# Compilatori

**A.A. 2023/2024**

# Indice

# 1. Introduzione

Dato un programma $P$ e una memoria concreta $\sigma$, vogliamo trovare una relazione $\rightarrow^\star P', \sigma'$. Questa è chiamata semantica operazionale matematica (quest'ultima per dimostrare dei teoremi). Nell'astrazione, ad esempio, possiamo trovare `int` che è rappresentante di un insieme grande di valori; in quella concreta potremmo avere semplicemente `42`. Si deve fare controllo d'errori di tipo; se non ci sono errori deve essere corretto (nessun falso positivo).

Le varie differenze che vi sono tra compilatori ed interpreti sono in merito a:
- velocità (compilatori più veloci).
- cosa prendono (compilatori l'intero programma; interprete singola linea).
- compilatori trasformano tutte le righe in codice macchina e poi le eseguono; gli interpreti trasformano in codice macchina riga per riga.

Gli step di compilazione e interpretazione di Java:
1. Analisi lessicale: si individuano gli elementi "interessanti" e dunque si rimuovono commenti, tab e spazi superflui.
2. Analisi sintattica: si controlla la conformità ad una grammatica (quest'ultima più complicata della grammatica dell'analisi lessicale).
3. Analisi semantica: controllare se il programma è ben tipato, variabili tutte inizializzate, codice morto e così via. L'analizzatore deve essere progettato e poi dimostrare che sia corretto.
4. Generazione di codice intermedio, bytecode: la macchina che lo esegue è una VM.

Nel caso di altri linguaggi, come C, bisogna invece:
4. Generare codice intermedio per una macchina astratta.
5. Ottimizzazione del codice (la semantica deve rimanere invariata).
6. Generazione del codice macchina.

Il codice assembler è untyped.

Con le grammatiche si definiscono insiemi di regole finite per definire tutte le cose esprimibili da quel linguaggio. Si usano grammatiche libere dal contesto (context-free) perché a destra può avere simboli non terminali. Si usano automi a pila. È una tupla $(N, T, \rightarrow, S)$ dove $N$ è insieme finito di simboli non terminali. $T$ è insieme finito di simboli terminali. $\rightarrow$ è insieme finito di regole del tipo $A \rightarrow \alpha_1 ... \alpha_n$ con $A \in N$ e $\alpha_1 ... a_n \in N \cup T$. $S \in N$ è simbolo iniziale. Se prendiamo $\gamma$ e $\delta$ come sequenza di simboli t.c. $N \cup T$. La **derivazione** la si può vedere come una semplificazione che porta ad un programma con soli simboli terminali $\gamma \rightarrow^\star \delta$. Una derivazione a one-step è definita come $\gamma A \delta \rightarrow \gamma \alpha_1 ... \alpha_n \delta$ con $A \rightarrow \alpha_1 ... \alpha_n$ Si ha la derivazione a $0+$ passi $\rightarrow^\star$ e la derivazione a $1+$ passi $\rightarrow^+$.

Data una grammatica $G$ si definisce il linguaggio come

$$\mathcal{L}(G) = \{\gamma \mid \gamma \in T^\star \wedge S \rightarrow^+ \gamma\}$$

con $T^\star$ chiusura di Kleene.

## 1.1. ANTLR

Da una grammatica genera lexer e parser. Presa una grammatica definita come

```
grammar ArrayOfInt;

init : '{' value (',' value)* '}';
value : init | INT;
INT : [0-9]+;
```

`INT` è una grammatica regolare. Si usano automi a stati finiti. Viene usata per l'analisi lessicale. Le grammatiche context-free sono usate per l'analisi sintattica.

Con ANTLR4 si può testare il parsing e generarlo (insieme al lexer).

```
$ antlr4-parse parser/ArrayOfInt.g4 init -tree
{3,4}

(init:1 { (value:2 3) , (value:2 4) })
```

```
$ antlr4-parse parser/ArrayOfInt.g4 init -tree
{3,4,5,}
line 1:7 mismatched input '}' expecting {'{', INT}
(init:1 { (value:2 3) , (value:2 4) , (value:2 5) , (value:1 }) })
```

# 2. Analisi lessicale

La struttura lessicale sono tutti gli elementi (parole chiavi, parentesi, interi e reali, ...). Quindi deve restituire un errore se non vengono rispettate cose specificate (tipo l'inizio di una variabile per numero piuttosto che da una lettera). Si divide la grammatica in **lexer** e **parser**.

L'analisi lessicale divide il testo in **token**. Durante la creazione di essi vengono esclusi (non nel caso di Python) tabulazioni, spazi e commenti. I lessemi includono sequenze di caratteri non usati come i token (tipo gli spazi).

Da un codice

```
if (x == y)
    z = 1;
else
    z = 2;
```

si ha che dal testo

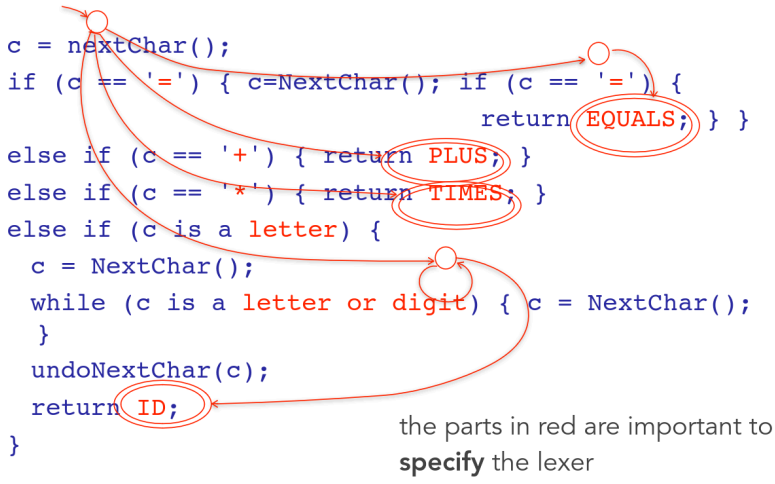
```
if (x == y)\n\tz = 1;\nelse\n\tz = 2;
```

troviamo i token

```
IF, LPAR, ID("x"), EQUALS, ID("y"), RPAR, ...
```

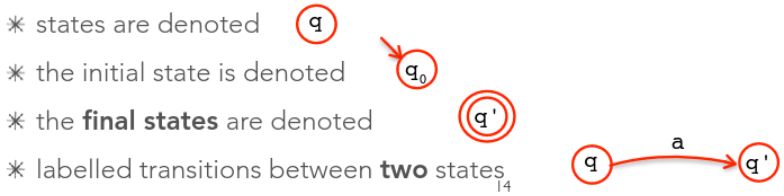
La creazione di un lexer potrebbe esser fatto facendo uno scanner di carattere a carattere. Quello che fa il lexer è chiamato maximal match, e si vede nel caso di fare matching di sequenze di caratteri con un singolo token.

Le frecce tra i token (i pallini nella foto sotto) si chiamano transizioni ed hanno delle etichette. Questo lexer abstract model è un automa a stati finiti non deterministici (NFA). Il non determinismo complica l'esecuzione del programma.




Si ha un sistema di precedenza a first-rule. Il primo elemento preso è quello a cui fa riferimento.

Un NFA è $(Q, \Sigma, \delta, q_0, F)$ e lo rappresentiamo con una notazione grafica:

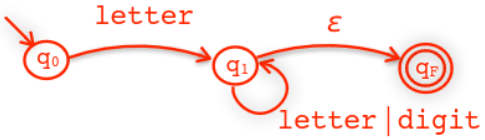


Il linguaggio dato dall'NFA $M$ è $\mathcal{L}(M)$ è infinito, dato che si possono fare infiniti passi in $\delta$.

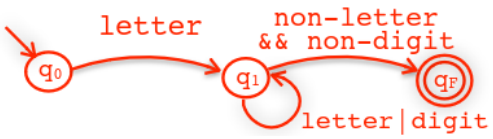
Il design di un lexer avviene prima definendo la descrizione su cosa fa, in cui si descrivono tutti i token in modo formale e poi viene implementato il comportamento di componenti a "basso livello".

L'algoritmo che il lexer deve seguire è:
1. Definizione di un NFA per ogni lessema;
2. Combinazione di NFA con stati iniziali;
3. Usando sempre la maximal match rule, si punta ad avere un automa deterministico (DFA) dai due NFA risultanti;
4. Se si raggiunge uno stato finale allora si prende la posizione dell'input e si legge finché non si cambia stato. Se non ci sono altre transizioni possibili allora si torna indietro all'ultimo stato finale e si ritorna il token a quello stato.

L'identificazione di un token è una $\varepsilon$ transizione da uno stato $q_1$ ad uno finale.




Anche se sarebbe meglio specificare una etichetta al posto di $\varepsilon$ dato che potrebbe finire in $q_f$ in qualsiasi modo rendendo il parsing valido.




In quest'ultimo caso, una volta raggiunto lo stato finale, bisognerebbe fare un passo indietro prima di ritornare l'ID trovato.

In realtà è più comodo usare un DFA al posto di un NFA.

**Teorema di costruzione del sottoinsieme** Dato un NFA $M$ è possibile definire un DFA $M'$ tale che $\mathcal{L}(M) = \mathcal{L}(M')$. La trasformazione si può fare mediante $\varepsilon$ chiusura.

**Teorema dell'algoritmo di Hopcroft** Si va a minimizzare il DFA per numero di stati tali che $\mathcal{L}(M) = \mathcal{L}(M')$. I lexer non fanno nessuna minimizzazione.

Nella pratica vi possono essere dei problemi, come ad esempio di ambiguità. In un automa a stati finiti un `if` potrebbe corrispondere sia ad un `ID` che ad una keyword `IF` (il livello lessicale si occupa di disambiguare ciò).

**Principio di match più lungo** Un lexer ritorna il token che consuma il percorso più lungo.

**Principio di primo match** I token hanno la priorità (ma sulla stessa lunghezza); il lexer può decidere quale token riconoscere se vi è ambiguità su due.

```
stm : ... | 'while' '(' ...;

ID : ('a'...'z');
```

il `while` viene riconosciuto come terminale prima di essere riconosciuto come `ID`.

I **lookaheads** possono essere lunghi a piacere, basta memorizzare l'indice dell'ultimo input letto.

Un equivalente descrizione degli automi (DFA o NFA) sono le grammatiche regolari ed espressioni regolari.

**Algoritmo per convertire un DFA a una grammatica regolare** È sempre possibile farlo. I non terminali della grammatica sono stati dell'automa.

- Se $q \xrightarrow{a} q'$ nell'automa e $q'$ non è finale, allora $Q \to aQ'$ è nella grammatica.
- Se $q \xrightarrow{a} q'$ nell'automa e $q'$ è finale, allora $Q \to aQ' \mid a$ è nella grammatica.
- Se $q$ è iniziale e finale, allora $Q \to \varepsilon$ è nella grammatica.

In ANTLR $\varepsilon$ si traduce in "spazio bianco". `stm : 'a' | ε;` viene usato `stm : 'a' | ;`.

Un'implementazione del lexer può essere fatta usando una matrice in cui una dimensione descrive gli stati e l'altra i simboli di input. Per ogni transazione $S_i \xrightarrow{a} S_k$ basta definire `T[i, a] = k`. Se l'automa è nello stato $s_i$ e il carattere è `'a'`, allora legge `T[i, a] = k` e salta a $s_k$.

```
index = 0
state = s1
state = T[S1, input[0]]

if (isfinal(state)) ...
```

Un lexer legge un carattere per volta finché non trova una regola e in tal caso ritorna un token. Visto che si usa la regola longest-match, con degli esempi del tipo

```
SHORTTOKEN : 'abc';
LONGTOKEN  : 'abcabc';
FUFFA      : 'abc';
ID         : (CHAR)+;
```

avremmo

```
abcab
SHORTOKEN ERROR 'a' ERROR 'b'

---

abcabc
LONGTOKEN
```

il `FUFFA` non verrà mai riconosciuto come token, dato che `abc` è già riconosciuto da `SHORTTOKEN`. L'`ID` non riconosce `abc` e `abcabc`.

