10. Analisi dipendente dal contesto

Per generare il codice, il compilatore deve rispondere a molte domande.

È "x" uno scalare, un array o una funzione? È "x" dichiarato? Ci sono nomi che non sono dichiarati? Oppure dichiarati ma non utilizzati? A quale dichiarazione di "x" fa riferimento un certo uso? L'espressione $x \cdot y + z$ è coerente dal punto di vista del tipo? In $a[i, j, k]$, "a" ha tre dimensioni? Dove può essere memorizzato "z"? (nel registro, a livello locale, globale, heap, statico). In $f \leftarrow 15$, come dovrebbe essere rappresentato il valore 15? Quanti argomenti accetta la funzione $fie()$? E per $printf()$? Il puntatore $*p$ fa riferimento al risultato di una chiamata a $malloc()$? I puntatori p e q si riferiscono alla stessa area di memoria? Il valore di "x" è definito prima di essere utilizzato?

Queste domande fanno parte dell'analisi contestuale. Le risposte dipendono dai valori, non dalle parti del discorso. Domande e risposte implicano informazioni non locali e talvolta richiedono calcoli per essere determinate. Come possiamo rispondere a queste domande? Possiamo usare metodi formali: grammatiche sensibili al contesto o grammatiche con attributi (attributed grammars). Possiamo anche fare uso di tecniche ad-hoc, come tabelle dei simboli o codice specifico (routine d'azione).

Nel parsing, i formalismi hanno prevalso, mentre nell'analisi contestuale dominano le tecniche ad-hoc.

Studieremo il formalismo — una grammatica con attributi. Questo ci permetterà di chiarire molte questioni in modo immediato e conciso, separando i problemi di analisi dalle loro implementazioni. Vedremo come i limiti delle grammatiche con attributi motivino la pratica ad-hoc attuale, dovendo affrontare calcoli non locali e la necessità di informazioni centralizzate. Alcuni sostengono ancora l'uso delle grammatiche con attributi, con l'idea che la conoscenza sia potere e l'informazione sia una forma di immunizzazione. Tratteremo le grammatiche con attributi per poi passare alle idee ad-hoc.

Grammatiche con Attributi

Che cos'è una grammatica con attributi? È una grammatica libera dal contesto, arricchita da un insieme di regole. Ogni simbolo nella derivazione (o nell'albero di parsing) ha un insieme di valori denominati, o attributi. Le regole specificano come calcolare un valore per ciascun attributo. Le regole di attribuzione sono funzionali e definiscono in modo univoco il valore.

Esempio. Questa grammatica descrive numeri binari con segno e vorremmo arricchirla con regole che calcolino il valore decimale di ogni stringa di input valida.

1	*Number*	→	*Sign List*
2	*Sign*	→	+
3		|	-
4	*List*	→	*List Bit*
5		|	*Bit*
6	*Bit*	→	0
7		|	1

Gli attributi sono associati ai nodi dell'albero di parsing, e le regole sono assegnazioni di valori associate alle produzioni della grammatica. Ogni attributo viene definito una sola volta, utilizzando informazioni locali. È necessario etichettare termini identici nella produzione per garantire l'unicità.

Le regole e l'albero di parsing definiscono un grafo di dipendenza degli attributi. Questo grafo deve essere non circolare, il che garantisce una specifica funzionale di alto livello.

Gli attributi possono essere di due tipi:

- **Attributo sintetizzato**: dipende dai valori dei figli del nodo.
- **Attributo ereditato**: dipende dai valori dei fratelli e del nodo padre.

Le grammatiche con attributi permettono di specificare azioni sensibili al contesto, utilizzando valori derivati dalla sintassi, eseguendo calcoli e inserendo test o logiche

Attributi Sintetizzati

Gli attributi sintetizzati utilizzano valori derivati dai figli e da costanti. Sono associati alle grammatiche S-attributed e possono essere valutati con una singola passata dal basso verso l'alto, rendendoli particolarmente adatti al parsing LR.

