

Compiladores e Intérpretes

Trabajo Práctico 3 y 4

Grupo 15

Integrantes:

Maria Eugenia Arriaga marriaga@alumnos.exa.unicen.edu.ar
Lopez Milagros milopez@alumnos.exa.unicen.edu.ar
Rizzalli Bianca brizzalli@alumnos.exa.unicen.edu.ar

Profesora asignada:

Marcela Ridao

Temas asignados:

4 6 7 12 17 23 26 27 30 33

Introducción

El presente informe desarrolla el proceso de construcción de dos etapas del compilador: la generación de código intermedio (Trabajo Práctico No3) y la generación de código Assembler (Trabajo Práctico No4).

En la primera etapa del desarrollo del compilador, trabajamos la generación de código intermedio en formato Polaca Inversa, junto con la detección de errores semánticos y el manejo de ámbitos mediante name mangling, siguiendo las instrucciones dadas por la cátedra.

Según el enunciado del TP3 debíamos generar código intermedio almacenado en una estructura dinámica, cubrir todas las sentencias ejecutables, registrar tipo/uso/atributos de cada símbolo e implementar los chequeos semánticos de:

- variables no declaradas,
- variables redeclaradas,
- funciones no declaradas,
- funciones redeclaradas,
- reglas de alcance y visibilidad.

En la segunda etapa, el objetivo fue generar el código assembler a través del código intermedio generado en la etapa anterior, utilizando variables auxiliares.

Optamos por utilizar Assembler para Pentium de 32 bits, utilizando variables auxiliares, tal como se indica en el enunciado.

En el proceso de traducción nos encargamos tanto de traducir de la polaca inversa a assembler como de incorporar chequeos en tiempo de ejecución, asegurando que el código resultante se ejecute sin errores y detecte correctamente situaciones como divisiones inválidas.

En esta etapa pudimos integrar el análisis semántico con la generación de código de máquina, completando el compilador

A lo largo de este informe, se va a explicar y detallar cada consigna dada por la cátedra, indicando tanto su solución como el razonamiento que se tuvo para llegar a la misma.

Generación de código intermedio

Temas particulares asignados

- **Representación asignada:** Polaca inversa.
- **Tema 12:** while do
while (<condicion>) do <bloque_de_sentencias_ejecutables> ;
El bloque de sentencias ejecutables se ejecutará mientras la condición sea verdadera

- **Tema 17:** Asignaciones que pueden tener menor número de elementos del lado izquierdo

Se deberá generar código para asignar cada elemento de la derecha al correspondiente de la izquierda, en el orden en que se presenten. En caso que del lado izquierdo haya menos elementos que del lado derecho, se descartarán las expresiones sobrantes del lado derecho, y se informará la situación con un Warning. En caso que del lado izquierdo haya más elementos que del lado derecho, se informará como error.

Por ejemplo, para:

```
A,B,C = 1I, 23I, 5I; // Se deberán generar 3 asignaciones: A:=1, B:=23 y C:= 5
I, K = 1I , 23I, 5I; // Se deberán generar 2 asignaciones: A:=1, B:=23 e informar un
warning
I,J,K = 1I , 23I; // Se deberá informar error
```

Se deberá chequear la compatibilidad de tipos en cada asignación individual, en forma independiente, según las reglas detalladas en la sección Chequeo de Compatibilidad de Tipos y Conversiones.

- **Tema 23:** Prefijado Opcional

El prefijado será opcional, tanto para variables locales o no declaradas localmente.

Por lo tanto:

Si una variable no contiene el prefijado, se buscará en el ámbito local y hacia arriba en el árbol de anidamiento, utilizando name mangling

Si una variable tiene prefijado, se debe buscar en el ámbito indicado por el prefijado, chequeando su visibilidad mediante name mangling.

