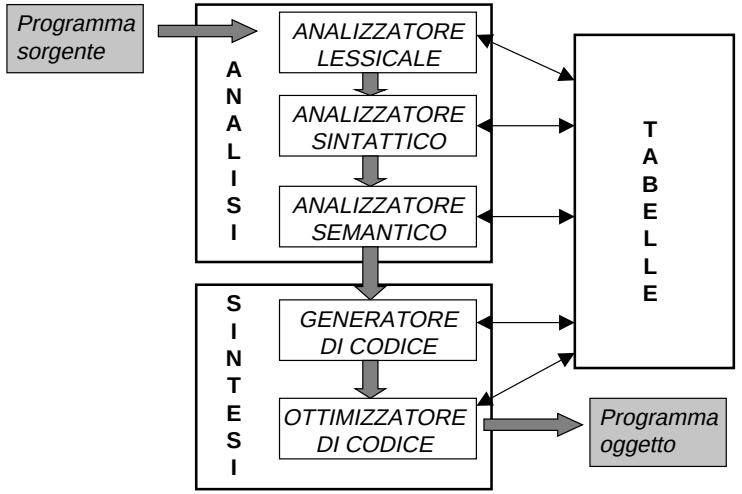
COMPILATORI: MODELLO

- La costruzione di un compilatore per un particolare linguaggio di programmazione è complessa.
  - La complessità dipende dal linguaggio sorgente
- Compilatore: traduce il programma sorgente in programma oggetto.
- Due compiti:
  - ANALISI del programma sorgente
  - SINTESI del programma oggetto

COMPILATORI: MODELLO



ANALIZZATORE LESSICALE

- Un programma sorgente è una stringa di simboli
- Analizzatore lessicale o *scanner*: esamina il programma sorgente per identificare i simboli che lo compongono (*tokens*) classificando parole chiave, identificatori, operatori, costanti…..
- Ad ogni classe di tokens è associato un numero unico che la identifica.
- Vengono ignorati spazi bianchi e commenti

ANALIZZATORE LESSICALE

- Esempio: **`if A>B then X:=Y;`** è trasformata in

TOKEN	NUMERO
`if`	`20`
`A`	`1`
`>`	`15`
`B`	`1`
`then`	`21`
`X`	`1`
`:=`	`10`
`Y`	`1`
`;`	`27`

Si noti che le variabili sono associate allo stesso numero identificativo

Nella TABELLA dei simboli per ogni variabile ho un elemento contenente:

NOME, TIPO, INDIRIZZO, VALORE, LINEA DICHIARAZIONE

ANALIZZATORE LESSICALE

- Esempio: **X1:= a+bb*12;**
  X2:= a/2+bb*12;

TOKEN	CLASSI
X1	id
:=	op
a	id
+	op
bb	id
*	op
12	lit
;	punct

TOKEN	CLASSI
X2	id
:=	op
a	id
/	op
2	lit
+	op
bb	id
*	op
12	lit
;	punct

ANALIZZATORE SINTATTICO

- Progetto di uno scanner e la sua realizzazione
- Compiti:
  - Eliminare bianchi, commenti ecc;
  - Isolare il prossimo token dalla sequenza di caratteri in input;
  - Isolare identificatori e parole-chiave;
  - Generare la symbol-table.
- I tokens possono essere descritti in modi differenti. Spesso sono descritti utilizzando le grammatiche regolari.

ANALIZZATORE SINTATTICO

- Analizzatore sintattico o *parser*: individua la struttura sintattica della stringa in esame a partire dal programma sorgente già trasformato sotto forma di tokens: identifica espressioni, istruzioni, procedure…
- Esempio in linguaggio Pascal

