Trabajos prácticos 3 y 4

Diseño de compiladores I

Integrantes:

- Borgato, Franco Federico - francofederico.6v@gmail.com
- Costes, Juan Mateo - mateocostes@hotmail.com

Grupo n° 18

Docente asignado:

- Marcela Ridao.

Temas asignados:

- 4 8 9 14 17 19 21 25 26

Índice

Correcciones de la entrega anterior

- ***En la salida ponen: Estructuras sintácticas, pero están informando también los tokens detectados, en algunos casos. Sería más correcto decir: Estructuras léxicas y sintácticas, por ejemplo.***
- ***Informa todo como estructuras del parser. Pero si dice se detectó una constante entera, se trata del Léxico no del parser***

Se creó una estructura de lista en la cual almacenamos las estructuras léxicas detectadas para luego ser mostradas.

- ***NO informan los tipos de tokens (solo informan los números que los identifican)***

Se modificó la estructura que almacena los tokens. Se guarda el símbolo que representa a los mismos en vez de su valor numérico.

Errores en el Análisis Léxico

- ***Están informando sólo los números de token. Se pidió informar los tipos de tokens. Ejemplo: identificador, no 257.***

Corrección comentada anteriormente.

- ***Las constantes enteras negativas deben ser informadas como error***
- ***Revisar el tratamiento de constantes negativas: Si aparece una constante positiva, por ejemplo 2., y luego -2., pierden la constante positiva. Y si se coloca una constante negativa grande, la deja en la TdeS cómo -0.0.***

Se replanteo el método “actualizarRango” en el cual actualizamos el rango de las constantes negativas en la gramática, ya que el chequeo del rango lo realizamos en el análisis léxico. En este método para el caso de las constantes negativas enteras, el rango actualizado es el módulo de la misma. Para el caso de las constantes negativas dobles, solo las negamos concatenando el símbolo ‘-’ delante del número.

- ***Si bien detectan correctamente el identificador con más de 25 caracteres, el reporte debe ser como warning, ya que se está truncando, y entregando como identificador válido.***

Se creó una estructura en la cual almacenamos los warnings, para diferenciarlos de los errores.

- ***(Operadores) No se puede chequear sin tener la tabla de equivalencia del número con tipo de token***
- ***(Palabras reservadas) No se puede chequear sin tener la tabla de equivalencia del número con tipo de token***

Corrección comentada anteriormente.

- ***Las palabras reservadas son informadas correctamente, pero también quedan registradas en la TdeS, como si fueran identificadores. No deben ser registradas en la Tabla.***

Las palabras reservadas ya no quedan registradas en la tabla de símbolos.

- ***En el caso de prueba presentado para cadenas, el error de la cadena sin cerrar se informa 2 veces***

Se acomodo un error en la matriz de transición de estados para solucionar este problema.

- ***Como las cadenas se guardan sin las comillas, se confunden con identificadores. Si hay un id con lexema hola, y una cadena 'hola', queda una sola entrada con el lexema hola.***

Se corrigió en el analizador sintáctico agregando un atributo "uso" para cada estructura.

Errores en el Análisis Sintáctico

- ***Informa que se realizó conversión explícita en líneas de asignaciones sin conversión. Esta información está mal ubicada en la gramática***

Se borró dicha notificación ya que había sido colocada en una posición incorrecta.

- ***No reconoce correctamente las expresiones***

Al realizar revisiones y arreglos en descripciones que involucran a las expresiones dentro de la gramática, este inconveniente fue solucionado

- ***La gramática exige que un factor dentro de un término esté entre (). Eso es incorrecto.***

Se eliminó la petición en los paréntesis al utilizar un factor.

- ***No reconoce correctamente las expresiones con conversiones***

Se modificó la descripción de la gramática, ubicando las conversiones a nivel factor.

- ***Break con etiqueta aceptado pero no informado***

Se incorporó un mensaje que informa el break con etiqueta.

