Trabajo Práctico:
Compilador LIS para MVS

Programación Funcional
Tecnicatura en Programación Informática
Universidad Nacional de Quilmes

Introducción

LIS es un lenguaje imperativo simple en el que se pueden manipular valores enteros y booleanos. En LIS puede realizarse cómputo con efectos, almacenando y alterando valores en variables en memoria. Por su parte, *MVS* es una máquina virtual simple que consta únicamente de dos registros enteros, una pila de trabajo y dos memorias, de instrucciones y datos respectivamente. Este trabajo consiste en implementar un compilador de LIS al lenguaje ensamblador propio de la MVS utilizando mónadas para manejar el estado de la compilación. Con el objeto de guiar esta tarea se proveen los siguientes archivos base:

- **Main.hs**, punto de entrada a la aplicación. Permite procesar un archivo de código *.lis* y realiza las operaciones necesarias para compilar y ejecutar el programa. (ARCHIVO COMPLETO)[1]
- **StackL.hs**, implementación de una pila, utilizada por la MVS como stack.
- **StateMonad.hs**, mónada State para realizar cómputos que requieran de un estado general.
- **ParserLib.hs**, biblioteca de funciones auxiliares para realizar el parseo de archivos *.lis*. (ARCHIVO COMPLETO)
- **AssemblyRepresentation.hs**, definición del lenguaje Assembly al cual se compilan los programas LIS. (ARCHIVO COMPLETO)
- **LISRepresentation.hs**, representación de LIS en Haskell como estructura de datos. (ARCHIVO COMPLETO)
- **LISParser.hs**, lexer y parser de LIS para obtener un programa en la representación Haskell a partir de un String. (ARCHIVO COMPLETO)
- **AssemblyExecution.hs**, definición de la máquina virtual y su semántica operacional. (ARCHIVO COMPLETO)
- **LISCompilation.hs**, módulo de compilación de LIS. Recibe un programa LIS en representación Haskell y retorna su representación en lenguaje Assembly.
- **AssemblyOptimization.hs**, módulo de optimización de Assembly. Recibe un programa Assembly, analiza e introduce posibles optimizaciones.
- **Pretty.hs**, biblioteca de funciones auxiliares para representar en pantalla de las estructuras. (ARCHIVO COMPLETO)

[1] Los items marcados como (ARCHIVO COMPLETO) son aquellos cuya implementación provee la cátedra. Los demás se espera que sean completados durante el desarrollo del trabajo práctico.

1. Implementación de la máquina virtual

La semántica operacional de la MVS se encuentra definida en el módulo `AssemblyExecution`. La máquina consta de dos registros (`A` y `B`) y un *Instruction Pointer (IP)* que apunta a la próxima instrucción Assembly a ejecutarse, dentro de la memoria de instrucciones. Una vez identificada la instrucción simplemente de procede a realizar la operación correspondiente actualizando el estado de la máquina. La ejecución termina cuando el IP apunta a una posición de memoria indefinida.

A su vez la MVS posee una memoria de datos que almacena las distintas variables de las que consta el programa, la cual es indexada por nombre de variable. Para facilitar el uso de datos temporarios también se cuenta con una pila o *stack*, donde se pueden almacenar datos sin necesidad de darles un nombre.

Para entender la interacción entre los distintos componentes de la MVS, a continuación se tiene un resumen de sus operaciones y la forma en que son ejecutadas:

