Corso di Laurea Magistrale in Ingegneria Informatica
A.A. 2010-2011

Linguaggi Formali e Compilatori

Backus Naur Form (BNF)

Giacomo PISCITELLI

Descrizione compatta delle grammatiche: la BNF

Per descrivere in maniera compatta le grammatiche si usa la BNF (*Backus–Naur form*).

I terminali e i nonterminali vengono descritti da parole; per poterli distinguere, i terminali vengono scritti in **grassetto**.

Una regola BNF è data da un nonterminale e da un'espressione (anzichè una semplice sequenza di terminali e nonterminali).

Al posto di '$\rightarrow$' scriviamo '::='.

La regola spiega, come una produzione, il modo di riscrivere il nonterminale, ma ha una sintassi molto più ricca.

- ✓ Ogni terminale e ogni nonterminale è un'espressione BNF; una regola contenente solo un terminale o un nonterminale corrisponde in modo ovvio a una produzione.
- ✓ La giustapposizione di espressioni BNF è ancora un'espressione BNF.

Descrizione compatta delle grammatiche: la BNF

- La notazione $A ::= e_0 \mid e_1 \mid \cdots \mid e_{n-1}$ significa che la grammatica contiene le produzioni $A \rightarrow e_0$, $A \rightarrow e_1$,. . . , $A \rightarrow e_{n-1}$, e cioè che per riscrivere A bisogna scegliere uno degli e_i.
- La notazione [e] nel membro destro di una produzione significa che l'espressione e può essere usata o meno, cioè è opzionale.
- La notazione {e} nel membro destro di una produzione significa che l'espressione e può comparire zero o più volte.
- Le parentesi tonde possono essere utilizzate per raggruppare: per esempio, (a | b) [c] è diverso da a | (b [c]), che è uguale a a | b [c].

Descrizione compatta delle grammatiche: la BNF

Esempi

- parola ::= {**a**}
- parola ::= {**ab**} **a** | **b** {**ab**} **a** | ε
- ◦ cifra ::= **0** | **1** | **2** | · · · | **9**
  ◦ cifrapari ::= **0** | **2** | **4** | **6** | **8**
  ◦ numeropari ::= {cifra} cifrapari
- ◦ underscore ::= **_**
  ◦ lettera ::= **A** | **B** | · · · | **Z** | **a** | **b** | · · · | **z**
  ◦ cifra ::= **0** | **1** | **2** | · · · | **9**
  ◦ identificatore ::= (lettera | underscore){lettera | underscore | cifra}
- istr-if ::= **if (** espr **)** istr [**else** istr]
- ciclo-for ::= **for** ([espr] ; [espr] ; [espr]) istr
- ciclo-while ::= **while** (espr) istr
- ciclo-do ::= **do** istr **while** (espr);

Analisi lessicale

Dato che un file è una sequenza di caratteri, per poter stabilire se è un programma C occorre tokenizzarlo, cioè assegnare sequenze di caratteri ASCII a token del linguaggio C.

Per farlo, il compilatore utilizza una logica di massimizzazione: durante l'analisi lessicale, il massimo numero di caratteri consecutivi viene accumulato in un token.

Per esempio, se scrivete return0 invece di return 0 l'analizzatore lessicale decide che avete voluto specificare l'identificatore return0 invece di return seguito dal numero 0.

Al contrario, scrivere return(0) o return (0) è equivalente, perchè quando il compilatore arriva alla parentesi aperta si accorge che il token corrente è terminato, ed è return.

La BNF del linguaggio C: elementi lessicali

Le **parole chiave** sono token a cui sono assegnati significati particolari, e che non possono essere utilizzate dal programmatore:

auto	**do**	**goto**	**signed**	**unsigned**
break	**double**	**if**	**sizeof**	**void**
case	**else**	**int**	**static**	**volatile**
char	**enum**	**long**	**struct**	**while**
const	**extern**	**register**	**switch**	
continue	**float**	**return**	**typedef**	
default	**for**	**short**	**union**	

Gli **identificatori** sono un altro tipo di token, definiti in BNF come segue:

- lettera ::= **A** | **B** | · · · | **Z** | **a** | **b** | · · · | **z**
- cifra ::= **0** | **1** | **2** | · · · | **9**
- ident ::= (_ | lettera){_ | lettera | cifra}

La BNF del linguaggio C: elementi lessicali

Costanti

Le costanti (caratteri o numeri) rappresentano ancora un altro tipo di token.

