

01-interpreti-compilatori

- [Interprete e Compilatore](#)
 - [Interprete VS Compilatore](#)
- [Struttura compilatore](#)
 - [Front-End](#)
 - [Lexer](#)
 - [Parser](#)
 - [Checker](#)
 - [Back-End](#)
 - [Instruction Selection](#)
 - [Register allocation](#)
 - [Instruction Scheduling](#)
 - [Middle-End](#)
 - [Passi di analisi](#)
 - [Passi di trasformazione](#)

Interprete e Compilatore

Un **interprete** è un programma che prende in input un programma eseguibile (espresso nel linguaggio L) e lo **esegue**, producendo l'output corrispondente.

Un **compilatore** è un programma che prende in input un programma eseguibile (espresso nel linguaggio L) e lo **traduce**, producendo in output un programma equivalente (espresso nel linguaggio M).

Per eseguire il compilato serve un interprete per il linguaggio M

Il compilatore traduce il programma in modo da ottenere un miglioramento di qualche metrica (tempo di esecuzione, memoria usata, consumo energetico, ...).

Interprete VS Compilatore

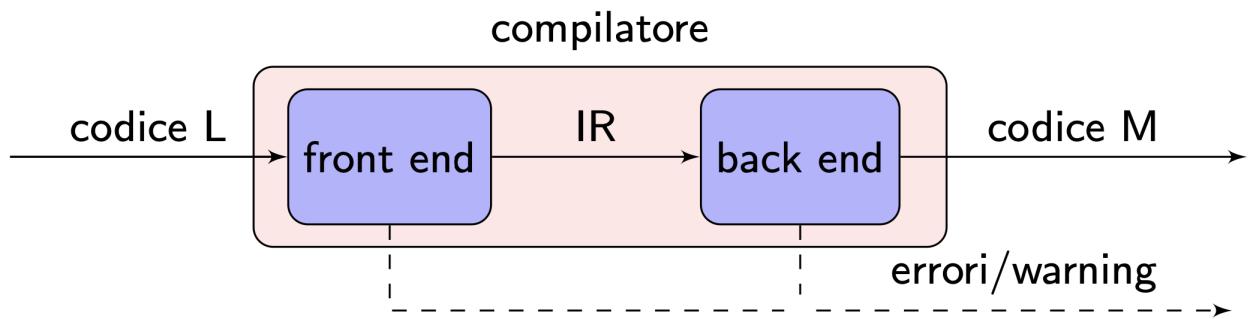
La compilazione è un'attività che viene compiuta off-line (non a run-time) questo perché il compilatore deve identificare alcuni errori di programmazione prima dell'esecuzione del programma, deve migliorare l'efficienza e deve rendere utilizzabili alcuni costrutti dei linguaggi ad alto livello.

Esistono approcci che sono tipicamente compilati (C, C++; Pascal, ...), approcci tipicamente interpretati (PHP, Matlab, ...) e approcci misti (Java, Python, SQL, ...).

Quindi è meglio interpretare o compilare? In realtà non c'è un migliore, dipende dalle necessità, l'importante è stabilire compromessi:

- Bilanciamento tra attività off-line ed on-line
- Il tempo di compilazione dev'essere accettabile
- L'occupazione in spazio del programma compilato deve essere accettabile

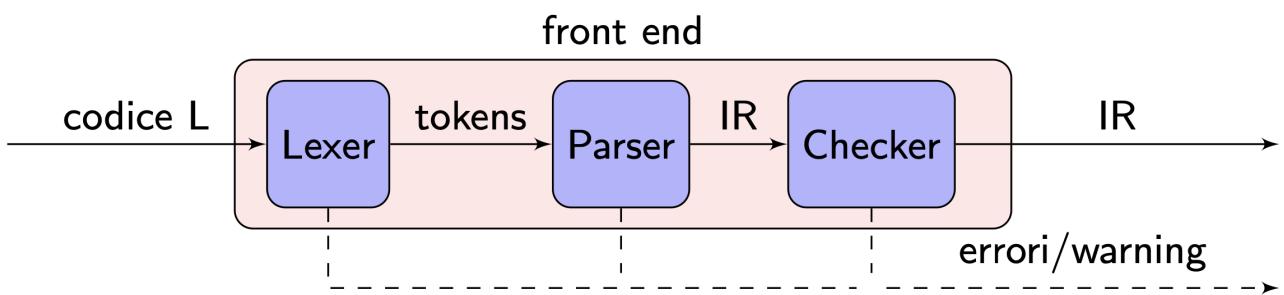
Struttura compilatore



Front-End

Il front-end è in grado di riconoscere programmi validi (e invalidi), segnalare errori e warning facilmente leggibili e produce codice IR (rappresentazione intermedia).

Esso si decompone nel seguente modo:



Come primo passo abbiamo il **Lexer**, che si occupa dell'analisi lessicale, questo ha il compito di suddividere il codice L in una sequenza di **token**, che darà in pasto al **Parser**.

Questo si occupa di eseguire l'analisi sintattica (correttezza della *struttura*) indipendente dal contesto, esso produce una rappresentazione intermedia (IR) e quindi il suo compito è riordinare la sequenza di token secondo una sintassi definita, spesso però l'analisi sintattica non è sufficiente per garantire la completa traduzione ed entra in aiuto il **Checker**.

Quest'ultimo si occupa dunque dell'analisi semantica (correttezza del *significato*) dipendente dal contesto.

Lexer

Prende in input una sequenza di caratteri e restituisce in output una sequenza di token, costruiti in questo modo: *token* = ⟨parte del discorso, lessema⟩, ovvero parte del discorso indica di che tipologia è la parola che ho letto, mentre il lessema è la parola letta.

Esempi: ⟨*KWD*, while⟩ (ho letto la keyword while), ⟨*IDENT*, somma⟩, ⟨*INT*, 42⟩, ⟨*STR*, "Hello"⟩.

Come definisco in modo rigoroso quali sono i token validi? Lo faccio definendo un linguaggio adeguato come le espressioni regolari che permettono di identificare schemi specifici all'interno di linguaggi, quindi un linguaggio comprensibile all'essere umano.

Quando implementiamo un lexer dobbiamo includere un riconoscitore che si occupa dunque di identificare e classificare i token, esso è un DFSA, cioè un automa a stati finiti deterministico, questo viene spesso generato automaticamente partendo dalla specifica.

Esempio:

- DIGIT = [0-9]
- LETTER = [a-zA-Z] | []
- ID = LETTER (LETTER | DIGIT)*

Abbiamo i digit che è un numero da 0 a 9, lettere che è una lettera dell'alfabeto minuscola o maiuscola oppure l'underscore e infine abbiamo gli ID che sono composti da una lettere seguita da una sequenza di lettere o numeri che possono apparire zero o più volte.

Parser

Prende in input una sequenza di token e restituisce in output una rappresentazione IR della struttura sintattica, creando un AST (Abstract Syntax Tree).

Viene definito un linguaggio adatto per implementarlo, utilizzando grammatiche libere dal contesto, significa che quando leggo un carattere non mi interessa cosa viene prima o dopo di esso perché appunto sto affrontando un analisi sintattica.

Il riconoscitore che dovremmo implementare è un PDA (automa a pila non deterministico).

Checker

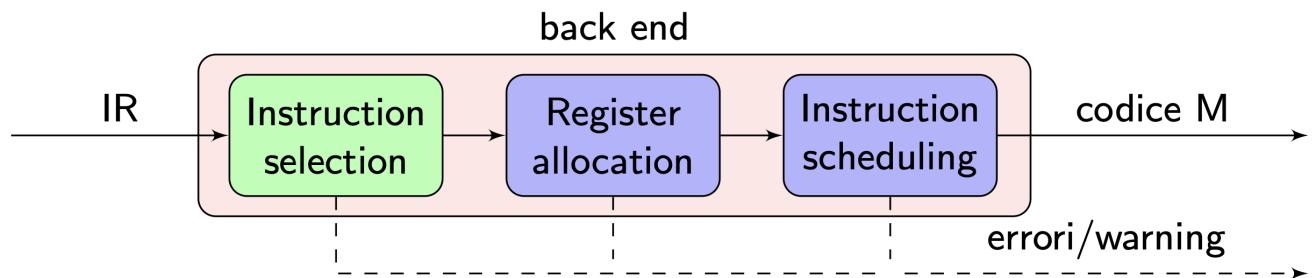
In input abbiamo un AST generato dal parser che viene definito grezzo in quanto il checker ha il compito di arricchirlo ed ultimarla restituendolo in output.

Come posso definire in modo rigoroso quali programmi sono validi? Lo posso fare tramite il

linguaggio naturale servendomi dello **standard** del linguaggio, della documentazione del compilatore ecc., utilizzando anche semantiche formali (sistemi di regole).

Back-End

Si occupa di tradurre da IR a linguaggio M (macchina), decide quali valori mantenere nei registri, sceglie le istruzioni per implementare le operazioni e rispetta l'interfaccia di sistema. Esso viene decomposto nel modo seguente:



Instruction Selection

Si occupa di traduzione della rappresentazione intermedia (IR) del codice sorgente in istruzioni specifiche per l'architettura di destinazione (la macchina su cui eseguiamo il codice).

Register allocation

Si occupa di assegnare variabili e temporanei a registri del processore in modo efficiente, massimizzando l'uso di queste risorse e minimizzando l'accesso alla memoria. Le tecniche di register allocation sono cruciali per migliorare le prestazioni del codice generato, riducendo il tempo di esecuzione e l'overhead associato all'accesso alla memoria (LOAD e STORE).

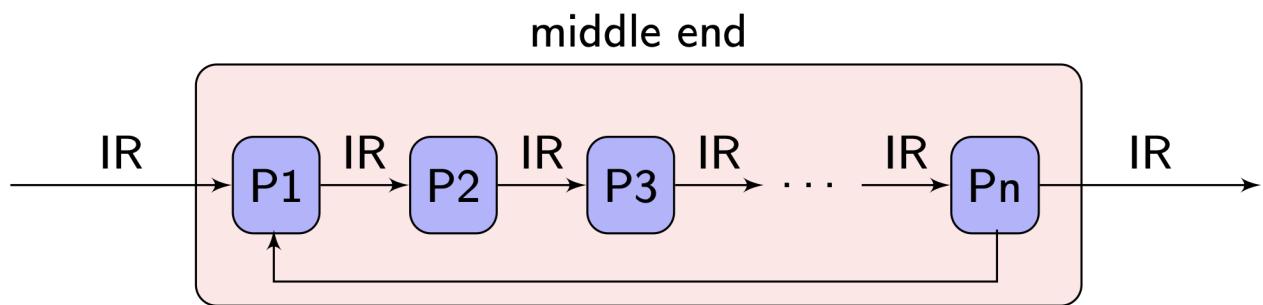
Instruction Scheduling

Si occupa di ottimizzare l'ordine delle istruzioni per migliorare l'efficienza dell'esecuzione. Attraverso tecniche di scheduling statico e dinamico, il compilatore cerca di massimizzare l'uso delle risorse del processore, ridurre i tempi di attesa e minimizzare il numero totale di cicli di clock necessari per l'esecuzione del programma.

Middle-End

Abbiamo anche una fase intermedia tra front-end e back-end, chiamata appunto middle-end. Il suo compito è analizzare il codice IR e trasformarlo, con l'obiettivo di migliorarlo preservando la semantica del programma.

Il middle-end viene decomposto in più passi, ognuno di essi implementa un'**analisi** o **trasformazione** dell'IR e un passo può dipendere o invalidare altri passi.



Passi di analisi

- identificazione di valori costanti
- identificazione di codice o valori inutili
- analisi di aliasing

Passi di trasformazione

- propagazione di valori costanti
- rimozione di codice inutile
- inlining di chiamate a funzione
- loop unrolling ("srotolo" i loop per poi ottimizzarli)

02-analisi-lessicale

- [RE - NFA - DFA](#)
 - [NFA](#)
 - [NFA vs DFA](#)
 - [DFA](#)
- [Token](#)
- [Flex](#)
 - [1 - Sezione delle definizioni](#)
 - [Literal block](#)
 - [Pattern con nome](#)
 - [Opzioni per flex](#)
 - [Start States](#)
 - [2 - Sezione delle regole](#)
 - [Regole lessicali](#)
 - [3- Sezione del codice utente](#)

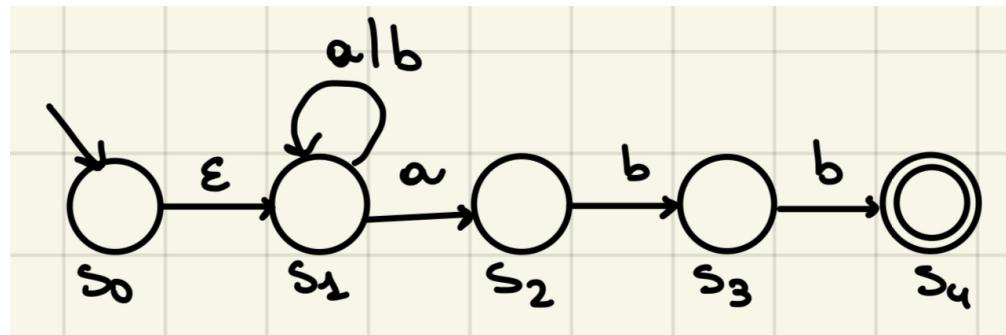
RE - NFA - DFA

Nel seguente paragrafo costruiremo un automa a stati finiti deterministico (DFA) per riconoscere un'espressione regolare (RE).

Come prima cosa, partendo da un'espressione regolare, costruiremo un automa a stati finiti non deterministico (NFA). Successivamente dal NFA passeremo all'automa a stati finiti deterministico (DFA) per poi minimizzarlo (renderlo più semplice).

NFA

Costruiamo un NFA per la seguente RE: $(a|b)^*abb$



Partendo dallo stato iniziale S_0 e seguendo le frecce possiamo ricostruire la RE precedente, che ci diceva: "possiamo avere una sequenza di zero o più (*) a o b, l'importante è che la stringa si concluda con la sequenza abb".

Dunque se noi dovessimo avere una stringa del tipo "baabababb" in questo caso il nostro automa sarebbe in grado di riconoscerla e accettarla, analizziamola:

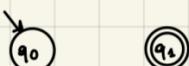
Carattere letto	Stato corrente	Stato successivo
leggo nulla	S_0	S_1
b	S_1	S_1
a	S_1	S_1
a	S_1	S_1
b	S_1	S_1
a	S_1	S_1
b	S_1	S_1
a	S_1	S_2
b	S_2	S_3
b	S_3	S_4

Finiamo dunque in uno stato finale accettante S_4 quindi l'espressione è corretta e accettata dall'automa.

le ϵ transizioni permettono all'automa di passare da uno stato a un altro **senza leggere alcun simbolo dell'input**. In altre parole, l'automa può "saltare" da uno stato a un altro senza consumare nessun carattere della stringa in input.

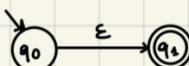
Vediamo più nella teoria come si costruisce un NFA, tramite la costruzione di **Thompson**. Generalizziamo i casi base:

1) Linguaggio vuoto $\{\emptyset\}$:

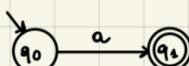


(non esistono transizioni)

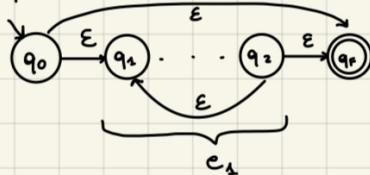
2) Linguaggio vuoto $\{\epsilon\}$:



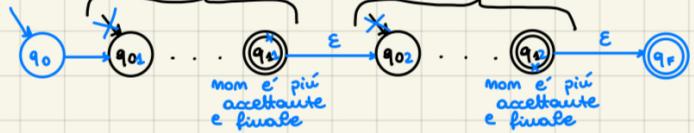
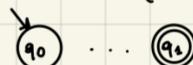
3) Leggo un singolo simbolo dell'alfabeto $\{a\}$:



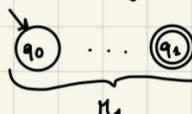
6) Espressione ripetuta 0 o più volte: e_1^*



4) Concatenazione di due espressioni $\{e_1 \rightarrow M_1\} \& \{e_2 \rightarrow M_2\}$:



5) Disjunzione di due espressioni $\{e_1 \rightarrow M_1\} \mid \{e_2 \rightarrow M_2\}$:

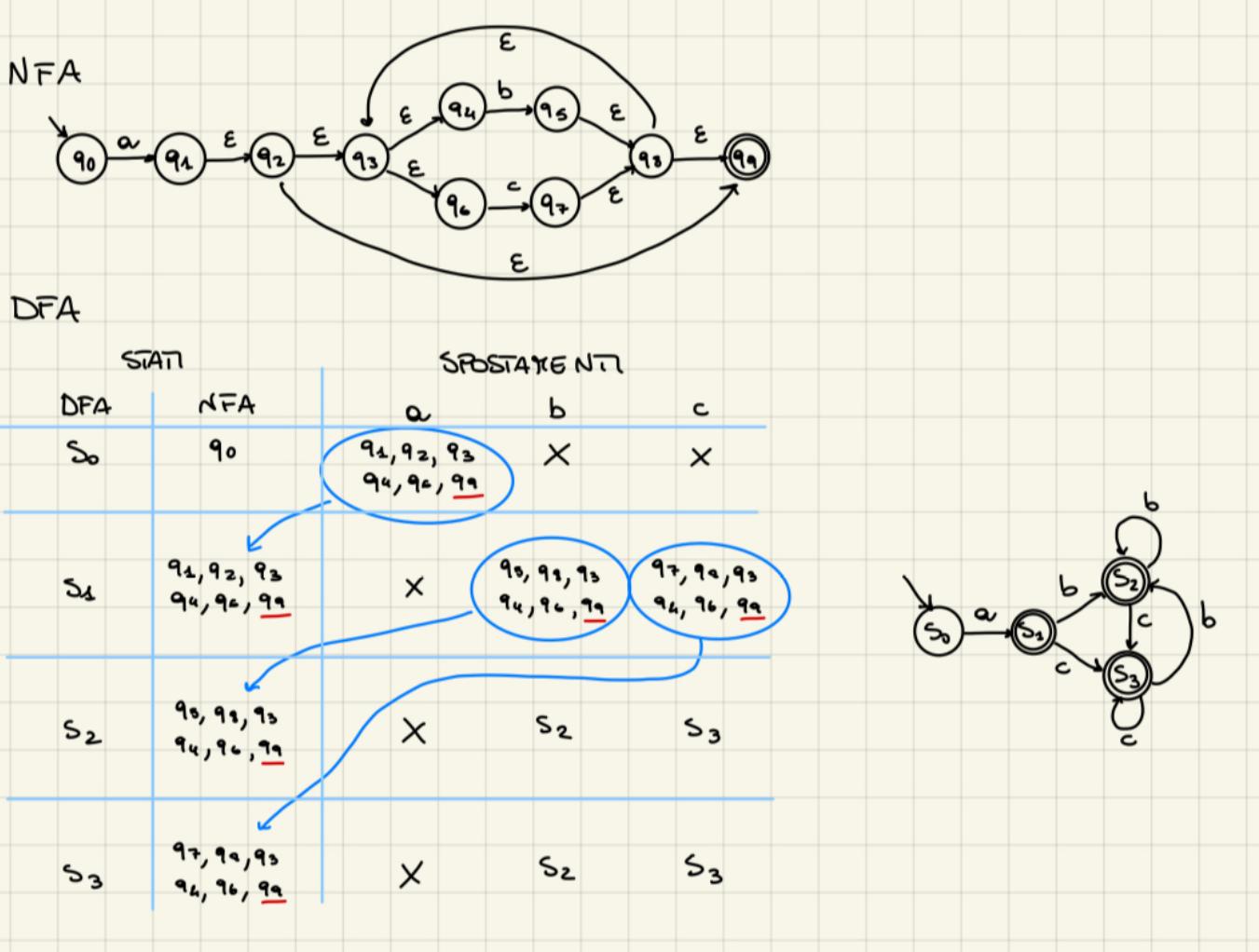


NFA vs DFA

Un automa a stati finiti deterministico è un caso speciale di automa a stati finiti non deterministico, questo perché un DFA non contiene le ϵ transizioni, di conseguenza questo porta anche al fatto che un DFA sia deterministico e quindi significa che per ogni stato e simbolo dell'alfabeto esiste una sola transizione verso un altro stato specifico (non ci sono ulteriori opzioni), cosa che nel NFA non accade in quanto non è deterministico, quindi possiamo avere più di una transizione o anche nessuna.

