72.39 Autómatas, Teoría de Lenguajes y Compiladores
Trabajo Práctico Especial
1er Cuatrimestre 2021

Docentes:

- Rodrigo Ezequiel Ramele
- Ana Maria Arias Roig
- Juan Miguel Santos

Integrantes:

- Quintairos, Juan Ignacio - 59715 - jquintairos@itba.edu.ar
- Revich, Igal Leonel - 60390 - irevich@itba.edu.ar
- Sicardi, Julian Nicolas - 60347 - jsicardi@itba.edu.ar
- Spitzner, Agustín - 60142 - aspitzner@itba.edu.ar

Índice:

Idea subyacente y objetivo del lenguaje

Para este proyecto, se presenta el lenguaje *"Geome"*. Este es un lenguaje que podría describirse como "orientado a geometría", presentando tipos de datos que representan figuras geométricas (en esta versión triángulo, círculo y rectángulo) de forma nativa. Geome permite la creación de figuras especificando sus propiedades, así como también obtener perímetro y área y poder compararlas entre sí, sumado a una capa simple de manejo de expresiones aritméticas. Esto permite que pueda usarse de forma rápida para poder obtener estas propiedades sin necesidad de recordar la amplia gama de fórmulas específicas que posee la geometría. El enfoque de nuestro lenguaje no es sobre la graficación, sino sobre la parte matemática de la geometría.

Si bien el lenguaje posee una sintaxis similar a aquella del lenguaje C para manejo de tipos de datos en común, también presenta distintas simplificaciones con el objetivo de hacerlo más descriptivo y apto para personas con una menor experiencia en programación. Esta búsqueda de la simplicidad y comprensión rápida es otro de nuestros objetivos.

Consideraciones realizadas

Como se menciona en la sección previa, la sintaxis de Geome está influenciada por aquella del lenguaje C, tomando varios elementos de esta. Por esta razón se decidió utilizar el mismo como lenguaje de salida de nuestro compilador.

A la hora de utilizar el lenguaje Geome, debe tenerse en cuenta ciertas acciones que no son realizables dentro del mismo :

- **Declaración de variables con nombre de palabras *reservadas* :** No pueden declararse variables con el nombre *int, string , circle, triangle, rectangle, radius, base, height, side_1, side_2, side_3, start, end, if, otherwise, print, read_num, and, or, eq, not_eq, create_circle, create_rectangle, create_triangle, get_perimeter* y *get_area,* pues se consideran *palabras reservadas* del lenguaje dado que corresponden a tipos de datos, o propiedades y funciones geométricas. De utilizar dichas palabras como nombre de variable se obtendrá un *syntax error* en la línea correspondiente.

- **Declaración de variables sin valor inicial :** El lenguaje Geome no permite la declaración de variables sin inicialización. En caso de hacerlo se obtendrá un *syntax error* en la línea correspondiente.

- **Definición de funciones :** Este lenguaje no le permite al usuario definir sus propias funciones. Solo puede utilizar las funciones geométricas otorgadas por el mismo (Ver en "*Funciones built-in*" dentro de *"Descripción de la gramática"*).

- **Manejo de números con punto flotante :** Actualmente el lenguaje Geome no permite el manejo de números con punto flotante, siendo un tipo de dato no

permitido. Por lo tanto, todas las expresiones asociadas a cálculos algebraicos siempre dan un resultado entero.

- **Lectura de datos de entrada estándar :** En esta versión, nuestro lenguaje solo permite leer de entrada estándar datos de tipo int. Esto se debe a que el manejo de strings está más asociado en su mayor parte a impresiones en salida estándar, por lo cual se prefirió implementar otras funcionalidades con una mayor utilidad para el lenguaje.

Descripción del desarrollo del TP

Dentro de lo que es el desarrollo de nuestro compilador se pueden distinguir tres partes fundamentales: análisis léxico, análisis sintáctico y semántico y traducción a lenguaje de salida.

