Compilatori e Interpreti

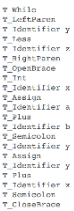
COMPILATORE	INTERPRETE
• Trasforma il codice scritto in un linguaggio di programmazione di alto livello nel codice macchina, immediatamente, prima dell'esecuzione del programma • Il codice compilato viene eseguito più velocemente • Visualizza tutti gli errori dopo la compilazione • Usa un intero programma	• Converte ogni istruzione di programma di alto livello, una per una, nel codice macchina, durante l'esecuzione del programma • Il codice interpretato è più lento • Visualizza gli errori di ogni riga uno per uno • Usa una singola riga di codice
Linguaggi compilati: C, C++, C#, Scala, Java	**Linguaggi interpretati:** PHP, Perl, Ruby, JavaScript, Python (Python viene compilato in byte code che viene quindi salvato ed eseguito in seguito al posto del codice sorgente)

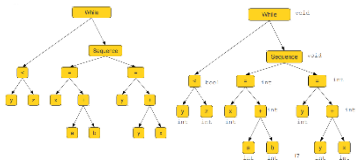
Java

Viene compilato in **byte code** che viene poi **interpretato** ed **eseguito** dalla **Java Virtual Machine**

C (senza interpreti)


```
while (y < z) {
    int x = a + b;
    y = y + x;
}
```





```
Loop: _t1 = y < z
      iffalse _t1 goto Exit
        x = a + b
      y = x+y
      goto Loop
Exit:
```

```
        add $1, $2, $3
LLoop:  slt $6, $4, $5
        beq $6, Exit     Exit
        add $4, $1, $4
        b Loop
EExit:
```

Grammatica senza contesto: tupla (**N**, **T**, →, S) dove

- **N** è un insieme finito di **simboli non terminali**
- **T** è un insieme finito di **simboli terminali**
- → è un insieme finito di produzioni di tipo: A → α1 …αn con A ∈ **N** e α1 …αn ∈ **N** ∪ **T**
- S ∈ **N** è chiamato **simbolo iniziale**
- **Esempio:** BExp →(BExp)→Digit→0|1|…|9 (sintassi compatta che rappresenta 10 produzioni)
- **Manterremo sempre la tupla IMPLICITA**

Derivazioni: sia G = (**N**, **T**, →, S) una grammatica priva di contesto e ɣ e ɛ siano sequenze di simboli in **N** ∪ **T**. Una **derivazione a un passo** di G è ɣAɛ ⟹ ɣα1 …αnɛ dove A ⟹ α1 …αn ∈ ⟹* (0 o più passaggi) ⟹+ (1 o più passaggi) ɣ ⟹* ɛ è chiamato **derivazione**

- **Esempio:** BExp →(BExp) →((BExp)) →((Digit)) → ((1))
  - È una **derivazione**
  - La **sequenza dei terminali** ((1)) appartiene alla lingua generata dalla grammatica
  - le sequenze ((9)) e (((((1)))) e 3 **appartengono** al linguaggio, così come
  - le sequenze ((BExp)) e ((((10)))) e ((3)) **non appartengono** al linguaggio
- Una **derivazione più a sinistra** è una derivazione in cui il simbolo non terminale che viene sostituito ogni volta è quello **più a sinistra** (vale la cosa inversa per la **derivazione più a destra**)
  - Exp ⟹ Exp - Exp ⟹ Digit - Exp ⟹ 3 - Exp ⟹ 3 - Exp - Exp ⟹ 3 – Digit - Exp ⟹ 3 - 2 - Exp ⟹ 3 - 2 - Digit ⟹ 3 - 2 -1

Il **linguaggio generato da una grammatica senza contesto** da G = (**N**, **T**, →, S) è l'insieme L(G) = {ɣ | ɣ ∈ **T*** e S ⟹+ɣ}

- **T*** è la **chiusura di Kleene:** ogni sequenza di simboli in **T**
- **Esempio:** se **T** = {a, b}, **T***= {ε, a, b, aa, ab, ba, bb, aaa, ...}

ANTLR

La sintassi

```
block       : '{' statement* '}';
statement   : assignment ';'
            | deletion ';'
            | print ';'
            | block;
assignment  : ID '=' exp;
deletion    : 'delete' ID;
print       : 'print' exp;
exp         : '(' exp ')'
            | '-' exp
            | exp ('*' | '/') exp
            | exp ('+' | '-') exp
            | ID
            | NUMBER
```

Un programma

```
{
    a = 4; //variable a has value 4
    print a; //value 4 has been recorded
    {
        a = 2; //a is redefined
        print a; //value 2 has been recorded
        delete a; //a has been deleted
        print a; //value 4 has been recorded
    }
    delete a; //a has been deleted
    b = 4; //variable b has value 4
    print b * 2 + 4; //value 12 has been recorded
}
```

ANTLR: L'ANALISI

Il corretto utilizzo delle variabili

- **una variabile deve essere dichiarata prima dell'uso**: una variabile non può apparire sul lato destro di un compito o come argomento di un'operazione senza prima apparire sul lato sinistro di un compito
- **una variabile non può essere utilizzata dopo essere stata eliminata:** delete rimuove solo la dichiarazione più vicina, quindi se un'altra variabile con lo stesso nome esiste in un blocco esterno, questo non sarà influenzato

Analisi comportamentale: estrai una tupla [max, total, {prints}] dove:

- **max** è il numero massimo di variabili che possono esistere contemporaneamente
- **total** è il numero di variabili che rimangono non eliminate dopo la fine
- **prints** è la sequenza di valori che è stata scritta dalla stampa

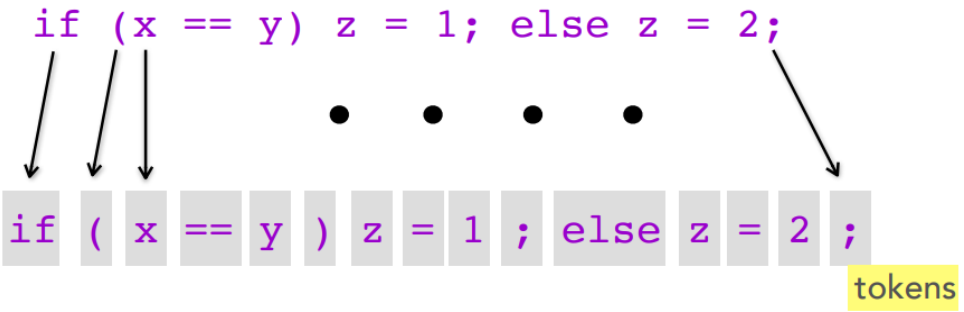
Analisi Lessicale

Suddividiamo grammatiche molto grandi in **blocchi logici** (proprio come facciamo con il software)

Un modo per farlo è dividere una grammatica in una **grammatica lexer** e una **grammatica parser**

- questa non è una cattiva idea perché c'è una sorprendente quantità di sovrapposizione tra lingue diverse
- **ad esempio**, identificatori e numeri sono generalmente gli stessi in tutte le lingue.
- scomporre le regole lessicali in un "modulo" significa che **possiamo usarlo per diverse grammatiche del parser**

L'analisi lessicale divide i testi del programma in token o parole



Design di un Lexer

PARTE 1: **descrizione** (definisci **cosa** fa il lexer)

- **Descrivi ogni token in un modo preciso**, con un **modello formale** come gli automi a stati finiti
- Definisci l'**associazione lessema-token** per ogni possibile lessema nella lingua di input (e la corrispondente **azione da fare**)

PARTE 2: **implementazione** (definisci **come** si comporta il lexer)

- Costruisci l'automa corrispondente al lexer, gli elementi utilizzati **sono comuni a ogni lexer** (è una libreria)
- Definisci lo scanner del programma di input, ad esempio NextChar() e undoNextChar(c)

L'**input** è solo una sequenza di caratteri. **Ad esempio:**

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

L'**obiettivo** è trovare i **lessemi** e mapparli sui **token**: partizionare la stringa di input in sottostringhe (chiamate **lessemi**) e classificare i lessemi in base al loro ruolo (ruolo = **token**)

La stringa di input è **\t if (x == y)\n\t\tz = 1;\n\telse\n\t\tz = 2;**

Il partizionamento in lessemi è **\t if (x == y)\n\t\tz = 1;\n\telse\n\t\tz = 2;**

- **19 lessemi** mappati su una sequenza di token: **IF, LPAR, ID("x"), EQUALS, ID("y"), RPAR …**
- I lessemi costituiti da \n e \t vengono cancellati e non producono token
- Alcuni token hanno attributi: il lessema e il numero di riga

È scomodo costruire un lexer da soli: è noioso ripetitivo, soggetto a errori e non mantenibile

È molto meglio usare un generatore di lexer: con un generatore a portata di mano **ci si può concentrare direttamente sulla definizione dei lessemi e dei token**, ovvero fornire la **descrizione lessicale del linguaggio** e viene **generato automaticamente** il codice che esegue il partizionamento in lessemi/token (il codice generato automaticamente può avere ripetizioni)

Design di un Lexer a mano

Costruiamo un (semplice) lexer **a mano** in Java: l'obiettivo è vedere **come si fa** e capire dove sono le **ripetizioni di codice che vogliamo nascondere**

Il nostro semplice lexer deve riconoscere quattro tokens

token	lexeme
ID	a sequence of one or more letters or digits starting with a letter
EQUALS	"=="
PLUS	"+"
TIMES	"*"

Pseudocodice del Lexer in Java

```
c = nextChar();
if (c == '=') { c=NextChar(); if (c == '=') {
                                    return EQUALS; } }
else if (c == '+') { return PLUS; }
else if (c == '*') { return TIMES; }
else if (c is a letter) {
    c = NextChar();
    while (c is a letter or digit) { c = NextChar(); }
    undoNextChar(c);
    return ID;
}
```

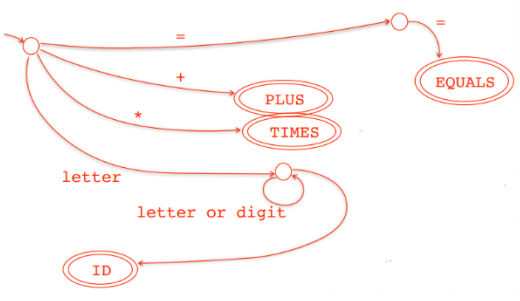
undoNextChar(): esegue un look-ahead per determinare se l'ID lessema può essere più lungo o meno

Il codice precedente mostra un'istanza della **regola di corrispondenza massima** (questa regola è usata da **ogni** lexer): **il flusso di caratteri in input viene partizionato in lessemi il più lunghi possibile**

- Esempio: in Java, “iffy” non è partizionato in “if” (la parola chiave IF) e “fy” (che è un ID), ma in “iffy” (ID)

```
c = nextChar();
if (c == '=') { c=NextChar(); if (c == '=') {
                                    return EQUALS; } }
else if (c == '+') { return PLUS; }
else if (c == '*') { return TIMES; }
else if (c is a letter) {
  c = NextChar();
  while (c is a letter or digit) { c = NextChar();
  }
  undoNextChar(c);
  return ID;
}
```

Le parti in rosso sono importanti per **specificare** il lexer



Modello Astratto Lexer

Il codice segue questo schema: leggi il carattere successivo e confrontalo con uno predeterminato**, se** c'è una corrispondenza **allora** restituisci un token, **altrimenti** ripeti finché non sarà possibile restituire un token

Automi a stati finiti non deterministici (NFA): modelli computazionali che permettono di definire il comportamento di lexer. Sono formati da una tupla (Q, ∑, δ, q_0, F) dove

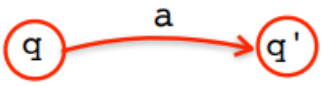
- Q è un insieme finito di stati
- ∑ è un insieme finito di simboli (l'alfabeto di input)
- δ, chiamata relazione di transizione, è una relazione Q x (∑ {ε}) x Q [invece di scrivere δ (q, a) = q' scriviamo q $a\rightarrow$ q']
- q_0 ∈ Q è lo stato iniziale
- F ⊆ Q sono gli stati finali

NFA ha una **notazione grafica**:

- gli stati sono indicati 

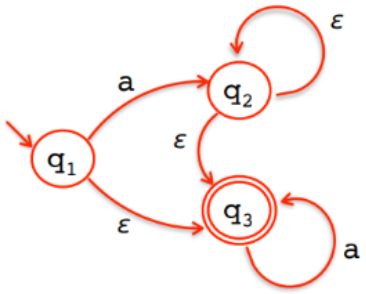
- lo stato iniziale è 

indicato
- gli **stati finali** sono 

- transizioni etichettate tra **due** 

indicati stati

Esempio tupla: ({q1, q2, q3}, {a}, δ, q1, {q3}) dove δ = [(q1, a) ⟼ q2, (q1, ε) ⟼ q3, (q2, ε) ⟼ q2, (q2, ε) ⟼ q3, (q3, a) ⟼ q2]

Esempio: quando **M** è 

L(M) = { ε, a, aa, aaa, …}

Linguaggio definito da una NFA: M = (Q, ∑, δ, q_0, F), scritto L(M), è l'insieme {ϫ | ϫ ∑* e [ϫ = a1 … an implica (q_{i-1} $a_i\rightarrow$ q_1 ∈ $\overline{\delta}$)$^{i\in 1..n}$ e q_n ∈ F]} dove $\overline{\delta}$ è la relazione definita come segue $\overline{\delta}$(q_1, a) = q_n se q_1 ε→ q_2 ε→ … ε→ q_i $a\rightarrow$ q_{i+1} ε→ q_n sono transizioni in δ

Stringa **accettata/rifiutata** da un NFA

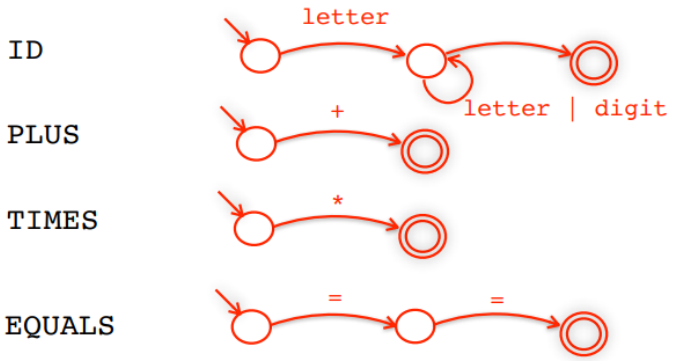
1. Inizia nello stato iniziale unico
2. Quindi inizia a leggere la stringa di input un carattere alla volta
3. Al termine della **lettura**
    - Se lo stato in cui arrivi è **finale**, la **stringa viene accettata**
    - Se lo stato in cui arrivi **NON è definitivo**, la **stringa viene rifiutata**

4. Se nel frattempo **non** è possibile alcuna transizione, la **stringa viene rifiutata**

PARTE 1: DESCRIZIONE DI UN LEXER

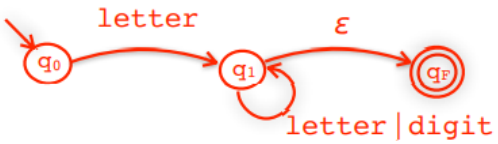
- Definire un **NFA** per ogni lessema del linguaggio
- Associare l'NFA al token riconosciuto

Esempio:

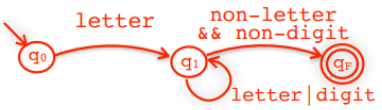


PARTE 2: IMPLEMENTAZIONE DEL LEXER

L'identificazione del token ID ha una **transizione non etichettata** (in realtà una transizione ε) quella da q1 a qF

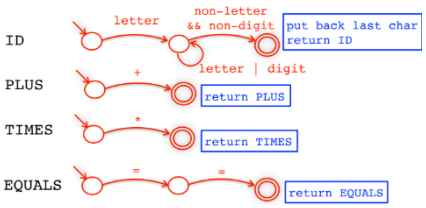


- L'automa **non** è **deterministico**: nello stato q1 non è chiaro cosa succede quando arriva una lettera o una cifra, transiti in q1 o transiti in qF **senza aspettare** il carattere successivo
- È più conveniente utilizzare l'automa **deterministico** (**DFA**: Deterministic Finite-state Automata) e utilizzare look-ahead (per lessemi a lunghezza variabile)



Quando un token viene riconosciuto, l'NFA deve eseguire le **azioni**:

- “return TOKEN”: il chiamante del lexer (il parser) recupera il token riconosciuto e il **lexer riparte dallo stato iniziale**



  - Questa **azione riporta il lexer allo stato iniziale** (in realtà il lexer viene invocato dal parser, ogni volta che viene restituito esattamente un token)
  - In corrispondenza degli stati finali, occorre precisare le azioni del lexer

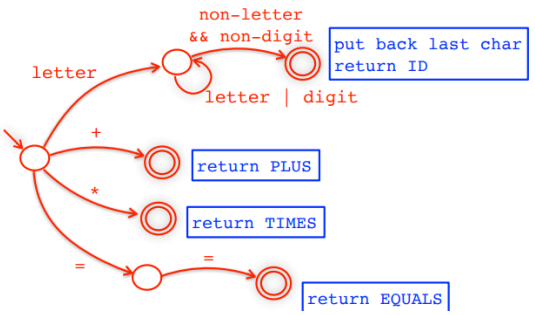
- **Azioni di gestione dei lookahead**, ad esempio undoNextChar(c) per i token che corrispondono a lessemi di lunghezza variabile (nel nostro caso, token ID). I lookaheads possono avere **qualsiasi lunghezza**; nel caso in cui sia necessario eseguire undoNextChar(c) più volte

La NFA della descrizione diventa una **NFA estesa** e è necessario **combinarla.**

Problema: il lexer deve avere un punto di ingresso univoco

Algoritmo: identifica gli stati iniziali dell'NFA che corrispondono ai token e viene utilizzato da ogni lexer

La combinazione non è esattamente così! in questo caso l'automa risultante è DETERMINISTICO



Problema: la combinazione delle NFA può dare in generale il non determinismo

Costruiamo un semplice lexer che riconosce **cinque** token (ASSIGN è un prefisso di EQUALS)

token	lexeme
ID	a sequence of one or more letters or digits starting with a letter
EQUALS	"=="
PLUS	"+"
TIMES	"*"
ASSIGN	"="

Algoritmo: lexer

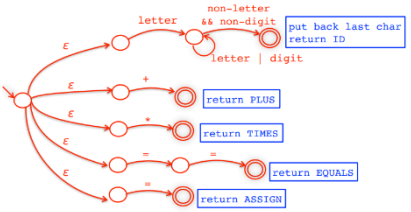
Descrizione: questa è la parte richiesta al designer del linguaggio

1. Definire un NFA per ogni lessema

Implementazione: questa è la parte che viene eseguita automaticamente da un generatore di lexer

2. Combinare la NFA individuando gli stati iniziali
3. Se l'NFA risultante in due non è deterministico, **trasformare l'automa in automa deterministico a stati finiti (DFA):** un DFA è un NFA (Q, ∑, δ, q_0, F) tale che δ è una funzione Q x ∑ ⟼ Q

    a. **Teorema della costruzione di sottoinsiemi [Morgensen]**: dato un NFA M, è possibile definire un DFA M' tale che L(M) = L(M').
    b. **Teorema dell'algoritmo di Hopcroft [Morgensen]:** dato un DFA M, è possibile definire un DFA M' con un insieme minimo di stati tale che L(M) = L(M').
4. Utilizzare le seguenti regole:
    a. Quando si raggiunge uno **stato finale**:
        i. Memorizzare la posizione in input (quindi è possibile leggere altri caratteri)
        ii. Continua a leggere altri caratteri in transito da uno stato all'altro
    b. Se **non sono possibili altre transizioni** con il carattere successivo: tornare all'ultimo stato finale (d'ora in poi esegui undo delle letture corrispondenti) e restituire il token corrispondente all'ultimo stato finale



Ambiguità

- **Problema:** il lexer raggiunge diversi stati finali
    - Esempio: "**if**" corrisponde sia a **ID** che a **IF** (parola chiave riservata)
- **Problema:** durante la lettura dei caratteri, gli automi lexer attraversano diversi stati finali
    - Esempio: **"="** corrisponde sia ad **ASSIGN** che **"=="** a **EQUAL**

Principio della corrispondenza più lunga: un lexer emette sempre il token che consuma la parte più lunga dell'input. Questo è importante quando si leggono identificatori e numeri (altrimenti i prefissi verrebbero riconosciuti anche come token)

Principio della prima corrispondenza: i token hanno sempre la priorità; quindi, il lexer può decidere quale token riconoscere se sono possibili due token per lo stesso input. Questo è importante quando si leggono le parole chiave (altrimenti potrebbero essere riconosciute come identificatori)

Errori nell'input

- **Problema:** rimuovere i lessemi illegali e stampare un messaggio di errore
- **Soluzione: rimuovi un carattere alla volta** e aggiungi un lessema che corrisponda a ogni carattere. **Questo lessema ha la priorità più**

bassa: l'azione corrispondente verrà eseguita quando nessun altro lessema viene riconosciuto

Rimuovi gli spazi bianchi \n, \t **e** \r**:** gli stati finali dei lessemi con questi caratteri sono **speciali**, non restituiscono un token ma invocano ricorsivamente il lexer (= tornare allo stato iniziale)

Specificare gli Atomi: Espressioni Regolari

Gli automi consentono di definire **visivamente** i lessemi che corrispondono a un token, ma **non sono adeguati** come linguaggio di specificazione

Una **descrizione equivalente** agli automi (DFA e NFA) sono le **grammatiche/espressioni regolari:** le grammatiche/espressioni regolari sono un modo compatto per definire un linguaggio accettato da un FA

Le grammatiche/espressioni regolari **vengono utilizzate come input** per i generatori di lexer: definiscono ogni lessema, comprese le sequenze di spazi bianchi e i commenti, che devono essere riconosciuti ma non associati a un token

Grammatica regolare: una grammatica (**N**, **T**, ⟼, S) è **regolare** se le sue produzioni ⟼ hanno la forma

- A ⟼ a
- A ⟼ aB
- A ⟼ ε

In **letteratura**, le grammatiche regolari sono anche chiamate grammatiche lineari a destra

Esempio: definizione dell'identificatore Java come grammatica regolare

- ID ⟼ ('a'..'z'| 'A'..'Z') CONT
- CONT ⟼ ('a'..'z'| 'A'..'Z'| '0'..'9' | '_') CONT
- CONT ⟼ ε

cont = (letter | digit | ‘_’) *

Teorema dal DFA alle grammatiche regolari: per ogni automa finito M, esiste una grammatica regolare G dove L(M) = L(G))

Algoritmo da DFA a grammatica regolare: i non terminali della grammatica sono gli stati degli automi (scritti in maiuscolo, per semplicità). Le produzioni sono

- se q $a\rightarrow$ q' negli automi e q' non è finale allora Q $\mapsto$ a Q' nella grammatica
- se q $a\rightarrow$ q' negli automi e q' è finale allora Q $\mapsto$ a Q' | a nella grammatica
- se q è iniziale e finale allora Q $\mapsto$ ε nella grammatica

Esempio: un identificatore java come espressione regolare

Definizione lessicale (in **inglese**): una lettera, seguita da zero o più lettere, cifre o simboli '_'

Definizione lessicale (come **espressione regolare**): letter (letter | digit | '_') *

- **ε** significa "stringa vuota"
- **|** significa "o"
- **string1 string2** significa "sequenza"
- * significa "ripetere 0 o più volte"
- **( )** significa "raggruppamento"

Esiste una precedenza tra gli operatori delle espressioni regolari: * ha la precedenza sulla concatenazione che ha la precedenza su |

Linguaggio definito da un'espressione regolare: insieme di stringhe che **corrispondono** all'espressione

Esempi

regular expressions	corresponding language
'00' | '1' | ε	{00, 1, ε}
'0'*	{ε, 0, 00, 000, …}
ε*	{ε}
('0'|'1')*	{ε, 0, 1, 00, 01, 10, …}
('0'|'1')('0'|'1')*	{0, 1, 00, 01, 10, …}
('1'|ε)('01')*('0'|ε)	sequenze anche vuote di 0 e 1 alternati

Gli operandi di un'espressione regolare

- Corrispondono alle etichette delle transizioni della FA
- Sono caratteri singoli tra apici o sequenze di caratteri tra apici, esempi: 'a' e 'while'
- Sono il carattere speciale ε (la stringa vuota)

Esempio:

letter: 'a' | 'b' | 'c' | ... | 'z' | 'A' | ... | 'Z'

digit: '0' | '1' | ... | '9'

In molti lexer (incluso ANTLR) puoi anche scrivere

letter: ('a'..'z') | ('A'..'Z')

digit: ('0'..'9')

Altri operatori utili di espressioni regolari

- Operatore **+** (una o più ripetizioni): **digit+** è uguale **a digit (digit)***
  - numeri_naturali: digit+;
- Operatore **?** (zero o una ripetizione): ('+' | '-')? è uguale a ε | '+' | '-'
  - integer: ('+' | '-')? numeri naturali
- (ANTLR) operatore **~('a'..'z')** sono i caratteri che sono diversi da **'a'..'z'**
- (ANTLR) operatore **.** sta per ogni carattere (quindi **.*** è ogni sequenza di caratteri)

Generatori Lexer

Input: le espressioni regolari che descrivono i lessemi

Generare codice (C, C++, Java, …) che implementa l'algoritmo completo di lexer:

- tradurre le espressioni regolari in FA
- unire la FA in un unico automa
- tradurre l'automa unito in un FA deterministico (più efficiente da simulare)
- produrre il codice che implementa la simulazione “speciale” del DFA (lookahead per regola di corrispondenza massima, priorità in caso di corrispondenza multipla, operazioni da eseguire al momento della corrispondenza e ritorno allo stato iniziale)

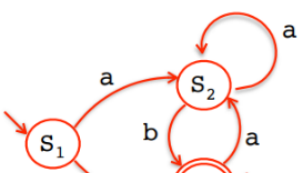
Implementazione Lexer

Un DFA può essere implementato da una tabella a 2 dimensioni: sia T

- una dimensione descrive gli “stati degli automi”
- l'altra dimensione descrive i "simboli di input"
- per ogni transizione degli automi $S_i \rightarrow a\ S_k$, è sufficiente definire T[i, a] = k

L'esecuzione del DFA **è molto efficiente:** se l'automa è nello stato S_i e il carattere di input è 'a', allora leggi T[i, a] = k e salta allo stato S_k

Esempio di tabella che implementa un DFA



ANTLR LEXER

Il lexer ANTLR (come ogni lexer): legge i caratteri finché non viene selezionata una regola; quindi, stampa il token corrispondente e poi ricomincia dal carattere successivo

La prima regola di corrispondenza più lunga

Se ci sono più regole che corrispondono all'input, quella selezionata è quella corrispondente alla stringa più lunga (in ANTLR, i non terminali che iniziano con una lettera maiuscola sono le regole lessicali (token)).

