Progetto di Informatica e Computazione

Armando Tacchella

Anno Accademico 2023/2024

L'obiettivo del progetto è sviluppare un interprete per il sottoinsieme del linguaggio C 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 sintattica 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 vettori 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 tramite espressioni regolari:

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

in cui [a_1 - a_n] è un'abbreviazione per $a_1 \mid \ldots \mid a_n$, e $[a, b]$ è un'abbreviazione per $a \mid b$. Gli altri simboli terminali del linguaggio sono i segni di interpunzione, gli operatori aritmetici, relazionali e booleani e le seguenti parole chiave (riservate): if, else, while, do, break, int, boolean, true e false. La struttura del programma 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:

$$\begin{aligned} \text{<program>} &\rightarrow \text{<block>} \\ \text{<block>} &\rightarrow \{\ \text{<decls>}\ \text{<stmts>}\ \} \\ \text{<decls>} &\rightarrow \text{<decl>}\ \text{<decls>} \mid \epsilon \\ \text{<decl>} &\rightarrow \text{<type>}\ \text{id}\ ; \\ \text{<type>} &\rightarrow \text{<type>}\ [\ \text{num}\ ] \mid \text{<basic>} \\ \text{<basic>} &\rightarrow \text{int} \mid \text{boolean} \\ \text{<stmts>} &\rightarrow \text{<stmt>}\ \text{<stmts>} \mid \epsilon \end{aligned}$$

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

$$
\begin{aligned}
<\text{stmt}> \rightarrow\ & <\text{loc}> = <\text{bool}> ; \\
\mid\ & \text{if } (<\text{bool}>) <\text{stmt}> \\
\mid\ & \text{if } (<\text{bool}>) <\text{stmt}> \text{ else } <\text{stmt}> \\
\mid\ & \text{while } (<\text{bool}>) <\text{stmt}> \\
\mid\ & \text{do } <\text{stmt}> \text{ while } (<\text{bool}>) ; \\
\mid\ & \text{break} ; \\
\mid\ & \text{print}(<\text{bool}>) ; \\
\mid\ & <\text{block}> \\
<\text{loc}> \rightarrow\ & <\text{loc}> [<\text{bool}>] \mid \text{id}
\end{aligned}
$$

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

$$
\begin{aligned}
<\text{bool}> &\rightarrow <\text{bool}> \,||\, <\text{join}> \mid <\text{join}> \\
<\text{join}> &\rightarrow <\text{join}> \,\&\&\, <\text{equality}> \mid <\text{equality}> \\
<\text{equality}> &\rightarrow <\text{equality}> == <\text{rel}> \mid <\text{equality}> \,!= <\text{rel}> \mid <\text{rel}> \\
<\text{rel}> &\rightarrow <\text{expr}> < <\text{expr}> \mid <\text{expr}> <= <\text{expr}> \\
&\quad \mid <\text{expr}> >= <\text{expr}> \mid <\text{expr}> > <\text{expr}> \mid <\text{expr}> \\
<\text{expr}> &\rightarrow <\text{expr}> + <\text{term}> \mid <\text{expr}> - <\text{term}> \mid <\text{term}> \\
<\text{term}> &\rightarrow <\text{term}> * <\text{unary}> \mid <\text{term}> / <\text{unary}> \mid <\text{unary}> \\
<\text{unary}> &\rightarrow ! <\text{unary}> \mid - <\text{unary}> \mid <\text{factor}> \\
<\text{factor}> &\rightarrow (<\text{bool}>) \mid <\text{loc}> \mid \text{num} \mid \text{true} \mid \text{false}
\end{aligned}
$$

2 Semantica del linguaggio

Definiamo (in modo informale) la semantica dei vari costrutti introdotti nella sezione precedente:

- In un $<$ block $>$ viene valutato prima il simbolo $<$ decls $>$ e poi il simbolo $<\text{stmts}>$.
- La valutazione di un simbolo $<\text{decls}>$ comporta la valutazione di ogni simbolo $<\text{decl}>$ nell'ordine in cui compare; analogamente per il simbolo $<\text{stmts}>$.
- La valutazione di un simbolo $<$ decl $>$ comporta la creazione di uno spazio in memoria identificato dall'etichetta "id"; lo spazio deve essere dimensionato per contenere un dato di tipo "int", "boolean", oppure un array di dimensione "num" di dati "int" o "boolean"; "int" e "boolean" sono interpretati come interi (con segno) e booleani rispettivamente; esiste un unico spazio di memoria pertanto una variabile, ovunque venga dichiarata, è sempre disponibile a valle della sua dichiarazione soggetta al vincolo che uno stesso "id" non può comparire in due dichiarazioni diverse.

Per un simbolo $<\text{stmt}>$ vi sono i seguenti casi:

- $<$ **loc** $>$ **=** $<$ **bool** $>$: il risultato della valutazione del simbolo $<$ bool $>$ viene assegnato allo spazio di memoria identificato da $<$ loc $>$ soggetto ai seguenti vincoli:

  - $<$ loc $>$ è stato oggetto di una dichiarazione che precede l'istruzione in esame;
  - il tipo associato a $<$ loc $>$ in fase di dichiarazione è lo stesso del risultato ottenuto valutando $<$ bool $>$.
  - nel caso in cui $<$ loc $>$ sia una cella di un vettore, è necessario verificare che si tratti di una cella valida;

- **if (**$<$ **bool** $>$**)** $<$ **stmt** $>$: viene valutato il simbolo $<$ bool $>$; se la valutazione restituisce vero viene valutato il simbolo $<$ stmt $>$; il vincolo è che il tipo del risultato della valutazione del simbolo $<$ bool $>$ sia lo stesso di "boolean".

- **if (**$<$ **bool** $>$**)** $<$ **stmt** $>^1$ **else** $<$ **stmt** $>^2$: viene valutato il simbolo $<$ bool $>$; viene valutato il simbolo $<$ stmt $>^1$ se la valutazione restituisce vero, mentre viene valutato il simbolo $<$ stmt $>^2$ in caso contrario; il vincolo è che il tipo del risultato della valutazione del simbolo $<$ bool $>$ sia lo stesso di "boolean".

- **while (**$<$ **bool** $>$**)** $<$ **stmt** $>$: viene ripetuta la valutazione del simbolo $<$ stmt $>$ fintanto che la valutazione del simbolo $<$ bool $>$ restituisce vero; pertanto la valutazione del simbolo $<$ stmt $>$ avviene zero o più volte; il vincolo è che il tipo del risultato della valutazione del simbolo $<$ bool $>$ sia lo stesso di "boolean".

- **do** $<$ **stmt** $>$ **while (**$<$ **bool** $>$**);**: viene effettuata la valutazione del simbolo $<$ stmt $>$ e poi viene ripetuta fintanto che la valutazione del simbolo $<$ bool $>$ restituisce vero; pertanto la valutazione del simbolo $<$ stmt $>$ avviene una o più volte; il vincolo è che il tipo del risultato della valutazione del simbolo $<$ bool $>$ sia lo stesso di "boolean".

- **print(**$<$ **bool** $>$**)**: viene valutata l'espressione $<$ **bool** $>$ e viene visualizzato il risultato in console.

- **break**: viene interrotta la valutazione dello statement "while" o "do-while" all'interno del quale si trova il break;

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.

- Il codice sorgente dovrà essere compatibile con il C++ standard ISO 2017; l'utilizzo di standard **precedenti** è consentito, mentre non è consentito l'utilizzo di standard **successivi**.

- 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 Linux; pertanto, il suggerimento è testare la compilazione e l'esecuzione del programma in un ambiente Linux oppure accertarsi che non vi siano elementi che pregiudichino la compilazione in tale ambiente (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)

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; viene fornito il progetto UML dell'interprete (0-2 punti)

I test che verranno forniti avranno sia il codice sorgente di input, sia l'output atteso. 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.

