POLITECNICO DI BARI

Dipartimento di Ingegneria Elettrica e dell'Informazione - DEI

Corso di Laurea Magistrale in
INGEGNERIA INFORMATICA

Corso di
LINGUAGGI FORMALI E COMPILATORI

Kotlin2Swift Transpiler

Studentessa
Angela Cassanelli

Anno Accademico 2024 - 2025

Indice

1. Introduzione

Il progetto propone lo sviluppo di un transpiler per convertire codice *Kotlin* in *Swift*, utilizzando una grammatica semplificata di Kotlin. Le principali funzionalità includono la definizione della grammatica, la generazione automatica di *lexer* e *parser*, nonché l'analisi lessicale e sintattica, tutte realizzate tramite *ANTLR* (ANother Tool for Language Recognition); l'analisi semantica e la traduzione del *parse tree* sono state sviluppate manualmente. Per semplificare il processo di compilazione ed esecuzione del transpiler, è stato creato un *Makefile* con i seguenti target: *generate_antlr* per generare il lexer e il parser; *build* per installare le dipendenze necessarie, con riferimento al file *requirements.txt*; *all* per eseguire in sequenza i target *generate_antlr* e *build*; *run* per avviare il transpiler; infine *clean* per rimuovere i file di output e i file temporanei generati.

1.1 Esecuzione del transpiler

Il codice sorgente è disponibile su GitHub e include la documentazione tecnica. Per eseguire il transpiler, è necessario soddisfare i seguenti prerequisiti (il progetto è stato sviluppato su macOS, alcune istruzioni potrebbero variare su altri sistemi operativi):

- Installare Java, con una versione minima di 17.0.2 (consigliata Java 17.0.12). Se si utilizza una versione diversa, modificare la variabile `JAVA_HOME` nel file `.zshrc`.
- Installare Python 3.x (consigliata 3.13.0). A seconda della versione di Python installata, potrebbero verificarsi errori nella verifica dei certificati: per risolvere il problema, è necessario installare i certificati manualmente eseguendo il file `Install Certificates.command` in `/Applications/Python 3.x`.
- Attivare un ambiente virtuale (`venv`) per isolare le dipendenze del progetto tramite i comandi `python3 -m venv venv` e `source venv/bin/activate`.
- Installare ANTLR tramite il comando: `pip install antlr4-tools`.
- Configurare correttamente le variabili d'ambiente. Su sistemi operativi Unix, è necessario eseguire il comando: `source Kotlin2SwiftTranspiler/.zshrc`. Su sistemi operativi non Unix, potrebbero essere necessari comandi differenti per ottenere lo stesso comportamento in relazione alle variabili d'ambiente.

Dopo aver soddisfatto i prerequisiti, è necessario posizionarsi nella directory principale del progetto ed eseguire i seguenti comandi da terminale:

```
make clean
make all
make run KOTLIN_FILE=path_to_kotlin_file
```

dove *path_to_kotlin_file* rappresenta il percorso del file Kotlin da tradurre (ad esempio: `tests/test_case_1`). È possibile passare in input i test case presenti nella cartella `tests`.

Il transpiler mostrerà sulla console i messaggi relativi alla lettura del codice Kotlin, alla generazione del *parse tree* e alla visita dei nodi sintattici, insieme ai controlli semantici eseguiti. Eventuali errori lessicali, sintattici o semantici verranno segnalati con dettagli sulla tipologia e sulla posizione. Se invece non ci sono errori, il codice Swift generato verrà mostrato in console e scritto nel file `output/output.swift`.

2. Architettura del Transpiler

Il progetto è stato strutturato seguendo l'architettura generale di un transpiler. L'architettura risultante, riportata di seguito, è modulare e permette una chiara separazione delle responsabilità, rendendo il sistema facilmente estendibile e manutenibile.

```
root/
├── antlr/
│   └── Kotlin.g4
├── generated/antlr/
│   ├── KotlinLexer.py
│   ├── KotlinListener.py
│   └── KotlinParser.py
├── output/
│   └── output.swift
├── src/
│   ├── KotlinToSwiftVisitor.py
│   ├── LexicalErrorListener.py
│   ├── SemanticErrorListener.py
│   ├── SymbolTable.py
│   ├── SyntaxErrorListener.py
│   ├── transpiler.py
│   ├── Types.py
│   └── Utils.py
└── tests/
    ├── test_case_1
    ├── test_case_2
    ├── test_case_3
    ├── test_case_4
    └── test_case_5
```

2.1 Componenti e Funzionalità

La cartella `antlr/` contiene la definizione formale della grammatica nel file `Kotlin.g4`. Questa grammatica stabilisce i token lessicali, che sono le unità atomiche del linguaggio, come parole chiave (ad esempio `class`, `fun`), operatori (come `+` e `-`) e simboli (`(`, `{`), e definisce le regole sintattiche, che descrivono le strutture gerarchiche e i costrutti del linguaggio Kotlin, come classi, funzioni ed espressioni. Il file `Kotlin.g4` viene utilizzato da ANTLR per generare automaticamente componenti fondamentali richiesti nelle diverse fasi del processo di traduzione.

La cartella `generated/antlr/` contiene i file generati automaticamente da ANTLR a partire dalla grammatica `Kotlin.g4`. Questi sono: **KotlinLexer.py**, responsabile della tokenizzazione e dell'analisi lessicale; **KotlinParser.py**, che esegue l'analisi sintattica costruendo il *parse tree* dai token riconosciuti; **KotlinListener.py**, che gestisce l'attraversamento e la manipolazione del *parse tree*.

La cartella `src/` contiene i moduli principali del transpiler, che implementano le diverse fasi del processo di traduzione. I file più rilevanti sono:

- **transpiler.py**: Gestisce l'intero processo di traduzione da Kotlin a Swift attraverso diverse funzioni. Il flusso inizia con il caricamento del codice sorgente Kotlin da un file di input, che viene analizzato dal lexer generato da ANTLR. Successivamente, il parser costruisce un *parse tree*, che rappresenta la struttura del codice sorgente. Il *parse tree* viene quindi attraversato dal visitor `KotlinToSwiftVisitor.py`, che traduce i nodi del *parse tree* in codice Swift, applicando le opportune trasformazioni. Il sistema implementa anche meccanismi per il rilevamento e la gestione di errori

lessicali, sintattici e semantici attraverso l'uso di appositi listener. Se non vengono rilevati errori e la traduzione ha successo, il codice Swift risultante viene scritto nel file di output `output/output.swift`. In caso contrario, vengono sollevati errori con messaggi dettagliati, che aiutano l'utente a identificare e correggere eventuali problemi nel codice sorgente.

- **KotlinToSwiftVisitor.py**: Implementa il pattern *Visitor* per attraversare il *parse tree* generato da ANTLR e tradurre ogni nodo nella sua controparte Swift. Contestualmente alla visita, esegue l'analisi semantica per verificare la correttezza logica del codice. Tra gli elementi gestiti rientrano classi, funzioni, variabili, costrutti condizionali e iterativi, istruzioni di ritorno, e altri componenti del linguaggio Kotlin.

- **LexicalErrorListener.py, SyntaxErrorListener.py, SemanticErrorListener.py**: Ereditano dalla classe `ErrorListener` di ANTLR e ridefiniscono i metodi per gestire gli errori nelle rispettive fasi di analisi. Ogni classe raccoglie i messaggi di errore in una lista per una gestione centralizzata. `LexicalErrorListener` e `SyntaxErrorListener` personalizzano i messaggi per gli errori lessicali e sintattici, che vengono intercettati automaticamente da ANTLR, mentre `SemanticErrorListener` lancia e gestisce gli errori rilevati durante l'analisi semantica implementata nel processo di traduzione.

- **Symbol.py**: Rappresenta una variabile simbolica all'interno della tabella dei simboli, e ne registra nome, tipo, mutabilità e valore. Viene utilizzata per tracciare le variabili durante l'analisi semantica, garantendo che siano utilizzate correttamente.

- **SymbolTable.py**: Implementa una tabella dei simboli come una pila di scope, ciascuno rappresentato da un dizionario che memorizza informazioni su variabili, funzioni e classi. Gestisce l'aggiunta, la rimozione e la ricerca di questi elementi negli scope attivi, supportando l'analisi semantica e permettendo la rilevazione di duplicati e la validazione dei dati.

- **Types.py**: Definisce i tipi di dati supportati dal transpiler, per il linguaggio Kotlin (`Int`, `String` e `Boolean`) e per il linguaggio Swift (`Int`, `String` e `Bool`).

- **Utils.py**: Contiene due principali strutture di utility. `KOTLIN_2_SWIFT_TYPES` è un dizionario che associa i tipi di dati di Kotlin ai corrispondenti tipi in Swift, mentre `RESERVED_KEYWORDS` è un insieme di parole chiave e simboli riservati in entrambi i linguaggi, che non possono essere utilizzati come nomi di variabili, funzioni o classi.

La cartella `output/` contiene il file `output.swift`, che rappresenta il risultato finale della traduzione, generato dal transpiler solo se il codice di input è privo di errori lessicali, sintattici o semantici. Il codice Swift risultante è sintatticamente corretto, rispecchia la struttura e i costrutti del linguaggio Swift corrispondenti a quelli del linguaggio Kotlin di partenza, e può essere direttamente eseguito.

La cartella `tests/` contiene i file di codice sorgente in Kotlin utilizzati per testare il transpiler. I casi di test comprendono sia esempi corretti che scenari con errori lessicali, sintattici e semantici, e permettono di verificare non solo la corretta traduzione del codice Kotlin in Swift, ma anche l'efficacia dell'analisi semantica implementata. Questo approccio assicura che il transpiler funzioni correttamente sotto diversi scenari, garantendo sia la qualità del codice generato che l'affidabilità del processo di analisi.

