Universidad Nacional de San Luis

Práctico de Máquina

San Luis - 2023

Asignatura: Diseño y Construcción de Compiladores

Estudiante: Kwist Daniel

Docentes: Aragón Victoria Soledad

Ferreti Edgardo

Índice

Introducción

El presente informe es realizado con la intención de documentar e informar aspectos sobre las decisiones tomadas en el diseño, desarrollo e implementación del practico de maquina, como tambien asi sobre las experiencias y dificultades en el transcurso del mismo. Parte de la esencia de este trabajo es integrar los conocimientos adquiridos durante el dictado de las unidades temáticas vistas durante el curso de la materia.

El objetivo es implementar un análisis sintáctico mediante un parser descendente recursivo provisto por la cátedra añadiendo una estrategia de recuperación de errores antipánico, para luego realizar un análisis semántico sobre un sistema de tipos provisto implementando, en conjunto de la tabla de símbolos, controles pertinentes.

Durante el informe se hará un recorrido sobre dos ejes principales en los que fue dividido el trabajo, los cuales son el análisis sintáctico y el análisis semántico, previamente mencionados.

En cada uno de estos se detalla información recopilada sobre las decisiones de diseño tomadas, los lotes de prueba que se utilizan y algunos problemas atravesados y las soluciones encontradas.

Desarrollo

El desarrollo de este trabajo comienza mediante los lineamientos de la primera entrega, donde al alumno se le ofrecen un conjunto de archivos, software y manuales con el fin de analizar, instalar y seguir para así dar inicio al trabajo con una base establecida. Además se incluye la gramática en formato bnf y bnfe.

Analizar estos archivos en conjunto con la gramática, y madurar este contenido, conllevo su respectivo tiempo. Por lo tanto al principio fue laborioso entender cómo funcionaba el parser y los demás archivos provistos, como también que cadenas nos permite producir y de qué manera se realizaban las proyecciones de la gramática.

En los archivos nombrados previamente nos encontramos, entre otras cosas, un analizador lexicográfico, un parser descendente recursivo (TopDown), un archivo makefile utilizado para integrar todos los archivos del directorio y ejecutar el compilador, funciones de utilidad y distintos códigos de errores, constantes o funcionalidades extras a utilizar. A esto se le añade en la segunda etapa una actualización a los códigos de errores y archivos referentes a las estructuras utilizadas en el funcionamiento de la tabla de símbolos.

Luego de instalar el compilador ucc provisto por la cátedra, se dio inicio al trabajo de la primera etapa de la cual se habla en el apartado Entrega 1.

Ahora nos concentraremos en algunas consideraciones y detalles para luego referirnos a ambas etapas nombradas previamente.

Como punto de inicio, se decidió crear un archivo batch (run.bat) para facilitar el trabajo de ejecución de los lotes con el compilador, para facilitar las modificaciones y pruebas que se realizaban durante el desarrollo, el mismo contiene 5 comandos claves:

- make clean : Para eliminar los archivos objeto generados y el ucc.
- make : Para compilar y generar el ejecutable ucc .
- ucc -c “lote”.c : Para utilizar un archivo fuente “lote” con el ucc. Este se repite con cada lote de prueba distinto para ejecutar todos los lotes de pruebas disponibles de una pasada.
- PAUSE : Para no cerrar la ventana del cmd (Simbolo de Sistema/Command Prompt)
- DEL ucc.exe : Se decidió incorporar este comando ya que ante algunos casos, al inicio del desarrollo, no se actualizaba correctamente el ejecutable ucc ante un make / make clean.

Además se hace mención a que no se implementó ninguna decisión sobre la utilización de un sistema de debugging más que la utilización de impresión por pantalla (printf) en el compilador. Esto es por la facilidad de uso, rapidez y bajo coste de las impresiones de pantalla, como también al desconocimiento de las posibilidades que ofrece la herramienta que se utilizó para compilar este compilador. Aunque esto tiene sus desventajas ya que era difícil ver paso a paso que sucedía con el compilador y tan solo se podía intentar encapsular de a poco los errores.

