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:
- 2022.10.31: Versión base (v1).

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 comprendido los detalles relativos a las fases de síntesis de un procesador del lenguaje Δ (Triángulo) extendido: generación de código para la máquina virtual TAM e interpretación de ese código. El lenguaje Triángulo extendido será denominado **Δxt** en el resto de este documento. 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 Δxt, según se describe en la sección *Lenguaje fuente* de las especificaciones de los proyectos #1 y #2, y conforme con las indicaciones que se dan abajo. 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***.

Algunos temas en azul son obligatorios para grupos de 4 personas. Darán puntos extra para los demás grupos. Se distinguen de los demás párrafos por tener una línea vertical a su izquierda.

Este documento **no especifica** lo indicado como puntos extra, marcados con fondo gris en las especificaciones de los Proyectos #1 y #2.

Bases

Además de la base dada para los proyectos #1 y #2, Ud. 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* (capítulos {6, 7, 8}, apéndices {B, C, D}). Debe estudiar y entender la estructura y las técnicas empleadas en la construcción del generador de código que traduce programas de Δ (`.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'. En clases, discutimos algunos de los retos que plantea este proyecto y posibles abordajes para resolverlos.

Para este tercer proyecto se supone que los analizadores léxicos, sintáctico y contextual funcionan bien. Vamos a generar código únicamente para programas que ***no*** tienen errores léxicos, sintácticos o contextuales.

Si su Proyecto #2 fue deficiente, *puede utilizar el programa desarrollado (para el Proyecto #2) por otr@s compañer@s como base para este Proyecto #3*, ***pero deberá pedir la autorización de reutilización a sus compañer@ y darles créditos explícitamente*** en la documentación del proyecto que presenta el equipo del cual Ud. es miembro[2]. La autorización de uso, dada por los autores, debe llegar al profesor, vía un mensaje a su cuenta itrejos@itcr.ac.cr o por mensaje directo a su cuenta en Telegram, ***antes*** de transcurrida *una semana* desde la publicación de la presente especificación de este Proyecto #3.

---

[1] El profesor es un ser humano, falible como cualquier otro ser humano.
[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.

Entradas
Los programas de entrada serán suministrados en archivos de texto. Los archivos fuente deben tener la extensión `.tri`. Si su equipo 'domestica' un IDE, como el suministrado por el profesor o algún IDE alterno, puede usarlo en este proyecto. En tal IDE, el usuario debe ser capaz de seleccionar el archivo que contiene el texto del programa fuente por procesar, editarlo, guardarlo de manera persistente, compilarlo y enviarlo a ejecución (una vez compilado); además, deberá tener 'pestañas' semejantes a las que ofrece el IDE suministrado por el profesor.

Lenguaje fuente

Sintaxis y léxico
Remítase a la especificación del Proyecto #1 ('`IC-5701 2022-2 Proyecto 1 v2.pdf`). En el IDE: ***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. Vea el documento '`IC-5701 2022-2 Proyecto 2 v2.pdf`'.

Es importante que comprenda cómo funciona la variable de control en el comando de repetición controlada por contador, **`loop for`** ***Id*** **`from`** ***Exp₁*** **`to`** ***Exp₂*** **`do`** ***Com₁*** **`end`** (con sus variantes correspondientes a la salida o permanencia condicionada por **`until`** o **`while`**), a fin de que darle la semántica deseada en tiempo de ejecución. Además, deben asegurarse las propiedades contextuales indicadas en el Proyecto #2 (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 con referencias hacia el sub-AST donde aparece la ocurrencia de *definición* correspondiente[3]; esto se logra vía la tabla de identificación, donde la variable de control *Id* tiene como atributo asociado una referencia al AST que contiene su declaración en relación con el valor inicial que será 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 Δxt que han pasado el análisis contextual. No se debe generar código para programas con errores contextuales, sintácticos o léxicos.

array

- Para los valores declarados con denotador de tipo **`array`** *IL* **`of`** *T*, el primer elemento tiene índice 0 y el último elemento tiene índice *IL*-1. Hay *IL* elementos en arreglos definidos de esta manera.

