Programmazione B
Ingegneria e Scienze Informatiche - Cesena
A.A. 2021-2022

La struttura dati PILA

Andrea Piroddi - andrea.piroddi@unibo.it
Credit: Pietro Di Lena

Introduzione

- Una **pila** (o `stack`) è una **struttura dati astratta** le cui modalità di accesso sono di tipo **LIFO**.
  - LIFO (Last In First Out): i dati sono estratti in ordine inverso rispetto al loro inserimento.
  - FIFO (First In First Out): i dati sono estratti rispetto all'ordine di inserimento.
- Una struttura dati pila supporta essenzialmente due sole operazioni.
  1. **Push**. Inserisce un oggetto in cima alla pila.
  2. **Pop**. Rimuove l'oggetto in cima alla pila e ne ritorna il valore.
- Nonostante la sua semplicità, la struttura dati astratta pila è utilizzata in numerosi contesti informatici.
  - Ad esempio, quasi tutti i linguaggi di alto livello utilizzano una pila di record di attivazione per gestire la chiamata a funzione.
- Una struttura dati di tipo pila può essere vista come una struttura dati di tipo lista che supporta un numero limitato di operazioni.
  - Le operazioni `Push` e `Pop` sono equivalenti a inserimento e rimozione in testa (o, equivalentemente, in coda) della lista.

Implementazione del tipo di dato astratto pila in C

- Vediamo alcuni approcci per implementare una struttura dati di tipo pila in C.
  - Implementazione di pila con capacità *illimitata*. Utilizziamo una libreria che implementa una struttura dati di tipo lista (potremmo implementare direttamente le funzioni richieste).
  - Implementazione efficiente di pila con capacità *limitata*. Mostriamo una implementazione ad-hoc.
- Anche in questo caso ci concentriamo unicamente su implementazioni di pile che ci permettano di memorizzare interi (tipo di dato `int`).
- Scelte progettuali generali (le stesse adottate per la struttura dati lista).
  - Vogliamo che il nostro codice sia **efficiente**: dobbiamo fornire l'implementazione più efficiente in termini di **utilizzo di memoria** e **tempo di calcolo** (queste caratteristiche dipendono da come abbiamo definito il tipo di dato lista).
  - Vogliamo che il nostro codice sia **modulare**: le funzioni devono essere **semplici** (poche righe di codice), **specifiche** (un solo compito) e **riutilizzabili** (vogliamo poter riutilizzare una funzione in tutti i contesti in cui sia possibile).

Implementazioni delle operazioni sulla struttura dati pila di interi

- Consideriamo i seguenti prototipi, che definiscono le operazioni richieste su una pila.
- Aggiungiamo altre funzioni utili oltre alle due essenziali `push()` e `pop()`.
- I prototipi prendono in input un puntatore al tipo di dato `stack` (da definire).

```
1  stack init(void);
2
3  void clear(stack *S);
4
5  int isempty(stack *S);
6
7  int push(stack *S, int elem);
8
9  int pop(stack *S, int *elem);
10
11 int top(stack *S, int *elem);
```

Operazioni sulla struttura dati pila: creazione, distruzione e test

- Le seguenti funzioni ci permettono di:
  - *creare* un oggetto di tipo pila. Inizialmente la pila non contiene elementi,
  - *svuotare* l'intero contenuto di un oggetto di tipo pila.
  - *testare* se la pila è vuota.

```
1  /*
2   * Crea e ritorna una pila vuota.
3   */
4  stack init(void);
5
6  /*
7   * Elimina tutti gli elementi contenuti nella pila
8   * puntata da S.
9   */
10 void  clear(stack *S);
11
12 /*
13  * Ritorna 0 se la pila Ã¨ vuota,
14  * un intero diverso da zero altrimenti.
15  */
16 int isempty(stack *S);
```

