Instituto Tecnológico de Costa Rica
Escuela de Ingeniería en Computación
IC-5701 Compiladores e intérpretes

Proyecto #3, Las fases de síntesis: generación e interpretación de código

Historial de revisiones:

- 2020.07.23 a 2020.07.30: Comunicaciones orales retos, con ejemplos en clase.
- 2020.08.03: Versión v0. Color azul para problemas que dan puntos extra.

Lea con cuidado este documento. Si encuentra errores en el planteamiento[1], por favor comuníqueselos inmediatamente al profesor.

Objetivo

Al concluir este tercer proyecto, Ud. habrá terminado de comprender los detalles relativos a las fases de síntesis de un procesador del lenguaje Δ extendido (generación de código para la máquina virtual TAM e interpretación de ese código). Como en los proyectos anteriores, el compilador y el intérprete aplican los principios, los métodos y las técnicas expuestos por Watt y Brown en su libro *Programming Language Processors in Java*. Ud. deberá extender el par (compilador, intérprete) de (Δ, TAM) desarrollado por Watt y Brown, de manera que sea capaz de procesar el lenguaje Δ extendido, según se describe en la sección *Lenguaje fuente* de la especificación de los proyectos #1 y #2, y conforme con las indicaciones que se hacen abajo. Además, su compilador deberá coexistir con un ambiente de edición, compilación y ejecución ("IDE"). Se le suministra un IDE construido en Java por el Dr. Luis Leopoldo Pérez, ajustado por estudiantes de Ingeniería en Computación del TEC.

En negro se plantean los temas que deberá cubrir en este proyecto ***obligatoriamente***.

En azul se plantean los temas que dan puntos extra.

Base

La base es la misma dada para los proyectos #1 y #2. Debe estudiar y comprender los capítulos y apéndices del libro *Programming Language Processors in Java* correspondientes a *organización en tiempo de ejecución*, *generación de código*, *interpretación* y *descripción de TAM* (6, 7, 8, B, C, D). Deberá estudiar y entender la estructura y las técnicas empleadas en la construcción del generador de código que traduce programas de Triángulo extendido (`.tri`) hacia código de la Máquina abstracta TAM; estos componentes están en la carpeta 'CodeGenerator'. También deberá estudiar el intérprete de TAM, cuyos componentes están en la carpeta 'TAM'.

Si su proyecto ***anterior*** fue deficiente, *puede utilizar el programa desarrollado (para el Proyecto #1 o Proyecto #2) por otros compañeros como base para este Proyecto #3,* ***pero deberá pedir la autorización de reutilización a sus compañeros y darles créditos explícitamente*** en la documentación del proyecto que presenta el equipo del cual Ud. es miembro[2].

Entradas

Los programas de entrada serán suministrados en archivos de texto. Los archivos fuente deben tener la terminación `.tri`. Si su equipo 'domestica' un IDE, como el suministrado por el profesor o alguno alterno, puede usarlo en este proyecto. En ese caso, el usuario seleccionará el archivo que contiene el texto del programa fuente desde el ambiente de programación (IDE), o bien lo editará en la ventana que el IDE provea para el efecto (que también debe permitir guardarlo de manera persistente).

[1] El profesor es un ser humano, falible como cualquiera.
[2] Asegúrese de comprender bien la representación que sus compañeros hicieron para los árboles de sintaxis abstracta, la forma en que estos deben ser recorridos, y las implicaciones que tienen las decisiones de diseño tomadas por ellos, pero en el contexto de los requerimientos de este Proyecto #3.

Lenguaje fuente

Sintaxis

Remítase a la especificación del Proyecto #1. ***Asegúrese de presentar los árboles de sintaxis abstracta en la ventana correspondiente***, en caso de que no lo hubiera hecho en los proyectos anteriores.

Contexto: identificación y tipos

No hay cambios.

