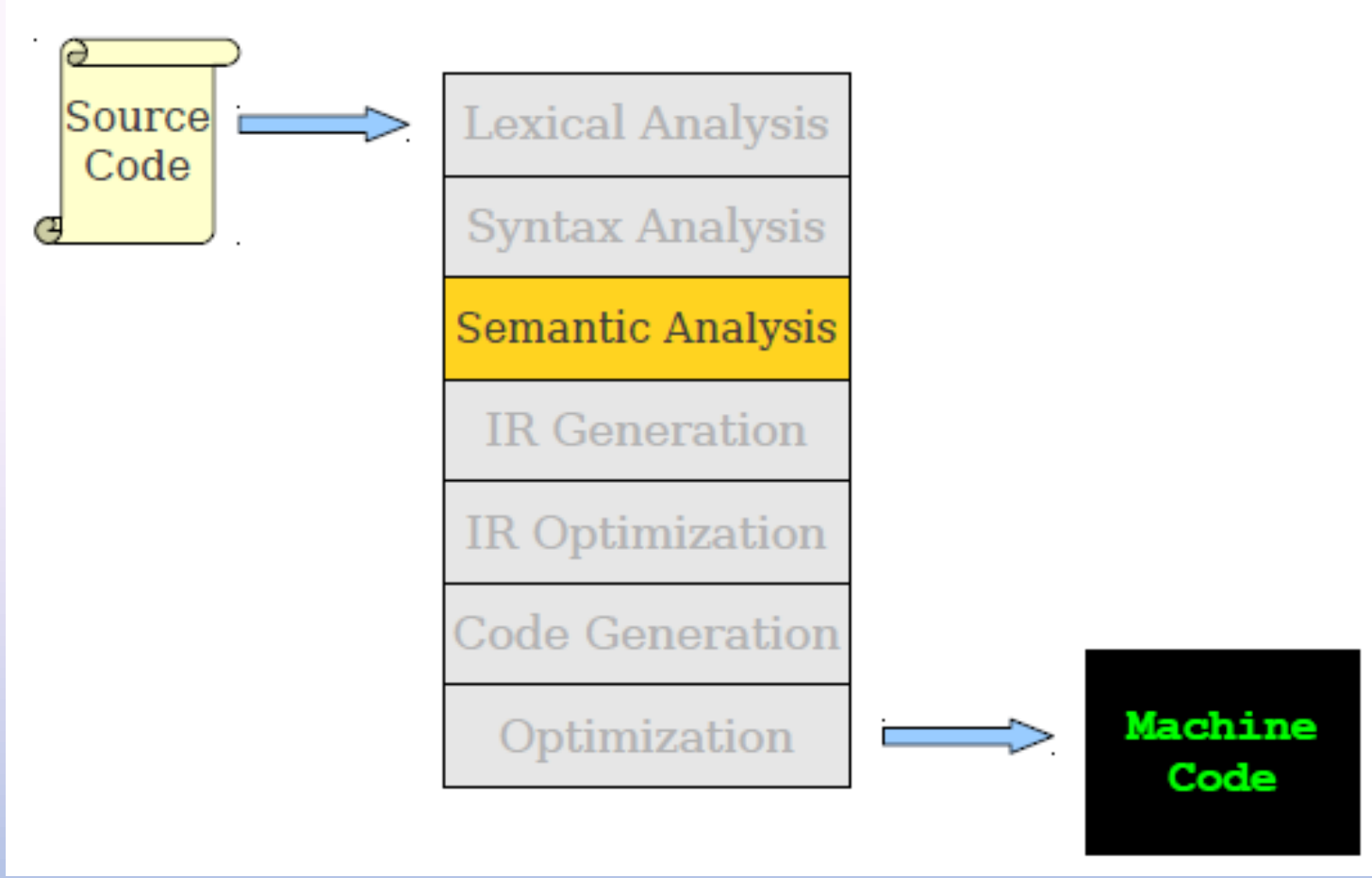
ANALISI SEMANTICA

ANALISI SEMANTICA

Questa fase segue la fase dell'analisi sintattica



OBIETTIVI

La fase di analisi semantica

- verifica ciò che non può essere controllato nelle fasi precedenti, ovvero per esempio:
  - Se le variabili sono state dichiarate prima di essere usate
  - Se le espressioni hanno i tipi corretti
  - Che una variabile o una classe non sia definita due volte
  - Che variabili di tipo diverso siano diverse
  - Che una classe implementi tutti i metodi
  - …
- Raccoglie le informazioni relative agli identificatori introdotti nella tabella dei simboli;
- Verifica la correttezza d'impiego degli identificatori e dei costrutti del linguaggio;
- Segnala gli errori in modo chiaro e senza ridondanze.

In sintesi, interpreta il significato associato alla struttura sintattica e verifica che le regole di impiego del linguaggio siano soddisfatte.

SEMANTICA STATICA O DINAMICA?

SEMANTICA STATICA

Indipendente dai dati su cui opera il programma sorgente. Verifica:

- che una variabile sia dichiarata una sola volta e che venga usata coerentemente al tipo dichiarato;
- il rispetto delle regole che governano i tipi degli operandi nelle espressioni e negli assegnamenti (type checking)
- il rispetto delle regole di visibilità e univocità degli identificatori;
- la correttezza delle strutture di controllo del linguaggio;
- il rispetto delle regole di comunicazione fra i vari moduli (interni e/o esterni) che costituiscono il programma.
- che le chiamate dei sottoprogrammi siano congruenti con le loro dichiarazioni.

SEMANTICA DINAMICA

Dipendente dai dati su cui opera il programma sorgente. Si occupa di:

- controllo sui cicli infiniti;
- controllo sulla dereferenziazione di puntatori NULL;
- controllo sui limiti degli array (per esempio, l'indice di un array non superi i limiti stabiliti dalla sua dichiarazione);
- un dato letto in input sia compatibile con il tipo della variabile a cui è destinato.

L'analizzatore semantico si occupa della semantica statica, mentre la semantica dinamica spetta **all'interprete** o al **supporto esecutivo**.

ESEMPIO DI SEMANTICA STATICA E DINAMICA

Semantica statica: indipendente dai dati su cui opera il programma sorgente.

```
Var i : real;
A : array [1..100] of integer;
.....
i:=3.5;
A[i]:=3;
```

Deve rilevare un errore perché il valore di i non è intero

Semantica dinamica: dipendente dai dati su cui opera il programma sorgente.

```
Var i : integer;
A : array [1..100] of integer;
.....
Read(i);
A[i]:=3;
```

Rileva un errore se il valore inserito non è un intero

ANALISI SEMANTICA STATICA

Nell'analisi sintattica esistono formalismi standard per descrivere la sintassi e i vari algoritmi di parsing per implementare la sintassi stessa.

Nell'analisi semantica la situazione non è così definita: non esiste una metodologia standard per definire o descrivere la semantica statica di un linguaggio; esiste un'enorme varietà di controlli semantici statici nei vari linguaggi.

METODI PER LA DESCRIZIONE DELLA SEMANTICA

L'idea di base, è in ogni caso, quella di integrare il lavoro del parser con azioni speciali di tipo semantico

Esistono diversi approcci per la descrizione della semantica statica di un linguaggio.

