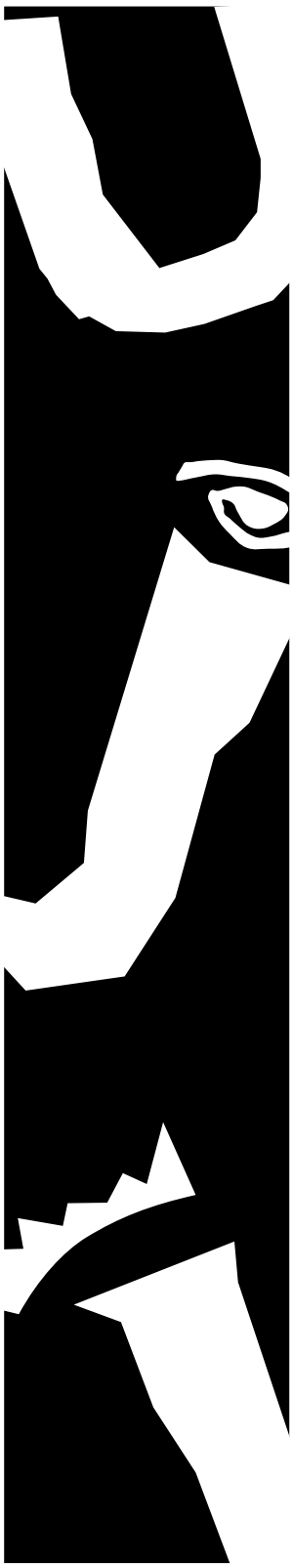

4º Ingeniería Informática

II26 Procesadores de lenguaje

Minicomp, un compilador sencillo

UNIVERSITAT JAUME·I

Índice

© Universitat Jaume I 2006-2007

1. Introducción

Esta práctica nos muestra un compilador para un lenguaje sencillo. Este compilador (`minicomp`) traduce de un lenguaje de alto nivel al lenguaje de la máquina virtual ROSSI.

2. Especificación del compilador

2.1. Descripción informal del lenguaje

Un programa consta de tres partes:

- Una definición de variables globales.
- Una zona de definiciones de funciones.
- Una sentencia compuesta que representa el programa principal.

Las partes primera y segunda son opcionales. La definición de variables globales comienza con la palabra reservada `globales` y termina con la palabra reservada `fin`. Entre ambas se escribe una lista de definiciones, cada una de las cuales consiste en una lista de identificadores separados por comas, un carácter dos puntos, un tipo y un punto y coma.

Los tipos elementales que admite el lenguaje son los enteros y las cadenas. Además, el lenguaje cuenta con tipos compuestos vectoriales: se permite la creación de vectores de cualquier tipo del lenguaje, ya sea elemental o compuesto. Los tipos entero y cadena se representan mediante las palabras reservadas `entero` y `cadena`, respectivamente. Para representar un vector se emplea la palabra reservada `vector` seguida de un número que indica la talla (los índices del vector van de 0 al tamaño especificado menos uno) entre corchetes, la palabra reservada `de` y el tipo base del vector.

Un ejemplo de definiciones globales sería:

```
globales
  x, y: entero;
  nombre: cadena;
  lista: vector [10] de cadena;
  matriz: vector [5] de vector [5] de entero;
fin
```

Después de definir las variables globales, se definen cero o más funciones. Cada función se define mediante la palabra reservada `funcion` (sin acento), seguida del nombre de la función, su perfil, la palabra reservada `es`, una definición opcional de variables locales y una sentencia compuesta. El perfil de la función se escribe mediante una lista de definiciones de parámetros entre paréntesis, un carácter dos puntos y el tipo devuelto. La sintaxis de definición de los parámetros es la misma que la de las variables globales salvo que el punto y coma sirve para separar las definiciones, no para terminarlas. Si la función no tiene parámetros, la lista es vacía.

Un ejemplo de función sería:

```
funcion compara(x,y: entero): cadena
es
  secuencia
    si x<y entonces
      devuelve "menor";
    si_no
      si x>y entonces
        devuelve "mayor";
      si_no
        devuelve "igual";
```

© Universitat Jaume I 2006-2007

```
      fin
    fin
  fin
```

Como se puede ver, la sentencia compuesta comienza por la palabra reservada `secuencia` y termina en `fin`. La sentencia `devuelve` se emplea para volver de la función y su presencia es responsabilidad del programador: el comportamiento de las funciones sin sentencia `devuelve` está indefinido. Para definir variables locales se emplea la palabra `locales` y la misma estructura que para las variables globales. Podríamos haber escrito la función anterior como:

```
funcion compara(x,y: entero): cadena
es
  locales
    resultado: cadena;
  fin
  secuencia
    si x<y entonces
      resultado:= "menor";
    si_no
      si x>y entonces
        resultado:= "mayor";
      si_no
        resultado:= "igual";
      fin
    fin
    devuelve resultado;
  fin
```

Tanto las variables locales como los parámetros y el resultado de la función tienen que ser de tipos simples.

