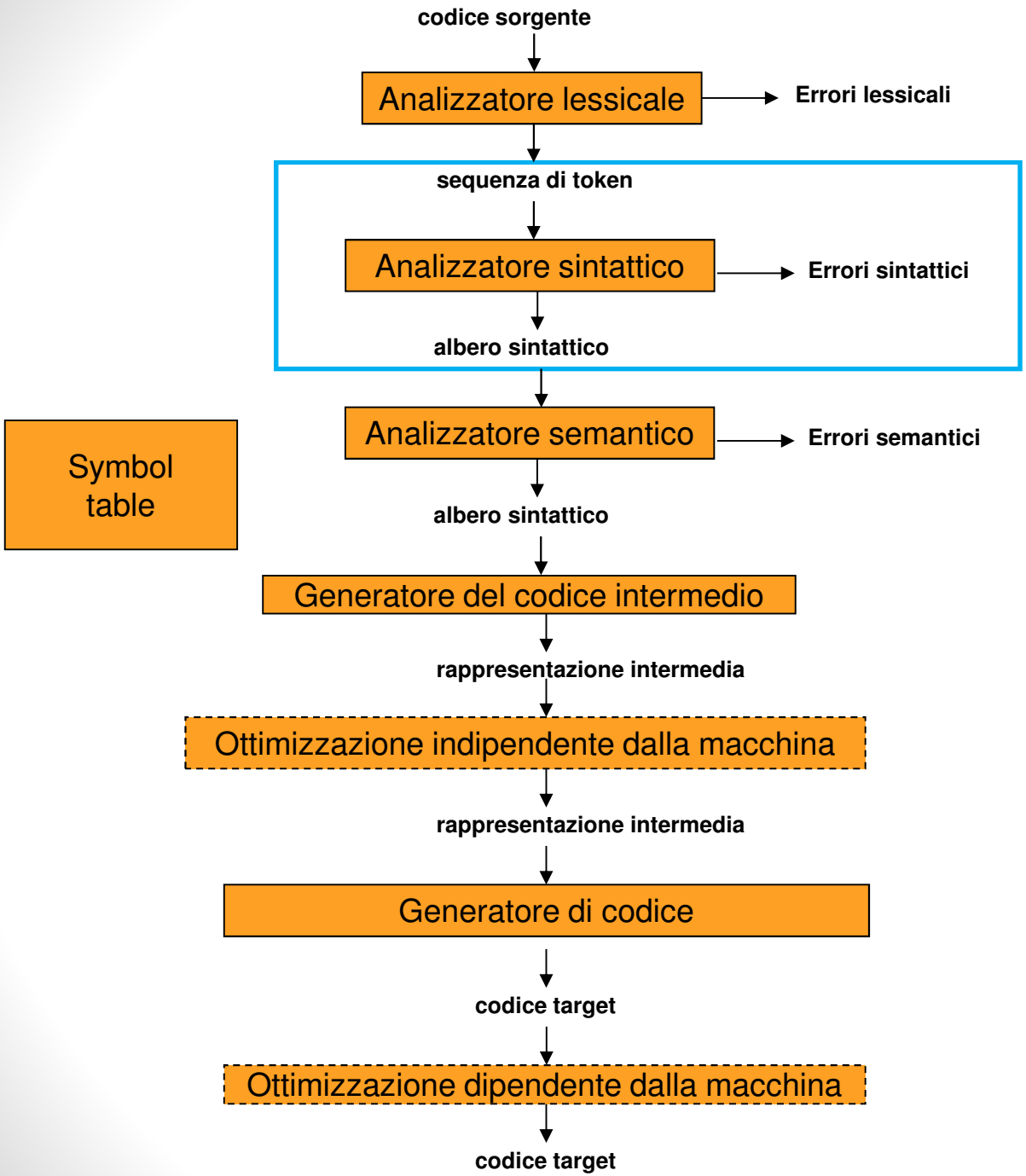
Analisi sintattica e grammatiche context-free

Lunedì 22 Ottobre

Le fasi di un compilatore



Perché parliamo di linguaggi context-free nell'ambito dei compilatori?

