Compilatori Core Concepts

Introduzione

Un *linguaggio formale* è una qualsiasi collezione di stringhe su un qualche alfabeto (composto da simboli qualsiasi, in senso astratto).
Le *grammatiche* sono un modo compatto per definire come un linguaggio è costruito su un certo alfabeto.
Una *stringa* corrisponde ad una possibile produzione di una grammatica.

Formalmente una grammatica è una quadrupla (N, T, → , S) in cui:
- **N** è un insieme finito di simboli non terminali, ovvero simboli intermedi usati per fornire la struttura delle frasi della grammatica.
- **T** è un insieme finito di simboli terminali, ovvero l'alfabeto/vocabolario su cui sono costruite le produzioni della grammatica.
- → è un insieme finito di produzioni, non è una funzione bensì una relazione; ad esempio $A \rightarrow a_1 \dots a_n$ con $A \in N$ ed $a_1 \dots a_n \in N \cup T$ (a sinistra vi sono sempre simboli non terminali, a destra possono essere terminali o non terminali).
- **S** ∈ N, chiamato simbolo iniziale, ovvero dove comincia la produzione.

Derivazioni

Una derivazione (=>) è la riscrittura di una stringa composta da terminali e non terminali in un'altra stringa sempre composta da terminali e/o non terminali usando una o più regole di produzione (→).

Data la grammatica:

BExp → (BExp) BExp → Digit Digit → 0 | 1 | . . . | 9

((BExp)) è il frutto di 2 derivazioni ma non è una stringa del linguaggio perché contiene un non terminale, se vi applichiamo la seconda regola della grammatica vista prima otterremmo la derivazione: ((BExp)) => ((Digit)).

Il *linguaggio generato* è l'insieme delle sequenze terminali che possono essere derivate con più passi di derivazione a partire dal simbolo iniziale. Il linguaggio generato dalla grammatica precedente è definito come: $L(G) = \{ (^n\ d\ )^n \mid n \geq 0, d \in 0 \dots 9 \}$

Con =>* indichiamo che la derivazione avviene in 0 o più passaggi (non ha importanza il numero preciso).
Con $=>^+$ indichiamo che la derivazione avviene in 1 o più passaggi.
Con T* indichiamo la *chiusura di Kleene*, ovvero se T è un insieme di simboli, si indicano tutte le stringhe finite costruite usando 0 o più di quei simboli.

Derivazione leftmost

La derivazione leftmost prevede che ogni volta si vada a riscrivere il non terminale

più a sinistra e si applichi una delle regole che ha quel non terminale come simbolo a sinistra (speculare è la derivazione rightmost).

Esempio date le seguenti regole di derivazione:
Exp → Exp - Exp
Exp → Digit
Digit → 0 | 1 | ... | 9

Una derivazione leftmost è (in blu le applicazioni delle regole):
Exp => Exp - Exp => Exp - Exp - Exp => Digit - Exp - Exp => 3 - Exp - Exp =>
3 - Digit - Exp => 3 - 2 - Exp => 3 - 2 - Digit => 3 - 2 - 1

Un'altra uguale è:
Exp => Exp - Exp => Digit - Exp => 3 - Exp => 3 - Exp - Exp => 3 - Digit - Exp =>
3 - 2 - Exp => 3 - 2 - Digit => 3 - 2 - 1

Data una grammatica è possibile derivare in maniera leftmost la stessa stringa in modi diversi (ovvero in più alberi distinti). Dal punto di vista del linguaggio generato questo è irrilevante, perché la stringa è la stessa, ma la struttura sintattica è diversa, e se la semantica dipende dalla struttura sintattica questo risulta un problema (vi sono quindi frasi/stringhe uguali che vogliono dire cose diverse in base a come sono state derivate).
Per ovviare a questo problema si aggiungono regole di associatività e precedenza, oppure "zucchero sintattico" (simboli che non hanno significato ma servono a disambiguare la sintassi, come ad esempio le parentesi).

Una grammatica che ha più alberi di derivazione per la stessa stringa è chiamata *ambigua*. In questi casi è possibile risolvere il problema ponendo regole di associatività e precedenza.

L'obiettivo dell'analisi lessicale è appunto *riconoscere le strutture lessicali*, l'idea dietro a questo passaggio consiste nel suddividere grammatiche molto grandi in pezzi logici. Un modo per farlo è dividerle in: grammatica *lexer* (lessico) e grammatica *parser* (sintassi).

Analisi Lessicale

Lexer grammar: definisce e permette di riconoscere le *parole* (sotto forma di *token*) a partire da uno stream di caratteri.
Parser grammar: definisce e permette di riconoscere l'albero sintattico sotteso dall'ordine delle parole (token) del programma.

L'analisi lessicale divide i programmi testuali in *tokens* o *words*. Ad esempio:
`if (x == y) z = 1; else z = 2;` viene tokenizzato in
`if, (, x, ==, y, ), z, =, 1, ;, else, z, =, 2, ;`

I lessemi sono analizzati, riconosciuti e convertiti in token:
`IF, LPAR, ID("x"), EQUALS, ID("y"), RPAR, ID("z"), etc...`

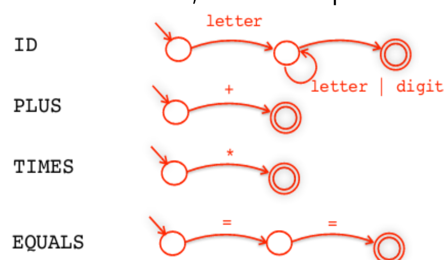
La costruzione di un lexer prevede che l'input sia composto da una sequenza di caratteri, l'obiettivo è trovare i lessemi (lexemes) e mapparli in tokens, questo si ottiene partizionando la stringa di input in sottostringhe (lexemes) e classificarli in base al loro ruolo (token). I lexemes che sono formati da soli \n, \t e \r sono solitamente cancellati senza produrre alcun token.

Regola del matching massimo
Questa regola prevede semplicemente che il flusso di input di caratteri è partizionato in lessemi che sono il *più lunghi possibile*. È possibile definire il comportamento del lexer tramite un Automa a Stati Finiti (FA):

Automi a stati finiti
Gli automi a stati finiti (FA) sono grafi i cui nodi sono chiamati *stati* e i cui archi sono chiamati *transizioni*. Le transizioni hanno etichette e permettono di transitare da uno stato all'altro in base al carattere letto. Gli elementi principali sono gli stati, uno dei quali è chiamato stato *iniziale* ed altri sono chiamati stati *finali* (accepting). La semantica di un automa a stati finiti è semplice:
- si inizia nell'unico stato iniziale
- si inizia a leggere la stringa di input un carattere alla volta
- quando la lettura termina se lo stato in cui ci si trova è finale allora la stringa viene *accettata*, se non è finale allora è rifiutata.
- se non è possibile nessuna transizione, allora la stringa è chiaramente rifiutata.

Progettazione di un lexer
La progettazione di un lexer prevede due parti:
1. *Descrizione*: ovvero *cosa* fa il lexer. Si descrive ciascun token in modo preciso, con un modello formale come l'automa a stati finiti.
2. *Implementazione*: ovvero *come* si comporta il lexer. Si costruisce l'automazione corrispondente al lexer, gli elementi usati sono comuni ad ognuno (come libreria).

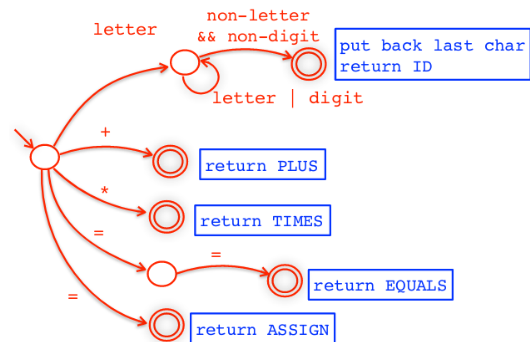
- *Descrizione del lexer*
Si definisce un FA per ogni lessema del linguaggio e si associa la FA al token riconosciuto, ad esempio:



- *Implementazione del lexer*
Nell'esempio precedente, il token ID ha una transizione senza etichetta, quella

che punta allo stato finale, questo rende l'*automazione non deterministica (NFA)*: nello stato q_1 infatti non è chiaro cosa succeda quando arriva una lettera o una cifra, se si aggiungesse l'etichetta "*non-letter && non-digit*" diventerebbe una DFA (automa a stati finiti deterministico).
Quando viene riconosciuto un token il DFA deve eseguire due azioni:
1. return TOKEN: il chiamante del lexer (ovvero il parser) ottiene indietro il token riconosciuto e il lexer ricomincia dallo stato iniziale cercando il token successivo.
2. azioni di gestione dei lookaheads, ad esempio undoNextChar(c) per i token che corrispondono ai lessemi di lunghezza variabile (es. ID).