Para el análisis léxico se utilizó LEX, definiendo las distintas palabras claves y símbolos de nuestro lenguaje. Para el análisis sintáctico se utilizó YACC, construyendo un parser con análisis ascendente (LALR(1) en específico), complementado con una implementación de un árbol AST (*Abstract Syntax Tree*) acorde, para así lograr una mejor gestión de los datos necesarios para etapas posteriores al parseo.

El análisis semántico se realiza tanto en las reglas de cada producción como en la creación de los nodos del árbol de acuerdo a la complejidad del análisis, generando mensajes de errores apropiados por salida de error. Cabe destacar que se implementó además una versión de tabla de símbolos más completa en forma de lista de listas, donde se representan primero los distintos scopes del programa asociados a los varios bloques de código y luego, por cada scope, las variables declaradas en este. Esto permite un mayor control en las declaraciones en distintos bloques de código, evitando repetición de declaraciones y descartando variables una vez que se terminó de procesar el bloque completo.

Finalmente y usando el AST, se realiza una última etapa de traducción en lenguaje C (elegido como lenguaje de salida) para construir un archivo .c, que luego debe compilarse con gcc (recordando incluir el flag -lm para el uso de la librería math.h) para generar el ejecutable final. Para más detalles sobre esto, ver el "*README.md*" incluido en el repositorio.

Descripción de la gramática

- Para comenzar, todo programa debe comenzar con *"start"* y terminar con un *"end"*

- Esta versión del lenguaje no acepta un programa vacío (ya que se considera sin sentido), como mínimo debe haber una instrucción.

- Las instrucciones deben terminar con ";"

Tipos de datos

Nuestra gramática permite la utilización de los siguientes tipos de datos:

- *int* : El cual representa un número entero, tal como en el lenguaje C
- *string*: El cual representa una cadena de caracteres
- *circle* : Representa una figura de tipo circulo
- *rectangle*: Representa una figura de tipo rectángulo
- *triangle*: Representa una figura de tipo triángulo

Definiciones y asignaciones

- Toda definición de variable debe venir acompañada de una asignación, es decir, toda definición debe ser de la forma: tipo_de_variable nombre_variable = valor;

- Las definiciones de variables son dependientes del scope en el que están. Solo se permite definir variables que no estén definidas en un scope menor (más general)

- Por cada ciclo o bloque condicional anidado, se le otorga un número de scope mayor, y cuando el parseo llega al final de este, se liberan todas las definiciones de variables de tal manera de que se puedan volver a definir de nuevo

- Para definir una variable, el nombre de esta debe empezar con una letra y contener únicamente: letras en minuscula y mayuscula, números o el símbolo "_"

Condicionales y ciclos

- La gramática también permite la utilización del condicional *"if"*, seguido de una expresión booleana, sea relacional o lógica, y el bloque de código a ejecutar entre llaves. En caso de que no se cumpla la expresión booleana, se puede incluir un bloque de código a ejecutar utilizando el término *"otherwise"* (Análogo al *"else"* en c), poniendo dicho bloque de código entre llaves seguido del término recientemente mencionado.

- Las expresiones relacionales se ven identificadas por el uso de los operadores:
    - *<,<=,>,>=* (análogos de C)
    - *eq* (equivalente a == en C) y *not_eq* (equivalente a != en C). La decisión de usar eq y not_eq fue para hacer algo más descriptivo el lenguaje para usuarios que podrían no conocer que el símbolo "==" se usa para igualdades por ejemplo.

- Los operadores eq y not_eq sirven tanto para comparar datos o variables de tipo int como para comparar variables de tipo figura, devolviendo en este caso verdadero solo si ambas variables son del mismo tipo de figura y poseen las mismas propiedades.

- Las expresiones lógicas en cambio se ven identificadas por el uso de los operadores *"and"* (equivalente al && en C) y *"or"* (equivalente al || en C). Este cambio también se

hizo en busca de hacer más descriptivo el lenguaje para usuarios que podrían no conocer el uso estándar de && o || para expresiones lógicas.

