UNIVERSITÀ DEGLI STUDI DI VERONA

DIPARTIMENTO DI INFORMATICA

Corso di Laurea in Informatica

Tesi di Laurea Triennale

Analisi degli strumenti informatici a supporto della programmazione in presenza di disabilità visiva

Relatore
Prof. Davide Quaglia

Candidato
Mattia Donà

Matricola
VR445883

ANNO ACCADEMICO 2022/2023

Indice

1 Introduzione

L'obiettivo di questo documento è analizzare e approfondire il tirocinio degli studenti Hazem e Mattia, che partì con il tema: "analisi di strumenti informatici a supporto della programmazione in presenza di soggetti con disabilità visiva e motoria". Questo documento sarà strutturato come una guida al percorso che i due studenti hanno intrapreso per ridurre la vasta quantità di informazioni relative a questo argomento in un elenco di tecniche più specifiche ed efficaci al fine dello sviluppo di codice. In particolare, verrà approfondita una specifica sezione del tema principale, ovvero la lettura di codice scritto al calcolatore mediante tecnologie che soddisfino le esigenze degli utenti con disabilità visiva.

2 Analisi

2.1 Approfondimento del tema

La prima fase di questo percorso è stata una ricerca con lo scopo di approfondire il tema del progetto, poiché le informazioni possedute riguardo alle difficoltà delle persone con questa tipologia di disabilità erano alquanto misere. Durante alcune ricerche sul web, la prime conoscenze sono emerse dalla visione di un'intervista a due programmatori ipovedenti che sviluppano software lato backend per l'azienda in cui lavorano. La loro testimonianza è stata assai importante perché ha fornito delle informazioni chiave sugli strumenti che i programmatori ipovedenti utilizzano per lo sviluppo di codice e sui criteri che assumono per valutare queste teconlogie. In particolare, i requisiti principali che tali strumenti devono possedere sono una buona usabilità ed un' ampia personalizzazione in base alle esigenze di ogni utente. Il link all'intervista viene lasciato a piè di pagina.
[1]

2.2 Criteri e requisiti degli strumenti

I criteri utilizzati per l'analisi degli strumenti per lo sviluppo di codice da parte di soggetti con disabilità visive o motorie sono stati: accessibilità, autonomia, impostazioni e preferenze. Riguardo l'accessibilità, è ritenuto fondamentale che gli strumenti siano in grado di garantire ai soggetti disabili l'accesso ai servizi, offrendo inoltre un'interazione utente-dispositivo che sia la più simile possibile a quella di un utente senza limitazioni fisiche. Parlando invece di l'autonomia si tocca un punt critico su cui bisgona riflettere attentamente. I servizi web, i software che usiamo quotidianamente spesso richiedono l'inserimento, l'utilizzo di dati sensibili, oppure di effettuare download o installazioni. Per

[1] https://www.youtube.com/watch?v=jun_hLBWxds

un utente ipovedente, queste attività potrebbero risultare infattibili senza un buon supporto da parte della tecnologia. Infine, la modifica delle impostazioni e delle preferenze viene incontro alla persona che ha le sue esigenze specifiche, che possono variare in base a seconda della propria disabilità. Un'ampia gamma di opzioni di modifica, come la lingua, la velocità di lettura e il tono della voce, rende un determinato dispositivo più attraente rispetto ad un altro. Dunque, questi saranno i tre criteri cruciali per valutare la qualità e l'efficacia di uno strumento per lo sviluppo di codice da parte di soggetti con disabilità visive.

3 Restrizione della tipologia di utenti

Inizialmente, le tipologie di utenti su cui volevamo concentrare il nostro studio erano sia soggetti con disabilità visive e motorie. Dopo una valutazione attenta e una successiva discussione con i docenti, abbiamo preso coscienza del fatto che la categoria di persone con disabilità visiva presenta già di per se molte sfide da affrontare, di conseguenza una vasta gamma di tematiche da esplorare, ad esempio potremmo concentrarci su soggetti ipovedenti che hanno già acquisito conoscenze riguardo le tecniche di sviluppo di codice, oppure su coloro che desiderano avvicinarsi senza alcuna esperienza al mondo dell'informatica. Questa considerazione ci ha indotto dunque ad una revisione dei nostri obiettivi di studio, per poi spostare l'attenzione dei nostri studi solo sulle persone con disabilità visive, escludendo quelle con disabilità motorie. Inoltre, è importante considerare che i dispositivi per facilitare la programmazione a soggetti con difficoltà legate al movimento sono principalmente di natura hardware, il che significa che effettuare anche solo un semplice test richiederebbe un acquisto ed uno studio specifico della modalità d'uso del dispositivo.

4 Dispositivi che sfruttano l'udito: gli Screen reader

La teconlogia che maggiormente ha dimostrato di possedere i requisti cercati sono stati gli screen reader. Questi software consentono di comprendere l'informazione visualizzata sullo schermo del proprio computer senza l'uso della vista, traducendola in un formato audio e leggendola attraverso un sintetizzatore vocale. Esistono diversi screen reader installabili a seconda del sistema operativo, oppure integrabilli come plug-in per browser web o applicazioni specifiche. Tuttavia, questi dispositivi richiedono un'interazione da parte dell'utente, poichè non sono in grado di essere autonomi, la persona deve essere in grado di muovere il cursore sulla zona d'interesse dello schermo mediante mouse o tastiera. I prossimi obbiettivi saranno esaminare i principali screen reader disponibili sul mercato, descrivendo brevemente i test effettuati su questi software e condividendo le personali impressioni su di essi.

4.1 Screen reader per Mac-OS: Utility Voice-Over

VoiceOver è un'applicazione per sistemi Apple, progettata per aiutare gli utenti con disabilità visive a navigare e interagire con il loro dispositivi. Questo screen reader è integrato all'interno di macOS e iOS e non richiede alcun download o installazione, rendendolo dunque facilmente accessibile per gli utenti. La sua attivazione può essere effettuata attraverso comandi rapidi o mediante l'assistente vocale "Siri". Una volta attivato, VoiceOver offre un'esperienza del tutto personalizzabile per l'utente, come l'impostazione della lingua (in base alla lingua del sistema), la velocità di lettura e il volume della voce. Inoltre, VoiceOver è in grado di rilevare l'utilizzo di una barra braille, rendendo la navigazione ancora più semplici per gli utenti non-vedenti in grado di utilizzarla. La voce guida non solo legge il testo sullo schermo, ma avvisa anche l'utente sul tipo di contenuto che sta leggendo, ad esempio se esso è una pagina web, un elenco puntato, un bottone, una check-box, oppure un'input. VoiceOver è uno strumento gratuito per chi possiede un dispositivo Apple, ma non è open-source. Tuttavia, a prima impressione sembra molto semplice da usare e svolge efficacemente il suo ruolo di guida alla navigazione del sistema. Uno svantaggio che possiamo sottolineare è l'esclusività ai dispositivi Apple, il quale non è certamente il sistema operativo più diffuso ed open-source esistente.

4.2 Screen reader per Windows: Nvda

NVDA è un lettore schermo disponibile per i sistemi operativi Windows e, a differenza di altri screen reader, richiede un download ed un'installazione prima di poter essere utilizzato. È completamente gratuito ed è uno dei pochi lettori schermo ad essere open-source. All'avvio dell'applicazione non viene fornita una guida all'utilizzo, ma è possibile acquisire una maggiore conoscenza dell'ambiente grazie all'ampia documentazione disponibile sul sito ufficiale [2]. NVDA è un lettore schermo molto efficiente, in grado di leggere tutte le interazioni tra l'utente con il sistema, oppure col web, ed offre una vasta gamma di impostazioni e preferenze, come la velocità di lettura, la lingua e il timbro della voce. È in grado, come VoiceOver, di rilevare la presenza di una barra braille. L'utilizzo di NVDA può essere ulteriormente agevolato grazie alle shortcuts di Windows, che permettono una migliore navigazione nelle finestre all'interno dello schermo. In sintesi, NVDA rappresenta un eccellente screen reader per i sistemi Windows in grado di esaudire la maggior parte delle esigenze degli utenti non-vedenti.

4.3 Screen reader per Windows: Jaws

JAWS è un lettore schermo per sistemi operativi Windows. Analogamente a NVDA, richiede un processo di download ed installazione ma, a differenza di quest'ultimo, JAWS

[2]https://www.nvda.it/documentazione

è un software a pagamento. É possibile testare una versione demo limitata per una durata di 40 minuti. Una caratteristica distintiva di JAWS è poter navigare sullo schermo ed interagire con le finestre aperte sullo schermo utilizzando solamente le frecce direzionali ed il comando TAB, senza che sia necessario per l'utente conoscere i numerosi comandi rapidi di Windows. Nonostante la fase di test sia limitata a causa del costo del software, è stato possibile approfondire la conoscenza di JAWS attraverso la sua documentazione [3]. Durante la prova della versione demo, JAWS è risultato un lettore schermo molto efficiente, riesce a leggere ogni riga su cui il cursore è posizionato, inoltre fornisce anche una descrizione dettagliata riguardo la zona della finestra su cui si trova. La versione a pagamento include le solite prefrenze come il multilingua, impostazioni di velocità e timbro della voce.

4.4 Screen reader per Linux: Orca

Parlando del sistema operativo Linux, esso dispone di un proprio lettore di schermo esclusivo chiamato Orca. Questo software è integrato nel sistema operativo e non richiede di alcun download, la procedura d'installazione è guidata da un sintetizzatore vocale. Orca ha una vasta gamma di opzioni, tra cui una modalità "allenamento", che permette all'utente di familiarizzare con l'utilizzo del software. Tra le preferenze impostabili vi sono, come abbiamo visto spesso essere presenti, la velocità ed il timbro della voce, mentre la lingua si basa sulla lingua del sistema operativo. Orca è un software gratuito, esso risulta essere disponibile a chiunque installi una distribuzione Linux, di conseguenza questo rende il software accessibile a tutti nonostante la sua esclusività come tool integrato nel sistema operativo. La documentazione di Orca è molto dettagliata e approfondita per quanto riguarda la guida e l'uso. [4].

5 Considerazioni dopo l'analisi

L'analisi degli screen reader non può essere considerata del tutto oggettiva poichè essi non sono stati analizzati da utenti che possiedono delle difficoltà visive. Personalmente dunque ho trovato difficile immedesimarmi come tester e valutare le differenze ed i particolari specifici dei lettori schermo analizzati, inoltre diversi di questi, per ottenere il massimo rendimento, vengono utilizzati interagendo con dispositivi hardware, come una barra braille, che richiedono conoscenza dell'utilizzo ed un aquisto per poter ottenere le migliori prestazioni. Come è stato descritto nelle varie analisi degli screen reader, le principali esigenze degli utenti ipovedenti sono la possibilità di modificare lingua, la velocità ed il timbro della voce. Il motivo di impostare un ritmo di lettura sostenuto proviene dalla

[3] https://support.freedomscientific.com/Content/Documents/Manuals/JAWS/JAWS-Quick-Start-Guide.pdf

[4] https://help.gnome.org/users/orca/stable/

capacità dei soggetti ipovedenti di sviluppare una velocità di ascolto superiore alla norma e quindi di poter comprendere le informazioni lette dallo screen reader più velocemente rispetto alle persone con capacità visive comuni.

