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:
- 2021.05.05: Versión base (v0).
- 2021.05.20: Fecha de entrega cambiada a **lunes 2021.05.31**.

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 terminado de comprender 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*. Ud. deberá extender el compilador del lenguaje Δ (escrito en Java), desarrollado por Watt y Brown, de manera que sea capaz de procesar el lenguaje descrito en el enunciado del Proyecto #1 y en la sección *Lenguaje fuente* que aparece abajo. Su compilador será la modificación de uno existente, el cual debe transformar en uno capaz de procesar el lenguaje Δ extendido conforme las características contextuales (identificación y tipos) especificadas en este documento. 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.

Base

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 (@ tecDigital), para que Ud. los descargue y estudie, a fin de modificarlos conforme a lo especificado en los proyectos del curso. En el 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 estudiantes Pablo Navarro y Jimmy Fallas. Si desea aprender acerca de cómo integrar el IDE y el compilador, siga las indicaciones preparadas por nuestro ex-asistente, 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, 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 técnicas por aplicar, el lenguaje fuente original (Δ, que vamos a extender) 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. 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 otros compañeros como base para esta tarea programada*, ***pero deberá pedir la autorización de reutilización a sus compañeros y darles créditos explícitamente*** en la documentación del proyecto que presenta el equipo del cual Ud. es miembro[2].

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

Sintaxis
Remítase a la especificación del Proyecto #1 ('`IC-5701 2021-1 Proyecto 1 v2.pdf`').

Contexto: identificación y tipos
Usaremos las reglas sintácticas del lenguaje Δ extendido para indicar las restricciones de contexto. Interprete las 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 árboles de sintaxis abstractas de las clases sintácticas indicadas[3]:

• *V*: `V-name`	• PF_1, PF_2, PF_i, PF_n : `Proc-Func`
• *Vn*: `Var-name`	• *PFs* : `Proc-Func*`
• *Id*, *N*, Id_1, Id_2, *Idp*, *Idq*, *Idr* : `Identifier`	• *Cases* : `Cases`
• *R*: `Long-Identifier`	• *C* : `Case`
• *Exp*, Exp_1, Exp_2, Exp_3, Exp_i : `Expression`	• *Cs* : `Case*`
• *Com*, Com_1, Com_2 : `Command`	• *Cl*, Cl_1, Cl_2 : `Case-Literal`
• *Dec*, Dec_1, Dec_2, Dec_3 : `Declaration`	• *Cls* : `Case-Literal*`
• *TD* : `Type-denoter`	• *Il* : `Integer-Literal`
• *FPS* : `Formal-Parameter-Sequence`	• *Cl* : `Character-Literal`
• *APS* : `Actual-Parameter-Sequence`	

Expresiones
No hay cambios en las expresiones, salvo en lo concerniente a los accesos a variables y las llamadas a funciones declaradas dentro de paquetes. Ver en la sub-sección relacionada con *acceso cualificado a entidades declaradas en paquetes*.

Comandos
El comando **`nothing`** no requiere de revisión contextual, pues no depende de tipos ni de declaraciones de identificadores.

Para el *comando* de asignación, *V* **`:=`** *Exp*, las restricciones contextuales son:

- Si *V* es de la forma *Idp***`$`***Vn*, *Idp* debe corresponder a un identificador de paquete declarado *antes* de la ocurrencia de *V*. *Vn* debe iniciar con un identificador Id_v declarado dentro del paquete *Idp*. Dentro del paquete *Idp*, Id_v debe haber sido declarada como una *variable*[4]. [Ver la sub-sección relacionada con *acceso cualificado a entidades declaradas en paquetes*.]
- Si *V* es de la forma *Vn*, aplican las restricciones del lenguaje Δ original.
- *Exp* debe tener el mismo tipo que se declaró o se infirió para la variable *Vn*.

Considere estos comandos repetitivos añadidos a Δ:

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

[3] Usamos los subíndices de manera liberal.
[4] Recuerde que tenemos *dos* formas de declaración de variables: variable con tipo y variable inicializada. También los parámetros formales **`var`** declaran variables. Además, las variables pueden aparecer declaradas dentro de un paquete.

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

Las restricciones contextuales son:

- *Exp* debe ser de tipo `Boolean`.
- *Com* y sus partes deben satisfacer las restricciones contextuales[5].