- La sintassi dei costrutti di un comune linguaggio di programmazione può essere descritta da una grammatica context-free.
- Un linguaggio context-free è generato da una grammatica context-free.

Grammatiche context-free

Una grammatica context-free (CFG) è una quadrupla G=(T,N,S,P) dove:

- T è l'alfabeto dei simboli terminali (= token lessicali);
- N è l'alfabeto dei simboli intermedi o variabili o non terminali (= categorie grammaticali);
- $S \in N$ è l'assioma;
- P è l'insieme delle produzioni o regole grammaticali della forma
  $A \rightarrow \alpha$, dove $A \in N$ e $\alpha \in (N \cup T)^*$

  Esempi :

  instr $\rightarrow$ **if** *expr* **then** *instr* **else** *instr*

  frase $\rightarrow$ *soggetto verbo complemento*

Linguaggio generato da una grammatica context-free

- Il linguaggio generato da una grammatica G è l'insieme delle stringhe di simboli terminali ottenute a partire dall'assioma con una o più derivazioni.
- Una derivazione consiste nell'applicazione di una sequenza di una o più produzioni: $\eta A\delta \Rightarrow \eta\alpha\delta$

Altri esempi

- Grammatica che genera la struttura di un libro:

  S->fA (f frontespizio, A serie di capitoli)
  A->AtB|tB (t titolo, B serie di righe)
  B->rB|r (r riga)

- Il linguaggio generato è l'insieme di tutte le stringhe che rappresentano la struttura corretta di un libro, ftrtrrr
- Tale linguaggio è anche regolare, $L=f(tr^{+})^{+}$
- Esistono linguaggi context-free non regolari, per esempio: $L=\{a^{n}b^{n}, n>0\}$

  S->aSb|ab

Linguaggi finiti e infiniti

- Linguaggio finito:
  S->aBc B->ab|Ca C->c
  L={aabc, acac}
- Linguaggio infinito
  G=({E,T,F}, {i,+,*,),(}, P, E)
  E->E+T|T T->T*F|F F->(E)|i
  L={i, i+i, i+i+i, i*i, (i+i)*i,…}
- Analizziamo le tre derivazioni:
  $E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow i+T \Rightarrow i+T*F \Rightarrow i+F*F \Rightarrow i+i*F \Rightarrow i+i*i$
  $E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*i \Rightarrow E+F*i \Rightarrow E+i*i \Rightarrow T+i*i \Rightarrow F+i*i$
  $\Rightarrow i+i*i$
  $E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow T+T*F \Rightarrow T+F*F \Rightarrow T+F*i \Rightarrow F+F*i$
  $\Rightarrow F+i*i \Rightarrow i+i*i$

Regole ricorsive

- Sia G una grammatica pulita o ridotta, cioè:
  1. ogni simbolo non terminale A è raggiungibile dall'assioma;
  2. ogni simbolo non terminale A genera un linguaggio non vuoto;
  3. non sono consentite derivazioni circolari: $A \Rightarrow^{*} A$
- Esiste un algoritmo che per "ripulire" una grammatica.

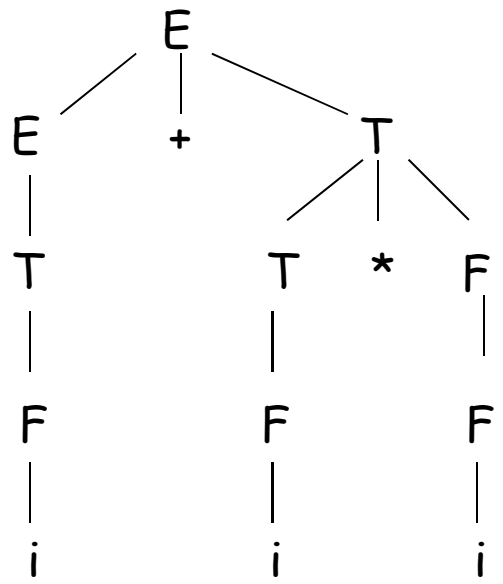
Condizione necessaria e sufficiente affinchè L(G) sia infinito è che G permetta derivazioni ricorsive, ovvero del tipo $A \Rightarrow^{n} xAy$

Alberi di derivazione o di parsing

Un ***albero di derivazione*** è una rappresentazione ad albero di una derivazione, in cui

- ogni simbolo è un nodo
- ogni simbolo è connesso al simbolo che l'ha generato

Nel caso dell'esempio precedente:



Lo stesso albero rappresenta le tre derivazioni precedenti.