Operazioni sulla struttura dati pila: `push()` e `pop()`

- Le seguenti funzioni permettono di inserire e rimuovere un elemento in cima alla lista.
- La funzione `pop()` restituisce il valore dell'elemento in cima alla pila oltre a rimuoverlo.
- La funzione `top()` ci permette solo di recuperare il valore dell'elemento in cima.

```
1  /*
2   * Inserisce elem in cima alla pila S.
3   *
4   * Ritorna 0 se l'inserimento ha successo,
5   * un intero diverso da zero altrimenti.
6   */
7  int push(stack *S, int elem);
8  /*
9   * Rimuove l'elemento in testa alla pila S e ne salva
10  * il valore in elem.
11  *
12  * Ritorna 0 se la rimozione ha successo,
13  * un intero diverso da zero altrimenti.
14  */
15 int pop(stack *S, int *elem);
16 /*
17  * Salva in elem il valore in cima alla pila S.
18  *
19  * Ritorna 0 se l'operazione ha successo,
20  * un intero diverso da zero altrimenti.
21  */
22 int top(stack *S, int *elem);
```

Il tipo di dato *pila illimitata*

- Una struttura dati di tipo pila con capienza illimitata può essere implementata tramite una lista concatenata.
- Utilizziamo una libreria che implementa liste concatenate per implementare una struttura dati di tipo pila.
  - Le operazioni di inserimento e rimozione in testa sono computazionalmente efficienti per tutte le rappresentazioni concatenate viste (lista concatenata semplice, doppiamente concatenata e circolare).
  - Le operazioni in cima alla pila possono essere implementate efficientemente come operazioni in testa ad una lista.
- Il tipo di dato pila (`stack`) è equivalente al tipo di dato lista (`list`).
- E' necessario includere il file di header in cui è definito il tipo di dato `list` per poter definire il tipo di dato `stack`.

```
1 #include "list.h"
2
3 typedef list stack;
```

- Utilizziamo unicamente le funzioni della libreria `list.h` per poter implementare le funzioni sulla struttura dati pila: possiamo completamente tralasciare i dettagli implementativi della libreria `list.h`.
- Quale rappresentazione è maggiormente conveniente tra liste concatenate semplici, doppiamente concatenate e circolari?

Creazione, distruzione e test su pila vuota: `init()`, `clear()` e `isempty()`

- Le seguenti funzioni permettono di creare una pila, di svuotare completamente la pila e di verificare se una pila è vuota.
- L'implementazione richiama semplicemente le corrispondenti implementazioni su liste concatenate.
- Da notare che abbiamo definito il tipo di dato `stack` come sinonimo del tipo di dato `list`: il tipo di ritorno della funzione `list_create()` è quindi perfettamente equivalente al tipo di ritorno della funzione `init()`.

```
1 stack init(void) {
2   return list_create();
3 }
```

```
1 void clear(stack *S) {
2   list_delete(S);
3 }
```

```
1 int isempty(stack *S) {
2   return is_empty(S);
3 }
```

Inserimento, rimozione e selezione di dati nella pila: `push()`,`pop()`,`top()`

- Le funzioni di inserimento e rimozione in cima alla pila sono implementate come funzioni di inserimento e rimozione in testa ad una lista concatenata.

```
1 int push(stack *S, int elem) {
2   return head_insert(S,elem);
3 }
```

```
1 int pop(stack *S, int *elem) {
2   if(is_empty(S) || elem == NULL) {
3     return 1;
4   } else {
5     *elem = head_select(S);
6     return  head_delete(S);
7   }
8 }
```

```
1 int top(stack *S, int *elem) {
2   if(is_empty(S) || elem == NULL) {
3     return 1;
4   } else {
5     *elem = head_select(S);
6     return 0;
7   }
8 }
```