- ***No reconoce correctamente el anidamiento de sentencias de control***

Se acomodo el bloque de sentencias en la gramática, permitiendo utilizar varias sentencias declarativas y ejecutables.

- ***Break aceptado pero no informado***

Se incorporó un mensaje que informa el break.

- ***Si se hace una invocación sin discard, da syntax error sin indicar de qué se trata el error****.*

Se incorporó dicho error junto con su descripción. De igual manera, luego de arrojar dicho mensaje del error mencionado, se muestra syntax error ya que dicho caso corresponde a varios errores, se intentó solucionar sin éxito.

Errores:

- ***Si se coloca un '.' solo, lo informa correctamente como un error léxico, pero da syntax error y aborta la compilación. Lo mismo ocurre si aparece la D del exponente, pero no hay dígitos a continuación***

Se agregaron diferentes sentencias para reconocer dichos errores en la gramática, aunque se sigue mostrando syntax error.

Léxicos

- ***Cuando hay un símbolo no reconocido informa que no es reconocido por la gramática. Sería mejor informar como no reconocido por el compilador, o el lenguaje***.

Se cambió dicho mensaje en la acción semántica correspondiente.

Sintácticos

- ***Informa falta de llave de cierre en un bloque que tiene su llave***

Se agregaron diferentes sentencias para reconocer dicho error en la gramática.

- ***Algunos errores los marca como syntax error sin ninguna descripción***

Se agregaron sentencias para reconocer distintos errores en la gramática, de igual manera en algunos casos, sigue marcando syntax error pero con la descripción del error.

- ***Para el caso de las constantes, si bien consideran diferente tipo de token para cte entera que para constante flotante, deben registrar el tipo (i16 o f64) en la entrada correspondiente en la Tabla de símbolos.***

Se registra el tipo ui16 o f64 en la tabla de símbolos para las constantes enteras y flotantes.

- ***Las palabras reservadas no se guardan en la TdeS***

Corrección comentada anteriormente.

Informe

- ***Se indican acciones semánticas que tratan errores. Para los casos en que la acción se encargue de modificar el token con inserción/borrado/reemplazo para entregar un token válido , se debe reportar el problema como warning, no como error***

El token modificado se informa como warning y no como error.

Introducción

En este trabajo se lleva a cabo la generación de código intermedio, además de los chequeos semánticos y la generación de código assembler para un compilador cuyo desarrollo de la parte léxica y semántica, ya fue desarrollada en una anterior entrega.
En el presente informe se darán detalles de las consignas planteadas por la cátedra junto con las soluciones propuestas y el razonamiento necesario para la obtención de dichas soluciones.

Temas Particulares Asignados Práctico 3

Funciones: Registro de información en la Tabla de Símbolos En esta etapa, se deberá registrar: Información referida a la función: o Tipo devuelto o Para los parámetros se deberá registrar el tipo Otros atributos que el compilador requiera para permitir esta funcionalidad en el lenguaje. Código intermedio Se deberá generar código para cada función declarada. El código de cada función se podrá generar en una única estructura, junto con el programa principal, o bien, utilizando una estructura independiente para el programa principal y para cada función. Se deberá generar código para la invocación a cada función, chequeando el tipo de los parámetros en la invocación, así como otros chequeos asociados al uso de dicha función en el contexto en que se está invocando. El pasaje de parámetros será por copia-valor. Se deberá generar el código para el pasaje de parámetros correspondiente a cada invocación.

Tema 9. **discard**

discard ID();

La invocación se efectuará normalmente, pero el valor de retorno de la función será descartado. Ejemplo: discard f1(2); // el valor de retorno de la función f1 será descartado.

Tema 14. **Do until con expresión**

do until ( ) : () ;

El bloque se ejecutará hasta que la condición sea verdadera Si se detecta una sentencia break, se debe salir de la iteración en la que se encuentra. La asignación del final de la sentencia, en caso de estar presente, se deberá ejecutar al final de cada ciclo.