Mediante el uso de esta estrategia se intentaba, en la mayoría de casos, como parte inicial encontrar las entradas que generaban errores, para así ubicarlas de a una en un lote de prueba solo y luego empezar a encapsular los procedimientos hasta dar con el causal total o parcial del error, para luego corregir las secciones de código implementadas de manera incorrecta. Esto se realizaba mediante mensajes al entrar en ciertas secciones de código o procedimientos, como también desplegando por pantalla información respecto de las entradas en la tabla de símbolos para corroborar su correcta inserción y consulta de los datos.

En algunos casos, tanto en la primera como segunda entrega, aparecieron errores más difíciles de identificar para los cuales se tuvo que ejecutar a mano en una pizarra para encontrar cual (en el tp1 fue un error en la implementación de la recuperación de errores antipánico y para el segundo tp errores de inserción en la TS). Estas ejecuciones a mano fueron bastante laboriosas, en las cuales era demasiado fácil perderse en la ejecución y llevaba demasiado tiempo, pero permitió ver qué secciones de código incorrectas, procedimientos que se saltaban, y cuales eran llamados y de qué forma, y gracias a esto se encontraban los fallos de manera fehaciente.

Entrega 1

Para esta primera etapa de la construcción del compilador, se comienza con el parser descendente recursivo provisto por la cátedra, al cual hay que añadirle una estrategia de recuperación de errores antipánicos.

Para realizar esta tarea, se comenzó con el desarrollo del ejercicio 11 de la unidad temática 2, con el objetivo de afianzarse con el contenido y archivos, y así utilizarlo de punto de partida para el cumplimiento de la tarea encargada.

Agregando con el ejercicio mencionado anteriormente, se incluyó, a partir de una sugerencia del cuerpo docente, la definición del conjunto FIRST para cada no terminal, para poder simplificar la escritura de los argumentos de los conjuntos tests en la recuperación de errores antipánico, este fue incluido en util.c en conjunto con una declaración de enumeración de los procedimientos.

Luego se incluyó los parámetros folset y, después de un análisis realizado, los test iniciales y finales correspondientes a cada procedimiento. Para finalmente realizar algunas pruebas y corregir según se iban detectaban errores en los argumentos elegidos para los tests.

Un conflicto particular, fue la definición incorrecta de algunos conjuntos first, lo cual conllevo bastante tiempo en búsqueda y análisis hasta que se encontraron los errores de la construcción de estos conjuntos. Particularmente el de Lista de Proposiciones, el cual estaba construido de forma incorrecta por lo que generaba errores inesperados y, además, difíciles de detectar. También se han presentado conflictos con la importación incorrecta de las librerías, entradas forzadas mal realizadas y errores reiterados por la falta de afianzamiento con la gramática. Esto provocó una lentitud para avanzar al inicio.

Para esta presente etapa, aprovecharemos a listar decisiones de diseño tomadas por parte del alumno con su respectiva finalidad, estas son:

1. En lista_declaraciones_param: se agrega en el while el first de declaración parámetro para forzar la entrada
2. En declaracion_parametro: se agrega el CCOR_CIE por si se olvida al usuario la inclusión del mismo.
3. En lista_declaraciones_init: se agrega el first de declarador_init al while para forzar la entrada al while.
4. En declaracion_variable: se agrega el first de lista_declarac_init en el if para forzar la entrada a la sentencia condicional por si el usuario se olvida la coma.
5. En declarador_init: se fuerzan entradas al switch con flotante y carácter para la rama de asignación o con], { y } para la otra rama
6. En lista_inicializadores: se agrega el first de constante al while para forzar la entrada por si el usuario se olvida la coma.
7. En proposición: se agregan para forzar la entrada en entrada y salida el CSHL y CSHR.
8. En proposicion_e_s: se fuerzan respectivamente los CSHL y CSHR donde corresponden por el ítem (7).
9. En expresion_simple: se fuerza en el while con el first de termino por si el usuario se olvida un operando.

