**Motivazione (breve riassunto)**

L’esecuzione a tempo di compilazione (Compile-Time Execution) è un potente strumento che può essere utilizzato in numerosi contesti, ma trova particolare impiego nella metaprogrammazione, ovvero nella scrittura di programmi che generano a loro volta altri programmi (o che li manipolano, o ispezionano).

I linguaggi sviluppati fino ad ora, con qualche eccezione di cui si parlerà in seguito, adottano uno o più tra i seguenti approcci che permettono di sfruttare la metaprogrammazione:

* Macro, segmenti di codice che effettuano manipolazioni testuali (come nel caso delle macro in C o C++) oppure sintattiche (come le macro di Lisp). Un uso delle macro particolarmente importante è quello delle “X-macro” in C, una tecnica ampiamente utilizzata che è spiegata in questo articolo <https://en.wikipedia.org/wiki/X_macro> .
* Template-metaprogramming, ovvero una tecnica in cui si usa un linguaggio di tipo funzionale per specificare codice in forma generica. Si tratta di una filosofia differente da CTE, anche se [“è una sfumatura”, trovare la parola giusta], perché anziché usare il linguaggio stesso

Di fatto i linguaggi di programmazione che non forniscono un potente sistema di CTE, in certe situazioni inducono il programmatore a inserire computazioni superflue a tempo di esecuzione, oppure a scrivere un programma separato apposta per il parsing e la modifica del codice. Questo è tipico nei programmi scritti in C, soprattutto quando le X-macro non bastano e/o quando si ha bisogno di ispezionare la struttura dei tipi durante l’esecuzione. Si analizzano ora tre casi di system-language dotati di una qualche forma di CTE.

C++ (a partire da C++11) ha introdotto il costrutto *constexpr*, associato alla definizione di una funzione, per specificare che la funzione può essere eseguita a tempo di compilazione. Questo necessita di supporto da parte del front-end del compilatore in quanto un valore calcolato da una funzione *constexpr* può essere utilizzato come parametro di template. Anche nella versione attuale questo costrutto è relativamente limitante, per i seguenti motivi:

* Una particolare funzione deve essere ‘taggata’ come *constexpr*, e può solamente chiamare funzioni che sono a loro volta *constexpr*; lasciando dunque l’onere al programmatore di marcare le proprie funzioni ogni qualvolta le voglia utilizzare a tempo di compilazione.
* Il programmatore non ha pieno controllo su quando una determinata funzione *constexpr* debba essere eseguita a tempo di compilazione e quando a tempo di esecuzione, con l’eccezione delle chiamate inserite come argomenti di template, che di fatto devono necessariamente eseguire a tempo di compilazione. Nel caso di una funzione fibonacci ricorsiva utilizzata per l’inizializzazione di una variabile, ad esempio, viene impropriamente trasformata in una chiamata alla funzione a run-time, piuttosto che cercare di calcolare il valore a tempo di compilazione. La motivazione è in parte data dal prossimo punto.
* A tempo di compilazione, non si può stampare su standard output, o scrivere file, o qualsiasi cosa che potrebbe avere side-effect. Questo perché le funzioni *constexpr* devono essere ‘pure’, dunque il loro unico scopo deve essere quello di calcolare un valore.
* Non è possibile ispezionare/fare controlli su tipi/valori noti a tempo di compilazione. Questa non è una limitazione del *constexpr*, di per sé, ma anche dei template, che per loro natura non permettono di fare questo tipo di operazioni. Ad esempio, un template qualsiasi non dà la possibilità di controllare il valore dei suoi argomenti, e di stampare un opportuno messaggio nel caso in cui i valori non siano ritenuti accettabili. Questo vale sia per i template associati alle struct/classi sia per i template di funzione.

Si suppone che queste limitazioni siano dovute alla complessità del linguaggio e la vastità del language-specification, rendendo difficile il miglioramento di questo aspetto.

Zig, in contrapposizione a C++, offre un sistema di CTE molto più flessibile.

Jai, un linguaggio di programmazione tutt’ora in sviluppo, colma tutte le mancanze di cui ho parlato; si tratta di un progetto molto più complesso e ambizioso, che tra l’altro sta sperimentando con i concetti di puntatori relativi, macro-igieniche, ecc. Permette la lettura e modifica […]

Si propone dunque il linguaggio di programmazione Ryu. Si tratta di un semplice linguaggio, ma che fornisce comunque le comodità ormai comuni in una pletora di linguaggi moderni, ed è anche utilizzabile per progetti non-triviali (quelle più importanti sono la gestione di dichiarazioni fuori ordine, operator-overloading e function-overloading). Il linguaggio permette di ispezionare le informazioni sintattiche e semantiche del programma mentre viene compilato (per mezzo di un message-loop oppure un altro sistema), di contrassegnare elementi sintattici con un “tag” testuale, utilizzabile dal metaprogramma (consente il filtraggio da parte di sub-routine del metaprogramma per tag). Si potrebbe anche implementare la modifica degli elementi sintattici (è ancora incerto se si tratta di una cosa facilmente implementabile oppure no) Permetterà ai programmatori di sperimentare con queste funzionalità per trovare nuove applicazioni, e potrebbe potenzialmente essere un trampolino di lancio per avanzare lo state-of-the-art del language-design [troppo pretenzioso? Forse cambiare] nel caso questa applicazione del CTE verrà considerata come “standard” nei linguaggi di programmazione procedurali del futuro.