TPE Re-C-tas

Instituto Tecnológico de Buenos Aires

Autómatas, teoría de lenguajes y compiladores

Primer Cuatrimestre 2020

- Alejandro Marcelo Rolandelli 56644
- Damian Nicolas Nuñez Köhler 54365
- Franco Agustin Baliarda 58306
- Marc Holste 56306
- Martin Galderisi Isola 58404

Objetivo del Lenguaje

La idea subyacente del lenguaje es dar la habilidad de programar en formato de recetas de cocina, donde las distintas variables son los ingredientes a usar para cocinar y las funciones son las recetas con sus procedimientos y vocabulario de cocina.

El objetivo es simplificar el código para una persona común que tenga un poco de conocimiento de cocina, para que escriba y lea recetas, no código. Esto elimina la barrera de tener que aprender un nuevo lenguaje, ya que el lenguaje de cocina es muy usado y no tiene un vocabulario tan dificil. El lenguaje está orientado a realizar tareas que requieran interacción del usuario, pudiendo obtener texto de entrada estándar y escribir por salida estándar.

La traducción elegida es al lenguaje C debido a que todos los integrantes tienen experiencia con este lenguaje, además de tener la herramienta de código abierto GCC para crear el ejecutable.

Consideraciones Realizadas

Una de las consideraciones que tuvimos en cuenta es el hecho de que un programa debe corresponder a una determinada receta específica. Es decir, que en un archivo no es posible definir múltiples recetas, sino que se tendrá un único conjunto de ingredientes seguido del procedimiento que se necesita seguir instrucción por instrucción. Esto implica que una receta está contenida en un único archivo, la cual deberá especificar todos los ingredientes y todas las instrucciones a ejecutar.

Otra de las consideraciones que se deben tener al crear una receta es que los ingredientes se deben definir obligatoriamente previo al procedimiento. No tendría sentido utilizar ingredientes en una receta que no se hayan mencionado como los que se van a necesitar y, por lo tanto, no se ofrecen instrucciones para definir nuevos ingredientes dentro del procedimiento de la receta. Esto tiene un beneficio para el pensamiento del desarrollador también, ya que se puede considerar una buena práctica definir las variables que se van a utilizar antes de la lógica del código.

Por último, otra consideración que se debió tener en cuenta es el manejo de la memoria que utiliza el programa en sí. Debido a que el lenguaje está orientado a permitir la interacción con el usuario, se necesita obtener memoria de manera dinámica para almacenar el texto que es ingresado para luego ser utilizado. Esto no presenta un problema trivial, ya que esa memoria luego debe ser liberada para hacer un uso eficiente y no tener leaks. Por lo tanto, debido a la naturaleza imperativa del lenguaje y el hecho de que una receta está contenida en sí misma, la memoria que se necesita obtener de manera dinámica se libera al llegar a una instrucción de retorno.

Desarrollo del trabajo

Para este trabajo, se utilizaron 2 herramientas vistas en clase (Yacc y Lex) y el compilador GCC.

1. Lex que consiste en leer la recetas entrada, analizar por patrones y obtener sus tokens correspondientes.

2. Yacc que toma los tokens generados por Lex y construye el árbol de código C de salida utilizando la gramática elegida para el lenguaje.

3. Una vez que Yacc arma el árbol con el código C, este se imprime en un archivo .c que elija el usuario. Luego, se toma este archivo y se compila el código con GCC para obtener el ejecutable final.

Descripción de la gramática

De manera de poder entender la gramática correspondiente al lenguaje, es importante reconocer en principio dos símbolos no terminales principales que determinan el formato del lenguaje: *ingredients* y *recipe*. Estos símbolos no terminales corresponden a las dos partes principales en un archivo escrito en este lenguaje: la sección de ingredientes y la sección del procedimiento de la receta respectivamente. Como se puede observar en la producción del símbolo distinguido, la parte sentencial derecha consiste únicamente de estos dos símbolos no terminales (*program: ingredients recipe)*. Luego, cada uno de estos símbolos tendrá las producciones necesarias para expandirse en sus correspondientes secciones.