costante ::= cost-car | cost-int | cost-float

cost-car ::= 'c' | 'seq-escape' dove c è un qualunque carattere ASCII tranne l'apostrofo, il backslash e il newline

seq-escape ::= **\'** | **** | **** | **\n** | **\t** | **\x**cifra-esa{cifra-esa}

cifra-esa ::= cifra | **A** | **B** | **C** | **D** | **E** | **F** | **a** | **b** | **c** | **d** | **e** | **f**

cost-int ::= (cost-dec | cost-esa) [su_-int]

nonzero ::= **1** | **2** | · · · | **9**

cost-dec ::= **0** | nonzero{cifra}

cost-esa ::= (**0x** | **0X**)cifra-esa{cifra-esa}

su_-int ::= [**U** | **u**][**L** | **l**]

cifre ::= cifra {cifra}

cost-float ::= cost-fraz [esponente] [su_-float] |
cifre esponente [su_-float]

cost-fraz ::= [cifre] . cifre | cifre .

esponente ::= [**e** | **E**][**+** | **-**] cifre

su_-float ::= **f** | **F** | **l** | **L**

Un programma C

programma ::= file {file}
file ::= {dichiarazione | def-fun}
def-fun := tipo dichiaratore**(**dich-param**)** istr-comp
istr-comp ::= **{** {dichiarazione} {istr} **}**
istr ::= istr-comp | istr-espr | istr-salto | istr-if | istr-iter | istr-switch | istr-vuota
istr-vuota ::= ;
istr-espr ::= espr;
istr-salto ::= **break**; | **continue**; | **return** [espr];
istr-if ::= **if (**espr**)** istr [**else** istr]
istr-switch ::= **switch**(espr)
{ {**case** espr-cost: {istr} } [**default:** {istr}] **}**
istr-iter ::= ciclo-for | ciclo-while | ciclo-do
ciclo-for ::= **for (**[espr] ; [espr] ; [espr]**)** istr
ciclo-while ::= **while (**espr**)** istr ciclo-do ::= **do** istr **while (**espr**)**;

Commento alle produzioni delle istruzioni (1)

programma ::= file {file}

Un programma C è composto da uno o più file.

file ::= {dichiarazione | def-fun}

Ogni file è composto da una sequenza di dichiarazioni (per esempio di variabili) e di definizioni di funzioni. Le variabili dichiarate nel corso del file sono globali, e sono visibili da qualunque funzione.

def-fun := tipo dichiaratore**(**dich-param**)** istr-comp

Una definizione di funzione richiede il tipo, il dichiaratore (che nel caso più semplice è il nome della funzione), la lista dei parametri e dei loro tipi fra parentesi, e infine il corpo (un'istruzione composta).

istr-comp ::= **{** {dichiarazione} {istr} **}**

Un'istruzione composta è un blocco, delimitato da parentesi graffe, che contiene una lista di dichiarazioni seguita da una lista di istruzioni.

istr ::= istr-comp | istr-espr | istr-salto | istr-if | istr-iter | istr-switch | istr-vuota

Un'istruzione può essere un'istruzione composta, un'istruzione formata da un'espressione, un'istruzione di salto, un'istruzione condizionale (if), un'istruzione di iterazione, un'istruzione di switch o l'istruzione vuota.

Commento alle produzioni delle istruzioni (2)

istr-vuota ::= ;

L'istruzione vuota non ha alcun effetto.

istr-espr ::= espr;

L'effetto di un'istruzione formata da un'espressione è la valutazione dell'espressione (il risultato viene scartato).

istr-salto ::= **break**; | **continue**; | **return** [espr];

Le istruzioni di salto servono a modificare il normale flusso di esecuzione.

istr-if ::= **if (**espr**)** istr [**else** istr]

Un'istruzione condizionale esegue la prima istruzione se l'espressione ha un valore diverso da zero, la seconda (se presente) altrimenti.

istr-switch ::= **switch**(espr)
{ {**case** espr-cost: {istr} } [**default:** {istr}] **}**

L'istruzione **switch** evita di usare un gran numero di espressioni condizionali. L'espressione viene valutata, e il controllo passa alla riga di programma preceduta da un **case** la cui espressione costante coincide con il valore di espr. L'esecuzione continua normalmente fino a un **break**, che fa uscire dall'intero **switch**. Se non è presente nessun case come sopra, viene eseguito, se presente, il codice che segue il **default:**.

