Trinity

Analizador sintáctico

Versión 1 - 2 de octubre de 2014

Índice general

---

1 Introducción

Esta sección es informativa.

1. Este documento especifica los requerimientos para la segunda entrega del intérprete para el lenguaje de programación *Trinity*.

2. Todas las versiones publicadas de este documento, incluyendo la más reciente y vigente, se encontrarán siempre disponibles en la página de la práctica, en http://ldc.usb.ve/~05-38235/cursos/CI3725/2014SD/.

3. La especificación del lenguaje de programación *Trinity* **no** es parte de este documento — la definición del lenguaje se encuentra en un documento separado. Solo una parte de lo especificado en *ese* documento debe ser implantada para la entrega especificada en *este* documento. La especificación del lenguaje está disponible en la página del curso.

4. Los requerimientos establecidos por este documento se dividen en dos secciones: los *requerimientos generales*, que aplican para el proyecto en general y en todas sus etapas, y los *requerimientos específicos*, que aplican únicamente para la etapa específica correspondiente a este documento.

2 Requerimientos generales

Esta sección es normativa.

1. En este curso, Ud. debe escribir un intérprete para el lenguaje de programación *Trinity* según lo especificado en el documento de especificación del lenguaje, este documento, y los documentos de requerimientos para las demás entregas del proyecto.
2. El proyecto se desarrollará en cuatro partes, a saber:
    1. El analizador lexicográfico,
    2. el analizador sintáctico,
    3. la tabla de símbolos y el análisis estático, y
    4. la ejecución de programas en forma interpretada.
3. Debe desarrollar su proyecto usando **uno** de los tres conjuntos de herramientas seleccionados para el curso:
    1. El lenguaje de programación *Haskell*, junto con el generador de analizadores lexicográficos *Alex* y el generador de analizadores sintácticos *Happy*.
    2. El lenguaje de programación *Ruby*, junto con el generador de analizadores sintácticos *Racc*.
    3. El lenguaje de programación *Python*, junto con el generador de analizadores sintácticos *PLY*.

    Una vez seleccione un conjunto de herramientas para realizar su proyecto, deberá realizar las entregas de los demás en el mismo. No es factible y no se aceptará que se combinen herramientas ni que cambien sus herramientas una vez realizada la primera entrega.
4. Las entregas deben realizarse vía correo electrónico a los encargados de la práctica en un archivo adjunto de nombre `CI3725-2014SD-n-gg.tar.gz`, donde `gg` es su número de grupo con dos dígitos decimales, y `n` es el número de la etapa (entre 1 y 4). El archivo deberá contener un directorio con todo lo necesario para compilar su proyecto y ejecutarlo. El número de carné debe escribirse con todos sus dígitos consecutivos; por ejemplo, `0538235`.

5. No incluya en su entrega archivos producidos por herramientas generadoras de código ni compiladores que Ud. utilice para su proyecto, sino los archivos de fuente que Ud. escribió para suministrarle a esas herramientas.

6. Su proyecto debe incluir un `Makefile` que permita generar todo lo necesario para su ejecución al ejecutar el comando `make` en el directorio contenido en su entrega. Si Ud. elige realizar su proyecto en *Haskell*, se acepta y se recomienda que utilice *Cabal* en vez de Make para especificar lo necesario para que su proyecto compile.

7. Es **deseable** que su proyecto funcione sin modificaciones en las computadoras del Laboratorio Docente de Computación. Si su proyecto requiere alguna dependencia adicional por alguna situación excepcional, debe indicar las razones claramente al realizar su entrega y justificarlo en la documentación de su proyecto.

8. Una vez efectuada la generación de código y compilación correspondiente a su proyecto, éste debe poder ejecutarse con la orden `./trinity archivo.ty` desde el directorio raíz de su proyecto, donde `archivo.ty` será la ruta (relativa o absoluta) de un archivo de entrada para su proyecto. En caso de que utilice *Cabal* para compilar su proyecto usando las herramientas de *Haskell*, es esperado y aceptable que el ejecutable se genere en las ubicaciones usuales de esa herramienta, y no es necesario que pueda ejecutarse con ese comando exacto.

