Parseo y Generación de Código – 2do semestre 2017
Licenciatura en Informática con Orientación en Desarrollo de Software
Universidad Nacional de Quilmes

Práctica 8
Generación de código intermedio

Nota: en los ejercicios siguientes, supondremos que el símbolo no terminal **n** representa algún número natural, y tiene asociado un atributo `valor` que indica su valor numérico.

Ejercicio 1. Dada la siguiente gramática de expresiones aritméticas sobre números enteros:

$$G = (\{E, O\}, \{+, -, *, /, (,), \mathbf{n}\}, P, E)$$

$$\begin{array}{lcl} E & \rightarrow & \mathbf{n} \mid EOE \mid (E) \\ O & \rightarrow & + \mid - \mid * \mid / \end{array}$$

1. Dar una gramática de atributos para calcular el valor numérico de una expresión, utilizando únicamente atributos **sintetizados**.

   Recordar que para esto se debe elegir el conjunto de atributos que se asocia a cada símbolo no terminal de manera conveniente, y agregar acciones semánticas asociadas a cada producción.

2. Dar el árbol de derivación para la expresión $50 - 16/2$, e indicar los valores de los atributos en cada nodo del árbol.

Ejercicio 2. Dada la siguiente gramática $G = (\{E, T, F\}, \{+, *, \mathbf{n}, (,)\}, P, E)$:

$$\begin{array}{lcl} E & \rightarrow & T \mid T + E \\ T & \rightarrow & F \mid F * T \\ F & \rightarrow & \mathbf{n} \mid (E) \end{array}$$

1. Dar una gramática de atributos que asocie a cada símbolo no terminal un atributo sintetizado `post` de tipo *string*, que represente la expresión en notación postfija.

2. Comprobar que `2 * (3 * 4 + 1)` se traduce a `2 3 4 * 1 + *`.

3. Análogamente, dar una gramática de atributos para sintetizar una expresión equivalente en notación prefija, agregando un atributo sintetizado `pre` a cada símbolo no terminal.

Ejercicio 3. Dada la siguiente gramática de expresiones que involucran constantes numéricas (**n**) y variables (**id**) $G = (\{E\}, \{+, *, \mathbf{n}, \mathbf{id}, (,)\}, P, E)$:

$$E \rightarrow \mathbf{id} \mid \mathbf{n} \mid E + E \mid E * E \mid (E)$$

1. Dar una gramática de atributos para sintetizar el AST asociado a una expresión. Suponer que los nodos del AST se construyen así:

   - `Var(`x`)` donde x es el nombre de un identificador, para las variables.
   - `Const(`n`)` donde n es un número, para las constantes numéricas.
   - `Add(`a_1`, `a_2`)` donde a_1 y a_2 son ASTs, para la suma.
   - `Mul(`a_1`, `a_2`)` donde a_1 y a_2 son ASTs, para el producto.

   Suponer que el símbolo no terminal **id** tiene un atributo `nombre` de tipo *string* que indica el nombre del identificador. Por ejemplo, el resultado de procesar la entrada `2 * x + 2 * 3` debe ser:

   ```
   Add(Mul(Const(2), Var("x")), Mul(Const(2), Const(3))).
   ```

2. Dar una gramática de atributos para sintetizar el AST asociado a una expresión, de tal manera que las subexpresiones que *no involucran variables* se precalculen. Por ejemplo, el resultado de procesar la entrada `2 * x + 2 * 3` debe ser:

   ```
   Add(Mul(Const(2), Var("x")), Const(6)).
   ```

3. Dar una gramática de atributos para sintetizar el AST asociado a una expresión, aplicando las siguientes simplificaciones siempre que sea posible:

   $$0 + A = A \quad A + 0 = A$$
   $$0 * A = 0 \quad A * 0 = 0$$
   $$1 * A = A \quad A * 1 = A$$

   Por ejemplo, el resultado de procesar la entrada `(0 + 1) * x + 0 * y` debe ser `Var("x")`.

