Valutazione di espressioni

- La valutazione di espressioni è un problema che appare banale ma non lo è affatto…
- Come rappresentare l'espressione?
- Qual è il modo migliore per fare "capire" l'espressione alla macchina?
- Come costruire un algoritmo di valutazione che sia anche estendibile?

1

Valutazione di espressioni

- Per semplicità da ora in poi si farà riferimento ad espressioni di tipo aritmetico in cui sono coinvolte le sole quattro operazioni fondamentali (con relativa priorità)
- Prima si studierà una rappresentazione che consenta la valutazione nel modo più semplice possibile
- Poi si studierà una soluzione per la valutazione della rappresentazione standard (infissa)

2

Rappresentazione postfissa

- Anche detta RPN (Reverse Polish Notation)
- Nella RPN gli operandi **precedono** gli operatori, rimuovendo sia la necessità di parentesi sia di tenere conto della priorità degli operatori

- 4 + 7 → 4 7 +
- 3 * (4 + 7) → 3 4 7 + *
- 5 – 3 – 1 → 5 3 – 1 – → Attenzione!!!

- Ogni operatore è direttamente preceduto da entrambi i propri operandi
- Ogni volta che si incontra un operatore si può eseguire l'operazione e sostituire il risultato al posto dell'operazione stessa (operandi e operatore)

- 3 4 7 + * 5 - → 3 11 * 5 -
- 3 11 * 5 - → 33 5 -
- 33 5 - → 28

3

Algoritmo di valutazione – RPN

- L'algoritmo di valutazione di una espressione RPN è basato su uno stack
  - Ogni volta che si incontra un operando, lo si inserisce nello stack
  - Ogni volta che si incontra un operatore
    - Si estraggono due operandi (i due operandi che con l'operatore compongono l'operazione)
    - Si esegue l'operazione
    - Si inserisce il risultato nello stack
  - Al termine della valutazione, nello stack c'è un solo valore che rappresenta il risultato della valutazione dell'epressione
- Il tutto sembrerà un po' cervellotico, ma le espressioni RPN hanno il vantaggio di essere valutate in modo semplice e veloce

4

Algoritmo di valutazione – RPN

- Implicazioni pratiche:
  - Il calcolo procede in modo ordinato da sinistra a destra
  - Non ci sono parentesi: non sono necessarie
  - Gli operandi precedono il loro operatore: operandi e operatore sono rimossi dall'espressione nel momento in cui l'operazione viene valutata
  - Quando un'operazione viene valutata, il risultato diventa esso stesso un operando per operatori che vengono successivamente

5

Scelte implementative

- L'espressione è rappresentata tramite una stringa
- Occorre un modulo per la suddivisione della stringa in operandi ed operatori
- Una volta costruito il modulo (meglio un ADT), è possibile implementare l'algoritmo precedentemente citato…
- …facendo uso di un opportuno *stack* (ancora ADT) per gli operandi

6

Tokenizer

- Un *tokenizer* è un modulo software che scandisce una stringa di testo e determina se le varie sottostringhe della stringa possono essere riconosciute come *token* (gettoni) significativi
- Mangia la stringa un po' alla volta restituendo i componenti significativi
- Nel nostro caso, mangia l'espressione e restituisce (riconoscendoli) gli operandi e altri simboli (operatori, parentesi, …)
- Non effettua alcuna valutazione!!

7

Tokenizer – operazioni

- Costruzione (e distruzione) del *tokenizer* data una stringa contenente un'espressione
- Avanzamento del *tokenizer* – tenta di leggere il *token* successivo e riporta il successo dell'operazione
  - L'eventuale *token* letto diventa il *token* corrente
  - L'avanzamento fallisce quando il *tokenizer* è arrivato alla fine dell'espressione – tutti i tentativi di avanzamento successivi ad un tentativo fallito riporteranno insuccesso
- Lettura del *token* corrente
- Lettura del tipo di *token* corrente (numero o simbolo)

8

Tokenizer – Struttura

- Il *tokenizer* può essere modellato come un ADT la cui struttura sottostante contiene:
  - Una stringa contenente l'espressione
  - Il *token* corrente – una stringa
  - Il tipo di *token* corrente – un enumerativo
  - L'indice di scorrimento della stringa – un intero

9

Definizioni comuni

```
#ifndef COMMON_DEFINES
#define COMMON_DEFINES

#define MAX_STACK_DIM 512

typedef enum {false, true} boolean;

typedef int OperandType;
#define OPERAND_FORMAT "%d"

typedef char SymbolType;

#endif
```

10

ADT Tokenizer – Definizioni

```
#define EXP_BUFFER_DIM 512
#define TOKEN_DIM 16

#ifndef TOKENIZER
#define TOKENIZER
typedef char ExpressionBuffer[EXP_BUFFER_DIM];
typedef char Token[TOKEN_DIM];

typedef enum
{
    None,
    Number,
    Symbol,
} TokenType;

typedef struct
{
    ExpressionBuffer buffer;
    Token currentToken;
    TokenType currentTokenType;
    int currentIdx;
} Tokenizer;
#endif
```

