Universitat de Girona
Escola Politècnica Superior

Projecte/Treball Final de Carrera

Estudi: Eng. Tèc. Informàtica de Sistemes

Títol:

Traductor de pseudocodi a java

Document: Resum

Alumne: Carles Royán Salvatella

Director/Tutor: Josep Suy
Departament: Informàtica i Matemàtica Aplicada
Àrea: Llenguatges i Sistemes Informàtics

Convocatòria: Juny 2003

Índex

1. Introducció

En les diferents assignatures de programació de les carreres informàtiques, els professors donen als alumnes diversos codis font de programes d'exemple o d'estructures de dades. Però els algorismes no poden ésser verificats, ja que no existeix cap eina per compilar i provar el programa escrit en pseudocodi. En projectes de fi de carrera anteriors s'havia intentat realitzar compiladors de pseudocodi, però no es podien aplicar a programes que solucionessin problemes d'una mida mitja.

L'objectiu del projecte és la realització d'una eina capaç de traduir el pseudocodi al llenguatge de programació **Java**, utilitzant la programació orientada a objectes en la realització del projecte, concretament el llenguatge **Java**. D'aquesta manera s'obté un programa en **Java** del qual es podrà comprovar el seu bon funcionament, tot compilant-lo amb qualsevol compilador estàndard de **Java** com el distribuït per *Sun Microsystems*. El projecte està basat em teories de llenguatges i creació d'autòmats reconeixedors de gramàtiques, ja que són els fonaments per tal de realitzar un compilador/traductor.

En el traductor s'implementa tant l'anàlisi lèxica, com la sintàctica i la semàntica. Les etapes de generació de codi intermedi, optimització i generació de codi final són substituïdes per la generació de codi **Java**. Per tant, el traductor implementa les etapes més importants del procés de compilació i es capaç de generar programes que ressolin problemes de mida mitjana i gran.

El traductor de pseudocodi no es limita solament a traduir, sinó que realitza totes les comprovacions sintàctiques i semàntiques que realitza un compilador. Si el fitxer pseudocodi no genera cap error al ser tractat per el traductor, tampoc generarà cap error al ser compilat el fitxer resultant amb el compilador **Java**. D'igual manera, si generés algun error el fitxer de pseudocodi, es pot estar segur que el fitxer de sortida tampoc es podria compilar.

S'espera que aquest projecte pugui servir de recolzament per a la docència, ja que permet implementar ràpidament algorismes sense haver de passar per la rigidesa dels llenguatges de programació tradicionals. A més, permet que els algorismes i les estructures de dades exposats a les classes de programació puguin ésser verificats, de manera que l'alumnat tingui accés a codis font sense error. I finalment, els alumnes poden fer-ne ús, per estudiar els algorismes exposats a classe, així com iniciar-se en el llenguatge **Java** tot observant com actua el traductor.

2. Estructura del traductor

El traductor es divideix en 4 grans etapes: anàlisi lèxica, anàlisi sintàctica, anàlisi semàntica i generació de codi **Java**. Cadascuna d'aquestes etapes s'encarrega de funcions diferents dintre el procés de traducció. Vegem-ho amb més detall:

- L'anàlisi lèxica llegeix el programa font en pseudocodi com una seqüència de caràcters d'esquerre a dreta i genera una llista de components lèxics anomenats *tokens*. Els espais en blanc i els comentaris són ignorats. Aquests *tokens* són un grup de caràcters amb significat col·lectiu i representen la informació del fitxer, per exemple un identificador, un número decimal, etc.
- L'anàlisi sintàctica agrupa els *tokens* del programa font tot construint un arbre de derivació o sintàctic que expressa com s'agrupen aquests *tokens*, és a dir, com s'estructura el codi font. Llavors, comprova si aquest arbre pot ésser generat amb les regles de la gramàtica, prèviament definides, que descriuen el llenguatge pseudocodi. En cas afirmatiu, l'entrada del programa és correcte, altrament es genera un error sintàctic.
- L'anàlisi semàntica utilitza l'arbre de derivació generat en la fase sintàctica i comprova diferents errors semàntics, com ara la concordança dels tipus, la visibilitat, l'herència, unicitat en les declaracions dels objectes, entre d'altres. La comprovació semàntica s'anomena també comprovació estàtica, per distingir-la de la comprovació dinàmica que es realitza en temps d'execució del programa objecte, com fan per exemple molts llenguatges *script*, com el **JavaScript**.
- Finalment, gràcies a la informació obtinguda en les etapes anteriors, es genera el fitxer de sortida en **Java**, el qual pot ésser compilat directament amb qualsevol compilador estàndard de **Java**.

