Diseño de Compiladores

Cursada 2019

Grupo N°: 19

Docente asignado: José Massa

Integrantes:

Freiberger Agustín - agustin.freiberger@gmail.com

San Juan Roque - roque.m.sanjuan@gmail.com

Violante Facundo - facundo.violante@gmail.com

Temas asignados:

- Generacion de codigo Intermedio : Tercetos
- Conversiones: Explícitas
- Mecanismo de Traducción : Seguimiento de Registros
- Controles en Tiempo de Ejecución: - División por cero
  - Pérdida de información en conversiones
- Chequeos semánticos: Sólo se permitirán operaciones con operandos de distinto tipo, si se efectúa la conversión explícita.
- Sentencia de control : Do until.
- Registro de información en la Tabla de Símbolos:
  1. Se deberá registrar si se trata de nombres de clases, de atributos, de métodos, de objetos, etc.
  2. Se deberá registrar para variables, atributos, métodos y objetos, el tipo de datos asignado en sus respectivas declaraciones.
  3. Se deberá incorporar la información necesaria para implementar la herencia entre clases.
- Chequeos semánticos:
  1. No se permite más de una variable / clase / atributo / método / objeto con el mismo nombre.
  2. No se permite usar el mismo nombre para una variable / clase / método / objeto o atributo.
  3. No se debe permitir el nombre de uno de los elementos antes mencionados, en una sentencia donde corresponde usar un elemento diferente.
  4. Chequear el uso correcto de atributos y métodos heredados.

- Código intermedio
  1. Se deberá generar código para los métodos declarados dentro de las clases.
  2. Cuando se encuentre la invocación a alguno de los métodos declarados, se deberá generar la invocación al método correspondiente.

Introducción:

El objetivo de este trabajo fue en primera instancia , crear el código intermedio a partir de la gramática generada en la primera entrega. Este,fue implementado a través de tercetos. Como segunda instancia, se generó el código Assembler, utilizando como mecanismo el seguimiento de registros, dando como resultado la compilación completa del código.

Diseño e Implementación:

Generación de Código Intermedio:

Para la generación de los tercetos se crearon 3 clases , por un lado Terceto la cual tiene la estructura (Nro Terceto, Operador , Operando1, Operando2). Esta clase posee todos los get , set y funciones de impresión de la estructura de un terceto.

Por otro lado creamos la clase TercetoOperacion , la cual hereda de Terceto, esta clase posee un tipo y será utilizada para las operaciones matemáticas, de asignación y de comparación, además esta clase implementa la interfaz Asignable.

Asignable fue creada porque los tercetos en sus operandos pueden tener Símbolos o Tercetos, en ambos casos es diferente la forma de obtener el Lexema y el tipo, por lo tanto fue necesaria la utilización de esta interface.

Se utilizó como estructura un ArrayList dinámico al cual se agregan los tercetos dependiendo de la gramática del programa. Para realizar las marcas de salto y sentencias de control , se implementó una estructura de pila auxiliar, ya que es más fácil de implementar y evita los problemas de anidamiento.

En la gramatica , para crear los tercetos , se utilizaron los operadores de posición provistos por Yacc ($$, $n). Al utilizar $n podemos identificar el Token o Símbolo que se encuentra en la posición n de la regla gramatical. Al utilizar $$ le estamos pasando el valor a otra regla que esté utilizando a esta. Tanto $$ como $n son utilizados por el Parser, creando un ParseVal por cada uno, esto nos permite hacer uso por ejemplo de $1.obj , y modificar el objeto que está dentro de ParseVal.

Pseudocódigo para las bifurcaciones de las Sentencias de Control

Sentencia IF

Mientras no leo end_if

si termino condicion

genero bf sin dirección de salto y apilo.

si leo else

genero bi incompleto y label_incio_else.

desapilo, seteo la dirección de salto de bf.

apilo bi.

genero label_fin_de_if .

desapilo.

seteo la dirección de salto .

