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:
- 2021.06.02: Versión base (v0).

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 las especificaciones 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 darán 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). Debe 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'. En clases, discutimos sobre 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 ***anterior*** fue deficiente, *puede utilizar el programa desarrollado por otros compañeros como base para esta tarea programada*, ***pero deberá pedir la autorización de reutilización de 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 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).

Lenguaje fuente

[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.

Sintaxis
Remítase a la especificación del Proyecto #1 (‘`IC-5701 2021-1 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.

Es importante que entienda bien cómo funciona la variable de control en el comando de repetición controlada por contador, **`loop for`** *Id* **`from`** Exp_1 **`to`** Exp_2 **`do`** *Com* **`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 de sus propiedades contextuales aseguradas 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] (vía la tabla de identificación, donde la variable de control *Id* está asociada 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 Δ que han pasado el análisis contextual. No se debe generar código para programas con errores contextuales, sintácticos o léxicos.

> Para los valores declarados como 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. En un acceso a arreglo, el valor que resulta de evaluar la expresión seleccionadora (indexadora o indizadora) 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`". El compilador original no genera código que compruebe índices dentro de rango al acceder arreglos. Note que hay tanto *variables* como *constantes* de tipo **`array`**.

El comando nulo **`nothing`** 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].

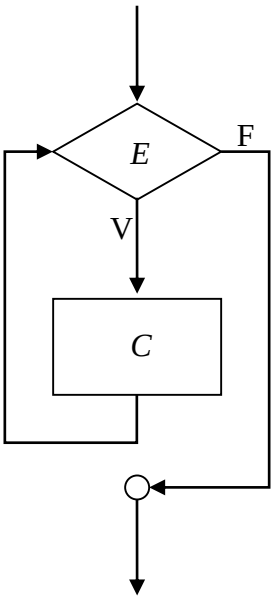
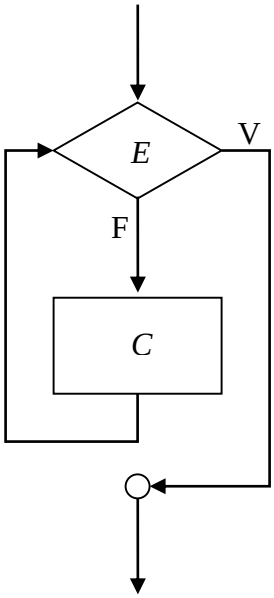
> En el comando **`choose`**, la expresión selectora se evalúa *una sola vez*. El valor resultante selecciona aquella rama que incluye una literal que coincide con el valor de la expresión; si se especificó la opción **`else`** y *ninguna* de las literales comprendidas en las ramas (**`when`**) del **`choose`** coincidió con el valor de la expresión, entonces se selecciona

---

[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 **`loop_for_from_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 ni el de una variable*.
[5] Pista: el comando nulo ya existía en el compilador original. Estudie eso para que comprenda por qué trabajar el **`nothing`** es un *no-problema* para el analizador contextual y para el generador de código.
[6] 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.

la rama del **`else`**[7]. Si ***no*** se especificó una rama **`else`** y ninguna de las literales coincide con el valor de la expresión selectora, entonces la ejecución debe ***abortar*** con un mensaje de error apropiado[8]. (Tiene sentido proceder de "izquierda a derecha" en las comparaciones de literales o etiquetas con el valor selector, y mantener el valor selector en la pila mientras se ejecute el comando[9].)

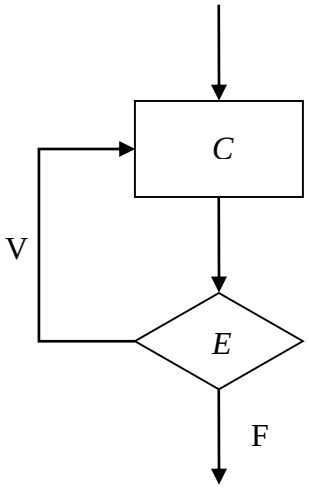
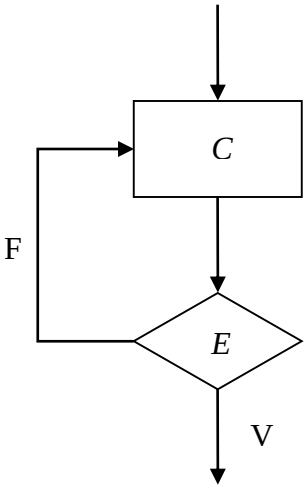
Los comandos repetitivos pueden ser ejecutados cero, una o más veces, según se indica a continuación.