Tema 17. **Break con etiquetado**

Al detectar una etiqueta precediendo a la sentencia de control asignada (temas 11 al 16), y un break con una etiqueta, se deberá proceder de la siguiente forma: Se debe salir de la iteración etiquetada con la etiqueta indicada en el break. Se deberá verificar que exista tal etiqueta, y que la sentencia break se encuentre en la sentencia de control que lleva esa etiqueta, o en una sentencia anidada en la misma. Por ejemplo: outer: while (i < 10) { while (j < 5) { break :outer; //El break provocará salir de la iteración etiquetada con outer }; };

Tema 19. **Diferimiento**

Al detectarse la presencia de la palabra reservada defer al comienzo de una sentencia o bloque de selección o iteración, la ejecución de dicha sentencia o bloque deberá diferirse al final del ámbito donde se encuentra la sentencia diferida.

Tema 21. **Conversiones Explícitas**

Todos los grupos que tienen asignado el tema 21, deben reconocer una conversión explícita, indicada mediante la palabra reservada para tal fin. Por ejemplo, un grupo que tiene

asignada la conversión tof64, debe considerar que cuando se indique la conversión tof64(expresión), el compilador debe efectuar una conversión del tipo del argumento al tipo indicado por la palabra reservada. (en este caso f64). Dado que el otro tipo asignado al grupo es entero (1-2-3-4-5-6), el argumento de una conversión, debe ser de dicho tipo. Entonces, sólo se podrán efectuar operaciones entre dos operandos de distinto tipo, si se convierte el operando de tipo entero (1-2-3-4-5-6) al tipo de punto flotante asignado, mediante la conversión explícita que corresponda. En otro caso, se debe informar error. En el caso de asignaciones, si el lado izquierdo es del tipo entero (1-2-3-4-5-6), y la expresión del lado derecho es de tipo diferente, se deberá informar error por incompatibilidad de tipos.

Descripción de la Generación de Código Intermedio

Para la generación de código intermedio del compilador se utilizó una estructura de fila, en la cual se guardaron tanto los operadores como los identificadores. Cada vez que se leía un identificador se guardaba en la polaca, y cuando se leía un operador también, de tal manera que si por ejemplo tenemos “a + b”, lo que nos quedaría en la polaca sería: a b +. Cuando se debe hacer una bifurcación en una sentencia de control, lo que se hace en la polaca es poner la condición y luego colocar un símbolo especial que indica la misma. Antes de este símbolo, se guarda la dirección de salto. Por ejemplo, si se encuentra un if-else “ if (a > b) { a =: b } else { b =: a} end-if ”, en la polaca quedaría:

a	b	>	11	BF	b	a	=:	13	BI	a	b	=:

Lo que también utilizamos fue la notación posicional del lenguaje YACC cuando queríamos guardar el nombre de un identificador al detectarlo en la gramática. Por ejemplo si en la gramática está definido: factor: ID | CTE
Lo que debemos hacer es guardar el valor de ese factor en la polaca, el cual se accede con el símbolo $1.sval (siendo sval el valor de ese identificador o constante en String).
Por lo tanto nos queda de la siguiente manera:

```
factor: ID { String id = $1.sval; polaca.addElementPolaca(id) }
      |  CTE {String cte = $1.sval; polaca.addElementPolaca(cte) }
```

Para comenzar con esté trabajo, correspondiente al proceso de generación de código intermedio, se agregó información a la tabla de símbolos correspondiente al tipo, uso, y otros atributos que resultan necesarios para el manejo de funciones y otras estructuras. Los mismos, se agregan a través de un diccionario o mapa, al cual agregamos y quitamos

elementos a través de su clave (número de orden en el cual se agrega el elemento en el mapa, vinculado con el nombre del atributo).
Lo primero que se hizo fue declarar una variable “ambito”, la cual se modifica cada vez que se ingresa al programa principal o a una declaración de función. Se renombraron los identificadores ya cargados en la tabla de símbolos utilizando la técnica de Name Mangling vista en la cursada, basada en concatenar los nombres de los identificadores, nombre del programa principal y los nombres de las funciones en las cuales estos fueron declarados separados por ‘.’.
Con esto último podemos reconocer el ámbito de las diferentes sentencias que se reconocen y chequear si una variable o función puede o no ser declarada o utilizada en ese ámbito.
En cuanto al proceso de chequeo de compatibilidad de tipos y conversiones explícitas, debió ser postergado hasta la última etapa de la generación de código.

