Progetto di Informatica e Computazione

Armando Tacchella

Anno Accademico 2024/2025

L'obiettivo del progetto è sviluppare un interprete per il sottoinsieme del linguaggio Python la cui sintassi e semantica sono definite nel seguito. L'interprete dovrà leggere un programma contenuto in un unico file di testo sorgente, valutarne la correttezza sintatttica ed, eventualmente, eseguirlo. Programmi sintatticamente errati dovranno essere scartati con opportuni messaggi di errore, mentre i programmi sintatticamente corretti saranno eseguiti. Eventuali errori di esecuzione (per esempio, divisione per zero oppure accesso a liste con indici non validi) dovranno essere identificati e segnalati, come da specifiche date nel seguito.

1 Sintassi del linguaggio

Nel seguito diamo la specifica della struttura lessicale e sintattica del linguaggio che si intende trattare in input. Definiamo innanzitutto i seguenti simboli terminali (token) tramite espressioni regolari:

$$\begin{aligned} \textbf{num} &\rightarrow \text{[1-9] [0-9]}^* \\ \textbf{id} &\rightarrow \text{[a-z, A-Z] [0-9, a-z, A-Z]}^* \\ \textbf{newline} &\rightarrow \backslash\text{n} \\ \textbf{endmarker} &\rightarrow \text{EOF} \end{aligned}$$

in cui [a_1 - a_n] è un'abbreviazione per $a_1 \mid \ldots \mid a_n$, e $[a, b]$ è un'abbreviazione per $a \mid b$. Per "EOF" si intende la fine del file in input e per "\n" si intende il ritorno a capo. Gli altri token del linguaggio sono i segni di interpunzione, gli operatori aritmetici, relazionali e booleani e le seguenti parole chiave (riservate): **if**, **elif**, **else**, **while**, **continue**, **break**, **list**, **True** e **False**. Vi sono poi due token speciali **indent** e **dedent** che sono generati utilizzando i livelli di indentazione delle linee consecutive presenti nel file di input.[1] In particolare:

- Si utilizza uno stack di interi S in cui, prima di iniziare la lettura del file, viene inserito uno zero; questo dato non verrà mai estratto.
- I numeri inseriti in S saranno sempre in ordine strettamente crescente dal fondo alla cima.

[1] Tratto da `https://docs.python.org/3/reference/lexical_analysis.html#indentation`

- All'inizio di ogni linea del codice sorgente, il livello di indentazione della linea n_t (numero di caratteri di tabulazione presenti inizialmente) viene confrontato con la cima dello stack $top(S)$:
  - se $n_t = top(S)$, non viene generato nessun token;
  - se $n_t > top(S)$, allora n_t viene inserito nello stack e viene generato un token **indent**;
  - se $n_t < top(S)$, allora *deve* essere vero che $n_t \in S$ (altrimenti c'è un errore di tabulazione inconsistente) e tutti i numeri più grandi di n_t vengono estratti dallo stack generando un token **dedent** per ciascuno.
- al termine del file (prima di generare il token **endmarker**), viene generato un token **dedent** per ogni $x \in S$ con $x > 0$.

La struttura del programma[2] è delineata dal seguente insieme di produzioni dove, per convenzione, indichiamo i simboli non terminali tra parentesi acute e i simboli terminali in testo normale (in grassetto):

$$
\begin{aligned}
&< \text{program} > \rightarrow < \text{stmts} > \textbf{ endmarker} \\
&< \text{stmts} > \rightarrow < \text{stmt} > < \text{stmts} > \mid \epsilon \\
&< \text{stmt} > \rightarrow < \text{compound\_stmt} > \mid < \text{simple\_stmt} >
\end{aligned}
$$

La struttura delle istruzioni (< simple_stmt >) è delineata nel seguente insieme di produzioni:

$$
\begin{aligned}
< \text{simple\_stmt} > \rightarrow\ & < \text{loc} > \textbf{ = } < \text{expr} > \textbf{ newline} \\
\mid\ & < \text{id} > \textbf{ = list() newline} \\
\mid\ & < \text{id} > \textbf{ . append (} < \text{expr} > \textbf{) newline} \\
\mid\ & \textbf{break newline} \\
\mid\ & \textbf{continue newline} \\
\mid\ & \textbf{print (} < \text{expr} > \textbf{) newline}
\end{aligned}
$$

