Diseño de Compiladores

**Trabajo de Cursada - Segunda Parte**

Grupo 20

# Temas particulares asignados

**Práctico 3**

* + - * + Representación intermedia: notación polaca inversa

**Práctico 4**

* + - * + Generación de código assembler: Seguimiento de registros
        + Chequeos en tiempo de ejecución: overflow en sumas y productos

# Introducción

Habiendo realizado en las etapas anteriores del trabajo práctico el analizador léxico y el sintáctico, en estas últimas dos se prosiguió con la generación de código intermedio y la generación de código ensamblador.

En nuestro caso, el código intermedio debía ser generado en notación polaca inversa o notación posfija (desde ahora, RPN), y luego traducirla a código ensamblador, listo para ser compilado a código objeto en formato ejecutable.

# Generación de código intermedio

El código fuente del programa se traduce en una forma más adecuada para el código de la mejora de transformaciones antes de ser utilizado para generar *object code* o código de máquina para una máquina de destino. El diseño de un lenguaje intermedio típicamente difiere de la de un lenguaje práctico de la máquina de tres maneras fundamentales:

1. Cada instrucción representa exactamente una operación fundamental; por ejemplo, “shift-add", modos de direccionamiento comunes en los microprocesadores no están presentes.
2. Información sobre el flujo de control no puede ser incluido en el conjunto de instrucciones.
3. El número de registros del procesador disponible puede ser grande, incluso ilimitada.

Como consigna, debía ser representado en RPN que, simplificadamente, consiste en colocar primero los operandos y luego el operador.

## 

## Polaca Inversa

Su principio es el de evaluar los datos directamente cuando se introducen y manejarlos dentro de una estructura LIFO *(Last In First Out)*, lo que optimiza los procesos a la hora de programar.

Básicamente la diferencias con el método algebraico o notación de infijo es que, al evaluar los datos directamente al introducirlos, no es necesario ordenar la evaluación de los mismos, y que para ejecutar un comando, primero se deben introducir todos sus argumentos, así, para hacer una suma 'a+b=c' el RPN lo manejaría a b +, dejando el resultado 'c' directamente.

Nótese que la notación polaca inversa no es literalmente la imagen especular de la notación polaca: el orden de los operandos es igual en la tres notaciones (infijo, prefijo o polaca, y postfijo o polaca inversa), lo que cambia es que el lugar donde va el operador. En la notación infija, el operador va en el medio de los operandos, mientras que en la notación polaca va antes y en la notación polaca inversa va después. Así pues, "640 / 16" (en notación de infijo), se escribe como "/ 640 16" (en notación polaca) y como "640 16 /" en notación polaca inversa. El orden de los operandos es importante cuando se manejan operadores no conmutativos (como la resta o la división), así, si dividimos 10 entre 2, por ejemplo, en las tres notaciones se debe escribir de la siguiente manera: "10 / 2", "/ 10 2", "10 2 /".

## El Algoritmo RPN

El algoritmo que utilizan las calculadoras RPN es relativamente simple:

* + Si hay elementos en la bandeja de entrada
    - Leer el primer elemento de la bandeja de entrada.
    - Si el elemento es un operando.
      * Poner el operando en la [pila](http://es.wikipedia.org/wiki/Pila_(inform%C3%A1tica)).
    - Si no, el elemento es una función (los operadores, como "+", no son más que funciones que toman dos argumentos).
      * Se sabe que la función x toma n argumentos.
      * Si hay menos de n argumentos en la pila
        + **(Error)** El usuario no ha introducido suficientes argumentos en la expresión.
      * Si no, tomar los últimos n operandos de la pila.
      * Evaluar la función con respecto a los operandos.
      * Introducir el resultado (si lo hubiere) en la pila.
  + Si hay un sólo elemento en la pila
    - El valor de ese elemento es el resultado del cálculo.
  + Si hay más de un elemento en la pila
    - **(Error)** El usuario ha introducido demasiados elementos.

