Dispensa 1

1. Introduzione ai compilatori

Compilatore

Un compilatore è un programma che legge un programma scritto in un linguaggio (*sorgente*) e lo traduce in un programma equivalente in un altro linguaggio (*target*).

programma sorgente ⇒ compilatore ⇒ programma target

I compilatori sono quindi essenzialmente traduttori, inoltre essi fanno diagnostica (ossia riportano all'utente la presenza di eventuali errori nel programma sorgente) e sono supportati da routine di "error recovery" che permettono la gestione di particolari tipi di anomalie senza bloccare il processo di compilazione.

Il modello analisi/sintesi

Da un punto di vista logico l'architettura di un compilatore si può suddividere in due parti: Analisi e Sintesi.

programma sorgente ⇒ Analisi ⇒ rappresentazione intermedia ⇒ Sintesi ⇒ programma target

La parte di Analisi analizza la correttezza sintattica e semantica del programma sorgente, raggruppa il programma sorgente (inizialmente visto come una mera sequenza di caratteri) nei suoi elementi costituenti e crea una rappresentazione intermedia su cui è più facile lavorare.
La parte di Sintesi costruisce il desiderato programma target a partire dalla rappresentazione intermedia.

Interpreti

Mentre un compilatore produce un codice oggetto eseguibile, un interprete non produce target, ma esegue direttamente (sulla rappresentazione intermedia) le operazioni implicate dal programma sorgente.

programma sorgente ⇒ Analisi ⇒ rappresentazione intermedia --- > esecuzione

Solitamente l'uso di un interprete software produce un'esecuzione più lenta (in quanto ogni volta bisogna rifare la parte di analisi), tuttavia la convenienza e l'uso di un compilatore o di un interprete dipendono caso per caso dalle caratteristiche intrinseche del linguaggio.

Fasi della compilazione

Un compilatore opera in fasi ciascuna delle quali trasforma il programma sorgente da una rappresentazione a un'altra.

programma sorgente ⇒ Analisi Lessicale ⇒ Analisi Sintattica (Parsing) ⇒ Analisi Semantica ⇒ Generazione del Codice Intermedio ⇒ Ottimizzazione del Codice ⇒ Generazione del Codice ⇒programma target

Ogni fase può rilevare la presenza di errori nel programma sorgente, e interagisce con una struttura dati, chiamata *Symbol Table*, dove vengono memorizzati i token del programma sorgente.

2. Analisi Lessicale

Il ruolo dell'analizzatore lessicale

L'analizzatore lessicale fa lo *scanning* del programma sorgente (ossia legge un carattere alla volta da sinistra a destra il programma sorgente, inizialmente visto come una lunga stringa di caratteri indistinguibili e senza alcuna struttura) e lo raggruppa in *tokens*.
Un token è una sequenza di caratteri con significato collettivo. Esempi di token nei linguaggi di programmazione sono le parole chiave, gli identificatori, le costanti, etc.
Compiti minori dell'analizzatore lessicale sono anche l'eliminazione di spazi (blank, newline, tab), l'eliminazione di commenti, etc, in modo da produrre in output una stringa concatenata di token che poi il parser userà per l'analisi sintattica.

Quando individua un token, l'analizzatore lessicale aggiorna la Symbol Table e restituisce al parser la "categoria" del token (ad esempio ID per gli identificatori, NUM per le costanti numeriche, etc.) e il puntatore alla Symbol Table.

Definizione (Pattern):
Associato a ogni token del linguaggio sorgente c'è una regola, chiamata Pattern, che descrive l'insieme delle stringhe che rappresentano quel token

Definizione (Lessema):
Un Lessema è una sequenza di caratteri nel programma sorgente che "fa match" col pattern di un token

Ad esempio, nella stringa sorgente `TEMP:= X + 60` l'analizzatore lessicale, istruito da una opportuna regola/pattern per gli identificatori, individuerà il token ID sia per il lessema `TEMP` che per il lessema `X`. Ad esempio, una regola/pattern per gli identificatori Pascal potrebbe essere la definizione "stringa di cifre e/o lettere che inizia con una lettera". Tuttavia essendo il linguaggio naturale ambiguo, è preferibile utilizzare per la descrizione di pattern notazioni più formali, in particolare le espressioni regolari in quanto i token sono costrutti che cadono nella famiglia dei linguaggi regolari.