Uno tra questi si definisce attraverso le **grammatiche con attributi (attribute grammars)**

GRAMMATICHE CON ATTRIBUTI

Sono grammatiche context-free in cui sono state aggiunte proprietà delle entità sintattiche del linguaggio (**attributi**) e regole di valutazione di tali proprietà (**regole semantiche o equazioni di attributi**).

Una grammatica con attributi specifica quindi sia azioni sintattiche che semantiche.

Le grammatiche con attributi sono utilizzabili in tutti i linguaggi di programmazione che obbediscono al principio della SEMANTICA GUIDATA DALLA SINTASSI che asserisce che la semantica non dipende solo dal contesto ma è strettamente legata alla sintassi.

Ciò accade per tutti i moderni linguaggi di programmazione.

ATTRIBUTI

Un **attributo** è qualunque proprietà delle entità sintattiche di un linguaggio.

Gli attributi possono variare molto rispetto al loro contenuto, alla loro complessità e principalmente in relazione al momento in cui essi sono calcolati.

Gli algoritmi per l'implementazione dell'analisi semantica non sono chiaramente esprimibili come quelli di parsing.

Esempi di attributi sono:

- Il nome della variabile;
- il tipo di una variabile
- la locazione di memoria di una variabile
- il valore di una espressione
- Il numero dei parametri di una procedura (o dimensione della variabile);
- La linea sorgente in cui la variabile è dichiarata;
- Le linee sorgenti in cui la variabile è referenziata;
- …

ATTRIBUTI

Gli attributi possono essere:

STATICI: calcolati al tempo di compilazione (i tipi di dati, il numero delle cifre significative…)

DINAMICI: calcolati al tempo di esecuzione (valore di una variabile o di un'espressione, le locazioni di memoria di una struttura dati dinamica)

Il calcolo di un attributo e l'associazione del suo valore ad un costrutto sintattico è detto **binding** (Legame) dell'attributo.

Il momento in cui questo legame avviene è detto **Binding time**. Differenti attributi possono avere binding time diversi, o anche lo stesso attributo può avere differenti binding time che variano da linguaggio a linguaggio.

Nella fase di analisi semantica statica siamo interessati agli attributi statici, ossia quelli analizzabili in fase di compilazione

ESEMPI DI BINDING TIME

In linguaggi dichiarativi come C e pascal, il tipo di una variabile o di un'espressione è un attributo definito al tempo di compilazione. Il **type checker** (parte dell'analizzatore semantico che calcola tale attributo per tutte le entità del linguaggio per le quali è definito e verifica che tali tipi siano conformi alle regole dei tipi del linguaggio) in C e in pascal agisce durante la fase di compilazione, mentre in linguaggi come LISP o alcuni linguaggi ad oggetti tale processo avviene durante l'esecuzione.

Il valore di un'espressione è generalmente calcolato al tempo di esecuzione. In alcuni casi però (se si ha l'istruzione `x=3+4*5` per esempio) l'analizzatore semantico può scegliere di valutare l'espressione durante la compilazione, poiché possiede già tutti i dati che servono per effettuare la valutazione).

L'allocazione di una variabile può essere sia statica che dinamica e dipende dal linguaggio e dal tipo di variabile: in FORTRAN tutte le variabili sono statiche, in LISP sono tutte dinamiche mentre in C e pascal possono essere sia statiche che dinamiche.

GRAMMATICHE CON ATTRIBUTI

GRAMMATICHE CON ATTRIBUTI

Le **Grammatiche con attributi** sono grammatiche context-free in cui sono state aggiunte proprietà delle entità sintattiche del linguaggio (**attributi**) e regole di valutazione di tali proprietà (**regole semantiche o equazioni di attributi**).

Una grammatica con attributi è quindi una terna **(G, A, R)** dove :

- **G** è una grammatica context-free
- **A** è l'insieme degli attributi associato ad ogni simbolo terminale e non
- **R** è l'insieme di regole associate alle varie produzioni di G.

RAPPRESENTAZIONE DI UN ATTRIBUTO

In queste ipotesi gli attributi sono associati direttamente ai simboli della grammatica (terminali e non).

Se X è un simbolo sintattico e a è un attributo associato ad X scriveremo:

X.a

per accedere al valore corrispondente.

GRAMMATICHE CON ATTRIBUTI E SEMANTICA GUIDATA DALLA SINTASSI

Dato un insieme di attributi

$$a_1, a_2, \ldots\ldots, a_k$$

Il principio della **semantica guidata dalla sintassi** afferma che per ogni produzione del tipo

$$X_0 \rightarrow X_1 \ldots. X_n$$

i valori degli attributi $X_i.a_j$ per ogni simbolo sintattico X_i sono legati ai valori degli attributi degli altri simboli presenti nella produzione.

Questo legame è definito nella forma:

$$X_i.a_j = f_{ij}\ (X_0.a_0,\ X_0.a_1, \ldots,\ X_0.a_k, \ldots,\ X_n.a_0,\ X_n.a_1, \ldots,\ X_n.a_k)$$

E costituisce una **regola semantica (attribute equation)**

Si definisce **grammatica con attributi** l'insieme di tali regole per ogni produzione del linguaggio.

GRAMMATICHE CON ATTRIBUTI

Le grammatiche con attributi sono specificate mediante tabelle, in cui, accanto ad ogni produzione, sono elencate le regole semantiche associate.

regola grammaticale	regole semantiche
regola 1	equazione 1.1 equazione 1.2 equazione 1.3
regola n	equazione n.1 equazione n.2

OSSERVAZIONI

Le grammatiche con attributi sono uno strumento molto potente per l'analisi semantica.

Pur non essendo semplici da usare, le grammatiche con attributi sono meno complesse di quanto sembrano per i seguenti motivi:

1. Di solito gli attributi sono pochi
2. Le regole semantiche dipendono raramente da tutti gli attributi
3. Spesso gli attributi possono essere separati in sottoinsiemi e le regole semantiche possono essere scritte separatamente per ogni sottoinsieme

I manuali dei linguaggi di programmazione non definiscono la grammatica con attributi, sicché il progettista del compilatore deve scriversela a mano.

Nonostante ciò, è importante studiare le grammatiche con attributi perché consentono di definire analisi semantiche più semplici, concise, con meno errori, e che consentono una più semplice comprensione del codice.

ESEMPIO 1

Si consideri la seguente grammatica per esprimere un numero in binario:

number ->number digit | digit

digit -> 0|1

Un attributo significativo potrebbe essere il suo valore in decimale.

Definiamo un attributo val per i simboli non-terminali number e digit (number.val e digit.val).

Come diventa la grammatica con attibuto val ?