Un problema del maximal match rule si vede con le regole

```
SHORT: 'aaa';
LONG:  'aaaa';
```

e l'input `aaaaaa` il quale ritorna `LONG ERROR 'a' ERROR 'a'`.

Ad ogni terminale corrisponde un token lessicale nell'albero lessicale. Non è comodo per ogni carattere, dunque, qualcosa del tipo

```
fragment CHAR : 'a'..'z' | 'A'..'Z';
```

è "ausiliario" così da non creare nodi per ogni carattere della stringa.

L'analizzatore lessicale usa espressioni o grammatiche regolari, non context-free. Non è il caso di ANTLR4 che permette l'uso di grammatiche context-free.

```
start : (BINDIGIT PIU DIGIT)+ ;
PIU : '+' ;
BINDIGIT : ('0' | '1')+ ;
DIGIT : ('0'..'9')+ ;
WS : (' ' | '\t' | '\n' | '\r' ) -> skip
```

`1+1` dà errore a causa del first match rule con `BINDIGIT`.

Ora, si vede però come la grammatica

```
init : token (',' token)*;
token : 'a'token'b' | 'ab';
```

restituisce

```
aabb
(init:1 (token:1 a (token:2 ab) b))
```

ma la grammatica context free è ancora valida e

```
init : TOKEN (',' TOKEN)*;
TOKEN : 'a'TOKEN'b' | 'ab';
```

```
aabb
(init:1 aabb)
```

Questa analisi riesce a risolvere problemi lessicali del tipo `mainf` usato al posto di `main`.

Cosa importante che CL vuole in tutte le grammatiche:

```
WS : (' '|'\t'|'\n'|'\r')-> skip;
```

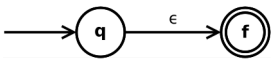
## 2.1. Algoritmo McNaughton–Yamada–Thompson

Usata per convertire un'espressione regolare in NFA.

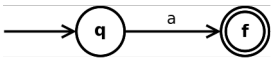
Partendo da uno stato $s$ si definisce l'insieme $\varepsilon$-closure($s$) come l'insieme raggiungibile da tutti gli stati dell'NFA che raggiungono lo stato $s$ solo mediante transizioni $\varepsilon$.

La conversione può avvenire strutturalmente.

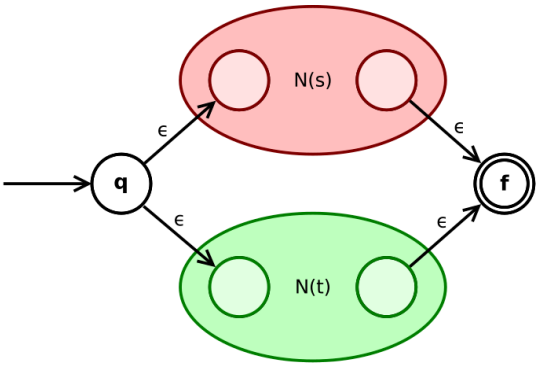
- espressione $\varepsilon$



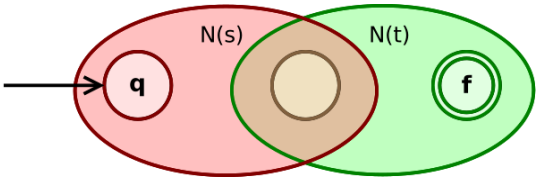

- simbolo $a$



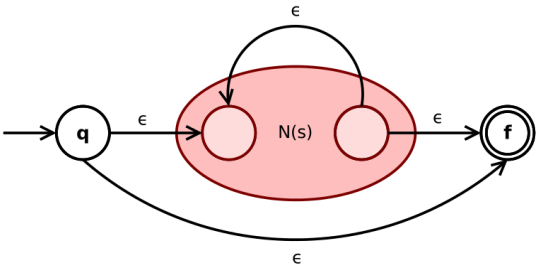

- $r = s|t$




- $r = st$




- $r = s^*$




## 2.2. Minimizzazione del DFA

Per minimizzare di un DFA si intende creare un DFA equivalente con un numero minimo di stati. Per far ciò:

1. Si rimuovono gli stati morti (nessuno stato finale è raggiungibile) e non raggiungibili.
2. Si fondono gli stati non distinguibili (una stringa può fare match con più stati).

L'algoritmo per fare il punto 2 è quello di Hopcroft.

# 3. Analisi sintattica

Si può costruire, data una grammatica, un albero sintattico (parse tree) tale che:

- la radice è il simbolo iniziale.
- le foglie sono token lessicali (o $\varepsilon$).
- i nodi interni sono non-terminali.

Data la grammatica context-free

```
exp : exp - exp | digit;
digit : DIGI;
DIGI : ('0'..'9')+
```

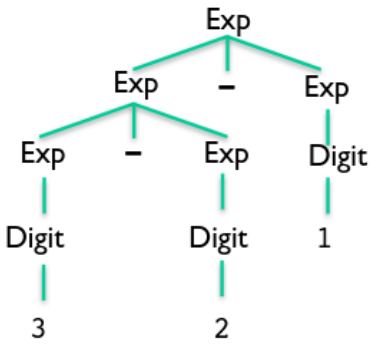
un'esecuzione del tipo

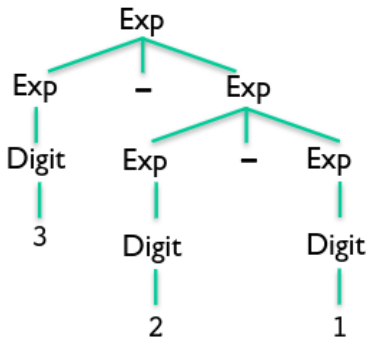
```
exp -> exp - exp
    -> exp - exp - exp
    -> digit - exp - exp
    -> 3 - digit - exp
    -> 3 - 2 - exp
    -> 3 - 2 - digit
    -> 3 - 2 - 1
```

è diversa da

```
exp -> exp - exp
    -> digit - exp
    -> 3 - exp
    -> 3 - exp - exp
    -> 3 - digit - exp
    -> 3 - 2 - exp
    -> 3 - 2 - digit
    -> 3 - 2 - 1
```

perché restituiscono due alberi sintattici diversi .



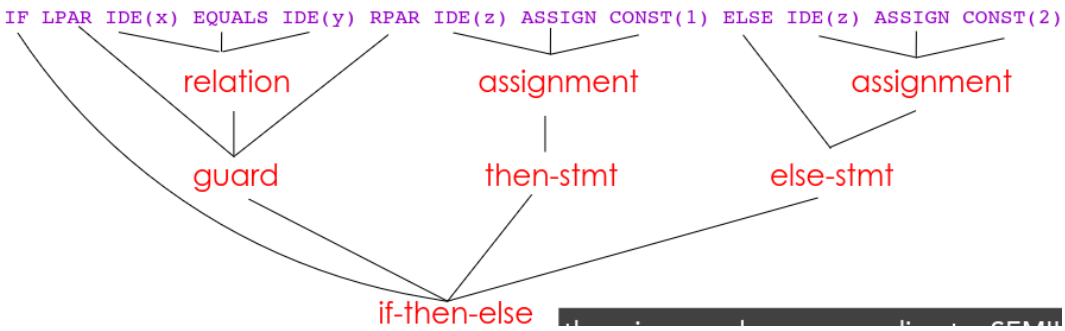

5

L'associatività, in genere, è a sinistra. Quindi `3-2-1` dovrebbe essere `(3 - 2) - 1`.

Se dalla grammatica $G$ la stringa presa in $\mathcal{L}(G)$ ha più derivazioni leftmost (o ha più alberi sintattici) diciamo che $G$ è **ambigua**.

La funzione di parsing prende una sequenza di token in input e ritorna un'AST (abstract syntax tree).

Il parse tree di `if (x == y) z = 1; else z = 2;` potrebbe essere (senza il `;` perché non c'era spazio):




si vede come la sequenza dei token è più verbosa rispetto all'AST.

Presa una grammatica

```
bexp : '(' bexp ')' | DIGIT;
DIGIT : ('0'..'9')+;
```

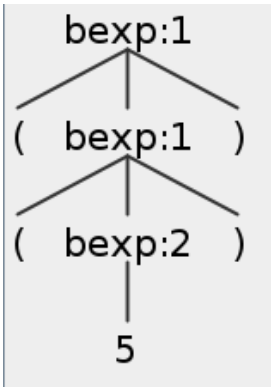
un parser potrebbe essere

```
# input ((5))
#       ^
#       | next

def bexp:
    if input[next] == LPAR:
        next += 1
        if bexp():
            if input[next] == RPAR:
                return 1
            else:
                raise SyntaxError
```

un DFA non potrebbe descrivere quella grammatica.

Il parse tree non servirebbe in questo caso ma serve per la generazione del codice successivo-project. Questo perché con quell'input `((5))` si genera qualcosa ricorsivamente (il lexer invece ritorna `LPAR LPAR DIGIT RPAR RPAR`).




Quindi il flow è: (1) lexer prende sequenza di caratteri e genera sequenza di token, (2) parser prende sequenza di token e genera AST.

## 3.1. Ricorsione a sinistra

Una grammatica è ricorsiva a sinistra se ha un non terminale $A$ t.c. $A \to^+ A\gamma$ per qualche $\gamma$.

Può essere una cosa del tipo

$$A \to A\gamma_1 \mid ... \mid A\gamma_m \mid \delta_1 \mid ... \mid \delta_n$$

che è equivalente a

$$(\delta_1 \mid ... \mid \delta_n)(\gamma_1 \mid ... \mid \gamma_m)^*$$

Per trasformare $A \to A\gamma_1$ in maniera non ricorsiva a sinistra si può introdurre un nuovo non terminale $A'$ e aggiungere

$A \to \delta_1 A' \mid ... \mid \delta_n A'$

$A' \to \gamma_1 A' \mid ... \mid \gamma_m A' \mid \varepsilon$

Ad esempio

$E \to E + F$ diventa

$E \to FE'$

$E' \to +FE'$

La **ricorsione a sinistra indiretta** si mostra in maniera diversa.

Può essere del tipo

$$A_1 \to A_2\gamma_1$$

$$A_2 \to A_3\gamma_2$$

$$\vdots$$

$$A_{k_1} \to A_k\gamma_{k-1}$$

$$A_k \to A_1\gamma_k$$

oppure

$$A \to \gamma A \delta$$

con NULLABLE($\gamma$).

## 3.2. Fattorizzazione a sinistra

Presa ad esempio

$$E \to T + E \mid T$$

$$T \to (E) \mid (E) * T \mid \text{int} \mid \text{int} * T$$

vediamo come per fattorizzarla si può prendere la più grande sottosequenza simile per le regole e aggiungere un simbolo ausiliario. Ad esempio, per $E$ si vede che la più grande sottosequenza simile è "T".

$$E \to TE'$$

$$E' \to +E \mid \varepsilon$$

Per $T$ vi sono due sottosequenze simili.

$$T \to (E)T' \mid \text{int } T'$$

$$T' \to * T \mid \varepsilon$$

È risolta automaticamente da ANTLR.

## 3.3. Parsing a discesa ricorsiva

**Formalmente**, una grammatica $(N, T, \to, S)$ è ricorsiva a sinistra se $\exists A \in N$ t.c. $A \to^+ A\gamma$ per qualche $\gamma$.

Si analizzano i token e si prova a la ricostruzione con derivazione left-most. Sono chiamati parser **top-down** perché somiglia ad una visita su un albero sintattico.

Un non terminale $NT$ ha un metodo che riconosce $NT$, ogniuna differita da costrutto di selezione `if-else` o `case`.

