**Appunti SLIDE INTRODUZIONE DEI COMPILATORI**

**Processamento dei linguaggi:** Preparare un programma per far sì che sia eseguibile su un computer.

Assicura che il programma conformi con la semantica intesa.

Traduce in una forma più facilmente eseguibile da computer. I due estremi sono:

**1)Interprete,** gira il programma esaminando la struttura dei costrutti e simulandone le azioni.

**2)Compilatore,** traduce il programma in istruzioni macchina eseguibili direttamente nel computer.

**Linguaggio macchina puro:** Codice per un particolare set di istruzioni macchina non assumendol’esistenza di alcun sistema operativo o libreria di funzioni**.** Questo approccio è molto raro, relegato a linguaggi perimplementazione di sistemi operativi o applicazioni “embedded”.

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

**Assembler:** Viene prodotto un file di testo contenente il codice sorgente assembler. Un certo numero di decisioni nella generazione del codice (target delle istruzioni di salto, forma degli indirizzi) sono lasciate all’assemblatore.

Molte volte il C è usato come assembler. Il C viene chiamato linguaggio assembler universale, per il fatto che è relativamente a basso livello, ma anche indipendente dalla piattaforma.

**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 dascrivere.

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.

**Vantaggi:**

1)Semplificare un compilatore fornendo le primitive adatte (ad esempio come chiamate di metodo, manipolazione di stringhe).

2)Diminuzione della dimensione del codice generato se le istruzioni sono progettate per una particolare linguaggio di programmazione (per esempio codice JVM per Java)

3)Trasportabilità del compilatore ovvero la portabilità del mio codice.

4)Possibilità di verificare che il codice non contenga istruzioni “maliziose”

**TRADUZIONE DIRETTA DALLA SINTASSI**

Sarebbe difficile progettare un algoritmo di traduzione di un linguaggio complesso senza prima analizzare la frase sorgente nei suoi costituenti definiti dalla sintassi.

**L’analisi sintattica riduce la complessità del problema del calcolo della traduzione, decomponendo il problema in sotto-problemi più semplici.**

**(In modo simile l’analisi lessicale riduce la complessità dell’analisi sintattica.)**

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 modularizzato secondo la struttura descritta dall’albero sintattico della frase.

La traduzione diretta dalla sintassi è diretta da produzioni.

**Questo tipo di traduttori è detto guidato dalla sintassi.**

**APPROCCI ALLA TRADUZIONE**

1)approccio in due fasi:

1.1) Il parser sintetizza **l’Abstract Syntax Tree (AST**), che è un albero di parsing in forma astratta nella fase di analisi sintattica.

1.2) Si utilizza l’AST per l’analisi semantica e poi per la sintesi 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.

Nel primo caso si ha a disposizione l’AST per fare analisi semantica ed elaborazione dell’output (si possono fare diverse passate sull’AST).

Nel secondo 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 (ha molti meno nodi!).

Immagine che contiene testo

Descrizione generata automaticamente

Manca una t alla Start 🡪 Stmt $

**Linguaggio a parentesi bilanciate.**

**NODI DI ABS PER IL LINGUAGGIO**

Sintetizzano l’informazione derivata dal parsing. Alcuni alberi standard per costrutti di linguaggi.

1) if, 2) while, 3) assegnamento, 4) operatori binari, 5) blocco

**STRUTTURA DI UN COMPILATORE**

Ci sono due fasi fondamentali in un compilatore:

**Analisi:** viene creata una rappresentazione intermedia del programma sorgente.

**1)Analisi Lessicale** è il processo che prende in ingresso una sequenza di caratteri e produce in uscita una sequenza di token.

**2)Analisi Sintattica (o parsing)** è un processo che analizza un flusso continuo di dati in ingresso in modo da determinare la correttezza della sua struttura grazie ad una data grammatica formale. Un parser è un programma che esegue questo compito.