6 Software che sfruttano la voce

Un'altro approccio pensato per facilitare l'interazione tra il sistema e utente ipovedente è l'uso della voce. Attraverso vari software di dettatura è possibile poter coordinare la navigazione sullo schermo mediante comandi vocali, essi sono disponibili in ogni sistema operativo e funzionano tutti secondo il processo di trasformare ciò che l'utente dice in testo o comando vocale da sottoporre al sistema. Applicando queste tecnlogie allo sviluppo del codice, dettare codice al sistema che lo trascrive direttamente nel file risulta essere un'approccio limitato, perchè a differenza di uno screen reader in cui posso sempre leggere e, se scrivo qualcosa, posso poi leggerlo, in un software di dettatura posso sempre scrivere, ma non posso leggere ciò che scrivo. Quest'ultima considerazione risulta una pesante limitazione, quindi l'utilizzo di questi software è ritenuto superato dalla tecnologia degli screen reader e per tale motivo sono stati scartati da questa ricerca.

7 Software che sfruttano l'IA

7.1 Open-Api

I software che sfruttano l'intelligenza artificiale sono in grado di tradurre un testo scritto in linguaggio naturale in codice eseguibile, secondo un qualiasi linguaggio di programmazione. Sfruttare software come Open-Api nel nostro intento di facilitare la programmazione a studenti ipovedenti può risultare parzialmente efficace, infatti queste teconlogie risultano efficienti nel creare codice senza particolari capacità di programmazione e ,come si può immaginare, produrre una frase in linguaggio naturale è più semplice di produrre del codice programmando. Tuttavia, questa semplificazione non è necessaria ai nostri utenti, poichè assumiamo come già detto che siano dei programmatori, il fatto che siano ipovedenti non esclude il fatto che essi siano in grado di programmare, di conseguenza, questa strategia è stata scartata in favore di strumenti che possano comunque sfruttare lo studio e le abilità di programmazione dei nostri utenti.

8 Ulteriori considerazioni

Dopo aver completato l'analisi dei vari strumenti disponibili per facilitare la programmazione a soggetti ipovedenti, è stato deciso di dedicarsi alla categoria degli screen reader. Il prossimo obbiettivo sarà quello di testare gli screen reader analizzati in precedenza con

vari ambienti di sviluppo, al fine di testare quali tra loro siano più efficenti nel mondo dello sviluppo del codice.

9 Prima fase di test

Il seguente step in questo progetto è stato testare i vari screen reader all'interno di vari ambienti di sviluppo di codice (IDE), per poter avvicinarsi ancora di più al tema iniziale di questa tesi. Prima di tutto sono stati classificati i vari IDE in 2 categorie: la prima riguarda ambienti di sviluppo scaricabili ed installabili su un calcolatore, la seconda invece riguarda ambienti di sviluppo in cloud. Per quanto riguarda la prima classe, è stato scelto di testare come ambiente di sviluppo "Visual Studio Code", la scelta dell'IDE ricade sulla numerosa tipologia di linguaggi di programmazione (python, c, php, html, css, java, javascript, ...) disponibili mediante la semplice installazione dei pacchetti. Inoltre, un secondo punto a favore riguarda la sua accessibilità (se volessi programmare in Java, Visual Studio mi chiede se voglio installare il compilatore e l'interprete di java mediante una finestra di pop-up, in seguito sono subito pronto a scrivere il mio primo programma in java senza dover fare altro), il che rende il lavoro molto più gestibile.

9.1 Test 1: Uso di Voice-Over con Visual Studio Code

Il primo test si è basato sull'obiettivo di riuscire a creare un programma e leggerne l'output dal terminale. Prima di discutere del risultato ottenuto, è necessario spiegare in maniera più accurata le potenzialità dei plug-in installabili in Visual Studio Code, come ad esempio l'attivazione dei colori per ipovedenti, l'ottimizzazione dell'editor in presenza di uno screen reader, la navigazione rapida all'interno delle impostazioni dell'editor, tutte preferenze modificabili al fine di migliorare le prestazioni e l'accessbilità di un utente con disabilità visiva. Dopo aver attivato VoiceOver su Mac, viene aperto l'editor mediante il comando [COMMAND + Spazio] e digitando nella barra di ricerca "Visual Studio code". Entrati all'interno dell'editor, la navigazione nelle varie opzioni è possibile cliccando [TAB] insieme alle frecce direzionali. Tutte le opzioni vengono lette dallo screen reader, specificando inoltre la tipologia del contenuto che si sta leggendo('Campo di testo', 'opzione', 'terminale', ...). Creato e salvato un file di prova scritto in python, si è scritto del semplice codice giusto per avere del materiale per il testing, si è scoperta la possibilità di poter rileggere le righe mediante il comand [TAB] ed evidenziando ciò che si desidera venga letto dallo screen reader. Una volta terminata la scrittura del codice, è possibile spostarsi sulla schermata fino ad arrivare all'opzione "esegui codice", infine è necessario eseguire un ultimo movimento verso la zona del terminale (disponibile nell'editor stesso) per leggere la riga relativa all' output del programma. Possiamo dunque considerare raggiunto l'obiettivo prefissato all'inizio di questo sotto-capitolo, nonostante però siano

emerse alcune problematiche nel utilizzo di questa tecnlogia. La principale difficoltà sono gli spostamenti all'interno dell' editor che, per quanto possano essere rapidi, possono facilmente disorientare il programmatore e, di conseguenza, si ritrova nella situazione di non essere più in grado di ritrovare la zona in cui voleva operare. Mentre i punti di forza riscontrati sono stati il risultato dell'unione delle potenzialità di Utility VoiceOver come screen reader e di Visual Studio code come editor, entrambi già trattati precedentemente, la cui combinazione porta ad un risultato tutto sommato soddisfacente. [5]

9.2 Test 2: Uso di Voice-Over con Replit

Replit è un ambiente di sviuluppo online, rende possibile la scrittura e l'esecuzione di codice nei principlai linguaggi noti. Non dispone di strumenti che facilitano l'accessibilità a soggetti con difficoltà visive. L'obiettivo del secondo test è simile al primo, ovvero provare a leggere l'output di un programma scritto nel linguaggio Python. La prima differenza notata rispetto al test precedente è la navigazione all'interno di contenuti presenti in una pagina web rispetto ad una schermata di un editor installato. La pagina web infatti presenta molte più interfacce,opzioni, inserzioni che aumentano molto la difficoltà di movimento e, di conseguenza, la facilità di sentirsi disorientati. Lo screen reader non legge il contenuto della zona evidenziata all'interno di una pagina web, come invece accade in un editor installato, inoltre vi è un grosso problema di navigazione dovuto all'utilizzo del terminale, poichè, una volta entrati al suo interno, i comandi di movimento vengono interpretati come comandi di Bash, creando una situzione in cui si resta bloccati al suo interno. Resta possibile come fatto nel test precedente creare un file, scrivere del codice al suo interno ed eseguirlo, ma non è possibile visualizzare l'output sul terminale, il che porta alla luce un problema abbastanza grave. Infine, non poter rileggere le righe di codice come accadeva su Visual Studio Code, comporta a continui problemi di sintassi errata. L'insieme delle problematiche discusse sono state trovate anche in tutti i numerosi editor di sviluppo di codice online testati.

9.3 Test 3: Uso di Nvda con Visual Studio Code

Spostandosi sul sistema operativo Windows, dal momento che Jaws risulta essere un lettore schermo a pagamento, i test sono stati effettuati sullo screen reader Nvda. Come già trattato nella sua analisi, il lettore schermo è ben strutturato e i suoi comandi sono piuttosto semplici e facilmente comprensibili, la navigazione sullo schermo avviene mediante il tasto [TAB] ed ogni zona viene spiegata nel dettaglio (contenuto tipo input, box, avviso di sistema, ...). Ad una prima interazione, l'utilizzo sembra essere più preciso rispetto a VoiceOver, infatti, in caso di disorientamento, si riesce sempre, grazie alla divisione dello schermo in zone, a tornare in un'area principale da cui ritentare il raggiungimento del

[5] https://www.dropbox.com/s/sopqp67ntdmiuuf/test1_tirocinio.mov?dl=0

percorso deisderato. Testato insieme a Visual Studio Code per Windows, nel tentativo di visualizzare l'output di un programma scritto in linguaggio Python, come per i test con VoiceOver, l'operazione è avvenuta con successo, senza risultare troppo complicata o stressante.

9.4 Test 4: Uso di Nvda con Replit

L'ultima combinazione screen reader-editor provata è stata il lettore schermo Nvda insieme all'IDE online Replit, sempre con l'obiettivo di visualizzare l'output di un file Python mandato in esecuzione. Risulta ancora una volta impossibile visualizzare l'output del file sul terminale a causa della non-interpretazione del comando [TAB], utilizzato dallo screen reader per navigare sullo schermo, infatti risulta solo un comando dato in input al terminale, impossibilitando dunque il movimento all'interno dell'IDE Online. In aggiunta, non è possibile leggere le righe di testo presente nel terminale. Giunto a questo punto, il problema lo si può attribuire all'editor online, che non essendo configurato per essere accessibile e performante con uno screen reader, non è in grado di gestire le situazioni critiche che abbiamo generato nei diversi test. Possiamo sottolineare ancora una volta tra le criticità il problema di sentirsi disorientati all'interno di un IDE online, vista l'innumerevole quantità di pop-up, inserzioni, elementi presenti in una pagina web rispetto ad un editor installato.

10 Ulteriori considerazioni

Giunto a questo punto dell'analisi, gli editor di codice online sono risultati non adeguati per venire incontro alle necessità di utenti con disabilità visive, al contrario di quelli installabili, come Visual Studio Code, che invece hanno integrati al loro interno dei plug-in che facilitano il lavoro a soggeti appartenenti alla categoria d'interesse, inoltre sono in grado di essere ottimizzati per lavorare in sincronia con lo screen reader. Dopo un incontro con i docenti, si è spostato il focus della nostra analisi su degli aspetti non ancora approfonditi in tema di utilizzo di editor insieme agli screen reader. Come motivato sopra, si è deciso di concentrare lo studio sulle teconlogie che hanno portato i risultati più soddisfacenti, ovvero editor scaricabili ed i più noti screen reader disponibli per i principali sistemi operativi, ovvero VoiceOver per MacOs e Nvda per Windows. I principali aspetti evidenziati sono stati:

- se in un programma vi è un' errore, come l'utente non-vedente può correggerlo? (Sapere a che riga si trova, risalire a quella determinata riga, correggere l'errore).
- può l'autocompilazione del codice essere uno strumento utile anche ad utenti che non possono leggerla con gli occhi?

- la lettura di codice da parte di uno screen reader può essere modificata in modo tale da rendere più comprensibile il contenuto del programma? (Il codice di un programma fino ad ora è stato letto come scritto, senza punteggiatura, quindi senza un' intonazione, un' enfasi particolare).

11 Seconda fase di test

La seconda fase di test ha avuto l'obbiettivo di trovare delle risposte alle domande poste nel capitolo precedente.