Caratteristiche della libreria per pile limitate

- Per alcuni problemi algoritmici che fanno uso di una struttura dati di tipo pila il numero massimo di oggetti da memorizzare in una pila può essere determinato a priori.
  - Implementazione efficiente tramite rappresentazione della pila con array.
- Caratteristiche specifiche della libreria per pile con capienza limitata:
  - La funzione `init()` richiede come argomento la dimensione della pila.
  - Aggiungiamo una nuova funzione di libreria per determinare se la pila è piena.

```
1  typedef struct stack {
2    unsigned int size;
3    unsigned int last;
4    int *data;
5  } stack;
6
7  stack init(unsigned int n);
8
9  void clear(stack *S);
10
11 int isempty(stack *S);
12
13 int isfull(stack *S);
14
15 int push(stack *S, int elem);
16
17 int pop(stack *S, int *elem);
18
19 int top(stack *S, int *elem);
```

Creazione e distruzione do una pila: `init()` e `clear()`

- La funzione di creazione di una pila (`init()`) è equivalente alla funzione di creazione di una lista rappresentata con array. A differenza del tipo di dato lista, in questo caso la dimensione della pila non verrà modificata a run-time.
- La funzione di distruzione (`clear()`) si occupa di deallocare l'array dinamico: la pila non può essere ulteriormente utilizzata, deve essere riallocata.

```
1 stack init(unsigned int n) {
2   stack tmp = {0,0,NULL};
3
4   tmp.data = (int *)malloc(n*sizeof(int));
5   if(tmp.data != NULL)
6     tmp.size = n;
7   return tmp;
8 }
```

```
1 void clear(stack *S) {
2   if(S != NULL) {
3     free(S->data);
4     S->data = NULL;
5     S->last = 0;
6     S->size = 0;
7   }
8 }
```

Funzioni di test sul numero di elementi nella lista: `isempty()` e `isfull()`

- Le funzioni di test verificano che la pila sia piena o vuota. Ritornano un valore booleano.
- Da notare che se la pila viene deallocata utilizzando la funzione `clear()`, chiamate a `is_empty()` e `is_full()` su tale struttura ritornano *true*: una pila deallocata ha capacità nulla (è quindi sempre piena) e non contiene elementi.

```
1 int isempty(stack *S) {
2   return S == NULL || S->last == 0;
3 }
```

```
1 int isfull(stack *S) {
2   return S == NULL || S->last == S->size;
3 }
```

Inserimento, rimozione e selezione di dati nella pila: `push()`, `pop()`, `top()`

- Le funzioni di inserimento e rimozione in cima alla pila sono implementate come funzioni di inserimento e rimozione in coda ad un array.
- Da notare che l'unica differenza tra la funzione `top()` e la funzione `pop()` consiste nell'aggiornamento del campo `last` (riga 5): in `top()` non è necessario aggiornalo.
- Su pile deallocate queste funzioni non hanno effetto (ritornano 1).

```
1 int push(stack *S, int elem) {
2   if(isfull(S)) {
3     return 1;
4   } else {
5     S->data[S->last++] = elem;
6     return 0;
7   }
8 }
```

```
1 int pop(stack *S, int *elem) {
2   if(isempty(S) || elem == NULL) {
3     return 1;
4   } else {
5     *elem = S->data[S->last-- - 1];
6     return 0;
7   }
8 }
```

```
1 int top(stack *S, int *elem) {
2   if(isempty(S) || elem == NULL) {
3     return 1;
4   } else {
5     *elem = S->data[S->last - 1];
6     return 0;
7   }
8 }
```

Parentesi bilanciate

- Un problema algoritmico che può essere facilmente risolto facendo uso di una struttura dati pila è quello di verificare il **bilanciamento delle parentesi** in una stringa.
- Assumiamo che la stringa possa contenere altre informazioni oltre alle parentesi: ci interessa unicamente verificare che le parentesi siano bilanciate.
- La lunghezza della stringa è un upper bound al numero massimo di elementi che la pila dovrà contenere: possiamo quindi fare uso di una struttura dati pila con capienza limitata.
- Algoritmo per valutare il bilanciamento delle parentesi.
  - Scorriamo la stringa in input da sinistra verso destra.
  - Se il carattere corrente non è una parentesi (aperta o chiusa), ignoriamo il carattere.
  - Se il carattere corrente è una parentesi aperta, inseriamo nella pila tale carattere.
  - Se il carattere corrente è una parentesi chiusa, rimuoviamo la cima della pila e verifichiamo che sia una parentesi aperta di tipo compatibile con il carattere corrente. Se le due parentesi non sono compatibili, ritorniamo `false`.
  - Al termine della procedura se la pila è vuota ritorniamo `true`, altrimenti `false`.