**3)Analisi Semantica**

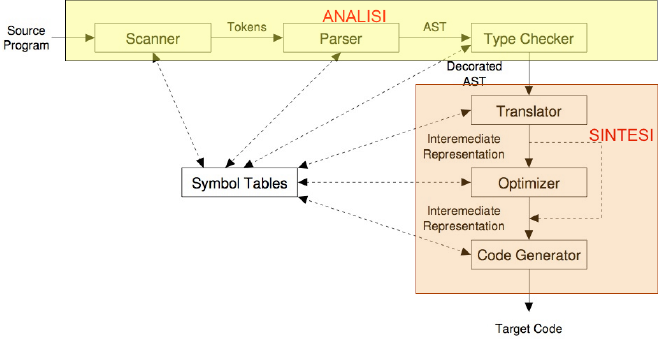
**Sintesi:** viene creato, partendo dalla rappresentazione intermedia, il programma target equivalente

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

**Fasi di compilazione con compilatore a due fasi**



Nella fase di analisi ci sono le analisi lessicale, sintattica e semantica

**1)ANALISI LESSICALE**

**Lo scanner legge** **l'istruzione e genera un token e lo passa al parser.**

**L’analisi lessicale (scanner)** legge il programma sorgente e produce i token delle istruzioni lette.

In questa fase si scoprono errori lessicali come, per esempio, aver inserito un carattere che non doveva essere messo in quel punto (ad esempio, il simbolo del $ al posto del + per una somma) e inoltre in questa fase si eliminano informazioni non necessarie come ad esempio i commenti.

**Token:** descrive un insieme di caratteri che hanno lo stesso significato come ad esempio, identificatori, operatori, keywords, numeri, delimitatori. Astrazione di una certa stringa sintattica che in un linguaggio diventa standard come, ad esempio, il ++ o l'operatore if ecc. Le espressioni regolari sono usate per descrivere i token.

**Gli automi a stati finiti deterministici possono essere usati per implementare un analizzatore lessicale.**

Esempio analisi lessicale

Input🡪 newVal = oldVal + 1 - 3.2

Output🡪 ⟨id : newVal⟩⟨assign⟩⟨id : oldVal⟩⟨plus⟩⟨intNum : 1⟩⟨minus⟩⟨floatNum : 3.2⟩

**2)ANALISI SINTATTICA**

In questa fase si riconosce se un token è derivabile.

La sintassi di un linguaggio è specificata per mezzo di una grammatica context-free (CFG).

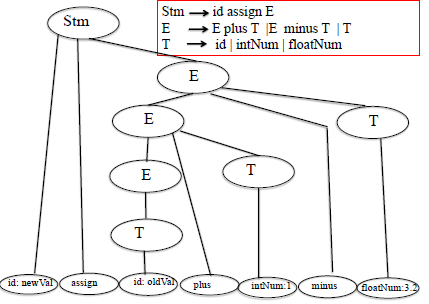
Il parser legge la sequenza di tokens e riconosce la struttura sintattica, sotto forma di un albero sintattico. Cioè riconosce quali produzioni sono state usate per generare la sequenza di token.

Un analizzatore sintattico verifica se un dato programma è derivabile dal simbolo iniziale della grammatica del suo linguaggio, cioè verifica se soddisfa le regole derivate dalle produzioni della sua CFG. In questa fase si trovano errori sintattici.

Dalla sequenza di token

⟨id : newVal⟩⟨assign⟩⟨id : oldVal⟩⟨plus⟩⟨intNum : 1⟩⟨minus⟩⟨floatNum : 3.2⟩

ottengo un parse tree dal quale posso capire le produzioni che sono state usate per generare la stringa dalla grammatica:



**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.

In relazione a come viene costruito il parse-tree ci sono due differenti tecniche:

**1)Top-Down**

La costruzione inizia dalla radice e procede verso le foglie. Efficienti parser top-down possono facilmente essere costruiti a mano. **Parsing a discesa ricorsiva predittivo** (implementato con funzioni mutuamente ricorsive), **Parsing predittivo non a discesa risorsiva** (implementato usando

l’automa pushdown).

Viene prodotta la derivazione “Leftmost” della stringa analizzata.

**2)Botton-Up**

La costruzione inizia dalle foglie e procede verso la radice, producendo la derivazione “Rightmost” della stringa analizzata. Di norma efficienti parser bottom-up parsers sono creati con l’ausilio di tools software. I parser Bottom-up sono anche conosciuti con il nome di parser shift-reduce.

Ci sono molte variazioni di LR Parsing: LR, SLR, LALR

**3)ANALISI SEMANTICA**

Controlla che le variabili siano correttamente dichiarate e che i tipi siano corretti. Queste informazioni non possono essere rappresentate con una grammatica context-free.