11.1 Test 5: Uso di Voice-Over con Visual Studio Code

Il primo obiettivo della seconda serie di test è stato la gestione di un errore all'interno di un programma. Come già osservato nel Test numero 1, con Voice-Over e Visual-Studio siamo riusciti ad arrivare alla lettura dell'output di un programma scritto e creato, simulando di interagire col sistema come un soggetto non vedente. In caso di errore, Visual Studio mostra sul terminale riguardo la sua tipologia, mostrandoci poi un dato che sarà fondamentale alla risoluzione di questo problema, ovvero la linea di codice in cui si trova l'errore. Tramite la documentazione di Visual Studio Code, è possibile utilizzare il comando rapido disposnibile sull'editor [control + G], che permette l'apertura di una barra di ricerca nella quale può essere inserito un valore. Una volta premuto invio, l'editor porta il cursore dell' utente alla linea di codice esplicitata sulla barra di ricerca. Questa possibilità, combinara con gli esiti del test 1, permette di eseguire un programma e, in caso di errore, di leggere sul terminale la tipologia di errore insieme alla riga di codice in cui esso si trova e, tramite il comando rapido [control + G], di spostarci nel nostro codice alla riga selezionata per leggere tramite lo screen-reader il suo contenuto. Questo processo rende dunque possibile la correzione di errori sintattici ed eventuali errori semantici presenti nel codice ad utenti non-vedenti.[6] Il secondo obiettivo riguarda l'auto-compilazione, è possibile sfruttare il suo potenziale anche se l'utente non può leggere a schermo le varie opzioni che essa rende disponibili? In questo caso, la versione ottimizzata per screen reader di Visual Studio Code viene incontro a questo problema. Nel momento in cui qualcuno digita dei caratteri di una funzione o di una keyword nota del linguaggio in cui si sta programmando, l'autocompliazione si attiva e l'utente, mediante le frecce direzionali su e giù, può ascoltare tutti i suggerimenti dell'autocompliazione, premendo invio una volta trovato quello che cercava. In questo modo, tutti i vantaggi dell' autocompilazione risultano accessibili anche ad utenti non-vedenti senza grosse difficoltà. Prima di parlare di come si possa modificare il metodo di lettura di uno screen reader quando il contenuto è del codice scritto in un linguaggio di programmazione, è necessario prima porsi la seguente

[6] https://www.dropbox.com/s/is11td8b6hof1a5/gestione_errore.mov?dl=0

domanda: come si legge del codice? È una questione di intonazione, di enfasi che rende più comprensibile la lettura di un mio programma? La risposta emersa da questa domanda è stata no, perchè, ad esempio, se si dovesse spiegare ad una persona la stesura di un programma che implementa una funzione, variare l'enfasi o il tono in cui leggo le mie righe di codice non risulterebbe più comprensibile al mio interlocutore, invece la comprensione risulta maggiore se alcune espressioni venissero lette in un linguaggio simile a quello naturale. Per fare un esempio molto pratico: si deve spiegare ad una persona la creazione di una funzione media in Java, scritta in questo modo:

```
private int mean(List x){
    return(x[1]+x[2]+x[3]+x[4]) / len(x);
}
```

per spiegare questo codice ad un mio interlocutore direi "Ho creato una funzione di nome mean che prende in input una lista x e ritorna la somma degli elementi della lista diviso il numero di elementi della stessa". Da quest'osservazione, l'idea pensata per migliorare la lettura del codice da parte di uno screen reader è quella di tradurre la sintassi di un linguaggio programmazione in altre espressioni più vicine al linguaggio naturale. Facendo degli esempi sul linguaggio di programmazione "Java", si può tradurre il simbolo "=" con l'espressione "valore assegnato", oppure "System.out.println" con "chiamata a sistema sullo standard output, stampa e vai a capo". Traducendo in questo modo il nostro programma, si può rendere molto più comprensibile il contenuto di ogni riga di codice, sia dal punto di vista sintattico che semantico. Su VoiceOver, è possibile modificare la lettura delle parole in determinate applicazioni, ovvero posso inserire una parola, ad esempio "ciao", e far sì che ogni volta che lo screen reader vede la parola "ciao", legga "come stai?", facendo si che ciò si verifichi solo all'interno di Visual Studio Code. È stato dunque eseguito un nuovo test, nel quale sono state inserite nel menu delle pronuncie di VoiceOver le principali keyword del linguaggio Java, verificando poi se esse venissero lette correttamente nello sviluppo di un semplice programma. Il risultato di questo test è stato, come previsto, una maggior comprensione di alcune particolari righe di codice che, invece di essere lette solo come puro testo senza punteggiatura, veniva compreso dall'utente il costrutto del linguaggio ("questo è un ciclo, un array, stampa a video, valore di ritorno, assegnamento, variabile di tipo intero") rendendo ancora più accessibile la lettura da parte di utenti non vedenti. [7]

11.2 Test 6: Uso di Nvda Con Visual Studio Code

Il test numero 6 è stato un continuo del risultato ottenuto nel test numero 3, applicando gli stessi criteri ed obiettivi del test numero 5. L'unica differenza riguarda il sistema operativo e, di conseguenza, lo screen reader utilizzato. Fatta questa premessa, le soluzioni relative

[7] https://www.dropbox.com/s/74xb7qrmz9wr60h/test_voicevore_dizionario.mp4?dl=0

ai problemi di accessibilità dell'autocompilatore e della gestione di errori sono le medesime trovate in ambiente MacOS, poiché indipendenti dallo screen reader e risolte mediante gli strumenti di Visual Studio Code, il quale resta l'editor utilizzato in questo test. Quindi l'attenzione in questo test è rivolta all'ultimo dei nostri quesiti, è possibile, anche tramite Nvda, creare un dizionario che associ alle keywords di un linguaggio di programmazione, delle parole che forniscano una maggior comprensione del contenuto all'interno delle righe di codice, ottenendo un risultato il più vicino possibile al linguaggio naturale? La risposta è stata positiva, Nvda mette a disposizione dei dizionari personalizzabili che permettono all'utente di impostare un'associazione tra due parole diverse. Ma le potenzialità di Nvda non si fermano qui, infatti, rispetto a VoiceOver, l'associazione non viene eseguita solo su singole parole, ma può essere fatta su intere frasi, range di caratteri, sotto-stringhe, poichè i dizionari di Nvda possono essere programmati utlizzando una serie di caratteri speciali che NVDA mette a disposizione. [8]. Rispetto a VoiceOver, il risultato è stato più interessante, perché non esiste solo un associazione parola-parola, ma è possibile sostituire interi scheletri o macro di codice con delle parole (un esempio pratico: è stato possibile sostituire la lettura di "private static void main(String [] args)" con "blocco main del programma"). Il test è stato, come nel test 5, creare un programma scritto nel linguaggio Java e analizzare se alcuni blocchi statici del codice, come il main, la dichiarazione di una funzione, la stampa a video, risultassero maggiormente comprensibili rispetto alla semplice lettura di puro testo. A conclusione di questo test, il test numero 6 ha portato alla luce risultati più interessanti rispetto a quelli del test 5, che restano comunque soddisfacenti. Le potenzialità dei dizionari di Nvda, per essere comprese al meglio, necessitiano uno studio adeguato di come funzioni la programmazione delle sotto-stringhe, dei range di valori e di tutto ciò che si discosta da una semplice associazione parola-parola, ma già con una breve infarinatura abbiamo ottenuto una possibile soluzione al problema di partenza. [9]

12 Analisi dei plugin di Visual Studio Code a supporto di utenti con disabilità visiva

Dopo i test usando gli screen reader, si è voluto approfondire le capacità dell'editor Visual Studio Code, come ad esempio l'installazione di plugin. Oltre al fornire un supporto per l'installazione e la configurazione di diversi linguaggi di programmazione, l'aggiunta di estensioni offre una vastità di personalizzazioni disponibili all'utente, tra le quali delle impostazioni che migliorano l'interazione tra soggetto non-vedente e Visual Studio. Un plugin interessante che si ha avuto modo di testare è un modificatore dei colori dell'editor,

[8] https://www.nvda.it/i-tre-dizionari-di-nvda
[9] https://www.dropbox.com/s/4c7oru6ee5wirl9/test_dizionario_nvda.mp4?dl=0

chiamato “color blind” che, una volta scaricato, imposta i colori “per ipovedenti”, ovvero dei colori con un alto contrasto per facilitare la lettura di testo. Di questi plugin ne esistono diversi, divisi in base al tipo di difetto visivo (protanopia= no rosso, deuteranopia = no verde, tritanopia = no blu, monocromia = no rosso, verde, blu).

13 Possibilità di esportazione ed importazione di un dizionario

Il successivo step del progetto è stato rispondere alla domanda: ”è possibile sviluppare un dizionario specifico per un determinato linguaggio di programmazione e riuscire ad esportarlo?” In questo modo si potrebbe diffondere ed installare il dizionario creato in tutti i computer che supportano Nvda. Per quanto riguarda VoiceOver, la metodologia è piuttosto differente, infatti esso si limita ad una esportazione generale delle preferenze ipostate sul proprio calcolatore, permettendo tramite unità rimuovibili di poterle importare in altri computer MacOs. Al contrario, con Nvda è possbilie accedere ad uno dei 3 file che memorizzano i diversi dizionari disponibili allo screen reader. Questi file sono in formato .dic, ma è possibile accedervi in lettura e scrittura tramite il più comune editor di testo per Windows, il Blocco Note. All’interno di uno dei file, le varie impostazioni del dizionario sono salvate mediante stringhe e flag, che memorizzano le voci e le preferenze impostate nel dizionario di Nvda. Modificando direttamente il contenuto del dizionario dal file .dic, riavviando poi Nvda, è stato possibile scrivere delle nuove voci e visualizzarle nell’editor di Nvda. Grazie a questa possibiltà si può creare un file .txt contenente tutti i valori che si vogliono inserire nel dizionario, trasferirlo successivamente tra i diversi utenti e, una volta ricevuto, copiarne il contenuto nel proprio file .dic di Nvda. Dai risultati raccolti, è possibile in ambiente Windows poter creare una configurazione specifica di un dizionario ed esportarla o importarla tra i diversi calcolatori che possiedono lo screen reader Nvda.

13.1 Lettura di variabili e dichiarazioni mediante il dizionario

È noto a chi se ne intende un minimo di sviluppo di codice che, nei più noti linguaggi di programmazione, il nome che viene dichiarato per una variabile deve seguire delle precise regole sintattiche. Sfruttando le potenzialità di un dizionario, è sorta la domanda se fosse necessario inserire delle considerazioni riguardo la forma delle variabili, in modo tale che la lettura tramite il lettore schermo sia più fluente, precisa e chiara. Si è eseguito un test su un programma scritto nel lingauggio Java, inizializzando una variabile di tipo intero e attribuendoli nomei secondo criteri diversi. I casi testati sono stati 4:

- `int la_mia_variabile;`

- `int lamiavariabile;`

- `int la-mia-variabile;`

- `int laMiaVariabile;`

I risultati sono i seguenti:

- caso 1: Lo screen reader legge il carattere “trattino basso”, di conseguenza la lettura della variabile risulta molto confusa, decisamente non chiara. Nonostante sia possibile risolvere questo problema mediante i dizionari, i casi 2 e 4 risultano decisamente più fluidi e chiari.