Ad esempio, per la grammatica

```
E -> T + E | T
T -> (E) | (E) * T | int(DIGI) | int(DIGI) * T
```

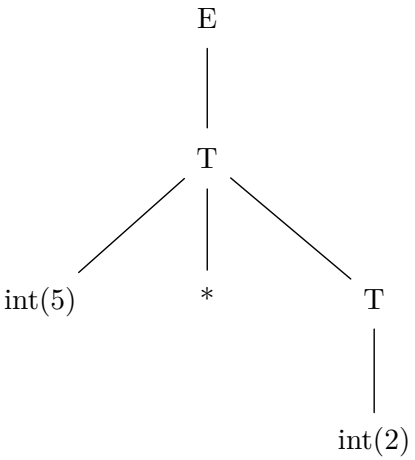
non si ha occorrenza per input

```
int(5) * int(2)
```

perché i passi di analisi sono:

1. Prova `E -> T + E`.
   - Prova `T -> (E)`: nessuna occorrenza per `int(5)`, skip.
   - Prova `T -> (E) * T`: nessuna occorrenza per `int(5)`, skip.
   - Prova `T -> int(DIGI)`: trovata occorrenza per `int(5)` ma non trovata occorrenza per `* int(2)`, dato che la regola da cui siamo partiti si aspetta un `+ E`, skip.
   - Prova `T -> int(DIG) * T`: trovata occorrenza per `int(5)` e anche per `* int(2)`, ma non trova occorrenza per `+ E` dato che i token sono finiti, skip.
2. Prova `E -> T`.
   - Prova `T -> (E)`: nessuna occorrenza per `int(5)`, skip.
   - Prova `T -> (E) * T`: nessuna occorrenza per `int(5)`, skip.
   - Prova `T -> int(DIGI)`: trovata occorrenza per `int(5)` ma non trovata occorrenza per `* int(2)`, skip.
   - Prova `T -> int(DIG) * T`: trovata occorrenza per `int(5)` e anche per `* int(2)`, accetta.

Quest'ultimo crea un albero del tipo



Questa tipologia di costruzione non funziona per grammatiche LR perché vi sarebbe un ciclo infinito.

## 3.4. Parser LR

La sequenza di token viene espansa a partire dalla derivazione più a sinistra.

Vi è un problema con le grammatiche ricorsive a sinistra **LR** come `S -> S a` in cui il processo va in ciclo infinito. Una grammatica ricorsiva a sinistra è del tipo $A \rightarrow^{+} A\gamma$, con $\gamma$ generica sequenza di simboli.

La ricorsione avviene anche per un backtrack del tipo `T -> (E) | (E) + NUM`.

Usare la fattorizzazione a sinistra permette di ovviare questo problema di ricorsione a sinistra, ma poi bisogna tener conto delle valutazioni a sinistra dell'espressione. Preso ad esempio l'ordine delle produzioni `T -> int * T | int` e `T -> int | int * T` vediamo come non siano intercambiabili perché il secondo non riconoscerebbe mai qualcosa del tipo `int * int`.

## 3.5. Parser LL(1)

Il nome sta per $L$ = left-to-right input scan, $L$ = derivazione leftmost, $1$ = predice usando $k$ token di lookahead. Quindi per ogni non-terminale e input token c'è al massimo 1 produzione che può essere usata.

In ANTLR si usa un parser LL(*) che usa un tot di token di lookahead.

I parser LL(1) possono essere definiti usando una tabella in cui la 1^ è per i non terminali e la 2^ per il next-token.

Ad esempio, per la grammatica

```
E -> T X
T -> (E) Y | int Y
X -> + E | ε
Y -> * T | ε
```

si ha

| | 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` |

| Stack | Input | Azione |
|---|---|---|
| `E $` | `int * int $` | `T X` |
| `T X $` | `int * int $` | `int Y` |
| `int Y X $` | `int * int $` | `<terminal>` |
| `Y X $` | `* int $` | `* T` |
| `* T X $` | `* int $` | `<terminal>` |
| `T X $` | `int $` | `int Y` |
| `int Y X $` | `int $` | `<terminal>` |
| `Y X $` | `$` | `ε` |

| Stack | Input | Azione |
|---|---|---|
| `X $` | `$` | `ε` |
| `$` | `$` | `<terminal>/ACCEPT` |

ad esempio, per `[E, int]` quando il non terminale presente in cima allo stack è `E` e il next-token è `int` allora si usa la produzione `E -> T X`.

La tecnica per fare parsing della tabella è: per ogni non terminale `S` vedere se il next-token `a` è presente guardando `[S, a]`. Si usa uno stack per tenere traccia dei terminali / non-terminali per le produzioni a destra. Per `[S, a]` l'input è accettato quando l'entry contiene il token EOF (`$`).

Formalmente si ha la grammatica $G = (N, T, \to, S)$ in cui la sua tabella LL(1) è definta come:
1. $N$ come righe, $T$ come colonne.
2. Per ogni regola $X \to \gamma$ e per $t$ t.c $\gamma \to^* t\gamma$ si aggiunge la regola $X \to \gamma$ nella entry `[X, t]`.
3. Per ogni regola $X \to \gamma$ t.c $\gamma \to^* \varepsilon$ si aggiunge la regola $X \to \gamma$ nella entry `[X, t]` per ogni $t$ t.c. $S \to^* \gamma X t \gamma'$.

### 3.5.1. Insieme `NULLABLE`

Presa una grammatica context-free $G = (N, T, \to, S)$ vediamo che `NULLABLE` è la funzione def. su $G$ t.c.

$$\text{NULLABLE}(G) = \{A \mid A \to^* \varepsilon\}$$

con $\text{NULLABLE}(G) \subseteq N$.

Ad esempio, per la grammatica

```
E -> T X
T -> (E) Y | int Y
X -> + E | ε
Y -> * T | ε
```

si ha $\text{NULLABLE}(G) = \{X, Y, T, E\}$.

Si può usare la teoria del punto fisso per calcolare l'insieme. dato che

$$\text{NULLABLE}_0(G) = \{A \mid A \to \varepsilon\}$$

$$\text{NULLABLE}_{i+1}(G) = \text{NULLABLE}_i(G) \cup \{A \mid A \to A_1, ..., A_n \wedge A_1, ...A_n \in \text{NULLABLE}_i(G)\}$$

e si vede che $\text{NULLABLE}_i(G) \subseteq \text{NULLABLE}_{i+1}(G) \subseteq N$ e dunque $\exists k$ t.c. $\text{NULLABLE}_k(G) = \text{NULLABLE}_{k+1}(G)$.

Un modo più semplice per definirlo è:

$$\text{NULLABLE}(\varepsilon) = \text{True}$$

$$\text{NULLABLE}(t) = \text{False}, t \in T$$

$$\text{NULLABLE}(\alpha\gamma) = \begin{cases} \text{False} & \text{NULLABLE}(\alpha) = \text{False} \\ \text{NULLABLE}(\gamma) & \text{NULLABLE}(\alpha) = \text{True} \end{cases}$$

$$\text{NULLABLE}(x) = \bigvee_{x \to \gamma \text{ in } G} \text{NULLABLE}(\gamma)$$

### 3.5.2. Insieme `FIRST`

Presa una grammatica context-free $G = (N, T, \to, S)$ vediamo che `FIRST` è la funzione def. su $N \cup T$ t.c.

$$\text{FIRST}_i(t) = \{t\}, t \in T$$

$$\text{FIRST}_0(A) = \begin{cases} \{e\} \text{ if } A \in \text{NULLABLE}(G) \\ \emptyset \quad \text{if } A \notin \text{NULLABLE } (G) \wedge A \in N \end{cases}$$

$$\text{FIRST}_{i+1}(A) = \text{FIRST}_i(A) \cup \bigcup_{\substack{A \to \alpha_1 \dots \alpha_n \\ \forall i \in 1 \dots k-1: \\ \alpha_i \in \text{NULLABLE}(G)}} \text{FIRST}(\alpha_k) \setminus \{\varepsilon\}$$

Un modo più semplice per definirlo è:

$$\text{FIRST}(\varepsilon) = \emptyset$$

$$\text{FIRST}(t) = \{t\}, t \in T$$

$$\text{FIRST}(\alpha\gamma) = \begin{cases} \text{FIRST}(\alpha) & \text{NULLABLE}(\alpha) = \text{False} \\ \text{FIRST}(\alpha) \setminus \{\varepsilon\} \cup \text{FIRST}(\gamma) & \text{NULLABLE}(a) = \text{True} \end{cases}$$

Anche qui può essere usata la teoria del punto fisso vista sopra.

Ad esempio, per la grammatica

```
E -> T X
T -> (E) Y | int Y
X -> + E | ε
Y -> * T | ε
```

| | X | Y | E | T |
|---|---|---|---|---|
| $\text{FIRST}_0$ | {ε} | {ε} | ∅ | ∅ |
| $\text{FIRST}_1$ | {+, ε} | {*, ε} | ∅ | {(, int)} |
| $\text{FIRST}_2$ | {+, ε} | {*, ε} | {(, int} | {(, int} |

si estende $\text{FIRST}(\gamma)$ con $\gamma \in (N \cup T)^\star$ t.c.

$$\text{FIRST}(\varepsilon) = \{\varepsilon\}$$

$$\text{FIRST}(t\gamma) = \{t\}, t \in T$$

$$\text{FIRST}(A\gamma) = \text{FIRST}(A), A \notin \text{NULLABLE}(G)$$

$$\text{FIRST}(A\gamma) = \text{FIRST}(A) \setminus \{\varepsilon\} \cup \text{FIRST}(\gamma), A \in \text{NULLABLE}(G)$$

### 3.5.3. Insieme `FOLLOW`

Presa una grammatica context-free $G = (N, T, \to, S)$ vediamo che `FOLLOW` è la funzione def. su $N$ t.c.

$$\text{FOLLOW}_0(S) = \{\$\}$$

$$\text{FOLLOW}_0(A) = \emptyset$$

$$\text{FOLLOW}_{i+1}(X) = \text{FOLLOW}_i(X) \cup \bigcup_{Z \to \delta X \gamma} \text{FIRST}(\gamma) \setminus \{\varepsilon\} \cup \bigcup_{\substack{Z \to \delta X \gamma \\ \wedge \\ \text{NULLABLE}(\gamma)}} \text{FOLLOW}_i(Z)$$

`FOLLOW` non contiene mai $\varepsilon$.

Ad esempio, per la grammatica

```
E -> T X
T -> (E) Y | int Y
X -> + E | ε
Y -> * T | ε
```

| | E | Y | X | T |
|---|---|---|---|---|
| $\text{FOLLOW}_0$ | {$} | ∅ | ∅ | ∅ |
| $\text{FOLLOW}_1$ | {$, )} | {+, $} | {$} | {+, $} |
| $\text{FOLLOW}_2$ | {$, )} | {+, $, )} | {$, )} | {+, $} |
| $\text{FOLLOW}_2$ | {$, )} | {+, $, )} | {$, )} | {+, $, )} |

### 3.5.4. Tabella di parsing

Per una grammatica context-free $G$ si def. $\text{LL}^1_G$ per $G$ in cui per ogni $A \to \alpha$ si

- Per ogni terminale

  $t \in \text{FIRST}(\alpha) \Rightarrow \text{LL}^1_G[A, t] = A \to \alpha$
- Se $\varepsilon \in \text{FIRST}(\alpha)$ per ogni

  $t \in \text{FOLLOW}(A) \Rightarrow \text{LL}^1_G[A, t] = A \to \alpha$

L'ultima regola si applica anche per `$`.

Se

$$\varepsilon \in \text{FIRST}(\alpha) \wedge \$ \in \text{FOLLOW}(A) \Rightarrow \text{LL}^1_G[A, \$] = A \to \alpha$$

Se un entry è definita più volte in $G$ allora non è LL(1). Questo si ha anche quando $G$ è ricorsiva a sx oppure non fattorizzata a sinistra oppure ambigua.ric

# 4. Analisi semantica

Si vorrebbe verificare che un identificatore è definito (o se lo è più volte): questo non riguarda in merito al tipaggio. Si può dividere in due fasi:
1. check semantico (scope e tabella dei simboli).
2. type checking.