- **Tema 26:** Copia-Valor-Resultado

Casos posibles:

```
<parámetro_formal> es <tipo> ID // semántica por defecto: copia-valor
<parámetro_formal> es cvr <tipo> ID // copia-valor-resultado
```

Cuando se invoque una función, el pasaje de parámetros deberá efectuarse según lo consignado en la declaración: copia-valor o copia- resultado

- **Tema 27:** En Línea

Al detectar una expresión Lambda, con la estructura:

```
<parámetro><cuerpo><argumento>
```

se deberá generar el código que implemente las acciones determinadas por dicha expresión. Ejemplo:

```
(int A){if (A>1I) print ("hola");}(3I) // imprime hola
```

El código generado debe copiar el valor del parámetro formal (3I en el ejemplo), y ejecutar el cuerpo.

Descripción general de la generación de código intermedio

Para realizar la generación de código intermedio la estructura que utilizamos fue un **HashMap**, donde la clave es un String que representa el ámbito y su valor es un ArrayList de String que representa la polaca inversa generada en ese ámbito. De esta forma, podemos almacenar dinámicamente la información de cada función y separarla.

Luego, fuimos guardando las sentencias que leímos en su ArrayList correspondiente, agregando a la polaca los símbolos: identificadores, constantes, operadores, bifurcaciones, entre otros.

Cuando el compilador lee una **asignación**, guarda los identificadores, constantes y “:=” en la polaca, es decir que si leyó “H := 3UL;”, en la polaca aparecerá “H, 3, :=”.

Para las **sentencias de control** utilizamos labels y saltos, ubicando primero la condición y luego la instrucción bifurcación por falso (BF) o bifurcación incondicional (BI) según la sentencia leída.

Por otro lado, cuando se lee el **header de una función**, se modifica el ámbito, se registra tipo/uso/parámetros y se agrega al HashMap la polaca de esta función. Además, se modifica la clave de los parámetros formales, agregándoles el ámbito de la función a la que pertenecen.

También, al leer un **llamado a una función**, se chequea que esté al alcance y se agregan a la polaca los parámetros formales, el identificador de la función y “call”.

Por último, para las **lambdas**, primero se agrega a la polaca la asignación del parámetro y su valor y, luego, el cuerpo de la función.

Descripción detallada de la generación de código intermedio

Lo primero que realizamos en esta etapa del trabajo fue **agregar información a la tabla de símbolos**: tipo, uso y otros atributos necesarios. Para esto utilizamos un HashMap, donde la clave es el nombre del atributo con name mangling y el valor es un ArrayList con la información mencionada. El uso es utilizado para distinguir entre nombre de variables, de funciones y de parámetros.

Realizamos **chequeos semánticos** para evitar conflictos de variables o llamados a funciones. Lo primero que hicimos fue declarar la variable ámbito, que se inicia con el valor “MAIN”, la cual cada vez que se declara una función se actualiza y se anida, y luego se desconcatena cuando termina el bloque de sentencias de dicha función. Por ejemplo, si declaras FUNC1 dentro del main, el ámbito pasaría de “MAIN” a “MAIN:FUNC1”.

Para el analizador sintáctico creamos una nueva tabla de símbolos que copia la información de la tabla de símbolos del analizador léxico, pero le concatena el ámbito al nombre de los identificadores utilizando técnica name mangling para que sean únicos.

Por otro lado, con respecto al chequeo de variables o funciones no definidas, decidimos fijarnos si la tabla de símbolos contenía su nombre como clave y si tenía el uso definido en su valor. De ser así, significa que la variable o función ya fue definida y se puede usar en ese ámbito.

Resolución de temas asignados

- **Tema 12: while do.**

Para llevar a cabo este pedido optamos por agregar **labels y bifurcaciones**, el objetivo es traducir el while a una estructura de saltos en la polaca de esta forma:

LABEL inicio:, condición, BF salto a fin, bloque, BI salto a inicio, LABEL fin:.

Primero, cuando estamos en el header del while, añadimos el “LABEL <número>” a la polaca y guardamos su número en la estructura pilaWhile, de esta forma podemos manejar whiles anidados sin problemas de repetición de labels.

Luego, manejamos la condición de igual manera que en las sentencias if. Agregamos a la polaca las expresiones y luego “cond”, que más tarde será reemplazado por un salto condicional BF.

Cuando finaliza el bloque del while, agregamos el marcador “cuerpo” a la polaca, que luego será sustituido por el salto incondicional al inicio del while (BI). Esto asegura que al terminar las sentencias del bloque se reevaluará la condición.

