Appunti lezione 1

Processamento dei linguaggi: Preparare un programma per far sì che sia eseguibile su un computer. Esistono due modalità di esecuzione ovvero due modi per preparare un programma per far sì che sia eseguito su un computer: 1) Interpretato o 2) Compilato.

1)Interpretato: Un programma detto interprete esamina ogni istruzione del programma e le esegue. Tale interpretazione del programma termina se il programma finisce o in una certa riga del programma si trova un errore. I vantaggi sono la portabilità e la sicurezza mentre gli svantaggi sono che non viene controllato a priori che la sintassi sia corretta o che una variabile sia stata inizializzata.

2)Compilato: Un programma detto compilatore traduce il programma in un programma analogo scritto in un altro linguaggio (il linguaggio macchina). I vantaggi sono che è efficiente e che è presente controllo a basso livello mentre gli svantaggi sono che c'è una scarsa portabilità e scarsa sicurezza.

Linguaggio macchina: I compilatori generano codice per un particolare set di istruzioni macchina arricchito con routine di sistema operativo o di supporto (I/O, allocazione di memoria), che vanno linkate al codice oggetto.

Codice per una Macchina Virtuale: Permette di generare codice eseguibile indipendente dall'hardware. Se la macchina virtuale è semplice, il suo interprete può essere facile da scrivere.
Questo approccio penalizza la velocità di esecuzione di un fattore da 3:1 a 10:1.
Esiste la compilazione **"Just in Time" (JIT)** che può tradurre porzioni di codice virtuale in codice nativo. Il JJT rappresenta una modalità di compilazione di tipo dinamico che consente di migliorare le prestazioni di linguaggi di programmazione che usano il bytecode.

Java è un esempio di linguaggio che fa uso di una macchina virtuale, infatti, fa uso di un modello di compilazione misto in cui il sorgente viene prima compilato in un bytecode da un compilatore e quindi non viene compilato subito in linguaggio macchina. Dopodiché, il bytecode viene interpretato da un interprete che è la JVM (java virtual machine).
Questo porta due vantaggi:
Il primo è la portabilità ovvero non ci interessa la macchina o il sistema operativo che genera il bytecode basta che la macchina che deve interpretare il bytecode abbia la JVM.
Il secondo è la sicurezza ovvero dato che il sistema operativo non va a leggere il bytecode non c'è nessun rischio che esso venga danneggiato.
Lo svantaggio è dato che c'è un interprete non stiamo lavorando con un linguaggio efficiente che viene solo compilato.

TRADUZIONE DETTA DIRETTA DALLA SINTASSI

Per ogni costituente della frase sorgente il traduttore esegue le azioni appropriate per preparare la traduzione, azioni che consistono nel raccogliere informazioni sui nomi (nella tabella dei simboli), nel fare dei controlli semantici, ecc. Il lavoro del traduttore è guidato dalla struttura descritta dall'albero sintattico della frase. La traduzione diretta dalla sintassi è guidata dalle produzioni.

APPROCCI ALLA TRADUZIONE

Esistono due tipi di approcci:

1)approccio in due fasi:

Il parser sintetizza **l'Abstract Syntax Tree (AST**), che è un albero di parsing in forma astratta nella fase di analisi sintattica. Si utilizza l'AST per l'analisi semantica e poi per la sintesi dell'output.
In questo caso si ha a disposizione l'AST per fare analisi semantica ed elaborazione dell'output.

2)approccio semplificato una fase:

Si sintetizza l'output (ovvero genero il mio codice) direttamente durante l'analisi sintattica, senza costruire l'AST.
In questo caso non si memorizza l'albero, ma si è legati a come si fa il parsing (se top-down) o bottom-up. Non si ha a disposizione tutto l'albero per cui si possono fare SOLO elaborazioni semplici.

Il parse tree è l'albero corrispondente alla derivazione di una stringa della grammatica. Il parse tree contiene tutti i dettagli del parsing (cioè i nodi per tutti i simboli usati nella derivazione).
L'AST è definito per rendere possibile l'analisi semantica e la traduzione ed è una astrazione del parse tree, infatti, ha molti meno nodi.

STRUTTURA DI UN COMPILATORE

Ci sono due fasi fondamentali in un compilatore:

1)Analisi: Viene creata una rappresentazione intermedia del programma sorgente. Tale analisi si divide in tre analisi: analisi lessicale, analisi sintattica e analisi semantica.

2)Sintesi: Viene creato, partendo dalla rappresentazione intermedia, il programma target equivalente.

1)Generazione di codice intermedio
2)Ottimizzazione relative al programma in esse e non al programma macchina. Tali ottimizzazioni possono essere anche più di una.
3)Generazione di Codice

ANALISI LESSICALE

Lo scanner legge l'istruzione e genera un token e lo passa al parser. È eseguita dallo scanner, che scompone il programma in token. Lo scanner e quindi l'analizzatore lessicale vengono rappresentati da automi a stati finiti deterministici.

