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:

- 2019.04.24: Versión base (v0). Precedido por comunicaciones y ejemplos en clase.

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 de un compilador escrito "a mano" usando 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, de manera que sea 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

Ud. usará como base 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. Los compiladores e intérpretes han sido ubicados en la carpeta 'Recursos' del curso, para que Ud. los descargue, pueda estudiarlos y modificarlos. 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. Siga también las indicaciones preparadas por nuestro ex-asistente, Diego Ramírez Rodríguez, en cuanto a las partes del compilador que debe desactivar para poder desarrollar este trabajo, así como los ajustes necesarios para que el IDE y el compilador trabajen 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.

¡Lea los apéndices B y D del libro de Watt y Brown, así como el código del compilador para comprender el lenguaje fuente original 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 ***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. El usuario seleccionará cuál es el archivo que contiene el texto del programa fuente desde el ambiente de programación (IDE) base suministrado, o bien lo editará en la ventana que el IDE provee para el efecto (que también permite guardarlo de manera persistente). Los archivos fuente deben tener terminación `.tri`.

---

[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.

Lenguaje fuente

Sintaxis
Remítase a la especificación del Proyecto #1 ('IC-5701 2019-1 Proyecto 1 v3').

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 sintácticos de las clases indicadas[3]:

• *V*: `V-name`	• *Cases* : `Cases`
• *Id, N, Id₁, Id₂, Id_p, Id_q* : `Identifier`	• *C* : `Case`
• *Exp, Exp₁, Exp₂, Exp₃, Exp_i* : `Expression`	• *Cs* : `Case*`
• *Com, Com₁, Com₂* : `Command`	• *Cl, Cl₁, Cl₂* : `Case-Literal`
• *Dec, Dec₁, Dec₂, Dec₃* : `Declaration`	• *Cls* : `Case-Literal*`
• *PF₁, PF₂, PF_i, PF_n* : `Proc-Func`	• *Il* : `Integer-Literal`
• *PFs* : `Proc-Func*`	• *Cl* : `Character-Literal`
• *FPS* : `Formal-Parameter-Sequence`	• *TD* : `Type-denoter`

El comando **`pass`** no requiere de revisión contextual, pues no depende de tipos ni de declaraciones de identificadores.

Considere el *comando* de asignación[4], *V* **`:=`** *Exp*. Las restricciones contextuales son:
- *V* debe haber sido declarada como una *variable*[5].
- *Exp* debe tener el mismo tipo que se declaró o se infirió para la variable *V*.

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:
- *Exp* debe ser de tipo `Boolean`.
- *Com* y sus partes deben satisfacer las restricciones contextuales[6].

En el comando de repetición controlada por contador

`loop for` *Id* **`from`** *Exp₁* **`to`** *Exp₂* **`do`** *Com* **`end`**

las restricciones son:
- *Exp₁* y *Exp₂* deben ser ambas de tipo *entero*. Los tipos de *Exp₁* y *Exp₂* 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 *Exp₁* ***ni*** por *Exp₂*.

---

[3] Usamos los subíndices de manera liberal.
[4] Esta es una asignación. No la confunda con la *declaración* de variable inicializada.
[5] Recuerde que tenemos *dos* formas de declaración de variables: variable con tipo y variable inicializada. También los parámetros formales **`var`** están declarando variables.
[6] Observe que ésta, como muchas de las restricciones contextuales, se formulan de manera recursiva. Es decir, 'están bien' en todos sus niveles de anidamiento.

- *Com* debe cumplir con las restricciones contextuales. Su contexto es el del comando **loop_for_from_to_do_end** en que aparece *Com*, enriquecido por la variable de control *Id*.
- La variable de control *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_from_to_do_end**[8].
- 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 **loop_for_from_to_do_end**, que es 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 condicional de Δ extendido fue modificado solo sintácticamente. La expresión *Exp* debe ser booleana. Com_1 y Com_2, así como sus partes, deben satisfacer las restricciones contextuales

if *Exp* **then** Com_1 **else** Com_2 **end**

El comando

let `Declaration` **in** `Command` **end**

se analiza contextualmente de manera idéntica al comando correspondiente en Δ original (el original no tiene **end,** recuerde que ahora permitimos `Command` en lugar del `single-Command` original).