- caso numero 2: La lettura è chiara, fluida, da l'impressione che stia leggendo il nome della variabile come un nome unico (scritto tutto in minuscolo e attacato è giusto che sia così) e per questo, rispetto al caso numero 4, vi è un leggero cambiamento del tono di voce.

- caso numero 3: i principali linguaggi di programmazione vedono la dichiarazione di una variabile composta da più nomi separati da un ”-” come un'errore sintattico, dunque non è possibile poter dichiarare una variabile in questo modo (anche se lo screen reader la leggerà in maniera corretta).

- caso numero 4: La lettura assume un tono di voce e di pronuncia leggermente diverso dal caso 2, ma resta comunque tra le letture di variabili più fluide e chiare.

In conclusione di questo test, possiamo affermare che una buona norma sulla dichiarazioni di variabili con nomi composti può essere quella di unire i nomi in unico nome tutto attaccato, senza caratteri speciali divisori, scegliendo se mettere le lettere inizali in maiuscolo oppure no, dal momento che la differenza si sostanzia in una semplice differenza di intonazione sulla lettura.[10]

14 Creazione di un dizionario Nvda per Python

In questo capitolo si è voluto creare un semplice dizionario che possa facilitare la programmazione, mediante lo screen reader di Nvda, in Python, uno dei linguaggi più noti e diffusi al mondo. Prima di descrivere come è avvenuta la creazione del dizionario, è opportuno fare una considerazione: per evitare problemi di mancata lettura di una nuova voce del dizionario, ogni operatore o keyword del linguaggio deve essere scritta tra due caratteri spazio, esempi: [a = 2] va bene, [a=2] non va bene, [for i in range(a,b)] va bene, [a = 3 * 2] va bene, [w = 2*2] non va bene. Questo è stato deciso per evitare che alcune

[10] https://www.dropbox.com/s/m5jitlc0mtqcqkk/nomi_variabili.mov?dl=0

sotto-voci andassero in conflitto con altre voci (esempio: [elif] viene interpretato leggendo “el”, più la voce dell'istruzione [if]). Questo dizionario è in grado di leggere le principale keyword e costrutti di Python tramite lo screen reader e riprodurre il risultato in base alla voce corrispondente impostata nel dizionario. Come anticipato nei test precedenti, Nvda è in grado di creare un dizionario capace di sfruttare i meta-caratteri e la loro combinazione per creare dei propri range di simboli che, in una determinata sequenza, possono corrispondere ad una e una sola voce del dizionario.

Azione elementare	Simbolo in Python	Voce di Nvda assegnata
Assegnamento variabile	=	”valore assegnato”
Stampa a video	print	”stampa a video”
Virgolette	”	”contenuto fra apici”
Commento	#	”commento su una riga”
Diviso	/	”diviso”
Più	+	”più”
Meno	-	”meno”
Per	*	”per”
Divisione percentuale	%	”diviso percentuale”
Divisione intera	//	”divisone intera”
Esponente	**	”esponente”
And	and	”operatore logico intersezione”
Or	or	”operatore logico unione”
Not	not	”operatore logico negazione”
Array	[	”contenuto di un array”
For	for	”ciclo for di indice”
Range	range	”intervallo”
Minore	<	”minore di”
Minore uguale	<=	”minore uguale di”
Maggiore	>	”maggiore di”
Maggiore uguale	>=	”maggiore uguale di”
Uguale	==	”uguale a”
While	while	”ciclo while di condizione”
If	ˆif — if	”ramo if con condizione”
Elif	elif	”ramo elif con condizione”
Else	else:	”ramo else”
True	True	”valore logico True”
False	False	”valore logico False”
Funzione	def	”definizione funzione”
Return	return	”valore di ritorno”
Dizionario	dict	”dizionario”
Incremento	+=	”incrementato di”
Decremento	-=	”decrementato di”
Main	main	”main del programma”
Classe	class	”classe del programma”

Tabella 1: Sintassi del dizionario Nvda per Python

È possibile vedere la differenza tra un programma Python letto da un screen reader senza l'implementazione del dizionario [11] e lo stesso programma in cui invece viene implementato.[12]

15 Analisi dell'editor Visual Studio Code Online

Nonostante l'utlizzo di editor di codice online fosse già stato testato e scartato poichè meno efficente rispetto agli editor installabili, è stata riaperta la questione per testare le funzionalità della versione di Visual Studio Code per Web. Questo editor possiede le principali opzioni della sua versione per desktop, non necessitando di un download o di un'installazione, inoltre, rispetto alla versione scaricabile, possiede l'opzione di poter creare un account personale, con il quale poter salvare il proprio codice e poter proseguire il lavoro su un'altro dispositivo. Se questa versione Web funzionasse come la versione Desktop, questa versione in Cloud offrirebbe una piattaforma in cui si può accedere allo stesso codice mediante diversi dispositivi, grazie al proprio account. Le principali opzioni della versione desktop sono disponibili anche nella versione Web, come ad esempio la molteplicità di linguaggi disponibili, la funzionalità di alcuni comandi rapidi, l'ottimizzazione dell'editor quando si utilizza uno screen reader, invece una prima grave mancanza riscontrata è l'impossibilità d'esecuzione o di debug del codice all'interno dell'editor web. Il secondo problema, leggermente meno grave, è la non capacità di lettura delle righe di codice, infatti è possibile leggere solamente parola per parola. Queste due mancanze, nonostante l'enorme potenziale di poter creare codice in cloud e poterci accedere mediante un qualunque dispositivo grazie al proprio account, rendono la versione web di Visual Studio Code parecchio penalizzata rispetto all'originale, poichè la possibilità di poter eseguire il codice scritto è un'esigenza fondamentale per un programmatore. Portata a termine questa fase di test, l'editor Visual Studio Code offre maggiori opzioni e preferenze nella sua versione per Desktop rispetto alla sua versione Online, la quale si limita ad essere uno strumento che permette di scrivere del codice, ma non di verificarne il suo corretto funzionamento.

16 Utlizzo di alberi sintattici del linguaggio

Dopo aver analizzato in maniera approfondita le potenzialità dei dizionari all'interno dei più noti screen reader, non siamo ancora in grado del tutto a dare una particolare enfasi ai costrutti ed alla sintassi di un particolare linguaggio di programmazione. Infatti quello che

[11] https://www.dropbox.com/s/ppe1yu590gspgps/non-dizionarionvda.mp4?dl=0
[12] https://www.dropbox.com/s/xo3aawt77jd1033/dizionario_nvda.mp4?dl=0

il dizionario effettivamente fa è uno "string-replacement", ovvero individua la sequenza di lettere che compongono una parola e ne associa una particolare traduzione da parte dello screen reader (esempio, individuo un ciclo for all'interno del mio programma poichè ho letto la combinazione di lettere 'f','o','r'). Quest'aspetto evidenzia diverse problematiche, come l'analisi di sotto-stringhe (caso di un 'elif', che contiene al suo interno la sequenza di lettere 'if', che corrisponde ad un'altra istruzione), la distinzione tra nomi di variabili, nomi di funzioni, nomi di liste, oppure l'apertura e chiusura di cicli, condizioni. Un'idea in grado di poter risolvere questo problema è sfruttare il parsing del codice. Il parsing si basa sull'analisi sintattica dei costrutti di un linguaggio, è un processo che analizza il flusso continuo di dati in ingresso (input letti, per esempio, da un file o una tastiera) in modo da determinare la correttezza della struttura del codice confrontandola con la grammatica del linguaggio. Poter utilizzare un parser, ovvero un programma che applica tecniche di parsing, può essere un metodo funzionante per analizzare del codice fornito in ingresso e, in base alla struttura e alla sintassi dei suoi particolari elementi, poter creare determinati eventi una volta individuato un determinato costrutto grammaticale del linguaggio, come la riscrittura di quel costrutto in linguaggio simile al naturale.

16.1 Creazione di un parser per il linguaggio Python

Dopo aver introdotto brevemente come funziona un parser, si vuole crearne uno per il linguaggio di programmazione Python, estendendo il codice in modo tale che sia possibile inserire in ingresso al programma il nome di un file Python, leggerne il contenuto, applicare l'albero di parsing al codice, gestendo per ogni costrutto del linguaggio la riscrittura di esso in un linguaggio più simile al naturale, in modo tale che, una volta completato il processo, ottengo una stringa contente una spiegazione dettagliata sul codice appena parsato, che riscriverò sul file originale come commento. Per costruire l'albero di parsing sono state sfruttate le potenzialità della libreria 'ast' di Python, che offre numerosi metodi per programmare un determinato evento quando il parser, durante il suo cammino, riscontra uno specifico costrutto del linguaggio (un ciclo, una costante, un'istruzione if, una variabile,...). Una volta ottenute queste funzionalità, è possibile implementare un parser che dato il nome di un file Python come input, restituisca come output una stringa nella quale sono state "tradotte" tutte le strutture sintattiche del linguaggio Python in un linguaggio simile al naturale, una vera e propria spiegazione del codice scritta sul file originale come commmento all'inizio del codice.

16.2 Testing del parser su un programma python

Una volta costruito quello che d'ora in poi chiameremo "parser avanzato", si è voluto testare le sue funzionalità su un programma scritto in Python, in modo da visualizzare l'output come commento inserito all'interno del codice. Il nome del programma su cui è

stato fatto il test è "file.py", che verifica su una serie di valori di un ciclo [for] quale tra questi è pari e quali tra questi è dispari. Il codice è il seguente:

```
def myfun(s):
    if s % 2 == 0:
        print("pari")
    else:
        print("dispari")

def main():
    for i in range(0, 10):
        myfun(i)
```

Per eseguire il parser è necessario chiamare su una shell il nome del file insieme a quello del file che si vuole "tradurre", ad esempio:

```
python3 parser.py file.py
```

Una volta lanciato in esecuzione il programma, in cima al codice del file tradotto troviamo il seguente commmento:

```
'''
Definizione della funzione myfun, con parametri: s, .
Istruzione if, Con condizione, s, variabile. modulo 2, costante. uguale a 0, costante. .
Chiamata di funzione print , parametri pari, costante. .
Fine istruzione if.
istruzione else,
Chiamata di funzione print, parametri dispari, costante. .
Fine condizione else.
Fine corpo della funzione myfun.
Definizione della funzione main, nessun parametro.
ciclo For, con indice i, variabile. nell' intervallo: start uguale a 0,
stop uguale a 10, step uguale a 1. .
Chiamata di funzione myfun , parametri i, variabile. .
fine ciclo For.
Fine corpo della funzione main.
'''
```

Questa spiegazione risulta essere più vicina al linguaggio naturale rispetto alla scrittura del codice. Si nota subito la presenza di molta punteggiatura all'interno del commento, ciò è dovuto al fatto che i punti e le virgole sono i caratteri che gestiscono maggiormente l'enfasi sulle parole e sulle pause, in modo tale che il sintetizzatore vocale legga il contenuto del commento in maniera chiara e non troppo veloce, per evitare di confondere l'utente.

