1 Richiami di Sintassi

Un programma viene in genere descritto mendiante una sequenza finita di caratteri (stringa). Nella fase di analisi lessicale (presente sia nei compilatori che negli interpreti) si effettuano alcuni controlli di tipo puramente sintattico e si produce una descrizione piu' astratta del programma, che va sotto il nome di albero astratto.

1.1 Linguaggi e grammatiche

Fissato un insieme (finito) T di simboli terminali, un **linguaggio** L e' un sottoinsieme di T^*, dove T^* e' l'insieme delle sequenze finite di elementi di T (stringhe su T). Una **grammatica libera da contesto** (Context Free Grammar) e' una quadrupla $G = (T, N, P, S)$ t.c.

- T e' un insieme (finito)
- N e' un insieme (finito) disgiunto da T
- P e' un sottoinsieme (finito) di $N \times (N + T)^*$
- S e' un elemento di N

Gli elementi di T sono detti simboli terminali e vengono indicati con lettere minuscole $a, b, \ldots$. Gli elementi di N sono detti simboli non-terminali e vengono indicati con lettere mauiscole $A, B, \ldots$. Gli elementi di P sono detti produzioni e vengono rappresentati da $A ::= \alpha$ con $A \in N$ e $\alpha \in (N + T)^*$. S e' detto non-terminale iniziale.

1.2 Linguaggio associato ad una CFG

Una CFG $G = (T, N, P, S)$ identifica un insieme di regole di inferenza per derivare asserzioni della forma $\alpha \in A$ dove $A \in N$ e $\alpha \in T^*$. Piu' precisamente, ad ogni produzione $A ::= \alpha$ (di *nome* p) associamo la regola di inferenza

$$(p)\ \frac{\beta_i \in A_i \quad 1 \le i \le n}{\alpha_0 \beta_1 \alpha_1 \ldots \beta_n \alpha_n \in A}$$

dove $\alpha = \alpha_0 A_1 \alpha_1 \ldots A_n \alpha_n$ con $n \ge 0$, $\alpha_i \in T^*$ e $A_i \in N$ (tale decomposizione di α in stringhe di terminali alternate a simboli non-terminali e' unica).

Date queste regole possiamo associare ad ogni non-terminale $A \in N$ un linguaggio $[\![A]\!]_G \subseteq T^*$ cosi' definito: $\alpha \in [\![A]\!]_G$ sse esiste una derivazione di $\alpha \in A$ mediante le regole di inferenza associate alle produzioni di G.

Il **linguaggio** associato alla CFG G e' $L_G \stackrel{\Delta}{=} [\![S]\!]_G \subseteq T^*$. Diciamo che una CFG e' **ambigua** se esiste una stringa $\alpha \in L_G$ che ammette due alberi di derivazioni distinti, ovvero due modi diversi di inferire $\alpha \in S$.

1.3 Formalismi per descrivere CFG

Il formalismo piu' noto per descrivere CFG e' la Backus Naur Form (BNF). Altri formalismi sono vari forme di BNF estesa, ed i diagrammi sintattici.

Nella BNF i simboli non-terminali (detti anche categorie sintattiche) sono indicati con `<Nome>`, dove `Nome` dovrebbe indicare in modo informale il tipo di entita' (p.e. Espressioni, Variabili, Comandi, Identificatori) descritte con le stringhe del linguaggio corrispondente a quel non-terminale. Il formalismo BNF consente anche di raggruppare le produzioni corrispondenti ad una certa categoria sintattica scrivendo $A ::= \alpha_1 \mid \ldots \mid \alpha_n$, invece delle n produzioni $A ::= \alpha_1, \ldots, A ::= \alpha_n$.

Nelle estensioni del formalismo BNF si tendono a fornire ulteriori construtti idicare ripetizione oppure opzione (in genere si usano le notazioni $\{\alpha\}$ e $[\alpha]$). Questi costrutti possono essere eliminati introducendo nuove categorie sintattiche e produzioni.