  ALFA1:= 5 + A*B

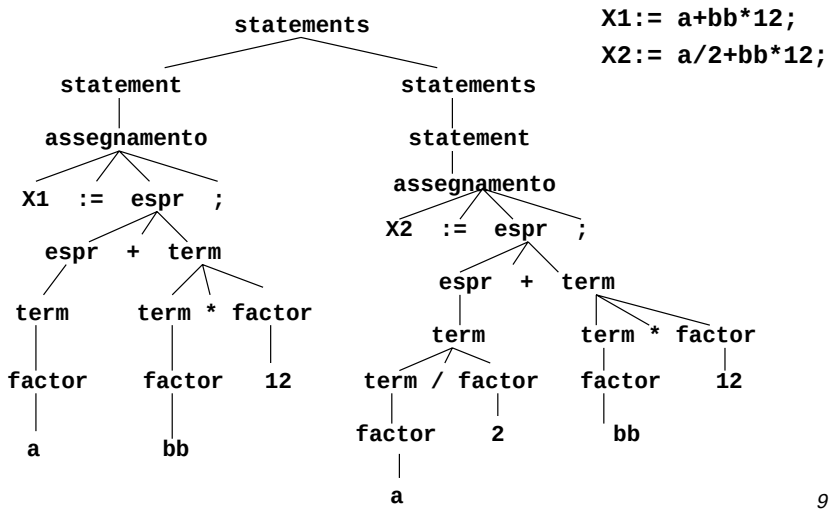
  - la sottostringa 5 + A * B viene riconosciuta come **<espressione>** mentre la stringa completa come **<assegnamento>** secondo la regola sintattica Pascal

  <assegnamento>::= <variabile> := <espressione>
- In realtà viene usata la rappresentazione a classi di token **id1:= lit1 + id2*id3**

ANALIZZATORE SINTATTICO

- Il controllo sintattico si basa sulle regole grammaticali utilizzate per definire formalmente il linguaggio
- Durante il controllo sintattico si genera l'albero di derivazione o *albero sintattico*

ALBERI SINTATTICI




```
X1:= a+bb*12;
X2:= a/2+bb*12;
```


ANALIZZATORE SEMANTICO

- Analizzatore semantico: riceve come ingresso l'albero sintattico generato dal parser.
- Fasi principali
  - CONTROLLO STATICO: vengono svolti i controlli sui tipi, dichiarazioni ecc…
  - AZIONI DA COMPIERE: associazione di routine semantiche associate agli operatori che specificano quali azioni compiere (esempio: tipo operandi conforme all'operatore)
  - GENERAZIONE DI RAPPRESENTAZIONE INTERMEDIA

ANALIZZATORE SEMANTICO

- Considera gli aspetti dipendenti dal contesto
- Esempio (per un linguaggio a blocchi):

```
BEGIN
  INT I;
  ...
END
...
J:=I*K;
```

- Se **I** non è dichiarato anche nel blocco esterno, **J:=I*K;** è un'istruzione illegale, anche se sintatticamente corretta.

ANALIZZATORE SEMANTICO

- La soluzione comune per manipolare queste situazioni è aumentare il lavoro del parser (context-free) con azioni speciali di tipo semantico.
- Esempio: Supponiamo che le dichiarazioni di variabili intere siano considerate con la seguente produzione:

<decl statement>::= INT <identifier>;

- Il nome simbolico per **<identifier>** è riconosciuto dallo scanner e inserito nella tabella dei simboli durante l'analisi lessicale.

ANALIZZATORE SEMANTICO

- Quindi, quando il compilatore incontra l'istruzione:

  `J = I * K;`

  la correttezza dell'utilizzo di **`I`** e' determinata esaminando la tabella dei simboli.

- Un formalismo ormai accettato per descrivere le azioni semantiche necessarie per il controllo statico nei linguaggi (static checking) sono le **attribute grammars**.
  - Il controllo statico si riferisce al controllo di tipo, il controllo che una variabile sia stata dichiarata, il corretto utilizzo degli indici negli array ecc (dipendono dal particolare linguaggio di programmazione).

ATTRIBUTE GRAMMARS

- Grammatiche context-free in cui sono stati aggiunti attributi e regole di valutazione degli attributi chiamate *funzioni semantiche*.
  - Una attribute grammar specifica sia azioni semantiche sia sintattiche.

