visto en el input
  - una hilera constante escrita entre comillas como `'mul'` o `'add'`
  - las constantes `*1` o `*2`
- los parámetros del `.out` se graban en orden en el output, pero no se empiezan a grabar desde la primera columna (si hiciera ésto, el output del `.out` automáticamente se convierte en una etiqueta)

Los parámetros especiales tienen la siguiente interpretación:

- el * representa el valor del último token visto en el input
- el *1 y *2 corresponden a etiquetas nuevas, creadas específicamente para esa regla de la gramática, las etiquetas se generan en orden con los nombres de `A01`, `A02`, ...

Por ejemplo: dada la gramática anterior, y parseando el archivo con el texto:

```
3.2 + 4.1 * 5
```

El parser debe generar el código

```
loadl 3.2
loadl 4.1
loadl 5
mult
add
```

La otra posible instrucción que se agrega en la gramática es `.label` la cual crea una etiqueta en la linea actual.
Debe probar su programa con la siguiente especificación:

Archivo `valgol.mtl`

```
Primary        =  .id .out('load ' *) | .num .out('loadl ' *)' | '(' Expr ')';
Term           =  Primary $ ('*' Primary .out('mult')) ;
Exp1           =  Term $ ('+' Term .out('add') | '-' Term .out('sub'));
Exp            =  Exp1 ( '=' Exp1 out('equal')) | .null);
AssignST       =  Exp '->' .id .out('store' *);
UntilST        =  'until' .label *1 Exp 'do' .out('jmpnz' *2) ST .out('jmp ' *1) .label *2;
ConditionalST  =  'if' Exp
                  'then' .out('jmpz' *1 ) ST
                  'else' .out('jmp' *2) .label *1 ST .label *2;
IOST           =  'edit' '(' Exp ',' .string .out('edit' *) ')' | 'print' .out('print');
IDSeq1         =  .id .label * .out('block 1');
IDSeq          =  IDSeq1 $ (',' IDSeq1);
Dec            =  'real' .out('jmp ' *1) IDSeq .label *1 ;
Block          =  'begin' (Dec ';' | .null) ST $ (';' ST) 'end' ;
ST             =  IOST | AssignST | UntilST | ConditionalST | Block;
Program        =  Block .out('halt') .out('SP 1') .out('end');
```

Archivo `ciclo.val`

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

Corriendo su programa final

```
> metaII1.0 vagol.mtl ciclo.val
    jmp A01
x
    block 1
A01
    loadll 0
    store x
A02
    load x
    loadl 3
    equal
    jmpz A03
    edit( x*x*10+1, '*')
    load x
    load x
    mult
    loadl 10
    mult
```

```
    loadl 1
    add
    edit '*'
    load x
    load 0.1
    add
    store x
    jmp A02
A03
    halt
    space 1
    end
```