En primer lugar, se tiene la sección de ingredientes comenzada por el símbolo no terminal *ingredients*. Como se puede ver en su producción, la sección comienza con el símbolo terminal *INGREDIENTS_TOKEN* seguido de las múltiples sentencias que definirán las variables a utilizar. Las instrucciones en concreto que pueden aparecer en esta sección se pueden ver de mejor manera en las producciones del símbolo no terminal *ingredient*. Cada producción comienza con el tipo de la variable (entero, cadena o booleano), seguido por el símbolo terminal *ID* el cual representa el nombre de la variable que se está definiendo. Además, en el caso de que se quiera inicializarla, estará seguida por el símbolo terminal *EQUAL_TO*, el cual tendrá a su derecha el valor con el que se le quiera inicializar. Esto puede ser una expresión (explicada más adelante en el símbolo no terminal *expression*), una cadena de texto o un valor booleano. Cabe destacar que en todos los casos la línea deberá estar terminada en el símbolo terminal *END_LINE*.

Luego de la sección de ingredientes, sigue la parte del procedimiento correspondiente al símbolo no terminal *recipe*. Al igual que la sección anterior, esta comienza con un símbolo terminal, en este caso *RECIPE_TOKEN*. Este símbolo estará seguido por múltiples instrucciones detalladas en las producciones de *instruction*, las cuales todas terminan con un símbolo *END_LINE*. Como se puede observar, se dividen en cinco producciones:

- *assignment END_LINE*: En este caso la instrucción se basa en una asignación. Si se observan las producciones del símbolo *assignment*, estas asignan un valor a la variable denotada por *ID*. Este valor puede ser una expresión, una cadena de texto o un valor booleano, al igual que en la inicialización de la variable en ingredientes.

- *call END_LINE:* El símbolo *call* corresponde a la llamada de una función. Esta llamada consiste en el símbolo terminal *PREPARE*, seguido por el nombre de la función a llamar. En el caso de requerir argumentos, estos se podrán ingresar de la manera *WITH* arg1,argn. Se ofrecen tres funciones para utilizar de manera nativa: *READ* la cual permite leer una cadena de texto de la entrada estándar, *WRITE* que permite escribir una cadena de texto con un determinado formato (del mismo modo que printf en C), y *TO_INT* que permite transformar una cadena de texto en un entero, útil en el caso de querer obtener valores numéricos de la entrada estándar.

- *return END_LINE:* Como el nombre lo indica, esta producción se utiliza en caso de tener una instrucción de retorno. El símbolo terminal para retornar es *EAT*, el cual puede estar seguido de una expresión, condición o cadena de texto.

- *if*: Esta producción corresponde al comienzo de un bloque condicional. El bloque *if* funciona de la misma manera que en C, evaluando una condición y, en el caso de que sea verdadera, ejecutar el código dentro del bloque. Este bloque comienza con el símbolo terminal *THEN,* está compuesto por una o más instrucciones y finaliza con el símbolo terminal *END*.

- *while:* El bloque *while* funciona de la misma manera que en C. Las instrucciones dentro del bloque (delimitado por *DO...END)* se ejecutarán en ciclo hasta que la condición en la instrucción del comienzo del bloque sea falsa.

Por último, caben destacar ciertos símbolos y producciones que poseen cierta relevancia en la gramática. En primer lugar se tiene el símbolo no terminal *expression*, el cual corresponderá al valor de una expresión aritmética. Este valor, como se observa en las producciones, puede ser obtenido directamente de un número *NUM*, de una variable denotada por *ID*, de una llamada a una función, o de un cálculo matemático (adición, resta, multiplicación, división). Otro símbolo que cabe destacar es *condition*, el cual representa una condición que puede ser evaluada para obtener un valor verdadero o falso. Para esto se pueden hacer comparaciones numéricas (mayor, menor, igual) y utilizar operadores lógicos. Por último, el símbolo no terminal *recipe_args* toma relevancia al momento de hacer una llamada a función. Este símbolo se utiliza para expresar los argumentos que recibe la función, todos separados por el símbolo terminal *COMMA*.

En el caso de querer ver la gramática y sus producciones de manera más clara, dentro del proyecto se encuentra el archivo *grammar.bnf*, el cual contiene las producciones en formato BNF.