10. En termino: se fuerza en el while con el first de factor por si el usuario se olvida un operando.
11. En lista_expresiones: se fuerza el while con el first de expresion por si el usuario se olvida la coma.

Para la creación de lotes propios en esta entrega, se siguió lo sugerido por la cátedra, de incluir lotes sin errores para probar que efectivamente funcionen bien (aquí también hubo problemas de no tener una plena comprensión de la gramática ya que, por ejemplo, en las proposiciones compuestas primero se realizan opcionalmente las declaraciones y luego las proposiciones, estas dos no pueden estar mezcladas, asunto que alumno confundia al inicio con regularidad) y también crear lotes con errores, tales como empezar con 1 solo error, e ir agregando de a poco distintos errores que se querían probar.

Los lotes propios del alumno fueron pequeños y escasos ya que al intentar arreglar algunos errores, la cátedra suministro algunos lotes, con los cuales se terminó de probar y corregir el funcionamiento de la recuperación de errores.

Entrega 2

Comenzando esta segunda entrega, se presentan los lineamientos y se requiere analizar la semántica (encontrada entre los archivos y guías ofrecidas previamente por la cátedra) para la cual se deben seleccionar una serie de controles a realizar.

Como punto mínimo se requieren implementar los controles referidos a las variables, dos de arreglos, dos de asignación, uno de entrada/salida, cuatro de funciones/parámetros (incluyendo además, el ítem 31), uno de lógica booleana y como mínimo coerción en la asignación.

Los controles semánticos seleccionados por el alumno para implementar son los siguientes:

- Variables:
    4. Las variables deben ser declaradas antes de su uso.
    5. Las variables simples no pueden ser declaradas con tipo void.
    6. Las variables tienen alcance de bloque.
- Arreglos:
    7. El tipo base de los arreglos puede ser: char, int o float.
    8. La cantidad de elementos de un arreglo puede estar dada por un número natural (es decir, mayor a 0) y/o a través de la inicialización del mismo.
- Asignación:
    10. Los tipos de ambos lados de la asignación deben ser estructuralmente equivalentes.
    11. No se permite la asignación de arreglos como un todo.
    13. En la inicialización de arreglos en el momento de la declaración se debe chequear que la cantidad de valores sea igual o menor a la constante y que el tipo también se corresponda.

El ítem 11 es realizado de manera indirecta por decisiones de implementación que para utilizar una variable definida como arreglo en una proposición se requiere un índice (excepto cuando se utiliza como parámetro en la invocación de una función).

- Coerciones:
    - Para la asignación
- Entrada/Salida:
    19. Las proposiciones de E/S aceptan variables y/o expresiones de tipo char, int y float.
- Funciones/Parámetros:
    23. El reconocimiento de los parámetros se realiza por posición.
    26. a.La cantidad de parámetros reales debe coincidir con la cantidad de parámetros formales.
    27. El nombre de un arreglo en el parámetro real implica proveer la dirección del arreglo. En este caso el parámetro formal debe ser de nido como pasaje por valor.
    28. No se permite <tipo> & <nombre arreglo> [] en la definición de un parámetro.
    31. main() debe ser un procedimiento sin parámetros que tiene que aparecer exactamente una vez en el programa fuente.
- Logica Booleana
    33. Las condiciones de las proposiciones de selección e iteración pueden ser de tipo char, int y float

Se decidió, de parte del alumno, implementar los controles que se pedían desde el instructivo como mínimo. Previo a esta implementación se realizó un análisis para considerar cuales se pensaban que no insumían demasiado esfuerzo ni modificación de los procedimientos actuales a la hora de integrarlos buscando justamente reducir el esfuerzo de implementación, ya que el tiempo para las tareas es finito.

La idea fue empezar con las que el alumno consideraba que eran más fáciles para finalizar lo requerido lo más rápido posible y desde ahí considerar si se disponía el tiempo y recursos para seguir implementando más controles, pero este no fue el caso ya que estas implementaciones consumieron bastante tiempo y correcciones, y al finalizar, el tiempo se destinó al resto de las actividades de la asignatura.

