Garion Musetta 20010613@studenti.uniupo.it Maggio 2017

RELAZIONE PROGETTO
COMPILATORE AC - DC

STRUTTURA

Il compilatore trasforma un linguaggio immaginario AC in uno comprensibile dal programma unix DC, una calcolatrice a precisione infinita.
Il linguaggio AC ha una grammatica che prevede dichiarazioni di variabili i/f (INT/FLOAT) con id composto da una sola lettera, assegnamenti a variabile di espressioni formate da costanti, variabili o operatori binari + e -, e stampe di variabili. Il DC invece è un linguaggio basato su stack e su registri e esegue operazioni scritte in notazione postfissa.
Il compilatore consiste in 4 componenti principali che interagiscono fra loro: **Scanner**, **Parser**, **TypeChecker** e **CodeGenerator**, oltre alle strutture dati, gli oggetti di appoggio e le funzioni di utilità: Token, TokenType, AST, SymbolTable, TypeCheckingUtil, e Eccezioni: LexicalException, SyntacticException, TypeException e VariableNotInitializedException.

I Packages sono:

- ast: contiene le classi dei nodi dell'albero sintattico astratto, AST, oltre ai tipi di operazioni del linguaggio (LangOper) e i tipi di variabili (LangType).
- parser: è il package che contiene uno dei componenti principali, cioè il Parser. É definita anche l'eccezione sintattica SyntacticException.
- scanner: contiene un altro componente principale, lo Scanner, e l'eccezione lessicale LexicalException.
- symTable: il package contenente la SymbolTable, una mappa fra stringhe-STEntry, dove vengono salvate le variabili con le loro informazioni sul tipo e valore.
- test: test di unità sui moduli principali.
- typecheck: contiene alcune funzioni di utilità usate dal Typechecker, per analizzare i tipi di due espressioni e renderle compatibili, e l'eccezione di tipo TypeException.
- visitor: contiene la definizione di visitor astratto e le sue due implementazioni, TypeChecker e CodeGenerator, gli ultimi due componenti principali del compilatore. É presente anche la classe VariableNotInitializedException.

Componenti principali:

Scanner è il modulo che si occupa di leggere carattere per carattere da un file di input e produrre uno stream di Token corrispondenti alle parole lette; può riconoscere tutti gli elementi più piccoli della grammatica, i Token, che rappresentano dichiarazioni, nomi di variabili, costanti, operazioni... Lo Scanner crea i Token e assegna loro un TokenType adatto a seconda di cosa legge. La sua interfaccia verso l'esterno permette di richiedere il prossimo Token consumando dell'input, oppure di restituire il Token appena letto.
Se viene letta una “parola” non definita dalla grammatica viene lanciata un'eccezione lessicale (LexicalException).

Parser preleva Token per Token dallo stream che produce lo Scanner e analizza la loro sequenza, per capire se sono conformi alle regole della grammatica: per ogni Token che legge, capisce se si tratta di una dichiarazione o di uno statement e quale, quindi sa quale prossimo Token si aspetterà. Se la sequenza non è in ordine secondo le regole, viene lanciata un'eccezione sintattica (SyntacticException).
Il Parser costruisce inoltre un AST, albero sintattico astratto, che rappresenta la struttura del programma: alla radice c'è un NodeProgram che contiene i riferimenti alle dichiarazioni e agli statement; le dichiarazioni sono composte da un NodeDcl che ha informazioni sul nome (NodeId) e tipo della variabile; gli statement invece possono essere istruzioni di stampa (NodePrint) oppure assegnamenti di costanti, variabili, o operazioni. I NodeAssign hanno informazione sull'id della variabile a cui assegnare (NodeId) e sulla parte destra dell'assegnamento, cioè una NodeExpr. Questa può essere un'operazione binaria (NodeBinOp) che può contenerne altre, oppure una costante (NodeCost) o una variabile (NodeDeref).
Leggendo lo stream di Token il parser crea tutti questi nodi e li dispone nell'ordine giusto, “collegandoli” fra di loro.