1.4 Sintassi astratta associata ad una CFG

Ad una CFG $G = (T, N, P, S)$ possiamo associare una segnatura many-sorted Σ_G che astrae dai dettagli sintattici concreti, ovvero dai simboli terminali:

- $Sort(\Sigma) = N$, cioe' le sort sono esattamente i simboli non-terminali
- per ogni produzione $A ::= \alpha_0 A_1 \alpha_1 \ldots A_n \alpha_n$ (di *nome* p) abbiamo un simbolo di operazione $p: A_1 \ldots A_n \to A$.

I termini chiusi $t \in T_A^{\Sigma_G}$ sono in corrispondenza biunivoca con gli alberi di derivazione di asserzioni della forma $\alpha \in A$.
A partire dalla CFG G possiamo definire una particolare Σ_G-algebra:

- la sort A e' interpretata dall'insieme $[\![A]\!]_G \subseteq T^*$
- l'operazione $p: A_1 \ldots A_n \to A$, corrispondente a $A ::= \alpha_0 A_1 \alpha_1 \ldots A_n \alpha_n$, e' interpretata dalla funzione $[\![p]\!](\beta_1, \ldots, \beta_n) = \alpha_0 \beta_1 \alpha_1 \ldots \beta_n \alpha_n$

In tale algebra l'interpretazione $[\![t]\!]$ di un termine chiuso $t \in T_A^{\Sigma_G}$ e' esattamente la stringa $\alpha \in T^*$ t.c. $\alpha \in A$ e' la conclusione dell'albero di derivazione corrispondente al termine t.

1.5 Esempi di CFG

Consideriamo varie CFG per descrivere semplici espressioni, che hanno come non-terminale iniziale e' E:

$$\begin{array}{ll}
G_0 & E ::= 0 \mid 1 \mid E + E \mid E * E \\
G_1 & E ::= 0 \mid 1 \mid (E + E) \mid (E * E) \\
G_2 & E ::= 0 \mid 1 \mid (E) \mid E + E \mid E * E \\
G_3 & E ::= F \mid F + E \\
 & F ::= A \mid A * E \\
 & A ::= 0 \mid 1 \mid (E)
\end{array}$$

A livello di sintassi astratta G_0 e G_1 sono uguali, ma G_0 e' ambigua (vi sono due derivazioni di $0 + 0 + 1$), mentre G_1 genera espressioni piene di parentesi. G_2 ha gli stessi problemi di ambiguita' di G_0, ma offre la possibilita' di risolvere l'ambiguita' facendo uso delle parentesi. A livello di sintassi astratta G_2 introduce una operazione unaria, di scarso significato semantico. G_3 e' una grammatica non ambigua che descrive lo stesso linguaggio di G_2 (non e' immediato dimostrare che $L_2 \subseteq L_3$, si consiglia di procedere per induzione sulla lunghezza di una stringa di terminali), pero' per evitare l'ambiguita' e limitare l'uso delle parentesi, si complica notevolmente la sintassi astratta.

1.6 Alberi sintattici ed alberi astratti

Fissata una CFG G, diciamo che un albero ordinato con etichette sui nodi e' un **albero sintattico** per G sse:

- le etichette sono elementi di $N + T$, cioe' o terminali o non-terminali
- la radice e' etichettata con S
- se un nodo e' etichettato da un terminale allora non ha figli
- se un nodo e' etichettatto da un non-terminale A e α e' la sequenza delle etichette dei suoi figli, allora $A ::= \alpha$ e' una produzione

Le foglie di un albero sintattico sono o terminali o non-terminali A t.c. $A ::=$ e' una produzione. Un albero sintattico identifica una stringa in T^* data dalle etichette terminali lette da sinistra a destra.
Sia Σ_G la segnatura (sintassi astratta) associata a G, diciamo che un albero ordinato con etichette sui nodi e' un **albero astratto** per G sse:

- le etichette sono operazioni di Σ_G
- la radice e' etichettata da una operazione la cui sort di output e' S
- se un nodo e' etichettato da una operazione $p: A_1, \ldots, A_n \to A$, allora ha esattamente n figli e l'etichetta dell'i-esimo figlio deve avere sort di output A_i.

Un albero astratto non e' altro che la rappresentazione di un termine chiuso $t \in T_S^{\Sigma_G}$.

2 Richiami di semantica

In genere si utilizzano due tipi di semantica per descrivere i linguaggi di programmazione: quella denotazionale e quella operazionale.

