

## Trabajo práctico especial

### Detalles de la segunda entrega

El proyecto deberá ser capaz de realizar, además de todas las actividades de la primera entrega, la generación de tablas de símbolos, código intermedio sin optimizar, y las siguientes optimizaciones de código: propagación de constantes, simplificación algebraica (elementos neutro y absorbente en las operaciones básicas), eliminación de asignaciones inútiles, eliminación de código muerto, eliminación de subexpresiones comunes, análisis de flujo intra bloque, análisis de flujo interbloque, extracción de invariantes, *loop-unrolling* e *inlining*.

Además, el compilador deberá ser capaz de reportar errores de incongruencia de tipos, y validaciones semánticas con respecto a el tipo y cantidad de parámetros formales en las definiciones parametrizadas.

El jar resultante deberá aceptar los **nuevos** modos de operación:

```
java -jar dcc.jar -T <archivo>
```

Analizará el archivo y emitirá en la salida estándar una tabla de símbolos en la cual se recojan los símbolos reconocidos al procesar el archivo. Dicha tabla tendrá las siguientes columnas: Nombre, tipo, categoría.

Los posibles tipos se corresponden con los del lenguaje o con tipos definidos. Las categorías son: (Var = variable; Fn = función, Pen = parámetro formal entrada, Psa=parámetro formal de salida, res = restricción, typ = tipo definido, com= componente de un tipo definido). Se espera que los datos se muestren ordenados jerárquicamente: los parámetros formales de una función inmediatamente después de esta y en orden; que los componentes de un tipo sigan a la definición del mismo. Si un símbolo depende de otro (ej: parámetro de una función), su nombre deberá estar indentado dos espacios (por nivel de anidación).

Por ejemplo, el siguiente programa:

```
use constraints; // modulo donde se define la restricción 'positive'

initialized:constraint x:int {
    x != 0
}

result:type {
    success:boolean,
    msg:string
}
```

```

function1:function param1:int positive, param2:real -> result:result {
    var1:int;
    function2();
}
function2:function -> result:result {
    var1:int positive;
}

```

Daria la siguiente salida:

res	initialized	
pen	x	int
typ	err	
com	success	bool
com	msg	string
fn	funcion1	
pen	param1	int
res	positive	
pen	param2	real
psal	result	result
var	var1	int
fn	function2	
psal	result	result
var	var1	int
res	positive	

```
java -jar dcc.jar -E <archivo>
```

Analizará el archivo lexica, sintactica y semanticamente, reportando cualquier error o advertencia que encuentre. Si el programa fuera correcto, no se emitirá ninguna salida.

```
java -jar dcc.jar -I <archivo>
```

Analizará el archivo y emitirá código intermedio a la salida estándar, sin optimizar

```
java -jar dcc.jar -I <archivo> [-fxxx -fyyy .... ]
```

Analizará el archivo y emitirá código intermedio optimizado según las técnicas solicitadas con -f (puede haber más de un -f en la línea de comandos)

Los valores posibles de optimización son:

~~-fconstant~~ --> propagación de constantes

~~-fexpr~~ --> simplificación algebraica

~~-fassign~~ --> eliminación de asignaciones inútiles

~~-fdead~~ --> eliminación de código muerto

~~-fcse~~ --> eliminación de subexpresiones comunes

~~-ffax~~ --> extender el análisis de flujo a relaciones entre bloques (opcional, ver

apendice)

~~-floop~~ --> extracción de invariantes y loop unrolling

~~-finline~~ --> inline de funciones

-o --> equivalente a todas las optimizaciones juntas

En todos los casos mencionados, se espera que el programa emita errores lexicos, sintacticos y semanticos de existir. En el caso de encontrar errores, no se deberá generar (y por ende emitir) codigo intermedio.

**Nota:**

Se espera que los parametros de linea de comenados de la primera entrega se comporten de la misma manera. Por ejemplo, el parametro -L solo realizara un analisis lexico y mostrará solo errores de ese tipo.

## Lenguaje intermedio

Para estandarizar los compiladores, todos los grupos utilizarán el mismo lenguaje intermedio, conformado por codigos de tres direcciones, con las siguientes operaciones:

Operaciones simples (de enteros):

$x := y + z$

$x := y - z$

$x := y * z$

$x := y / z$

$x := y \% z$

$x := y$

Referencias y punteros:

$x := \&y$

$x := *y$

$*x := y$

Accesos indexados:

$x := y[z]$

$x[z] := y$

Salto:

goto z

if x = y goto z

```

if x != y goto z
if x > y goto z
if x < y goto z
if x >= y goto z
if x <= y goto z

```

Invocaciones:

```

param x
call y, z , donde y es el simbolo de la funcion a invocar, y z es el número de parametros
ret

```

Independientemente de la representacion interna elegida, la impresion del codigo intermedio debe seguir la nomenclatura mostrada anteriormente. Adicionalmente, cada instrucción podrá tener una etiqueta, de hasta 15 caracteres.

Los operadores se indicaran de la siguiente forma:

Si es una constante, utilizar el numero directamente. Ej: 0, -5, 20

Si es un simbolo (ej: variables), prefijar el nombre con #. ej: #var, #tmp, #par1

Si es una expresión temporal, prefijar con \$. Ej: \$t1, \$t2, \$r1

Si es una etiqueta, encerrar entre parentesis. Ej: (L1), (DEST1).

A modo de ejemplo (considerar que esta no es la unica posible representación), el siguiente fragmento de codigo:

```

a:int;
b:int[4];
c:real;
...
a = 5;
b[2] = 8 * (a + 1);
if (a > 0) {
    a = a + 1;
    c = 3.2;
}

```

Podría mostrar el siguiente codigo intermedio:

```

#a      :=      5
$t1     :=      #a      +      1
$t2     :=      $t1     *      8
$t3     :=      2      *      4  <-- asumiendo que int ocupa 4 bytes
#b[$t3] :=      $t2

```

```
        if #a      >      0      goto (L1)
        goto (L2)
L1:      #a        :=      #a      +      1
        $t4        :=      &#c
        $t5        :=      &#const_3_2
        param $t4
        param $t5
        call assign_real, 2
        goto (L2)
L2:
```

Notar que las operaciones que involucren reales y/o strings deberán ser realizadas a través de invocaciones.

Apendice:

Actualización 16/May/2008:

- La instrucción de código intermedio call es de la forma: call y, z, y se clarifican los parametros
- Se agrega intrucción ret al codigo intermedio
- Desaparece la opcion de optimización -ffa (analisis de flujo intra bloques), dado que es el alcance mínimo de cualquier optimización.
- Se clarifica el significado de la opcion -ffax

Actualizacion 24/May/2007:

- Solo son obligatorios las optimizaciones de propagaciones de constantes y eliminacion de subexpresiones comunes, con alcance intra-bloque.
- Extender dichas optimizaciones a analisis entre bloques (siempre y cuando funcione correctamente) supondrá una bonificación de +2 puntos para la nota de la entrega.