Cabe destacar que no todos los controles seleccionados fueron precisamente los mas faciles o rapidos, ya que, por ejemplo, se debieron hacer muchas modificaciones y varias iteraciones para completar el ítem 27 el cual el alumno confundia constantemente su propósito y control pertinente, lo que incurrió a varios errores en distribuidos durante todo el desarrollo de la actual entrega y a la inclusión de bastantes controles para los cuales se requirieron guardar los tipos que se heredan y sintetizan de distinta forma y así representar más información sobre el tipo siendo utilizado.

Esta representación fue realizada mediante la decisión de guardar en siete variables globales valores auxiliares para facilitar el uso y comparación de arreglos, variables y strings, estas son:

```
int ARRCHAR = -41, ARRINT = -42, ARRFLOAT = -43, VARCHAR = -11,
VARINT = -12, VARFLOAT = -13, STRING = -14;
```

Se optó por utilizar valores negativos para evitar colisionar con otros valores devueltos por la tabla de símbolos en caso de ser añadidos más tipos en el futuro.

Añadiremos también que, para poder realizar diferentes comparativas, se ha decidido guardar en variables globales, luego de la inicialización de la tabla de símbolos, la ubicación de los punteros a los tipos insertados en tabla de símbolos, para ser utilizados fácilmente en ejecución y evitar consultas a la tabla de símbolos. Estas variables son:

```
int TIPOVOID, TIPOCHAR, TIPOINT, TIPOFLOAT, TIPOARREGLO,
TIPOERROR;
```

Además, se optó por hacer un traceback de los tipos, de tal forma que cuando se identifica una expresión, cada llamado que haga expresión a otro procedimiento, ira sintetizando el tipo heredado de dicho procedimiento (y en caso necesario, se sintetiza TIPOERROR para ser heredado por el resto de procedimientos cuando se encuentra una situación que no es válida semánticamente)

Se añade, a lo mencionado anteriormente, que para controlar en la lista de expresiones (el procedimiento lista_expresiones) encontrada al momento de ser invocada a una función, si se pasa un parametro por direccion, para poder ver si efectivamente corresponde una

variable o el resultado de una expresión se utilizan los valores previamente mostrados correspondientes a VARCHAR, VARINT y VARFLOAT. Esto se hace de la siguiente manera:

```
if(ptr_inf_res->tipo_pje == 'd'){
      if(tipo_parametro_actual == VARCHAR || tipo_parametro_actual
== VARINT || tipo_parametro_actual == VARFLOAT){
          if(tipo_parametro_actual == VARCHAR){
              tipo_parametro_actual = TIPOCHAR;
          }
          else if(tipo_parametro_actual == VARINT){
              tipo_parametro_actual = TIPOINT;
          }
          else if(tipo_parametro_actual == VARFLOAT){
              tipo_parametro_actual = TIPOFLOAT;
          }
      }
      else{
          if((tipo_parametro_actual != ARRCHAR) &&
(tipo_parametro_actual != ARRINT) && (tipo_parametro_actual !=
ARRFLOAT)){
              error_handler(93); //Si el pasaje es por REFERENCIA,
el parametro real debe ser una variable
          }
      }
  }
```

Se decidió hacer uso de esta estrategia, ya que se sintetiza el tipo que se retorna en cada procedimiento (donde se ve involucrado el control de tipo) para ser heredado por otro procedimiento y continuar con su control pertinente.

Para entender un poco más esta situación, supondremos el siguiente ejemplo de cadena de entrada (extraido de CE2Lote7.c):

```
1-void func4(int &a){}
2-
3-void main(){
4-    int b;
5-    func4(b);
6-    func4(2);
7-}
```

La primera invocación a funcion en (5) es correcta, mientras la segunda en (6) no, ya que por mas que sea de tipo entero, al estar recuperando el valor de VARINT a través de los procedimientos involucrados (El camino realizado es: expresión -> expresión_simple -> factor -> variable) se puede corroborar la necesidad de referirse a una variable de tipo entero y no a una constante (o expresión que resulta en entero, como pudiera ser a+2 o 2+2).

