ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Utilizzo di Neverlang per la modellazione di Domain Specific Languages

Terenzi Mirco: TODO: chiedere conferma titolo

Tesi di laurea in:
Programmazione ad Oggetti

Relatore
Prof. Viroli Mirko

Correlatore
Prof. Aguzzi Gianluca

Candidato
Terenzi Mirco

II Sessione di Laurea
Anno Accademico 2023-2024

Sommario

Max 2000 characters, strict.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Terenzi Mirco: TODO

Struttura della Tesi

Terenzi Mirco: TODO

Capitolo 2

Background

Terenzi Mirco: TODO: è corretto background o sarebbe meglio utilizzare un termine italiano (es. contesto)?

2.1 Domain-Specific Languages

In modo del tutto opposto rispetto ai linguaggi general-purpose, progettati per poter essere utilizzati in ogni contesto con un'efficienza e un grado d'espressività relativamente uguali, i Domain-Specific Language (DSL) sono ottimizzati per uno specifico ambito e risultano essere, in molti casi, una soluzione molto più naturale rispetto a quella fornita dai primi [Hud97]. Tra gli esempi più comuni di DSL troviamo SQL, LaTeX (utilizzato anche per la scrittura di questo documento) e CSS.

Nonostante la definizione di DSL sia chiara, non è altrettanto immediato definire se un linguaggio sia o meno un DSL. In questo caso, vi sono alcuni principi chiave da osservare [Fow10]:

Terenzi Mirco: TODO: è corretto inserire qui il riferimento alla documentazione utilizzata per ciò che è scritto all'interno dell'elenco puntato?

- Un DSL è un linguaggio di programmazione e, come tale, la sua struttura dovrebbe essere progettata in modo da essere facile da comprendere per gli esseri umani e, al tempo stesso, eseguibile da un compilatore.
- Essendo un linguaggio, deve avere un senso di fluidità e la sua espressività deve essere derivata non solo da un'espressione individuale, ma anche dalla combinazione di più istruzioni.

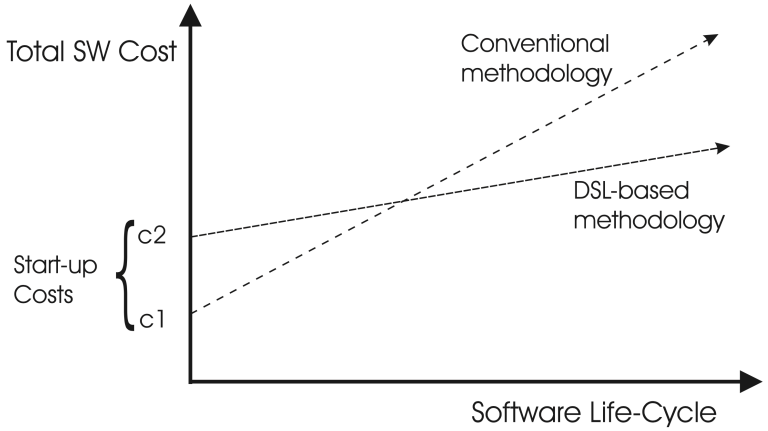

Figura 2.1: Il vantaggio ottenuto dall'utilizzo di DSL.

- Coerentemente alla definizione, un DSL dovrebbe implementare l'insieme minimo di caratteristiche necessarie per poter supportare il dominio applicativo di interesse ed evitare funzioni non strettamente necessarie che potrebbero rendere il linguaggio più difficile, sia da utilizzare, sia da comprendere.