Parentesi bilanciate: implementazione

```
1  /* Ritorna true se le parentesi in str sono bilanciate */
2  int balanced_parenthesis(char *str) {
3    unsigned int i,n = strlen(str);
4    stack S = init(n);
5    int c;
6    for(i=0; i<n; i++) {
7      switch(str[i]) {
8        case '(': push(&S,str[i]); break;
9        case '{': push(&S,str[i]); break;
10       case '[': push(&S,str[i]); break;
11       case ')': if(pop(&S,&c) != 0 || c != '(') {
12                   clear(&S); return 0;
13                 }
14                 break;
15       case '}': if(pop(&S,&c) != 0 || c != '{') {
16                   clear(&S);
17                   return 0;
18                 }
19                 break;
20       case ']': if(pop(&S,&c) != 0 || c != '[') {
21                   clear(&S); return 0;
22                 }
23                 break;
24     }
25   }
26   if(!isempty(&S)) { clear(&S); return 0; }
27   else             { clear(&S); return 1; }
28 }
```

Notazione polacca inversa

- La **notazione polacca inversa** (**reverse polish notation**) è un particolare tipo di sintassi, utilizzata per le espressioni matematiche.
- La sintassi tipicamente utilizzata per le espressioni matematiche è *infissa*: l'operatore matematico è indicato tra gli operandi.
  - La sintassi infissa richiede regole di associatività e precedenza tra operatori ed è necessario l'uso di parentesi per alterare le regole di precedenza.

```
(1+2)*(3+4) = 21
```

- La notazione polacca inversa fa uso di una sintassi *postfissa*: l'operatore matematico è indicato dopo gli operandi.
  - La sintassi postfissa non richiede regole di associatività e precedenza tra operatori e quindi non richiede l'uso di parentesi.

```
1 2 + 3 4 + * = 21
```

- La valutazione di una espressione in notazione polacca inversa è facilmente implementabile algoritmicamente, facendo uso di una struttura dati di tipo pila.

Notazione polacca inversa: esempio

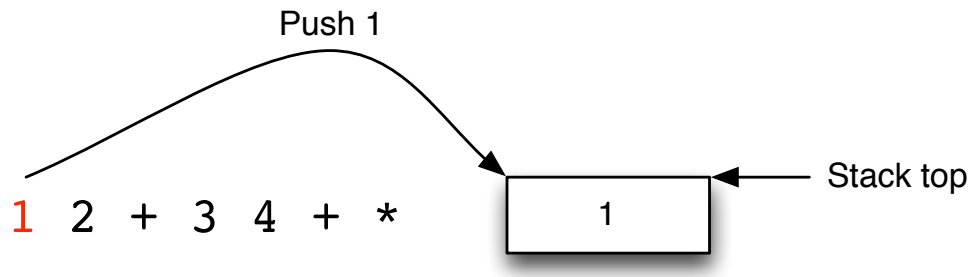
- Algoritmo per valutare una espressione scritta in notazione polacca inversa.
  - Scorriamo l'espressione da sinistra verso destra.
  - Ogni volta che incontriamo un numero: lo inseriamo sulla cima della pila.
  - Ogni volta che incontriamo un operatore n-ario: rimuoviamo dalla pila gli ultimi n numeri, applichiamo l'operatore agli n numeri e inseriamo sulla pila il risultato dell'operazione.
  - Non appena abbiamo terminato di valutare l'espressione, la pila contiene il risultato della valutazione.
- Vediamo come viene valutata la seguente espressione in notazione polacca inversa.

```
1 2 + 3 4 + *
```

