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

Proyecto #2, la fase de análisis contextual de un compilador

Historial de revisiones:
- 2022.09.20 Versión base (v0).
- 2022.10.06 Versión base, sin marca de agua (v1).

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

Objetivo

Al concluir este proyecto Ud. habrá comprendido los detalles relativos a fase de análisis contextual de un compilador escrito "a mano" al aplicar las técnicas expuestas por Watt y Brown en su libro *Programming Language Processors in Java* y las explicaciones dadas en clases. Ud. deberá extender el compilador del lenguaje imperativo Δ desarrollado por Watt y Brown (en Java), de manera que sea capaz de procesar las extensiones descritas en el enunciado del Proyecto #1 y en la sección *Lenguaje fuente* que aparece abajo. El lenguaje extendido será denominado **Δxt** en el resto de este documento. El lenguaje tiene estructura de bloques y su sistema de tipos es monomórfico.

Además de realizar el trabajo de análisis léxico y sintáctico logrado en el Proyecto #1, su compilador debe ser capaz de procesar el lenguaje Δxt conforme las características contextuales (identificación y tipos) especificadas en este documento. 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.

Bases

Para entender las técnicas expuestas en el libro de texto, Ud. deberá estudiar el compilador del lenguaje imperativo Δ y el intérprete de la máquina abstracta TAM, ambos desarrollados en Java por los profesores David Watt y Deryck Brown. El compilador y el intérprete han sido ubicados en la carpeta 'Recursos' del curso (@ OneDrive y @ tecDigital, bajo 'Documentos'), para que Ud. los descargue y los estudie; posteriormente, los modificará según lo especificado en los proyectos del curso. En ese repositorio también pueden encontrar un ambiente interactivo de edición, compilación y ejecución (IDE) desarrollado por el Dr. Luis Leopoldo Pérez (implementado en Java) y corregido por los ex-estudiantes Ing. Pablo Navarro y Ing. Jimmy Fallas. Si desea aprender acerca de cómo integrar el IDE y el compilador, siga las indicaciones preparadas por el Dr. Pérez y nuestro ex-asistente, Ing. Diego Ramírez Rodríguez, en cuanto a las partes del compilador que debe desactivar para poder trabajar, así como los ajustes necesarios para que el IDE y el compilador funcionen bien conjuntamente. *No se darán puntos extra a los estudiantes que desarrollen su propio IDE*; ***no*** *es objetivo de este curso desarrollar IDEs para lenguajes de programación*. Sin embargo, si uno o más grupos de estudiantes aportan un IDE interesante y confiable, podemos considerarlo para futuras ediciones de este curso.

Estudie el capítulo **5** y los apéndices **B** y **D** del libro de Watt y Brown, así como el código del compilador, para comprender los principios y las técnicas por aplicar, el lenguaje fuente original (Δ, que estamos extendiendo) y las interdependencias entre las partes del compilador. En clases dimos sugerencias sobre cómo abordar algunos de los retos que plantea la extensión de Δ sujeto de este proyecto (Δxt). Ud. extenderá el compilador que resultó de su trabajo en el Proyecto #1 de este curso (aunque es lícito basarse en trabajo de otros compañeros, según indicamos abajo).

Si su Proyecto #1 fue deficiente, *puede utilizar el programa desarrollado (para el Proyecto #1) por otr@s compañer@s como base para esta tarea programada,* ***pero deberá pedir la autorización de reutilización a sus***

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

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.

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

Remítase a la especificación del Proyecto #1 ('`IC-5701 2022-2 Proyecto 1 v0.pdf`').