La struttura delle istruzioni (< compund_stmt >) è delineata nel seguente insieme di produzioni:

$$
\begin{aligned}
< \text{compound\_stmt} > \rightarrow\ & < \text{if\_stmt} > \mid < \text{while\_stmt} > \\
< \text{if\_stmt} > \rightarrow\ & \textbf{if } < \text{expr} > \textbf{ : } < \text{block} > \\
\mid\ & \textbf{if } < \text{expr} > \textbf{ : } < \text{block} > < \text{elif\_block} > \\
\mid\ & \textbf{if } < \text{expr} > \textbf{ : } < \text{block} > < \text{else\_block} > \\
< \text{elif\_block} > \rightarrow\ & \textbf{elif } < \text{expr} > \textbf{ : } < \text{block} > \\
\mid\ & \textbf{elif } < \text{expr} > \textbf{ : } < \text{block} > < \text{elif\_block} > \\
\mid\ & \textbf{elif } < \text{expr} > \textbf{ : } < \text{block} > < \text{else\_block} > \\
< \text{else\_block} > \rightarrow\ & \textbf{else : } < \text{block} > \\
< \text{while\_stmt} > \rightarrow\ & \textbf{while } < \text{expr} > \textbf{ : } < \text{block} > \\
< \text{block} > \rightarrow\ & \textbf{newline indent } < \text{stmts} > \textbf{ dedent}
\end{aligned}
$$

Infine, la struttura delle espressioni, inclusi gli aspetti di precedenza e associatività, è gestita con le seguenti produzioni:

[2] Tratta da: `https://docs.python.org/3/reference/grammar.html`

$$
\begin{aligned}
<\text{expr}> &\rightarrow <\text{expr}> \textbf{ or } <\text{join}> \mid <\text{join}> \\
<\text{join}> &\rightarrow <\text{join}> \textbf{ and } <\text{equality}> \mid <\text{equality}> \\
<\text{equality}> &\rightarrow <\text{equality}> \textbf{ == } <\text{rel}> \mid <\text{equality}> \textbf{ != } <\text{rel}> \mid <\text{rel}> \\
<\text{rel}> &\rightarrow <\text{numexpr}> \textbf{ < } <\text{numexpr}> \mid <\text{numexpr}> \textbf{ <= } <\text{numexpr}> \\
&\quad \mid <\text{numexpr}> \textbf{ >= } <\text{numexpr}> \mid <\text{numexpr}> \textbf{ > } <\text{numexpr}> \mid <\text{numexpr}> \\
<\text{numexpr}> &\rightarrow <\text{numexpr}> \textbf{ + } <\text{term}> \mid <\text{numexpr}> \textbf{ - } <\text{term}> \mid <\text{term}> \\
<\text{term}> &\rightarrow <\text{term}> \textbf{ * } <\text{unary}> \mid <\text{term}> \textbf{ // } <\text{unary}> \mid <\text{unary}> \\
<\text{unary}> &\rightarrow \textbf{not } <\text{unary}> \mid \textbf{ - } <\text{unary}> \mid <\text{factor}> \\
<\text{factor}> &\rightarrow \textbf{ (} <\text{expr}> \textbf{)} \mid <\text{loc}> \mid \textbf{num} \mid \textbf{True} \mid \textbf{False} \\
<\text{loc}> &\rightarrow \textbf{id} \mid \textbf{id [} <\text{expr}> \textbf{]}
\end{aligned}
$$

2 Semantica del linguaggio

Definiamo (in modo informale) la semantica dei vari costrutti introdotti nella sezione precedente. Nel seguito, quando si parla di *numero intero* (tipo di dato intero), si intende interi a 64 bit con segno, mentre nel caso di un *booleano* (tipo di dato booleano), ci si riferisce a un tipo con soli due valori di verità logica, "vero" e "falso". Consideriamo dapprima i $<$simple_stmt$>$ che riguardano le variabili:

- La valutazione di $<$loc$>$ **=** $<$expr$>$ ricade in uno fra i casi seguenti:

  1. se $<$loc$>$ è un **id** (identificativo di variabile), viene (ri)allocato uno spazio in memoria identificato dall'etichetta **id**; lo spazio deve essere dimensionato per contenere un dato corrispondente a quello del tipo dell'espressione $<$expr$>$, ossia un intero oppure un booleano;
  2. se $<$loc$>$ è un **id [** $<$expr$>$ **]**, ossia l'accesso (in scrittura) alla posizione di una lista **id** identificata dall'indice i ottenuto valutando l'espressione $<$ expr $>$ (che deve essere di tipo intero e di valore positivo), si suppone che sia stata dichiarata in precedenza una lista **id** che contiene abbastanza elementi (almeno $i + 1$) in modo da poter memorizzarne uno di indice i.

- La valutazione di **id = list ( )** richiede la creazione di uno spazio in memora per l'allocazione di un vettore dinamico di valori interi o booleani (non sono ammesse liste di liste).

- La valutazione di **id . append (** $<$ expr $>$ **)**" comporta l'aggiunta in fondo alla lista **id** il risultato della valutazione dell'espressione $<$ expr $>$ (che non può essere una lista).