3. Grammatica Kotlin

Il transpiler sviluppato implementa una versione ristretta della grammatica di Kotlin, focalizzandosi su un sottoinsieme di costrutti fondamentali del linguaggio. Tra le funzionalità incluse, vi sono le istruzioni per l'input e l'output, come la lettura di dati tramite `readLine()` e la stampa a console mediante `println()`; le istruzioni iterative sono limitate al costrutto `for`, che consente di iterare su intervalli numerici; le istruzioni condizionali supportano le strutture `if` e `if-else`, utili per la gestione del flusso decisionale.

La programmazione orientata agli oggetti è rappresentata in modo semplificato, permettendo la dichiarazione di classi che possono contenere proprietà e metodi, ma senza possibilità di istanziazione; questa limitazione è legata al supporto ristretto dei tipi di dato, che include soltanto i tipi primitivi `Int`, `Boolean` e `String`; il sistema consente inoltre di dichiarare e assegnare variabili mutabili e immutabili; un'altra caratteristica chiave del transpiler è la gestione delle funzioni, che possono essere definite con parametri e valori di ritorno e successivamente richiamate nel codice.

3.1 Implementazione della Grammatica Kotlin

La grammatica è stata progettata per semplificare la traduzione da Kotlin a Swift, mantenendo le caratteristiche principali di Kotlin e garantendo un mapping chiaro verso Swift: ogni costrutto grammaticale è stato sviluppato per ridurre le ambiguità, facilitare l'analisi semantica e ottimizzare la traduzione.

La grammatica definisce le regole lessicali e sintattiche fondamentali di Kotlin. Le regole lessicali definiscono le unità lessicali, i token, come le parole chiave, i tipi di dato, gli operatori aritmetici e logici, i simboli di punteggiatura e i valori letterali; sono espresse tramite espressioni regolari e, di solito, sono scritte in maiuscolo. Le regole sintattiche, invece, definiscono la struttura del linguaggio, specificando come i token devono essere combinati per formare frasi e istruzioni valide, e sono generalmente scritte in minuscolo. Segue una descrizione dei costrutti implementati.

La regola `program` è l'entry point del programma e rappresenta un file di codice Kotlin composto da zero o più `topLevelStatement`, seguiti dal termine del programma (`EOF`).

La regola `topLevelStatement` definisce le dichiarazioni che possono apparire direttamente all'interno del programma: queste sono `classDeclaration`, che rappresenta la dichiarazione di una classe, e `commentStatement`, che rappresenta un commento.

La regola `statement` definisce i costrutti eseguibili all'interno di classi, funzioni (`functionDeclaration`) o blocchi di codice (`block`). Tra questi, troviamo `readStatement` per leggere l'input dell'utente, `printStatement` per stampare un output sulla console, `ifElseStatement` come istruzione di diramazione, `forStatement` come istruzione di iterazione, `assignmentStatement` per assegnare un valore a una variabile, `varDeclaration` per dichiarare una variabile, `returnStatement` per ritornare un valore da una funzione e, infine, `commentStatement`, che rappresenta un commento.

Una sequenza di zero o più `statement`, racchiusa tra parentesi graffe, costituisce un blocco di codice, definito dalla regola `block`.

La regola `classDeclaration` consente la dichiarazione di classi, attraverso il token `CLASS`, che possono includere un costruttore opzionale definito utilizzando `parameterList` o `propertyList`, e un corpo definito come blocco di codice tramite la regola `classBody`,

che può includere dichiarazioni e assegnazioni di variabili, dichiarazioni di funzioni e commenti. A causa della limitazione sui tipi di dato supportati (interi, booleani e stringhe), non è possibile creare istanze delle classi.

La regola `functionDeclaration` consente la dichiarazione di funzioni, attraverso il token `FUN`, che possono includere parametri opzionali, un tipo di ritorno opzionale e un corpo definito come blocco di codice tramite la regola `block`. I parametri sono specificati tramite la regola `parameterList`, che consente di definire più parametri separati da virgole e, opzionalmente, assegnare loro valori di default.

La regola `varDeclaration` gestisce le dichiarazioni di variabili, sia mutabili con il token `VAR` che immutabili con il token `VAL`. Una variabile può essere dichiarata specificando il tipo e, opzionalmente, un valore iniziale, oppure può essere assegnato direttamente un valore senza dichiarare esplicitamente il tipo.

La regola `assignmentStatement` consente di assegnare un valore a una variabile e prevede tre varianti. La prima variante, `IDENTIFIER EQ expression`, assegna alla variabile (identificata da `IDENTIFIER`) il valore di un'espressione. La seconda variante, `IDENTIFIER EQ readStatement`, assegna alla variabile il valore ritornato da un'istruzione di input di tipo `readStatement`. L'ultima variante include `callExpression`, per risolvere l'ambiguità tra l'istruzione di assegnazione e la chiamata a funzione. Questo approccio evita conflitti nei casi in cui un `IDENTIFIER` seguito dal simbolo di assegnazione `=` potrebbe essere interpretato sia come un'operazione di assegnazione che come una chiamata di funzione.

L'istruzione di diramazione `ifElseStatement` valuta un'espressione racchiusa tra parentesi tonde: se l'espressione è vera, viene eseguito il corpo dell'istruzione `if` (token `IF`), definito dalla regola `ifBody`; altrimenti viene eseguito il corpo dell'istruzione `else` (token `ELSE`), definito dalla regola `elseBody`. Il blocco `else` è opzionale: l'intero costrutto `ELSE` seguito dal suo corpo può non essere presente. Entrambi i corpi (`ifBody` e `elseBody`) possono essere costituiti da un blocco di istruzioni racchiuso tra parentesi graffe o da una singola istruzione.

La regola `forStatement` descrive un ciclo `for` attraverso il token `FOR`. La condizione di iterazione è definita dalla regola `membershipExpression`, che verifica l'appartenenza di un'espressione a un intervallo numerico definito come `rangeExpression` su cui iterare. A causa delle limitazioni sui tipi, il ciclo `for` può iterare solo su intervalli di numeri interi. Il corpo del ciclo può essere costituito da un blocco di istruzioni racchiuso tra parentesi graffe oppure da una singola istruzione.

Le espressioni sono definite in modo ricorsivo, consentendo la costruzione di espressioni complesse a partire da quelle più semplici. Ogni tipo di espressione può includere altri tipi, creando così espressioni annidate. L'operatore logico OR (`OR`) è il primo ad essere definito nella grammatica e, per questo motivo, ha la precedenza più bassa tra tutti gli operatori. Può contenere ricorsivamente tutte le altre tipologie di espressioni, che verranno valutate prima di essere eventualmente combinate con l'operatore OR.

La regola principale che definisce un'espressione è `expression`, che include un'espressione logica OR. Quest'ultima è gestita dalla regola `logicalOrExpression`, che si occupa dell'operatore `OR` e può contenere ricorsivamente espressioni logiche AND, definite dalla regola `logicalAndExpression`. Questa, a sua volta, gestisce l'operatore `AND` e può includere espressioni di uguaglianza, descritte dalla regola `equalityExpression`.

Le espressioni di uguaglianza utilizzano gli operatori di uguaglianza (`EQEQ`) e diversità (`NEQ`) e possono includere espressioni relazionali, regolate dalla `relationalExpression`.

Quest'ultima gestisce gli operatori relazionali (`GT`, `GTE`, `LT`, `LTE`) per i confronti di grandezza e può includere espressioni additive, definite dalla `additiveExpression`.

Le espressioni additive trattano le operazioni aritmetiche di somma (`PLUS`) e sottrazione (`MINUS`) e possono includere espressioni moltiplicative, gestite dalla regola `multiplicativeExpression`. Questa regola, a sua volta, si occupa delle operazioni di moltiplicazione (`MULT`), divisione (`DIV`) e modulo (`MOD`) e può includere espressioni unarie, definite dalla `unaryExpression`.

Le espressioni unarie comprendono la negazione aritmetica (`MINUS`), la negazione logica (`NOT`) e le espressioni di appartenenza, descritte dalla regola `membershipExpression`. Quest'ultima gestisce l'operatore di appartenenza (`IN`), che verifica se un valore appartiene a un intervallo definito da una `rangeExpression`.

Alla base della struttura delle espressioni si trova la regola `primaryExpression`, che può rappresentare un identificatore, una chiamata a funzione, un valore letterale o un'altra espressione racchiusa tra parentesi tonde.

Le chiamate a funzione sono definite dalla regola `callExpression`, che descrive la sintassi composta dal nome della funzione (`IDENTIFIER`), seguito da una lista di argomenti separati da virgole e racchiusi tra parentesi tonde, specificati tramite la regola `argumentList`.

Infine, i valori letterali, gestiti dalla regola `literal`, sono valori fissi e immutabili utilizzati nel programma. Essi possono essere di diversi tipi, tra cui `TYPE_INT` per rappresentare numeri interi, `TYPE_BOOLEAN` per valori booleani (`true` o `false`), e `TYPE_STRING` per stringhe di caratteri racchiusi tra virgolette.

La regola `type` definisce i tipi di dato supportati nel linguaggio: `TYPE_INT` per i numeri interi, `TYPE_STRING` per le stringhe, e `TYPE_BOOLEAN` per i valori booleani.

L'istruzione `IDENTIFIER` definisce un identificatore composto da un carattere alfabetico (maiuscolo o minuscolo) o un underscore, seguito da una sequenza di caratteri alfanumerici o underscore (`[a-zA-Z_] [a-zA-Z_0-9]*`). Un identificatore è un nome che rappresenta variabili, funzioni o altre entità nel programma.

Per l'interazione con l'utente, sono stati implementati due costrutti specifici. Il primo è rappresentato dalla regola `readStatement`, che gestisce la lettura di input tramite il metodo Kotlin `readLine`. Il secondo costrutto è definito dalla regola `printStatement`, che consente di stampare un'espressione sulla console utilizzando il metodo Kotlin `println`.

La grammatica supporta due tipi di commenti: su singola riga e a blocchi. I commenti su singola riga sono definiti dalla regola `LINE_COMMENT` e possono essere utilizzati per annotazioni che si estendono su una singola riga, ma devono essere scritti su una riga separata dal codice, non accanto ad esso. I commenti a blocchi, definiti dalla regola `BLOCK_COMMENT`, consentono di scrivere annotazioni che si estendono su più righe. Entrambi i costrutti sono integrati nella grammatica tramite la regola `commentStatement`.