Sentencia DO UNTIL

Leo do

genero label_inicio_do

Mientras no termina condicion

genero terceto.

guardo en lista.

genero bf sin dirección de salto

apilo

guardo bf en lista de tercetos

genero bi.

seteo dirección de salto de inicio_do a bi.

guardo en lista.

desapilo y seteo dirección de salto.

Seguimiento de Registros

Para hacer el seguimiento de registros se trató a cada registro como un entero que identifica un número de terceto. Para ello, creamos una clase Registros que se encarga de administrar los registros mediante los métodos:

- **ocuparRegistro**: Este método almacena el número de un terceto en un identificador de un registro en la clase Registros. En caso de tratarse de un int, el valor será guardado en los 16 bits menos significativos del registro, pero de todas formas se marcará el registro entero como ocupado. En caso de tratarse de un tipo de dato ulong, será guardado en la totalidad del registro. Para un registro disponible, lo identificamos guardando 0.
- **encontrarTerceto**: este método devuelve el registro donde se encuentra un terceto. Los tercetos ejecutados que deban mantenerse en un registro para su posterior uso son almacenados en un identificador del registro como se explica arriba.
- **estaOcupado**: Devuelve true si el registro que se está consultando tiene algún terceto tanto en los bits menos significativos como en su totalidad

Operaciones entre registros

Multiplicación

Para la multiplicación entre operandos de tipo ulong se almacenó un factor en el registro EAX, y el otro en el registro EBX. Mediante la llamada a MUL se realiza la operación y se guarda el resultado en EAX.

Para realizar esta operación entre números enteros el proceso difiere ya que debe utilizarse la implementación de IMUL. Para esta operación se cargan los 16 bits de entero a multiplicar en AX mientras que se extiende el signo en DX, de esta forma el factor queda comprendido en DX:AX. Luego de cargar el primer operando, se carga el valor por el cual queremos multiplicar este factor en BX y se efectúa el llamado a IMUL con BX. El resultado es almacenado en AX.

División

Primeramente para la división se comparó que el divisor no sea cero mediante la instrucción CMP. Está activa o no el flag equal lo que controla la siguiente instrucción, un Jump Equal a la invocación del mensaje de error correspondiente.

Al igual que con la multiplicación, es necesario discernir si se trata de una división entre tipos int o ulong. Para el caso de enteros, análogamente a la multiplicación, es necesario cargar el dividendo en DX:AX donde se extiende el signo del mismo en DX. El divisor es cargado en BX y se llama a la operación IDIV que realiza la división almacenando el resultado en AX y el resto en DX. Para los ulong, no debemos controlar el signo, pero si se realiza la comprobación de que el divisor no sea cero. Luego, se cargan el dividendo y divisor en los respectivos registros y se procede a invocación de DIV.

Suma y Resta

Para la suma y la resta se utilizan las operaciones SUB y ADD. Las mismas poseen dos operandos que pueden ser Simbolos o otros Tercetos. En caso de ser tercetos que fueron ejecutados con anterioridad, indica que están almacenados en un registro, mientras el símbolo debe ser movido a un registro en caso de que ninguno de los dos operandos esté cargado. Estas operaciones deben verificar el estado de los registros tanto en su ejecución como al final, liberando los registros que no sean necesarios y realizando las operaciones sobre el mismo registro.

Generación de Etiquetas

La generación de etiquetas se realizó mediante marcas en la lista de Tercetos. Es decir, que en cada bifurcación se generó un Terceto Label correspondiente, indicando la posición del terceto en la lista a modo de referencia como operando. Así, un Label que marca la aparición de un else en una sentencia IF quedaría como (“Label”, “11”, -). No sólo en las bifurcaciones se emitieron marcas sino también en el fin mediante el terceto (“FINIF”, “-”,”-”).

Por otro lado se utilizaron Labels para determinar el inicio y fin de una sentencia DO UNTIL. Dichas etiquetas tienen como “operador” los valores “INIDO” y “FIN_DO”.

Estas etiquetas se interpretan directamente desde el generador de código.