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 mismo que rige el comando **loop_for_from_to_do_end** dentro del cual 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**[6] en la invocación de un procedimiento o función dentro del cuerpo del comando repetitivo **loop_for_from_to_do_end**[7].
- Este comando repetitivo *funciona como un bloque* al declarar una variable local (la variable de control). Las reglas usuales de anidamiento aplican aquí (si se re-declara la variable en un comando, en una expresión, en una lista de parámetros o en cualquier bloque anidado, esta es *distinta* y hace *inaccesible* la variable de control declarada en el comando **loop_for_from_to_do_end**, que sería 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*.

El comando condicional de Δ extendido fue modificado solamente en forma sintáctica. Considere:

if *Exp* **then** Com_1 **(elsif** 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

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 **choose** *Exp* **from** *Cases* **end**

- La expresión selectora *Exp* debe ser de tipo entero (`Integer`) o de tipo carácter (`Char`).

---

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

- Todas las literales de un mismo **`choose`** deben ser del mismo tipo que el de la expresión selectora[8].
- Cada literal entera *Il* o literal carácter *Cl* debe aparecer *una sola vez* como caso dentro de un *mismo* **`choose`**.[9]
- Los **`choose`**s pueden anidarse y sus literales tienen *alcance*; una literal de un **`choose`** externo puede *reaparecer* dentro de un **`choose`** *anidado* y esto *no* es un error.
- Todos los comandos subordinados (que aparecen dentro del **`choose`**) deben satisfacer las restricciones contextuales[10].

Declaraciones

Los parámetros de una abstracción (*función* o *procedimiento*) forman parte de un nivel léxico (alcance) inmediatamente más profundo que aquel en el cual aparece el identificador de la abstracción que los declara[11]. *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* **`:=`** *Exp*

Aquí, el tipo de la *variable* inicializada *Id* debe inferirse a partir del tipo de la expresión inicializadora *Exp*[12]. *Id* se "exporta" al resto del bloque donde aparece esta declaración.

En una declaración **`private`** 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`)[13].

En una declaración **`recursive`** *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: **`recursive`** PF_1 **`|`** PF_2 **`end`**. Allí se declaran funciones y(o) procedimientos mutuamente recursivos: el identificador[14] declarado en PF_1 es conocido por PF_2, y viceversa.
- Los nombres 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 mayor que el de los nombres de la función o procedimiento que les corresponden.
- Se "exportan"[15] los identificadores de función o de procedimiento declarados en PF_1 y PF_2. Tanto en PF_1 como 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: **`recursive`** 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 son conocidos al procesar los cuerpos de *todas* las funciones o procedimientos declarados en las PF_i.

---

[8] Si la expresión selectora es de tipo entero, entonces las literales que aparecen en los casos deben ser literales enteras. Si la expresión selectora es de tipo carácter, entonces las literales que aparecen en los casos deben ser caracteres (literales).

[9] Tengan particular cuidado con los valores comprendidos en un sub-rango Cl_1 **`..`** Cl_2. Ese sub-rango comprende todos los valores que están en el conjunto $\{ lit : Literal \mid Cl_1 \leqslant lit \leqslant Cl_2 \}$. *Literal* puede corresponder a `Integer` o a `Char`.

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

[11] Es decir, los parámetros 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, así como lo discutido en clases.

[12] El procesamiento es semejante al que se hace para la declaración **`const`** *Id* **`~`** *Exp*. La diferencia es que, en el caso que nos ocupa, se está declarando a *Id* 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 una invocación a un procedimiento o a una función.

[13] Revise lo explicado en clases.

[14] Ese identificador da nombre a la función o al procedimiento.

[15] *Después* de la declaración **`recursive`** los identificadores de procedimiento o de función serán visibles y podrán ser utilizados por declaraciones subsecuentes.

- Es un error declarar más de una vez el mismo identificador[16] en las declaraciones simples que componen una misma declaración compuesta **`recursive`**.
- Todos los identificadores de procedimiento o de función declarados vía **`recursive`** son "exportados"; esto es, son agregados al contexto subsiguiente y son conocidos después de la declaración compuesta **`recursive`**[17].
- Funciones o procedimientos *distintos* declarados vía **`recursive`** pueden declarar ***parámetros*** con nombres idénticos, pero en listas de parámetros distintas. Eso *no* es problema.