- Estas expresiones booleanas representan al falso como 0 y al verdadero como 1, tal como hace C.

- La gramática permite el uso de ciclos a partir del uso del comando "*repeat while*", seguido de una expresión booleana entre paréntesis, y el bloque de código a ejecutar entre llaves.

Funciones Built-In

- La gramática dispone de alguna funciones built-in para poder inicializar las variables de tipo figura, tales como son:
    - ***circle create_circle(int radius)*** : La cual recibe el radio de la variable círculo que se quiere inicializar, y devuelve un círculo conteniendo tal propiedad.

    - ***rectangle create_rectangle(int base,int height)*** : La cual recibe la base y la altura de la variable rectángulo que se quiere inicializar, y devuelve un rectángulo conteniendo tales propiedades.

    - ***triangle create_triangle(triangle_side_1,triangle_side_2, triangle_side_3)*** : La cual recibe la longitud de cada lado de la variable triángulo que se quiere inicializar, y devuelve un triángulo conteniendo tales propiedades.

    - ***int get_perimeter(figure_type figure)*** : La cual recibe una variable de tipo figura (circle, rectangle o triangle) y devuelve su perímetro.

    - ***int get_area(figure_type figure)*** : La cual recibe una variable de tipo figura (circle, rectangle o triangle) y devuelve su área. En caso del área del triángulo, esta se calcula mediante la *"Fórmula de Herón" ,* que utiliza el semiperímetro del triángulo y cada uno de sus lados. (Ver en *"Referencias"*)

    - ***void print(string text | int num | figure_type figure)*** : La cual permite imprimir a salida estándar el texto especificado. Esta función permite tanto imprimir texto especificado entre comillas dobles (por ejemplo: print("Texto");) como valores asociados a variables de tipo int, string y figuras (ejemplo: print(circle->radius) o print(my_triangle).

    - ***void read_num(int num)***: La cual lee de STDIN esperando un número que luego será almacenado en la variable num.

  Cabe aclarar que no existe el tipo de dato *figure_type* dentro del lenguaje, pero se usa para indicar que las funciones que reciben ese tipo de parámetro pueden recibir cualquiera de las figuras geométricas manejadas (círculo, rectángulo o triángulo).

  Para revisar las producciones asociadas a la gramática, ver el archivo *"grammar.bnf"* que se encuentra en el repositorio de Github.

Dificultades encontradas en el desarrollo del TP

La primera dificultad encontrada en instancias previas al desarrollo fue la creación del lenguaje, sobre todo la elección de una temática suficientemente acotada y específica para poder ser realizada, pero a su vez innovadora y que aportase al mundo de la programación.

La definición de la gramática es otra dificultad mencionable. Primero, el grupo completo se dedicó a discutir sobre las distintas producciones a utilizar, con el objetivo de agrupar todas las capacidades del lenguaje para tener una mayor visión de este. Esta gramática luego se fue adaptando a medida que se implementaba el compilador, comenzando por representar lenguajes reducidos para luego ir agregando nuevas producciones para expandirlo. También se debió hacer frente a los clásicos conflictos de reduce/reduce y shift/reduce vistos en las clases de análisis sintáctico ascendente.

Otra dificultad en el desarrollo fue la definición y creación del árbol AST. En una primera instancia hubo varias dificultades para comprender el concepto, teniendo en cuenta las amplias implementaciones posibles. El grupo se vio obligado a hacer un análisis profundo y discutir cómo lograr representar las distintas producciones en los distintos nodos. Una vez implementado, el AST probó ser una alternativa óptima para realizar el parseo y luego la traducción, dando una mayor independencia entre cada producción.