Un problema che sorge però è che il lexer deve avere un unico entry point, è necessaria quindi la combinazione dei DFA che corrispondono ai singoli tokens. Questa operazione tuttavia potrebbe creare non determinismo.



L'algoritmo del lexer può essere definito con i passaggi seguenti:
1. definire un FA per ogni lessema
2. combinare gli FA identificando gli stati iniziali
3. se gli FA risultanti dal punto 2 sono non deterministici allora trasforma l'automa in deterministico (applicando un algoritmo specifico).
4. utilizzare le seguenti regole:
- quando uno stato finale è raggiunto: si memorizza la posizione presa in input (in modo che sia possibile leggere altri caratteri) e si continua a leggere gli altri caratteri da stato a stato.
- se non sono possibili altre transizioni con il carattere successivo: si effettua il rollback all'ultimo stato finale (effettuando undo delle letture corrispondenti) e si ritorna il token corrispondente all'ultimo stato finale.

Per risolvere problemi di **ambiguità** (come "if" se è token ID o keyword IF) si definiscono delle *priorità* tra i tokens. Oltre che a rimuovere tutti gli spazi vuoti.
Ad esempio dato il DFA seguente:

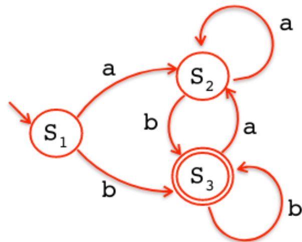
La stringa di input "*then_-*" restituirà un errore sul simbolo "-".

L'automa permette di definire i lessemi che corrispondono a token, e per avere una descrizione più compatta e formale dell'automa si utilizzano le espressioni regolari, utilizzate come input per i generatori lexer.

Gli operandi nelle regex corrispondono alle etichette delle transizioni del FA, sono singoli caratteri tra apici o sequenze di caratteri tra apici.
L'operatore "*****" indica zero o più ripetizioni.
L'operatore "**+**" indica una o più ripetizioni ed equivale ad: c(c)*

L'operatore "**?**" indica zero o una ripetizione, es. int: ('+' | '-')?
L'operatore "**~**" (not) indica i caratteri diversi da quelli specificati, es. ~('a'...'z')
L'operatore "**.**" indica un singolo carattere, es. .* è ogni sequenza di caratteri.

Una DFA è solitamente rappresentata tramite una matrice a 2 dimensioni come ogni grafo, di seguito un esempio.



	a	b
S_1	2	3
S_2	2	3
S_3	2	3

La chiusura di Kleene indica combinazioni, anche nulle, degli elementi nella sequenza. Questa chiusura ha la precedenza su tutti gli altri operatori (es. a | b* la chiusura è su b, se no era necessario mettere le parentesi). Tutte le espressioni che non contengono la chiusura di Kleene sono insiemi finiti, se hanno la chiusura sono invece infiniti.

ANTLR lexer
Il lexer ANTLR, come ogni lexer, legge i caratteri fino a quando una regola non è selezionata, viene stampato il corrispondente token e si riparte dal carattere successivo. La regola usata è il primo match più lungo, a differenza degli altri lexers il matching è greedy, il token selezionato utilizza il massimo numero di caratteri in input. Da notare che il lexer non effettua backtracking, non cambia le decisioni precedenti. In ANTLR è il parser che man mano invoca il lexer, token per token.

Regola della prima corrispondenza più lunga
Se vi sono varie regole che corrispondono all'input, quella selezionata è la corrispondente alla stringa più lunga.

Greedy token matching
ANTLR è greedy, il token selezionato è quello che consuma il numero maggiore di caratteri.

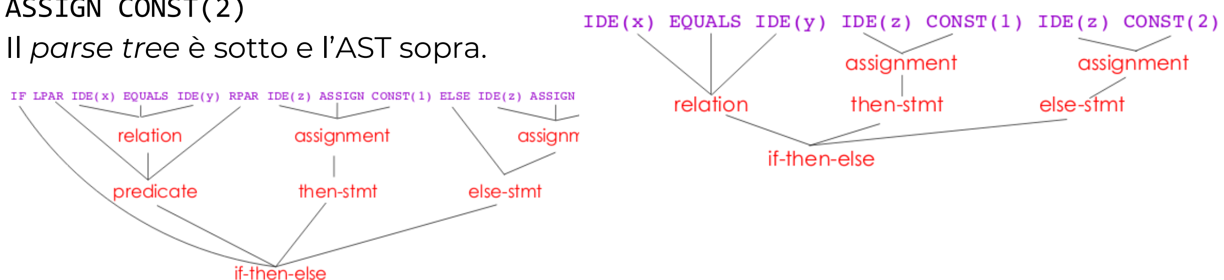
NOTA: i nomi che iniziano con la lettera minuscola sono non terminali del parser, i nomi che iniziano con la lettera maiuscola sono non terminali del lexer, a destra del lexer si possono avere solo espressioni regolari.

Analisi Sintattica

Quando si scrive una grammatica per un linguaggio di programmazione, si inizia normalmente dividendo i costrutti del linguaggio in categorie sintatticamente differenti. Una categoria sintattica è un sotto-linguaggio che incorpora un

particolare concetto. Esempi di categorie sintattiche comuni nei linguaggi di programmazione sono:
- *Espressioni*: sono utilizzate per esprimere il calcolo di valori.
- *Statement*: esprimono azioni che si verificano in una particolare sequenza.
- Dichiarazioni: esprimono proprietà dei nomi utilizzati in altre parti del programma.

Con *parsing* intendiamo prendere in input sequenze di token e ritornare un *albero di sintassi astratto* (AST). Parse tree e abstract syntax tree sono leggermente diversi: ad esempio `if (x == y) z = 1; else z = 2;` diventa `IF LPAR IDE(x) EQUALS IDE(y) RPAR IDE(z) ASSIGN CONST(1) ELSE IDE(z) ASSIGN CONST(2)`
Il *parse tree* è sotto e l'AST sopra.



Differenza tra parse tree e AST:
- Il *Parse Tree*: ha tutti i tokens, compresi quelli che il parser utilizza per identificare l'annidamento di sotto-espressioni (come le parentesi) e la punteggiatura; tecnicamente mostra tutta la sintassi concreta. Il difetto di questi alberi è che sono troppo verbosi, quindi spesso non vengono costruiti esplicitamente ma utilizzati durante la computazione dei parser.
- L'*Abstract Syntax Tree* (AST): rimuove i risultati parziali, cancellando token inutili, appiattendo l'albero rimuovendone i nodi, ecc.; tecnicamente l'AST mostra una versione astratta della sintassi.

Grammatiche context-free
Le espressioni regolari non catturano situazioni in cui ad esempio è necessario avere specularità nelle stringhe prodotte [ad esempio: $a^n b^n$ o $(^n a)^n$]. Questo può essere raggiunto tramite **grammatiche context-free**, ovvero grammatiche in cui a sinistra si hanno sempre non terminali, mentre a destra possono esserci sia terminali che non terminali (come già detto in precedenza), con libero dal contesto si intende che ogni non terminale a destra può essere sostituito, quindi il contesto non influisce. Questo tipo di grammatiche sono una notazione naturale per la struttura ricorsiva della sintassi dei linguaggi di programmazione.
Nell'esempio di prima è possibile scrivere A → "(" A ")" per mantenere il bilanciamento delle parentesi, o A → "a" A "b".

Le grammatiche viste finora hanno forma $A \to \alpha_1 \dots \alpha_n$ con $\alpha_i \in N \cup T$, e sono grammatiche context-free. Se gli elementi α fossero raggruppati in due gruppi, ad esempio prima tutti non terminali e poi tutti terminali (o viceversa) saremmo in presenza di una ***grammatica regolare***. Ad esempio BExp → (BExp) non è regolare in quanto si ha non terminale - terminale - non terminale e questo non

rispetta le caratteristiche di grammatica regolare, è solo context-free.
ANTLR ad esempio vuole che a destra compaiano sempre non terminali e i terminali a sinistra.

Derivazione
L'idea di base della derivazione è considerare le produzioni come regole di riscrittura: ogni volta che abbiamo un non terminale, possiamo sostituirlo con il lato destro di qualsiasi produzione in cui il non terminale appare sul lato sinistro. Possiamo farlo ovunque in una sequenza di simboli (terminali e non terminali) e ripetere la procedura fino a che non restano solo terminali. La sequenza risultante di terminali è una stringa nel linguaggio definito dalla grammatica.

Possiamo disegnare una derivazione come un albero: la radice dell'albero è il simbolo iniziale della grammatica, e ogni volta che riscriviamo un non-terminale aggiungiamo come suoi simboli i simboli sul lato destro della produzione che è stata usata. Le foglie degli alberi sono terminali che, letti da sinistra a destra, formano la stringa derivata.