Por último, creamos el label de salida y, como mencionamos anteriormente, se recorre hacia atrás la polaca en busca del marcador “cond” para reemplazarlo por un salto hacia el label recién creado.

Con este mecanismo podemos manejar whiles anidados y generar la polaca correctamente combinando labels, saltos y marcadores reemplazables.

- **Tema 17: asignaciones múltiples.**

Para poder realizar esto verificamos que el número de elementos del lado izquierdo sea menor al de lado derecho y que las variables estén declaradas. Si se cumplen estas condiciones, podemos recorrer la lista de identificadores e ir agregando a la polaca el identificador con la constante que le corresponde. Si hay más constantes que identificadores, se descartan las que sobran. Es decir, si el código es “X, Y = 1 UL, 2 UL, 3 UL” la polaca va a quedar “X, 1, :=, Y, 2, :=”.

- **Tema 23: prefijado opcional.**

Para este tema lo primero que hicimos fue tener la regla ID PUNTO ID en la gramática, además de la regla ID.

Luego, a la hora de chequear si la variable está al alcance, si tiene prefijado, nos fijamos que el ámbito contenga el prefijo para no buscarla en la tabla de símbolos.

Por ejemplo, si tenemos “FUNC.X” y el ámbito es “MAIN:FUNC1”, la variable está permitida.

- **Tema 26: Copia-Valor-Resultado.**

Para cumplir con lo solicitado incorporamos en los parámetros formales tanto “cvr” (copia-valor-resultado) como “cv” (copia-valor). Luego, durante la generación de código intermedio del llamado a función, identificamos los parámetros declarados con “cvr” y, al finalizar la ejecución de la función, se le asigna a su parámetro real.

Dicha asignación se representa con “<-”, esta decisión será justificada y explicada más adelante en el informe.

- **Tema 27: Lambda.**

Para implementar esta tarea optamos por actualizar el ámbito y generar la polaca inversa de la función lambda en el momento en que se procesa su encabezado. Luego, su bloque de sentencias se maneja de igual manera que los de las funciones normales y se agrega el marcador “returnLambda” al final de la polaca, lo que nos permite manejar adecuadamente su retorno durante la generación de código assembler.

Finalmente, se restaura el ámbito y agregamos a la polaca la asignación de parámetros y el llamado a la función lambda. Dicha asignación fue representada con el símbolo “->”, para luego tratarlo como una asignación de parámetros de una función normal.

Además, para permitir la existencia y el llamado a más de una función lambda en nuestro código, decidimos enumerarlas con un contador global.

Problemas ocurridos y decisiones tomadas

Funciones lambda y Shift/Reduce.

Al diseñar la gramática de las funciones lambda, nos dimos cuenta que la estructura propuesta generaba un conflicto shift/reduce por su similitud a la gramática de los parámetros formales de las funciones normales.

Esto se generaba porque las lambdas comenzaban con “PARENTESISIA TIPO ID”, mientras que los parámetros formales pueden comenzar con “TIPO ID”, y son invocados desde header_funcion, que se define como “PARENTESISIA parametros_formales”. Es decir, ambas construcciones podían empezar con “PARENTESISIA TIPO ID”, haciendo que se genere el conflicto shift/reduce.

Para solucionarlo, optamos por agregar el símbolo “->” al inicio del header de las funciones lambda, de esta manera el parser ya no produce el conflicto mencionado entre ambas estructuras.

Cambio de ámbito en funciones.

Al ingresar al bloque de una función se debe modificar el ámbito antes de ejecutar su bloque de sentencias y, una vez finalizado, restaurarlo.

Sin embargo, en nuestra gramática original, la estructura de la función estaba definida en una sola línea, por lo tanto nos resultaba imposible actualizar el ámbito en el momento correcto. Esto se debía a que el parser primero reducía la regla bloque y luego la regla función, donde se actualizaba el ámbito.

Para solucionarlo, optamos por separar la regla original en dos reglas distintas: función y header_funcion. De esta forma, primero se reduce header_funcion, donde actualizamos el ámbito, luego el bloque con el ámbito correcto, y por último reduce función, donde restauramos el ámbito.