Ejercicio 4. Recordar las siguientes definiciones vistas en clase, simplificadas ignorando la multiplicación:

- **Arquitectura basada en código de tres direcciones.**

```
type Reg = Int

data Op =
    OpMovInt Reg Int    -- t_i := n
  | OpAdd Reg Reg Reg   -- t_i := t_j + t_k

type Memory = Dictionary Reg Int

run :: [Op] -> Memory -> Memory
run []                    m = m
run (OpMovInt i n : ops) m = run ops (insert i n m)
run (OpAdd i j k : ops)  m =
  run ops (insert i (lookup j m + lookup k m) m)
```

- **Lenguaje de expresiones.**

```
data Expr =
    ExprInt Int
  | ExprAdd Expr Expr

eval :: Expr -> Int
eval (ExprInt n)     = n
eval (ExprAdd e1 e2) = eval e1 + eval e2
```

- **Compilador del lenguaje de expresiones a código para la representación intermedia.**

```
compile :: Reg -> Expr -> [Op]
compile i (ExprInt n) = [OpMovInt n]
compile i (ExprAdd e1 e2) = compile i e1 ++
                            compile (i + 1) e2 ++
                            [OpAdd i i (i + 1)]
```

Demostrar que valen las siguientes propiedades:

1. `run (prog1 ++ prog2) memory = run prog2 (run prog1 memory)`

   Sugerencia: por inducción en la estructura de `prog1 :: [Op]`.

2. Si $j < i$, entonces:

   `lookup (run (compile i expr) memory) j = lookup memory j`

   Sugerencia: por inducción en la estructura de `expr :: Expr`, usando la propiedad 1.

3. `lookup (run (compile i expr) memory) i = eval expr`

   Sugerencia: por inducción en la estructura de `expr :: Expr`, usando las propiedades 1 y 2.

Ejercicio 5. Dada la siguiente expresión:

$$\frac{\sqrt{b^2 - 4ac} - b}{2a}$$

1. Compilarla para una máquina de pila. Asumir que se dispone de operaciones que operan con los elementos de la pila para sumar, restar, multiplicar y tomar raíz cuadrada. Expresar b^2 como $b * b$.

2. Compilarla para código de tres direcciones. Asumir operaciones como en el ítem anterior, pero que operan con registros.

3. Si el programa que se obtiene del ítem anterior requiere más de tres registros, encontrar otra manera de calcular la expresión que utilice a lo sumo tres registros.

Ejercicio 6. Determinar los números de Ershov de las siguientes expresiones, y dar códigos de tres direcciones para calcularlas utilizando la menor cantidad posible de registros. Los operadores $\oplus$ y $\otimes$ representan operadores binarios de los que no asumimos ninguna propiedad específica:

1. $a \otimes (b \otimes (c \otimes d))$
2. $(a \otimes b) \oplus (c \otimes d)$
3. $(a \otimes (b \oplus c)) \oplus ((d \oplus e) \otimes f)$

Ejercicio 7. Considerar una representación intermedia que cuenta con una cantidad fija R de registros $r_1, r_2, \ldots, r_R$, y con una cantidad en principio ilimitada de posiciones en memoria $m_1, m_2, \ldots, m_n, \ldots$. Se cuenta con el tipo `Op` para las instrucciones siguientes; observar que las operaciones aritméticas manipulan únicamente registros:

`OpMovInt` i n	carga una constante numérica en un registro	$(r_i := n)$
`OpAdd` i j k	suma entre registros	$(r_i := r_j + r_k)$
`OpMul` i j k	producto entre registros	$(r_i := r_j * r_k)$
`OpLoad` i j	carga un valor de la memoria en un registro	$(r_i := m_j)$
`OpStore` i j	guarda el valor de un registro en la memoria	$(m_i := r_j)$

Definir una función `compile :: Expr → [Op]` para compilar el siguiente lenguaje de expresiones aritméticas a un programa en la representación intermedia descripta arriba:

```
data Expr =
    ExprInt Int
  | ExprAdd Expr Expr
  | ExprMul Expr Expr
```

Introducir todas las funciones auxiliares que sean necesarias. Se puede suponer que la cantidad de registros no es "ridículamente chica", es decir que se dispone de al menos cuatro o cinco registros. *Nota:* no hay una única manera posible de resolver este ejercicio.

Ejercicio 8. Considerar la estructura de control `for`:

```
for x = expr1 to expr2
    cuerpo
end
```

1. Describir cómo se compilaría esta estructura para una representación intermedia basada en una máquina de pila y para código de tres direcciones. Notar que las expresiones `expr1` y `expr2` se deben evaluar exactamente una vez al comienzo de la ejecución.
2. Indicar en cada caso cómo se compilaría la expresión `x` cuando se la utiliza dentro del cuerpo del `for`.