La regola `WS` definisce i caratteri di spaziatura, che includono spazi bianchi, tabulazioni e ritorni a capo: questi caratteri sono ignorati durante il parsing del codice, in quanto non influiscono sulla sintassi del programma.

4. Implementazione del Visitor

ANTLR utilizza il pattern *Visitor* per l'attraversamento dell'albero di parsing. Questo modello consente di separare la logica di visita dalla struttura dei dati, rappresentata dall'albero di parsing generato da ANTLR. Il principale vantaggio del Visitor è la possibilità di controllare esplicitamente l'ordine di visita dei nodi, offrendo maggiore flessibilità nel processo di traversamento. Inoltre, il pattern è particolarmente vantaggioso per la sua modularità e scalabilità: a differenza di altre soluzioni, il Visitor permette di aggiungere nuovi metodi senza modificare quelli già esistenti, semplificando l'integrazione di modifiche alla grammatica o l'introduzione di nuove funzionalità.

Nel progetto, è stato scelto il pattern Visitor per gestire in modo modulare e scalabile la traduzione del codice. ANTLR genera automaticamente una classe base, `ParseTreeVisitor`, per ogni grammatica definita, che include metodi predefiniti per ogni tipo di nodo dell'albero di parsing. Estendendo questa classe, è possibile implementare operazioni personalizzate per ciascun tipo di nodo, adattando la logica di traduzione alle specifiche esigenze del progetto.

4.1 Dettaglio dei principali metodi del Visitor

La classe `KotlinToSwiftVisitor` estende la classe `ParseTreeVisitor` e implementa metodi specifici per tradurre ogni nodo dell'albero sintattico in codice Swift, seguendo la grammatica definita. Ogni costrutto rilevante della grammatica ha un metodo dedicato, progettato per estrarre dal parse tree le informazioni necessarie, come identificatori, tipi e valori, e generare la corrispondente rappresentazione in Swift. Ogni metodo della classe `KotlinToSwiftVisitor` prende in input un parametro `ctx`, che rappresenta un oggetto `ParserRuleContext` o una sua sottoclasse. ANTLR, infatti, genera una classe *Context* specifica per ciascuna regola grammaticale, estendendo la classe base `ParserRuleContext`. Questi oggetti *Context* rappresentano i nodi dell'albero sintattico associati alle regole della grammatica e offrono metodi e proprietà per accedere alle informazioni strutturali del nodo. In particolare, è possibile estrarre specifici token tramite metodi come `getText()`, accedere ai figli del nodo corrente, che corrispondono ai sottocostrutti grammaticali, e visitarli ricorsivamente con `visit(ctx.childNode)`, per costruire l'output combinando i risultati delle sottoespressioni. Di seguito sono descritti i metodi principali e il loro funzionamento in relazione alla sola traduzione sintattica.

Il metodo `visit_program` elabora le istruzioni a livello superiore di un programma. Prende in input un oggetto `ProgramContext`, generato da ANTLR per la regola `program` definita nella grammatica, che include il metodo `topLevelStatement` per attraversare gli statement a livello superiore. Il metodo verifica se il nodo corrente dell'albero, rappresentato da `ctx`, contiene uno o più top-level statement: se non sono presenti, il metodo solleva un'eccezione; altrimenti, per ogni top-level statement, viene invocato il metodo `visit_top_level_statement()` che visita ciascun `topLevelStatement` e ritorna il codice Swift equivalente. Tutti i codici Swift generati vengono raccolti in una lista e, dopo aver visitato tutti gli statement, il metodo unisce le stringhe nella lista con un separatore di newline, rimuovendo eventuali valori `None`. Infine, ritorna la stringa risultante.

Il metodo `visit_top_level_statement` gestisce le istruzioni a livello superiore di un

programma, indirizzandole ai metodi specifici per la traduzione. Prende in input un oggetto `TopLevelStatementContext`, generato da ANTLR per la regola `topLevelStatement`, che offre i metodi `commentStatement`, per elaborare i commenti, e `classDeclaration`, per accedere alle dichiarazioni di classe. Il metodo analizza lo statement verificandone la natura: se si tratta di un commento, affida l'elaborazione al metodo `visit_comment_statement`; se invece è una dichiarazione di classe, delega la traduzione al metodo `visit_class_declaration`. In caso di statement non riconosciuto, il metodo mostra un messaggio di errore.

Il metodo `visit_statement` traduce vari tipi di istruzioni, delegando ciascuna traduzione al metodo appropriato in base al tipo di istruzione. Riceve in input un oggetto `StatementContext`, che espone i metodi per identificare il tipo di istruzione e accedere al suo contenuto, come `readStatement()`, `printStatement()`, `ifElseStatement()`, `forStatement()`, `assignmentStatement()`, `varDeclaration()`, `returnStatement()` e `commentStatement()`. Il metodo identifica il tipo di istruzione e invoca la funzione corrispondente per generare la sintassi Swift equivalente, restituendo una stringa vuota se l'istruzione non è riconosciuta.

Il metodo `visit_block` gestisce un blocco di istruzioni. Prende in input un oggetto `BlockContext`, che rappresenta il blocco di codice, e fornisce il metodo `statement()` per accedere alle singole istruzioni al suo interno. Visita ciascuna istruzione, invocando il metodo di traduzione appropriato per generare la sintassi Swift equivalente. Le istruzioni vuote o non riconosciute vengono escluse dalla traduzione. Il risultato finale è una stringa che rappresenta il blocco di codice Swift, con ogni istruzione separata da una nuova riga.

Il metodo `visit_comment_statement` gestisce i commenti. Prende in input un oggetto `CommentStatementContext`, generato da ANTLR per la regola `commentStatement`, che include i metodi `LINE_COMMENT` e `BLOCK_COMMENT` per riconoscere rispettivamente i commenti su singola riga e i commenti a blocco. Il metodo invoca `visit_line_comment` per convertire i commenti su singola riga nella forma `# comment` e `visit_block_comment` per convertire i commenti a blocco nella forma `/* comment */`.

Il metodo `visit_class_declaration` traduce una dichiarazione di classe. Prende in input un oggetto `ClassDeclarationContext`, generato da ANTLR per la regola `classDeclaration` definita nella grammatica, che include metodi come `IDENTIFIER` per riconoscere l'identificatore della classe, `classBody` per accedere al corpo della classe, `propertyList` per elaborare le proprietà e `parameterList` per gestire i parametri. In Kotlin, il costruttore principale è definito nella dichiarazione della classe, senza la necessità di un blocco `init`, e i parametri inizializzano automaticamente le proprietà. In Swift, invece, il costruttore è separato dalla dichiarazione della classe e le proprietà devono essere inizializzate esplicitamente nel blocco `init`. Il processo inizia con l'estrazione del nome della classe, utilizzando il metodo `visit_identifier`. Se la classe include delle proprietà, queste vengono visitate tramite `visit_property_list`, e un costruttore Swift viene creato per dichiarare e inizializzare, se necessario, tali proprietà. Nel caso in cui non ci siano proprietà, ma siano presenti parametri di input, questi vengono visitati tramite `visit_parameter_list`, e viene generato un costruttore senza corpo. Se la classe non ha né proprietà né parametri, viene creata solo la dichiarazione della classe. In seguito, se è presente un corpo della classe, viene visitato tramite `visit_class_body` e aggiunto alla dichiarazione della classe. Al termine del processo, il metodo ritorna la dichiarazione della classe Swift nella forma *`class_declaration`* `{` *`properties_declarations`* `\n` *`constructor`* `\n` *`body`* `}`, che include proprietà, costruttore e corpo, se presenti.

Il metodo `visit_identifier` gestisce gli identificatori, assicurandosi che non siano

parole chiave riservate. Prende in input un contesto `KotlinParser`, che rappresenta il nodo dell'albero sintattico associato all'identificatore. Poiché gli identificatori sono definiti tramite un'espressione regolare nella grammatica, non viene creato un contesto specifico come per altre regole, ma il contesto generico del parser è sufficiente per estrarre il nome dell'identificatore utilizzando il metodo `getText()`. Se l'identificatore è una parola chiave riservata, viene sollevato un errore; altrimenti, il metodo ritorna il nome dell'identificatore.

Il metodo `visit_property_list` traduce una lista di proprietà rappresentata da un oggetto `PropertyListContext`, generato da ANTLR per la regola `propertyList`. Quest'ultimo fornisce il metodo `property_` per accedere alle singole proprietà della lista. Il metodo itera sugli elementi della lista, invocando `visit_property` per tradurre ciascuna proprietà. Alla fine, ritorna una lista di stringhe, ognuna delle quali rappresenta una proprietà tradotta nel formato Swift corrispondente. Il metodo `visit_property` traduce una singola proprietà rappresentata da un oggetto `PropertyContext`, generato da ANTLR per la regola `property`. Nella grammatica, le proprietà sono definite come variabili, pertanto l'oggetto `PropertyContext` offre il metodo `varDeclaration` per accedere alla dichiarazione della variabile associata. Il metodo `visit_property` invoca quindi `visit_var_declaration` per tradurre la dichiarazione della variabile in Swift.

Il metodo `visit_parameter_list` traduce una lista di parametri rappresentata da un oggetto `ParameterListContext`, generato da ANTLR per la regola `parameterList`. Quest'ultimo include il metodo `parameter` per accedere ai singoli parametri nella lista. Il metodo itera sugli elementi della lista, chiamando il metodo `visit_parameter` per ciascun parametro. Al termine, ritorna una stringa che rappresenta la lista dei parametri Swift, separati da virgole. Il metodo `visit_parameter` traduce un singolo parametro rappresentato da un oggetto `ParameterContext`, generato da ANTLR per la regola `parameter`. Quest'ultimo include i metodi `IDENTIFIER` per riconoscere l'identificatore del parametro, `type_` per accedere al tipo e `expression` per accedere a un eventuale valore associato. Il metodo estrae il nome e il tipo del parametro, utilizzando rispettivamente il metodo `visit_identifier` e `visit_type`. Se il parametro ha un valore di default, questo viene visitato tramite `visit_expression`. Infine, il metodo ritorna una stringa che rappresenta il parametro in Swift in forma *param_name* `:` *param_type* `=` *param_value*, includendo il nome, il tipo e il valore di default se presenti.