Parámetros formales y Copia-Valor-Resultado.

Como nos pidió la cátedra, los parámetros formales de las funciones deben permitir dos modalidades de pasaje: “cvr” (copia-valor-resultado) y la versión sin prefijo (copia-valor).

En la estructura de la gramática permitimos que los parámetros formales sean tanto “CVR tipo ID” como “tipo ID”. Sin embargo, que se manejen de manera tan distinta complicaba la modificación de sus ámbitos y usos en la tabla de símbolos, generando lógica adicional innecesaria.

Para simplificar esto, decidimos guardar internamente los parámetros sin prefijo con “cv”. De esta forma ambos parámetros se pueden tratar de manera similar, reduciendo la complejidad del código y mejorando su eficiencia.

Llamado a función y CVR

Siguiendo con la problemática anterior, al momento de realizar una llamada a función se debe chequear si sus parámetros formales son de tipo "cvt". De ser así, se debe guardar el valor de los parámetros formales en los reales luego de ejecutarla.

Para llevar a cabo este requerimiento, cuando reducimos llamado_funcion chequeamos si sus parámetros formales son de tipo cvt. De ser así, generamos en la polaca una asignación posterior al llamado de la función, en la que cada parámetro formal se copia en el parámetro real asociado.

Polaca inversa y Main.

En nuestra gramática, lo último en reducirse es prog, que es justamente la encargada de generar la polaca correspondiente al ámbito main. Al igual que nos pasó con las funciones, el problema es que primero se reduce el bloque de sentencias interno y luego prog. Esto implica que cuando ejecutemos las sentencias del main, la estructura de la polaca aún no existe, por lo que no tenemos donde guardarlas.

Nuestra solución fue crear un arreglo global "mainArreglo", en donde fuimos almacenando las instrucciones generadas en el main. Luego, cuando se reduce prog, se inserta este arreglo en la estructura de la polaca.

Además, siguiendo con el mismo problema, no podíamos saber cuál sería el nombre del programa principal hasta después de ejecutar todas sus sentencias. Esto nos imposibilitaba saber el ámbito correcto durante toda la generación de código intermedio.

Como solución, inicializamos la variable global ámbito con "MAIN". De esta forma perdemos el nombre original del programa pero podemos generar el código intermedio sin problemas.

Polaca inversa y errores.

Para evitar la generación de código innecesario en la etapa siguiente, optamos por devolver una polaca nula en caso de encontrar errores durante la generación de código intermedio. Al reducir la regla final (prog), verificamos si hubo errores semánticos, sintácticos o léxicos. De ser así, la polaca inversa se vuelve null.

También, durante la reducción de reglas intermedias, realizamos un mecanismo que consiste en ir devolviendo null a las reglas superiores en caso de haber errores, de esta forma también evitamos agregar sentencias erróneas en la estructura final.

Generación de código Assembler

Temas particulares asignados

Conversiones Implícitas:

(Este tema fue asignado en el TP3 pero resuelto en esta instancia)

El compilador debe incorporar conversiones en forma implícita, cuando se intente utilizar una constante de punto flotante como operando de una expresión o comparación, o como parámetro real de una función.

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. Si esto sucede, se deberá emitir un mensaje de error y terminar la ejecución. Este chequeo deberá efectuarse para el o los tipos de datos asignados al grupo.

Overflow en productos de datos de punto flotante: El código Assembler deberá controlar el resultado de la operación indicada, para el tipo de datos de punto flotante asignado al grupo. Si el mismo excede el rango del tipo del resultado, se deberá emitir un mensaje de error y terminar la ejecución.

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. Si una resta entre datos de este tipo arroja un resultado negativo, se deberá emitir un mensaje de error y terminar la ejecución.

Descripción general de la generación de Assembler

En esta instancia del trabajo lo que se realizó fue: obtener el código intermedio con la forma de polaca inversa generado anteriormente y traducirlo en Assembler para Pentium de 32 bits. Esta etapa genera un archivo .asm con el nombre original del .txt a compilar. Este .asm está listo para ser compilado y ejecutado en sistema operativo Windows.