En cuanto a las sentencias, ya conocemos la sentencia compuesta, la sentencia devuelve y la sentencia condicional, cuya parte `si_no` es obligatoria. Además existen las siguientes sentencias:

- Sentencia de escritura: comienza por la palabra reservada `escribe`, seguida de una expresión de tipo simple y un punto y coma. Su ejecución provoca la evaluación de la expresión y la escritura del resultado por pantalla.
- Sentencia nueva línea: consta de la palabra reservada `nl` y un punto y coma. Su ejecución provoca la escritura del carácter nueva línea por pantalla.
- Sentencia de asignación: consta de un identificador, posiblemente seguido de una secuencia de expresiones entre corchetes si se trata de un vector, el signo `:=`, una expresión y un punto y coma. Su ejecución provoca la evaluación de la expresión y la asignación del resultado a la parte izquierda. Sólo se admiten asignaciones entre elementos del mismo tipo, que debe ser simple.

Las expresiones elementales que admite el lenguaje son los literales enteros (secuencias de dígitos), literales de cadena (secuencias de caracteres encerradas entre comillas y sin saltos de línea ni comillas en su interior[1]), accesos a variables (que son identificadores posiblemente seguidos de expresiones encerradas entre corchetes) y llamadas a funciones (que se escriben mediante la palabra reservada `llama`, el nombre de la función y los parámetros de hecho entre paréntesis).

El lenguaje tiene los siguientes operadores, por orden de prioridad creciente:

- Operadores de comparación: `<`, `>`, `<=`, `>=`, `=`, `!=`.
- Operadores aditivos: `+`, `-`.

[1] Tampoco se interpretan de manera especial las secuencias de escape.

- Operadores multiplicativos: `*`, `/`, `%`.

Además, se pueden utilizar paréntesis para cambiar las prioridades.

Todos los operadores admiten únicamente operandos enteros. Los operadores de comparación devuelven resultados de tipo lógico, que es el único admitido en la sentencia condicional, y el resto devuelve resultados de tipo entero.

2.2. Especificación léxica

Utilizaremos la especificación léxica del cuadro 1.

Puedes observar que hemos introducido los comentarios: comienzan con `//` y terminan con el fin de línea. Además, hemos tenido en cuenta que mayúsculas y minúsculas son distintas.

2.3. Especificación sintáctica

La gramática que emplearemos es la siguiente:

⟨Programa⟩ → ⟨Globales⟩?⟨Función⟩*⟨Compuesta⟩
⟨Globales⟩ → **globales** (⟨Definición⟩ **pyc**)+ **fin**
⟨Definición⟩ → **id** (**coma id**)* **dp** ⟨Tipo⟩
⟨Tipo⟩ → **entero** | **cadena** | **vector ca num cc de** ⟨Tipo⟩
⟨Función⟩ → **funcion id** ⟨Perfil⟩ **es** (**locales** (⟨Definición⟩ **pyc**)+**fin**)?⟨Compuesta⟩
⟨Perfil⟩ → **pa** (⟨Definición⟩(**pyc** ⟨Definición⟩)*)? **pc dp** ⟨Tipo⟩
⟨Compuesta⟩ → **secuencia** (⟨Sentencia⟩)* **fin**
⟨Sentencia⟩ → ⟨Compuesta⟩
⟨Sentencia⟩ → **escribe** ⟨Expresión⟩ **pyc** | **nl pyc**
⟨Sentencia⟩ → **si** ⟨Expresión⟩ **entonces** ⟨Sentencia⟩ **si_no** ⟨Sentencia⟩ **fin**
⟨Sentencia⟩ → ⟨AccesoVariable⟩ **asig** ⟨Expresión⟩ **pyc**
⟨Sentencia⟩ → **devuelve** ⟨Expresión⟩ **pyc**
⟨Expresión⟩ → ⟨Comparado⟩(**opcom** ⟨Comparado⟩)*
⟨Comparado⟩ → ⟨Producto⟩(**opad** ⟨Producto⟩)*
⟨Producto⟩ → ⟨Termino⟩(**opmul** ⟨Termino⟩)*
⟨Termino⟩ → ⟨AccesoVariable⟩|⟨Llamada⟩| **num** | **pa** ⟨Expresión⟩ **pc** | **cad**
⟨AccesoVariable⟩ → **id** (**ca** ⟨Expresión⟩ **cc**)*
⟨Llamada⟩ → **llama id pa**(⟨Expresión⟩(**coma** ⟨Expresión⟩)*)? **pc**

2.4. Árboles de sintaxis abstracta

Representaremos el programa mediante un conjunto de árboles: uno por cada función más uno que representa el programa principal.

Los tipos de nodo que se emplearán para representar las sentencias están representados en el cuadro 2. Para las expresiones, emplearemos los nodos del cuadro 3.

