Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3725 – Traductores e Interpretadores
Enero – Marzo 2020

Version: 27/01/2020

Interpretador de `Willy`*
Análisis Sintáctico

La segunda etapa del proyecto corresponde al módulo de análisis sintáctico del interpretador del lenguaje `Willy`*. Este módulo toma como entrada la salida generada por el analizador lexicográfico que se implemento en la primera etapa. Por lo tanto, este etapa del proyecto también comprende cualquier modificación necesaria al módulo anterior, ya sea para terminarlo o para integrarlo con el analizador sintáctico. El analizador sintáctico requiere el uso de tablas de símbolos para poder realizar verificaciones estáticas sobre los programas en el lenguaje `Willy`*.

El analizador sintáctico debe aceptar como entrada la lista de tokens producidos por el analizador lexicográfico y construir una representación intermedia del programa `Willy`* que es analizada estáticamente y luego pasada al interpretador desarrollado en la última etapa del proyecto. El analizador debe *aceptar* cualquier entrada que corresponda a un programa válido en el lenguaje `Willy`*, y *rechazar* cualquier otra entrada que haya sido aceptada por el analizador lexicográfico pero que no se corresponda con un programa válido. Errores de ejecución de programas válidos son detectados por el interpretador y no por el analizador sintáctico. El analizador debe reportar los errores encontrados con mensajes de error que muestren la posición dentro del archivo de entrada del lugar en donde ocurre el error así como también información de "contexto" que permitan al programador `Willy`* identificar y corregir los errores con facilidad.

Para implementar el analizador sintáctico del lenguaje `Willy`* se debe:

- Diseñar una gramática libre de contexto que genere el lenguaje `Willy`*, y expresarla en la herramienta de software utilizada. *La gramática no puede ser ambigüa y no debe utilizar recursión por derecha.*
- Diseñar los tipos de datos abstractos necesarios para el análisis estático requerido. Los tipos de datos pueden hacer uso de los tipos de datos ofrecidos por en el lenguaje de programación utilizado. En particular, se necesita implementar una tabla de símbolos y un tipo de datos recursivo para representar el programa "abstracto" (i.e., representación intermedia del programa `Willy`*).
- El analizador sintáctico debe reportar de manera clara los siquientes errores de contexto que que deben ser detectados durante el análisis estático:
  - Por simplicidad, no se puede usar el mismo identificador para distintos usos. Por ejemplo, adentro de un mundo no se puede utilizar el mismo identificador para nombrar un booleano y para nombrar un goal.
  - Es un error definir más de un mundo con el mismo nombre.
  - Es un error definir más de una tarea con el mismo nombre.
  - Es un error definir un mundo con 0 filas o 0 columnas.
  - Adentro de un mismo mundo, es un error definir más de un tipo de objeto con el mismo nombre.
  - Es un error ubicar 0 objetos de un tipo en una localidad, celda o cesta.
  - Es un error definir una cesta de capacidad 0.
  - Es un error referirse a objetos de un tipo inexistente.

- Adentro de un mismo mundo, es un error definir más de un booleano con el mismo nombre.
- Es un error utilizar un identificador de un tipo en una condición que espera un identificador de otro tipo.
- Es un error definir una tarea para un (identificador de) mundo inexistente.
- Adentro de la misma tarea, es un error definir más de una instrucción con el mismo nombre.
- Es un error utilizar una instrucción inexistente.
- Es un error colocar objetos fuera de los límites del mundo.
- Es un error localizar inicialmente a `willy` fuera de los límites del mundo.
- Es un error definir una iteración acotada que repita un número no positivo de veces.
- Es un error tomar o dejar objetos de un tipo inexistente.
- Es un error cambiar o definir el valor de un booleano inexistente.
- Es un error utilizar un booleano inexistente o referirse a un tipo de objeto inexistente en una condición.

Tabla de Símbolos

La tabla de símbolos es un componente crucial en la implementación de cualquier compilador o interpretador de un lenguage. Ella es utilizada para guardar los símbolos definidos y para verificar si un símbolo existe. Las tablas de símbolos se manejan dentro de una estructura de pila (stack). Cada vez que se entra en un contexto sintáctico, una nueva tabla se empila, y cada vez que se sale de un contexto, la tabla actual se desempila. Una tabla que se desempila puede destruirse o guardarse para ser utilizada posteriormente.

Las búsquedas se realizan en orden comenzando en la tabla de símbolos en el tope de la pila: si el símbolo no se encuentra en esa tabla, se intenta con la immediatamente siguiente en la pila, y así hasta encontrar el símbolo o terminar de revisar todas las tablas en la pila, en cuyo caso se retorna que el símbolo es inexistente.

La tabla de símbolos a implementar debe asociar a cada identificador en la tabla, el tipo de identificador (mundo, tipo de objeto, booleano, goal e instrucción) junto con información útil para el análisis estático. Por ejemplo, se puede guardar la localización en el programa de entrada del lugar donde el símbolo se define. Para los mundos, el identificador puede asociarse a la tabla de símbolos que se utilizó al análizar la especificación del mundo.

