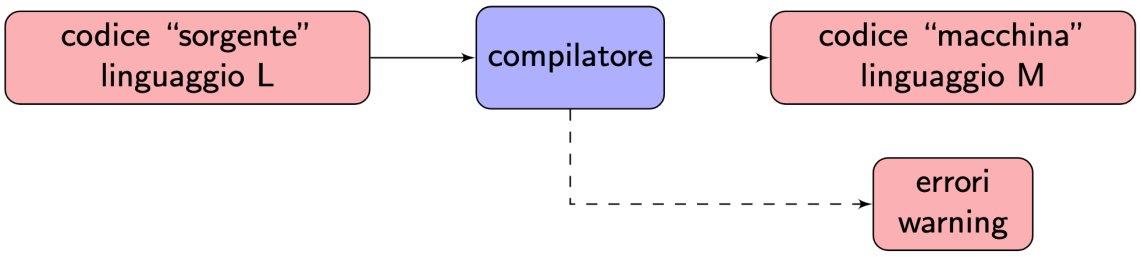
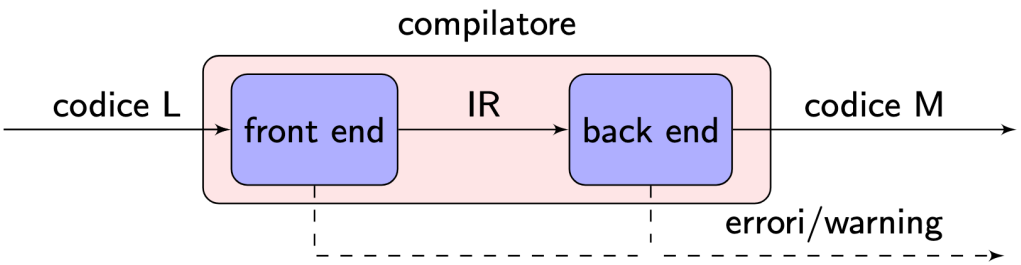
2. La struttura di un compilatore



Un **compilatore** è in grado di:

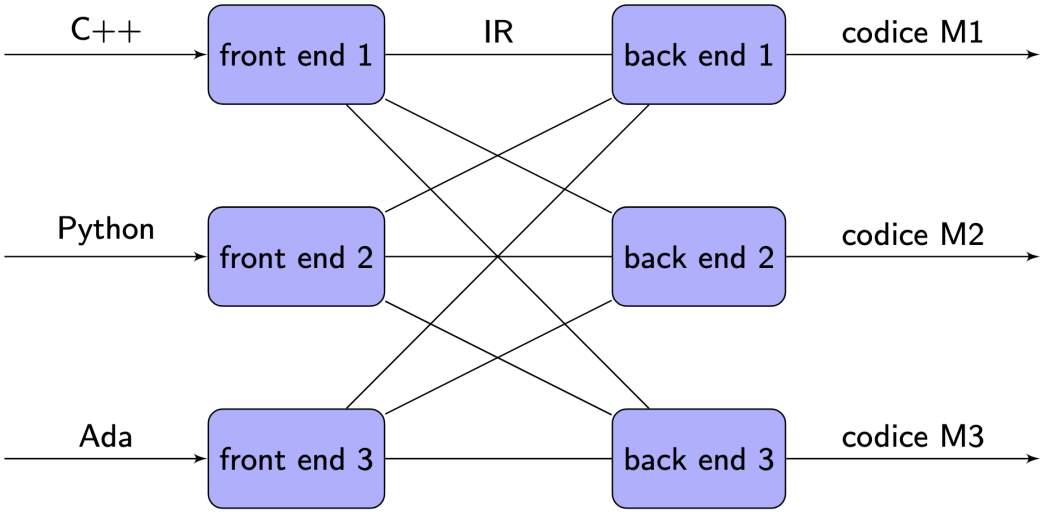
- **Riconoscere la validità o l'invalidità del programma**. Il compilatore verifica che il programma rispetti le regole sintattiche e semantiche del linguaggio di programmazione. In caso di errori o potenziali problemi, genera messaggi di errore e warning chiari per aiutare il programmatore a correggerli.
- **Generare codice corretto**. Dopo l'analisi, il compilatore deve produrre un codice macchina o bytecode che sia efficiente e semanticamente equivalente al codice sorgente. Il codice generato deve essere eseguibile correttamente sull'architettura target.
- **Gestire le risorse**. Il compilatore è responsabile dell'allocazione e deallocazione della memoria durante l'esecuzione del programma. Deve assegnare le risorse in modo efficiente, minimizzando gli sprechi e prevenendo problemi come la frammentazione della memoria.
- **Interagire con il sistema operativo**. Il compilatore collabora con alcune componenti del sistema operativo, come il linker dinamico, per risolvere riferimenti esterni (ad esempio, librerie) e garantire che il programma funzioni correttamente nell'ambiente di esecuzione target.



Il lavoro di un **compilatore** è tradizionalmente suddiviso in due parti, ciascuna con ruoli specifici nella trasformazione del codice sorgente:

- **Front end**. Questa parte riceve in input il codice sorgente scritto in un linguaggio di alto livello L. Il suo compito è analizzare il codice, verificarne la correttezza sintattica e semantica, e successivamente generare una **rappresentazione intermedia (IR)**. La rappresentazione intermedia è una forma astratta e indipendente dalla macchina, ancora strettamente legata al linguaggio sorgente L. Il front end è anche responsabile di individuare e segnalare eventuali **errori e warning**, come errori di sintassi o incongruenze semantiche.
- **Back end**. Questa parte prende in input la rappresentazione intermedia IR e la trasforma in **codice eseguibile** o **bytecode** per una macchina target specifica, utilizzando il linguaggio M. Il back end è responsabile dell'ottimizzazione del codice e della generazione del codice macchina in modo efficiente, tenendo conto delle risorse della piattaforma hardware e del sistema operativo.