Ad esempio, nel codice seguente

```
class MyClass implements MyInterface {
  string myInteger;
  void doSomething() {
    int[] x = new string;
    x[5] = myInteger * y;
  }
  void doSomething() {
  }
  int fibonacci(int n) {
    return doSomething() + fibonacci(n - 1);
  }
}
```

vi sono alcuni errori semantici:
- in `fibonacci` viene ritornato il valore di una funzione che ritorna `void`.
- è definita una moltiplicazione tra stringhe e `y` (ma `y` non è manco definita).
- `x` è un `int[]` quindi non può essere inizializzato con una stringa.
- `doSomething()` è ridefinita (questo dipende dal linguaggio però).

che però, a livello *sintattico*, non sono errori.

Il check semantico (scope checking) attraversa l'AST e:
- processa le dichiarazioni (famo mediante HashMap) che inserisce la chiave nella tabella dei simboli o segnala multiple dichiarazioni.
- processa i blocchi che va ad aggiornare l'AST in modo da puntare alla tabella dei simboli aggiornata.

Il type checking va ad aggiornare il nodo dell'AST in modo da tener traccia del tipo e dell'offset all'interno dello stack (piuttosto dell'intera tabella di simboli).

## 4.1. Scope checking

### 4.1.1. Symbol table

Una tabella dei simboli che visualmente potrebbe essere

| **key** | **value** |
|---|---|
| $X$ | $T_x$ |
| $Y$ | $T_y$ |
| $Z$ | $T_z$ |

che può essere rappresentanta, dunque, mediante una HashMap $H$. Il dominio degli identificatori $ID$ è finito. La funzione $\Gamma$ è definita, usando un oggetto di tipo STentry (così l'ha chiamata sul codice).

$$\Gamma : ID \longrightarrow \text{STentry}$$

$\Gamma$ è detta **ambiente**.

Le operazioni sono:
- $\emptyset$
- $\Gamma[x \mapsto T]$ che aggiorna un elemento di chiave $x$ di valore $T$.
- $\Gamma(a)$ in cui $a$ è una chiave di cui vogliamo prender eil valore.

Ad esempio, se abbiamo già

$\Gamma = [a \mapsto T_a, b \mapsto T_b]$

e volessimo inserire il primo elemento della lista, avremmo

$\Gamma[x \mapsto T_x] = [a \mapsto T_a, b \mapsto T_b, x \mapsto T_x]$

Un esempio su un codice C, che però ignora gli identificatori delle funzioni, è:

```c
int x = 137;
int z = 42;
int foo(int x, int y) {
  printf("%d %d %d", x, y, z);
  {
    int x, z;
    z = y;
    x = z;
    {
      int y = x;
      printf("%d %d %d", x, y, z);
    }
  }
}
```

si vede come vi sia un offset tra i dati di `y` e quelli espressi dentro la funzione `foo` e poi così via ogni volta si cambia scope. La tabella rappresenta la coppia (ID, lina nel codice).

| | |
|---|---|
| x | 1 |
| y | 2 |
| | |
| x | 3 |
| y | 3 |
| | |

| | |
|---|---|
| x | 6 |
| z | 6 |
| | |
| y | 10 |

Possono essere viste come diverse HashMap collegate fra di loro. A partire dal $H_4$ che ha solo $(y, T''_y)$ che punta alla symbol table $H_3$ con $((x, T''_x), (z, T'_z))$ e così via.

Una nuova variabile controlla a partire da $H_4$ se è presente. Se non lo è la si aggiunge in $H_4$, sennò genera errore di inizializzazione multipla (o si fa overloading eh, dipende).

Il codice riscritto che punta alle varie hash map è:

```
int x = 137;
int z = 42;
int foo(int x, int y) {
  printf("%d %d %d", x@3, y@3, z@2);
  {
    int x, z;
    z@6 = y@3;
    x@6 = z@6;
    {
      int y = x@6;
      printf("%d %d %d", x@6, y@10, z@6);
    }
  }
}
```

dove la `<x>` di `@<x>` rappresenta la linea di codice a cui riferisce quella variabile.

L'operazione di **lookup**, formalmente, viene definita come:

$$\Gamma_1 \cdot \Gamma_2 \cdot \Gamma_3(x) = \begin{cases} \Gamma_3(x) & \text{if } x \in \text{dom}(\Gamma_3) \\ \\ \Gamma_1 \cdot \Gamma_2(x) & \text{else} \end{cases}$$

che è $\Gamma(a)$ dove $a$ è l'ID che vogliamo cercare.

Si abusa la notazione in modo da avere

$$\underbrace{\Gamma_1 \cdot \Gamma_2 \cdot \Gamma_3}_{\Gamma \cdot \Gamma'}$$

in cui $\Gamma$ è sequenza e $\Gamma'$ ambiente di testa.

*Esempio*

$\Gamma = [x \mapsto \text{int}, y \mapsto \text{bool}]$

$$\Gamma[z \mapsto \text{string}] = [x \mapsto \text{int}, y \mapsto \text{bool}, z \mapsto \text{string}]$$

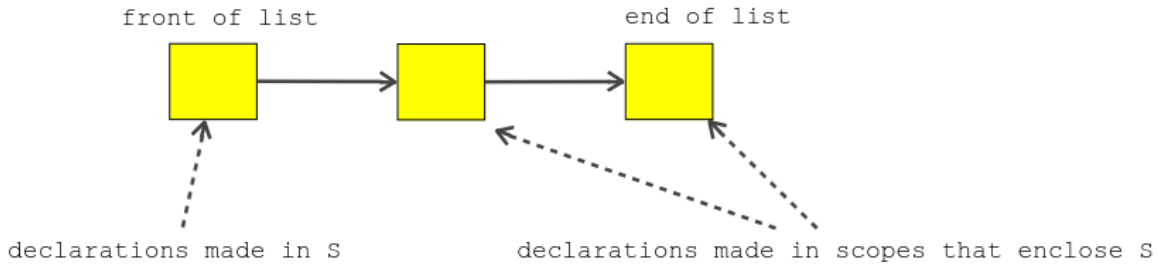
$$\Gamma[z \mapsto \text{string}](a) = \begin{cases} \text{string if } a{=}z \\ \\ \Gamma(z) \quad \text{else} \end{cases}$$

Vedendolo in un modo diverso che mette in paragone la symbol table con l'ambiente $\Gamma$, le operazioni che devono essere implementate nel pratico sono 4:

| Tabella dei simboli | Environment $\Gamma$ |
|---|---|
| add | $\Gamma[x \mapsto T]$ |
| lookup | $\Gamma(x)$ |
| scope entry | $\Gamma \cdot \emptyset \equiv \Gamma \cdot []$ |
| scope exit | $\Gamma \cdot \Gamma' \Rightarrow \Gamma$ |

### 4.1.2. Implementazione con lista di hash table

Assumiamo che la lunghezza della lista è il numero di suo annidamento.



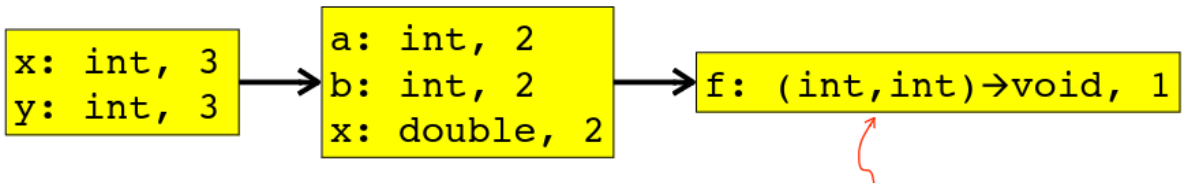

*Esempio*

Preso il codice

```
void f(int a, int b) {
  double x;
  while (...) { int x, y; ... ###}
}

void g() { f(4,5); }
```

la symbol table nel punto `###` sarà composta in modo tale che l'ambiente `g()` non è ancora stato creato, dunque si avrà




Con

$\Gamma_1 \cdot \Gamma_2 \cdot \Gamma_3(x) = \Gamma_3(x)$

$\Gamma_1 \cdot \Gamma_2 \cdot \Gamma_3(a) = \Gamma_2(a)$

$\Gamma_1 \cdot \Gamma_2 \cdot \Gamma_3(g) = g$ is not defined

Ricordando l'uso di due visite dell'albero sintattico, vediamo come i prototipi evitano l'impiego di questa seconda visita. In realtà, questa seconda visita riesce a ricostruire da sola i tipi.

Si potrebbe provare a fare tutto in una sola visita ma si dovrebbe fare inferenza di tipi.

Nell'esempio sopra vi è la possibilità di fare ricorsione dato che `f` è in scope. Disabilitare la ricorsione potrebbe essere fatta aggiungendo l'ambiente `f: (int, int) -> void, 1` solo alla fine dello scope; questo potrebbe generare un problema di `f` definita più volte.

| | |
|---|---|
| add | si fa lookup di x e se non c'è allora aggiunge x alla prima tabella in lista. $O(1)$ |
| lookup | si cerca dalla prima lista e così via. $O(depth)$ |
| scope entry | incrementa il livello corrente e aggiunge un hashtable vuota all'inizio della lista |
| scope exit | rimuove la prima tabella in lista e decrementa il livello corrente. $O(1)$ |

In Java i parametri locali e globali si trovano nella medesima hash table

```
void g(int x, int a) { }
void f(int x, int y) { int a, x; }
```

dunque, per `f`, si avrebbe un'unica tabella del tipo

| | | |
|---|---|---|
| x | int | 2 |
| y | int | 2 |
| a | int | 2 |
| x | int | 2 |

### 4.1.3. Implementazione con hash table di liste

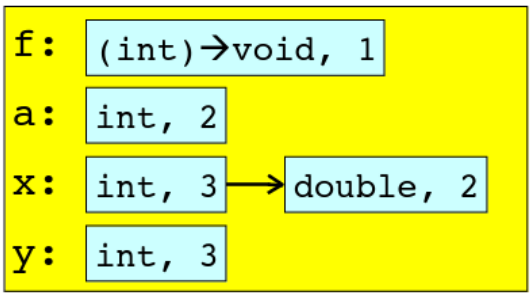
Ogni scope ha associata una lista di valori di tabella di simboli, usando il livello di nesting.

*Esempio*

Preso il precedente codice

```
void f(int a) {
  double x;
  while (...) { int x, y; ### }
  void g() { f(); }
}
```

la tabella di simboli nel punto ### diviene




In cui il numero dopo la virgola è il livello di nesting.

Le operazioni che vengono svolte, in modo analogo all'altra implementazione, sono:

| | |
|---|---|
| add | si fa lookup di x e se non c'è allora, una volta preso il valore corrente, si vede se è = al numero corrente: se sì, allora dà errore. Altrimenti lo aggiunge in fronte alla lista con il valore corrente. $O(1)$ |
| lookup | si cerca il valore nella symbol table. $O(1)$ |
| scope entry | si incrementa il numero di livello corrente. $O(1)$ |
| scope exit | cerca tutti i valori della tabella dei simboli dalla prima, se il livello dell'elemento corrente è = al livelo corrente, allora lo rimuove dalla lista. Alla fine0 decrementa il numero di livello corrente. |

*Esempio*

```
void g(int x, int a) {
  double d;
  while (...) {
    int d, w;
    double x, b;
    if (...) { int a,b,c; }
  }
  while (...) { int x,y,z; }
}
```

Si ha la tabella

```
a: [int, 4] -> [int, 2]
b: [int, 4] -> [double, 3]
c: [int, 4]
d: [int, 3] -> [double, 2]
g: [int -> int -> void, 1]
x: [int, 3] -> [int, 2]
w: [int, 3]
y: [int, 3]
z: [int, 3]
```

## 4.2. Type checking

Un tipo è un insieme di valori o di operazioni su tali valori. Diciamo che un linguaggio è type-safe se le uniche operazioni che possono essere svolte sono quelle su quei tipi. (Ad esempio la somma può essere fatta solo per i numeri).