- `NoOp`:
  Representa la ausencia de operación. Simplemente incrementa el IP pasando a la siguiente instrucción.
- `Load r n`:
  Carga el valor constante `n` en el registro `r`.
- `Read r v`:
  Carga de la memoria de datos el valor de la variable `v` en el registro `r`.
- `Store r v`:
  Guarda el valor del registro `r` en la posición de memoria correspondiente a la variable `v`.
- `ADD r1 r2`:
  Suma el contenido de los registros `r1` y `r2`, dejando el resultado de la operación en `r1`.
- `ADDmod2 r1 r2`:
  Calcula el resto de dividir por 2 a la suma de los registros (`(r1 + r2) % 2`), almacenando el resultado en `r1`.
- `MUL r1 r2`:
  Guarda en `r1` el resultado de multiplicar el contenido de los regitros.
- `SUB r1 r2`:
  Computa `r1 - r2`, dejando el resultado de la operación en `r1`.
- `DIV r1 r2`:
  Calcula la división enterea entre el contenido de `r1` y `r2`, el resultado es almacenado en `r1`.
- `MOD r1 r2`:
  Calcula el resto de la división entera entre `r1` y `r2`, dejando el resultado de la operación en `r1`.
- `CompEq r1 r2`:
  Compara por igualdad los registros, dejando en `r1` un 1 en caso positivo o un 0 si no.
- `CompGt r1 r2`:
  Realiza la comparación `r1 > r2` codificando el resultado en `r1` como 0 o 1 al igual que `CompEq`.
- `Mark l`:
  Nombra la posición actual con la etiqueta `l`. Operacionalmente se comporta como `NoOp`.
- `Jump l`:
  Actualiza el IP redireccionandolo a la posición marcada con la etiqueta `l`.

- `JumpIfZ r l`: si el contenido de `r` es 0 aputa el IP a la posición marcada con `l`, si no simplemente lo incrementa.
- `Push r`: agrega al tope del stack el contenido del registro `r`.
- `Pop r`: quita el valor del tope del stack y lo guarda en el registro `r`.

1.1. Stack

Al ejecutar la MVS se utiliza un `Stack` para almacenar datos temporarios. Este tipo se utiliza de manera abstracta, con la siguiente interfaz:

1. `empty :: Stack a`
   Expresión que denota un `Stack` vacío.
2. `isEmpty :: Stack a -> Bool`
   Que dado un `Stack` dice si se encuentra vacío.
3. `push :: a -> Stack a -> Stack a`
   Que retorna el `Stack` que resulta de agrega un valor al tope del `Stack` pasado por párametro.
4. `top :: Stack a -> a`
   Que retorna el valor que se encuentra a la tope del `Stack`.
5. `pop :: Stack a -> Stack a`
   Que retorna el `Stack` que resulta de eliminar el valor del tope al `Stack` pasado por párametro.

Ejercicio 1

Completar el módulo `StackL`.

2. Definición de LIS

La estructura de un programa `LIS` queda definida en el módulo `LISRepresentation`. Puede observarse allí que un programa LIS es simplemente una bloque o secuencia de comandos (`Command`), donde éstos pueden ser de asignación, de alternativa condicional, de repetición condicional o simplemente la ausencia de una operación. Los comandos utilizan expresiones que denotan valores enteros (`NExp`) o booleanos (`BExp`). Adicionalmente se tiene el tipo `ROp` que representa las posibles relaciones de comparación entre expresiones enteras.

Para cada comando y expresión es necesario identificar la secuencia de instrucciones Assembly que lo/la implementan. Por ejemplo, dada la declaración de una constante `NCte n`, ésta se traduce a cargar el un registro y guardarlo en el stack para su posterior uso. Es decir, `[Load A n, Push A]`. Notar que la declaración de un valor constante no viene acompañada de un nombre para el mismo. Si quisiéramos en cambio guardarlo en la memoria de datos, necesitaríamos generar un nombre de variable fresco con el cual indexar la memoria. De aquí la utilidad del stack. Todo resultado de una operación que no tenga un nombre asociado debe quedar almacenado en el stack, para poder ser recuperado por las operaciones subsiguientes.

Ejercicio 2

Identificar la secuencia de instrucciones Assembly asociada a cada comando o expresión de LIS.

2.1. La mónada State

Al compilar el lenguaje LIS es necesario llevar cuenta de las distintas etiquetas que se generan para implementar los comandos que controlan el flujo de ejecución, de modo de garantizar que no hay colisión de nombres entre ellas y el código Assembly resultante se corresponde con el programa LIS original.