Esempio: SHORTTOKEN: **'abc'**;

LONGTOKEN: **'abcabc'**;

Sia **SHORTTOKEN** che **LONGTOKEN** corrispondono alla parte iniziale della stringa **abcabc**

- Poiché **LONGTOKEN** ha 6 caratteri e **SHORTTOKEN** ne ha solo 3, il lexer restituisce **LONGTOKEN**
- Se ci sono più regole che corrispondono, quella restituita è la prima nell'elenco

Esempio: SHORTTOKEN: **'a'**;

FIRSTTOKEN: **'abc'**;

SAMELENGTHTOKEN: **'ab'**. ;

Con l'input **abc**, **ANTLR** seleziona **FIRSTTOKEN**

Decisioni irreversibili

Una volta presa la decisione, il lexer **non esegue** il rollback

Esempio: in teoria la grammatica

SHORT: **'aaa'**;

LONG: **'aaaa'**;

Potrebbe dividere l'input **aaaaaa** nella sequenza **SHORT SHORT**, **ma**

- Il lexer sceglierà la corrispondenza più lunga e quindi riconoscerà **LONG**
- Poiché la coda aa non corrisponde a nessun token, il lexer emetterà gli errori: **start:1:4: errore di riconoscimento del token in: 'aa\n'**

Soliti errori

Il lexer sceglie il token successivo consumando i caratteri in **input senza far corrispondere completamente l'input**, cancellando le regole più brevi: **se il token selezionato non corrisponde all'input, viene segnalato un errore**

Esempio: l'input abcabQ con la grammatica

SHORTTOKEN: **'abc'**;

LONGTOKEN: **'abcabc'**;

- **SHORTTOKEN** corrisponde a 3 caratteri, **LONGTOKEN** corrisponde a più di 4 caratteri
- D'ora in poi ANTLR sceglie **LONGTOKEN**
- Purtroppo, **LONGTOKEN** non corrisponde all'input e quindi il lexer torna indietro e riconosce **SHORTTOKEN** dando un errore per abQ

L'esempio Simple

```
grammar Simple;
// THIS IS THE INPUT FOR THE PARSER
block     : '{' statement* '}';
statement: assignment ';'
         | deletion ';'
         | print ';'
         | block;
assignment    : ID '=' exp;
deletion : 'delete' ID;
print     : 'print' exp;
exp           : '(' exp ')'                          #baseExp
         | '-' exp                                   #negExp
         | left=exp op=('*' | '/') right=exp         #binExp
         | left=exp op=('+' | '-') right=exp         #binExp
         | ID                                        #varExp
         | NUMBER                                    #valExp;
// THIS IS THE INPUT FOR THE LEXER
fragment CHAR     : 'a'..'z' |'A'..'Z' ;
ID              : CHAR (CHAR | DIGIT)* ;
fragment DIGIT    : '0'..'9';
NUMBER          : DIGIT+;
// ESCAPE SEQUENCES
WS            : (' '|'\t'|'\n'|'\r')-> skip;
LINECOMENTS   : '//' (~('\n'|'\r'))* -> skip;
BLOCKCOMENTS  : '/*'( ~('/'|'*')|'/'~'*'|'*'~'/'|BLOCKCOMENTS)* '*/' -> skip;
```

nodes in the syntax tree are not "exp" but what is specified by #

no node in the syntax tree is generated! CHAR and DIGIT are collected in the tokens ID and NUMBER (they are not referenced from parser's rules)

no node in the syntax tree is generated! The characters are skipped

the syntax

parser rules start with lowercase letters, lexer rules with uppercase

```
grammar ArrayofInt;

init  : LPAREN value (COMMA value)* RPAREN ; THIS IS THE PARSER
value : init | INT ;                          PART — init is the initial
                                              symbol
INT : [0-9]+  ;
LPAREN : '{' ;          THIS IS THE LEXER PART
RPAREN : '}' ;          — rhs are always regular expressions —
COMMA : ',' ;

WS  : [ \t\r\n]+ -> skip ;  // define whitespace rule, toss it out
ERR : .          -> channel(HIDDEN) ;
```

Analisi Sintattica

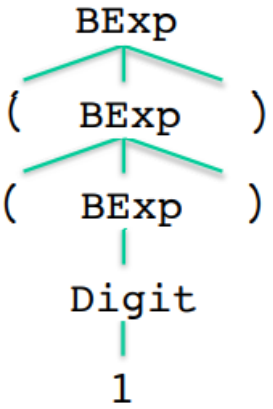
Derivazioni e Parse Trees

Prendi la grammatica BExp → (BExp)

BExp → Digit

Digit → 0 | 1 | . . . | 9

La derivazione BExp ⟹ (BExp) ⟹ ((BExp)) ⟹ ((Digit)) ⟹ ((1)) può essere rappresentata graficamente da **alberi** dove



- la radice (**root**) è il simbolo iniziale
- la foglia (**leaf**) è un terminale o ε
- ogni nodo interno (**internal node**) è un non terminale
- gli spigoli nodo-discendente (**edges node-descendant**) rappresentano una produzione

Questi alberi sono chiamati alberi di analisi (**parse trees**)

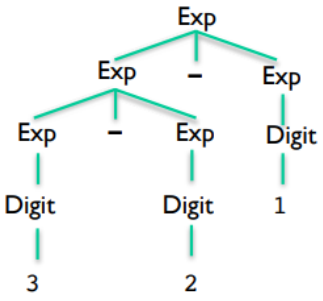
Parse Trees e ambiguità

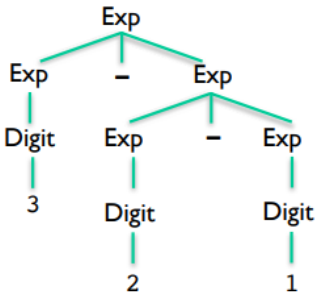
Le due derivazioni più a sinistra

```
Exp  ⟹ Exp - Exp
     ⟹ Exp - Exp - Exp
     ⟹ Digit - Exp - Exp
     ⟹ 3 - Exp - Exp
     ⟹ 3 - Digit - Exp
     ⟹ 3 - 2 - Exp
     ⟹ 3 - 2 - Digit
     ⟹ 3 - 2 -1
```

```
Exp  ⟹ Exp - Exp
     ⟹ Digit - Exp
     ⟹ 3 - Exp
     ⟹ 3 - Exp - Exp
     ⟹ 3 - Digit - Exp
     ⟹ 3 - 2 - Exp
     ⟹ 3 - 2 - Digit
     ⟹ 3 - 2 - 1
```

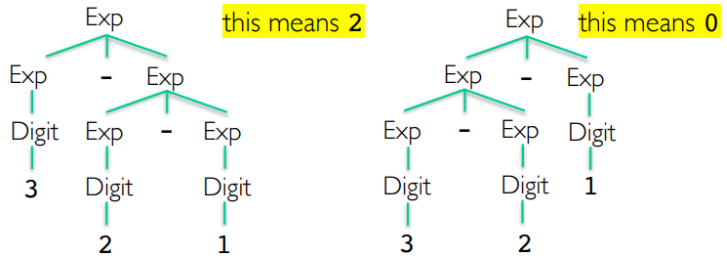
Corrispondono ai **due** alberi di analisi





Derivazioni a sinistra, parse trees e ambiguità

Grammatica ambigua: sia G una grammatica, se una stringa in L(G) ha **diverse derivazioni più a sinistra** (o più derivazioni più a destra) o è **rappresentata da diversi alberi di analisi**, allora G è **ambigua** (ambiguità significa **semantica diversa** della stessa frase). **L'ambiguità è problematica e deve essere risolta!**



Parse trees

- **Ha tutti i token**, inclusi quelli che il parser usa per il rilevamento
  - annidamento di sotto espressioni (come le parentesi)
  - segni di punteggiatura (punto e virgola, due punti, ecc.)
- Tecnicamente, gli alberi di analisi mostrano tutta la sintassi concreta
- Gli alberi di analisi non sono quasi mai costruiti in modo esplicito: **sono troppo prolissi**; vengono utilizzati durante i calcoli dei parser

Analisi

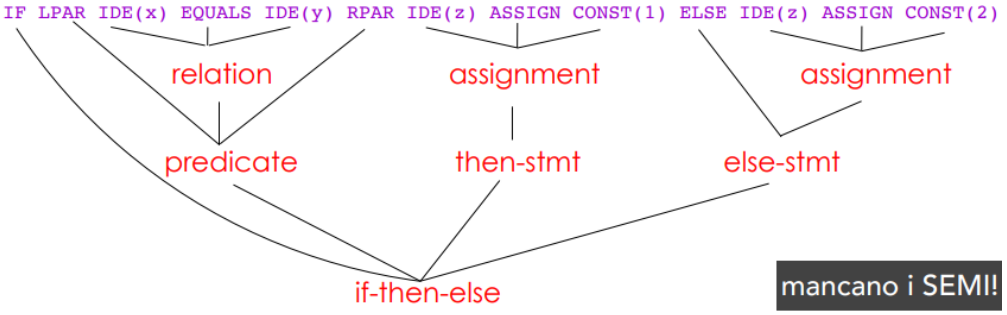
Una volta che le sequenze di caratteri sono state riconosciute in token, è necessario **analizzare la struttura sintattica** delle frasi/programmi per **verificare se appartengono o meno al linguaggio**

parsing = accetta le sequenze di token di input e restituisce **l'albero della sintassi astratta** (AST)

Esempio: if (x == y) z = 1; else z = 2;

Corrisponde alla sequenza di token (output di lexer)

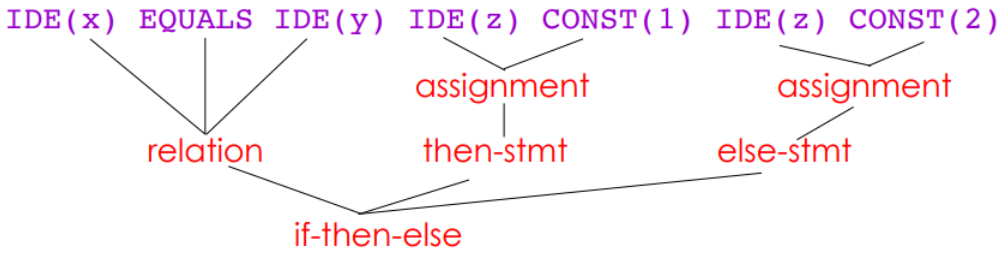
IF LPAR IDE(x) EQUALS IDE(y) RPAR IDE(z) ASSIGN CONST(1) SEMI ELSE IDE(z) ASSIGN CONST(2) SEMI



Abstract Syntax Tree (AST)

- Rimuovere i risultati parziali dell'analisi, cancellare i token inutili, appiattire l'albero rimuovendo i nodi interni, ecc.
- Tecnicamente, l'AST mostra una versione "astratta" della sintassi

Il **parser** restituisce l'**albero della sintassi astratto**: nell'albero della sintassi astratta vengono rimossi diversi token!



Progettazione di un parser

Può essere fatto "**a mano**", ovviamente: ok per i piccoli linguaggi, ma molto difficile per i veri linguaggi di programmazione

Oppure, come per il lexer, è possibile **utilizzare un generatore di parser automatico**: è necessario specificare la struttura sintattica del linguaggio (le produzioni) e il generatore emette il parser

Per quanto riguarda il lexer, **iniziamo con un parser fatto "a mano"** (così puoi capire perché è meglio usare un generatore di parser)

Primo esempio: la grammatica BExp

BExp → (BExp)

BExp → Digit

Digit → 0 | 1 | . . . | 9

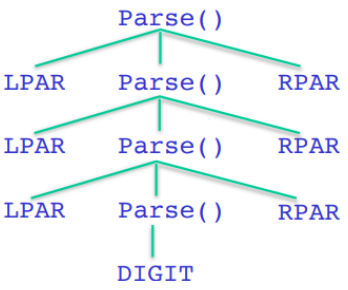
Preliminari del codice parser

- Sia **TOKEN** un tipo di dati enumerato che definisce i possibili token: **LPAR, RPAR, DIGIT**
- Sia **in[]** un array (globale) i cui elementi sono di tipo **TOKEN** e che rappresenta la sequenza di token restituiti dal lexer
- Sia **next** un numero intero (globale) che rappresenta l'indice della sequenza di token

Il codice parser fatto “a mano”

```
public void Parse() {
   next = next+1 ;
   TOKEN nextToken = in[next];
   if (nextToken == DIGIT) return() ;
   else if (nextToken == LPAR) {
            Parse();
            next = next+1 ;
            if (in[next] == RPAR) return() ;
             else System.out.print("syntax error")
            } else System.out.print("syntax error");
}
```

nextToken è inutile!



Non viene costruito un parse tree, tuttavia è possibile estendere il metodo **Parse** per costruire l'albero di analisi seguendo le invocazioni. **Ad esempio,** con input **(((1)))** il lexer restituisce **LPAR LPAR LPAR DIGIT RPAR RPAR RPAR** e il parser (esteso) compila:

Secondo esempio: Il linguaggio Exp

Exp → Exp - Exp

Esp → Digit

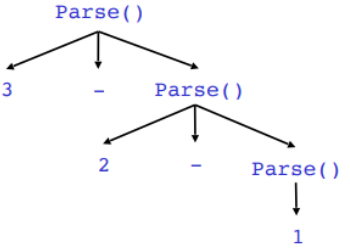
Digit → 0 | 1 | . . . | 9

Sia **TOKEN** un tipo di dati enumerato che definisce i possibili token (come prima), abbiamo anche **MINUS**, **DIGIT**

```
public void Parse(){
      next = next+1; TOKEN nextToken = in[next];
      if (nextToken==DIGIT) {
          if (in[next+1]==MINUS)  {
             next = next+1; Parse(); return();
          } else return();
       } else System.out.print("syntax error");
}
```

Continuano le espressioni di sottrazione

Un linguaggio più complesso: quindi, più difficile vedere come funziona il parser (e se funziona correttamente)



L'albero di analisi in realtà non è proprio quello che vogliamo: considera l'input 3-2-1

Abbiamo bisogno di una descrizione sintattica pulita: proprio come con lo scanner, scrivere il parser a mano è doloroso e soggetto a errori (considera di aggiungere +, *, / all'ultimo esempio)

Il Cosa: grammatica senza contesto (CFG)

Possiamo descrivere la struttura sintattica usando grammatiche prive di contesto!

I costrutti del linguaggio di programmazione hanno una **struttura ricorsiva:** questo è il motivo per cui anche il nostro parser scritto a mano aveva questa struttura

Esempio: un'espressione può essere: un numero, una variabile, un'espressione + espressione, un'espressione - espressione, o un (espressione), …

simple arithmetic expressions:

```
Exp → n | id | ( Exp )
    | Exp - Exp | Exp + Exp
```

I CFG sono una notazione naturale per la struttura ricorsiva della sintassi del linguaggio di programmazione

Grammatica per le espressioni parentesi bilanciate: BExp → '0'..'9' | (BExp)

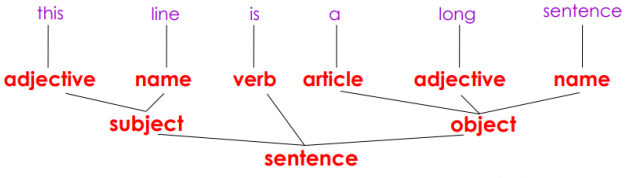
- descrive (genera) stringhe di simboli: **0, (1), ((1)), (((9))), ...**
- sono simili alle espressioni regolari ma possono fare riferimento a altre espressioni (here, BExp) e farlo ricorsivamente (thus, è "**stato non finito**")

Il Come: usare le derivazioni per il parsing

Un programma (una stringa di token) **non** ha **errori di sintassi** se può essere derivato dalla grammatica: finora sai solo come derivare qualche (qualsiasi) stringa e non sai come verificare se una determinata stringa è derivabile o no

Analisi: una volta riconosciuta la sequenza di caratteri come sequenza di token, occorre analizzare la struttura sintattica delle frasi/programmi per verificare se appartengono o meno alla lingua

- **parsing** = accetta sequenze di input di token e restituisce alberi di sintassi astratti (AST)
- **Esempio: albero di sintassi**



Confronto con l'analisi lessicale

Fase	Ingresso	Uscita
Lexer	Sequenza di caratteri	Sequenza di token
Parser	Sequenza di token	AST costruito dal parse tree

Il linguaggio SIMPLAN = **S**imple **Imp**erative **Lan**guage

- È un linguaggio imperativo con due tipi di dati (**int** e **bool**): istruzione di assegnazione standard **x = exp ;**
- Ammette dichiarazioni di variabili: dichiarazione standard **let int x = 4 ; in x+1**
- Ammette definizioni di funzioni: definizione standard **let int foo(int x) x+1; in foo(34) ;**
- Una funzione di libreria: **print**

Esempi

- print(5+3) ;
- let int x = 5; in print(x+3) ;
- let int f(int i, int j) i+j; in print(f(3,5));
- let int x = 1; bool b = false; in print (if (b) then { x+1 } else { x+2 });
- let int x = 1; bool b = true; in print (if (b) then { x+1 } else { x+2 });
- let int y = 5+2; bool f(int n, int m) let int x = m; in x==y; in print (if f(y,8) then { false } else { 10 });

Analisi della discesa ricorsiva e analisi LL

Analisi della discesa ricorsiva

Analizzare la sequenza di token cercando di ricostruire i passaggi di una **derivazione più a sinistra:** questi parser sono chiamati **top-down** perché imitano una visita anticipata dell'albero della sintassi, vale a dire dalla radicem alle foglie

Le regole per un A non terminale **definiscono un metodo** che riconosce A

- Il lato destro delle regole definisce la struttura del codice del metodo
- La sequenza di terminali e non terminali nelle regole corrisponde ad una verifica della corrispondenza terminali e ad invocazioni dei metodi corrispondenti ai simboli non terminali
- La presenza di regole diverse per A è implementata da un caso o da un if

Esempio: prendi la grammatica

E → T + E | T

T → (E) | (E) * T | int | int * t

I token restituiti dal lexer sono: lpar rpar plus times int(k) [k ∈ Nat]

Supponiamo di analizzare il flusso di token: **int(5) times int(2)**

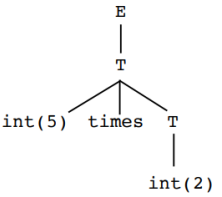
L'analisi inizia con l'espansione del simbolo iniziale E e **ogni regola** di E viene verificata, una alla volta ...

Prova E → T + E

1. Quindi controlli la **prima regola** per T: T → (E), ma non c'è corrispondenza con il token di input int(5)
2. Quindi controlli la **seconda regola** per T: T → (E) * T, ma non c'è corrispondenza con il token di input int(5)
3. Quindi controlli la **terza regola** per T: T → int, c'è corrispondenza con il token di input int(5), ma non c'è corrispondenza con il token **plus** dopo **T** e il token **times** del flusso di input

4. Quindi controlli la **quarta regola** per T: T → int *T, c'è corrispondenza con int(5) e poi times e int(2), ma non c'è corrispondenza con il token plus perché il flusso di input termina
5. **Abbiamo saturato le scelte per** T **senza riuscirci**, torniamo alle altre scelte per E

Quindi prova E → T ed esegui gli stessi passaggi fatti per E → T + E: l'analisi riesce con la regola **T → int*T** e **T → int** (a destra l'albero di analisi restituito)



Implementazione: definisci un metodo booleano che verifica le corrispondenze del flusso di token, ogni metodo viene incrementato **successivamente**

- Verificare la corrispondenza con un determinato **terminale**

```
public boolean term(TOKEN tok) {
    TOKEN x = in[next] ;
    next = next + 1 ;
    return x == tok;
}
```

- Verifica la corrispondenza con una regola di S (diciamo l'n-esimo)

```
public boolean S_n() { ... }
```

- Verificare la corrispondenza con una qualsiasi regola di S:

```
public boolean S() { ... }
```

for the rule E → T + E

```
public boolean E_1( ) {
    return (T() && term(plus) && E()) ;
    }
```

this corresponds to

```
boolean B1 = T();
boolean B2 = B1 && term(plus);
return(B2 && E()) ;
```

for the rule E → T

```
public boolean E_2( ) {
    return T();
    }
```

for (all) the rules of E (with backtracking)

backtrack!

```
public boolean E() {
   int saved = next ;
   if (E_1()) return true ;
     else { next = saved ; return (E_2()) ; }
     }
```

Metodi per il non terminale T

```
public boolean T_1() {
   return ( term(lpar) && E() && term(rpar) ); }
public boolean T_2() {
   return ( term(lpar) && E() && term(rpar) &&
         term(times) && T() ); }
public boolean T_3() { return ( term(int) ); }
public boolean T_4() {
   return ( term(int) && term(times) && T() ); }
public boolean T() {
  int saved = next;
  if (T_1()) return true ;
  else { next = saved ;
         if (T_2()) return true ;
         else { next = saved ;
                  if (T_3()) return true ;
                 else { next = saved ; return T_4() ; }
                }
          }
}
```

Per attivare l'analisi: inizializza **next** in modo tale che punti al primo token e invoca E(). Supponiamo che un carattere speciale $ rappresenti la fine del flusso di input nell'array in[]. L'analisi **termina con successo** se, al termine dell'esecuzione, in[next] == $. L'esecuzione dell'analisi ricorsiva della discesa coincide con l'esecuzione astratta calcolata all'inizio. Questo è semplice da implementare (anche a mano) **ma non funziona, a volte**!

Grammatica sinistra-ricorsiva: prendi la regola "S → S a" e prova ad analizzare questa regola nell'analisi della discesa ricorsiva

- Una grammatica (**N**, **T**, →, S) è **ricorsiva a sinistra** se esiste A ∈ **N** tale che A ⟹+ A γ, per alcuni γ
- L'analisi ricorsiva della discesa **non funziona** per le grammatiche ricorsive a sinistra, perché esegue **un ciclo infinito.** In questi casi è necessario **modificare la grammatica rimuovendo la ricorsione a sinistra**

La strategia di analisi è **estremamente semplice**: nel caso sia necessario **rimuovere la ricorsione a sinistra**... ma questa operazione può essere eseguita **automaticamente**

Non è **comune** perché utilizza il backtracking: è molto inefficiente e in pratica il backtracking può essere ridotto o eliminato modificando la grammatica (**fattorizzazione a sinistra**)

Va **bene per piccole grammatiche:** l'ordine delle produzioni è importante anche dopo l'eliminazione della ricorsione a sinistra.

Parser predittivi

Motivazioni: per evitare il backtracking, sarebbe utile se il parser di discesa ricorsiva **conosce la prossima produzione da espandere**

Sostituire il codice

```
saved = next;
if (E_1()) return true;
else { next = saved; return E_2(); }
```

con

```
switch (something) {
    case L1: return E_1();
    case L2: return E_2();
```

```
                default: System.out.print("errore di sintassi");
        }
```

“something”, L1, L2 sono definiti da un **lookahead** (analisi dei prossimi token)

Sono simili ai parser a discesa ricorsiva tranne per il fatto che possono **prevedere** quale produzione utilizzare guardando i prossimi token e senza tornare indietro

I parser predittivi accettano grammatiche LL(k).