L'utilizzo di questa tipologia di linguaggi di programmazione comporta una serie di vantaggi:

- **Produttività**: È possibile aumentare il livello d'astrazione e, di conseguenza, aumentare la produttività. Questo perché è possibile utilizzare direttamente i concetti propri del dominio applicativo, non essendo limitati dalla necessità di mantenere una generalità atta ad ottenere un linguaggio applicabile in molteplici contesti [KT08]. Come indicato da Paul Hudak [Hud97] ed illustrato nella fig. 2.1, assumendo che il costo di sviluppo di un programma sia lineare, possiamo ipotizzare che il costo iniziale richiesto sia maggiore nel caso si utilizzasse un DSL rispetto a metodologie più convenzionali (considerando il caso in cui il linguaggio andasse sviluppato, se ne venisse utilizzato uno esistente tale costo sarebbe significativamente minore). Ciò nonostante, la pendenza della curva è considerevolmente più bassa e quindi, da un determinato punto in poi, l'utilizzo di DSL porterebbe un risparmio significativo.

- **Qualità del codice**: L'utilizzo di DSL favorisce anche una migliore qualità del codice. Infatti, il linguaggio può includere regole direttamente trasposte dal dominio all'interno del quale è applicato. In questo modo risulta molto più difficile, talvolta impossibile, ottenere dei risultati non attesi. Ad esempio, Antti Raunio, capo ingegnere del progetto EADS [Met07], afferma che "la qualità del codice generato è chiaramente migliore [...] perché il linguaggio di modellazione è stato progettato per adattarsi all'architettura del nostro terminale"[1]. Inoltre, l'offuscamento della reale complessità del problema, dovuto all'utilizzo di DSL, consente ai nuovi sviluppatori di lavorare ad un alto livello d'astrazione, senza dover conoscere tutti i dettagli inerenti all'implementazione del linguaggio [Met07].

- **Migliore manutenibilità**: Sebbene l'uso di DSL non renda l'implementazione necessariamente meno complessa di quanto si possa ottenere utilizzando un linguaggio *general-purpose*, la manutenibilità del codice risulta essere accentuata [KvdSV10]. Infatti, considerando il volume del codice, l'utilizzo di DSL comporta una minor quantità di codice da comprendere, facilitandone la modifica. Inoltre, è possibile ignorare il problema di mantenere coerente ciò che è definito dalla grammatica con la struttura gerarchica definita dall'Abstract Syntax Tree (AST) in quanto quest'ultimo si evolve con la prima [BM10].

2.2 Grammatica non contestuale

Nel campo dell'informatica, una grammatica formale è costituita da regole che descrivono precisamente come vengono generati i simboli di un linguaggio formale, partendo da un insieme finito chiamato alfabeto. Ogni regola sintattica, detta anche produzione, è espressa nella forma $A \rightarrow x$, dove A è un simbolo non terminale e x è una combinazione di simboli terminali e/o non terminali.

In particolare, la grammatica non contestuale, o Context-Free Grammar (CFG), è un tipo specifico di grammatica formale che deriva il proprio nome dal fatto che

[1] Di seguito riportata l'affermazione citata, in lingua originale: "the quality of the generated code is clearly better, simply because the modelling language was designed to fit our terminal architecture"

le regole sintattiche mantengono la loro validità indipendentemente dai simboli che precedono o seguono il simbolo non terminale a cui si applicano. Questa caratteristica è dovuta al fatto che le regole sintattiche di una grammatica non contestuale ammettono soltanto una variabile non terminale sul lato sinistro della regola [LR22].

Le grammatiche non contestuali assumono un ruolo fondamentale nella teoria dei linguaggi formali, in particolare nella definizione e nell'analisi dei linguaggi di programmazione. La CFG è utilizzata principalmente per la modellazione della sintassi, ma viene impiegata anche nella costruzione di interpreti e compilatori [LR22].

2.3 Compilazione del codice

Il compilatore è un software fondamentale nel campo della programmazione e dell'informatica, il cui compito è tradurre il codice sorgente, scritto in un linguaggio di alto livello, in un linguaggio di basso livello, tipicamente il linguaggio macchina o codice oggetto, che può essere eseguito dal calcolatore.

