Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información.
CI–4721 – Lenguajes de Programación II
Enero – Marzo 2025

Tarea II

(10 puntos)

A continuación encontrará 2 preguntas, cada una compuesta de diferentes sub–preguntas. El valor de cada pregunta (y sub–pregunta) estará expresado entre paréntesis al inicio de las mismas.

En aquellas preguntas donde se le pida ejecutar un algoritmo o procesar una entrada, incluya los pasos relevantes de la ejecución del mismo con los cuales usted pudo alcanzar su conclusión. Sea lo más detallado y preciso posible en sus razonamientos y procedimientos.

En aquellas preguntas donde se le pida implementar un programa, mantenga su código en un repositorio `git` remoto (preferiblemente `Github`) y coloque un enlace al mismo en lugar de su respuesta. Todo su código debe ser legible y estar debidamente documentado.

La entrega se realizará por correo electrónico a rmonascal@gmail.com hasta las 11:59pm. VET del Jueves 4 de Marzo de 2025.

1. (4 puntos) Sea una gramática $IFT = (\{n, +, \ if, \ then, \ else\}, \ \{E\}, \ P, \ E)$, con P definido de la siguiente manera:

$$\begin{array}{rcl} E & \rightarrow & if\ E\ then\ E\ else\ E \\ & | & if\ E\ then\ E \\ & | & E\ +\ E \\ & | & n \end{array}$$

   a) (1.5 puntos) Aumente el símbolo no–terminal E con un atributo *val*, que contenga el valor de evaluar la expresión en cuestión. Puede suponer que el lenguaje evalúa los booleanos como en `C`, donde 0 es $false$ y todo lo contrario es $true$. Puede suponer, además, que el símbolo n tiene un atributo intrínseco *val* que tiene el valor para ese terminal (un número entero). Puede agregar todos los atributos adicionales que desee a E, cuidando que la gramática resultante sea S–atribuida.

   b) (1.5 puntos) Tenemos a nuestra disposición un reconocedor descendente. La gramática anterior tiene prefijos comunes y recursión izquierda. Transforme la gramática de tal forma que sea apropiada para un reconocedor descendente. Recuerde agregar atributos y reglas de tal forma que aún se calcule el valor de la expresión en *val*, cuidando que la gramática resultante sea L–atribuida.

   c) (1 punto) Construya un reconocedor recursivo descendente a partir de su gramática. Esto es, escriba las funciones (en pseudocódigo) que reconozca frases en el lenguaje y calculen el atributo *val* para una expresión bien formada. Puede suponer que tiene una variable *lookahead* que contiene el siguiente símbolo de la entrada en todo momento.

2. (6 puntos) Se desea que modele e implemente, en el lenguaje de su elección, un generador de analizadores sintácticos para gramáticas de operadores:

   a) Debe saber manejar símbolos terminales (en minúscula o signos ascii, menos el marcador de borde $) y no terminales (en mayúscula) que se comprenden de un sólo caracter.

   b) Una vez iniciado el programa, pedirá repetidamente al usuario una acción para proceder. Tal acción puede ser:

   1) `RULE <no-terminal> [<simbolo>]`
      Define una nueva regla en la gramática para el símbolo `<no-terminal>`. La lista de símbolos en `[<simbolo>]` es una lista (potencialmente vacía), separada por espacios, de símbolos terminales o no terminales.

      Por ejemplo:
      - `RULE A a A b` — Representa a la regla: $A \to a\ A\ b$
      - `RULE B` — Representa a la regla: $B \to \lambda$

      El programa debe reportar un error e ignorar la acción si el símbolo que se coloca del lado izquierdo de la regla no es no–terminal o si la regla expresada no corresponde a una gramática de operadores.
   2) `INIT <no-terminal>`
      Establece que el símbolo inicial de la gramática es el símbolo en `<no-terminal>`.

      Por ejemplo: `INIT B` — Establece el símbolo B como símbolo inicial de la gramática.

      El programa debe reportar un error e ignorar la acción si el símbolo no es no–terminal.
   3) `PREC <terminal> <op> <terminal>`
      Establece la relación entre dos terminales (o $). Esta operación `<op>` puede ser:
      - `<` cuando el primer terminal tiene menor precedencia que el segundo
      - `>` cuando el primer terminal tiene mayor precedencia que el segundo
      - `=` cuando el primer terminal tiene igual precedencia que el segundo

      Por ejemplo:
      - `PREC + < *` — Establece que `+` tiene menor precedencia que `*`
      - `PREC ( = )` — Establece que `(` tiene igual precedencia que `)`
      - `PREC $ > n` — Establece que `$` (marcador de borde) tiene mayor precedencia que `n`

      El programa debe reportar un error e ignorar la acción si los símbolos involucrados no son símbolos terminales o si el operador en `<op>` es inválido.
   4) `BUILD`
      Construye en analizador sintáctico con la información suministrada hasta el momento.