Es interesante reflexionar acerca del atributo tipo que hemos incluido en todos los nodos de las expresiones. En principio, podríamos pensar que lo restringido del lenguaje hace que los únicos nodos que en rigor lo necesitarían fueran los de llamadas a función y de acceso a variables y vectores (los operadores aritméticos siempre tienen como resultado un tipo entero y los de comparación uno lógico). Sin embargo, en un compilador más realista, no se podría deducir el tipo de un nodo de

© Universitat Jaume I 2006-2007

Cuadro 1: Especificación léxica de minicomp

Categoría	Expresión regular	Atributos	Acciones
blanco	[␣\t\n]$^+$	—	omitir
comentario	//[^\n]*\n	—	omitir
pyc	;	—	emitir
coma	,	—	emitir
dp	:	—	emitir
ca	\[	—	emitir
cc	\]	—	emitir
pa	\(	—	emitir
pc	\)	—	emitir
asig	:=	—	emitir
cad	"[^"\n]*"	v	calcular v emitir
cadena	cadena	—	emitir
de	de	—	emitir
devuelve	devuelve	—	emitir
entero	entero	—	emitir
entonces	entonces	—	emitir
es	es	—	emitir
escribe	escribe	—	emitir
fin	fin	—	emitir
funcion	funcion	—	emitir
globales	globales	—	emitir
llama	llama	—	emitir
locales	locales	—	emitir
nl	nl	—	emitir
secuencia	secuencia	—	emitir
si	si	—	emitir
si_no	si_no	—	emitir
vector	vector	—	emitir
id	[a–zA–Z][a–zA–Z0–9_]*	lexema	copiar lexema emitir
num	[0–9]$^+$	v	calcular v emitir
opcom	[<>]=?|!?=	lexema	copiar lexema emitir
opad	[-+]	lexema	copiar lexema emitir
opmul	[*/%]	lexema	copiar lexema emitir
eof	$^{e}_{f}$	—	emitir

Cuadro 2: Nodos empleados para representar sentencias.

Tipo de nodo	Representa	Atributos	Hijos
`Asignación`	sentencia asignación	—	las expresiones del valor-i y del valor por asignar
`Devuelve`	devolución de un resultado en una función	la función	la expresión devuelta
`Si`	sentencia condicional	—	la expresión de la condición, las sentencias correspondientes al `entonces` y al `si_no`
`Escribe`	escritura de una expresión	—	la expresión
`Compuesta`	sentencia compuesta	—	las sentencias que la componen

Cuadro 3: Nodos empleados para representar expresiones.

Tipo de nodo	Representa	Atributos	Hijos
`Comparación`	una comparación	la operación, el tipo	los operandos
`Aritmética`	una operación aritmética	la operación, el tipo	los operandos
`Entero`	un entero	el valor, el tipo	—
`Cadena`	una cadena	el valor, el tipo	—
`Llamada`	una llamada a función	la función, el tipo	las expresiones de los parámetros de hecho
`AccesoVariable`	acceso a una variable	la variable, el tipo	—
`AccesoVector`	acceso a un vector	el tipo	el vector, la expresión con el índice

manera directa (excepto en los casos de los nodos `Entero` y `Cadena`). Por otro lado la uniformidad facilita las comprobaciones de tipos. Por ello, hemos decidido poner el atributo en todos ellos.

Además, tenemos un tipo de nodo para representar las funciones, el `Función`. Como atributos, tendrá el identificador de la función, sus listas de variables locales y de parámetros y el tipo devuelto, además de un atributo tipo que indica que se trata de una función. Tendrá como hijo la sentencia compuesta que representa su cuerpo.

Finalmente, hemos tomado la decisión de crear árboles incluso para programas que tengan errores. Esto hace necesario crear un nodo especial, al que llamamos `Vacío` y que representa las construcciones mal formadas encontradas en la entrada.

3. Implementación

Hemos dividido la implementación en distintos ficheros:

- `AST.py` contiene las clases para representar los nodos del AST, excepto los nodos de las funciones.
- `cadenas.py` contiene la clase que se utilizará para representar las cadenas del programa.
- `errores.py` contiene funciones auxiliares para unificar el tratamiento de errores.
- `errsintactico.py` contiene funciones auxiliares para el tratamiento de errores sintácticos.
- `etiquetas.py` contiene la función que genera etiquetas para el correspondiente código ROSSI.
- `funciones.py` contiene la clase que se utiliza para representar las funciones.

© Universitat Jaume I 2006-2007

- `gencodigo.py` contiene las funciones que llaman a los métodos de generación de código del programa principal y las funciones, además de generar las instrucciones de inicialización y finalización.
- `memoria.py` contiene las funciones que asignan posiciones de memoria a las variables globales así como representaciones de las direcciones utilizadas como registros.
- `minicomp.mc` es el fichero de entrada para `metacomp`.
- `registros.py` contiene las funciones para controlar la reserva de registros.
- `Rossi.py` contiene clases para representar las instrucciones de la máquina virtual ROSSI.
- `TDS.py` contiene las estructuras que representan la tabla de símbolos.
- `tipos.py` contiene las clases empleadas para representar tipos.
- `variables.py` contiene la clase empleada para representar las variables del programa.