9. Es un hecho común y natural del diseño y la implantación de lenguajes de programación que el diseño se modifique durante el desarrollo para adaptarse al contexto en el cual el lenguaje se utilizará, el cual es, en este caso, el curso. En función de esto, tanto el documento de especificación del lenguaje como la especficación de cada etapa y entrega estarán sujetos a cambios y aclaratorias. Los encargados de la práctica le notificarán de tales cambios y aclaratorias y actualizarán los documentos relevantes siempre que ocurran tales modificaciones a la especificación original.

10. Se acostumbra a hacer públicas las respuestas a consultas hechas por el equipo de la práctica referentes a la especificación del lenguaje y a las condiciones de entrega de cada etapa, lo cual beneficia al resto de los alumnos. Si Ud. hace una consulta y no desea que se publique, notifíquelo al hacer su consulta. **Todas las consultas que se publiquen serán anónimas.**

11. Deberá enviar sus entregas antes de la medianoche al final del día especificado para las entregas de cada etapa del proyecto, en hora legal de Venezuela. Las entregas se realizarán siempre los viernes de la semana indicada para cada etapa. Cualquier modificación al cronograma de entregas será notificada por los encargados de la práctica en la página del curso y en las horas de práctica.

12. Las entregas de cada etapa se realizarán en las siguientes semanas y tendrán la siguiente ponderación correspondiente en la evaluación del curso:

Etapa	Semana	Valor (puntos)	Asignación
1	3	5	Analizador lexicográfico
2	5	5	Analizador sintáctico
3	8	9	Tabla de símbolos y verificaciones estáticas
4	11	11	Ejecución con verificaciones dinámicas

13. Deberá realizar el proyecto individualmente o en equipos de dos integrantes que estén actualmente cursando CI3725.

3 Requerimientos específicos

Esta sección es normativa.

1. Para esta etapa del proyecto, Ud. debe escribir un analizador sintáctico LALR(1) para el lenguaje de programación *Trinity* utilizando las herramientas que haya seleccionado.

2. En caso de haber seleccionado las herramientas de *Haskell*, debe utilizar el generador de analizadores sintácticos *Happy*. En caso de haber seleccionado las herramientas de *Ruby*, debe utilizar el generador de analizadores sintácticos *Racc*. En caso de haber seleccionado las herramientas de *Python*, debe utilizar el generador de analizadores sintácticos *PLY*. Debe especificar una gramática para *Trinity* en el formato correspondiente a su herramienta.

3. La porción de su gramática correspondiente a las expresiones de *Trinity* debe ser ambigua y toda ambigüedad de su gramática debe ser resuelta utilizando reglas de precedencia adecuadas según la especificación del lenguaje. Su analizador sintáctico **no** debe tener conflictos *shift/reduce* ni *reduce/reduce* que no hayan sido resueltos por las reglas de precedencia especificadas junto con su gramática.

4. En caso de haber seleccionado las herramientas de *Haskell*, debe definir los tipos de datos algebráicos que Ud. determine necesarios para representar los distintos tipos de nodos de árboles de sintaxis abstracta de *Trinity*. En caso de haber seleccionado las herramientas de *Ruby* o *Python*, debe definir las jerarquías de clases que Ud. determine necesarias para representar los distintos tipos de nodos de árboles de sintaxis abstracta de *Trinity*. **No** se considera aceptable que utilice un único tipo o constructor, o una única clase concreta, para representar todos los nodos de su árbol de sintaxis abstracta.

5. Su analizador sintáctico debe sintetizar un valor de su lenguaje de programación que represente al árbol de sintaxis abstracta correspondiente al programa de *Trinity* analizado en caso de que éste sea sintácticamente correcto. Para esto, debe asociar acciones adecuadas a las producciones de su gramática que construyan los nodos del árbol de sintaxis abstracta correspondientes a cada producción. No necesariamente debe haber un tipo o una clase distinta para *cada* producción de su gramática.

6. Al ejecutarse, su programa debe abrir el archivo de entrada e intentar consumirla por completo produciendo un árbol de sintaxis abstracta para el programa contenido en ella en caso de que su sintaxis sea válida. El archivo de entrada se encontrará en la ruta especificada en el primer y único argumento de línea de comando suministrado a su programa.