Cal dir que durant l'execució de totes les etapes anteriors s'administra la taula de símbols i es tracten els errors corresponents que poden sorgir en qualsevol de les etapes. A la pàgina següent hi ha una figura que mostra la descomposició en fases del traductor implementat.

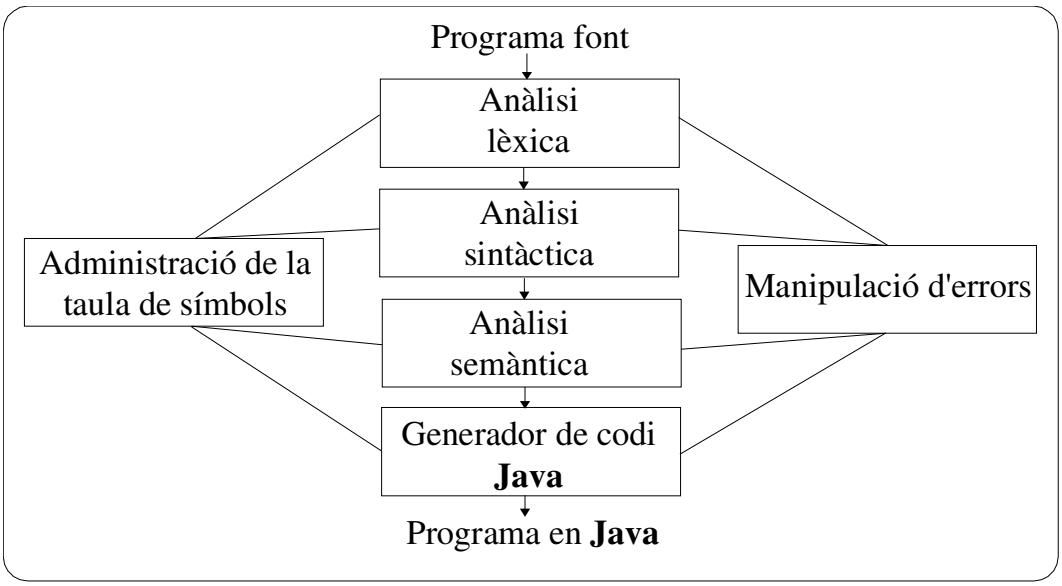

Figura 1 Divisió del procés de traducció en les 4 etapes

3. Implementació amb l'eina ANTLR

"The ANTLR Translator generator" és una versió completament redissenyada del PCCTS implementada en **Java** que genera codi en: **Java**, **C++** i **Sather**. Es pot dir que el PCCTS va ser la primera generació de la eina programada per Terence Parr, i el ANTLR és la segona generació. Està programat totalment orientat a objectes, i la manipulació d'errors es fa mitjançant d'excepcions. La definició completa de la gramàtica (anàlisi lèxica i sintàctica) està tota en un fitxer. Amb l'eina *antlr.Tool* es genera els fitxers amb el codi del *parser* que després han d'ésser compilats amb el llenguatge escollit a l'hora de generar el codi (recordem: **Java**, **C++** o **Sather**).

Aquesta eina ens permet treballar en un nivell superior, ja que ens hem de preocupar només de definir bé la gramàtica per tal que no surtin indeterminismes. Canvis en la gramàtica es reproduiran automàticament en el codi, tot evitant que s'introdueixin *bugs* en el codi si l'haguéssim de programar a mà. El ANTLR genera *parsers* de tipus LL(k), que significa que són *parsers descendents* que partint del símbol inicial de la gramàtica i aplicant una sèrie de derivacions, arriben al conjunt de *tokens* de l'entrada.