Una declaración de la forma **`package`** *Id* ~ *Dec* **`end`** se elabora como sigue. *Id* se asocia con un paquete de entidades declaradas en *Dec*. Las entidades empacadas pueden incluir constantes, variables, tipos, procedimientos o funciones (los paquetes *no se anidan* y eso lo garantiza la especificación sintáctica de Δ extendido). Una entidad declarada con nombre *N* dentro de un paquete con nombre *Id* es conocida con el nombre *Id*`$`*N* fuera del paquete. Los paquetes son espacios de nombres, no son clases de objetos; sólo hay *una* instancia de cada variable declarada en un paquete. El nombre *Id* del paquete cualifica los nombres *N* de las entidades declaradas en *Dec*; sintácticamente, *Idp*`$`*N* es un `Long-Identifier` e *Idp***`$`***Vn* es un `V-name` (y *Vn* es un `Var-name`).

Paquetes

Los paquetes se procesan en secuencia. Una vez procesada una declaración de paquete, los identificadores declarados allí quedarán a disposición de los paquetes subsecuentes o del "comando principal" que debe venir después de la declaración del último paquete.

Considere un programa como:

`package` *Idp* ~ Dec_p **`end;`**
`package` *Idq* ~ Dec_q **`end;`**
Com

Dentro de *Idq* – es decir, Dec_q – se puede utilizar cualquier identificador Id_1 exportado por la declaración Dec_p del paquete *Idp* haciendo la cualificación de esta forma: *Idp*$\$Id_1$. Dentro de *Com* se pueden acceder los identificadores declarados dentro de *Idp* o de *Idq*, de manera cualificada mediante los prefijos correspondientes; por ejemplo: *Idp*$\$Id_1$, *Idq*$\$Id_2$. En Dec_p o Dec_q no es necesario cualificar los identificadores declarados dentro de su propio paquete (*Idp* e *Idq* respectivamente); por ejemplo, si *N* fue declarado dentro de Dec_p, dentro del paquete *Idp* podemos accederlo usando *N* simplemente, aunque también debe ser posible accederlo vía *Idp*$\$N$.

Ud. debe *exigir* que todos los paquetes que anteceden al "comando principal" *Com*, del programa, tengan nombres distintos.

Acceso cualificado a entidades declaradas en paquetes

Acceso a variables y constantes declaradas en paquetes

Los accesos cualificados a constantes o variables *V* se dan cuando *V* es de la forma *Idp***`$`***Vn*, conforme lo permite la regla sintáctica modificada para Δ extendido:

```
V-name ::= [ Package-Identifier "$" ] Var-name
```

Recuerde que esa regla sintáctica especifica *dos* formas, pues lo encerrado entre [y] es opcional:

```
V-name ::= Package-Identifier "$" Var-name
V-name ::= Var-name
```

Aquí nos interesa el caso en que *sí* aparece el identificador de paquete.

En lo que sigue, suponga:

- *V*: `V-name`

[16] De función o de procedimiento.
[17] Esto no sucede si la declaración **`recursive`** aparece dentro de una declaración **`private`** *antes* del **`in`**.

- *Id$_v$*: `Identifier`,
- *Idp*: `Identifier`
- *Vn*: `Var-name`
- *V* tiene la forma *Idp***`$`***Vn*
- *Vn* inicia con *Id$_v$*

Conforme a la sintaxis de Δ extendido, los accesos cualificados pueden ocurrir en estas situaciones:

No terminal (regla sintáctica)	Forma sintáctica
`single-Command`	`V-name` **`:=`** `Expression`
`primary-Expression`	`V-name`
`Actual-Parameter`	**`var`** `V-name`

En general, si *V* es de la forma *Idp***`$`***Vn*, las restricciones contextuales son:

- *Idp* debe corresponder a un identificador de paquete declarado *antes* de la ocurrencia de *V*.
- *Vn* debe iniciar con un identificador *Id$_v$* declarado dentro del paquete *Idp*. Dentro del paquete *Idp*, *Id$_v$* debe haber sido definido por alguna de estas formas de declaración:
  - **`const`** `Identifier` **`~`** `Expression`
  - **`var`** `Identifier` **`:`** `Type-denoter`
  - **`var`** `Identifier` **`:=`** `Expression`