Un sitema di tipi è un sistema formale contenente un insieme di regole che assegna una proprietà (un tipo) ad un'operazione. Questo sistema può essere svolto a tempo di compilazione (staticamente) o durante l'esecuzione (dinamicamente). Vi è anche un concetto di "untyped" in cui vi è una combinazione di essi.

Il type checking è il processo di verifica di tali operazioni. Anch'esso viene fatto in maniera statica o dinamica. Gli errori di tipo sorgono quando le operazioni non sono supportate su tali valori.

Un esempio di formalismo sull'and è

$$\frac{\vdash e_1 : \text{bool} \qquad \vdash e_2 : \text{bool}}{\vdash e_1 \ \&\& \ e_2 : \text{bool}}$$

Dato che formalmente si ha la regola descritta come

$$\frac{J_1 \ldots J_n}{J}$$

Ci si aspetta una regola per ogni produzione $J_i$ a meno di ottimizzazioni.

Il tipo di una variabile viene riconosciuta in base al contesto $\Gamma$. Quindi si scrive nella maniera $\Gamma \vdash e : \text{bool}$.

Ad esempio, per la grammatica

```
exp : NUM | ID | 'true' | 'false' | expr '+' expr | exp '==' expr;
```

si ha un assioma (non judgement) per la variabile.

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

e poi in seguito

$$[\text{Num}]\ \frac{}{\Gamma \vdash \text{NUM} : \text{int}}$$

$$[\text{True}]\ \frac{}{\Gamma \vdash \text{true} : \text{bool}}$$

$$[\text{False}]\ \frac{}{\Gamma \vdash \text{false} : \text{bool}}$$

$$[\text{Plus}]\ \frac{\Gamma \vdash e_1 : \text{int} \qquad \Gamma \vdash e_2 : \text{int} \qquad + : \text{int} \times \text{int} \to \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

$$[\text{Eq}]\ \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad T_1 = T_2 \qquad ==: T_1 \times T_1 \to \text{bool}}{\Gamma \vdash e_1 == e_2 : \text{bool}}$$

Potremmo anche estendere tale gramamatica in modo da supportare altre operazioni e dunque avere anche:

$$[\text{If}]\ \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad \Gamma \vdash e_3 : T_3 \qquad T_1 = \text{bool} \qquad T_2 = T_3}{\Gamma \vdash \text{if } e_1\{e_2\} \text{ else } \{e_3\} : T_2}$$

$$[\text{Invk}]\ \frac{\Gamma \vdash f : T_1 \times ...T_n \to T \qquad (\Gamma \vdash e_i : T_i')^{i \in 1..n} \qquad (T_i = T_i')^{i \in 1..n}}{\Gamma \vdash f(e_1, ..., e_n) : T}$$

$$[\text{VarD}]\ \frac{\Gamma \vdash e : T' \qquad x \notin \text{dom}(\text{top}(\Gamma)) \qquad T = T'}{\Gamma \vdash Tx = e; : \Gamma[x \mapsto T]}$$

$$[\text{SeqD}]\ \frac{\Gamma \vdash d : \Gamma' \qquad \Gamma' \vdash D : \Gamma''}{\Gamma \vdash dD : \Gamma''}$$

Quando viene richiesta la costruzione di un sistema di tipi per un'espressione tipo `(x+5) == (y+2)` bisogna costruire l'**albero di prova**: albero finito in cui i nodi sono istance di regole di inferenze; le foglie sono istanze di assiomi; la radice dell'albero contiene il giudizio che deve essere dimostrato (l'albero è visto al contrario).

Bisogna derivare, in questo caso, $\Gamma \vdash (x + 5) == (y + 2) : \text{bool}$ avendo come contesto $\Gamma = [x \mapsto \text{int}, y \mapsto \text{int}]$

$$[\text{Eq}]\ \frac{[\text{Plus}]\ \dfrac{[\text{Var}]\ \dfrac{\Gamma(x) = \text{int}}{\Gamma \vdash x : \text{int}} \quad \dfrac{}{\Gamma \vdash 5 : \text{int}}}{\Gamma \vdash (x + 5) : \text{int}} \quad [\text{Plus}]\ \dfrac{[\text{Var}]\ \dfrac{\Gamma(y) = \text{int}}{\Gamma \vdash y : \text{int}} \quad \dfrac{}{\Gamma \vdash 2 : \text{int}}}{\Gamma \vdash (y + 2) : \text{int}} \qquad + : \text{int} \times \text{int} \to \text{int}}{==: \text{int} \times \text{int} \to \text{bool}}}{\Gamma \vdash (x + 5) == (y + 2) : \text{bool}}$$

Paragonando il contesto alla tabella dei simboli si vede come l'operazione di lookup in $\Gamma$ è $\Gamma(\text{id})$; l'operazione di inserimento ed estenzione di un nuovo identificatore id è $\Gamma[\text{id} \mapsto \text{''<type>''}]$.

A livello di codice si potrebbe avere qualcosa del tipo

```
type typeChecking(SymbolTable ST, ExprNode e) {
  switch(e) {
    case num: return int;
    case "true": case "false": return bool;
    case id: type t = ST(id);
            if (t == "<unbound>") "error";
            return t;
    case e1 + e2:
            type t1 = typeChecking(ST, e1);
            type t2 = typeChecking(ST, e2);
            if ((t1 == int) && (t2 == int)) return int;

            "error";
    case e1 == e2:
            type t1 = typeChecking(ST, e1);
            type t2 = typeChecking(ST, e2);
            if (t1 == t2) return bool;

            "error";
    case if e1 { e2 } else { e3 }:
            type t1 = typeChecking(ST, e1);
            type t2 = typeChecking(ST, e2);
            type t3 = typeChecking(ST, e3);
            if (t1 == bool && t2 == t3) return t2;

            "error"
    case id(elist):
            type t = ST.lookup(id);
            switch(t) {
              case "<unbound>": "error";
              case (t1, ..., tn) -> t0:
              [t1', ..., tn'] = typeCheckingTuple(ST, elist);
              if (n == m && t1 == t1' && ... && tn == tn') return t0;

              "error"
            }
  }
}
tuple_type typeCheckingTuple(SymbolTable ST, ExpNodeList L) {
  match L with
    | nil -> []
    | e -> [typeChecking(ST, e)]
    | e :: l1 -> typeChecking(ST, e) :: typeCheckingTuple(ST, l1)
}
```

Una regola per una funzione pu essere definita come (preso da SimpLan):

$$[\text{Fun}]\ \frac{\Gamma \cdot [x_1 \mapsto T_1, ..., x_n \mapsto T_n] \vdash e : T' \qquad T' = T \qquad f \notin \text{dom}(\text{top}(\Gamma))}{\Gamma \vdash T f(T_1 x_1, ..., T_n x_n) = e; : \Gamma[f \mapsto (T_1, ..., T_n) \to T]}$$

Per ammettere la ricorsione bisogna estendere il contesto iniziale, e quindi come se ci fossero "2 passate" nel type checking.

$$[\text{FunR}]\frac{\Gamma[f \mapsto (T_1, ..., T_n) \to T] \cdot [x_1 \mapsto T_1, ..., x_n \mapsto T_n] \vdash e : T' \qquad T' = T \qquad f \notin \text{dom}(\text{top}(\Gamma))}{\Gamma \vdash Tf(T_1x_1, ..., T_nx_n) = e; : \Gamma[f \mapsto (T_1, ..., T_n) \to T]}$$

#### 4.2.1. Let

Bisogna specificare dove i parametri sono validi in uno scope dato un `D` nell'istruzione `let D in E`. Ricordando che in esso non vi sono funzioni.

$$[\text{Let}]\frac{\Gamma \cdot [] \vdash D : \Gamma' \qquad \Gamma' \vdash e : T}{\Gamma \vdash \text{let } D \text{ in } e : T}$$

$\Gamma \cdot []$ è equivalente a `newScope()`.

$\Gamma \vdash ...$ è equivalente a `remove()`.

Lo scope delle dichiarazioni di $D$ sono $e$, fuori dal `let` le dichiarazioni non sono più accessibili.

#### 4.2.2. Subtyping

Una relazione $<:$ è detta subtyping quando, presa uan coppia di tipi $P$:

- $T <: T$
- $T <: T'$, se $(T, T') \in P$
- $T <: T'$, se $T <: T'' \land T'' <: T'$

Perché è utile? Preso ad esempio `let T x = e in e'` si ha l'albero di prova:

$$[\text{VarD}]\frac{\Gamma \cdot [] \vdash e : T'' \qquad x \notin \text{dom}(\text{top}(\Gamma \cdot [])) \qquad T = T''}{[\text{Prog}]\frac{\Gamma \cdot [] \vdash (Tx = e) : \Gamma \cdot [x \mapsto T] \qquad \Gamma \cdot [x \mapsto T] \vdash e' : T'}{\Gamma \vdash (\text{let } Tx = e \text{ in } e') : T'}}$$

L'uguaglianza $T'' = T$ è troppo forte, eprché non reggerebbe con

```
class C extends P { ... }
let P x = new C in ...
```

Con i sottotipi si avrebbe:

$$[\text{Var-Subt}]\frac{\Gamma \vdash e : T'' \qquad x \notin \text{dom}(\text{top}(\Gamma)) \qquad T <: T''}{[\text{Prog}]\frac{\Gamma \vdash (Tx = e) : \Gamma[x \mapsto T] \qquad \Gamma \cdot [x \mapsto T] \vdash e' : T'}{\Gamma \vdash (\text{let } Tx = e \text{ in } e') : T'}}$$

che tra l'altro può essere compattato come

$$[\text{CompactLet}]\frac{\Gamma \vdash e : T \qquad \Gamma[x \mapsto T'] \vdash e' : T'' \qquad T <: T'}{\Gamma \vdash \text{let } T'x = e \text{ in } e' : T''}$$

Non si ha bisogno di un nuovo scope (il $\Gamma \cdot []$ per intendersi).

La sintassi per un'espressione condizionale del tipo `if e { e1 } else { e2 }` si può tipizzare col subtyping come

$$[\text{If-Subt}]\frac{\Gamma \vdash e : T \qquad \Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad T <: \text{bool} \qquad T_1 <: T' \qquad T_2 <: T'}{\Gamma \vdash \text{if } (e)\{e_1\} \text{ else } \{e_2\} : T'}$$

Con un linguaggio OO si ha che $B <: A$ permette `A x = new B()`.

#### 4.2.3. Typesystem per le dichiarazioni

Prese ad esempio le dichiarazioni con la grammatica

```
stats : stat (';' stat)*;
stat  : ID ':=' exp | 'if' exp '{' stats '}' 'else' '{' stats '}';
```

il tipaggio può essere

$$[\text{Asgn}]\ \frac{\Gamma(x) = T \qquad \Gamma \vdash e : T' \qquad T = T'}{\Gamma \vdash x = e; : \text{void}}$$

$$[\text{IfS}]\ \frac{\Gamma \vdash e : T \qquad \Gamma \vdash s_1 : T' \qquad \Gamma \vdash s_2 : T'' \qquad T = \text{bool} \qquad T' = \text{void} = T''}{\Gamma \vdash \text{if } (e)\{s_1\} \text{ else } \{s_2\} : \text{void}}$$

in questo caso si vede come l'`if` non abbia un tipo di ritorno perché $s_1$ e $s_2$ sono degli statements.

$$[\text{SeqS}]\ \frac{\Gamma \vdash s : T \qquad \Gamma \vdash S : T' \qquad T = \text{void} = T'}{\Gamma \vdash sS : \text{void}}$$

col subtyping si avrebbe

$$[\text{Asgn-Subt}]\ \frac{\Gamma(x) = T \qquad \Gamma \vdash e : T' \qquad T <: T'}{\Gamma \vdash x = e; : \text{void}}$$

$$[\text{IfS-Subt}]\ \frac{\Gamma \vdash e : T \qquad \Gamma \vdash s_1 : T' \qquad \Gamma \vdash s_2 : T'' \qquad T <: \text{bool} \qquad T' <: \text{void} \qquad T'' <: \text{void}}{\Gamma \vdash \text{if } (e)\{s_1\} \text{ else } \{s_2\} : \text{void}}$$

$$[\text{SeqS}]\ \frac{\Gamma \vdash s : T \qquad \Gamma \vdash S : T' \qquad T <: \text{void} \qquad T' <: \text{void}}{\Gamma \vdash sS : \text{void}}$$

anche se $T <:$ void è uguale a dire $T =$ void visto che non c'è un sottotipo per il void.