Questa struttura a due livelli è vantaggiosa perché consente la **combinazione modulare** di front end e back end diversi. Ciò significa che un compilatore può essere facilmente adattato per supportare nuovi linguaggi di programmazione o nuove piattaforme hardware, semplicemente combinando un front end per il linguaggio L con un back end per la macchina target M. Questo approccio riduce notevolmente la complessità nella creazione di nuovi compilatori.



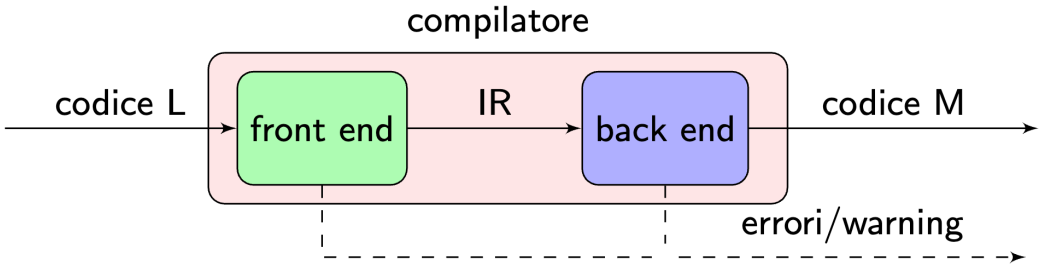
Nota. Questa demarcazione così marcata tra front end e back end è però perlopiù teorica. Nella realtà, infatti, non è sempre possibile una divisione così netta dei compiti

Rappresentazione Intermedia

La **rappresentazione intermedia** (IR) deve essere sufficientemente espressiva da catturare tutte le informazioni raccolte dal compilatore. Esistono diverse tipologie di IR, ciascuna con caratteristiche specifiche:

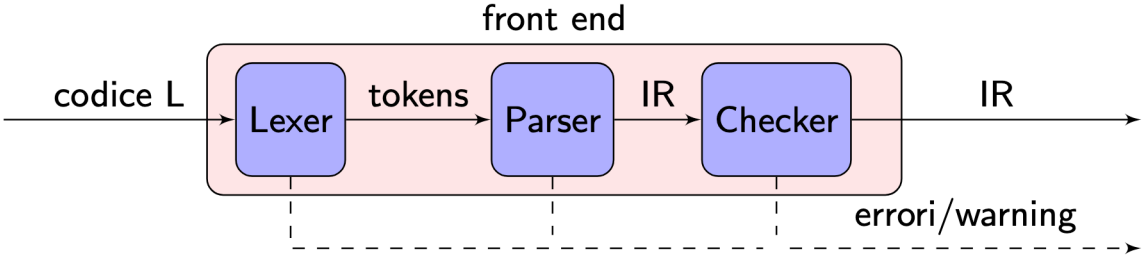
- **Strutturali**. Utilizzano strutture come alberi, grafi e DAG (grafi aciclici diretti) per rappresentare la struttura del programma
- **Lineari**. Rappresentano il programma come una sequenza lineare di istruzioni. Esempi includono il **codice a tre indirizzi** (3-address code) e il **codice per macchine a stack**.
- **Ibridi**. Combinano elementi sia strutturali che lineari per sfruttare i vantaggi di entrambi gli approcci. Il **Control Flow Graph** (CFG) è un esempio prominente, dove il programma è suddiviso in **Basic Blocks** (BB) collegati tra loro

Il front end del compilatore



Come detto, la parte **front end** di un compilatore ha il compito di analizzare il codice sorgente, verificarne la correttezza e trasformarlo in una rappresentazione intermedia (IR) adatta per le fasi successive del processo di compilazione. La parte front end del compilatore deve quindi essere in grado di:

- **Riconoscere i programmi validi.** Il front-end deve identificare se il codice sorgente rispetta le regole sintattiche e semantiche del linguaggio di programmazione. Questo implica distinguere i programmi corretti da quelli contenenti errori.
- **Segnalare errori e warning leggibili.** In caso di errori, il compilatore deve fornire messaggi di errore e avvertimenti chiari e facilmente comprensibili, indicando il tipo di problema e il contesto in cui si verifica, per facilitare la correzione del codice da parte del programmatore.
- **Produrre codice IR e strutture ausiliarie.** Alla fine dell'analisi, il front end genera una rappresentazione intermedia del programma (IR) che servirà come base per le ottimizzazioni e la generazione del codice finale. Inoltre, crea strutture dati ausiliarie, come la tabella dei simboli, contenenti informazioni sulle variabili, funzioni e altri elementi del programma.

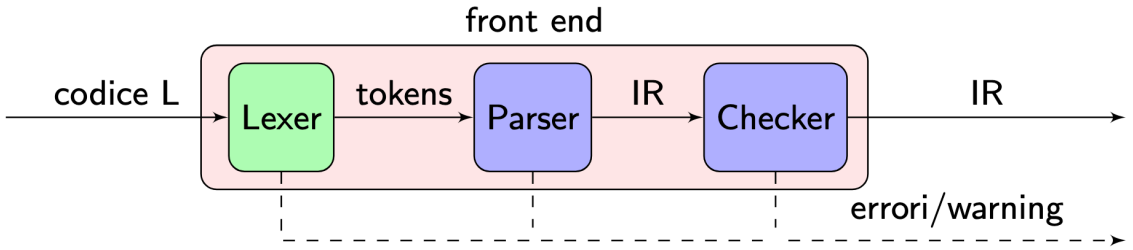


Il front end è suddiviso (tradizionalmente) in tre componenti:

- **Lexer (analisi lessicale).** Questa fase rappresenta la lettura del flusso di caratteri del codice sorgente per suddividerlo in unità sintattiche fondamentali, dette *token*. Il lexer identifica elementi come parole chiave, identificatori, numeri, operatori e simboli speciali, eliminando al contempo spazi bianchi e commenti.
- **Parser (analisi sintattica).** Il parser prende i token generati dal lexer e li organizza secondo la grammatica del linguaggio di programmazione, costruendo strutture dati come l'**albero sintattico astratto** (AST, abstract syntax tree). Questa fase verifica che la sequenza di token rispetti le regole sintattiche definite (grammatiche libere da contesto) e individua errori come parentesi non chiuse o sintassi invalide.
- **Checker (analisi semantica statica).** Durante l'analisi semantica, il compilatore verifica che il codice sia non solo sintatticamente corretto, ma anche semanticamente valido. Questa fase dipende dal contesto e include controlli come la verifica dei tipi di dati, l'esistenza di variabili e funzioni, l'accesso corretto agli scope e il rispetto delle regole del linguaggio (ad esempio, l'assegnazione di un intero a una variabile dichiarata come stringa). Il checker può anche costruire la **tabella dei simboli**, che traccia le dichiarazioni e le definizioni all'interno del codice.

Attenzione. Poiché un compilatore deve terminare in un tempo finito, il checker può verificare unicamente quelle proprietà del programma che sono **decidibili** (è impossibile individuare errori come la non terminazione di un ciclo, che l'indice di un array sia valido per quell'array o che la somma di due variabili non generi un overflow)

Il Lexer



L'**analisi lessicale** è la prima fase del processo di compilazione, in cui il **lexer** trasforma una sequenza di caratteri grezzi (input) in una sequenza di **token** (output). Un token rappresenta l'unità sintattica minima significativa del linguaggio, e viene utilizzato nelle fasi successive del processo di compilazione.

Un **token** è formalmente rappresentato da una coppia $\langle \text{part of speech}, \text{lexeme} \rangle$, dove:

- Il **part of speech** è la **categoria lessicale** che identifica la tipologia del token (ad esempio, `KWD` per le parole chiave, `INT` per i numeri interi, `STR` per le stringhe).

- Il **lexeme** (o *lessema*) è la stringa di caratteri effettivamente letta dal codice sorgente, ad esempio `while`, `42`, o `3.1415`

e.g. ⟨ `KWD`, `while` ⟩, ⟨ `INT`, `42` ⟩, ⟨ `FLOAT`, `3.1415` ⟩

Specifica (cosa deve fare?)

Per definire in modo rigoroso quali token sono validi, si utilizzano le **espressioni regolari (RE)**. Le espressioni regolari forniscono un modo compatto e formale per descrivere le sequenze di caratteri che corrispondono a un token specifico. Questo metodo:

- Permette di descrivere in maniera formale i token validi per un linguaggio di programmazione.
- È abbastanza leggibile e intuitivo da essere utilizzato da un progettista umano per definire la specifica dei token.

Implementazione (come deve farlo?)

L'efficienza è un fattore critico per il **lexer**, poiché ogni carattere del codice sorgente deve essere analizzato rapidamente. Il lexer è anche chiamato **scanner** poiché "scansiona" il flusso di caratteri per individuare i token. Per implementare un lexer efficiente, viene utilizzato un **DFSA (automa a stati finiti deterministico)**

Nota. Un DFSA è una macchina che, leggendo il flusso di caratteri, passa attraverso una serie di stati per identificare correttamente i token validi. L'automa termina quando raggiunge uno stato accettante, producendo il token corrispondente.

In molti casi, l'implementazione viene **generata automaticamente** a partire dalla specifica delle espressioni regolari

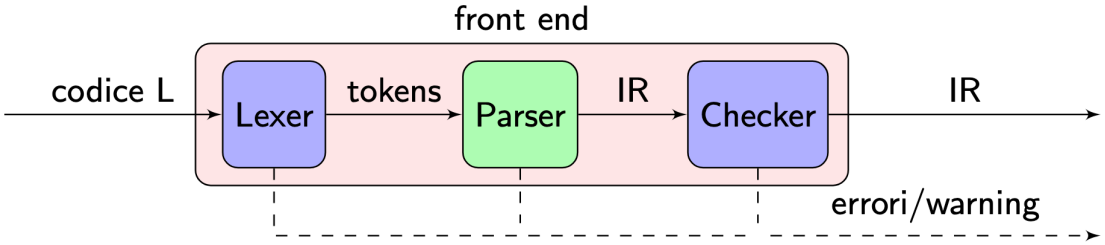
---

Esempio. RE per identificatori del linguaggio

```
DIGIT = [0–9]
LETTER = [a–zA–Z] | [_]
ID = LETTER ( LETTER | DIGIT )*
```

Attenzione. I caratteri 0-9, a-z, A-Z e _ sono caratteri del linguaggio, mentre LETTER, DIGIT, ID, (,), = e | rappresentano la *meta-sintassi*. La meta-sintassi ammette varie forme di abbreviazione
(iterazione positiva, complemento, . . .)

Il Parser



Il **parser** è la componente del compilatore che esegue l'**analisi sintattica**. Il suo compito è prendere in input una sequenza di **token** generati dal lexer e costruire una rappresentazione strutturale che catturi la sintassi del programma. Il risultato di questa fase è una rappresentazione intermedia (IR) che riflette la struttura del programma in modo adeguato per le fasi successive del compilatore.