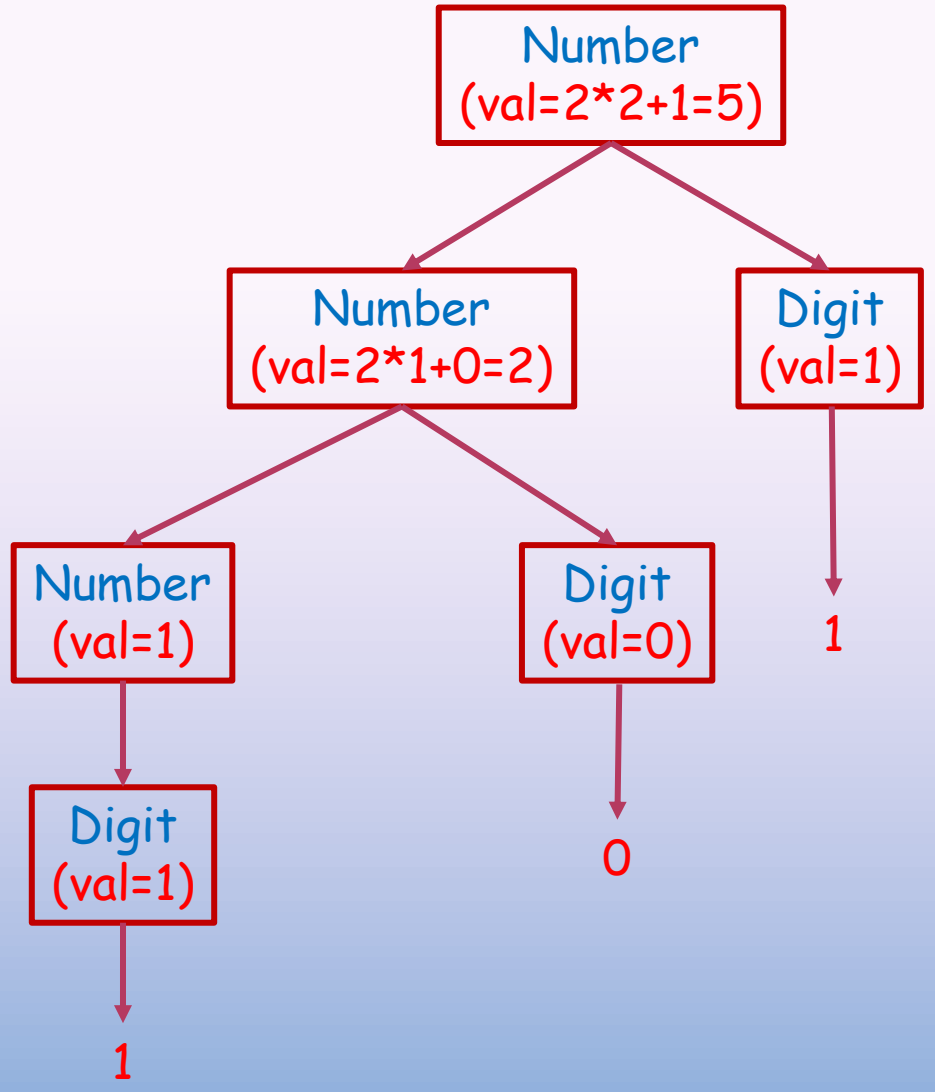
regola grammaticale	regole semantiche
$number_1 \rightarrow number_2\ digit$	$number_1.val = 2*number_2.val + digit.val$
$number \rightarrow digit$	$number.val = digit.val$
$digit \rightarrow 1$	$digit.val = 1$
$digit \rightarrow 0$	$digit.val = 0$

1. La regola **digit->1** implica che digit ha il valore che la cifra stessa rappresenta, e quindi **digit.val=1**
2. Analogamente **digit->0** implica che **digit.val=0**
3. Se il numero è derivato usando la regola **number->digit** allora il suo valore è **number.val=digit.val**
4. Consideriamo che il numero sia derivato usando la regola **number->number digit**. Riscriviamo la regola distinguendo le due occorrenze di number nella testa e nella coda della regola: **number1->number2 digit** da cui si ottiene **number1.val=2*number2.val+digit.val.**

Si noti la differenza tra la rappresentazione sintattica di digit e il suo contenuto semantico (valore). Nella regola **digit->1** il simbolo 1 è un token (simbolo terminale) mentre in **digit.val=1** il simbolo 1 il valore numerico.

ALBERO SINTATTICO DECORATO

- Il significato delle regole semantiche per una particolare stringa può essere descritto usando l'albero sintattico associato agli attributi (**Albero sintattico decorato**).
- Per esempio descriviamo l'albero sintattico decorato per la stringa 101.
- Il calcolo dell'apposita regola semantica è indicato all'interno del nodo.
- E' importante osservare come avviene il calcolo degli attributi per avere un'idea di come possano funzionare gli algoritmi di calcolo degli attributi stessi.



ESEMPIO 2

Completiamo la grammatica dell'esempio 1 in modo da esprimere numeri sia in codice binario (b) che in codice ternario (t):

Gli attributi sono:

based_num : val , base

base : val

number : val , base

digit : val , base

based_num → *number base*

base → ***b*** | ***t***

number → *number digit* | *digit*

digit → 0 | 1 | 2

based_num → number base

*base → **b** | **t***

number → number digit | digit

digit → 0 | 1 | 2

La grammatica con attributi è:

regola grammaticale	regole semantiche
based_num → number base	***based_num.val = number.val*** ***number.base = base.val***
base → b	***base.val = 2***
base → t	***base.val = 3***
$number_1$ → $number_2$ digit	***$number_2$.base = $number_1$.base*** ***digit.base = $number_1$.base*** ***$number_1$.val =*** `if` ((*digit.val == error*) `or` (*$number_2$.val == error*)) `then` ***error*** `else` ***$number_2$.base*$number_2$.val + digit.val***
number → digit	***number.val = digit.val*** ***digit.base = number.base***
digit → 2	***digit.val =*** `if` (***digit.base == 2***) `then` **error** `else` ***digit.val = 2***
digit → 1	***digit.val = 1***
digit → 0	***digit.val = 0***

OSSERVAZIONI

1. Questa grammatica in assenza degli attributi e delle regole semantiche può generare stringhe corrette sintatticamente ma non semanticamente: per esempio la grammatica può generare la stringa **21b** senza rilevare un errore sintattico. E' l'analizzatore semantico che è in grado di rilevare questo errore.
2. Nelle regole semantiche è stato utilizzato il costrutto if-then-else.
3. Ciò è perfettamente lecito perché le regole semantiche saranno scritte in un linguaggio di programmazione (sarà quindi possibile invocare anche funzioni).