- **Attributi:**
  - Sono variabili a cui sono assegnati dei valori.
  - Ogni variabile attributo e' associata a uno o più terminali e non terminali della grammatica.
  - Se si scrive E.*Value* si indica che il non terminale E ha l'attributo *Value* .

ANALIZZATORE SEMANTICO

```
Declaration --> Type Var
Type --> Real | Int
```

- Si possono aggiungere i seguenti attributi:

```
Declaration --> Type Var  Var.Class:=Type.Class
Type --> Real|Int         Type.Class:=LexValue
```

- **`LexValue`** e' il valore del token (**`Real`** o **`Int`**) determinato dall'analizzatore lessicale.

FORMATO INTERMEDIO

Il compilatore, durante la traslazione, può creare una forma sorgente intermedia.

- Nella forma intermedia si ignorano alcuni dettagli tipici della macchina target.

- Alcune forme intermedie
  - Notazione polacca;
  - Notazione a n-tuple;
  - Alberi sintattici astratti;
  - Codice di una macchina astratta.

ANALIZZATORE SEMANTICO

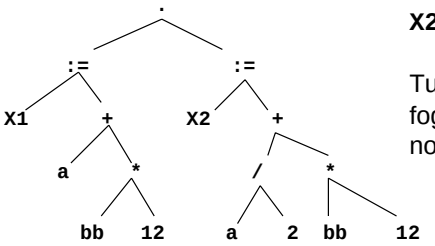
- Esempio **`X1:= a+bb*12;`**
  `X2:= a/2+bb*12;`
  - controlla che il tipo di **`X1`**, **`a`** e **`bb`** sia corretto rispetto agli operatori e al loro risultato. Inoltre, controlla le dichiarazioni delle variabili.
  - un esempio di rappresentazione intermedia può essere in forma a quadruple

`(*,B,12,R1)`
`(+,A,R1,X1)`

ALBERO SINTATTICO ASTRATTO

- Un esempio di rappresentazione intermedia può essere quella che rimuove dall'albero sintattico alcune categorie intermedie e mantiene solo la struttura essenziale.

`X1:= a+bb*12;`
`X2:= a/2+bb*12;`



Tutti i nodi sono tokens. Le foglie sono operandi, mentre i nodi intermedi sono operatori

OTTIMIZZAZIONI

- Spesso a valle dell'analizzatore semantico ci può essere un ottimizzatore del codice intermedio.
  - Propagazione di costanti

```
X:=3;          X:=3;
A:=B+X;        A:=B+3;
```

  evitando un accesso alla memoria

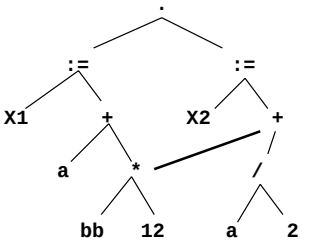
  - Eliminazione di sottoespressioni comuni

```
A:=B*C;        T:=B*C;
D:=B*C;        A:=T;
               B:=T;
```


ALBERO SINTATTICO ASTRATTO

- Eliminando le sotto-espressioni comuni, l'albero sintattico diventa un grafo

`X1:= a+bb*12;`
`X2:= a/2+bb*12;`



ANALISI: Riassumendo….

- Il compilatore nel corso dell'analisi del programma sorgente verifica la correttezza sintattica e semantica del programma:
  - ANALISI LESSICALE verifica che i simboli utilizzati siano legali cioè appartengano all'alfabeto
  - ANALISI SINTATTICA verifica che le regole grammaticali siano rispettate
  - ANALISI SEMANTICA verifica i vincoli imposti dal contesto

GENERATORE DI CODICE

- Generatore di codice: trasla la forma intermedia in linguaggio assembler o macchina
- Prima della generazione di codice:
  - ALLOCAZIONE DELLA MEMORIA
  - ALLOCAZIONE DEI REGISTRI
- Esempio:

```
X1:= a+bb*12;
X2:= a/2+bb*12;
```

allocare l'espressione **bb*12** al registro 1 e una copia del valore di **a** al registro 3 e il valore **a/2** nel registro 2. Le variabili si potrebbero allocare sullo stack con **a** al top e poi, nell'ordine **bb**, **X1**, **X2**. Il registro S punta al top dello stack. (S) accede al top dello stack, 1(S) a una posizione successiva, 2(S) a due posizioni successive...

