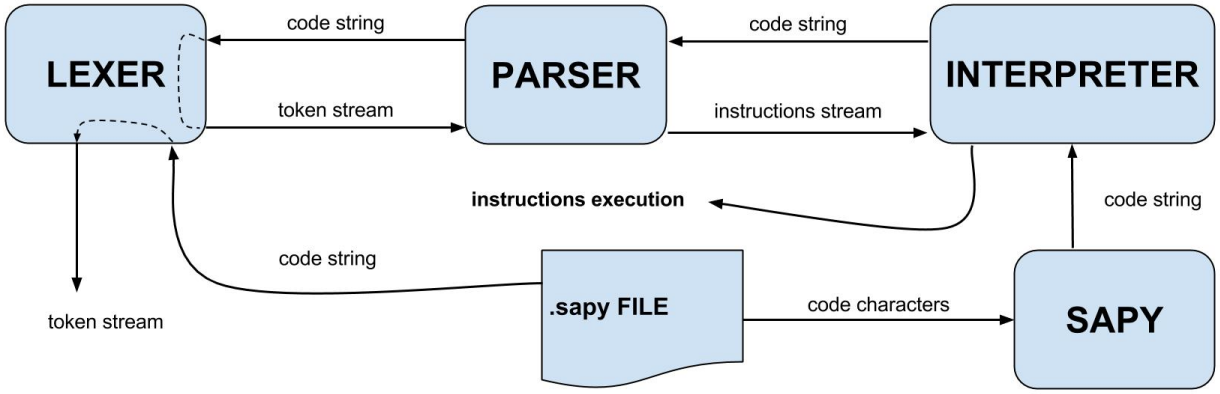
Documentazione del progetto "Sapienza BASIC v1.0" (o "Sapy")
Studente: Michele Reale, matricola 1315785

Architettura di Sapy

Sapy è un programma interprete dell'omonimo dialetto del linguaggio BASIC (la cui specifica grammaticale, sintattica e semantica è riportata nelle specifiche fornite durante l'insegnamento). L'input che esso riceve è un frammento di codice in linguaggio Sapy di cui simulare l'esecuzione. Il frammento di codice ricevuto subisce una serie di elaborazioni da parte del programma, il quale può essere concepito come una catena di montaggio, o una *pipeline*. Qui sotto è visualizzata l'architettura modulare del sistema.



Di seguito sono riportate le principali scelte progettuali e implementative compiute in ciascuno dei moduli del sistema.

NOTA: il **diagramma delle classi** è stato incluso nel pacchetto del progetto, sottoforma di immagine JPEG ad alta risoluzione. Il nome del file è DiagrammaClassiSapy.jpg, e si trova nella directory principale del file archivio.

Analizzatore lessicale (Lexer)

Il lexer è stato implementato come un riconoscitore *recursive-descent* di token. Le scelte progettuali più rilevanti sono state le seguenti.

- È stata definita la classe CodeCursor, interna alla classe Lexer, per astrarre lo stream di caratteri da analizzare. La classe permette di visualizzare il carattere corrente, di

consumarlo, e di verificare se lo stream di caratteri è finito. La classe Lexer può dunque essere implementata in modo più pulito, concentrandosi sull'implementazione dell'analisi lessicale.
- È stata definita la classe KeywordDictionary, interna alla classe Lexer, per istanziare token relativi alle parole chiave di Sapy, a partire da un dizionario ordinato delle parole chiave, analizzato con ricerca binaria.
- La classe Token è l'unica classe che mantiene le informazioni sui singoli token. La classe mantiene il contenuto del token (solo per i token che contengono effettivamente un valore) ed il tipo del token, definito dall'enumerazione TokenType.
- È stata implementata una tokenizzazione sofisticata, documentata in dettaglio nel javadoc.

Analizzatore sintattico (Parser)

Il parser è stato implementato come un riconoscitore *recursive-descent* di istruzioni ed espressioni. Le scelte progettuali più importanti sono state le seguenti.

- È stata definita la classe TokenCursor, interna alla classe Parser, che si comporta in modo analogo alla classe CodeCursor vista nel lexer.
- È stata definita la classe InstructionIdentifiers, interna alla classe Parser, che permette di riconoscere correttamente gli identificatori di riga, verificandone sia la correttezza sintattica sia l'univocità (mantenendo una struttura dati ordinata di ID).
- È stata delegata l'analisi sintattica delle espressioni alla classe SapyExpressionParser. La classe Parser, dunque, è più snella, e si concentra sull'analisi sintattica di istruzioni. La classe SapyExpressionParser utilizza la stessa istanza di TokenCursor del Parser, per iniziare l'analisi dalla posizione esatta da cui deve cominciare.
- È stata implementata la grammatica ricorsiva fornita nelle specifiche, ed è stata implementata l'analisi sintattica di tutte le istruzioni assegnate ai team da 1 studente; in aggiunta, è stato implementato anche il parsing della clausola ELSE dell'IF e della clausola STEP del FOR.
- I token FUNZIONE non sono passati al modulo interprete.

Interprete e classe Sapy

Le scelte progettuali ed implementative principali sono state le seguenti.

- La classe Interprete funge solo da *launcher* del programma eseguibile; quest'ultimo è il vero ambiente di esecuzione. Essa ottiene un'istanza di ProgrammaEseguibile dal Parser, passandogli la stringa di codice del file. Quindi imposta i riferimenti all'ambiente di esecuzione per le espressioni e le istruzioni. Infine, ne lancia l'esecuzione.