TypeChecker implementa il pattern visitor, lavora sull'albero prodotto dal Parser e lo decora con informazioni sul tipo delle variabili e espressioni: ogni nodo dell'albero che rappresenta una NodeExpr deve essere corredata da un LangType. Il TypeChecker effettua anche conversioni di tipo: se compare una variabile INT in un'espressione FLOAT, o a sinistra di un assegnamento a FLOAT, il suo NodeId corrispondente viene contenuto in un NodeConv, a rappresentare che è stata necessaria una conversione.
Se si tenta di scrivere espressioni con variabili aventi tipi non coerenti o compatibili fra loro, il TypeChecker lancia un'eccezione di tipo (TypeException).

CodeGenerator è l'ultimo modulo del traduttore, quello che si occupa dell'effettiva traduzione del codice. È un'altra implementazione del pattern visitor, infatti anche questo componente visita l'albero sintattico astratto precedentemente costruito e decorato, e per ogni tipo di nodo produce il codice DC corrispondente, trasformando le espressioni da notazione infissa a postfissa e traducendo la sintassi del linguaggio AC in quella di DC. Sono possibili assegnamenti e stampe di variabili, operazioni binarie + e -, - unario ('_' in DC).
A questo livello vengono controllati i valori delle variabili e se si tenta di usare una variabile non inizializzata a destra di un assegnamento, occorre un'eccezione "variabile non inizializzata" (VariableNotInitializedException).

COMMENTI

Questo progetto è stato sicuramente il più "grosso" mai prodotto. Anche se in realtà la complessità non è così elevata, ci si è trovati davanti ai problemi che derivano dalla scrittura di molti moduli complessi che interagiscono fra di loro e quindi si sono dovute adottare alcune accortezze, come per esempio: isolare il più possibile i moduli dall'esterno, esportando solo le operazioni indispensabili e mantenendo privato tutto il resto; testare esaustivamente ogni modulo prima di passare al successivo, per non dover "tornare indietro" e per evitare che dei malfunzionamenti in un componente influiscano sugli altri; effettuare periodicamente operazioni di riduzione della complessità e miglioramento della leggibilità, altrimenti col passare del tempo e con l'aumento delle righe di codice era difficile capire cosa e come era stato fatto in precedenza.
Ci si è quindi resi conto di quanto siano importanti le tecniche di ingegneria quali test di unità e refactoring, uso di software di versioning (è stato usato GIT fin dall'inizio del progetto), supporti dell'IDE e altri strumenti utili a gestire progetti non semplici.
In generale si è imparato anche di più sul linguaggio Java, implementando cose nuove, usando design pattern non visti in altri corsi e facendo molta attività di debug; la costruzione di un traduttore completo ha inoltre aperto la strada alla comprensione di come funzionano realmente i parser dei linguaggi di programmazione e quindi si riescono a capire molte più cose anche sui linguaggi stessi.

DIFFICOLTÀ INCONTRATE

Le principali sono state quelle legate alla gestione di progetti non semplici, quindi soprattutto difficoltà a modificare parti dei moduli vecchi, (se per qualche motivo veniva richiesta qualche modifica) perché non era facile ricordarsi di come erano state implementate alcune parti; oppure se un componente non è stato testato in modo esaustivo, capitava che poi implementandone di nuovi si riscontrassero errori incomprensibili causati da vecchi moduli non testati bene e quindi malfunzionanti: questo portava alla situazione sopra descritta, ovvero difficoltà nel capire cosa andava modificato e come, perché non ci si ricordava bene come erano state implementate alcune parti. Inoltre era necessario fermarsi e tornare a testare in modo più completo il modulo vecchio finché non si comportasse in modo stabilito, prima di ripartire (e magari non era neanche l'ultima volta che si andava a modificare il modulo che ora si pensava funzionante).
Un'ultima “difficoltà” è stata sicuramente affrontare le scelte fatte. Se in una situazione si possono scegliere diverse soluzioni per uno stesso (sotto-)problema, non è detto che tutte siano corrette e compatibili con i prossimi problemi a venire; alcune volte è capitato che, per mancanza di esperienza o per errori di valutazione, si scegliesse una certa soluzione (esempio, per trasformare le espressioni in notazione postfissa uso funzioni ricorsive sui nodi dell'AST, oppure trasformo subito tutto in stringa e poi visito sequenzialmente i caratteri della stringa?) ma poi la stessa non era più compatibile con altri requisiti o comunque si rivelava poco elegante o efficiente: a questo punto bisognava tornare indietro e cambiare implementazione.