Token: Un Token descrive un insieme di caratteri che hanno lo stesso significato come ad esempio, identificatori, operatori, keywords, numeri, delimitatori. Le espressioni regolari sono usate per descrivere i token.

Esempio analisi lessicale

Input→ newVal = oldVal + 1 - 3.2
Output→ ⟨id : newVal⟩⟨assign⟩⟨id : oldVal⟩⟨plus⟩⟨intNum : 1⟩⟨minus⟩⟨floatNum : 3.2⟩

ANALISI SINTATTICA

In questa fase si riconosce se un token è derivabile e si trovano errori sintattici. Tale analisi è eseguita dal parser che organizza i token in un parse tree e inoltre, riconosce quali produzioni sono state usate per generare la sequenza di Token.

Il **parse tree** è l'albero di derivazione della grammatica. Nel nostro caso il parse tree generato è un **ast**, ossia un parse tree semplificato, con meno informazioni (con meno nodi). L'ast ricostruisce la grammatica lineare del codice.

Esistono due tecniche di parsing:

1)TOP-DOWN: In questo caso si costruisce l'ast a partire dalle grammatiche LL e si costruisce l'albero partendo dalla radice andando verso e foglie.

2)BOTTOM-UP: L'ast si costruisce a partire dalla grammatica LR e si costruisce l'albero partendo dalle foglie e proseguendo verso la radice.

ANALISI SEMANTICA

Tale analisi è eseguita dal **type checker** che visita l'ast e controlla:

1)Che le variabili siano state dichiarate prima di essere usate e siano dichiarate una volta sola,

2)Che siano stati usati i giusti tipi. All'occorrenza effettua una conversione,

3)La raggiungibilità del codice,

4)Corretto ritorno di valori dai metodi.

In questa fase, nel caso di approccio alla traduzione in due fasi il type checker aggiunge nell'ast delle informazioni di tipo.

TRANSLATOR, OPTMIZER e CODE GENERATOR

Il translator genera il codice intermedio (IR) che ha poco della macchina di destinazione, mentre l'ottimizzatore lo ottimizza con operazioni come l'eliminazione delle parti di codice non raggiungibili e delle operazioni inutili. Il code generator aggiunge al codice intermedio le informazioni riguardanti la macchina su cui verrà eseguito il codice, come gli indirizzi di memoria, i registri, eccetera.

STATIC SINGLE ASSIGNMENT (SSA): È un codice intermedio nel quale ogni variabile è assegnata esattamente una volta ed è definita prima del suo uso. Le variabili originali sono replicate in versioni in modo tale da tracciarne la catena di definizione ed uso. Questo formato semplifica e migliora i risultati che si possono ottenere nella fase di ottimizzazione, semplificando le proprietà delle variabili.

TABELLA DEI SIMBOLI

Una tabella dei simboli mantiene l'associazione fra gli identificatori e le informazioni ad essi associati (tipo, definizione, ecc.). È condivisa dalle varie fasi della compilazione ed ogni volta che un identificatore viene usato, la tabella di simboli consente di accedere alle informazioni raccolte quando è stato definito. In genere, si implementa tramite una tabella hash.

STRUMENTI PER LA REALIZZAZIONE DI COMPILATORI

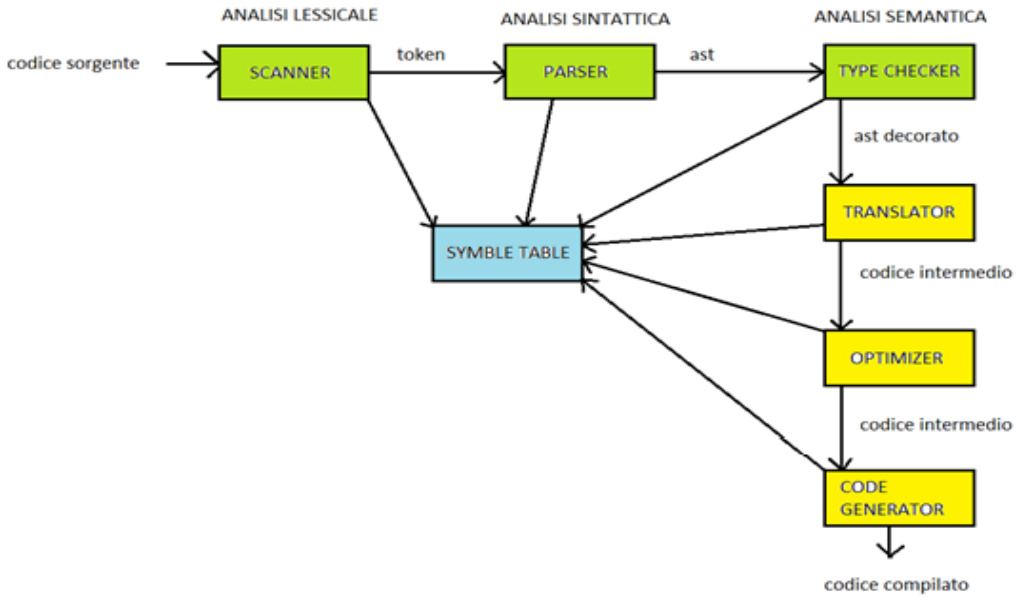
1)Generatori di scanner: producono analizzatori lessicali (input in generale sono espressioni regolari che descrivono i token).