Para crear el fichero ejecutable del compilador, se empleará `metacomp`:

```
metacomp -s minicomp minicomp.mc
```

3.1. Analizador léxico

La parte correspondiente al analizador léxico es bastante breve:

```
1  None    None         [ \t\n]+
2  None    None         //[^\n]*\n
3  cad     trataCad     "[^"\n]*"
4  id      trataId      [a-zA-Z][a-zA-Z0-9_]*
5  num     trataEntero  [0-9]+
6  opcom   None         [<>]=?|!?=
7  opad    None         [-+]
8  opmul   None         [*/%]
```

Hemos empleado la capacidad de `metacomp` para representar directamente categorías con un solo lexema (escribiendo el lexema entre comillas directamente en las reglas) para ahorrarnos muchas categorías "auxiliares" como **coma** y **pyc**. Además, hemos incluido el reconocimiento de las palabras reservadas en el tratamiento de los identificadores:

```
27  _reservadas=ImmutableSet(["cadena", "de", "devuelve",
28          "entero", "entonces", "es", "escribe", "fin",
29          "funcion", "globales", "llama",  "locales",
30          "nl", "secuencia", "si", "si_no", "vector"])
31
32  def trataId(c):
33    if c.lexema in _reservadas:
34      c.cat= c.lexema
```

Fíjate cómo aprovechamos el atributo `cat` que utiliza `metacomp` para cambiar la categoría del componente. Además, hemos utilizado un `ImmutableSet` de palabras reservadas, y no una lista, para hacer que la comprobación resulte más eficiente.

Para el tratamiento de errores léxicos, hemos utilizado una función prácticamente idéntica a la del manual:

```
42  def error_lexico(linea, cars):
43    if len(cars)> 1:
44      if len(cars)> 10:
45        cars= cars[:10]+"..."
```

```
46        errores.lexico("No he podido analizar la cadena %s." % repr(cars), linea)
47      else:
48        errores.lexico("No he podido analizar el carácter %s." % repr(cars), linea)
```

Observa que hemos añadido un test para evitar escribir cadenas muy largas.

3.2. Analizador sintáctico

Respecto a la organización del analizador sintáctico no hay nada especialmente destacable, ya que nos hemos limitado a transcribir la gramática. En cuanto al tratamiento de errores, hemos incluido reglas de error en diversos no terminales. La política de tratamiento de errores ha sido sincronizar con los primeros y siguientes del no terminal que trata el error. Si el componente con el que se ha sincronizado ha sido uno de los primeros, se reintenta el análisis; si ha sido uno de los siguientes, se da un valor adecuado (ficticio) a los atributos sintetizados. Para implementar más fácilmente esta política, se ha utilizado un módulo auxiliar, `errsintactico`, como se explica en la sección 3.11.5.

Además, hemos introducido tratamiento para el error que se produce si aparece entrada después del fin del programa:

```
52  $errores.sintactico("He encontrado entrada después del último fin.", mc_al.linea())$
53  $mc_al.sincroniza(["mc_EOF"])$
```

3.3. Análisis semántico

Siguiendo la estructura habitual, hemos divido el análisis semántico en dos partes. Las acciones semánticas del esquema de traducción se han destinado a construir un AST. Sobre el AST se hacen muchas de las comprobaciones utilizando el método `compsemanticas` de cada nodo. Un ejemplo típico de construcción del árbol es la parte del esquema correspondiente a las comparaciones:

```
196  <Expresion> ->
197    <Comparado> @arb= Comparado.arb@
198    ( opcom <Comparado>
199      @arb= AST.NodoComparacion(opcom.lexema, arb, Comparado_.arb, opcom.nlinea)@
200    )*
201    @Expresion.arb= arb@
202    ;
```

Para guardar la información relativa a variables y funciones se emplean objetos que representan estas entidades. Así, al definir una variable, se crea un objeto de la clase `Variable` que será el que almacene el tipo, la talla, un *flag* que indica si es o no local y, más adelante, la dirección.

El código correspondiente a la definición es:

```
83  <Definicion> ->
84    id @l=[id]@
85    ( "," id @l.append(id2)@ )*
86    ":" <Tipo>
87    @for id in l:@
88    @  v= variables.Variable(id.lexema, Tipo.tipo, TDS.enFuncion()!= None, id.nlinea)@
89    @  Definicion.variables.append(v)@
90    @  TDS.define(id.lexema, v, id.nlinea)@
91    ;
```

Como se explica en 3.10, la propia tabla de símbolos anuncia el error correspondiente a definiciones repetidas.