Il processo di compilazione è suddiviso in diverse fasi, ognuna delle quali ha un ruolo specifico nel convertire il codice sorgente in un programma eseguibile. Le fasi principali sono di seguito elencate [ALSU06]:

- **Analisi lessicale**: Durante la fase dell'analisi lessicale, talvolta definita anche *scanning*, la sequenza di caratteri di input viene analizzata e divisa in porzioni significative, chiamate lessemi. Per ogni lessema viene prodotto come output un token, definito come coppia di valori `<nome-token, valore-attributo>`, che viene processato dalla fase successiva.

- **Analisi sintattica**: La fase dell'analisi sintattica, più comunemente indicata come *parsing* o, in italiano, parsificazione (allo stesso modo, il programma che svolge queste operazioni è chiamato *parser*), prevede che i token siano utilizzati per creare una rappresentazione ad albero che descriva efficacemente la struttura grammaticale dell'input. Un esempio di questo tipo di rappresentazione è costituito dall'albero sintattico. Al suo interno, ogni

nodo rappresenta un'operazione e i nodi ad esso discendenti, chiamati anche figli, rappresentano gli operandi dell'operazione definita dal primo. La parsificazione è svolta principalmente secondo due metodologie [GJ06]:

  - l'analisi **top-down** consiste nell'emulare il processo di produzione della frase. In questo modo, partendo dal simbolo iniziale si procede per fasi ad evolvere l'output in modo da farlo corrispondere alla frase ottenuta come input. Il nome indica che l'albero sintattico viene costruito partendo dall'alto, ossia dal nodo radice, proseguendo verso il basso;
  - l'analisi **bottom-up**, al contrario, prevede di invertire il processo di produzione della frase, ossia ha come obiettivo quello di ridurre la frase di input al simbolo iniziale.

- **Analisi semantica**: In questo stadio della compilazione, il fine è analizzare la correttezza semantica del programma. Più nello specifico, vengono utilizzati l'albero sintattico ottenuto dalla fase precedente e la tabella dei simboli per controllare che l'input sia semanticamente coerente con quanto è definito dal linguaggio. Una parte fondamentale dell'analisi semantica è quella del controllo del tipo (in inglese *type-checking*), durante la quale il compilatore si accerta che il valore assegnato ad una variabile sia ammissibile con il tipo di tale variabile e, allo stesso modo, che gli operandi utilizzati in un'operazione siano compatibili con l'operazione stessa. Talora, le specifiche del linguaggio permettono delle conversioni di tipo chiamate coercizioni.

- **Generazione di codice intermedio**: Durante la compilazione, il compilatore genera una o più rappresentazioni intermedie. L'albero sintattico prima descritto rappresenta una delle varie forme che tali rappresentazioni possono assumere. In particolare, devono essere rispettate due proprietà significative: le rappresentazioni devono essere facili da generare e facili da trasporre in codice target (spesso si tratta di linguaggio macchina).

- **Ottimizzazione del codice**: Successivamente, si procede a perfezionare, per quanto possibile, il codice intermedio in modo che si possa poi ottenere un codice finale migliore.

- **Generazione del codice**: Infine, il codice intermedio viene utilizzato per generare il codice target.

2.4 Programmazione feature-oriented

La programmazione orientata alle funzionalità, comunemente nota come Feature-Oriented Programming (FOP), è un paradigma di programmazione focalizzato sulla modularizzazione del software. L'obiettivo principale della FOP è facilitare la gestione e lo sviluppo di programmi complessi, consentendo ai programmatori di scomporli in unità modulari. Questa metodologia è ampiamente applicata in contesti di linee produttive di software e viene utilizzata per gestire la creazione di diverse versioni di un programma, adatte alle specifiche richieste dei clienti [ABKS13]. Inoltre, la FOP è impiegata per costruire sistemi configurabili, che prevedono l'attivazione o la disattivazione di funzionalità specifiche e offrono un elevato grado di personalizzazione.

Le caratteristiche principali della FOP sono le seguenti:

- **Riusabilità**: Confrontando la FOP con l'approccio più classico della Object-Oriented Programming (OOP), si nota come la prima offra una maggiore modularità e flessibilità del codice. Di conseguenza, la riusabilità è amplificata, poiché ogni funzionalità implementata separa il nucleo dalla parte relativa alle interazioni [Pre97].