Commento alle produzioni delle istruzioni (3)

istr-iter ::= ciclo-for | ciclo-while | ciclo-do

Le istruzioni di iterazione sono tre tipi di cicli.

ciclo-for ::= **for (**[espr] ; [espr] ; [espr]**)** istr

Nel ciclo **for** viene inizialmente valutata la prima espressione, quindi viene controllato il valore della seconda. Se è diverso da zero, si esegue l'istruzione, altrimenti si esce dal ciclo. Dopo l'esecuzione dell'istruzione, viene valutata la terza espressione, si torna a controllare il valore della seconda e così via.

ciclo-while ::= **while (**espr**)** istr

Nel ciclo **while** viene controllato il valore dell'espressione, e, se è diverso da zero, viene eseguita l'istruzione. Si torna quindi a controllare il valore dell'espressione, e così via.

ciclo-do ::= **do** istr **while (**espr**)**;

Il ciclo **do** è simile al ciclo **while**, ma l'istruzione viene eseguita prima di controllare il valore dell'espressione.

Espressioni C

espr ::= lvalue | costante | str-lett | **(** espr **)** | espr-ass | +espr | −espr | espr-fun | espr (oper-rel | oper-arit) espr | espr-log | **sizeof** espr | **sizeof (**tipo**)** | **(**tipo**)**espr | espr **?** espr **:** espr | **&**lvalue | **+ +**lvalue | **−−**lvalue | lvalue **+ +** | lvalue **−−**

lvalue ::= ident | _espr | espr**[**espr**]**

espr-ass ::= lvalue oper-ass espr

oper-ass ::= **=** | **+=** | **−=** | ***=** | **/=** | **%=**

oper-arit ::= **+** | **−** | ***** | **/** | **%**

oper-rel ::= **==** | **! =** |**<** |**>** |**<=** |**>=**

espr-log ::= **!**espr | espr **||** espr | espr **&&** espr

espr-fun ::= ident**(** [espr { , espr}] **)**

str-lett ::= **”** {*c* | seq-escape} **”**

dove *c* è un qualunque carattere ASCII tranne le virgolette, il backslash e il newline.

Commento alle produzioni delle espressioni (1)

lvalue ::= ident | _espr | espr**[**espr**]**

Prima di tutto notiamo che un lvalue (left value) è un particolare tipo di espressione che può stare a sinistra di un assegnamento.
Intuitivamente, un lvalue identifica una variabile, e quindi è possibile assegnarlo, incrementarlo e così via. Gli lvalue sono identificatori (nomi di variabili), dereferenziazioni di puntatori, e accessi a elementi di vettori.

espr ::= lvalue | costante | str-lett | **(** espr **)** | espr-ass | +espr | −espr | espr-fun | espr (oper-rel | oper-arit) espr | espr-log | **sizeof** espr | **sizeof (**tipo**)** | **(**tipo**)**espr | espr **?** espr **:** espr | **&**lvalue | **+ +**lvalue | **−−**lvalue | lvalue **+ +** | lvalue **−−**

Un'espressione è un lvalue, una costante (eventualmente stringa), un'espressione tra parentesi, un'espressione di assegnamento, un'espressione preceduta da **+** o **−**, una chiamata di funzione, una coppia di espressioni tra cui poniamo un operatore relazionale o aritmetico, un'espressione logica, un'espressione costante di tipo **sizeof**, una conversione di tipo, un'espressione costruita con l'operatore ternario, una referenziazione di un lvalue, e infine un pre/post incremento/decremento di un lvalue.

Commento alle produzioni delle espressioni (2)

espr-ass ::= lvalue oper-ass espr

oper-ass ::= **=** | **+=** | **−=** | ***=** | **/=** | **%=**

Un'espressione di assegnamento assegna all'lvalue il valore di espr nel caso l'operatore sia **=**. Se l'operatore è un altro, viene prima calcolato il valore dell'operazione indicata tra lvalue e espr, e poi viene assegnato a lvalue.

oper-arit ::= **+** | **−** | ***** | **/** | **%**

oper-rel ::= **==** | **! =** |**<** |**>** |**<=** |**>=**

Gli operatori aritmetici e logici permettono di costruire nuove espressioni.

espr-log ::= **!**espr | espr **||** espr | espr **&&** espr

Le espressioni logiche combinano i valori logici con una valutazione cortocircuitata (non appena è noto il valore dell'espressione la valutazione si ferma).