16.3 Codice del parser avanzato

Il seguente codice è il corpo del parser avanzato costruito, esso fa riferimento all'ultimo aggiornamento che verrà trattato negli ultimi capitoli di questa tesi.

```
# Librerie necessarie
import ast
import io
import os.path
import tokenize
from sys import argv
import pyttsx3
import sys

# inizializzazione del sintetizzatore
engine = pyttsx3.init()
# in questa stringa memorizzo il contenuto del parsing del codice Python in ingresso
parsing_trace = ""


# i commenti non vengono rilevati dall'albero di parsing
# con questo metodo vengono estratti dal codice
def extract_comment(source_code):
    comments = []
    for token in tokenize.tokenize(io.BytesIO(source_code.encode("utf-8")).readline):
        if token.type == tokenize.COMMENT:
            comm = token.string.strip()
            if comm.startswith("#"):
                comments.append(comm)
    return comments


# classe che definisce tutte le funzioni eseguire il parsing del mio codice
class NodeVisitor(ast.NodeVisitor):

    # trigger di un nome di una variabile
    def visit_Name(self, node):
        global parsing_trace
        parsing_trace += f"{node.id}, variabile, . "
        self.generic_visit(node)

    # trigger di un valore costante (numero, carattere, stringa)
    def visit_Constant(self, node):
        global parsing_trace
        parsing_trace += f"{node.value}, costante, . "
```

```
# trigger di nomi costanti del linguaggio
def visit_NameConstant(self, node):
    global parsing_trace
    # gestione dei valori booleani
    if node.value in [True, False]:
        parsing_trace += f"Valore logico {node.value}, "

# trigger su una espressione
def visit_Expr(self, node):
    # se l'espressione    una costante
    if isinstance(node, ast.Constant):
        visitor.visit_Constant(node)
    # se l'espressione    un assegnamento
    elif isinstance(node, ast.Assign):
        visitor.visit_Assign(node)
    # se l'espressione    una comparazione
    elif isinstance(node, ast.Compare):
        visitor.visit_Compare(node)
    # se l'espressione    una definizione di funzione
    elif isinstance(node, ast.FunctionDef):
        visitor.visit_FunctionDef(node)
    # se l'espressione    una chiamata di funzione
    elif isinstance(node, ast.Call):
        visitor.visit_Call(node)
    # se l'espressione    un operatore di assegnamento
    elif isinstance(node, ast.AugAssign):
        visitor.visit_AugAssign(node)
    # se l'espressione    una variabile
    elif isinstance(node, ast.Name):
        visitor.visit_Name(node)
    # se l'espressione    un costrutto if
    elif isinstance(node, ast.If):
        visitor.visit_If(node)
    # se l'espressione    un ciclo For
    elif isinstance(node, ast.For):
        visitor.visit_For(node)
    # se l'espressione    un ciclo While
    elif isinstance(node, ast.While):
        visitor.visit_While(node)
    # se l'espressione    un array
    elif isinstance(node, ast.List):
        visitor.visit_List(node)
    # se l'espressione    un elemento di un array
    elif isinstance(node, ast.Subscript):
        visitor.visit_Subscript(node)
    # se l'espressione    un costrutto Try-except
```

```
        elif isinstance(node, ast.Try):
            visitor.visit_Try(node)
        # se l'espressione    un import di una libreria
        elif isinstance(node, ast.Import):
            visitor.visit_Import(node)
        # se l'espressione    un istruzione yield
        elif isinstance(node, ast.Yield):
            visitor.visit_Yield(node)
        # se l'espressione    un operazione bin
        elif isinstance(node, ast.BinOp):
            visitor.visit_BinOp(node)
        # se l'espressione    un operazione bool
        elif isinstance(node, ast.BoolOp):
            visitor.visit_BoolOp(node)
        # se l'espressione    l'istanza di una classe
        elif isinstance(node, ast.ClassDef):
            visitor.visit_ClassDef(node)
        # se l'espressione    la definizione di un dizionario
        elif isinstance(node, ast.Dict):
            visitor.visit_Dict(node)
        # se l'espressione    un istruzione pass
        elif isinstance(node, ast.Pass):
            visitor.visit_Pass(node)
        # se l'espressione    un metodo di una classe
        elif isinstance(node, ast.Attribute):
            visitor.visit_Attribute(node)
        # se l'espressione    una espressione: valuto la sotto espressione
        elif isinstance(node, ast.Expr):
            visitor.visit_Expr(node.value)
        # quando la condizione    semplicemente o True o False
        elif node.value is True or node.value is False:
            visitor.visit_NameConstant(node)
        # gestione di sotto-espressione
        else:
            # sotto espressione    una chiamata di funzione
            if isinstance(node.value, ast.Call):
                visitor.visit_Call(node.value)
            # sotto espressione    una istruzione yield
            if isinstance(node.value, ast.Yield):
                visitor.visit_Yield(node.value)
            # casi non ancora gestiti
            else:
                self.generic_visit(node.value)

    # trigger operatori di comparazione
    def visit_Compare(self, node):
        global parsing_trace
```

```
    # se il valore a sinistra dell'operatore    una costante
    if isinstance(node.left, ast.Constant):
        visitor.visit_Constant(node.left)
    # se il valore a sinistra dell'operatore    una variabile
    elif isinstance(node.left, ast.Name):
        visitor.visit_Name(node.left)
    # se il valore a sinistra    un insieme di operazioni
    elif isinstance(node.left, ast.BinOp):
        visitor.visit_BinOp(node.left)
    # se il valore a sinistra    una chiamata di funzione
    elif isinstance(node.left, ast.Call):
        visitor.visit_Call(node.left)

    for op in node.ops:
        # operatore di uguaglianza  ==
        if isinstance(op, ast.Eq):
            parsing_trace += f"uguale a "
        # operatore maggiore >
        if isinstance(op, ast.Gt):
            parsing_trace += f"maggiore di "
        # operatore maggiore uguale >=
        if isinstance(op, ast.GtE):
            parsing_trace += f"maggiore uguale di "
        # operatore minore <
        if isinstance(op, ast.Lt):
            parsing_trace += f"minore di "
        # operatore minore uguale <=
        if isinstance(op, ast.LtE):
            parsing_trace += f"minore uguale di "

    # prendo il valore a destra dell'operatore
    comparators = node.comparators
    right = comparators[0]
    # se il valore a destra dell'operatore    una costante
    if isinstance(right, ast.Constant):
        visitor.visit_Constant(right)
    # se il valore a destra dell'operatore    una variabile
    elif isinstance(right, ast.Name):
        visitor.visit_Name(right)
    # se il valore a destra    un insieme di operazioni
    elif isinstance(right, ast.BinOp):
        visitor.visit_BinOp(right)
    # se il valore a destra    una chiamata di funzione
    elif isinstance(right, ast.Call):
        visitor.visit_Call(right)

# trigger di un ciclo for
```

```
def visit_For(self, node):
    global parsing_trace
    parsing_trace += f"ciclo For, "
    # indice del mio ciclo
    parsing_trace += f" con indice "
    visitor.visit_Expr(node.target)
    # Gestione condizione del ciclo for
    iter_ = node.iter
    if isinstance(iter_, ast.Call) and isinstance(iter_.func, ast.Name):
        # se utilizzo la funzione range per iterare, faccio il parsing di start, sto
        if iter_.func.id == 'range':
            parsing_trace += f" nell'intervallo: "
            start = iter_.args[0].n if len(iter_.args) >= 1 else 0
            stop = iter_.args[1].n if len(iter_.args) >= 2 else None
            step = iter_.args[2].n if len(iter_.args) >= 3 else 1
            parsing_trace += f" start uguale a {start}, stop uguale a {stop}, step u
        # altrimenti analizzo l'espressione fornita
    # se itero su una Lista, espressione, metodo, faccio il parsing dell' espression
    else:
        parsing_trace += f"Iterando su: "
        visitor.visit_Expr(iter_)

    parsing_trace += ".\n"

    # corpo del ciclo
    for el in node.body:
        # faccio il parsing su tutte le espressioni del corpo
        visitor.visit_Expr(el)

    parsing_trace += "fine ciclo For. \n"

# trigger di un ciclo while
def visit_While(self, node):
    global parsing_trace
    parsing_trace += f"ciclo While "
    # condizione del ciclo
    parsing_trace += f"con condizione "
    visitor.visit_Expr(node.test)
    parsing_trace += ".\n"
    # corpo del ciclo
    for el in node.body:
        # faccio il parsing su tutte le espressioni del corpo
        visitor.visit_Expr(el)

    parsing_trace += "\n"
    parsing_trace += "fine ciclo While.\n"
```

```
# trigger di un argomento di funzione
def visit_arg(self, node):
    global parsing_trace
    parsing_trace += f"{node.arg}, "

# trigger di una stringa composta da variabili
def visit_JoinedStr(self, node):
    # faccio il parsing di ogni valore che compone la stringa
    for el in node.values:
        # se il contenuto    composto da una costante
        if isinstance(el, ast.Constant):
            visitor.visit_Constant(el)
        # casi generici
        else:
            self.generic_visit(el)

# trigger di una definizione di funzione
def visit_FunctionDef(self, node):
    global parsing_trace
    # definizione funzione senza alcun parametro
    if not node.args.args:
        parsing_trace += f"Definizione della funzione {node.name}, nessun parametro.
    # definizione funzione con parametri passati in ingresso
    else:
        parsing_trace += f"Definizione della funzione {node.name}, con parametri: "
        for arg in node.args.args:
            visitor.visit_arg(arg)
        parsing_trace += ".\n"

    # parsing del corpo della funzione, insieme di espressioni
    for el in node.body:
        # se il corpo    un istruzione return
        if isinstance(el, ast.Return):
            visitor.visit_Return(el)
        # casi generici
        else:
            visitor.visit_Expr(el)

    parsing_trace += f"Fine corpo della funzione {node.name}. \n"

# trigger di una chiamata di funzione
def visit_Call(self, node):
    global parsing_trace
    # nome della funzione
    if isinstance(node.func, ast.Name):
        parsing_trace += f'Chiamata di funzione '
        for t in node.func.id:
```

```
            parsing_trace += t
        parsing_trace += "␣"
        parsing_trace += f",␣parametri:␣"
        # gestisco i parametri passati alla funzione
        for arg in node.args:
            # il parametro    un nome di variabile
            if isinstance(arg, ast.Name):
                visitor.visit_Name(arg)
            # il parametro contiene un operatore
            elif isinstance(arg, ast.BinOp):
                visitor.visit_BinOp(arg)
            # il parametro    una costante
            elif isinstance(arg, ast.Constant):
                visitor.visit_Constant(arg)
            # il parametro    una chiamata di una seconda funzione
            elif isinstance(arg, ast.Call):
                visitor.visit_Call(arg)
            # il parametro    una lista, array
            elif isinstance(arg, ast.Subscript):
                visitor.visit_Subscript(arg)
            # il parametro    una stringa composta da variabili (es : f"")
            elif isinstance(arg, ast.JoinedStr):
                visitor.visit_JoinedStr(arg)
            # Parametro non gestito
            else:
                # se un parametro non    gestito, stampo la sua tipologia
                print(arg)
        # Se non viene passato nessun parametro
        if not node.args:
            parsing_trace += f"vuoto.␣"
        parsing_trace += f".\n"

# trigger operatori
def visit_BinOp(self, node):
    global parsing_trace

    # se il valore a sinistra dell'operatore    una costante
    if isinstance(node.left, ast.Constant):
        visitor.visit_Constant(node.left)
    # se il valore a sinistra dell'operatore    una variabile
    elif isinstance(node.left, ast.Name):
        visitor.visit_Name(node.left)
    # se il valore a sinistra dell'operatore    una funzione
    elif isinstance(node.left, ast.Call):
        visitor.visit_Call(node.left)

    parsing_trace += f"operatore␣"
```

```
        # operatore +
        if isinstance(node.op, ast.Add):
            parsing_trace += f"somma con "
        # operatore -
        elif isinstance(node.op, ast.Sub):
            parsing_trace += f"sottrazione con "
        # operatore *
        elif isinstance(node.op, ast.Mult):
            parsing_trace += f"moltiplicazione con "
        # operatore /
        elif isinstance(node.op, ast.Div):
            parsing_trace += f"divisione con "
        # operatore %
        elif isinstance(node.op, ast.Mod):
            parsing_trace += f"modulo della divisione con "
        # operatore potenza
        elif isinstance(node.op, ast.Pow):
            parsing_trace += f"potenza con esponente "

        # se il valore a destra dell'operatore    una costante
        if isinstance(node.right, ast.Constant):
            visitor.visit_Constant(node.right)
        # se il valore a destra dell'operatore    una variabile
        elif isinstance(node.right, ast.Name):
            visitor.visit_Name(node.right)
        # se il valore a destra dell'operatore    una funzione
        elif isinstance(node.right, ast.Call):
            visitor.visit_Call(node.right)

# trigger di una tupla
def visit_Tuple(self, node):
    for el in node.elts:
        visitor.visit_Expr(el)

# trigger assegnamento variabile
def visit_Assign(self, node):
    global parsing_trace
    parsing_trace += f"Assegnamento a , "
    # scansione dei target di assegnamento
    for target in node.targets:
        # se il target    una variabile
        if isinstance(target, ast.Name):
            visitor.visit_Name(target)
        # se il target    una tupla di variabili
        if isinstance(target, ast.Tuple):
            visitor.visit_Tuple(target)
```

```
            # se il target    un elemento di un array
            if isinstance(target, ast.Subscript):
                visitor.visit_Subscript(target)

        parsing_trace += f"il valore "
        # il valore assegnato    un espressione con operatori