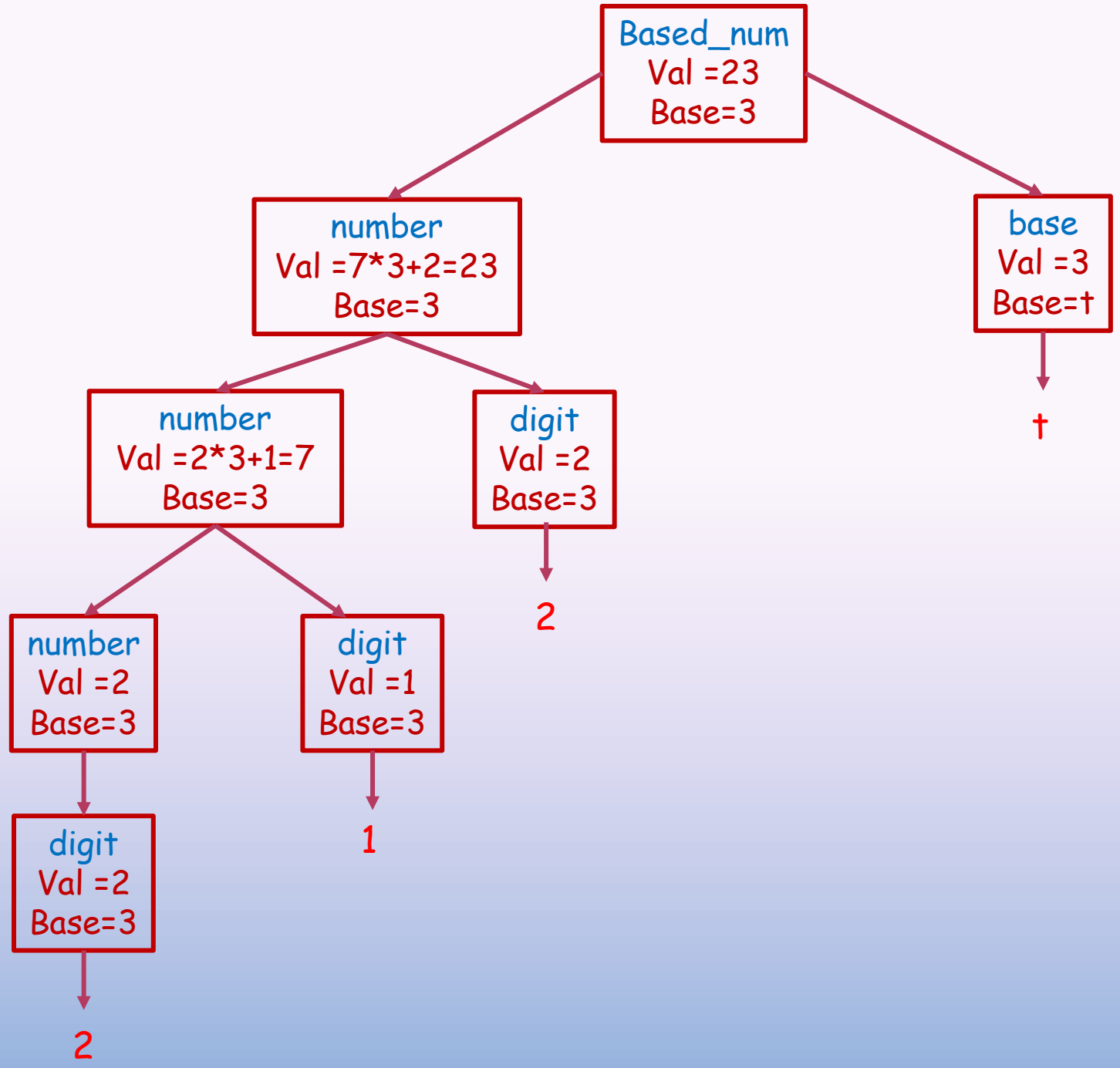
Albero sintattico decorato per la stringa 212t

based_num → *number base*

base → ***b*** | ***t***

number → *number digit* | *digit*

digit → 0 | 1 | 2



CALCOLO DEGLI ATTRIBUTI

Nel primo esempio il calcolo degli attributi è avvenuto con una semplice visita dell'albero di parsing.

In alcuni casi invece può servire completare l'analisi sintattica e la costruzione dell'albero di parsing per poi procedere con l'analisi semantica. Ciò implica che il compilatore deve effettuare più di una passata.

PROBLEMI

1. Come definire gli attributi?
2. Come definire le regole semantiche?
3. Come specificare l'ordine in cui calcolarli?
4. Quali algoritmi utilizzare per calcolarli?

DEFINIRE E CALCOLARE GLI ATTRIBUTI

La definizione degli attributi è relativamente semplice: basta inserire le opportune dichiarazioni nella parte iniziale di qualunque analizzatore sintattico.

Le parti destre delle regole devono essere espressioni effettivamente calcolabili al momento della derivazione della produzione.

Questo vuol dire che gli attributi coinvolti devono essere già disponibili.

CALCOLO DEGLI ATTRIBUTI:

In alcuni casi è semplice. Dipendono dal tipo di visita dell'albero sintattico decorato.

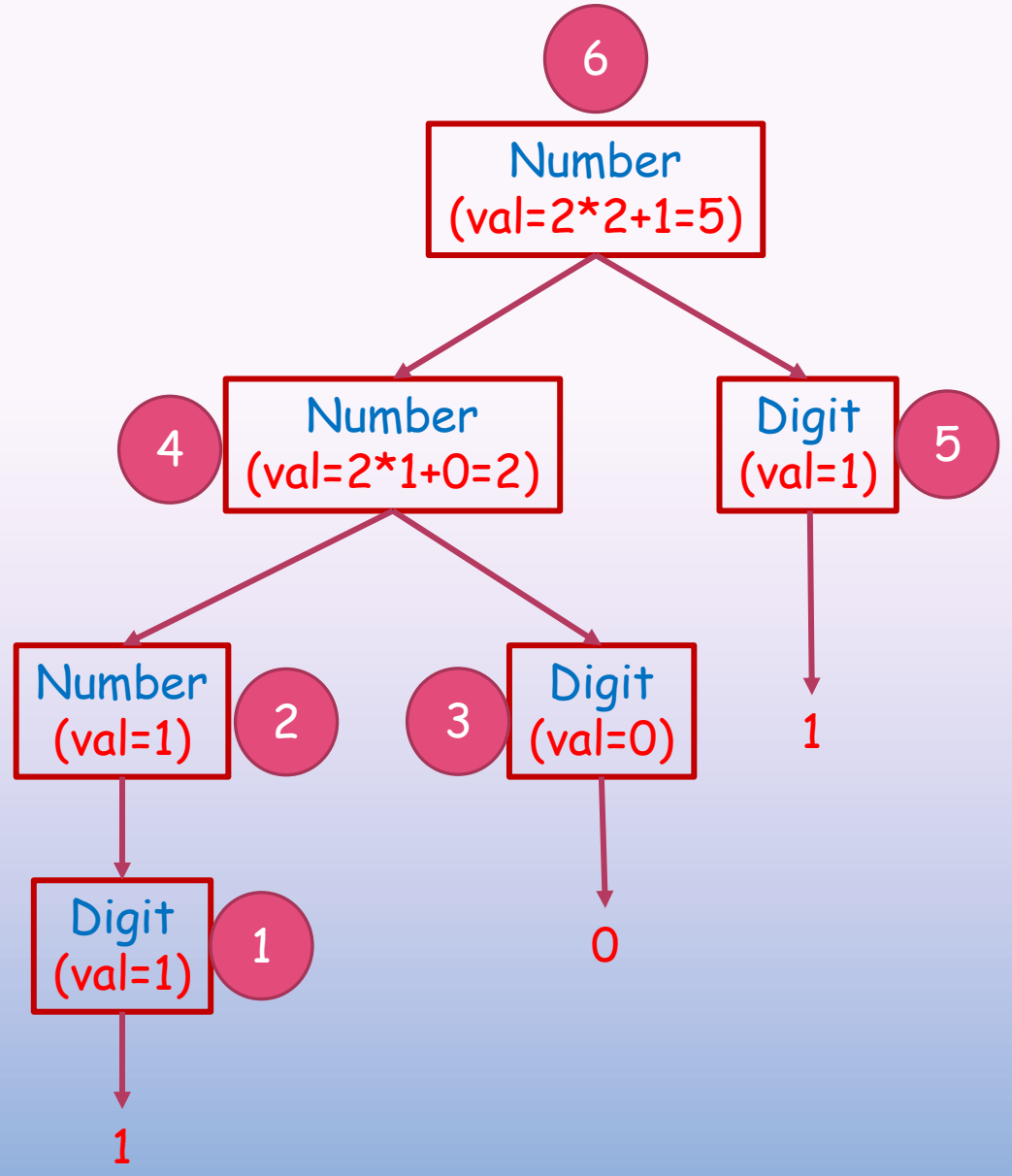
Esempio:

number → number digit | digit

digit → 0 | 1

regola grammaticale	regole semantiche
$number_1 \rightarrow number_2\ digit$	$number_1.val = 2*number_2.val + digit.val$
number → digit	*number.val = digit.val*
digit → 1	*digit.val = 1*
digit → 0	*digit.val = 0*

In questo caso il calcolo del valore dell'attributo val si ottiene con una visita in ordine posticipato dell'albero decorato.



DIPENDENZE FUNZIONALI DEGLI ATTRIBUTI

Per capire come il diverso modo di visitare l'albero sintattico decorato può influire nella corretta valutazione degli attributi occorre introdurre il concetto di dipendenze funzionali degli attributi.

Una regola semantica assegna un valore ad un attributo di un nodo dell'albero sintattico, in funzione dei valori di altri attributi del nodo stesso e dei nodi vicini (padre, fratelli e figli).

L'attributo dipende dunque funzionalmente da altri attributi, i cui valori devono essere noti per consentire il calcolo.

GRAFO DELLE DIPENDENZE

Data una grammatica con attributi, ad ogni regola è associato un **grafo delle dipendenze**.

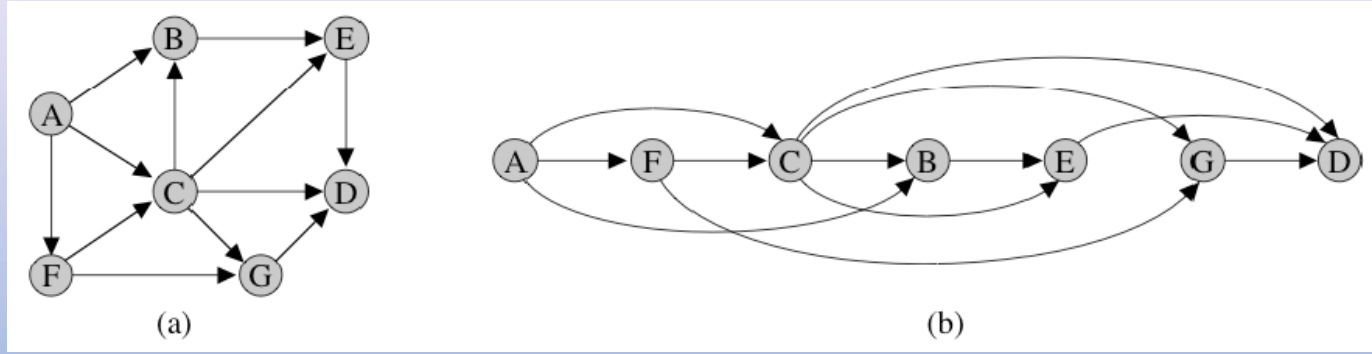
Per ogni nodo dell'albero di parsing etichettato con il simbolo X, il grafo delle dipendenze avrà un nodo per ogni attributo di X.

Se una regola definisce il valore dell'attributo A.b in funzione di X.c, allora esisterà un arco da X.c a A.b.

ALGORITMO GENERALE PER IL CALCOLO DEGLI ATTRIBUTI

L'algoritmo deve calcolare l'attributo di un nodo prima del calcolo dell'attributo del nodo successore; bisogna cioè trovare un ordinamento topologico del grafo.

- Il grafo deve essere un DAG (direct acycic graph). Quindi in particolare può essere un albero
- Il tempo di calcolo può diventare eccessivo.
- L'ordinamento topologico consiste nel trovare una sequenza dei vertici tale che se esiste un arco da u a v, allora u precede v nell'ordinamento.