Il metodo `visit_class_body` gestisce le dichiarazioni di classe. Prende in input un oggetto `ClassBodyContext`, generato da ANTLR per la regola `classBody` definita nella grammatica, che rappresenta il corpo della classe e consente di accedere alle singole istruzioni contenute al suo interno. Il processo inizia iterando sulle istruzioni presenti nel corpo della classe, identificate come figli dell'oggetto `ClassBodyContext`. Se l'istruzione corrisponde a una dichiarazione di variabile (`VarDeclarationContext`), viene elaborata tramite `visit_var_declaration`; se corrisponde a una dichiarazione di funzione (`FunctionDeclarationContext`), viene elaborata utilizzando `visit_function_declaration`; le assegnazioni (`AssignmentStatementContext`) sono gestite tramite `visit_assignment_statement`; i commenti (`CommentStatementContext`) vengono convertiti utilizzando `visit_comment_statement`. In caso di istruzioni non riconosciute o non valide, viene registrato un errore o sollevata un'eccezione rispettivamente. Infine, tutte le istruzioni tradotte vengono unite in una singola stringa, con ogni istruzione separata da una nuova riga. Il metodo ritorna la stringa risultante.

Il metodo `visit_var_declaration` gestisce le dichiarazioni di variabili mutabili e immutabili. Prende in input un oggetto `VarDeclarationContext`, generato da ANTLR per

la regola `varDeclaration`, che fornisce metodi per accedere ai componenti della dichiarazione: il metodo `IDENTIFIER()` ritorna il nodo contenente il nome della variabile; il metodo `type_()` ritorna il contesto del tipo dichiarato per la variabile; i metodi `VAR()` e `VAL()` identificano se la variabile è mutabile (`var`) o immutabile (`val`); il metodo `expression()` consente di accedere all'espressione di inizializzazione; il metodo `readStatement()` fornisce accesso a una dichiarazione di lettura da input. Il nome della variabile viene estratto utilizzando `visit_identifier`, il tipo della variabile, se presente, viene gestito attraverso `visit_type`, mentre il valore, se presente, viene determinato attraverso un controllo condizionale: se l'inizializzazione è basata su un'espressione (`ctx.expression()`), il valore viene tradotto in Swift utilizzando il metodo `visit_expression`; se, invece, l'inizializzazione avviene tramite una lettura da input (`ctx.readStatement()`), il valore viene elaborato con il metodo `visit_read_statement`. Infine, il metodo costruisce e ritorna una stringa rappresentante la dichiarazione Swift nel formato `var` *`var_name`* `:` *`var_type`* `=` *`var_value`* per variabili mutabili, `let` *`var_name`* `:` *`var_type`* `=` *`var_value`* per variabili immutabili, oppure `None` in caso di errori. In Kotlin e Swift, `var` viene usato per dichiarare variabili mutabili, senza differenze tra i due linguaggi. In Kotlin, `val` rende immutabile il riferimento della variabile, ma consente modifiche al contenuto di oggetti mutabili. In Swift, `let` rende immutabili sia il riferimento che l'oggetto. Poiché il transpiler supporta solo tipi intrinsecamente immutabili come `String`, `Boolean` e `Int`, la distinzione tra `val` e `let` non sussiste, quindi `val` è tradotto direttamente in `let`.

Il metodo `visit_assignment_statement` gestisce sia le semplici assegnazioni di variabili sia le chiamate a funzioni. Prende in input un oggetto `AssignmentStatementContext` che include i metodi per accedere ai componenti della dichiarazione: il metodo `IDENTIFIER()` ritorna il nodo contenente il nome della variabile; il metodo `expression()` consente di accedere all'espressione di inizializzazione; il metodo `readStatement()` fornisce accesso a una dichiarazione di lettura da input; il metodo `callExpression` fornisce accesso a una chiamata di funzione. La gestione delle chiamate a funzioni in questo contesto rappresenta un workaround reso necessario dall'ambiguità tra l'istruzione di assegnazione e l'espressione di chiamata a funzione: entrambe condividono una struttura sintattica simile. Per evitare conflitti nell'analisi del codice, i due casi vengono gestiti all'interno della stessa regola. Se l'assegnazione è una chiamata a funzione (`ctx.callExpression()`), la traduzione è delegata al metodo `visit_call_expression`. In caso contrario, il metodo elabora un'assegnazione diretta di variabile: il nome della variabile viene determinato tramite `visit_identifier` e, nel caso di un'inizializzazione basata su espressione (`ctx.expression()`), il valore viene tradotto con `visit_expression`, mentre per assegnazioni basate su `readStatement`, il valore è tradotto con `visit_read_statement`. Infine, il metodo ritorna la dichiarazione Swift nel formato *`var_name`* `=` *`var_value`* o, in caso di errori, `None`.

Il metodo `visit_function_declaration` gestisce le dichiarazioni di funzioni. Prende in input un oggetto `FunctionDeclarationContext`, generato da ANTLR per la regola `functionDeclaration`, che fornisce metodi per accedere ai componenti della dichiarazione di funzione: `IDENTIFIER()` ritorna il nodo contenente il nome della funzione, `parameterList()` permette di accedere alla lista dei parametri, `type_()` ritorna il contesto del tipo di ritorno (se presente) e `block()` consente di ottenere il corpo della funzione. La traduzione inizia con la generazione del nome della funzione tramite il metodo `visit_identifier` e la creazione della lista dei parametri tramite `visit_parameter_list`. Il corpo della funzione viene tradotto tramite `visit_block`, mentre il tipo di ritorno viene elaborato attraverso `visit_type`, se specificato. Infine,

il metodo ritorna la dichiarazione Swift della funzione `func` *`fun_name`*`(`*`parameters`*`) ->` *`return_type`* `{` *`body`* `}`, omettendo il tipo di ritorno se non dichiarato.

Il metodo `visit_return_statement` analizza il contesto rappresentato dall'oggetto `ReturnStatementContext`, generato da ANTLR per la regola `returnStatement`, che offre i metodi `RETURN()` per identificare il token di ritorno e `expression()` per accedere all'eventuale espressione associata al valore restituito. La traduzione viene effettuata distinguendo due casi: se l'istruzione `return` contiene un'espressione (determinata dalla presenza di `ctx.expression()`), l'espressione viene tradotta mediante il metodo `visit_expression`, e il risultato viene integrato nella sintassi di Swift come `return` *`expression`*. Nel caso in cui non sia presente alcuna espressione, il metodo ritorna semplicemente la stringa `return`, corrispondente all'istruzione di ritorno vuota.

Il metodo `visit_for_statement` traduce un ciclo `for`. Prende in input un oggetto `ForStatementContext`, generato da ANTLR per la regola `forStatement`, che rappresenta l'istruzione di iterazione e fornisce il metodo `membershipExpression()` per accedere all'espressione di appartenenza e i metodi `block()` e `statement()` per accedere al corpo del ciclo. Il metodo invoca `visit_membership_expression` per tradurre la condizione di iterazione e gestisce il corpo del ciclo con `visit_block` se è un blocco, con `visit_statement` se si tratta di una singola istruzione. Infine, il metodo restituisce una stringa che rappresenta il ciclo `for` in Swift, nella forma `for` *`expression`* `{` *`body`* `}`, con il corpo racchiuso tra parentesi graffe se si tratta di un blocco.

Il metodo `visit_if_else_statement` traduce un'istruzione `if-else`, con il blocco `else` opzionale. Prende in input un oggetto `IfElseStatementContext`, generato da ANTLR per la regola `ifElseStatement`, che rappresenta l'istruzione di diramazione. Esso fornisce i metodi `expression()` per l'espressione condizionale, `ifBody()` per il corpo del blocco `if`, e `elseBody()` per il corpo del blocco `else`. Il metodo utilizza `visit_expression` per tradurre la condizione e invoca `visit_if_body` e `visit_else_body` per tradurre i rispettivi blocchi di codice. Ritorna una stringa che rappresenta l'istruzione `if-else` in Swift, nel formato `if` *`expression if_body`* oppure `if` *`expression if_body`* `else` *`else_body`*, a seconda della presenza del blocco `else`. I metodi `visit_if_body` e `visit_else_body` traducono i corpi delle istruzioni `if` e `else`. Prendono in input oggetti `IfBodyContext` e `ElseBodyContext`, che rappresentano i rispettivi corpi. Questi oggetti forniscono i metodi `block()` per accedere a un blocco di istruzioni e `statement()` per una singola istruzione. Se il corpo è un blocco, viene tradotto tramite `visit_block`, altrimenti tramite `visit_statement`. Entrambi i metodi restituiscono una stringa che rappresenta il corpo dell'istruzione `if` o `else` in Swift, racchiuso tra parentesi graffe se si tratta di un blocco.

Il metodo `visit_expression` rappresenta il punto di ingresso per visitare le espressioni. Prende in input un oggetto `ExpressionContext`, generato da ANTLR per la regola `expression`, che rappresenta un'espressione generica e fornisce il metodo `logicalOrExpression` per accedere a un'espressione logica OR. Gestisce qualsiasi espressione delegando la traduzione al metodo `visit_logical_or_expression`, e ritorna una stringa con il codice Swift equivalente.