        if isinstance(node.value, ast.BinOp):
            visitor.visit_BinOp(node.value)
        # il valore assegnato    una variabile
        elif isinstance(node.value, ast.Name):
            visitor.visit_Name(node.value)
        # il valore assegnato    una costante
        elif isinstance(node.value, ast.Constant):
            visitor.visit_Constant(node.value)
        # il valore assegnato    una lista
        elif isinstance(node.value, ast.List):
            visitor.visit_List(node.value)
        # il valore assegnato    una funzione
        elif isinstance(node.value, ast.Call):
            visitor.visit_Call(node.value)
        # se il valore assegnato    una tupla di valori
        if isinstance(node.value, ast.Tuple):
            visitor.visit_Tuple(node.value)
        # se il valore assegnato    un dizionario
        if isinstance(node.value, ast.Dict):
            visitor.visit_Dict(node.value)
        # se il valore assegnato    un elemento di un array
        if isinstance(node.value, ast.Subscript):
            visitor.visit_Subscript(node.value)


        parsing_trace += ".\n"

    # trigger clausola if-else
    def visit_If(self, node):
        global parsing_trace

        if isinstance(node, ast.If):
            # se siamo nel corpo del if
            parsing_trace += f"Istruzione if, "
            # prima faccio il parsing della condizione del if
            parsing_trace += f"Con condizione, "
            # se la condizione    un' operazione di confronto
            if isinstance(node.test, ast.Compare):
                visitor.visit_Compare(node.test)
            # se la condizione    un nome costante (es: True, False)
            elif isinstance(node.test, ast.NameConstant):
```

```
                visitor.visit_NameConstant(node.test)
            # se la condizione    una variabile
            elif isinstance(node.test, ast.Name):
                visitor.visit_Name(node.test)
            # se la condizione    una costante
            elif isinstance(node.test, ast.Constant):
                visitor.visit_Constant(node.test)
            # se la condizione    una chiamata di funzione
            elif isinstance(node.test, ast.Call):
                visitor.visit_Call(node.test)
            # se la condizione    un'espressione
            elif isinstance(node.test, ast.Expr):
                visitor.visit_Expr(node.test)
            elif isinstance(node.test, ast.BoolOp):
                visitor.visit_BoolOp(node.test)
            # casi da gestire
            else:
                print(node.test)

            parsing_trace += ".\n"
            # poi faccio il parsing del corpo, che    un insieme di espressioni
            for body in node.body:
                visitor.visit_Expr(body)

            parsing_trace += f"Fine istruzione if, .\n"

        # se vi    anche un istruzione elif, quanti ce ne sono
        if len(node.orelse) > 0 and isinstance(node.orelse[0], ast.If):
            if isinstance(node.orelse[0], ast.If):
                # chiamo una funzione ricorsiva che visiti tutti gli elif presenti
                self.elif_if_tree(node)

        # se vi    anche un istruzione else senza elif precedenti
        else:
            # else non ha condizioni
            parsing_trace += f"Istruzione else, \n"
            # faccio il parsing del corpo, insieme di espressioni
            for body in node.orelse:
                visitor.visit_Expr(body)

            parsing_trace += f"Fine condizione else.\n"

    def elif_if_tree(self, node):
        global parsing_trace
        parsing_trace += f"Istruzione elif, "
        # prima faccio il parsing della condizione del elif
        parsing_trace += f"Con condizione, "
```

```
        # se la condizione    un' operazione di confronto
        if isinstance(node.orelse[0].test, ast.Compare):
            visitor.visit_Compare(node.orelse[0].test)
        # se la condizione    un nome costante (es: True, False)
        elif isinstance(node.orelse[0].test, ast.NameConstant):
            visitor.visit_NameConstant(node.orelse[0].test)
        # se la condizione    una variabile
        elif isinstance(node.orelse[0].test, ast.Name):
            visitor.visit_Name(node.orelse[0].test)
        # se la condizione    una costante
        elif isinstance(node.orelse[0].test, ast.Constant):
            visitor.visit_Constant(node.orelse[0].test)
        # se la condizione    una chiamata di funzione
        elif isinstance(node.orelse[0].test, ast.Call):
            visitor.visit_Call(node.orelse[0].test)
        # se la condizione    un'espressione
        elif isinstance(node.orelse[0].test, ast.Expr):
            visitor.visit_Expr(node.orelse[0].test)
        elif isinstance(node.orelse[0].test, ast.BoolOp):
            visitor.visit_BoolOp(node.orelse[0].test)
        # casi da gestire
        else:
            print(node.orelse[0].test)

        parsing_trace += ".\n"
        # poi faccio il parsing del corpo, che    un insieme di espressioni
        for body in node.orelse[0].body:
            visitor.visit_Expr(body)

        parsing_trace += f"Fine istruzione elif, .\n"

        # se vi    un ulteriore elif di seguito a questo
        if len(node.orelse[0].orelse) > 0 and isinstance(node.orelse[0].orelse[0], ast.I
            self.elif_if_tree(node.orelse[0])
        # se vi    un else dopo questo elif
        elif len(node.orelse[0].orelse) > 0 and isinstance(node.orelse[0].orelse[0], ast
            # else non ha condizioni
            parsing_trace += f"Istruzione else, \n"
            # faccio il parsing del corpo, insieme di espressioni
            for body in node.orelse[0].orelse:
                visitor.visit_Expr(body)

            parsing_trace += f"Fine condizione else.\n"

    # trigger degli operatori booleani
    def visit_BoolOp(self, node):
        global parsing_trace
```

```
    # se il valore a sinistra dell'operatore    una costante
    if isinstance(node.values[0], ast.Constant):
        visitor.visit_Constant(node.values[0])
    # se il valore a sinistra dell'operatore    una variabile
    if isinstance(node.values[0], ast.Name):
        visitor.visit_Name(node.values[0])
    # se il valore a sinistra dell'operatore    una funzione
    if isinstance(node.values[0], ast.Call):
        visitor.visit_Call(node.values[0])

    # operatore and
    if isinstance(node.op, ast.And):
        visitor.visit_And(node.op)
    # operatore or
    elif isinstance(node.op, ast.Or):
        visitor.visit_Or(node.op)
    # operatore not
    elif isinstance(node.op, ast.Not):
        visitor.visit_Not(node.op)

    # se il valore a destra dell'operatore    una costante
    if isinstance(node.values[1], ast.Constant):
        visitor.visit_Constant(node.values[1])
    # se il valore a destra dell'operatore    una variabile
    if isinstance(node.values[1], ast.Name):
        visitor.visit_Name(node.values[1])
    # se il valore a destra dell'operatore    una funzione
    if isinstance(node.values[1], ast.Call):
        visitor.visit_Call(node.values[1])

# trigger operatore AND
def visit_And(self, node):
    global parsing_trace
    parsing_trace += f"Operatore Booleano And, "
    self.generic_visit(node)

# trigger operatore OR
def visit_Or(self, node):
    global parsing_trace
    parsing_trace += f"Operatore Booleano Or, "
    self.generic_visit(node)

# trigger operatore NOT
def visit_Not(self, node):
    global parsing_trace
    parsing_trace += f"Operatore Booleano Not, "
    self.generic_visit(node)
```

```
# trigger di una variabile globale
def visit_Global(self, node):
    global parsing_trace
    parsing_trace += f"Variabile globale {node.names}. \n"
    self.generic_visit(node)

# trigger istruzione return di una funzione
def visit_Return(self, node):
    global parsing_trace
    parsing_trace += f"Valore di ritorno, "
    self.generic_visit(node)
    parsing_trace += ".\n"

# trigger su una creazione di una lista
def visit_List(self, node):
    global parsing_trace
    # definizione lista
    parsing_trace += f"Array "
    # se il contenuto della lista    vuoto
    if len(node.elts) == 0:
        parsing_trace += f"Con contenuto vuoto."
    # se il contenuto    un insieme di valori
    else:
        parsing_trace += f"Di elementi: "
        for el in node.elts:
            # se l'elemento    una costante
            if isinstance(el, ast.Constant):
                visitor.visit_Constant(el)
            # se l'elemento    una variabile
            elif isinstance(el, ast.Name):
                visitor.visit_Name(el)
        print(", .")

# trigger su un elemento di una lista
def visit_Subscript(self, node):
    global parsing_trace
    # indice della lista
    parsing_trace += "elemento di indice "
    # se l'indice    una costante
    if isinstance(node.slice, ast.Constant):
        visitor.visit_Constant(node.slice)
    # se l'indice    una variabile
    if isinstance(node.slice, ast.Name):
        visitor.visit_Name(node.slice)
    # nome della lista
    parsing_trace += "dell' array "
```

```
        visitor.visit_Name(node.value)

    # trigger dell' operatore try-except
    def visit_Try(self, node):
        global parsing_trace
        # nome istruzione
        parsing_trace += f"Istruzione Try, \n"
        # itero sul corpo del try
        for el in node.body:
            visitor.visit_Expr(el)
        parsing_trace += f"Fine Istruzione Try. \n"

        # istruzione except
        parsing_trace += f"Istruzione Except, "
        # nome del exception catturata dal handler
        parsing_trace += f"Con Handler di errore di tipo: "
        visitor.visit_ExceptHandler(node.handlers[0])
        parsing_trace += ", \n"
        # corpo del except
        for el in node.handlers[0].body:
            visitor.visit_Expr(el)
        parsing_trace += f"Fine Istruzione Except. \n"

    # trigger del tipo di exceptionHandler
    def visit_ExceptHandler(self, node):
        global parsing_trace
        # Gestione del nome del Handler
        parsing_trace += f"{node.type.id} "

    # trigger degli operatori assegnamento
    def visit_AugAssign(self, node):
        global parsing_trace
        # valore a sinistra dell' operatore
        visitor.visit_Name(node.target)

        # trigger operatore incremento
        if isinstance(node.op, ast.Add):
            parsing_trace += f"incrementato di, "
        # trigger operatore decremento
        if isinstance(node.op, ast.Sub):
            parsing_trace += f"decrementato di , "

        # valore a destra dell' operatore
        # se il valore e' una costante
        if isinstance(node.value, ast.Constant):
            visitor.visit_Constant(node.value)
        # se il valore e' una variabile
```

```
        elif isinstance(node.value, ast.Name):
            visitor.visit_Name(node.value)

        parsing_trace += "\n"

    # trigger del nome di una libreria
    def visit_alias(self, node):
        global parsing_trace
        parsing_trace += node.name

    # tigger sull'importazione di una libreria
    def visit_Import(self, node):
        global parsing_trace
        parsing_trace += f"Importazione della libreria: "
        for name in node.names:
            visitor.visit_alias(name)
        parsing_trace += ".\n"

    # trigger sull' istruzione Yield
    def visit_Yield(self, node):
        global parsing_trace
        parsing_trace += f"Istruzione yield con: "
        visitor.visit_Expr(node.value)
        parsing_trace += ", . \n"

    # trigger sulla definizione di una classe
    def visit_ClassDef(self, node):
        global parsing_trace
        # istanza della classe e nome della classe
        parsing_trace += f"Definizione della classe: {node.name}. \n"
        parsing_trace += f"corpo della classe: \n"
        # parsing del corpo della classe
        for el in node.body:
            visitor.visit_Expr(el)
        parsing_trace += f"Fine corpo della classe: {node.name}. \n"

    # trigger sulla definizione di un dizionario
    def visit_Dict(self, node):
        global parsing_trace
        # definizione di un dizionario
        parsing_trace += f"Dizionario, con valori: ,"
        # corpo del dizionario
        for key, value in zip(node.keys, node.values):
            parsing_trace += f". Nome Chiave: , "
            visitor.visit_Constant(key)
            parsing_trace += f" Valore assegnato: , "
            visitor.visit_Constant(value)
```

```
        parsing_trace += f"Fine definizione dizionario."

    # trigger sull'invocazione di un metodo di una classe
    def visit_Attribute(self, node):
        global parsing_trace
        parsing_trace += f"Metodo della classe o libreria: {node.value.id}, "
        parsing_trace += f"Con nome: ,,, {node.attr} ."

    # trigger su un istruzione pass
    def visit_Pass(self, node):
        global parsing_trace
        # pass: lo screen reader non lo annuncia
        parsing_trace += ""



if __name__ == '__main__':
    print(argv[3])
    # controllo che sia stato passato il nome di un file
    if len(sys.argv) < 3:
        print("Nome file non esplicitato")
    else:
        # prendo la selezione di testo evidenziato
        code = argv[3]

        try:
            # faccio il parsing del blocco di codice
            tree = ast.parse(str(code).strip())
        except:
            # il blocco di codice passato e' parziale rispetto alla sua sintassi:
            # es: passare la riga di condizione del if senza il suo corpo, o solo l'istr
            engine.say("Il blocco di codice passato e' parziale, e' necessario passare u
            engine.runAndWait()
            exit(1)

        visitor = NodeVisitor()
        visitor.visit(tree)

        # stampo il parsing ottenuto
        print(parsing_trace)
        # riproduco con il sintetizzatore il parsing ottenuto
        engine.say(parsing_trace)

        # stampa dei commenti trovati nel codice
        comments_in_code = extract_comment(code)
        if(len(comments_in_code) > 0):
            engine.say("\nCommenti trovati all'interno del codice: \n")
```

```
        print(comments_in_code)
        # Lettura dei commenti trovati nel codice
        for comment in comments_in_code:
            engine.say("Nuovo commento, contenuto: , ")
            engine.say(comment + ".\n")

    engine.runAndWait()
```