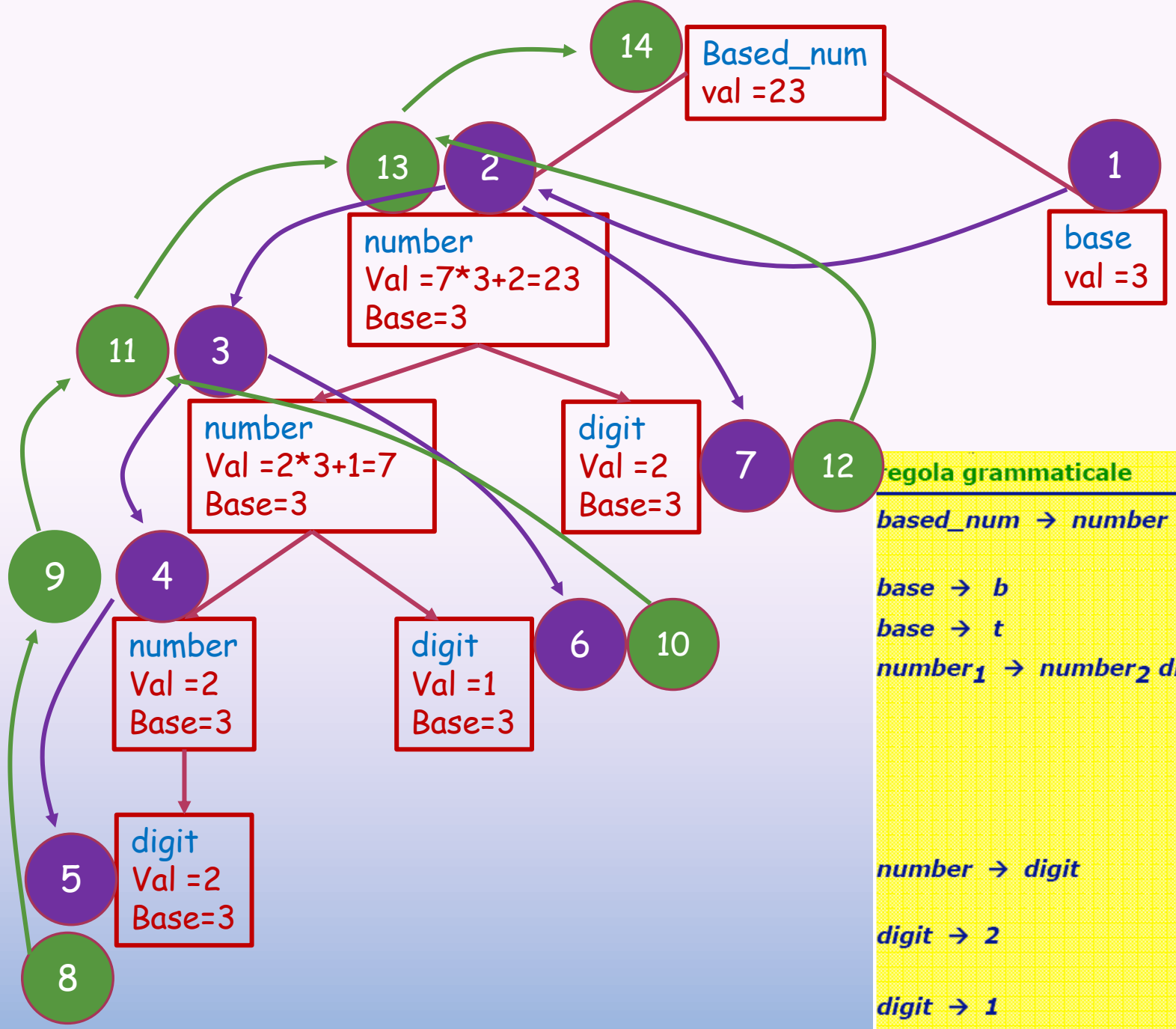


(a) Esempio di grafo aciclico; (b) Ordinamento topologico del grafo (a)

ORDINAMENTO TOPOLOGICO

```
Topological_sort (G, L) /*L restituisce la sequenza dei nodi
                        nell'ordinamento topologico */
INIZIALIZZA (G)         //colora di bianco tutti i nodi
      for all u ∈ V do
            If color [u] = white then dfs-topologica (G, u, L)


dfs-topologica (G, u, pila)
      Color [u] ← gray
      For all v ∈ ADJ [u] do
            If color [v] = white then dfs-topologica (G, v, pila)
      Color [u] ← black
      Inserimentointesta (u, pila)
```



In viola il grafo per il calcolo di base
In verde il grafo per il calcolo di val

regola grammaticale	regole semantiche
based_num → number base	based_num.val = number.val number.base = base.val
base → b	base.val = 2
base → t	base.val = 3
$number_1$ → $number_2$ digit	$number_2$.base = $number_1$.base digit.base = $number_1$.base $number_1$.val = if ((digit.val == error) or ($number_2$.val == error)) then error else $number_2$.base*$number_2$.val + digit.val
number → digit	number.val = digit.val digit.base = number.base
digit → 2	digit.val = if (digit.base == 2) then error else digit.val = 2
digit → 1	digit.val = 1
digit → 0	digit.val = 0

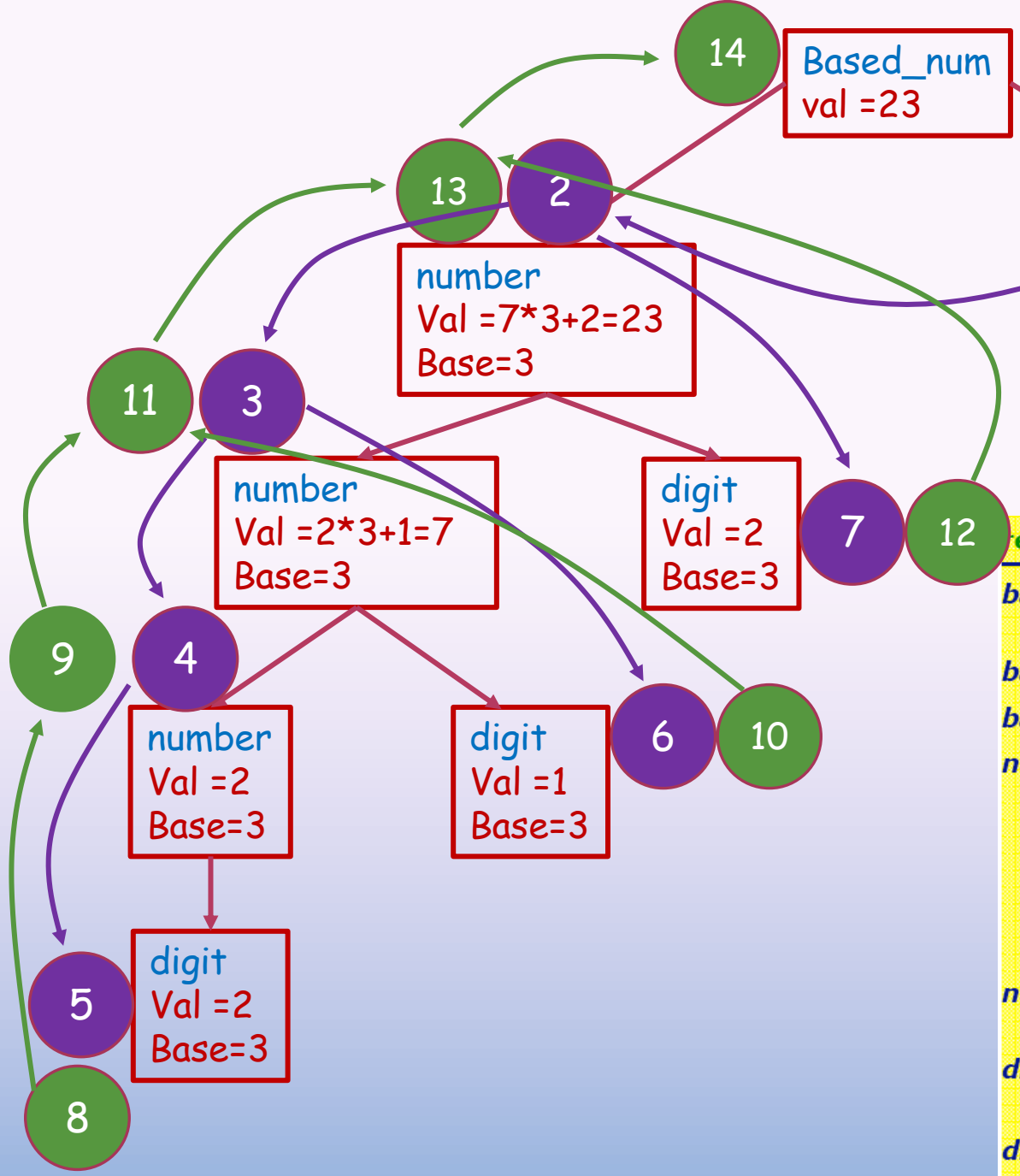
SI RESTRINGE LA CLASSE DELLE GRAMMATICHE CON ATTRIBUTI

Esistono particolari classi di grammatiche con attributi non circolari:

Attributi sintetizzati (synthesized attributes)

Un attributo di un nodo si dice **sintetizzato** se il suo valore dipende solo dai valori degli attributi dei nodi figli.