Compilación

Para compilar el lenguaje, el usuario tiene que estar ubicado en el directorio src, dentro el proyecto y necesita tener instalado Bison, Flex, gcc y make. Un vez instalados los programas, se debe correr la siguiente instrucción en bash:

```
$> make all
```

Esta instrucción compila el lenguaje y genera el archivo “recetas” que contiene el parser del lenguaje. Una vez generado el lenguaje, para compilar una receta se debe correr la siguiente instrucción:

```
$> ./reCtas.sh nombreReceta.rc program.out
```

Donde reCtas.sh es el archivo bash que compila y nombreReceta.rc es el nombre del archivo que contiene la receta a compilar. Esto genera el ejecutable program.out donde program es el nombre deseado para el ejecutable, y parsedCode.c que contiene la receta en lenguaje C.

Dificultades encontradas

Una de las principales dificultades que se encontraron al principio del proyecto fue el aprendizaje y el entendimiento de las herramientas Yacc y Lex para el desarrollo del compilador del lenguaje. Además, para utilizar estas herramientas correctamente, se debió aprender acerca de gramáticas libre de contexto y árboles AST de manera de poder validar que el archivo pertenezca al lenguaje y traducirlo al lenguaje de salida.

Otra dificultad que se presentó durante el desarrollo fue al momento de implementar la funcionalidad de comentarios en el lenguaje. Debido a que estos son externos a la gramática, no deben aparecer en las producciones de esta. Esto implica que lex, al leer texto encerrado entre los caracteres ‘(‘ y ‘)’, debe omitir el comentario y no generar ningún token permitiendo ser transparente a la gramática sin tener que traducir los comentarios a C. De todas maneras, por más de que esta sea una solución óptima, presenta la dificultad de que los comentarios deben estar contenidos en una sola línea ya que de otra forma lex no podrá reconocer donde este comienza y termina.

Futuras extensiones

En el caso de requerir mayor funcionalidad a la que otorga el lenguaje, este se puede extender en varios aspectos para cumplir otros requerimientos.

En primer lugar, el lenguaje solo soporta tres tipos de datos en particular: numérico, cadena de texto y booleano. Esto significa que el lenguaje en principio no soporta tipos de datos complejos o colecciones de datos de manera nativa, una funcionalidad que puede ser agregada en una futura extensión.

Debido a que la naturaleza del lenguaje consiste en un archivo conteniendo una receta, no es posible refactorizar el código en recetas más pequeñas o importar recetas de otros archivos. Esto implica que, en caso de tareas más complejas, se pueden llegar a tener recetas considerablemente largas. Por otro lado, sin contar las funciones otorgadas de manera nativa para leer de entrada estándar y escribir en salida estándar, el lenguaje no posee de interfaces o funciones para realizar operaciones a más bajo nivel (como puede otorgar C). Estas funcionalidades de refactorización de código, importación de recetas e interfaces de bajo nivel pueden ser posibles features en futuras extensiones del lenguaje.

Por último, hay ciertas instrucciones para manejar el flujo de la ejecución del código que no se encuentran de manera nativa (ej: bucles *for*, bloques *switch* o *else*). Por más de que estas funcionalidades puedan ser reemplazadas por instrucciones análogas, como un *switch* por múltiples sentencias *if*, estas pueden ser parte de una futura extensión del lenguaje de manera de poder escribir código de manera más legible e intuitiva para el desarrollador.

Ejemplos

Dentro del proyecto, en la carpeta src/examples, se tienen cinco ejemplos de programas escritos en el lenguaje

- N_pares.rc: Solicita al usuario un número N e imprime los primeros N pares.
- max.rc: Pide al usuario que ingrese tres números y calcula el máximo.
- par_impar.rc: Imprime si un número ingresado por el usuario es par o impar.
- reverse_number.rc: Solicita al usuario un número e imprime el reverso de este.
- fizzbuzz.rc: Juega el juego FizzBuzz hasta el número ingresado por el usuario.

Referencias

- Implementación de árboles n-arios: https://github.com/ChuOkupai/n-ary-tree
- Bibliografía: Ley & Yacc - John R. Levine, Tony Mason & Doug Brown
- Proyectos de ejemplo:
    - https://github.com/faturita/YetAnotherCompilerClass
    - https://github.com/jengelsma/yacc-tutorial
- Videos útiles:
    - https://www.youtube.com/watch?v=54bo1qaHAfk
    - https://www.youtube.com/watch?v=__-wUHG2rfM