Per le variabili (sia scalari, sia liste) esiste un unico spazio di memoria pertanto una variabile, ovunque venga definita, è sempre disponibile a valle della sua definizione. Il tipo di una variabile può essere ridefinito dinamicamente, ossia se una variabile **id** compare in due istruzioni del tipo **id** = $<$expr$>$ e il tipo dell'espressione è diverso, la variabile avrà il tipo dell'ultima espressione in ordine di esecuzione. Analogamente, il tipo degli elementi di una lista è dinamico e pertanto una lista può contenere valori

eterogenei (interi e booleani) in base al tipo delle espressioni che sono state usate per aggiungere gli elementi o modificarne il valore.

Gli ulteriori casi del simbolo < simple_stmt > sono i seguenti:

- Nel caso di **break**, viene interrotta la valutazione dello statement **while** all'interno del quale si trova l'istruzione.
- Nel caso di **continue**, vengono saltate tutte le istruzioni che sono nel corpo del **while** e che seguono l'istruzione, e il ciclo ricomincia dall'inizio.
- Nel caso di **print (** < expr > **)**, viene valutata l'espressione < expr > e viene visualizzato il risultato in console.

Le istruzioni **break** e **continue** modificano solo il ciclo in cui si trovano. Nel caso di cicli annidati, se le istruzioni appaiono nel ciclo più interno, viene modificato solo il comportamento di quest'ultimo. Queste istruzioni non hanno alcun effetto se si trovano fuori da un ciclo.

Per quanto riguarda il simbolo <compound_stmt> abbiamo i seguenti casi:

- Nel caso dell'istruzione condizionale **if** <expr> **:** <block>, eventualmente seguita da <elif_block> o <else_block>, viene valutata l'espressione <expr>; se la valutazione restituisce vero viene valutato il simbolo < block >; altrimenti vengono valutati i blocchi < elif_block > o < else_block >, se presenti. Il vincolo è che il tipo del risultato della valutazione del simbolo < expr > sia booleano.
- Nel caso del blocco **elif** < expr > **:** < block >, eventualmente seguito da < elif_block > o < else_block >, si procede esattamente come per l'istruzione **if**, con le stesse condizioni.
- Nel caso del blocco **else :** <block>, viene valutato il simbolo <block>.
- Nel caso dell'istruzione **while** < expr > **:** < block >, viene ripetuta la valutazione del simbolo < block > fintanto che la valutazione del simbolo < expr > restituisce vero; pertanto la valutazione del simbolo < block > avviene zero o più volte. Il vincolo è che lil risultato della valutazione del simbolo <expr> sia di tipo booleano.

La valutazione del simbolo <expr> comporta l'attribuzione di un *valore* e un *tipo*. Il tipo può essere intero, booleano o non definito (e in questo caso si genera un errore di valutazione). A seconda dell'espressione, il tipo è definito ricorsivamente come segue:

- Il tipo di **id** è lo stesso della variabile **id**.
- Il tipo di **id[**< expr >**]** è lo stesso dell'elemento della lista a cui si riferisce l'indice ottenuto dalla valutazione di <expr> se è un indice lecito, altrimenti è non determinato.
- < factor > ha lo stesso tipo dell'elemento che lo definisce, quindi il tipo di <loc>, il tipo di <expr>, intero nel caso di **num**, oppure booleano nel caso di **True** e **False**.

- il tipo di $<$ unary $>$ è booleano nella forma **not** $<$ unary $>$ se l'espressione che segue il **not** ha tipo booleano, altrimenti è indeterminato; il tipo è intero nella forma **-** $<$ unary $>$ se l'espressione che segue il meno ha tipo intero, altrimenti è indeterminato; infine, nella forma $<$ factor $>$, il tipo è lo stesso dell'espressione $<$ factor $>$.

- il tipo di $<$ term $>$ è intero se le espressioni $<$ term $>$ e $<$ unary $>$ che lo compongono hanno tipo intero, altrimenti è indeterminato; nella forma $<$ unary $>$, il tipo è lo stesso di $<$ unary $>$.

- per il tipo di $<$ numexpr $>$ vale lo stesso che per $<$ term $>$.

- nel caso $<$ rel $>$ ::= $<$ numexpr $>$, il tipo di $<$ rel $>$ è lo stesso di $<$ numexpr $>$; negli altri casi è booleano a patto che le espressioni che lo compongono abbiano tutte tipo intero, altrimenti è indeterminato.

- nel caso $<$ equality $>$ ::= $<$ rel $>$, il tipo di $<$ equality $>$ è lo stesso di $<$ rel $>$; negli altri casi è booleano a patto che le espressioni confrontate abbiano lo stesso tipo, altrimenti è indeterminato.

