UNIVERSIDAD RAFAEL LANDÍVAR, FACULTAD DE INGENIERÍA

Proyecto (segunda fase)

Ing. Msc. Víctor Orozco

29 de octubre de 2023

1. INTRODUCCIÓN

En este proyecto, su objetivo es implementar dos analizadores sintácticos para CHILERO. Para esto necesitara 4 herramientas:

1. JFlex y las expresiones regulares creadas en la fase previa de su proyecto
2. CUP como el generador de analizadores sintácticos para su primera versión de parser
3. ANTLR como el generador de analizadores léxicos y sintácticos para su segunda versión de parser
4. Un paquete para creación/manipulación de arboles AST creado por la Universidad de Berkley, California

La salida de su analizador será un árbol de sintaxis abstracta (AST), el cual debe ser construido usando acciones semánticas de CUP (versión 1) y ANTLR(versión 2).

Como primer paso y de forma similar a la fase número uno, se le sugiere consultar la estructura sintáctica de Cool, que se encuentra en la Figura 1 de "The Cool Manual".

Como sugerencia final se le recomienda que antes de iniciar con la programación, analice detenidamente el contenido de este documento.

2. Resolución

Para iniciar la resolución de esta primera fase su instructor le proporcionara un enlace hacia un repositorio en GitHub Classroom.

3. Probando el parser

Como tarea previa a esta fase del proyecto usted ha elaborado un analizador léxico para el lenguaje de programación CHILERO, debera reutilizar este analizador para esta fase como paso previo en CUP. Y posteriormente replicar la funcionalidad de JFlex y CUP en ANTLR.

De forma similar a la fase anterior el área de trabajo proporcionada mediante GitHub Classroom incluye el archivo Makefile correspondiente a 1- Generación de analizador léxico con JFLEX, 2- Generación de analizador sintáctico con CUP y 3- Generación de analizador léxico y sintáctico con ANTLR.

Se incluye nuevamente un generador de script para ejecutar una versión de coolc denominada `cupcoolc` la cual es capaz de 1- Ejecutar el análisis léxico con el analizador léxico generado por JFlex y 2- Ejecutar el análisis sintáctico con CUP.

Así mismo, se incluye nuevamente un generador de script para ejecutar una versión de coolc denominada `antlrcoolc` la cual es capaz de ejecutar el análisis léxico y sintáctico producido por ANTLR, junto a las otras piezas que hacen posible el funcionamiento del programa con SPIM.

Usted debera realizar sus pruebas sobre los parser tanto en archivos bien definidos de COOL y CHILERO, como en malos, para ver si todo está funcionando correctamente. Recuerden, los errores en su parser se pueden manifestar en alguna otra parte del proceso de compilación. Su parser va a ser calificado utilizando tanto su analizador léxico, como el analizador léxico de referencia de COOL.

4. Salida del parser

Sus acciones semánticas deberían de construir un árbol AST. La raíz (y solamente la raíz) del AST debería de ser de tipo `program` . Para los programas que son parseados satisfactoriamente, la salida del parser es una secuencia de nodos del AST.

Para programas que contengan errores (léxicos o sintácticos), la salida son mensajes de error del parser. El esqueleto le proporciona función que reporta errores imprimiendo los mensajes en un formato estándar; por favor NO modifiquen esto. Ustedes no deberían de invocar esta función directamente en las acciones semánticas en la versión 1, ya que CUP automáticamente invoca a esta función cuando un error es detectado. En el caso de ANTLR deberá

analizar el contenido de la función y replicar su funcionamiento mediante acciones semánticas.

Para algunas expresiones que puedan abarcar varias líneas de código, por ejemplo:

```
1 foo(
2   1,
3   2,
4   3
5 )
```

Note que abarca 5 líneas, de la 1 a la 5. Cuando construyan alguna entrada en el árbol AST que abarque múltiples líneas son libres de indicar a que número de línea pertenece esta entrada, siempre y cuando, el número esté en el rango que abarque la entrada en el árbol AST., en el ejemplo anterior podría ser 1, 2, 3, 4 o 5.

Para facilitar su solución, los parser solo deberán de funcionar para programas que estén contenidos en un solo archivo. El soporte a compilación de archivos múltiples es optativo.

5. ERRORES

Para la emisión de errores, el pseudo no terminal `error` debera ser utilizado como una transición de manejo de errores en el parser. El propósito de error es permitirle al parser continuar después de un error detectado, considere que a pesar de que la transición es funcional, el parser puede volverse completamente confuso. Vean la documentación de CUP para saber como utilizar error correctamente. Para recibir toda la nota, sus parser debería de recuperarse por lo menos en las siguientes situaciones:

1. Si hay algún error en una definición de clase pero la clase es terminada correctamente y la siguiente clase está correcta sintácticamente, el parser debería de ser capaz de empezar de nuevo en la siguiente definición de clase.

2. El parser debe recuperarse reconociendo marcas de finales de expresiones, por ejemplo en un bloque loop delimitado por pool, debe reportar el error en la estructura y continuar su analisis luego de pool, o en caso de un error en let, el parser debe continuar desde el final de la declaración de variables.

No se enfoque demasiado en los números de línea que aparecen en los mensajes de error que su parser genera. Si su parser está funcionando correctamente, el número de línea generalmente va a ser la línea donde se encontró el error. Para construcciones erroneas que abarquen múltiples líneas, el número de línea por lo general va a ser la última línea de esa construcción.

6. OBSERVACIONES

En el proyecto serán necesarias declaraciones de precedencia, pero solo para las expresiones. No utilicen declaraciones de precedencia ciegamente (es decir, no resuelvan un conflicto shift-reduce en su gramática agregando reglas de precedencia hasta que ya no aparezca).

El let de COOL introduce una ambiguedad en el lenguage (traten de construir un ejemplo si es que no están convencidos). El manual resuelve esta ambigüedad diciendo que el let se extiende a la derecha tanto como se pueda. Dependiendo de como su gramática sea escrita, esta ambigüedad puede aparecer en su parser como un conflicto shift-reduce involucrando las producciones del let. Si ustedes se encuentran en esta situación, talvez quieran considerar resolver el problema utilizando características CUP y ANTLR que permitan que la precedencia sea asociada a las producciones (no solamente a los operadores). Se recomienda revisar la documentación de CUP y ANTLR para obtener información en como utilizar esta característica.

Debe declarar los "tipos"de CUP para sus no terminales y terminales que tienen atributos. Por ejemplo, en el esqueleto cool.cup está la declaración:

```
1 nonterminal programc program;
```

Esta declaración dice que el no terminal `program` tiene tipo `programc`.

Es fundamental que declare los tipos correctos para los atributos de los símbolos gramaticales; si no se ejecutau usted está garantizando que su analizador no funcionará. No necesita declarar tipos para símbolos de gramática que no tienen atributos. Así mismo luego de observar el funcionamiento en CUP, debera replicar este comportamiento en ANTLR.
La verificación de tipos del compilador de java `javac` se puede quejar si utilizan los constructores de los nodos del árbol con el tipo incorrecto. Si ustedes corrigen los errores con casting, su programa puede lanzar una excepción cuando el constructor note que está siendo utilizado con los tipos incorrectos.

7. REFERENCIAS

1. Manual de JFlex
2. Manual de ANTLR
3. Manual de CUP
4. Manual de la clase Tree

8. NOTAS FINALES

Fecha límite de entrega: Jueves 16 de noviembre, 23:55