Vediamo nell'esempio precedente che quando mi trovo nello stato S_1 e leggo il carattere a , posso avere diverse transizioni in stato differenti, infatti posso continuare a rimanere nello stato S_1 oppure passare allo stato S_2 .

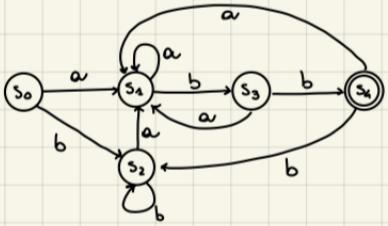
DFA



Vediamo come siamo passati da un NFA a un DFA:

- il nostro stato S_0 del DFA corrisponde allo stato q_0 dell'NFA, a questo punto, per ogni lettera presente nel nell'NFA guardo quali stati attraverso partendo da q_0 (contando anche le ϵ), nel nostro caso con la lettera a attraverso gli stati $\{q_1, q_2, q_3, q_4, q_6, q_9\}$, mentre per le lettere b e c non faccio spostamenti in quanto partendo da q_0 devo per forza avere una a per passare agli stati successivi.
 - Il nuovo stato S_1 dunque corrisponderà all'insieme di stati trovati prima, ora da un qualsiasi stato dichiarato con la lettera a non mi sposto in un nessun nuovo stato, mentre con b posso raggiungere nuovi stati $\{q_5, q_8, q_3, q_4, q_6, q_9\}$, mentre con il carattere c raggiungo gli stati $\{q_7, q_8, q_3, q_4, q_6, q_9\}$.
 - Questi stati trovati precedentemente diventano nuovi stati del DFA S_2 e S_3 , a questo punto da S_2 con il carattere b incontro nuovamente ancora tutti gli stati di S_2 , stessa cosa per c che incontro nuovamente tutti gli stati di S_3 .
 - Infine nello stato S_3 , si ripete ancora la stessa cosa precedente.
- Da qui costruiamo il DFA.

Ora vediamo come **minimizzare** il DFA:



	a	b
S0	S1 S2	
S1	S1 S3	
S2	S1 S2	
S3	S2 S4	
S4	S4 S2	

1) $\{ \underline{S_0}, \underline{S_1}, \underline{S_2}, \underline{S_3} \}$ $\{ \underline{S_4} \}$

2) $S_0 \rightarrow \boxed{S_1 \text{ e } S_2}$
 $S_1 \rightarrow \boxed{S_1 \text{ e } S_3}$

questi stanno stati stanno
in nello stesso set, quindi S_0 e S_1 sono
equivalenti.

3) $\{ \underline{S_0}, \underline{S_1}, \underline{S_2} \}$ $\{ \underline{S_3} \}$ $\{ \underline{S_4} \}$

$S_0 \rightarrow \{ S_1 \text{ e } S_2 \}$ sono equivalenti
 $S_2 \rightarrow \{ S_1 \text{ e } S_2 \}$ quindi lo aggiungiamo al set.

$S_2 \rightarrow \{ S_1 \text{ e } S_2 \}$ NON sono equivalenti perché
 $S_3 \rightarrow \{ S_1 \text{ e } \cancel{S_4} \}$ S_4 non è nello stesso set di S_2 e S_3 .
Lo mettiamo in un altro set separato.

4) $\{ \cancel{S_0}, \cancel{S_1}, \cancel{S_2} \}$ $\{ \underline{S_3} \}$ $\{ \underline{S_4} \}$

$S_0 \rightarrow S_1 \text{ e } S_2 \}$ NON sono nello stesso set
 $S_4 \rightarrow S_1 \text{ e } \cancel{S_3} \}$ perché S_3 non è più nello
stesso set di S_0 e S_1 .

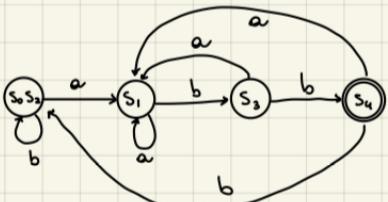
$\{ S_0 \}$ $\{ S_1 \}$ $\{ S_3 \}$ $\{ S_4 \}$

a questo punto separamo S_0 e S_1
quindi S_2 deve confrontarla con S_0
e S_1 :

$S_0 \rightarrow S_1 \text{ e } S_2 \}$ sono equivalenti
 $S_2 \rightarrow S_1 \text{ e } S_2 \}$ quindi mi ferma

$\{ S_0, S_2 \}$ $\{ S_1 \}$ $\{ S_3 \}$ $\{ S_4 \}$

5) A questo punto possiamo **riconoscere** il DFA minimizzato:



Un DFA - Automa a Stati Finiti Deterministico è una quintupla $M = \langle \Sigma, Q, \delta, q_0, F \rangle$, dove:

- Σ : alfabeto finito ($\Sigma^* =$ insieme di tutte le stringhe finite sull'alfabeto Σ)
- Q : insieme finito degli stati dell'automa
- $\delta: Q \times \Sigma \rightarrow Q$: funzione di transizione
- $q_0 \in Q$ lo stato iniziale
- $F \subseteq Q$: sottoinsieme degli stati finali (accettanti)

Il linguaggio riconosciuto da un DFA è l'insieme delle stringhe che sono **accettate** dall'automa, partendo dallo stato iniziale q_0 , sono quelle per le quali la transizione estesa termina in una configurazione finale accettante.

Linguaggi formali:

- Σ : alfabeto finito
- Σ^* : insieme di tutte le stringhe finite sull'alfabeto Σ

- $\epsilon \in \Sigma^*$: indica la stringa vuota
- $L \subseteq \Sigma^*$: linguaggio
 - $\emptyset = \{\}$ è un linguaggio
 - $\{\epsilon\}$ è un linguaggio
 - Σ è un linguaggio
 - Σ^* è un linguaggio

Token

Come vengono classificati i token:

- parole chiavi
- identificatori
- costanti letterali (interi, floating point, stringa, ...)
- operatori (matematici, logici, ...)
- "punteggiatura" (parentesi, virgola, punto e virgola, ...)
- commenti (singola linea, multi linea)

Esempi:

- **Keyword:** if, then, else, while, ... (attenzione al case sensitive).
- **Identificatori:**
 - $[a-zA-Z][0-9a-zA-Z]^*$
 - $[a-zA-Z]([0-9] | [a-zA-Z])^*$
 - oppure
 - DIGIT = $[0-9]$
 - LETTER = $[a-zA-Z] | []$
 - LETTER (LETTER | DIGIT) *
- **Costanti:**
 - intere: DIGIT $^+$ (accetta 000000 non accetta -1)
 - floating point: $[+-]?[0-9]+.[0-9]$
 - carattere: $'[^']'$ ('^' = qualsiasi carattere che non sia l'apostrofo)

- **Operatori e punteggiatura**, ogni lessema ha la sua categoria lessicale:

lessema	categoria	lessema	categoria
(OPEN_PAREN)	CLOSE_PAREN
[OPEN_BRACKET]	CLOSE_BRACKET
{	OPEN_BRACE	}	CLOSE_BRACE
+	PLUS	-	MINUS
+=	PLUS_ASSIGN	-=	MINUS_ASSIGN
:	COLON	::	SCOPE
<	LESS_THAN	<<	SHIFT_LEFT
>	GREATER_THAN	>>	SHIFT_RIGHT
.	DOT	...	ELLIPSIS
...	...		

- **Commenti:**

- //[^\\n]* \\n (C++)
- --[^\\n]* \\n (SQL)
- commento multilinea:
 - ^*([^*] | * + [^/*])**/

Flex

Lo strumento flex è un generatore di analizzatori lessicali, che è quindi un compilatore.

Viene diviso in 3 sezioni generali:

1 - Sezione delle definizioni

Come prima sezione in un flexer abbiamo quella parte delle definizioni che può contenere:

- Literal Block
- Definizioni di pattern con nome
- Opzioni per flex
- Start states

Literal block

Il literal block è un blocco di codice C racchiuso da `%{ ... %}` a inizio riga, questo viene copiato verbatim (letteralmente così come lo scriviamo) nella parte iniziale del sorgente generato da flex, solitamente contiene:

- definizioni di costanti per categorie lessicali
- dichiarazioni di variabili (usate nelle regole)
- dichiarazioni di funzioni (invocate nelle regole)
- definizioni di funzioni inline

```

enum P_LANGUAGE {
    KEY_W = 1,
    IDENT,
    BOOL,
    INTEGER,
    FLOAT,
    CHAR,
    STRING,
    COMMENT,
};


```

Nell'esempio soprastante ho definito delle costanti che rappresentano i tipi di token che posso riconoscere e restituire, queste vengono utilizzate successivamente nella sezione delle regole per classificare i pattern.

Pattern con nome

Successivamente al literal block vengono definiti i pattern con nome, cioè i pattern riutilizzabili che vengono associati a nomi specifici che verranno poi utilizzati nelle regole del lexer. La sintassi di questa sezione è del tipo: NOME_PATTERN espressione_regolare.

```

DIGIT [0-9]
LETTER [a-zA-Z]
IDENTIFIER {LETTER}({LETTER}|{DIGIT})*
NUMBER {DIGIT}+
WHITE SPACE [\t\n]+

```

Nell'esempio vediamo che il pattern DIGIT rappresenta una cifra da 0 a 9, il pattern LETTER rappresenta una lettera maiuscola o minuscola, il pattern IDENTIFIER rappresenta un identificatore, formato da una lettera iniziale seguita da lettere o cifre, il pattern NUMBER rappresenta una sequenza di una o più cifre consecutive, infine, il pattern WHITESPACE rappresenta una sequenza di spazi bianchi (spazio, tabulazione, o nuova riga).

Come specifico i pattern in flex?

- Uso le **virgolette** per simboli non alfanumerici e le ***parentesi**
 - ("/*)([^*] | ("") +[^*/]) * ((***) + "/") questa espressione viene utilizzata per riconoscere i commenti multi-linea in C.
- Usare nomi di pattern per evitare ripetizioni:
 - STARS ("**")+ nella sezione delle definizioni
 - ("/*<")([^\"] | {STARS} +[^*/]) * ({STARS} "/")

Opzioni per flex

Sono due quelle da usare sempre:

- `%option noyywrap`: evita la generazione della funzione `yywrap()` e della sua chiamata fine input
- `%option nodefault`: evita la generazione della regola *catch-all* (`. ECHO;`) che causa la stampa dei token non riconosciuti.

Due sono quelle da usare quando utile:

- `%option yylineno`: definisce la variabile intera `yylineno` che mantiene il numero di riga della posizione corrente (la fine del lessema, usarla causa una perdita di efficienza)
- `%option case-insensitive`: rende case-insensitive i pattern, non modifica il file di input (i lessemi riconosciuti rimangono case-sensitive)

Start States

Servono a limitare l'applicabilità di alcune regole, le regole che vediamo successivamente si applicano quando il lexer è nello stato/condition INITIAL, possiamo definire altri stati/condition nella sezione delle definizioni, con la sintassi: `%x NOME_STATO`.

`%x` indica che si tratta di uno stato esclusivo, cioè che il lexer quando entra in quello stato deve uscire dagli altri stati, `%s` definirebbe uno stato shared, consentendo al lexer di essere contemporaneamente in più stati (complicato).

2 - Sezione delle regole

Questa sezione inizia dopo il marker `%%`, serve a fornire la definizione della funzione `int yylex()`, questa deve leggere un lessema dall'input e restituire al chiamante il token corrispondente.

La sezione quindi contiene le regole lessicali per riconoscere i token.

Regole lessicali

Il formato di ogni regola è: *pattern codice*.

Il pattern è un'**espressione regolare** che identifica una specifica sequenza di caratteri (lessema) che il lexer deve riconoscere, questo rappresenta la struttura dei caratteri che corrispondono a un token specifico.

Il codice è un blocco di codice associato al pattern, che viene eseguito quando il pattern viene riconosciuto nell'input, questo specifica cosa fare quando si incontra il lessema: in genere, restituire il token corrispondente o gestire il lessema in modo particolare.

```
bool          { return BOOL; }
if           { return KEY_W; }
```

```

{NUMBER}      { return INTEGER; }
{LETTER}      { return IDENTIFIER; }
{WHITESPACE}   { }

```

Nell'esempio soprastante vediamo:

- il pattern `bool` che riconosce un booleano e il codice `{ return BOOL; }` restituisce il token `BOOL` quando viene riconosciuto il pattern (costante definita nel literal block).
- il pattern `if` che riconosce un if e il codice `{ return KEY_W; }` restituisce il token `KEY_W` quando viene riconosciuto il pattern (costante definita nel literal block).
- Pattern `NUMBER` riconosce un numero intero, il codice `{ return INTEGER; }` restituisce il token `INTEGER`.
- Pattern `LETTER` riconosce una stringa, il codice `{ return IDENTIFIER; }` restituisce il token `IDENTIFIER`.
- Infine, il pattern `WHITESPACE` riconosce uno o più spazi, tabulazioni o nuove righe, il codice associato `{ }` indica di ignorare quel tipo di lessema senza restituire alcun token.

Se volessi conoscere il lessema che è stato identificato dal lexer, esso viene contenuto nelle variabili globali `yytext` (puntatore al primo carattere) e la sua lunghezza tramite la variabile `yyleng`.

Il pattern deve essere specificato ad inizio riga, il codice deve iniziare nella stessa riga del pattern, è possibile andare a capo nel codice se lo si racchiude in un blocco: {codice}. E' possibile andare a capo con pattern disgiuntivi usando | al posto del codice, **l'ordine delle regole ne stabilisce la priorità**.

Come specificare i pattern in Flex:

pattern	significato
<code>c</code>	carattere non speciale sta per se stesso
<code>\c</code>	carattere di escape (per i caratteri speciali)
<code>(pattern)</code>	parentesi (per specificare precedenze)
<code>pattern₁pattern₂</code>	concatenazione
<code>pattern₁ pattern₂</code>	alternanza
<code>pattern*</code>	iterazione di Kleene (zero o più occorrenze)
<code>pattern+</code>	iterazione positiva (una o più occorrenze)
<code>pattern?</code>	opzionalità (zero o una occorrenza)
<code>pattern{m,M}</code>	iterazione limitata
<code>.</code>	qualsiasi carattere <i>singolo tranne newline</i>
<code>[chars]</code>	classe di caratteri (match singolo)
<code>[^chars]</code>	complemento di classe di caratteri
<code>"string"</code>	match letterale di <i>string</i>
<code>{name}</code>	uso di pattern tramite nome
<code>pattern₁/pattern₂</code>	trailing context: <i>pattern₁</i> solo se seguito da <i>pattern₂</i>
<code>^pattern</code>	start-of-line context (se primo carattere del pattern)
<code>pattern\$</code>	end-of-line context (se ultimo carattere del pattern)

3- Sezione del codice utente

La sezione del codice utente inizia dopo il secondo marker `%%`, può contenere codice utente arbitrario, inserito verbatim dopo la definizione `yylex`.

Tipicamente vengono definite delle funzioni ausiliarie precedentemente dichiarate nella sezione delle definizioni e la funzione **main**.

```
int main() {
    int token;
    while(1) {
        token = yylex();
        if (token == 0)
            break;
        if (token == ERROR)
            exit(1);
    }
    return 0;
}
```

03-analisi-sintattica

- [Parser](#)
 - [Context-Free Grammar e Derivazioni](#)
 - [Top-Down Parsing - LL\(1\)](#)
 - [Ricorsione a sinistra](#)
 - [Backtracking](#)
 - [Parsing Predittivo](#)
 - [Trasformazione di una grammatica non LL\(1\) in una LL\(1\)](#)
 - [Bottom-Up Parsing - LR\(1\)](#)
 - [Shift-Reduce](#)
 - [LR\(1\) vs LL\(1\)](#)
 - [Bison](#)
 - [Sezione delle definizioni](#)
 - [Literal Block](#)
 - [Union declaration](#)
 - [Start declaration](#)
 - [Token declaration](#)
 - [Type declaration](#)
 - [Sezione delle regole](#)
 - [Sezione codice utente](#)

Parser

I compiti principali del parser, sono:

- **Controlla la sequenza di parole e le loro parti del discorso per verificarne la correttezza grammaticale** - riceve dallo scanner una sequenza di token con informazioni sulla loro categoria grammaticale e controlla che siano organizzate in una struttura grammaticalmente valida secondo le regole del linguaggio di programmazione.
- **Determinare se l'input è sintatticamente corretto** - verifica che il codice sorgente rispetti la sintassi prevista dal linguaggio
- **Guidare il controllo a livelli più profondi rispetto alla sintassi** - non si limita alla sola sintassi, ma può anche preparare il terreno per ulteriori controlli semantici, come verifiche di tipo o coerenza logica.
- **Costruire una rappresentazione intermedia del codice (IR)** - crea una struttura dati che rappresenta in modo strutturato il codice sorgente, spesso sotto forma di un albero

sintattico astratto

Quando analizziamo un programma in un linguaggio, cerchiamo di trovare una **derivazione**, cioè una sequenza di passi che permetta di costruire quella frase partendo dalle regole del linguaggio.

Per poter analizzare la struttura di un programma, abbiamo bisogno di una **grammatica formale G** che descriva le regole sintattiche del linguaggio, questo modello matematico definisce quali combinazioni di simboli sono valide. Di solito, per descrivere le grammatiche dei linguaggi di programmazione si usano quelle di tipo **Context-Free** (CFG - grammatiche libere dal contesto), poiché sono sufficientemente potenti per rappresentare la maggior parte delle strutture linguistiche.

Una volta definita la grammatica G, abbiamo un linguaggio formale $L(G)$ che include tutte le frasi che possono essere generate da G, dobbiamo quindi avere un **algoritmo** che ci permetta di verificare se una determinata sequenza di token appartiene a $L(G)$.

Context-Free Grammar e Derivazioni

Una grammatica è una quadrupla $G = (S, N, T, P)$ dove:

- S è lo start simbolo (simbolo iniziale da cui parte ogni derivazione - es. *Expr*)
- N è l'insieme dei simboli non terminali (variabili che rappresentano strutture più complesse che verranno riscritte/espanse secondo le regole)
- T è l'insieme dei simboli terminali (simboli finali concreti che compaiono nella stringa di output)
- P è l'insieme delle produzioni o delle regole riscritte (insieme delle regole che descrivono l'espansione/sostituzione dei simboli non terminali)

Perché non posso usare i linguaggi regolari e i DFA anche in questo caso? I linguaggi regolari sono quelli che possono essere descritti da espressioni regolari e possono essere riconosciuti da automi a stati finiti deterministici (DFA), codesti sono adatti a descrivere linguaggi con una struttura molto semplice, come sequenze ripetitive o pattern statici.

I linguaggi di programmazione richiedono spesso strutture più complesse, come l'annidamento di parentesi, le strutture condizionali e i blocchi di codice, questi schemi richiedono una capacità di tenere traccia di contesti diversi, come aprire e chiudere parentesi o iniziare e terminare blocchi di codice, i linguaggi regolari non hanno questa capacità.

La soluzione è dunque utilizzare **grammatiche libere da contesto**, che vengono gestite da automi più potenti, quelli a pila.

Una grammatica si definisce libera da contesto perché, ogni produzione ha la forma del tipo $A \rightarrow \alpha$ dove: A è un simbolo **non terminale** e α è una stringa di simboli terminali e/o non terminali, questa forma significa che ogni produzione può riscrivere il non terminale A indipendentemente dal **contesto** in cui A appare, può sempre essere sostituito da α

in qualunque punto del derivato.

In una **grammatica sensibile al contesto** (context-sensitive grammar), le regole di produzione possono invece dipendere dal contesto in cui il non terminale appare.

Una regola di produzione sensibile al contesto ha una forma più generale: $\alpha A \beta \rightarrow \alpha \gamma \beta$, dove A è un simbolo non terminale, α e β sono sequenze, anche vuote, di simboli terminali che rappresentano il **contesto** e γ è una sequenza di terminale e\o non, che sostituisce A in quel contesto specifico.

In questo caso, A può essere riscritto in γ solo se appare circondato dai simboli α e β , questo rende la grammatica **dipendente dal contesto**.