La rappresentazione intermedia generalmente utilizzata è l'**AST (Abstract Syntax Tree)**, che rappresenta la struttura logica del programma. Un'altra possibile rappresentazione è il **parse tree** (albero di derivazione o **concrete syntax tree**), che però viene usato raramente nei compilatori moderni, poiché include troppi dettagli sintattici non necessari per le fasi successive (l'output generato dal parser deve essere una struttura semplificata, come l'AST, che elimini dettagli superflui e sia ottimizzata per ulteriori analisi e trasformazioni.)

Specifica

Per definire in modo rigoroso la sintassi valida del linguaggio di programmazione, viene utilizzata una **grammatica libera dal contesto** (**CFG**) $G = \langle S, N, T, P \rangle$, dove

- N : simboli non terminali (qui rappresentano le categorie sintattiche)
- T : simboli terminali (qui rappresentano le categorie lessicali dei token)
- $S \in N$: simbolo iniziale
- $P \subseteq N \times (N \cup T)^*)$: produzioni della grammatica

La CFG descrive le regole di produzione che definiscono come i token possono essere combinati per formare costrutti sintattici validi.

- La **CFG** è adatta a descrivere la sintassi di linguaggi complessi.
- Deve essere abbastanza chiara e leggibile per essere compresa e gestita dai progettisti umani.
- Tuttavia, l'implementazione del parser può affrontare problemi non banali come la gestione del **determinismo**, l'**efficienza** e la risoluzione di **ambiguità** nella grammatica.

Implementazione

Il parser può essere implementato utilizzando diversi approcci. Il **riconoscitore** utilizzato per analizzare la grammatica è un **PDA**, ossia un automa a pila non deterministico.

- L'implementazione può essere **codificata direttamente**, spesso utilizzando tecniche come la **ricorsione** e il **backtracking** per risolvere le regole della grammatica.
- In molti casi, il parser viene **generato automaticamente** partendo dalla grammatica
- Esistono diverse **tipologie di generatori di parser**, ciascuna ottimizzata per specifiche sottoclassi di grammatiche (ad esempio, LL e LR), con diversi compromessi in termini di potenza espressiva e complessità computazionale.

Esempio. Derivazione di $x + 2 - y$

Data la seguente grammatica per le espressioni additive

$$
\begin{aligned}
&S = \mathit{Expr} \\
&N = \{\mathit{Expr}, \mathit{Op}, \mathit{Term}\} \\
&T = \{\mathrm{id}, \mathrm{num}, +, -\} \\
&P = \left\{ \begin{array}{l} S \to \mathit{Expr}, \\ \mathit{Expr} \to \mathit{Term} \mid \mathit{Expr}\ \ \mathit{Op}\ \ \mathit{Term}, \\ \mathit{Op} \to + \mid -, \\ \mathit{Term} \to \mathrm{id} \mid \mathrm{num} \end{array} \right\}
\end{aligned}
$$

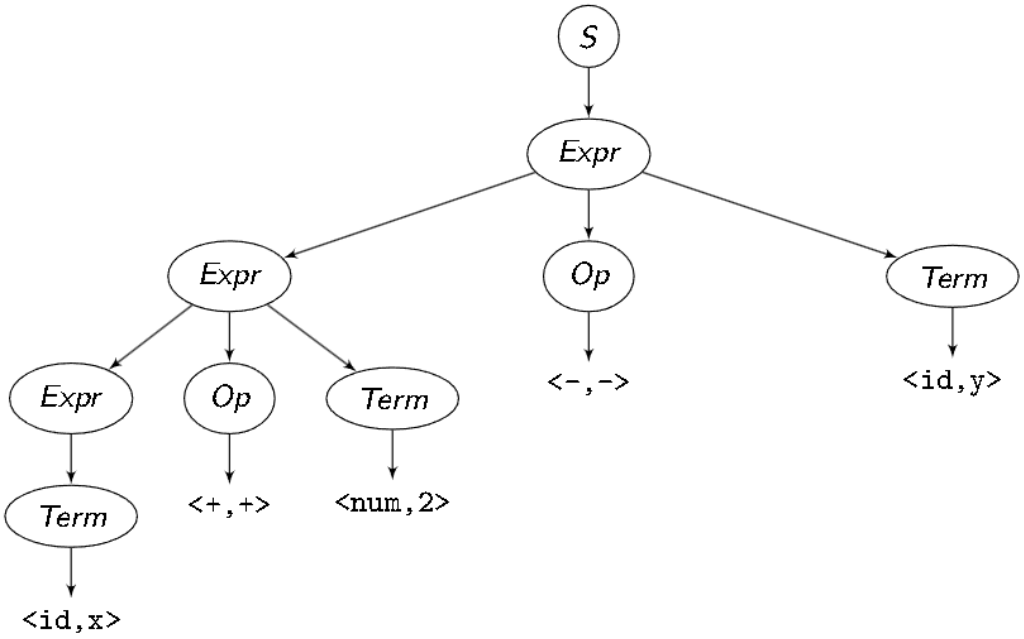
che introducendo le seguenti abbreviazioni sintattiche

$$
\begin{aligned}
&(N \to \gamma) \equiv (N, \gamma) \in P \\
&(N \to \gamma_1 \mid \gamma_2) \equiv (N \to \gamma_1, N \to \gamma_2)
\end{aligned}
$$