In questa fase, nel caso di compilatore a 2 fasi, il **type-checker** decora l’AST aggiungendogli le informazioni di tipo. Questa fase dipende dalle regole semantiche del linguaggio sorgente (è indipendente dal linguaggio target).

In Java vengono fatti ulteriori controlli durante l’analisi semantica:

🡪Raggiungibilità del codice

🡪Inizializzazione variabili prima dell’uso

🡪Corretto ritorno di valori da funzioni/metodi

**Generazione di un codice intermedio**

Un compilatore può produrre una rappresentazione esplicita del codice intermedio. Questo codice intermedio (o Intermedie Representation = IR) è indipendente dall’architettura del computer per cui si genera codice, ma può essere più o meno ad alto livello.

Esempio:

newVal = oldVal + 1 - 3.2 potrebbe essere tradotto in

ADD id2,#1,temp1

SUB temp1,#3.2,temp2

MOV temp2,id1

La generazione del codice è diretta dalla sintassi.

**GENERAZIONE CODICE**

La generazione del codice richiede che venga catturata la semantica dinamica (cosa fa a runtime) di un costrutto.

Per esempio, l’AST di un “while” contiene due sotto-alberi, uno per controllare l’espressione di controllo, e l’altro per il corpo del ciclo, ma non dice che un “while” ripete il body. Questo è catturato quando un AST di un ciclo while viene tradotto.

Nella IR, la nozione di verificare il valore dell’espressione di controllo e dell’esecuzione condizionale del corpo del ciclo diventa esplicito.

**Codice intermedio**

Il codice intermedio ha poco della macchina di destinazione. Le informazioni sulla macchina di destinazione (operazioni disponibili, indirizzamento, registri, ecc.) sono riservate per la fase finale di

generazione del codice.

In semplici compilatori non ottimizzati il traduttore genera il codice di riferimento direttamente, senza utilizzare una IR.

In compilatori complessi, ad esempio GNU Compiler Collection (GCC) prima si genera un IR di alto livello (orientato al linguaggio sorgente) e poi questo viene tradotto in un IR a basso livello (orientato alla macchina). Questo approccio consente di separare le dipendenze derivanti dal sorgente e dal target.

**Ottimizzazione**

Il codice IR generato dal traduttore viene analizzato e trasformato in codice IR equivalente ottimizzato. Un ottimizzatore ben progettato può incrementare in modo significativo la velocità di esecuzione. Le ottimizzazioni possono essere:

-spostare o eliminare operazioni non necessarie

-rimuovere codice non raggiungibile

**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. Per esempio:

y = 1

y = 2

x = y

il primo assegnamento è inutile, ma non è facile da capire, mentre nella sua forma SSA

y\_1 = 1

y\_2 = 2

x\_1 = y\_2

il risultato è immediato!

**GENERAZIONE CODICE**

Il codice IR prodotto dal traduttore viene tradotto nel codice della macchina target.

IL programma target è normalmente un codice oggetto “rilocabile” contenente codice macchina.

Questa fase fa uso di informazioni dettagliate sulla macchina target e include ottimizzazione legate alla macchina specifica quali l’allocazione dei registri e lo scheduling del codice.

Il generatore di codice può essere piuttosto complesso. Per produrre buon codice target bisogna considerare di molti casi particolari.

È possibile generare i generatori di codice in modo automatico, definendo dei template che mettano in corrispondenza le istruzioni di un IR a basso livello con quelle della macchina target.

Il compilatore GNU GCC è un compilatore fortemente ottimizzato che usa files di descrizione per più di dieci architetture PC, e di almeno due linguaggi (C and C++).

**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. Ogni volta che un identificatore viene usato, la tabella di simboli consente di accedere alle informazioni raccolte quando è stato definito.

**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”).

**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**

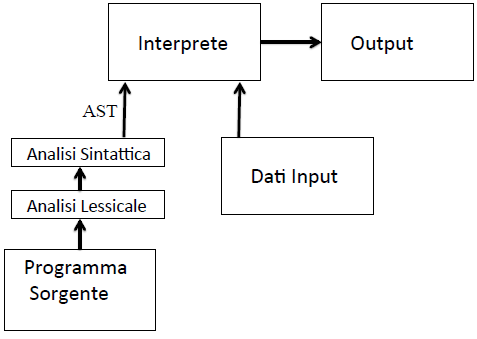
Simulano l’esecuzione di un programma compilato per una particolare architettura.