I metodi `visit_logical_or_expression`, `visit_logical_and_expression` e `visit_equality_expression` gestiscono rispettivamente le espressioni logiche OR (`||`), AND (`&&`) e di uguaglianza (`==`, `!=`). Ciascun metodo prende in input il contesto generato da ANTLR per la relativa regola grammaticale (`LogicalOrExpressionContext` per `logicalOrExpression`, `LogicalAndExpressionContext` per `logicalAndExpression`

o `EqualityExpressionContext` per `equalityExpression`) e accede agli operandi attraverso i metodi associati (`logicalAndExpression`, `equalityExpression` o `relationalExpression`). Ogni metodo segue un approccio modulare: elabora sempre il primo operando e verifica la presenza di operandi aggiuntivi, che vengono elaborati uno alla volta e concatenati con l'operatore logico o di uguaglianza corrispondente. Se è presente un solo operando, viene restituito direttamente l'operando stesso. Questo approccio ricorsivo garantisce una gestione modulare e coerente delle espressioni.

Il metodo `visit_relational_expression` traduce espressioni relazionali (`<`, `>`, `<=`, `>=`). Prende in input un oggetto `RelationalExpressionContext`, generato da ANTLR per la regola `relationalExpression`, e accede agli operandi attraverso `additiveExpression`. Elabora sempre il primo operando utilizzando `visit_additive_expression`. Se è presente un secondo operando, lo concatena con l'operatore relazionale corrispondente; altrimenti, ritorna il primo operando.

I metodi `visit_additive_expression` e `visit_multiplicative_expression` si occupano delle espressioni additive (`+`, `-`) e moltiplicative (`*`, `/`, `%`). Entrambi prendono in input un oggetto generato da ANTLR per le rispettive regole grammaticali (`AdditiveExpressionContext` per `additiveExpression` e `MultiplicativeExpressionContext` per `multiplicativeExpression`), che forniscono metodi per accedere agli operandi sottostanti. Ogni metodo elabora sempre il primo operando, visitando ricorsivamente il contesto corrispondente. Se sono presenti ulteriori operandi, questi vengono elaborati uno alla volta e concatenati utilizzando gli operatori pertinenti. Se è presente un solo operando, viene restituito l'operando stesso. Questo approccio modulare e ricorsivo consente di gestire in modo efficiente sia espressioni semplici che complesse, sfruttando la struttura ricorsiva della grammatica.

Il metodo `visit_unary_expression` traduce le espressioni unarie, come la negazione logica (`!`) e la negazione aritmetica (`-`). Prende in input un oggetto `UnaryExpressionContext`, generato da ANTLR per la regola `unaryExpression`. Questo oggetto rappresenta un'espressione unaria e fornisce i metodi `membershipExpression` per accedere a un'espressione di appartenenza e `primaryExpression` per accedere a un'espressione primaria. Il metodo controlla la presenza dei token `NOT` o `MINUS` nel contesto dell'espressione unaria e applica l'operatore Swift corrispondente; altrimenti, il metodo delega la gestione delle espressioni di appartenenza al metodo `visit_membership_expression`.

Il metodo `visit_membership_expression` traduce le espressioni di appartenenza, come `a in 1..10`. Prende in input un oggetto `MembershipExpressionContext`, generato da ANTLR per la regola `membershipExpression`, che rappresenta l'espressione di appartenenza e fornisce i metodi `primaryExpression` per accedere alle espressioni primarie e `rangeExpression` agli intervalli. Il metodo verifica la presenza degli operatori `in` o `!in`, elabora i lati sinistro e destro dell'espressione e applica gli operatori Swift appropriati. Se è presente un intervallo, lo visita e lo include nella stringa finale; altrimenti, ritorna solo il lato sinistro dell'espressione.

Il metodo `visit_primary_expression` traduce le espressioni primarie, che possono essere identificatori, espressioni tra parentesi, chiamate di funzione o letterali. Prende in input un oggetto `PrimaryExpressionContext`, generato da ANTLR per la regola `primaryExpression`, che fornisce i metodi `IDENTIFIER` per accedere a un identificatore, `expression` per accedere a un'espressione tra parentesi, `callExpression` per accedere alle chiamate di funzione e `literal` per accedere a un letterale. Se l'espressione è un identificatore, ritorna il nome dell'identificatore. Se è un'espressione tra parentesi, visita ricorsivamente l'espressione interna. Per una chiamata di funzione, invoca il metodo

`visit_call_expression`, mentre per un letterale chiama `visit_literal`. Il risultato finale è il codice Swift equivalente.

Il metodo `visit_range_expression` traduce le espressioni di intervallo, come `a..b`. Prende in input un oggetto `RangeExpressionContext`, generato da ANTLR per la regola `rangeExpression`, che fornisce il metodo `visit_additive_expression` per accedere a un'espressione additiva. Il metodo esamina i due operandi dell'espressione di intervallo e ritorna la rappresentazione Swift con i tre puntini (`...`) tra di essi.

Il metodo `visit_call_expression` traduce le chiamate di funzione. Prende in input un oggetto `CallExpressionContext`, generato da ANTLR per la regola `callExpression`, che fornisce i metodi `IDENTIFIER()` per estrarre il nome della funzione, e `argumentList()` per estrarre gli eventuali argomenti. Il metodo visita l'identificatore della funzione e, se presenti, gli argomenti tramite `visit_argument_list`. Successivamente, costruisce la stringa corrispondente alla chiamata di funzione in Swift: la sintassi è del tipo *`fun_name`*`(`*`argument_list`*`)` se sono presenti argomenti, *`fun_name`*`()` altrimenti.

Il metodo `visit_argument_list` traduce una lista di argomenti in una chiamata di funzione, rappresentata da un oggetto `ArgumentListContext`, generato da ANTLR per la regola `argumentList`. Quest'ultimo include il metodo `argument` per accedere ai singoli argomenti presenti nella lista. Il metodo itera sugli argomenti e invoca `visit_argument` per ciascun argomento. Infine, ritorna una stringa contenente tutti gli argomenti elaborati, separati da virgole. Il metodo `visit_argument` traduce un singolo argomento rappresentato da un oggetto `ArgumentContext`, generato da ANTLR per la regola `argument`. Quest'ultimo include i metodi `IDENTIFIER` per riconoscere l'identificatore del parametro e `expression` per accedere a un eventuale valore associato. Se l'argomento è nominato, il metodo utilizza `visit_identifier` per estrarre il nome e `visit_expression` per il valore; altrimenti, elabora solo il valore tramite `visit_expression`. Alla fine, il metodo ritorna l'argomento Swift nella forma *`argument_name`* `:` *`argument_value`*.

Il metodo `visit_literal` gestisce le espressioni letterali, come valori numerici, booleani o stringhe, rappresentate da un oggetto `LiteralContext`, generato da ANTLR per la regola `literal`. Quest'ultimo fornisce i metodi `INT_LITERAL`, `STRING_LITERAL` e `booleanLiteral`, per accedere ai rispettivi tipi di letterali. Il metodo `visit_literal` ritorna direttamente il testo dell'espressione letterale, senza modificarlo o interpretarlo.

Il metodo `visit_type` traduce i tipi di dato. Prende in input un oggetto `TypeContext`, che rappresenta un tipo e fornisce i metodi `TYPE_INT`, `TYPE_STRING` e `TYPE_BOOLEAN`, per riconoscere i rispettivi tipi Kotlin. Utilizza la mappa `KOTLIN_2_SWIFT_TYPES` per restituire il tipo Swift equivalente. Se il tipo Kotlin non è supportato, il metodo ritorna `None`, altrimenti ritorna il tipo Swift corrispondente.

Il metodo `visit_read_statement` analizza il contesto dell'istruzione `readLine()` utilizzando l'oggetto `ReadStatementContext`, generato da ANTLR per la regola `readStatement`, che fornisce il metodo `READLINE()` per identificare il token `readLine()`. Il metodo ritorna la stringa `readLine()` in Swift, che corrisponde direttamente alla funzione `readLine()` di Kotlin, senza necessitare di modifiche sintattiche.

Il metodo `visit_print_statement` analizza il contesto dell'istruzione `println()` attraverso l'oggetto `PrintStatementContext`, generato da ANTLR per la regola `printStatement`, che offre il metodo `PRINTLN()` per identificare il token `println()`. L'espressione all'interno dell'istruzione di stampa viene elaborata dal metodo `visit_expression`. Il risultato finale è una stringa Swift che rappresenta la funzione `print()` con l'espressione tradotta come argomento, nella forma `print(`*`expression`*`)`.

5. Gestione degli Errori

Il processo di *error handling* si articola in più fasi, progettate per garantire la validità e la correttezza del codice tradotto. La gestione degli errori inizia con l'analisi lessicale, che identifica token malformati o caratteri non validi, e prosegue con l'analisi sintattica, volta a verificare la conformità del codice alla grammatica definita. In entrambe le fasi, gli eventuali errori vengono raccolti senza bloccare immediatamente il processo. Alla fine, se sono stati rilevati errori in almeno una delle due fasi, questi vengono mostrati all'utente e il processo di transpiling viene interrotto, evitando la generazione di codice Swift non valido. In caso contrario, il transpiler avvia la visita del *parse tree*, che integra l'analisi semantica, finalizzata a garantire che il codice abbia senso dal punto di vista logico: per ogni nodo dell'albero vengono eseguiti i relativi controlli logici, contestualmente alla generazione del codice Swift. Al termine del processo, se sono stati rilevati errori semantici, questi vengono mostrati all'utente e il processo di traduzione viene interrotto per evitare la generazione di codice Swift logicamente errato; altrimenti, il codice Swift viene generato correttamente.

5.1 Gestione degli Errori Lessicali e Sintattici

Il sistema di gestione degli errori lessicali e sintattici si basa sulla classe `ErrorListener` di ANTLR: quando il lexer o il parser incontrano un errore, viene automaticamente invocato il metodo `syntaxError` della classe `ErrorListener`, che fornisce diverse informazioni, come la posizione dell'errore (riga e colonna), il simbolo che ha causato l'errore e un messaggio descrittivo. Estendendo la classe `ErrorListener` è possibile personalizzarne il comportamento: attraverso l'override del metodo `syntaxError`, ad esempio, è possibile raccogliere i messaggi di errore in una struttura dati, formattarli per una migliore leggibilità o implementare strategie specifiche di gestione.