TO_ULONG

Pérdida de información en conversiones:

El código Assembler deberá controlar que al efectuar una conversión, no haya pérdida de información del dato convertido. En caso que se produzca tal pérdida, se debe emitir un mensaje de error y finalizar la ejecución.

Para esta implementación se implementó un Label el cual lleva a un mensaje de error el cual notifica que se ha producido una pérdida de información. Luego de esto, el programa termina. En caso de que el entero a convertir no sea negativo, se realiza la conversión de la siguiente manera. En primer lugar se setea en 0 el registro ECX. Luego se carga en CX el entero y posteriormente se realiza un MOV de ECX a EBX. De esta forma, se logra que el entero sea movido a los bits menos significativos de ECX y posteriormente se mueve con los bits más significativos en cero a EBX.

Tema 12: Objetos con Herencia Multiple

Para la implementación de el tema especial asignado, se tuvieron que tener en cuenta muchos factores. En primer lugar, se añadió la siguiente información a la tabla de símbolos:

Uso: Define el uso que tendrá el símbolo en el momento de generar el código. El uso puede ser CLASE(Nombre de clase), OBJETO (Nombre del Objeto de una clase), ATRIBUTO(Variable declarada dentro del alcance de una clase), OBJETO_ATRIBUTO(Atributo utilizado en un objeto perteneciente a una clase), METODO (Nombre del método).

Ambiente: define el ambiente donde se permite el uso de un método atributo o variable. Puede ser GLOBAL si es que fue definido en el bloque declarativo del programa, o en caso de haber sido definido dentro de una clase, tendrá en Ambiente el nombre de dicha clase.

Tipo: Si bien ya teníamos el tipo de las variables como ULONG e INT, se implementó de tal manera que para un Objeto de una Clase, se almacena en Tipo el nombre de la clase de quien fue creado.

Declarado: Indica si el Símbolo fue declarado correctamente o no.

Además de esta información añadida a la tabla de símbolos, fue necesario añadir una lista de Herencias a los símbolos, la misma solo se utiliza en Símbolos cuyo uso sea Clase. En esta lista se guardan los nombres de las clases de las que hereda la clase en cuestión y en

caso de tener herencia sus padres, también se guarda esta información. De esta forma se logra que cada clase tenga todas sus herencias en esta lista.

También se implementó un método DeclaradoYAccesible la cual, además de preguntar si esta declarado el Símbolo, si el ambiente corresponde a al ambiente actual.

Mediante esta información agregada, fuimos capaces de reconocer nuevos tipos de errores, los cuales incluyen:

- Nombres de clases/variables/objetos/métodos repetidos.
- Variables/métodos/objetos/clases no declaradas o fuera de alcance.
- Variables/métodos/objetos/clases redeclaradas.

Para poder utilizar los atributos de clase de cada objeto, fue necesario agregar a la tabla de símbolos nuevos símbolos que nos ayudan a diferenciarlos de un objeto y otro. Aquí se introdujo la existencia de Objeto_Atributo como uso de un Simbolo.

Como el nombre lo indica, son Símbolos que poseen de nombre en primer lugar el atributo a cual pertenecen y el atributo que representan. Estos mismos Objetos_Atributos fueron utilizados en la generación de Data del código assembler.

Salida

El compilador genera el código assembler en el archivo llamado “Salida.asm”, en caso de tener errores , este archivo no se genera.

Se presentan las salidas de errores , los tercetos , y la tabla de símbolos en el archivo llamado “Resultados.txt”.

Conclusiones:

Como conclusión de este trabajo podemos decir que nos sirvió para comprender el funcionamiento profundo de los compiladores que utilizamos en los lenguajes de programación , además de ponernos a prueba en el diseño de un software complejo con muchas funcionalidades.

Se nos presentaron muchas dificultades debido a la capacidad de nuestro compilador de tener que manejar una herencia múltiple, además del tiempo que fue necesario para realizar todas las comprobaciones de su correcto funcionamiento.