Es importante que entienda bien cómo funciona la variable de control en el comando de repetición controlada por contador, **`repeat var`** *Id* **`in`** Exp_1 **`to`** Exp_2 **`do`** *Com* **`end`**, para que le dé la semántica deseada (alcance léxico, efecto en la estructura de bloques, variable protegida para que no pueda ser modificada vía asignación o paso de parámetros por referencia). Asegúrese de que el analizador contextual haya dejado el árbol de sintaxis abstracta (AST) decorado apropiadamente, esto es, que haya introducido información de tipos en los ASTs correspondientes a expresiones, así como dejar "amarradas" las ocurrencias *aplicadas* de identificadores vía la tabla de identificación (las ocurrencias aplicadas deben tener referencias hacia el sub-AST donde aparece la ocurrencia de *definición* correspondiente[3], es decir, la declaración de la variable *Id* asociada a un valor inicial dado por la expresión Exp_1). Asimismo, deberá determinar si un identificador corresponde a una variable[4], etc. Refiérase a la especificación del proyecto #2 y repase el capítulo 5 del libro.

Semántica

Esta es la parte que influye en la generación e interpretación de código.

Solo se describe la semántica de las frases de Δ que han pasado el análisis contextual. No se debe generar código para programas con errores contextuales, sintácticos o léxicos.

El comando nulo **`nil`** no tiene efectos en la ejecución: no se afectan la memoria, ni la entrada ni la salida. Tenga cuidado con el esquema de generación de código para que de veras el comportamiento sea "nulo"[5].

En el comando **`if`** *Exp* **`then`** Com_1 (**`elsif`** Exp_i **`then`** Com_i)* **`else`** Com_2 **`end`** se procede como en el lenguaje Δ original[6].

Los comandos repetitivos pueden ser ejecutados cero, una o más veces, según se indica a continuación.

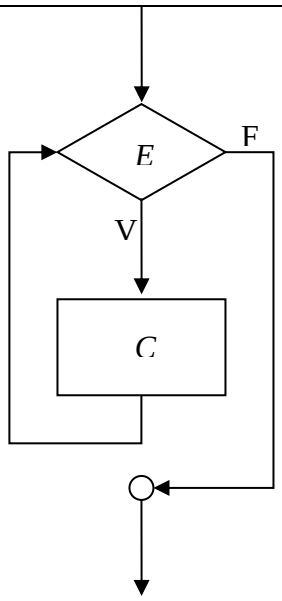
`repeat while` *E* **`do`** *C* **`end`**	**`repeat until`** *E* **`do`** *C* **`end`**
(*Repetición con entrada condicionada positiva*) Se evalúa la expresión *E*. Si ésta es verdadera, se procede a ejecutar el comando *C*; luego se procede a re-evaluar *E* y determinar si se repite *C* o no. Si *E* evalúa a falso, se termina la repetición. Note que *C* podría ejecutarse 0 veces (si *E* evalúa a falso al inicio).	(*Repetición con entrada condicionada negativa*) Se evalúa la expresión *E*. Si ésta es falsa, se procede a ejecutar el comando *C*; luego se procede a re-evaluar *E* y determinar si se repite *C* o no. Si *E* evalúa a verdadero, se termina la repetición. Note que *C* podría ejecutarse 0 veces (si *E* evalúa a verdadero al inicio).

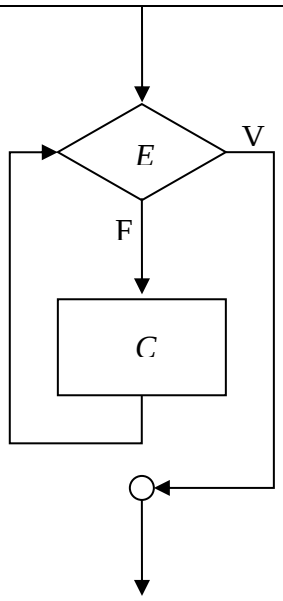
[3] En las filminas esas referencias aparecen dibujadas como flechas rojas. Nos hemos referido a ellas en clases como 'punteros rojos'.

[4] Como discutimos ampliamente en clases, la variable de control de un **`repeat_var_in_to_do_end`** se comporta como una *constante* (no se la puede asignar, no se la puede pasar por referencia), pero es preferible usar un constructor que corresponda a esta situación: *no usar el constructor de una constante*.

[5] Pista: el comando nulo ya existía en el compilador original. Estudie eso.