Ogni frase di una grammatica CF può essere generata da una derivazione sinistra (oppure destra).

Se esiste una frase per cui è possibile trovare due derivazioni sinistre la grammatica si dice AMBIGUA.

Grammatiche ambigue

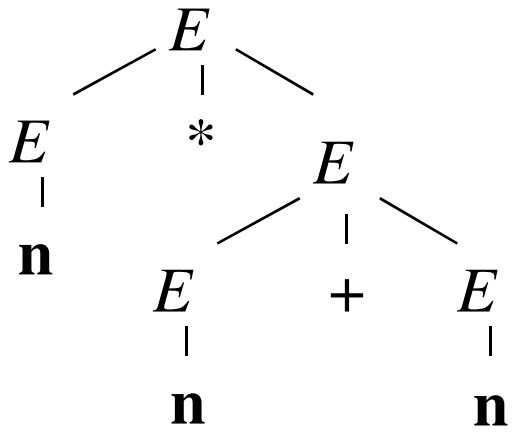
- **Una grammatica si dice ambigua** *quando ci sono due alberi di parsing differenti per la stessa frase (o, equivalentem., due derivazioni leftmost o sinistre per la stessa frase)*
- *Un linguaggio per cui esistono solo grammatiche ambigue si dice* **inerentemente ambiguo***;*
- ***Stabilire se una data CFG sia ambigua o se un dato linguaggio sia inerentemente ambiguo sono problemi indecidibili.***
- *Per alcune applicazioni si usano metodi che coinvolgono grammatiche ambigue unite alle regole che servono per eliminare le ambiguità.*
- *I costrutti per cui il parsing è difficile coinvolgono per lo più regole di precedenza o associatività*

Esempio di grammatica ambigua

Espressioni aritmetiche

$$E \rightarrow \mathbf{n}$$
$$E \rightarrow (E)$$
$$E \rightarrow E + E$$
$$E \rightarrow E - E$$
$$E \rightarrow E * E$$
$$E \rightarrow E / E$$

Parse tree di n*n+n



Esistono altri alberi di derivazione per n*n+n

Alcuni tipi di ambiguità

- Ricorsione sinistra e destra nella stessa regola:
  E->E+E|i i+i+i ha due derivazioni sinistre
  Si elimina stabilendo un ordine di derivazione, per esempio:
  E->i+E|i oppure E->E+i|i

- Ricorsione sinistra e destra in regole diverse:
  A->aA|Ab|c L=a*cb*
  Si stabilisce un ordine tra le derivazioni:
  S->aS|X X->Xb|c

Eliminare l'ambiguità

Grammatica non ambigua per espressioni aritmetiche (operatori postfissi):

$$E \rightarrow EE\ + \mid EE\ - \mid EE\ * \mid EE\ / \mid \mathbf{number}$$

Grammatica non ambigua per espressioni aritmetiche:

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow (E) \mid \mathbf{n}$$

Scelta: associatività a sinistra
precedenza di * e / su + e –

Ambiguità dell'else "pendente"

Un comune tipo di ambiguità riguarda le frasi condizionali.
Si consideri la grammatica

stmt->**if** expr **then** stmt | **if** expr **then** stmt **else** stmt | **other**

Ad esempio: **if** E_1 **then** **if** E_2 **then** S_1 **else** S_2 ha due parse tree.