#### 4.2.4. Classi

Preso il codice

class $A$ {

$T_1\ f_1;$

...

$T_n\ f_n;$

$T_1{}'\ m_1(\overline{T_1{}''\ x_1})\{\}$

...

$T_h{}'\ m_h(\overline{T_h{}''\ x_h})\{\}$

}

Il tipaggio avviene come

$$D = T_1 f_1; ...; T_n f_n; T_1{}'\ m_1(\overline{T_1{}''\ x_1})\{\}; ..., T_h{}'\ m_h(\overline{T_h{}''\ x_h})\{\}$$

$$[\text{Class}]\ \frac{\Gamma[A \mapsto [f_i \mapsto T_i, m_j \mapsto \overline{T_j{}''} \to T_j{}']^{i\in 1..n, j\in 1..h}] \vdash D : \Gamma'}{\Gamma \vdash \text{class } A\{...\} : \Gamma[A \mapsto [f_i \mapsto T_i, m_j \mapsto \overline{T_j{}''} \to T_j{}']^{i\in 1..n, j\in 1..h}]}$$

Nel caso di sotto classi `class A extends B {}` si può definire l'albero per la chiamata di un override di un metodo.

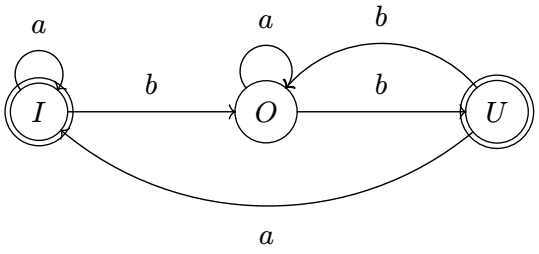
$$[\text{MethodInvk-Subt}]\ \frac{\Gamma(x) = C \qquad \Gamma(C.m) = T_1 \times ... \times T_n \to T \qquad (\Gamma \vdash e_i : T_i' \quad T_i' <: T_i)^{i\in 1..n}}{\Gamma \vdash x.m(e_1, ..., e_n) : T}$$

La regola generale per il subtyping per le funzioni definisce la covarianza come l'output tipo di ritorno e la controvarianzia l'input tipi dei parametri.

# 5. Esercizi

## 5.1. 16 Settembre 2022

Si definisca un analizzatore lessicale in ANTLR che accetta sequenze di token che a loro volta sono stringhe (non vuote) sull'alfabeto `a, b` che contengono un numero pari di occorrenze di `b`.

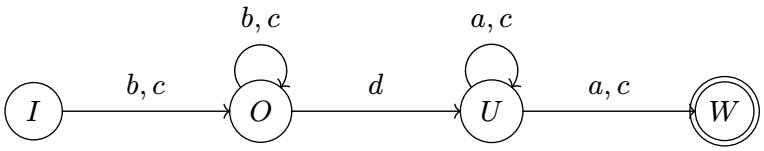



```
init : I*;
I : 'a' I | 'b' O | 'a';
O : 'a' O | 'b' U | 'b';
U : 'a' I | 'b' O | 'a';
```

## 5.2. 20 dicembre 2021

$L$ linguaggio su $\{a, b, c, d\}$ costituto da sequenze non vuote di token $\alpha d \beta$ con $\alpha$ stringa non vuota che contiene $\{b, c\}$ e $\beta$ stringa non vuota che contiene $\{a, c\}$. Definire il lexer in ANTLR senza $*$ o $+$.

eg: `ccdcbdc` è valida, `ccada` no.




```
init : I*;

I : 'b' O | 'c' O;
O : 'b' O | 'c' O | 'd' U;
U : 'a' U | 'c' U | 'a' | 'c';
```

## 5.3. 17 settembre 2021

1. Dare le regole di inferenza dei tipi.

- `exp`

$\Gamma \vdash e : T$

$$\frac{n \in \text{Int}}{\Gamma \vdash n : \text{Int}}$$

$$\frac{}{\Gamma \vdash \text{true} : \text{Bool}}$$

$$\frac{}{\Gamma \vdash \text{false} : \text{Bool}}$$

$$\frac{\Gamma \vdash e : \text{Int} \qquad \Gamma \vdash e' : \text{Int}}{\Gamma \vdash e + e' : \text{Int}}$$

$$\frac{\Gamma \vdash e : \text{Int} \qquad \Gamma \vdash e' : \text{Int}}{\Gamma \vdash e - e' : \text{Int}}$$

$$\frac{\Gamma \vdash e : \text{Int} \qquad \Gamma \vdash e' : \text{Int}}{\Gamma \vdash e > e' : \text{Bool}}$$

$$\frac{\Gamma \vdash e : T \qquad \Gamma \vdash e' : T}{\Gamma \vdash e == e' : \text{Bool}}$$

$$\frac{\Gamma \vdash e : \text{Bool} \qquad \Gamma \vdash e' : \text{Bool}}{\Gamma \vdash e \parallel e' : \text{Bool}}$$

$$\frac{\Gamma \vdash e : \text{Bool} \qquad \Gamma \vdash e' : \text{Bool}}{\Gamma \vdash ee' : \text{Bool}}$$

- `dec`

$$\Gamma \vdash D : \Gamma'$$

$$\frac{x \notin \text{dom}(\text{top}(\Gamma))}{\Gamma \vdash Tx : \Gamma[x \mapsto T]}$$

$$\frac{\Gamma \vdash d : \Gamma' \qquad \Gamma' \vdash D : \Gamma''}{\Gamma \vdash d\ D : \Gamma'}$$

- `stm`

$$\Gamma \vdash S$$

$$\frac{\Gamma \vdash e : T \qquad \Gamma(x) = T}{\Gamma \vdash x = e}$$

$$\frac{\Gamma \cdot \emptyset \vdash D : \Gamma' \qquad \Gamma' \vdash S}{\Gamma \vdash \{D\ S\}}$$

$$\frac{\Gamma \vdash s \qquad \Gamma \vdash S}{\Gamma \vdash s\ S}$$

- `prg`

$$\Gamma \vdash P$$

$$\frac{\emptyset \vdash P : \text{stm}}{\Gamma \vdash P}$$

ma va bene anche dire che il programma è ben tipato, come

$$\vdash P$$

$$\frac{\emptyset \vdash P : \text{stm}}{\emptyset \vdash P}$$

2. Creare l'albero di prova per il programma

$$\dfrac{\dfrac{\dfrac{x \notin \mathrm{dom}(\mathrm{top}(\emptyset \cdot \emptyset))}{\emptyset \cdot \emptyset \vdash \text{int x} : \emptyset[\text{int} \mapsto \text{Int}]} \quad \dfrac{y \notin \mathrm{dom}(\mathrm{top}(\emptyset \cdot [\text{int} \mapsto \text{Int}]))}{\emptyset \cdot [\text{int} \mapsto \text{Int}] \vdash \text{int y} : \Gamma'}}{\emptyset \cdot \emptyset \vdash \text{int x; int y;} : \Gamma'} \quad \dfrac{\dfrac{\Gamma' \vdash 5 : \text{Int} \qquad \Gamma'(x) = \text{Int}}{\Gamma' \vdash \text{x=5}} \quad \dfrac{\Gamma' \cdot \emptyset \vdash \text{bool z} : \Gamma''}{\Gamma' \vdash \{\text{S1}\}}}{\Gamma' \vdash \text{x} = 5; \{\text{S1}\}}}{\dfrac{\emptyset \vdash \{ \text{ int x; int y; x} = 5; \{\text{S1}\}\}}{\vdash \{ \text{ int x; int y; x} = 5; \{\text{S1}\}\}}}$$

Poi si ha anche

```
S1 = {bool z; z = (x > 5)||false; {S2}}
S2 = {int z; y=6+x; z=3+y;}
```

$$\dfrac{\dfrac{\Gamma'' \vdash (\text{x} > 5) \,||\, \text{false} : \text{Bool} \qquad \Gamma'(z) = \text{Bool}}{\Gamma'' \vdash \text{z} = (\text{x} > 5) \,||\, \text{false} : \text{Bool}} \quad \dfrac{\dfrac{z \notin \mathrm{dom}(\mathrm{top}(\Gamma''' \cdot \emptyset))}{\Gamma'' \cdot \emptyset \vdash \text{int z} : \Gamma'''} \qquad \Gamma''' \vdash \text{y=6+x; z=3+y}}{\Gamma'' \vdash \{\text{S2}\}}}{\Gamma'' \vdash \{\text{bool z; z} = (\text{x} > 5)||\text{false; } \{\text{S2}\}\}}$$

TODO: estendere il $\Gamma'''$

## 5.4. 21 giugno 2021

1. Regole di inferenza

Qui il tipo $\mathcal{T} = \{\bot, k, \top\}$

- `exp`

$\Gamma \vdash \mathcal{T}$

$$\frac{x \in \mathrm{dom}(\Gamma')}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{}{\Gamma \vdash n : k}$$

$$\frac{\Gamma \vdash e : t \qquad \Gamma \vdash e' : t'}{\Gamma \vdash e + e' : \#(t, t')}$$

dove

$\#(\bot, k) = \bot \quad \#(k, k) = k \quad \#(\top, -) = \top \quad \#(\bot, \bot) = \bot$

- `stm`

$\Gamma \vdash S : \Gamma'$

$$\frac{\Gamma(x) = \bot \qquad \Gamma \vdash e : k}{\Gamma \vdash x = e : \Gamma[x \mapsto k]}$$

$$\frac{\Gamma(x) \neq \bot \qquad \Gamma \vdash e : T \vee \Gamma \vdash e : k}{\Gamma \vdash x = e : \Gamma[x \mapsto T]}$$

$$\frac{\Gamma \vdash e : t \qquad t \in \{k, \top\} \qquad \Gamma \vdash S : \Gamma' \qquad \Gamma \vdash S' : \Gamma''}{\Gamma \vdash \text{if } (e)\ S \text{ else } S' : \Gamma' \sqcup \Gamma''}$$

dove

$$(\Gamma' \sqcup \Gamma'')(x) = \begin{cases} \bot \text{ se } \Gamma'(x) = \Gamma''(x) = \bot \\ k \text{ se } \Gamma'(x) = \Gamma''(k) = k \\ \top \text{ altrimenti} \end{cases}$$

## 5.5. Esercizio 2 di esame 19 giugno 2019

Preso $\Gamma = [x \mapsto C_x, y \mapsto C_y, z \mapsto C_z]$ trovare l'albero sintattico per il comando `x := y; y := z; z := new C()`.

$$\dfrac{\dfrac{\Gamma(x) = C_x \qquad \Gamma \vdash y : C_y \qquad C_y <: C_x}{\Gamma \vdash \text{x} := \text{y}} \qquad \dfrac{\dfrac{C_z <: C_y}{\Gamma \vdash \text{y} := \text{z}} \qquad \dfrac{C <: C_z}{\Gamma \vdash \text{z} := \text{new C()}}}{\Gamma \vdash \text{y} := \text{z; z} := \text{new C()}}}{\Gamma \vdash \text{x} := \text{y; y} := \text{z; z} := \text{new C()}}$$

## 5.6. 16 luglio 2021

I programmi di un linguaggio di programmazione sono blocchi `{ Dec Stm }` dove

- `Dec` sono sequenze di dichiarazioni di identificatori interi;
- `Stm` sono sequenze di comandi che possono essere

– assegnamenti;

– condizionali (la guardia del condizionale ha tipo intero, la semantica è quella di C);

– blocchi `{ Dec Stm }`. Attenzione: i blocchi possono essere annidati.