Vediamo un esempio:

Grammatica	Derivazione
$Expr \rightarrow Expr \quad Op \quad Expr$	0 $Expr$
number	1 $Expr \quad Op \quad Expr$
id	2 $\langle id, x \rangle \quad Op \quad Expr$
$Op \rightarrow +$	3 $\langle id, x \rangle \quad - \quad Expr$
-	4 $\langle id, x \rangle \quad - \quad Expr \quad Op \quad Expr$
*	5 $\langle id, x \rangle \quad - \quad \langle \dots, 2 \rangle \quad Op \quad Expr$
/	6 $\langle id, x \rangle \quad - \quad \langle \dots, 2 \rangle \quad * \quad Expr$
	7 $\langle id, x \rangle \quad - \quad \langle \dots, 2 \rangle \quad * \quad \langle id, y \rangle$

Analizzando la grammatica abbiamo che:

- Expr può essere espansa con:

- Expr Op Expr
- number
- id

- Op può essere espanso con:

- +
- -
- *
- /

Analizzando la derivazione, espandendo ad ogni step i simboli non terminali evidenziati in viola con i possibili simboli concessi, arriviamo ad avere la stringa $x - 2 * y$.

Notiamo a questo punto che la derivazione si può provare in due differenti modi:

- **Leftmost derivation** sostituendo il simbolo non terminale che si trova più a sinistra ad ogni step (esempio precedente).
- **Rightmost derivation** sostituendo il simbolo non terminale che si trova più a destra ad ogni step.

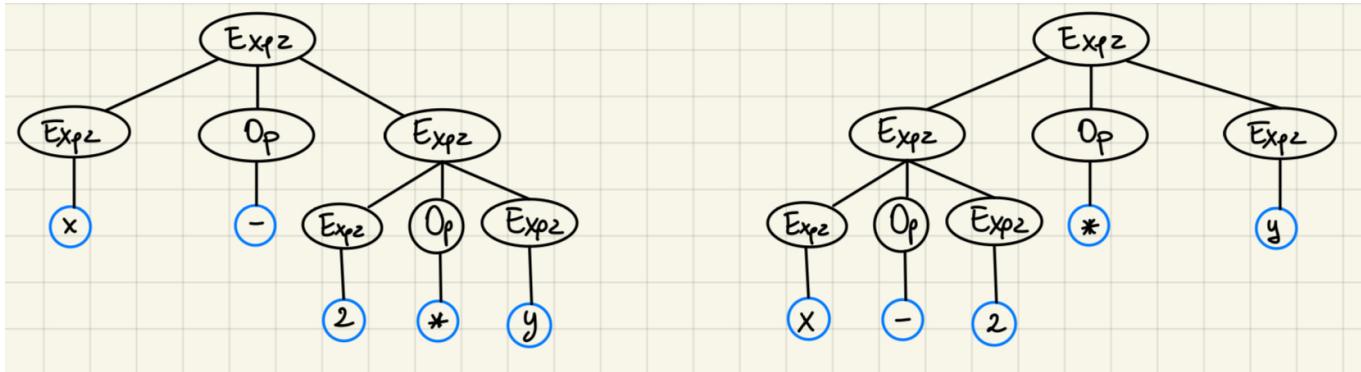
Derivazione sinistra

- 0 Exp_2
- 1 $\text{Exp}_2 \text{ Op } \text{Exp}_2$
- 2 $\langle \text{id}, x \rangle \text{ Op } \text{Exp}_2$
- 3 $\langle \text{id}, x \rangle - \text{Exp}_2$
- 4 $\langle \text{id}, x \rangle - \text{Exp}_2 \text{ Op } \text{Exp}_2$
- 5 $\langle \text{id}, x \rangle - \langle \text{num}, 2 \rangle \text{ Op } \text{Exp}_2$
- 6 $\langle \text{id}, x \rangle - \langle \text{num}, 2 \rangle * \text{Exp}_2$
- 7 $\langle \text{id}, x \rangle - \langle \text{num}, 2 \rangle * \langle \text{id}, y \rangle$

Derivazione destra

- 0 Exp_2
- 1 $\text{Exp}_2 \text{ Op } \text{Exp}_2$
- 2 $\text{Exp}_2 \text{ Op } \langle \text{id}, y \rangle$
- 3 $\text{Exp}_2 * \langle \text{id}, y \rangle$
- 4 $\text{Exp}_2 \text{ Op } \text{Exp}_2 * \langle \text{id}, y \rangle$
- 5 $\text{Exp}_2 \text{ Op } \langle \text{num}, 2 \rangle * \langle \text{id}, y \rangle$
- 6 $\text{Exp}_2 - \langle \text{num}, 2 \rangle * \langle \text{id}, y \rangle$
- 7 $\langle \text{id}, x \rangle - \langle \text{num}, 2 \rangle * \langle \text{id}, y \rangle$

Il risultato è lo stesso, ma producono differenti alberi di parsing:



L'ambiguità che ci suggerisce l'albero è: $x - (2 * y)$ o $(x - 2) * y$?

La grammatica ha un problema, non abbiamo nessuna informazione sulla precedenza degli

operatori o sull'ordine di valutazione, bisogna riscriverla correttamente.

Grammatica

```

Goal → Expr
Expr → Expr + Term } 3
| Expr - Term
| Term
Term → Term * Factor } 2
| Term / Factor
| Factor
Factor → (Expr) } 1
| number
| id
  
```

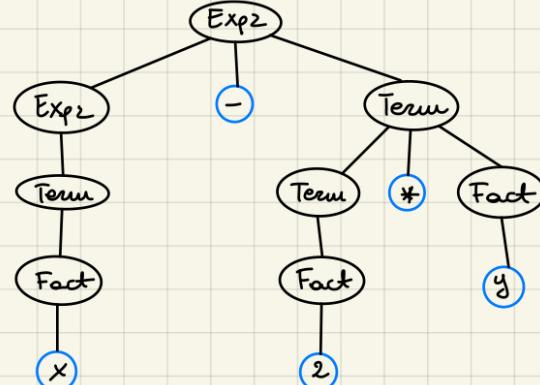
In matematica:

- 1) Parentesi
- 2) * e /
- 3) + e -

Derivazione destra

```

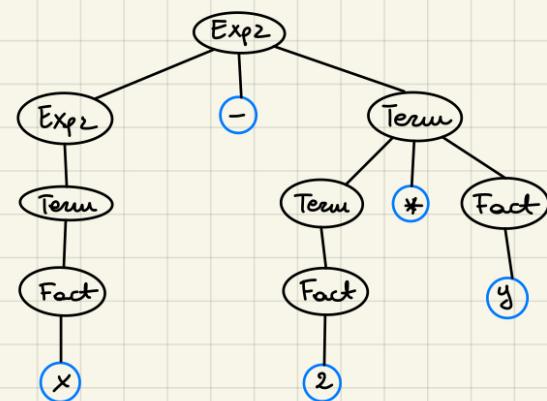
0 Goal
1 Expr
2 Expr - Term
3 Expr - Term * Fact
4 Expr - Term * < id, y >
5 Expr - Factor * < id, y >
6 Expr - < number, 2 > * < id, y >
7 Term - < number, 2 > * < id, y >
8 Factor - < number, 2 > * < id, y >
9 < id, x > - < number, 2 > * < id, y >
  
```



Derivazione sinistra

```

0 Goal
1 Expr
2 Expr - Term
3 Term - Term
4 Fact - Term
5 < id, x > - Term
6 < id, x > - Term * Fact
7 < id, x > - Fact * Fact
8 < id, x > - < number, 2 > * Fact
9 < id, x > - < number, 2 > * < id, y >
  
```



Vediamo che gli alberi di parsing sono identici e la soluzione finale è $x - (2 * y)$.

Un ultimo problema di questa grammatica è che se abbiamo più di una possibilità di derivazione sia nel caso a sinistra che nel caso a destra per un singolo step, la grammatica è ambigua.

Ad esempio nello step 1 della prima [derivazione](#), quello può essere espanso anche come `Expr Op Expr`, finendo in un loop infinito di espansione, NON lo vogliamo (soluzione nel prossimo [paragrafo](#)).

Top-Down Parsing - LL(1)

Il Top-down parser è una delle tecniche che viene usata per fare parsing, il suo funzionamento consiste nell'analisi dal nodo radice dell'albero sintattico e procede verso le foglie, cercando di far corrispondere la sequenza di input alle regole della grammatica.

LL(1):

- **Left-to-right:** il parser legge l'input da sinistra a destra
- **Leftmost derivation:** costruisce l'albero sintattico espandendo sempre il simbolo non terminale più a sinistra
- **(1): lookahead**, cioè il parser guarda solo un simbolo (in questo caso) in anticipo per decidere quale regola applicare.

Quando si fa una scelta sbagliata nel fare match con la grammatica, il parser deve "backtrackare", cioè annullare la scelta della regola di produzione, e provare con un'altra regola per lo stesso simbolo.

Il **backtracking** implica tornare al punto in cui il parser ha fatto una scelta, scegliere una produzione diversa e riprendere l'analisi da lì, questo processo è inefficiente e può portare a tempi di parsing esponenziali nel caso peggiore.

Alcune grammatiche sono **senza backtracking**, il che significa che il parser può decidere quale regola di produzione usare basandosi solo sul simbolo corrente di input e (al massimo) su un simbolo di lookahead, eliminando quindi la necessità di backtracking.

Le grammatiche LL(1) ne sono un esempio, perché strutturate in modo che ogni simbolo non terminale abbia una sola produzione possibile per ciascun token di lookahead. Questo le rende compatibili con i **parser predittivi**, che usano il lookahead per "predire" quale produzione applicare senza bisogno di fare tentativi e backtracking, per assicurarsi che non ci sia, di solito viene eliminata la **ricorsione a sinistra** nella grammatica e si rimuovono ambiguità.

Come accennato precedentemente la grammatica vista prima ha due problematiche:

- è ricorsiva a sinistra;
- troppe possibilità di fare match con diversi non-terminali per un unico non-terminale, dovendo dunque fare backtracking più volte;

Ricorsione a sinistra

Una grammatica è considerata **ricorsiva a sinistra** se $\exists A \leftarrow \text{Non-Terminale}$ tale che \exists una derivazione $A \Rightarrow^+ A\alpha$, per qualche stringa $\alpha \in (\text{Non-Terminale} \cup \text{Terminale}^+)$
 Dunque per eliminarla, dobbiamo trasformare la ricorsione a sinistra come una ricorsione a destra riscrivendo la grammatica.
 Ad esempio, consideriamo il pezzo di grammatica:

```
Fee -> Fee α
|   β
//ricorsiva a sinistra
```

Possiamo riscriverla nel seguente modo:

```
Fee -> β Fie
Fie -> α Fie
|   ε
//ricorsiva a destra
```

Vediamo che è la stessa identica grammatica trasformata in una ricorsione destra e con l'aggiunta del riferimento ad una stringa vuota ε .

Aggiorniamo ora la nostra vecchia grammatica con ricorsione a sinistra:

```
Expr -> Term Expr'
Expr' -> + Term Expr'
|   - Term Expr'
|   ε
Term -> Factor Term'
Term' -> * Factor Term'
|   / Factor Term'
|   ε
```

Questa parte riscritta usa solamente la ricorsione a destra, quindi possiamo integrarla nella grammatica finale come segue:

```
Goal -> Expr
Expr -> Term Expr'
Expr' -> + Term Expr'
|   - Term Expr'
|   ε
Term -> Factor Term'
Term' -> * Factor Term'
```

```

    | / Factor Term'
    | ε
Factor -> (Expr)
    | number
    | id

```

Backtracking

Se sceglie la produzione sbagliata, un parser top-down potrebbe tornare indietro.

Un'alternativa è guardare avanti nell'input e usare il contesto per scegliere correttamente, questa metodologia del parser si chiama parsing predittivo.

Parsing Predittivo

L'idea di fondo è che avendo $A \rightarrow \alpha \mid \beta$ il parser dovrebbe essere in grado di scegliere tra α o β . Per fare questa scelta, il parser utilizza l'insieme **FIRST** di ogni espansione, cioè per una stringa α (che può essere composta da simboli terminali e non-terminali), l'insieme $\text{FIRST}(\alpha)$ è definito come l'insieme di tutti i token (simboli terminali) che possono comparire come **primo simbolo** in qualche stringa derivata da α .

$$x \in \text{FIRST}(\alpha) \text{ se } \alpha \Rightarrow^+ x\gamma$$

dove:

- x è un simbolo terminale
- γ è una sequenza qualsiasi di simboli (terminali e non)
- \Rightarrow^+ indica una derivazione in zero o più passaggi

Esempio

Supponiamo di avere le seguenti produzioni per un non-terminale A:

$$A \rightarrow aX \mid bY$$

dove:

- a e b sono simboli terminali.
- X e Y sono sequenze di simboli (terminali o non-terminali).

Per costruire l'insieme FIRST per ciascuna produzione:

1. $\text{FIRST}(aX)$, dato che la stringa aX inizia con a, abbiamo:

$$\text{FIRST}(aX) = \{a\}$$

2. $\text{FIRST}(bY)$, dato che la stringa bY inizia con b, abbiamo:

$$\text{FIRST}(bY) = \{b\}$$

Ora, l'insieme **FIRST(A)**, che rappresenta i simboli terminali che possono iniziare una stringa derivata da A, è dato dall'unione degli insiemi FIRST delle sue espansioni:

$$FIRST(A) = FIRST(aX) \cup FIRST(bY) = \{a, b\}$$

Attenzione, questo funziona solo se gli insiemi $FIRST(\alpha)$ e $FIRST(\beta)$ sono disgiuntivi:

$$FIRST(\alpha) \cap FIRST(\beta) = \emptyset$$

Le **ϵ -produzioni** (produzioni che permettono a un non-terminale di derivare la stringa vuota) complicano la definizione di una grammatica LL(1).

$$FIRST^+(A \rightarrow \alpha) = \begin{cases} FIRST(\alpha) \cup FOLLOW(A), & \text{se } \epsilon \in FIRST(\alpha) \\ FIRST(\alpha), & \text{altrimenti} \end{cases}$$

FOLLOW(A) è l'insieme dei simboli terminali che possono immediatamente seguire il simbolo non terminale A in una forma sentenziale.

Trasformazione di una grammatica non LL(1) in una LL(1)

Generalmente questa trasformazione non la si può fare, ma in alcuni casi sì.

Esiste una tecnica chiamata factoring della grammatica che riorganizza le produzioni per renderle più predittive.

Consideriamo una grammatica G con le produzioni:

$$A \rightarrow \alpha\beta_1 \quad \text{e} \quad A \rightarrow \alpha\beta_2$$

In questo caso, entrambe le produzioni di A iniziano con lo stesso prefisso α , questo significa che quando il parser vede un simbolo iniziale corrispondente a α , non può sapere quale delle due produzioni usare (se quella con β_1 o quella con β_2), poiché entrambe condividono α . Questa ambiguità si traduce in una violazione della proprietà LL(1), poiché il parser non può decidere in modo univoco basandosi su un solo simbolo di lookahead, di conseguenza non stiamo rispettando la regola:

$$FIRST^+(A \rightarrow \alpha\beta_1) \cap FIRST^+(A \rightarrow \alpha\beta_2) \neq \emptyset$$

Per risolvere questo problema, possiamo **estrarre il prefisso comune** α e creare una nuova produzione, rendendo la grammatica più adatta al parsing predittivo, questa tecnica si chiama **factoring**.

Riscriviamo G:

- Creiamo un nuovo non terminale A' per gestire le varianti che seguono il prefisso comunque
- Riscriviamo le produzioni di A in modo che utilizzino A' dopo α .

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \quad \text{e} \quad A' \rightarrow \beta_2$$

Bottom-Up Parsing - LR(1)

Rappresentano un approccio opposto rispetto ai parser top-down, mentre questi costruiscono l'albero di derivazione dall'alto verso il basso, i parser **bottom-up** partono dalle **foglie** (i simboli terminali) e cercano di risalire all'albero fino al **nodo radice**.

1. Il parser inizia con la stringa di input, questi vengono "consumati" uno per uno, cercando di formare sequenze che possano corrispondere alle produzioni della grammatica.
2. Man mano che l'input viene consumato, il parser aggiorna uno **stato interno** che rappresenta tutte le possibili produzioni o configurazioni della grammatica che potrebbero essere state applicate finora.
3. Quando il parser riconosce una sequenza di simboli che corrisponde al lato destro di una produzione, applica una **riduzione**. Questo significa che sostituisce quella sequenza con il non-terminale corrispondente, riducendo la parte dell'albero che ha appena "costruito".
4. Quando il parser inizia, è in uno stato che è valido per il primo simbolo dell'input, e da lì costruisce gradualmente l'albero.

Il parser bottom-up costruisce una derivazione di tipo rightmost.

Una **riduzione** in un parser bottom-up è il processo in cui una sequenza di simboli viene sostituita con il simbolo non-terminale sul lato sinistro della stessa produzione. Ad esempio, se abbiamo la produzione: $A \rightarrow \beta$ e il parser riconosce che una parte dell'input corrente corrisponde a β , sostituirà quella parte con A.

Grammatica	Input String	Derivazione
Goal	a b c d e	0 a b c d e
A		1 a A b c d e
	b	2 a A d e
B	d	3 a A B e
6		4 Goal

Nell'esempio sopra abbiamo sviluppato ridotto la stringa "abbcde" con la grammatica a fianco, i caratteri evidenziati in viola sono quelli che sono stati ridotti nel passo successivo. E' un rightmost reverse in quanto se partiamo nella derivazione dal goal e saliamo al passo 0 vediamo che i caratteri derivati sono quelli più a destra.

Nella gestione delle espressioni aritmetiche, si preferiscono le grammatiche ricorsive a sinistra, perché la valutazione segue solitamente una direzione **da sinistra a destra**, per facilitare questa valutazione in modo coerente, utilizziamo grammatiche ricorsive a sinistra, che riflettono meglio questa **associatività a sinistra** comune nelle operazioni come somma e sottrazione (per i bottom-up la ricorsione a sinistra non è un problema).

Shift-Reduce

Per implementare un parser bottom-up si utilizza il paradigma di **shift-reduce**, basandosi su una macchia a stati con uno stack. Questo modello ci sono quattro azioni principali:

1. **Shift**: si sposta il prossimo simbolo dell'input sullo stack (Push).
2. **Reduce**: si riconosce un **handle** (una sequenza che corrisponde al lato destro di una produzione) in cima allo stack e si riduce sostituendolo con il simbolo non-terminale relativo alla produzione (Pop + Push).
3. **Accept**: Il parser ha analizzato tutto l'input e la stringa è accettata con successo.
4. **Error**: Si riscontra un errore e il parser attiva una routine di gestione o recupero.

Il parser deve determinare dinamicamente quando applicare Shift e quando applicare Reduce: il parser continua a shiftare simboli dallo stack fino a formare un handle, quando un handle è in cima allo stack, è possibile applicare la Reduce.

Il parser riconosce un handle utilizzando una **tabella di parsing** che specifica, per ogni stato e simbolo, se deve eseguire un'operazione di Shift o di Reduce, questa si basa sull'analisi della grammatica, determinando in quali condizioni è possibile applicare una certa produzione.