Con el objetivo de evitar una proliferación de mensajes, se ha hecho que los accesos a variables no definidas provoquen el correspondiente error y una definición de la variable con tipo error, que es compatible con cualquier otro tipo:

© Universitat Jaume I 2006-2007

```
239   <AccesoVariable> ->
240     id
241     @if not TDS.existe(id.lexema):@
242     @  errores.semantico("La variable %s no está definida." % id.lexema, id.nlinea)@
243     @  v= variables.Variable(id.lexema, tipos.Error, TDS.enFuncion()!= None, id.nlinea)@
244     @  TDS.define(id.lexema, v, id.nlinea)@
245     @arb= AST.NodoAccesoVariable(TDS.recupera(id.lexema), id.nlinea)@
246     (
247       "[" @nlinea= mc_al.linea()@ <Expresion> "]"
248       @arb= AST.NodoAccesoVector(arb, Expresion.arb, nlinea)@
249     )*
250     @AccesoVariable.arb= arb@
251     ;
```

Otra comprobación semántica que se hace antes de construir el árbol es si la sentencia `devuelve` aparece dentro de una función. Para ello, se emplea la función `enFuncion` de la tabla de símbolos. Si estamos en una función, `enFuncion` devuelve el objeto correspondiente, en caso contrario, devuelve `None`. La regla correspondiente es:

```
179   <Sentencia> ->
180     devuelve <Expresion> ";"
181     @f= TDS.enFuncion()@
182     @if not f:@
183     @  errores.semantico("Sólo puede aparecer devuelve dentro de una función.",@
184     @    devuelve.nlinea)@
185     @  f= TDS.recupera(funcionError)@
186     @Sentencia.arb= AST.NodoDevuelve(Expresion.arb, f, devuelve.nlinea)@
187     ;
```

Como puedes ver, en caso de error, se crea el nodo como si la sentencia perteneciera a una "función error". Esto nos permite realizar posteriormente las comprobaciones semánticas sobre la expresión.

3.4. Estructuras globales

Durante la compilación se mantienen tres listas globales que contienen las funciones definidas en el programa, las variables globales y las cadenas. La lista de funciones se emplea durante la fase de generación de código para generar cada una de ellas. La lista de variables globales se emplea para asignarles una dirección de memoria a los objetos correspondientes (las funciones guardan listas similares con sus variables locales y parámetros). Finalmente, la lista de cadenas se emplea para generar las correspondientes inicializaciones. En la sección 3.8 se explica con más detalle cómo se representan las cadenas.

La inicialización de estas variables se hace en la función `inicializaGlobales`:

```
271   def inicializaGlobales():
272     global Funciones, VariablesGlobales, Cadenas, funcionError, cadenaNL
273     Funciones= []
274     VariablesGlobales= []
275     Cadenas= []
276     cadenaNL= cadenas.Cadena("\n")
277     Cadenas.append(cadenaNL)
278     funcionError= "#ferror"
279     ferror= funciones.NodoFuncion(funcionError,0)
280     ferror.fijaPerfil([], tipos.Error)
281     TDS.define(funcionError, ferror, 0)
282     errsintactico.inicializa(mc_primeros, mc_siguientes)
```

Emplearemos la `cadenaNL` para implementar la instrucción `nl`. Para ello, utilizaremos un nodo `Escribe` con esta cadena:

```
163   <Sentencia> ->
164     nl ";"
165     @Sentencia.arb= AST.NodoEscribe(AST.NodoCadena(cadenaNL, nl.nlinea),nl.nlinea)@
166     ;
```

Como puedes ver, tenemos también una "función error". Ésta es la función que se emplea cuando se detectan llamadas a funciones no definidas o sentencias `devuelve` fuera de funciones. El nombre que le hemos dado garantiza que no puede coincidir con ninguna creada por el usuario. El código correspondiente a las llamadas es:

```
253   <Llamada> ->
254     llama id
255     "("
256       @l=[]@
257       ( <Expresion> @l.append(Expresion_.arb)@
258         ( "," <Expresion> @l.append(Expresion_.arb)@)*
259       )?
260     ")"
261     @if not TDS.existe(id.lexema):@
262     @  f= TDS.recupera(funcionError)@
263     @  errores.semantico("La función %s no está definida." % id.lexema, id.nlinea)@
264     @else:@
265     @  f= TDS.recupera(id.lexema)@
266     @Llamada.arb= AST.NodoLlamada(f, l, llama.nlinea)@
267     ;
```

La última línea de `inicializaGlobales` permite que `errsintactico` tenga acceso a las tablas de primeros y siguientes.

3.5. Implementación de los AST

Los distintos nodos del AST se han representado de una manera bastante uniforme. Se ha hecho que sean clases derivadas de la clase `AST` definida en el fichero `AST.py`. Todos los nodos tienen un método de inicialización que, aparte de la información propia del nodo, guarda el número de línea para identificar la posición de posibles errores. Además, tienen los métodos `compsemanticas`, `generaCodigo` y `arbol`. Los nodos de tipo lógico (en nuestro caso, sólo las comparaciones) tienen el método `codigoControl` en lugar del `generaCodigo`.

El método `compsemanticas` realiza las comprobaciones semánticas oportunas, visitando a los hijos si los hay y emitiendo los mensajes pertinentes. Se ha hecho que, en los nodos correspondientes a expresiones, sea este método el encargado de inferir el tipo y guardarlo en el atributo tipo. Como ya se ha comentado, para evitar un exceso de mensajes de error, se emplea un tipo especial cuando ha habido errores.