7. Su analizador sintáctico debe operar sobre una secuencia de lexemas obtenida de su analizador lexicográfico.

8. En caso de que el programa a analizar tenga errores lexicográficos, debe reportarlos **todos** al igual que en la primera etapa; sin embargo, no debe indicar qué lexemas fueron reconocidos indivudualmente. En caso de que el programa a analizar no tenga errores lexicográficos y no tenga sintaxis válida, solo debe reportar el **primer** error de sintaxis que encuentre, indicando la ubicación del primer caracter del lexema que produjo la detección del error de sintaxis, y luego abortar su ejecución.

9. Al final de la ejecución, su programa debe terminar con un estado de salida distinto de cero si se encontraron errores, y con el estado de salida cero si no se encontraron errores.

10. El árbol de sintaxis abstracta producido por su analizador sintáctico al analizar un programa sin errores de sintaxis debe representarse en la salida estándar en forma de texto.

11. Cada nodo del árbol de sintaxis abstracta debe representarse indicando el tipo de estructura sintáctica a la cual corresponde, la información propia de esa estructura sintáctica, y la representación textual de los subárboles que contiene. La representación textual del árbol de sintaxis abstracta debe utilizar indentación para indicar la contención de unos nodos del árbol dentro de otros.

12. Como ejemplo concreto, la representación textual en la salida del analizador sintáctico de un fragmento de programa que represente al código `set foo = 21 * 2 + bar + 1.99;`, se podría imprimir algo similar a

```
Asignación:
  lado izquierdo:
    Identificador:
```


```
    nombre: foo
lado derecho:
  Suma:
    operando izquierdo:
      Suma:
        operando izquierdo:
          Multiplicación:
            operando izquierdo:
              Literal numérico:
                valor: 21
            operando derecho:
              Literal numérico:
                valor: 2
        operando derecho:
          Identificador:
            nombre: bar
    operando derecho:
      Literal numérico:
        valor: 1.99
```

En este ejemplo, el fragmento de árbol de sintaxis abstracta mostrado tiene 9 nodos:

- Uno del tipo de nodo `Asignación` (que representa a un tipo de instrucción), con dos campos `lado izquierdo` que deben representar alguna de las formas válidas en el lado izquierdo de una asignación, y `lado derecho` que debe representar una expresión.
- Dos del tipo de nodo `Identificador` (que representa a un tipo de expresión), con un campo `nombre` que debe ser el texto del identificador.
- Dos del tipo de nodo `Suma` (que representa a un tipo de expresión), con dos campos `operando izquierdo` y `operando derecho` que deben representar expresiones.
- Uno del tipo de nodo `Multiplicación` que es similar a `Suma`.
- Tres del tipo de nodo `Literal numérico` (que representa a un tipo de expresión), con un campo `valor` que debe ser el valor del literal numérico.

Note que los fragmentos de programas como estos no serán entradas válidas para el analizador sintáctico, ya que éste funcionará únicamente con programas completos — estos ejemplos de fragmentos de programas se utilizan únicamente para ilustrar un ejemplo de formato de salida.

13. Como ejemplo concreto, si se encuentra un lexema inesperado `if` que comience en la columna 9 de la línea 84 y que produzca un error de sintaxis, como en


```
set a = if;
```

se podría imprimir algo similar a

```
Línea 84, columna 9: Lexema inesperado: Palabra reservada: if
```

14. Tanto los nombres asociados a cada tipo de nodo del árbol de sintaxis abstracta, como la selección exacta de cuáles elementos sintácticos corresponden a nodos del árbol, como el formato exacto de salida para los nodos y los errores de sintaxis quedan a su criterio — se requiere únicamente que contengan al menos la información requerida e ilustrada en estos ejemplos.

15. Para esta etapa del proyecto, **no** debe realizar análisis de tipos sobre el programa a analizar, ni verificar que las variables usadas estén correctamente declaradas, ni ningún otro análisis de contexto. Tampoco debe realizar ningún tipo de ejecución de los programas a analizar.