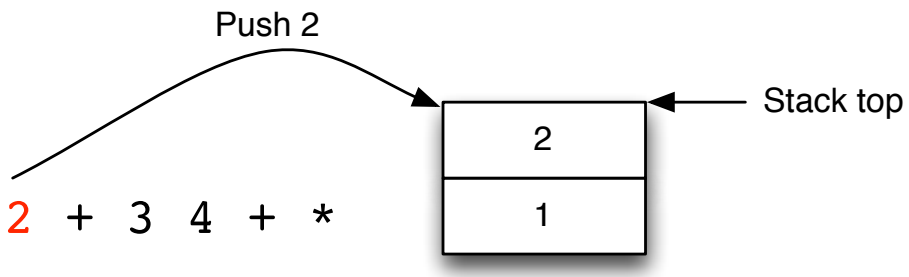
- Nel nostro esempio consideriamo unicamente operatori binari.

Notazione polacca inversa: esempio

- Il primo elemento dell'espressione è un intero: lo inseriamo nella pila.

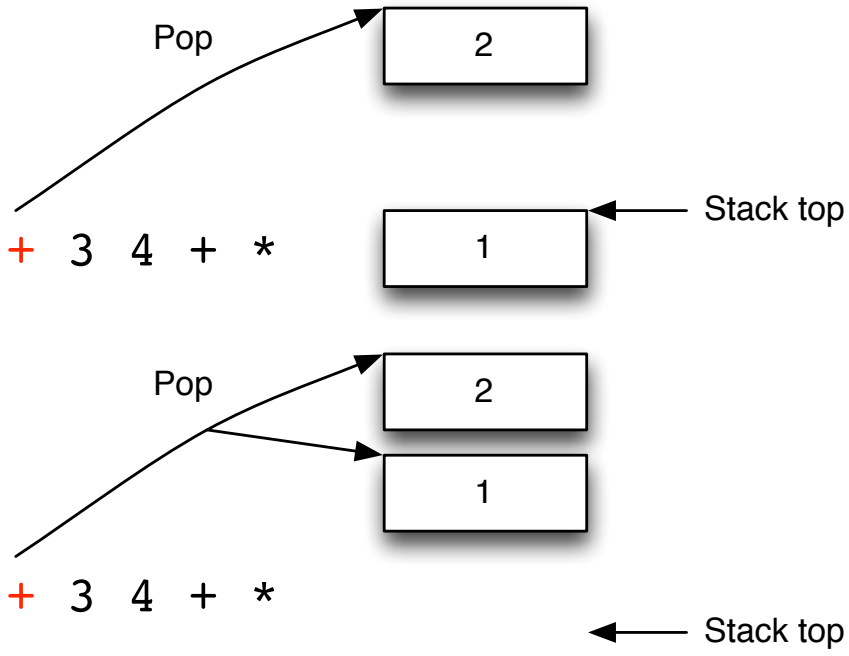


- Il secondo elemento dell'espressione è un intero: lo inseriamo nella pila.

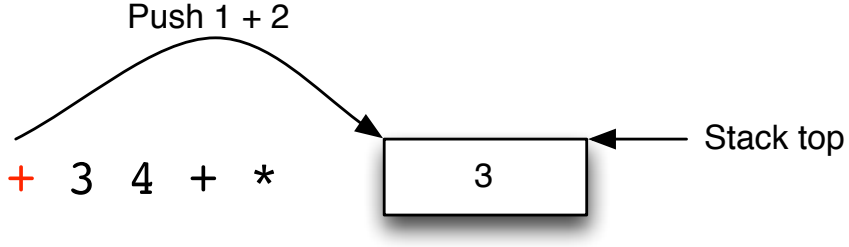


Notazione polacca inversa: esempio

- Il terzo elemento dell'espressione è un'operatore binario: rimuoviamo il secondo e il primo operando dalla pila.