4 Architettura del codice (suggerimenti)

In figura 1 riportiamo il diagramma a classi relativo all'organizzazione dei costrutti del linguaggio sorgente. Si tratta delle classi che vengono utilizzate per costruire l'albero sintattico astratto (AST) corrispondente al codice sorgente. Alla base della gerarchia

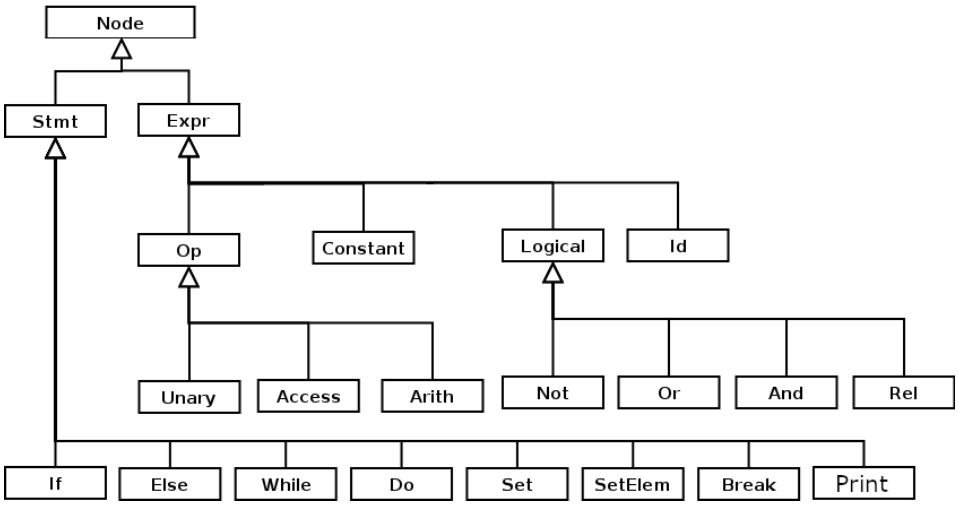

Figura 1: Classi corrispondenti ai costrutti del linguaggio sorgente.

di figura 1 troviamo la classe Node con due sottoclassi principali: Stmt per la gestione delle istruzioni e Expr per la gestione delle espressioni.

In particolare, per quanto riguarda la gestione delle espressioni, vediamo che la classe Expr ha quattro sottoclassi dirette. La classe Op viene utilizzata per gestire gli operatori che non restituiscono un valore di verità, quindi operatori aritmetici con uno (Unary) o due operandi (Arith) e l'operatore "[ ]" di indicizzazione dei vettori (Access). La classe Constant corrisponde alle costanti, ossia numeri interi e costanti booleane `true` e `false`. La classe Logical fornisce il supporto per gli operatori logici (Not, Or, And) e per quelli relazionali (Rel). La gestione separata degli operatori logici e relazionali rispetto a quelli aritmetici è opzionale. Infine, l'ultima sottoclasse diretta di Expr collegata alla struttura del codice sorgente è Id per la gestione degli identificatori, ossia nomi di variabili e vettori.

Per quanto riguarda la gestione delle istruzioni, la classe Stmt ha otto sottoclassi. Le classi If e Else servono per la gestione delle istruzioni condizionali, mentre le classi While e Do, servono per la gestione dei costrutti di iterazione. Le classi Set e SetElem servono, rispettivamente, per supportare le istruzioni di assegnamento a variabile e a vettore. La classe Break gestisce le omonime istruzioni di interruzione – in sostanza, salti incondizionati. Infine, la classe Seq consente di costruire sequenze di istruzioni concatenando fra loro oggetti della superclasse. Complessivamente, la gerarchia mostrata in figura 1 costituisce un esempio di pattern "Interpreter" combinata con dei pattern "Composite", che rendono conto della composizione gerarchica dei costrutti del linguaggio sorgente.