GENERATORE DI CODICE

- Istruzioni pseudo-assembler per una macchina di nostra invenzione

```
X1:= a+bb*12;
X2:= a/2+bb*12;
```

```
PUSHADDR X2
PUSHADDR X1
PUSH bb
PUSH a
LOAD R1 1(S)      mette bb in R1
MPY  12 R1        mette bb*12 in R1
LOAD R2 S         mette a in R2
STORE R2 R3       copia R2 in R3
ADD R1 R3         mette a+b*12 in R3
STORE @2(S)       mette a+b*12 in X1
DIV  2 R2         mette a/2 in R2
ADD R1 R2         mette a/2+b*12 in R2
STORE @3(S)       mette a/2+b*12 in X2
```


GENERATORE DI CODICE IN PROLOG

- Macchina con un **singolo registro accumulatore** e con le usuali operazioni aritmetiche più le operazioni di:

  `LOAD Value` *(valore o R-value di variabile)*

  `STORE Var` *(L-value di variabile)*

- Algoritmo
  - Quando riconosce una **variabile** la inserisce nello stack
  - Quando riconosce una **operazione**, carica il primo operando nell'accumulatore ed esegue l'operazione con il secondo operando come argomento:

    `LOAD` ***`FirstOperand`***

    `Operation SecondOperand`

GENERATORE DI CODICE IN PROLOG

IPOTESI: espressioni in notazione polacca postfissa

- Le ***variabili*** sono rappresentate da termini **`v(Name)`**
- Le ***costanti*** sono rappresentate da termini **`v(val)`**
- Gli ***operatori*** sono rappresentati da termini **`op(Op)`**
- Le ***locazioni temporanee*** sono rappresentate da **`t(Num)`**

La procedura principale è:

```
gen_code(Polish, Stack, Temps)
```

dove

- la lista **`Polish`** rappresenta l'ingresso in forma polacca postfissa
- la lista **`Stack`** rappresenta lo stack (inizialmente vuoto, **`[]`**)
- la lista **`Temps`** rappresenta le locaz. temporanee (inizialmente **`[0]`**)

GENERATORE DI CODICE IN PROLOG

POSSIBILI QUERY:

- ```
  (a + 3) * b
  :- gen_code([   v(a), v(3), op(+), v(b), op(*)], [],
  [0]).
  ```
- ```
  (a - (3+8)) * b
  :- gen_code([   v(a), v(3), v(8), op(+), op(-), v(b),
            op(*)], [], [0]).
  ```
- ```
  a - (3+8) - (-2 / -4)
  :- gen_code([   v(a), v(3), v(8), op(+), op(-), v(-2),
            v(-4), op(/), op(-)], [], [0]).
  ```

Il risultato è stampato usando **`write`**.

GENERATORE DI CODICE IN PROLOG

Schema generale di top-level:

```
gen_code([op(Op)|Rest], Stack,Temps):-
  operator(Op, Stack, NewStack, Temps, NewTemps),
  gen_code(Rest, NewStack,NewTemps).

gen_code([v(X)|Rest], Stack,Temps):-
  operand(X, Stack, NewStack, Temps, NewTemps),
  gen_code(Rest, NewStack,NewTemps).

gen_code([], _AnyStack, _AnyTemps).
```


GENERATORE DI CODICE IN PROLOG

Il problema dell'accumulatore

- Poiché la nostra macchina ha un solo registro (l'accumulatore), occorre ***evitare di sovrascriverlo***.

Quand'è che l'accumulatore è "*a rischio*" ?

- *NON è a rischio* se il suo valore serve subito nella prossima operazione
- *È a rischio* se il suo valore *non serve* subito nella prossima operazione, perché ciò significa che *esiste un'operazione intermedia* che ne altererà inevitabilmente il valore attuale.