- **Gestione della complessità**: La FOP permette di gestire progetti di considerevole complessità grazie allo sviluppo modulare del codice in unità distinte. Queste unità possono essere sviluppate e mantenute in modo indipendente, riducendo significativamente l'interdipendenza e la complessità del sistema complessivo. Inoltre, la separazione delle funzionalità semplifica l'evoluzione del software, poiché ogni modifica può essere limitata a una singola funzionalità senza influenzare l'intero sistema [Pre97].

- **Adattabilità**: La FOP consente una facile configurazione e personalizzazione dei sistemi software attraverso la selezione e la composizione delle funzionalità. Questo approccio permette di creare diverse varianti di un prodotto

software a seconda dei requisiti specifici, facilitando l'adattamento a nuovi contesti o a esigenze mutate nel tempo [ABKS13].

2.5 Neverlang

Neverlang Language Workbench è un framework sviluppato presso l'Università di Milano dal professor Cazzola e dai suoi collaboratori, il cui scopo è favorire lo sviluppo di linguaggi di programmazione, in particolare seguendo il paradigma della FOP.

È basato sull'idea che i linguaggi di programmazione abbiano un'intrinseca divisione modulare in più caratteristiche, o *features*, ciascuna delle quali è implementata da un componente specifico. In accordo con tale visione, l'obiettivo del framework è definire i linguaggi tramite una divisione in frammenti, chiamati moduli, ognuno dei quali si occupa di implementare una specifica caratteristica e, infine, tramite la combinazione dei diversi moduli, ottenere un linguaggio di programmazione specifico per il contesto applicativo richiesto, ossia un DSL [Caz].

Quando il codice viene compilato, ogni costrutto viene inserito in una rappresentazione chiamata AST, o albero sintattico astratto. L'AST è simile all'albero sintattico descritto in precedenza ma con la fondamentale differenza che non rappresenta ogni dettaglio della reale sintassi. È essenzialmente una versione alternativa e semplificata che consente di focalizzarsi sulla logica descritta dal linguaggio.

All'interno di ogni modulo vengono definite due parti principali:

- la **sintassi**, utilizzando una grammatica formale non contestuale;
- e la **semantica**, in funzione della sintassi e sfruttando i vari elementi nonterminali e i loro attributi. Inoltre, il comportamento del componente può essere suddiviso in diverse fasi, ciascuna definita da un ruolo specifico all'interno del modulo. Anche l'ordine d'esecuzione dei ruoli può essere definito, le tre tipologie di visite predefinite sono:
  - Semi-automatica: Tale strategia prevede che i nodi siano visitati partendo dal nodo radice e discendendo ai nodi figli finché non viene indi-

viduata una definizione semantica. Una volta trovata, il controllo della visita viene trasferito all'esecuzione di tale azione.

- *Post-order*: In questo caso la visita dell'albero predilige la discesa in profondità, ciò significa che l'esecuzione delle azioni definite dalla semantica dei vari nodi è posticipata a dopo che tutti i nodi figli sono stati valutati.
- Giustapposizione: Nei due metodi precedenti, ad ogni ruolo corrisponde una visita all'albero. Al contrario, quando due ruoli sono giustapposti, la loro esecuzione viene eseguita "in una volta" e cioè tutte le azioni corrispondenti a tali ruoli sono eseguite in sequenza in una singola visita all'albero.

Successivamente, i componenti del linguaggio vengono definiti combinando definizioni di sintassi e semantica provenienti da diversi moduli all'interno di costrutti denominati *slice*, ognuno dei quali contiene la definizione di un singolo componente del linguaggio [VC15]. Infine, il linguaggio viene generando combinando i vari *slice* insieme.

Tra i vantaggi principali di Neverlang troviamo [Caz12]:

- **Modularità**: Ognuno dei moduli che compongono il linguaggio viene compilato separatamente, permettendo di utilizzarne uno o più di uno (in tal caso aggregandoli in uno *slice*) all'interno di altri linguaggi.