Manejo de instrucciones de control

Sentencia de selección
Al momento de generar la sentencia de selección, luego de la condición se marca un salto (*) para el caso de que la misma sea falsa luego de generar el código correspondiente al bloque de sentencias ejecutables. Una vez finalizado dicho bloque se apila la dirección de salto mencionada anteriormente. En caso de que haya un ‘else’ se agrega un salto incondicional, marcando la posición del salto de forma semejante a como se mencionó en el caso previo.

* *A la hora de generar una instrucción de salto a una posición donde aún no se ha generado parte del programa, agregamos un espacio en blanco y almacenamos la posición en una pila. Después de esto, continuamos hasta encontrar la instrucción a la cual se debe saltar y guardar esa posición en el lugar en blanco que dejamos con anterioridad.*

Do until con expresión
Cuando se detecta una expresión **Do until**, se guarda la posición de comienzo. Luego de la condición, si la misma es falsa se vuelve a esa posición mediante un salto, después de ejecutar la asignación presente en la expresión. En caso de que la condición sea verdadera, se continúa con la instrucción siguiente a la sentencia de control.
Dentro de esté tipo de sentencias también podrían encontrarse una o más sentencias **BREAK**, las cuales harán que la ejecución del programa continúe en la siguiente instrucción al **UNTIL**(‘condición’).

Declaración e Invocación de funciones
Al momento de detectar una declaración de función se marca la misma con la palabra “fun”, que indica el comienzo de la misma. En el momento que finaliza la función se indica con la

palabra “ret”. Ambas marcas se apilan en la polaca para utilizarse en la generación de código assembler.

Cuando se detecta una invocación, posteriormente a la misma, se almacena en la polaca la palabra “call”.

Sentencia discard

Cuando se detecta una invocación de función junto con la palabra **discard**, se almacena en la polaca la palabra “discard”.

Sentencia de diferimiento

Al igual que la declaración de función, se marca el inicio y finalización de la sentencia de diferimiento con las palabras “defer” y “findefer”, para luego ser utilizadas en la generación de código assembler

Break con etiquetado

En nuestro caso cuando en una sentencia **do until** se detecta un **break con etiquetado**, se almacena en la polaca una posición en blanco junto a una marca de salto incondicional y una vez finalizado el **do until**, se reemplaza la posición en blanco con la dirección del fin del bloque.

Aclaración: en nuestra implementación, funcionan los casos en donde no se encuentra anidamiento de dicha sentencia, ya que por la estructura de nuestra gramática se permite que el break se encuentre solamente al final del bloque de sentencias.

Trabajo Práctico 4

Temas Particulares

División por cero para datos enteros y de punto flotante.
El código Assembler deberá chequear que el divisor sea diferente de cero antes de efectuar una división. Este chequeo deberá efectuarse para los dos tipos de datos asignados al grupo.

Resultados negativos en restas de enteros sin signo.
El código Assembler deberá controlar el resultado de la operación indicada. Este control se aplicará a operaciones entre enteros sin signo. En caso que una resta entre datos de este tipo arroje un resultado negativo, deberá emitir un mensaje de error y terminar.

Recursión en invocaciones de funciones (este tema fue consultado y cambiado por el tema h)
El código Assembler deberá controlar que una función no pueda invocarse a sí misma.

Conversiones explícitas.
Los grupos cuyos lenguajes incluyen conversiones explícitas, deberán traducir el código de las conversiones a código Assembler.