Per gestire separatamente gli errori lessicali e sintattici, sono state implementate due classi personalizzate, `LexicalErrorListener` e `SyntaxErrorListener`, che estendono `ErrorListener` offrendo un controllo preciso e dettagliato sulle fasi di tokenizzazione e parsing del codice. La classe `LexicalErrorListener` è responsabile della gestione degli errori durante la fase di tokenizzazione, come token malformati o caratteri non validi, mentre la classe `SyntaxErrorListener` si occupa di rilevare errori durante la fase di parsing, come violazioni della grammatica definita. Entrambe le classi personalizzate sovrascrivono il metodo `syntaxError`: i messaggi di errore vengono formattati in modo da fornire informazioni dettagliate, come la riga e la colonna in cui si è verificato l'errore, nonché una descrizione del tipo di problema riscontrato. Inoltre, sono stati implementati i metodi `has_errors` e `get_errors`, per verificare se sono presenti errori e accedere agli errori rilevati, rispettivamente.

5.2 Gestione degli Errori Semantici

Per la gestione degli errori semantici, è stata implementata la classe personalizzata `SemanticErrorListener`, la cui struttura è analoga a quella delle classi `SyntaxErrorListener` e `LexicalErrorListener`. Include metodi per formattare messaggi di errore dettagliati (`semantic_error`), verificare la presenza di errori (`has_errors`) e recuperarne i dettagli (`get_errors`). La principale differenza risiede nel fatto che non

estende la classe `ErrorListener`, ma è gestita manualmente: i controlli semantici vengono effettuati esplicitamente all'interno del visitor, durante la visita dei nodi del *parse tree*, e il metodo `semantic_error` viene invocato solo in caso di violazioni semantiche.

Per supportare adeguatamente i controlli semantici, è stata implementata nel file `SymbolTable.py` la tabella dei simboli, componente fondamentale per gestire le dichiarazioni di variabili, funzioni e classi all'interno degli scope annidati, garantendo che vengano rispettate le regole semantiche del linguaggio. La tabella dei simboli implementa lo scoping a livello globale, di classi e funzioni. Non è stato implementato, invece, lo scoping per blocchi come quelli definiti da istruzioni `if` e `for`: questa scelta riflette la natura semplificata del linguaggio tradotto, in cui non si prevede un utilizzo avanzato degli scope locali. La tabella dei simboli è stata progettata come uno stack di scope, ognuno dei quali è rappresentato da un dizionario contenente le dichiarazioni di variabili, funzioni e classi. L'utilizzo dello stack per gestire gli scope consente di implementare un controllo preciso della visibilità dei simboli, assicurando che quelli dichiarati in uno scope annidato siano visibili solo all'interno degli scope superiori. Quando si entra in un nuovo scope, ad esempio all'interno di una funzione, viene creato un nuovo livello nello stack, mentre quando si esce da uno scope, questo viene rimosso dallo stack, rendendo i simboli in esso contenuti non più accessibili. Inizialmente, la tabella contiene uno scope globale vuoto, utilizzato per memorizzare le dichiarazioni globali. Per gestire dinamicamente gli scope, la tabella dei simboli mette a disposizione i metodi `add_scope()` per aggiungere uno scope e `remove_scope()` per rimuovere lo scope corrente. Lo scope globale, tuttavia, non può essere rimosso. Ogni dizionario associato a uno scope include tre categorie principali: `variables`, che contiene le variabili, `functions`, che memorizza le funzioni e `classes`, che conserva le classi dichiarate nello scope.

Per quanto riguarda le variabili, queste sono rappresentate da oggetti della classe `Symbol`, che memorizza informazioni come il nome, il tipo, la mutabilità e il valore della variabile. Le variabili vengono aggiunte o aggiornate nello scope corrente tramite i metodi `add_variable()` e `update_variable()`. La ricerca di una variabile può avvenire all'interno del solo scope corrente, tramite `lookup_variable_in_current_scope()`, oppure in tutti gli scope attivi, tramite `lookup_variable()`. Inoltre, è possibile ottenere informazioni specifiche sulle variabili attraverso i metodi `get_variable_info` e `get_variable_assigned`. Le funzioni sono memorizzate in un dizionario che contiene il nome della funzione, il tipo di ritorno e i tipi e valori dei parametri. Ogni funzione è identificata in modo univoco dalla sua firma, che consiste nel nome e nei tipi dei parametri. La tabella dei simboli supporta il sovraccarico delle funzioni, permettendo di avere più versioni della stessa funzione con firme differenti. In caso di sovraccarico, viene utilizzato un unico dizionario per tutte le versioni della stessa funzione, in cui il nome e il tipo di ritorno rimangono invariati, mentre per ogni versione sovraccaricata vengono aggiunti i tipi e i valori dei parametri. Le funzioni vengono aggiunte tramite il metodo `add_function()` e possono essere ricercate con `lookup_function()`. È inoltre possibile ottenere informazioni specifiche sulle funzioni tramite i metodi `get_function_return_type()` e `get_function_params()`. Le classi, infine, sono gestite tramite un set che memorizza i nomi delle classi dichiarate in ogni scope. Ogni classe è identificata univocamente dal suo nome, e viene aggiunta tramite il metodo `add_class()`. La ricerca di una classe avviene tramite il metodo `lookup_class()`.

Per garantire una corretta analisi e traduzione del codice, è fondamentale che le variabili e le funzioni siano dichiarate prima del loro utilizzo. Questo requisito deriva dal fatto che il transpiler elabora il codice sequenzialmente, visitando ogni istruzione nell'ordine in

cui appare: se una funzione o una variabile venisse utilizzata prima della sua dichiarazione, il sistema non sarebbe in grado di riconoscerla correttamente al momento dell'accesso, generando un errore semantico per segnalare il riferimento a un'entità non definita.

I controlli semantici sono implementati all'interno di funzioni dedicate nella classe `KotlinToSwiftVisitor` e vengono eseguiti durante la visita del *parse tree*, garantendo che il codice rispetti le regole semantiche prima di essere trascritto in Swift. Di seguito vengono descritti i principali controlli semantici implementati.

La gestione delle classi si concentra sulla corretta dichiarazione e visibilità delle classi nel codice, con l'obiettivo di evitare dichiarazioni duplicate e garantire un utilizzo appropriato. In particolare, il metodo `check_class_already_declared_in_current_scope` verifica se una classe è già stata dichiarata all'interno dello scope corrente, consultando la tabella dei simboli per controllare l'esistenza di una classe con lo stesso nome. Se la classe risulta già dichiarata, il metodo solleva un errore semantico.

Per quanto riguarda la gestione di variabili e costanti, il transpiler verifica che queste siano dichiarate e inizializzate prima dell'uso, controllando anche la coerenza dei tipi per prevenire operazioni non valide; inoltre, assicura che le variabili immutabili non vengano modificate dopo l'assegnazione iniziale.

Il controllo semantico sulle variabili duplicate è gestito dai metodi `check_variable_already_declared_in_current_scope` e `check_variable_already_declared`. Il primo metodo è progettato per il controllo delle dichiarazioni di nuove variabili all'interno dello scope corrente, sia all'interno di un blocco di codice, sia come parametro di una funzione, per garantire che non esista già una variabile con lo stesso nome nello stesso scope. Il secondo metodo, invece, verifica se una variabile è già stata dichiarata in uno qualsiasi degli scope accessibili ed è particolarmente utile quando si verificano espressioni o assegnazioni, in quanto la variabile potrebbe essere stata dichiarata in uno scope precedente, e utilizzata nello scope corrente: in altre parole, il metodo è utilizzato per garantire che la variabile venga effettivamente dichiarata prima del suo utilizzo, evitando accessi a variabili non dichiarate. Entrambi i metodi consultano la tabella dei simboli per verificare se una variabile con lo stesso nome è già presente e, se viene rilevata una variabile duplicata, sollevano un errore semantico. Tuttavia, ciascun metodo si applica in contesti diversi: il primo per prevenire conflitti durante la dichiarazione di nuove variabili, il secondo per evitare il riuso di variabili già dichiarate.

Il metodo `check_variable_already_assigned` verifica che una variabile sia stata assegnata prima del suo utilizzo. Prima di eseguire il controllo sull'assegnazione, verifica se la variabile è stata dichiarata utilizzando il metodo `check_variable_already_declared`: solo se la variabile è dichiarata, viene controllata l'assegnazione nella tabella dei simboli. Se la variabile risulta dichiarata ma non assegnata, viene sollevato un errore semantico. Questo metodo, infatti, è utilizzato in altri metodi, come `check_membership_expression_type` e `check_primary_expression_type`, per garantire che le variabili siano correttamente dichiarate e assegnate prima del loro utilizzo.

Il metodo `check_mutability` viene utilizzato per validare la mutabilità di una variabile prima di consentire un'assegnazione, garantendo che le variabili immutabili non vengano modificate dopo la loro assegnazione iniziale. Questo metodo è invocato durante le operazioni di assegnazione, in cui verifica se la variabile è mutabile o immutabile. Se la variabile è immutabile e si tenta di assegnarle un nuovo valore, il metodo utilizza

`check_variable_not_assigned` per controllare che la variabile non sia già stata assegnata. Se la variabile è immutabile e già assegnata, l'assegnazione non è permessa e viene segnalato un errore; se, invece, la variabile è mutabile, o immutabile e non assegnata, l'assegnazione è permessa. Un caso particolare riguarda i parametri delle funzioni, che sono immutabili per definizione. Se un parametro ha un valore di default, esso non può essere modificato, come previsto. Tuttavia, se il parametro non è inizializzato con un valore di default, l'attuale implementazione consente la sua modifica, violando il principio di immutabilità. Questo comportamento si applica anche alle proprietà immutabili delle classi e rappresenta una limitazione del sistema, che potrebbe essere risolta aggiungendo un controllo specifico per impedire le modifiche a parametri delle funzioni e proprietà immutabili delle classi, a prescindere dalla loro configurazione iniziale.