Entre otras cosas, la estructura de tabla de símbolos debe soportar las siguientes operaciones:

- `find(<symbol>)`: busca un símbolo en la pila de tablas. Retorna los datos asociados al símbolo si el símbolo se encuentra en la tabla. De lo contrario indica que el símbolo no existe.
- `insert(<symbol>,<data>)`: inserta el símbolo dado en la tabla al tope de la pila y lo asocia a los datos dados. Los datos deben instanciar una clase predefinida que acompañan a cada símbolo en la tabla. Es un error insertar en la tabla un símbolo que ya existe.
- `push_empty_table()`: empila una nueva tabla de símbolos vacía en la pila de tablas.
- `pop()`: desempila y retorna la tabla en el tope de la pila. Es un error desempilar una tabla de un stack vacío.
- `empty()`: retorna un booleano que indica si la pila de tablas está vacía.
- `push(<table>)`: empila la tabla dada en la pila de tablas.

Representación Intermedia o Abstracta de Programas

El analizador sintáctico debe aceptar o rechazar una entrada. La acepta cuando la entrada se corresponde con un programa en el lenguaje `Willy`*, y dicho programa pasa todos los chequeos estáticos implementados. El analizador rechaza la entrada si no se corresponde con un programa, o si el programa no pasa alguno de los chequeos estáticos. Si la entrada es aceptada, el analizador debe retornar una representación intermedia (abstracta) del archivo de entrada.

Un programa en el lenguaje `Willy`* contiene uno o más mundos, y una o más tareas. Un mundo es representado de forma abstracta como una instancia de la clase `World` que debe ser definida por Ud. La clase debe contener campos o registros para almacenar los diferentes elementos que puede estar definidos, y debe contener métodos para accesar y modificar dichos elementos. Entre otras cosas, un mundo debe soportar métodos para acceder a las dimensiones del mundo, determinar si cierta celda está ocupada por una pared o está libre, determinar si un identificador es un tipo de objeto válido, determinar el color asociado a un tipo de objeto, determinar cuantos objetos de un tipo dado están en una celda dada, determinar cuantos objetos de un tipo dado están en la cesta de `willy`, determinar la celda y orientación inicial de `willy`, determinar la capacidad de la cesta, determinar si un identificador dado se corresponde con un booleano del mundo, determinar el valor de verdad de un booleano dado, determinar el valor inicial de un booleano dado, determinar las condiciones goal y la condición final de goal, etc.

Una tarea se representa de forma abstracta como una instancia de la clase `Task`. Al igual que antes, la clase `Task` debe implementar métodos para acceder y modificar los elementos que definen la tarea. En particular, el programa contenido en una tarea debe representarse con una estructura recursiva de árbol.

Segunda Entrega

- La segunda entrega consiste en implementar el módulo sintáctico en el lenguaje de su elección. Este módulo debe estar integrado con el módulo lexicográfico que se implementó para la primera entrega.
- Se debe entregar un programa principal que primero alimente al módulo lexicográfico con la entrada y luego utilize la salida de este módulo para alimentar el módulo sintáctico.
- Todos los módulos del sistema deben estar completa y correctamente documentados. Tambíen debe entregarse un pequeño informa en formato `.md` en el repositorio Github del grupo, en el directorio correspondiente a la primera entrega. La forma correcta de invocar el programa principal debe estar claramente descrita.
- Un `Makefile` que permita compilar el programa.
- El programa principal será compilado y utilizado desde la línea de comandos de dos manera diferentes:

  1. Si se invoca el programa suministrando exactamente un argumento en la línea, el programa debe intentar abrir ese archivo y procesarlo:
     a) Si el programa es sintácticamente correcto y no tiene errores de contexto estático, mostrar en pantalla todos los símbolos definidos y su naturaleza, junto con una representación de las clases abstractas que representan los elementos en el programa. Notar que toda instrucción y condición dentro de una tarea debe estar presente en la representación abstracta.
     b) Si el programa tiene errores lexicográficos, el programa debe mostrarlos tal como se hizo en la Primera Parte.
     c) Si el programa no tiene errores lexicográficos, pero tiene errores de sintaxis, el programa debe mostrar el primer error de sintaxis, indicando la línea y columna donde ocurre, e indicando los tokens del contexto.
     d) Si el programa no tiene errores lexicográficos ni sintácticos, pero tiene errores de contexto, se debe mostrar *todos* los errores de contexto, indicando la línea y columna donde ocurre, así como la definición, símbolo o instrucción que ocasiona el error y su naturaleza.

Si el archivo suministrado como argumento no existe o no puede abrirse por la razón que sea, el programa debe terminar indicando el problema. Es inaceptable que el programa aborte o tenga una terminación anormal.

2. 2. Si se invoca el programa sin argumentos, el programa debe ofrecer un prompt en el cual escribir el nombre del archivo a procesar. El procesamiento debe retornar todos los tokens procesados o todos los errores detectados en la línea.

- **Fecha de Entrega:** Martes 18 de Febrero 2020 hasta la media noche.
- **Valor:** 10 puntos.