- L significa scansione dell'input "da sinistra a destra".
- L significa "derivazione più a sinistra"
- k significa "prevedere usando k token di previsione"

ANTLR utilizza LL(*), una tecnica sofisticata che considera tutti i token necessari

Parsing predittivo e factoring di sinistra

La grammatica

$$E \rightarrow T + E \mid T$$

$$T \rightarrow (E) \mid (E)*T \mid int \mid int * t$$

È impossibile da prevedere perché la T non terminale ha **due produzioni** che iniziano con "(" e **due produzioni** che iniziano con "int", la E non terminale ha le due produzioni che iniziano con T e **non è evidente come predire.** Questa grammatica **deve essere fattorizzata a sinistra** prima di utilizzare parser predittivi

PARSER LL(1)

Utilizziamo una **tabella** LL(1) e uno **stack di analisi:** la tabella LL(1) sostituirà l'istruzione di commutazione e lo stack di analisi sostituirà lo stack

the LL(1) parsing table of

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

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

di chiamate

- **Per la voce** [E, int]**:** quando il non terminale nello stack è E e il prossimo token in input è int, utilizziamo la produzione E → T X
- **Per la voce** [Y, +]**:** quando il non-terminale sullo stack è Y e il prossimo token in input è + allora rimuovi Y
- Le voci vuote indicano un **errore**: esempio [E, *]

La tabella di analisi

La tecnica è **simile alla discesa ricorsiva**, ma invece del non determinismo (e del backtrack) per ogni S non terminale, guarda il token successivo, diciamo a, e la voce [S, a] nella tabella

Utilizziamo uno **stack** per registrare i terminali e non terminali nel dx della produzione in [S, a]

- L'input viene **rifiutato** quando viene rilevato uno stato errato (voce vuota nella tabella di analisi)
- L'input viene **accettato** quando l'immissione contiene un token di **fine input**

Pseudo-algoritmo di LL(1) parsing

```
add $ at the end of the array TOKENS ;
next = 0 ;
stack =<S $> ;
repeat
   switch stack
   case <X rest>: if (LL1_TABLE[X,TOKENS[next]] = α1…
  αn)
                        stack = <α1…αn rest>;
                    else  System.out.println("error") ;
        <t rest>: if (t == TOKENS[next]) {
                             stack = <rest> ;
                             next = next+1 ;
                    } else  System.out.println("error") ;
until (stack == < >)
```

LL(1) Parsing: esempio

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

La definizione della tabella di Parsing LL(1)

Sia G = (**N**, **T**, →, S), la sua tabella LL(1) è definita come segue:

1. Ha simboli non terminali nelle righe e simboli terminali nelle colonne

2. Per ogni regola X → ɣ in G e per ogni **t** tale che ɣ ⟹*t δ aggiungiamo la regola X → ɣ nella voce (X, t) (sembra difficile da calcolare)

3. per ogni regola X → ɣ in G tale che ɣ ⟹* ε aggiungi la regola X → ɣ nella voce (X, t), per ogni t tale che S ⟹* δ X t δ' (sembra difficile da calcolare)

Le grammatiche LL(1) sono quelle con tabelle di analisi LL(1) che **non hanno più voci**

Il predicato NULLABLE: sia G = (**N**, **T**, →, S) una grammatica senza contesto. NULLABLE è un predicato su (**N** U **T**)* definito come segue

$$\text{NULLABLE}(\gamma) = \begin{cases} \texttt{true} & \text{if } \gamma \Longrightarrow^* \varepsilon \\ \texttt{false} & \text{otherwise} \end{cases}$$

Esempio:

$$\begin{array}{llll} E \to T\ X & & X \to +\ E \mid \varepsilon \\ T \to (E)\ Y \mid Y & & Y \to *\ T \mid \varepsilon \end{array}$$

Quindi **NULLABLE(X) = NULLABLE(Y) = true**. Che dire di **NULLABLE(E)**?

Questa definizione **non è algoritmica**

- Definisci un **insieme di equazioni** per i non terminali
- Calcola la soluzione più piccola trovata con la tecnica del **punto fisso minimo**
- Risolvi simultaneamente le equazioni per tutti i lati destri delle produzioni

Il predicato NULLABLE: sia G = (**N**, **T**, →, S) una grammatica senza contesto. NULLABLE è il **più piccolo predicato** su (**N** U **T**)* definito come

1. $\text{NULLABLE}(\varepsilon) = \text{true}$
2. $\text{NULLABLE}(t) = \text{false}\,, \quad \text{for every} \quad t \in \mathbf{T}$
3. $$\text{NULLABLE}(\alpha\gamma) = \begin{cases} \text{NULLABLE}(\gamma) & \text{if} \quad \text{NULLABLE}(\alpha) = \text{true} \\ \text{false} & \text{if} \quad \text{NULLABLE}(\alpha) = \text{false} \end{cases}$$
4. $\text{NULLABLE}(X) = \bigvee_{X \to \gamma \text{ in } G} \text{NULLABLE}(\gamma)$

$\alpha \in \mathbf{N} \cup \mathbf{T}$

$\vee$ is logical or

segue

Questa è la definizione standard di **NULLABLE**, però si preferisce riscrivere l'elemento 3 in: **NULLABILE(α ϫ) = NULLABILE(α) ∧ NULLABILE(ϫ)**

Per esempio NULLABILE(α1…αn) = NULLABILE(α1) ∧ . . . ∧ NULLABILE(αn)

Esempio: grammatica

$$E \to T\ X \qquad X \to +\ E \mid \varepsilon$$
$$T \to (E)Y \mid Y \qquad Y \to *\ T \mid \varepsilon$$

Predicato NULLABLE

$$\begin{aligned} \text{NULLABLE}(E) &= \text{NULLABLE}(TX) \\ &= \text{NULLABLE}(T) \wedge \text{NULLABLE}(X) \end{aligned} \qquad = \text{false}$$

$$\begin{aligned} \text{NULLABLE}(X) &= \text{NULLABLE}(+E) \vee \text{NULLABLE}(\varepsilon) \\ &= \text{NULLABLE}(+E) \end{aligned} \qquad = \text{false}$$

$$\begin{aligned} \text{NULLABLE}(T) &= \text{NULLABLE}((E)Y) \vee \text{NULLABLE}(Y) \\ &= \text{NULLABLE}((E)Y) \\ &= \text{NULLABLE}(() \wedge \text{NULLABLE}(E) \wedge \text{NULLABLE}())) \wedge \text{NULLABLE}(Y) \end{aligned} \qquad = \text{false}$$

$$\begin{aligned} \text{NULLABLE}(Y) &= \text{NULLABLE}(*T) \vee \text{NULLABLE}(\varepsilon) \\ &= \text{NULLABLE}(*T) = \text{NULLABLE}(*) \wedge \text{NULLABLE}(T) \end{aligned} \qquad = \text{false}$$

grammar $E \to T\ X \qquad X \to +\ E \mid \varepsilon$
$T \to (E)Y \mid \text{int}\ Y \qquad Y \to *\ T \mid \varepsilon$

predicate NULLABLE

$$\begin{aligned} \text{NULLABLE}(E) &= \text{NULLABLE}(TX) \\ &= \text{NULLABLE}(T) \wedge \text{NULLABLE}(X) \end{aligned} \qquad = \text{false}$$

$$\begin{aligned} \text{NULLABLE}(X) &= \text{NULLABLE}(+E) \vee \text{NULLABLE}(\varepsilon) \\ &= \text{NULLABLE}(\varepsilon) \end{aligned} \qquad = \text{true}$$

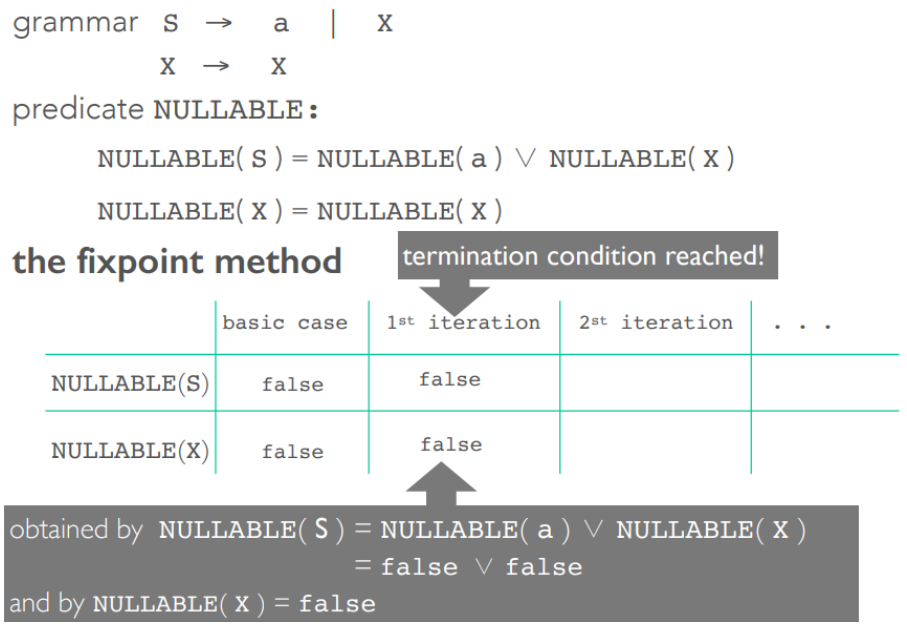
$$\begin{aligned} \text{NULLABLE}(T) &= \text{NULLABLE}((E)Y) \vee \text{NULLABLE}(\text{int}\ Y) \\ &= \text{NULLABLE}((E)Y) \\ &= \text{NULLABLE}(() \wedge \text{NULLABLE}(E) \wedge \text{NULLABLE}())) \wedge \text{NULLABLE}(Y) \end{aligned} \qquad = \text{false}$$

$$\begin{aligned} \text{NULLABLE}(Y) &= \text{NULLABLE}(*T) \vee \text{NULLABLE}(\varepsilon) \\ &= \text{NULLABLE}(\varepsilon) \end{aligned} \qquad = \text{true}$$

grammar $Z \to b \mid X\ Y\ Z$
$X \to Y \mid a$
$Y \to \varepsilon \mid c$

predicate NULLABLE

$$\text{NULLABLE}(Z) = \text{NULLABLE}(b) \vee \text{NULLABLE}(XYZ) = ??? \qquad [\text{false}]$$



Il metodo fixpoint con booleani

La teoria si applica anche agli ordini parziali infiniti (**cpos**, in quel caso la monotonia è sostituita dalla **continuità**)

Permette di risolvere equazioni come **x = ψ(x)** purché

1. Il **dominio** (es. i valori che x può assumere) di x è un ordine **parziale finito con un minimo elemento** (un insieme con una **relazione** riflessiva, antisimmetrica e transitiva)

2. Le operazioni utilizzate da **ψ** sono **monotone** rispetto alla relazione di ordine parziale

Nel nostro caso

1. il dominio è {true, false} con la relazione di ordinamento false ≤ true

2. **ψ** usa l'operatore booleano ∨ e ∧, che sono monotoni, ad es. se A ≤ A' e B ≤ B' allora A ∨ B ≤ A' ∨ B' e A ∧ B ≤ A' ∧ B'

L'algoritmo del metodo fixpoint per risolvere x = ψ(x)

- Inizializzazione: inizia con x = false (false è l'elemento minimo)
- Prima iterazione: calcola ψ(falso), per definizione è maggiore (o uguale a) falso; assegna questo valore a x
- Seconda iterazione: calcola ψ(ψ(falso)), questo è maggiore (o uguale a) ψ(falso) per monotonia (perché falso ⊆ ψ(falso) implica ψ(falso) ⊆ ψ(ψ(falso))); assegna questo valore a x
- ...: la catena non può crescere per sempre perché il dominio è finito. Pertanto, esiste n tale che ψn(falso) = ψ(ψn(falso))

- ψn(false) è detto **punto fisso minimo**

La funzione FIRST: sia G = (**N**, **T**, →, S) una grammatica senza contesto. FIRST è il **punto fisso minimo** che soddisfa le seguenti equazioni

1. $\mathtt{FIRST}(\varepsilon) = \{\varepsilon\}$
2. $\mathtt{FIRST}(t) = \{t\}$, for every $t \in \mathbf{T}$
3. $\mathtt{FIRST}(\alpha\gamma) = \begin{cases} \mathtt{FIRST}(\alpha) & \text{if } \mathtt{NULLABLE}(\alpha) = \mathtt{false} \\ \mathtt{FIRST}(\alpha)\setminus\{\varepsilon\} \cup \mathtt{FIRST}(\gamma) & \text{if } \mathtt{NULLABLE}(\alpha) = \mathtt{true} \end{cases}$
4. $\mathtt{FIRST}(X) = \bigcup_{X \to \gamma \text{ in } G} \mathtt{FIRST}(\gamma)$

$\alpha \in \mathbf{N} \cup \mathbf{T}$

Questa è la definizione standard di FIRST (in MORGENSEN, FIRST(ε) = ∅): come per NULLABLE, la voce tre è un dispiegamento degli altri casi con elementi in **N** U **T**

Set FIRST: esempi

grammar

$E \to T\ X \qquad X \to +\ E \mid \varepsilon$
$T \to (E)\ Y \mid \mathtt{int}\ Y \qquad Y \to *\ T \mid \varepsilon$

sets FIRST

$\mathtt{FIRST}(+) = \{+\}$ $\mathtt{FIRST}(*) = \{*\}$ $\mathtt{FIRST}(() = \{(\}$
$\mathtt{FIRST}()) = \{)\}$ $\mathtt{FIRST}(\mathtt{int}) = \{\mathtt{int}\}$

$\mathtt{FIRST}(T) = \mathtt{FIRST}((E)\ Y) \cup \mathtt{FIRST}(\mathtt{int}\ Y) = \{(, \mathtt{int}\}$
$\mathtt{FIRST}(X) = \mathtt{FIRST}(+\ E) \cup \mathtt{FIRST}(\varepsilon) = \{+, \varepsilon\}$
$\mathtt{FIRST}(Y) = \mathtt{FIRST}(*\ T) \cup \mathtt{FIRST}(\varepsilon) = \{*, \varepsilon\}$
$\mathtt{FIRST}(E) = \mathtt{FIRST}(T\ X) = \mathtt{FIRST}(T) = \{\mathtt{int}, (\}$

grammar

$S \to (S)\ S \mid \varepsilon$

sets FIRST

$\mathtt{FIRST}(() = \{(\}$ $\mathtt{FIRST}()) = \{)\}$

$\mathtt{FIRST}(S) = \mathtt{FIRST}((S)S) \cup \mathtt{FIRST}(\varepsilon) = \{(, \varepsilon\}$

grammar

$Z \to b \mid X\ Y\ Z$
$X \to Y \mid a$
$Y \to \varepsilon \mid c$

sets FIRST:

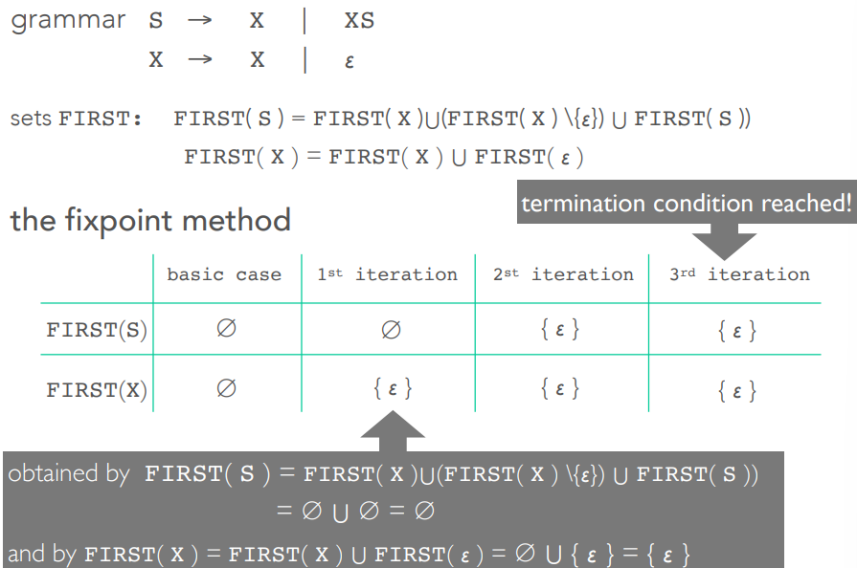
$\mathtt{FIRST}(a) = \{a\}$ $\mathtt{FIRST}(b) = \{b\}$ $\mathtt{FIRST}(c) = \{c\}$

$\mathtt{FIRST}(Y) = \mathtt{FIRST}(\varepsilon) \cup \mathtt{FIRST}(c) = \{c, \varepsilon\}$

$\mathtt{FIRST}(X) = \mathtt{FIRST}(Y) \cup \mathtt{FIRST}(a) = \{a, c, \varepsilon\}$

$\mathtt{FIRST}(Z) = \mathtt{FIRST}(b) \cup \mathtt{FIRST}(XYZ)$
$= \{b\} \cup \mathtt{FIRST}(X)\setminus\{\varepsilon\} \cup \mathtt{FIRST}(YZ)$
$= \{b, a, c\} \cup \mathtt{FIRST}(Z) = \{b, a, c\}$

L'algoritmo per il calcolo **FIRST** può utilizzare **un metodo a punto fisso**



Il metodo fixpoint con i set (finiti)

1. Il dominio è P(**T** U { ε }), il powerset di **T** U { ε } (es. l'insieme dei sottoinsiemi dei terminali **T** e ε), e la relazione di ordinamento è ⊆ la relazione di contenimento

2. Utilizza l'operatore di unione U, che è monotono, ad es. se A ⊆ A' e B ⊆ B' allora A U B ⊆ A' U B' e l'operatore setminus A\{ε} anch'esso monotono [setminus in not monotone sul secondo argomento]

L'algoritmo del metodo fixpoint per risolvere **x = ψ(x)**

- Inizializzazione: inizia con x = ∅ (∅ è l'elemento minimo)
- Prima iterazione: calcola ψ(∅), per definizione è maggiore (o uguale a) ∅; assegna questo valore a x
- Seconda iterazione: calcola ψ(ψ(∅)), questo è maggiore (o uguale a) ψ(∅) per monotonia (perché falso ⊆ ψ(∅) implica ψ(∅) ⊆ ψ(ψ(∅)))); assegna questo valore a x
- ...: la catena non può crescere per sempre perché il dominio è finito. Pertanto, esiste n tale che ψn(∅) = ψ(ψn(∅))
- ψn(∅) è detto **punto fisso minimo**

Definizione operativa di FIRST: sia G = (**N**, **T**, →, S) e α **N** ∪ **T**, FIRST(α) è definito come segue

1. **Inizializzazione:**

- se α è un terminale t allora FIRST(α) = {t}
- se α è un X non terminale allora FIRST(α) = ∅

2. **Ciclo:** ripetere fino a quando non viene modificato FIRST(α).

- per ogni regola X → ε, FIRST(X) = FIRST(X) ∪ {ε}
- per ogni regola X → α1 …αn e ogni 1 ≤ i ≤ n, se ε ∈ FIRST(α1), ..., ε ∈ FIRST(αi-1) quindi FIRST(X) = FIRST(X) ∪ FIRST(αi) \ {ε}
- per ogni regola X → α1 …αn, se ε ∈ FIRST(α1), ..., ε ∈ FIRST(αn) quindi FIRST(X) = FIRST(X) ∪ {ε}

La funzione FOLLOW: sia G = (**N**, **T**, →, S) una grammatica senza contesto. FOLLOW è il **punto fisso minimo** che soddisfa le seguenti equazioni

1. $\texttt{FOLLOW}(S) = \{ \$ \}$
2. $\texttt{FOLLOW}(X) = \bigcup_{Z \to \delta X \gamma \text{ in } G} \texttt{FIRST}(\gamma) \setminus \{\varepsilon\} \cup \bigcup_{Z \to \delta X \gamma \text{ in } G \text{ and } \texttt{NULLABLE}(\gamma)} \texttt{FOLLOW}(Z)$

1. Quando il simbolo iniziale non compare sul dx delle produzioni, “$” è il simbolo unico nel suo FOLLOW

2. FOLLOW non contiene mai “ε”

3. FOLLOW è definito solo per i **non terminali**: potremmo trasferire la definizione ai terminali ma questo è inutile per la tabella LL(1)

Set FOLLOW: esempi

grammar

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

sets FOLLOW

FOLLOW(E) = { $ } ∪ FOLLOW(X) ∪ {) } = { $,) }

FOLLOW(T) = FIRST(X)\{ε} ∪ FOLLOW(E) ∪ FOLLOW(Y)
= {+, $,) }

FOLLOW(Y) = FOLLOW(T) = {+, $,) }

FOLLOW(X) = FOLLOW(E) = { $,) }

grammar

S → (S) S | ε

FOLLOW sets

FOLLOW(S) = { $ } ∪ FIRST( ) S) ∪ FOLLOW(S)
= { $,) } ∪ FOLLOW(S)

grammar

Z → b | X Y Z
X → Y | a
Y → ε | c

FOLLOW sets

FOLLOW(Z) = { $ } ∪ FOLLOW(Z) = { $ }

FOLLOW(X) = FIRST(Y)\{ε} ∪ FIRST(Z) = { c } ∪ {b, a, c}

FOLLOW(Y) = FIRST(Z)\{ε} ∪ FOLLOW(X) = {b, a, c} ∪ {b, a, c}

grammar

$$S \rightarrow X \mid XS$$
$$X \rightarrow X \mid \varepsilon$$

sets FOLLOW:

$$FOLLOW(S) = \{\$\} \cup FOLLOW(S)$$
$$FOLLOW(X) = FIRST(S) \setminus \{\varepsilon\} \cup FOLLOW(S) \cup FOLLOW(X)$$

the fixpoint algorithm:

	basic case	1st iteration	2st iteration	3rd iteration
FOLLOW(S)	∅	{ $ }	{ $ }	{ $ }
FOLLOW(X)	∅	∅	{ $ }	{ $ }

Definizione di LL(1) tabelle di analisi

La tabella di analisi LL1G per una grammatica G: per ogni A → α in G fare:

1. per ogni terminale a ∈ FIRST(α) fai LL1G[A, a] = A → α

2. se ε ∈ FIRST(α), per ogni b ∈ FOLLOW(A) fai LL1G[A, b] = A → α

Questa regola si applica anche a $: se ε ∈ FIRST(α) e $ ∈ FOLLOW(A) fai LL1G[A, $] = A → α

Esempio

take the grammar

$$E \rightarrow T\ X \qquad X \rightarrow +\ E \mid \varepsilon$$
$$T \rightarrow (E)\ Y \mid int\ Y \qquad Y \rightarrow *\ T \mid \varepsilon$$

where in the line of Y we put $Y \rightarrow *T$?
- in the columns of FIRST(*T) = { * }

where in the line of Y we put $Y \rightarrow \varepsilon$?
- in the columns of FOLLOW(Y) = { $, +,) }

	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

Se una voce è **definita moltiplicata**, G non è LL(1) in particolare quando G è **ricorsivo a sinistra**, G **non** è **a sinistra**, G è **ambiguo** e anche in altri casi

La maggior parte delle grammatiche del linguaggio di programmazione non sono LL(1): ci sono strumenti che creano tabelle LL(1). Il generatore di parser ANTLR utilizza l'approccio LL

Rimozione della ricursione sinistra: caso di **recursione sinistra diretta**, ovvero esiste A tale che (la grammatica è equivalente all'espressione regolare (δ1| . . . | δn) (𝛄1 | . . . | 𝛄 m)*)

$$\left.\begin{array}{ll} A \to A\gamma_1 & A \to \delta_1 \\ \vdots & \vdots \\ A \to A\gamma_m & A \to \delta_n \end{array}\right\} \delta_1 \ldots \delta_n \text{ do not start with } A$$

is rewritten into

$$\begin{array}{lll} A \to \delta_1 A* & A* \to \gamma_1 A* & A* \to \varepsilon \\ \vdots & \vdots & \\ A \to \delta_n A* & A* \to \gamma_m A* & \end{array}$$

- poiché gli δ_i non iniziano con A non c'è più la ricorsione diretta a sinistra
- poiché la A* è un nuovo non terminale, la 𝛄i **non può iniziare con essa**
- potrebbero esserci **ricorsioni a sinistra indirette** se, per alcuni i, NULLABLE(𝛄i)

Esempio

$$\begin{array}{l} E \to E + F \\ E \to E - F \\ E \to F \\ F \to F * T \\ F \to F / T \\ F \to T \\ T \to num \\ T \to (E) \end{array} \quad \text{is rewritten into} \quad \begin{array}{ll} E & \to F\ E* \\ E* & \to +\ F\ E* \\ E* & \to -\ F\ E* \\ E* & \to \varepsilon \\ F & \to T\ F* \\ F* & \to *\ T\ F* \\ F* & \to /\ T\ F* \\ F* & \to \varepsilon \\ T & \to num \\ T & \to (E) \end{array}$$

Ci sono diverse possibilità

1. Caso di RECIPROCA SINISTRA-RICURSIONE:

$$\left.\begin{array}{ll} A_1 & \to A_2\gamma_1 \\ A_2 & \to A_3\gamma_2 \\ & \vdots \\ A_{k-1} & \to A_k\gamma_{k-1} \\ A_k & \to A_1\gamma_k \end{array}\right\}$$

break the mutual recursion, e.g. replace

$A_1 \to A_2\gamma_1$

with

$A_1 \to A_1\gamma_k \ldots \gamma_1$

and solve the direct left recursion

2. C'è una produzione: A → 𝛄A δ dove NULLABLE(𝛄)

3. Qualsiasi combinazione di 1 e 2

È sempre possibile riscrivere la ricorsione sinistra indiretta in quella diretta (il **processo è un po' complesso**)

Fattore di sinistra

La grammatica

$$E \rightarrow T + E \mid T$$

$$T \rightarrow (E) \mid (E)*T \mid int \mid int * t$$

È impossibile da prevedere perché

- la T non terminale ha **due produzioni** che iniziano con "(" e **due produzioni** che iniziano con "int"
- la E non terminale ha le due produzioni che iniziano con T e **non è evidente come predire**

La grammatica precedente **deve essere calcolata a sinistra** prima di utilizzare parser predittivi

Esempio:

the grammar

```
E  →  T + E | T
T  →  (E) | (E)*T | int  |  int * T
```

is left-factored as follows

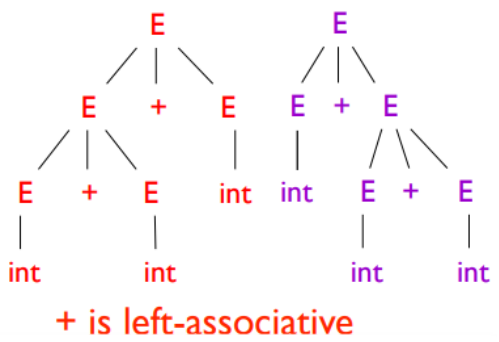
```
E   →  T E*
E*  → + E  |  ε
T   → (E) T* | int T*
T*  → *T  |  ε
```

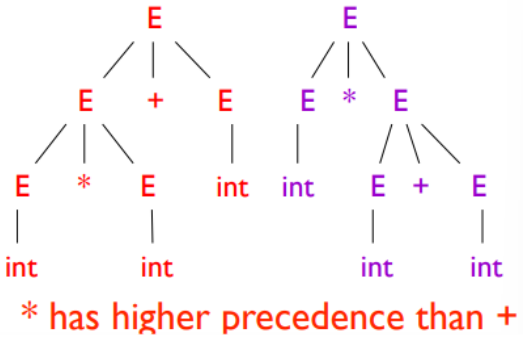
PROBLEM: left-factoring the standard `if-then-else` statement

```
Stat → if Exp then Stat else Stat  |  if Exp then Stat
```