La sematica dei linguaggi di programmazione puo' essere usata in vari modi: come specifica rivolta al programmatore di cosa fa' un programma (in questo si pone come alternativa a descrizioni piu' informali quali i manuali), come strumento per validare regole di tipo logiche (p.e. logica di Hoare) usate nelle dimostrazioni di correttezza, come strumento per dimostrare la correttezza di una implementazione.

Ad un certo linguaggio e' possibile dare semantiche a diverso livello di dettagli e descritte in modi molto diversi. Tuttavia e' ragionevole aspettarsi che ci sia concordanza nel predire il *comportamento osservabile*. Per esempio nel caso di un programma sequenziale quello che ci interessa osservare e' il comportamento input/output (piu' in generale l'interazione con il mondo esterno, che ha senso anche per programmi che non terminano) piuttosto che lo stato della memoria interna. Un tipo particolarmente semplice di osservazione (che spesso e' sufficiente per definire una equivalenza osservazione non banale per *parti di programma*) e' osservare la terminazione.

2.1 Semantica Denotazionale

La semantica denotazionale definisce una interpretazione (a la Tarski) dei programmi in un dominio semantico (p.e. un insieme o una struttura matematica piu' complessa). La semantica denotazionale dei linguaggi di programmazione si sviluppo a partire dagli anni '70, utilizzando le strutture matematiche proposte da Dana Scott per modellare il λ-calcolo non tipato. Tale interpretazione e' data per induzione sulla struttura astratta dei programmi (alberi astratti), e spesso si definisce l'interpretazione solo per programmi *staticamente* corretti.

La semantica denotazionale si puo' pensare anche come una traduzione (definita per induzione sulla struttura dei programmi) del linguaggio di programmazione in un linguaggio matematico. Da questo punto di vista una semantica denotazionale e' come un copilatore che ha come linguaggio target un formalismo matematico invece di un linguaggio di programmazione (a piu' basso livello).

2.2 Semantica Operazionale

La semantica operazionale e' essenzialmente una descrizione di un interprete per il linguaggio su una macchina astratta (in genere non realizzabile fisicamente). Un modo elegante di dare la semantica operazionale e' tramite regole di inferenza. Il metodo va sotto il nome di Structured Operational Semantics (SOS) e fu proposto da Plotkin verso il 1980.

Al contrario della semantica denotazionale, quella operazionale non e' necessariamente definita per induzione sulla struttura astratta dei programmi, ma e' comunque conveniente manipolare i programmi in forma astratta.

2.3 Equivalenza denotazionale e osservazionale

In semantica denotazionale vi e' una nozione ovvia di **equivalenza denotazionale** indotta dall'interpretazione: due comandi/espressioni sono denotazionalmente equivalenti se hanno la stessa interpretazioni. In effetti tale equivalenza e' una congruenza, cioe' dato un qualsiasi comando $c[\_]$ *con un buco per un comando* e due comandi denotazionalmente equivalenti $c_1 =_d c_2$, allora $c[c_1] =_d c[c_2]$.

Un'altra nozione di equivalenza definita in termini delle osservazioni e' la cosidetta **equivalenza osservazionale**, diciamo che $c_1 =_o c_2$ sse per ogni programma $p[\_]$ con un buco per un comando e test osservabile t si ha che $p[c_1]$ passa il test t sse lo passa $p[c_2]$. Anche l'equivalenza osservazionale e' una congurenza. Si osservi che la definizione di $=_o$ e' sensibile non solo alle osservazioni che e' possibile fare su un programma, ma anche all'insieme dei programmi. Quindi estendendo il set dei programmi puo' succedere che due comandi che prima dell'estensione risultavano osservazionamente equivalenti diventino distinguibili.