Come tenere conto di questa situazione?
```

GENERATORE DI CODICE IN PROLOG

Per considerare lo "stato di rischio" **dell'accumulatore si introduce un "marcatore" `acc` sullo stack che indica *quando verrà usato* quel valore**

- se **`acc`** è al top dello stack, ***l'accumulatore*** verrà usato per primo e dunque ***non corre pericoli immediati***
- se invece **`acc`** è *appena sotto il top dello stack* (penultima posizione), ***l'accumulatore è in pericolo***
  - ci sarà infatti una operazione intermedia che *non userà* il valore attuale dell'accumulatore, ma *lo sovrascriverà*
  - il valore attuale dell'accumulatore servirà però *in una operazione successiva*

  → ***salvare*** **`acc` in una locazione temporanea Ti**

GENERATORE DI CODICE IN PROLOG

```
operand( X, [A, acc | Stack], [ v(X), A, t(I) | Stack],
         Temps, NewTemps):-
  get_temp(t(I), Temps, NewTemps),
  write('store '), write(t(I)), nl.

operand( X, Stack, [ v(X) |Stack ], Temps, Temps).
```

- CASO 1: **`acc`** è *appena sotto il top dello stack*, in pericolo
  → **generazione di codice per salvare l'accumulatore** in una locazione temporanea Ti
  → inserimento del nuovo valore sullo stack (*senza più marcatore*)
- CASO 2: **`acc`** non corre pericoli
  → semplice inserimento del nuovo valore sullo stack.

GENERATORE DI CODICE IN PROLOG

La gestione degli operandi

- **La nostra macchina elabora espressioni *già in forma polacca postfissa*, quindi opera *in sequenza***
- **Le operazioni devono quindi avere la forma**

  `acc` *`value`* `OP`

Quindi:

- **se va bene che `acc` sia il *primo operando*, tutto ok**
- **altrimenti, *occorre salvarlo* e girare l'operazione:**

  `temp := acc;  acc := value;  acc temp OP`
- **Questo problema riguarda evidentemente *solo le operazioni non commutative*** (differenza, divisione), poiché per le altre l'ordine degli operandi è indifferente.

GENERATORE DI CODICE IN PROLOG

Esempi

`4 a * 5 - 2 +`	diventa (caricando **`4`** in **`acc`**)
`acc a * 5 - 2 +`	che diventa (eseguendo **`acc a *`**)
`acc 5 - 2 +`	che diventa (eseguendo **`acc 5 -`**)
`acc 2 +`	che dà il risultato (eseguendo **`acc 2 +`**)
`a 3 8 + -`	comporta (caricando **`3`** in **`acc`**)
`a acc 8 + -`	che diventa (eseguendo **`acc 8 +`**)
`a acc -`	che *non può essere svolta così*

Occorre il temporaneo in cui salvare **`acc`**. Si ha allora:

`a temp -`	che diventa (caricando **`a`** in **`acc`**)
`acc temp -`	che dà il risultato.

GENERATORE DI CODICE IN PROLOG

Nel nostro caso:

- il nostro generatore di codice pone sullo stack nell'ordine *prima il primo operando*, *poi il secondo*: ad esempio, **`4 a *`** comporta nello stack **`[ ]`** → **`[v(4)]`** → **`[v(a),v(4)]`**
- quindi, ***il primo operando diventa il secondo sullo stack***

Dunque, operativamente:

- se **acc** è <u>in seconda posizione sullo stack</u>, **no problem**
- se invece è <u>al top</u> (e dunque è il *secondo* argomento):
  - **nessun problema se l'operazione è *commutativa***
  - **necessità del temporaneo se l'operazione <u>non è</u> commutativa.**

GENERATORE DI CODICE IN PROLOG

```
operator(Op, [A, acc|Stack], [acc|Stack], Temps, NewTemps):-
   codeop(Op, Instruction, _AnyOpType),
   gen_instr(Instruction, A, Temps, NewTemps).

operator(Op, [acc, A|Stack], [acc|Stack], Temps, NewTemps):-
   codeop(Op, Instruction, commute),
   gen_instr(Instruction, A, Temps, NewTemps).