Ambiguità: una grammatica è **ambigua** se ha **più di un albero di analisi** per una stringa, equivalentemente, c'è **più di una derivazione più a destra** o **più a sinistra** per una stringa

Esempio: E → E + E | E * E | (E) | int è ambiguo perché int+int+int int*int+int hanno due alberi di analisi





L'ambiguità è **cattiva:** lascia il significato di alcuni programmi mal definito

L'ambiguità è **comune** nei linguaggi di programmazione: espressioni aritmetiche, if-then-else

Nell'analisi LL è possibile gestire l'ambiguità: **riscrivere le grammatiche**

Non esistono tecniche generali per gestire l'ambiguità trasformando la grammatica (è sempre preferibile **non modificare una grammatica**)

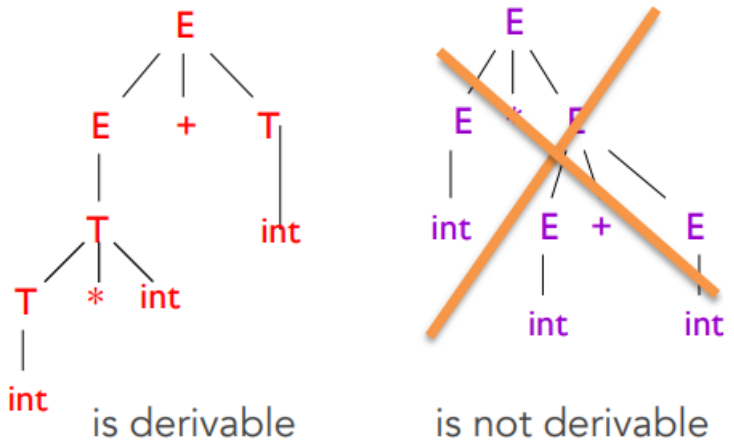
Invece di riscrivere la grammatica usa la grammatica più naturale (ambigua) insieme a **dichiarazioni disambiguate**

Affrontare l'ambiguità: riscrivi la grammatica per le espressioni in modo univoco

$$E \rightarrow E + T \mid T$$

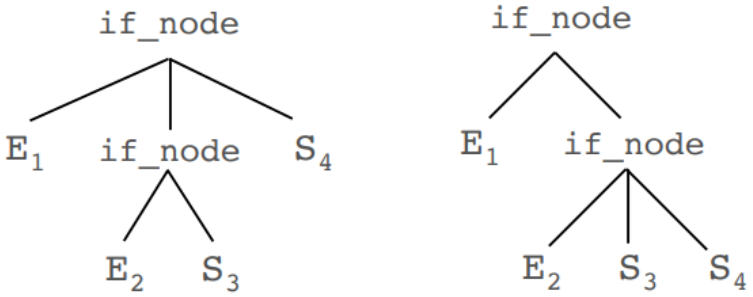
$$T \rightarrow T * int \mid T * (E) \mid int \mid (E)$$

- impone la precedenza di * su +
- impone l'associatività a sinistra di + e *



La nuova grammatica non è né LL(1) né adeguata per l'analisi discendente (ricorsione a sinistra)

La grammatica “S → ID '=' E | ‘if’ E 'then' S | 'if' E 'then' S 'else' S” è anche **ambigua** perché l'affermazione “if E1 then if E2 then S3 else S4” ha due



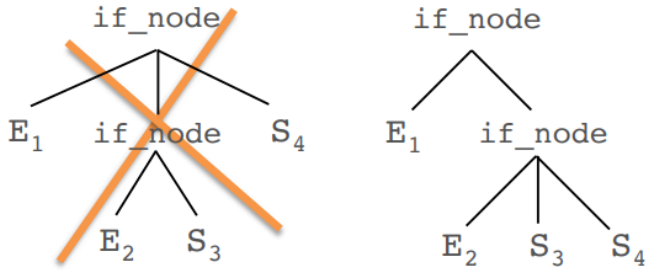
alberi di analisi (astratti). Nei linguaggi di programmazione vogliamo l'albero a destra

else corrisponde al più vicino senza eguali allora possiamo **fattorizzare** la parte if-then e **riscrivere** la grammatica come segue (questa nuova grammatica descrive lo stesso insieme di stringhe e consente le stesse derivazioni):

$$S \rightarrow ID = E \mid \text{if } E \text{ then } S \text{ ELSE}$$

$$ELSE \rightarrow \text{else } S \mid \boldsymbol{\varepsilon}$$

Esiste un hack standard per vietare questa derivazione: dare priorità al token "else".



Una soluzione teorica (vedi Gabbrielli-Martini)

```
statement           : statementNoTrailing
                    | ifThenStatement
                    | ifThenElseStatement
                    ;

statementNoTrailing : // the statements without if
                    ;
ifThenStatement     : 'if' '(' exp ')' 'then' statement ;

ifThenElseStatement : 'if' '(' exp ')' 'then' statementNoShortIf 'else'
                      statement ;

statementNoShortIf  : statementNoTrailing
                    | ifThenElseStatementNoShortIf  ;

ifThenElseStatementNoShortIf : 'if' '(' exp ')' 'then'
                               statementNoShortIf 'else' statementNoShortIf ;
```

La grammatica NON è LL(1): deve essere fattorizzata a sinistra, anche se è accettato in ANTLR!

ANTLR

Regola d'inizio: qualsiasi grammatica necessita di una cosiddetta regola di inizio (non è più così in ANTLR v4)

- la regola iniziale è una regola a cui non fa riferimento un'altra regola

- se la tua grammatica non ha tale regola, il generatore ANTLR emetterà un avviso: nessuna regola di partenza (nessuna regola può ovviamente essere seguita da EOF)

Per evitarlo, aggiungi una regola di inizio fittizia alla tua grammatica: **start_rule: someOtherRule ;**

Ambiguità esempio:

```
exp : exp '+' exp | exp '*' exp | NUM | '(' exp ')';

NUM: ('0'..'9')+;
```

Questa grammatica dovrebbe riconoscere gli input per una semplice calcolatrice

- ANTLR v4 si comporta male
- per capire il problema, ricorda che ANTLR va **da sinistra a destra** ogni volta che si analizza un input
  - **o** decide prima quale alternativa utilizzerà **seguendo l'ordine delle regole**
  - o poi si attiene alla decisione
- la fattorizzazione a sinistra viene risolta automaticamente da ANTLR

Prova a simulare in ingresso: 1 + 3 * 4

Corrisponde a un'alternativa **exp '+' exp** o a un'alternativa **exp '*' exp**?

L'errore suggerisce di riordinare le regole come segue:

```
exp : exp '*' exp | exp '+' exp | NUMBER | '(' exp ')';

NUMBER: ('0'..'9')+;
```

Problemi con l'associatività: estendere la grammatica con "/" e forzarla ad essere associativa corretta

```
exp : exp '/' exp | exp '+' exp | exp '*' exp | NUM | '(' exp ')';

NUM: ('0'..'9')+;
```

Soluzione: usa un nuovo non terminale!

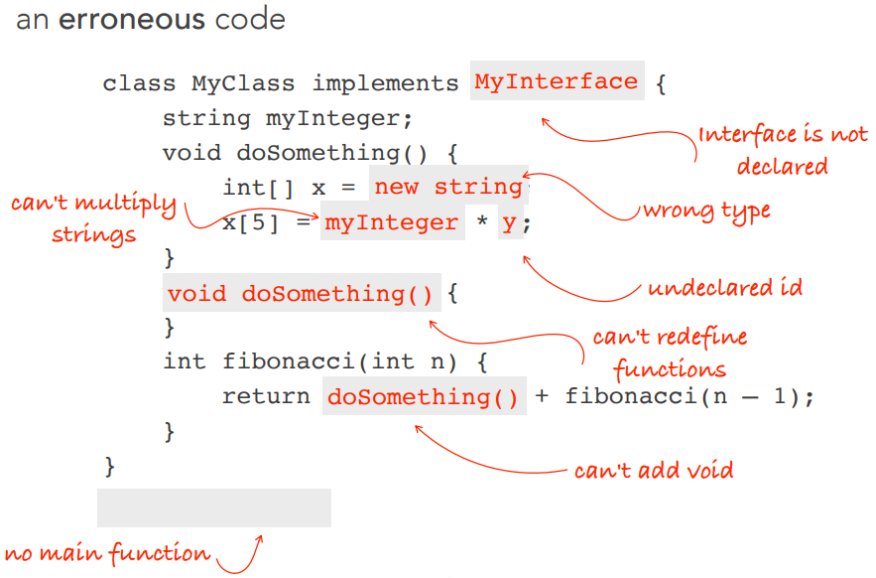
```
exp : term | exp '+' term ;

term : factor | factor '/' term | term '*' factor ;

factor : '(' exp ')' | NUM ;
```

Analisi Semantica: ambiti e tabelle dei simboli

Il compilatore finora

- **Analisi lessicale:** rileva gli input con token illegali, ad es. main£ ();
- **Parsing:** rileva gli input con **alberi di analisi mal formati**, ad es. punto e virgola mancanti

Lo scopo dell'analisi semantica: un esempio



Analisi semantica: cattura gli errori che non sono stati trovati dal lexer e dal parser

Tipici **errori semantici**:

- **Dichiarazioni multiple:** una variabile deve essere dichiarata (nello stesso ambito) al massimo una volta
- **Variabile non dichiarata:** una variabile non deve essere utilizzata prima di essere dichiarata
- **Tipo non corrispondente:** il tipo del lato sinistro di un compito deve corrispondere al tipo del lato destro
- **Argomenti sbagliati:** le funzioni/metodi dovrebbero essere chiamati con il numero e il tipo di argomenti corretti

Domanda: perché questi errori non possono essere rilevati prima?

Limitazioni della grammatica senza contesto

Utilizzando grammatiche prive di contesto

- come puoi prevenire le definizioni di identificatori **duplicate**?
- come differenziare le variabili di un tipo dalle variabili di un altro tipo?
- come garantiresti che le classi implementino tutti i metodi di interfaccia?

Per la maggior parte dei linguaggi di programmazione, questi sono **probabilmente** impossibili: usa il **pumping lemma per i linguaggi privi di contesto**

Tipi e terminologie

Type safety: un linguaggio è **indipendente dal tipo** se le sole operazioni che possono essere eseguite sui dati nella lingua sono quelle definite dal tipo di dati. L'imposizione del tipo può essere:

- **statica**, rilevando potenziali errori in fase di compilazione
- **dinamica**, associando informazioni sul tipo a valori in fase di esecuzione e consultandole secondo necessità per rilevare errori imminenti
- una **combinazione** di entrambe

Type system è un sistema formale costituito da un insieme di regole che assegna una proprietà denominata type alle operazioni di un programma

- **controllo del tipo statico** = eseguito in fase di compilazione
- **controllo dinamico del tipo** = eseguito in fase di esecuzione; associa ad ogni oggetto di runtime un tag di tipo contenente informazioni sul tipo che possono essere utilizzate anche per implementare downcasting, riflessione, ecc.
- combinazioni…

Type expressions: i tipi di dati possono essere definiti dai programmatori (tipi di dati strutturati)

- **Conformità ed equivalenza del tipo:** quando due tipi sono uguali? (nominale vs strutturale)

a table for mainstream languages

TYPE SYSTEMS

Language	Type Safety	Type Expr.	Type Comp. & Equiv.	Type Checking
C	weak	explicit	nominal	checking/inference
C#	weak	implicit/explicit	nominal	checking/inference
F#	strong	implicit	nominal	inference
Go	strong	implicit/explicit	structural	inference
Haskell	strong	implicit/explicit	nominal	inference
Java	strong	explicit	nominal	checking/inference
JavaScript	weak	implicit	no	dynamic
OCaml	strong	implicit/explicit	nominal	inference

Falsi positivi e falsi negativi

Supponiamo che ci sia un codice per il controllo del tipo: TypeCheck(P)

- TypeCheck(P) prende in input un programma P
- restituisce true se il programma è corretto rispetto ai tipi
- falso altrimenti

Falsi positivi: TypeCheck(P) = true e quando si esegue P, l'esecuzione termina con un errore di programma dovuto ai tipi (**i falsi positivi sono problematici**)

Falsi negativi: TypeCheck(P) = false e quando si esegue P, l'esecuzione non mostra mai un errore di tipo

- esempio: int x = 0; se (vero) x = 1; altrimenti x = vero;

Un semplice analizzatore semantica

Funziona in **due** fasi

1. attraversa l'AST creato dal parser e, per ogni **ambito** nel programma:
   - **elabora le dichiarazioni** ovvero **(a)** aggiunge nuove voci alla tabella dei simboli e **(b)** riporta qualsiasi variabile dichiarata moltiplicata
   - **elabora le istruzioni** che sono **(a)** trova usi di variabili non dichiarate e **(b)** aggiorna i nodi "ID" dell'AST in modo che puntino alla voce della tabella dei simboli appropriata
2. attraversa l'AST **di nuovo** ed elabora tutte le istruzioni nel programma
   - utilizza le informazioni della tabella dei simboli per determinare il **tipo** di ciascuna espressione e per trovare errori di tipo

Il controllo dell'ambito e **il controllo del tipo** sono le due sfide dell'analisi semantica. I moderni analizzatori semantici effettuano **diverse visite** (non solo due) dell'AST

Controllo dello scopo: che cosa c'è in un nome?

Lo stesso nome nei programmi delle lingue moderne può riferirsi a cose fondamentalmente diverse

Questo è un codice Java perfettamente legale:

```
public class A {
    char A;
    A A(A A) {
        A.A = 'A';
        return A((A) A);
    }
}
```

what all these 'A' are?

Questo è un codice C++ perfettamente legale:

```
int Awful() {
                int x = 137;
                {
                  string x = "Scope!";
                  if (float x = 0)
                          double x = x;
                }
                if (x == 137) cout << "Y";
}
```

what all these 'x' are?

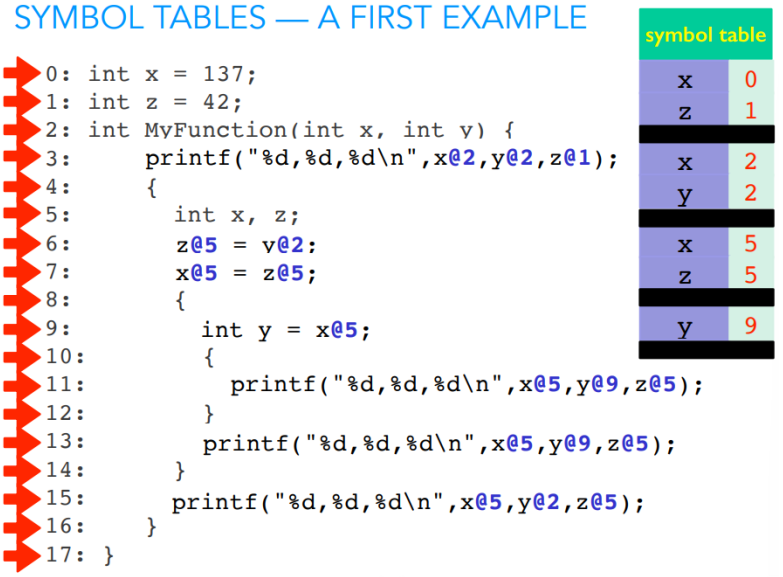
Campi di applicazione e tabelle dei simboli

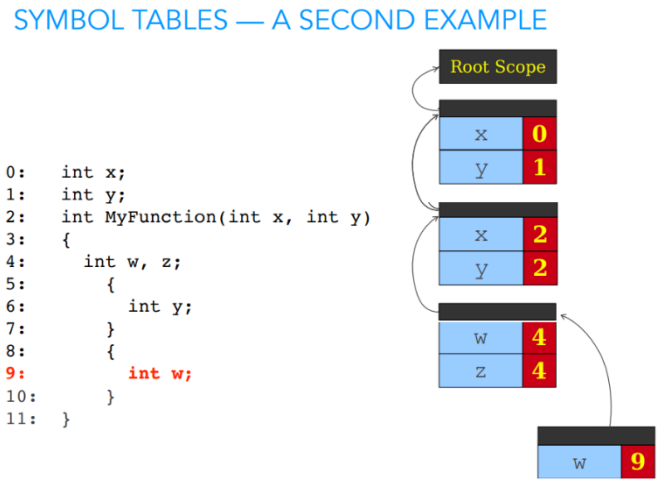
Lo **scopo di una dichiarazione** è l'insieme di posizioni in un programma in cui il nome si riferisce al nome della dichiarazione: l'introduzione di nuove variabili nell'ambito può nascondere i nomi precedenti, come possiamo tenere traccia di **ciò che è visibile**? Usiamo le **tabelle dei simboli**

- una tabella dei simboli è una mappa da un nome alla cosa a cui il nome si riferisce
- mentre eseguiamo la nostra analisi semantica, aggiorniamo continuamente la tabella dei simboli con informazioni su ciò che è nell'ambito
- il design della tabella dei simboli è **influenzato dal tipo di ambito** utilizzato dal linguaggio di programmazione

Domande:

- come appare in pratica la tabella dei simboli?
- quali operazioni devono essere definite su di esso e come lo implementiamo?





La struttura non menziona MyFunction, che non è corretta!

Pila di spaghetti

Logicamente, la tabella dei simboli è una **struttura collegata di ambiti**

- ogni ambito memorizza un puntatore ai suoi genitori, ma non viceversa
- da qualsiasi punto del programma, la tabella dei simboli sembra essere una pila
    - ogni punto del programma (es. ogni nodo dell'albero della sintassi) ha la propria tabella dei simboli
    - l'implementazione della tabella dei simboli può utilizzare puntatori tra i nodi per evitare la copia

Questa struttura logica è chiamata **pila di spaghetti**

I pacchi di spaghetti sono strutture statiche; gli stack espliciti sono una struttura dinamica

Semplificazione

D'ora in poi, supponiamo che la nostra lingua:

- utilizza l'**ambito statico**
- richiede che **tutti i nomi siano dichiarati** prima di essere utilizzati
- **non consente più dichiarazioni multiple** di un nome nello stesso ambito (anche per diversi tipi di nomi)
- consente di dichiarare **lo stesso nome in più ambiti nidificati** (ma solo una volta per ambito)
- utilizza lo stesso ambito per i parametri della funzione/del metodo e per le variabili locali dichiarate all'inizio del metodo

Implementazione della tabella dei simboli

Supponiamo che la tabella dei simboli verrà utilizzata per rispondere a due domande (ulteriore semplificazione):

1. data una dichiarazione di un nome**, esiste già una dichiarazione con lo stesso nome nell'ambito di applicazione attuale**?
2. dato l'uso di un nome, **a quale dichiarazione corrisponde (usando la regola "più strettamente annidato"), o non è dichiarato**? (Questo è rilevante anche quando si genera il codice perché è necessario memorizzare un offset nel record di attivazione, pertanto la tabella dei simboli, o parte di essa, deve essere conservata fino al termine della compilazione)

Di quale operazione abbiamo bisogno?

Date le ipotesi di cui sopra, avremo bisogno di:

- **insert** un nuovo nome nella tabella dei simboli con i suoi attributi
- **look up** un nome negli **ambiti correnti e di inclusione**
    - per verificare se è dichiarato moltiplicato
    - verificare l'uso di un nome non dichiarato, e
    - per collegare un utilizzo con la corrispondente voce della tabella dei simboli
- fare **ciò che deve essere fatto** quando viene **inserito un nuovo ambito**
- fare **ciò che deve essere fatto** quando **si esce da un ambito**

Due possibili implementazioni della tabella dei simboli

1. **un elenco di tabelle (hash)**
2. **una tabella (hash) di elenchi**

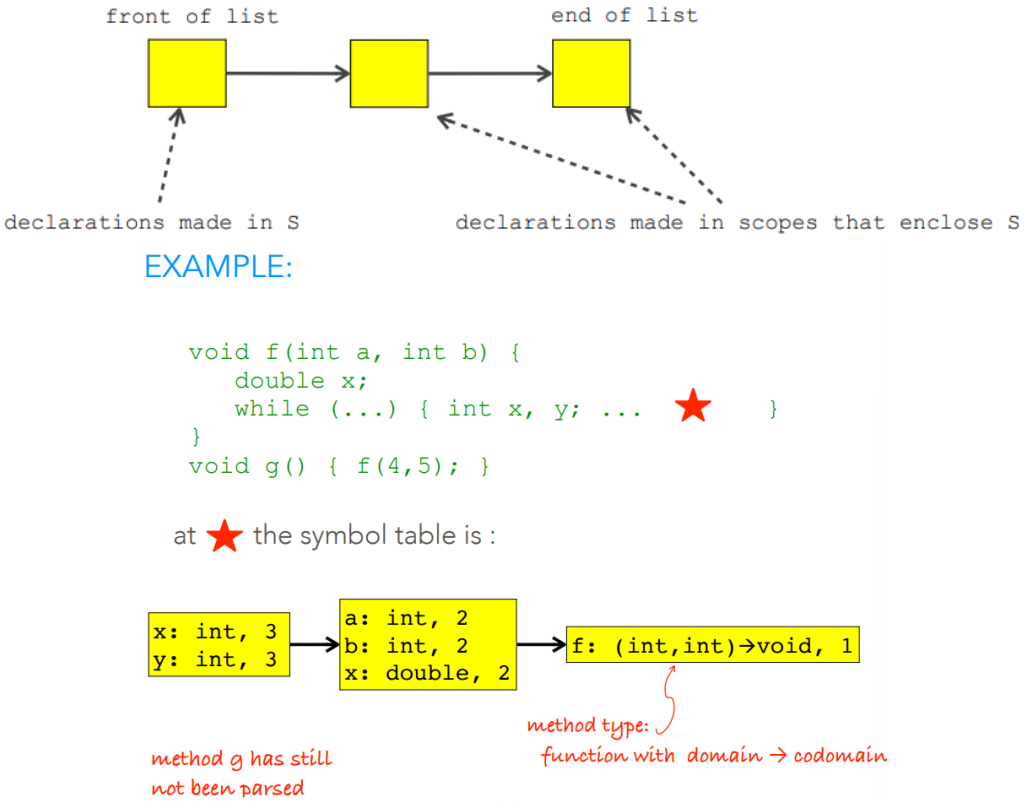
per ogni approccio, considereremo: cosa deve essere fatto durante l'elaborazione di una dichiarazione, durante l'elaborazione di un utilizzo, e quando si entra e si esce da un ambito.