Contexto: identificación y tipos

Nos referiremos a las reglas sintácticas del lenguaje Δxt para indicar las restricciones de contexto. Interprete esas reglas sintácticas desde una perspectiva *contextual*, esto es, vea en ellas *árboles de sintaxis abstracta* y no secuencias de símbolos. En lo que sigue usaremos las siguientes *meta-variables* para hacer referencia a sub-árboles de sintaxis abstracta de las clases sintácticas que se indican[3]:

• *V*: `V-name`	• PF_1, PF_2, PF_i, PF_n : `Proc-Func`
• *Id*, Id_1, Id_2, Id_L: `Identifier`	• *PFs*: `Proc-Func*`
• *Exp*, Exp_1, Exp_2, Exp_3, Exp_i : `Expression`	• *Cases*: `Cases`
• *Com*, Com_1, Com_2, Com_i : `Command`	• *C*: `Case`
• *Dec*, Dec_1, Dec_2, Dec_3 : `Declaration`	• *Cs*: `Case*`
• *TD*: `Type-denoter`	• Cl_1, Cl_2: `Case-Literal`
• *FPS*: `Formal-Parameter-Sequence`	• *InL* : `Integer-Literal`
• *APS*: `Actual-Parameter-Sequence`	• *ChL* : `Character-Literal`

Expresiones

No hay cambios en las expresiones.

Comandos

El comando **`nil`** no requiere de revisión contextual, pues no depende de tipos ni de declaraciones de identificadores. Para el *comando* de asignación, *V* **`:=`** *Exp* aplican las restricciones contextuales de Δ. Para el *comando* de llamada a procedimiento *Id* (*APS*) aplican las restricciones contextuales de Δ y las que se explican más abajo para las declaraciones mutualmente recursivas (**`rec`**).

Considere estos comandos repetitivos añadidos a Δ:

`loop while` *Exp* **`do`** *Com* **`end`**
`loop until` *Exp* **`do`** *Com* **`end`**
`loop do` *Com* **`while`** *Exp* **`end`**
`loop do` *Com* **`until`** *Exp* **`end`**

Las restricciones contextuales son:

---

[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 #2.

[3] Usamos los subíndices de manera liberal.

- *Exp* debe ser de tipo `Boolean`.
- *Com*, y sus partes, deben satisfacer las restricciones contextuales[4].

En el comando de repetición controlada por contador

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

las restricciones son:

- Exp_1 y Exp_2 deben ser ambas de tipo *entero*. Los tipos de Exp_1 y Exp_2 deben ser determinados en el contexto en el que aparece ***este*** comando **`loop_for_from_to_do_end`**
- *Id* es conocida como la "variable de control". *Id* es de tipo entero. *Id* es ***declarada*** en este comando y su *alcance* es *Com*. ***Esta*** declaración de *Id* ***no*** es conocida por Exp_1 ***ni*** por Exp_2.
- *Com* debe cumplir con las restricciones contextuales. Su contexto es el que rige al comando **`loop_for_from_to_do_end`** dentro del que aparece *Com*, ***extendido*** por la declaración de la variable de control *Id*.
- La variable de control *Id* ***no*** puede aparecer a la izquierda de una asignación ni pasarse como parámetro **`var`**[5] en la invocación de un procedimiento o función dentro del cuerpo del comando repetitivo **`loop_for_from_to_do_end`**[6].
- Este comando repetitivo *funciona como un bloque* al declarar una variable local (*Id*, la variable de control). Las reglas usuales de anidamiento de bloques aplican aquí: si se re-declara el identificador *Id* en un comando **`let`** anidado, en una expresión **`let`** anidada, en una secuencia de parámetros (*FPS*) o en cualquier bloque anidado, este es *distinto* y hace *inaccesible* la variable de control declarada en el comando **`loop_for_from_to_do_end`**, pues sería no-local (más externa).

En las variantes condicionadas del **`loop_for_from_to_do_end`**

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

aplican las restricciones anteriores y se extienden así:

- Exp_3 debe ser de tipo *booleano*. El tipo de Exp_3 debe ser determinado en el contexto en el que aparece ***este*** comando **`loop_for_from_to_do_end`**, ***extendido*** por la variable de control *Id* (que es de tipo entero, como se indicó antes). Es decir Exp_3 've' a *Id*, además del contexto circundante.