## Sentencia IF

**IF** *< condición >* **THEN** Rule 1 *< bloque >* Rule 2**ELSE** *< bloque >* Rule 3

**Rule 1**

Guarda la dirección de índice donde el BF va a ser guardado

**Rule 2**

Guarda la dirección donde el final del bloque else va a ser guardado. Agrega el jump.

**Rule 3**

Completa la dirección de fin del bloque en el índice que se guardo en la regla 1

## Sentencia FOR

Se aplica una metodología similar a la del IF; Luego de crear la polaca para el encabezado del FOR, se guarda el índice del BF. Se crea la polaca para todas las ejecuciones siguientes y al final se agrega a la polaca para el incremento de la variable utilizada para el control del loop.

# Implementación

El código intermedio es generado por el analizador sintáctico a medida que este consume tokens. Luego, la gramática, se encarga de invocar a las funciones correspondientes que van agregando al código intermedio los elementos.

La traducción a RPN de la mayoría de las sentencias es sencilla: un proceso de apilación permite guardar resultados intermedios para luego ser utilizados. Su implementación no requiere procedimientos extra a la lógica explicada para la RPN en la sección anterior, por lo que no amerita más explicaciones. Sin embargo, para generar sentencias que utilizan saltos, debieron agregarse elementos temporales dentro del código intermedio, de forma que el analizador sintáctico pueda calcular las direcciones de salto dentro de la RPN. A continuación se detallan los algoritmos y la representación de las mismas.

# Generación de Código Assembler

El lenguaje ensamblador refleja directamente la arquitectura y las instrucciones en lenguaje de máquina de la CPU, y pueden ser muy diferentes de una arquitectura de CPU a otra. Cada arquitectura de microprocesador tiene su propio lenguaje de máquina, y en consecuencia su propio lenguaje ensamblador ya que este se encuentra muy ligado a la estructura del hardware para el cual se programa. Los microprocesadores difieren en el tipo y número de operaciones que soportan; también pueden tener diferente cantidad de registros, y distinta representación de los tipos de datos en memoria. Aunque la mayoría de los microprocesadores son capaces de cumplir esencialmente las mismas funciones, la forma en que lo hacen difiere y los respectivos lenguajes ensamblador reflejan tal diferencia.

En esta etapa, se toma como entrada el código intermedio y se genera el código ensamblador para la arquitectura Pentium. El proceso consiste en traducir los elementos de la RPN en una secuencia de instrucciones de máquina adecuada.

Nos tocó el tipo de datos ULONG, por lo que no necesitamos ningún tratamiento especial , ya que estos datos los podemos meter dentro de los registros de 32 bits.

Se dividió la generación del código ensamblador en tres pasos, intercalados con inserciones de partes estáticas del código, como la cabecera o variables auxiliares que están presentes siempre: primero, se inserta la declaración de todas las variables, que representan variables o constantes de la tabla de símbolos, luego se recorre toda la RPN y se generan las etiquetas que marcan las direcciones a dónde se producen todos los saltos y, finalmente, se da paso a la sección principal, que lee cada uno de los elementos de la RPN y los traduce a ensamblador, utilizando donde lo requiera tanto las etiquetas como las variables declaradas.

## Declaración

Para iniciar declaramos las variables que se encuentran en la tabla de símbolos. Para ello, se recorre la misma y se agrega el nombre de variable seguido del tipo y el valor si lo tuviese (las constantes son declaradas con valor).

## Etiquetas

Para agregar etiquetas se recorre la polaca buscando los índices a los que se saltará y se agregan a una lista aparte de la forma “Label\_nro”. Cuando se recorra nuevamente la polaca se agregaran esas etiquetas en el lugar correspondiente.

## Ejecuciones

La declaración del cuerpo principal del programa se realiza en una función cuyo algoritmo es el siguiente:

Por cada elemento del código intermedio

Si no es un **operador**

* Agregar al stack de operandos

Si no

* Realizar la operación correspondiente

**Seguimiento de registros**

Para satisfacer este punto se programó una clase que permite la administración de registros: adquisición, realocación y liberación de los mismos.