- **Riutilizzo**: Neverlang offre la possibilità di riutilizzare frammenti di linguaggio in più di un contesto. Ad esempio, un frammento può utilizzare la sintassi di un altro frammento definito in precedenza e ridefinire la semantica, o viceversa. Inoltre, è possibile ridefinire l'ordine dei simboli non-terminali utilizzati nella sintassi o nella semantica importata.

- **Estensibilità**: L'architettura modulare utilizzata all'interno di Neverlang facilita l'estensione di linguaggi esistenti. Per aggiungere nuove funzionalità non è necessario modificare il codice, ma è sufficiente integrare un nuovo *slice*.

2.6 Java

In aggiunta a Neverlang, per la realizzazione del progetto è stato utilizzato Java. Java è un linguaggio di programmazione ad alto livello, orientato agli oggetti e a tipizzazione statica, sviluppato da Sun Microsystems nel 1991. È molto diffuso e ben supportato, con una vasta comunità di sviluppatori e una grande quantità di librerie. Uno degli obiettivi principali di Java è quello di essere il più possibile autonomo rispetto alla piattaforma di esecuzione, permettendo di scrivere una volta il codice e farlo eseguire su qualsiasi Java Virtual Machine (JVM), indipendentemente dall'architettura del calcolatore [IBM].

Java è stato utilizzato per la realizzazione del progetto in quanto Neverlang è progettato per essere completamente integrato con esso. Il suo compilatore (nlgc) è stato sviluppato per poter convertire il codice scritto utilizzando il DSL di Neverlang in un nuovo codice supportato dalla JVM. Inoltre, Neverlang permette di utilizzare Java (ma non solo; anche Scala, ad esempio, è supportato) come linguaggio per la definizione della semantica all'interno dei moduli del DSL. Ciò è possibile in quanto gli accessi a variabili non-terminali, definiti all'interno della sintassi, sono sostituiti dallo specifico plug-in con accessi alla reale rappresentazione interna del linguaggio. In particolare, l'accesso alle variabili viene effettuato tramite una chiamata all'n-esimo figlio dell'AST [CV13].

Bibliografia

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[ALSU06] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[BM10] Claus Brabrand and Pierre-Etienne Moreau. *Proceedings of the of the Tenth Workshop on Language Descriptions, Tools and Applications (LDTA 2010)*. Association for Computing Machinery, New York, NY, USA, 2010.

[Caz] Walter Cazzola. Neverlang 2: Language workbench. Ultimo accesso: 23 Agosto 2024.

[Caz12] Walter Cazzola. Domain-specific languages in few steps. In Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book, editors, *Software Composition*, pages 162–177, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[CV13] Walter Cazzola and Edoardo Vacchi. Neverlang 2 – componentised language development for the jvm. In Walter Binder, Eric Bodden, and Welf Löwe, editors, *Software Composition*, pages 17–32, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Fow10] Martin Fowler. *Domain-specific languages*. Addison-Wesley Professional, 2010.

[GJ06] Dick Grune and Ceriel JH Jacobs. Parsing techniques (monographs in computer science), 2006.

[Hud97] Paul Hudak. Domain specific languages. *Handbook of programming languages*, 3(39-60):21, 1997.

[IBM] IBM. What is java? Ultimo accesso: 23 Agosto 2024.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation*. Wiley-IEEE Computer Society Pr, 2008.

[KvdSV10] Paul Klint, Tijs van der Storm, and Jurgen Vinju. On the impact of dsl tools on the maintainability of language implementations. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, New York, NY, USA, 2010. Association for Computing Machinery.

[LR22] Peter Linz and Susan H Rodger. *An introduction to formal languages and automata (7th Edition)*. Jones & Bartlett Learning, 2022.

[Met07] MetaCase. Eads case study, 2007.

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, pages 419–443, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[VC15] Edoardo Vacchi and Walter Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages Systems & Structures*, 43:1–40, 2015.