Quando una grammatica permette la produzione di diversi alberi di sintassi per alcune stringhe, chiamiamo la grammatica *ambigua*. Se il nostro unico uso della grammatica è descrivere insiemi di stringhe, l'ambiguità non è un problema. Tuttavia, quando vogliamo usare la grammatica per imporre una struttura alle stringhe, la struttura dovrebbe essere sempre la stessa ogni volta.

Disambiguazione
Per disambiguare si utilizzano regole associative (ad esempio le operazioni - e / hanno associatività a sinistra, mentre per + e * è indifferente). La correttezza della struttura dipende dall'associatività dell'operazione, usiamo regole di riscrittura diverse in base ad associatività diverse. Se è associativa a sinistra, creiamo una grammatica left-recursive avendo un riferimento ricorsivo a sinistra solo del simbolo dell'operatore. In presenza di un associatività a destra si crea una grammatica right-recursive.

$$\begin{array}{lcl} E & \rightarrow & E \oplus E \\ E & \rightarrow & \textbf{num} \end{array} \qquad \begin{array}{lcl} E & \rightarrow & E \oplus E' \\ E & \rightarrow & E' \\ E' & \rightarrow & \textbf{num} \end{array} \qquad \begin{array}{lcl} E & \rightarrow & E' \oplus E \\ E & \rightarrow & E' \\ E' & \rightarrow & \textbf{num} \end{array}$$

Un altro modo per disambiguare una grammatica è utilizzare regole di precedenza durante il parsing per risolvere i conflitti (ad esempio * e / hanno la precedenza).

Il *linguaggio* **FOOL** è un semplice linguaggio imperativo che comprende tipi di dato intero e booleano, ha dichiarazioni di variabili e assegnamento, la possibilità di definire funzioni (senza ricorsione) e la print. La struttura di ogni espressione in FOOL è: **let** [variabili]; **in** [scope in cui saranno usate le variabili];

Parsing a discendenza ricorsiva
In questa tipologia di parsing si analizza la sequenza di token provando a ricostruire i passaggi di una *derivazione leftmost*. Questi parser sono chiamati top-down perché simulano una visita anticipata dell'albero di sintassi, ovvero dalla radice alle foglie.

L'idea dietro a questo metodo di parsing è che le regole per un non terminale A definiscono una "funzione" che riconosce A. I lati destri delle regole definiscono la struttura del codice della "funzione". La sequenza di terminali e non terminali nelle regole corrisponde a controllare che i terminali combacino e alle invocazioni delle "funzioni" corrispondenti ai simboli non terminali. La presenza di regole differenti per A è implementata da un "case" o un "if".

Parsing predittivo
I parser predittivi sono simili ai discendenti ricorsivi tranne per il fatto che possono predire quale produzione usare controllando i token successivi e senza backtracking. I parser predittivi accettano grammatiche LL(k), in cui la prima L sta per Left-to-right, ovvero come viene scansionato lo stream di input, la seconda L sta per Leftmost derivation, ovvero l'ordine di derivazione, e infine *k* indica il numero di token successivi utilizzati per predire.

Funzioni
Nullable è una funzione che per una sequenza di simboli grammaticali indica se tale sequenza può derivare o meno dalla stringa vuota.

Una sequenza di simboli grammaticali è **NULLABLE** (lo scriviamo come Nullable(α)) se e solo se $\alpha \Rightarrow \varepsilon$. Una produzione $N \rightarrow \alpha$ è chiamata nullable se Nullable(α) ritorna true.

$$\begin{aligned}
Nullable(\varepsilon) &= true \\
Nullable(\mathtt{a}) &= false \\
Nullable(\alpha\beta) &= Nullable(\alpha) \wedge Nullable(\beta) \\
Nullable(N) &= Nullable(\alpha_1) \vee \ldots \vee Nullable(\alpha_n), \\
&\quad \textit{where the productions for } N \textit{ are} \\
&\quad N \rightarrow \alpha_1, \quad \ldots \quad, N \rightarrow \alpha_n
\end{aligned}$$

Definiamo la funzione **FIRST**, che data una sequenza di simboli della grammatica (ad esempio, il lato destro di una produzione) restituisce l'insieme dei simboli con cui possono iniziare le stringhe derivate da quella sequenza.

Un simbolo c è in <u>FIRST</u>(α) se e solo se $\alpha \Rightarrow c\beta$ per qualche sequenza β (anche vuota) di simboli grammaticali.

$$
\begin{array}{lcl}
FIRST(\varepsilon) & = & \emptyset \\
FIRST(\mathtt{a}) & = & \{\mathtt{a}\} \\
FIRST(\alpha\beta) & = & \begin{cases} FIRST(\alpha) \cup FIRST(\beta) & \textit{if } Nullable(\alpha) \\ FIRST(\alpha) & \textit{if not } Nullable(\alpha) \end{cases} \\
FIRST(N) & = & FIRST(\alpha_1) \cup \ldots \cup FIRST(\alpha_n) \\
 & & \textit{where the productions for } N \textit{ are} \\
 & & N \rightarrow \alpha_1, \quad \ldots \quad , N \rightarrow \alpha_n
\end{array}
$$

Per determinare quando possiamo selezionare produzioni nullable durante l'analisi predittiva, introduciamo gli insiemi **FOLLOW** per non terminali.
Un simbolo terminale a è in <u>FOLLOW</u>(N) se e solo se vi è una derivazione dal simbolo di inizio S della grammatica tale che $S \Rightarrow \alpha N a \beta$, dove α e β sono sequenze (anche vuote) di simboli grammaticali.

In altre parole, un terminale c è in FOLLOW(N) se può seguire N in qualche punto in una derivazione. A differenza di FIRST(N), questa non è una proprietà delle produzioni per N, ma delle produzioni che (direttamente o indirettamente) usano N nel loro lato destro.

Sia data una grammatica G = (N, T, →, S) e sia X ∈ N (X è un non terminale).
FOLLOW(X) è l'insieme dei simboli terminali e $ definiti come:
Se lo stato S è lo stato iniziale allora:
FOLLOW(S) = { $ }
Se la produzione ha la forma δXγ con γ non ε il FOLLOW(X) è il FIRST(γ) senza ε
$FOLLOW(X) = \bigcup_{Z \rightarrow \delta X \gamma \text{ in } G} FIRST(\gamma) \setminus \{\varepsilon\}$
Se la produzione ha la forma δXγ e γ può essere ε il FOLLOW(X) è il FOLLOW(Z)
$FOLLOW(X) = \bigcup_{Z \rightarrow \delta X \gamma \text{ in } G \text{ and } NULLABLE(\gamma)} FOLLOW(Z)$

- Quando il simbolo iniziale non appare alla destra delle produzioni, "$" è l'unico simbolo nel suo FOLLOW.
- FOLLOW non contiene mai "ε".
- FOLLOW è definito solo per non-terminali: si potrebbe estendere la definizione ai terminali ma sarebbe inutile per la tabella LL(1).

Parsing LL(1)
Dato il simbolo *c* viene scelta la produzione $N \rightarrow \alpha$ se
$c \in FIRST(\alpha)$, or

$Nullable(\alpha)$ and $c \in FOLLOW(N)$

Se è sempre possibile scegliere una produzione unicamente utilizzando queste regole, stiamo parlando di parsing LL(1), dove 1 indica il numero di simboli di lookahead.

Tabella di parsing LL(1)
Si crea una tabella con i non terminali sulle righe e i terminali sulle colonne, inserendo in ogni cella la produzione che porta dal non terminale della riga al terminale nella colonna. Una produzione N → α viene scritta nella cella (N, a) se a è in FIRST(α) o se si ha Nullable(α) = true e a è in FOLLOW(N).

Ad esempio nel caso della grammatica seguente, si avrebbe la tabella a destra:

E → T X
T → (E) Y | int Y
X → + E | ε
Y → *T | ε

	int	*	+	(	)	$
T	T → int Y			T → (E) Y		
E	E → T X			E → T X		
X			X → + E		X → ε	X → ε
Y		Y → *T	Y → ε		Y → ε	Y → ε

La tecnica è simile al discendente ricorsivo ma invece che il non determinismo (e il backtracking) per ogni non terminale S, si guarda il token successivo, diciamo a, e si prende l'entry [S, a] nella tabella. Si usa uno stack in modo da registrare i terminali e i non terminali nello sviluppo della produzione in [S, a]. L'input è rigettato quando uno stato erroneo viene trovato (entry vuota nella tabella di parsing). L'input è accettato quando l'entry contiene il token di fine input ($).