2)Generatori di parser: producono AST (input la grammatica). Inoltre, si possono specificare "attributi" e regole per realizzare analizzatori semantici.

3)Traduttori diretti dalla sintassi: producono collezioni di routine che visitano l'AST e generano il codice intermedio.

4)Generatori di codice: prendono regole per tradurre da IL a linguaggio macchina (soluzioni alternative selezionate con "template matching").

NOTA→Un programma può avere diversi **scope**, ossia zone di codice delimitate da delimitatori in cui una variabile dichiarata può essere usata. In questo caso, la symbol table viene implementata come un albero formato da più tabelle. Nella radice ci sono le variabili appartenenti allo scope globale. Le altre tabelle contengono invece le variabili appartenenti ai vari scope locali.



Differenze tra analisi lessicale e analisi sintattica

Quali costrutti di un programma sono riconosciuti da un analizzatore lessicale, e quali da un analizzatore sintattico?

→L'analizzatore lessicale tratta solo i costrutti del linguaggio non propriamente ricorsivi.

→L'analizzatore sintattico consente di trattare i costrutti del linguaggio ricorsivi.

→L'analizzatore lessicale semplifica il lavoro dell'analizzatore sintattico, riconoscendo i componenti più piccoli, i tokens, del programma sorgente.

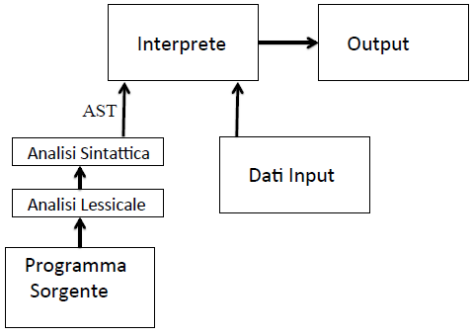
→L'analizzatore sintattico lavora sui tokens riconoscendo le strutture del linguaggio.

INTERPRETI

Ci sono due tipi diversi di interpreti che supportano l'esecuzione di programmi: interpreti di una macchina (astratta/concreta), e interpreti di un linguaggio.

Interpreti di Macchine: Simulano l'esecuzione di un programma compilato per una particolare architettura. Java usa un interprete bytecode per simulare l'effetto del programma compilato per la Java Virtual Machine (JVM).

Interpreti di Linguaggi: L'interprete del linguaggio simula l'effetto dell'esecuzione di un programma senza compilarlo in un particolare linguaggio macchina. Per l'esecuzione viene usato direttamente l'AST.



VANTAGGI

1) È possibile aggiungere codice a tempo di esecuzione.
2) È possibile generare codice a tempo di esecuzione.
3) In Python e Javascript, per esempio, qualsiasi variabile stringa può essere interpretata come una espressione ed eseguita.
4) Il tipo di una variabile può cambiare dinamicamente. I tipi vengono testati a tempo di esecuzione (non compilazione).
5) Gli interpreti supportano l'indipendenza dalla macchina. Tutte le operazioni sono eseguite nell'interprete.

SVANTAGGI

1) Non è possibile effettuare ottimizzazione del codice
2) Durante l'esecuzione il testo del programma è continuamente riesaminato, i tipi e le operazioni sono ricalcolate anche ad ogni uso.
3) Per linguaggi molto dinamici questo rappresenta un'overheads 100:1 nella velocità di esecuzione rispetto al codice compilato.
4) Per linguaggi più statici (come C o Java), il degrado della velocità è dell'ordine di 10:1.
5)Il programma sorgente spesso non è compatto come se fosse compilato. Ciò può causare una limitazione nella dimensione dei programmi eseguibili.

ANALISI LESSICALE

Il **lessico** è l'insieme delle parole e dei simboli appartenenti ad un linguaggio. Ogni elemento del lessico appartiene ad una delle seguenti sei **classi lessicali**:
1) PAROLE CHIAVE: non possono essere usate come identificatori (for, if, else ...)
2) DELIMITATORI: (; {) ...) **3) OPERATORI:** (*, +, - ...)
4) COMMENTI **5) IDENTIFICATORI** **6) COSTANTI**
Le prime quattro sono classi lessicali **chiuse**, e i loro elementi sono composti solo dal loro nome.
Le ultime due sono classi lessicali **aperte**, ed i loro elementi sono composti da nome e da **attributo semantico**.
Le classi lessicali sono rappresentate da espressioni regolari.