En el comando de repetición para acceder a elementos de un arreglo

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

deben satisfacerse estas restricciones:

- *Exp* debe ser de tipo **`array`** *InL* **`of`** *TD*.
- *Id* es conocida como la "variable de iteración". *Id* es de tipo *TD*. *Id* es ***declarada*** en este comando y su alcance es *Com*; ***esta*** declaración de *Id* ***no*** es conocida por *Exp*.
- *Com* debe cumplir con las restricciones contextuales. Su contexto es el mismo que rige el comando **`loop_for_in_do_end`** dentro del cual aparece *Com*, ***extendido*** por la declaración de la variable de iteración *Id*.
- La variable de iteración *Id* ***no*** puede aparecer a la izquierda de una asignación ni pasarse como parámetro **`var`**[7] en la invocación de un procedimiento o función dentro del cuerpo del comando repetitivo **`loop_for_in_do_end`**[8].
- Este comando repetitivo *funciona como un bloque* al declarar una variable local (*Id*, la variable de iteración). Las reglas usuales de anidamiento aplican aquí (si se re-declara el identificador *Id* en un

---

[4] Observe que ésta, como muchas de las restricciones contextuales, se formulan de manera *recursiva*. Es decir, 'están bien' en todos sus niveles de anidamiento.
[5] No podrá pasarse por referencia.
[6] En clase discutimos que, en el cuerpo de este comando repetitivo (*Com*), la 'variable' de control *Id* en realidad debe ***comportarse*** como una *constante* (aunque, técnicamente, no sea una constante).
[7] No podrá pasarse por referencia.
[8] En clase discutimos que, en el cuerpo de este comando repetitivo (*Com*), la 'variable' de iteración *Id* en realidad debe ***comportarse*** como una *constante* (técnicamente, no lo es, pero debe ser tratada de manera semejante a una constante).

comando **`let`** anidado, en una expresión **`let`**, en una secuencia de parámetros (*FPS*) o en cualquier bloque anidado, este es *distinto* y hace *inaccesible* la variable de iteración declarada en el comando **`loop_for_in_do_end`**, que sería más externa).

El comando condicional de Δxt fue modificado solamente en cuanto a sintaxis. Considere:

`if` *Exp* **`then`** *Com*$_1$ (**`|`** *Exp*$_i$ **`then`** *Com*$_i$)* **`else`** *Com*$_2$ **`end`**

Las expresiones *Exp* y *Exp*$_i$ deben ser de tipo booleano. *Com*$_1$, *Com*$_i$ y *Com*$_2$, así como sus partes, deben satisfacer las restricciones contextuales.

El comando bloque

`let` *Dec* **`in`** *Com* **`end`**

se analiza contextualmente de manera idéntica al comando correspondiente en Δ original (el original no tiene **`end`** y ahora permitimos `Command` en lugar del `single-Command` original, pero esos son asuntos sintácticos).

Considere el comando de selección **`select`** *Exp* **`from`** *Cases* **`end`**

- La expresión selectora *Exp* debe ser de tipo entero (`Integer`) o de tipo carácter (`Char`).
- Todas las literales de un mismo **`select`** deben ser del mismo tipo que el de la expresión selectora[9].
- Cada literal entera *InL* o literal carácter *ChL* debe aparecer *una sola vez*, como caso, dentro de un *mismo* **`select`**[10]. Esto incluye a los sub-rangos: las intersecciones entre literales solitarias y rangos de literales deben ser vacías (dentro de un *mismo* **`select`**).
- Los **`select`**s pueden anidarse y sus literales tienen *alcance*; una literal de un **`select`** externo puede *reaparecer* dentro de un **`select`** *anidado* y esto ***no*** es un error.
- Todos los comandos subordinados (que aparecen dentro del **`select`**) deben satisfacer las restricciones contextuales[11].

Declaraciones

Los parámetros formales (*FPS*) de una abstracción (declaración de *función* vía **`func`** o declaración de *procedimiento* vía **`proc`**) forman parte de un nivel léxico (alcance) inmediatamente más profundo (+ 1) que aquel en el cual aparece el identificador de la abstracción que los declara[12]. *Usted debe verificar que ningún nombre de parámetro se repita dentro de la declaración de una función o procedimiento (en el encabezado).*

Veamos la declaración de variable inicializada:

`var` *Id* **`init`** *Exp*

Aquí, el tipo de la *variable* inicializada *Id* debe inferirse a partir del tipo de la expresión inicializadora *Exp*[13]. *Id* se "exporta" al resto del bloque donde aparece esta declaración.