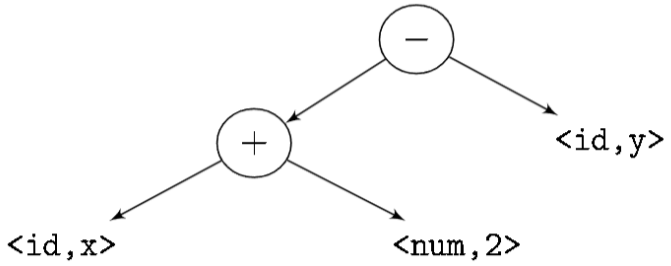
può essere riscritta come

$$
\begin{array}{rcl}
S & \to & \mathit{Expr} \\
\mathit{Expr} & \to & \mathit{Term} \\
& \mid & \mathit{Expr}\ \ \mathit{Op}\ \ \mathit{Term} \\
\mathit{Op} & \to & + \\
& \mid & - \\
\mathit{Term} & \to & \mathrm{id} \\
& \mid & \mathrm{num}
\end{array}
$$

Il parsing di $x + 2 - y$ genererà il seguente parse tree (o concrete syntax tree)

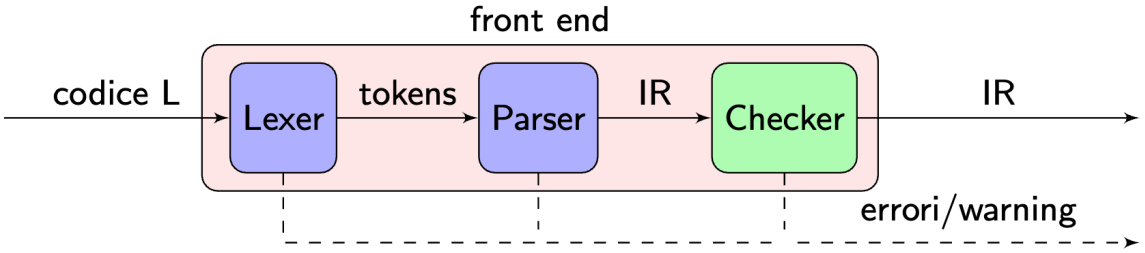


L'abstract syntax tree che può essere utilizzato per semplificare questa derivazione è invece il seguente



Il Checker

Il **checker** è la componente del compilatore responsabile dell'**analisi di semantica statica**, nota anche come **CSA (Context-Sensitive Analysis)**.



Questa fase verifica la correttezza semantica del codice sorgente, assicurandosi che le operazioni siano sensate nel contesto in cui si trovano e rispettino le regole del linguaggio di programmazione.

In particolare, il checker prende in input l'AST grezzo generato dal parser, che rappresenta la struttura sintattica del programma senza tenere conto delle regole semantiche dipendenti dal contesto, e produce in output un **AST arricchito** con informazioni contestuali.

Durante questa fase, l'albero viene annotato con dettagli come:

- Tipi di dato di variabili ed espressioni.
- Conversioni implicite tra tipi di dati, ove necessario.
- Risoluzione di funzionalità polimorfiche come l'**overloading** delle funzioni, in cui la scelta della funzione corretta dipende dal tipo degli argomenti.
- Verifica di correttezza nella definizione di variabili, scope e accessi.

Nota. Spesso il **checker** e il **parser** sono strettamente integrati, al punto che l'AST "grezzo" non viene effettivamente prodotto separatamente. In molti compilatori, invece di creare prima l'AST grezzo e poi arricchirlo, si costruisce direttamente un **AST arricchito** a partire dal **parse tree**, combinando analisi sintattica e semantica in un'unica fase. Questo approccio ottimizza il processo e riduce la duplicazione di lavoro.

Specifica

Per definire in modo rigoroso quali sono i **programmi validi**, esistono vari approcci, che vanno dal linguaggio naturale alle formalizzazioni rigorose:

- **Linguaggio naturale.** Spesso, la specifica di un linguaggio di programmazione viene descritta in documenti come lo **standard del linguaggio**, la **manualistica** o la **documentazione del compilatore**. Questi documenti sono pensati per essere comprensibili dagli sviluppatori, ma tendono a essere imprecisi o soggetti a interpretazioni diverse.
- **Semantiche formali.** Per una definizione più rigorosa, vengono utilizzate **semantiche formali**, che descrivono il comportamento di un linguaggio attraverso sistemi di regole matematiche. Questi sistemi possono includere regole per la valutazione delle espressioni, il controllo dei tipi e la gestione del flusso di controllo. Tuttavia, le semantiche formali sono spesso complesse e non sempre facilmente comprensibili da progettisti o sviluppatori.

Implementazione

In questo caso, un fattore cruciale è la **correttezza**. Il compilatore deve implementare fedelmente le regole definite nella specifica del linguaggio, sia essa in linguaggio naturale o in forma formale.

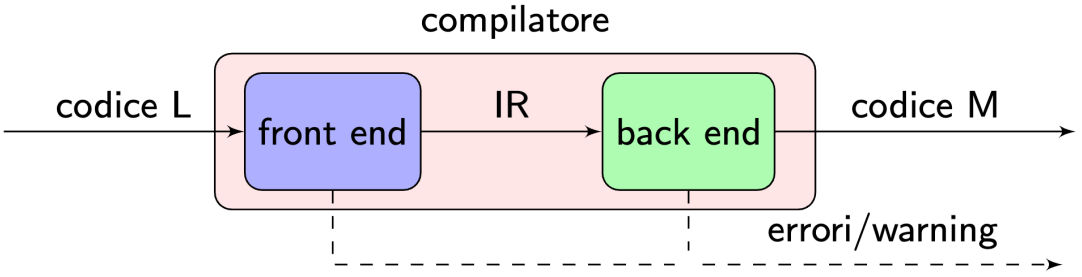
- **In passato.** Si utilizzavano **grammatiche arricchite** da **attributi calcolati**, in cui gli attributi venivano associati ai nodi della grammatica per rappresentare informazioni come i tipi di dati o la posizione nel codice sorgente. Questi attributi venivano calcolati durante il parsing e l'analisi semantica, in modo da garantire che il programma rispettasse le regole.