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`).
- Todas las literales de un mismo **choose** deben ser del mismo tipo que el de la expresión selectora[9].
- Cada literal entera *Il* o literal carácter *Cl* debe aparecer *una sola vez* como caso dentro de un *mismo* **choose**.[10]
- 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[11].

Los parámetros de una abstracción (función o procedimiento) forman parte de un nivel léxico (alcance) inmediatamente más profundo que aquel en el que 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* **::=** *Exp*

---

[7] No podrá pasarse por referencia.
[8] En clase discutimos que, en el cuerpo de este comando repetitivo (*Com*), la 'variable' de control *Id* en realidad se comporta como una *constante*.
[9] 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).
[10] 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`.
[11] Es decir, las restricciones contextuales deben cumplirse *recursivamente* en los comandos, expresiones y declaraciones subordinados.
[12] 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, así como el código del compilador de Δ original en Java.

Aquí, el tipo de la *variable* inicializada *Id* debe deducirse a partir del tipo de la expresión inicializadora *Exp*[13]. *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`)[14].

En una declaración **`par`** Dec_1 **`|`** Dec_2 **`end`**, el identificador declarado en Dec_1 no es conocido por Dec_2, ni viceversa. Las declaraciones compuestas introducidas con **`par`** son denominadas 'paralelas' o 'colaterales'. Si la declaración compuesta **`par`** Dec_1 **`|`** Dec_2 **`end`** tiene *Amb* como contexto, ese mismo *Amb* es el contexto para elaborar tanto Dec_1 como Dec_2. Además, Dec_1 y Dec_2 deben declarar identificadores distintos[15]. Observe que todas las declaraciones paralelas son simples (`single-Declaration`). Se "exportan" todos los identificadores declarados en Dec_1 y Dec_2. Es posible declarar paralelamente dos o más declaraciones simples; lo descrito antes corresponde al caso de ***dos*** declaraciones y *debe ser generalizado a más de dos declaraciones*: **`par`** Dec_1 **`|`** … **`|`** Dec_n **`end`** ($n \geq 2$).

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

- Consideremos primero el caso de *dos* declaraciones: en **`recursive`** PF_1 **`|`** PF_2 **`end`** se declaran funciones y(o) procedimientos mutuamente recursivos: el identificador[16] declarado en PF_1 es conocido por PF_2, y viceversa.
- PF_1 y PF_2 deben declarar identificadores (de función o procedimiento) *distintos*.
- Se "exportan"[17] los identificadores de función o de procedimiento declarados en PF_1 y PF_2. Los *parámetros* 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 *procedimiento* introducidos por las declaraciones PF_i ($1 \leq i \leq n$) deben ser *distintos* y son conocidos al 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[18] en las declaraciones simples que componen una misma declaración compuesta **`recursive`**.
- Todos los identificadores de procedimiento o función declarados vía **`recursive`** son "exportados"; esto es, son agregados al contexto subsiguiente y son conocidos después de la declaración compuesta **`recursive`**[19].
- 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

---

[13] 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.

[14] Revise lo explicado en clases.

[15] Es un error declarar el mismo identificador en varias declaraciones que componen una declaración **`par`**.

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

[17] *Después* de la declaración **`recursive`** los identificadores podrán ser utilizados.

[18] De función o procedimiento.

[19] Esto no sucede cuando **`recursive`** aparece dentro de una declaración **`private`** (antes del **`in`**).

paquete. El nombre *Id* del paquete cualifica los nombres *N* de las entidades declaradas en *Dec*; sintácticamente, *Id*`$`*N* es un `Long-Identifier`.

Los paquetes se procesan en secuencia. Considere un programa como:

`package` Id_p ~ Dec_p **`end;`**
`package` Id_q ~ Dec_q **`end;`**
Com

Dentro de Id_q – es decir, Dec_q – se puede utilizar cualquier identificador Id_1 exportado por la declaración Dec_p del paquete Id_p haciendo la cualificación de esta forma: $Id_p\$ Id_1$. Dentro de *Com* se pueden acceder los identificadores declarados dentro de Id_p o de Id_q, con los prefijos correspondientes; por ejemplo: $Id_p\$Id_1$, $Id_q\$Id_2$. En Dec_p o Dec_q no es necesario cualificar los identificadores declarados dentro de ellos mismos; por ejemplo, si *N* fue declarado dentro de Dec_p, podemos accederlo usando *N* simplemente, aunque también debe ser posible accederlo vía $Id_p\$N$.