Generación de Código Ensamblador

Para realizar la traducción a código ensamblador se utilizó la clase “Assembler”. La misma está compuesta por un string builder que almacena el código generado a partir de los elementos contenidos en la polaca construida en la etapa anterior.
Como primer medida se genera el código traducido desde el código intermedio, leyendo uno a uno los tokens generados por el parser. Para esto, a medida que se detecta un operador binario o unario (+, -, ...), un comparador (>, =, ...), una asignación, un salto (BF, BI, BT) o demás “marcas” (“CALL”, “FUN”, …) se genera código ensamblador para cada caso. En caso contrario, es decir, que el token sea distinto a lo mencionado, consultamos si es una etiqueta y si no, el mismo se almacena en una pila para luego ser utilizado.

En cuanto a las **condiciones** y los tokens comparadores, decidimos manejarlo de manera tal que al encontrar un token comparador cual sea, se guarda la misma en una variable que presenta la última comparación, actualizado constantemente.
En cuanto al **chequeo de tipos y conversiones**, se utilizó la clase “Tipos”, encargada de devolver el tipo de una operación entre dos operandos, devolviendo el tipo correspondiente a los operandos en caso de que ambos sean del mismo tipo o devolviendo error en caso contrario. En nuestro caso nos tocó el tema de conversiones explícitas de

enteros a flotantes utilizando la nomenclatura “tof64”. En caso de detectar una conversión explícita, se realiza el chequeo de tipos entre el resultado de la expresión convertida explícitamente y el resto de los operandos en la operación. En caso de que la expresión dentro de la conversión explícita contenga operandos de diferente tipos, consideramos necesario que cada operando deba ser convertido explícitamente, en caso contrario se arrojaría incompatibilidad de tipos.

En lo pertinente a la **declaración de funciones**, previamente a iniciar el recorrido de la polaca para generar el código ensamblador, se realiza un recorrido en el cual las declaraciones de funciones almacenadas en la polaca, se trasladan al inicio de la misma, para generar su código correspondiente. Para luego, cuando se detecta una **invocación de función**, reemplazar los parámetros reales sobre los parámetros formales e invocar a la misma, almacenando la variable de retorno en la tabla de símbolos, para luego ser utilizada. En cambio, para el caso de una **invocación discard**, la variable de retorno no es utilizada.

Ante un bloque de sentencias de **diferimiento**, previamente se guardó en la polaca el comienzo y fin del mismo. En esta etapa, se realiza un corrimiento de las sentencias al final del ámbito correspondiente para luego ser concatenado en el string builder.

Cuando se detecta un **break con etiquetado** se genera código Assembler el cual realiza un salto incondicional a la dirección final, almacenada en la polaca, del bloque con dicha etiqueta.

Modificaciones en Etapas anteriores

Además de las modificaciones en el analizador lexico y sintactico por los errores surgidos en las etapas 1 y 2, se realizaron diversos cambios en la conformación de gramática, como por ejemplo: agregado del comienzo y fin de declaracion de funciones, concatenación del ámbito a los identificadores (para su búsqueda en la tabla de símbolos), alteraciones en la ubicación de saltos y direcciones a donde se realizan los mismos, agregado de etiquetas para las sentencias de control, agregado y modificación de atributos en la tabla de símbolos, como la incorporación de los parámetros de una función, entre otros.

Conclusión

En estas dos últimas etapas de generación de código intermedio y assembler se pudo dar un cierre al trabajo propuesto por la cátedra y completar el funcionamiento general de un compilador. Durante el desarrollo del mismo, se debieron realizar modificaciones en etapas anteriores acorde a los nuevos requisitos, y se llevaron adelante las implementaciones necesarias para poder poner en funcionamiento a estos últimos. Además, en el trayecto aprendimos y empleamos diversas herramientas provistas por la cátedra como lo son el yacc y el masm32, lo cual fue una experiencia interesante.
En general todas las etapas de desarrollo llevadas a cabo en este trabajo práctico permiten dar una vista integral de las dificultades de programar un compilador desde cero.