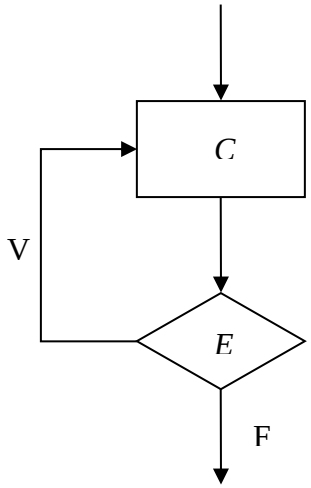
[6] Su analizador sintáctico ya debe haber dejado los ASTs con la forma apropiada, de manera que esto sea un no-problema para el analizador contextual y para el generador de código.





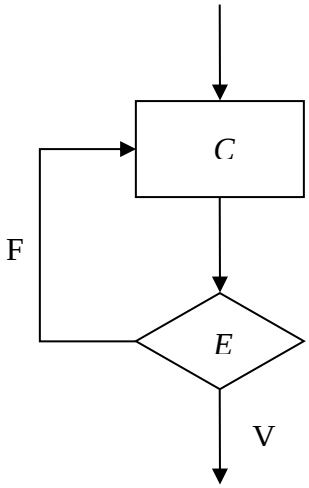
`repeat do` *C* `while` *E* `end`

(*Repetición con entrada garantizada y salida negativa*) Se ejecuta el comando *C*. Se evalúa la expresión *E*. Si *E* evalúa a verdadera, se procede a ejecutar el comando *C* de nuevo. Si *E* evalúa a falsa, se termina la repetición. Note que *C* se ejecuta al menos una vez.



`repeat do` *C* `until` *E* `end`

(*Repetición con entrada garantizada y salida positiva*) Se ejecuta el comando *C*. Se evalúa la expresión *E*. Si *E* evalúa a falsa, se procede a ejecutar el comando *C* de nuevo. Si *E* evalúa a verdadera, se termina la repetición. Note que *C* se ejecuta al menos una vez.



El comando

repeat var *Id* **in** *Exp₁* **to** *Exp₂* **do** *Com* **end**

declara una variable *Id* que es ***local*** a la repetición[7] y cumple con las restricciones indicadas en el Proyecto #2. Tal comando se ***comporta*** como el código que sigue[8]:

```
let const $Final ~ Exp2 ; !el valor final se evalúa solo una vez
    var   Id := Exp1 !Id obtiene como primer valor el que tiene Exp1
in repeat while Id <= $Final do !mientras no se haya excedido el límite superior
     let
       const Id ~ Id !se re-declara Id como constante para usarlo en Com
                     !esto protege a Id dentro de Com, dada la estructura de bloques de Δ
     in Com
     end ; !el let interno abarca únicamente Com, que llega hasta aquí
     Id := Id + 1 !se incrementa la variable de control, Id, declarada en el primer let
     !continuar con las repeticiones
   end
end
```

En particular observe que:

- La expresión *Exp₂* se evalúa *una sola vez*, al inicio de la repetición (por eso se asocia el valor a una constante, `$Final`), y debe ser entera – como lo garantiza el analizador contextual. Si el valor se guarda en memoria, ese espacio debe ser liberado al terminar la repetición.
- La expresión *Exp₁* se evalúa *una sola vez*, al inicio de la repetición, debe ser entera y ése es el valor *inicial* de la variable de control `Id`.
- La variable de control (`Id`) es *declarada* por el comando **`repeat_var_in_to_end`**; cuando la repetición termina, esta variable debe desaparecer de la memoria[9].
- La variable de control *debe 'funcionar' como una constante* en el cuerpo del **`repeat_var_in_to_end`** (`Com`): en el comando `Com` no se le pueden asignar valores; estas restricciones se suponen aseguradas por el analizador contextual, al 'instalar' adecuadamente una asociación en el ambiente. El generador de código debe emitir las instrucciones necesarias para *actualizar* el valor de la variable de control (`Id`)[10].

Considere la declaración de variable inicializada:

var *Id* **:=** *Exp*

En esta declaración se evalúa la expresión, cuyo valor resultante queda en la *cima* de la pila de TAM[11] y da valor inicial a la variable. La dirección donde inicia ese valor debe asociarse al identificador (*Id*) de la variable; recuerde que las direcciones se forman mediante desplazamientos relativos al contenido de un registro que sirve como *base*. *Exp* puede ser de cualquier tipo; el almacenamiento requerido para guardar la variable *Id* dependerá del tamaño de los datos del *tipo* inferido para *Exp* (lo cual ya fue hecho por el analizador contextual).