L'analizzatore lessicale non deve solo verificare che una sottostringa del testo sorgente corrisponde ad un elemento lessicale valido, ma deve anche tradurla in una opportuna codifica che faciliti la successiva elaborazione da parte del traduttore o interprete.
La codifica deve contenere:
1) L'identificativo della classe lessicale cui l'elemento appartiene.
2) Gli attributi semantici (nel caso ve ne siano associati a tale classe).

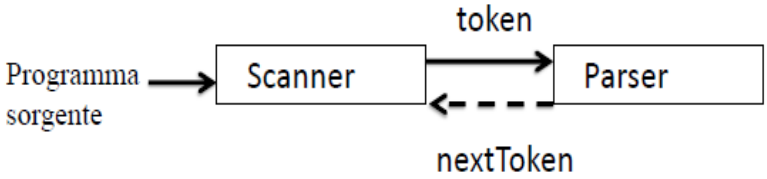
Ruolo dell'analizzatore lessicale
1) Il ruolo dell'analizzatore è quello di fornire un modo per isolare le regole di basso livello dalle strutture che costituiscono la sintassi del linguaggio.
2) Suddividere la frase in ingresso in elementi lessicali (detti tokens) da fornire al parser.
3) Eliminare gli spazi bianchi e i commenti.

Terminologia
Token: unità lessicale restituita dall'analizzatore lessicale e fornita come ingresso al parser (e.g., costante, identificatore, operatore, ...)
Lessico: stringa di caratteri che rappresentano un particolare token.
Pattern: una descrizione del lessico che corrisponde al token.

Come fa lo scanner a capire quando una stringa è un identificatore o una parola chiave?
La tecnica si chiama **lookup delle parole chiave**, e consiste nel tenere una tabella contenente tutte le parole chiavi. Quando lo scanner trova una stringa, cerca se è presente nella tabella o meno.
Quando il codice termina lo scanner produce un token **EOF**. Quando il parser lo riceve, controlla che la fine fisica del codice coincida con la sua fine logica.
Un errore lessicale è dato quando per una stringa non esiste un token corrispondente.

L'interazione tra scanner e parser può avvenire in due diversi modi:

1)Lo scanner processa tutto il programma e invia tutti i token al parser. Serve però molto spazio in memoria per contenere tutti i token
2)Lo scanner produce token solo quando richiesti dal parser.



Come realizzare un analizzatore lessicale

Per realizzare un programma ad-hoc che riconosce tutti gli elementi lessicali e produce i corrispondenti token si può partire:

1)Dalle espressioni regolari
2)Dall'automa a stati finiti
3)Dalla grammatica regolare

Conclusione dello scanning

Cosa accade quando viene raggiunta la fine del file di input?

In genere (ed è quello che faremo nel nostro compilatore) si crea uno pseudocarattere (il token EOF). (In corrispondenza al -1 ritornato da InputStream.read() viene generato il token EOF.)

Il token EOF è utile perché permette al parser di verificare che la fine logica di un programma corrisponde con la fine fisica. Molti parser richiedono l'esistenza di un tale token.

Recupero dagli errori lessicali

Una sequenza di caratteri per la quale non esiste un token valido è un errore lessicale. Gli errori lessicali non sono comuni ma devono comunque essere gestiti dallo scanner. Non è opportuno bloccare il processo di compilazione per errore lessicale. Le strategie di recupero sono:

1) Cancellare i caratteri letti fino al momento dell'errore e ricominciare le operazioni di scanning,
2) Eliminare il primo carattere letto dallo scanner e riprendere la scansione in corrispondenza del carattere successivo.

Di solito, un errore lessicale è causato dalla comparsa di qualche carattere illegale, soprattutto all'inizio di un token. In questo caso i due approcci sono equivalenti.

Appunti lezione 2

Derivazione left-most (o sinistra)

Data una grammatica G = (V, Σ, P, S), una stringa w $\in \Sigma^*$ è nel linguaggio generato da G (w $\in$ L(G)), se S $\rightarrow^*$ w.

Il processo di applicare le produzioni per derivare stringhe è semplice, ad esempio:

Data le seguenti regole di produzione:

0. S → E $ 1. E → Pr (E) 2. E → v Tl 3. Pr → f 4. Pr → ϵ 5. Tl → + E 6. Tl → ϵ

deriviamo la stringa f (v + (v)) $ con una derivazione left-most (o sinistra) (cioè rimpiazzando sempre il non terminale più a sinistra):

NOTA: tra le parentesi indico il numero della regola di produzione usata.