Attributi Ereditati

Gli attributi ereditati utilizzano valori derivati dal nodo padre, da costanti e dai fratelli. Esprimono direttamente il contesto e possono essere riscritti per evitarne l'uso, se necessario. Sono spesso considerati più "naturali", ma non è semplice gestirli durante il parsing.

Uso Combinato di Attributi

L'uso combinato degli attributi sintetizzati ed ereditati è desiderato, in quanto consente di prendere valori dalla sintassi e utilizzarli per effettuare calcoli o per inserire condizioni e logiche nel processo di parsing.

Metodi di Valutazione

Metodi Dinamici Basati sulle Dipendenze
Questi metodi prevedono la costruzione dell'albero di parsing e del grafo di dipendenze. Successivamente, viene effettuato un ordinamento topologico del grafo di dipendenze, e gli attributi vengono definiti seguendo l'ordine topologico.

Metodi Basati su Regole (Treewalk)
Questi metodi analizzano le regole durante la generazione del compilatore, determinando un ordine fisso (statico) per la valutazione. I nodi vengono quindi valutati seguendo questo ordine predefinito.

Metodi "Oblivious" (Passate, Dataflow)
Questi metodi ignorano le regole e l'albero di parsing. Durante la progettazione, viene scelta un'ordinazione comoda che verrà poi utilizzata per la valutazione.

Circolarità

Possiamo valutare solo istanze acicliche. Il problema generale del test di circolarità è intrinsecamente esponenziale! Tuttavia, possiamo dimostrare che alcune grammatiche generano solo istanze con grafi di dipendenza aciclici. La più ampia classe di queste grammatiche è quella delle grammatiche "fortemente non circolari" (SNC). Le grammatiche SNC possono essere testate in tempo polinomiale. Se un test SNC fallisce, ciò non implica necessariamente che la grammatica sia circolare. Molti metodi di valutazione rilevano la circolarità in modo dinamico, il che è una proprietà sfavorevole per un compilatore.

Esempio. Grammatica con attributi circolare

Productions			Attribution Rules
$Number$	$\rightarrow$	$List$	$List.a \leftarrow 0$
$List_0$	$\rightarrow$	$List_1\ Bit$	$List_1.a \leftarrow List_0.a + 1$ $List_0.b \leftarrow List_1.b$ $List_1.c \leftarrow List_1.b + Bit.val$
	|	Bit	$List_0.b \leftarrow List_0.a + List_0.c + Bit.val$
Bit	$\rightarrow$	0	$Bit.val \leftarrow 0$
	|	1	$Bit.val \leftarrow 1$

La tabella descrive una grammatica con attributi assegnati alle produzioni. Gli attributi vengono calcolati in base alle regole di attribuzione, ma in questo caso le regole portano a una dipendenza circolare, impedendo una valutazione corretta degli attributi.

- **Produzione** `Number → List`:
  - `List.a ← 0`: L'attributo `a` del nodo `List` viene inizializzato a 0.
- **Produzione** `List_0 → List_1 Bit`:
  - `List_1.a ← List_0.a + 1`: L'attributo `a` di `List_1` dipende da `List_0.a`, incrementato di 1.
  - `List_0.b ← List_1.b`: L'attributo `b` di `List_0` viene assegnato in base a `List_1.b`.
  - `List_1.c ← List_1.b + Bit.val`: L'attributo `c` di `List_1` è determinato dalla somma di `List_1.b` e `Bit.val`.
- **Produzione** `List_0 → Bit`:
  - `List_0.b ← List_0.a + List_0.c + Bit.val`: L'attributo `b` di `List_0` viene calcolato utilizzando `List_0.a`, `List_0.c` e `Bit.val`. Questa dipendenza porta alla circolarità.
- **Produzione** `Bit → 0`:
  - `Bit.val ← 0`: L'attributo `val` per il simbolo `Bit` è impostato a 0.
- **Produzione** `Bit → 1`:
  - `Bit.val ← 1`: L'attributo `val` per il simbolo `Bit` è impostato a 1.