- La classe ProgrammaEseguibile è l'ambiente di esecuzione. Essa ha un'interfaccia per eseguire le operazioni di ambiente delle istruzioni, dette *primitive*. Ci sono primitive per l'I/O, per la gestione delle variabili, e per la gestione del flusso di esecuzione. Quest'ultima sarà descritta a breve.
- La classe ProgrammaEseguibile si interfaccia esclusivamente con l'Interprete e con le classi astratte Istruzione ed Espressione.
- Le classi Istruzione ed Espressione forniscono un *redirect* alle primitive alle sole loro sottoclassi. Esse si trovano, rispettivamente, nei package *istruzioni* ed *espressioni*. Per poter utilizzare l'ambiente di esecuzione, esse possono fare riferimento esclusivamente alle due superclassi astratte.
- La classe Literal, utilizzata per rappresentare i valori delle espressioni, è analoga alla classe Token, ma può contenere esclusivamente valori interi, booleani o stringa.
- I token FUNZIONE non sono analizzati in quanto il Parser li ha scartati.
- La classe Sapy, al termine dell'interpretazione del codice, fornisce il tempo di esecuzione totale dell'applicazione, dall'inizio dell'analisi lessicale al termine dell'interpretazione. Non è conteggiato il tempo di accesso e lettura del file del codice.

La gestione del flusso di esecuzione delle istruzioni del programma è stata introdotta per implementare le istruzioni di alterazione del flusso di esecuzione: **IF**, **FOR**, **GOTO**, **END**, WHILE, GOSUB, RETURN (le ultime tre non sono state implementate).

IF, FOR e WHILE generano un nuovo flusso di esecuzione da quello corrente, contenente le istruzioni al loro interno. GOSUB genera un nuovo flusso a partire da uno dei flussi attivi. RETURN torna al flusso più recente che sta eseguendo una GOSUB, scartando tutti gli altri flussi che incontra. GOTO torna all'unico flusso contenente l'istruzione con l'ID cercato. END distrugge l'ambiente di esecuzione, e quindi anche il flusso di esecuzione.

È stata definita la classe Stack, interna a ProgrammaEseguibile, che mantiene i flussi di esecuzione attivi al momento con la logica LIFO, tipica delle chiamate di funzione. Le istruzioni IF, FOR, WHILE e GOSUB, infatti, sono viste come chiamate di funzione, gestite mediante un nuovo flusso di esecuzione inserito sul top dello stack.

La classe Stack astrae la gestione dei singoli flussi, in particolare la gestione dei loro cursori di esecuzione (o *contatori*), ma la gestione dei flussi nel loro complesso è affidata alla classe ProgrammaEseguibile, mediante una catena di chiamate ricorsive del metodo `esegui(StackRow)`.

È stata definita la classe StackRow, interna a Stack, che permette di gestire un unico flusso di esecuzione. Ogni flusso ha un proprio cursore che indica l'istruzione attualmente in esecuzione, e deve essere eliminato dallo stack quando la sua esecuzione è terminata.

La catena di invocazioni ricorsive al metodo `esegui(StackRow)` è così impiegata.

- Inizialmente, si invoca il metodo sull'unico elemento dello stack: la lista di istruzioni fornita

dal Parser, flusso principale di esecuzione.

- Non appena si esegue un'istruzione di alterazione del flusso (la quale utilizza una opportuna primitiva dell'ambiente di esecuzione), si procede per casi.
    - Se l'istruzione è IF o FOR, si lancia la primitiva `executeInstructions`, che inserisce la lista di istruzioni nello stack e lancia il metodo `esegui(StackRow)`, che eseguirà il nuovo flusso al top dello stack.
    - Se l'istruzione è GOTO, si lancia la primitiva `jumpTo`, che scorre lo stack, dal top in poi, verificando che esista un flusso in esso contenente l'istruzione con l'ID cercato. Se lo trova, si ferma, e l'esecuzione riprende dall'istruzione successiva. Altrimenti, si elimina il top dello stack e si itera il procedimento sulla catena di chiamate a `esegui(StackRow)`, sfruttando il meccanismo delle eccezioni: si lancia l'eccezione `MissingJumpIDException`, che il metodo `esegui(StackRow)` deve gestire controllando che l'ID specificato sia presente nel suo flusso: se è presente, si imposta il contatore su di esso e si lancia il metodo `esegui` sul flusso; altrimenti, si lascia scorrere l'eccezione nelle chiamate ricorsive di `esegui(StackRow)`.
    - Se l'istruzione è END, si sfrutta il meccanismo delle eccezioni per terminare la catena di chiamate di `esegui(StackRow)`, lanciando l'eccezione `HaltException`; il metodo `esegui(StackRow)` la lascia passare. In questo modo, le chiamate ricorsive di `esegui(StackRow)` sono tutte terminate, e lo stack è vuotato.

- Ogni volta che si deve decidere quale istruzione eseguire, si procede sempre per casi. La decisione è mantenuta in un flag booleano membro della classe ProgrammaEseguibile, chiamato `setStandardPC`.
    - Se l'istruzione non altera il flusso di esecuzione, si incrementa il contatore prima di estrarre l'istruzione successiva.
    - Se l'istruzione è IF o FOR, non si deve incrementare il contatore prima dell'esecuzione, perché esso è già posto sulla prima istruzione del nuovo flusso.