El método `generaCodigo` es el encargado de generar el código correspondiente al nodo. La emisión de instrucciones se realiza añadiéndolas a la lista que recibe como parámetro. Por ejemplo, la generación de código para el nodo correspondiente al `si` es:

```
78    def generaCodigo(self, c):
79      c.append(R.Comentario("Condicional en línea: %d" % self.linea))
80      siguiente= etiquetas.nueva()
81      falso= etiquetas.nueva()
82      self.cond.codigoControl(c,None,falso)
83      self.si.generaCodigo(c)
```

© Universitat Jaume I 2006-2007

```
84        c.append(R.j(siguiente))
85        c.append(R.Etiqueta(falso))
86        self.sino.generaCodigo(c)
87        c.append(R.Etiqueta(siguiente))
```

Hemos hecho que `R` tenga una referencia al módulo `Rossi` (consulta la sección 3.11.1) para simplificar la escritura de las instrucciones.

Para la reserva de registros se ha utilizado la misma estrategia que se comenta en los apuntes. Los manejamos de manera similar a una pila. Por ejemplo, en el caso de las operaciones aritméticas

```
188     def generaCodigo(self, c):
189       iz= self.izdo.generaCodigo(c)
190       de= self.dcho.generaCodigo(c)
191       if self.op== "+":
192         c.append(R.add(iz, iz, de))
193       elif self.op== "-":
194         c.append(R.sub(iz, iz, de))
195       elif self.op== "*":
196         c.append(R.mult(iz, iz, de))
197       elif self.op== "/":
198         c.append(R.div(iz, iz, de))
199       elif self.op== "%":
200         c.append(R.mod(iz, iz, de))
201       registros.libera(de)
202       return iz
```

Guardamos los resultados de los operandos izquierdo y derecho en sendos registros. Después hacemos la operación sobreescribiendo el registro donde está el operando izquierdo y liberamos el derecho. El resultado de `generaCodigo` es el registro donde se encuentra el resultado (en nuestro caso, el que contenía el operando izquierdo).

El método `codigoControl` es análogo al `generaCodigo`, pero recibe dos parámetros: las etiquetas a las que saltar en función de si la comparación es cierta o no. Seguimos el convenio de que si alguno de ellos es `None`, el salto es a la instrucción siguiente.

Para generar código en las asignaciones y los accesos a vector, empleamos el método `generaDir`. Este método genera el código necesario para que la dirección de la variable correspondiente se guarde en un registro, que se devuelve como resultado del método.

Finalmente, el método `arbol` es el encargado de escribir los árboles en formato `verArbol`. No se ha intentado sangrarlos, por lo que su legibilidad no es muy buena. La definición de `__str__` en la clase base hace que éste sea el método llamado al escribir los árboles.

3.6. Representación de los tipos

Para representar los tipos, hemos creado una clase base, `Tipo`, que no se utilizará directamente, sólo a través de sus clases derivadas. El código correspondiente es:

```
3   class Tipo:
4     def __init__(self):
5       pass
6
7     def __ne__(self, otro):
8       return not self.__eq__(otro)
9
10    def talla(self):
11      return self.tamanyo
```

El método `__ne__` es el que se llama cuando se hace una comparación "distinto de". Hemos hecho que sea simplemente la negación de la comparación de igualdad. De esta manera, las clases derivadas sólo tienen que definir esta última (mediante el método `__eq__`). Utilizaremos el atributo `nombre` para guardar el nombre del tipo de modo que la implementación de `__eq__` sea más sencilla.

Como clases derivadas se ha creado la clase `Elemental` que representa los tipos elementales del lenguaje, entero y cadena, además del tipo implícito lógico y el tipo auxiliar error. El código correspondiente es:

```
13  class Elemental(Tipo):
14    def __init__(self, nombre):
15      self.nombre= nombre
16      self.tamanyo= 1
17
18    def __eq__(self, otro):
19      return self.nombre== otro.nombre
20
21    def __str__(self):
22      return self.nombre
23
24    def elemental(self):
25      return True
```

Cada uno de los tipos elementales es simplemente una instancia de la clase `Elemental`.

El tipo `Array` tiene un atributo para guardar el tipo base y otro para el número de elementos.

Finalmente, hemos creado un tipo función "descafeinado" que es prácticamente idéntico al tipo elemental, excepto por el método `elemental` que devuelve cero.

Como en muchas ocasiones se necesita saber si dos tipos son iguales o alguno de ellos es un error, se ha incluido en el módulo `tipos` la función `igualOError` que hace exactamente esa comprobación.

3.7. Representación de las funciones

Para las funciones hemos empleado una clase aparte, que no es derivada de `AST`. Cada función guarda su identificador, una lista con sus parámetros y otra con sus variables locales así como el tipo devuelto y el código de la función.

Quizá lo más interesante de la función es cómo se organiza el registro de activación y la secuencia de llamada. Hemos optado por una secuencia razonablemente simple de montar, aunque no sea la más eficiente. Una llamada a la función consistirá en ir apilando los parámetros e incrementando el `$sp`. Así, la llamada a una función `f` con los valores 1 y 2 para los parámetros enteros `a` y `b` genera el código siguiente:

```
# Llamada a f en línea 10
addi $r0, $zero, 1 # Valor entero
sw $r0, 0($sp) # Parámetro a
addi $sp, $sp, 1
addi $r0, $zero, 2 # Valor entero
sw $r0, 0($sp) # Parámetro b
addi $sp, $sp, 1
jal et1 # Salto a la función
```

Como ves, se calcula el valor de cada parámetro, se guarda en la dirección apuntada por `$sp` y se incrementa `$sp`.