- **Oggi.** Più frequentemente si utilizzano approcci come la **Syntax Directed Translation (SDT)**, che combina la sintassi del linguaggio con la traduzione automatica. L'SDT consente di associare alle regole grammaticali specifici **algoritmi di visita** che permettono di eseguire traduzioni o verifiche semantiche durante l'analisi sintattica. Questo approccio è più flessibile e consente di integrare facilmente la semantica del linguaggio nel processo di parsing.

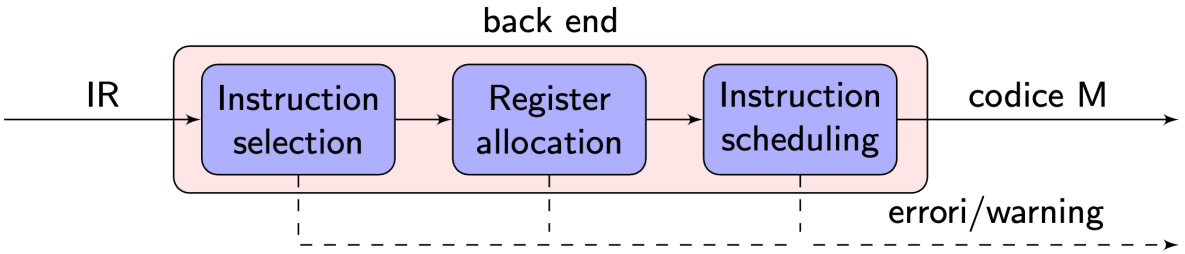
Esempio. Il dump di un AST prodotto da clang

```
$ clang -Xclang -ast-dump -fsyntax-only a.c
TranslationUnitDecl 0x829008 <<invalid sloc>> <invalid sloc>
[ ... omissis ...]
‘-FunctionDecl 0x887de0 <a.c:1:1, line:3:1> line:1:5 main ’int (int, char **)’
  |-ParmVarDecl 0x887c50 <col:10, col:14> col:14 used argc ’int’
  |-ParmVarDecl 0x887d00 <col:20, col:27> col:27 argv ’char **’
  ‘-CompoundStmt 0x887f60 <col:33, line:3:1>
    ‘-ReturnStmt 0x887f50 <line:2:3, col:18>
      ‘-BinaryOperator 0x887f30 <col:10, col:18> ’int’ ’==’
        |-ImplicitCastExpr 0x887f18 <col:10> ’int’ <LValueToRValue>
        | ‘-DeclRefExpr 0x887ed8 <col:10> ’int’ lvalue ParmVar 0x887c50 ’argc’
        ‘-IntegerLiteral 0x887ef8 <col:18> ’int’ 1
```

Il back end del compilatore



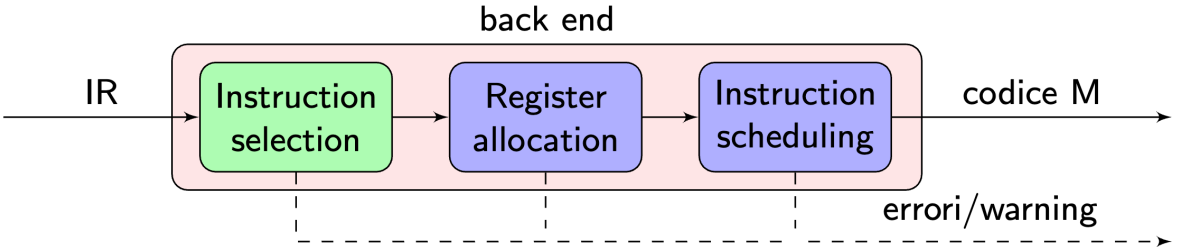
La parte **back end** di un compilatore si occupa della traduzione dal linguaggio intermedio (IR) al linguaggio macchina (M). In questo processo, seleziona le istruzioni necessarie per implementare le varie operazioni, decide quali valori devono essere conservati nei registri e assicura il rispetto delle interfacce di sistema



Il lavoro del **back end** è tradizionalmente suddiviso in tre parti: la selezione delle istruzioni, l'allocazione dei registri e lo scheduling delle istruzioni.

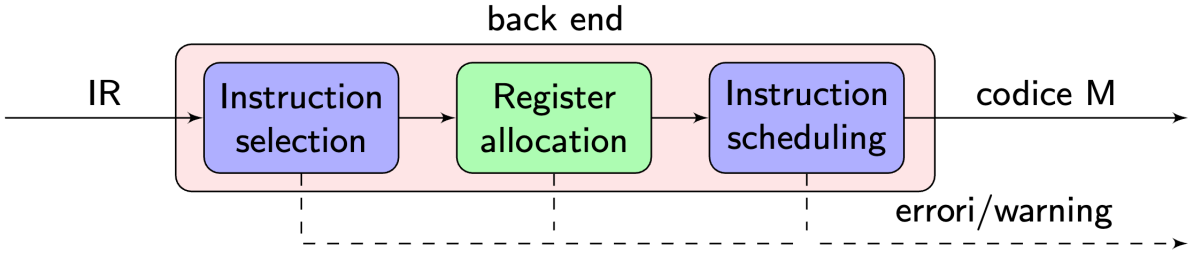
Durante questa fase, il compilatore si trova ad affrontare problemi intrinsecamente NP-completi, il che rende necessario l'uso di tecniche euristiche per trovare soluzioni efficaci.