Vediamo l'esempio che abbiamo usato anche per il top-down, l'espressione $x - 2 * y$:

<u>Grammatica</u>		<u>Riduzioni : $x - 2 * y$</u>	<u>Stack</u>	<u>Input</u>	<u>Azione</u>
0	Goal \rightarrow Expr				
1	Expr \rightarrow Expr + Term	0		id - mmm * id	shift
2	Expr - Term	1 id		- mmm * id	reduce 3
3	Term	2 Factor		- mmm * id	reduce 6
4	Term \rightarrow Term * Factor	3 Term		- mmm * id	reduce 3
5	Term / Factor	4 Expr		mmm * id	shift
6	Factor	5 Expr -		mmm * id	shift
7	Factor \rightarrow number	6 Expr - mmm		* id	reduce 7
8	id	7 Expr - Factor		* id	reduce 6
9	(Expr)	8 Expr - Term		* id	shift
		9 Expr - Term *		id	shift
		10 Expr - Term * id		reduce 8	
		11 Expr - Term * Factor		reduce 4	
		12 Expr - Term		reduce 2	
		13 Expr		reduce 0	
		14 Goal			ACCEPT

Ma come fare a riconoscere esattamente quando abbiamo un handle in cima allo stack? Invece di provare tutte le possibili derivazioni, possiamo fare riferimento a un insieme di informazioni pre-calcolate:

- **Contesto a Sinistra:** gli stati del parser contengono informazioni sul contesto a sinistra, cioè cosa è stato visto fino a quel punto nell'input, questo è memorizzato nella **forma sentenziale parziale** presente nello stack del parser.
- **Stati del Parser:** il parser mantiene informazioni aggiuntive attraverso uno stato, che rappresenta il contesto corrente del parsing e aiuta a determinare quando siamo pronti a effettuare una riduzione, questi stati vengono calcolati in anticipo attraverso un'**analisi di raggiungibilità** sulla grammatica, per capire quali sequenze di simboli possono portare a certi stati in un automa.
- **Lookahead di un Simbolo:** il parser guarda anche un simbolo in avanti (lookahead), cioè il simbolo successivo nell'input, che aiuta a determinare se il simbolo attuale può effettivamente essere l'inizio di un handle.

Per implementare questo riconoscimento senza ambiguità, il parser utilizza un automa a stati finiti (DFA) che è stato progettato per riconoscere gli handle.

LR(1) vs LL(1)

Il **contesto a sinistra** permette al parser LR(1) (Left-to-right; Rightmost derivation reverse; lookahead) di distinguere situazioni che un parser LL(1) non sarebbe in grado di distinguere

solo con il lookahead. Per esempio, in una grammatica ambigua o con ricorsione sinistra, il parser LR(1) può comunque prendere la decisione corretta basandosi sul contesto a sinistra, mentre il parser LL(1) non potrebbe.

Bison

Bison è uno strumento utilizzato per generare analizzatori sintattici (o parser) basati su grammatiche specificate dall'utente, è comunemente impiegato in combinazione con Flex, che si occupa dell'analisi lessicale.

- Bison riceve in input una descrizione grammaticale scritta in un linguaggio simile a una grammatica di contesto, che specifica la sintassi del linguaggio da analizzare.
- Dal file di definizione della grammatica, Bison genera automaticamente un parser scritto in C o C++ che segue il modello **LR(1)**.
- Il parser prodotto è in grado di riconoscere strutture sintattiche definite dalla grammatica e di eseguire azioni associate, come costruire alberi sintattici, calcolare espressioni, o interpretare comandi.

La struttura del file sorgente di Bison ha 3 sezioni differenti:

1. Sezione delle definizioni
2. Sezione delle regole
3. Sezione del codice utente

Sezione delle definizioni

Cosa può contenere questa fase?

- Literal Block
- Union declaration
- Start declaration
- Token declaration
- Type declaration

Literal Block

Un literal block è una sezione di codice C racchiusa tra `%{` e `%}`, che appare all'inizio del file di input Bison, questo viene copiato letteralmente nella parte iniziale del file C generato da Bison, senza alcuna modifica.

Tipicamente, un literal block contiene:

- Inclusioni di file header (ad esempio, `#include <stdio.h>`) necessari per il codice generato,
- Dichiarazioni di variabili che saranno usate nelle regole grammaticali,
- Dichiarazioni di funzioni che saranno invocate all'interno delle regole,
- Definizioni di funzioni inline per calcoli o operazioni ausiliarie.

Union declaration

La dichiarazione `%union` permette di definire un insieme di attributi associabili ai simboli della grammatica (sia terminali che non-terminali), si utilizza per specificare quali tipi di dati possono essere contenuti nei simboli, consentendo al parser di gestire più tipi di valori (es. interi, stringhe, strutture) durante l'analisi sintattica.

```
%union {
    tipo_1 nome_1;
    ...
    tipo_n nome_n;
};
```

quindi, il blocco `%union` definisce un tipo union in C, in cui ogni campo (es. `tipo1 nome1`) rappresenta un tipo di dato specifico che un simbolo può assumere, per i simboli della grammatica che richiedono un attributo (es. valori di numeri o stringhe), è possibile specificare quale membro della union utilizzare.

Dopo averla definita, si possono associare i campi della union ai simboli tramite direttive `%type` (per non-terminali) e `%token` (per terminali), consentendo al parser di accedere al dato corretto per ciascun simbolo durante la derivazione.

Start declaration

La dichiarazione `%start` permette di specificare esplicitamente il simbolo di inizio (**start symbol**) della grammatica.

```
%start non-terminale
```

Il simbolo indicato da `%start` sarà il punto di partenza per l'analisi sintattica: il parser cercherà di derivare l'intero input a partire da questo simbolo non-terminal.

Se non si specifica un simbolo di start con `%start`, Bison utilizza automaticamente il non-terminal che si trova sul lato sinistro della prima produzione definita nella sezione delle regole.

Token declaration

La dichiarazione `%token` in Bison serve a definire un simbolo come terminale e, optionalmente, associarlo a un attributo specificato nella union, inoltre, se fornito, viene specificato anche il lessema.

```
%token [<nome>] terminale
```

oppure

```
%token [<nome>] terminale "TOKEN"
```

Scopo:

- **Dichiarare un terminale:** un simbolo che appare nel linguaggio di input (di solito definito dal lexer).
- **Attributo:** se viene fornito un nome, questo indica il campo nella union che memorizza l'attributo associato al terminale, il lexer assegnerà il valore dell'attributo a `yyval.nome`.
- **Lessema:** se è fornito un valore come stringa (es. `"TOKEN"`), viene definito il lessema, cioè la sequenza di caratteri che rappresenta il simbolo nel codice sorgente.

```
%union {  
    double dval;  
    /* ... */  
};  
  
%token <dval> NUMBER
```

Viene dichiarato una union, che può contenere vari tipi di dati, in questo caso, c'è solo un campo, `dval`, che è di tipo `double`.

La direttiva `%token` dichiara che `NUMBER` è un terminale, ciò significa che il valore associato al terminale `NUMBER` sarà di tipo `double`, e verrà memorizzato in `yyval.dval`.

Le dichiarazioni `%left`, `%right`, e `%nonassoc` servono a specificare l'associatività e la precedenza di un terminale, queste aiutano a risolvere ambiguità quando ci sono operatori con la stessa precedenza.

```
%left [<nome>] terminale  
%right [<nome>] terminale  
%nonassoc [<nome>] terminale
```

Type declaration

La dichiarazione `%type` serve per specificare il tipo di dato associato a un simbolo non-terminale della grammatica, utilizzando uno dei membri definiti nella union.

```
%type <nome> non-terminale
```

Sezione delle regole

La sezione delle regole in un file Bison contiene le regole di produzione della grammatica e definisce la funzione `yyparse()`, che è il cuore dell'analizzatore sintattico, questa ha il compito di analizzare la sequenza di token generata dall'analizzatore lessicale `yylex()`, riconoscere le strutture sintattiche del linguaggio e applicare azioni associate alle produzioni grammaticali.

Funzione di `yyparse()`:

- `yyparse()` chiama ripetutamente la funzione `yylex()`, che fornisce i token uno alla volta dall'input.
- `yyparse()` esegue un'analisi bottom-up, costruendo gradualmente la struttura sintattica completa, partendo dai singoli token fino ad arrivare al simbolo di inizio della grammatica.
- Spesso, il valore associato al simbolo lhs (lato sinistro della produzione) viene calcolato usando i valori dei simboli nel rhs (lato destro della produzione), ad esempio sommando due numeri o concatenando stringhe.

```
nonterm: rhs_1 codice_1
        | rhs_2 codice_2
        |
        | ...
        | rhs_n codice_n
;
```

formato di una regola.

Esempio:

```
expr:
      term
    | expr `+` term { $$ = $1 + $3; }
    | expr `-` term { $$ = $1 - $3; }
;
```

Regola di default: se non viene scritto codice esplicito, Bison assume che il valore di `$$` sia uguale al primo simbolo a destra (in questo caso `$$ = &1`).

Grammatiche accettate da Bison:

Normalmente, le grammatiche devono essere non ambigue anche se le può accettare

specificando precedenza e associatività degli operatori:

- **associatività**: `%left` , `%right` , `%noassoc`
- **precedenza**: data dall'ordine (inverso) delle associatività
- **Esempio**:
`%left '+' '-'`
`%left '*' '/'`
`%right POW`
- **precedenza regole**: quella del simbolo terminale più a destra

Sezione codice utente

Inizia dopo il secondo marker `%%` può contenere codice utente arbitrario, inserito *verbatim* dopo la definizione di `yyparse`, tipicamente:

- definizione delle funzioni ausiliarie precedentemente dichiarate (nella sezione delle definizioni)
- la funzione main (non usuale)
Best practice non mettere le definizioni delle funzioni, usare piuttosto un'altra unità di traduzione.

```
/*...*/
%%
/*...*/
%%

int main() {
    return yyparse();
}
```

04-analisi-semantica

- [Analisi semantica](#)
 - [Grammatiche attribuite](#)
 - [Tipologie di Attributi](#)
 - [Metodi per calcolare gli attributi](#)
 - [Circolarità nelle grammatiche attribuite](#)
 - [Dalle grammatiche attribuite ai metodi ad-hoc](#)
 - [Problemi Grammatiche Attribuite](#)
 - [Traduzione Sintattica Diretta Ad-Hoc](#)
 - [Limitazioni](#)
 - [Come adattare la traduzione sintattica ad-hoc in un parser LR\(1\)?](#)
 - [Alternativa: Passeggiata sull'Albero Sintattico Astratto \(AST\)](#)

Analisi semantica

Per generare codice abbiamo bisogno di capire il suo significato, quindi il compilatore ha bisogno di porsi tante domande, ad esempio:

- "x" è uno scalare, un array o una funzione? "x" è dichiarata?
 - Ci sono nomi che non sono stati dichiarati? Magari dichiarati e non usati? etc.
- Queste domande fanno parte dell'analisi context-sensitive, cioè che hanno bisogno di un contesto per avere senso.
- Come possiamo dunque rispondere a queste domande?
- Usando metodi formali
 - Grammatiche context-sensitive: consentono di definire regole in cui la produzione di un simbolo dipende dai simboli circostanti (difficili da implementare e poco efficienti).
 - Grammatiche attribuite: ai simboli vengono associati attributi (valori o informazioni aggiuntive) e regole semantiche per calcolare questi attributi (molto utili).
 - Tecniche ad-hoc
 - Tabelle dei simboli: strutture dati utilizzate per tenere traccia delle informazioni sulle entità del programma, come variabili, funzioni, classi e scope.
 - Codice ad-hoc (action routines): frammenti di codice specifici (o funzioni) eseguiti durante la compilazione per risolvere problemi contestuali.

Nel parsing (analisi sintattica) vincono le grammatiche libere dal contesto, mentre nell'analisi semantica (context-sensitive) le tecniche ad-hoc dominano la pratica.

Grammatiche attribuite

Le grammatiche attribuite combinano la struttura di una grammatica sintattica con regole semantiche per arricchire i simboli del linguaggio con **attributi**, che rappresentano informazioni aggiuntive come il tipo di una variabile o il valore di un'espressione, e le regole semantiche permettono di **calcolare e verificare** questi attributi.

PROBLEMI:

- calcoli non locali: se un attributo in un punto del codice dipende da informazioni lontane (come l'assegnazione di una variabile definita in un altro blocco), risulta difficile gestirlo in una grammatica attribuita.
- informazioni centralizzate: per molte analisi, serve una tabella centrale delle informazioni (tabella dei simboli) per tenere traccia delle variabili, funzioni e tipi le grammatiche attribuite non sono pensate per gestire direttamente strutture centralizzate, risultando quindi poco pratiche.

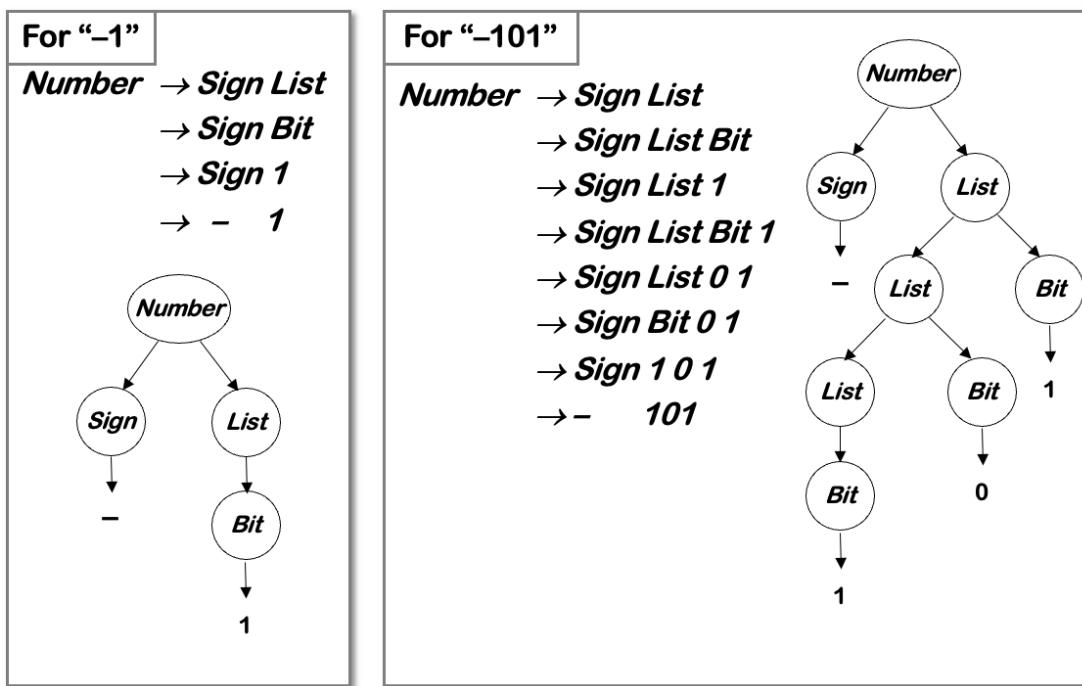
Arriviamo dunque a prediligere le **Tecniche ad-hoc**, più flessibili e pratiche per problemi complessi, per ora però capiamo come funzionano queste.

Gli elementi principali sono:

- **Simboli e Attributi**: ogni simbolo della grammatica (sia terminale che non terminale) è arricchito con un insieme di attributi, quest'ultimi sono valori associati ai simboli, che possono contenere informazioni semantiche come il tipo, il valore o altre proprietà rilevanti, esistono due tipi principali di attributi:
 1. Attributi sintetizzati: calcolati dalle regole semantiche sulla base dei figli del simbolo nell'albero sintattico
 2. Attributi ereditati: derivano da informazioni che provengono dai genitori o dai fratelli del simbolo nell'albero.
- **Regole di attribuzione**: per ogni produzione della grammatica vengono definite regole che descrivono come calcolare gli attributi. Le regole sono funzionali, cioè determinano univocamente il valore di ogni attributo basandosi sugli attributi disponibili.
- **Funzioni semantiche**: le regole di attribuzione spesso usano funzioni definite dall'utente, che calcolano gli attributi in base a valori disponibili o logica specifica.
Vediamo un esempio, la seguente grammatica descrive i numeri binari con segno:

1	<i>Number</i>	\rightarrow	<i>Sign List</i>
2	<i>Sign</i>	\rightarrow	+
3			-
4	<i>List</i>	\rightarrow	<i>List Bit</i>
5			<i>Bit</i>
6	<i>Bit</i>	\rightarrow	0
7			1

Formiamo due AST per due input differenti (-1 e -101):



Vogliamo dunque calcolare il valore decimale, dobbiamo quindi aggiungere delle regole per farlo:

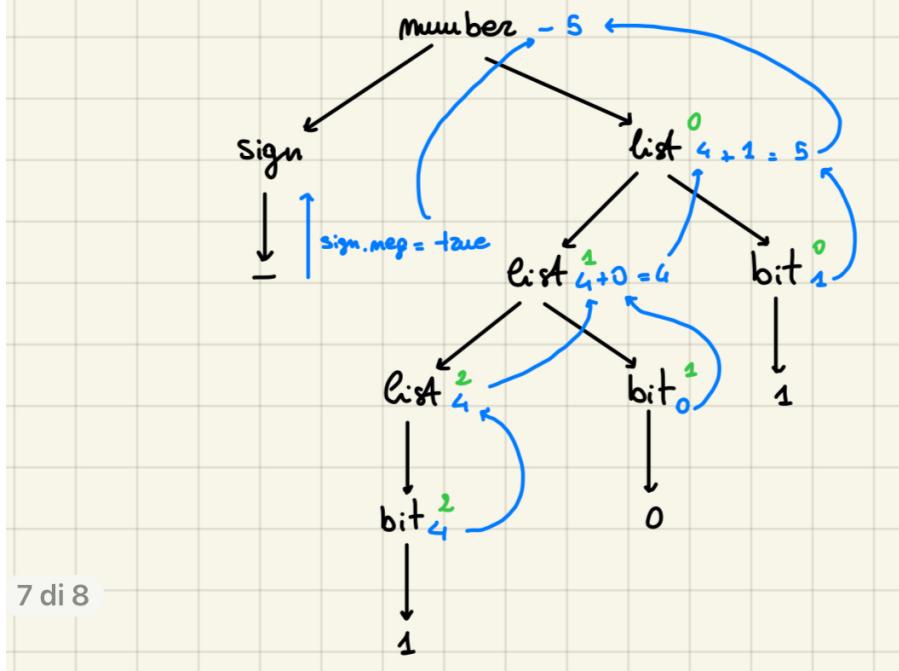
Productions		Attribution Rules	
Symbol	Attribute s		
Number	\rightarrow Sign List	$List.pos \leftarrow 0$ $if\ Sign.neg$ $then\ Number.val \leftarrow -List.val$ $else\ Number.val \leftarrow List.val$	Number val
Sign	\rightarrow + -	$Sign.neg \leftarrow false$ $Sign.neg \leftarrow true$	Sign neg
List ₀	\rightarrow List ₁ Bit	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$	List pos, val
	Bit	$Bit.pos \leftarrow List.pos$ $List.val \leftarrow Bit.val$	Bit pos, val
Bit	\rightarrow 0 1	$Bit.val \leftarrow 0$ $Bit.val \leftarrow 2^{Bit.pos}$	

Possiamo dedurre che, dall'immagine sopra, abbiamo degli attributi per ogni tipo di produzione, ad esempio banalmente per il "Bit" e "List" abbiamo due attributi necessari: posizione e valore. Andiamo con ordine vediamo come valutare un AST con gli attributi:

Input String : -101 → value

position

Solution: $1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 = 1 + 0 + 4 = -5$



Spiegazione

In input abbiamo la stringa -101, che dovrà essere trasformata in decimale tramite una grammatica con attributi.

Partiamo a sinistra dove abbiamo il segno -, questo porterà la produzione di Sign con l'attributo neg a true e lo "assegna" a number.

Ora spostiamoci a destra, il primo List che incontriamo avrà l'attributo position uguale a 0 il secondo list attributo position uguale ad 1 e così via anche per i bit, il primo bit che incontriamo avrà position = 0 etc.

Ora risaliamo l'albero dalle foglie quindi iniziamo dal primo bit più a sinistra che è uguale ad 1, assegnamo quindi alla sua produzione "bit" il valore 4 dato dal calcolo di $2^{\text{bit}.pos}$, riporto ora il valore alla produzione superiore "list", nel mentre guardo anche il bit 0, che ha come valore 0 dato dalla condizione che se $\text{bit}.val = 0$ allora 0.

Arriviamo ora alla produzione "List" con posizione 1, che valore avrà? Devo semplicemente riportare il valore dei due figli sommato, dunque abbiamo la list in posizione 2 con valore 4 e il bit in posizione 1 con valore 0 sommando troviamo che list in posizione 1 vale 4, questo meccanismo è applicato ricorsivamente a tutte le produzioni dell'AST trovando quindi in automatico che number avrà valore finale -5.