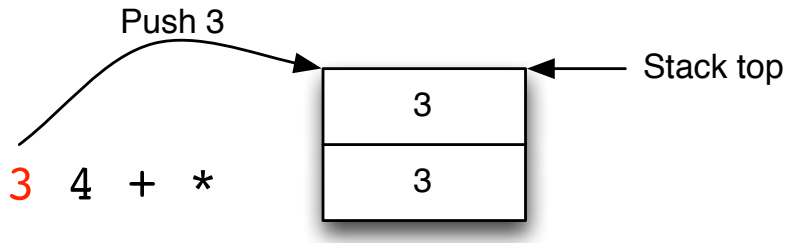


- Dopo avere recuperato i due operandi, applichiamo l'operatore ai due operandi e inseriamo il risultato dell'operazione nella pila.

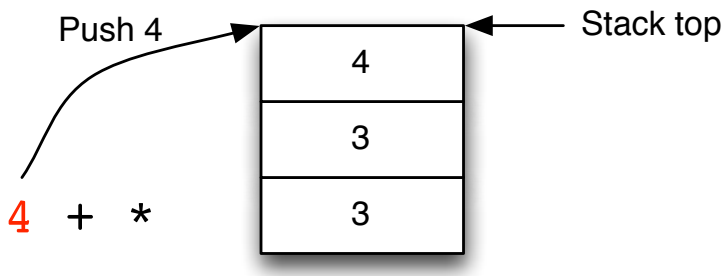


Notazione polacca inversa: esempio

- Il quarto elemento dell'espressione è un intero: lo inseriamo nella pila.

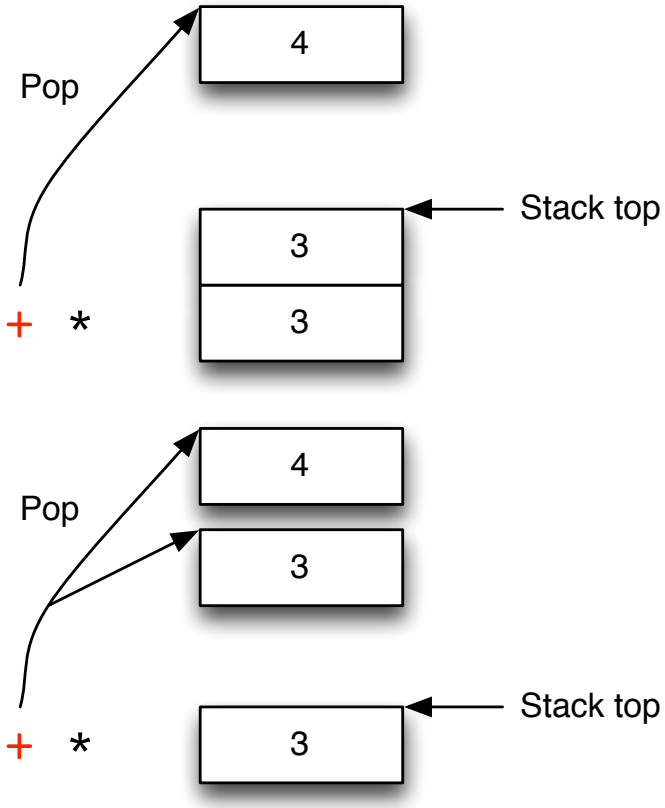


- Il quinto elemento dell'espressione è un intero: lo inseriamo nella pila.

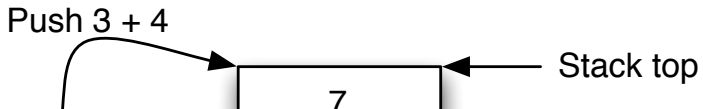


Notazione polacca inversa: esempio

- Il sesto elemento dell'espressione è un'operatore binario: rimuoviamo il secondo e il primo operando dalla pila.