`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).
	

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

[8] 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. 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 choose command`".

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

loop do *C* **while** *E* **end**	**loop do** *C* **until** *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.	(*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

loop for *Id* **from** *Exp*$_1$ **to** *Exp*$_2$ **do** *Com* **end**

declara una variable *Id* que es ***local*** a la repetición[10] y cumple con las restricciones indicadas en el Proyecto #2. Tal comando se ***comporta*** como el código que sigue[11]:

```
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
       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 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:

[10] La 'variable de control' se crea para cada activación del **loop-for-from-to-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.

[11] Ese código es para *explicar* el comportamiento del **loop-for-from-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.

- 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 **`loop_for_from_to_end`**; cuando la repetición termina, esta variable debe desaparecer de la memoria[12].
- La variable de control *debe 'funcionar' como una constante* en el cuerpo del **`loop_for_from_to_end`** (`Com`): en el comando `Com` no se le pueden asignar valores; estas restricciones se presuponen aseguradas por el analizador contextual, al registrar 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`)[13].

En las variantes condicionadas del **`loop_for_from_to_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 una sola vez, 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`**.

- **`loop for`** *Id* **`from`** Exp_1 **`to`** 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`**.

En detalle: El comando

`loop for` *Id* **`from`** Exp_1 **`to`** Exp_2 **`while`** Exp_3 **`do`** *Com* **`end`**

declara una variable *Id* que es *local* a la repetición. Tal comando se ***comporta*** como[14]:

```
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_to_do_end`** valen aquí también.

---

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

[13] 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 **`loop-for-from-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.

[14] 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.

El comando

loop for *Id* **from** *Exp*$_1$ **to** *Exp*$_2$ **until** *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 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_to_do_end** valen aquí también.

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[16] 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 ya realizado por el analizador contextual).

Debe asignarse espacio y direcciones *aparte* para cada una de las entidades declaradas dentro de las declaraciones compuestas **private** y **recursive**.

Las entidades que pueden declararse mediante **recursive** son *procedimientos* y *funciones*, por lo que el espacio y las direcciones son en memoria de *código*. Las entidades declaradas en un **recursive** deben conocer las direcciones (de código) de todas las demás entidades introducidas en la misma declaración. En las declaraciones **recursive**, las direcciones de las entidades declaradas son de *código*, que corresponden al 'punto de entrada' del procedimiento o función en cuestión. Es natural que el procesamiento de **recursive** requiera *dos* pasadas sobre el árbol de esa declaración compuesta, para poder resolver las referencias 'hacia adelante' mediante un proceso de "parchado"[17]. Por lo demás, desde la perspectiva de la generación de código, el procesamiento de las declaraciones compuestas **recursive** 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").

private exporta entidades, pero estas deben tener acceso a lo declarado como *local* ('privado')[18]. Las declaraciones de la parte *privada* (*Dec*$_1$) de una declaración **private** *Dec*$_1$ **in** *Dec*$_2$ **end** también requieren espacio de almacenamiento. Desde la perspectiva de generación de código, el procesamiento de las declaraciones compuestas **private** no ofrece otras complicaciones adicionales a las que presenta el procesamiento de las declaraciones

[15] Ver nota al pie precedente.
[16] Debe quedar inmediatamente debajo de la celda de memoria apuntada por el registro `ST`, cuando este haya sido actualizado.
[17] En clases se sugiere aprovechar el patrón 'publicador-suscriptores' ('publish-subscribe').
[18] Las declaraciones de la parte privada (*Dec*$_1$) de una declaración **private** *Dec*$_1$ **in** *Dec*$_2$ **end** también requieren espacio de almacenamiento.

secuenciales. Suponemos que el analizador contextual dejó correctamente establecidas las referencias dentro del AST decorado (los "punteros rojos").