El compilador original no genera código que compruebe *índices dentro de rango* al acceder arreglos en tiempo de ejecución. En un acceso a arreglo, el valor que resulta de evaluar la expresión selectora (indexadora) debe estar dentro del rango de los índices; de lo contrario se deberá ***abortar*** la ejecución del programa con un mensaje de error apropiado: "`array out of bounds`"[5]. Note que hay tanto *variables* como *constantes* de tipo **`array`**.

[3] En las presentaciones esas referencias aparecen dibujadas como flechas rojas: van desde la ocurrencia aplicada hasta la ocurrencia de definición (un AST de declaración). En clases nos hemos referido a ellas como los "punteros rojos".
[4] Como discutimos ampliamente en clases, la variable de control de un **`loop_for_from_do_end`** se comporta como una *constante* (no se la puede asignar, no se la puede pasar por referencia), pero *debe* usarse un constructor *distinto* que corresponda a esta situación: ***no*** *usen el constructor de una constante ni el de una variable*.
[5] Esto puede lograrse ampliando el rango de uso de la instrucción `HALT`. Ver nota al pie más adelante.

nil

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 el comportamiento sea realmente "nulo"[6].

if

En el comando **`if`** *Exp* **`then`** Com_1 (**`|`** Exp_i **`then`** Com_i)* **`else`** Com_2 **`end`** se procede como en el lenguaje Δ original[7], según las transformaciones explicadas en clases (*RestOfIf*, etc.).

select

En el comando **`select`**, la expresión selectora se evalúa *una sola vez*. Llamaremos '*valor selector*' al valor que resulta de la evaluación de la expresión. El valor selector se usa para seleccionar aquella rama (**`when`**) que incluye una literal que coincide con él. Si se especificó la opción **`else`** y *ninguna* de las literales comprendidas en las ramas del **`select`** coincidió con el valor selector, entonces se selecciona la rama del **`else`**[8]. Si ***no*** se especificó una rama **`else`** y *ninguna* de las literales coincide con el valor selector, entonces la ejecución debe ***abortar*** con un mensaje de error apropiado[9], como "`Unmatched expression value in select command`". Debe procederse de "izquierda a derecha" en las comparaciones del valor selector con las literales, y mantener el valor selector accesible en la pila mientras se esté procesando este comando[10]; al concluir la ejecución del comando **`select`**, el valor selector debe ser desalojado de la pila.)

loops condicionados

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

`loop while` *E* **`do`** *C* **`end`**	**`loop 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).

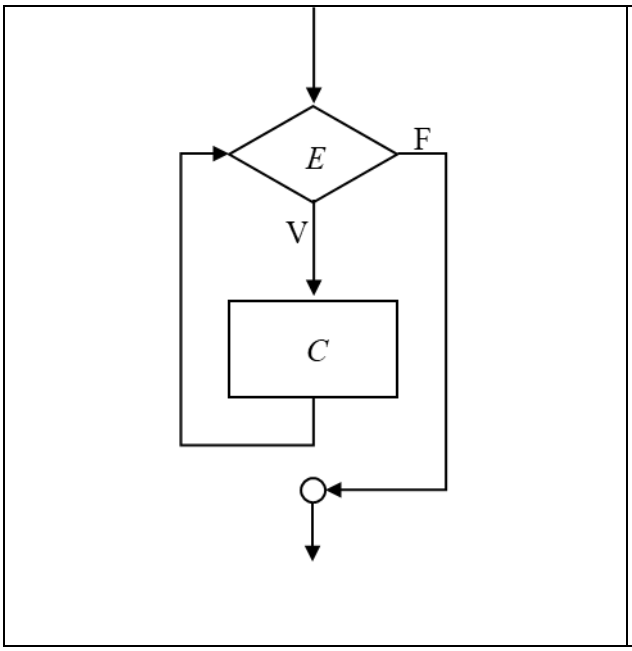
[6] Pista: el comando nulo ya existía en el compilador original. Estudie eso para que comprenda por qué trabajar el **`nil`** es un *no-problema* para el analizador contextual y para el generador de código.