Descripción detallada de la generación de Assembler

Esta parte del trabajo se implementa en la clase Assembler, que recibe una polaca inversa y una tabla de símbolos (ambas representadas como *HashMap<String, ArrayList<String>>*). A través del método *generarAssembler*, que recibe el path del archivo de salida, se genera un .asm con la traducción de la polaca inversa.

Como primer paso, se realiza una lista lineal de la polaca inversa y se agrega una etiqueta al inicio de cada función para permitir los saltos correspondientes. El código comienza por el main, seguido de un salto incondicional a *FIN* para evitar que, una vez finalizado el main, la ejecución continúe con las funciones declaradas abajo. Luego se procesan las funciones restantes.

Para la generación se utilizaron dos *StringBuilder*:

- **data:** donde se inicializan las variables que necesitan espacio en memoria.

- **code**: que contiene las instrucciones ejecutables: asignaciones, comparaciones, operaciones aritméticas, saltos, conversiones de 64 a 32 bits, llamadas y retornos de funciones, e impresiones.

Se eligió *StringBuilder* por la inmutabilidad de los strings simples, que volvería ineficiente la construcción del archivo.

En la sección data se agregan todas las entradas correspondientes a variables y parámetros de la tabla de símbolos, ya que son las únicas que requieren espacio en memoria. Como el lenguaje solo trabaja con “ulong”, se declararon como DD (32 bits). Las constantes no se declararon aquí, sino que se agregan como auxiliares durante la generación del código.

Para generar las instrucciones se utilizó una pila, que se va completando al leer identificadores, constantes o flags. Al encontrarse un operador o palabra reservada (como print o return), se desapilan los operandos necesarios.

- Las operaciones binarias (aritméticas, comparaciones, asignaciones) desapilan 2 valores.
- Las unarias (llamados a función, saltos, print) desapilan 1.
- *return* desapila todos los valores hasta encontrar un flag.
La asignación múltiple se dividió previamente en asignaciones simples.

Para operar, los operandos se cargan en los registros **EAX** y **EBX**. En algunos casos (por ejemplo, algunas asignaciones), ciertos valores se cargan en registros aunque no se usan directamente, debido que se usó el mismo método general para las operaciones binarias.

Las impresiones se implementaron mediante *MessageBox*, permitiendo mostrar cadenas o valores de registros o memoria.

Las operaciones aritméticas incluyen manejo de errores: si ocurre una condición inválida (como overflow), la ejecución salta directamente a una etiqueta de error definida en un tercer *StringBuilder*, que contiene el mensaje correspondiente y un JMP final para evitar que se impriman otros errores.

Finalmente, las tres secciones (data, code y errores) se combinan en una salida final (*out*), junto con modelo usado, “includes” y encabezados adecuados para generar un programa ejecutable en assembler.

Problemas ocurridos y decisiones tomadas

Return y “empieza lista”

A la hora de generar el código assembler para la sentencia return, detectamos que no sabíamos cuantos de las expresiones o identificadores leídos pertenecían a la lista de elementos que devuelve.

Para solucionar esto evaluamos distintas alternativas, como agregar los paréntesis del return directamente en la polaca, pero optamos por añadir un marcador “empieza lista” antes de agregar los elementos a devolver.

De esta forma, a la hora de traducir a código assembler, cuando se procesa un “return”, el compilador puede extraer elementos de la pila hasta encontrar el marcador “empieza lista” e identificar correctamente los elementos que debe retornar.

Operar en ULONG

Dado que nuestro lenguaje no permite declarar variables de tipo DFLOAT, pero sí exige realizar conversiones implícitas, adoptamos la estrategia de unificar todas las operaciones aritméticas en el tipo ULONG.

Cuando aparece una expresión que involucra valores DFLOAT, primero convertimos cada operando a ULONG y luego realizamos la operación.

De esta forma evitamos mezclar tipos y simplificamos la generación de código, trabajando siempre sobre los registros enteros de 32 bits en lugar de los registros del coprocesador.