Riscrivere una grammatica per il parsing LL(1)
Rimuovere la ricorsione sinistra
In situazioni di ricorsione sinistra diretta, in cui è presente un non-terminale che genera alcune produzioni ricorsive a sinistra, ed altre no, come ad esempio l'immagine a sinistra. La grammatica potrebbe essere trasformata nell'espressione regolare equivalente: $(\beta_1 | \ldots | \beta_n)(\alpha_1 | \ldots | \alpha_m)*$ in cui si hanno due non-terminali distinti e la possibilità di produrre ε (immagine a destra).

$$\begin{array}{lcl} N & \rightarrow & N\alpha_1 \\ & \vdots & \\ N & \rightarrow & N\alpha_m \\ N & \rightarrow & \beta_1 \\ & \vdots & \\ N & \rightarrow & \beta_n \end{array}$$

$$\begin{array}{lcl} N & \rightarrow & \beta_1 N_* \\ & \vdots & \\ N & \rightarrow & \beta_n N_* \\ N_* & \rightarrow & \alpha_1 N_* \\ & \vdots & \\ N_* & \rightarrow & \alpha_m N_* \\ N_* & \rightarrow & \end{array}$$

Si parla di ricorsione sinistra indiretta in due casi, o si hanno produzioni mutuamente ricorsive sinistre, o si ha una produzione N → αNβ dove α è *Nullable* (o una combinazione delle due). Anche questo tipo di ricorsioni sinistre possono

essere risolte ma è abbastanza complicato e non viene affrontato.

$$\begin{array}{lcl} N_1 & \rightarrow & N_2\alpha_1 \\ N_2 & \rightarrow & N_3\alpha_2 \\ & \vdots & \\ N_{k-1} & \rightarrow & N_k\alpha_{k-1} \\ N_k & \rightarrow & N_1\alpha_k \end{array}$$

Fattorizzazione sinistra
Se due produzioni per lo stesso non-terminale iniziano con la stessa sequenza di simboli, hanno ovviamente i gli insiemi FIRST sovrapposti. È necessario riscrivere questo in modo tale che le produzioni sovrapposte siano trasformate in un'unica produzione che contiene il prefisso comune delle produzioni e utilizza un nuovo non ausiliario non terminale per i diversi suffissi. Dalla grammatica a sinistra si passa a quella a destra.

$$\begin{array}{lcl} Stat & \rightarrow & \textbf{id} := Exp \\ Stat & \rightarrow & Stat \; ; Stat \\ Stat & \rightarrow & \textbf{if}\, Exp\, \textbf{then}\, Stat\, \textbf{else}\, Stat \\ Stat & \rightarrow & \textbf{if}\, Exp\, \textbf{then}\, Stat \end{array}$$

$$\begin{array}{lcl} Stat & \rightarrow & \textbf{id} := Exp \\ Stat & \rightarrow & \texttt{if}\; Exp\; \texttt{then}\; Stat\; Elsepart \\ & & \\ Elsepart & \rightarrow & \texttt{else}\; Stat \\ Elsepart & \rightarrow & \end{array}$$

Riassumendo la costruzione di parser LL(1)
1. eliminare ambiguità
2. eliminare ricorsione sinistra
3. applicare fattorizzazione sinistra dove richiesta
4. aggiungere una produzione iniziale extra S’ → S$ alla grammatica
5. Calcolare il FIRST per ogni produzione, e il FOLLOW per ogni non terminale
6. Per il non terminale N e il simbolo di input c, si sceglie la produzione N → α dove: c ∈ FIRST(α) o Nullable(α) e c ∈ FOLLOW(N); questa scelta è valida sia per tabelle che per programmi a discendenza ricorsiva.

Le seguenti espressioni ANTLR:
```
exp: ‘a’ | exp ‘a’;
exp: ‘a’+;
```
Sono entrambe corrette, ma la prima avrà un albero sintattico molto profondo (in base al numero di ‘a’ in input), il secondo sarà un albero in cui la radice ha tanti figli ‘a’ quante sono in input, molto più compatto.

Semantic Analysis

Scopes e symbol table

L’analisi semantica cattura gli errori non rilevati dal lexer e dal parser, alcuni errori semantici tipici sono dichiarazioni multiple per la stessa variabile (nello stesso scope), una variabile non dichiarata ma utilizzata nel codice, type mismatch: il lato sinistro di un assegnamento ha tipo diverso rispetto al lato destro, argomenti sbagliati quando si invoca un metodo, ed altri ancora.

Un semplice *analizzatore semantico* lavora in due fasi:
1. controllo semantica con popolamento symbol-table: attraversa l’AST creato dal parser e, per ogni scope nel programma

- elabora le dichiarazioni e aggiunge nuove entries alla tabella dei simboli, riportando le variabili dichiarate più volte.
- elabora gli statement e cerca gli utilizzi di variabili non dichiarate, aggiornando i nodi "ID" dell'AST per farli puntare all'entry appropriata della tabella dei simboli.

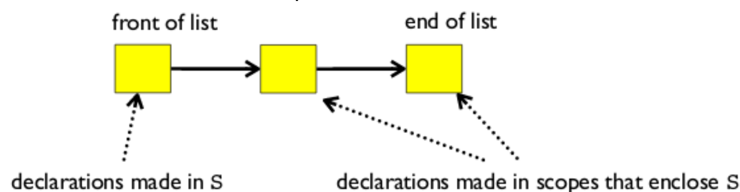
2. type-checking: attraversa l'AST nuovamente e processa tutti gli statement nel programma, ovvero usa le informazioni nella tabella dei simboli per determinare il tipo di ogni espressione e per trovare errori di tipo.

Lo *scope* di una dichiarazione è l'insieme delle posizioni in un programma in cui il nome si riferisce alla dichiarazione. Per tenere traccia di ciò che è visibile o meno si usa la *tabella dei simboli*, ovvero una mappatura tra il nome e l'oggetto a cui il nome si riferisce. Man mano che viene usata questa tabella, si aggiorna continuamente in base a cosa è presente nello scope in cui ci si trova.

Spaghetti Stacks
Uno *spaghetti stack* è una struttura ad albero in cui ogni elemento mantiene un puntatore al padre (e non viceversa). Di conseguenza, essendo la tabella dei simboli una struttura collegata di scopes in cui ognuno salva il collegamento al padre, è di fatto uno spaghetti stack. Ogni punto nel programma (ad es. ogni nodo dell'albero della sintassi) ha la propria tabella dei simboli.

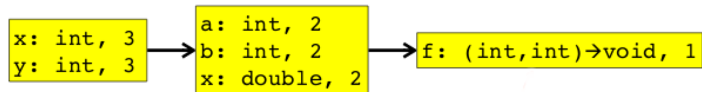
Le operazioni fondamentali che una tabella dei simboli deve fare sono:
- inserire un nuovo nome nella tabella dei simboli, con i vari attributi;
- cercare un nome negli scope correnti (per controllare se è dichiarato più volte o se vi è un utilizzo di un nome non dichiarato);
- fare ciò che va fatto quando si entra in uno scope;
- fare ciò che va fatto quando si esce da uno scope.

Vi sono due possibili implementazioni per la tabella dei simboli:
1. **Lista di hashtables**
La tabella dei simboli non è altro che una lista di hashtables in cui ogni hashtable corrisponde ad ogni scope attualmente visibile (durante la visita del codice). Ad esempio quando si elabora uno scope S:



Prendiamo come esempio il seguente codice:
```
void f(int a, int b) {
   double x;
   while(...) { int x, y; … }
}
```

```
void g() { f(4, 5); }
```
Trovandoci alla linea evidenziata la situazione della lista di hashtables è la seguente:

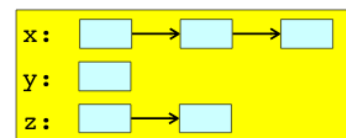


La terza tabella, ad esempio, contiene `nome: dominio → codominio`

Questo tipo di implementazione non permette di applicare la ricorsione. Per poter usare la ricorsione si possono usare interfacce, oppure si usano due passate, nella prima si crea l'interfaccia e poi si va a valutare.

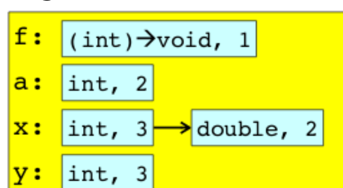
Le operazioni effettuabili sono:
- quando si *entra* in un nuovo scope: si incrementa il numero di livello corrente e si aggiunge una nuova hashtable in testa alla lista.
- per elaborare una *dichiarazione* di x: si cerca x nella prima tabella della lista, se è presente si lancia un errore "variabile dichiarata più volte", altrimenti si aggiunge x alla prima tabella nella lista.
- per elaborare un *utilizzo* di x: si cerca x partendo dalla prima tabella nella lista, se non è presente si cerca in quelle successive, se non è presente in nessuna tabella si lancia un errore "variabile non dichiarata".
- quando si *esce* da uno scope: si rimuove la prima tabella dalla lista e si decrementa il valore del livello corrente.