---

[7] La 'variable de control' se crea para cada activación del **`repeat_var_in_to_end`**. El espacio que esta ocupe debe ser liberado al terminar la ejecución de dicho comando iterativo, así como cualquier otro espacio (en memoria) requerido para ejecutar ese comando.

[8] Ese código es para *explicar* el comportamiento del **`repeat_var_in_to_end`**. Que se comporte así *no implica* que Ud. deba transliterar esta descripción y hacer una generación de código rudimentaria basada en ella. La pseudo-constante `$Final` ha sido introducida únicamente para *explicar* el comportamiento de este comando repetitivo. Tal constante no es parte del código del programa fuente. Haga una interpretación clara y eficiente de la semántica deseada para este comando.

[9] Es decir, Id sale de la pila (se desaloja de memoria).

[10] En la explicación, la *constante* `Id` en el **`let`** interno toma el valor actual de la *variable* `Id` del **`let`** externo. La declaración de `Id` en el **`let`** interno asegura que `Com` no puede usar a `Id` como variable. *Después* de ejecutar el **`let`** interno se actualiza la variable de control. Pero esto solo *explica* el funcionamiento de este comando repetitivo; una vez comprendida la semántica del comando, Ud. debe desarrollar un buen esquema de generación de código para el comando – en clases ofrecimos una plantilla que puede ayudarle a plantear su propio esquema de generación de código para el comando **`repeat_var_in_to_end`**. Recuerde que el analizador contextual debió asegurar que `Id` no puede ser modificada vía asignaciones por *Com* ni puede ser pasada por referencia.

[11] Debe quedar inmediatamente debajo de la celda de memoria apuntada por el registro `ST`, cuando este haya sido actualizado.

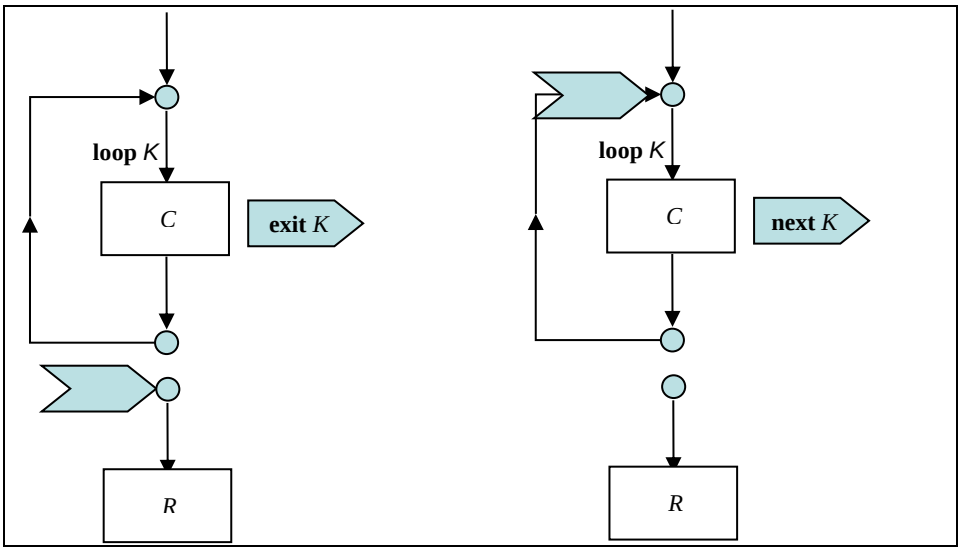
Debe asignarse espacio y direcciones aparte para *cada una* de las entidades declaradas dentro de las declaraciones compuestas **`private`** y **`rec`**[12]. Las entidades declaradas en un **`rec`** deben conocer las direcciones de todas las demás entidades introducidas en la misma declaración[13]. Recuerde que **`private`** exporta entidades, pero las entidades exportadas deben tener acceso a lo declarado como *local* ('privado'). Desde la perspectiva de generación de código, el procesamiento de las declaraciones compuestas **`private`** y **`rec`** no ofrece otras complicaciones adicionales a las que presenta el procesamiento de las declaraciones secuenciales. Suponemos que el analizador contextual dejó correctamente establecidas las referencias dentro del AST decorado (los "punteros rojos").