Tipologie di Attributi

- **Attributi sintetizzati:** sono calcolati a partire dagli attributi dei figli e da eventuali costanti o informazioni esterne (dal basso verso l'alto). Le caratteristiche principali sono:
 - Gli attributi sintetizzati di un nodo dipendono dai valori dei nodi figli (o terminali associati) e possono includere costanti.
 - Se una grammatica utilizza solo attributi sintetizzati, si dice **S-attributed** (compatibile con il parsing LR, bottom-up)
 - Facile da implementare durante il parsing, il parsing LR costruisce l'albero sintattico dal basso verso l'alto, calcolare gli attributi sintetizzati durante il parsing è diretto e non richiede trasformazioni particolari.
- **Attributi ereditati:** sono calcolati usando valori provenienti dal genitore del nodo nell'albero sintattico, informazioni provenienti dai fratelli e tramite costanti o informazione esterne. Le caratteristiche principali sono:
 - Gli attributi ereditati sono particolarmente utili per rappresentare il **contesto** del nodo corrente.
 - Gli attributi ereditati spesso richiedono che le informazioni siano calcolate prima che il nodo sia visitato (non è facilmente compatibile con il parsing LR).
 - Spesso, un'analisi basata su attributi ereditati può essere riscritta per evitare questi attributi, trasformandoli in sintetizzati o utilizzando strutture di supporto.
 - Nonostante le difficoltà pratiche, gli attributi ereditati sono ritenuti più "naturali" per esprimere certe relazioni semantiche, poiché modellano esplicitamente il contesto.

Metodi per calcolare gli attributi

Metodi dinamici basati su dipendenze: questi metodi si basano sulle **dipendenze** tra gli attributi, rappresentate in un grafo, per determinare l'ordine di calcolo degli attributi in tempo reale.

Procedura:

1. Costruzione dell'albero sintattico (AST)
2. Costruzione del grafo delle dipendenze: ogni attributo è rappresentato come un nodo del grafo, le dipendenze vengono rappresentate come archi diretti
3. Ordinamento topologico: si effettua un ordinamento topologico del grafo che garantisce che gli attributi siano calcolati nell'ordine corretto.
4. Calcolo degli attributi: vengono valutati in base all'ordine derivato dall'ordinamento topologico.

Vantaggi: funziona con qualsiasi grammatica attribuita.

Svantaggi: overhead, la costruzione e l'elaborazione del grafo delle dipendenze può essere costoso in termini di tempo e memoria.

Metodi basati su regole (Treewalk): l'ordine di calcolo degli attributi è determinato **in anticipo**, durante la generazione del compilatore, analizzando le regole semantiche associate alla grammatica.

Procedure:

1. Analisi delle regole: durante la fase di generazione del compilatore, le regole di attribuzione sono analizzate per identificare le dipendenze tra gli attributi.
 2. Si determina un **ordine fisso** per il calcolo degli attributi, basato su una strategia che garantisce che tutti gli attributi necessari per un calcolo siano disponibili al momento giusto.
 3. Durante l'elaborazione dell'albero sintattico, i nodi vengono visitati in un ordine specifico (determinato in precedenza) e gli attributi vengono calcolati di conseguenza.
- Vantaggi:** L'ordine statico elimina il bisogno di costruire grafi di dipendenze a runtime, una volta determinato l'ordine, il processo è diretto.
- Svantaggi:** Non tutte le grammatiche attribuite possono essere risolte facilmente con un ordine statico. Alcune richiedono trasformazioni o modifiche.

Metodi oblivious (Passes, Dataflow)

Questi metodi ignorano le regole semantiche dell'albero sintattico, e invece scelgono un ordine predeterminato (in fase di progettazione del compilatore) per valutare gli attributi.

Procedure:

1. Gli sviluppatori decidono un ordine fisso di calcolo degli attributi, questo è basato su ipotesi pratiche e non richiede l'analisi delle regole semantiche.

2. Gli attributi sono calcolati in **più passaggi** sull'albero sintattico, questo approccio può essere simile a un'analisi dataflow: si propagano le informazioni attraverso i nodi fino a quando tutti gli attributi non sono calcolati.

Vantaggi: Non richiede analisi sofisticate di dipendenze o costruzione di grafi, può gestire attributi complessi con un numero sufficiente di passaggi.

Svantaggi: L'approccio può richiedere più passaggi rispetto ai metodi dinamici o basati su regole, con un aumento dei tempi di esecuzione.

Il grafo delle dipendenze dev'essere aciclico.

Circolarità nelle grammatiche attribuite

Circolarità nelle grammatiche si riferisce alla presenza di **dipendenze circolari** tra gli attributi. Se le regole di valutazione creano dipendenze cicliche, ovvero un attributo dipende da un altro che, a sua volta, dipende dal primo, si ha una situazione di **circolarità**, che è problematica per la valutazione, soprattutto in un compilatore.

Esistono **grammatiche non circolari** che non presentano questa problematica. La più grande classe di grammatiche che non genera dipendenze circolari è rappresentata dalle **grammatiche fortemente non circolari (SNC)**. La buona notizia è che **testare se una grammatica è SNC** può essere fatto in **tempo polinomiale**, quindi è possibile farlo in modo efficiente.

Il testo fornisce un esempio di grammatica in cui tutti gli attributi sono **sintetizzati**, quindi si tratta di una **grammatica S-attribuita**

Un possibile miglioramento è il **tracciamento dei carichi** (cioè, la gestione delle variabili o valori già caricati) nelle produzioni, per evitare di caricare un valore più volte.

- È necessario introdurre dei **set Before e After** per ogni produzione, che permettano di tracciare quali valori sono stati già caricati e quali no. Questo richiede una gestione più complessa degli attributi, ma è utile per migliorare l'efficienza.
- L'aggiunta di questi set aumenta la **complessità** della grammatica, poiché per ogni produzione bisogna aggiungere delle **regole di copia** per propagare i valori tra i vari set.

Un ulteriore passo nell'evoluzione del modello è la gestione di un **set di registri finito**, cioè un numero limitato di variabili che possono essere utilizzate per tracciare i valori.

- Questo complica la produzione **Factor → Identifier**, perché è necessario tener traccia dei registri disponibili e fare un'allocazione adeguata per ciascun identificatore. Anche se non sono necessari cambiamenti sostanziali, questa gestione richiede un'**inizializzazione più complessa**.

La difficoltà di alcune modifiche dipende dal fatto che il tracciamento dei carichi comporta l'introduzione di molte **regole di copia**, mentre la gestione di un set di registri finiti è un cambiamento relativamente semplice, una volta che il tracciamento dei carichi è già stato implementato.

Conclusione finale

- Il **tracciamento dei carichi** e l'introduzione di set Before e After aumenta notevolmente la **complessità della grammatica**, mentre la gestione di un set di registri finiti è una modifica relativamente **più semplice** in quanto si basa su un meccanismo già introdotto.
- La **complessità aumenta con l'introduzione di regole di copia**, che devono essere scritte per ogni produzione e che comportano un grande aumento del numero di regole nella grammatica.

Dalle grammatiche attribuite ai metodi ad-hoc

Problemi Grammatiche Attribuite

- Le **regole di copia** (copy rules) aumentano la **difficoltà cognitiva** di comprensione e manutenzione del codice.
- Aumentano anche i **requisiti di spazio**, poiché bisogna copiare gli attributi tra i vari nodi dell'albero sintattico, e l'uso di **puntatori** può peggiorare ulteriormente la difficoltà cognitiva.
- Il risultato finale è un **albero attribuito**, in cui le informazioni semantiche sono assegnate ai nodi dell'albero. Questo comporta due possibili soluzioni:
 - **Costruire l'albero di parsing** e poi cercare le risposte all'interno di esso.
 - **Copiare le informazioni nei nodi** e fare riferimento a queste informazioni tramite il nodo radice dell'albero.

Traduzione Sintattica Diretta Ad-Hoc

Un altro approccio più flessibile è l'uso della **traduzione sintattica diretta ad-hoc** (Ad-hoc Syntax-Directed Translation, SDT), che si basa su un parser bottom-up:

- **Associazione di frammenti di codice con ogni produzione**: ad ogni riduzione della produzione, viene eseguito il frammento di codice associato.
- La **flessibilità completa** è offerta dalla possibilità di includere codice arbitrario (anche codice che potrebbe essere problematico), il che permette di adattare il compilatore a vari scenari.

Per fare funzionare questo approccio:

- È necessario **dare un nome agli attributi** di ciascun simbolo a sinistra e a destra della produzione.
 - Yacc, per esempio, usa i simboli `$$`, `$1`, `$2`, ..., `$n` per fare riferimento agli attributi dei simboli nelle produzioni.
- È necessario avere uno **schema di valutazione** che si inserisce bene nell'algoritmo LR(1).

La maggior parte dei parser utilizza questo **approccio ad-hoc** per l'analisi semantica, con vari vantaggi e svantaggi:

- **Vantaggi:**
 - Supera le limitazioni del paradigma delle grammatiche attribuite.
 - È più **efficiente** e **flessibile**, poiché consente l'esecuzione di codice arbitrario in qualsiasi punto della produzione.
- **Svantaggi:**
 - È necessario scrivere il codice manualmente, senza l'assistenza di strumenti.
 - Il programmatore si occupa direttamente dei dettagli complessi.

Molti generatori di parser, come Yacc, supportano una **notazione simile a Yacc**, che facilita questo approccio.

Usi Tipici

- **Costruire una tabella dei simboli:**
 - Si possono inserire le informazioni di dichiarazione durante il parsing.
 - Alla fine della sintassi della dichiarazione, si può fare un post-processing per verificare errori.
- **Controllo degli errori e verifica dei tipi:**
 - **Definire prima dell'uso e controllare al momento del riferimento** (ad esempio, verificando la dimensione, il tipo o la compatibilità di tipo di un'espressione).
 - Si può eseguire un controllo di tipo **bottom-up** o **verifica dell'interfaccia di una funzione**.

Questa è davvero "Ad-hoc"?

Il confronto tra **pratica** e **grammatiche attribuite** rivela:

- **Somiglianze:**
 - Entrambi gli approcci associano regole o azioni alle produzioni.
 - L'ordine di applicazione delle regole è determinato dagli **strumenti** (parser), non dall'autore.
 - I nomi dei simboli sono **astratti**.
- **Differenze:**

- Le azioni sono applicate come un'unità nel caso della traduzione sintattica diretta, mentre nelle grammatiche attribuite le regole sono applicate in modo **funzionale**.
- Nel caso delle azioni ad-hoc, **qualsiasi cosa è permessa**, mentre nelle grammatiche attribuite le regole sono più strutturate e formali.
- Le grammatiche attribuite sono più **astratte** rispetto alle azioni ad-hoc.

Limitazioni

L'approccio ad-hoc richiede che le azioni vengano eseguite in un **ordine rigido** (post-ordine, da sinistra a destra o bottom-up), con alcune implicazioni:

- **Dichiarazioni prima dell'uso.**
- Non è possibile passare informazioni contestuali **dal basso verso l'alto**.
- La soluzione potrebbe necessitare di **variabili globali**, che richiedono una gestione più complessa.

Come adattare la traduzione sintattica ad-hoc in un parser LR(1)?

- Per adattare il codice in un parser LR(1), è necessario:
 - Memorizzare gli attributi nella **pila** insieme allo stato e al simbolo.
 - Usare uno **schema di nominazione** per accedere agli attributi, come `$n` che fa riferimento alla posizione nella pila.
 - Sequenziare l'applicazione delle regole in base alla riduzione, aggiungendo una grande dichiarazione `case` al parser.

Alternativa: Passeggiata sull'Albero Sintattico Astratto (AST)

Quando le azioni necessarie non si adattano bene alla traduzione sintattica diretta ad-hoc, è possibile utilizzare una **strategia basata sull'albero sintattico astratto (AST)**:

1. Costruire l'AST:

- Invece di eseguire azioni durante il parsing, si costruisce un **albero sintattico astratto** che rappresenta la struttura della grammatica in modo più compatto e astratto rispetto all'albero di derivazione.
- Ad esempio:
 - Un nodo per ogni operazione (es. somma, moltiplicazione).
 - Foglie per ogni valore (es. numeri o variabili).
- Ogni nodo dell'albero contiene i dati e le regole necessarie per la successiva analisi o elaborazione.

2. Eseguire azioni durante le passeggiate sull'AST:

- Una volta costruito l'AST, le azioni semantiche vengono eseguite tramite una o più **visite dell'albero (tree walks)**.
- Questo approccio è particolarmente comune nei linguaggi orientati agli oggetti, dove si utilizza il **pattern Visitor**:
 - Ogni nodo dell'albero implementa un metodo che definisce come elaborare quel nodo.
 - Un **Visitor** percorre l'albero e richiama i metodi corrispondenti ai nodi.

3. Passaggi multipli per maggiore flessibilità:

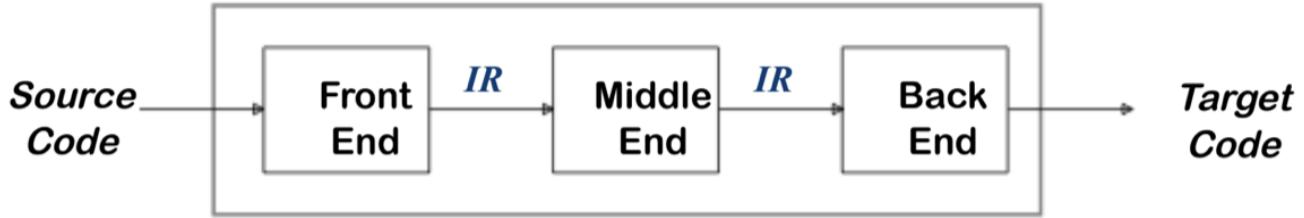
- L'AST consente di effettuare più passaggi (o **tree walks**) per attività diverse:
 - Primo passaggio: verifica e calcolo degli attributi.
 - Secondo passaggio: generazione di codice intermedio.
 - Terzo passaggio: ottimizzazioni.
- Questo approccio offre una maggiore **flessibilità**, consentendo di separare le fasi di analisi e generazione del codice.

05-rappresentazione-intermedia

- [Rappresentazione Intermedia](#)
 - [Tipi di IR](#)
 - [Livello di astrazione](#)
 - [Usare rappresentazioni multiple](#)
 - [Modelli di Memoria](#)
 - [Tabelle dei simboli](#)
 - [Tabelle dei Simboli Senza Hash](#)
 - [La procedura di astrazione](#)
 - [Compilazione separata](#)
 - [La procedura come astrazione di controllo](#)
 - [La procedura come spazio dei nomi](#)
 - [La procedura come interfaccia esterna](#)
 - [Stabilire l'indirizzabilità](#)
 - [Creazione di indirizzi di base](#)
 - [Accesso alle variabili non locali](#)
 - [Collegamenti di accesso VS Display](#)
 - [Creazione e distruzione di record di attivazione](#)
 - [Creazione degli AR](#)
 - [Distruzione degli AR](#)
 - [Registri di salvataggio](#)
 - [Procedura di chiamata](#)
 - [Allocazione degli AR](#)
 - [Senza ricorsione](#)
 - [Comunicazione tra le procedure](#)
 - [Meccanismi di passaggio dei parametri](#)
 - [Record di Attivazione - AR](#)

Rappresentazione Intermedia

La parte di front-end produce una rappresentazione intermedia, il middle-end trasforma la rappresentazione intermedia prodotta dal front-end in una rappresentazione intermedia equivalente, attraverso diverse passate ognuna delle quali applica tecniche di ottimizzazione, che funziona in modo più efficiente e, infine, il back-end trasforma la rappresentazione intermedia in codice nativo eseguibile dal processore.



Le decisioni nella progettazione IR influenzano la velocità e l'efficienza del compilatore, vediamo alcuni importanti proprietà dell'IR:

- Facilità di generazione: l'IR deve essere semplice da produrre a partire dal codice sorgente durante il front end.
- Facilità di manipolazione: deve essere facile analizzare e trasformare l'IR durante il middle end.
- Dimensione delle procedure: l'IR non dovrebbe essere troppo grande, perché procedure molto ampie possono richiedere molta memoria e rallentare le analisi.
- Libertà di espressione: l'IR deve essere abbastanza flessibile da rappresentare tutti i costrutti del linguaggio sorgente senza perdita di informazioni.
- Livello di astrazione: l'IR deve essere abbastanza astratto da essere indipendente dall'architettura hardware, ma non così astratto da rendere difficile la generazione di codice macchina nella fase di back end.

Tipi di IR

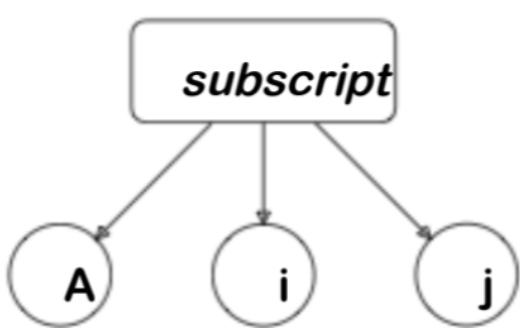
Le rappresentazioni intermedie si dividono in tre principali categorie:

1. **Strutturali:** questo tipo di IR è rappresentato in modo grafico, come grafi o alberi, che catturano la struttura logica e sintattica del programma, queste tendono a occupare molta memoria, specialmente per programmi complessi. Esempi: AST(Abstract Syntax Tree), DAG (Directed Acyclic Graph)
2. **Linearici:** questo tipo di IR rappresenta il programma come una sequenza di istruzioni, simile a un linguaggio assembly o pseudo-codice per una macchina astratta, queste sono rappresentate con strutture dati più leggere, rendendo più facile lavorare con grandi programmi. Poiché le istruzioni sono disposte in sequenza, è più semplice riorganizzarle per ottimizzazioni come l'ordinamento dei cicli o la rimozione del codice morto. Esempio: TAC (Three-Address Code), Bytecode.
3. **Ibridi:** combina caratteristiche delle IR strutturali e linearici per ottenere i vantaggi di entrambe, di solito, unisce una rappresentazione grafica (per il controllo del flusso) con istruzioni lineari per le operazioni locali, è meno grande delle IR strutturali e più potente delle IR linearici per analisi complesse. Esempi: SSA (Static Single Assignment), Control-Flow Graph con TAC.

Livello di astrazione

Il **livello di astrazione** di una IR riguarda il grado di dettaglio che la rappresentazione intermedia fornisce sul programma, abbiamo due casi differenti di rappresentazioni per un accesso a un array:

1. ad alto livello (strutturale) -> AST
2. a basso livello (lineare) -> Linear Code



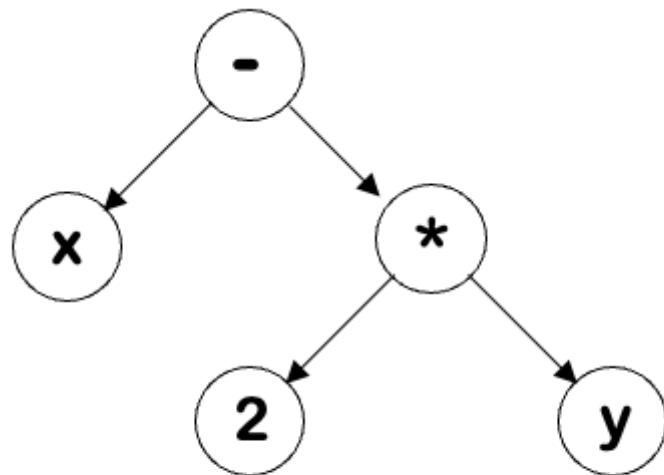
```
loadI 1      => r1
sub    rj, r1 => r2
loadI 10     => r3
mult   r2, r3 => r4
sub    ri, r1 => r5
add    r4, r5 => r6
loadI @A     => r7
add    r7, r6 => r8
load   r8     => rAij
```

Non sempre le IR strutturali vengono utilizzati per l'alto livello e le IR lineari per il basso livello.

- **Abstract Syntax Tree - AST:** è una rappresentazione ad albero della struttura sintattica di un programma, ma con un livello di astrazione maggiore rispetto al parse tree, in particolare, l'AST elimina nodi non essenziali e rappresenta solo le informazioni necessarie per comprendere la semantica del programma.