Instruction selection



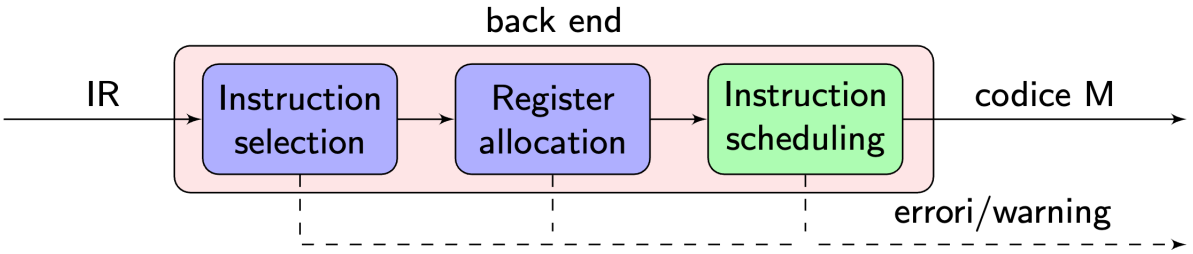
La produzione di codice deve essere sia veloce in termini di tempo che compatta in termini di spazio, sfruttando al massimo le caratteristiche specifiche della macchina target M. Questo processo è spesso affrontato come un problema di pattern matching, in cui si ricercano soluzioni ottimali a livello locale tramite metodi di approssimazione.

Register allocation



La gestione di un insieme finito di risorse, come i registri della CPU, richiede una strategia efficace per allocare questi registri in modo ottimale. Quando i registri disponibili non sono sufficienti, si ricorre allo **spilling**, che consiste nell'uso di istruzioni di **LOAD** e **STORE** per trasferire dati tra i registri e la memoria. L'algoritmo sottostante a questo processo è basato sulla **colorazione di grafi**.

Instruction scheduling



La gestione delle dipendenze a livello hardware mira a evitare le attese attraverso il riordino delle istruzioni per ottimizzare l'esecuzione. Questa fase si concentra anche sull'ottimizzazione dell'uso delle unità funzionali della CPU, sfruttando il parallelismo. Tale processo può influenzare il tempo di vita dei valori nei registri, incidendo di conseguenza sulla loro allocazione.

Esempio. Instruction Scheduling

Consideriamo la seguente rappresentazione intermedia ad alto livello

$$a \leftarrow b \times c + d$$
$$e \leftarrow f + a$$

Una possibile implementazione di questa rappresentazione, scritta in pseudo-codice per la macchina M, potrebbe essere la seguente

```
unit 1
load @b ⇒ r1
load @c ⇒ r2
mult r1, r2 ⇒ r3
load @d ⇒ r4
add r3, r4 ⇒ r5
store r5 ⇒ @a
load @f ⇒ r6
add r5, r6 ⇒ r7
store r7 ⇒ @e
```

Supponendo di avere alcune informazioni sulla latenza delle istruzioni, come mostrato qui

load, store: 2 cicli
altre istruzioni: 1 ciclo

L'esecuzione delle seguenti istruzioni potrebbe risultare problematica in termini di dipendenze e tempi di attesa

unit 1
load @b ⇒ r1
load @c ⇒ r2
mult r1, r2 ⇒ r3
load @d ⇒ r4
add r3, r4 ⇒ r5
store r5 ⇒ @a
load @f ⇒ r6
add r5, r6 ⇒ r7
store r7 ⇒ @e

Una possibile soluzione consiste nell'introdurre il parallelismo, riordinando le istruzioni in modo da sfruttare al meglio le unità funzionali della CPU

unit 1	unit 2
load @b ⇒ r1	load @c ⇒ r2
load @d ⇒ r4	load @f ⇒ r6
mult r1, r2 ⇒ r3	nop
add r3, r4 ⇒ r5	nop
store r5 ⇒ @a	nop
add r5, r6 ⇒ r7	nop
store r7 ⇒ @e	nop

Nota. Nell'immagine, l'uso dell'istruzione *nop* (no operation) rappresenta un'attesa, evitando conflitti e sincronizzando l'esecuzione delle istruzioni.

Successivamente, ci si può concentrare su un utilizzo più efficiente dei registri della CPU per ottimizzare ulteriormente l'esecuzione del codice

unit 1	unit 2
load @b ⇒ r1	load @c ⇒ r2
load @d ⇒ r4	load @f ⇒ r6
mult r1, r2 ⇒ r3	nop
add r3, r4 ⇒ r5	nop
store r5 ⇒ @a	nop
add r5, r6 ⇒ r7	nop
store r7 ⇒ @e	nop

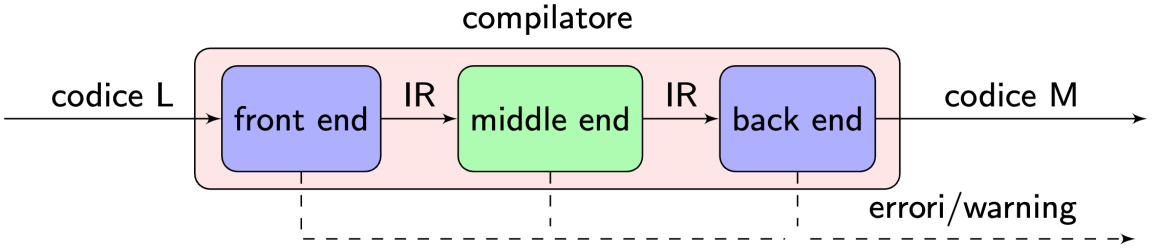
unit 1	unit 2
load @b ⇒ r1	load @c ⇒ r2
load @d ⇒ r4	nop
mult r1, r2 ⇒ r3	nop
add r3, r4 ⇒ r5	load @f ⇒ r6
store r5 ⇒ @a	nop
add r5, r6 ⇒ r7	nop
store r7 ⇒ @e	nop

Il middle end

Riassumendo quanto detto finora, la struttura di un compilatore, tradizionalmente suddivisa in **front end** e **back end**, funziona nel seguente modo:

- **Front end**: Si occupa dell'elaborazione dei linguaggi sorgente. Grazie all'uso di generatori automatici, i problemi legati a questa fase sono ormai considerati “risolti”.
- **Back end**: Si concentra sulle macchine target, richiedendo lo sviluppo di soluzioni specifiche e ad hoc. Questo è un campo “in evoluzione”, poiché nuove architetture come CPU e GPU continuano a emergere.