2. **Hashtable di liste**



L'idea è che quando si processa uno scope S, Si ha solo una hashtable che contiene un'entry per ogni nome che abbia una qualche dichiarazione nello scope S o in uno scope che racchiude S. Ogni nome ha una lista associata di entries della symbol-table: il primo elemento della lista corrisponde alla dichiarazione nella chiusura più vicina, quelli successivi corrispondono a dichiarazioni negli scope precedenti.

Prendiamo ad esempio il seguente codice:
```
void f(int a) {
   double x;
   while(...) { int x, y; … }
   void g() { f(); }
}
```
Trovandoci alla linea evidenziata la situazione della hashtable di liste è la seguente:



Il livello di informazione dell'annidamento è cruciale, l'attributo del livello salvato in ogni elemento della lista permette di determinare se la dichiarazione nella chiusura più vicina è stata effettuata nello scope corrente o in uno scope che lo racchiude.

Le operazioni effettuabili sono:
- quando si *entra* in nuovo scope: si incrementa il numero del livello corrente.
- per elaborare una *dichiarazione* di x: si cerca x nella tabella dei simboli, se x esiste si prende il numero di livello dal primo elemento, se è uguale al livello corrente si lancia un errore "variabile dichiarata più volte", altrimenti si aggiunge un nuovo elemento in testa alla lista con tipo appropriato e livello corrente.
- per elaborare un *utilizzo* di x: si cerca x nella tabella dei simboli, se non è presente si lancia un errore "undeclared variable".
- quando si *esce* da uno scope: si passa da tutte le entries della tabella dei simboli, controllando ogni primo elemento di ogni lista, se il livello è uguale a quello corrente, allora viene rimosso dalla propria lista (se la lista rimane vuote si elimina la riga nella hashtable), alla fine si decrementa il livello corrente.

Type checking
Le proprietà di conformità dei tipi sono spesso legate al contesto, cioè non è sufficiente controllare l'appartenenza a un linguaggio context-free. Di conseguenza, il controllo viene fatto da una fase che (concettualmente) viene dopo l'analisi della sintassi (sebbene possa essere intervallata con essa).

Un *tipaggio forte* significa che l'implementazione del linguaggio assicura che ogniqualvolta viene eseguita un'operazione, gli argomenti dell'operazione sono di un tipo per cui è stata definita l'operazione, quindi, ad esempio, non è possibile concatenare una stringa e un numero a virgola mobile. Ciò è indipendente dal fatto che sia garantito staticamente (prima dell'esecuzione) o dinamicamente (durante l'esecuzione). Al contrario, un linguaggio *debolmente tipizzato* non garantisce che le operazioni vengano eseguite su argomenti che hanno senso per l'operazione.

Una variabile è associata a uno dei due tipi *int* o *bool*. Una funzione è legata al suo tipo, che consiste nei tipi dei suoi argomenti più il tipo del suo risultato. Un tipo è un insieme di valori e un insieme di operazioni su quei valori.

Definizione formale di type checking
Il type checking è il processo in cui si controlla che il programma obbedisca al type system. Spesso include l'*inferenza di tipi* in parti del programma. Le regole di inferenza hanno la forma "se l'ipotesi è vera allora la conclusione è vera", e il type checking viene calcolato ragionando come "se E_1 ed E_2 hanno certi tipi, allora E_3 ha un certo tipo.

Chiamiamo **ambiente** la somma Γ = vtable + ftable. Γ è una mappa che ritorna il

tipo delle variabili (es. Γ(x) = int) e il tipo delle funzioni (es. Γ(f) = int → bool).
Se per "`num`" indichiamo "`return(int)`" nel CheckExp vogliamo dire che il tipo num sarà sempre intero, e nell'ambiente si indica come `Γ ⊢ num : int` ("Γ implica che num sia di tipo int"). In questo caso si parla di assioma, ovvero un caso in cui non è richiesta alcuna invocazione ricorsiva.

L'ambiente Γ è una mappa che prende identificatori (di variabili e funzioni) e ritorna tipi. Γ può essere formalizzata esplicitando la lista di bindings, come ad esempio: Γ = [x → int, y → bool], e quindi Γ(x) = int e Γ(y) = bool. Per estendere Γ ad esempio con z, in notazione Γ · [z → char], la stessa notazione è usata per sovrascrivere un elemento esistente.

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ [\text{Var}]$$

Nell'immagine a lato il dividendo (sopra) indica l'ipotesi ovvero che x sia di tipo T, il divisore (sotto) indica la conclusione ovvero che x sia mappato in T all'interno di Γ. E infine a destra è presente il nome della regola di typing.

Per formalizzare invocazioni ricorsive, dato l'esempio della somma, come segue:

```
Exp1 + Exp2 : t1 = CheckExp(Exp1, vtable, ftable)
              t2 = CheckExp(Exp2, vtable, ftable)
              if ((t1 = int) && (t2 = int)) then return(int)
              else error()
```

Si ha la formalizzazione:

$$\frac{\Gamma \vdash \text{Exp1} : t1 \qquad \Gamma \vdash \text{Exp2} : t2 \qquad t1 = int = t2 \qquad \Gamma(+) = int \times int \rightarrow int}{\Gamma \vdash \text{Exp1} + \text{Exp2} : int}\ [\text{Plus}]$$

O in caso di if-then-else ad esempio:

$$\frac{\Gamma \vdash \text{Exp1} : T1 \qquad \Gamma \vdash \text{Exp2} : T2 \qquad \Gamma \vdash \text{Exp3} : T3 \qquad T1 = bool \qquad T2 = T3}{\Gamma \vdash \text{if Exp then \{ Exp1 \} else \{ Exp2 \}} : T2}\ [\text{If}]$$

In caso di molteplici dichiarazioni di variabili (decs è una lista):

$$\frac{\Gamma \vdash exp : T \qquad \Gamma \bullet [id \mapsto T] \vdash decs : \Gamma'}{\Gamma \vdash T\ id = exp\ ;\ decs : [id \mapsto T] \bullet \Gamma'}\ [\text{VarDec}]$$

Nota: solo dichiarazioni di variabili/funzioni producono un nuovo ambiente (Γ') con le modifiche apportate dalla dichiarazione, tutto il resto (chiamate di funzioni, utilizzi di variabili, ...) producono solo un tipo.

Prendiamo l'esempio: `int x = 5; bool x = true;` dato un certo ambiente Γ:

____

Γ ⊢ int x = 5; bool x = true;   ? ? ?

Step 1: si pone in basso la conclusione che si vuole raggiungere, i punti interrogativi verranno sostituiti dall'ambiente finale ottenuto.

Γ ⊢ 5 : int   Γ · [x → int] ⊢ bool x = true : Γ’
————————————————————————————

Γ ⊢ int x = 5; bool x = true;   ? ? ?

Step 2: il passo successivo sviluppa banalmente 5 come intero, quindi il primo elemento della lista di dichiarazioni è risolto, si passa poi al secondo, che partendo dall’ambiente modificato con l’elemento intero aggiunto, ritorna un altro ambiente Γ’ essendo una dichiarazione di variabile.

                                        Γ · [x → int] ⊢ true : bool
                                        —————————————————

Γ ⊢ 5 : int   Γ · [x → int] ⊢ bool x = true : Γ’
————————————————————————————

Γ ⊢ int x = 5; bool x = true;   ? ? ?

Step 3: si sviluppa questa dichiarazione, vedendola come una lista di altre dichiarazione (in questo caso però solo 1) e risolvendone il primo elemento.

                                        Γ · [x → int] · [x → bool] ⊢ ε : Γ’’ (con Γ’’ = ∅)

                                        Γ · [x → int] ⊢ true : bool
                                        —————————————————

Γ ⊢ 5 : int   Γ · [x → int] ⊢ bool x = true : Γ’
————————————————————————————

Γ ⊢ int x = 5; bool x = true;   **[x → int] · [x → bool]**

Step 4: infine si continua ad aggiornare l’ambiente di partenza, prendere “ricorsivamente” il primo elemento della lista e lasciare per dopo il resto della lista. In questo caso l’elemento successo in lista è nulla, quindi ε, che produce per definizione sempre un ambiente vuoto. A questo punto è possibile aggiornare la conclusione con l’ambiente risultante.

Il risultato è che nell’ambiente finale x viene sovrascritto, ed è utilizzabile solo come bool.