S (0)⟶E $ (1)⟶ Pr (E) $ (3)⟶ f (E) $ (2)⟶ f (v Tl) $ (5)⟶ f (v + E) $ (1) ⟶ f (v + Pr(E))$ (4)⟶ f(v + (E)) $(2)⟶f(v + (v Tl)) $(6)⟶ f (v + (v)) $

Processo di parsing

Il processo di applicare le produzioni per derivare stringhe è semplice, mentre rovesciare il processo, non lo è. Il parsing è il processo inverso alla derivazione, cioè, data una stringa ricostruire, se possibile, le produzioni che sono state applicate partendo dal simbolo iniziale per ottenere la stringa. Ad esempio data la stringa f(v + (v)) $ e la grammatica della indicata sopra il parsing si deve dire la sequenza delle produzioni che sono state applicate per ottenere tale stringa.

Il parsing deve anche dire quando una stringa NON è derivabile dall'assioma come nel caso di questa stringa f(v) + v $.

La tecnica di parsing top-down cerca di ricostruire la derivazione left-most (sinistra) e fallisce se la stringa NON fa parte del linguaggio.

Questa tecnica è nota con diversi nomi:
→**Parsing top-down,** perché il parser comincia dal simbolo iniziale della grammatica e fa crescere l'albero di parsing dalla radice alle foglie.
→**Predittivo**, perché il parser deve predire qual è la prossima derivazione che deve essere applicata. Da notare che il parser LR non è predittivo.
→**LL(k)**, perché l'input è scandito da sinistra a destra (questo è indicato con la prima L), produce la derivazione left-most (questo indicato con la seconda L) e usa k simboli di "lookahead" (noi ci limiteremo a LL(1)).

Implementazioni del Parsing top-down, Parsing a Discesa Ricorsiva (che implementeremo noi).
Il programma viene scritto a mano a partire dalla grammatica. Le funzioni/metodi che definiscono il parsing sono mutuamente ricorsive.
Parsing LL basato su tabelle (top della pila e simbolo in input).
Questo è simile all'implementazione del riconoscimento dei linguaggi regolari attraverso l'automa, ma in questo caso si ha un programma che simula un automa a pila. Il suo funzionamento è diverso rispetto all'automa shift-reduce per il parsing LR.

Entrambe le implementazioni partono dalla tabella PREDICT che ci deve dire per ogni produzione quali simboli dell'input ne predicono l'uso. Questa tabella si basa sul calcolo del FIRST e del FOLLOW. Fare questa tabella ci permetterà anche di dire se la grammatica è LL(1) o se non lo è.

FIRST E FOLLOW
Il first(B) è l'insieme dei simboli terminali che appaiono per primi derivando B usando le produzioni della grammatica.
Il follow(B) è l'insieme dei terminali che vengono per primi dopo B in ogni produzione. Ad essi bisogna unire l'insieme follow di ogni non terminali che invocano B. Ovvero nelle regole di produzione in cui B appare dopo la freccia.

IL PREDICT
Il predict di una produzione P ci dice quale è l'insieme di simboli terminali che sono prodotti all'inizio di una stringa generata dalla produzione P.
La produzione che non produce la epsillon avrà predict uguale al first mentre la produzione che produce epsillon avrà predict uguale al follow.

Parsing LL(1)
Supponiamo di avere un insieme di simboli di tokens (i ***Tokens*** sono i nostri simboli terminali).
La stringa di input di cui vogliamo fare il parsing è: ***w a w'***
dove ***w, w'∈ Tokens**** e ***a ∈ Tokens*** e che il parser ha costruito (fino ad ora) la derivazione left-most
$S \rightarrow^* w A X_1 \dots X_n$
Supponendo che $Q = \{p : A \rightarrow \alpha \in P \mid a \in Predict(p)\}$
Si possono verificare i seguenti casi:
1) Q è vuoto, per cui nessuna produzione per A può generare il token a. Questo è un errore sintattico, (le produzioni per A potrebbero aiutare a capire che tipo di errore!)
2) Q contiene più di una produzione. In questo caso il parsing è non deterministico. Questo produrrebbe un parsing inefficiente (che necessita di backtracking). La grammatica deve essere resa deterministica.
3) Q contiene esattamente una produzione. In questo caso si procede applicandola.

Definizione di Grammatica LL(1) e Linguaggio LL(1)

Una grammatica è LL(1), se per ogni simbolo non-terminale A, un token predice al più una produzione. Cioè una volta fatta la tabella Predict per ogni simbolo non terminale A:

Se le produzioni, p_1, ...,p_n associate ad A e $Pred_1$, ...,$Pred_n$ sono gli insiemi predict associati a p_1, ...,p_n, **allora $Pred_i \cap Pred_j = \emptyset$** per tutti $1 \leq i \neq j \leq n$.

Un linguaggio è LL(1), se ha una grammatica LL(1) che lo genera.