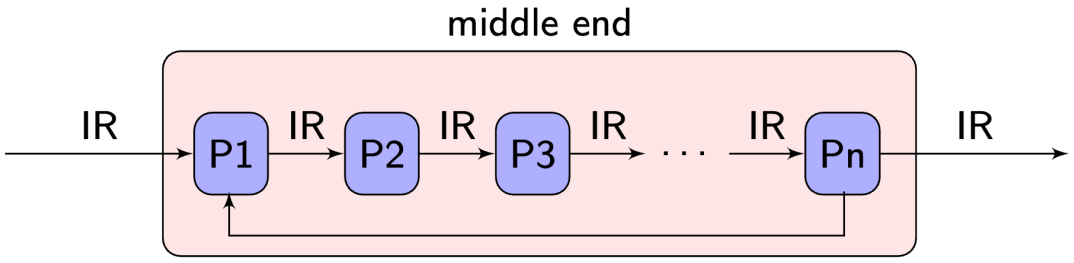
Con il tempo, la struttura tradizionale del compilatore è stata ampliata con l'introduzione del **middle end**. Questa componente intermedia si occupa di ottimizzazioni più generiche che non dipendono né dal linguaggio sorgente né dalla macchina target, migliorando ulteriormente l'efficienza della compilazione



L'analisi e la trasformazione del codice IR mirano a ottimizzare varie metriche, sia classiche che recenti. Tra le metriche classiche ci sono il tempo di esecuzione e lo spazio di memoria, mentre tra quelle più recenti troviamo il consumo energetico e l'uso efficiente delle risorse hardware.

Nota. L'ottimizzazione indipendente dal linguaggio, ovvero che non dipende né dal linguaggio sorgente né dal linguaggio macchina, avviene quasi interamente a questo livello, soprattutto per quanto riguarda i linguaggi imperativi.

Durante questa fase, è fondamentale che le trasformazioni preservino la **semantica** del programma originale. Tuttavia, garantire questa preservazione può risultare complesso, in quanto la semantica non è sempre chiaramente definita. Inoltre, la rappresentazione intermedia in output non deve necessariamente essere la stessa di quella di input, il che offre flessibilità per applicare varie tecniche di ottimizzazione



Il lavoro del **middle end** è tradizionalmente suddiviso in diversi passi, i quali possono essere ripetuti per ottenere il miglior risultato possibile. Tuttavia, è importante assicurarsi che l'insieme delle operazioni eseguite dal compilatore termini entro un tempo finito. Il numero di passi varia in base a diversi fattori, come il livello di ottimizzazione richiesto o il tipo di compilatore utilizzato.

Come detto, ogni passo deve mantenere invariata la semantica del programma per garantire che il comportamento originale non venga alterato. Un singolo passo può essere un'**analisi** o una **trasformazione** dell'IR. L'analisi ha il compito di aggiungere informazioni supplementari al codice, come attributi aggiuntivi, mentre la trasformazione si basa su queste informazioni per applicare ottimizzazioni.

Poiché una trasformazione potrebbe invalidare i risultati dell'analisi precedente, spesso è necessario ripetere l'analisi e, se opportuno, applicare nuove trasformazioni. Inoltre, è possibile che un passo dipenda dai risultati di altri o che, al contrario, possa invalidarne gli effetti, rendendo fondamentale una gestione accurata delle dipendenze tra i vari passi del processo di ottimizzazione.

Ecco alcuni esempi di passi di **analisi** nel middle end

- **Identificazione di valori costanti**: Rilevare variabili o espressioni che mantengono un valore costante durante l'esecuzione del programma, consentendo al compilatore di ottimizzare il codice successivo.
- **Identificazione di codice o valori inutili**: Individuare segmenti di codice o variabili che non influenzano l'esecuzione finale del programma, in modo che possano essere rimossi.
- **Analisi di aliasing**: Determinare quando due nomi (alias) fanno riferimento alla stessa entità in memoria. Ad esempio, in C++, l'aliasing entra in gioco quando un parametro viene passato per riferimento a una funzione. Passando invece un parametro per

valore si evita l'aliasing, consentendo al compilatore di applicare ottimizzazioni più efficaci.

Ecco alcuni esempi di passi di **trasformazione**

- **Propagazione di valori costanti**: Sostituire l’uso di variabili costanti con il loro valore diretto, semplificando così il codice e migliorando le prestazioni.
- **Rimozione di codice inutile**: Eliminare codice che non ha alcun effetto sull'esecuzione del programma, come segmenti ridondanti o codice che non viene mai eseguito. Questo include anche l'eliminazione di codice che deve essere portabile su diverse piattaforme ma che non è necessario nella versione compilata.
- **Inlining di chiamate a funzioni**: Sostituire una chiamata a funzione con il corpo della funzione stessa, riducendo l'overhead delle chiamate e migliorando la velocità di esecuzione.
- **Loop unrolling**: Se il numero di iterazioni di un ciclo è noto, è possibile "srotolare" il ciclo ripetendo il corpo del loop un numero specifico di volte. Questo evita l'overhead del controllo condizionale a ogni iterazione e consente ulteriori ottimizzazioni sul corpo del ciclo.