      Debe reportar los valores calculados para las funciones f y g (vistas en clase) o reportar que construir dichas funciones es imposible, mostrando evidencias para ello.
   5) `PARSE <string>`
      Realiza el proceso de análisis sintáctico sobre la cadena suministrada en `<string>`. Debe mostrar cada uno de los pasos, incluyendo:
      - **Pila** — Estado actual de la pila
      - **Entrada** — Estado actual de la entrada. Este estado debe mostrar claramente las relaciones de precedencias y el punto donde se está leyendo actualmente (ver ejemplo).
      - **Accion** — Acción tomada (leer o reducir por una regla particular)

      Por ejemplo: `PARSE n + n * n` — Realizará el proceso sobre la cadena `n + n * n`

      El programa debe reportar un error e ignorar la acción si los símbolos involucrados no son símbolos terminales, hay símbolos no–comparables o si no ha hecho `BUILD` previamente. Entre cada para de símbolo terminales en `<string>` puede haber una cantidad cualquiera (potencialmente cero) de espacios en blanco.

6) `EXIT`
   Debe salir del simulador.

A continuación se muestra un ejemplo de corrida para la gramática de expresiones aritméticas:

```
$> RULE E E + E
Regla "E -> E * E" agregada a la gramática
$> RULE E E + E
Regla "E -> E * E" agregada a la gramática
$> RULE E E E
ERROR: Regla "E -> E E" no corresponde a una gramática de operadores
$> RULE E n
Regla "E -> n" agregada a la gramática
$> INIT e
ERROR: "e" no es un símbolo no-terminal
$> INIT E
"E" es ahora el símbolo inicial de la gramática
$> PREC n > +
"n" tiene mayor precedencia que "+"
$> PREC n > *
"n" tiene mayor precedencia que "*"
$> PREC n > $
"n" tiene mayor precedencia que "$"
$> PREC + < n
"+" tiene menor precedencia que "n"
$> PREC + > +
"+" tiene mayor precedencia que "+"
$> PREC + < *
"+" tiene menor precedencia que "*"
$> PREC + > $
"+" tiene mayor precedencia que "$"
$> PREC * < n
"*" tiene menor precedencia que "n"
$> PREC * > +
"*" tiene mayor precedencia que "+"
$> PREC * > *
"*" tiene mayor precedencia que "*"
$> PREC * > $
"*" tiene mayor precedencia que "$"
$> PREC $ < n
"$" tiene menor precedencia que "n"
$> PREC $ < +
"$" tiene menor precedencia que "+"
$> PREC $ < *
"$" tiene menor precedencia que "*"
$> PARSE n + n * n
ERROR: Aún no se ha construido el analizador sintáctico
$> BUILD
Analizador sintáctico construido.
Valores para f:
  n: 4
  +: 2
  *: 4
  $: 0
Valores para g:
  n: 5
  +: 1
  *: 3
  $: 0
```

```
$> PARSE n + n * n
Pila           Entrada                                    Accion
                           $    <    n > + < n > * < n > $     leer
n                    $ < n    >    + < n > * < n > $         reducir E -> n
E                          $    <    + n > * < n > $         leer
E +                  $ < +    <    n > * < n > $           leer
E + n           $ < + < n    >    * < n > $               reducir E -> n
E + E                $ < +    <    * < n > $               leer
E + E *         $ < + < *    <    n > $                   leer
E + E * n  $ < + < * < n    >    $                       reducir E -> n
E + E * E       $ < + < *    >    $                       reducir E -> E * E
E + E                $ < +    >    $                       reducir E -> E + E
E                          $         $                       aceptar

$> PARSE n + * n
Pila           Entrada                                    Accion
                           $    <    n > + < * < n > $     leer
n                    $ < n    >    + < * < n > $         reducir E -> n
E                          $    <    + < * < n > $         leer
E +                  $ < +    <    * < n > $             leer
E + *           $ < + < *    <    n > $                 leer
E + * n    $ < + < * < n    >    $                       reducir E -> n
E + * E         $ < + < *    >    $                       rechazar, no se puede reducir por E -> E * E

$> PARSE n
Pila           Entrada                Accion
                    $    <    n > $     leer
n              $ < n    >    $         reducir E -> n
E                   $         $         aceptar

$> PARSE a + b * c
ERROR: Los siguientes símbolos no son terminales de la gramática: "a", "b", "c"

$> PARSE n n
ERROR: "n" no es comparable con "n"

$> PARSE
ERROR: "$" no es comparable con "$"
```

Investigue herramientas para pruebas unitarias y cobertura en su lenguaje escogido y agregue pruebas a su programa que permitan corroborar su correcto funcionamiento. Como regla general, su programa debería tener una cobertura (de líneas de código y de bifuración) mayor al 80 %.