Il metodo `validate_value` è utilizzato per convalidare il valore assegnato a una variabile, verificando la compatibilità tra il tipo del valore e il tipo dichiarato per la variabile. Questo controllo viene eseguito in vari contesti, tra cui il metodo `visit_assignment_statement`, che si occupa di elaborare le dichiarazioni di assegnazione, il metodo `visit_var_declaration`, che valida la dichiarazione delle variabili, e il metodo `validate_return_statement`, che verifica che il tipo del valore restituito corrisponda al tipo di ritorno dichiarato per la funzione. Il metodo `validate_value` sfrutta `check_expression_type` per determinare il tipo dell'espressione assegnata e, se questo non corrisponde al tipo dichiarato per la variabile, solleva un errore semantico.

Il metodo `check_supported_type` si occupa di verificare che un tipo Kotlin sia supportato dal transpiler: il controllo avviene confrontando il tipo di dato specificato con quelli definiti nell'enumerazione `KotlinTypes`, che rappresenta i tipi supportati dal transpiler. Se il tipo specificato non è supportato, viene generato un errore semantico. Questo metodo viene impiegato in vari contesti, come nel metodo `visit_var_declaration`, che assicura che il tipo di una variabile sia valido, nel metodo `visit_function_declaration`, che verifica la validità del tipo di ritorno delle funzioni, e nel metodo `check_parameter_type`, che garantisce la validità dei tipi dei parametri delle funzioni.

Per quanto riguarda le espressioni, sono stati implementati controlli semantici per rilevare l'uso di tipi incompatibili o l'uso improprio degli operatori.

Il metodo `check_expression_type` è il punto di ingresso per il controllo del tipo di un'espressione e si occupa di delegare il controllo a metodi specifici. Il controllo parte da un'espressione logica OR e prosegue lungo la catena di espressioni in base ai tipi di operatori coinvolti, seguendo la gerarchia dei metodi di visita delle espressioni.

Le espressioni logiche OR e AND sono gestite rispettivamente dai metodi `check_logical_or_expression_type` e `check_logical_and_expression_type`. Quando l'espressione coinvolge due o più operandi, i metodi verificano che tutti gli operandi siano di tipo `Boolean`. Se la condizione è soddisfatta, il tipo risultante dell'espressione è `Boolean`; in caso contrario, viene generato un errore semantico. Nel caso di un singolo operando, il metodo restituisce direttamente il tipo di quest'ultimo, rinviando eventuali verifiche aggiuntive ai controlli successivi.

I metodi `check_equality_expression_type` e `check_relational_expression_type` si occupano invece delle espressioni di uguaglianza e relazionali. In entrambi i casi, se sono presenti due o più operandi, viene verificato che tutti siano dello stesso tipo. Per le espressioni relazionali, viene inoltre controllato che il tipo degli operandi sia compatibile con l'operazione, ossia `Int`. Se le condizioni sono verificate, il tipo restituito è `Boolean`; in caso contrario, viene segnalato un errore semantico. Nel caso di un singolo operando, il metodo restituisce direttamente il tipo di quest'ultimo, rinviando eventuali verifiche

aggiuntive ai controlli successivi.

Le espressioni additive e moltiplicative sono gestite dai metodi `check_additive_expression_type` e `check_multiplicative_expression_type` che, nel caso ci siano due o più operandi, verificano che questi siano dello stesso tipo e interi. Se la condizione è soddisfatta, il tipo restituito è `Int`; in caso contrario, viene generato un errore semantico. Nel caso di un singolo operando, il metodo restituisce direttamente il tipo di quest'ultimo, rinviando eventuali verifiche aggiuntive ai controlli successivi.

Il metodo `check_unary_expression_type` gestisce le espressioni unarie. Verifica che l'operando sia compatibile con l'operatore: la negazione logica è valida solo per espressioni di tipo `Boolean`, mentre la negazione aritmetica è applicabile solo su espressioni di tipo `Int`. Se l'operazione è valida, viene restituito il tipo dell'espressione elaborata; se ci sono incompatibilità, viene segnalato un errore semantico.

Per le espressioni di membership, come l'operatore `in`, il metodo `check_membership_expression_type` verifica che l'espressione a sinistra dell'operatore sia una variabile dichiarata e assegnata correttamente, di tipo `Int` e mutabile. Se l'operazione è valida, viene restituito il tipo dell'espressione elaborata; se ci sono incompatibilità, viene segnalato un errore semantico.

Le espressioni di intervallo, rappresentate dall'operatore `..`, sono verificate dal metodo `check_range_expression_type`, che assicura che entrambi gli operandi siano di tipo `Int`, segnalando un errore semantico nel caso contrario. Anche in questo caso, se l'operazione è valida, viene restituito il tipo dell'espressione elaborata; se ci sono incompatibilità, viene segnalato un errore semantico.

Infine, il metodo `check_primary_expression_type` gestisce le espressioni primarie come variabili, espressioni tra parentesi, letterali e chiamate a funzione. Per le variabili, controlla che siano dichiarate e assegnate utilizzando i metodi `check_variable_already_declared` e `check_variable_already_assigned`; le chiamate a funzione sono verificate tramite il metodo `check_call_expression`; i letterali sono validati con `check_literal_type`, che determina il tipo (ad esempio, `Int`, `String`, `Boolean`) o solleva un errore semantico se il tipo non è supportato; per le espressioni tra parentesi, il tipo dell'espressione è verificato ricorsivamente tramite `check_expression_type`.

La gestione delle funzioni include il controllo delle dichiarazioni duplicate, la validazione dei tipi di ritorno e dei parametri, la verifica dell'esistenza delle funzioni invocate nonché la verifica della corrispondenza tra argomenti e parametri nelle invocazioni.

Il metodo `check_function_already_declared_in_current_scope` viene utilizzato durante la dichiarazione di una funzione per evitare la duplicazione di funzioni con la stessa firma. Il metodo consulta la tabella dei simboli per verificare se una funzione con lo stesso nome e gli stessi tipi di parametri esiste già nello scope corrente. In caso affermativo, viene sollevato un errore semantico.

Il metodo `check_function_not_declared_in_current_scope` viene utilizzato per verificare se una funzione viene invocata prima della sua dichiarazione nello scope corrente. Quando una funzione viene invocata, il metodo consulta la tabella dei simboli per verificare se esiste una funzione con la stessa firma, ovvero con lo stesso nome e la stessa lista di tipi di argomenti. Se non viene trovata alcuna corrispondenza, viene sollevato un errore semantico. Questo controllo viene applicato nei metodi `check_call_expression`, che si occupa di garantire che la funzione chiamata sia stata dichiarata correttamente e

che la firma corrisponda, e `check_argument_types`, che verifica che i tipi degli argomenti passati alla funzione corrispondano ai tipi dei parametri dichiarati nella funzione.

Il metodo `check_call_expression` viene utilizzato all'interno del metodo `visit_call_expression` per verificare che una chiamata a funzione sia dichiarata correttamente e che i suoi parametri siano conformi alla firma definita. Il processo inizia con l'estrazione del nome della funzione e dei tipi degli argomenti. Successivamente, il metodo verifica che la funzione sia già stata dichiarata correttamente, con la firma attesa. Se la funzione non è dichiarata o la firma non corrisponde, viene generato un errore semantico.

I controlli semantici sui parametri di una funzione sono progettati per garantire che i parametri siano dichiarati correttamente e che non siano duplicati. Il metodo `check_parameter_type_list` si occupa della validazione di una lista di parametri. La funzione itera su ciascun parametro, applicando il controllo di tipo definito in `check_parameter_type`, che verifica se il tipo di un parametro appartiene a un insieme di tipi supportati. Se il tipo non è valido, viene sollevato un errore semantico. Il metodo `check_parameter_name_value_list` restituisce una stringa contenente i nomi e i valori di ciascun parametro, elaborati da `check_parameter_name_value`. Quest'ultimo verifica sia il nome che il valore di ciascun parametro. In un contesto semantico, il controllo assicura che ogni parametro abbia un nome e, se previsto, un valore associato. Se un parametro non ha un valore associato, viene restituito `name: None`. Questo metodo è utile per la validazione dei parametri con valori opzionali o non dichiarati esplicitamente. Il metodo `check_duplicate_parameters` verifica che non vi siano parametri duplicati all'interno di una funzione. Se vengono rilevati parametri duplicati, il metodo solleva un errore semantico.

Nel caso dei parametri senza valore di default, la logica di analisi semantica potrebbe erroneamente generare un errore di *variabile non assegnata* quando i parametri vengono utilizzati all'interno del corpo della funzione: questo accade perché il controllo sull'assegnazione avviene prima che i parametri vengano effettivamente valorizzati al momento della chiamata a funzione (il comportamento corretto, infatti, prevede che i parametri vengano assegnati durante la chiamata a funzione e non all'interno del corpo della funzione stessa). Un problema analogo si verifica anche per le proprietà immutabili delle classi, che vengono valorizzate nel momento in cui l'oggetto viene istanziato. Questo problema rappresenta una limitazione del sistema, che potrebbe essere risolta implementando una logica che escluda i parametri delle funzioni e le proprietà immutabili delle classi dal controllo sull'assegnazione, riconoscendo che questi elementi possono essere valorizzati successivamente.

I controlli semantici sugli argomenti di una funzione sono finalizzati a verificare che gli argomenti passati siano dichiarati correttamente e che i loro nomi e tipi siano coerenti con quelli definiti nella firma della funzione. Il metodo `check_arguments` integra i controlli relativi alla validità dei tipi e dei nomi degli argomenti, eseguiti dai metodi `check_argument_types` e `check_argument_names`. In particolare, `check_argument_types` si occupa di verificare che i tipi degli argomenti in una chiamata di funzione corrispondano ai tipi dei parametri della dichiarazione della stessa funzione: se non viene trovata una corrispondenza, viene generato un errore semantico. Per fare ciò, il metodo `check_argument_types` analizza la lista degli argomenti tramite il metodo `check_argument_type_list`, che verifica il tipo di ciascun argomento utilizzando a sua volta il metodo `check_argument_type`: quest'ultimo determina il tipo di un singolo argomento basandosi sul tipo dell'espressione associata. Analogamente, `check_argument_names` verifica che i nomi degli argomenti siano coerenti con i nomi

dei parametri definiti nella firma della funzione: se non viene trovata una corrispondenza, viene generato un errore semantico. Questa verifica avviene tramite il metodo `check_argument_name_list`, che raccoglie e verifica i nomi degli argomenti passati alla funzione, utilizzando a sua volta il metodo `check_argument_name` per estrarre il nome di ogni singolo argomento, oppure la stringa `"None"` se l'argomento non ha un identificatore.