Commento alle produzioni delle espressioni (3)

espr-fun ::= ident**(** [espr { , espr}] **)**

Una chiamata di funzione passa il controllo alla funzione di nome ident, fornendo come lista di parametri attuali una lista di espressioni.

str-lett ::= ” {*c* | seq-escape} ”

dove *c* è un qualunque carattere ASCII tranne le virgolette, il backslash e il newline. Una stringa letterale è una sequenza di caratteri o di sequenze di escape (vedi la BNF delle costanti) racchiusa tra virgolette, e ha come valore un puntatore all’inizio della stringa.

Ambiguità

Una grammatica è detta ambigua se esistono due diverse sequenze di produzioni che generano la stessa parola:

espr → espr * espr → espr * espr + espr → 5 * 3 + 1

espr → espr + espr → espr * espr + espr → 5 * 3 + 1

La semantica della prima espressione vuol essere 20, quella della seconda 16.
Per risolvere il problema si definisce una precedenza tra le produzioni. Ad esempio, possiamo stabilire che una produzione

espr → espr + espr

debba sempre precedere, e mai seguire, una produzione

espr → espr * espr

Lo stesso problema si presenta se usiamo due volte lo stesso operatore, e l'operatore non è associativo (per esempio, **<**).
In tal caso viene scelta una direzione per l'associatività (da sinistra a destra o viceversa).

Ambiguità nelle istruzioni

L'ambiguità non è un problema relegato alle espressioni.

```
if ( x == 0 )
if ( y == 0 ) printf("pippo")
else printf("pluto");
```

Ci sono due modi di produrre il frammento precedente:

istr → **if (**espr**)** istr **else** istr → **if (**espr**) if (**espr**)** istr **else** istr

istr → **if (**espr**)** istr → **if (**espr**)** if **(**espr**)** istr **else** istr

Nel primo caso l'**else** è associato al primo **if**: se x non è zero, verrà stampato pluto, altrimenti se y è zero verrà stampato pippo.
Nel secondo caso l'**else** è associato al secondo **if**: se x non è zero, viene esaminato y: se quest'ultimo è zero verrà stampato pippo, altrimenti verrà stampato pluto.
Le due semantiche sono chiaramente diverse (la prima stampa pluto in ogni caso, una volta che x non è zero).
Anche qui siamo di fronte a un problema di associatività, che assumiamo verso destra: l'ultimo **else** è associato all'ultimo **if**.

Precedenza degli operatori

Operatori	Associatività
() [] . – >	da sinistra a destra
++ – – ! **sizeof** (tipo) + – (*unari*) & * (*puntatori*)	da destra a sinistra
* / %	da sinistra a destra
+ –	da sinistra a destra
< > <= >=	da sinistra a destra
== !=	da sinistra a destra
&&	da sinistra a destra
||	da sinistra a destra
?:	da destra a sinistra
= += –= *= /= %=	da destra a sinistra

Tipi delle espressioni

Quando si combinano più operandi di tipi diversi, il compilatore promuove gli operandi e determina il tipo dell'espressione risultante in maniera da non perdere informazione. Quindi se l'espressione contiene un **double**, tutti gli operandi sono convertiti a **double**; altrimenti, se l'espressione contiene un **float**, tutti gli operandi sono convertiti a **float**, e così via fino a **char**, che viene sempre promosso a **int**.

Il problema fondamentale è che se è presente un'espressione di tipo **int (long)** e una di tipo **unsigned int** (**unsigned long**, rispettivamente) tutti gli operandi sono convertiti nel tipo senza segno. Questo può dare risultati indesiderati.

Attenzione però: se un operando è un puntatore, è possibile solo aggiungergli o toglierli interi, e il risultato è un puntatore.

Per forzare il tipo di un'espressione si usano le conversioni di tipo (per esempio, **(int)f**, dove **f** è di tipo **float**).

Dichiarazioni in C

Le dichiarazioni C servono a due scopi: definire le variabili, e dare i prototipi delle funzioni non ancora definite, che servono al compilatore per conoscere il tipo dei loro argomenti e del loro risultato prima della loro definizione.