Il ruolo dell'analizzatore lessicale nei compilatori può essere così formalizzato:
"L'analizzatore lessicale fa lo scanning dell'input da sinistra verso destra cercando di trovare i più lunghi lessemi che fanno match con qualche pattern, trovato un match restituisce il token associato a quel pattern"

Il buffering dell'input

Il buffering dell'input è una tecnica usata in analisi lessicale per implementare lo scanning del programma sorgente. Si usa un buffer diviso in due parti capaci di contenere ciascuna N caratteri. Si caricano dal programma sorgente N caratteri input in una metà e, ogni qualvolta una metà è esaminata, viene caricata l'altra metà e continua lo scanning. Nel caso rimangano meno di N

caratteri nel sorgente, il corrispondente caricamento verrà delimitato dal carattere speciale EOF (end_of_file) nel buffer.
L'idea alla base dell'individuazione dei lessemi quando si usa il buffering dell'input è la seguente. La tecnica usa due puntatori nel buffer: il puntatore *forward* e il puntatore *lexeme_beginning*. All'inizio entrambi puntano allo stesso carattere (cioè il primo carattere dell'input quando comincia lo scanning, oppure il carattere successivo a un lessema durante lo scanning). Il puntatore lexeme_beginning rimane fermo e il puntatore forward va avanti finchè viene trovato un match con un pattern. La stringa di caratteri fra i due puntatori è il corrente lessema. A questo punto (una volta processato il lessema) entrambi i puntatori sono settati al carattere immediatamente a destra del lessema, e si procede iterativamente in questo modo fino a che l'input non viene completamente esaminato.
In questo procedimento quando il puntatore forward arriva alla fine della prima metà, la seconda metà è caricata con N nuovi caratteri input e il puntatore forward è semplicemente incrementato. Quando il puntatore forward arriva alla fine della seconda metà, la prima metà è caricata con N caratteri e il puntatore forward è settato all'inizio del buffer. Più in dettaglio, il frammento di codice che implementa l'avanzamento del puntatore forward (questo è il task più oneroso nello scanning col buffering dell'input) è il seguente:

```
IF forward è alla fine della prima metà THEN BEGIN
        carica la seconda metà
        forward := forward + 1
END
ELSE IF forward è alla fine della seconda metà THEN BEGIN
        carica la prima metà
        muovi forward all'inizio della prima metà
END
ELSE  forward := forward + 1
```

Questo codice è molto naturale ma risulta inefficiente in quanto, tranne nel caso in cui si è alla fine di una delle due metà, esso richiede sempre due test per far avanzare il puntatore forward. Il metodo può essere migliorato utilizzando due "sentinelle", ossia due caratteri EOF, posizionati nel buffer alla fine di ciascuna metà. Il codice per far avanzare il puntatore forward modificato con le "sentinelle" è:

```
forward := forward + 1
IF forward punta ad EOF THEN BEGIN
        IF forward è alla fine della prima metà THEN BEGIN
                carica la seconda metà
                forward := forward + 1
        END
        ELSE IF forward è alla fine della seconda metà THEN BEGIN
        carica la prima metà
        muovi forward all'inizio della prima metà
        END
        ELSE  termina lo scanning  \* infatti in questo caso EOF significa end_of_file
END
```

In media il nuovo codice con le "sentinelle" velocizza lo scanning in quanto effettua solo un test per verificare se forward punta ad EOF. Solo quando forward raggiunge la fine di una delle due metà (ma questo capita raramente, ogni N caratteri) oppure la fine del file si fanno più test.

L'algoritmo di Thompson

L'algoritmo di Thompson è un risultato proveniente dalla teoria dei linguaggi formali che permette di costruire riconoscitori a partire da espressioni regolari. Esso può risultare utile nello sviluppo dell'analizzatore lessicale del compilatore in quanto la fase di analisi lessicale ha come scopo proprio quello di riconoscere token, che sono costrutti regolari.

Input: un'espressione regolare *r* su un alfabeto Σ
Output: un'automa a stati finiti non-deterministico (NFA) che accetta L(*r*)
Metodo:
L'espressione regolare *r* deve essere anzitutto decomposta nelle sue sottoespressioni costituenti. L'algoritmo consiste di tre regole: le prime due permettono di costruire separati NFA per ciascuna sottoespressione, la terza combina questi automi ottenendo l'NFA risultante per *r*. Le tre regole sono le seguenti:

1) Per la stringa vuota ε, costruisci l'NFA con stato iniziale *i*, finale *f* e una ε-transizione da *i* a *f*