Attributi ereditati (inherited attributes) Un attributo di un nodo si dice **ereditato** se il suo valore dipende dai valori degli attributi del nodo padre e/o dei nodi fratelli (utili per esprimere le dipendenze di un costrutto di un linguaggio rispetto al suo contesto)



In viola il grafo per il calcolo di base.
Si tratta di un attributo ereditato
In verde il grafo per il calcolo di val.
E' un attributo sintetizzato

regola grammaticale	regole semantiche
based_num → number base	based_num.val = number.val number.base = base.val
base → b	base.val = 2
base → t	base.val = 3
$number_1$ → $number_2$ digit	$number_2$.base = $number_1$.base digit.base = $number_1$.base $number_1$.val = if ((digit.val == error) or ($number_2$.val == error)) then error else $number_2$.base*$number_2$.val + digit.val
number → digit	number.val = digit.val digit.base = number.base
digit → 2	digit.val = if (digit.base == 2) then error else digit.val = 2
digit → 1	digit.val = 1
digit → 0	digit.val = 0

ALGORITMI PER IL CALCOLO DEGLI ATTRIBUTI

Nel caso di grammatiche con **attributi sintetizzati** il calcolo del valore degli attributi si ottiene con una sola visita in ordine posticipato dell'albero sintattico decorato (più adatti per un parser ascendente).

Nel caso di grammatiche con **attributi ereditati** che dipendono dagli attributi del nodo padre oppure dagli attributi ereditati dei nodi fratelli che lo precedono il calcolo del valore degli attributi si ottiene con una sola visita in ordine anticipato dell'albero sintattico decorato (più adatti per un parser discendente).

PASSATE PER IL CALCOLO DEGLI ATTRIBUTI

- Le strategie di calcolo degli attributi in una passata (discendente o ascendente) sono applicabili quindi quando le dipendenze fra gli attributi soddisfano le condizioni piuttosto restrittive viste in precedenza.
- In certi casi non è possibile ricondursi a tali condizioni per cui è necessario ricorrere a strategie più potenti come quelle a più passate.
- Ogni passata visita l'albero parzialmente decorato con i valori degli attributi calcolati dalle passate precedenti, e calcola quel (sotto)insieme degli attributi per cui gli insiemi delle dipendenze sono disponibili nell'albero.
- A seconda delle modalità di visita (ascendenti, discendenti, da sx a dx o da dx a sx) si possono trattare le diverse classi di grammatiche ad attributi.

UN IMPORTANTE COMPITO DELL'ANALIZZATORE SEMATICO: IL TYPE CHECKING

TYPE CHECKING (CONTROLLO DEI TIPI)

La ST, o meglio il suo contenuto, è molto importante per uno dei principali compiti della fase di analisi semantica: il **type checking**.

In effetti, con **type checking** si fa riferimento a due attività separate ma strettamente correlate fra di loro (eseguite insieme):

- Calcolo e mantenimento delle informazioni sui tipi dei dati;
- Controllo che ogni parte del programma abbia un senso per le regole dei tipi ammessi nel linguaggio.

TYPE CHECKING (CONTROLLO DEI TIPI)

Le informazioni sui tipi dei dati possono essere **statiche** o **dinamiche**.

In alcuni linguaggi (lisp) la gestione dei tipi dei dati è interamente dinamica. In questo caso il **type checking** verrà effettuato durante l'esecuzione del programma.

In altri linguaggi (c, pascal) le informazioni sono prevalentemente statiche. In questo caso il **type checking** verrà eseguito durante il processo di compilazione.

Le informazioni statiche sono anche utili per determinare lo spazio di memoria necessario per allocare le variabili.

Ovviamente noi ci occuperemo solo del **type checking** statico.

COMPATIBILITÀ O EQUIVALENZA DI TIPI

Il type checker controlla che in un'espressione gli operandi siano compatibili fra di loro e, se necessario, che il risultato sia compatibile con la variabile destinata a riceverlo.

Quando due operandi sono compatibili? Non basta dire: "quando sono dello stesso tipo".

Infatti l'introduzione delle dichiarazioni di tipo nei moderni linguaggi di programmazione pone una serie di questioni che il progettista di compilatori (e non solo) deve tenere in considerazione.

COMPATIBILITÀ O EQUIVALENZA DI TIPI

La risposta sulla compatibilità dei tipi dipende infatti dalla nozione di compatibilità o equivalenza fra tipi che viene adottata.

Si considerano due diverse nozioni di equivalenza:

- Equivalenza nominale
- Equivalenza strutturale

EQUIVALENZA DI TIPI

Equivalenza nominale: secondo questa nozione due variabili sono dello stesso tipo se e solo se sono state dichiarate con lo stesso **<nome-tipo>** (sia esso predefinito o definito dall'utente).

Equivalenza strutturale: secondo questa nozione due variabili sono dello stesso tipo se hanno la stessa struttura.

EQUIVALENZA DI TIPI

L'equivalenza nominale risulta molto più restrittiva dell'**equivalenza strutturale**: due tipi equivalenti nominalmente lo sono anche strutturalmente ma non è generalmente vero il contrario.

Solo da alcuni anni, per evitare differenti interpretazioni da parte dei compilatori si è ritenuto essenziale includere nelle descrizione di un linguaggio anche le regole di equivalenza dei tipi.

REALIZZAZIONE DI UN SEMPLICE TYPE CHECKER

Faremo adesso un esempio di un semplice linguaggio per il quale descriveremo il type checking in termini di azioni semantiche.

Introdurremo i due attributi:

Name

Type

Faremo uso della symbol table.

GRAMMATICA

program->var-decls; stmts

var-decls-> var-decls; var-decl | var-decl

var-decl -> **id**:type-exp

type-exp -> **int** | **bool**

stmts -> stmts; stmt | stmt

stmt-> **if** exp **then** stmt | **id**:=exp

exp-> exp + exp |exp **or** exp | **true** | **false** | **id**

TYPE CHECKING DELLE DICHIARAZIONI

`var-decl->id:type-exp`	`insert(id.name,type-exp.type)`
`type-exp->int`	`type-esp.type:=integer`
`type-exp->bool`	`type-esp.type:=boolean`

La regola semantica

`insert(id.nome, type-exp.type)`

Inserisce l'identificatore nella Symbol table, associandogli il suo tipo.

Le altre due regole assegnano all'attributo type dell'espressione tipo intero o booleano in base alla regola grammaticale

TYPE CHECKING DEGLI STATEMENT

Gli statement non hanno un loro tipo ma bisogna sempre verificarne la correttezza.

- Nel caso del costrutto **if** si richiede che l'espressione condizionale sia di tipo boolean.

`stmt->if exp then stmt`	`if not typeEqual(exp.type, boolean) then type-error(stmt)`

- La funzione **`typeEqual`** stabilisce se i tipi rappresentati dai suoi due parametri sono nominalmente equivalenti.
- La funzione **`type-error`** segnalerà opportunamente un errore in funzione dello statement esaminato.

TYPE CHECKING DEGLI STATEMENT

Nel caso dell'assegnazione si richiede che il risultato dell'espressione sia di tipo nominalmente compatibile con quello della variabile assegnata.

`stmt->id:=exp`	`if not typeEqual(lookup(id.name).type, exp.type) then type-error(stmt)`

TYPE CHECKING DELLE ESPRESSIONI

`exp1-> exp2 + exp3`	`if not(typeEqual(exp2.type, integer) and typeEqual(exp3.type, integer))) then type-error(exp1); exp1.type:=integer;`
`exp1-> exp2 or exp3`	`if not(typeEqual(exp2.type, boolean) and typeEqual(exp3.type, boolean))) then type-error(exp1); exp1.type:=boolean`

Nella prima espressione si assegna il tipo integer a expr1 dopo aver verificato che gli operandi exp2 e exp3 siano entrambi interi, altrimenti si attribuisce un errore all'espressione che ne deriva.
Nella seconda espressione si assegna il tipo boolean a expr1 dopo aver verificato che gli operandi exp2 e exp3 siano entrambi booleani, altrimenti si attribuisce un errore all'espressione che ne deriva.