Por lo tanto, cuando se encuentra, por ejemplo, un identificador que es un arreglo de enteros, se codifica bajo el tipo -42 para ser decodificado luego (ya que para mantener simplicidad, devolvemos un tipo simple en el retorno de los procedimientos).
Otra decisión del alumno fue, en los casos que se repiten las declaraciones de identificadores, dar de alta el identificador de todas formas, pero utilizando el puntero a tipo de TIPOERROR, así, al volver a utilizar el identificador, no se indica la inexistencia del mismo por un motivo de error de inicialización o declaración. Así también al reconocer un identificador que no se encuentra en tabla de símbolos, se da de alta como TIPOERROR, o también cuando hay errores en la declaración de una variable o función.

También, en lista de inicializadores, al encontrar un valor de la lista que no coincida con el tipo base del arreglo, se opta por detener el procedimiento, devolviendo un valor de -1 (recordar que este procedimiento retorna un número entero indicando la cantidad de elementos presentes en la lista de inicializadores) y cuando se captura en declarador_init se asigna tipo error y el valor de cantidad de elementos en la entrada de la tabla de símbolos será -1.

Además, al ser una gramática de un lenguaje estructurado a bloques, se utiliza un flag para indicar en una proposición compuesta si es necesario hacer un push a la tabla de bloques (cuando se viene desde proposición, debido a que en declaración de función se realiza este push a la tabla de bloques previamente).

Otra consideración fue controlar, además de como indica la guía semántica sobre variables y arreglos, que un parámetro no puede ser de tipo void, este control surgió al momento de leer el error 73.

Un aspecto que quedaba a consideración del usuario, era, en factor, el case CIDENT, si no existe el identificador se da de alta, y si existe el alumno deberá resolver cómo identificar si se trata de una función o una variable. Para esto, se ha optado por resolver la ramificación con la funcion Clase_Ident() provista para verificar el tipo de identificador, si es una funcion se hace llamado a llamada_funcion, y en caso contrario se lo trata como variable (pudiendo ser CLASVAR o CLASPAR, o errores caen en este caso también). Se puede ver en factor:

```
case CIDENT:
   if(en_tabla(sbol->lexema) == NIL){ //en_tabla devuelve NIL (-1)
si no esta en tabla
       error_handler(71); //Identificador no declarado
       strcpy(inf_id->nbre,sbol->lexema);
       inf_id->ptr_tipo = TIPOERROR;
       tipo = TIPOERROR;
       inf_id->clase = -1; //Le asigno una clase incorrecta
       scanner(); //Consumo el identificador
       insertarTS();
   }
   else if(Clase_Ident(sbol->lexema) == CLASFUNC){
       tipo = llamada_funcion(folset);
   }
   else{
```

```
        tipo = variable(folset, necesito_indice);
    }
    break;
```

Se decidió verificar también, que el valor de un índice debe ser de tipo entero, en variable:

```
if(tipoaux != TIPOINT){ //Control propio ajeno al sistema de tipos
    error_handler(103); //Los índices de los arreglos deben ser una
constante entera
}
```

Para la confección de lotes de pruebas de esta entrega se ha decidido crear 8 lotes, en los cuales se intentan controlar la mayoría de posibilidades de errores semánticos para visualizar la respectiva notificación del mismo y así verificar la correcta funcionalidad. Dentro del encabezado de cada lote de prueba se ofrece información sobre la finalidad del respectivo lote.

Se incluyeron los siguientes mensajes de error:

```
case 101:printf("\t Error %d: La cantidad de valores
inicializadores no puede ser 0\n", ne); break;
case 102:printf("\t Error %d: La función main() ya se encuentra
declarada\n", ne); break;
case 103:printf("\t Error %d: El índice de un arreglo debe ser una
constante entera\n", ne); break;
case 104:printf("\t Error %d: Tipo de la asignación no valido\n",
ne); break;
case 105:printf("\t Error %d: Los tipos de ambos lados de los
operadores lógicos o aritméticos deben ser estructuralmente
equivalentes\n", ne); break;
case 106:printf("\t Error %d: Los operandos de los operadores
lógicos o aritméticos solo pueden ser de tipo char, int o
float\n", ne); break;
```

Las altas en la tabla de símbolos se realizan en:

- definicion_funcion: por cada función.
- declaracion_parametro: por cada parámetro de una función.
- declarador_init: por cada variable.
- proposicion_e_s: si se encuentra una variable que no está definida (se inserta como TIPOERROR).
- factor: si se encuentra un identificador que no está definido (se inserta como TIPOERROR).