operator(Op, [acc, A|Stack], [acc|Stack], Temps, NewTemps):-
   codeop(Op, Instruction, noncommute),
   get_temp(t(I), Temps, TempsChanged1),
   write('store '), write(t(I)), nl,
   gen_instr(load, A, TempsChanged1, TempsChanged2),
   gen_instr(Instruction, t(I) ,TempsChanged2, NewTemps).

operator(Op, [A, B|Stack], [acc|Stack], Temps, NewTemps):-
   A \= acc, B \= acc,
   codeop(Op,Instruction,OpType),
   gen_instr(load, B, Temps, TempsChanged),
   gen_instr(Instruction, A, TempsChanged, NewTemps).
```


GENERATORE DI CODICE IN PROLOG

La gestione degli operandi

```
codeop(+, add, commute).
codeop(-, sub, noncommute).
codeop(*, mul, commute).
codeop(/, div, noncommute).

get_temp( t(I), [I, J|R], [J|R] ).
get_temp( t(I), [I],      [J]   ):-
    J is I + 1.
```

Recupera i temporanei

Genera i nuovi temporanei

Traccia il temporaneo corrente

```
gen_instr(Instruction, t(I), Temps, [ I | Temps ]):-
  write(Instruction), write(' '), write(t(I)), nl.

gen_instr(Instruction, v(A), Temps, Temps):-
  write(Instruction), write(' '), write(A), nl.
```


GENERATORE DI CODICE IN PROLOG

IL RISULTATO DELLE QUERY (1/2):

- **`(a + 3) * b`**
  `:- gen_code([v(a), v(3), op(+), v(b), op(*)], [], [0]).`

```
load a
add 3
mul b
```

- **`(a - (3+8)) * b`**
  `:- gen_code([ v(a),v(3),v(8),op(+),op(-),v(b),op(*)],[],[0]).`

```
load 3
add 8
store t(0)
load a
sub t(0)
mul b
```

GENERATORE DI CODICE IN PROLOG

IL RISULTATO DELLE QUERY (2/2):

- **a - (3+8) - (-2 / -4)**
  :-gen_code([v(a),v(3),v(8),op(+),op(-),v(-2),
  v(-4), op(/), op(-)], [], [0]).

```
load 3
add 8
store t(0)
load a
sub t(0)
store t(0)
load -2
div -4
store t(1)
load t(0)
sub t(1)
```

Variante: **a - (3+8) + (-2 / -4)**

Operazione COMMUTATIVA → evita t(1)

```
load 3
add 8
store t(0)
load a
sub t(0)
store t(0)
load -2
div -4
add t(0)
```


OTTIMIZZATORE DI CODICE

- Il codice generato può essere ottimizzato:
  - ottimizzazioni indipendenti dalla macchina
    - ad esempio rimozione di invarianti di ciclo, rimozione di espressioni duplicate
  - ottimizzazioni dipendenti dalla macchina
    - che riguardano ad esempio l'ottimizzazione sull'uso dei registri

COMPILATORI

- Per ora abbiamo visto tutti i passi effettuati da un compilatore separatamente. In realtà questi passi possono essere uniti.
  - Scanner e parser possono essere eseguiti in sequenza, oppure lo scanner può essere chiamato dal parser ogni volta che necessita di un nuovo token.
  - Parser e analizzatore semantico possono essere uniti.
- Altri aspetti:
  - error detection e recovery
  - tabelle dei simboli generate dai vari moduli
  - gestione della memoria

CONCLUDENDO...

- Spunto per un'esercitazione: come si realizza un generatore di codice a partire da alberi sintattici ?
- Realizzare le varie parti di un compilatore in Prolog.
- Per maggiori dettagli
  - *J Cohen, T.J. Hickey: "Parsing and Compiling Using Prolog" ACM TOPLAS, Vol. 9, N. 2, Aprile 1987;*
  - *D.H.D. Warren: "Logic Programming and Compiler Writing", Software Practice and Experience, Vol 10, 97-125 (1980).*
  - *L.Sterling, E.Shapiro: "The Art of Prolog", The Mit Press, 1987*