Por último, se deben mencionar dos dificultades asociadas a nuestro lenguaje. La primera es el manejo de los distintos scopes, tema que requirió mucho análisis hasta lograr encontrar una alternativa adecuada, que en nuestro caso fue la lista de listas mencionada en la sección *“Descripción del desarrollo del TP”*. La segunda fue la representación de nuestros tipos de datos de figuras geométricas, que requirió generar distintas funciones de sistema para lograr el manejo equivalente en C, funciones que se copian en el archivo resultante previo a la traducción del código de nuestro lenguaje. En nuestro caso el compilador agrega estas funciones directamente sobre el código resultante, pues se buscó que el compilador fuese autosuficiente sin necesidad de archivos externos o de forzar la compilación del programa con una librería de nuestra autoría.

Futuras extensiones

Las posibles extensiones que se pueden incorporar en un futuro son :

- ***Manejo de nuevas figuras :*** Se pueden incluir nuevos tipos de figuras, como rombos, trapecios, entre otras. Esto no involucraria demasiada complejidad, pues consiste en crear nuevos tipos de datos correspondientes a dichas figuras, junto con sus respectivos nodos en el AST y su correspondiente implementación de las funciones geométricas en C.

- ***Implementación de comentarios :*** Una posible extensión es la implementación de comentarios dentro del código. Esta no involucraria demasiada complejidad, pues debería tomarse algún carácter que no se use para algún propósito en particular dentro del lenguaje (Por ejemplo el *"#"*), y entre 2 de esos caracteres debería colocarse el cuerpo del comentario, que en c se traduciría como ese texto entre */* y */*. La mayor complejidad de dicha funcionalidad consistiria en permitirle al usuario que dentro del comentario pueda poner todo tipo de texto. Durante el desarrollo del trabajo práctico se consideró fuertemente implementar esta funcionalidad, pero decidimos priorizar otras funcionalidades relacionadas al manejo de figuras geométricas, que es el foco de nuestro lenguaje.

- ***Declaración de variables sin ser inicializadas :*** Por el momento, nuestro lenguaje no permite la declaración de variables sin un valor inicial asignado. Sin embargo, esto en un futuro podría incluirse sin involucrar demasiada complejidad. Esto consistiría en modificar la producción correspondiente a la declaración de variables dentro de la gramática, para que en vez de que tenga una asignación por defecto, en el final de dicha producción tenga un nuevo símbolo no terminal que derive en la asignación de un valor o en λ (lambda). Esta funcionalidad también debatimos si implementarla pero no la consideramos de gran valor en nuestro lenguaje, aunque se podría agregar a futuro.

- **Definición de funciones de usuario :** Actualmente nuestro lenguaje no le permite al usuario la definición de funciones propias, sino que solamente puede usar las otorgadas por el mismo utilizadas para creación de figuras y obtención de perímetro y área de las mismas (Ver en "*Funciones built-in*" dentro de *"Descripción de la gramática"*). Esta funcionalidad podría incluirse en un futuro, teniendo en cuenta la complejidad del manejo de parámetros variables. Cabe destacar que se ha realizado un manejo aproximado de esto al guardar las funciones que ofrece el lenguaje dentro de una "tabla de símbolos" que funciona como "tabla de definiciones de funciones" y además se utiliza un nodo de función genérica para el árbol AST. Sin embargo, deberían hacerse otro tipo de modificaciones para lograr un manejo más adecuado y lograr el objetivo mencionado anteriormente.

- **Manejo de números con punto flotante :** Por el momento nuestro lenguaje no permite el manejo de números en punto flotante. Dicho manejo podría incluirse en un futuro, pero su complejidad reside en el parseo de parte entera y parte decimal, así como también realizar las diferentes operaciones relacionales con los mismos (Esto involucraria la definición de algún tipo de Epsilon).

Referencias

- Clases dadas por la cátedra
- http://www.cs.ecu.edu/karl/5220/spr16/SFLIMP/Assn3/ast.html Arbol ast
- https://www.youtube.com/watch?v=54bo1qaHAfk Ejemplo Lex
- https://www.youtube.com/watch?v=__-wUHG2rfM Ejemplo Yacc
- https://es.wikipedia.org/wiki/F%C3%B3rmula_de_Her%C3%B3n Formula de Heron para el calculo del area del triangulo