Semplificazione: supponiamo che ogni voce della tabella dei simboli includa solo il nome del simbolo, il suo tipo e il livello di annidamento della sua dichiarazione

Implementazione 1: elenco di hashtables

- la tabella dei simboli **è** un elenco di tabelle hash
- una tabella hash **per ogni ambito attualmente visibile**

Durante l'elaborazione di un ambito S:



Elenco degli hashtables: le operazioni

1. **sull'ingresso nell'ambito:** incrementa **il numero del livello corrente** e **aggiunge** una nuova tabella hash vuota all'inizio dell'elenco
2. **elaborare una dichiarazione** di x**: look up** x nella prima tabella dell'elenco, se è presente, emette un errore "moltiplica variabile dichiarata"; in caso contrario, aggiungi x alla prima tabella dell'elenco
3. **per elaborare un uso** di x**:** look up x a partire dalla prima tabella dell'elenco, se non è presente, cerca x in ogni tabella successiva nell'elenco e se non è in nessuna tabella, emette un errore di "variabile non dichiarata".
4. in **uscita dall'ambito: rimuove** la prima tabella dall'elenco e **diminuisce** il numero del livello corrente

I nomi di funzioni/metodi appartengono alla tabella hash per l'ambito più esterno, non nella stessa tabella delle variabili del metodo. **Ad esempio**, nell'esempio sopra:

- il nome della funzione f è nella tabella dei simboli per l'ambito più esterno

- il nome f non è nello stesso ambito dei parametri aeb e della variabile x
- pertanto, quando viene elaborato l'uso del nome f nel metodo g, il nome viene trovato in una tabella dell'ambito di inclusione

La complessità computazionale delle operazioni

1. scope entry: tempo per inizializzare una nuova tabella hash vuota, probabilmente proporzionale alla dimensione della tabella hash

2. elaborare una dichiarazione: utilizzando l'hashing, tempo previsto costante (O(1))

3. elaborare un utilizzo: utilizzando l'hashing per eseguire la ricerca in ogni tabella nell'elenco, il tempo peggiore è O (profondità di annidamento), quando tutte le tabelle nell'elenco devono essere esaminate

4. **scope exit:** tempo per rimuovere una tabella dall'elenco, che dovrebbe essere O(1)

Esercizio

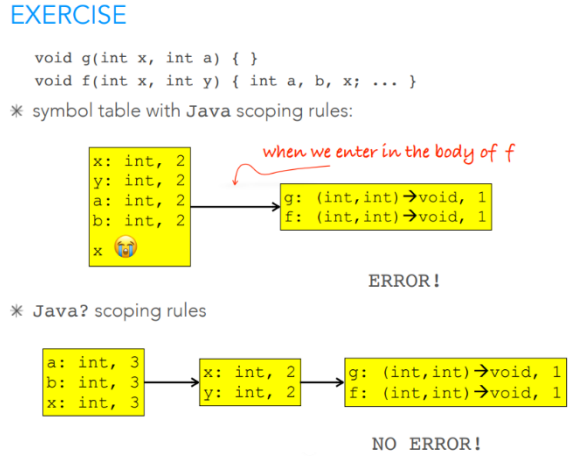
Supponi di avere **Java**? (un **linguaggio immaginario** diverso da Java) che permette ad una funzione di avere sia un parametro che una variabile locale con lo stesso nome (qualsiasi uso del nome nel corpo della funzione si **riferisce alla variabile locale**). Considera il codice:

void g(int x, int a) { }

void f(int x, int y, int z) { int a, b, x; ... }

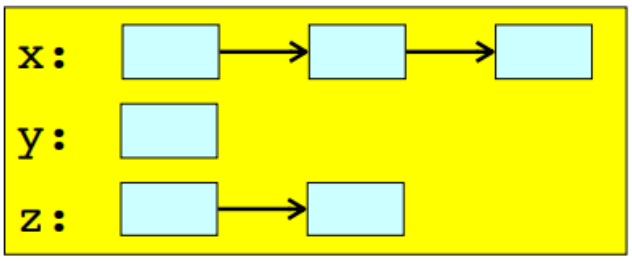
Disegna la tabella dei simboli come sarebbe dopo aver elaborato le dichiarazioni nel corpo di f sotto:

- le regole di scoping che abbiamo assunto
- Java? regole di ambito



Implementazione 2: hashtable degli elenchi

Quando si elabora un ambito S, la struttura della tabella dei simboli è



C'è solo **una grande tabella hash**, contenente una voce per ogni nome per il quale esiste: qualche dichiarazione in ambito S o in un ambito che racchiude S

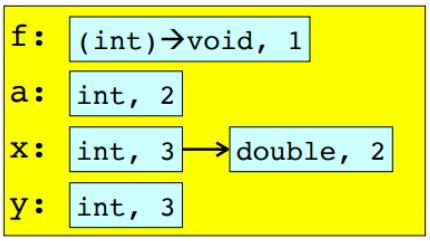
Ogni nome ha un elenco associato di voci della tabella dei simboli

- il primo elemento corrisponde alla dichiarazione più ravvicinata
- le altre voci dell'elenco corrispondono a dichiarazioni negli ambiti di inclusione

EXAMPLE

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

at ★ the symbol table is



the **nesting level information is crucial**

L'attributo level-number memorizzato in ciascuna voce dell'elenco ci consente di determinare se è stata fatta la dichiarazione più ravvicinata: nell'**ambito attuale** o in un **ambito racchiuso**

Hashtable delle liste: le operazioni

1. sull'**ingresso nell'ambito:** incrementare il **numero del livello corrente**
2. elaborare una **dichiarazione** di x**:** cerca x nella tabella dei simboli, se c'è x, prendi il numero del livello dal primo elemento dell'elenco, se quel numero di livello = il livello corrente, emette **un errore** "moltiplica variabile dichiarata". In caso contrario, **aggiunge** un nuovo elemento all'inizio dell'elenco con il tipo appropriato e il numero di livello corrente

3. per **elaborare un uso** di x**:** cerca x nella tabella dei simboli, se non è presente, emette un **errore di** "variabile non dichiarata".
4. in **uscita dall'ambito:** scansiona tutte le voci nella tabella dei simboli, guardando la prima voce di ogni lista, se il numero del livello dell'elemento = il numero del livello corrente, **lo rimuove dalla sua lista** (e se la lista diventa vuota, rimuovere l'intera voce della tabella dei simboli), infine, decrementa il **numero del livello corrente**

La complessità computazionale delle operazioni

1. **scope entry:** tempo per incrementare il numero di livello, O(1)
2. **elaborare una dichiarazione:** utilizzando hashing, tempo previsto costante (O(1))
3. **elaborare un uso:** utilizzando hashing, tempo previsto costante (O(1))
4. **uscita dall'ambito:** tempo proporzionale al numero di nomi nella tabella dei simboli

Esercizio

Supponiamo che la tabella dei simboli sia implementata utilizzando una **tabella hash di liste**

Disegna immagini per mostrare come cambia la tabella dei simboli quando viene elaborata ogni dichiarazione nel codice seguente

```
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;
               }
}
```

Scoping

Le **regole di portata di un linguaggio:**

- determinare quale dichiarazione di un identificatore corrisponde a ciascuna occorrenza dell'identificatore
- vale a dire, le regole di scoping associano le occorrenze degli identificatori alle loro dichiarazioni

C++ e Java usano l'**ambito statico:**

- la mappatura dagli usi alle dichiarazioni viene effettuata in fase di compilazione
- C++ usa la regola "**nidificato più da vicino**".
    - un'occorrenza di x corrisponde alla dichiarazione nell'ambito più strettamente racchiuso in modo tale che la dichiarazione preceda l'uso
    - una variabile profondamente annidata x nasconde x dichiarata in un ambito esterno
- in Java: gli ambiti interni non possono definire variabili definite negli ambiti esterni

Livelli di scope

Ogni funzione in linguaggi come Java e C ha uno o più ambiti: **uno** per i parametri **e** per il corpo della funzione ed eventualmente ambiti aggiuntivi nella funzione (**per ogni ciclo for** e **per ogni blocco annidato** delimitato da parentesi graffe)

example:

```
void f(int k) {
      int y = 0;
      int x = 3;
      while (y) {
          int x = 1;   // another local var x
      }                // (legal in C++, not legal in Java)
}
```

l'ambito più esterno include solo il nome "f"

La stessa funzione f ha due ambiti (nidificati) in Java (e in C++):

1. il primo include k, y e x
2. l'ambito più interno è per il corpo del ciclo while e include la variabile x inizializzata a 1

Esercizio

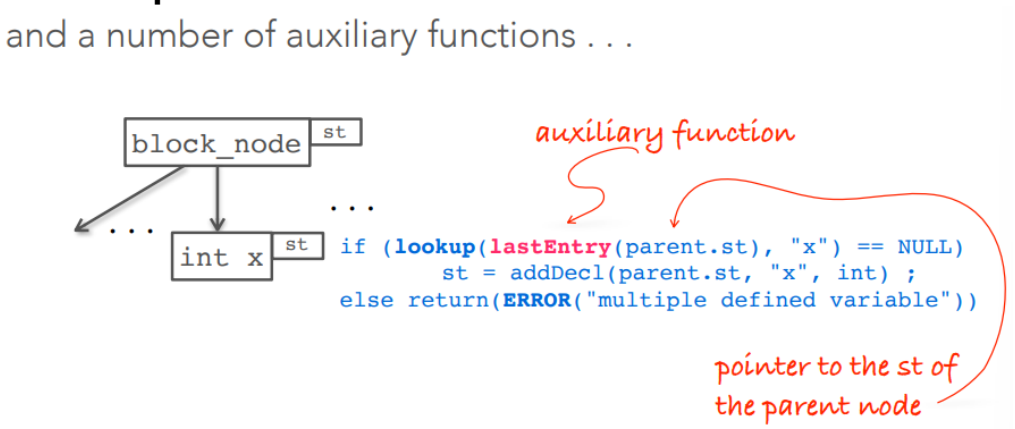
Questo è un programma C++: abbina ogni var-occorrenza alla sua dichiarazione, o dice quando un'occorrenza non è dichiarata

```
class Foo {
    int k=10, x=20;
    void foo(int k) {
       int a = x; int x = k; int b = x;
       while (...) {
                    int x=11;
                    if (x == k) {
                           int k, y;
                           k = (y = x);
                    }
                    if (x == k) { int x, y; }
       }
    }
}
```

Per gestire le tabelle dei simboli sono necessarie quattro funzioni:

1. SymTable **newScope**(SymTable st) // estende la st con un nuovo scope
2. SymTable **addDecl**(SymTable st, String id, Type t) // se non c'è scontro di nomi, aggiunge id ⟼ t a st
3. Type **lookup**(SymTable st, String id) // cerca il tipo di id in st, se presente
4. SymTable **exitScope**(SymTable st) // esce dall'ambito corrente

È fondamentale prendersi cura delle eccezioni!



Si può usare un nome prima che sia definito? In Java è possibile utilizzare un metodo o un nome di campo prima che appaia la definizione, ma non è vero per una variabile!

example:

```
class Test {
  void f() {
        val = 0;   // field val has not yet been declared -- OK
        g();       // method g has not yet been declared -- OK
        x = 1;     // var x has not yet been declared -- ERROR
        int x;
    }
    void g() {}
    int val;
}
```

Scoping: esempio

Java: puoi usare lo stesso nome per una classe, un campo della classe, un metodo della classe, e una variabile locale del metodo

esempio: programma Java legale

```
class Test {
        int Test;
        Test( ) { double Test; }
}
```

Scoping: sovraccarico

Java e C++ (ma non in Pascal o C): può usare lo stesso nome per più di un metodo, purché il numero e/o i tipi di parametri siano univoci

example:

```
int add(int a, int b);
float add(float a, float b);
```

SNIPPETS OF THE SIMPLAN COMPILER

the `ProgLetInNode.java` has the following method for defining symbol tables:

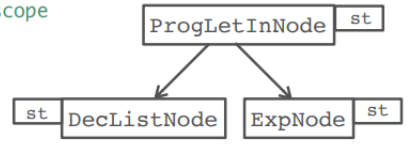
```
public ArrayList<SemanticError> checkSemantics(Environment env) {
    env.nestingLevel++;
     HashMap<String,STentry> hm = new HashMap<String,STentry> ();
     env.symTable.add(hm);
     // declare resulting error list
     ArrayList<SemanticError> res = new ArrayList<SemanticError>();
     // check semantics in the dec list
     if(declist.size() > 0){
         env.offset = -2;
         // check semantics for every child and save the results
         for(Node n : declist)
             res.addAll(n.checkSemantics(env));
     }
     //check semantics in the exp body
     res.addAll(exp.checkSemantics(env));
     //clean the scope, we are leaving a let scope
     env.symTable.remove(env.nestingLevel);
     env.nestingLevel = env.nestingLevel - 1 ;
     return res;
   }
```

the nesting level is increased!

hashtable

list of hashtables

this is used for code generation



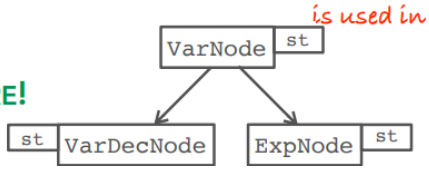
the `VarNode.java` has the following method for defining symbol tables:

extract the symbol table in front of the list

```
public ArrayList<SemanticError> checkSemantics(Environment env) {
       //create result list
       ArrayList<SemanticError> res = new ArrayList<SemanticError>();

       HashMap<String,STentry> hm = env.symTable.get(env.nestingLevel);
       STentry entry = new STentry(env.nestingLevel,type, env.offset);
       env.offset = env.offset - 1 ;

       if ( hm.put(id,entry) != null )
               res.add(new SemanticError("Var id "+id+" already declared"));

       res.addAll(exp.checkSemantics(env));

       return res;
}
```

buid an entry with right nesting-level and type (offset is used in the code generation)

C'È UN GRAVE ERRORE!



the `FunNode.java` has the following method for defining symbol tables:

```
public ArrayList<SemanticError> checkSemantics(Environment env) {
      ArrayList<SemanticError> res = new ArrayList<SemanticError>();
      HashMap<String,STentry> hm = env.symTable.get(env.nestingLevel);
      STentry entry = new STentry(env.nestingLevel,env.offset);
      env.offset = env.offset - 1;
      if ( hm.put(id,entry) != null )
        res.add(new SemanticError("Fun id "+id+" already declared"));
      else {
         env.nestingLevel = env.nestingLevel + 1;
         HashMap<String,STentry> hmn = new HashMap<String,STentry> ();
         env.symTable.add(hmn);
         ArrayList<Node> parTypes = new ArrayList<Node>();
         int paroffset=1;
         for (Node a : parlist){
           ParNode arg = (ParNode) a;
           parTypes.add(arg.getType());
           if (hmn.put(arg.getId(),new STentry(env.nestingLevel,arg.getType(),paroffset)) != null){
                paroffset = paroffset + 1 ;
                res.add(new SemanticError("Parameter id "+arg.getId()+" already declared")) ;
            }
      }
      entry.addType( new ArrowTypeNode(parTypes, type) );
      if (declist.size() > 0){
                env.offset = -2;
                for(Node n : declist)
                     res.addAll(n.checkSemantics(env));
      }
      res.addAll(body.checkSemantics(env));
      env.symTable.remove(env.nestingLevel--); env.nestingLevel = env.nestingLevel - 1 ;
     }
      return res;
}
```

extract the symbol table in front of the list

buid an entry with right nesting-level and type (offset is used in the code generation)

look for comments in the code!

Analisi Semantica: Type Checking

Tipi e sistemi di tipo

Un tipo è

- un insieme di valori
- un insieme di operazioni su quei valori

esempio: le classi sono un'istanza della moderna nozione di tipo

Perché usiamo i tipi? La maggior parte delle operazioni sono **legali** solo per valori di alcuni tipi: **non ha senso** aggiungere un puntatore a funzione e un numero intero in C. **Ha senso** aggiungere due numeri interi, ma entrambi hanno la stessa implementazione del linguaggio assembly!

esempio: qual è il tipo di **addi $r1, $r2, $r3**

Controllo del tipo

Il **controllo del tipo** è il processo di verifica che le operazioni vengano utilizzate con i tipi corretti

Gli **errori di tipo** si verificano quando le operazioni vengono eseguite su valori che non supportano tale operazione

Il controllo del tipo può rilevare alcuni tipi importanti di errori

- errori di memoria: lettura da un puntatore non valido, ecc.
- violazione dei confini di astrazione

```
class FileSystem {
     private File open(String x){
          . . .
      }
      . . .
}
```

```
class Client {
   void f(FileSystem fs){
       File fdesc = fs.open("foo")
       . . .
   } // f cannot see inside FileSystem !
}
```

Il controllo del tipo infastidisce i programmatori: scrivi un pezzo di codice che è significativo per te e ottieni uno stupido errore di tipo ...

Ci sono tre tipi di linguaggi:

- **untyped:** non esiste un tipo in giro: gli errori sono gestiti in fase di esecuzione (Python, JavaScript, ecc.)
- **statically typed:** tutto o quasi il controllo dei tipi viene eseguito come parte della compilazione (la maggior parte dei linguaggi: C, Java, C++, ML, Haskell, ecc.). lo static typing evita il sovraccarico dei controlli del tipo di runtime
- **dinamically typed:** quasi tutti i controlli dei tipi vengono eseguiti come parte dell'esecuzione del programma (Schema, ecc.). Il dynamic typing è più potente rispetto a quello statico, ma più costoso

Il controllo del tipo: il formalismo

Non esiste uno strumento standard per i controllori di tipo: devono essere scritti in un linguaggio di programmazione generico

Ci sono notazioni standard che possono essere utilizzate per specificare le regole del controllo del tipo: possono essere facilmente convertite in codice in qualsiasi lingua host

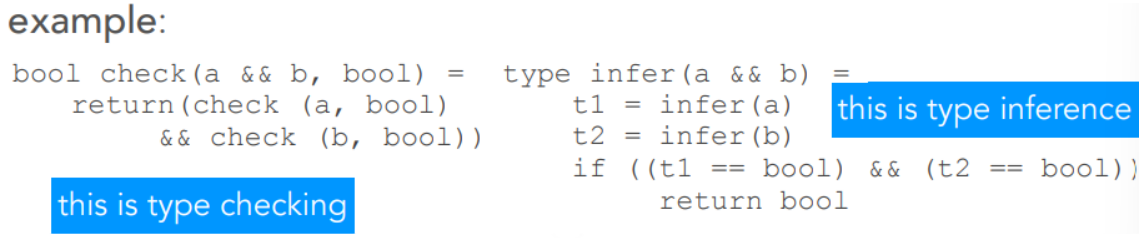
La notazione più comune è **regole di inferenza:** una regola di inferenza ha un insieme di premesse $J_1, ..., J_n$ e una conclusione J, convenzionalmente separate da una retta: $\frac{J_1 ... J_n}{J}$ (quando l'insieme delle premesse è vuoto, la regola si chiama **assioma**)

La **regola di inferenza** $\frac{J_1 ... J_n}{J}$ viene letta se le **Premesse** $J_1, ..., J_n$ sono vere allora la **Conclusione** J è vera; i simboli $J_1, ..., J_n$, J sono chiamati **giudizi**. Il giudizio più comune è $\vdash$ e:T che si legge "l'espressione e ha tipo T". Ad esempio, $\frac{\vdash e_1 : bool \ \vdash e_2 : bool}{\vdash e_1 \&\& e_2 : bool}$ si legge: se e1 ed e2 hanno tipo bool, allora e1 && e2 ha tipo bool

Sistema di tipo, verifica del tipo, inferenza del tipo

L'insieme delle regole di inferenza per i tipi di un linguaggio è chiamato **type system:** regole di inferenza per costrutti linguistici possono essere implementate per mezzo di **funzioni ricorsive** sugli alberi sintattici astratti. Ci sono due tipi di funzioni:

- **verifica del tipo:** dati un termine e e un tipo T, verificare se $\vdash$ e:T
- **inferenza di tipo:** dato un termine e, trova un tipo T, tale che $\vdash$ e:T



Contesto e ambiente: come controlliamo le variabili?

- variabili, come x, possono avere qualsiasi tipo disponibile in un linguaggio di programmazione
- il tipo che ha in un particolare programma dipende dal **contesto**

Il contesto è definito da dichiarazioni che legano le variabili al loro tipo, è una **struttura di dati** in cui è possibile **cercare** una variabile e **ottenerne il tipo: formalmente, i contesti sono mappe da variabili a tipi**

- nelle regole di inferenza, il contesto è indicato dalla lettera greca Γ, ed è chiamato **ambiente** (environment): Γ ⊢ e:bool si legge: e ha tipo bool nell'ambiente Γ

Nei compilatori, l'ambiente è implementato da tabelle di simboli

Verifica del tipo: definizione formale di Γ

L'ambiente Γ è una **mappa parziale finita** Id → Types che accetta identificatori (variabili e simboli di funzione) e restituisce tipi. Γ **può essere esteso**: ad esempio vogliamo estendere Γ con il binding [z ⟼ char]

Γ[z ⟼ char] indica l'ambiente Γ' tale che $\Gamma'(u) = \{char \quad if\ u = z\ \Gamma(u)\ altrimenti$

Γ può essere formalizzato **esplicitando l'elenco dei vincoli**

- **esempio:** sia Γ = [x ⟼ int, y ⟼ bool] allora Γ(x) = int e Γ(y) = bool
- ∅ è la **mappa vuota** (che può essere annotata anche con [])
- la notazione Γ[x ⟼ char] significa anche **aggiornare**: Γ[x ⟼ char] = [x ⟼ char, y ⟼ bool]

A TYPE SYSTEM FOR SIMPLE EXPRESSIONS

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

the type system contains the rules (for type checking)

$$\frac{}{\Gamma \vdash \text{num}: \text{int}}\,[\text{Num}] \qquad \frac{}{\Gamma \vdash \text{true}: \text{bool}}\,[\text{True}] \qquad \frac{}{\Gamma \vdash \text{false}: \text{bool}}\,[\text{False}]$$

$$\frac{\Gamma \vdash e1: \text{int} \quad \Gamma \vdash e2: \text{int} \quad + : \text{int}\times\text{int} \to \text{int}}{\Gamma \vdash e1 + e2 : \text{int}}\,[\text{Plus}] \qquad \frac{\Gamma(\text{id}) = T}{\Gamma \vdash \text{id}: T}\,[\text{Var}]$$

this is not a judgment: it is a condition!

== is polymorphic!

$$\frac{\Gamma \vdash e1: \text{bool} \quad \Gamma \vdash e2: \text{bool} \quad == : \text{bool}\times\text{bool} \to \text{bool}}{\Gamma \vdash e1 == e2 : \text{bool}}\,[\text{Eq-b}]$$

$$\frac{\Gamma \vdash e1: \text{int} \quad \Gamma \vdash e2: \text{int} \quad == : \text{int}\times\text{int} \to \text{bool}}{\Gamma \vdash e1 == e2 : \text{bool}}\,[\text{Eq-i}]$$

type checking

$$\frac{\Gamma \vdash e1: T1 \quad \Gamma \vdash e2: T2 \quad T1{=}T2 \quad == : T1\times T1 \to \text{bool}}{\Gamma \vdash e1 == e2 : \text{bool}}\,[\text{Eq}]$$

type inference

PROOF TREES

with the type system we can derive

$$\Gamma \vdash (x+5)==(y+2) : \text{bool}$$

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

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

Questo è chiamato **PROOF TREE:** gli alberi di prova sono alberi **finiti** dove i nodi sono **istanze** delle regole di inferenza, in particolare, le foglie sono esempi di assiomi; la radice dell'albero contiene il giudizio che è dimostrato

IMPLEMENTATION OF THE TYPE SYSTEM

the type checking is implemented by a recursive function on the nodes of the AST

```
bool check (Env Γ, Exp e, Type t){
   case e of
      e1+e2 : if (t == int) return(check(Γ,e1,int) && check(Γ,e2,int));
              else return false ;
      e1==e2: if (t == bool)
                  if (check(Γ,e1,int) && check(Γ,e2,int)) return true ;
                  else return (check(Γ,e1,bool) && check(Γ,e2,bool)) ;
              else return false ;
      id    : if (Γ(id.name) is undefined)
                      error("Undeclared id") ; return(false) ;
              else return(t == Γ(id.name)) ;
      num   : return(t == int) ;
      true  : return(t == bool) ;
      false : return(t == bool) ;
```

È tail recursive!

remark: the execution of the recursive function mimicks the construction of the proof tree

A TYPE INFERENCE SYSTEM FOR SIMPLE EXPRESSIONS

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

the type system contains the rules (for type inference)

$$\frac{}{\Gamma \vdash \text{num}: \text{int}}\,[\text{Num}] \qquad \frac{}{\Gamma \vdash \text{true}: \text{bool}}\,[\text{True}] \qquad \frac{}{\Gamma \vdash \text{false}: \text{bool}}\,[\text{False}]$$

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

$$\frac{\Gamma \vdash e1: T1 \quad \Gamma \vdash e2: T2 \quad T1 = \text{int} = T2 \quad + : \text{int}\times\text{int} \to \text{int}}{\Gamma \vdash e1 + e2 : \text{int}}\,[\text{Plus}]$$

$$\frac{\Gamma \vdash e1: T1 \quad \Gamma \vdash e2: T2 \quad T1 = T2 \quad == : T1\times T1 \to \text{bool}}{\Gamma \vdash e1 == e2 : \text{bool}}\,[\text{Eq}]$$

```
infer: Env × Exp → Type
lookup : Env × Ide → Type

infer (Gamma,x) =
    return lookup(Gamma,x)
```

read the rule in this way

the polymorphism of == is evident!

PROOFS TREES IN A TYPE INFERENCE SYSTEM

with the type inference system we can derive

$$\Gamma \vdash (x+5)==(y+2) : bool$$

assuming that $\Gamma = [x \mapsto int,\ y \mapsto int]$

$$\dfrac{\dfrac{\dfrac{\Gamma(x) = int}{\Gamma \vdash x:T2\ \ T2=int}[Var] \quad \dfrac{}{\Gamma \vdash 5:T3\ \ T3=int}[Num]}{\dfrac{T2=int=T3 \quad + : int\times int \to int}{\Gamma \vdash (x+5):T1\ \ T1=int}[Plus]} \quad \dfrac{\dfrac{\dfrac{\Gamma(y) = int}{\Gamma \vdash y:T5\ \ T5=int}[Var] \quad \dfrac{}{\Gamma \vdash 2:T6\ \ T6=int}[Num]}{\dfrac{T5=int=T6 \quad + : int\times int \to int}{\Gamma \vdash (y+2):T4\ \ T4=int}[Plus]}}{\dfrac{T1=T4 \quad == : T1\times T1 \to bool}{\Gamma \vdash (x+5)==(y+2): T\ \ T = bool}[Eq]}$$

the text in red are **UNIFICATIONS**

unification amounts to find a substitution σ such that two different terms `t` and `t'`, containing variables, may be identified, e.g. $t\sigma = t'\sigma$

example: $x \to bool \sim int \to y$ with $\sigma = [x \mapsto int,\ y \mapsto bool]$

THE IMPLEMENTATION OF TYPE INFERENCE

similarly to type checking, type inference is implemented by a recursive function on the nodes of the AST

```
Type infer (Env Γ, Exp e){
   case e of
       e1+e2 : t1 = infer(Γ,e1); t2 = infer(Γ,e2);
               if ((t1==int)&&(t2==int)) return int ;
               else error("Wrong invocation of addition") ;
       e1==e2: t1 = infer(Γ,e1); t2 = infer(Γ,e2);
               if (t1==t2) return bool ;
               else error("Wrong invocation of conjunction") ;
       id    : if (Γ(id.name) is undefined) error("Undeclared id") ;
               else return(Γ(id.name)) ;
       num   : return(int) ;
       true  : return(bool) ;
       false : return(bool) ;
```

NON È tail recursive!

remark: the equalities in the guards of conditionals are unifications, in general!