Le regole di attribuzione nella grammatica creano una dipendenza circolare tra gli attributi di `List_0` e `List_1`, rendendo impossibile una valutazione univoca degli attributi. Questo tipo di circolarità è una caratteristica indesiderata nelle grammatiche con attributi, poiché impedisce al compilatore di determinare i valori in modo coerente e prevedibile.

Le grammatiche circolari presentano valori indeterminati e gli algoritmi di valutazione falliranno su di esse. Le grammatiche non circolari, invece, si valutano in un insieme unico di valori. Sebbene una grammatica circolare possa dare origine a un'istanza non circolare, non è saggio affidarsi a questa possibilità per il compilatore.

Dovremmo (senza dubbio) utilizzare grammatiche dimostrabilmente non circolari. Ricordiamoci che stiamo studiando le grammatiche con attributi (AG) per ottenere una maggiore comprensione; quindi, è importante evitare calcoli circolari e indeterminati. Se ci atteniamo a schemi dimostrabilmente non circolari, la valutazione risulterà più semplice.

Considerazioni

I calcoli non locali richiedono numerose regole di supporto, mentre i calcoli locali complessi risultano relativamente semplici da gestire.

I Problemi

Le regole di copia aumentano il carico cognitivo e le esigenze di spazio, in quanto richiedono copie degli attributi. È possibile utilizzare puntatori per ridurre la duplicazione, ma questo aumenta ulteriormente il carico cognitivo. Una buona regola pratica è che il compilatore utilizzi tutto lo spazio che alloca, solitamente più volte. Il risultato è un albero con attributi, ma la gestione presenta alcune sottigliezze.

È necessario costruire l'albero di parsing e, successivamente, cercare le risposte all'interno dell'albero oppure copiarle nella radice

Affrontare il Problema

Se affidassimo il problema di stimare il numero di cicli a uno studente competente del terzo o quarto anno di informatica, potremmo introdurre un archivio centrale per gestire le informazioni. Si potrebbe creare una tabella dei nomi, con un campo per lo stato di caricamento/non caricamento. Questo approccio permette di evitare tutte le complicazioni legate alle regole di copia, all'allocazione e alla memorizzazione degli attributi. Il flusso degli attributi tra le assegnazioni avviene interamente tramite la tabella, garantendo un'implementazione pulita ed efficiente.

Esistono buone tecniche per implementare la tabella, come l'hashing. Al termine del processo, tutte le informazioni si troveranno nella tabella! Questo risolve la maggior parte dei problemi.

Purtroppo, però, questo design viola il paradigma funzionale delle grammatiche con attributi (AG). La domanda è: ci importa davvero?

Alternativa realistica

La traduzione ad-hoc diretta dalla sintassi si basa su un parser bottom-up, di tipo shift-reduce. A ogni produzione viene associato un frammento di codice, e ad ogni riduzione viene eseguito il frammento corrispondente. L'inserimento di codice arbitrario fornisce una flessibilità completa, ma include anche la possibilità di fare cose di cattivo gusto o implementazioni poco eleganti.

Per far funzionare questo approccio, è necessario avere nomi per gli attributi di ciascun simbolo sul lato sinistro e destro della produzione. Di solito, un attributo viene passato attraverso il parser insieme al codice arbitrario (strutture, variabili globali, statiche, …). `Yacc` ha introdotto `$`, `$$`, `$1`, `$2`, … `$n`, da sinistra a destra, per gestire questi attributi.

È necessario definire uno schema di valutazione, che si integra perfettamente nell'algoritmo di parsing LR(1). La maggior parte dei parser si basa su uno stile ad-hoc di analisi contestuale.

Vantaggi
Questo approccio affronta le carenze del paradigma delle grammatiche con attributi (AG), risultando efficiente e flessibile.

Svantaggi
Il programmatore deve scrivere il codice con poca assistenza e gestire direttamente i dettagli implementativi. La maggior parte dei generatori di parser supporta una notazione simile a `yacc`.