16.4 Uso della libreria 'pyttsx3' di Python

Se si analizza il codice del parser avanzato, si può notare l'utilizzo della libreria 'pyttsx3', la cui utilità non è ancora stata spiegata. Essa mette a disposizione dei metodi per sfruttare il sintetizzatore vocale del proprio sistema per leggere del codice fornito in ingresso argomento di una stringa. Combinato col parser avanzato, è possibile, una volta ottenuta la stringa output del parsing, leggerne il contenuto tramite i metodi della libreria 'pyttsx3'. Quest'approccio permette di implementare alcune funzionalità di uno screen reader per il codice del file passato in ingresso al parser avanzato, con diversi pro e contro rispetto agli screen reader noti e analizzati precedentemente. Tra i principali vantaggi della libreria 'pyttsx3' abbiamo la sua indipendenza dal sistema operativo, infatti necessita solo dell'installazione di Python, inoltre è gratuito ed open-source, dal momento che sia i metodi del parser che del sintetizzatore vocale possono essere programmati in base alle esigenze dell'utente. Questo approccio funziona su qualsiasi ambiente di sviluppo per python, non è necessario alcun particolare IDE. Gli svantaggi più noti sono il non utilizzo in real-time, ovvero la libreria legge il contenuto passatogli in ingresso solo quando il programma viene mandato in esecuzione, perdiamo la capacità di muovere il cursore ed ascoltarre la traduzione di una determinata linea come avveniva con gli screen reader. Fatte queste considerazioni, il parser, insieme alla liberia scoperta in questo sotto capitolo copre le mancanze trovate nell'utilizzo di uno screen reader con il suo dizionario, anche se una solzione ancora più efficente potrebbe essere l'utilizzo di entrambe le tecniche, combinandole secondo le proprie esigenze, per avvicinarci sempre di più ad una capacità di analisi e comprensione del codice più efficente per un utente con difficoltà visive. [13]

17 Ulteriori considerazioni

L'ultimo punto sull'analisi delle tecnologie a supporto dello sviluppo di codice da parte di utenti con disabilità visive ha portato risultati mai raggiunti fino ad ora, grazie alla potenza e all'efficenza dell'analisi tramite il parsing di codice. Il risultato però è ancora grezzo, ovvero, per eseguire il parsing del codice e la rispettiva lettura mediante il sintetizzatore vocale, dobbiamo eseguire il file parser.py dal terminale, passando come input

[13] https://www.dropbox.com/s/vofggt41z8mgy78/test_python_parser.mov?dl=0

l'intero file. Vogliamo migliorare questo processo in modo da rendere il più accessibile possibile l'utilizzo del parser avanzato, in modo che possa essere eseguito, per esempio, tramite un semplice combinazione di tasti sulla tastiera.

18 Utilizzo del parser con Visual Studio Code

Per rendere il processo di parsing più accessibile all'utente non-vedente, si è tornati a sfruttare alcune proprietà dell'editor Visual Studio Code, in particolare, questo IDE ci offre la possibilità di creare dei processi (programmi in esecuzione) richiamabili tramite un comando rapido. In questo modo, risulta possibile eseguire il codice che fa il parsing avanzato di programmi scritti in Python ed ottenerne il risultato semplicemente attraverso una determimanata combinazione di tasti. Per fare ciò, è necessario configurare dei determinati file presenti nel editor secondo dei determinati criteri.

18.1 Configurazione del file "task.json"

La prima configurazione da implementare è creare un Task all'interno dell'editor, che potrà poi essere richiamato senza dover eseguire e compilare il file "parser.py" ogni volta che si intende fare il parsing avanzato del codice. Un Task può essere richiamato dall'editor mediante una combinazione di tasti rapidi. Per compiere questo processo su Visual Studio Code, sulla schermata dell' editor in cui abbiamo il file di cui si vuole fare il parsing, premiamo i tasti [Ctrl + Shift +] su Windows o [Cmd + Shift + P] su mac, per aprire la 'barra dei comandi'. Sulla barra, digitare "Configure Tasks", scegliendo l'opzione "Tasks: Configure Tasks" dal menù in discesa che appare. Sarà poi necessario cliccare "Create tasks.json file from template", una volta cliccato, apparirà una finestra in cui si dovrà cliccare su "Others" come tipo di attività. Dopo aver seguito questo processo, troveremo di fronte a noi il file "task.json", il cui contenuto potrebbe essere vuoto o meno. Il file "task.json" dovrà essere configurato come segue:

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "Print selected line",
            "type": "shell",
            "command": "python",
            "args": [
                "print_selected_line.py",
                "${file}",
                "${lineNumber}",
                "'${selectedText}'"
            ],
```

```
            "presentation": {
                "reveal": "silent"
            },
            "group": {
                "kind" : "test",
                "isDefault" : true
            }
        }
    ]
}
```

I campi principali di questo file sono: “label”: il nome del Task, “command”: il linguaggio del contenuto del Task, “args”: gli argomenti che il Task prende come input quando viene eseguito sul determinato file su cui eseguire il parsing. In particolare “print-selected-line.py” è il nome del file parser, “file”, il nome del file su cui eseguire il parsing, “lineNumber” il numero di riga da cui parte il parsing, “selectedText” il testo evidenziato dal cursore (verrà spiegato successivamente il suo funzionamento) . Una volta configurato il file “task.json”, salvare e chiudere il file.

18.2 Configurazione del file “keybindings.json”

Vi è un secondo file che necessita essere configurato prima di ottenere il risultato voluto, ovvero impostare una combinazione di tasti rapidi per eseguire il Task creato precedentemente. Per fare ciò su Visual Studio Code bisogna: dalla finestra dell’ editor in cui visualizzo il file su cui fare il parsing, premere la combinazione di tasti [Ctrl+Shift+P] su Windows, o [Cmd+Shift+P] su mac, per aprire la barra dei comandi, digitare: “Open Keyboard Shortcuts”. Scegliere poi l’opzione “Preferences: Open Keyboard ShortCuts(JSON)”. Apparirà ora il file “keybindings.json”, il cui contenuto può esssere vuoto o meno. Inserire all’interno del file il seguente codice:

```
[
    {
        "key": "ctrl+shift+x",
        "command": "workbench.action.tasks.runTask",
        "args": "Print selected line",
        "when": "editorTextFocus && !editorReadonly"
    }
]
```

Descriviamo brevemente la sintassi: “key” rappresenta la combinazione di tasti che premeremo per attivare il Task, “command” sarà il comando che verrà lanciato tramite la combinazione dei tasti rapidi (in questo caso: esegui il Task), “args” il nome del Task da eseguire, “when” specifica in che condizione verrà eseguito il task (“editorTextFocus”:

il contenuto su cui fare il parsing è quello evidenziato dal cursore, "!editorReadonly": in base ai permessi dell'editor). Inserito il codice, salvare e chiudere il file.

18.3 Considerazioni

Per fare il parsing di alcune righe di codice invece che di un intero programma scritto in Python, è necessario fare una dovuta considerazione. I costrutti del linguaggio necessitano di una particolare struttura per essere riconosciuti ed analizzati, il parser sarà dunque impostato in modo tale che, se una riga o più di codice viene passata in input e non corrisponde interamente ad un costrutto del linguaggio Python, verrà segnalato un errore (esempio: una riga passata per metà, una definzione di un ciclo for senza il suo corpo, ...).

18.4 Test parsing di codice Python

Ora il nostro editor è configurato correttamente per poter eseguire il nostro parser avanzato tramite il comando rapido "ctrl+shift+x". Nel file "print selected line.py" inseriamo il contenuto del parser già mostrato in un sotto capitolo precedente. Ora creiamo un file Python, nominandolo ad esempio "lorem.py", al cui interno mettiamo del codice. Ora possiamo evidenziare delle righe all'interno del file "lorem.py", cliccare la combinazione di tasti, ottenendo l'ouput del parser avanzato che verrà letto all'utente tramite sintetizzatore vocale.[14]

19 Richiesta di parsing mediante un'istanza client-server

Il test di questo capitolo ha voluto rispondere alla seguente domanda: " se il parser non fosse sulla stessa macchina del file di cui si vuole fare il parsing?". Si è voluto provare a mettere in relazione i 2 programmi mediante un socket che inizializzi un'istanza client-server su 2 calcolatori diversi. In questo modo, colui che possiede il programma di cui vuole ottenere il parsing si connette come client, mentre il programma che fa il parsing viene mantenuto sempre attivo su una macchina che funge da server. Per fare ciò, d'ora in poi il programma "print selected line.py" non conterrà più il codice del parser, ma crea una connessione lato client tramite socket e, una volta instaurata la connessione, passa al server gli argomenti necessari al parsing, si mette in attesa, fin quando il server risponde con quello che siamo abituati essere il "lungo commento", che una volta ricevuto verrà letto dal client tramite il sintetizzatore vocale. Ora il codice del file file "print selected line.py" presente sul client è stato implementato nel seguente modo:

[14] https://www.dropbox.com/s/6bcm027vcg5mlyb/simulazionetasti_rapidivs_codenointernet.mov

```
# questo programma diventa un client che fara' una richiesta di connessione
# al server sul quale si deposita il parser.

# attualmente si utilizzera' l'indirizzo di localhost per gestire sia client
# che server, depositati in 2 cartelle diverse

import socket
from sys import argv
import pyttsx3

engine = pyttsx3.init() # inizializzo il mio lettore vocale


HOST = 'localhost' # Indirizzo ip del server
PORT = 8000 # Porta su cui il server ascolta

# Creo un oggetto socket
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT)) # connessione al server

    # una volta connesso, invio i parametri necessari al parsing
    # LEGENDA:
    #
    # argv[0] il nome del percorso completo di questo file
    # argv[1] il nome del file di cui si vuole fare il parsing
    # argv[2] la linea iniziale di codice da qui si vuole fare il parsing
    # argv[3] il contenuto della linea evidenziata di cui si vuole fare il parsing

    # 1
    s.send(argv[0].encode())
    # attendo ack 1
    ack_1 = s.recv(1024)
    # 2
    s.send(argv[1].encode())
    # attendo ack 2
    ack_2 = s.recv(1024)
    # 3
    s.send(argv[2].encode())
    # attendo ack 3
    ack_3 = s.recv(1024)
    # 4
    s.send(argv[3].encode())
    # attendo ack 4
    ack_4 = s.recv(1024)

    # ricevo la risposta del parsing eseguito dal server
```

```
response = s.recv(1024).decode()

# la riproduco con il lettore vocale
engine.say(response)
engine.runAndWait()
```

Questo approccio è stato testato sulla stessa macchina, mediante la connessione 'localhost', poi tra 2 macchine connesse alla stessa rete Wi-fi, una che simula il client, l'altra il server. [15]

20 Sfruttare la tecnica di Parsing in altri linguaggi

Gli esiti dei test precedenti sono risultati positivi e hanno compiuto un gran passo in avanti nelle nostre ricerche per quanto riguarda la lettura del codice tramite tecniche non-visisve. Questa tecnologia, purtoppo, è resa disponbile solo per il linguaggio di programmazione Python, non siamo in grado attualmente di eseguire il parsing di programmi scritti in linguaggi differenti da Python. Come visto in precedenza, il parsing del codice viene eseguito mediante la libreria 'ast', che è limitata solo al parsing di codice Python, è necessario allora ricercare delle nuove tecniche per poter ottenere il medesimo risultato anche in altri linguaggi di programmazione. Dopo alcune riflessioni, un approccio che avrebbe potuto risolvere il problema di non poter fare il parsing di codice in altri linguaggi è la costruzione di "grammatiche" mediante strumenti come 'lex' e 'yacc', o 'antlr', ma subito è emerso un problema nel momento in cui volessimo fare il parsing di "righe di codice" e non dell'intero programma. Per essere più chiari: se utilizzassi una grammatica costruita con le tecnologie citate precedentemente per, ad esempio, un programma scritto in Java, avrò bisogno di una regola detta "di partenza", in pratica dovrei definire la regola per stabilire che un programma in Java è composto da delle importazioni di librerie e da una definzione di una classe principale, successivamente una classe è definita da un'intestazione e da un corpo, poi un corpo di una classe è composto da un'insieme di espressioni, le espressioni a loro volta posso essere costrutti if, for, ..., fino ad arrivare agli elementi atomici della nostra grammatica. Se io però volessi fare il parsing di un singolo costrutto, ciò non sarebbe possibile, ad esempio, nel fare il parsing di un costrutto for non verrebbe riconosciuta la sua regola, poichè la grammatica si aspetta in primis una serie di 'import' e una classe, la regola del for quindi può essere raggiunta solo se il cammino viene percorso come prestabilito. Arrivati a questa consapevolezza, si è deciso di abbandonare l'uso delle grammatiche. Giunto alla conclusione delle ricerche, è emersa una possibile soluzione, ovvero tentare di ricostruire il codice dei metodi che la libreria 'ast' mette a disposizione, cercando di applicarli ad un'altro linguaggio di programmazione.

[15] https://www.dropbox.com/s/g32fp1x2o7ln2vh/parser_online.mov?dl=0

Per fare ciò, è stato necessario costruire numerose regole in modo da poter dividire il codice in blocchi sempre più elementari, in modo che possano venire confrontati con dei "template", che saranno univoci per ogni costrutto esistente nel linguaggio. Facciamo un esempio pratico: voglio fare il parsing del seguente blocco di codice scritto in Java:

```
while(i < 3) {
    j = 2;
    System.out.println("il valore e': " + j);
    i++;
}
```

Ricevuto in input questo codice, il programma costruito andrà, secondo opportune regole, a dividere il codice in blocchi. In questo caso, il risultato della divisione risulta:

```
['while (i < 3) {','j = 2','System.out.println("il valore e': " + j)','i++','}']
```

Una volta inserite in una lista le righe di codice, viene valutata ognuna di essa mediante degli opportuni template, infatti ogni elemento della lista può essere riconducibile ad un determinato costrutto del linguaggio Java. Questi sono alcuni dei template utilizzati:

```
def is_while(lista : List[str]) -> bool:
    # regole di verifica di un ciclo while
def is_assign(lista : List[str]) -> bool:
    # regole di verifica di un assegnamento
def is_increment(lista : List[str]) -> bool:
    # regole di verifica di un incremento
def is_call_method(lista : List[str]) -> bool:
    # regole di verifica di una chiamata di metodo

def check_template(l : List[str]):
    for lista in l:
        if is_while(lista):
            print("ciclo while...")
        elif is_assing(lista):
            print("assegnamento...")
        elif is_increment(lista):
            print("incremento...")
        elif is_call_method(lista):
            print("metodo...")
```

L'output risulta essere all'incirca lo stesso che si otteneva precedentemente con la libreria 'ast' in Python. Il parser di codice scritto in Java è stato costruito usando il linguaggio Python, dando modo di poterlo fondere col codice precedente per creare un

parser sia di codice Java che di Python. Seguendo questo meccanismo, diventa possibile costruire parser appositi per un qualsiasi linguaggio, per poi fonderlo insieme a quelli precedentemente creati, creando quello che possiamo definire un "parser avanzato universale". [16]

21 Conclusioni

L'obiettivo principale di questo progetto è stato fin dall'inizio di poter proporre un qualcosa di concreto ed utilizzabile, in modo che la domanda da cui questa tesi si è sviluppata possa dar frutto ad un prodotto usufruibile e non sia solamente una trattazione senza alcun risultato reale. Infatti, se riprendiamo la domanda di partenza presente nei capitoli iniziali:" È possibile facilitare lo sviluppo di codice ad utenti programmatori con difficoltà visiva? ", abbiamo sviluppato una risposta esaustiva che comprende un insieme di programmi, tecniche, buone attenzioni, in grado di superare l'ostacolo fisico della categoria di utenti a cui si è fatto riferimento per tutto il progetto. Giunti alla fine di questo progetto, la risposta definitiva che si può dare, secondo le nostre analisi, è la seguente: "Si, una tecnica che permette di rendere più accessibile la programmazione ad utenti non-vedenti è l'utilizzo degli screen reader, dispositivi che leggono al posto del utente il contenuto testuale presente sullo schermo, disponibli a tutti i sistemi operativi. I lettori di schermo possono essere utilizzati in combinazione con gli IDE come Visual Studio Code, editor che possiede una versione ottimizzata quando avverte l'utilizzo di uno screen reader, oltre a fornire la possibilità di produrre ed eseguire del codice scritto nei principali linguaggi di programmazione. Per migliorare la compresione della lettura di codice tramite lettore schermo è possibile sfruttare la tecnica del parsing di codice, ovvero usare a proprio vantaggio la struttura dei costrutti sintattici e semantici di un linguaggio di programmazione, in modo da poter creare un algoritmo di rilevazione di blocchi di codice riconducibili alle strutture predefinite e , una volta rilevate, creare una sequenza di azioni in base al tipo di costrutto rilevato, ad esempio, una traduzione del blocco di codice in linguaggio naturale. Questo è possibile sfruttando delle tecnologie già esistenti, come la libreria 'ast' con Python, oppure costruendo queste logiche per lo specifico linguaggio di cui vogliamo eseguire il parsing avanzato del codice".

[16] https://www.dropbox.com/s/3dt7uzwptmw9eoj/Testjavaparser.mov?dl=0