ASSIOMA: **Γ ⊢ ε : ∅**

Un altro assioma che vale in qualunque ambiente è: ∅ ⊢ == : T x T → bool, di seguito un esempio:

                                        ∅ · [x → int](x) = int
                                        ∅ · [x → int] ⊢ 1 : int
                                        la primitiva == dati due tipi uguali ritorna bool
                                        ∅ · [x → int] ⊢ x == 1 : bool   ∅ · [x → int] · [x → bool] ⊢ ε : ∅
                                        x in questo ambiente è un intero
        ∅ ⊢ 1 : int   ∅ · [x → int] ⊢ bool x = (x == 1);   ε : [x → bool]
                                                                                ottengo l’ambiente ε : [x → bool]
        ∅ ⊢ int x = 1; bool x = (x == 1);   [x → int] · [x → bool]
                                            ambiente (sarebbe uguale a [x → bool] perché si sovrascrive)

Soundness

Un sistema di tipaggio è *sound* se ogni volta che $\Gamma \vdash e : T$, durante la computazione, e realmente viene valutato come tipo T. Quindi non ci saranno mai errori di tipo a runtime, ma eventualmente solo in compilazione.

Subtyping

Si consideri il seguente programma: `let T x = e in e'`

Secondo il sistema di tipo corrente si ha l'albero di prova:

$$\dfrac{\dfrac{\Gamma \vdash e: T \qquad \Gamma \bullet [x \mapsto T] \vdash \varepsilon: \varnothing}{\Gamma \vdash T\ x\ =\ e:\ [x \mapsto T]}\,[\text{VarDec}] \qquad \Gamma \bullet [x \mapsto T] \vdash e' : T'}{\Gamma \vdash \text{let } T\ x = e \text{ in } e' : T'}\,[\text{Prog}]$$

Il codice precedente non tipa il caso in cui:

`let P x = new C in . . .` con C sottoclasse di P

Si definisce allora la relazione `X <: Y` sui tipi per indicare che un oggetto di tipo X può essere usato quando uno di tipo Y è accettabile ("X è sottoclasse di Y").

Una relazione <: sui tipi è chiamata subtyping quando:

- X <: X
- X <: Y se X estende Y
- X <: Z se X <: Y e Y <: Z

Ridefinendo quindi VarDec perché accetti i sottotipi avremmo:

$$\dfrac{\Gamma \vdash e: T' \qquad \Gamma \bullet [id \mapsto T] \vdash decs: \Gamma' \qquad T' <: T}{\Gamma \vdash T\ id\ =\ e\ ;\ decs\ :\ [id \mapsto T] \bullet \Gamma'}\,[\text{SubTVarDec}]$$

Quindi si "spezza" il tipo di e con il tipo di id, specificando che uno è il sottotipo dell'altro.

L'invocazione di funzione con il subtyping ha forma:

$$\dfrac{\Gamma \vdash f: T_1 \times \ldots \times T_n \to T \qquad (\Gamma \vdash e_i : T_i' \qquad T_i' <: T_i)^{i \in 1..n}}{\Gamma \vdash f(e_1, \ldots, e_n) : T}\,[\text{Sbt-Call}]$$

È quindi possibile invocare una funzione con degli argomenti che sono sottotipi dei parametri formali.

Method invocation

$$\dfrac{\Gamma(o) = C \qquad \Gamma(C)(m) = T_1 \times \ldots \times T_n \to T \qquad (\Gamma \vdash e_i : T_i' \qquad T_i' <: T_i)^{i \in 1..n}}{\Gamma \vdash o.m(e_1, \ldots, e_n) : T}\,[\text{Sbt-MtdCall}]$$

Method overriding

$$\dfrac{T_1 <: T_1' \ldots T_n <: T_n' \quad \text{and} \quad T' <: T}{T_1' \times \ldots \times T_n' \to T' <: T_1 \times \ldots \times T_n \to T}$$

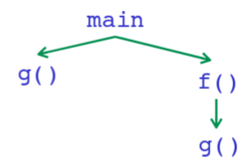
Consideriamo: `let T x = o.m(e₁, . . . , eₙ) in e`

Assumendo o sia di tipo A che può essere istanziato da un oggetto di tipo B (che è una sua sottoclasse). Allora il tipo T' che è il ritorno di B deve essere utilizzabile al posto del tipo T che è il ritorno di A, abbiamo bisogno che T' <: T. Inoltre i tipi di parametro di A devono essere utilizzabili al posto dei tipi di parametro di B, abbiamo bisogno che T_i <: T_i'. In sostanza se faccio l'override di un metodo m in una classe B che estende una classe A si è obbligati a prendere argomenti del tipo della superclasse.

Bytecode Generation

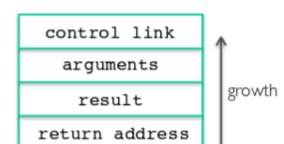
È necessario assumere che l'*esecuzione* sia *sequenziale*, ovvero il controllo si sposti da un punto del programma ad un altro in un modo ben definito. E quando una procedura è chiamata, il *controllo* alla fine *ritorna il punto immediatamente successivo alla chiamata*.

Attivazione e ciclo di vita
L'invocazione di funzione f è un'attivazione di f. Il *ciclo di vita di un'attivazione* di f consiste in tutti i passaggi per eseguire f, ed include tutti i passaggi nelle procedure di chiamata ad f.
Il *ciclo di vita di una variabile* x è la porzione di esecuzione in cui x è definita. Da notare che il ciclo di vita è un concetto dinamico (run-time), mentre lo scope è un concetto statico.
Ricordiamo che l'albero di attivazione dipende dal comportamento a run-time e può differire per ogni programma in input.

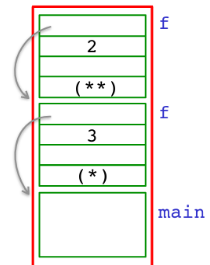


Le informazioni necessarie per gestire un attivazione di funzione è chiamata **activation record** (AR) o **frame**. Se f chiama g, allora il record di attivazione di g contiene un mix di informazioni sia su f che su g, e l'AR di g si comporta:
- f è "sospesa" fino a quando g non ha terminato, quando ciò accade si riesuma f.
- l'AR di g contiene informazioni di cui si ha bisogno per l'esecuzione di f.
- l'AR di g può anche contenere: il valore di ritorno di g (di cui ha bisogno f), i reali parametri per g (forniti da f), lo spazio per le variabili locali di g.

Il contenuto di un tipico AR per g è:
1. spazio per il *valore di ritorno* di g
2. *parametri reali*
3. *puntatore all'AR precedente* (il *control link* punta all'AR del chiamante di g)
4. lo *stato* della macchina *precedente alla chiamata* g (contenuto dei registri e il program counter)
5. variabili locali
6. altri valori temporanei.



Il main non ha argomenti o variabili locali, e il suo risultato non è utilizzato da nessuna parte, mentre (*) e (**) sono indirizzi di ritorno delle invocazioni di f.



Il compilatore deve determinare, a tempo di compilazione, il layout degli AR e generare codice che correttamente acceda indirizzi nel record di attivazione. Quindi il layout degli AR e il generatore di codice devono essere progettati insieme.

Variabili globali

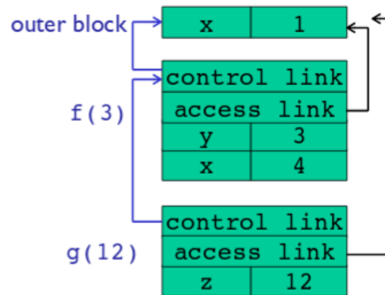
Tutte le referenze ad una variabile globale puntano allo stesso elemento. È sbagliato memorizzare una variabile globale in un AR. Le variabili globali sono assegnate ad un indirizzo *fisso* una volta, le variabili con indirizzi fissi sono allocate staticamente. In base al linguaggio possono essere altri valori allocati staticamente.

Variabili dichiarate in scope esterni

Le referenze ad una variabile dichiarata in uno scope esterno dovrebbero puntare ad una variabile memorizzata in un altro AR. Precisamente al record di attivazione secondo la regola dell'annidato più vicino.

```
int x = 1;
  { int g(int z){ return x+z; }
          . . .
          { int f(int y){
                int x = y+1;
                return g(y*x) }
            f(3);
          }
          . . .
  }
```

the symbol table at this point



Il valore dell'access link di un nuovo record di attivazione è stabilito come segue:
- quando si entra in un blocco interno o quando viene chiamata una funzione dichiarata nello scope corrente:
`ACCESS_LINK = indirizzo dell'ACCESS_LINK nell'AR corrente`
- quando una funzione chiama se stessa ricorsivamente o chiama un'altra funzione dichiarata nell'enclosing syntactical block:
`ACCESS_LINK = valore dell'ACCESS_LINK dell'AR corrente`
- quando in generale si chiama una funzione fuori dallo scope corrente:
`ACCESS_LINK =  segue la catena degli ACCESS_LINK per la differenza tra il livello di annidamento corrente e quello della dichiarazione di funzione.`

Heap storage
Un valore che sopravvive alla procedura che lo crea non può essere mantenuto nell'AR, ad esempio: `Bar foo() { return new Bar(); }`
I linguaggi con dati allocati dinamicamente utilizzano un *heap* per memorizzare i dati dinamici. Gli AR nello stack sono deallocati quando il controllo esce dallo scope corrispondente (la funzione termina), l'heap no.