El registro de activación comenzará por tanto por los parámetros tras los que guardaremos la dirección de retorno, el valor del `$fp` al entrar en la función y las variables locales. El comienzo de la función `f` comentada antes es:

© Universitat Jaume I 2006-2007

```
# Función f:
et1:
sw $ra, 0($sp) # Guardamos la dirección de retorno
sw $fp, 1($sp) # Guardamos el FP
add $fp, $sp, $zero # Nuevo FP
addi $sp, $sp, 3 # Incrementamos el SP
```

Se incrementa el `$sp` en 3 para guardar el `$fp` y la dirección de retorno y tener espacio para una variable local.

Esta manera de preparar el registro de activación hace que los parámetros se organicen de modo que si tenemos n parámetros, el primero tenga un desplazamiento $-n$ respecto al FP, el segundo un desplazamiento $-(n-1)$ y así hasta el parámetro n que tiene un desplazamiento -1. En cuanto a las variables locales, la primera tiene un desplazamiento 2, la siguiente 3 y así sucesivamente.

En caso de que la llamada forme parte de una expresión, habrá que preservar los registros activos en ese momento. Para eso, los apilamos primero y los recuperamos tras la llamada:

```
# Llamada a f en línea 11
# Guardamos los registros activos:
save $r0, 0($sp)
addi $sp, $sp, 1
addi $r1, $zero, 1 # Valor entero
sw $r1, 0($sp) # Parámetro a
addi $sp, $sp, 1
addi $r1, $zero, 2 # Valor entero
sw $r1, 0($sp) # Parámetro b
addi $sp, $sp, 1
jal et1 # Salto a la función
add $r1, $a0, $zero
# Recuperamos los registros activos:
subi $sp, $sp, 1
rest $r0, 0($sp)
```

Para devolver el resultado de la función, empleamos el registro `$a0`, como puedes ver en la línea `add $r1, $a0, $zero`.

3.8. Representación de las cadenas

Representaremos las cadenas guardando en posiciones consecutivas de memoria los caracteres, terminando la cadena con un carácter cero. Las variables de cadena contendrán la dirección de memoria donde comienzan. Con esto nos bastará porque sólo tenemos dos posibilidades para las cadenas: copiarlas y escribirlas. Para la primera, haremos una copia de la dirección y, para la segunda, utilizaremos un `syscall`, con el valor 2 en el registro `$sc`.

Utilizaremos las directivas de la máquina ROSSI para inicializar la memoria correspondiente a las cadenas.

3.9. Módulo gencodigo

Este módulo contiene la función `gencodigo` que es la que se encarga de crear el fichero con el código compilado. Para ello recibe el árbol correspondiente al programa principal y las listas con las funciones, las cadenas y las variables.

En primer lugar asigna direcciones (mediante el módulo `memoria`) a las variables globales y las cadenas, después genera código e intenta escribirlo en el fichero `a.rossi`.

El programa generado tiene las siguientes partes:

- Región de datos con las cadenas.

- Inicialización de registros.
- Código del programa principal.
- Código de parada (`syscall` con 6 en `$sc`).
- Código de las funciones.

3.10. Tabla de símbolos

Como el lenguaje permite únicamente dos ámbitos, hemos optado por implementar la tabla de símbolos mediante dos diccionarios, uno para las variables globales y otro para las locales (incluidos los parámetros). El acceso a estos diccionarios se hace a través de las siguientes funciones:

- `existe` devuelve cierto si un identificador está definido.
- `define` guarda la información asociada a un identificador; si el identificador ya estaba definido, se emite un mensaje de error semántico y no se cambia la información original.
- `recupera` devuelve la información asociada a un identificador, si existe, o `None` en caso contrario.

Además, relacionado con los cambios de ámbito, tenemos las funciones siguientes:

- `entraFuncion` se utiliza para indicar que hemos entrado en una función.
- `salFuncion` se utiliza para indicar que hemos salido de la función.
- `enFuncion` devuelve la función actual o, si no se está en ninguna, `None`.

3.11. Módulos auxiliares

Algunos de los módulos son simplemente auxiliares para facilitar la escritura del compilador.

3.11.1. Módulo Rossi

Para emitir de forma cómoda las instrucciones, utilizamos este módulo. En él tenemos una clase por cada posible modo de direccionamiento de operandos. Además tenemos una clase (`ROSSI`) que guardará las instrucciones en sí. Los objetos de la clase guardan el código de la operación, los operandos, la posible etiqueta de la línea y un comentario opcional. El método `__str__` se encarga de formatearlo según la sintaxis de ROSSI. Esta clase no está pensada para ser utilizada directamente, sino a través de las funciones auxiliares que se definen: una por cada instrucción.