2) Per il simbolo *a* in Σ , costruisci l'NFA con stato iniziale *i*, finale *f* e una *a*-transizione da *i* a *f*

3) Se N(*s*) e N(*t*) sono gli NFA per le espressioni regolari *s* e *t*, allora:

3.1) per l'espressione regolare *s*|*t* costruisci l'NFA composto da N(*s*) e N(*t*) aggiungendo un nuovo stato iniziale *i*, un nuovo stato finale *f*, e quattro nuove ε-transizioni: due che vanno da *i* ai vecchi stati iniziali di N(*s*) e N(*t*), rispettivamente, e due che vanno dai vecchi stati finali di N(*s*) e N(*t*), rispettivamente, ad *f*

3.2) per l'espressione regolare *st* costruisci l'NFA composto da N(*s*) e N(*t*) fondendo in un unico stato il vecchio stato finale di N(*s*) e il vecchio stato iniziale di N(*t*), e considerando come stati iniziale e finale del nuovo NFA composto rispettivamente lo stato iniziale di N(*s*) e quello finale di N(*t*)

3.3) per l'espressione regolare *s** costruisci l'NFA composto da N(*s*) aggiungendo un nuovo stato iniziale *i*, un nuovo stato finale *f*, e quattro nuove ε-transizioni: una che va direttamente da *i* ad *f*, una che va da *i* al vecchio stato iniziale di N(*s*), una che va dal vecchio stato finale di N(*s*) ad *f*, ed una che va dal vecchio stato finale di N(*s*) al vecchio stato iniziale di N(*s*), quest'ultima per implementare il loop.

3.4) per l'espressione regolare parentesizzata *(s)* usa lo stesso NFA N(*s*), in quanto le parentesi non contribuiscono al linguaggio

Nota: per ogni NFA intermedio costruito dall'algoritmo di Thompson valgono le seguenti proprietà:
- ha esattamente uno stato iniziale e uno stato finale
- nessun arco entra nello stato iniziale
- nessun arco esce dallo stato finale

3. Analisi Sintattica (Parsing)

Nei compilatori il parser determina se il programma sorgente (ottenuto come sequenza di token dall'analizzatore lessicale) è sintatticamente corretto e produce in output il suo parse tree. Se si indica con P il programma sorgente e con G la grammatica context free che specifica il linguaggio sorgente del compilatore, ciò equivale a verificare che $P \in L(G)$.
Esistono tre tipi di parser per grammatiche context free: i parser *universali*, i parser *top-down* e i parser *bottom-up*. I parser universali sono generali, ossia riescono ad analizzare qualsiasi grammatica, ma non sono efficienti e quindi non si usano nei compilatori. Le tecniche utilizzate nella pratica sono quelle top-down e bottom-up che pur non essendo universali sono efficienti e comunque applicabili a un ampio sottinsieme delle grammatiche context free. In entrambi i metodi l'input del parser è analizzato da sinistra verso destra un simbolo alla volta; in particolare, i parser top-down costruiscono il parse tree dall'alto verso il basso, viceversa i parser bottom-up dal basso verso l'alto, ossia dalle foglie verso la radice.

3.1 Parsing Predittivo Non Ricorsivo

La tecnica di Parsing Predittivo Non Ricorsivo è un metodo di analisi sintattica top-down che analizza una stringa cercando di trovarne una derivazione leftmost, ossia una derivazione in cui ad ogni passo è espanso il non-terminale più a sinistra. Il termine "predittivo" è dovuto al fatto che questo tipo di parser non ha bisogno di backtracking per riconoscere una stringa, cioè può farlo deterministicamente. Il parser è inoltre "tabellare" nel senso che utilizza una parsing table, chiamata *predittiva*, dove è precompilata la grammatica del linguaggio sorgente.

Le componenti dell'architettura software di un parser Predittivo Non Ricorsivo sono le seguenti:

- *input* il buffer input contiene la stringa di token da analizzare seguita dal simbolo speciale end_marker $, con un puntatore al simbolo correntemente esaminato
- *stack* lo stack contiene simboli grammaticali (terminali e non-terminali) con il simbolo $ sul fondo
- *output* è il parse tree prodotto per la stringa input se essa è sintatticamente corretta
- *parsing table* è un array bidimensionale della forma M[*X*, *a*], dove *X* è un non-terminale e *a* è un terminale oppure il simbolo $. La generica entrata della parsing table è una produzione della grammatica sorgente avente il non-terminale *X* nella parte sinistra oppure è una entrata di errore che solitamente si lascia vuota
- *parsing program* è l'algoritmo di parsing che, guidato deterministicamente dalla parsing table predittiva, esegue l'analisi sintattica della stringa input