Asignación de llamado a funciones

En nuestra gramática no permitimos que una asignación múltiple reciba directamente el resultado de un llamado a función. Por lo tanto, cuando una función es llamada dentro de una asignación a una variable ULONG, aún si dicha función retorna una lista de valores, solo se asigna el primer elemento de esa lista.

Para implementar este comportamiento utilizamos un *flag* especial en la polaca inversa: *reemplazar_<nombreFuncion>*,

que nos permite identificar qué valor debe utilizarse como resultado efectivo de la función (el primero de la lista retornada).

Dado que el código intermedio se construye en un StringBuilder (mutable), para realizar el reemplazo convertimos primero su contenido a String, aplicamos el reemplazo sobre el marcador *reemplazar_<funcion>*, y luego recargamos nuevamente el StringBuilder con el resultado final.

Copia valor resultado

Para implementar el paso por *copia-valor-resultado* en nuestro lenguaje, incorporamos a la polaca inversa un flag especial <->.

Este operador marca el punto en el que, una vez finalizada la ejecución de una función, debe copiarse el valor final del parámetro formal sobre el parámetro real correspondiente.

En el generador de código, cuando encontramos <->, realizamos esta secuencia:

1. Extraemos de la pila el parámetro real y luego el parámetro formal.
2. Cargamos ambos operando.
3. Generamos una instrucción MOV que copia el valor returned desde la variable del ámbito de la función hacia la variable real original.

De este modo garantizamos que, finalizada la función, el parámetro real recibe el valor actualizado del formal, cumpliendo correctamente la semántica CVR.

Conclusión

En conclusión, la implementación de este compilador nos permitió integrar todos los conceptos de la materia en un solo trabajo, desde el análisis léxico hasta la generación de código assembler, implementando cada etapa del proceso de un compilador. A lo largo del desarrollo resolvimos problemáticas sobre ámbitos, alcance, manejo de funciones y lambdas, tipos de pasaje de parámetros, representación intermedia y generación de código Assembler, siguiendo las especificaciones de la cátedra.

En la etapa de generación de código intermedio, organizamos las instrucciones de cada ámbito a través de un `HashMap`, utilizado como Polaca Inversa. Con esta estructura dinámica pudimos diferenciar correctamente los ámbitos, manejar la anidación de funciones y aplicar técnicas de name mangling para garantizar la unicidad de los símbolos.

Además, solucionamos situaciones como conflictos de shift/reduce, la gestión del ámbito main y la manipulación de parámetros `cvr`, e implementamos los chequeos semánticos requeridos: alcance, redeclaración, uso de variables, prefijado opcional, pasaje de parámetros y expresiones lambda.

En la etapa de generación de código Assembler completamos el funcionamiento del compilador. Tradujimos la polaca inversa a instrucciones de Pentium de 32 bits usando registros de propósito general y el coprocesador 80x87 para datos de punto flotante. También incorporamos los chequeos de tiempo de ejecución asignados al grupo, garantizando que el código generado detecte errores como divisiones por cero, overflows y resultados inválidos en restas sin signo. De esta manera, pudimos generar archivos `.asm` ejecutables.

Durante el desarrollo de este compilador tomamos diversas decisiones de diseño como agregar el prefijo “`cv`” a los parámetros formales que no eran “`cvr`”, crear el arreglo para la polaca del main, contadores globales para las funciones lambdas, uso de marcadores para identificar retornos múltiples y asignaciones posteriores a llamados de funciones cuyos parámetros formales son de tipo “`cvr`”. También optamos por reescribir nuestra gramática luego de la entrega del trabajo práctico dos, dado que tenía varios errores que nos resultaban difíciles de encontrar y corregir.

Todas estas decisiones surgieron de conflictos en la gramática y el orden de reducción del parser y nos permitieron obtener un compilador funcional.

Para cerrar, el proyecto nos ayudó a entender en profundidad cómo se diseñan y construyen las distintas etapas de un compilador. El resultado final es un compilador funcional, capaz de procesar un lenguaje de alto nivel, detectar errores en todas sus etapas y generar código assembler ejecutable, cumpliendo con los objetivos planteados por la cátedra.