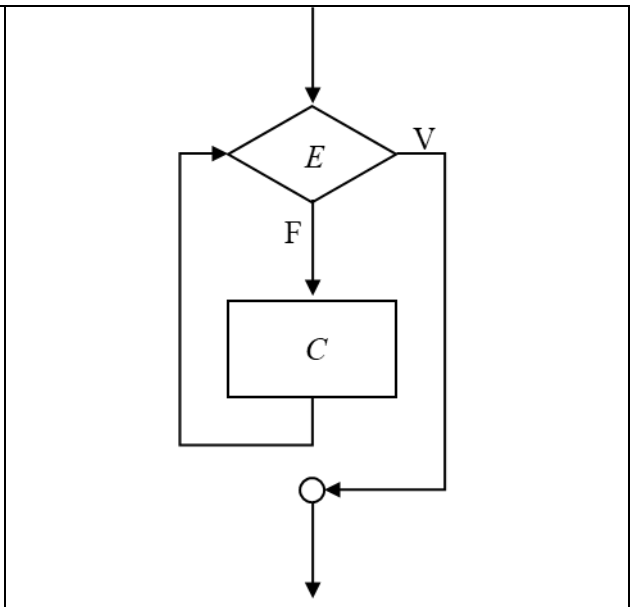
[7] Su analizador sintáctico ya debe haber dejado los ASTs con la forma apropiada, de manera que trabajar el **`if`** sea un *no-problema* para el analizador contextual y para el generador de código.

[8] Recuerde que el analizador contextual aseguró que las literales son únicas en un mismo **`select`**.

[9] Esto *no* implica añadir una nueva instrucción a TAM. Basta con usar `HALT` y *codificar* de manera distinta las diversas causas de error que llevan a una terminación abortiva. El código de operación ocupa 4 bits, por lo que hay 28 bits disponibles para codificar otros datos relacionados con las razones para detener la ejecución de un programa. Tanto el intérprete como el desensamblador deben ser modificados para que el comportamiento del programa y la interfaz de usuario sean consistentes (incluyendo lo que se muestra en la consola). Un mensaje de error podría ser "`Unmatched expression value in select command`".

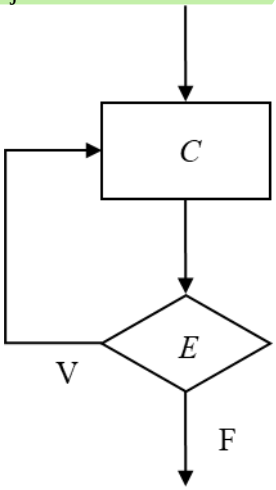
[10] TAM tiene 15 códigos de instrucción y usa 4 bits para codificar las instrucciones. Existe a posibilidad de añadir una instrucción a TAM para facilitar el procesamiento de las comparaciones en el **`select`**.





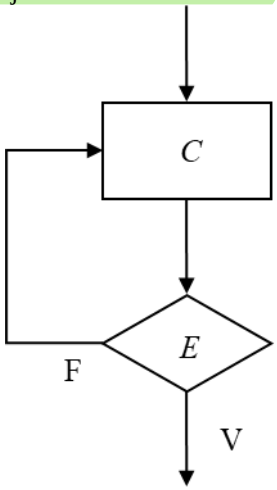
`loop do` *C* `while` *E* `end`

(*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.