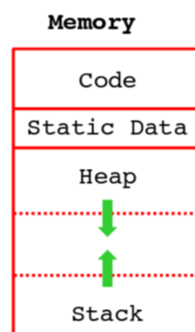
Garbage collection
I dati nell'heap possono essere rimossi quando diventano "spazzatura", in un certo punto p nell'esecuzione di un programma una locazione di memoria m è spazzatura se nessuna continuazione di p può accedere alla locazione m.
Il garbage collector trova la "spazzatura" durante l'esecuzione di un programma, viene invocato quando c'è bisogno di più memoria e la decisione viene presa dal run-time system, non dal programma.

In alcuni programmi, come il C, la deallocazione è responsabilità del programmatore, tuttavia se si effettua un free() ad esempio di un puntatore che viene puntato da un altro non può essere eliminato (si parla di dangling pointers).

Algoritmo di mark-and-sweep
Si assumono bit tag associati ai dati e si assume che gli indirizzi delle locazioni create da un programma siano collezionate in una tabella. L'algoritmo ha 3 step:
1. imposta tutti i bit tag a 0
2. parte da ogni locazione utilizzata direttamente dal programma (le cerca dall'AR) e segue tutti i link, cambiando i bit tag ad 1.
3. considera *spazzatura* tutte le celle con tag = 0

Algoritmo di reference counting
Si assume che ogni datum in memoria abbia un reference counter associato. L'algoritmo ha 4 step:
1. quando un datum è allocato in memoria, si inizializza il contatore a 0
2. quando un puntatore ad un datum è settato, si incrementa il counter
3. quando un puntatore ad un datum è resettato, si decrementa il counter
4. quando il counter risulta essere 0, il datum è *spazzatura*.

La memoria segue il verso delle frecce, quando queste si incontrano si va out of memory:



Generazione di codice per una macchina a pila
Le macchine a pila sono un modello di valutazione semplice. Non sono presenti variabili o registri, si utilizza infatti una pila di valori per i risultati intermedi. Ci si basa quindi su operazioni di push e pop. Nelle macchine a pila ogni operazione può prendere operandi sia dallo stack che dai registri e li inserisce nello stack o in un registro. La posizione degli operandi può essere sia implicita (lo stack) che esplicita (i registri). Quando gli operandi sono impliciti, sono sempre in cima alla pila, quindi non c'è bisogno di specificarli esplicitamente.

Istruzione: add
L'istruzione add effettua 3 operazioni in memoria: 2 letture ed 1 scrittura nello stack. L'idea è utilizzare un registro, in quanto gli accessi ai registri sono veloci, questo diminuisce le operazioni in memoria da 3 ad 1, l'istruzione add diventa:
`$r0 ← $r0 + top_of_stack`.

Esempi di istruzioni assembly per il bytecode
`lw $r1 offset($r2)`
[load word] carica una word di 32-bit dall'indirizzo $r2 + offset in $r1.

`add $r1 $r2 $r3`
[add] $r1 ← $r2 + $r3 , il valore di $r2 più il valore di $r3 è memorizzato in $r1, equivalente l'espressione `sub` che effettua la sottrazione.

`sw $r1 offset($r2)`
[store word] memorizza una word di 32-bit in $r1 all'indirizzo $r2 + offset.

`addi $r1 $r2 n`
[add integer] $r1 ← $r2 + n , il valore di $r2 più n è memorizzato in $r1.

`li $r1 n`
[load integer] $r1 ← n , n è salvato in $r1.

`move $r1 $r2`
[move] $r1 ← $r2 , il valore di $r2 è memorizzato in $r1.

`beq $r1 $r2 label`
[branch equivalence] va al branch label se $r1 = $r2.

`b label`
[branch] va al branch label senza controllare nessuna condizione.

Esempi di macro utilizzate
Nota: **$sp** è il registro che memorizza lo stack pointer
`push $t`
equivale a: `addi $sp $sp -4   sw $t 0($sp)`

pop
equivale a: `addi $sp $sp 4`

$t ← top
equivale a: `lw $t 0($sp)`

La strategia per la generazione di codice prevede che per ogni espressione **e** si generi codice assembly che calcoli il valore di **e** e lo memorizzi in $a0 ($a0 è un registro speciale chiamato *accumulatore*), inoltre il codice assembly deve anche preservare $sp e i contenuti dello stack. Definiamo una funzione per la generazione di codice **cgen(e)** i cui risultati sono generati per e. Deve essere *invariante*, ovvero il risultato del calcolo di un'espressione è sempre nell'accumulatore, e dopo aver calcolato un espressione lo stack torna come prima (questo deve essere effettuato all'interno di cgen(e)).

Per *valutare una costante* semplicemente si copia nell'accumulatore, questa operazione preserva lo stack (invarianza), come richiesto:
`cgen(i) = li $a0 i`

Per valutare un'espressione di *somma* invece:

```
cgen(e1 + e2) = cgen(e1)
                push $a0
                cgen(e2)
                $t1 ← top
                add $a0 $a0 $t1
                pop
```

Si potrebbe pensare di salvare $a0 direttamente in $t1, questo sarebbe sbagliato in quanto ad esempio in 3 + (7 + 5) il registro $t1 verrebbe sovrascritto più volte.

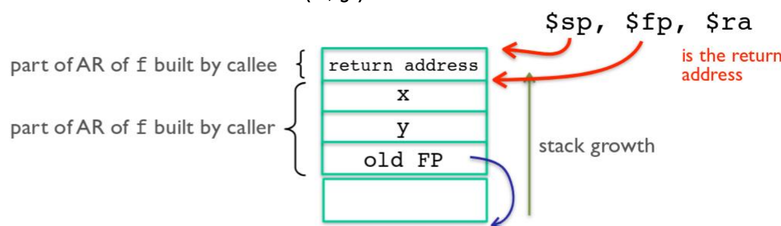
Per valutare un'espressione *condizionale* invece si introduce il concetto di etichetta, ovvero un nome che viene associato ad una porzione di codice, a cui è possibile "saltare" durante l'esecuzione:

```
cgen(if e1 = e2 then e3 else e4) =
        false_branch = newlabel();
        true_branch = newlabel();
        end_if = newlabel();
        cgen(e1)
        push $a0
        cgen(e2)
        $t1 ← top
        pop
        beq $a0 $t1 true_branch
      false_branch:
        cgen(e4)
        b end_if
      true_branch:
        cgen(e3)
      end_if:
```

the labels must be fresh to avoid name clashes

L'activation record

Il codice per le chiamate di funzione e per le definizioni di funzioni dipende dal layout dell'activation record. Per questo linguaggio supponiamo un AR molto semplice: il risultato è sempre nell'accumulatore (non c'è bisogno di salvarlo nell'AR), l'AR mantiene i parametri reali (per $f(x_1, \dots, x_n)$ si pushano $x_n, \dots, x_1$ nello stack), e la disciplina dello stack garantisce che all'uscita dalla funzione $sp sia lo stesso di quando si è entrati (non c'è bisogno di salvare $sp nell'AR), e infine è necessario salvare l'indirizzo di ritorno. Non c'è bisogno di access link in quanto non abbiamo dichiarazioni annidate. È necessario però avere un puntatore ad una reference position nell'AR corrente.
Per una chiamata a f(x, y) l'AR sarà:



La parte in figura è nota col nome di parte statica della chiamata di funzione. I risultati intermedi che la funzione chiamata pusherà in cima alla pila si chiameranno invece parte dinamica. Se all'interno della parte dinamica viene referenziato un parametro della funzione, come facciamo a sapere la profondità a cui si trova nello stack? Siccome determinare tale profondità è problematico, ricorriamo all'utilizzo di un registro $fp, noto come *frame pointer*, per mantenere un link alla parte statica dell'invocazione di funzione. All'interno di tale parte statica, l'offset dei parametri formali è noto, per cui accedervi è banale.
Si noti come la parte statica contiene essa stessa un frame pointer: old FP, che costituisce un link alla parte statica del precedente record di attivazione.

Invocazione di funzione

Detto questo, il codice per l'invocazione di funzione è:

```
cgen( f(e₁, ..., eₙ) )   =   push $fp;          salva il frame pointer
                             cgen(eₙ);          salva i parametri reali in ordine inverso
                             push $a0;
                             …
                             cgen(e₁);
                             push $a0;
                             jal f_entry;       salva l'indirizzo di ritorno in $ra
```

Dove <u>jal label</u> salta all'etichetta e salva l'indirizzo dell'istruzione successiva in $ra.