Java usa un interprete bytecode per simulare l’effetto del programma compilato per la Java Virtual Machine (JVM).

I programmi come SPIM simula l’esecuzione di programma MIPS su un computer non-MIPS (uso didattico).

**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.

**Il lessico**

Il lessico descrive, le parole o elementi lessicali che compongono le frasi. Nei linguaggi artificiali gli elementi lessicali possono essere assegnati alle seguenti classi:

1) Parole chiave: sono particolari parole fisse che caratterizzano vari tipi di frasi o strutture. Ad es.: if, for, class.

2) Delimitatori (;) operatori (+,++,..): come i precedenti sono delle parole fisse composte di caratteri anche non alfabetici.

3) Commenti in Java sono aperti da /\* e chiusi da \*/.

4) Classi lessicali aperte: queste comprendono un numero illimitato di elementi lessicali, che devono avere la struttura di un linguaggio regolare ossia a stati finiti. Esempi tipici sono:

🡪nomi o identificatori di variabili, di funzioni, o altro (classi, metodi, ...) definiti dalla espressione regolare: Id = (Lettera | \_) (Lettera | Cifra | \_)\*

🡪costanti quali i numeri interi o reali o le stringhe alfanumeriche.

**Analisi lessicale**

La differenza fra parole chiave e classi lessicali aperte:

1)Le parole chiave non hanno altra informazione che il proprio nome,

2)Le classi lessicali denotano delle entità che hanno un valore o altre proprietà (che chiameremo attributi semantici). Ad esempio, le costanti hanno un valore.

Le classi lessicali sono delle stringhe appartenenti ad un linguaggio formale del tipo regolare. Questi linguaggi sono descritti dalle espressioni regolari e riconosciuti dagli automi a stati finiti deterministici.

**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**

Fornire un modo per isolare le regole di basso livello dalle strutture che costituiscono la sintassi del linguaggio.

Suddividere la frase in ingresso in elementi lessicali (detti tokens) da fornire al parser.

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.

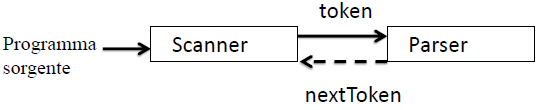
**Pattern:** una descrizione del lessico che corrisponde al token.

**Interazione tra scanner e parser**

Può avvenire in due modi:

1) Lo scanner processa tutto il programma sorgente prima che inizi il parsing (tutti i token sono memorizzati in un file o tabella).

2) Lo scanner è chiamato dal parser quando c’è bisogno di un altro token, per cui non si deve memorizzare la sequenza di token.



**Come realizzare un analizzatore lessicale**

Realizzazione procedurale: un programma ad-hoc che riconosce tutti gli elementi lessicali e produce i corrispondenti token.

🡪A partire dalle espressioni regolari

🡪A partire dall’automa a stati finiti

🡪A partire dalla grammatica regolare

Realizzazione tabulare interpretata: una struttura dati (tabella) rappresenta il DFA riconoscitore della grammatica G e un programma indipendente dalla grammatica G realizza il funzionamento del DFA.

Automaticamente con un generatore di Scanner, ad esempio, **JFLex** (che prende come input le espressioni regolari corrispondenti ai token).

**Realizzazione procedurale**

Dall’Espressione Regolare al Codice per l’analisi. Si scrive

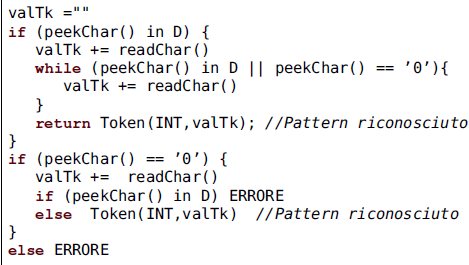
🡪una sequenza per ogni concatenazione

🡪un test per ogni unione

🡪un ciclo per ogni stella di kleene

Esempio, se D=1,...,9: l’espressione regolare per i letterali interi: D (D | 0)\* | 0

È riconosciuta dal seguente codice dove peekChar() restituisce il prossimo carattere senza rimuoverlo dall’input, mentre readChar() lo rimuove.



**Riconoscere un commento di linea**

L’espressione regolare, supponendo di avere Eol come simbolo di fine linea e Not che significa tutti i simboli eccetto quello specificato.