`loop 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.



loops for

El comando

`loop for` *Id* `from` Exp_1 `to` Exp_2 `do` *Com* `end`

declara una variable *Id* que es ***local*** a la repetición[11] y debe cumplir con las restricciones indicadas en el Proyecto #2. Tal comando se ***comporta*** como el código que sigue[12]:

```
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 loop while Id <= $Final do !mientras no se haya excedido el límite superior
     let
       !se re-declara Id como constante para usarlo en Com
       !esto protege a Id dentro de Com, dada la estructura de bloques de Δ
       const Id ~ Id
     in Com
     end ; !el let interno comprende 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 <u>una sola vez</u>, al inicio de la repetición (por eso se asocia el valor a una constante, `$Final`) que debe ser entera – como lo garantiza el analizador contextual del Proyecto #2. 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 da el valor *inicial* de la variable de control `Id`.
- La variable de control (`Id`) es *declarada* por el comando **`loop_for_from_do_end`**. Cuando la repetición termina, el espacio (memoria) ocupado por esa variable debe ser liberado[13].
- La variable de control, llamémosla `Id`, *debe 'funcionar como' una constante* en el comando subordinado del **`loop_for_from_do_end`** (`Com`). En el comando `Com`, a `Id` no se le pueden asignar valores; estas restricciones se presuponen aseguradas por el analizador contextual, que registró apropiadamente una asociación en el ambiente. El generador de código debe emitir las instrucciones necesarias para *actualizar* el valor de la variable de control (`Id`)[14].

En las variantes condicionadas del **`loop_for_from_do_end`**, Exp_3 puede 'ver' la variable de control *Id* y todo el contexto visible para el comando en que aparece. Exp_1 y Exp_2 deben ser evaluadas <u>una sola vez</u>, antes de iniciar las repeticiones. En las repeticiones, Exp_3 debe ser evaluada *antes* de ejecutar *Com*. La semántica explicada anteriormente se extiende así:

- **`loop for`** *Id* **`from`** Exp_1 **`to`** Exp_2 **`while`** Exp_3 **`do`** *Com* **`end`**:
  Si *Id* <= Exp_2 **y** Exp_3 es verdadera, se ejecuta *Com*; si Exp_3 es falsa **o** si *Id* > Exp_2, se termina la ejecución del **`loop for`**.

- **`for`** *Id* **`from`** Exp_1 **`..`** Exp_2 **`until`** Exp_3 **`do`** *Com* **`end`**:
  Si *Id* <= Exp_2 **y** Exp_3 es falsa, se ejecuta *Com*; si Exp_3 es verdadera **o** si *Id* > Exp_2, se termina la ejecución del **`loop for`**.

---

[11] La 'variable de control' se crea para cada activación del **`loop_for_from_do_end`**. Como en todo comando, 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 iterativo.

[12] El código busca *explicar* el comportamiento del **`loop_for_from_do_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. Esa constante no es parte del código del programa fuente. Haga una interpretación clara y eficiente de la semántica deseada para este comando.

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

[14] 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 esbozamos una plantilla que puede ayudarle a plantear su propio esquema de generación de código para el comando **`loop_for_from_do_end`**. Recuerde que el analizador contextual debió asegurar que `Id` no puede ser modificada vía asignaciones por `Com` ni puede ser pasada por referencia.

En detalle: El comando

for *Id* **from** *Exp*$_1$ **..** *Exp*$_2$ **while** *Exp*$_3$ **do** *Com* **end**

declara una variable *Id* que es *local* a la repetición. Tal comando se ***comporta*** como[15]:

```
let const $Final ~ Exp2 ; !el valor final se evalúa solo una vez
    var   Id := Exp1 !Id obtiene como primer valor el que tiene Exp1
    var   $Seguir := true !para controlar si la condición interna fuerza la terminación
in loop while Id <= $Final /\ $Seguir do
                !mientras no se haya excedido el límite superior y Exp3 sea verdadera
     let
       const Id ~ Id !se re-declara Id como constante para usarlo en Com
                     !esto protege a Id dentro de Com
     in if Exp3 then Com else $Seguir := false end
     end ; !el let interno comprende únicamente Com, que llega hasta aquí
     Id := Id + 1 !se incrementa la variable de control, Id, declarada en el primer let
     !seguir las repeticiones
   end
end
```

Observe que la expresión `Exp3` debe ser booleana, *conoce* a la variable de control `Id`, se re-evalúa en cada iteración y se usa su valor para determinar si debe continuarse iterando. Las demás características del **`loop_for_from_do_end`** valen aquí también.

El comando

for *Id* **from** *Exp*$_1$ **..** *Exp*$_2$ **until** *Exp*$_3$ **do** *Com* **end**

declara una variable *Id* que es *local* a la repetición. Tal comando se ***comporta*** como[16]:

```
let const $Final ~ Exp2 ; !el valor final se evalúa solo una vez
    var   Id := Exp1 !Id obtiene como primer valor el que tiene Exp1
    var   $Seguir := true !para controlar si la condición interna fuerza la terminación
in loop while Id <= $Final /\ $Seguir do
                !mientras no se haya excedido el límite superior y Exp3 sea falsa
     let
       const Id ~ Id !se re-declara Id como constante para usarlo en Com
                     !esto protege a Id dentro de Com
     in if Exp3 then $Seguir := false else Com end
     end ; !el let interno comprende únicamente Com, que llega hasta aquí
     Id := Id + 1 !se incrementa la variable de control, Id, declarada en el primer let
     !seguir las repeticiones
   end
end
```

Observe que la expresión `Exp3` debe ser booleana, *conoce* a la variable de control `Id`, se re-evalúa en cada iteración y se usa su valor para determinar si debe continuarse iterando. Las demás características del **`loop_for_from_do_end`** valen aquí también.

El comando de repetición para acceder a elementos de un arreglo

loop for *Id* **in** *Exp* **do** *Com* **end**

declara una variable *Id* que es ***local*** a la repetición[17]. Damos por sentado que este comando cumple con las restricciones indicadas en el Proyecto #2. Supongamos que *Exp* tiene tipo **array** *InL* **of** *TD*. Llamemos *arr* al arreglo denotado por *Exp*. La literal entera *InL* determinará el número de elementos del arreglo; el Analizador

[15] No translitere esta descripción. Piense en la semántica en términos de un diagrama de flujo (de control) semejante a los anteriores para plantear su esquema de generación de código.
[16] Ver nota al pie precedente.
[17] La 'variable de iteración' se crea para cada activación del **`loop_for_in_do_end`**. Como en todo comando, 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 iterativo.

Contextual no permite declarar arreglos con 0 elementos. Los elementos de *arr* serán recorridos uno-a-uno, desde el índice 0 hasta el índice *InL* – 1. La variable *Id* tomará, consecutivamente los valores *arr*[0], *arr*[1], …, *arr*[*InL* – 1]. Como cada elemento es de tipo *TD*, la variable de iteración *Id* tiene tipo *TD*. El descriptor de la entidad *Id* debe ser planteado cuidadosamente.

***var** inicializada*

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[18] 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* (un trabajo que adelanta el analizador contextual).

local** y **rec

Debe asignarse espacio y direcciones *aparte* para cada una de las entidades declaradas dentro de las declaraciones compuestas **`local`** y **`rec`**.

Las entidades que pueden declararse mediante **`rec`** son *procedimientos* y *funciones*, de manera que puedan 'verse' unas a otras e invocarse de manera (mutuamente) recursiva. El espacio y las direcciones de estas entidades se refieren a memoria de *código*. Estas entidades no requieren memoria de datos mientras no sean invocadas; al ser invocadas, se creará un marco (*frame*) para registrar cada activación; no hay diferencia con los **`proc`** y las **`func`** de Δ original. Las entidades declaradas en un **`rec`** deben conocer las direcciones (de código) de todas las demás entidades introducidas en la *misma* declaración **`rec`** (las entidades son mutuamente recursivas). En las declaraciones **`rec`**, las direcciones de las entidades declaradas son de *código*; estas direcciones corresponden al 'punto de entrada' del procedimiento o función en cuestión. Es natural que la generación de código para **`rec`** requiera *dos* pasadas sobre el árbol de esa declaración compuesta, a fin de resolver las referencias 'hacia adelante' mediante un proceso de "parchado"[19]. 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. Suponemos que el analizador contextual dejó correctamente establecidas las referencias dentro del AST decorado (los "punteros rojos").