Con ese fin se introduce la mónada *State*, implementada en el módulo `StateMonad`.

1. `return :: a -> State s a`
   Que toma un valor y lo introduce a un contexto mínimo de la mónada `State`.
2. `(>>=) :: State s a -> (a -> State s b) -> State s b`
   Que extrae el valor de tipo `a` de la mónada y aplica la función propagando el estado.
3. `evalState :: State s a -> s -> a`
   Que dada una mónada y un estado retorna el valor que resulta de evaluar la mónada.
4. `execState :: State s a -> s -> s`
   Que dada una mónada y un estado retorna el estado que resulta de ejecutar la mónada.
5. `getState :: State s s`
   Que denota la mónada que retorna su estado actual.
6. `updState :: (s -> s) -> State s ()`
   Que dada una función `f` retorna la mónada que resulta de actualizar el estado aplicando `f`.

Ejercicio 3

Completar el módulo `StateMonad`.

2.2. Compilando LIS

El proceso de compilación consiste en tomar un bloque de código y traducir cada instrucción que lo compone en la correspondiente secuencia de instrucciones Assembly que implementan dicha instrucción.

2.3. Memoria de compilación

El tipo `Memory` debe ser definido teniendo en cuenta cuáles son los datos que se desean preservar a lo largo del proceso de compilación para garantizar que no hay colisión entre las etiquetas que se van generando. Es primordial tener en claro esta definición antes de comenzar la implementación de las funciones `compile*` (presentadas a continuación). Adicionalmente, es posible que sea necesaria una función auxiliar para generar dichas etiquetas.

Ejercicio 4

Idear e implementar un mecanismo que creación de etiquetas que pueda ser usado en el proceso de compilación, basado en la definición de un tipo `Memory` adecuado y haciendo uso de la mónada `State`.

2.4. Proceso de compilación

Para cada comando o tipo de expresión se tiene una función que realiza este proceso, e interactúan entre ellas logrando el resultado final. Se proporciona el esqueleto de la implementación de las siguientes funciones, las cuales deben ser completadas:

1. `compileBlock :: Block -> State Memory [Mnemonic]`

2. `compileComm :: Command -> State Memory [Mnemonic]`
3. `compileNExp :: NExp -> State Memory [Mnemonic]`
4. `compileBExp :: BExp -> State Memory [Mnemonic]`

Cada una toma un tipo de datos de la representación de LIS en Haskell y retorna el estado general de la compilación luego de haber traducido la instrucción en cuestión. Notar que ya identificamos las instrucciones Assembly a usar en cada caso en el Ejercicio 2.

Ejercicio 5

Completar el módulo `LISCompilation`.

3. Optimizando Assembly

Muchas veces el código Assembly que se obtiene del proceso de compilación resulta ineficiente porque realiza operaciones innecesarias, que no tienen efecto real en el cómputo final del programa en cuestión. Una vez terminado este proceso es posible analizar el código y detectar varias de esas situaciones, como por elemplo un `Push` seguido inmediatamente de un `Pop` al stack, ambos realizados sobre el mismo registro.

Ejercicio 6

Opcional: Analizar posibles optimizaciones de código Assembly e incorporarlas a la implementación del módulo `AssemblyOptimization`.

4. Testeando la compilación

4.1. Implementación de los TADs

Una vez que todas las operaciones estén funcionando, el programa debería capaz de compilar archivos de código *.lis* arbitrarios. Para eso la función `run` del módulo `Main` toma por parámetro el path a un archivo y realiza el proceso completo de parseo, compilación, optimización y ejecución del mismo.

Se adjuntan a los códigos del trabajo práctico una carpeta `samples` con algunos ejemplos de programas que deberían compilar y ejecutar correctamente.

Ejercicio 7

Proponer algunos ejemplos más y probar que todo funciona correctamente. Verificar no solo que el programa compile sin errores, sino también que el resultado de la ejecución sea el correcto.