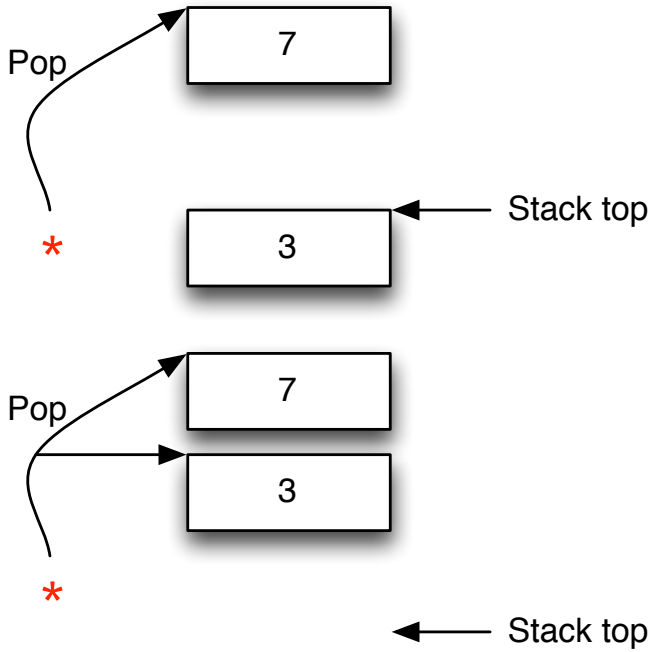


- Dopo avere recuperato i due operandi, applichiamo l'operatore ai due operandi e inseriamo il risultato dell'operazione nella pila.

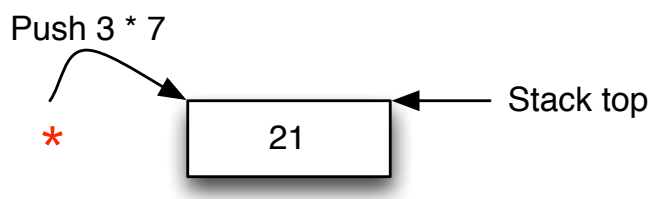


Notazione polacca inversa: esempio

- Il settimo elemento dell'espressione è un'operatore binario: rimuoviamo il secondo e il primo operando dalla pila.



- Dopo avere recuperato i due operandi, applichiamo l'operatore ai due operandi e inseriamo il risultato dell'operazione nella pila.



- L'espressione non contiene altri elementi: il risultato finale è in cima alla pila.

Notazione polacca inversa: implementazione

- Per valutare una espressione in notazione polacca inversa possiamo definire una funzione che prenda come parametro una stringa.
- Dato che la lunghezza della stringa è fissa, possiamo utilizzare per il calcolo una struttura dati di tipo pila con capienza limitata.
  - La lunghezza della stringa è un *upper bound* al numero massimo di elementi che memorizzeremo nella pila.
- L'implementazione richiede funzioni per la gestione del parsing della stringa in input.
  - Dobbiamo essere in grado di distinguere tra operandi e operatori nella stringa.
  - Abbiamo bisogno di una funzione che ritorni il successivo operando o operatore durante la fase di *scanning* della stringa in input.
- Per semplificare l'implementazione imponiamo che la stringa contenente l'espressione contenga operatori ed operandi separati da uno o più spazi.
  - Se la stringa in input non è *ben formata*, l'algoritmo di valutazione termina in condizione di errore.

Notazione polacca inversa: funzione di valutazione

```
1  /* Valuta l'espressione in notazione polacca inversa in una stringa. */
2  int evaluate(char *str, int *res) {
3    if (str == NULL || res == NULL) {
4      return 1;
5    } else {
6      stack S   = init(strlen(str));  // Inizilizza pila
7      char *tok = parse_string(str);  // Inizilizza parser
8      int x, y;
9      while(tok) {                    // Ripete finchè ci sono token
10       if(is_operand(tok)) {         // Il token è un operando
11         push(&S,get_operand(tok));
12       } else if(is_operator(tok) && pop(&S,&y) == 0 && pop(&S,&x) == 0) {
13         switch(get_operator(tok)) { // Il token è un operatore
14           case '+': push(&S,x+y); break;
15           case '-': push(&S,x-y); break;
16           case '*': push(&S,x*y); break;
17           case '/': push(&S,x/y); break;
18         }
19       } else {                      // Il token non è valido
20         clear(&S);
21         return 1;
22       }
23       tok = next_token();           // Recupera il token successivo
24     }
25     if(pop(&S,res) != 0 || !isempty(&S)) { clear(&S); return 1; }
26     else                                 { clear(&S); return 0; }
27   }
28 }
```