El resto de las características deberán ser procesadas como para el lenguaje Δ original.

Extras

En el comando de repetición indefinida **`repeat loop`** *Id* **`do`** *Com* **`end`**, *Com* se ejecuta repetidamente, salvo que el control alcance uno de los comandos de "escape" (**`exit`** o **`next`**):

- Si es un comando **`exit`** *Id*, se transfiere el control al comando que sigue al **`repeat-loop-do-end`**[14] cuyo nombre es *Id*.
- Si es un comando **`next`** *Id*, se transfiere el control al inicio del **`repeat-loop-do-end`**[15] cuyo nombre es *Id* (vuelve a ejecutar a partir de *Com*).



Cuando el comando **`repeat-loop-do-end`** no está etiquetado (sin *Id*), los comandos de escape son anónimos:

- Si es un comando **`exit`**, se transfiere el control al comando que sigue al **`repeat-loop-do-end`** más cercano ("desde adentro hacia afuera").
- Si es un comando **`next`**, se transfiere el control al inicio del **`repeat-loop-do-end`** más cercano ("desde adentro hacia afuera").

---

[12] Recuerde que las entidades que pueden declararse mediante **`rec`** son procedimientos y funciones, por lo que el espacio y las direcciones son en memoria de *código*.

[13] En las declaraciones **`rec`**, las direcciones son de código, pues lo que se declara como mutuamente recursivo son procedimientos y funciones. El procesamiento de **`rec`** requiere *dos* pasadas sobre el árbol de esa declaración compuesta, como se sugirió en clases, para poder resolver las referencias 'hacia adelante' mediante un proceso de "parchado" en que puede aprovecharse el patrón 'publicador-suscriptores'. Por lo demás, desde la perspectiva de la generación de código, el procesamiento de las declaraciones compuestas **`rec`** se asemeja al procesamiento de las declaraciones secuenciales de procedimientos y funciones.

[14] Las restricciones contextuales aseguran que un comando **`exit`** *Id* solo puede ser ejecutado en un contexto correcto.

[15] Las restricciones contextuales aseguran que un comando **`next`** *Id* solo puede ser ejecutado en un contexto correcto.

El escape de un comando **repeat-loop-end** es delicado, puesto que el **exit** puede aparecer dentro de un bloque **let** que incluye un **proc**edimiento, *salvo que su analizador contextual lo prohíba explícitamente*. Considere este ejemplo:

```
repeat K loop
  let proc P (…) ~
      begin
        …
        if … then
          exit K
        else
          …;
        …
      end
  in begin
    …
    P ();
    …
  end;
R
```

Si **exit** K se activa desde dentro de P, *no solamente* se trata de transferir el control a la dirección del resto del programa (*R* en este caso), *sino que también debe limpiarse la pila*: quitar el marco correspondiente a la activación de P antes de hacer esa transferencia de control. Como esto puede darse en un nivel arbitrariamente anidado (pero siempre dentro del alcance de *K*), entonces su plantilla de generación de código debe contemplar lo que debe hacerse en términos de la diferencia de niveles de léxico, para asegurar que la pila queda en un estado consistente (en lo concerniente a su tamaño).

El comando **return** provoca que el *procedimiento* cuyo cuerpo se está ejecutando termine y devuelva el control a su invocador, dejando el almacenamiento en un estado consistente[16]. Esto es, la pila debe quedar libre del marco correspondiente al procedimiento que termina y el registro ST debe apuntar a la posición de memoria a la que apuntaba antes de colocar los argumentos correspondientes a la invocación (CALL) de este procedimiento[17]. Debe tener cuidado si el **return** aparece dentro de un bloque **let**, para liberar el espacio ocupado por las constantes y variables declaradas dentro del bloque **let**.

Modificaciones a TAM

Este proyecto no requiere modificar a TAM, salvo que Ud. invente una solución a algún sub-problema mediante alguna extensión de TAM.

Proceso y salidas

Ud. modificará el procesador original de Δ y el intérprete de TAM, ambos en Java, para que sean capaces de procesar las extensiones especificadas arriba.