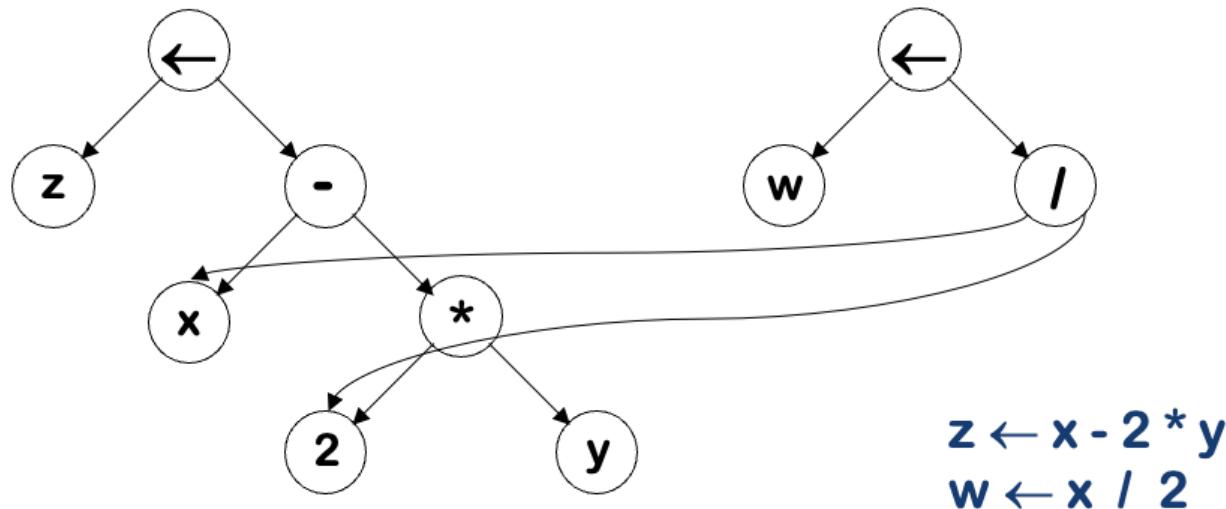
Sebbene l'AST sia naturalmente una struttura gerarchica, può essere linearizzato in forme come **notazione pre-fissa** (- 2 y x) o **post-fissa** (x 2 y -) per semplificare alcune

operazioni.



$$x - 2 * y$$

- **Direct Acyclic Graph - DAG:** è una rappresentazione grafica utilizzata per catturare le dipendenze tra operazioni o espressioni in un programma, è simile a un albero, ma con la differenza che consente la condivisione esplicita dei nodi quando ci sono ridondanze.



- **Stack Machine Code:** è una rappresentazione del codice che utilizza una macchina a pila per eseguire operazioni aritmetiche e logiche, in essa, tutti i calcoli avvengono manipolando una struttura dati a stack, dove gli operandi sono spinti (push) o estratti (pop) dalla pila. Questo approccio è semplice e compatto, ed è utilizzato da linguaggi e ambienti come Java (con il bytecode della JVM). Ad esempio $x - 2 * y$ la pila diventa:

- push x: [x]
- push 2: [x, 2]
- push y: [x, 2, y]
- multipl: [x, (2 * y)]

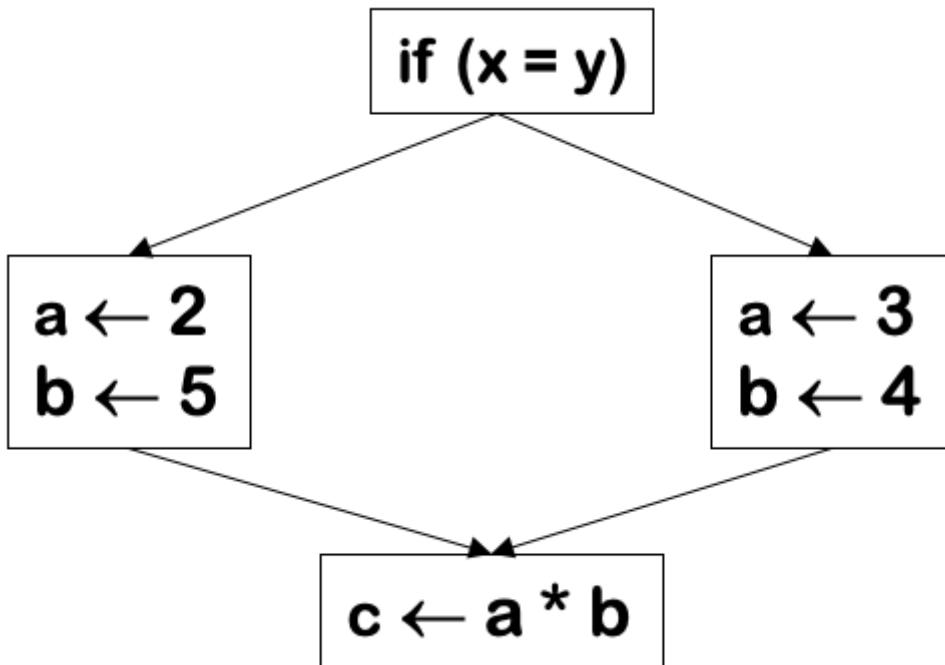
- subtract: $[(x - (2 * y))]$ (risultato finale)
- **Three Address Code - TAC**: è una rappresentazione intermedia del programma che utilizza istruzioni con al massimo tre operandi, è un formato comune nei compilatori perché bilancia semplicità e vicinanza al codice macchina reale. Ogni istruzione del TAC ha questa forma generale: $x \leftarrow y \text{ op } z$, ad esempio $z \leftarrow x - 2 * y$ diventa $t \leftarrow 2 * y; z \leftarrow x - t$.

Diverse varianti di TAC:

- **Tripla**: ogni istruzione è rappresentata come una tupla con tre campi: (operatore, operando1, operando2).
- **Quadrupla**: simile alla tripla, ma ogni istruzione esplicita anche il risultato.
- **Record lineare**: usa una sequenza di istruzioni simile a un linguaggio assembly, con nomi temporanei esplicativi.
- **Control-Flow Graph**: è una rappresentazione grafica utilizzata nei compilatori per modellare il flusso di controllo all'interno di un programma, in particolare in un singolo procedimento (funzione o metodo), questa rappresentazione è cruciale per analizzare e ottimizzare il codice, poiché mostra come le diverse istruzioni o blocchi di codice sono collegati tra loro, indicando come il controllo (ovvero l'esecuzione) si sposta da un'istruzione all'altra.

Struttura:

- Nodi (Basic Blocks): ogni nodo del grafo rappresenta un basic block, che è una sequenza di istruzioni che viene eseguita una volta iniziata, senza interruzioni. Un blocco base **non** può contenere istruzioni con salti (come `if`, `goto`, `return`) o biforazioni (come `if-else`), perché questi determinano cambiamenti nel flusso di controllo.
- Archi: gli archi nel grafo rappresentano il flusso di controllo tra i blocchi, indicano come l'esecuzione passa da un blocco all'altro.
- Se, per esempio, il codice contiene un'istruzione `if` che determina un salto, allora le istruzioni precedenti e quelle successive alla condizione `if` saranno in blocchi base separati, perché l'esecuzione può "saltare" da un punto all'altro, cambiando il flusso di controllo.



- **Static Single Assignment - SSA:** è una rappresentazione intermedia utilizzata nei compilatori che semplifica l'analisi e l'ottimizzazione del codice, l'idea principale di SSA è che ogni variabile o nome venga definito esattamente una volta nel programma, evitando ri-assegnazioni multiple.
Per ottenere la SSA, quando una variabile è assegnata più di una volta (ad esempio, in un ciclo o in una condizione `if-else`), si crea una nuova versione della variabile ogni volta che viene riassegnata, in questo modo, ogni "versione" della variabile ha una definizione unica.

```

if (condizione) {
    x = 1;
} else {
    x = 2;
}
y = x + 3;

//In SSA il codice diventa

if (condizione) {
    x1 = 1;
} else {
    x2 = 2;
}
x3 = phi(x1, x2); // phi funzione per scegliere tra x1 e x2
y = x3 + 3;
  
```

Usare rappresentazioni multiple

L'idea di utilizzare **più rappresentazioni intermedie (IR)** in un compilatore consiste nel passare gradualmente da rappresentazioni di alto livello, più astratte, a rappresentazioni di basso livello, più vicine al codice macchina. Ogni rappresentazione intermedia è progettata per essere ottimale per un determinato tipo di analisi o ottimizzazione, migliorando così l'efficienza e la qualità del codice generato.

Il compilatore **Open64** utilizza una rappresentazione intermedia chiamata **WHIRL**, che comprende **5 diversi livelli di IR**, ciascuno con un diverso grado di astrazione e dettagli.

Modelli di Memoria

I **modelli di memoria** definiscono come il compilatore gestisce i dati durante l'esecuzione del programma, ovvero se questi dati vengono memorizzati principalmente in **registri** (veloci ma limitati) o in **memoria principale** (lenta ma abbondante). Esistono due principali modelli di memoria, ciascuno con vantaggi e svantaggi:

1. Register-to-register model: il compilatore assume che tutti i valori che possono essere legalmente archiviati in un registro vengano effettivamente messi nei registri. Durante le prime fasi di compilazione, il compilatore non considera il numero limitato di registri della macchina reale, spetta al back-end del compilatore gestire il mapping effettivo ai registri fisici della macchina. Quando il numero di registri richiesti eccede quello disponibile, il back-end deve inserire istruzioni di **load** e **store** per scaricare temporaneamente i valori in memoria.
2. Memory-to-memory model: il compilatore assume che tutti i valori siano archiviati nella memoria principale, salvo quelli che vengono promossi temporaneamente ai registri subito prima di essere utilizzati. I dati vengono caricati nei registri solo quando strettamente necessari per un'operazione e poi scritti nuovamente in memoria. Il back-end del compilatore può eliminare i caricamenti e le scritture ridondanti per ottimizzare il codice.
3. Uso nei Compilatori RISC: I compilatori per architetture RISC (Reduced Instruction Set Computer) di solito adottano il **register-to-register model**.

Rappresentare il codice è solo una parte di una rappresentazione intermedia (IR), ci sono altri componenti necessari:

- **Tabella dei simboli**
- **Tabella delle costanti**
 - Rappresentazione, tipo
 - Classe di memorizzazione, offset
- **Mappa della memoria**

- Layout complessivo della memoria
- Informazioni sulle sovrapposizioni
- Assegnazioni di registri virtuali

Tabelle dei simboli

Le **tabelle dei simboli** sono strutture dati fondamentali in un compilatore, servono per memorizzare e gestire informazioni sui simboli del programma (es. variabili, funzioni, tipi), sono utilizzate per risolvere riferimenti e applicare regole di visibilità e scoping.

Un metodo classico per costruire una tabella dei simboli è utilizzare l'hashing, in questo approccio:

1. Indice basato su una funzione hash:

- La funzione hash calcola una posizione per ogni simbolo basandosi sul suo nome, questo riduce i tempi di ricerca rispetto a metodi lineari (es. lista o array).

2. Gestione delle collisioni:

- Se due simboli finiscono nello stesso indice (collisione), si usa una tecnica di gestione come il chaining (liste concatenate).

Tabelle dei Simboli Senza Hash

Le **tabelle dei simboli senza hash** sono un'alternativa, questo approccio nasce per evitare i problemi legati alle collisioni nella funzione hash, che possono degradare le prestazioni fino a una ricerca lineare nel caso peggiore.

"Perfect Hash": alcuni autori propongono di utilizzare funzioni di hashing perfette, ovvero funzioni che non generano collisioni, tuttavia, creare una funzione di hash perfetta per tutti i possibili simboli (soprattutto in compilatori generali) è complesso e poco pratico.

Un approccio alternativo è quello di utilizzare concetti derivati dalla **teoria degli automi** per costruire tabelle dei simboli che evitano completamente le collisioni e garantiscano prestazioni prevedibili e uniformi, indipendentemente dalla quantità e dalla distribuzione dei simboli.

Una proposta alternativa per eliminare l'uso delle funzioni hash nelle tabelle dei simboli è il metodo di **multiset discrimination** sviluppato da Paige & Cai. Questo si basa sull'**ordinamento offline** dello spazio dei nomi e sull'assegnazione di indici univoci, sfruttando tecniche derivate dagli automi deterministici a stati finiti (DFA).

Come funziona il metodo Paige & Cai:

1. Ordinamento Offline dello Spazio dei Nomi:

- Prima di elaborare il codice sorgente, lo spazio dei nomi (ossia i possibili identificatori, parole chiave, ecc.) viene ordinato in modo deterministico.

- Ad ogni nome, nello spazio dei nomi, viene assegnato un indice univoco basato su questo ordinamento.

2. Sostituzione dei Nomi con Indici:

- Durante l'elaborazione dell'input, i nomi nel codice sorgente vengono sostituiti dai loro indici predefiniti, questo elimina la necessità di calcolare una funzione hash dinamica per ogni nome, semplificando l'accesso e la gestione.
- Multiset Discrimination:
 - Dopo la sostituzione con gli indici, i simboli possono essere trattati come un insieme ordinato di interi (il multiset), questo rende le operazioni di confronto e ricerca molto più rapide, poiché si lavora su numeri interi anziché stringhe.

Un ulteriore miglioramento al metodo consiste nell'utilizzo di tecniche basate sui **DFA** per implementare una sostituzione lineare dei simboli senza ricorrere all'hashing.

Costruzione del DFA:

1. I nomi (parole chiave o identificatori) vengono rappresentati come un'espressione regolare composta del tipo:

```
r1 | r2 | r3 | ... | rk
```

```
// Dove ogni `ri` è un nome nello spazio dei nomi.
```

2. Questa espressione regolare viene convertita in un DFA aciclico, poiché le liste di parole chiave o nomi non hanno cicli.

La costruzione incrementale di un automa deterministico a stati finiti aciclico è una tecnica che consente di aggiungere nuove parole all'automa in modo dinamico, senza doverlo ricostruire completamente, questo approccio è particolarmente utile per gestire piccoli insiemi di chiavi (ad esempio, identificatori o parole chiave all'interno di una procedura).

Durante l'aggiunta di una parola, il DFA richiede un accesso in memoria per ogni carattere nella parola, fino a raggiungere lo stato di errore.

Se il DFA diventa molto grande, i costi di accesso alla memoria per ogni carattere possono aumentare significativamente, tuttavia, per piccoli insiemi di chiavi (come i nomi in una procedura), questo problema non è rilevante.

Ottimizzazioni per ridurre costi e dimensioni:

- Stato esplicito sull'ultimo nodo:
 - L'ultimo stato di ogni percorso del DFA può essere reso esplicito solo quando il percorso viene allungato.

- Vantaggi:
 - Riduzione sostanziale dei costi in memoria, poiché non si crea immediatamente lo stato finale.
 - La struttura è più compatta fino a quando non è necessario aggiungere nuove parole.
- Granularità vs. Dimensione degli stati:
 - È possibile regolare il livello di dettaglio con cui rappresentare ogni stato per trovare un equilibrio tra il costo di rappresentazione e l'efficienza.
 - Ad esempio, rappresentazioni più granulari degli stati possono aumentare le prestazioni, ma a costo di una maggiore memoria.
- Gestione separata della capitalizzazione:
 - La capitalizzazione delle lettere (es. Foo vs foo) può essere codificata separatamente usando stringhe di bit legate agli stati finali.
 - Vantaggi:
 - Evita di duplicare interi percorsi nel DFA per distinguere parole con lettere maiuscole e minuscole.
 - Permette un riconoscimento più efficiente.

La procedura di astrazione

Il compilatore deve fornire, per ogni costrutto del linguaggio di programmazione, un implementazione (o almeno una strategia).

Questi costrutti cadono in due principali categorie:

- Dichiarazioni individuali
- Procedure

Il compilatore deve implementare ogni costrutto del linguaggio, concentrandosi soprattutto sulle **procedure**, poiché forniscono il contesto necessario per gestire le dichiarazioni.

Le procedure sono un meccanismo fondamentale nei linguaggi di programmazione che permette di raggruppare un insieme di istruzioni in un blocco riutilizzabile e con un nome specifico = (funzioni).

Le procedure sono fondamentali per:

- Nascondere le informazioni (information hiding)
- Creare spazi dei nomi separati
- Definire interfacce uniformi

Poiché l'hardware non supporta direttamente queste astrazioni, il compilatore deve implementarle in modo efficiente, deve:

1. Decidere dove collocare i valori e come calcolarli.
2. Gestire il comportamento tra tempo di compilazione e tempo di esecuzione.
3. Integrare il codice con altri programmi e sistemi operativi.

Problemi chiave includono:

- Allocazione dello spazio di memoria e associazione dei nomi agli indirizzi.
- Creazione di codice per accedere a valori noti e non noti a tempo di compilazione.
- Garantire efficienza e compatibilità.

In sintesi, il compilatore deve produrre codice che combini correttamente tutte le procedure per formare un programma funzionante ed efficiente.

Compilazione separata

Le procedure permettono di sfruttare la **compilazione separata**, un meccanismo fondamentale per costruire programmi complessi e collaborativi.

Vantaggi della compilazione separata:

1. Programmi non banali: permette di gestire progetti grandi e modulari;
2. Tempi di compilazione ridotti: ogni file o modulo può essere compilato separatamente, evitando di ricompilare l'intero programma a ogni modifica;
3. Collaborazione tra programmatore: ogni sviluppatore può lavorare su parti diverse del codice in modo indipendente;
4. Procedure indipendenti: ogni procedura può essere compilata singolarmente e integrata successivamente;

La convenzione di collegamento delle procedure è un insieme di regole che definisce come le procedure (o funzioni) interagiscono con l'ambiente di runtime di un programma durante una chiamata:

- Garantisce che ogni procedura erediti un ambiente di runtime valido - quando una procedura viene chiamata, deve poter accedere a tutte le informazioni necessarie per funzionare correttamente, come i parametri di input, le variabili locali e lo stato generale del programma. La convenzione di collegamento si assicura che l'ambiente chiamante prepari tutto il necessario (parametri, spazio sullo stack, ecc.) prima di trasferire il controllo alla procedura chiamata;
- Assicura che al termine della procedura, l'ambiente chiamante venga ripristinato correttamente - dopo che la procedura termina la sua esecuzione, il controllo

- deve tornare al punto in cui è stata chiamata, con l'ambiente originale intatto;
- Il compilatore genera il codice per gestire questi aspetti in base alle convenzioni del sistema;

Supporto limitato dell'hardware:

L'hardware fornisce solo strumenti di base, come bit, byte, numeri interi e trasferimento di controllo (chiamata e ritorno). Tuttavia, non supporta direttamente:

- Meccanismi di chiamata e ritorno completi (ad esempio il salvataggio del contesto);
- Interfacce tra procedure;
- Spazi dei nomi e ambiti nidificati;

Chi fornisce queste funzionalità?

Queste astrazioni sono implementate tramite:

- **Compilatore**: genera il codice che gestisce la chiamata e il ritorno, spazi dei nomi e contesti;
- **Sistema di esecuzione**: gestisce l'ambiente di runtime;
- **Linker e caricatore**: collegano procedure compilate separatamente e le rendono eseguibili;
- **Sistema operativo**: supporta il caricamento del programma e la gestione della memoria;

La procedura come astrazione di controllo

Le procedure, come concetto astratto, ci permettono di definire un flusso di controllo ben preciso all'interno di un programma:

1. Quando una procedura viene chiamata, l'esecuzione si sposta dal punto della chiamata (il sito di invocazione) all'inizio della procedura.
2. Durante l'esecuzione, la procedura può lavorare con parametri passati dall'esterno e con variabili locali definite al suo interno.
3. Una volta terminata, la procedura restituisce il controllo al punto immediatamente successivo alla chiamata.

Questa semplicità apparente nasconde però delle sfide: per far funzionare tutto questo, il compilatore deve generare del codice che si occupi di gestire alcuni aspetti fondamentali in modo automatico.

Cosa deve fare il compilatore?

4. **Salvare e ripristinare l'indirizzo di ritorno**

Quando la procedura viene chiamata, il compilatore deve salvare da qualche parte l'indirizzo del punto successivo alla chiamata, così da sapere dove tornare quando la

procedura termina. Senza questo passaggio, il programma non saprebbe come riprendere l'esecuzione.

5. Gestire i parametri

I valori o riferimenti passati alla procedura (chiamati parametri effettivi) devono essere mappati ai parametri utilizzati all'interno della procedura (chiamati parametri formali). Questa corrispondenza garantisce che la procedura lavori con i dati giusti.

6. Creare spazio per le variabili locali

Ogni procedura può avere delle variabili locali, cioè variabili che esistono solo mentre la procedura è in esecuzione. Il compilatore deve riservare uno spazio in memoria per queste variabili, che deve essere creato all'avvio della procedura e liberato quando questa termina.