Eliminazione dell'ambiguità

- **Idea:** basta distinguere gli statement in matched o completi (statement che contengono sia **then** che **else** e tale che sia dopo il then che dopo l'else ci siano statement matched) e statement open (statement con condizionali semplici o tali che il primo statement sia matched e il secondo open)

  Solo gli statement matched possono precedere l'else
- Quindi la grammatica diventa:

stmt-> matched_stmt | open_stmt
matched_stmt -> **if** expr **then** matched_stmt **else** matched_stmt
| **other**
open_stmt-> **if** expr **then** stmt
| **if** expr **then** matched_stmt **else** open_stmt

Esempio di linguaggio inerentemente ambiguo

- $L=\{a^i b^j c^k , i,j,k>=0$ con i=j oppure j=k$\}$

- Le stringhe $a^i b^i c^i$ hanno due alberi di derivazione distinti
- Una grammatica:

  S->XC|AY
  X->aXb|ε
  C->cC|ε
  Y->bYc|ε
  A->aA|ε

Costrutti non context-free

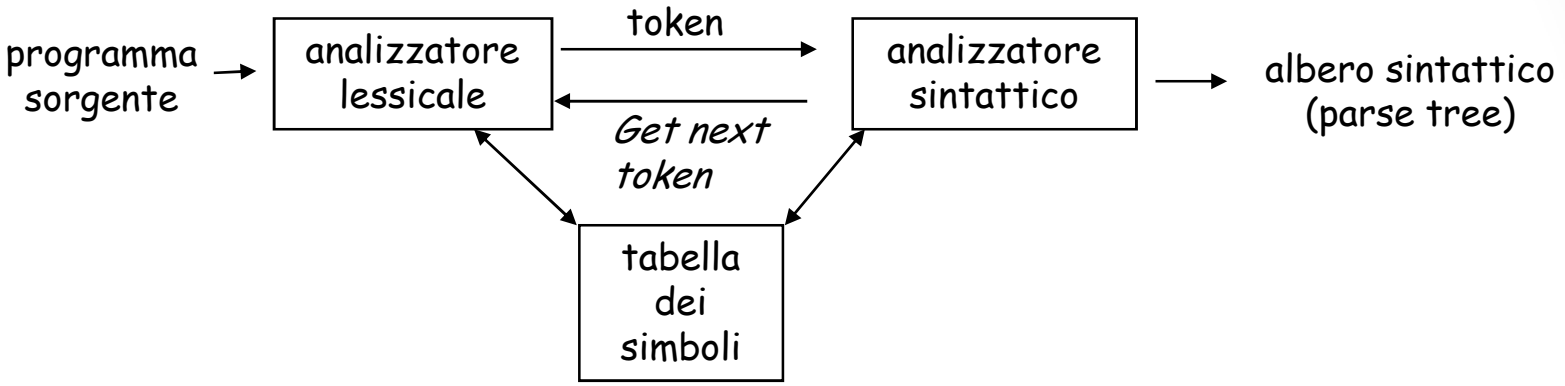
- $L=\{wcw \mid w \in (a|b)^*\}$
  Rappresenta il problema di controllare che gli identificatori siano dichiarati prima del loro uso.

- $L=\{a^n b^m c^n d^m \mid n,m>0\}$
  Rappresenta il problema di controllare il numero dei parametri formali di due funzioni coincida con quello dei parametri attuali delle rispettive funzioni.

Compito principale del parser

- Data una grammatica CF che genera un linguaggio L e data una frase x, rispondere alla domanda: x appartiene a L?
- Se la risposta è sì, esibire un albero di derivazione di x
- Se la risposta è no, esibire eventuali errori sintattici

Ruolo del parser nel processo di compilazione



Svolge un ruolo centrale nel front-end:

- attiva l'analizzatore lessicale con richieste
- verifica la correttezza sintattica
- costruisce l'**albero di parsing**
- gestisce gli errori comuni di sintassi
- prepara e anticipa la traduzione
- colleziona informazioni sui token nella symbol table
- realizza alcuni tipi di analisi sematica
- genera il codice intermedio.

Perché usare le grammatiche

- Una grammatica fornisce in modo semplice e facile da capire una specifica della sintassi di un linguaggio di programmazione
- A partire da certe classi di grammatiche è possibile costruire in modo automatico parser efficienti in grado di stabilire se un certo programma è ben formato.
- Un parser può rivelare alcune ambiguità sintattiche difficili da notare in fase di progettazione del linguaggi.
- Una grammatica progettata adeguatamente impartisce una struttura ad un linguaggio di programmazione che è utile per la traduzione in codice oggetto e per la ricerca degli errori.
- Aggiungere nuovi costrutti a linguaggi descritti mediante grammatiche è più facile.