// Not (Eol) \*Eol

Immagine che contiene testo

Descrizione generata automaticamente

Dalla Grammatica Regolare al Codice per l’analisi. Si scrive:

1) Una funzione/metodo per ogni non terminale,

2) Un test per ogni alternativa,

3) Si richiama la funzione per ogni non-terminale che compare nella parte destra della produzione.

**Poco usata per i linguaggi regolari perché questi vengono in genere descritti da espressioni regolari.**

Come mai? Per il nostro compilatore useremo:

1) La realizzazione procedurale a partire dall’espressione regolare per l’analisi lessicale

2) La realizzazione procedurale a partire dalla grammatica per l’analisi sintattica, ciò per una grammatica context free (LL).

**Realizzazione tabulare**

Dall’Espressione Regolare all’Automa a Stati Finiti.

Si costruisce una tabella T tale che dato lo stato s e il carattere c, se T(s, c) = s′ allora s′ è lo stato successivo. Si definisce la funzione che esegue l’automa, che NON dipende dalla particolare espressione regolare.

valTk =""

State = StartState

**while** ( (peekChar() != eof) && (State non in F) {

NextState = T[State][peekChar()]

**if**(NextState == error)

**return** ERRORE

State = NextState

valTk += readChar()

}

**if** (State in F) **return** Token(\_,valTk) *// il token riconosciuto*

**else return** ERRORE *// errore lessicale*

Questa realizzazione è usata dai generatori di analizzatori lessicali. Perché?

**Identificatori e parole chiave**

Tutti i linguaggi utilizzano parole chiave: if, while,.....

Per queste sequenze di caratteri lo scanner deve generare token diversi da quelli degli identificatori.

Come può uno scanner decidere quando una sequenza di caratteri è un identificatore e quando è una parola chiave?

Lo scanner può procedere utilizzando il modello degli identificatori e poi cercare il token in una speciale tabella delle parole chiave. (Come potrebbe essere implementata per avere un lookup efficiente?)

Si definiscono espressioni regolari per ogni parola riservata, e per gli identificatori e si definisce una priorità fra le varie espressioni regolari.

Ad esempio

If i f

While w h i l e

Id [a-zA-Z] ([a-zA-Z] | [0-9] | \_)\*

Il matching è con la prima espressione regolare (dall’alto in basso!). Notate comunque che dobbiamo fare il maching più lungo, cioè, se ho la stringa whiler NON mi devo fermare al riconoscimento di while, ma devo andare avanti fino a whiler e determinare che questo è un identificatore.

**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.

I generatori di scanner (Lex e JFlex) creano automaticamente un token EOF.

**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.

**JFLex: un generatore di scanner**

Questo software, scritto interamente in Java, produce in uscita delle classi Java che implementano i metodi per effettuare l’analisi lessicale di una stringa.

La classe principale prodotta è una classe per lo Scanner (Yylex). Il costruttore ha come parametro il file che vogliamo scannerizzare. La classe contiene i metodi:

Token yylex() dove Token è la classe che vogliamo sia restituita dallo scanner che restituisce il prossimo token.

String yytext() ritorna la stringa letta per riconoscere il token.

Per funzionare, JFLex ha bisogno in ingresso un file di specifica, contenente una lista di espressioni regolari definite per il lessico. A ciascuna può essere associata una azione da compiere. Ogni volta che ’input matcha l’espressione regolate è eseguita l’azione (che in genere è la generazione di un Token, ma che per i caratteri di skip non fa niente).

**Derivazione left-most (o sinistra)**

Data una grammatica G = (V,Σ, P, S), una stringa w ∈ Σ∗ è nel linguaggio generato da G (w ∈ L(G)), se S →∗ w.

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

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 ci 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. Ad esempio: 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 che avete visto 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 non lo è.

**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. Abbiamo visto che per ogni produzione P c’è un FIRST che è un insieme di simboli terminali che sono prodotti all’inizio di una stringa generata dalla produzione P.

Ma come facciamo a sapere l’insieme di simboli terminali che sono prodotti all’inizio di una stringa generata una produzione che genera ϵ, ad esempio A→ϵ, ma anche A→BC e sia B che C derivano ϵ? Per questo useremo il FOLLOW!

!!!!!!!!!!!!!!!!-----vedi slide algoritmi first e follow-----!!!!!!!!!