Los *paquetes* (**`package`**) se procesan en secuencia. El analizador contextual asegura que, una vez procesada una declaración de paquete, los identificadores declarados allí quedan a disposición de los paquetes subsecuentes o del "comando principal" que debe venir después de la declaración del último paquete. Las variables y constantes declaradas en los paquetes estarán en el nivel 0 (*global*), serán colocadas en las primeras celdas de la memoria de datos y serán direccionadas como globales (desplazamientos relativos al registro `SB`). Las funciones y los procedimientos declarados en los paquetes ocuparán las primeras celdas de la memoria de código, y estarán antes de las instrucciones que se generen para el "comando principal". Desde la perspectiva de generación de código, el procesamiento de las declaraciones de paquetes se asemeja al procesamiento de las declaraciones secuenciales, siempre que a las entidades declaradas en los paquetes les sean asociadas ubicaciones en la memoria que les corresponde (datos o código) *antes* de procesar el "comando principal". 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.

Generación de código y modificaciones a TAM
Para facilitar la realización de esta tarea, sugerimos modificar el repertorio de instrucciones de TAM, en un par de situaciones.

`choose`. En clase discutimos algunos esquemas como los requeridos para el manejo del **`choose`**. En algunos casos, la generación podría facilitarse extendiendo TAM con una instrucción específica, `CASE`, parecida a `JUMPIF`: 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, pero en caso contrario **no** da "pop" al elemento superior de la pila[19]. Para el manejo de errores en el contexto del **`choose`**, 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* (`HALT` tiene campos sin utilizar, por lo que *no* es necesario crear una nueva instrucción). La pseudo-instrucción `DUP` que vemos en clase puede simularse fácilmente con `POP` 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 seleccionadora (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[20]: 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[21].

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.

---

[19] `JUMPIF` *sí* haría el salto.
[20] Desde la cima (el "top") hacia "abajo", es decir, desde el `ST` hacia el `SB`.
[21] ¡Cuidado! Recuerde que `JUMPIF` saca valores de la pila una vez realizada la comparación.

- 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 cualquier cambo realizado a TAM, para que la salida sea significativa. `Disassembler.java` da facilidades para proveer esta funcionalidad.
- 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, *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.

Como hemos indicado, ustedes deben basarse en los programas que les han sido dados como punto de partida. Su programación debe ser consistente con el estilo presente en los programas en Java usados como base (compilador de Δ, intérprete de TAM), y ser respetuosa de ese estilo. En el código fuente deben incluir comentarios que indiquen claramente los lugares donde ustedes han introducido modificaciones.

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 al profesor y a los asistentes cuanto antes.*

Documentación

Nada en la documentación es opcional. 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 deben orientarse al *diseño* de la solución, ***no*** a mostrar el código. 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 para el comando **`nothing`**.
- Descripción del esquema de generación de código para el comando **`if_then_elsif_else_end`**.
- Descripción del esquema de generación de código para el comando **`choose`**.
- Descripción del esquema de generación de código para *todas* las variantes de **`loop`** … **`end`**.
- Descripción del esquema de generación de código para **`loop for`** … **`end`** y sus variantes con **`while`** y **`until`**.
- 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 (**`recursive`**).
- Su solución al problema de introducir declaraciones de paquetes (**`package`**) y al acceso a las entidades declaradas dentro de paquetes.
- 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.
- 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 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 los programas 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[22]. 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 Δ 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.
  - Programas (`.tri`) de prueba que preparados por su grupo.

Puntos extra

Los grupos de hasta tres miembros podrán obtener puntos extra por su procesamiento completo del comando **`choose`**.

Los grupos de *cuatro* miembros obligatoriamente deben procesar el comando **`choose`** completo.

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

Entrega

Fecha límite: **lunes 2021.06.28**, antes de las 13:00. No se recibirán trabajos después de la fecha y la hora indicadas.

Los grupos pueden ser de *hasta* **4** personas.

Debe enviar por correo-e el ***enlace***[23] a un archivo comprimido almacenado en la nube con todos los elementos de su solución a estas direcciones: `itrejos@itcr.ac.cr`, `susana.cob.3@gmail.com` (Susana Cob García, asistente), `a.tapia1908@gmail.com` (Alejandro Tapia Álvarez, asistente).

[22] Es decir, sin necesidad de compilar los programas fuente.

[23] Los sistemas de correo del TEC rechazan 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 sus asistentes mediante un mensaje de correo con el formato indicado.

El asunto (subject) 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 los asistentes (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 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.