Abbiamo già fatto notare come i parametri formali abbiano offset definito all'interno della parte statica. Ma tale offset deve essere salvato da qualche parte per poter essere inserito nel programma assembly durante la generazione del codice. L'offset viene salvato nella *symbol table*, come metadato della variabile stessa.

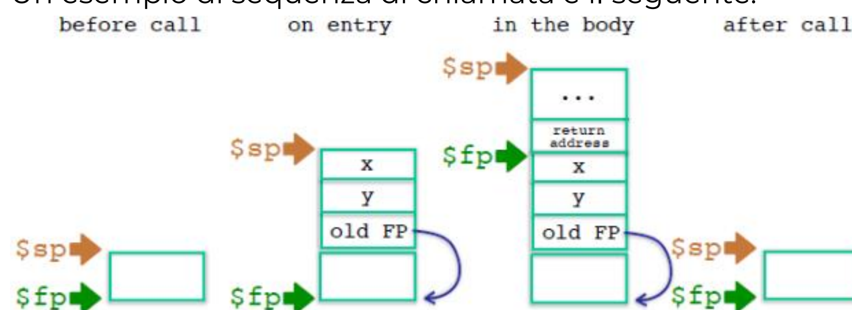
Definizione di funzione

Come si può leggere nello schema iniziale, la funzione chiamata si occupa di pushare l'indirizzo di ritorno nell'AR:

```
cgen( int f(int x1, ..., int xn) = e )   =     f_entry:
                              move $fp $sp;
                              push $ra;
                              cgen(e);
                              $ra ← top;
                              addi $sp $sp k;
                              $fp ← top;
                              pop;
                              jr $ra;
```

L'istruzione jr sposta l'esecuzione all'indirizzo memorizzato in $ra.
Invece k corrisponde a 4n + 4, di modo che l'istruzione relativa corrisponde a fare pop n + 1 volte. A questo punto è palese come: la forma dell'activation record debba essere disegnata attentamente, e soprattutto assieme al generatore del codice. Inoltre la generazione stessa può essere eseguita in modo ricorsivo.

Un esempio di sequenza di chiamata è il seguente:



Variabili globali e statement

L'aggiunta di variabili globali e statement al linguaggio in analisi richiede la modifica del codice generato e degli AR. Il linguaggio è ora:

```
P → D ; P | E | S
D → T id(ARGS) = P
 ARGS → id, ARGS | id
E →  int | id | if E1 = E2 then E3 else E4
     | E1 + E2 | E1 − E2 | id(E1,…,En)
S → ( id = E ; | id(E1,…,En) ; )+
T → int | void
```

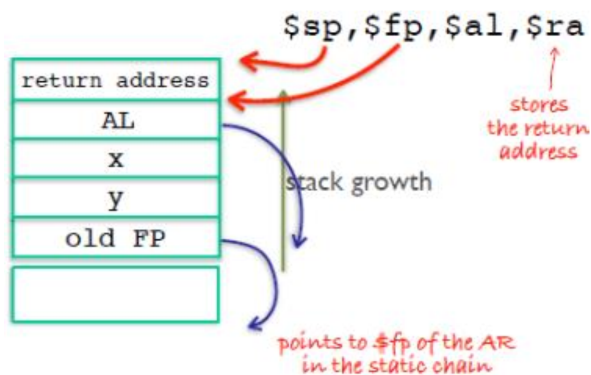
Come generiamo il codice per un assegnamento? Per esempio, per x=e? Se x esiste all'interno del record corrente, il lavoro da svolgere è molto semplice:

```
cgen(x = e)  =    cgen(e);
            sw $a0 offset[x]($fp);
```

Dove offset[x] non è altro che l'offset di x all'interno dell'AR, come dichiarato nella symbol table.

Se invece x è una variabile globale (o statica, etc.), gli AR per come li abbiamo visti finora non ci permettono di risalire alla sua posizione. In generale, la presenza di

variabili globali ci obbliga ad aggiungere un nuovo campo al record di attivazione. Questo è l'access link:



L'access link punta all'ambiente statico immediatamente esterno alla funzione. L'ambiente statico è l'AR del primo blocco di codice che racchiude il corpo della funzione, e non l'AR del chiamante della funzione (che è invece puntato da *old FP*, noto anche come control link). Gli access link costituiscono, complessivamente, la catena statica, che non dipende dall'esecuzione del programma, ma solo da come è scritto il suo codice.

Il bytecode generato per accedere una variabile globale e per modificarla è il seguente:

```
cgen(x)     =      lw $al 0($fp);
            for (i=0; i < nesting_level - lookup(vtable, x).nesting_level;
      i++) lw $al 0($al);
            lw $a0 lookup(vtable, x).offset($al);

cgen(x = e) =      cgen(e);
            lw $al 0($fp);
            for (i=0; i < nesting_level - lookup(vtable,x).nesting_level;
            i++) lw $al 0($al);
            sw $a0 lookup(vtable, x).offset($al);
```

Dove ovviamente i for e le invocazioni di lookup non sono istruzioni del linguaggio bytecode, ma piuttosto shorthand di operazioni di preprocessing di cui deve occuparsi il generatore.
L'invocazione di funzione diventa:

```
cgen(f(e1, ..., en))  = push $fp;
                  cgen(en);
                  push $a0;
                  …
                  cgen(e1);
                  push $a0;
                  lw $al 0($fp);
                  for (i=0; i < nesting_level - lookup(vtable,
            x).nesting_level; i++)
                        lw $al 0($al);
```

```
push $al;
jal lookup(vtable, ftable, f).label;
```

Mentre la definizione di funzione:

```
cgen(T f(T1 x1,...,Tn xn) = e) =    lookup(vtable, ftable, f).label:
                                    move $fp $sp
                                    push $ra
                                    cgen(e)
                                    $ra ← top
                                    addi $sp $sp k
                                    $fp ← top
                                    pop
                                    jr $ra
```

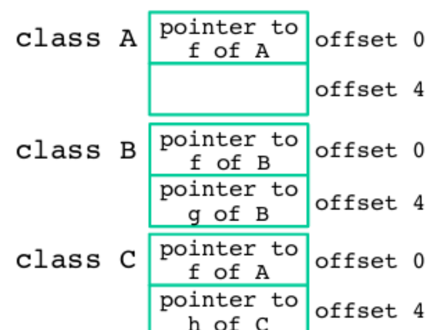
Funzioni di ordine superiore

In alcuni linguaggi le funzioni possono essere passate come parametri:
fun bool f(x:(int, int) → bool, a:int, b:int) { ... bool z = x(a, b); ... }
La funzione f dovrebbe preparare l'AR per l'esecuzione di x(a, b), tuttavia f non è in grado di impostare l'"access link", inoltre dovrebbe puntare all'AR più recente in cui la funzione di chiamata è stata dichiarata (basandosi sulla differenza con il livello di annidamento di tale dichiarazione, calcolata a tempo di compilazione).
La soluzione a questi problemi è che il chiamante di f, che passa il reale valore per il parametro x, dovrebbe anche passare una coppia contenente: l'indirizzo del codice della funzione g che viene realmente passata; l'indirizzo dell'AR più recente in cui g è stato dichiarato. Questa coppia è chiamata **closure** della funzione passata. Il valore di x è impostato a questa coppia e quando viene eseguito x(a, b) l'access link verrà settato con il secondo elemento di questa coppia.

Generazione di codice per linguaggi object-oriented

Ogni classe ha un insieme fissato di metodi (inclusi quelli ereditati). Una dispatch table (o virtual table) indicizza questi metodi generando un array di indirizzi dei metodi e un metodo f "esiste" ad un offset fissato nella dispatch table di una classe e di tutte le sue sottoclassi.
Ad esempio:

```
class A {
   int a = 0;
   int d = 1;
   int f(){ return a = a + d; }
}
class  B extends A {
   int b = 2;
   int f(){ return a; }//override
   int g(){ return a = a - b; }
}
class  C extends A {
   int c = 3;
   int h(A x) { return a = x.f()*c; }
}
```



Il dispatch pointer in un oggetto di classe *C* punta alla dispatch table per la classe *C*. Ad ogni metodo f della classe C è assegnato un offset O_f nella dispatch table a tempo di compilazione. L'offset è inserito nell'entry della symbol table del metodo f di classe C.
Per implementare un dispatch dinamico e.f(), è necessario che O_f sia l'offset del metodo f nella dispatch-table associato con il tipo statico di e. Si valuta e, ottenendo un oggetto o (che può essere di qualsiasi sottoclasse). E infine, data D la dispatch table di o, si esegue il metodo puntato da $D[O_f]$.