Parsing a discesa ricorsiva

L'input del parsing è la sequenza di token generata dallo scanner.

→A ogni non terminale, A, è associata una funzione.

→La funzione associata con A fa un passo di riduzione, scegliendo una delle produzioni associate ad A.

→Il parser sceglie la produzione da applicare ispezionando i prossimi k token dell'input. Per questo viene definito l'insieme di token Predict per ogni produzione $p \in P$.

→I token ispezionati sono il lookahead.

La funzione match

La funzione/metodo match(TokenType type) controlla che il prossimo token dello stream abbia uno specifico tipo, nel qual caso lo consuma e lo ritorna, mentre non ha lo stesso tipo match produce un ERRORE.

```
Token match(TokenType type) }
    if (type==peekToken().getType())
        then return nextToken()
        else ErroreSintattico
```

Implementazione Non Terminali

Per ogni non terminale A, a cui sono associate le produzioni p1, ..., pn, scriviamo una funzione/metodo, del tipo seguente:

```
parseA()
    Token nextTk = peekToken()
    if nextTk ∈ Predict(p1) then //codice per p1
    ...
    if nextTk ∈ Predict(pn) then //codice per pn
    else ErroreSintattico
```

supponiamo che $p : A \rightarrow X1 \cdot \cdot \cdot Xn$ il codice per p è la sequenza dei codici per Xi dove:

→se Xi è il non terminale B allora chiamiamo parseB(),

→se Xi è un token (cioè un terminale) allora chiamiamo match(token.getType()).

Esempio

Il parsing inizierà con la chiamata della funzione associata con il simbolo iniziale, parseS.

Vogliamo che alla fine sia stato consumato tutto l'input, cioè si trovi il token $ che denota la fine del programma.

Per il momento consideriamo che il parsing riconosca le stringhe del linguaggio generato, cioè ci basta non produrre un ErroreSintattico chiamando la funzione parseS. Possiamo considerare in modo alternativo di far ritornare un booleano che sia true se avete fatto il parsing corretto!

Funzioni parseS, parseE, parsePr e parse Tl

//FUN è il tipo del token per "f", PARA per "(" VAL per "v" PARC per ")" PLUS per "+"

```
parseS(){
    Token token=peekToken()
    switch (token.getType()) {
        case TokenType.FUN:
        case TokenType.PARA:
        case TokenType.VAL: // produzione S -> E $
            parseE()
            match(TokenType.EOF) // EOF e' il tipo del token per "$"
    }
    ErroreSintattico
}

parseE(){
    Token token=peekToken()
    switch (token.getType()) {
        case TokenType.FUN:
        case TokenType.PARA: // produzione E -> Pr ( E )
            parsePr()
            match(TokenType.PARA)
            parseE()
            match(TokenType.PARC) //
            return
        case TokenType.VAL: // produzione E -> v Tl
            match(TokenType.VAL)
            parseTl()
            return
    }
    ErroreSintattico
}

parsePr(){
    Token token=peekToken()
    switch (token.getType()) {
        case TokenType.FUN: // produzione Pr -> f
            match(TokenType.FUN)
            return
        case TokenType.PARA: // produzione Pr -> eps
            return
    }
    ErroreSintattico
}
```

```
parseTl(){
    Token token=peekToken()
    switch (token.getType()) {
        case TokenType.PLUS: // produzione Tl -> + E
            match(TokenType.PLUS)
            parseE()
            return
        case TokenType.PARC:
        case TokenType.EOF: // produzione Tl -> eps
            return
    }
    ErroreSintattico
}
```

Parser basato su tabelle

Come alternativa all'implementazione con insieme di funzioni ricorsive, il parser può essere implementato simulando un Automa a Pila (diverso dall'automa a pila shift-reduce usato per l'analisi LR).

Lo stack contiene simboli terminali e non terminali del linguaggio:

→Inizialmente ho S cioè il non terminale iniziale della grammatica,

→Alla fine mi aspetto di essere alla fine della stringa e avere lo stack vuoto (se la stringa è stata riconosciuta).

La tabella di parsing: array bidimensionale M: $V \times \Sigma \cup \{\$\} \rightarrow P$, che associa ad un simbolo non terminale A e un simbolo terminale a (che potrebbe anche essere $), una produzione $p \in P$.

Le azioni del parser

Se X è il simbolo top della pila e *a* il simbolo di ingresso, ci sono 4 mosse possibili:

1) X = $ e *a* = $, ⇒ il parsing termina con successo

2) X = a (cioè il top della pila è uguale al simbolo letto), ⇒ fare pop della pila e nextToken() (avanzare di un token l'input)

3) $X \in V$ (X è un simbolo non terminale) ⇒ se M(X, a) = p e p: $X \rightarrow X1 \cdot \cdot \cdot Xn$ si eseguono le mosse seguenti

   1)pop della pila (cioè rimuovere X)

   2)push di $Xn \cdot \cdot \cdot X1$ sulla pila

   3)output la produzione p : $X \rightarrow X1 \cdot \cdot \cdot Xn$ che indica che abbiamo usato questa produzione

4) Nessuno dei casi precedenti → ErroreSintattico (in particolare se M(X, a) è una casella vuota!)