SNIPPETS OF TYPE CHECKING EXPRESSIONS IN SIMPLAN

this is in `PlusNode.java`

isSubtype just verifies type equality!
it is there to support oo-extensions

```
public Node typeCheck() {
    if (! ( SimpLanlib.isSubtype(left.typeCheck(),new IntTypeNode()) &&
          SimpLanlib.isSubtype(right.typeCheck(),new IntTypeNode()) )) {
              System.out.println("Non integers in sum");
              System.exit(0);
    }
    else return new IntTypeNode();
}
```

returns INT

this is in `EqNode.java`

```
public Node typeCheck() {
    Node l = left.typeCheck();
    Node r = right.typeCheck();
    if (! ( SimpLanlib.isSubtype(l,r) ){
          System.out.println("Incompatible types in equal");
          System.exit(0);
    }
    else return new BoolTypeNode();
}
```

returns BOOL

Implementazione dell'ambiente

Gli ambienti sono implementati da tabelle di simboli

- **nel seguito: il tipo di tabelle dei simboli è Env**
- supponiamo che V sia di tipo Env
- l'operazione di lookup Γ(id) è quella che abbiamo definito per le tabelle dei simboli V.lookup(id.name)
- l'operazione di inserimento/estensione Γ[id ⟼ type] di un nuovo identificatore è l'operazione V.put(id,type)

TYPE INFERENCE OF `SimpLan`

the **miniSimpLan** language

```
prog   : 'let' (vardec | fundec)+ 'in' exp ';'            ;
fundec : type ID '(' ( args )? ')' fbody ';'              ;
vardec : type ID '=' exp ';'                              ;
fbody  : exp | 'let' (vardec)+ 'in' exp                   ;
args   : type ID ( ',' type ID)*                          ;
type   : 'int' | 'bool'                                   ;
exp    :  INTEGER | 'true' | 'false' | ID
       | exp '+' exp | exp '==' exp
       | 'if' '(' exp ')' '{' exp '}' 'else' '{' exp '}'
       | ID '(' (exps)? ')'                               ;
exps   : exp (',' exps )*                                 ;
```

TYPE INFERENCE OF EXPRESSIONS

we use a method **infer**(Env V, ExpNode e) that returns Type

the symbol table

the expression to type check — a pointer to the syntax tree —

```
Type infer(Env V, ExpNode e){
  case e of
      num            : return(int)
      true || false  : return(bool)
      id             : Type t = lookup(V, id);
                       if (t = unbound) error("Undeclared id");
                       else return(t);
      e1 + e2        : Type t1 = infer(V, e1);
                       Type t2 = infer(V, e2);
                       if ((t1==int)&&(t2==int)) return(int);
                       else error("Wrong invocation of addition");
      e1 == e2       : Type t1 = infer(V, e1);
                       Type t2 = infer(V, e2);
                       if (t1 == t2) return(bool);
                       else error("Wrong invocation of conjunction");
      . . .  }
```

the node of the syntax tree is a PlusExpNode?

the node of the syntax tree is a EqExpNode?

```
Type infer(Env V, ExpNode e){
   case e of
      . . .
      if (e1){ e2 } else { e3 } :
                  Type t1 = infer(V, e1);
                  Type t2 = infer(V, e2);
                  Type t3 = infer(V, e3);
                  if (t1==bool) && (t2==t3)) return(t2);
                  else error("Type mismatch in conditionals");
      id(e_list) : Type t = lookup(V, id)
                    case t of
                       unbound : error("Undeclared function id");
                       (t1,. . ., tn) -> t0 :
                         [t1',. . ., tm'] = inferTuple(V, e_list)
                         if ((n==m) && (t1 == t1') && ... && (tn == tn'))
                              return(t0);
                         else error("Wrong invocation of function");
```

the node of the syntax tree is a IfExpNode?

esercizio: vedere `CallNode`

```
TupleType inferTuple(Env V, ExpNodeList L){
 case L of
     null     : return([ ]);
     e        : return([infer(V, e)]);
     e :: L1  : return( infer(V, e) :: inferTuple(V, L1) );
```

element concatenation

Inferenza di tipo: regole avanzate

quali sono le regole per i condizionali e le chiamate di funzione?

$$\frac{\Gamma \vdash e1: T1 \quad \Gamma \vdash e2: T2 \quad \Gamma \vdash e3: T3 \qquad T1 = bool \quad T2 = T3}{\Gamma \vdash \text{if } (e1)\ e2 \text{ else } e3 : T2}\ [If]$$

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

e per quanto riguarda le dichiarazioni?

- in miniSimpLan ci sono due tipi di let-construct
  - l'intero programma **'let'** (vardec | fundec)+ **'in'** exp **';'**
  - la funzione corpi **'let'** (vardec)+ **'in'** exp
- cambiano la tabella dei simboli

Inferenza di tipo delle dichiarazioni

the judgments of `decs` are $\Gamma \vdash decs : \Gamma'$

the judgments return environments!

preliminaries: we use **stacks of environments** $\Gamma \bullet \Gamma'$ (Γ' is the top-envitonment)

$$\Gamma \bullet \Gamma'(x) = \begin{cases} \Gamma'(x) & \text{if } x = dom(\Gamma') \\ \Gamma(x) & \text{otherwise} \end{cases}$$

abuse of notation! Γ indicates a single environment and a sequence of environments!

- the operation $(\Gamma \bullet \Gamma')[x \mapsto T]$ becomes $\Gamma \bullet (\Gamma'[x \mapsto T])$, i.e. insert in the top environment
- if Γ is a sequence of environments, `top(Γ)` returns the top environment

rules for declarations are bisognerebbe chiamarla `Var_D` per evitare omonimie

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

La regola per le dichiarazioni di funzione è

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

[Fun] non ammette definizioni ricorsive: se le vuoi, devi sostituire il giudizio nella premessa con

$$\Gamma[f \mapsto (T_1, \ldots, T_n) \to T,\ x_1 \mapsto T_1, \ldots, x_n \mapsto T_n] \vdash e : T'$$

chiama [FunR] la nuova regola

Inferenza di tipo di let

la regola per let

this corresponds to a newScope() operation!

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

this corresponds to a exitScope() operation!

Lo scopo delle dichiarazioni D sono e: fuori let, le dichiarazioni non sono più accessibili!

Usiamo la regola più semplice

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

Inferenza di tipo: implementazione

```
Env inferDecs(Env V, DecsNode D){
        case D of
          T x = e ;     : Type T1 = infer(V,e) ;
                          if (T == T1) return insert(V,x,T);
                          else error("Type mismatch in id decl") ;

          T f (A) e ;   : Tuple<Types> T1 = getType(A);
                          Env V1 = insert(V,f, T1->T);
                          Env V2 = insertArgs(newScope(V1),A); // to be defined!
                          Type T2 = infer(V2,e);
                          if (T == T2) return exitScope(V1);
                          else error("Wrong function id declaration");

  T f(A) let D1 in e ;  : Tuple<Types> T1 = getType(A);
                          Env V1 = insert(V,f, T1->T);
                          V1 = newScope(V1) ;
                          V1 = insertArgs(V1,A);
                          V1 = inferDecs(V1,D1);
                          Type T2 = infer(V1,e);
                          if (T == T2) return exitScope(V1);
                          else error("Wrong function id declaration");

              d D1      : Env V1 = inferDecs(V,d);
                          return inferDecs(V1,D1);
}
```

this may return error("Multiple declaration of id")

type checks recursive functions (we are using [FunR])

formal parameters and local variables are in the same nesting level (we are using a variant of [FunR])

notice: mutual recursion is still not covered!

Programmi di verifica del tipo

i programmi sono prog : 'let' (vardec | fundec)+ 'in' exp ';' ; quindi la regola di inferenza è la stessa di let e l'implementazione lo è

```
Type infer(Env V, ProgNode p){
        Env V1= inferDecs(EMPTY_TABLE, p.decs);
        return infer(V1, p.exp) ;
}
```

// the symbol table is not used!

Programmi di verifica del tipo (ricorsione reciproca)

abbiamo bisogno di una visita preliminare che raccolga le definizioni delle funzioni

we need a pre-visit that collects function definitions

formally, there is a new judgment $\Gamma \Vdash D: \Gamma'$

$$\frac{}{\Gamma \Vdash T\ x = e\ ;\ : \Gamma}\ [VarM] \qquad \frac{\Gamma \Vdash d : \Gamma' \quad \Gamma' \Vdash D : \Gamma''}{\Gamma \Vdash d\ D : \Gamma''}\ [DecM]$$

$$\frac{f \notin dom(top(\Gamma))}{\Gamma \Vdash T\ f(T_1\ x_1,\ldots,T_n\ x_n) = e;\ : \Gamma[f \mapsto (T_1,\ldots,T_n) \to T]}\ [FunM]$$

the let-rule becomes

here we use [Fun] and not [FunR]. WHY? WHAT HAPPENS IF WE USE [FunR]?

$$\frac{\varnothing \Vdash D: \Gamma' \qquad \Gamma \bullet \Gamma' \bullet [] \vdash D: \Gamma'' \qquad \Gamma'' \vdash e: T}{\Gamma \vdash \text{let } D \text{ in } e: T}\ [LetM]$$

question: [LetM] has been written with the premise $\Gamma \bullet \Gamma' \bullet [] \vdash D: \Gamma''$ for reusing [Fun] (or [FunR]): may you provide a different rule in order to have the simpler premise $\Gamma \bullet \Gamma' \vdash D: \Gamma''$?

```
Env inferFuns(Env V, DecsNode D){
      case D of
              Empty   : return(V);
         T x = e ; D' : return inferFuns(F, D') ;
       T f(A) B ; D' : Tuple<Types> T1 = getType(A);
                       Env F = insert(V,f,T1->T);
                       return inferFuns(F, D') ;
}
```

the insert may also fail: multiple declarations of functions in the same scope

the type inference of the whole program managing mutual recursion is

```
Type infer(Env V, ProgNode p){
        Env F = inferFuns(EMPTY_TABLE, p.decs) ;
        F = newScope(F) ;
        Env V1 = inferDecs(F, p.decs);
        return infer(V1, p.exp) ;
}
```

Questioni avanzate

- sottotipizzazione
- dichiarazioni
- sottotipizzazione e assegnazione
- prevalere

Sottotipo

consideriamo il seguente programma in miniSimpLan: let T x = e in e'

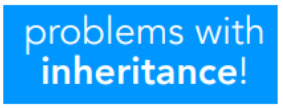
secondo l'attuale sistema di tipi abbiamo l'albero delle prove

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

l'uguaglianza tra T'' e T è un vincolo a volte troppo forte

esempio: con il sistema di digitazione sopra non è possibile digitare

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



definire una relazione T <: T' sui tipi per dire che un oggetto di tipo T può essere utilizzato quando uno di tipo T' è **accettabile**

Sia Inherits_from un insieme di coppie di tipi. Una relazione <: on types è chiamata **sottotipizzazione** quando

- T <: T
- T <: T' se (T, T') ϵ Inherits_from
- T <: T' se T <: T'' e T'' <: T'

VarD with subtyping:

$$\dfrac{\Gamma\vdash e : T \quad x\notin dom(top(\Gamma)) \quad T <: T'}{\Gamma\vdash T'\ x = e\ ; : \Gamma[x\mapsto T']}\ [Var\text{-}Subt]$$

the old rule was:

$$\dfrac{\Gamma\vdash e : T' \quad x\notin dom(top(\Gamma)) \quad T=T'}{\Gamma\vdash T\ x = e\ ; : \Gamma[x\mapsto T]}\ [VarD]$$

Attenzione: regola dec/let errata

quando le dichiarazioni sono singleton potresti avere una regola compact dec let:

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

no need to have • WHY?

1. consider the following wrong rule:

$$\frac{\Gamma \vdash e : T \quad \Gamma \vdash e' : T'' \quad T <: T'}{\Gamma \vdash \texttt{let } T' \texttt{ x = e in e' : } T''}\ [\text{WrongLet1}]$$

- the following good program does not typecheck

```
let int x = 0 in x + 1
```

- and some bad programs do typecheck

```
int foo(B x){ let A x = new A in x.b() }
```

[the problem was that `e'` was typed in a wrong env]

2. next, consider another hypothetical dec/let rule:

$$\frac{\Gamma \vdash e : T \quad \Gamma[x \mapsto T'] \vdash e' : T'' \quad T' <: T}{\Gamma \vdash \texttt{let } T' \texttt{ x = e in e' : } T''}\ [\text{WrongLet2}]$$

- the following bad program is well typed

```
let B x = new A in x.b()
```

[the problem is that we have inverted the subtyping relation]

3. then consider this dec/let rule:

$$\frac{\Gamma \vdash e : T \quad \Gamma[x \mapsto T] \vdash e' : T'' \quad T <: T'}{\Gamma \vdash \texttt{let } T' \texttt{ x = e in e' : } T''}\ [\text{WrongLet3}]$$

- the following good program is not typed

```
let A x = new B in {... x = new A; x.a(); }
```

[the problem is that `e'` has been typed with a wrong binding for `x`]

Invocazione di funzione con sottotipo

function `f` with type: $T_1 \times \ldots \times T_n \to T$

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

- therefore a function may be invoked with values of a subtype
- because of this rule, a function $g : T_1' \times \ldots \times T_n' \to T'$ may be used instead of `f` provided

$$(T_i <: T_i')^{i \in 1..n} \quad \text{and} \quad T' <: T$$

that is, we have the rule for the subtyping relation

$$\frac{(T_i <: T_i')^{i \in 1..n} \quad T' <: T}{T_1' \times \ldots \times T_n' \to T' <: T_1 \times \ldots T_n \to T}\ [\text{Arrow-Subt}]$$

controvariance ← $(T_i <: T_i')$ — $T' <:$ → covariance

- for oo and methods: see afterwards [Java, C# admit **invariance** of inputs]

Condizionata con sottotipo

la sintassi delle espressioni condizionali: if (e) {e1} else {e2}

il sistema di battitura deduce tipi che possono essere tutti diversi...

$$\frac{\Gamma \vdash e: T \quad \Gamma \vdash e1: T1 \quad \Gamma \vdash e2: T2 \quad T <: \texttt{bool} \quad T1 <: T' \quad T2 <: T'}{\Gamma \vdash \texttt{if (e) \{ e1 \} else \{ e2 \} : } T'}\ [\text{If-Subt}]$$

Dichiarazioni e verifica del tipo

estendere miniSimpLan con le istruzioni ottieni impSimpLan

```
prog   : 'let' decs 'in' ( exp | stats )  ';'                    ;
decs   : ( vardec | fundec )+                                    ;
vardec : type ID '=' exp ';'                                     ;
fundec : type ID '(' ( args )? ')' fbody ';'                     ;
fbody  : exp | stats | 'let' (vardec)+ 'in' ( exp | stats )       ;
args   : type ID ( ',' type ID)*                                 ;
type   : 'int' | 'bool' | 'void'                                 ;
exp    :  INTEGER | 'true' | 'false' | ID
       | exp '+' exp | exp '=' exp
       | 'if' exp 'then' '{' exp '}' 'else' '{' exp '}'
       | ID '(' (exps)? ')'                                      ;
exps   : exp (',' exps )*                                        ;
stats  : stat (';' stat )*                                       ;
stat   : ID ':=' exp
       | 'if' exp 'then' '{' stats '}' 'else' '{' stats '}'      ;
```

sequences of statements

bodies may also be statements

the type void!

the statements

Il sistema di digitazione delle dichiarazioni

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

$$\frac{\Gamma \vdash e : T \qquad \Gamma \vdash s1 : T' \qquad \Gamma \vdash s2 : T'' \qquad T = bool \qquad T' = void = T''}{\Gamma \vdash if\ (e)\ s1\ else\ s2 : void}\ [IfS]$$

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

not so difficult

Il sistema di digitazione con sottotipo per le dichiarazioni

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

assignment with subtyping!

T' <: void means that T1 = void because there is no subtype of void

$$\frac{\Gamma \vdash e : T \qquad \Gamma \vdash s1 : T' \qquad \Gamma \vdash s2 : T'' \qquad T <: bool \qquad T' <: void \qquad T'' <: void}{\Gamma \vdash if\ (e)\ s1\ else\ s2 : void}\ [IfS\text{-}Subt]$$

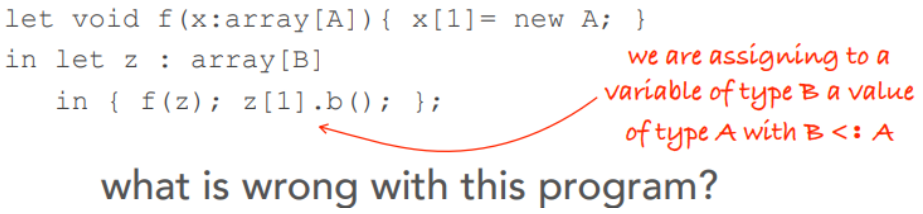
$$\frac{\Gamma \vdash s : T \qquad \Gamma \vdash S : T' \qquad T <: void \qquad T' <: void}{\Gamma \vdash s\ S : void}\ [SeqS\text{-}Subt]$$

* therefore, if `B <: A` , we may write

```
A x = new B() ;
```

Matrici e assegnazioni covarianti

let array[A] il tipo di un array contenente dati di tipo A si potrebbe assumere: if B <: A then array[B] <: array[A] (noto come **array covarianti**, presente ad esempio in Java) ma, considera il seguente programma (ben tipizzato secondo questa ipotesi):



A (generalmente) regola sottotipo errata

Quando l'**array di sottotipi** viene utilizzato al posto di un **array di supertipi** è possibile **inserire un supertipo** nell'array e quindi il supertipo può essere utilizzato al posto di un sottotipo (errore di tipo!). Ma se gli array **non possono essere scritti/modificati**, la "covarianza" è valida! È ammesso quando gli elementi dell'array **NON sono assegnati**

Sottotipo di classe

le sottoclassi di solito possono sovrascrivere alcune dichiarazioni del superclasse (di solito il corpo assegnato al metodo)

supponiamo che sia possibile **sovrascrivere sia i campi che i metodi**, modificandone anche i tipi

problema: class A{..., T f, ...}

class B inheriths A {..., T' f, ...}

con T' <: T (i campi possono essere visti come celle di array)

Solitamente non è ammesso il field override: se possono essere modificati dinamicamente, il tipo non può essere modificato; in un **linguaggio funzionale** (es. SimpLan) può essere supportata la sottotipizzazione dei campi **(campi covarianti)**

Invocazione del metodo

this is a small environment as well!

remark: the environment Γ now binds class types

- e.g. $\Gamma(C) = [a : T_a,\ b : T_b,\ m : T_1 \times \ldots \times T_n \to T]$
- therefore we may access to the type of a field with $\Gamma(C)(a)$ and of a method with $\Gamma(C)(m)$
- you may also use the notations $\Gamma(C.a)$ and $\Gamma(C.m)$

the type rule for method invocation:

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

Overriding del metodo (in scala)

esempio [da Scala**]**:

```
class A{..., T m(T1 p1,..., Tn pn){ e },… }
class B inherits A{..., T' m(T1' p1,…,Tn' pn){ e' }, ... }
```

consider `let T z = x.m(e1,…,en) in e`

assumendo x di tipo A che può essere istanziato da un oggetto di tipo B

- il tipo restituito B T' deve essere utilizzabile al posto del tipo restituito A T abbiamo bisogno di T' <: T
- I tipi di parametri A devono essere utilizzabili al posto dei tipi di parametri B di cui abbiamo bisogno Ti <: Ti'

Overriding del metodo: riassunto

if $T_1 <: T_1' \ldots T_n <: T_n'$ and $T' <: T$ and $B <: A$

the type of `m` in `A` is $T_1 \times \ldots \times T_n \to T$

the type of `m` in `B` is $T_1' \times \ldots \times T_n' \to T'$

then

$$\frac{T_1 <: T_1' \quad \ldots \quad T_n <: T_n' \qquad T' <: T}{T_1' \times \ldots \times T_n' \to T' \ <: \ T_1 \times \ldots \times T_n \to T}$$

che è la regola generale di sottotipizzazione per le funzioni

- tipo di output (ritorno) **covariante**

- tipi di input (parametro) **controvarianti**
- la maggior parte dei linguaggi di programmazione (Java, C#) ammette solo **invarianza** dei tipi di input!

Commenti

le regole di digitazione usano una notazione molto concisa, sono costruiti con molta cura, ma alcuni **buoni programmi** verranno comunque **rifiutati**

if (x == x) then return(x==1) else return(x)

la nozione di un buon programma è indecidibile

un sistema di tipi consente a un compilatore di rilevare molti errori di programmazione comuni: il costo è che alcuni programmi corretti non sono consentiti, si potrebbe avere un controllo del tipo statico più espressivo, ma i sistemi di tipo più espressivi sono anche più complessi

Analizzare effetti

considera il seguente linguaggio più istruzioni, condizionali e cancellazione - chiamalo ImpLan_d - [la sintassi per exp è omessa!]

```
block       : '{' dec* statement* '}'                          ;
statement   : assignment
              | deletion
              | conditional
              | block
              | ID '(' (exp (',' exp*))? ')'                   ;
dec         : type ID ';'
              | ID '(' (param (',' param*))? ')' block          ;
param       : ('var')? type ID                                 ;
assignment  : ID '=' exp ';'                                   ;
deletion    : 'delete' ID ';'                                  ;
conditional: 'if' '(' exp ')' statement 'else' statement       ;
type        : 'int' | 'bool'                                   ;
```

no new operation!

vogliamo verificare il corretto utilizzo di delete

- non è possibile accedere in lettura/scrittura ad una variabile cancellata: per esempio. elimina x ; y = x + 1; è sbagliato
- una funzione che elimina una variabile passata con var ha effetti collaterali sulla variabile: per esempio. f(var int x){ elimina x; } fa) ; a= a+1; è sbagliato

Funzioni problematiche in implan_d

- non è possibile accedere in lettura/scrittura ad una variabile cancellata da una funzione: **è necessario registrare gli effetti delle funzioni**

int a; f(var int x){ delete x ;} f(a); a= a + 1;

- problemi di aliasing: **devi stare attento con le identità dei nomi**

```
int a; int b;
f(var int x, var int y) {delete x; y = y + 1;}
f(a, b); f(b, b);
```

- problemi con i condizionali: **calcolare i tipi è difficile, è necessaria un'analisi del punto fisso**

```
int a; int b;
f(var int x, int n) {
    if (n == 0) {delete x;}
    else {int y; f(y, n - 1); delete y;}
}
f(a, 5);
```

Il dominio degli effetti e le operazioni

we assume programs are correctly typed (in the standard way)

effect types of identifiers $\mathcal{B} = \{ \bot, \mathtt{rw}, \mathtt{d}, \top \}$

$\mathcal{B}$ is a partial order with the ordering relation $\bot \leq \mathtt{rw} \leq \mathtt{d} \leq \top$

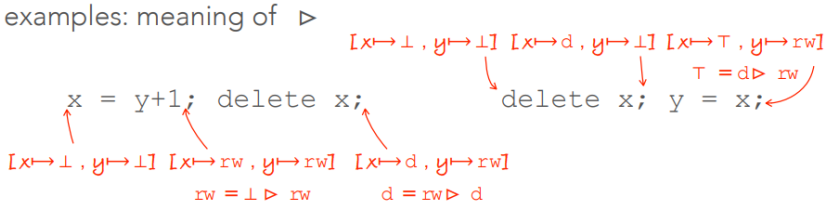
operations on $\mathcal{B}$

- **max**: $\mathtt{max\{a,\ b\}}$ returns the maximum of the two
- **seq**: $a \triangleright b = \begin{cases} \mathtt{max\{a,b\}} & \text{if } \mathtt{max\{a,b\}} \leq \mathtt{rw} \\ \mathtt{d} & \text{if } (a \leq \mathtt{rw} \text{ and } b=\mathtt{d}) \text{ or } (a=\mathtt{d} \text{ and } b=\bot) \\ \top & \text{otherwise} \end{cases}$
- **par**: $a \otimes b = \begin{cases} a & \text{if } b=\bot \\ b & \text{if } a=\bot \\ \mathtt{rw} & \text{if } a=\mathtt{rw} \text{ and } b=\mathtt{rw} \\ \top & \text{otherwise} \end{cases}$

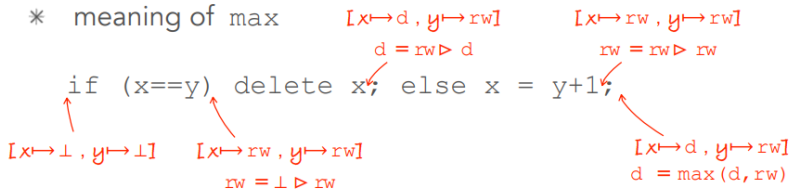
the operations are all monotone wrt $\leq$

Esempi sugli operatori sugli effetti

examples: meaning of $\triangleright$



- meaning of max



the meaning of $\otimes$ will be seen later on

Ambienti con effetti e regolamenti di scrittura

- Σ is the environment binding identifiers to $\mathscr{B}$
- operations on Σ, where we assume $dom(\Sigma') \subseteq dom(\Sigma)$

$$\Sigma \triangleright \Sigma'(x) = \begin{cases} \Sigma(x) \triangleright \Sigma'(x) & \text{if } x \in dom(\Sigma') \\ \Sigma(x) & \text{otherwise} \end{cases}$$

$$max(\Sigma, \Sigma')(x) = \begin{cases} max\{\Sigma(x), \Sigma'(x)\} & \text{if } x \in dom(\Sigma') \\ \Sigma(x) & \text{otherwise} \end{cases}$$

par will be discussed afterwards

the typing rules with effects for expressions and statements

$$\frac{ids(e) = \{x_1, \ldots, x_n\}}{\Sigma \vdash e : \Sigma \triangleright [x_1 \mapsto rw, \ldots, x_n \mapsto rw]}\ [Exp\text{-}e]$$

$$\frac{\Sigma \vdash e : \Sigma'}{\Sigma \vdash x = e; : \Sigma' \triangleright [x \mapsto rw]}\ [Asgn\text{-}e] \qquad \frac{}{\Sigma \vdash \text{delete } x; : \Sigma \triangleright [x \mapsto d]}\ [Del\text{-}e]$$

$$\frac{\Sigma \vdash e : \Sigma' \quad \Sigma' \vdash s1 : \Sigma_1 \quad \Sigma' \vdash s2 : \Sigma_2}{\Sigma \vdash \text{if (e) \{s1\} else}_{53}\{s2\} : max(\Sigma_1, \Sigma_2)}\ [If\text{-}e]$$

Sequenza e blocco

- the last two rules for statements

$$\frac{\Sigma \vdash s : \Sigma' \quad \Sigma' \vdash S : \Sigma''}{\Sigma \vdash s\ S : \Sigma''}\ [Seq\text{-}e]$$

why this condition?

$$\frac{\Sigma \bullet [\,] \vdash D : \Sigma' \quad \Sigma' \vdash S : \Sigma'' \quad \Sigma'' = \Sigma_0'' \bullet \Sigma_1'' \quad (\Sigma_1''(x) \leq d)^{x \in dom(\Sigma_1'')}}{\Sigma \vdash \{D\ S\} : \Sigma_0''}\ [Block\text{-}e]$$

exercises: verify

- `x = 1; delete x; y = x+1;`
- `x = 1; { int y; y = x+1; if (x == y) delete x; else delete y;}`
- `x = 1; { int y; y = x+1; if (x == y) delete x; else delete y;} x = 2;`
- `x = 1; { int y; y = x+1; if (x == y) y=y-1; else delete y;} x = 2;`

we still need rules for declarations and function invocations

Altre regole con effetti — le dure

$$\frac{}{\Sigma \vdash \mathtt{T}\ x;\ :\ \Sigma[x \mapsto \bot]}\ [\text{Var-e}]$$

$$\frac{\Sigma \vdash d : \Sigma' \qquad \Sigma' \vdash D : \Sigma''}{\Sigma \vdash d\ D : \Sigma''}\ [\text{Dseq-e}]$$