- Las técnicas por utilizar son las expuestas en clase y en el libro de Watt y Brown.
- Debe documentar *todas* las plantillas de generación de código correspondientes a las extensiones de Δ implementadas por su procesador.
- El algoritmo de generación de código debe corresponder a sus plantillas de generación de código.

---

[16] El cuerpo de un **proc**edimiento es un *comando*. Un **proc** es una abstracción sobre un comando. Por tanto, cuando termine de ejecutar, porque alcanzó su final o porque este fue forzado vía un **return**, la pila debe quedar del tamaño que tenía *antes de la invocación*. Se debe quitar el marco (*frame*) y también el espacio de los argumentos que están 'sobre' la dirección apuntada por el registro LB; además es necesario actualizar a LB y a ST. Estas cosas ya las logra la instrucción de RETURN de TAM. Es buena idea que un **return** transfiera el control donde está ese RETURN, pero cuidado si está en una situación como la ilustrada en el ejemplo. Δ no es C, al fin y al cabo.

[17] Recuerde que la ejecución de un comando no afecta el tamaño de la pila, aunque sí puede modificar su contenido. Un procedimiento es una *abstracción* sobre un comando. Una invocación a un procedimiento es una instanciación de esa abstracción y es, en sí misma, un comando. Consecuentemente, hay que dejar la pila con el tamaño que tenía antes de la invocación.

- Su programación debe ser consistente con el estilo aplicado en los programas en Java usados como base (compilador de Δ, intérprete de TAM), y ser respetuosa de ese estilo. En el código fuente debe estar claro dónde ha introducido Ud. sus modificaciones.
- Las salidas de la ejecución del programa Triangle, así como las entradas al programa, deben aparecer en la pestaña 'Console' del IDE.
- Los árboles de sintaxis abstracta deben desplegarse en la pestaña 'Abstract Syntax Trees' del IDE. Esto se logra extendiendo el trabajo que ya hace el IDE sobre el lenguaje Triangle original: visitando los ASTs y construyendo sub-ASTs apropiados que se presentan gráficamente en la pestaña.
- El código generado debe ser escrito en un archivo que tiene el mismo nombre del programa fuente, pero con extensión `.tam`. El archivo `.tam` debe quedar en la misma carpeta donde está el código fuente.
- El código generado debe aparecer en la pestaña 'TAM Code' del IDE. Esto se logra extendiendo el trabajo que ya hace el IDE al desensamblar el código TAM. Considere los cambios realizados a TAM, para que la salida sea significativa.
- Las entidades declaradas deberán aparecer en la pestaña 'Table Details' del IDE. En particular, nos interesa que el generador de código añada información apropiada en el árbol de sintaxis abstracta para poder reportar claramente las entidades declaradas (ya que esto se obtiene al recorrer el AST (decorado) para obtener los descriptores de entidades en tiempo de ejecución, y *no* de una tabla de identificación).
- Si un programa terminase anormalmente (aborta), en la consola debe indicarse claramente la razón. Escríbala en inglés, para ser consistente con los demás mensajes de error.
- El código TAM solo puede ser ejecutable cuando la compilación del programa fuente en Δ extendido está libre de errores léxicos, sintácticos y contextuales.
- **Debe ser posible activar la ejecución del IDE de su compilador desde el Explorador de Windows haciendo clics sobre el ícono de su archivo `.jar`, o bien generar un `.exe` a partir de su `.jar`**. *Por favor indique claramente cuál es el archivo ejecutable del IDE, para que el profesor o su asistente puedan someter a pruebas su programa sin dificultades*.
- *Debe dar crédito por escrito a cualquier fuente de información o ayuda.*

Documentación

Debe documentar clara y concisamente los siguientes puntos[18]:

- Descripción del esquema de generación de código para *todas* las variantes de **`repeat`** … **`end`**.
- Descripción del esquema de generación de código para **`repeat var`** … **`end`**.
- Solución dada al procesamiento de declaraciones de variable inicializada (**`var`** *Id* **`:=`** *Exp*).
- Su solución al problema de introducir declaraciones privadas (**`private`**).
- Su solución al problema de introducir declaraciones de procedimientos o funciones mutuamente recursivos (**`rec`**).
- Sus soluciones a la generación de código para las construcciones señaladas como *extras.*
- Nuevas rutinas de generación o interpretación de código, así como cualquier modificación a las existentes.
- Extensión realizada a los procedimientos o métodos que permiten visualizar la tabla de nombres (aparecen en la pestaña 'Table Details' del IDE).
- Descripción de cualquier modificación hecha a TAM y a su intérprete.
- Lista de errores de generación de código o de ejecución detectados.
- Describir cualquier cambio que requirió hacer al analizador sintáctico o al contextual para efectos de generación de código (incluida la representación de ASTs).
- Describir cualquier modificación hecha al ambiente de programación para hacer más fácil de usar su compilador.
- Dar crédito a los autores de cualquier programa utilizado como base (esto incluye el IDE y el código de los proyectos #1 y #2).
- Pruebas realizadas por su equipo para validar el compilador.
- Discusión y análisis de los resultados observados. Conclusiones obtenidas a partir de esto.
- Descripción resumida de las tareas realizadas por cada miembro del grupo.

[18] *Nada* en la documentación es opcional. La documentación tiene un peso importante en la calificación del proyecto. Si no modifica algo que es requerido para este proyecto, indíquelo explícitamente.

- Breve reflexión sobre la experiencia de modificar fragmentos de un (compilador | intérprete | ambiente) escrito por terceras personas, así como de trabajar en grupo para los proyectos de este curso.
- Indicar cómo debe compilarse su programa.
- Indicar cómo debe ejecutarse su programa.
- Una carpeta que contenga:
  - Sub-carpetas donde se encuentre el texto fuente de sus programas. El texto fuente debe indicar con claridad los puntos en los cuales se han hecho modificaciones.
  - Sub-carpetas donde se encuentre el código objeto del compilador+intérprete, en formato directamente ejecutable desde el sistema operativo Windows[19]. Todo el contenido del archivo comprimido debe venir libre de infecciones. Debe incluir el IDE enlazado a su compilador+intérprete, de manera que desde él se pueda ejecutar sus procesadores de Δ y de TAM.
- Debe reunir su trabajo en un archivo comprimido en formato **`.zip`**. Esto debe incluir:
  - Documentación indicada arriba, con una portada donde aparezcan los nombres y números de carnet de los miembros del grupo. Los documentos descriptivos deben estar en formato `.pdf`.
  - Código fuente, organizado en carpetas.
  - Código objeto. Recuerde que el código objeto de su compilador (programa principal) debe estar en un formato directamente ejecutable en Windows.
  - Programas (`.tri`) de prueba que han preparado.

Entrega
Fecha límite: **lunes 2020.08.17**, antes de las 23:55. No se recibirán trabajos después de la fecha y la hora indicadas.

Los grupos pueden ser de *hasta* **3** personas.

Debe enviar por correo-e el ***enlace***[20] a un archivo comprimido almacenado en la nube con todos los elementos de su solución a estas direcciones: `itrejos@itcr.ac.cr` y `andres.mirandaarias@gmail.com` (Andrés Miranda Arias, nuestro asistente).

> El asunto (subject) debe ser:
> "`IC-5701 - Proyecto 3 -` " <carnet> [" + " <carnet> [" + " <carnet>]].
>
> Los carnets deben ir ordenados ascendentemente.

Si su mensaje no tiene el asunto en la forma correcta, su proyecto será castigado con −10 puntos; podría darse el caso de que su proyecto no sea revisado del todo (y sea calificado con 0) sin responsabilidad alguna del profesor o del asistente (caso de que su mensaje fuera obviado por no tener el asunto apropiado). Si su mensaje no es legible (por cualquier motivo), o contiene un virus, la nota será 0.

La redacción y la ortografía deben ser correctas. El profesor tiene *altas expectativas* respecto de la calidad de los trabajos escritos y de la programación producidos por estudiantes universitarios de tercer año de la carrera de Ingeniería en Computación del Tecnológico de Costa Rica. Los profesores esperamos que los estudiantes tomen en serio la comunicación profesional.

[19] Es decir, sin necesidad de compilar los programas fuente.
[20] Los sistemas de correo han estado rechazando el envío o la recepción de carpetas comprimidas con componentes ejecutables. Suban su carpeta comprimida (en formato **`.zip`**) a algún 'lugar' en la nube y envíen el hipervínculo al profesor y a su asistente mediante un mensaje de correo con el formato indicado.