Esempio. Si parta dalla grammatica:

0. S → E $ 1. E → Pr (E) 2. E → v Tl 3. Pr → f 4. Pr → ϵ 5. Tl → + E 6. Tl → ϵ

Costruiamo la tabella per il nostro esempio e poi simuliamo l'esecuzione del parser sulla stringa di input: f (v + v) $

Per costruire la tabella usiamo Predict delle produzioni:

Produzione	0. *S*	1. *E*	2. *E*	3. Pr	4. Pr	5. *Tl*	6. *Tl*
Predict	{ f ,v ,(}	{ f ,(}	{ v }	{ f }	{ (}	{ + }	{),$ }

Tabella

	f	v	(	)	+	$
S	0.	0.	0.			
E	1.	2.	1.			
Pr	3.		4.			
Tl				6.	5.	6.

Input	Pila
f(v+v) $	*S*
f(v+v) $	*E* $
f(v+v) $	*Pr* (*E*) $
f(v+v) $	f (*E*) $
(v+v) $	(v *Tl*) $
v+v) $	v *Tl*) $
+v) $	*Tl*) $
+v) $	+ *E*) $
v) $	*E*) $
v) $	v *Tl*) $
) $	*Tl*) $
) $	) $
$	$

Grammatiche NON LL(1)

Una grammatica ricorsiva sinistra non può essere LL(1). Ad esempio, **E → E + T | E − T | v**

Perché? Assumi A → A α | β

→Se a ∈ First(β) allora a ∈ First(A α), perché A ⇒ A α ⇒ β α

→E β = ϵ allora a ∈ Follow(A) se a ∈ First(α) e quindi anche a ∈ First(A α), perché A ⇒ A α ⇒ α

Una grammatica con prefissi comuni, cioè 2 o più produzioni per lo stesso non terminale hanno la stessa parte iniziale, non può essere LL(1).

Ad esempio, **S → if E then E | if E then E else E.**

Perché?

→se A → α β1 | α β2, se a ∈ First(α β1) allora a ∈ First(α β2), e viceversa.

Ma il linguaggio generato può essere LL(1), se troviamo un'altra grammatica LL(1) che lo genera!

Trasformazioni per grammatiche

→Rimozione della ricorsione sinistra (Per le trasformazioni in forma normale di Greiback!)

→Fattorizzazione

Rimozione ricorsione sinistra

Le produzioni

A → A β1 | A β2 | ▪ ▪ ▪ | A βn dove n > 0

A → α1 | α2 | ▪ ▪ ▪ | αm dove m > 0

vengono rimpiazzate da

A → α1 A' | α2 A' | ▪ ▪ ▪ | αm A' dove m > 0

A' → ε |β1 A' | β2 A' | ▪ ▪ ▪ | βn A' dove n > 0

dove A' è un nuovo non-terminale usato per eliminare la ricorsione immediata a sinistra.

Fattorizzazione

Per ogni non terminale A consideriamo le produzioni

A → α β1 | α β2 | ▪ ▪ ▪ | α βn dove n > 1 e dove α è il prefisso comune più lungo

Rimpiazziamo queste produzioni con

A → α A'

A' → β1 | β2 | ▪ ▪ ▪ | βn

dove A' è un nuovo non-terminale.

Esempio

Definire una grammatica equivalente che sia LL(1).

St → if Exp then Sts endif

St → if Exp then Sts else Sts endif

Sts → Sts St;

Sts → St;

Exp → var + Exp

Exp → var

Trasformazioni

Fattorizzazione (delle produzioni per St e Exp)

St → if Exp then Sts St′

St′ → endif

St′ → else Sts endif

Sts → Sts St;

Sts → St;

Exp → var Exp′

Exp′ → + Exp

Exp′ → ε

Rimozione ricorsione sinistra dalle produzioni per Sts

A → A β | α *senza ricorsione sinistra diventa* A → α A'e A' → β A'| ε

St → if Exp then Sts St′

St′ → endif

St′ → else Sts endif

Sts → St; Sts′

Sts′ → St; Sts′

Sts′ → ϵ

Exp → var Exp′

Exp′ → + Exp

Exp′ → ε

Segnalare gli errori

Nel parsing top-down è facile segnalare gli errori sintattici.

Consideriamo il parser a discesa ricorsiva, falliamo quando:

→Il match non trova il token del tipo giusto nell'input

→Il token dell'input non è generato da una produzione del non-terminale che ci aspettavamo

In entrambi i casi possiamo segnalare il token se cui si è manifestato l'errore. Per questo nei token dovrebbe essere memorizzata la riga del programma sorgente nella quale è il token.