`local` exporta las entidades que están en la zona entre el **`in`** y el **`end`**, pero estas deben tener acceso a lo declarado como *local* ('privado')[20]. En una declaración **`local`** Dec_1 **`in`** Dec_2 **`end`**, tanto las declaraciones de la parte *privada* (Dec_1) como las de la parte *pública* (Dec_2) requieren espacio de almacenamiento. Aparte de esto, desde la perspectiva de la generación de código, el procesamiento de las declaraciones compuestas **`local`** 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").

Nota: En el generador de código *no* hay una tabla de identificación. Los "punteros rojos" creados por el analizador contextual cumplen ese propósito. Los descriptores de entidades declaradas, quedan 'colgados' a los ASTs de las declaraciones y contienen los datos necesarios para que se pueda asignarles espacio, asociarles direcciones y tamaño, y, con esos datos, generar el código para acceder a la entidad.

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

---

[18] Debe quedar inmediatamente debajo de la celda de memoria apuntada por el registro `ST`, cuando este haya sido actualizado.
[19] En clases sugerimos aprovechar el patrón 'publish-subscribe'. Hay otras maneras ingeniosas de resolver el problema.
[20] Lo privado (local) ha sido garantizado por el Analizador contextual. Lo privado está en la zona entre **`local`** e **`in`**. Observe que las declaraciones de la parte privada (Dec_1) de una declaración **`local`** Dec_1 **`in`** Dec_2 **`end`** también requieren espacio de almacenamiento.

Posibles modificaciones a TAM
Para facilitar la realización de este proyecto, sugerimos modificar el repertorio de instrucciones de TAM, en un par de situaciones.

`select`. En clase discutimos algunos esquemas como los requeridos para el manejo del **`select`**. En algunos casos, la generación podría facilitarse extendiendo TAM con una instrucción específica, `CASE`, parecida a `JUMPIF`. El comportamiento sería así: si el valor literal contenido por la instrucción *coincide* con el que está en la cima de la pila, se transfiere el control a la dirección (de código) indicada y libera el espacio ocupado por el valor de la expresión selectora (valor selector). En caso contrario **no** da "pop" al elemento superior de la pila[21] y la ejecución sigue 'hacia abajo'. Puede haber errores de ejecución para el comando **`select`**. Estos ocurren cuando el valor selector no coincide con *ninguna* de las literales contenidas en el **`select`**. En estos casos, es útil codificar información dentro de una instrucción `HALT` (llamémosla *`CASEERROR`*), que detiene la ejecución del programa (como el `HALT` original), pero que indica que esa detención es anormal mediante una modificación del *status* (dado que `HALT` tiene campos sin utilizar, *no* es necesario crear una nueva instrucción). La pseudo-instrucción `DUP` que vemos en clase puede simularse fácilmente con `LOAD` o con `STORE`.

Acceso a arreglos dentro de cotas. También podríamos modificar el repertorio de instrucciones de TAM, en cuanto a acceder arreglos de Δ, para comprobar, *en tiempo de ejecución*, que el valor correspondiente a la expresión selectora (la que indiza) está dentro del rango:

- entre 0 e *IL* – 1 (ambos inclusive) para arreglos declarados de tipo **`array`** *IL* **`of`** *T*.

Una posibilidad es introducir una nueva ***primitiva***, *`indexcheck`*, que acceda los tres elementos superiores de la pila, los cuales podrían ser[22]: índice, cota superior y cota inferior; el generador de código debe emitir las instrucciones para que esos valores estén en la pila antes de ejecutar el `CALL` *`indexcheck`*. Cuando el índice está fuera de rango debe abortarse la ejecución, indicando la naturaleza del error (`"array out of bounds"`), lo cual puede codificarse con bits disponibles en la instrucción `HALT`. Una alternativa es que la plantilla de generación de código para el acceso a arreglos tenga instrucciones `JUMPIF` idóneas para hacer las comparaciones con las cotas del (tipo) del arreglo[23].

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.