Algoritmo di Parsing Program Predittivo Non Ricorsivo

Input: una stringa di token *w*, una parsing table M per la grammatica sorgente
Output: il parse tree di *w* oppure syntax error
Metodo:
Al passo iniziale, il parsing program setta *w*$ nel buffer input con il puntatore sul token più a sinistra, e $S nello stack, con S al top, dove S è il non-terminale iniziale della grammatica sorgente.
Al passo generico, il parsing program considera la coppia (*X*, *a*), dove *X* è il simbolo al top dello stack e *a* il simbolo correntemente puntato dal puntatore input, e si comporta secondo queste possibilità:
1) Se *X* è un terminale ed è $X \neq a$, allora *syntax error*
2) Se $X = \$$, allora il parser si ferma con successo

3) Se X è un terminale ed è $X = a$, allora si fa il pop di X dallo stack e il puntatore viene incrementato di un token
4) Se X è un non-terminale, viene consultata l'entrata M[X, a] della parsing table:
4.1) Se M[X, a]= $X \rightarrow Y_1 \ldots Y_n$ allora si fa il pop di X dallo stack, il push di $Y_1 \ldots Y_n$ con Y_1 al top
4.2) Se M[X, a]=errore, allora *syntax error*

Nota: il parse tree output è ottenuto collezionando le produzioni ridotte al punto 4.1) dell'algoritmo.

Costruzione di parsing table predittive

Nella costruzione della parsing table per la tecnica di Parsing Predittivo Non Ricorsivo si utilizzano due funzioni chiamate *First* e *Follow*. Informalmente, la funzione First si applica ad una stringa di simboli grammaticali α e calcola l'insieme dei terminali che compaiono all'inizio di stringhe derivate da α, cioè:

- se $\alpha \overset{*}{\Rightarrow} a\beta$ allora a First(α)
- se $\alpha \overset{*}{\Rightarrow} \varepsilon$ allora ε First(α)

Invece, la funzione Follow si applica ai non-terminali di una grammatica e calcola per ciascuno di essi l'insieme dei terminali che appaiono alla sua destra in qualche forma sentenziale. La funzione Follow è formalmente definita dalle seguenti tre regole:

1) Inserisci \$ in Follow(S), dove S è il non-terminale iniziale della grammatica e \$ l'end_marker
2) Se nella grammatica c'è una produzione $A \rightarrow \alpha B \beta$ allora tutto ciò che è in First(β), eccetto ε se c'è, è inserito in Follow(B)
3) Se nella grammatica c'è una produzione $A \rightarrow \alpha B$ oppure una produzione $A \rightarrow \alpha B \beta$ con $\beta \overset{*}{\Rightarrow} \varepsilon$ allora tutto ciò che è in Follow(A) è inserito in Follow(B)

L'algoritmo per la costruzione di parsing table predittive è il seguente:

Input: una grammatica context free G
Output: la parsing table predittiva M
Metodo:
1) Per ogni produzione $A \rightarrow \alpha$ di G applica i passi 2) e 3)
2) Per ogni terminale a in First(α) inserisci $A \rightarrow \alpha$ in M[A, a]
3) Se ε è in First(α) inserisci $A \rightarrow \alpha$ in M[A, b] per ogni terminale b in Follow(A). Se ε è in First(α) e \$ è in Follow(A) inserisci $A \rightarrow \alpha$ in M[A, \$]
4) Poni a errore ogni entrata non definita di M

Definizione (Grammatica LL(1)):
Una grammatica LL(1) è una grammatica context free la cui parsing table predittiva non ha conflitti, ossia entrate multiple.

In definitiva, la tecnica di Parsing Predittivo Non Ricorsivo si può applicare soltanto al sottinsieme LL(1) delle grammatiche context free, ossia alle grammatiche che non hanno conflitti/entrate multiple nella parsing table (cioè la tabella deve avere una sola produzione in ogni entrata) e su cui il parser può quindi lavorare deterministicamente.

3.2 Parsing LR

Le caratteristiche principali che contraddistinguono il parsing LR sono le seguenti:

- *bottom-up*: i parser LR fanno l'analisi sintattica della stringa input un simbolo alla volta da sinistra a destra costruendo il parse tree dal basso verso l'alto.
- *shift-reduce*: i parser LR riconoscono una stringa input attraverso una derivazione rightmost al contrario riducendo ad ogni passo l'*handle*.