Notazione polacca inversa: commenti alla funzione di valutazione

- Valore di ritorno e risultato del calcolo.
  - La funzione `evaluate()` utilizza il valore di ritorno per segnalare se la valutazione è andata a buon fine (ritorna `0`) o meno (ritorna `1`).
  - Il risultato dell'espressione è passato al chiamate tramite l'argomento formale `res`.
- Struttura dati utilizzata.
  - Fa uso di una struttura dati pila con capienza massima pari alla lunghezza della stringa in input (riga 6).
  - La pila viene distrutta se durante la valutazione occorrono errori di parsing (riga 20 e riga 25) e viene distrutta al termine del calcolo (riga 26).
- Parsing della stringa.
  - La funzione `parse_string()` suddivide la stringa in input in *token* (riga 7). Un token può rappresentare un operando oppure un operatore. In questa implementazione i token sono stringhe.
  - La funzione `next_token()` restituisce in ordine (ad ogni chiamata successiva) i token in cui è stata scomposta la stringa (riga 23).
- Gestione dei token.
  - Le funzioni `is_operand()` (riga 10) e `is_operator()` (riga 12) verificano che il token passato sia un operando o operatore, rispettivamente.
  - Le funzioni `get_operand()` (riga 11) e `get_operator()` (riga 13) convertono il token in un operando o operatore, rispettivamente.

Notazione polacca inversa: funzioni di parsing della stringa

- Per il parsing della stringa utilizziamo la funzione di libreria `strtok()`, che suddivide la stringa in input in una serie di *token* (sotto-stringhe),
- I caratteri che *delimitano* i token nella stringa sono specificati come parametro di `strtok()`: nel nostro caso, il delimitatore è semplicemente uno spazio (riga 9).
- La funzione `strtok()` modifica la stringa in input, quindi in `parse_string()` passiamo a `strtok()` una copia della stringa originale (`strdup()`, riga 9).
- La funzione `next_token()` utilizza `strtok()` per ottenere il token successivo della stringa parsata (riga 15).

```
1  /*
2   * Suddivide la stringa s in una serie di token.
3   * Un token è una sottostringa di s non contentente
4   * spazi.
5   *
6   * Ritorna il primo token.
7   */
8  char *parse_string(char *s) {
9    return strtok(strdup(s)," ");
10 }
11 /*
12  * Ritorna il token successivo.
13  */
14 char *next_token(void) {
15   return strtok(NULL," ");
16 }
```

Notazione polacca inversa: funzioni di gestione dei token

- La funzione `is_operand()` verifica che la stringa token contenga solo cifre numeriche.
- La funzione `is_operator()` verifica che la stringa token abbia lunghezza 1 e contenga uno dei caratteri `+`, `-`, `*`, `\`.
- La funzione `get_operand()` converte il contenuto della stringa token in intero.

```
1  /* Verifica che il token tok sia una sequenza di sole cifre. */
2  int is_operand(char *tok) {
3    if(*tok == '\0') {
4      return 0;
5    } else {
6      while(*tok && isdigit(*tok)) tok++;
7      return *tok == '\0';
8    }
9  }
10 /* Verifica che il token tok sia uno dei caratteri '+', '-', '*', '\' */
11 int is_operator(char *tok) {
12   return  (*tok == '+' || *tok == '-'  ||
13            *tok == '*' || *tok == '/') && strlen(tok) == 1;
14 }
15 /* Ritorna il numero intero rappresentato nel token tok. */
16 int get_operand(char *tok) {
17   return atoi(tok);
18 }
19 /* Ritorna l'operazione rappresentata nel token tok. */
20 char get_operator(char *tok) {
21   return *tok;
22 }
```