I controlli semantici sul tipo di ritorno delle funzioni gestiscono i casi in cui il tipo dichiarato e il valore effettivamente restituito possono entrare in conflitto. Si distinguono due scenari principali: funzioni senza tipo di ritorno e funzioni con tipo di ritorno dichiarato. Nel primo caso, il corpo della funzione non deve contenere istruzioni di ritorno. Nel secondo caso, è necessario verificare che siano presenti istruzioni di ritorno che restituiscano valori compatibili con il tipo dichiarato, anche nei blocchi `for` e nei blocchi `if-else` dove entrambi i rami devono contenere istruzioni di ritorno valide. Eventuali discrepanze sollevano errori semantici. La funzione `check_return_statement` verifica che il corpo della funzione rispetti questi vincoli, controllando sia la presenza sia la correttezza delle istruzioni di ritorno. Ogni valore restituito è validato tramite il metodo `validate_return_statement`, che confronta il tipo dell'espressione con il tipo dichiarato nella funzione. Per scenari complessi, come strutture condizionali o cicli, vengono utilizzati metodi specifici. La funzione `check_return_statement_in_if_else_statement` assicura che nei blocchi `if-else` siano presenti istruzioni di ritorno coerenti con il tipo dichiarato, mentre `check_return_statement_in_for_statement` effettua verifiche analoghe nei cicli `for`. Per funzioni senza tipo di ritorno, i metodi `check_no_return_statement_in_if_else_statement` e `check_no_return_statement_in_for_statement` verificano che non siano presenti istruzioni di ritorno all'interno di blocchi condizionali o cicli.

Infine, per le strutture condizionali e iterative, vengono validati i tipi delle espressioni all'interno dei blocchi `if` e dei cicli `for`, per garantire che rispettino i requisiti semantici. Il metodo `validate_if_condition` è utilizzato all'interno del metodo `visit_if_else_statement` per convalidare la condizione di un'istruzione `if`. Questo verifica che la condizione di diramazione sia di tipo booleano e solleva un errore semantico nel caso in cui la condizione non corrisponda al tipo atteso. Il metodo `check_membership_expression_type`, utilizzato all'interno di `visit_for_statement`, verifica la correttezza della condizione di iterazione nei cicli `for`. Inizialmente, controlla che la variabile sul lato sinistro dell'operatore `in` sia dichiarata, assegnata, di tipo `Int` e mutabile. Questa verifica è necessaria poiché non è stata implementata la dichiarazione implicita della variabile nella condizione del ciclo `for`. Tale scelta è stata adottata per evitare problematiche nella gestione della variabile al di fuori del ciclo, considerando l'assenza di uno scoping specifico per blocchi `if` e `for`, come già discusso. Successivamente, il metodo valida l'intervallo di iterazione associato all'operatore `in`. Solo se entrambe le condizioni risultano soddisfatte, l'espressione di appartenenza viene considerata corretta; in caso contrario, viene generato un errore semantico.

6. Validazione del Transpiler

Per verificare il corretto funzionamento del transpiler, sono stati definiti cinque casi di test, associati ai file `test_case_1`, `test_case_2`, `test_case_3`, `test_case_4` e `test_case_5`, che includono sia programmi corretti, sia programmi contenenti errori intenzionali.

L'obiettivo principale dei test è verificare che il transpiler traduca correttamente il codice Kotlin valido in codice Swift equivalente. Inoltre, si punta a valutare l'efficacia degli error listener nel rilevare e segnalare errori lessicali e sintattici, nonché la correttezza dell'analisi semantica nell'identificare e segnalare eventuali errori logici con messaggi chiari e dettagliati. L'output atteso per ciascun caso di test è un codice Swift che sia sintatticamente e semanticamente equivalente al programma Kotlin se privo di errori, o che segnali correttamente gli eventuali errori lessicali, sintattici o semantici, impedendo la generazione di codice non valido.

6.1 Casi di Test

Il file `test_case_1` definisce la classe `Counter`, che implementa operazioni come incremento, reset e verifica della parità del contatore. Questo test è stato progettato per valutare la capacità del transpiler di gestire correttamente funzionalità di base, come la traduzione di dichiarazioni di classi, funzioni, variabili, nonché di istruzioni iterative e condizionali. Il file `test_case_2`, invece, definisce la classe `Rectangle`, che modella un rettangolo geometrico, include attributi e metodi per calcolare l'area, il perimetro, verificare se il rettangolo è un quadrato e definisce metodi sovraccaricati. Questo test ha l'obiettivo di verificare la capacità del transpiler di gestire correttamente funzionalità avanzate, come la gestione dei parametri opzionali e il sovraccarico dei metodi. L'analisi semantica, applicata a entrambi i test, si concentra su vari aspetti fondamentali del codice. In primo luogo, viene eseguito il **type checking** per le dichiarazioni e assegnazioni delle variabili, garantendo che il tipo del valore assegnato sia compatibile con il tipo dichiarato. Inoltre, viene verificato il **type mismatch** per le chiamate a funzione: il transpiler assicura che i tipi degli argomenti passati alle funzioni siano compatibili con i tipi dichiarati nei parametri delle funzioni stesse. Ogni funzione è soggetta a un controllo sul tipo di ritorno: se dichiarata senza tipo di ritorno (`void`), viene verificato che non contenga istruzioni `return`; se, invece, è dichiarata con un tipo di ritorno, viene verificato che la funzione restituisca un valore conforme a tale tipo. Un altro controllo fondamentale riguarda la **dichiarazione e assegnazione delle variabili e delle funzioni prima del loro utilizzo**: viene verificato che ogni elemento sia dichiarato prima di essere referenziato nel codice e assegnato prima di essere utilizzato. Inoltre, viene verificata la **modifica delle variabili immutabili già assegnate**: le variabili immutabili non possono essere modificate una volta assegnato loro un valore. L'analisi semantica include anche un controllo sulla **ri-dichiarazione di funzioni già dichiarate**, verificando che la definizione di una funzione con la stessa firma non compaia più di una volta all'interno dello stesso scope. Infine, viene verificata la **validità delle condizioni nelle diramazioni** `if-else` e nei cicli `for`, assicurando che le espressioni condizionali siano di tipo `Boolean` per le condizioni di diramazione e di appartenenza a un `Range` per le condizioni di iterazione.

Il file `test_case_3` presenta una versione errata della classe `Counter`, con diversi errori semantici. Il primo errore rilevato è la **violazione dell'immutabilità** alla riga 13, in

cui si tenta di modificare una variabile dichiarata come immutabile (dichiarata con la parola chiave `val`) a cui è già stato assegnato un valore. Successivamente viene rilevato l'**uso scorretto di `return`** alla riga 17: la funzione `reset`, è stata dichiarata senza un tipo di ritorno esplicito, tuttavia include un'istruzione di ritorno: questo è semanticamente errato, in quanto è implicito che la funzione non debba restituire alcun valore. Il successivo errore semantico riguarda un **errore di tipo nell'operatore range `..`** alla riga 36, dove l'operatore range, che supporta solo tipi numerici, viene utilizzato con un tipo `Boolean`: questo rende errata la definizione dell'intervallo. Infine, l'analisi semantica rileva una **condizione `if` non valida** alla riga 42, dove la condizione utilizza un tipo `Int` invece di un `Boolean`, che è il tipo richiesto per le espressioni condizionali negli `if`.

Il file `test_case_4` presenta una versione errata della classe `Rectangle`, con diversi errori semantici. In primo luogo, viene rilevata un'**assegnazione di tipo errato** alla riga 9, dove si tenta di assegnare una stringa a una variabile dichiarata come `Int`. Successivamente, l'analisi semantica evidenzia la **mancanza dell'istruzione `return` in una funzione con tipo di ritorno** alla riga 19: la funzione `calculateArea`, dichiarata come `Int`, non contiene alcuna istruzione di ritorno. Un altro errore semantico riguarda l'**uso di variabili non dichiarate** alla riga 18: la funzione `calculateAarea` tenta di utilizzare la variabile `depth` prima che questa venga definita o dichiarata. Un ulteriore problema riguarda la **doppia dichiarazione di funzioni** alla riga 28, dove la funzione `calculatePerimeter` viene ridefinita con la stessa firma, all'interno dello stesso scope. Infine, l'analisi semantica rileva un'**invocazione di una funzione non dichiarata** alla riga 35: la funzione `printDimensions` viene invocata nel metodo `main` prima della sua dichiarazione.

Infine, il file `test_case_5` si concentra sull'analisi di errori lessicali e sintattici in un programma semplice. In particolare, viene individuato un **errore lessicale** alla riga 6, causato dall'introduzione di token non riconosciuti, come `&=`, che non corrispondono a nessun simbolo valido definito nella grammatica del linguaggio. Inoltre, viene rilevato un **errore sintattico** alla riga 9, relativo a una dichiarazione di variabile errata (`val y : Int 5`), dove manca il simbolo `=` tra il tipo e il valore assegnato alla variabile.

I test effettuati confermano che il transpiler è in grado di tradurre correttamente il codice Kotlin valido in codice Swift equivalente, preservando la semantica e la logica del programma originale. Inoltre, il transpiler dimostra un'alta precisione nel rilevare errori a livello lessicale, sintattico e semantico, assicurando una gestione ottimale degli errori durante il processo di traduzione, con un approccio che previene la generazione di codice Swift non valido o errato, e fornisce messaggi di errore chiari, dettagliati e facilmente comprensibili.