Recupero dagli errori

→Uno dei meccanismi usati è il panic mode, cioè, trovato l'errore il parser scorre i token fino a trovare un delimitatore frequente, ad esempio; e chiama parseA() dove A è un non terminale che che deriva una stringa che segue il delimitatore. Ad esempio, nel linguaggio ac ricomincerebbe a fare il parsing da un non terminale da cui si derivano statements.

→Un altro meccanismo si basa sul raccogliere durante il parsing quei token che seguono la chiamata del parsing di un certo simbolo non terminale e cercare di completare la chiamata corrente scorrendo i token fino a trovarne uno in quell'insieme.

Io non vi chiederò di fare il recupero dagli errori, ma sei volete potete farlo.

Come calcolare il first, il follow e il predict

1)Per prima cosa vedo se la regola deriva la stringa vuota oppure no. Questa cosa la faccio per tutte le regole presenti in questo linguaggio.

2)Dopodiché calcolo il first. Il first è il primo simbolo che appare dopo la freccia nelle regole di produzione. A seconda del simbolo che si trova ci comportiamo in diversi modi:
→Se il carattere dopo la freccia è la stringa vuota ovvero ε allora il first è l'insieme vuoto.
→Se il carattere dopo la freccia è un terminale allora il first è il terminale stesso.
→Se il carattere dopo la freccia è un non terminale allora il first sarà pari al first di quel terminale e se tale first non è stato ancora calcolato si calcola prima il first di quel non terminale.

3)Dopodiché calcolo il follow. Per il follow bisogna guardare per ogni non terminale tutte le sue occorrenze nella parte di destra della freccia. Dopodiché dobbiamo guardare quale simbolo segue quel non terminale.
→Se il simbolo che segue è un terminale allora il follow sarà uguale al terminale trovato.
→Se il simbolo che segue è un non terminale allora il follow sarà il first del non terminale nel caso in cui questo non terminale non deriva la stringa vuota ε. Nel caso in cui il non terminale che segue derivi la stringa vuota ε allora il follow sarà uguale al first del non terminale unito al follow del non terminale.
→Se invece non c'è nessun simbolo che segue il non terminale allora il follow di quel non terminale sarà pari al follow del non terminale che produce tale produzione. Ad esempio:

A→ α T allora il follow(T)=follow(A)

4)Infine calcolo il predict. Se la regola non deriva la stringa vuota ε allora il predict è uguale al first mentre se la regola deriva la stringa vuota ε allora il predict è uguale al follow.

ESERCIZIO SUL FIRST, FOLLOW E PREDICT

Num.	LHS	RHS	Deriva ϵ	First	Follow	Predict
0	Prg	Dss $	No	=First(DSs)={float, int, id, print, $}	----------	{float, int, id, print, $}
1	DSs	Dcl DSs	No	=First(Dcl)={float, int}	{$}	{float, int}
2	DSs	Stm DSs	No	=First(Stm)={id, print}		{id, print}
3	DSs	ϵ	Sì	insieme vuoto		={$}
4	Dcl	Ty id Dcl'	No	=First(Ty)={float, int}	=First(DSs)+Follow(DSs)={float, int, id, print,$}	{float, int}
5	Dcl'	;	No	{;}	={$}Ufollow(Dcl)={float, int, id, print,$}	{;}
6	Dcl'	= Exp;	No	{=}		{=}
7	Stm	id = Exp;	No	{id}	=First(DSs)+Follow(DSs)={float, int, id, print,$}	{id}
8	Stm	print it;	No	{print}		{print}
9	Exp	Tr ExpP	No	=First(Tr)={intVal, floatVal, id}	{;}	{intVal, floatVal, id}
10	ExpP	+ Tr ExpP	No	{+}	=Follow(Exp)={;}	{+}
11	ExpP	- Tr ExpP	No	{-}		{-}
12	ExpP	ϵ	Sì	insieme vuoto		{;}
13	Tr	Val TrP	No	=First(Val)={intVal, floatVal, id}	=First(ExpP)UFollow(ExpP)={+,-,;}	{intVal, floatVal, id}
14	TrP	* Val TrP	No	{*}	=Follow(Tr)={+,-,;}	{*}
15	TrP	/ Val TrP	No	{/}		{/}
16	TrP	ϵ	Sì	insieme vuoto		{+,-,;}
17	Ty	float	No	{float}	{id}	{float}
18	Ty	int	No	{int}		{int}
19	Val	intVal	No	{intVal}	=First(TrP)UFollow(TrP)={*,/,+,-,;}	{intVal}
20	Val	floatVal	No	{floatVal}		{floatVal}
21	Val	id	No	{id}		{id}