POSSIBILI MIGLIORAMENTI

Una prima possibile ottimizzazione che salta all'occhio sarebbe “parametrizzare” Parser e Scanner, cioè permettere modifiche alla grammatica senza toccare il codice dei due moduli, in modo che questi riconoscano Token nuovi e nuove regole della grammatica definendo un file di configurazione (cioè un po' quello che viene permesso di fare in JCUP e JFLEX). Ovviamente non è un passo così facile da fare, ma una volta fatto, Parser e Scanner potrebbero teoricamente riconoscere moltissime grammatiche (LL (1)) e basterebbe ottimizzarli per avere un traduttore “universale”. Attualmente, se si volesse cambiare grammatica, bisognerebbe apportare modifiche pesanti a tutto il Parser, al TypeChecker, al CodeGenerator e, in misura minore, anche allo Scanner.

Altre modifiche più fattibili per noi potrebbero essere:

- “Compattare” tutto il traduttore in un'unica interfaccia, in modo che chi lo usa debba soltanto chiamare una funzione “traduci”, che prenda in input il file sorgente in ac, che si occupi di coordinare le varie chiamate fra i componenti e che restituisca soltanto il file di output in dc (come illustrato sullo schema UML).
- Salvare i valori delle variabili direttamente nella SymbolTable: questo eviterebbe la necessità di usare un attributo solo per sapere se la variabile è stata inizializzata. Attualmente è stata presa questa scelta perché il traduttore non tratta i valori delle variabili, ma questi vengono memorizzati nei registri del DC (quindi il traduttore non “vede” i valori, ma si salva soltanto se la variabile è stata inizializzata; per usare la variabile e quindi saperne il valore, viene caricata dal suo registro nello stack, con l'operazione di load del DC).
- Ottimizzare :

  (esempio, se ci sono due espressioni tipo: i = i + 1; i = i + 1;
  allora il codice DC tradotto fa due caricamenti della variabile sullo stack e due salvataggi della stessa nel registro; in realtà non sono necessarie tutte queste operazioni ma basterebbe compattare tutto in una sola istruzione: i = i + 2). In generale, analizzando il codice DC prodotto nei test si nota che spesso sono presenti molti salvataggi seguiti da caricamenti (.. 3 2 + sa la 2 + sb ..); magari prima di scrivere il file di output si potrebbe analizzare la stringa prodotta per eliminare queste situazioni (basterebbe eliminare “sa la” e l'espressione da lo stesso risultato, basta poi “ritardare” il salvataggio in a).
- Usare un file di log per informare l'utente su cosa sta succedendo: se la traduzione è andata a buon fine o se invece ci sono state eccezioni e qual è il carattere e la riga che ha causato l'errore. Attualmente viene riempito un solo file, **dcOut**, che contiene il codice DC tradotto, mentre le informazioni sulle eccezioni sono stampate sulla console di Java. Inoltre se capitano eccezioni in mezzo alla traduzione, il file dcOut rimane pieno a metà e non si capisce se la traduzione è andata bene o no. Quindi comunque non lasciare il file di output pieno di “spazzatura” se sono capitati errori, ma scrivere qualche messaggio all'interno del file stesso (oppure non crearlo proprio o cancellarlo, se la traduzione non va a buon fine).

- Un ultimo interessante miglioramento potrebbe essere implementare tutte le funzionalità che offre il programma DC (tutte le altre operazioni) e fornire supporto alla linea di comando, in modo che si possa chiamare un comando tipo:
  "$ acdcTranslate <fileInputAC> <fileOutputDC>"
  Per ora sono state implementate soltanto le operazioni '+' e '-' e la stampa variabili, ed è stato aggiunto il supporto al '–' unario su una costante e la stampa di variabili:
  (a = _1 ==> _1 sa);