- `Exp` possono essere naturali, identificatori o espressioni con somma.

1. definire l'input completo di ANTLR per la grammatica del linguaggio di sopra

```
prg : '{' dec stm '}';
stm : (ID '=' expr ';' | 'if' '(' exp ')' stm 'else' stm | '{' dec stm '}')*;
dec : ('int' ID ';')*;
exp : NAT | ID | exp '+' exp;

NAT : [0-9]+;
ID  : [a-zA-Z0-9]+;
```

2. dare tutte le regole di inferenza per verificare il corretto uso degli identificatori (identificatori non dichiarati o di dichiarazioni multiple) e per gestire gli offset nella generazione di codice.

Come strutturare il record di attivazione? Come lavorare a runtime?

Ipotesi 1: quando entro in un nuovo blocco creo un nuovo frame point. Mi serve anche la catena statica? No, è inutile perché ti basta seguire la catena dinamica in questo caso. $\Gamma, n, o \vdash S$. Ma questo ha senso nel caso di funzioni. $o$ è l'offset.

Ipotesi 2: ho un solo record di attivazione, dunque i judgement non hanno più livelli di nesting. $\Gamma, o \vdash S$

Seguendo l'ipotesi 2 avremo

- prg

$$\dfrac{\emptyset, o \vdash \text{dec stm} \qquad \text{prg} = \{\text{dec stm}\}}{\vdash \{\text{dec stm}\}}$$

$$\dfrac{\emptyset, o \vdash \text{dec} : \Gamma, o \qquad \Gamma, o \vdash \text{stm} : \Gamma', o'}{\vdash \text{prg} : o'}$$

- dec

$$\frac{x \notin \text{dom}(\text{top}(\Gamma))}{\Gamma, o \vdash \text{int x} : \Gamma[x \mapsto (\text{int}, o)], o + 1}$$

$$\frac{\Gamma, o \vdash d : \Gamma', o' \qquad \Gamma', o' \vdash D : \Gamma'', o''}{\Gamma, o \vdash dD : \Gamma'', o''}$$

- exp

$$\frac{\Gamma \vdash E \qquad \Gamma \vdash E'}{\Gamma \vdash E + E'}$$

$$\frac{\text{ID} \in \text{dom}(\Gamma)}{\Gamma \vdash \text{ID}}$$

$$\frac{}{\Gamma \vdash n}$$

- stm

Le dichiarazioni non sono più accessibili: ho un grande record di attivazione ed è possibile che ci siano porzioni di tale record non più accessibile. Alloco spazio di memoria grande quanto il numero di variabili.

$$\frac{\Gamma \cdot [], o \vdash \text{dec} : \Gamma \cdot \Gamma', o' \qquad \Gamma \cdot \Gamma', o' \vdash \text{stm} : \Gamma \cdot \Gamma'', o''}{\Gamma, o \vdash \{\text{dec stm}\} : \Gamma, o''}$$

$$\frac{x \in \text{dom}(\Gamma) \qquad \Gamma \vdash \text{exp}}{\Gamma, o \vdash \text{x} = \text{exp} : \Gamma, o}$$

$$\frac{\Gamma \vdash e \qquad \Gamma, o \vdash \text{stm} : \Gamma', o' \qquad \Gamma, o' \vdash \text{stm'} : \Gamma'', o''}{\Gamma, o \vdash \text{if (e) stm else stm'} : \max(o', o'')}$$

$$\frac{\Gamma, o \vdash \text{stm} : \Gamma', o' \qquad \Gamma', o' \vdash \text{STM} : \Gamma'', o''}{\Gamma, o \vdash \text{stm} : \text{STM} : \Gamma'', o''}$$

3. definire il codice intermedio per tutti i costrutti del linguaggio, in particolare allocando lo spazio necessario sulla pila per gestire i blocchi. Ricordate che la cgen prende come input anche l'ambiente/tabella dei simboli nei vari nodi dell'albero sintattico. Fate attenzione alla gestione degli accessi ai record di attivazione. Osservate anche che la grandezza massima del record di attivazione si può stabilire staticamente.

Tocca settare il frame pointer, dato che lo uso per accedere alle variabili. E lo setto una volta sola quando accedo al programma. Nel blocco di codice sotto diamo per assodato che `o' <= MAX_ADDRESS`. Lo `SP` in realtà serve per calcolare le somme locali.

```
cgen(∅, 0, prg) =
    storei    FT    MAX_ADDRESS
    storei    SP    MAX_ADDRESS
    addi      SP    o'
    popr      SP
```

```
    cgen(Γ, o', stm)
    halt
```

che è quello che si ottiene con $\emptyset, 0 \vdash \text{prg.dec} : \Gamma, o'$.

```
cgen(Γ, x) =
    store a0
    lookup(Γ, x).offset(FP)
```

```
cgen(Γ, e + e') =
    cgen(Γ, e)
    pushr     a0
    cgen(Γ, e')
    popr      t1
    add       a0      t1
    popr      a0
```

L'offset `lookup(Γ, x).offset` è $> 0$ per come è definito nei sistemi di tipo nell'esercizio 2 (La parte di dec che ritorna $o + 1$).

```
cgen(Γ, x = e) =
    cgen(Γ, e)
    load      a0      lookup(Γ, x).offset(FP)
```

## 5.7. 24 maggio 2024

I programmi di un linguaggio di programmazione sono termini `Dec Stm` dove

- `Dec` sono sequenze di dichiarazioni di identificatori interi o booleani:
- `Stm` sono sequenze di comandi che possono essere
  - assegnamenti;
  - condizionali `if (Exp) { Stm } else { Stm }` ;
  - iterazioni `while (Exp) { Stm }`.
- `Exp` possono essere naturali, booleani, identificatori, espressioni con ssomma `(Exp + Exp)` o sottrazione `(Exp - Exp)` o relazionali `(Exp < Exp)`.

1. definire l'input completo di ANTLR per la grammatica del linguaggio di sopra (incluse le escape sequences);

```
init : dec* stm*;

dec : typ ID ';' ;
stm : ID '=' exp ';'
    | 'if' '(' exp ')' '{' stm '}' 'else' '{' stm '}'
    | 'while' '(' exp ')' '{' stm '}';
exp : N | B | ID | exp '+' exp | exp '-' exp | exp '<' exp ;

typ : 'int' | 'bool';

N : '0'..'9'+;
B : 'true' | 'false';
fragment CHAR  : 'a'..'z' |'A'..'Z' ;
ID : CHAR (CHAR | N)* ;

WS : (' '|'\t'|'\n'|'\r')-> skip;
```

2. dare tutte le regole di inferenza per verificare il corretto uso degli identificatori (identifcatori non dichiarati, dichiarazioni multiple o errori di tipo) e per gestire gli offset nella generazione di codice.

Dato che le dichirazioni sono fatte solo al primo livello, considero il dominio di $\Gamma$ e non il dominio di top di $\Gamma$.

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : T}$$

$$[\mathrm{Ass}]\ \frac{\Gamma(x) = T \qquad \Gamma \vdash e : T' \qquad T' <: T}{\Gamma \vdash x = e; : T}$$

$$[\mathrm{Dec}]\ \frac{x \notin \mathrm{dom}(\Gamma) \qquad T = \mathrm{int} \vee \mathrm{bool}}{\Gamma \vdash T x; : \Gamma[x \mapsto T]}$$

Cosa extra non richiesta

$$\frac{\emptyset, o \vdash \mathrm{dec} : \Gamma, o \qquad \Gamma, o \vdash \mathrm{stm} : \Gamma', o}{\vdash \mathrm{prg} : o}$$

3. definire il codice intermedio per tutti i costrutti del linguaggio, in particolare quello per i programmi.

Generare il codice intermedio per il programma

```
int n ; int x ; n = 0; x = 7; while (1 < x) { n = x+n; x = x-1; }
```

```
cgen(Γ, n) =
  storei A0 n

cgen(Γ, x) =
  store A0 lookup(Γ, x).offset(FP)

cgen(Γ, x = e) =
  cgen(Γ, e)
  load A0 lookup(Γ, x).offset(FP)

cgen(Γ, e1 + e2) =
  cgen(Γ, e1)
  pushr A0
  cgen(Γ, e2)
  popr T1
  add A0 T1
  popr A0

cgen(Γ, e1 - e2) =
  cgen(Γ, e1)
  pushr A0
  cgen(Γ, e2)
  popr T1
  sub T1 A0
  popr A0



cgen(Γ, if (e1 < e2) { s1 } else { s2 }) =
```

```
  end_if = new_label()
  false_branch = new_label()
  cgen(Γ, e1)
  pushr A0
  cgen(Γ, e2)
  popr T1
  sub A0 T1  // e2 - e1
  popr A0
  blz A0 false_branch // e2 - e1 < 0 => e2 < e1 (e1 > e2)
  cgen(Γ, s1)
  b end_if
 false_branch:
  cgen(Γ, s2)
 end_if:

# prg = dec stm
cgen(Γ, prg) =
  storei SP max_value()
  storei FP max_value()
  addi SP o'
  cgen(Γ, stm)
```

```
cgen(ø, int n; int x; n = 0; x = 7; while (1 < x) { n = x+n; x = x-1; }) =
  storei SP max_value()
  storei FP max_value()
  cgen(Γ, n)
  pushr A0
  cgen(Γ, x)
  pushr A0

  // now the stack is
  // ________
  // |      |
  // |------|
  // |      | <-- SP
  // |------|
  // |  x   |
  // |------|
  // |__n__| <-- FP

  storei A0 7
  load A0 -2(FP) // or lookup(Γ, x).offset(FP)
                 // or 1(SP)

 b_while:
  store A0 -2(FP)
  storei T1 1
  bleq A0 T1 e_while
  store A0 -2(FP)
  pushr A0
  store A0 -1(FP)
  popr T1
  add A0 T1
  popr A0
  load A0 -1(FP)

  store A0 -2(FP)
  pushr A0
  storei A0 1
  popr T1
  sub T1 A0
  popr A0
  load A0 -2(FP)
  b b_while
 e_while:
```

$o'$ è l'offset formato dalle due variabili (in questo caso).