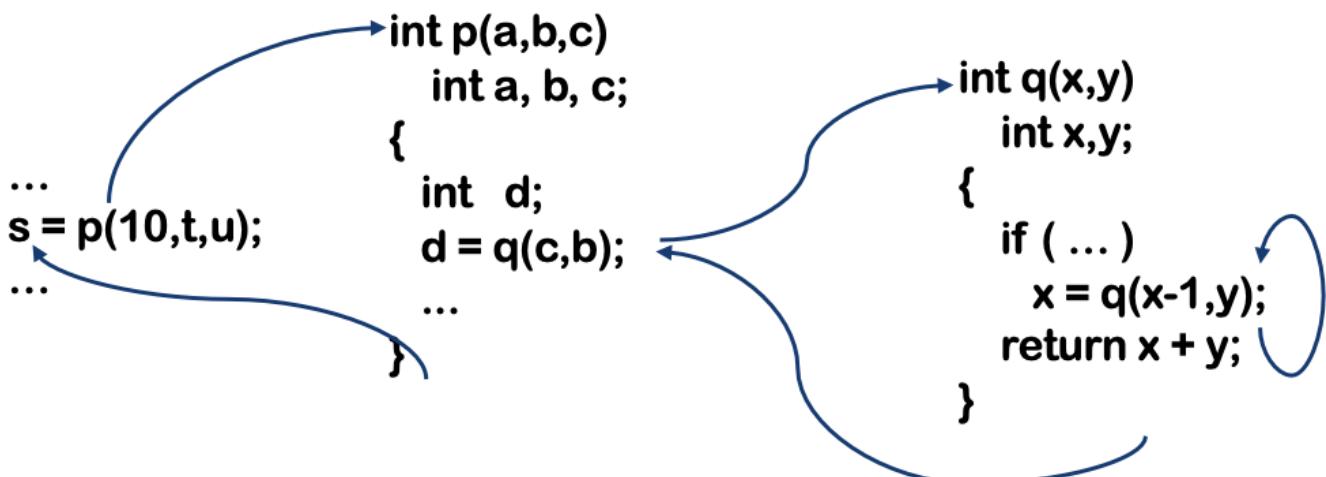
Il vero problema si presenta quando la procedura è **ricorsiva**, cioè quando può chiamare sé stessa prima di aver completato l'esecuzione precedente, in casi come questo, ogni chiamata deve mantenere il proprio stato indipendente.

La soluzione: la pila di attivazione

Per risolvere il problema, il sistema utilizza una pila di attivazione:

1. Quando una procedura viene chiamata, si aggiunge (push) in cima alla pila un blocco di memoria che contiene:
 - L'indirizzo di ritorno.
 - I parametri della procedura.
 - Lo spazio per le variabili locali.
2. Quando la procedura termina, il blocco in cima alla pila viene rimosso (pop) e il controllo ritorna all'indirizzo salvato.

Questa strategia garantisce che ogni chiamata, inclusa la ricorsione, abbia il proprio spazio separato per lavorare, evitando conflitti tra chiamate diverse.



La procedura come spazio dei nomi

Quando parliamo di una procedura come "spazio dei nomi", intendiamo che ogni procedura ha una **propria area isolata per dichiarare e gestire nomi** (variabili, costanti, funzioni). Questo concetto è regolato dalle regole dell'**ambito lessicale**, che definiscono come e dove i nomi (variabili o funzioni) possono essere utilizzati nel codice.

- Una procedura definisce un'area in cui i nomi dichiarati sono validi e accessibili. Fuori da quella procedura, i nomi locali non sono visibili né utilizzabili (scope).
- Se un nome è dichiarato localmente in una procedura, "oscura" qualsiasi altro nome con lo stesso identificatore dichiarato al di fuori della procedura.
- Una variabile dichiarata localmente a una procedura non è accessibile al di fuori di essa.

L'ambito (scope) lessicale è una convenzione che stabilisce le regole per determinare quale "versione" di una variabile è utilizzabile in una parte del codice, basandosi sulla posizione della variabile **nel codice sorgente**. È definito al momento della compilazione, non a runtime.

Il compilatore gestisce lo spazio dei nomi e l'ambito lessicale tramite **tabelle dei simboli**, strutture dati che memorizzano informazioni sui nomi dichiarati e sul loro contesto.

Interfaccia delle tabelle dei simboli:

- `insert(name, level)` : aggiunge un nuovo record per il nome dichiarato, indicando il livello (scope) in cui è valido.
 - Es: dichiarare `int x` a livello locale aggiunge `x` alla tabella per lo scope attuale.
 - `lookup(name, level)` : cerca il nome nel livello specificato o negli scope superiori (se permesso), restituendo un puntatore o indice alla dichiarazione corrispondente.
 - Es: cercare `x` in una procedura verifica se è dichiarata localmente o globalmente.
 - `delete(level)` : rimuove tutti i nomi associati a un livello quando si esce da quello scope.
 - Es: quando una funzione termina, elimina tutte le variabili locali dalla tabella.
- Sono stati proposti molti schemi di implementazione.

La procedura come interfaccia esterna

Le procedure fungono da punto di interfaccia per l'interazione tra programmi, sistema operativo e utenti.

Esecuzione di un programma: da dove inizia?

- Ogni programma ha un punto di inizio, come `main()` in C o il corpo principale in Python, il programmatore specifica questo punto, e il sistema operativo lo usa per avviare l'esecuzione.

Le variabili sono allocate in modo diverso a seconda della loro durata e visibilità.

- Variabili automatiche e locali:
 - Memorizzate nel record di attivazione della procedura (stack frame) o in registri della CPU.
 - Durata limitata: esistono solo durante l'esecuzione della procedura.
- Variabili statiche:
 - Durata: persistono per tutta l'esecuzione del programma.
 - Ambito della procedura: associate al nome della procedura in cui sono dichiarate.
 - Ambito del file: associate al file in cui sono dichiarate (es: `static int x;` in C).
 - Esistono in aree di memoria pre-allocate.
- Variabili globali:
 - Visibili a tutto il programma, memorizzate in aree dedicate della memoria globale.
 - Durata: l'intera esecuzione del programma.

La memoria di un programma è divisa in sezioni, ognuna con un ruolo specifico:

- Codice: contiene le istruzioni del programma, con dimensioni note a tempo di compilazione.
- Dati statici e globali: aree pre-allocate per variabili globali e statiche, note anch'esse a tempo di compilazione.
- Stack: utilizzato per memorizzare i record di attivazione (stack frame) delle procedure.
- Heap: utilizzato per allocazioni dinamiche (es: `malloc` in C).

Quando una procedura viene chiamata, il compilatore alloca un **record di attivazione (AR)**, che contiene:

- Variabili locali: allocate nello stack.
- Parametri formali: passati alla procedura.
- Indirizzo di ritorno: per tornare al chiamante.
- Valori di controllo: come lo stato dei registri.

Ricorsione:

- Per ogni invocazione ricorsiva, viene creato un nuovo record di attivazione.
- Questo permette a ciascuna invocazione di mantenere il proprio insieme di variabili locali.

Traduzione dei nomi locali

Il compilatore rappresenta ogni variabile come una coppia di coordinate statiche:

- `<level, offset>`:
 - `level`: livello di annidamento lessicale della procedura.
 - `offset`: posizione unica della variabile all'interno del livello.

Esempio: consideriamo il seguente codice

```

int x; // Variabile globale
void f() {
    int y; // Livello 1, offset 0
    void g() {
        int z; // Livello 2, offset 0
    }
}

```

- `x` (globale): non ha `level`, vive nell'area globale.
- `y` (in `f`): `level = 1, offset = 0`.
- `z` (in `g`): `level = 2, offset = 0`.

Generazione del codice

- In fase di compilazione:
 - Il compilatore assegna un `level` e un `offset` a ogni variabile e memorizza queste informazioni nella **tabella dei simboli**, questi dati vengono utilizzati per calcolare gli indirizzi di memoria e generare istruzioni che accedono alle variabili in fase di esecuzione.
- In fase di esecuzione:
 - Il codice generato usa le coordinate statiche per accedere alla posizione corretta della variabile nello stack o nella memoria globale.

Stabilire l'indirizzabilità

Creazione di indirizzi di base

Il compilatore deve sapere come trovare l'indirizzo in memoria delle variabili a seconda del loro tipo (locali, statiche, non locali):

- Variabili Locali:
 - Vengono tradotte in **coordinate statiche**: `<level, offset>`.
 - Il record di attivazione (AR) contiene un puntatore, chiamato **ARP** (Activation Record Pointer), che identifica il record della procedura corrente.
 - L'indirizzo della variabile è ottenuto calcolando: Indirizzo Variabile = ARP + offset
- Variabili Statiche: per queste variabili, il compilatore crea etichette simboliche (es. `&_fee` per una variabile chiamata `fee`), l'indirizzo è risolto una volta in fase di compilazione ed è fisso.
- Variabili non Locali: quando una variabile appartiene a un ambito esterno rispetto alla procedura corrente, il compilatore:
 - traduce il nome in coordinate statiche.

- trova l'ARP appropriato per il livello di annidamento richiesto.
- calcola l'indirizzo come: $\text{ARP}_{target} + \text{offset}$.

Accesso alle variabili non locali

1. Collegamenti di accesso - access link

- Ogni AR contiene un puntatore all'AR dell'antenato lessicale immediato.
- L'antenato lessicale è la procedura che "contiene" la procedura corrente, questo non è necessariamente il chiamante diretto.

Come funziona:

- Se una variabile non locale è al livello j e la procedura corrente è al livello k , il compilatore segue i collegamenti di accesso per risalire nella catena degli AR.
- Il costo dell'accesso dipende dalla distanza lessicale (quanto è lontano il livello j da k).
Esempio:

```
void outer() {
    int x; // livello 1
    void inner() {
        void nested() {
            // Accesso a x (livello 1) da nested (livello 3)
        }
    }
}
```

- `nested()` si trova al livello 3 e deve risalire due livelli di collegamenti di accesso per raggiungere `x` (al livello 1).

Pro: funziona sempre, anche se gli AR sopravvivono alla procedura.

Contro: il costo cresce con la distanza lessicale.

2. Display

- Usa un array globale chiamato `display`, che memorizza puntatori agli AR per ogni livello lessicale.
- L'indirizzo di una variabile non locale è trovato direttamente accedendo al `display`:
`Indirizzo Variabile = Display[j] + offset`.
- Il costo dell'accesso è costante, indipendentemente dalla distanza lessicale.
Esempio:

```
void outer() {
    int x; // livello 1
    void inner() {
        void nested() {
```

```

        // Accesso a x (livello 1) da nested (livello 3)
    }
}
}

```

- `nested()` accede a `x` al livello 1.
 - `Display[1]` contiene l'ARP per `outer`.
 - L'indirizzo di `x` è calcolato con `Display[1] + offset`.

Pro: accesso rapido e costante.

Contro: richiede aggiornamenti al display durante ogni chiamata e ritorno, e utilizza un registro aggiuntivo.

Collegamenti di accesso VS Display

Collegamenti di accesso

- Sovraccarico: ogni chiamata aggiunge un puntatore al record di attivazione.
- Costo di accesso: proporzionale alla distanza lessicale.
- Persistenza: funziona anche se gli AR vivono oltre la fine della procedura.

Display

- Sovraccarico: ogni chiamata aggiorna l'array globale.
- Costo di accesso: sempre costante.
- Efficienza: richiede un registro per il display, che può essere critico per la velocità complessiva.

Perché funzionano?

Entrambi gli approcci funzionano perché:

1. L'annidamento lessicale garantisce che una procedura acceda solo alle variabili visibili in base all'ambito.
2. Il compilatore inserisce automaticamente il codice necessario per:
 - Gestire i collegamenti di accesso o il display in ogni chiamata e ritorno.
 - Tradurre i nomi delle variabili in coordinate statiche.

Creazione e distruzione di record di attivazione

I record di attivazione (AR) sono strutture dati allocate in memoria durante l'esecuzione di una procedura, che memorizzano tutte le informazioni necessarie per eseguire la procedura stessa e per tornare correttamente al chiamante.

Creazione degli AR

Quando viene chiamata una procedura:

1. Chiamante:

- Alloca spazio per l'AR del chiamato.
- Salva i parametri, l'indirizzo di ritorno e l'ARP (Activation Record Pointer).
- (Se necessario) gestisce i collegamenti di accesso.

2. Chiamato:

- Completa la configurazione dell'AR (ad esempio, allocando lo spazio per le variabili locali).

Distruzione degli AR

Quando la procedura termina:

1. Chiamato:

- Ripristina lo stato del chiamante (come i registri salvati).
- Salva il valore di ritorno nell'AR del chiamante, se necessario.

2. Chiamante:

- Libera lo spazio dell'AR del chiamato.
- Ripristina l'ambiente originale per continuare l'esecuzione.

Registri di salvataggio

Chi salva i registri?

- Chiamante salva i registri che saranno utilizzati dopo la chiamata (registri LIVE).
- Chiamato salva i registri che utilizzerà durante l'esecuzione.

I registri sono divisi in:

1. Registri del chiamante
2. Registri del chiamato

Procedura di chiamata

Sequenza Pre-Chiamata

1. Il chiamante prepara lo stack:

- Salva i parametri, l'indirizzo di ritorno e l'ARP.

2. Passa il controllo al chiamato:

- Esegue il salto all'indirizzo iniziale della procedura chiamata.

Il prologo completa l'inizializzazione dell'AR:

3. Salva i registri.
4. Configura il display (se usato).
5. Alloca spazio per variabili locali.

Sequenza Post-Ritorno

6. Copia il valore di ritorno (se presente).
7. Libera lo stack.
8. Ripristina i registri salvati.

Allocazione degli AR

1. AR sullo Stack

- Vantaggi:
 - Facile da gestire con un puntatore allo stack.
 - Supporta la ricorsione (ogni chiamata ha il suo AR).
- Processo:
 1. Il chiamante alloca lo spazio sullo stack.
 2. Il chiamato usa lo spazio per le variabili locali.

2. AR sull'Heap

- Vantaggi:
 - Utile per procedure che richiedono durata prolungata (es. coroutine).
- Svantaggi:
 - Maggior costo di allocazione/deallocazione.
- Processo:
 1. Il chiamato alloca l'AR sull'heap.
 2. Libera l'AR manualmente alla fine.

Senza ricorsione

Gli AR possono essere statici (allocati una volta sola):

- Nessuna nuova allocazione per ogni chiamata.
- Utilizzabile solo se la procedura non è ricorsiva.

Comunicazione tra le procedure

La comunicazione tra le procedure si riferisce al passaggio di informazioni (parametri) tra una procedura chiamante e una procedura chiamata. Questo avviene attraverso meccanismi che collegano variabili del contesto del chiamante con quelle della procedura chiamata.

Meccanismi di passaggio dei parametri

1. Call-by-reference - Chiamata per riferimento

Passa un puntatore (indirizzo in memoria) al parametro effettivo, il chiamato può modificare direttamente il valore originale della variabile del chiamante.

Nell'activation record (AR) viene salvato l'indirizzo del parametro, se esso viene usato da più nomi, ogni nome fa riferimento allo stesso indirizzo (es. call foo(x, x, x)).

Vantaggi: efficiente in termini di spazio quando si passano strutture di grandi dimensioni.

Svantaggi: il chiamato può modificare i dati originali, causando effetti collaterali inattesi.

```
void increment(int &x) { // Passaggio per riferimento
    x++;
}

int main() {
    int value = 10;
    increment(value); // value è incrementato a 11
    return 0;
}
```

2. Call-by-value - Chiamata per valore

Passa una copia del valore del parametro al chiamato, esso può modificare la copia, ma l'originale nel chiamante rimane inalterato.

Una posizione di memoria distinta è allocata per il valore copiato, nell'AR vengono riservati slot per i valori copiati.

Vantaggi: evita effetti collaterali inaspettati sul chiamante

Svantaggi: inefficiente per strutture grandi perché ogni elemento viene copiato.

```
void increment(int x) { // Passaggio per valore
    x++;
}

int main() {
    int value = 10;
    increment(value); // value rimane 10
    return 0;
}
```

Record di Attivazione - AR

Dove sono allocati gli AR?

1. Sullo **Stack**:

- Ogni chiamata spinge un nuovo AR sullo stack.
- Facile da estendere: basta incrementare il puntatore dello stack.
- Responsabilità condivisa:
 - Il **chiamante**: spinge parametri, indirizzo di ritorno e altre informazioni.
 - Il **chiamato**: alloca spazio per le variabili locali.

Vantaggi:

- Efficienza nella gestione della memoria.
- Supporto naturale alla ricorsione.

2. Sull'**Heap**:

- AR allocato dinamicamente.
- Più flessibile, ma più costoso in termini di gestione.
- Adatto a scenari complessi, come procedure che vivono oltre la loro chiamata.

Esempio:

- I linguaggi funzionali che fanno uso di chiusure o di oggetti a durata estesa potrebbero allocare AR sull'heap.

3. **Statico**:

- Per funzioni non ricorsive, l'AR può essere allocato staticamente.
- Più efficiente perché evita l'overhead di allocazione e deallocazione.

05.1-LLVM-IR

- [Struttura di una file di bitcode LLVM](#)
 - [Identificatori](#)
 - [Tipi di dato semplici](#)
 - [Tipi di dato composti](#)
 - [Target Triple e Data Layout](#)
 - [Target Triple](#)
 - [Data Layout](#)
 - [Costanti](#)
 - [La struttura di una funzione](#)
 - [Istruzioni LLVM](#)

Struttura di una file di bitcode LLVM

LLVM (Low-Level Virtual Machine) è un'infrastruttura di compilazione modulare e riutilizzabile. In sostanza, è un insieme di strumenti e librerie che consentono di costruire compilatori, ottimizzatori e altri strumenti legati alla programmazione. È progettato per essere altamente flessibile e supportare un'ampia gamma di linguaggi di programmazione.

Un file di **bitcode LLVM** è la rappresentazione binaria del codice IR, serve come formato intermedio che può essere usato per:

- **Memorizzare il programma** durante le fasi di compilazione;
- **Riutilizzare** il codice ottimizzato;
- **Eseguire analisi o trasformazioni** su moduli già generati;

Un file di bitcode rappresenta un **modulo LLVM**, che è una collezione di oggetti del programma (funzioni, variabili, metadati, ecc.). La struttura è composta da:

- Variabili globali: possono avere diverse modalità di collegamento (linkage), come:
 - `external`: accessibile ad altri moduli;
 - `internal`: visibile solo all'interno del modulo;
 - `weak`: permette risoluzioni multiple (utile per le librerie condivise);
- Funzioni globali;
- Metadata;

Identificatori

Gli **identificatori** sono utilizzati per fare riferimento a **tipi**, **valori** e altre entità (come funzioni o variabili globali) presenti nel codice intermedio.

- entità visibili a livello **globale**: il nome inizia con il carattere @;
- entità **locali**, che esistono solo all'interno di una funzione: il nome inizia con il carattere %;
- Identificatori "anonimi": il nome è un intero non negativo
 - @123, %0, %1, %2;
- Identificatori con nome usano la RE: {[%@] [-a-zA-Z\$_] [-a-zA-Z\$_0-9]*}
 - @x, @main, @bond.james.bond.007;

Tipi di dato semplici

I tipi di dato semplici rappresentano i blocchi fondamentali per definire variabili, valori e operazioni.

- void (nessun valore): void;
- interi (con dimensione N in bits): i N
 - i1, i8, i23, i32, i123456;
- floating point (dimensione standard):
 - half, float, double, fp128;
 - anche specifici: x86_fp80, ppc_fp128;
- puntatori: type *
- i32*
- label, token;

Tipi di dato composti

- array: [n x type]
 - [10 x i32]
 - [10 x [20 x i8]]
- vector: <n x type>
- struct: {typelist}
 - i32, i8*, float, [5 x i32]
 - <{i16, i8, i32}>; packed
- funzioni: ret-type (typelist)
 - i32 (int32, i1)
 - i32 (int8*, ...)
- metadata, ...

Target Triple e Data Layout

Sono configurazioni fondamentali che definiscono le caratteristiche del sistema target su cui il codice generato verrà eseguito. Essi guidano il compilatore nella generazione di codice ottimizzato per una specifica piattaforma.