- no access to global variables
- no function invocation

$$\frac{\Sigma_0 = [x_1 \mapsto \bot, \ldots, x_m \mapsto \bot, y_1 \mapsto \bot, \ldots, y_n \mapsto \bot] \qquad \Sigma_0 \vdash s : \Sigma_1}{\Sigma \vdash f(\mathtt{var}\ T_1\ x_1, \ldots, \mathtt{var}\ T_m\ x_m, T_1'\ y_1, \ldots, T_n'\ y_n)\ s : \Sigma[f \mapsto \Sigma_0 \to \Sigma_1]}\ [\text{Fseq-e}]$$

we accept that, for some x, $\Sigma_1(x) = \top$!

the var-types of f — $x_1, \ldots, x_m$ — the standard types of f

$$\frac{\Gamma \vdash f : \&T_1 \times \ldots \times \&T_m \times T_1' \times \ldots \times T_n' \to \mathtt{void} \qquad \Sigma(f) = \Sigma_0 \to \Sigma_1 \qquad (\Sigma_1(y_i) \le d)^{1 \le i \le n} \qquad \Sigma' = \Sigma[(z_i \mapsto \Sigma(z_i) \rhd \mathtt{rw})^{z_i \in \mathtt{var}(e_1, \ldots, e_n)}] \qquad \Sigma'' = \otimes_{i \in 1..m}[u_i \mapsto \Sigma(u_i) \rhd \Sigma_1(x_i)]}{\Sigma \vdash f(u_1, \ldots, u_m, e_1, \ldots, e_n)\ ;\ :\ \mathtt{update}(\Sigma', \Sigma'')}\ [\text{Invk-e}]$$

the var used in $e_1, \ldots, e_n$ are accessed in rw

where $\Sigma \otimes \Sigma'$ is defined as follows $(\mathrm{dom}(\Sigma) \neq \mathrm{dom}(\Sigma'))$

$$\Sigma \otimes \Sigma'(x) = \begin{cases} \Sigma(x) & \text{if } x \notin \mathrm{dom}(\Sigma') \\ \Sigma'(x) & \text{if } x \notin \mathrm{dom}(\Sigma) \\ \Sigma(x) \otimes \Sigma'(x) & \text{otherwise} \end{cases}$$

and, assuming $\mathrm{dom}(\Sigma') \subseteq \mathrm{dom}(\Sigma)$, $\mathtt{update}(\Sigma, \Sigma')$ is

$$\mathtt{update}(\Sigma \bullet \Sigma_1,\ \Sigma') = \begin{cases} \mathtt{update}(\Sigma \bullet \Sigma_1[u \mapsto a],\ \Sigma'') & \text{if } \Sigma' = \Sigma''[u \mapsto a] \text{ and } u \in \mathrm{dom}(\Sigma_1) \\ \mathtt{update}(\mathtt{update}(\Sigma, [u \mapsto a]) \bullet \Sigma_1, \Sigma'') & \text{if } \Sigma' = \Sigma''[u \mapsto a] \text{ and } u \notin \mathrm{dom}(\Sigma_1) \\ \Sigma \bullet \Sigma_1 & \text{if } \Sigma' = \emptyset \end{cases}$$

let $\Sigma|_{FUN}$ be the environment that coincides with Σ on functions and it is undefined otherwise

- no access to global variables
- no mutual recursion

$$\frac{\Sigma_0 = [x_1 \mapsto \bot, \ldots, x_m \mapsto \bot, y_1 \mapsto \bot, \ldots, y_n \mapsto \bot] \qquad \Sigma|_{FUN} \bullet \Sigma_0[f \mapsto \Sigma_0 \to \Sigma_1] \vdash s : \Sigma|_{FUN} \bullet \Sigma_1[f \mapsto \Sigma_0 \to \Sigma_1]}{\Sigma \vdash f(\mathtt{var}\ T_1\ x_1, \ldots, \mathtt{var}\ T_m\ x_m, T_1'\ y_1, \ldots, T_n'\ y_n)\ s : \Sigma[f \mapsto \Sigma_0 \to \Sigma_1]}\ [\text{Fseq-e}]$$

the premise $\Sigma|_{FUN} \bullet \Sigma_0[f \mapsto \Sigma_0 \to \Sigma_1] \vdash s : \Sigma|_{FUN} \bullet \Sigma_1[f \mapsto \Sigma_0 \to \Sigma_1]$ requires a fixpoint computation

- we start with $\Sigma_1 = [x_1 \mapsto \bot, \ldots, x_m \mapsto \bot, y_1 \mapsto \bot, \ldots, y_n \mapsto \bot]$
- and we iterate till we find a fixpoint

example:

```
f(var int x, int n){
     if (n==0) { delete x; }
     else { int y ; f(y,n-1); delete x; }
}
```

initialization: $\Sigma_0=[x \mapsto \bot, n \mapsto \bot]$ $\Sigma_1=[x \mapsto \bot, n \mapsto \bot]$

iteration 1: compute

$$\Sigma|_{FUN} \bullet \Sigma_0[f \mapsto \Sigma_0 \to \Sigma_1] \vdash s : \Sigma|_{FUN} \bullet \Sigma_1[f \mapsto \Sigma_0 \to \Sigma_1]$$

then you obtain $\Sigma_0=[x \mapsto \bot, n \mapsto \bot]$ $\Sigma_1'=[x \mapsto d, n \mapsto rw]$

iteration 2: compute

$$\Sigma|_{FUN} \bullet \Sigma_0[f \mapsto \Sigma_0 \to \Sigma_1'] \vdash s : \Sigma|_{FUN} \bullet \Sigma_1'[f \mapsto \Sigma_0 \to \Sigma_1']$$

then you obtain $\Sigma_0=[x \mapsto \bot, n \mapsto \bot]$ $\Sigma_1'=[x \mapsto d, n \mapsto rw]$

therefore Σ_1' is the fixpoint!

Funzioni problematiche in implan_d

exercises:

1

```
int a; int b;
f(var int x, var int y) { delete x; y= y+1;}
f(a,b); f(b,b);
```

2

```
int a; int b;
f(var int x, int n){
     if (n==0) { delete x; }
     else { int y ; f(y,n-1); delete y; }
}
f(a,5);
```

3

```
int a; int b;
f(var int x, int n){
     if (n==0) { delete x; }
     else { int y ; f(y,n-1); delete x; }
}
f(a,5);
```

what happens if we remove this?

Snippet delle condizioni di controllo del tipo in simplan

this is in `IfNode.java`

```
public Node typeCheck() {
      if (!(SimpLanlib.isSubtype(cond.typeCheck(),new BoolTypeNode()))) {
          System.out.println("non boolean condition in if");
          System.exit(0);
      }
      Node t = th.typeCheck();
      Node e = el.typeCheck();
      if (SimpLanlib.isSubtype(t,e))
         return t;
      else { System.out.println("Incompatible types in then else branches");
           System.exit(0);
           return null;
      }
 }
```

SNIPPETS OF SIMPLAN

in `FunNode.java`:

```
public Node typeCheck () {
     if (declist!=null)
               for (Node dec:declist)
                     dec.typeCheck();
     if ( !(SimpLanlib.isSubtype(body.typeCheck(),type)) ){
               System.out.println("Wrong return type for function " + id) ;
               System.exit(0) ;
     } else return null;
}
```

in `VarNode.java`:

```
public Node typeCheck () {
     if (! (SimpLanlib.isSubtype(exp.typeCheck(),type)) ){
              System.out.println("Incompatible value for variable " + id);
              System.exit(0);
     } else return null;
}
```

Generazione del codice

Stato: abbiamo coperto le **fasi di front-end** (analisi lessicale, analisi, analisi semantica), le prossime sono le **fasi di back-end** (ottimizzazione, **generazione del codice**)

Ambienti run-time

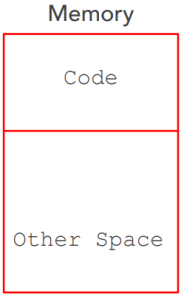
Prima di discutere la generazione del codice, dobbiamo capire cosa stiamo cercando di generare: esistono numerose tecniche standard per la strutturazione del codice eseguibile ampiamente utilizzate

L'esecuzione di un programma è **inizialmente sotto il controllo del sistema operativo** — quando un programma viene invocato:

1. il sistema operativo alloca spazio per il programma
2. il codice viene caricato in una parte dello spazio
3. il sistema operativo salta al punto di ingresso (cioè "main")

Disposizione della memoria

tradizionalmente, le immagini dell'organizzazione della macchina hanno aree per diversi tipi di dati: delimitata da linee



queste immagini sono **semplificazioni:** non tutta la memoria deve essere contigua

L'altro spazio contiene tutti i dati per il programma **(altro spazio = spazio dati)**

il compilatore è responsabile di: generazione di codice e **gestire** l'uso dello spazio dati.

Obiettivi e assunzioni di generazione del codice

due obiettivi:

1. correttezza
2. velocità

la maggior parte delle complicazioni nella generazione del codice deriva dal tentativo di essere veloce oltre che corretto

ipotesi:

1. **l'esecuzione è sequenziale**; il controllo si sposta da un punto all'altro di un programma in un ordine ben definito
2. quando viene chiamata una procedura, **il controllo ritorna eventualmente al punto immediatamente successivo alla chiamata**

Attivazioni e vita delle variabili

un'invocazione della funzione **f** è un'attivazione di **f**

la **durata di un'attivazione** di **f** è tutti i passaggi per eseguire **f,** comprese tutte le fasi delle procedure delle chiamate di **f**

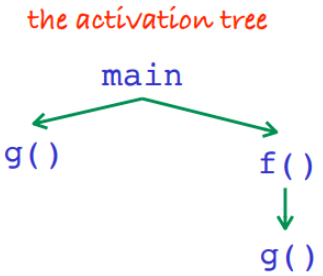
la **durata di una variabile x** è la **porzione di esecuzione** in cui **x** è definito

- La vita è un **concetto dinamico (di runtime)**.
- Lo scope è un **concetto statico**

Alberi di attivazione

example:

```
int g() { return 1 ; }
int f() { return g() ; }
void main() {
          g() ; f() ;
}
```

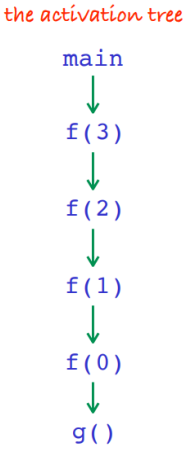


l'ipotesi (2) richiede che quando **f** chiama **g**, allora **g** ritorna prima che **f** continui

- le durate delle attivazioni delle procedure sono nidificate correttamente
- le vite di attivazione possono essere rappresentate come un albero: **l'albero di attivazione**

Esempio: calcolare l'albero di attivazione per

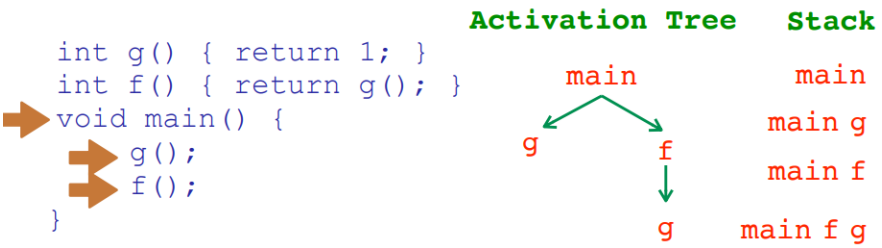
```
int g() { return 1; }
int f(int x) {
    if (x == 0) { return g(); }
    else { return f(x - 1); }
}
void main() { f(3); }
```



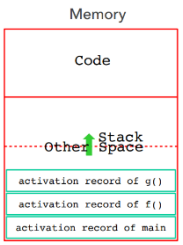
l'albero di attivazione **dipende dal comportamento in fase di esecuzione**

l'albero di attivazione **può essere diverso per ogni input del programma**

poiché le attivazioni sono nidificate, uno **stack** può tenere traccia delle procedure attualmente attive



Disposizione della memoria revisionata



le informazioni necessarie per gestire l'attivazione di una funzione sono chiamate **record di attivazione** (AR) o **frame**

se **f** chiama **g**, il record di attivazione di **g** contiene un mix di informazioni su **f** e **g**

Cosa c'è in g's ar quando f chiama g?

f viene "**sospesa**" fino al completamento di **g**, quando ciò accade, **f** riprende

L'AR di **g** contiene le informazioni necessarie per riprendere l'esecuzione di **f**

L'AR di **g** può anche contenere:

- valore di ritorno di g (necessario per f)
- parametri effettivi a g (fornito da f)
- spazio per le variabili locali di g

I contenuti di una tipica ar per g

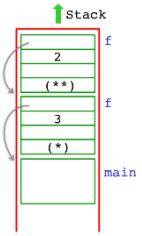
1. spazio per il **valore di ritorno** di **g**
2. **parametri effettivi**
3. **puntatore al record di attivazione precedente:** il **collegamento di controllo** punta ad AR del chiamante di **g**
4. **stato della macchina** prima della chiamata **g:** contenuto dei **registri e contatore dei programmi**
5. **variabili locali**
6. altri **valori temporanei**

THE EXERCISE, REVISITED

```
int g() { return 1; }
int f(int x) {
    if (x == 0) { return g(); }
    else { return f(x - 1); (**) }
}
void main() { f(3); (*) }
```

AR for f:

Stack dopo due chiamate a f

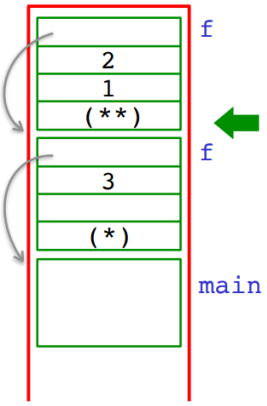


main non ha argomenti o variabili locali e il suo risultato non viene mai utilizzato; **il suo AR non è interessante.** (*) e (**) sono gli **indirizzi di ritorno** delle invocazioni di f, questo è solo uno dei tanti possibili progetti AR (funziona con molti linguaggi di programmazione)

Il punto principale

il compilatore deve **determinare**, in fase di compilazione, il **layout** dei record di attivazione e **generare codice** che acceda correttamente alle posizioni nel record di attivazione. Quindi **il layout AR e il generatore di codice devono essere progettati insieme!**

Esempio



l'immagine mostra lo stato dopo la chiamata alla seconda invocazione di **f ritorna** (la **freccia verde** indica l'AR **attivo**)

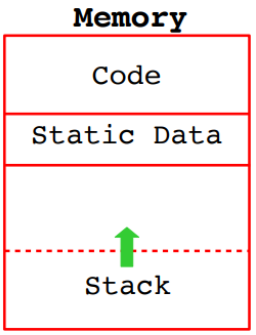
poiché il valore restituito è il primo in un frame, il chiamante può trovarlo a un offset fisso dal proprio frame

questa non è l'unica organizzazione possibile

- è possibile riorganizzare l'ordine degli elementi del telaio
- è possibile suddividere diversamente le responsabilità del chiamante/chiamato

Variabili globali

tutti i riferimenti a una variabile globale puntano allo stesso elemento: **è sbagliato** memorizzare una variabile globale in un AR



alle variabili globali viene assegnato un **indirizzo fisso** una volta: le variabili con indirizzo fisso sono "**allocate staticamente**"

a seconda della lingua, potrebbero esserci altri valori allocati staticamente

layout di memoria con dati statici:

Variabili dichiarate in ambiti esterni

Riferimenti a una **variabile dichiarata in una portata esterna:** dovrebbe indicare una variabile memorizzata in un **altro record di attivazione**

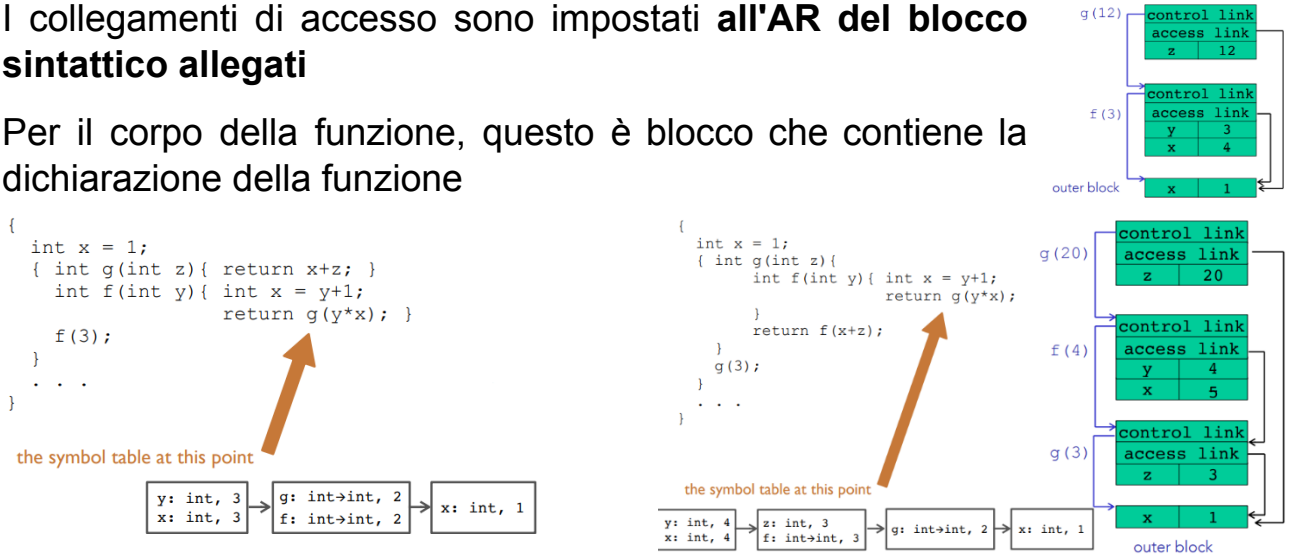
A quale record di attivazione? Secondo la regola **più strettamente nidificata**, un record di attivazione dovrebbe indicare il record di attivazione più recente del suo ambito di racchiusi immediatamente

Utilizzare i **collegamenti di accesso** ...

Scope statico con collegamenti di accesso

I collegamenti di accesso sono impostati **all'AR del blocco sintattico allegati**

Per il corpo della funzione, questo è blocco che contiene la dichiarazione della funzione



Impostazione del link di accesso

il valore del link di accesso di un nuovo record di attivazione è stabilito come segue:

- viene inserito un blocco interno o viene chiamata una funzione **dichiarata nell'ambito corrente:**
  - **ACCESS_LINK = indirizzo di ACCESS_LINK nell'AR corrente**
- una funzione chiama se stessa in modo ricorsivo o chiama un'altra funzione **dichiarata nel blocco sintattico che la racchiude:**
  - **ACCESS_LINK = valore di ACCESS_LINK dell'AR corrente**
- **in generale**, chiama una funzione al di **fuori dell'ambito corrente:**
  - **ACCESS LINK = seguire la catena di ACCESS_LINKs per la differenza tra il livello di annidamento corrente e quello di dichiarazione della funzione; sia AR' il record di attivazione (indirizzo di ACCESS_LINK di AR')**

Conservazione dell'heap

un valore che **sopravvive alla procedura** che lo crea non può essere mantenuto in AR: **Bar foo() { return new Bar() ; }**

il valore di **Bar** deve sopravvivere alla deallocazione dell'AR di foo

le lingue con **dati allocati dinamicamente** utilizzano un **heap** per **archiviare i dati dinamici**

gli AR nello stack vengono deallocati quando il controllo esce dall'ambito corrispondente

che dire dei dati nell'heap?

Raccolta dei rifiuti

Che dire dei dati nel mucchio?

- Possono essere rimossi quando diventano "**spazzatura**"
- A un determinato punto P nell'esecuzione di un programma, una posizione di memoria M è la **spazzatura** se nessuna continuazione di P può accedere alla posizione m

raccolta dei rifiuti:

- Rileva la spazzatura durante l'esecuzione del programma
- è invocato quando è necessaria più memoria
- **La decisione è effettuata dal sistema di runtime**, non dal programma

in alcuni linguaggi di programmazione, la deallocazione è sotto la responsabilità del programmatore

esempio di deallocazione in C

```
ptr = lst ; flag = true ;
while (ptr!=NULL  && flag){
    if (ptr->val == 1) { ptr = ptr->next ; flag = false ; }
    else { previous = ptr; ptr = ptr->next ; free(previous) ; }
    }
```

problema: in caso di **condivisione** (più puntatori nella stessa posizione) la cella non può essere effettivamente liberata (**puntatori penzolanti**)

nell'esempio sopra, 1st è un puntatore penzolante quando lst->val != 1

altre lingue hanno algoritmi di garbage collection **impliciti**

Algoritmo per collezione spazzatura / marchio e spazzata

algoritmo mark-and-sweep

- Assumere i **tag bit** associati ai dati
- Supponiamo che gli indirizzi delle posizioni create da un programma siano raccolti in una **tabella**

L'algoritmo:

1. Impostare tutti i tag bit su 0

2. Inizia da ciascuna posizione utilizzata direttamente nel programma (cercali dall'AR) e segui tutti i collegamenti, modificando il bit tag a 1
3. Considera come rifiuti tutte le celle con tag = 0

Algoritmo di raccolta rifiuti/conteggio di riferimento

algoritmo di conteggio dei riferimenti

supponiamo che ogni dato in memoria abbia un contatore di **riferimento associato**

l'algoritmo:

1. quando un dato è allocato in memoria, inizializzare il contatore a 0
2. quando è impostato un puntatore a un dato, incrementa il contatore
3. quando viene azzerato un puntatore a un dato, decrementare il contatore
4. quando il contatore diventa 0, il dato è spazzatura

Riassunto

l'**area del codice** contiene il **codice oggetto:** per la maggior parte delle lingue, **dimensione fissa e sola lettura**

l'**area statica** contiene **dati** (non codice) con indirizzi fissi (es. dati globali): **dimensione fissa**, può essere **leggibile o scrivibile**

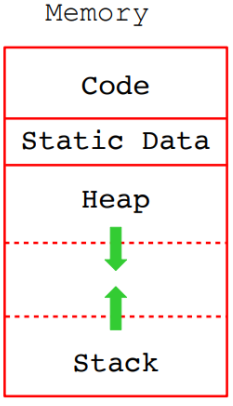
lo **stack** contiene un AR per ogni procedura attualmente attiva: ogni AR **di solito ha una dimensione fissa** (per una determinata procedura), contiene locali

l'heap contiene tutti gli altri dati

sia l'**heap** che lo **stack crescono**

- deve fare attenzione che **non crescano l'uno nell'altro**
- **soluzione:** avviare l'heap e impilare alle **estremità opposte della memoria** e lasciarle crescere l'una verso l'altra

Disposizione memoria con heap

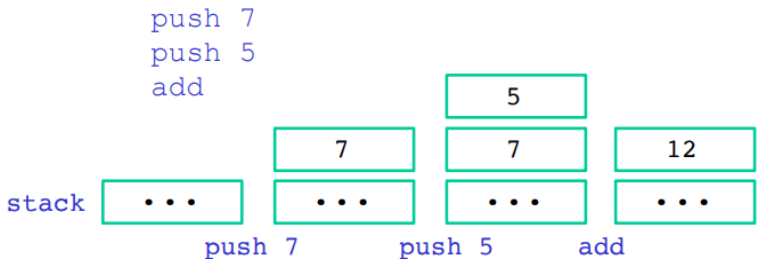


Generazione codice per macchina impilatore

le **macchine impilatrici** sono un semplice modello di valutazione: **non** hanno **variabili o registri**; utilizzano una **pila di valori** per risultati intermedi

esempio: considera due istruzioni

- **push i:** posiziona l'intero **i** in cima allo stack
- **add:** apre due elementi, li aggiunge e rimette il risultato in pila
- un programma che calcola 7 + 5:



Nelle macchine stack

ogni operazione prende gli operandi dallo stack o dai registri e inserisce i risultati nello stack o in un registro: questo significa un **semplice** schema di compilazione

- la **posizione** degli **operandi** è **implicita** (lo stack) o **esplicita** (i registri)
- quando gli operandi sono impliciti, sono sempre in **cima allo stack**
- pertanto, **non** è **necessario** specificare gli operandi in modo esplicito
- **non** è **necessario** specificare la posizione del risultato
- confrontare l'istruzione "**add**" con "**add r1 r2 r3**"

L'istruzione aggiungi con un registro

l'istruzione add esegue tre operazioni di memoria

- due letture e una scrittura nello stack
- si accede frequentemente alla parte superiore della pila

una soluzione migliore è l'ibrida: una macchina stack con registri

- gli accessi al registro sono **più rapidi**
- l'istruzione **add** è ora **$r0 ← $r0 + cima_della_pila**
- una **sola** operazione di memoria

la **Java Virtual Machine** è una macchina stack con registri

la **Simple Virtual Machine** (SVM) è una macchina stack con registri **(questa è la macchina di SimpLan)**

AN EXAMPLE: 3 + (7 + 5)

without a register

code	stack
push 3	3
push 7	7,3
push 5	5,7,3
add	12,3
add	15

with a register $r0

code	stack	$r0
push 3	3	
push 7	7 3	
$r0 ← 5	7 3	5
$r0 ← $r0+top_stack	7 3	12
pop	3	12
$r0 ← $r0+top_stack	3	15
pop		15

we will use a bytecode **with registers**!