4. Funcionament del traductor

El traductor accepta com entrada un fitxer de text pla escrit en pseudocodi. Aquest fitxer és tractat com s'ha vist i es genera un fitxer de sortida **Java** amb el pseudocodi traduït a aquest llenguatge. Llavors, aquest fitxer **Java** pot ésser compilat amb qualsevol compilador estàndard de **Java** per tal de generar un fitxer binari executable. Els binaris són classes escrites en *bytecode* i tenen l'extensió ".class". Aquests fitxers poden ésser executats amb la màquina virtual de **Java**.

Podem entendre que el traductor implementat en el projecte actuaria com una mena d'interfície entre el pseudocodi i el **Java**. El diagrama de flux del funcionament del traductor seria el següent:

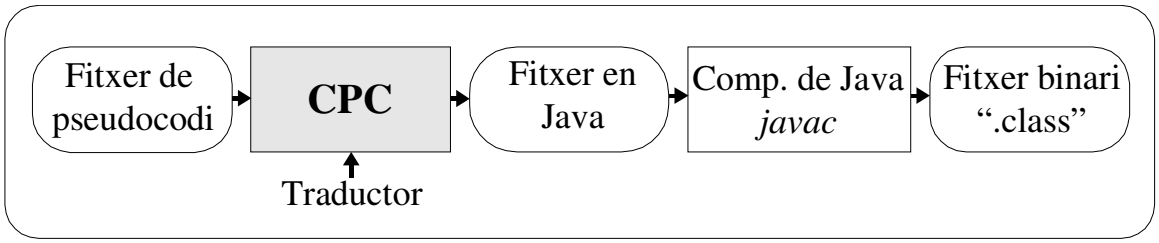

Figura 2 Diagrama de flux del funcionament del traductor

S'han realitzat proves amb problemes de mida mitjana i gran, obtenint resultats molt satisfactoris. Programes dissenyats amb el paradigma de l'orientació a objectes, completament dividit en classes, han funcionat correctament, cosa que demostra la versatilitat i la potència de l'eina creada durant el transcurs del projecte.

5. Conclusions

A continuació s'expliquen les conclusions extretes de l'elaboració d'aquest projecte.

El fet de disposar d'eines que realitzin la feina més mecànica de la generació de compiladors facilita el treball del programador en varis sentits. Per una part, no ha de perdre el temps implementant codi sistemàticament, sinó que es pot preocupar en les parts que requereixen més creativitat. D'altra banda, permet treballar a un nivell superior (a nivell de gramàtiques), i els canvis que realitzats a la gramàtica es veuran realitzats automàticament al codi del traductor. Sinó s'utilitzessin aquestes eines, un canvi en la gramàtica suposaria un canvi en el codi, i una major probabilitat d'introduir possibles *bugs* en el programa.

La programació orientada a objectes requereix d'uns compiladors molt més potents que la programació estructurada. Temes com la sobrecàrrega de mètodes o, sobretot, la herència ha complicat el desenvolupament del traductor, sobretot en l'etapa d'anàlisi semàntica on s'ha hagut de realitzar moltes comprovacions. El paradigma de programació orientada a objectes ofereix per al programador diverses facilitats a l'hora de dissenyar als programes, i aquestes facilitats són proporcionades en part per el llenguatge de programació i en part per el propi compilador, que ha de seguir estrictament les directrius del llenguatge.

L'anàlisi lèxica és la de més fàcil implementació, mentre que l'anàlisi sintàctica és una mica més complicada per què és la que defineix estrictament el llenguatge i s'ha d'anar en compte en no caure en indeterminismes quan es defineix la gramàtica. L'anàlisi semàntica és l'etapa més difícil, ja que s'ha de realitzar comprovacions de tipus, de visibilitat, herència, entre d'altres. La generació de codi **Java** només té el problema de que el text *parsejat* ha d'ésser passat entre fase i fase del traductor per tal de no perdre valors que hem d'escriure en aquesta fase, com expressions aritmètiques, identificadors, etc.

Finalment, com a valoració més personal, cal dir que he aprés amb exactitud quins són els diferents objectius de cada fase en què s'estructura un compilador/traductor. He aprofundit molt més en la construcció de cada etapa d'un compilador que no pas ho vaig fer en les pràctiques de l'assignatura optativa de *compiladors*.