Target Triple

La **target triple** è una stringa che identifica il sistema per cui il codice deve essere compilato. La stringa è composta da tre (o più) componenti separati da trattini (-), che specificano l'architettura, il produttore, il sistema operativo e, optionalmente, l'ambiente. Ecco un esempio: `x86_64-pc-linux-gnu`.

Data Layout

Il **data layout** è una stringa che descrive il formato di memoria e le regole di allineamento per i tipi di dati sulla piattaforma target. Definisce:

- L'endianness (ordine dei byte).
- La dimensione e l'allineamento dei tipi di dato.
- La granularità delle operazioni di memoria.

Vediamo un esempio: `e-m:e-i64:64-f80:128-n8:16:32:64-S128`

Costanti

- booleane (tipo i1): true, false
- intere: 4, 56, -1234 (anche negative)
- floating point: 123.0, 1.5e12
- puntatore: null, @global
- array: [i32 3023, i32 -12, i32 18]
- vector: <i32 3, i32 -12, i32 18, i32 4096>
- struct: {i32 3, float 2.5, i32* @global}
- zeroinitializer
- undef

La struttura di una funzione

Una **funzione** è una sequenza di basic block (BB), quindi una sequenza di istruzioni, solamente l'ultima istruzione del BB (terminatore) cambia il flusso di esecuzione (determina il BB successivo o l'uscita dalla funzione). Il codice è in forma SSA (Static Single Assignment),

per generare i nomi locali al BB (tipi, valori, etichette) si usa un unico contatore (poco leggibile), l'opzione `-fno-discard-value-names` cerca di mantenere i nomi locali (più leggibile).

Istruzioni LLVM

Consiglio la visione del pdf fornito dal prof. Zaffanella per la carrellata di istruzioni fornite da LLVM, il pdf in riferimento è chiamato `21-LLVM-IR.pdf`.

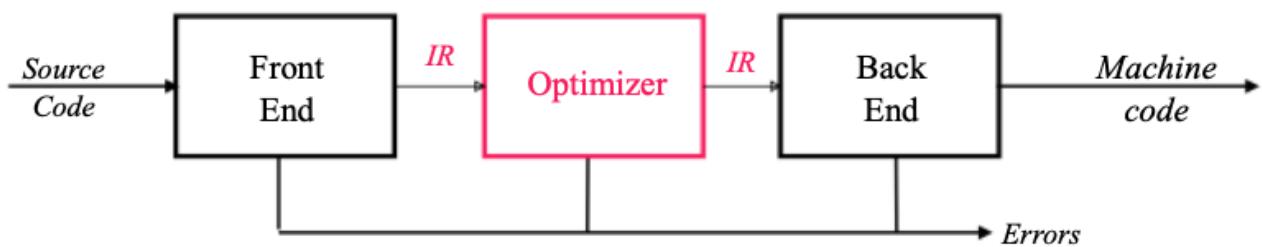
06-middle-end

- [Introduzione all'ottimizzazione di codice](#)
- [L'ottimizzatore](#)
 - [Le regole per l'ottimizzazione](#)
 - [Eliminazione della ridondanza: esempio](#)
 - [Value Numbering](#)
 - [Semplici estensioni alla numerazione dei valori](#)
 - [Ottimizzazione di Scope](#)
 - [Spazio dei nomi SSA](#)
 - [Loop Unrolling - Tecnica Regionale](#)
 - [Trovare variabili non inizializzate - Tecnica globale](#)

Introduzione all'ottimizzazione di codice

Tra il front-end e il back-end vi si trova appunto il middle-end, anche chiamato ottimizzatore. Il suo compito è: analizzare IR e riscriverlo o trasformarlo, questo per:

- l'obiettivo primario, cioè ridurre il tempo di esecuzione del codice compilato, migliorando lo spazio, il consumo energetico, ...;
- preservando il "significato" del codice;



L'ottimizzatore

L'ottimizzazione del codice può avvenire in più passi, le trasformazioni tipiche che ad ogni passo possono essere applicate sono:

- Scoprire che un valore rimane costante in tutto il flusso del programma;
- Spostare un'esecuzione in una parte di codice che viene eseguita meno frequentemente: se un pezzo di codice risulta invariato, che valuta sempre lo stesso valore, lo muovo in una parte del programma che viene eseguita meno frequentemente;
- Specializzare alcuni calcoli in base al contesto: inline delle funzioni;

- Scoprire che qualche calcolo è ridondante e rimuoverlo;
- Rimuovere codice che non viene mai eseguito o che non è raggiungibile;
- Codificare un idioma è una forma particolarmente efficiente;

Le regole per l'ottimizzazione

Il compilatore può implementare una procedura in molti modi, il compito dell'ottimizzatore è quello di trovare un'implementazione che sia "migliore": velocità, peso, spazio, ...

Per soddisfare questo:

- L'ottimizzatore analizza il codice per ricavare informazioni sul comportamento in fase di esecuzione;
- Utilizza tale conoscenza nel tentativo di migliorare il codice;

Eliminazione della ridondanza: esempio

Una espressione del tipo $x + y$ è ridondante se e solo se, lungo ogni percorso dall'ingresso della procedura, è stato valutato, e le sue sotto-espressioni costituenti (x e y) non sono state ridefinite.

Se il compilatore può provare che l'espressione è ridondante, può conservare i risultati delle valutazioni precedenti e può sostituire la valutazione corrente con un riferimento.

Due sono i problemi:

- Dimostrare che $x + y$ è ridondante o disponibile;
- Riscrivere il codice per eliminare la valutazione ridondante;

Una tecnica per realizzare entrambi è chiamata **numerazione dei valori**.

Value Numbering

La **numerazione dei valori** è una tecnica per risolvere entrambi i problemi. Consiste nel:

- Assegnare un identificatore univoco $V(n)$ a ogni valore calcolato, rappresentando ciascuna espressione con il suo risultato.
- Quando un'espressione $x + y$ viene calcolata, il risultato viene associato a un identificatore, $V(x + y) = V(j)$, se l'espressione compare nuovamente, il compilatore verifica:
 - Se l'identificatore, $V(j)$, è ancora valido (cioè x e y non sono cambiati).
 - In caso positivo, l'identificatore può essere riutilizzato.

Original code	Con VN	Riscrittura
$a \leftarrow x + y$	$a_3 \leftarrow x_1 + y_2$	$a_3 \leftarrow x_1 + y_2$
$b \leftarrow x + y$	$b_3 \leftarrow x_1 + y_2$	$b_3 \leftarrow a_3$
$a \leftarrow 17$	$a_4 \leftarrow 17$	$a_4 \leftarrow 17$ <i>e' già stato ridefinito</i>
$c \leftarrow x + y$	$c_3 \leftarrow x_1 + y_2$	$c_3 \leftarrow a_3$ <i>in quel momento, non vale più come prima</i>

Ma dando un valore univoco ad ogni variabile funziona:

Original Code	Con VN	Riscrittura
$a_0 \leftarrow x_0 + y_0$	$a_{03} \leftarrow x_{01} + y_{02}$	$a_{03} \leftarrow x_{01} + y_{02}$
$b_0 \leftarrow x_0 + y_0$	$b_{03} \leftarrow x_{01} + y_{02}$	$b_{03} \leftarrow a_{03}$
$a_1 \leftarrow 17$	$a_{14} \leftarrow 17$	$a_{14} \leftarrow 17$
$c_0 \leftarrow x_0 + y_0$	$c_{03} \leftarrow x_{01} + y_{02}$	$c_{03} \leftarrow a_{03}$

Semplici estensioni alla numerazione dei valori

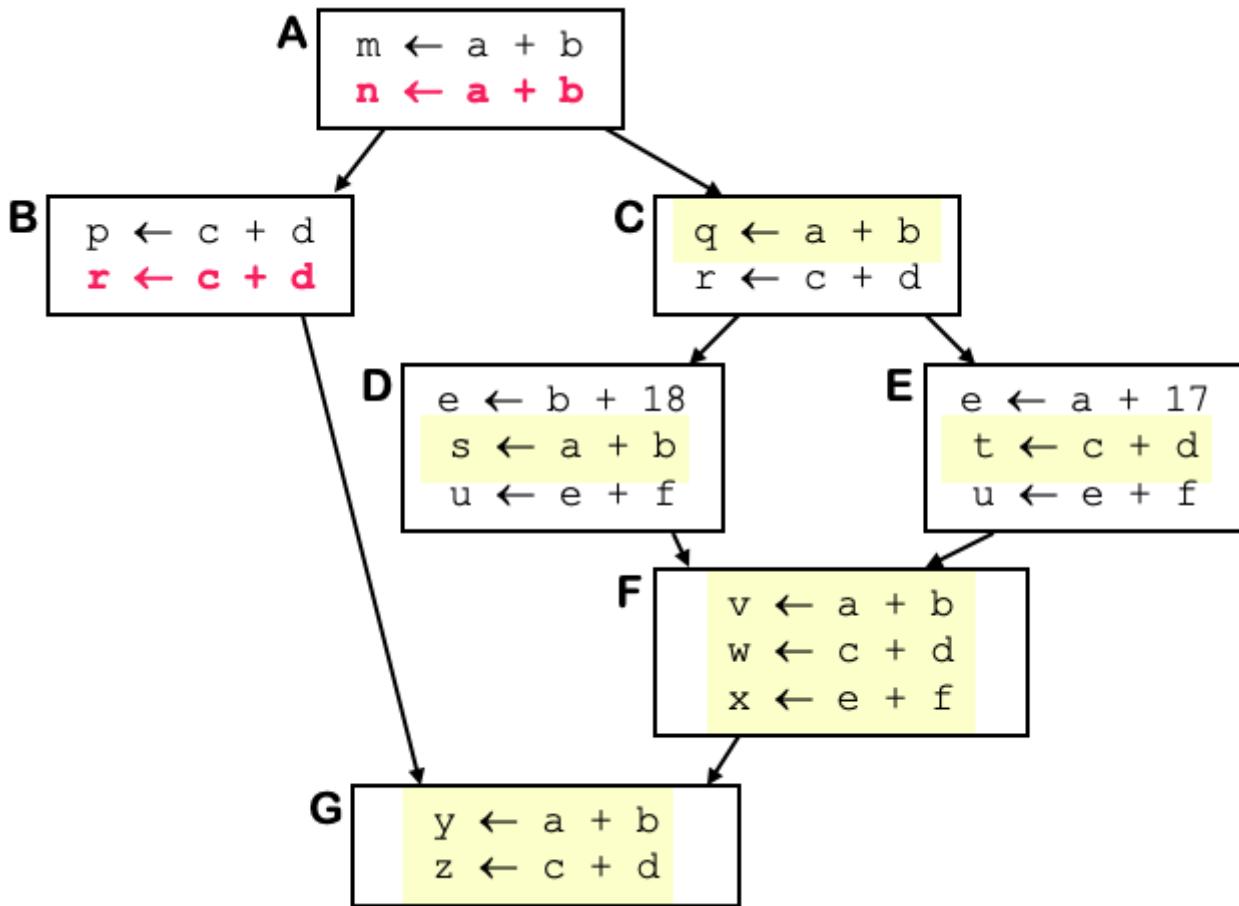
Il **folding delle costanti** consiste nel riconoscere e valutare espressioni composte esclusivamente da valori costanti direttamente in fase di compilazione, evitando calcoli a runtime.

- Aggiungere un bit che registra quando un valore è costante;
- Valutare i valori costanti in fase di compilazione;
- Sostituire con load immediate o operandi immediati: se il risultato è una costante, può essere caricato direttamente in un registro o utilizzato come operando immediato;

Le **identità algebriche** permettono al compilatore di semplificare ulteriormente le espressioni utilizzando proprietà matematiche.

- Deve controllare (molti) casi speciali: il compilatore deve verificare specifiche configurazioni di espressioni che possono essere semplificate tramite identità algebriche;
- Sostituisce il risultato con l'input VN: il compilatore utilizza la numerazione dei valori per sostituire il risultato di un'operazione con uno dei suoi input se l'operazione è semplificabile;
- Costruisce un albero decisionale sull'operazione: per operazioni più complesse, il compilatore costruisce un albero decisionale che valuta le regole algebriche applicabili;

Il problema principale del Value Numbering è che non si riescono a propagare le variabili di un Basic Block in un altro Basic Block. Vediamo un esempio:



Notiamo che nel blocco A c'è un'opportunità di ottimizzazione come nel blocco B, ma vediamo che l'ottimizzazione che dovrebbe essere fatta nel blocco A, potrei applicarla anche nel blocco C e non solo.

Ottimizzazione di Scope

L'**ottimizzazione di scope** si riferisce al livello di granularità su cui un compilatore esegue l'analisi e la trasformazione del codice per migliorare le prestazioni o ridurre l'occupazione di memoria. "Scope" qui identifica la regione del codice considerata durante l'ottimizzazione. A seconda dell'estensione dello scope, cambiano le strategie, le tecniche e le sfide dell'ottimizzazione.

Ottimizzazione Locale

- Opera interamente all'interno di un singolo basic block
- Le proprietà del blocco portano a forti ottimizzazioni

Ottimizzazione Regionale

- Operare su una regione nel CFG che contiene più blocchi collegati

Ottimizzazione dell'intera procedura

- Analizza e ottimizza l'intero grafo di controllo del flusso (**CFG**) di una singola procedura o funzione.

Ottimizzazione dell'intero programma

- Operare su alcuni o tutti i grafi delle chiamate (procedure multiple)
- Deve fare i conti con chiamata/ritorno e associazione dei parametri

Spazio dei nomi SSA

In SSA, ogni variabile (o nome) è assegnata una sola volta nel codice, garantendo unicità e rendendo più semplice tracciare l'origine e l'uso delle variabili.

Due principi fondamentali:

1. Ogni nome è definito da un'unica operazione: una variabile viene assegnata una sola volta e non cambia mai valore successivamente.
2. Ogni operando fa riferimento a un'unica definizione: quando si usa una variabile, il riferimento punta direttamente alla sua unica definizione, non ci sono ambiguità su quale valore di una variabile venga utilizzato.

Nel codice reale, percorsi di esecuzione multipli possono convergere in un singolo punto, e le variabili possono avere valori diversi a seconda del percorso eseguito, nel codice SSA, questo crea un problema, poiché dobbiamo rispettare i due principi.

Soluzione: Funzioni ϕ

Le **funzioni ϕ** sono un costrutto speciale che risolve il problema dei punti di unione, una funzione ϕ seleziona il valore corretto di una variabile in base al percorso eseguito.

```
if (cond) {  
    x1 = 1;  
} else {  
    x2 = 2;  
}  
x3 = φ(x1, x2); // x3 assume il valore di x1 o x2 a seconda del percorso
```

- La funzione ϕ viene inserita automaticamente dal compilatore nei punti di unione (ad esempio, dopo un if o un ciclo).
- Ogni ramo del codice contribuisce con una variabile definita nel suo percorso.

Loop Unrolling - Tecnica Regionale

Loop Unrolling (srotolamento del ciclo) è una tecnica di ottimizzazione utilizzata nei compilatori per migliorare l'efficienza dei loop, riducendo il sovraccarico legato ai test condizionali e ai salti, e migliorando le opportunità di ottimizzazione locale.

Le applicazioni trascorrono molto tempo nell'esecuzione dei loop. Ogni iterazione di un ciclo introduce un **sovraffaccarico** dovuto a:

1. **Test condizionali** (per verificare se il ciclo deve continuare).
2. **Rami** (salti al punto iniziale del ciclo per la prossima iterazione).
3. **Incrementi o modifiche** (variabili di controllo del ciclo).

Lo srotolamento riduce questo sovraccarico eliminando alcune iterazioni del ciclo e duplicando il corpo del loop per eseguirlo più volte in una sola iterazione.

Esempio:

```
for (int i = 0; i < 4; i++) {
    A[i] = A[i] + 1;
}
```

Unrolling:

```
A[0] = A[0] + 1;
A[1] = A[1] + 1;
A[2] = A[2] + 1;
A[3] = A[3] + 1;
```

Unrolling parziale

In molti casi, è inefficiente o impossibile srotolare completamente un ciclo, ad esempio quando il numero di iterazioni è molto grande o sconosciuto. In questi casi, si applica uno **srotolamento parziale** duplicando solo alcune iterazioni.

Unrolling con limiti sconosciuti

Se i limiti del ciclo non sono fissi (ad esempio, se dipendono da un input), è necessario aggiungere un **loop di guardia** per gestire i casi rimanenti.

- **Loop di guardia** gestisce le iterazioni rimanenti quando il numero totale di iterazioni non è divisibile per il fattore di srotolamento.
- La maggior parte dei benefici dello srotolamento è preservata.
- Funziona anche per limiti superiori e inferiori arbitrari.

Ottimizzazioni aggiuntive

- **Eliminazione delle copie alla fine del ciclo**: nei loop con **copie finali ridondanti**, lo srotolamento può eliminarle. Questo avviene calcolando il **minimo comune multiplo (LCM)** delle lunghezze del ciclo di copia.
- **Blocchi più lunghi per ottimizzazioni locali**: lo srotolamento crea **blocchi di codice più lunghi**, aumentando le opportunità di ottimizzazione locale (ad esempio, eliminazione di espressioni ridondanti).

Trovare variabili non inizializzate - Tecnica globale

Quando una variabile viene utilizzata prima di essere definita, può portare a errori logici nel programma. Questa situazione può essere individuata attraverso l'**analisi globale del flusso di dati**.

Flusso di dati e variabili non inizializzate:

Una variabile è detta **viva** in un punto del programma se il suo valore potrebbe essere utilizzato successivamente. Se una variabile è viva nel **blocco di ingresso** di una procedura senza essere inizializzata, rappresenta un **potenziale errore logico**.

L'obiettivo del compilatore è individuare e segnalare queste variabili.

Concetti chiave

1. Costruzione del CFG (Control Flow Graph)
2. Calcolo dei set UEVAR e VARKILL:
 - **UEVAR** (Used-Variable): insieme delle variabili usate nel blocco prima di essere ridefinite.
 - esempio: in `x = a + b`, `a` e `b` sono UEVAR se non sono ridefiniti precedentemente nel blocco.
 - **VARKILL**: insieme delle variabili definite (o "uccise") nel blocco.
 - esempio: in `x = a + b`, `x` appartiene a VARKILL.
3. LIVEOUT: LIVEOUT(`n`) è l'insieme delle variabili vive alla fine del blocco `n`, si calcola iterativamente utilizzando le definizioni di UEVAR e VARKILL.

Esempio:

```
x = a + b;
y = x * 2;
if (y > 10) {
    z = y - 5;
} else {
    w = y + 5;
}
```

- CFG:
 - Nodo 1: `x = a + b`
 - Nodo 2: `y = x * 2`
 - Nodo 3: `z = y - 5` (ramo `if`)
 - Nodo 4: `w = y + 5` (ramo `else`)
- UEVAR e VARKILL:

- Nodo 1: UEVAR = {a, b}, VARKILL = {x}
 - Nodo 2: UEVAR = {x}, VARKILL = {y}
 - Nodo 3: UEVAR = {y}, VARKILL = {z}
 - Nodo 4: UEVAR = {y}, VARKILL = {w}
- LIVEOUT calcolato iterativamente:
 - Nodo 3: LIVEOUT(3) = {} (nessun successore, z non viene usato dopo).
 - Nodo 4: LIVEOUT(4) = {} (nessun successore, w non viene usato dopo).
 - Nodo 2: LIVEOUT(2) = UEVAR(3) \cup UEVAR(4) = {y} .
 - Nodo 1: LIVEOUT(1) = UEVAR(2) \cup (LIVEOUT(2) - VARKILL(2)) = {x, y} .

Dopo aver calcolato LIVEOUT , controllare:

- Se una variabile è in LIVEOUT del blocco iniziale (LIVEOUT(n0)), ma non è mai stata inizializzata, il compilatore segnala un errore.
- Ad esempio, se a o b non sono inizializzati nel programma sopra, il compilatore lo segnala.

Ottimizzazioni del codice tramite LIVEOUT

1. Eliminazione delle istruzioni non necessarie;
2. Ordinare i calcoli per migliorare la convergenza
3. Calcolo RPO: PostOrder, ordine in cui i nodi sono visitati terminando ogni ramo del CFG.