La sintassi delle dichiarazioni è una delle parti più complesse del linguaggio.

dichiarazione ::= tipo dichiaratore {, dichiaratore} ;
tipo dichiaratore ::= ([**unsigned**] [**char** | **int** | **long** | **short**]) | **void** | **float** | **double**
dichiaratore ::= {*****} dich-dir
dich-dir ::= ident | **(**dichiaratore**)** |
dich-dir**[** [espr-cost] **]** | dich-dir**(**dich-param**)**
dich-param ::= void | (tipo dichiaratore { , tipo dichiaratore })

Commento alle produzioni delle dichiarazioni (1)

dichiarazione ::= tipo dichiaratore {, dichiaratore} ;

Una dichiarazione è formata da un tipo e da uno o più dichiaratori.

tipo dichiaratore ::= ([**unsigned**] [**char** | **int** | **long** | **short**]) | **void** | **float** | **double**

I tipi disponibili sono quelli interi e in virgola mobile. Premettendo **unsigned** i tipi interi diventano senza segno. I tipi **long** e **double** sono versioni con maggiore precisione di **int** e **float**, rispettivamente, mentre **short** è un tipo intero con precisione minore di **int**.

dichiaratore ::= {*} dich-dir

Un dichiaratore è formato da un dichiaratore diretto preceduto da una sequenza di *. Ogni * indica una dereferenziazione del tipo del dichiaratore diretto: se il dichiaratore diretto era di tipo “qualcosa di x”, allora un * davanti a esso dichiara il tipo “qualcosa di puntatore a x”.

Commento alle produzioni delle dichiarazioni (2)

dich-dir ::= ident | **(**dichiaratore**)** |
dich-dir**[** [espr-cost] **]** | dich-dir**(**dich-param**)**

Un dichiaratore diretto può essere un identificatore, nel qual caso il suo tipo è quello che compare nella dichiarazione. Le parentesi possono essere utilizzate per alterare l'associatività. Un dichiaratore diretto può essere seguito tra parentesi quadre contenenti un'espressione costante; se il dichiaratore diretto era di tipo "qualcosa di x", allora il tipo risultante è di "qualcosa di vettore di espr-cost elementi di x" (se l'espressione costante è omessa, l'effetto è equivalente ad anteporre *****).

Infine, se il dichiaratore è seguito da una coppia di parentesi tonde, eventualmente contenenti una lista di dichiarazioni di parametri, si dichiara con lo stesso meccanismo una funzione che restituisce elementi del tipo del dichiaratore, e che ha parametri formali corrispondenti a quelli indicati.

Commento alle produzioni delle dichiarazioni (3)

dich-param ::= void | (tipo dichiaratore { , tipo dichiaratore })

Una dichiarazione di parametri è data da **void** (la funzione non ha argomenti), oppure da una lista di tipi e dichiaratori. Si noti che non è possibile elencare più dichiaratori per tipo (contrariamente a quanto avviene in generale).

Le dichiarazioni che compaiono al di fuori di un'istruzione composta sono visibili ovunque; altrimenti, sono visibili solo all'interno dell'istruzione composta stessa.

La logica dietro le dichiarazioni C è che un dichiaratore viene letto come la seguente asserzione: quando l'identificatore appare in un'espressione della stessa forma del dichiaratore, produrrà un oggetto del tipo indicato.

Il preprocessore di C

I programmi C, prima di essere compilati, vengono elaborati da un preprocessore. Il preprocessore legge i caratteri che compongono il programma, effettua delle modifiche e quindi dà il risultato in pasto al compilatore.

Ci sono due usi fondamentali del preprocessore: includere nel testo altri file, e alterare la struttura lessicale del programma.

Il preprocessore è controllato da direttive, che devono stare interamente su una riga, e sono introdotte da un carattere # che deve essere necessariamente il primo carattere della riga.

preproc-dir ::= (dir-include | dir-define)

dir-include ::= **#include** (”nome-file” | <nome-file>)

dir-define ::= **#define** ident [(ident{, ident})] *qualsiasi cosa*

Commento alle produzioni del pre-processore (1)

preproc-dir ::= (dir-include | dir-define)

Una direttiva del preprocessore è una direttiva di inclusione, o una di definizione.

dir-include ::= **#include** (”nome-file” | <nome-file>)

Una direttiva di inclusione causa la sostituzione della direttiva con il contenuto del file indicato all’interno del file corrente. Se si utilizzano le virgolette, il file viene cercato anche nella directory corrente, altrimenti solo nelle directory standard.

dir-define ::= **#define** ident [(ident{, ident})] *qualsiasi cosa*

Una direttiva di definizione causa la sostituzione dell’identificatore specificato con il testo che lo segue in tutto il programma (in questo senso le definizioni alterano la struttura lessicale).

Se sono presenti le parentesi tonde dopo l’identificatore, viene effettuata una sostituzione più sofisticata, con effetto simile a una chiamata di funzione.