TYPE CHECKING DELLE ESPRESSIONI

`exp->true`	`exp.type:=boolean`
`exp->false`	`exp.type:=boolean`
`exp->id`	`exp.type:=lookup(id.name).type`

Queste tre regole si assegnano un tipo alle espressioni. Le prime due assegnano tipo booleano all'espressione.
La funzione **lookup** restituisce il tipo associato all'identificatore passato come parametro.

Regola grammaticale	Regola semantica
var-decl->id:type-exp	insert(id.name,type-exp.type)
type-exp->int	type-esp.type:=integer
type-exp->bool	type-esp.type:=boolean
stmt->if exp then stmt	if not typeEqual(exp.type, boolean) then type-error(stmt)
stmt->id:=exp	if not typeEqual(lookup(id.name), exp.type) then type-error(stmt)
exp1->exp2 + exp3	if not typeEqual(exp2.type, integer) and typeEqual(exp3.type, integer)) then type-error(exp1) exp1.type:=integer
exp1->exp2 or exp3	if not typeEqual(exp2.type, boolean) and typeEqual(exp3.type, boolean) then type-error(exp1); exp1.type:=boolean
exp->true	exp.type:=boolean
exp->false	exp.type:=boolean
exp->id	exp.type:=lookup(id.name)

TYPE COERCION
(CONVERSIONE AUTOMATICA DI TIPO)

In alcuni linguaggi di programmazione è possibile che alcune espressioni, che coinvolgono operandi di tipo diverso, vengano valutate correttamente dopo opportune conversioni (ogni linguaggio specifica opportune regole di conversione).

Le conversioni possono riguardare sia le espressioni che le assegnazioni.

Le conversioni possono essere:

Implicite: effettuate dal type checker automaticamente.

Esplicite: è il programmatore che forza la conversione (cast).

TYPE COERCION DELLE ESPRESSIONI

Consideriamo le seguenti regole:

```
exp1 -> exp2 + exp3
type-exp -> int
type-exp -> real
```

L'azione semantica che esegue la conversione di tipo può essere:

```
if typeEqual(exp2.type, integer) and typeEqual(exp3.type, integer)
then exp1.type:=integer else

if typeEqual(exp2.type, integer) and typeEqual(exp3.type, real) then
exp1.type:=real else

if typeEqual(exp2.type, real) and typeEqual(exp3.type, integer) then
exp1.type:=real else

if typeEqual(exp2.type, real) and typeEqual(exp3.type, real) then
exp1.type:=real

else type-error(exp1)
```

OVERLOADING

Un operatore (o una funzione) è overloaded se lo stesso nome di operatore ha diverso significato a seconda del contesto.

Es.1 2+3 (somma tra interi)

2.1 +3.0 (somma tra reali)

Es.2 int max(int x,y);

float max (float x,y);

char max (char x,y)

Un modo per eliminare l'ambiguità consiste nell'effettuare le operazioni di lookup nella Symbol Table controllando anche la lista dei parametri.

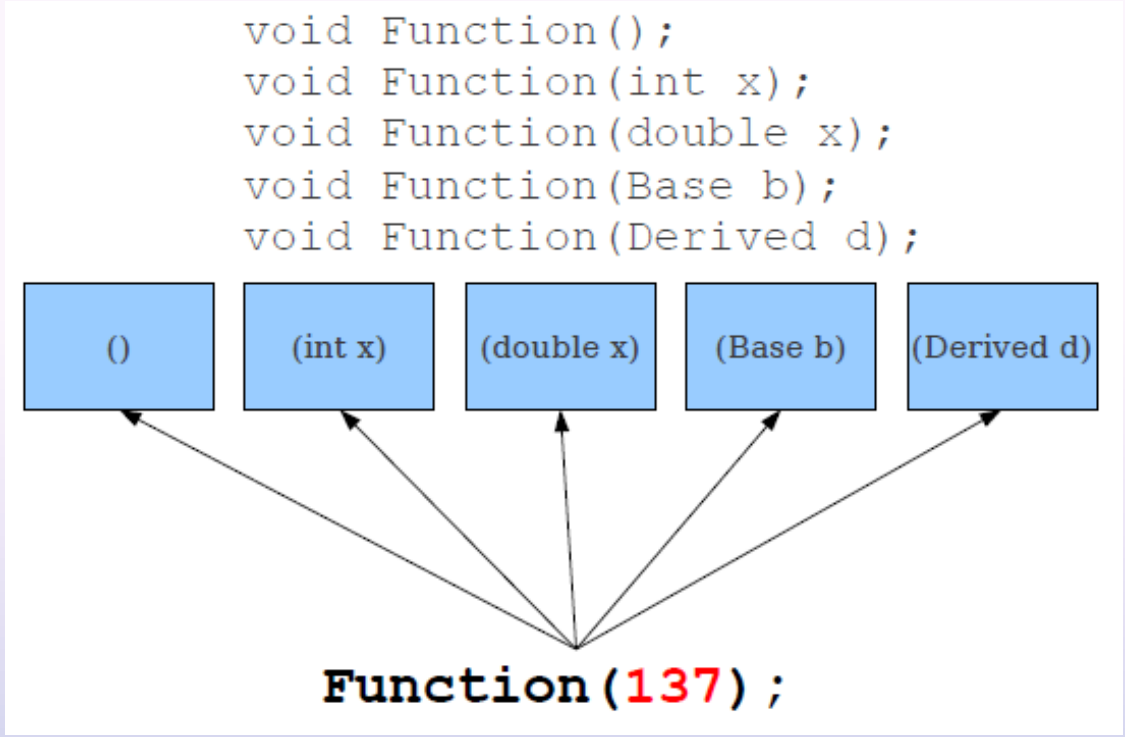
ESEMPIO

```
void Function();
void Function(int x);
void Function(double x);
void Function(Base b);
void Function(Derived d);



Function();
Function(137);
Function(42.0);
Function(new Base);
Function(new Derived);
```

COME SI REALIZZA?



Si verifica con quale lista di parametri formali, i parametri attuali hanno un match.

POLIMORFISMO

Una funzione è polimorfa se può avere parametri di qualsiasi tipo.

```
Procedure swap(var x,y:anytype);
```

A differenza dell'overloading in cui ci sono funzioni diverse con lo stesso nome, nel polimorfismo una stessa funzione può essere applicata a variabili di più tipi.

Esistono particolari type checker che, sfruttando particolari e sofisticate tecniche di pattern matching risolvono il problema.

EQUIVALENZA STRUTTURALE

L'unificazione è il problema di determinare se due espressioni s e t possono essere rese identiche sostituendo alle variabili di s e t nuove espressioni.

Si risolve attraverso algoritmi su grafi.

La verifica dell'equivalenza strutturale è un particolare problema di unificazione.