Conclusiones

Para concluir, por más que el compilador no incluye toda la guía semántica, funciona para los aspectos que se han implementado.

Las decisiones tomadas por parte del alumno dan lugar a la forma en la que funciona el mismo, por lo tanto, puede diferir respecto a haber tomado otras decisiones.

Para ejemplificar, ante identificadores no declarados, se dan de alta como TIPOERROR en vez de notificar que no existen cada vez que se encuentran, o por la forma en la que se fuerzan entradas como con los CSHR y CSHL >> << en Entrada/Salida por si el usuario olvida el cin o cout respectivamente.

Junto con estas y demás políticas detalladas en el apartado anterior, se le da la forma y el funcionamiento que ofrece el compilador desarrollado. Por lo que es el culmine de varias decisiones tomadas durante el transcurso de su construcción.

También, durante el trayecto del desarrollo se han presentado distintas situaciones que han aminorado el mismo, entre ellas nos encontramos con la falta de maduración del contenido por parte del alumno, como también falta de práctica del lenguaje utilizado para construir el compilador, C, el cual es visto por última vez en segundo año.

Esta falta de práctica significó en problemas tales como errores en la asignación de memoria a punteros, o también la incorrecta evaluación de expresiones de, por ejemplo, la sentencia de selección condicional (if), la cual, al pensar que solo evaluaba verdadero toda expresión igual a uno, generó confusiones y errores en el código, ya que el correcto funcionamiento por su definición es evaluar verdadero cualquier valor distinto a cero.

Respecto a la etapa de recuperación de errores también hubo confusiones a la hora de usar e importar librerías correctamente y con la confección de cadenas correctas o los conjuntos firsts para los parámetros folset.

Por otra parte resulta satisfactoria la posibilidad de integrar los contenidos vistos durante el cursado de la asignatura en un trabajo, permitiendo ver directamente la aplicación de los temas estudiados como también extender los conceptos y mejorar la comprensión del contenido. Agregando también la predisposición total desde el cuerpo docente que ha acompañado al alumno durante todo transcurso del desarrollo sirviendo de soporte ante las dificultades que se fueron presentando.

Apartados, Apéndices o Anexos

Cómo apartado, se aprovecha también a hacer mención sobre el software y herramientas utilizadas para el desarrollo:

- Intellij IDEA: Aunque su propósito es trabajar como un IDE (integrated development environment), se ha limitado su uso a procesador de texto, dado a su facilidad para mantener varios archivos abiertos al mismo tiempo y su gama de colores.
- MinGW GNU C/C++: Retomando el ítem anterior, esta herramienta provista por la cátedra es la que se ha utilizado para realizar la compilacion y ejecucion (con lotes) del compilador desarrollado
- git (GitHub): Al trabajar desde dos computadoras, se haría sencillo mantener el código actualizado en cada uno de los ambientes de trabajo mediante un software de control de versiones. Por más que no se utilicen branchs para la integración de trabajo (se mantiene en el branch main todo el desarrollo), sirvió para evitar complicaciones a la hora de traspasar archivos de una computadora a otra y poder controlar los cambios en líneas específicas que se realizaban en alguna actualización. Además, permite mantener el repositorio de manera pública para ser consultado, el cual se encuentra en https://github.com/KwistDaniel/PM-DCC.
- Google Docs: Este software fue utilizado para la redacción del presente informe.
- Gamma.app: Herramienta utilizada para la construcción de la presentación.

Bibliografía

En el presente informe no se han utilizado referencias a bibliografías externas o de terceros.