**Jerarquía de operadores**

Para la efectiva implementación de los distintos tipos de operadores, aprovechamos la herencia y polimorfismo que provee la orientación a objetos. En el código se puede encontrar una jerarquía que comienza con una clase abstracta que define el método principal que todos los operadores deben implementar (*operate*). Cada una de estas estos operadores es creado por un *factory*.

Un nivel por debajo se pueden encontrar las clases correspondientes a los operadores binarios y unarios, las cuales siguen siendo abstractas pero definen nueva funcionalidad mas especifica, para los binarios se agregaron métodos que permiten el manejo de los operandos (realocación de uno cuando ambos están en memoria, etc.). Los binarios necesitaran dos elementos del stack de operadores, mientras que los uranios solo necesitaran uno.

Dentro de los binarios podemos diferenciar operadores conmutativos y no conmutativos, los cuales se diferencian por el manejo de la situación “variable-registro”.

Por último, en el nivel más bajo se encuentran los operadores particulares que implementan el método operar definido en el padre de la jerarquía.

A continuación se explicara cómo se resolvieron cada uno de los operadores:

* **Asignación**: mueve el valor del segundo operando al primero.

**MOV A , B**

* **BranchFalse**: agrega una etiqueta con el respectivo tipo de salto a la dirección indicada. El tipo de salto lo guarda el comparador.

**JumpType label\_num**

* **Comparadores**: realiza la comparación entre el primer y el segundo operando y guarda el tipo de salto que luego utilizara el BF.

**CMP A , B**

**Store JumpType**

* **División**: en primer lugar se settea el registro “D” en 0 (se lo inicializa, aquí se guardará el resto).Ya que el primer operador debe estar en el registro “A” existe una función que lo reserva. Luego se procede a dividir el registro “A” por el segundo operador.

**reallocateEAX();**

**SUB EDX, EDX**

**IDIV B (with A now in EAX)**

* **Call**: llama a la función que se encuentra apilada en el stack

**CALL label\_functionName**

* **Ret**: cuando es llamado, solo agrega la sentencia ret en el código. Esta sirve para volver debajo del CALL llamado con anterioridad (en el llamado a una función)

**RET**

* **Jump**: Salta incondicionalmente a la etiqueta que indique el operador.

**JMP label\_num**

* **Resta**: substrae del primer operando el valor del segundo.

**SUB A, B**

* **Multiplicación**: realiza el producto entre los dos operadores. Existe una función que adquiere el registro A si el primer operador no se encuentra allí.

**reallocateEAX();**

**IMUL A, B (with A now in EAX)**

**CMP A, 65535 (if multiplication overflowed jump)**

**JG \_overflowed**

* **Print**: agrega la línea que posibilita la impresión a través de un *messageBox* con el parámetro que se encuentra en el *stack* como contenido y título del mismo.

**Invoke messageBox, NULL, addr A, addr A, MB\_OK; A is a variable of a String**

* **Suma**: Añade el segundo operando al primero.

**ADD A, B**

**CMP A, 65535 (if addition overflowed jump)**

**JG \_overflowed**

**Control de overflow en sumas y multiplicaciones**

Se agregaron a las respectivas clases un salto condicional para controlar el overflow, es decir, si el valor sobrepasa el asignado en el práctico 1; en nuestro caso 65535 (2^16-1). Si salta, muestra por pantalla una notificación y finaliza la ejecución.

# Conclusión

La creación del compilador fue una tarea ardua. Aunque la generación de código intermedio fue presta, pasar a assembler fue trabajoso (en parte, ya que fue necesario entender como se trabaja a bajo nivel), y se incurrieron en decenas de errores que se fueron arreglando.

Si bien se intentó realizar un diseño orientado a objetos lo más simple y fiel a la orientación a objetos, se incurrieron en ciertas prácticas que contradicen el paradigma; estos se debe principalmente a necesidades que de no ser resuelta como lo fueron hubieran implicado grandes cambios en el código.

Nos queda como conlusión que algo que parece ser simple (no eran muchas instrucciones, ni operaciones a tener en cuenta), puede llegar a complicarse.