---

[9] Si la expresión selectora es de tipo entero, entonces las literales que aparecen en los casos de ese **`select`** deben ser literales enteras. Si la expresión selectora es de tipo carácter, entonces las literales que aparecen en los casos del **`select`** deben ser caracteres (literales de un carácter).

[10] Tengan particular cuidado con los valores comprendidos en un sub-rango *Cl*$_1$ **`to`** *Cl*$_2$. Ese sub-rango comprende todos los valores que están en el conjunto { *lit* : *Literal* | *Cl*$_1$ ⩽ *lit* ⩽ *Cl*$_2$ }. *Literal* puede corresponder a `Integer` o a `Char`.

[11] Es decir, las restricciones contextuales deben cumplirse *recursivamente* (en profundidad) en los comandos, expresiones y declaraciones subordinados.

[12] Es decir, los parámetros formales se comportan como identificadores declarados *localmente* en el bloque de la función o procedimiento. Repase lo expuesto en el libro de Watt y Brown, el código del compilador de Δ original en Java (método `OpenScope`), así como lo discutido en clases.

[13] El procesamiento requerido es semejante al que se hace para la declaración **`const`** *Id* ~ *Exp*. La diferencia es que, en el caso que nos ocupa, *Id* es declarado como una ***variable***, **no** como una *constante*. Un identificador declarado como *variable* sí puede ser destino de una asignación o ser pasado por referencia en la invocación a un procedimiento o a una función.

En una declaración **`local`** *Dec*$_1$ **`in`** *Dec*$_2$ **`end`**, los identificadores declarados en *Dec*$_1$ son conocidos *exclusivamente* en *Dec*$_2$. Únicamente se "exportan" los identificadores declarados en *Dec*$_2$. Observe que tanto *Dec*$_1$ como *Dec*$_2$ son declaraciones generales (`Declaration`)[14].

En una declaración **`rec`** *PFs* **`end`** se permite combinar las declaraciones de varios procedimientos o funciones, de manera que puedan invocarse unos a otros (lo que pretende posibilitar la recursión mutua).

- Consideremos primero el caso de *dos* declaraciones: **`rec`** *PF*$_1$ **`|`** *PF*$_2$ **`end`**. Allí se declaran funciones y(o) procedimientos mutuamente recursivos: el identificador[15] declarado en *PF*$_1$ es conocido por *PF*$_2$, y viceversa.
- Los nombres (identificadores) de función o procedimiento que aparecen en *PF*$_1$ y *PF*$_2$ deben ser *distintos*.
- Las secuencias de parámetros que aparecen en *PF*$_1$ y *PF*$_2$ están en un nivel de profundidad léxica 1 mayor (+ 1) que el de los nombres de la función o procedimiento que les corresponden. stack
- Se "exportan"[16] los identificadores de función o de procedimiento declarados en *PF*$_1$ y *PF*$_2$. En *PF*$_1$ y en *PF*$_2$, los *parámetros* (*FPS*) declarados en un encabezado de función o de procedimiento son *privados* (es decir, *locales*) y, por lo tanto, no se "exportan".
- En el caso general, es posible declarar dos o más procedimientos o funciones como mutuamente recursivos. Sintácticamente: **`rec`** *PF*$_1$ **`|`** *PF*$_2$ **`|`** … **`|`** *PF*$_n$ **`end`**. Todos los identificadores de *función* o de *procedimiento* introducidos por las declaraciones *PF*$_i$ ($1 \le i \le n$) deben ser *distintos* y serán conocidos al momento de procesar los *cuerpos* de *todas* las funciones o procedimientos declarados en las *PF*$_i$.
- Es un error declarar más de una vez el mismo identificador[17] en las declaraciones simples que componen una *misma* declaración compuesta **`rec`**.
- Todos los identificadores de procedimiento o de función declarados vía **`rec`** son "exportados"; esto es, son agregados al contexto subsiguiente y son conocidos después de la declaración compuesta **`rec`**[18].
- Funciones o procedimientos *distintos* declarados vía **`rec`** pueden declarar ***parámetros*** con nombres idénticos, pero en secuencias de parámetros distintas. Eso *no* es problema.