3 Linguaggi imperativi

3.1 Un linguaggio RAM con goto

In questa sezione descriviamo un linguaggio di programmazione per una architettura RAM costituita da una memoria con infinite celle di tipo intero indirizzabili mediante interi, e nastri di input e output contenenti una sequenza finita di interi (tenderemo ad ignorare gli aspetti legati all'input/output). Al contrario dell'architettura von Neumann assumeremo che il programma si trovi in una memoria a parte, leggibile ma non modificabile.

$$
\begin{array}{rcl}
i \in < Natural > & ::= & \ldots \\
n \in < Integer > & ::= & \ldots \\
v \in < Variable > & ::= & n \mid \uparrow n \\
e \in < Expression > & ::= & n \mid op(v_1, v_2) \\
c \in < Command > & ::= & v := e \mid read(v) \mid write(v) \mid \\
 & & \text{if } v > 0 \text{ goto } i \\
p \in < Program > & ::= & \mid c; p
\end{array}
$$

La semantica operazionale a piccoli passi verra' descritta in termini di transizioni di stato $s \in S = Z^* \times (Z \to Z) \times Z^*$ e program counter $i \in N$. Per semplicita' ci limitiamo a modellare la sola memoria, cioe' prenderemo $S = Z \to Z$.

- $s \vdash v \Rightarrow m$

$$(m)\ \frac{}{s \vdash m \Rightarrow m} \qquad (\uparrow m)\ \frac{}{s \vdash \uparrow m \Rightarrow s[m]}$$

- $s \vdash e \Rightarrow n$

$$(n)\ \frac{}{s \vdash n \Rightarrow n} \qquad (op)\ \frac{s \vdash v_i m_i \quad i = 1, 2}{s \vdash op(v_1, v_2) \Rightarrow n}\ n = op(s[m_1], s[m_2])$$

- $p \vdash s, i \to s', i'$ dove $p = c_1; \ldots c_n;$

$$(:=)\ \frac{s \vdash v \Rightarrow m \quad s \vdash e \Rightarrow n}{p \vdash s, i \to s[m \mapsto n], i+1}\ \text{se } c_i \equiv (v := e)$$

$$(\text{goto.true})\ \frac{s \vdash v \Rightarrow m}{p \vdash s, i \to s, i+1}\ \text{se } c_i \equiv \text{if } v > 0 \text{ goto } i' \text{ e } s[m] > 0$$

$$(\text{goto.false})\ \frac{s \vdash v \Rightarrow m}{p \vdash s, i \to s, i'}\ \text{se } c_i \equiv \text{if } v > 0 \text{ goto } i' \text{ e } s[m] \leq 0$$

Una funzione $s: Z \to Z$ non e' rappresentabile in modo finitario, un per avere asserzioni rappresentabili in modo *finitario* e' considerare funzioni *quasi ovunque* nulle. Una semantica dovrebbe anzitutto specificare il comportamento input/output di un programma, che e' l'unico effettivamente osservabile. Quindi e' ragionevole associare ad un programma p una funzione parziale $[\![p]\!]: Z^* \rightharpoonup Z^*$ che ignora lo stato della memoria. Poiche' nella semantica operazionale abbiamo ingnorato l'input/output possiamo prendere come osservazione la terminazione: $p \downarrow$ sse eseguendo p a partire dalla prima istruzione con stato iniziale s_0 si termina. Piu' formalmente $p \downarrow$ sse $\exists s: S. p \vdash s_0, 1 \Rightarrow s$, dove s_0 e' uno stato prefissato (p.e. quello in cui tutte le celle di memoria contengono 0) e $p \vdash s, i \Rightarrow s'$ e' la semantica operazionale a grandi passi definita dalle regole

$$(\text{step})\ \frac{\begin{array}{c} p \vdash s, i \to s', i' \\ p \vdash s', i' \Rightarrow s'' \end{array}}{p \vdash s, i \Rightarrow s''} \qquad (\text{halt})\ \frac{p \vdash s, i \not\to}{p \vdash s, i \Rightarrow s}$$

3.2 Un linguaggio RAM strutturato

$$
\begin{array}{rcl}
n \in < Integer > & ::= & \ldots \\
v \in < Variable > & ::= & n \mid \uparrow v \mid v[e] \\
e \in < Expression > & ::= & n \mid v \mid op(e_1, e_2) \\
c \in < Command > & ::= & v := e \mid read(v) \mid write(v) \mid \\
 & & \text{if } e > 0 \text{ then } sc_1 \text{ else } sc_2 \text{ end} \mid \\
 & & \text{while } e > 0 \text{ do } sc \text{ end} \mid \\
 & & \text{case } e \text{ of } sa \text{ end} \mid \\
 & & \text{loop } sc \text{ end} \mid \text{exit} \mid \text{return} \\
sc \in < Command - Sequence > & ::= & \mid c; sc \\
sa \in < Alternative - Sequence > & ::= & \text{default } sc \mid n: sc | sa \\
p \in < Program > & ::= & sc
\end{array}
$$