Utilizzi tipici

Costruzione di una tabella dei simboli
Durante il parsing, si inseriscono informazioni sulle dichiarazioni man mano che vengono elaborate. Alla fine della sintassi delle dichiarazioni, si effettua un'elaborazione postuma. La tabella viene utilizzata per verificare la presenza di errori durante l'avanzamento del parsing, assumendo che la tabella sia globale.

Verifica degli errori / del tipo

- **Definizione prima dell'uso**: verifica tramite ricerca in riferimento.
- **Dimensione, tipo, ecc**: verifica quando vengono incontrati.
- **Conformità del tipo dell'espressione**: passeggiata dal basso verso l'alto.

Interfacce delle procedure
Le interfacce delle procedure sono più complesse da gestire. È necessario costruire una rappresentazione per la lista dei parametri e dei tipi, e creare una lista di punti da verificare. La verifica può essere fatta offline o si possono gestire i casi di ordinamento arbitrario.

È davvero "ad-hoc"?

Relazione tra pratica e grammatiche di attributi

Ci sono alcune somiglianze tra la pratica "ad-hoc" e le grammatiche di attributi. Entrambi associano regole e azioni alle produzioni e l'ordine di applicazione è determinato dagli strumenti, non dall'autore. Inoltre, utilizzano nomi astratti per i simboli, anche se in modo parziale.

Tuttavia, ci sono anche delle differenze significative. Le azioni "ad-hoc" vengono applicate come un'unità, mentre non è sempre vero per le regole delle grammatiche di attributi. Nelle azioni "ad-hoc" è ammesso qualsiasi approccio, mentre le regole delle grammatiche di attributi sono funzionali e di un livello più alto rispetto alle azioni "ad-hoc".

Limitazioni

L'ordine di valutazione è imposto e deve essere eseguito in un ordine specifico, ossia in postordine, da sinistra a destra e dal basso verso l'alto.

Implicazioni
Questo comporta che le dichiarazioni devono sempre precedere gli utilizzi, e le informazioni di contesto non possono essere passate verso il basso. Come si fa a sapere da quale regola si è stati chiamati? Un esempio comune è l'impossibilità di passare la posizione di bit dalla destra verso il basso.

Si potrebbero usare variabili globali?
In teoria, sì, ma richiederebbe un'inizializzazione adeguata e una rivalutazione della soluzione. Esiste la possibilità di riscrivere questo approccio in una forma che si adatti meglio alla soluzione "ad-hoc"?

Rendere efficaci le azioni semantiche ad-hoc
Come possiamo adattare questo approccio a un parser LR(1)? È necessario trovare un modo per memorizzare gli attributi. Possiamo inserirli nello stack insieme allo stato e al simbolo, spingendo quindi tre elementi ogni volta e prelevando $3 \times |\beta|$ simboli. Serve anche uno schema di denominazione per accedervi: l'uso di n può essere tradotto nella posizione nello stack come $(\text{top} - 3n)$.

È poi necessario stabilire una sequenza per l'applicazione delle regole: ad ogni azione di riduzione corrisponde l'esecuzione della regola associata. Per farlo, possiamo aggiungere una dichiarazione "case" gigante al parser, per valutare la regola corrispondente ad ogni riduzione.

Questo aggiunge una valutazione della regola ad ogni riduzione, ma di solito i frammenti di codice coinvolti sono relativamente semplici ed efficienti.

E se una regola deve essere applicata a metà produzione?
In questo caso, è possibile trasformare la grammatica. La produzione può essere divisa in due parti, esattamente nel punto in cui è necessario applicare la regola. A quel punto, la regola verrà applicata durante la riduzione della parte appropriata.

È anche possibile gestire riduzioni sulle azioni di "shift". In questo caso, si può aggiungere una produzione per creare una riduzione. Ad esempio, se avevamo: $fee \to fum$, possiamo trasformarlo in: $fee \to fie \to fum$, associando l'azione a questa nuova riduzione.