Extras: comandos repetitivos con nombre

Considere los comandos repetitivos con nombre *Id*$_L$

`loop` *Id*$_L$ **`while`** *Exp* **`do`** *Com* **`end`**
`loop` *Id*$_L$ **`until`** *Exp* **`do`** *Com* **`end`**
`loop` *Id*$_L$ **`do`** *Com* **`while`** *Exp* **`end`**
`loop` *Id*$_L$ **`do`** *Com* **`until`** *Exp* **`end`**
`loop` *Id*$_L$ **`for`** *Id* **`from`** *Exp*$_1$ **`to`** *Exp*$_2$ **`do`** *Com* **`end`**
`loop` *Id*$_L$ **`for`** *Id* **`from`** *Exp*$_1$ **`to`** *Exp*$_2$ **`while`** *Exp*$_3$ **`do`** *Com* **`end`**
`loop` *Id*$_L$ **`for`** *Id* **`from`** *Exp*$_1$ **`to`** *Exp*$_2$ **`until`** *Exp*$_3$ **`do`** *Com* **`end`**
`loop` *Id*$_L$ **`for`** *Id* **`in`** *Exp* **`do`** *Com* **`end`**

- *Id*$_L$ el nombre del comando repetitivo. *Id*$_L$ es ***declarado*** en este comando y su alcance es únicamente *Com*. ***Esta*** declaración de *Id*$_L$ puede ser *explícitamente* referenciada por comandos de escape **`leave`** *Id*$_L$ o **`next`** *Id*$_L$ que aparezcan dentro de *Com*. Ninguna otra referencia al nombre *Id*$_L$ es válida dentro del alcance de este comando **`loop`** *Id*$_L$ **`… do_end`**.
- Las reglas de anidamiento de bloques y alcances aplican también aquí. Esto es, *Id*$_L$ no es visible después del **`end`** de este comando repetitivo e *Id*$_L$ puede ser redefinido en el interior de *Com*.

---

[14] No son `single-Declaration` ni `compound-Declaration`. Revise lo explicado en clases.
[15] Ese identificador da nombre a la función o al procedimiento.
[16] *Después* de la declaración **`rec`** los identificadores de procedimiento o de función serán visibles y podrán ser utilizados por declaraciones subsecuentes. Tenga cuidado con la forma en que interactúan **`rec`** y **`local`**.
[17] Sintácticamente, esas declaraciones solo pueden ser de *función* o de *procedimiento*.
[18] Esto no sucede si la declaración **`rec`** aparece dentro de una declaración **`local`** *antes* del **`in`**, pues solamente será vista en la parte comprendida entre el **`in`** y el **`end`** de la declaración **`local`**.

- Un comando **loop** Id_L **... do_end** puede ser referenciado *implícitamente* por comandos de escape **leave** o **next** anónimos (sin Id_L), que aparezcan dentro de *Com*, siempre que el **loop** Id_L **... do_end** sea el más cercano a los comandos de escape **leave** o **next** (desde adentro hacia afuera).
- *Com* y sus partes deben respetar las restricciones contextuales.

Considere cualquier comando repetitivo *sin nombre* (*anónimo*)

loop ... do_end

- Este comando **loop ... do_end** puede ser referenciado *implícitamente* por comandos de escape **leave** o **next** anónimos (sin Id_L), que aparezcan dentro de *Com*, siempre que el **loop ... do_end** sea el más cercano a los comandos de escape **leave** o **next** (desde adentro hacia afuera).
- *Com* y sus partes deben satisfacer las restricciones contextuales.

Considere el comando de escape

leave Id_L

- El identificador Id_L debe corresponder a un comando repetitivo **loop** Id_L **... do_end** que lleve ese nombre y que establezca el contexto del comando de escape **leave** Id_L.