- $s \vdash v \Rightarrow m$

$$(m)\ \frac{}{s \vdash m \Rightarrow m} \qquad (\uparrow)\ \frac{s \vdash \uparrow v \Rightarrow m}{s \vdash \uparrow v \Rightarrow s[m]} \qquad ([\,])\ \frac{s \vdash v \Rightarrow m \quad s \vdash e \Rightarrow n}{s \vdash v[e] \Rightarrow m+n}$$

- $s \vdash e \Rightarrow n$

$$(n)\ \frac{}{s \vdash n \Rightarrow n} \qquad (v)\ \frac{s \vdash_V v \Rightarrow m}{s \vdash_E v \Rightarrow s[m]}$$

$$(op)\ \frac{s \vdash e_i n_i \quad i = 1,2}{s \vdash op(e_1, e_2) \Rightarrow n}\ n = op(n_1, n_2)$$

- $c \vdash s \Rightarrow s', r$ dove $r ::= ok \mid return \mid exit$ indica il tipo di terminazione

$$(:=)\ \frac{s \vdash v \Rightarrow m \quad s \vdash e \Rightarrow n}{x := v \vdash s \Rightarrow s[m \mapsto n], ok}$$

$$(\text{if.true})\ \frac{s \vdash e \Rightarrow n \quad sc_1 \vdash s \Rightarrow s', r}{\text{if } e > 0 \text{ then } sc_1 \text{ else } sc_2 \text{ end} \vdash s \Rightarrow s', r}\ n > 0$$

$$(\text{if.false})\ \frac{s \vdash e \Rightarrow n \quad sc_2 \vdash s \Rightarrow s', r}{\text{if } e > 0 \text{ then } sc_1 \text{ else } sc_2 \text{ end} \vdash s \Rightarrow s', r}\ n \leq 0$$

$$(\text{while.true})\ \frac{\begin{array}{c} s \vdash e \Rightarrow n \quad sc \vdash s \Rightarrow s', ok \\ \text{while } e > 0 \text{ do } sc \text{ end} \vdash s' \Rightarrow s'', r \end{array}}{\text{while } e > 0 \text{ do } sc \text{ end} \vdash s \Rightarrow s'', r}\ n > 0$$

$$(\text{while.true.fail})\ \frac{s \vdash e \Rightarrow n \quad sc \vdash s \Rightarrow s', r}{\text{while } e > 0 \text{ do } sc \text{ end} \vdash s \Rightarrow s', r}\ r \neq ok$$

$$(\text{while.false})\ \frac{s \vdash e \Rightarrow n}{\text{while } e > 0 \text{ do } sc \text{ end} \vdash s \Rightarrow s, ok}\ n \leq 0$$

$$(\text{case})\ \frac{s \vdash e \Rightarrow n \quad sa \vdash n, s \Rightarrow s', r}{\text{case } e \text{ of } sa \text{ end} \vdash s \Rightarrow s', r}$$

$$(\text{loop.ok})\ \frac{\begin{array}{l} sc \vdash s \Rightarrow s', ok \\ \text{loop } sc \text{ end} \vdash s' \Rightarrow s'', r \end{array}}{\text{loop } sc \text{ end} \vdash s \Rightarrow s'', r} \qquad (\text{loop.exit})\ \frac{sc \vdash s \Rightarrow s', exit}{\text{loop } sc \text{ end} \vdash s \Rightarrow s', ok}$$

$$(\text{loop.fail})\ \frac{sc \vdash s \Rightarrow s', r}{\text{loop } sc \text{ end} \vdash s \Rightarrow s', r}\ r = return$$

- $sc \vdash s \Rightarrow s', r$

$$(\,)\ \frac{}{\vdash s \Rightarrow s, ok}$$

$$(;)\ \frac{c \vdash s \Rightarrow s', ok \quad sc \vdash s' \Rightarrow s'', r}{c; sc \vdash s \Rightarrow s'', r} \qquad (;\text{fail})\ \frac{c \vdash s \Rightarrow s', r}{c; sc \vdash s \Rightarrow s', r}$$