Para el *comando* de asignación, *V* **`:=`** *Exp*, las restricciones contextuales son:

- Si *V* es de la forma *Idp***`$`***Vn*, *Idp* debe corresponder a un identificador de paquete declarado *antes* de la ocurrencia de *V*. *Vn* debe iniciar con un identificador *Id$_v$* declarado dentro del paquete *Idp*. Dentro del paquete *Idp*, *Id$_v$* debe haber sido declarada como una *variable*[18].
- *Exp* debe tener el mismo tipo que se declaró o se infirió para la variable *Vn.*

En una expresión de acceso a variable, *V*, las restricciones contextuales son:

- Si *V* es de la forma *Idp***`$`***Vn*, *Idp* debe corresponder a un identificador de paquete declarado *antes* de la ocurrencia de *V*. *Vn* debe iniciar con un identificador *Id$_v$* declarado dentro del paquete *Idp*. Dentro del paquete *Idp*, *Id$_v$* debe haber sido declarada como una *variable* o una *constante*[19].
- *V* tendrá el mismo tipo que se declaró o se infirió para el acceso a la variable *Vn.*

En un parámetro real ('actual parameter'), **`var`** *V*, las restricciones contextuales son:

- Si *V* es de la forma *Idp***`$`***Vn*, *Idp* debe corresponder a un identificador de paquete declarado *antes* de la ocurrencia de *V*. *Vn* debe iniciar con un identificador *Id$_v$* declarado dentro del paquete *Idp*. Dentro del paquete *Idp*, *Id$_v$* debe haber sido declarada como una *variable*[20].
- *V* tendrá el mismo tipo que se declaró o se infirió para la variable *Vn*.

Llamadas a funciones o procedimientos declarados en paquetes

Las llamadas a funciones y procedimientos *Idr*, declarados dentro de un paquete *Idp*, son permitidas mediante accesos *cualificados R* de la forma *Idp***`$`***Idr*.

En el Proyecto #1 añadimos esta regla sintáctica (a Δ extendido):

```
Long-Identifier ::= [ Package-Identifier "$" ] Identifier
```

[18] Recuerde que tenemos *dos* formas de declaración de variables: variable con tipo y variable inicializada. También los parámetros formales **`var`** declaran variables. Además, las variables pueden aparecer declaradas dentro de un paquete.

[19] Recuerde que tenemos *dos* formas de declaración de variables: variable con tipo y variable inicializada. También los parámetros formales **`var`** declaran variables. Además, las variables pueden aparecer declaradas dentro de un paquete.

[20] Recuerde que tenemos *dos* formas de declaración de variables: variable con tipo y variable inicializada. También los parámetros formales **`var`** declaran variables. Además, las variables pueden aparecer declaradas dentro de un paquete.

Recuerde que esa regla sintáctica especifica *dos* formas, pues lo encerrado entre [y] es opcional:

```
Long-Identifier ::= [ Package-Identifier "$" ] Identifier
Long-Identifier ::= Identifier
```

Aquí nos interesa el caso en que *sí* aparece el identificador de paquete.

Además, modificamos la forma de las llamadas a procedimientos en `single-Command` y a funciones en `primary-Expression`.

En lo que sigue, suponga:

- *Idr*: `Identifier`,
- *Idp*: `Identifier`,
- *R*: `Long-Identifier`
- *FPS* : `Formal-Parameter-Sequence`
- *APS* : `Actual-Parameter-Sequence`

Conforme a la sintaxis de Δ extendido, las llamadas cualificadas pueden ocurrir en estas situaciones:

No terminal (regla sintáctica)	Forma sintáctica
`single-Command`	`Long-Identifier ( Actual-Parameter-Sequence )`
`primary-Expression`	`Long-Identifier ( Actual-Parameter-Sequence )`

En general, si en una llamada *R*(*APS*), *R* es de la forma *Idp***$***Idr*, las restricciones contextuales son:

- *Idp* debe corresponder a un identificador de paquete declarado *antes* de la ocurrencia de *R*.
- *Idr* debe haber sido definido dentro del paquete *Idp*., mediante alguna de estas formas de declaración:
  - **proc** `Identifier (` *FPS* `) ~ Command` **end**
  - **func** `Identifier (` *FPS* `): Type-denoter ~ Expression`
- Si la llamada se da en un comando, *Idr* debe haber sido declarado como procedimiento (**proc**) y la correspondencia entre los parámetros formales (*FPS*) y los reales (*APS*) debe ser correcta en cuanto a la cantidad de parámetros, el modo del paso de parámetros y los tipos.
- Si la llamada se da en una expresión, *Idr* debe haber sido declarado como función (**func**) y la correspondencia entre los parámetros formales (*FPS*) y los reales (*APS*) debe ser correcta en cuanto a la cantidad de parámetros, el modo del paso de parámetros y los tipos. El tipo de la expresión *Idp***$***Idr* (*APS*) será el del denotador de tipo del resultado (`Type-denoter`) declarado para *Idr* dentro del paquete *Idp* vía la forma **func**.

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 la fase de generación de código subsiguiente (Proyecto #3)[21]. Esto es particularmente delicado para el procesamiento de la variante **for** del comando **loop**, las nuevas formas de declaración compuesta y el acceso a entidades declaradas dentro de los paquetes.

[21] Esto comprende introducir información de tipos en los árboles de sintaxis abstracta (ASTs) correspondientes a expresiones, 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. También interesa determinar si un identificador corresponde a una variable, etc.

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 el procesador en Java usado como base, 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
Debe documentar clara y concisamente los siguientes puntos[22]:

Analizador contextual

- Describir la manera en que se comprueban los tipos para todas las variantes de **`loop`** … **`end`**.
- Describir la solución dada al manejo de alcance, del tipo y de la protección de la variable de control del **`loop for`** … **`end`**.
- Describir la solución creada para resolver las restricciones contextuales del comando **`choose`** *Exp* **`from`** *Cases* **`end`**.
- Describir el procesamiento de la declaración de variable inicializada (**`var`** *Id* **`:=`** *Exp*).
- Descripción de su validación de la unicidad[23] de los nombres de parámetros en las declaraciones de funciones o procedimientos.
- Describir la manera en que se procesa la declaración compuesta **`private`**. Interesa que la primera declaración (antes del **`in`**) introduzca identificadores que son conocidos *privadamente* (*localmente*) por la segunda declaración (después del **`in`**); al finalizar, se "exporta" solamente lo introducido por la segunda declaración.
- Describir el procesamiento de la declaración compuesta **`recursive`**. Interesa la no-repetición de los identificadores *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 **`recursive`**.
- Describir cómo hacen el manejo de las declaraciones **`package`** y el acceso a las entidades allí declaradas, desde dentro o desde fuera del paquete.
- Nuevas rutinas de análisis contextual, así como cualquier modificación a las existentes.
- Lista de nuevos errores contextuales detectados.
- 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 con datos correctos) y pruebas *negativas* (para evidenciar la capacidad del compilador para detectar errores). Debe especificar lo siguiente para cada caso de prueba:
  - 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.

[22] *Nada* en la documentación es opcional. La documentación tiene un peso importante en la calificación del proyecto.
[23] I.e. no-repetición.

- 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.
- Archivos con el código objeto del compilador. **El compilador debe estar en un formato ejecutable directamente desde el sistema operativo Windows**[24] **o llegar a un acuerdo alterno con el profesor.**
- Debe guardar su trabajo en una carpeta comprimida (formato **.zip**) según se indica abajo[25]. Esto debe incluir:
  - Documentación indicada arriba, con una portada donde aparezcan los nombres y números de carnet de los miembros del grupo. Los documentos descriptivos deben estar en formato `.pdf`.
  - Código fuente, organizado en carpetas.
  - Código objeto. Recuerde que el código objeto de su compilador (programa principal) debe estar en un formato directamente ejecutable en Windows (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 **`choose`**. Los grupos de *cuatro* miembros obligatoriamente deben procesar el comando **`choose`** completo.

Entrega

Fecha límite: ~~**jueves 2021.05.27**~~**lunes 2021.05.31**, antes de las 23:55. 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***[26] 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).

> El asunto (subject) 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 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.

---

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

[25] **No use** formato **.rar**, porque es rechazado por el sistema de correo-e del TEC.

[26] 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.