3.11.2. Módulo memoria

Este módulo lleva la cuenta de las posiciones de memoria libres y se utiliza para asignar direcciones a los registros, las variables globales y las cadenas.

La función `asignaDir` asigna una dirección a cada uno de los objetos de la lista que se le pasa, actualizando cuál es la primera dirección libre.

3.11.3. Módulo registros

Este módulo permite reservar y liberar registros. Para reservar un registro, se utiliza `reserva` que proporciona un número de registro no utilizado en este momento. Con `libera` se indica que el registro deja de ser utilizado. Mediante `activos` se recupera la lista de registros activos.

También hemos incluido en este módulo la función `inicializaRegistros`, que emite las instrucciones necesarias para la inicialización de los registros `$sp` y `$fp`.

© Universitat Jaume I 2006-2007

3.11.4. Módulo errores

Define las funciones `lexico`, `sintactico` y `semantico` que son llamadas con un mensaje de error y una línea. Estas funciones guardan en una lista el error. Cuando posteriormente se llama a `escribeErrores`, se muestran todos los errores de la lista ordenados por número de línea.

3.11.5. Módulo errsintactico

Este módulo unifica el tratamiento de los errores sintácticos. El principal punto de entrada es la función `trataError`, que recibe como parámetros el analizador léxico, el no terminal en el que se produjo el error y el no terminal en que se trata el error. Con eso, escribe un mensaje de error estandarizado e intenta sincronizarse. Devuelve cierto si el no terminal debe reintentar y falso en caso contrario. De esta manera, las producciones de error se simplifican. Por ejemplo, el tratamiento de errores en el no terminal ⟨Sentencia⟩ es:

```
189   <Sentencia>-> error
190     @if errsintactico.trataError(mc_al, mc_nt, "<Sentencia>"):@
191     @  mc_reintentar()@
192     @else:@
193     @  Sentencia.arb= AST.NodoVacio(mc_al.linea())@
194     ;
```

Observa cómo en caso de no reintentar nos preocupamos de que el valor de `Sentencia.arb` tenga sentido. Cuando el no terminal no tenga atributos sintetizados, no hace falta tomar esta precaución. Por ejemplo, los errores que captura ⟨Globales⟩ se tratan así:

```
78   <Globales> -> error
79     @if errsintactico.trataError(mc_al, mc_nt, "<Globales>"):@
80     @  mc_reintentar()@
81     ;
```

Un caso especial es el de ⟨Programa⟩, que no tiene atributos sintetizados en sentido estricto pero que debe devolver un árbol al entorno:

```
62   <Programa> -> error
63     @if errsintactico.trataError(mc_al, mc_nt, "<Programa>"):@
64     @  mc_reintentar()@
65     @else:@
66     @  Programa.principal= AST.NodoVacio(mc_al.linea())@
67     ;
```

3.11.6. Módulo etiquetas

Este módulo define la función `nueva`, que devuelve una nueva etiqueta cada vez que es llamada. Las etiquetas están formadas por la cadena "`et`" seguida de un número que aumenta en cada llamada.

4. Ejercicios

Aquí proponemos una serie de modificaciones que te permiten comprobar si has entendido la organización de `minicomp`.

Ejercicio 1 Cambia `minicomp` para que el operador de asignación sea `=` y que el comparador de igualdad sea `==`.

Ejercicio 2 Haz que las palabras reservadas puedan escribirse íntegramente en mayúsculas o minúsculas, pero no en una combinación de cajas.

Ejercicio 3 Haz que se acepten comentarios de dos tipos. Los primeros nunca se extienden más allá de una línea: se abren con una secuencia de dos menores `<<` y finalizan con la primera secuencia de dos mayores `>>` que la siga dentro de la misma línea o, en su defecto, con el último carácter de la línea que no sea un eventual salto de línea. Los segundos, que se abren con la secuencia `<<<` de tres menores, sí pueden abarcar más de una línea y se cierran con la primera secuencia `>>>` que siga a la de apertura, si es que tal secuencia aparece en el programa; de no aparecer, se considerará que nos encontramos ante un comentario del primer tipo, abierto con `<<` y que tiene un menor adicional como primer carácter del interior del comentario.

Una advertencia: `metacomp` utiliza el módulo `re` para implementar el analizador léxico. Esto tiene como consecuencia que el operador de disyunción no busque el emparejamiento más largo, como puedes ver en la primera pregunta del apéndice B del manual de `metacomp`. En nuestro caso, probablemente lo más cómodo sea emplear dos expresiones regulares separadas, una para cada tipo de comentario.

Ejercicio 4 Adapta `minicomp` para que acepte un juego limitado de secuencias de escape. Las únicas "secuencias de escape" permitidas en los literales de cadena del lenguaje serán las siguientes:

Secuencia	Significado
`\n`	salto de línea
`\t`	tabulador
`\"`	comilla doble
`\\`	barra invertida

Además, si una barra invertida aparece en el interior de un literal de cadena, debe formar parte, obligatoriamente, de una secuencia de escape.

© Universitat Jaume I 2006-2007