- $sa \vdash n, s \Rightarrow s', r$

$$(\text{default})\ \frac{sc \vdash s \Rightarrow s', r}{\text{default } sc \vdash n, s \Rightarrow s', r} \qquad (\text{match})\ \frac{sc \vdash s \Rightarrow s', r}{n{:}\ sc|as \vdash n, s \Rightarrow s', r}$$

$$(\text{mismatch})\ \frac{as \vdash n, s \Rightarrow s', r}{m{:}\ sc|as \vdash n, s \Rightarrow s', r}\ m \neq n$$

Sui programmi RAM strutturati possiamo definire l'osservazione $p\downarrow$ come $\exists s, r.p \vdash s_0, 1 \Rightarrow s, r$, dove non discriminiamo tra i tre tipi di terminazione.

Sui programmi del linguaggio RAM strutturato e' sensato effettuare dei controlli statici per segnalare usi sospetti di certi costrutti che potrebbe essere sintomo di un errore logico. Per esempio, si potrebbe avvertire il programmatore che ha usato un "exit" fuori dal corpo di un loop (la semantica ci dice che cio' porta alla terminazione del programma), o che ha etichettatto due alternative di un "case" con la stessa constante (la semantica ci dice che solo la prima e' presa in considerazione).

Tali controlli non possono essere effettuati in fave di analisi lessicale (cioe' mediante usando una CFG), ma si possono descrivere con regole di inferenza *dirette dalla sintassi*, che permettono di decidere se un asserto e' derivabile o no. Questa proprieta' di decidibilita' contrasta con la impossibilita' di decidere se un asserto della forma $p \vdash s \Rightarrow s', r$ e' derivabile, ed e' la proprieta' cruciale che distingue la semantica statica da quella dinamica.

- $\vdash c \Rightarrow r$ dove $r \subseteq Error = \{exit, case\}$ indica il tipo di errori presenti in c

$$(:=)\ \frac{}{\vdash x := v \Rightarrow}$$

$$(\text{if})\ \frac{\vdash sc_i \Rightarrow r_i \quad i = 1, 2}{\vdash \text{if } e > 0 \text{ then } sc_1 \text{ else } sc_2 \text{ end} \Rightarrow r_1 \cup r_2}$$

$$(\text{while})\ \frac{\vdash sc \Rightarrow r}{\vdash \text{while } e > 0 \text{ do } sc \text{ end} \Rightarrow r}$$

$$(\text{case})\ \frac{\vdash sa \Rightarrow r}{\vdash \text{case } e \text{ of } sa \text{ end} \Rightarrow r \cap Error}$$

$$(\text{loop})\ \frac{\vdash sc \Rightarrow r}{\vdash \text{loop } sc \text{ end} \Rightarrow r \cap \{case\}}$$

$$(\ )\ \frac{}{\vdash \ \Rightarrow \emptyset}$$

$$(;)\ \frac{\vdash c \Rightarrow r \quad \vdash sc \Rightarrow r'}{\vdash c; sc \Rightarrow r \cup r'}$$

- $\vdash sa \Rightarrow r$ dove $r \subseteq Error \cup Z$

$$(\text{default})\ \frac{\vdash sc \Rightarrow r}{\vdash \text{default } sc \Rightarrow r}$$

$$(\text{case.ok})\ \frac{\vdash sc \Rightarrow r \quad \vdash sa \Rightarrow r'}{\vdash n{:}\, sc | as \Rightarrow r \cup r' \cup \{n\}}\ n \notin r'$$

$$(\text{case.fail})\ \frac{\vdash sc \Rightarrow r \quad \vdash sa \Rightarrow r'}{\vdash n{:}\, sc | as \Rightarrow r \cup r' \cup \{case\}}\ n \in r'$$

3.3 Esercizi

1. aggiungere al linguaggio RAM strutturato il comando "continue" del C, che permette di interrompere il ciclo corrente e passare al prossimo ciclo. Estendere la semantica operazionale statica e dinamica.
2. modificare la semantica dei costrutti "case" ed "exit", cercando di mimare il comportamento dei costrutti "switch" e "break" del C.