(Anche se non richiesto faccio l'albero di prova)

$$\dfrac{\dfrac{}{\emptyset \vdash \text{int } n;: \Gamma[n \mapsto \text{int}]} \quad \dfrac{}{\Gamma \vdash \text{int } x;: \Gamma[x \mapsto \text{int}]} \quad \dfrac{\Gamma(n) = T \quad \Gamma \vdash 0 : T' \quad T' <: T}{\Gamma \vdash n = 0;} \quad \dfrac{\Gamma(x) = T \quad \Gamma \vdash 7 : T' \quad T' <: T}{\Gamma \vdash x = 7;} \quad \star}{\vdash \text{int n ; int x ; n = 0; x = 7; while (1 < x) \{ n = x+n; x = x-1; \}}}$$

$\star$

$$\dfrac{\dfrac{}{\Gamma \vdash 1 : T} \quad \dfrac{\Gamma(x) = T'}{\Gamma \vdash x : T'} \quad T = T' \quad < : T \times T' \to \text{bool}}{\dfrac{\Gamma \vdash 1 < x : \text{bool} \qquad \star\star}{\Gamma \vdash \text{while } (1 < \text{x}) \{ \text{ n = x+n; x = x-1; } \}}}$$

$\star\star$

$$\dfrac{\dfrac{\Gamma(n) = T}{\Gamma \vdash n : T} \quad \dfrac{\dfrac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \dfrac{\Gamma(n) = T}{\Gamma \vdash n : T} \quad + : T \times T \to T}{\Gamma \vdash x + n : T}}{\Gamma \vdash \text{n = x + n} : T} \quad \dfrac{\dfrac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \dfrac{\dfrac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \dfrac{}{\Gamma \vdash 1 : T} \quad - : T \times T \to T}{\Gamma \vdash x - 1 : T}}{\Gamma \vdash \text{x = x - 1} : T}$$

$$\Gamma \vdash \text{n = x+n; x = x-1;}$$

## 5.8. 17 giugno 2024

1. definire l'input completo ANTLR per la grammatica.

```
prg : (dec ';')* exp ;

INT : '0'..'9'+;
fragment CHAR : 'a'..'z' | 'A'..'Z';
ID : CHAR (INT | CHAR)*;

dec : ID '(' ID (',' ID)* ')' '{' exp '}';
exp : INT | ID '(' exp (',' exp)* ')' | ID |  exp '+' exp | exp '-' exp;

WS : ('\t'|'\r'|'\n'| ' ')->skip;
```

2. regole di inferenza per verificare corretto uso di id (non dichiarati, multipli) e gestione offset nella generazione di codice.

$$[\text{Prg}]\ \dfrac{\emptyset, o \vdash \text{dec} : \Gamma, o \qquad \Gamma, o \vdash \text{stm} : \Gamma, o}{\vdash \text{prg} : \Gamma, o}$$

$$[\text{Dec}]\ \dfrac{\Gamma, o \vdash d : \Gamma', o' \qquad \Gamma', o' \vdash D : \Gamma'', o''}{\Gamma, o \vdash dD : \Gamma'', o''}$$

$$[\text{FunDec}]\ \dfrac{f \notin \text{dom}(\text{top}(\Gamma)) \qquad \Gamma \cdot \left[(p_i : \text{int})^{i \in 1..n}\right] \vdash e : \text{int}}{\Gamma \vdash f(p_1, ..., p_n)\{e\} : \Gamma[f \mapsto \text{int}_1 \times ... \times \text{int}_n \to \text{int}]}$$

$$[\text{FunCall}]\ \frac{\Gamma(f) = \text{int}_1 \times ... \times \text{int}_n \to \text{int} \qquad (\Gamma \vdash e_i : \text{int})^{1 \in 1..n}}{\Gamma \vdash f(e_1, ..., e_n) : \text{int}}$$

3. codice intermedio per codice

```
f(x,y) {x+y+3}; g(x) {2 + f(x, x+3)}; 7+g(8)
```

qui le uniche variabili usate sono quelle prese dai parametri.

```
cgen(, prg) =
  storei SP max_value()
  storei FP max_value()
  addi SP o'
  cgen(Γ, stm)
  halt

cgen(Γ, n) =
  storei A0 n

cgen(Γ, x = e) =
  cgen(Γ, e)
  load A0 lookup(Γ, x).offset

cgen(Γ, e1 + e2) =
  cgen(Γ, e1)
  pushr A0
  cgen(Γ, e2)
  popr T1
  addi A0 T1
  popr A0

cgen(Γ, e1 - e2) =
  cgen(Γ, e1)
  pushr A0
  cgen(Γ, e2)
  popr T1
  subi T1 A0 // e1-e2
  popr A0

cgen(Γ, f(x1, ..., xn) {e}) =
 lookup(Γ, f).offset:
  pushr RA
  cgen(Γ, e)
  popr RA
  addi SP n
  popr FP
  rsub RA

cgen(Γ, f(e1, ..., en)) =
  pushr FP
  cgen(e1)
  pushr A0
  ...
  cgen(en)
  pushr A0
  move SP FP
  addi FP n+1
```

```
  jsub lookup(Γ, f).label


----------------


cgen(, f(x,y) {x+y+3}; g(x) {2 + f(x, x+3)}; 7+g(8))=
  storei SP max_value()
  storei FP max_value()

  storei A0 7
  pushr A0

  // g(8)
  pushr FP
  storei A0 8
  pushr A0
  move SP FP
  addi FP 2
  jsub lookup(Γ, g).label // A0 = g(8)

  popr T1
  addi A0 T1 // A0 = g(8) + 7
  popr A0

  halt

 lookup(Γ, f).label:
  pushr RA

  store A0 -1(FP) // x
  store T1 -2(FP) // y
  addi A0 T1 // a0 = x+y
  popr A0
  storei T1 3
  addi A0 T1 // a0 = a0 (x+y) + 3
  popr A0

  popr RA
  addi SP 2
  popr FP
  rsub RA

 lookup(Γ, g).label:
  pushr RA

  // f(x, x+3)
  pushr FP

  store A0 -1(FP) // A0 = x
  pushr A0

  storei T1 3
  addi A0 T1 // A0 = x+3
  pushr A0
```

```
move SP FP
addi FP 3
jsub lookup(Γ, f).label // A0 = (x + (x+3)) + 3
pushr A0

storei A0 2
pushr A0

popr T1 // 2
popr A0 // f(x,x+3)

addi A0 T1 // f(x,x+3) + 2
popr A0

popr RA
addi SP 1
popr FP
rsub RA
```

## 5.9. 16 giugno 2023

Preso il linguaggio

```
prg : '{' (dec)* (stm)* exp '}' ;
stm : ID '=' exp ';' | 'if' '(' exp ')' stm ('else' stm)? ;
dec : ('int' | 'bool') ID '=' exp ';' ;
exp : ID | NUM | BOOL | exp '+' exp | exp '-' exp ;
```

2. dare regole di inferenza per corretto uso degli identificatori. Inoltre le regole devono calcolare il numero massimo di comandi condizionali annidati (un comando condizionale è annidato in un altro se si trova nel ramo then o else del secondo).

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

$$[\text{Ass}]\ \frac{\Gamma(\text{ID}) = T \qquad \Gamma \vdash e : T' \qquad T' <: T}{\Gamma, o \vdash \text{ID} = e : \Gamma, o}$$

$$[\text{Dec}]\ \frac{\text{ID} \notin \text{dom}(\text{top}(\Gamma)) \qquad \Gamma \vdash e : T}{\Gamma, o \vdash T\ \text{ID} = e : \Gamma[\text{ID} \mapsto T], o + 1}$$

$$[\text{If}]\ \frac{\Gamma, o \vdash e : \text{bool} \qquad \Gamma, o \vdash \text{stm} : \Gamma, o' \qquad o'' = \max(o, o')}{\Gamma, o \vdash \text{if}\ (e)\ \ \text{stm} : \Gamma, o''}$$

$$[\text{IfElse}]\ \frac{\Gamma, o \vdash e : \text{bool} \qquad \Gamma, o \vdash \text{stm}_1 : \Gamma, o' \qquad \Gamma, o \vdash \text{stm}_2 : \Gamma, o'' \qquad o''' = \max(o, o', o'')}{\Gamma, o \vdash \text{if}\ (e)\ \text{stm}_1\ \text{else}\ \text{stm}_2 : \Gamma, o'''}$$

## 5.10. 17 febbraio 2022

I programmi di un linguaggio di programmazione sono blocchi `Dec Stm` dove

- `Dec` sono sequenze di dichiarazioni interi (int) con inizializzazione `int X = E`;
- `Stm` sono sequenze di comandi del tipo
  - Assegnamenti di una `exp` ad una variabile;

- Iterazioni while (la guardia è un'espressione intera).

- `Exp` sono costanti intere, identificatori o espressioni con somma.

1. Definire input ANTLR completo.

```
prg: dec stm ;

dec : ('int' ID '=' exp ';')* ;
stm : (ID '=' exp ';' | 'while' '(' exp ')' '{' stm '}')* ;
exp : ID | NUM | exp '+' exp ;

fragment CHAR : 'a'..'z' | 'A'..'Z' ;
NUM  : '0'..'9'+;
ID : CHAR (CHAR | NUM)*;

WS : ('\n'| '\t'| '\r'| ' ')->skip;
```

2. Regole di inferenza per corretto uso di identificatori (non dichiarati o multiple), verifica di un identificatore che cambia valore (all'interno di `stm` c'è assegnamento con ID che compare a sinistra) e per gestire offset.

$$\frac{\Gamma(\mathrm{ID}) = \mathrm{int}}{\Gamma \vdash \mathrm{ID} : \mathrm{int}}$$

$$\frac{\mathrm{ID} \notin \mathrm{dom}(\mathrm{top}(\Gamma)) \qquad \Gamma \vdash e : \mathrm{int}}{\Gamma, o \vdash \mathrm{int}\ \mathrm{ID} = e; : \Gamma[\mathrm{ID} \mapsto \mathrm{int}], o + 1}$$

$$\frac{\Gamma \vdash e : \mathrm{int} \qquad \Gamma(\mathrm{ID}) = \mathrm{int}}{\Gamma \vdash \mathrm{ID} = e; \Gamma'}$$

$$\frac{\Gamma \vdash e : \mathrm{int} \qquad \Gamma, o \cdot [] \vdash s : \Gamma', o}{\Gamma, o \vdash \mathrm{while}\ (e)\{s\} : \Gamma', o}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{int} \qquad \Gamma \vdash e_2 : \mathrm{int}}{\Gamma \vdash e_1 + e_2 : \Gamma}$$

$$\frac{\Gamma \cdot [], o \vdash d : \Gamma', o' \qquad \Gamma', o' \vdash S : \Gamma'', o'}{\vdash \{dS\}}$$

$$\frac{\Gamma, o \vdash d : \Gamma', o' \qquad \Gamma', o' \vdash D : \Gamma'', o''}{\Gamma, o \vdash dD : \Gamma'', o''}$$

$$\frac{\Gamma, o \vdash s : \Gamma', o \qquad \Gamma, o \vdash S : \Gamma'', o}{\Gamma, o \vdash sS : \Gamma'', o}$$

3. Generare codice intermedio per

```
int x = 4; int z = x+5; while (z+3) { z = z+x; while (x) { x = x+1; } }
```

```
cgen(Γ, n) =
  storei A0 n

cgen(Γ, x) =
  move AL T1
  for (i = 0; i < nesting_level - lookup(Γ, x).nesting_level; ++i)
    store T1 0(T1)
  subi T1 lookup(Γ, x).offset
  storei A0 0(T1)

cgen(Γ, x = e) =
  cgen(Γ, e)
  move AL T1
  for(i = 0; i < nesting_level - lookup(Γ, x).nesting_level; ++i)
    store T1 0(T1)
  subi T1 lookup(Γ, x).offset
  store A0 0(T1)

cgen(Γ, while(e){s})
  bwhile = new_label()
  ewhile = new_label()
 bwhile:
  cgen(Γ, e)
  storei T1 0
  beq A0 T1 ewhile
  cgen(s)
  b bwhile
 ewhile:

cgen(, prg) =
  storei FP max_value()
  storei SP max_value()
  cgen(Γ, s)
  halt
  cgen(Γ, d)


cgen(, int x = 4; int z = x+5; while (z+3) { z = z+x; while (x) { x = x+1; }}) =
  storei FP max_value()
  storei SP max_value()

  bwhile1 = new_label()
  ewhile1 = new_label()
  bwhile2 = new_label()
  ewhile2 = new_label()

 bwhile1:
  store A0 -2(FP)
  storei T1 3
  addi A0 T1
  popr A0
  store T1 0
  beq A0 T1 ewhile1
```

```
 store A0 -2(FP)
 store T1 -1(FP)
 addi A0 T1
 popr A0
 load A0 -2(FP)

bwhile2:
 store A0 -1(FP)
 storei T1 0
 beq A0 T1 ewhile2
 store A0 -1(FP)
 storei T1 1
 addi A0 T1
 popr A0
 load A0 -1(FP)

 b bwhile1
ewhile2:
 b bwhile1
ewhile1:
 halt
 storei A0 4
 load A0 -1(FP)

 store A0 -1(FP)
 storei T1 5
 addi A0 T1
 popr A0
 load A0 -2(FP)
```