Considere el comando de escape

leave

- Este comando de escape anónimo debe tener en su contexto un comando **loop ... do_end**, que opcionalmente puede tener nombre.

Considere el comando de escape

next Id_L

- El identificador Id_L debe corresponder a un comando repetitivo **loop** Id_L **... do_end** que lleve ese nombre y que establezca el contexto del comando de escape **next** Id_L.

Considere el comando de escape

next

- Este comando de escape anónimo debe tener en su contexto un comando **loop ... do_end**, que opcionalmente puede tener nombre.

Considere el comando de retorno explícito de procedimiento

return

- Este comando debe aparecer dentro del cuerpo de un *procedimiento* (declarado vía **proc**). No debe aparecer dentro del cuerpo de una función (**func**) o como parte del programa (comando) principal.

Proceso y salidas

Ud. modificará el procesador de Δ extendido que preparó para el Proyecto #1, de manera que sea capaz de procesar las restricciones contextuales especificadas arriba.

- El analizador contextual debe realizar completamente el trabajo de identificación (manejo del alcance en la relación entre ocurrencias de definición y ocurrencias de aplicación de los identificadores) y realizar la comprobación de tipos sobre el lenguaje Δ extendido; debe reportar la posición de *cada uno* de los errores contextuales detectados (esto es, avisar de *todos* los errores contextuales encontrados en el proceso).
- Las técnicas por utilizar son las expuestas en clase y en el libro de Watt y Brown. Recuerde las reglas contextuales que el profesor expuso en clase.

- Nos interesa que el analizador contextual deje el árbol sintáctico 'decorado' apropiadamente para su uso en la fase de generación de código subsiguiente (Proyecto #3)[19]. Esto es particularmente delicado para el procesamiento de los comandos repetitivos **`loop for`** y de las nuevas formas de declaración compuesta.

Como hemos indicado, ustedes deben basarse en los programas que les han sido facilitados como punto de partida. 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. Debe documentar clara y concisamente los siguientes puntos[20]:

Analizador contextual

- Describir la manera en que se comprueban los tipos para todas las variantes de los comandos **`loop`** … **`end`** que *no* sean variantes del comando **`loop_for`**.
- Describir la solución dada al manejo de alcance, del tipo y de la protección de la variable de control del comando **`loop_for_from_to_do_end`** y sus variantes condicionadas (**`while`** y **`until`**).
- Describir la solución dada al manejo de alcance, del tipo y de la protección de la variable de iteración del comando **`loop_for_in_do_end`**.
- Describir la solución creada para resolver *todas* las restricciones contextuales del comando **`select`** *Exp* **`from`** *Cases* **`end`**.
- Describir el procesamiento de la declaración de variable inicializada (**`var`** *Id* **`init`** *Exp*).
- Descripción de su validación de la unicidad[21] de los nombres de parámetros en las declaraciones de funciones o procedimientos.
- Describir la manera en que se procesa la declaración compuesta **`local`**. Interesa que la *primera* declaración (entre **`local`** e **`in`**) introduzca identificadores que son conocidos *privadamente* (*localmente*) por la segunda declaración (entre **`in`** y **`end`**). Al finalizar el proceso, se "exporta" *solamente* lo introducido por la *segunda* declaración.
- Describir el procesamiento de la declaración compuesta **`rec`**. Interesa la no-repetición de los identificadores de *función* o de *procedimiento* (`Proc-Funcs`) declarados por esa declaración compuesta y que estos identificadores sean conocidos en los *cuerpos* de todas las declaraciones de funciones o procedimientos (`Proc-Funcs`) que aparecen en una misma declaración compuesta **`rec`**.
- Solución dada al procesamiento contextual de todos los comandos repetitivos con nombre, **`loop`** Id_L **`…do_end`**.
- Solución dada al procesamiento de los comandos de escape **`leave`** y **`next`**, con y sin nombre.