- 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. Deben extender el trabajo que ya hace el IDE sobre el lenguaje Δ original: visitar los ASTs y construir sub-ASTs apropiados que se presentan en la pestaña. *Presentar los ASTs de manera **gráfica** es opcional; pueden usar los componentes de la sub-carpeta 'TreeDrawer' bajo 'Triangle'. El compañero Leonardo Fariña grabó un tutorial explicativo, que pueden encontrar en la carpeta de videos en OneDrive*.
- 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 (`.tri`).
- 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 cualquier cambio realizado a TAM, para que la salida sea significativa. `Disassembler.java` contiene facilidades para crear esta funcionalidad.

---

[21] `JUMPIF` *sí* haría el salto y además haría "pop" a la pila**.**
[22] Desde la cima (el "top") hacia "abajo", es decir, desde el `[ST]-1` hacia el `SB`.
[23] ¡Cuidado! Recuerde que `JUMPIF` saca valores de la pila una vez realizada la comparación.

- 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; esto se obtiene al recorrer el AST (decorado) y acceder los descriptores asociados a las declaraciones de las entidades
- Si un programa terminase anormalmente (aborta), en la consola debe indicarse claramente la razón. Escríbala en inglés, para que haya consistencia con los demás mensajes de error.
- El código TAM solo debe ser generado (y ser ejecutable) cuando la compilación del programa fuente en Δxt está libre de errores léxicos, sintácticos y contextuales.

Ustedes deben basarse en los programas facilitados por el profesor como punto de partida; basarse en código correspondiente a proyectos de semestres anteriores es muy peligroso: además de ser algo fraudulento, probablemente los proyectos pasados tienen diferencias sutiles respecto del semestre actual. Su programación debe ser consistente con el estilo presente en el procesador en Java usado como base, y ser respetuosa de ese estilo. *En el código fuente debe estar claro dónde hay modificaciones introducidas por ustedes, mediante comentarios que indiquen el nombre de cada persona que hizo el cambio o la adición*.

Deben dar crédito por escrito a cualquier fuente de información o de ayuda que hayan consultado.

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 nuestros asistentes puedan someter a pruebas su procesador sin dificultades. Si Ud. trabaja en Linux, Mac OS o alguna variante de Unix, por favor avise cuanto antes al profesor y a los asistentes.*

Documentación
La documentación es un entregable y va en un solo documento. *Nada* en la documentación es opcional, salvo lo indicado explícitamente como 'extra'. La documentación tiene un peso importante en la calificación del proyecto. Si *no* modifica partes del generador de código o del intérprete de TAM para cumplir con algo requerido por este proyecto, indíquelo explícitamente. Las descripciones solicitadas como *esquemas de generación de código* deben orientarse al *diseño* de la solución, ***no*** a mostrar el código. Para ello, use plantillas como las de Watt & Brown, aumentadas con comentarios sobre los descriptores de entidades, etc.

Debe documentar clara y concisamente los siguientes puntos:
- Descripción del esquema de generación de código (plantilla) para el comando **`nil`**.
- Descripción del esquema de generación de código (plantilla) para el comando **`if`**.
- Descripción del esquema de generación de código (plantilla) para *todas* las variantes del comando **`loop`** con condiciones.
- Descripción del esquema de generación de código (plantilla) para el comando **`loop for`** y sus variantes con **`while`** y **`until`**.
- Descripción del esquema de generación de código (plantilla) para el comando **`select`**.
- Descripción del esquema de generación de código (plantilla) para el comando **`loop_for_in_do_end`**.
- Solución dada al procesamiento de declaraciones de variable inicializada (**`var`** *Id* **`:=`** *Exp*).
- Su solución al problema de introducir declaraciones privadas (**`local`**).
- Su solución al problema de introducir declaraciones de procedimientos o funciones mutuamente recursivos (**`rec`**).