Dalle macchine stack al linguaggio di assemblaggio

Consideriamo un compilatore che genera codice per una macchina pila con registri

- Vogliamo eseguire il codice risultante su qualche processore
- Definiremo una macchina, chiamata SVM (**semplice macchina virtuale**) e un **interprete** per il codice della macchina SVM
- L'interprete sarà in Java

La nostra **lingua di assemblaggio**

- ha operazioni aritmetiche che utilizzano registri per gli operandi e i risultati: useremo **$r0, $r1** ...
- Utilizzare il **carico** e **memorizzare** le istruzioni per utilizzare gli operandi e risultati in memoria
- ha **registri generici** (32 bit ciascuno): useremo **$sp**, **$a0** e **$t1** (un registro temporaneo)

A SAMPLE OF ASSEMBLY INSTRUCTIONS

usually $r2 will be a pointer to the top of the stack . . . $sp

```
lw $r1 offset($r2)
    loads 32-bit word from address $r2+offset into $r1
add $r1 $r2 $r3
    $r1 ← $r2 + $r3 // the value of $r2 plus the value
                    // of $r3 is stored in $r1
sw $r1 offset($r2)
    stores 32-bit word in $r1 at address $r2+offset
addi $r1 $r2 n
    $r1 ← $r2 + n   // the value of $r2 plus n is
                    // stored in $r1
li $r1 n
    $r1 ← n         // n is stored in $r1
move $r1 $r2
    $r1 ← $r2       // the value of $r2 is stored in $r1
```

EXAMPLE

the stack-machine code for 7+5 in the assembly language

the growth of the stack means that $sp decreases

$a0 ← 7	li $a0 7
push $a0	addi $sp $sp -4 sw $a0 0($sp)
$a0 ← 5	li $a0 5
$a0 ← $a0 + top_of_stack	lw $t1 0($sp) add $a0 $a0 $t1
pop	addi $sp $sp 4

the pop lets the stack decrease

Alcune utili macro

push $t addi $sp $sp -4 sw $t 0 ($sp)

pop addi $sp $sp 4

$t ← top lw $t 0 ($sp)

Un frammento di Simplan

Una lingua con numeri interi e operazioni intere

```
P → D ; P | E
D → T id(ARGS) = E
ARGS → T id, ARGS | T id
E →  Int | Id | if (E1 == E2) then E3 else E4
     | E1 + E2 | E1 - E2 | id(E1,…,En)
```

T is only int

in this language there is no variable declaration — there are formal parameters only!

Int sono costanti interi, **ID** sono gli identificatori, l'espressione più a destra è il "main"

Esempio: il programma per calcolare i numeri di Fibonacci

```
int fib(int x) = if (x == 1) then 0
                 else if (x = 2) then 1
                 else fib(x - 1) + fib(x - 2) ;
fib(5)
```

Strategia di generazione del codice

per ogni espressione **e** generiamo codice assembly che:

- calcola il valore di **e** e lo memorizza in **$a0** (**$a0** è un registro speciale chiamato **accumulatore**)
- **conserva $sp** e il **contenuto dello stack**

definiamo una **funzione di generazione del codice: cgen(Env stable, Node e)**

il cui risultato è il codice generato per **e**

invariante: il risultato del calcolo di un'espressione **è sempre nell'accumulatore, dopo** aver calcolato un'espressione **lo stack è come prima**

l'invariante deve essere soddisfatto da cgen(stable, e)

Generazione di codici per costanti e add

il codice per **valutare una costante** la copia semplicemente nell'accumulatore: **cgen(stable, i) = li $a0 i,** questo preserva lo stack, come richiesto

il codice per **valutare un'espressione add** è:

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

this is ok, but be carefull!
you need to verify that the
old value of $t1 is useless!

questo codice preserva lo stack, come richiesto

possibile ottimizzazione: mettere il risultato di e1 direttamente nel registro $t1?

Un altro codice per aggiungere

Possibile ottimizzazione: metti il risultato di **e1** direttamente in **$t1**

```
cgen(stable,e1 + e2) =
                       cgen(stable,e1)
                       move $t1 $a0   // $t1 ← $a0
                       cgen(stable,e2)
                       add $a0 $t1 $a0
```

questo è sbagliato!

Prova a generare il codice per: **3 + (7 + 5)**

Note sulla generazione del codice

1. Il codice per + è un modello con "**holes**" per il codice per valutare **e1** ed **e2**
2. La generazione di codice macchina pila è **ricorsiva**
3. Il codice per **e1 + e2** è costituito da codice per **e1** ed **e2 incollato insieme**
4. La generazione di codice può essere scritta come una **visita ricorsiva-discesa** dell'ast

Generazione codice per sub

c'è una nuova istruzione **sub $r1 $r2 $r3** che implementa **$r1 ← $r2 - $r3**

il codice è

```
cgen(stable, e1 - e2) =
                       cgen(stable,e1)
                       push $a0
                       cgen(stable,e2)
                       $t1 ← top
                       sub $a0 $t1 $a0
                       pop
```

questo codice preserva lo stack, come richiesto

il vecchio valore di $t1 è inutile

Generazione del codice per condizionato

abbiamo bisogno di **istruzioni per il controllo del flusso: beq $r1 $r2 label**

ramo da **label** se **$r1 = $r2**

un'altra istruzione di ramificazione: **b label** (**salto incondizionato** alla **label**)

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

we have added labels

the code for `if e1 = e2 then e3 else e4` :

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

the labels must be fresh to avoid name clashes

false_branch is useless!

RECAP OF THE BYTECODE

the bytecode language is the following one (up-to now)

```
bytecode =  ( lw $r1 offset($r2)
            | add $r1 $r2 $r3
            | sub $r1 $r2 $r3
            | sw $r1 offset($r2)
            | addi $r1 $r2 n
            | li $r1 n
            | move $r1 $r2
            | beq $r1 $r2 label
            | b label
            | label: )*          ;
```

we have used the registers `$a, $t1, $sp`

Il record di attivazione

Il codice per le chiamate di funzione e le definizioni delle funzioni dipende dal **layout del record di attivazione**

Un AR molto semplice è sufficiente per questa lingua:

- Il risultato è sempre nell'**accumulatore**: non c'è bisogno di memorizzare il risultato nell'AR
- Il record di attivazione **contiene parametri effettivi:** for f (**x1, ..., xn**) push **xn, ..., x1** on the stack, queste sono le uniche variabili in questa lingua
- La disciplina dello stack garantisce che sulla funzione Exit **$sp** sia la stessa che era sulla voce della voce: **non** è **necessario** memorizzare **$sp** nell'ar

- Dobbiamo memorizzare l'**indirizzo di ritorno**

e il **link di accesso**?

- il collegamento di accesso **non è necessario** perché non abbiamo dichiarazioni nidificate
- abbiamo solo dichiarazioni e funzioni di parametri locali che sono tutte dichiarate nell'ambito globale che è allocato staticamente

è necessario avere un puntatore ad una **posizione di riferimento** nell'AR corrente

questo puntatore è memorizzato nel registro **$fp** (**frame pointer**): **$fp punta alla parte "statica" del record di attivazione**

la posizione di riferimento fa parte della progettazione del layout AR: prendiamo **$fp** per puntare alla posizione del primo parametro

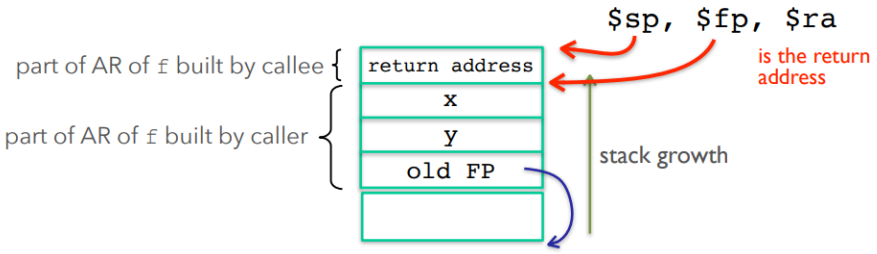
$fp viene utilizzato dal codice generato per individuare gli elementi AR, ad es. parametri, in base agli offset

dobbiamo memorizzare il **$fp** del chiamante nell'AR, in modo da poterlo ripristinare: in generale per il collegamento ad una AR utilizziamo l'indirizzo della sua posizione di riferimento

Sommario

Per la nostra lingua semplice, un AR con il puntatore del telaio del chiamante (collegamento di controllo), i parametri effettivi e l'indirizzo di ritorno è sufficiente

Considera una chiamata a **f (x, y)**, l'AR sarà:



Generazione codice per chiamata funzione

la sequenza di chiamata è costituita da istruzioni (sia del chiamante che del chiamato) che impostano una chiamata di funzione

nuova istruzione: **jal label**

salta alla label, salva l'indirizzo della prossima istruzione in **$ra**

```
cgen(stable,f(e1,…,en)) = push $fp
                          cgen(stable,en)
                          push $a0
                          . . .
                          cgen(stable,e1)
                          push $a0
                          jal f_entry
```

- il chiamante salva il valore del puntatore del frame
- quindi salva i parametri effettivi in ordine inverso
- quindi salva l'indirizzo di ritorno nel registro **$ra**
- l'AR finora è lungo **4*n+4** byte

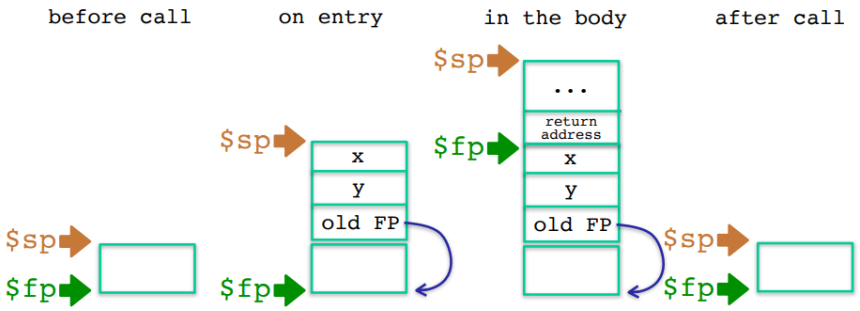
Generazione del codice per la definizione della funzione

nuova istruzione: **jr $ra // passa all'indirizzo in $ra**

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

- il puntatore della cornice non punta alla parte inferiore della cornice
- il chiamato visualizza l'indirizzo di ritorno, gli argomenti effettivi e il valore salvato del puntatore del frame
- k = 4*n + 4 // equivalente a pop n+1 volte
- il valore restituito viene lasciato in $a0

CALLING SEQUENCE: EXAMPLE FOR `F(X,Y)`



Poiché la **pila cresce quando i risultati intermedi vengono salvati**, AR **non** sono **adiacenti sullo stack!**

Esempio: **int f () = 3 + g (5)**

All'esecuzione, 3 è sullo stack tra l'AR di **F** e l'AR di **G**

Generazione di codice per variabili

Nella nostra lingua semplice le "variabili" di una funzione sono solo i suoi parametri

- Sono tutti nell'AR
- spinto dal **chiamante**

Problema: poiché la pila cresce quando i risultati intermedi vengono salvati, le variabili **non** sono **a un offset fisso** da **$sp**

Esempio: **int f (x) = 3 + x**

Soluzione: utilizzare il **puntatore del frame,** punta sempre alla prima variabile

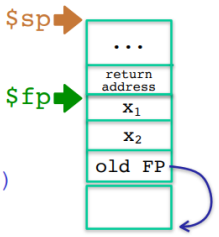
CODE GENERATION FOR VARIABLES: EXAMPLE

for `int f(x1,x2) = e` the activation and frame pointer are set up as follows:

- x_1 is at `$fp`
- x_2 is at `$fp + 4`
- thus, the access to x_i is

  `cgen(stable,xi) = lw $a0 z($fp)`

  with `z = 4*(i-1)`
- the offset of a parameter needs to be **inserted in its symbol table entry**



Sommario

il **layout del record di attivazione** deve essere progettato insieme al generatore di codice

la generazione del codice può essere eseguita mediante l'attraversamento ricorsivo dell'AST

per semplificare la presentazione non abbiamo discusso i link di accesso!

ma **i collegamenti di accesso possono essere facilmente aggiunti** (come discusso in "gestione della memoria"): verificare la differenza nel livello di annidamento tra il chiamante e la dichiarazione del chiamato e seguire di conseguenza i link di accesso già stabiliti

RECAP OF THE BYTECODE

the bytecode language is the following one

```
bytecode =  ( lw $r1 offset($r2)
            | add $r1 $r2 $r3
            | sub $r1 $r2 $r3
            | sw $r1 offset($r2)
            | addi $r1 $r2 n
            | li $r1 n
            | move $r1 $r2
            | beq $r1 $r2 label
            | b label
            | label:
            | jal label
            | jr $r1 )*          ;
```

in red: new instructions and new registers

we have used the registers `$a, $t1, $sp, $ra, $fp`

THE SMALL LANGUAGE WITH ACCESS LINKS AND STATEMENTS

the small language grows . . .

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

- `int` and `void` are the **types**
- there are **nested declarations** of functions
- **example:**

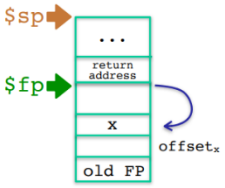
```
void foo(int x,int y) = void gee(int z) x = x+z ;
                        if y != 0 then ( gee(y) ; foo(x,y-1) ; )
                        else skip ;
foo(0,10) ;
```

skip is e.g. x=x;

STATEMENTS

we discuss assignments `x = e ;`

- **simple case:** `x` is in the current RA
- $offset_x$ is offset with respect to `$fp`

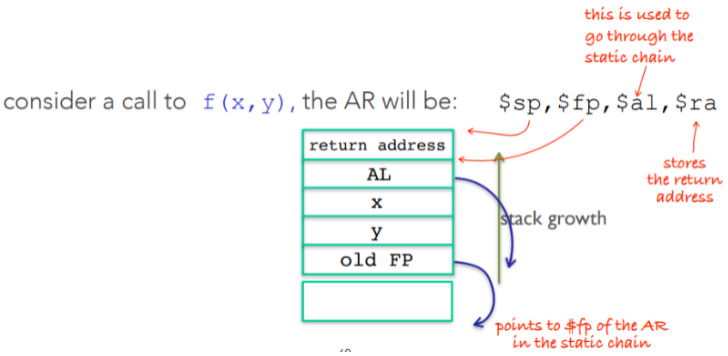


- thus `cgen(stable,x = e;) =`
  `cgen(stable,e)`
  `sw $a0 offset`$_x$`($fp)`

where is $offset_x$?

L'accesso alle variabili globali: collegamenti di accesso

L'AR per la nostra lingua ha: il puntatore del frame del chiamante, il **collegamento di accesso** all'ambiente di allevamento (nella catena statica), i parametri effettivi e l'indirizzo di ritorno




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

gives the nesting level of x

gives the offset of x inside the corresponding AR

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

exercise: define `cgen(stable, S ; S')`

Generazione codici per chiamata funzione con collegamenti di accesso

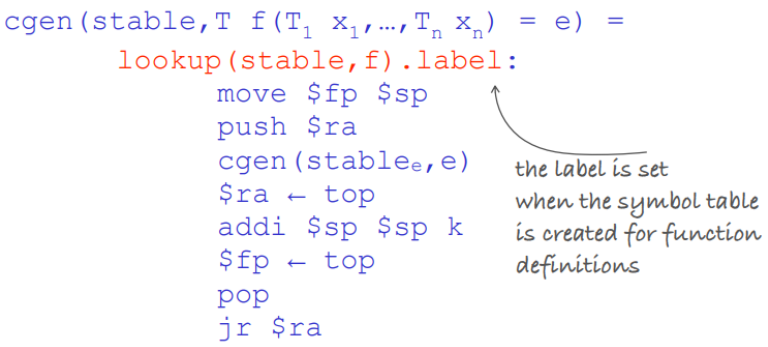
```
cgen(stable,f(e1,…,en)) =
        push $fp
        cgen(stable,en)
        push $a0
        . . .
        cgen(stable,e1)
        push $a0
        lw $al 0($fp)
        for (i=0;
             i < nesting_level -
                 lookup(stable,f).nesting_level;
             i++) lw $al 0($al) ;
 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

management of the access link

the label is taken from the symbol table

Nuova generazione di codici per la definizione delle funzioni

possono esserci più funzioni chiamate allo stesso modo: la label della prima istruzione deve essere presa dalla tabella dei simboli!



Argomenti avanzati

l'implementazione di **funzioni di ordine superiore**

l'implementazione di **oo linguaggi** e di **invio dinamico**

Funzioni di ordine superiore

in alcune lingue, le funzioni possono essere passate come parametri

```
fun bool f( x:(int,int)→bool, a:int, b:int){
    ...
    bool z = x(a,b) ;
    ...
    }
```

la funzione **f** dovrebbe preparare il record di attivazione per l'esecuzione di **x(a, b)**

- **PROBLEMA: f** non sa come impostare il “link di accesso”
- dovrebbe puntare **all'AR più recente in cui viene dichiarata la funzione chiamata** (in base alla differenza con il livello di annidamento di tale dichiarazione, calcolata in fase di compilazione)

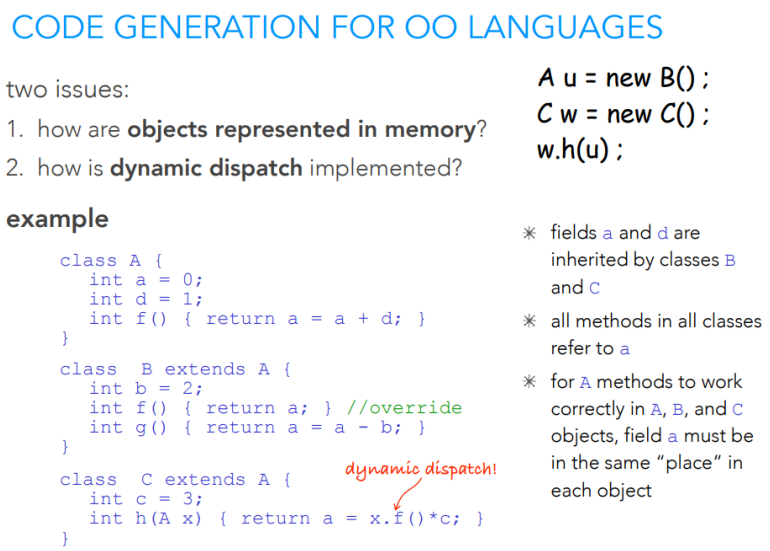
soluzione: il chiamante di **f**, che passa il valore effettivo per il parametro **x**, dovrebbe anche passare una coppia contenente:

1. l'indirizzo del codice della funzione **g** effettivamente passato (valore abituale dell'identificatore **g**)
2. l'indirizzo dell'AR più recente in cui è dichiarato **g** (informazioni di contesto aggiuntive)

questa coppia è chiamata **closure** della funzione da passare

il valore di **x** è impostato su tale coppia (i valori degli identificatori del tipo di funzione sono coppie)

quando **x(a, b)** viene eseguito, il collegamento di accesso verrà impostato con il secondo elemento di questa coppia



Problema 1: disposizione dell'oggetto

un oggetto è come una **struttura** in C

- il riferimento **foo.field** è un indice in una struttura **foo** a un offset corrispondente a **field**
- un oggetto è archiviato in una memoria contigua
  - ogni campo memorizzato a un offset fisso nell'oggetto
  - l'offset deve essere inserito nella relativa voce della tabella dei simboli
- alla creazione dell'oggetto, il layout corrispondente viene istanziato nell'heap (verrà eventualmente rimosso dal Garbage Collector)

Problema 1: rappresentazione oggetto di sottoclassi

Dato un layout per la classe **A**, un layout per la sottoclasse **B** può essere definito **estendendo il layout** di **A** con slot aggiuntivi per i campi aggiuntivi di **B:** questo lascia **invariato** il layout di **A** (il layout di **B** ne è un'estensione)

Problema 2: Dispatch dinamico

Considera ancora il nostro esempio

```
class A {
   int a = 0;
   int d = 1;
   int f(){ return a = a + d; }
}

class  B extends A {
   int b = 2;
   int f(){ return a; }//override
   int g(){ return a = a - b; }
}

class  C extends A {
   int c = 3;
   int h(A x) { return a = x.f()*c; }
}
```

- E.G () Chiamate Metodo G di B Se reta un oggetto B

- E.F () Chiamate Metodo F di A se i rendimenti IF e un oggetto A o C (F è ereditato nel caso di c) Chiamate Metodo F di B se la restituisce un oggetto B (anche se il tipo statico di E è A)

L'implementazione dei metodi e della spedizione dinamica **assomiglia fortemente all'attuazione dei campi**

Problema 2: tabelle di dispatch/discpatch dinamiche

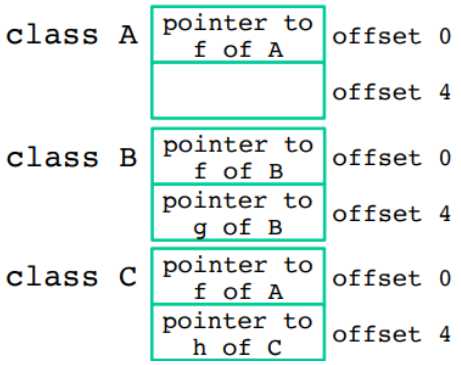
ogni classe ha un insieme fisso di metodi (inclusi i metodi ereditati) una tabella di invio indicizza questi metodi

- un array di indirizzi di metodo (**questo è anche chiamato VIRTUALE TABLE**)
- un metodo f vive a un offset fisso nella tabella di invio per una classe e tutte le sue sottoclassi

esempio: la tabella di spedizione per la classe **A** ha solo 1 metodo

le tabelle per **B** e **C** ampliano la tabella per **A**

poiché i metodi possono essere sovrascritti, il codice per **f** non è lo stesso in tutte le classi, ma è sempre allo stesso offset



Utilizzo delle tabelle dispatch

il puntatore di spedizione in un oggetto di classe **C** punta alla tabella di spedizione per la classe **C**

a ogni metodo **f** della classe **C** viene assegnato un offset **Of** nella tabella di invio in fase di compilazione: l'offset viene inserito come di consueto nella voce della tabella dei simboli del metodo **f** di classe **C**

per implementare un invio dinamico **e.f()** we

- sia **Of** l'offset del metodo **f** nella tabella di spedizione associata al tipo statico di **e**
- valutare **e**, ottenendo un oggetto **o** (che potrebbe essere di qualsiasi sottoclasse)
- sia **D** la tabella di spedizione di **o**
- eseguire il metodo indicato da **D[Of]**

THE SVM GRAMMAR

point to MEMORY[] — points to CODE[]

use MEMORY to store data; use registers SP, RA, RV, FP, HP, IP,

```
assembly: ('push' NUMBER     //push NUMBER on the stack
       |'push' LABEL    //push the location address pointed by LABEL on the stack
       |'pop'           //pop the top of the stack
       |'add'           //pop the two values x,y on top of the stack and push x+y
       |'sub'           //pop the two values x,y on top of the stack and push y-x
       |'mult'          //pop the two values x,y on top of the stack and push x*y
       |'div'           //pop the two values x,y on top of the stack and push y/x
       |'sw'         //pop the two values x,y on top of the stack and do MEMORY[x]=y
       |'lw'            //pop the value x on top of the stack and push MEMORY[x]
       | LABEL ':'      //LABEL points at the address of the subsequent instruction
       |'b' LABEL       //jump at the instruction pointed by LABEL
       |'beq' LABEL     //pop two values x,y on top of the stack and jump if x==y
       |'bleq' LABEL    //pop two values x,y on top of the stack and jump if x>=y
       |'js'           //pop x on top of the stack, copy the IP in RA and jump to x
       |'lra'          //push in the stack the value of RA
       |'sra'          //pop x on top of the stack and copy x in RA
       |'lrv'          //push in the stack the value of RV
       |'srv'          //pop x on top of the stack and copy x in RV
       |'lfp'          //push in the stack the value of FP
       |'sfp'          //pop x on top of the stack and copy x in FP
       |'cfp'          //copy in FP the value of SP
       |'lhp'          //push in the stack the value of HP
       |'shp'          //pop x on top of the stack and copy x in HP
       |'print'        //print the top of the stack without removing it
       |'halt'         //terminate the execution
       )* ;
```

Commenti su SVM

non usiamo più $a0 ma utilizziamo l'elemento in cima allo stack: la valutazione di un'espressione è in cima allo stack (**dopo la valutazione, lo stack è come prima, ad eccezione dell'elemento in alto**).

Esempio

$$cgen(stable, e_1 + e_2) = cgen(stable, e_1)\ cgen(stable, e_2)\ add$$

THE SVM GRAMMAR

- the code we have seen for function invocation

$cgen(stable, f(e_1, \ldots, e_n))$ = push $fp
$cgen(stable, e_n)$
push $a0
. . .
$cgen(stable, e_1)$
push $a0
jal f_entry

jump to f_entry and save the address in $ra
// omitted the management of AL

return address
AL
PARAMETRI
old FP

- the code in the FOOL compiler

Dichiarazioni Locali
Access Link
PARAMETRI ATTUALI
Frame Pointer

```
public String codeGeneration() {
    String parCode="", getAR="";
    for (int i=parlist.size()-1; i>=0; i--)
        parCode+=parlist.get(i).codeGeneration();
    for (int i=0; i<nestinglevel-STentry.getNestinglevel(); i++)
        getAR+="lw\n";
    return "lfp\n"+       // push del FP
           parCode+       // inserimento dei valori dei par. attuali sulla pila
           "lfp\n"+getAR+ // risalgo la catena statica per recuperare l'AL
           "push "+STentry.getOffset()+"\n"+ // metto offset sullo stack
           "lfp\n"+getAR+ // risalgo la catena statica per recuperare AL per
                          //indirizzo della funzione (si poteva copiare il valore preced.)
           "add\n"+
           "lw\n"+        // push del valore all'indirizzo ottenuto
           "js\n";
}
```

codice per individuare il puntatore all'ambiente statico