**Algoritmo PREDICT**

Dobbiamo completare la tabella della grammatica aggiungendo per ogni produzione il suo insieme Predict che è calcolato usando il First e Follow delle produzioni.

Predict(p : A → X1 ・ ・ ・Xn)

ris = First(X1 ・ ・ ・ Xn)

if DerEmpty(p) then ris = ris∪Follow(A)

return ris

Abbiamo calcolato First e Follow, ora calcoliamo il Predict per la grammatica di pag. 3 (guarda foto)

**Parsing LL(1)**

Supponiamo di avere un insieme di simboli di tokens Tokens (questi 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 →∗ w A X1 ・ ・ ・ Xn

Supponendo che Q = {p : A → α ∈ P | a ∈ Predict(p)}

Si possono verificare i seguenti casi:

🡪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!)

🡪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.

🡪Q contiene esattamente una produzione. In questo caso si procede applicandola.

Ad esempio, rispetto al nostro esempio potremo avere

f (v + (v)) $ e S →∗ f (v Tl) $

in questo caso Q = {Tl → + E}, mentre

f (v + (v)) $ e S →∗ f (v + Pr (E)) $

Q = {Pr → ϵ}.

**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, p1, ....,pn associate ad A e

🡪Pred1, ....,Predn sono gli insiemi predict associati a p1, ....,pn,

🡪**allora Predi ∩ Predj = ∅** per tutti 1 ≤ i ̸= j ≤ n

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

**Esempio**

La seguente grammatica è LL(1)?

0. S → E $

1. E → T E′

2. E′ → − E

3. E′ → ϵ

4. T → F T′

5. T′ → / T

6. T′ → ϵ

7. F → int

8. F → (E)

**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 ∈ 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()

se nextTk ∈ Predict(p1) allora //codice per p1

...

se nextTk ∈ Predict(pn) allora //codice per pn

altrimenti ErroreSintattico

supponiamo che p : A → X1 ・ ・ ・ 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 Alternativamente fate 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 × Σ ∪ {$} → P, che associa ad un simbolo non terminale A e un simbolo terminale a (che potrebbe anche essere $), una produzione p ∈ 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 ∈ V (X è un simbolo non terminale) ⇒ se M(X, a) = p e p: X → X1 ・ ・ ・ Xn si eseguono le mosse seguenti

1)pop della pila (cioè rimuovere X)

2)push di Xn ・ ・ ・X1 sulla pila

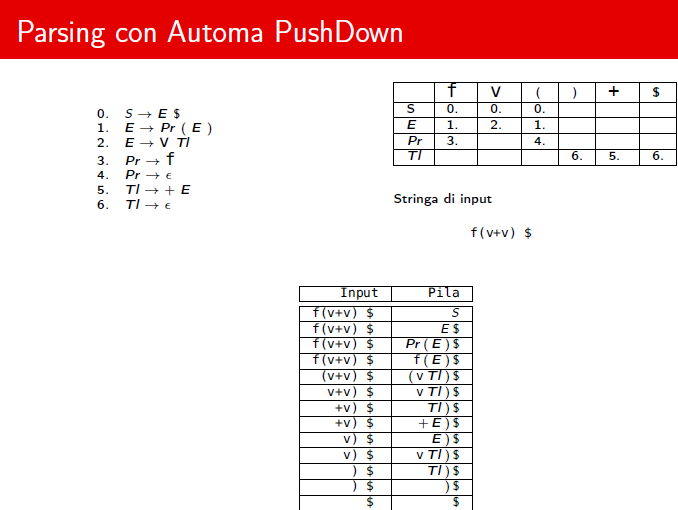
3)output la produzione p : X → X1 ・ ・ ・Xn che indica che abbiamo usato questa produzione

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

**Esercizio**

Immagine che contiene tavolo

Descrizione generata automaticamente



**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), perchè A ⇒ A α ⇒ β α

🡪se β = ϵ allora a ∈ Follow(A) se a ∈ First(α) e quindi anche a ∈ First(A α), perchè 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 (lo avete visto 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′ m > 0

A′ → **ε** |β1 A′ | β2 A′ | ・ ・ ・ | βn A′ n > 0

dove A′ è un nuovo non-terminale.

**Fattorizzazione**

Per ogni non terminale A consideriamo le produzioni

A → α β1 | α β2 | ・ ・ ・ | α βn n > 1 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 🡪 A → α A′, 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.