Definizione (handle):
Un handle di una forma sentenziale destra γ è una produzione $A \rightarrow \beta$ e una posizione di γ dove la stringa β può essere trovata e rimpiazzata da A per produrre la forma sentenziale destra precedente in una derivazione rightmost di γ

- *predittivo*: i parser LR non hanno bisogno di backtracking per riconoscere una stringa

- *tabellare*: i parser LR sono tabellari, ossia utilizzano una parsing table, in cui è precompilata la grammatica sorgente, che guida deterministicamente il parser nelle sue mosse

- *efficienza*: i parser LR effettuano l'analisi sintattica di una stringa input in tempo lineare

- *generazione automatica*: esistono generatori automatici di parser LR (ad esempio il tool Yacc)

- *potere riconoscitivo*: i parser LR sono applicabili alla intera famiglia dei linguaggi context free deterministici (nella quale cade la maggior parte dei costrutti dei linguaggi di programmazione)

Le componenti dell'architettura software di un parser LR sono le seguenti:

- *input* il buffer input contiene la stringa di token da analizzare seguita dal simbolo speciale end_marker $, con un puntatore al simbolo correntemente esaminato
- *stack* lo stack contiene una stringa che alterna simboli di stato e simboli grammaticali, sempre con uno stato al top
- *output* è il parse tree prodotto per la stringa input se la stringa è sintatticamente corretta
- *parsing table* è un array bidimensionale fatto di due parti *Action* e *Goto* le cui entrate specificano le azioni che corrispondono ad ogni stato dell'automa. La generica entrara Action[*s*, *a*], dove *s* è uno stato e *a* è un terminale oppure il simbolo $, può contenere un'azione di shift a un nuovo stato, oppure di riduzione di una produzione, oppure di errore, oppure di accettazione. La generica entrata Goto[*s*, *A*], dove *s* è uno stato e *A* è un non-terminale, può contenere un'azione di shift a un nuovo stato oppure di errore
- *parsing program* è l'algoritmo di parsing che, guidato deterministicamente dalla parsing table, effettua l'analisi sintattica shift-reduce della stringa input

Definizione (Configurazione):
La configurazione di un parser LR è una coppia la cui prima componente è il contenuto dello stack e la seconda componente è l'input non ancora espanso

Si noti che la configurazione generica di un parser LR (s_0 X_1 s_1 X_2 … X_m s_m, a_i a_{i+1} … a_n\$) corrisponde alla forma sentenziale destra X_1X_2 … X_m a_i a_{i+1} … a_n nel processo di derivazione rightmost al contario che il parser sta effettuando. La configurazione iniziale (s_0, a_1 a_2 … a_n\$) rappresenta ovviamente la forma sentenziale destra di partenza a_1 a_2 … a_n, ossia l'input da analizzare.

L'azione (o *mossa*) da intraprendere da parte del parser è decisa ad ogni passo considerando la coppia (s_m, a_i), dove s_m è il simbolo di stato al top dello stack e a_i è il token correntemente puntato nell'input, e quindi consultando l'entrata della parsing table Action[s_m, a_i].

Algoritmo di Parsing Program LR

Input: una stringa di token *w*, una parsing table per la grammatica sorgente
Output: il parse tree di *w* oppure syntax error
Metodo:
-Inizializzazione:
Al passo iniziale, il parsing program setta *w*$ nel buffer input con il puntatore sul token più a sinistra, e s_0 nello stack.
-Passo generico:
REPEAT FOREVER
BEGIN sia *s* lo stato al top dello stack e *a* il simbolo correntemente puntato nell'input
IF Action[*s*, *a*]=shift *s'* THEN
    BEGIN fai il push di *a* e *s'* sullo stack
        avanza il puntatore al prossimo simbolo input
    END
ELSE IF Action[*s*, *a*]=reduce$A \rightarrow \beta$ THEN * significa che si è formato un handle al top dello stack
    BEGIN fai il pop dallo stack di un numero di simboli doppio della taglia di β
        sia *s'* il nuovo stato al top dello stack
        fai il push di *A* e dello stato ottenuto da Goto[*s'*, *A*] sullo stack
    END
ELSE IF Action[*s*, *a*]=accetta THEN *successo*
ELSE syntax error
END

Nota: il parse tree output è ottenuto collezionando le produzioni $A \rightarrow \beta$ ridotte al passo di reduce dell'algoritmo.