---

[19] Esto comprende introducir información de tipos en los árboles de sintaxis abstracta (ASTs) correspondientes a expresiones y a variables, así como dejar "amarradas" las ocurrencias aplicadas de identificadores (poner referencias hacia el subárbol donde aparece la ocurrencia de definición correspondiente), vía la tabla de identificación [estos son los "punteros rojos" que hemos mencionado varias veces en clases]. También interesa determinar si un identificador corresponde a una variable, etc.; eso se sabrá según el tipo de subárbol de *declaración* que introdujo el identificador en el contexto.
[20] *Nada* en la documentación es opcional. La documentación tiene un peso importante en la calificación del proyecto.
[21] I.e. no-repetición. Sugerimos experimentar y entender cómo funciona el compilador de Δ original en ese aspecto.

- Solución dada al procesamiento del comando de retorno explícito de procedimiento **`return`**.
- Nuevas rutinas de análisis contextual, así como cualquier modificación a las existentes.
- Lista de nuevos errores contextuales detectados, con los nuevos mensajes de error.
- Plan de pruebas para validar el compilador. Debe separar las pruebas que validan la fase de análisis contextual, para no confundirlas con las pruebas dirigidas a validar los analizadores léxico y sintáctico. Debe incluir pruebas *positivas* (para confirmar funcionalidad del compilador al procesar programas correctos) y pruebas *negativas* (para evidenciar la capacidad del compilador para detectar errores). Debe especificar lo siguiente para cada caso de prueba[22]:
  - Objetivo del caso de prueba
  - Diseño del caso de prueba
  - Resultados esperados
  - Resultados observados
- En las pruebas del analizador contextual es importante comprobar que los componentes de análisis léxico y sintáctico siguen funcionando bien. Si hacen correcciones a ellos, deben documentar los cambios en un apéndice de la documentación de *este* Proyecto #2.
- Discusión y análisis de los resultados obtenidos. Conclusiones a partir de esto.
- Una reflexión sobre la experiencia de modificar fragmentos de un compilador/ambiente escrito por terceras personas.
- Descripción resumida de las tareas realizadas por cada miembro del grupo de trabajo.
- Indicar cómo debe compilarse su programa.
- Indicar cómo debe ejecutarse su programa.
- Archivos con el texto fuente de su compilador. El texto fuente debe incluir comentarios que indiquen con claridad los puntos en los cuales se han hecho modificaciones y cuál[es] persona[s] las realiz(ó | aron).
- Archivos con el código objeto del compilador. **El compilador debe estar en un formato ejecutable directamente desde el sistema operativo Windows[23], o llegar a un acuerdo alterno con el profesor.**
- Debe guardar su trabajo en una carpeta comprimida (formato **`.zip`**) según se indica abajo[24]. Esto debe incluir:
  - Documentación indicada arriba, con una portada donde aparezcan los nombres y números de carnet de los miembros del grupo. El documento debe 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 (o una alternativa, negociada con el profesor).
  - Programas (`.tri`) de entrada que han preparado para probar su analizador contextual.

Puntos extra

Los grupos de hasta *tres* miembros podrán obtener puntos extra por su procesamiento completo del comando **`select`**. Los grupos de *cuatro* miembros obligatoriamente deben procesar el comando **`select`** completo.

Entrega

Fecha límite: **2022.10.26**, antes de las 23:45. No se recibirán trabajos después de la fecha y la hora indicadas.

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`**. Para los grupos de *hasta* tres miembros el procesamiento del comando **`select`** y la variante **`loop for in`** es opcional y dará puntos extra. Los comandos repetitivos con nombre permitirán obtener puntos extra a los grupos, cualquiera que sea su cantidad de miembros.

---

[22] Puede usar como base los casos de prueba que publique el profesor.

[23] En principio, se permitirá entregar el trabajo en otro ambiente, pero debe avisar de previo al profesor y a los asistentes.

[24] **No use** formato **`.rar`**, porque es rechazado por el sistema de correo-e del TEC. Si usa **`.rar`** obtendrá un **`0`** como calificación.

Deben enviar por correo-e el ***enlace***[25] 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 2 carnet carnet carnet carnet**

El asunto (*subject*) de su mensaje debe ser: **IC-5701 Proyecto 2 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.

[25] 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 31 de enero del 2023.