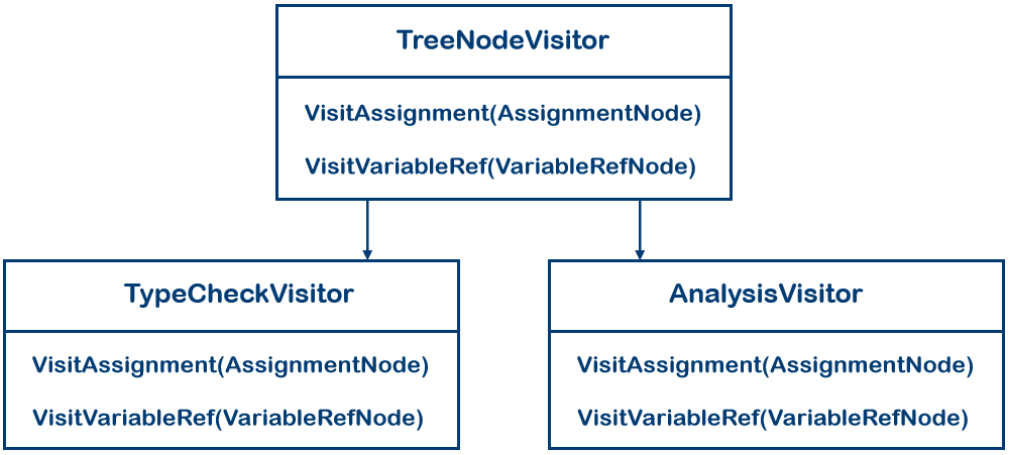
Insieme, queste tecniche ci permettono di applicare la regola in qualsiasi punto del processo di analisi.

Strategia alternativa

Cosa fare se è necessario eseguire azioni che non si adattano bene al framework di traduzione sintattica diretta "ad-hoc"? Un'opzione è costruire l'albero della sintassi astratta utilizzando la traduzione diretta, per poi eseguire le azioni durante uno o più attraversamenti dell'albero.

In un linguaggio orientato agli oggetti, questo problema può essere visto come un'applicazione classica del "visitor pattern". Questo approccio consente di eseguire qualsiasi tipo di calcolo nell'ordine dell'attraversamento dell'albero e di effettuare più passaggi se necessario.

Ancora una volta, una soluzione come questa potrebbe essere derivata da uno studente competente di informatica, di livello junior o senior, dopo pochi minuti di riflessione.



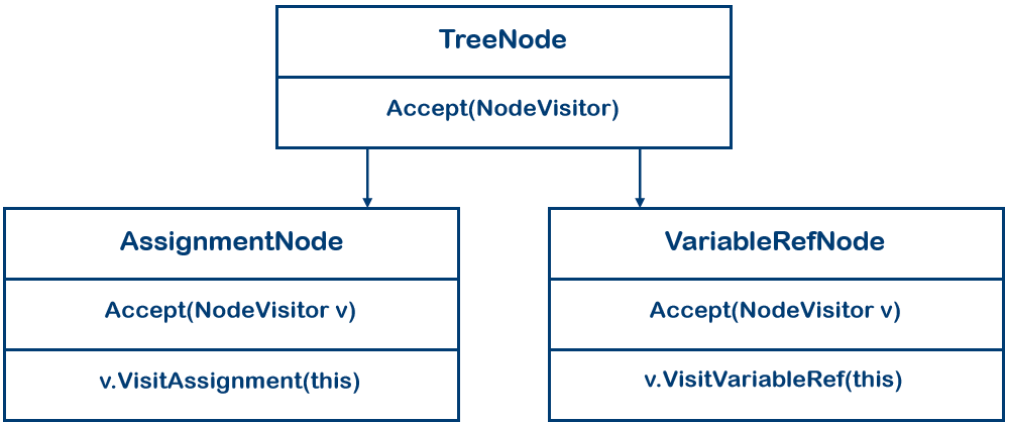
L'immagine mostra un diagramma che illustra l'uso del "visitor design pattern" per aggiungere funzionalità in una struttura ad albero (abstract syntax tree, AST). Questo approccio è spesso utilizzato nei parser e nei compilatori per separare la logica delle operazioni da quella della struttura dell'albero sintattico.