- analogamente, il tipo di $<$ join $>$ e $<$ expr $>$ è univocamente determinato da $<$ equality $>$ o $<$ join $>$, rispettivamente, oppure è booleano a patto che tutte le espressioni che lo compongono siano booleane.

Il calcolo del valore dell'espressione segue la semantica usuale per gli operatori aritmetici, relazionali e booleani. In particolare, per gli operatori booleani si utilizza la regola del "corto circuito" per cui se l'operatore è **and** e il primo operando valuta a falso, non viene valutato il secondo operando (e il valore dell'espressione è falso), mentre se l'operatore è **or** e il primo operando valuta a vero, non viene valutato il secondo operando (e il valore dell'espressione è vero). In assenza di errori di tipo, è considerato un errore da intercettare la divisione per zero, mentre eventuali overflow o underflow possono essere ignorati.

3 Specifiche di consegna e valutazione

Valgono i seguenti vincoli aggiuntivi rispetto alla consegna;

- L'interprete dovrà essere consegnato come codice sorgente **limitatamente** all'insieme di file .h e file .cpp che lo compongono in un'unica cartella zippata (consegna su Aulaweb).

- Il codice sorgente dovrà essere compatibile con il C++ standard ISO 2020. Non è possibile utilizzare altre librerie se non quelle standard del C++. Il progetto deve essere "stand-alone", ossia non dipendere da altri strumenti software per la sua corretta compilazione ed esecuzione.

- L'interprete dovrà leggere il nome del programma da compilare da linea di comando. Ad esempio, se l'interprete è stato compilato come `interpreter.exe`

sotto un sistema Windows, deve essere possibile invocarlo da prompt dei comandi come segue:

```
interpreter.exe fileSorgente.txt
```

dove `fileSorgente.txt` è un file di testo contenente un programma da interpretare.

- L'output del programma interpretato dovrà essere reso in console; in caso di errori di compilazione o errori di esecuzione, l'interprete dovrà fermarsi al primo di tali errori e segnalarlo in console con un messaggio **di una sola linea** avente come formato:

  ```
  Error: <descrizione errore>
  ```

  Il contenuto di $<$ descrizione errore $>$ non è ulteriormente specificato, ma dovrebbe fornire qualche indicazione utile a individuare l'errore nel codice sorgente.

- La compilazione e il test dell'interprete avverranno in ambiente WSL/Linux; per compilare il programma verrà utilizzato il comando:

  ```
  g++ -std=c++20 *.cpp -o interpreter
  ```

  Il programma verrà poi lanciato su tutti i test (sia quelli forniti, sia quelli non forniti) e il suo output confrontato con quello previsto in modo automatico con il comando `diff` o analogo. Pertanto, il suggerimento è testare la compilazione e l'esecuzione del programma nello stesso tipo di ambiente e accertarsi che non vi siano elementi che pregiudichino la compilazione (ad esempio, attenzione ai nomi dei file .h e i relativi include visto che il file system Windows non è case-sensitive mentre Linux sì) e che l'output sia esattamente quello previsto da specifica utilizzando un confronto automatico e non limitandosi a guardare a mano la corrispondenza dell'output.

- Il codice sorgente verrà esaminato per controllarne l'organizzazione e la presenza di commenti. I sorgenti verranno confrontati tra di loro per individuare (i) eccessive similarità tra elaborati diversi e (ii) probabile ricorso a strumenti di IA. In tutti i casi "sospetti" il docente si riserva di richiedere una verifica orale sul progetto per confermare il voto.

La consegna verrà valutata da 0 a 10 punti, sulla base della seguente griglia:

1. L'interprete compila e accetta input da linea di comando (0-2 punti).
2. L'esecuzione sui test forniti è corretta (0-4 punti).
3. L'esecuzione su test aggiuntivi (non forniti) è corretta (0-3 punti).
4. Il codice sorgente dell'interprete è organizzato e commentato adeguatamente (0-1 punto).

5. Viene fornito il progetto UML dell'interprete (0-1 punto).

I test che verranno forniti avranno sia il codice sorgente di input, sia l'output atteso. Se l'interprete va in crash su un programma in caso di errori lessicali, semantici o di valutazione, questo viene considerato come un test non passato. Se il programma non compila o non accetta input da linea di comando, il punteggio totale sarà comunque 0 (non verranno valutati i punti 2-4). Se il programma non passa nessuno dei test di cui al punto 2, il punteggio previsto è 2 (non verranno valutati i punti 3-4). Se il programma non dovesse passare nessuno dei test di cui al punto 3, verrà comunque valutato il punto 4. Si noti che la somma totale dei punti è 11 in modo da compensare eventuali perdite sui punti 2 e 3 oppure assegnare la lode in caso che si arrivi a 11 di progetto e il punteggio della prova scritta sia pari a 20.