- Descripción de su solución al acceso válido a elementos de arreglos (índices dentro del rango dado por las cotas inferior y superior).
- 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 nuevos errores de generación de código o de ejecución detectados.
- Describir cualquier cambio que requirió hacer al analizador sintáctico o al contextual para efectos de lograr la 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 y el intérprete[24].
- Discusión y análisis de los resultados observados. Conclusiones obtenidas a partir de esto.
- Descripción resumida de las tareas realizadas por cada miembro del grupo.
- Breve reflexión sobre la experiencia de modificar fragmentos de un (compilador | intérprete | ambiente) escrito por terceras personas, así como la 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 de todos los programas debe indicar con claridad los puntos en los cuales su equipo ha hecho modificaciones.
  - Sub-carpetas donde se encuentre el código objeto del compilador+intérprete, en formato directamente ejecutable desde el sistema operativo Windows[25]. Todo el contenido del archivo comprimido debe venir libre de infecciones. Debe incluir el IDE enlazado a su compilador+intérprete, de manera que desde el IDE se pueda ejecutar sus procesadores de Δxt y de TAM.
- La carpeta comprimida debe llamarse "`Proyecto 3`" y estar seguida por sus números de carnet separados por espacios en blanco.
- 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. El código objeto de su compilador (programa principal) debe estar en un formato directamente ejecutable en Windows (u otro sistema operativo).
  - Programas (`.tri`) de prueba que preparados por su grupo.

Grupos
Los grupos pueden ser de *hasta* **4** personas. Los grupos de *cuatro* miembros *obligatoriamente* deben procesar el comando **`select`** completo y la variante **`loop for in`**.

Puntos extra
Los grupos de *hasta* tres miembros podrán obtener puntos extra por su procesamiento completo del comando **`select`** completo y la variante **`loop for in`**.

Independientemente del número de miembros del grupo, la comprobación correcta de índices de arreglos (dentro de cotas) en tiempo de ejecución conlleva puntos extra.

Los comandos repetitivos con nombre y los escapes (**`leave`**, **`next`**, **`return`**) permitirán obtener puntos extra a los grupos, cualquiera que sea su cantidad de miembros. En este documento no se está especificando su semántica. Los grupos interesados en trabajar sobre esas extensiones deben comunicarse directamente con el profesor.

Entrega
Fecha límite: **lunes 2022.11.28,** antes de las 08:00 a.m. No se recibirán trabajos después de la fecha y la hora indicadas.

[24] Puede usar como base los casos de prueba que publique el profesor.
[25] Es decir, sin necesidad de compilar los programas fuente. En principio, se permitirá entregar el trabajo en otro ambiente, pero deben avisar de previo al profesor y a los asistentes.

Deben enviar por correo-e el ***enlace***[26] a un archivo comprimido almacenado en alguna nube, con todos los elementos de su solución a estas direcciones: itrejos@itcr.ac.cr, svengra01@gmail.com (Steven Granados Acuña, Asistente). El *archivo* comprimido debe llamarse **Proyecto 3 carnet carnet carnet carnet**

El asunto (*subject*) de su mensaje debe ser: **IC-5701 Proyecto 3 carnet carnet carnet carnet**

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 de l@s asistentes (caso de que su mensaje fuera obviado por no tener el asunto apropiado). Si su mensaje no es legible (por cualquier motivo), su carpeta está en formato **.rar**, o contiene un virus, la nota será **0**.

La documentación vale alrededor de un 25% de la nota de cada proyecto. 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.

[26] 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 nuestro asistente mediante un mensaje de correo con el formato indicado. Dar permiso de lectura a ambos. Deben mantener la carpeta viva hasta 30 de junio del 2023.