Il diagramma rappresenta una classe astratta chiamata `TreeNodeVisitor`, che definisce due metodi principali: `VisitAssignment` per visitare nodi di tipo `AssignmentNode` e `VisitVariableRef` per visitare nodi di tipo `VariableRefNode`. Da questa classe derivano due specifici visitatori:

1. **TypeCheckVisitor**: Implementa i metodi di visita per eseguire il controllo dei tipi sugli assegnamenti e i riferimenti a variabili.
2. **AnalysisVisitor**: Implementa gli stessi metodi di visita per eseguire analisi sui nodi dell'albero, come l'analisi semantica o delle dipendenze.

L'idea generale è che durante una passeggiata dell'albero sintattico, si possono utilizzare diversi visitatori per compiere compiti complessi (come verifiche o analisi) senza modificare direttamente la struttura dell'albero. Questo approccio consente di estendere facilmente le funzionalità del sistema.

Puoi usare questo design pattern per costruire un albero di sintassi astratta (AST) e fare lavori complessi durante la navigazione dell'albero utilizzando delle routine di cammino nell'albero



L'immagine rappresenta un esempio dell'applicazione del "visitor pattern" per l'attraversamento dell'albero. Nella struttura, ogni nodo dell'albero (ad esempio, `AssignmentNode` e `VariableRefNode`) eredita da una classe base (`TreeNode`) e implementa il metodo `Accept(NodeVisitor)` per gestire un oggetto visitatore (`NodeVisitor`).

Il codice di attraversamento parallelo segue la struttura dell'albero, separando il codice di attraversamento da quello specifico di gestione del nodo. Questo approccio facilita l'aggiornamento del processo senza modificare la struttura dell'albero stesso, consentendo una maggiore flessibilità e manutenibilità.

```
VisitAssignment(aNodePtr)
        // preprocess assignment
        (aNodePtr->rhs)->Accept(this);
        // postprocess rhs info;
        (aNodePtr->lhs)->Accept(this);
        // postprocess assignment;
```

Refers to current visitor!

To start the process:

```
AnalysisVisitor a; treeRoot->Accept(a);
```

L'immagine mostra un esempio di codice che implementa il "visitor pattern" per l'analisi di un'assegnazione nell'albero di sintassi. La funzione `VisitAssignment(aNodePtr)` gestisce il nodo di assegnazione passando attraverso tre fasi:

1. **Preprocessing dell'assegnazione**: viene eseguita una fase di preparazione sul nodo del lato destro (rhs) dell'assegnazione, chiamando il metodo `Accept(this)` per consentire al visitatore corrente di gestire il nodo.
2. **Postprocessing delle informazioni del lato destro**: successivamente, si passa al nodo del lato sinistro (lhs) dell'assegnazione, con un'altra chiamata al metodo

`Accept(this)`.

3. **Postprocessing dell'assegnazione**: infine, viene eseguita la fase di elaborazione finale dell'assegnazione.

Per avviare il processo di visita, viene istanziato un oggetto `AnalysisVisitor` e il metodo `Accept` viene invocato a partire dalla radice dell'albero (`treeRoot->Accept(a)`).

Questo approccio consente al visitatore di attraversare e manipolare i nodi dell'albero, mantenendo separate le operazioni di analisi dalla struttura dell'albero stesso.

Riassunto delle strategie per l'analisi dipendente dal contesto

---

Grammatiche di attributi
Vantaggi: Approccio formale e potente, in grado di gestire strategie di propagazione.
Svantaggi: Troppi passaggi di copia, assenza di tabelle globali, opera sull'albero di analisi sintattica.

Esecuzione del codice in postordine
Vantaggi: Semplice e funzionale, può essere specificato nella grammatica (ad esempio con Yacc) senza necessità di un albero di analisi sintattica.
Svantaggi: Ordine di valutazione rigido, assenza di ereditarietà del contesto.

Attraversamento generalizzato dell'albero
Vantaggi: Potente e generale, opera sull'albero di sintassi astratta utilizzando il "visitor pattern".
Svantaggi: Richiede codice specifico per ciascun tipo di nodo dell'albero ed è più complesso.