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

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 los libros 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)[20]. Esto es particularmente delicado para el procesamiento de las variantes **`for`** del comando **`loop`** y las nuevas formas de declaración compuesta.

Como se indicó, ustedes deben basarse en los programas que se le dan 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 han introducido modificaciones ustedes.

Debe dar crédito por escrito a cualquier otra fuente de información o ayuda.

Debe ser posible activar la ejecución del IDE de su compilador desde el Explorador de Windows haciendo clics sobre el ícono de su archivo .jar, o bien generar un .exe a partir de su .jar. *Por favor indique claramente cuál es el archivo ejecutable del IDE, para que el profesor o su asistente puedan someter a pruebas su programa sin dificultades.*

Documentación
Debe documentar clara y concisamente los siguientes puntos[21]:

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`**.

[20] 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 se determina si un identificador corresponde a una variable, etc.
[21] *Nada* en la documentación es opcional. La documentación tiene un peso importante en la calificación del proyecto.

- 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[22] 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 introduzca identificadores que son conocidos *privadamente* (*localmente*) por la segunda declaración; se "exporta" solamente lo introducido por la segunda declaración.
- Describir la manera en que se procesa la declaración compuesta **`par`**. Interesa la no-repetición de los identificadores introducidos por cada declaración colateral. Los identificadores declarados en una misma declaración compuesta **`par`** se "exportan" para ser conocidos en el resto del bloque donde aparece el **`par`**.
- Describir el procesamiento de la declaración compuesta **`recursive`**. Interesa la no-repetición de los identificadores (de función o procedimiento) declarados y que estos identificadores sean conocidos en los *cuerpos* de las funciones o procedimientos declarados en una misma declaración compuesta **`recursive`**.
- Nuevas rutinas de análisis contextual, así como cualquier modificación a las existentes.
- Lista de errores contextuales detectados.
- Plan de pruebas para validar el compilador. Debe separar las pruebas para la fase de análisis contextual. 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
- Análisis de la 'cobertura' del plan de pruebas (interesa que valide tanto el funcionamiento "normal" como la capacidad de detectar errores contextuales).
- 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 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.
- Archivos con el código objeto del compilador. **El compilador debe estar en un formato ejecutable directamente desde el sistema operativo Windows.**
- Debe enviar su trabajo en un archivo comprimido (formato **.zip**) según se indica abajo[23]. Esto debe incluir:
  - Documentación indicada arriba, con una portada donde aparezcan los nombres y números de carnet de los miembros del grupo. Los documentos descriptivos deben estar en formato .pdf.
  - Código fuente, organizado en carpetas.
  - Código objeto. Recuerde que el código objeto de su compilador (programa principal) debe estar en un formato directamente ejecutable en Windows.
  - Programas (.tri) de prueba que han preparado.

Entrega

Fecha límite: **viernes 2019.05.10** 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.

---

[22] I.e. no-repetición.

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

Debe enviar por correo-e el ***enlace***[24] a un archivo comprimido almacenado en la nube con todos los elementos de su solución a estas direcciones: itrejos@itcr.ac.cr y dramirez1705@gmail.com (Diego Ramírez Rodríguez, nuestro asistente). Asegúrese de dar los permisos de lectura y descarga suficientes para el profesor y su asistente.

> El asunto (subject) debe ser:
> "IC-5701 - Proyecto 1 - " <carnet> " + " <carnet> " + " <carnet> " + " <carnet>.
>
> Los carnets deben ir ordenados ascendentemente.

Si su mensaje no tiene el asunto en la forma correcta, su proyecto será castigado con −10 puntos; podría darse el caso de que su proyecto no sea revisado del todo (y sea calificado con 0) sin responsabilidad alguna del profesor (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.

Estén atentos a recibir notificación respecto de posible entrega alternativa de trabajos vía el tecDigital.

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] 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 envían el hipervínculo al profesor y a su asistente mediante un mensaje de correo con el formato indicado.