11

ADT Tokenizer – Interfaccia

```
Tokenizer* newTokenizer(char expression[]);

boolean readNextToken(Tokenizer *tokenizer);

void getCurrentToken(Tokenizer *tokenizer, Token t);

TokenType getCurrentTokenType(Tokenizer *tokenizer);
```

12

ADT Tokenizer – Costruzione

```
Tokenizer* newTokenizer(char expression[EXP_BUFFER_DIM])
{
    Tokenizer *tokenizer =
                  (Tokenizer*)malloc(sizeof(Tokenizer));
    strcpy(tokenizer->buffer, expression);
    tokenizer->currentIdx = 0;
    strcpy(tokenizer->currentToken, "");
    tokenizer->currentTokenType = None;
    return tokenizer;
}
```

13

ADT Tokenizer – Avanzamento

- Pseudocodice
  - Tralasciare eventuali spazi bianchi
    - Se dopo l'eliminazione degli spazi si è raggiunta la fine dell'espressione, terminare con insuccesso
  - Se il *token* successivo è un numero
    - Leggere il numero
    - Avanzare opportunamente l'indice di scorrimento
    - Copiare nel *token* corrente il numero letto
    - Assegnare al tipo di *token* corrente il tipo numero
  - Altrimenti il *token* successivo è un simbolo
    - Leggere il simbolo
    - Avanzare opportunamente l'indice di scorrimento
    - Copiare nel *token* corrente il simbolo letto
    - Assegnare al tipo di *token* corrente il tipo simbolo

14

Riconoscere e leggere un numero

- Per fattorizzare meglio il codice, si può prevedere una funzione di supporto
- Tale funzione prende come argomenti:
  - La stringa da analizzare (sottostringa "destra" dell'espressione)
  - Un intero per riferimento dove inserire il numero di caratteri che formano il numero
  - Un token dove inserire il numero in formato stringa

15

Riconoscere e leggere un numero

- Pseudocodice
  - Scandire la stringa partendo dall'inizio
    - Se il carattere corrente è una cifra (da 0 a 9) allora inserire tale cifra nel *token*
    - Altrimenti terminare la lettura
  - Se sono stati letti caratteri
    - Inserire il terminatore nel *token*
    - Assegnare al parametro per riferimento il numero di caratteri letti
    - Restituire successo
  - Altrimenti restituire insuccesso

16

Riconoscere e leggere un numero

```
boolean readNumber(char *buffer, int *readChars,
                                                Token number)
{
   int i;
   for (i = 0; buffer[i]>='0' && buffer[i]<='9'; i++)
       number[i] = buffer[i];
   if (i > 0)
   {
       number[i] = '\0';
       *readChars = i;
       return true;
   }
   return false;
}
```

17

ADT Tokenizer – Avanzamento (1)

```
boolean readNextToken(Tokenizer* tokenizer)
{
    int readChars;
    Token t;
    while (tokenizer->buffer[tokenizer->currentIdx] != '\0' &&
        tokenizer->buffer[tokenizer->currentIdx] == ' ')
    {
        tokenizer->currentIdx++; //trimLeft
    }
    if (tokenizer->buffer[tokenizer->currentIdx] == '\0')
        return false;
    //continua…
```

Elimina gli spazi bianchi…

18

ADT Tokenizer – Avanzamento (2)

```
    //…continua
    if (readNumber(tokenizer->buffer + tokenizer->currentIdx,
                                                &readChars, t))
    {
        strcpy(tokenizer->currentToken, t);
        tokenizer->currentTokenType = Number;
        tokenizer->currentIdx += readChars;
    }
    else
    {
        tokenizer->currentToken[0] =
                    tokenizer->buffer[tokenizer->currentIdx];
        tokenizer->currentToken[1] = '\0';
        tokenizer->currentTokenType = Symbol;
        tokenizer->currentIdx++;
    }
    return true;
}
```

Se legge un numero…

…altrimenti è un simbolo → un carattere

19

ADT Tokenizer – Accesso

```
void getCurrentToken(Tokenizer* tokenizer, Token t)
{
    strcpy(t, tokenizer->currentToken);
}

TokenType getCurrentTokenType(Tokenizer* tokenizer)
{
    return tokenizer->currentTokenType;
}
```

20

Stack degli operandi - Interfaccia

- Si riporta solo l'interfaccia dell'ADT OperandStack

```
OperandStack newOperandStack(void);
void operandPush(OperandType, OperandStack);
OperandType operandPop(OperandStack);
boolean isEmptyOperandStack(OperandStack);
boolean isFullOperandStack(OperandStack);
```

21

Postfix Expression Evaluation

Pseudocodice

- Creare stack degli operandi e *tokenizer*
- Finché nel *tokenizer* ci sono *token* da leggere
  - Se il *token* corrente è di tipo "numero"
    - Leggere il *token* corrente, tradurlo in valore numerico ed inserirlo nello stack degli operandi
  - Se il *token* corrente è di tipo "simbolo", si tratta di un operatore
    - Leggere il *token* corrente e tradurlo in operatore
    - Recuperare dallo stack due operandi (il primo recuperato è il secondo operando)
    - Eseguire l'operazione
    - Inserire nello stack il risultato dell'operazione
- Quando non ci sono più token, nello stack ci deve essere solamente un operando che rappresenta il risultato

22

Al solito: massima fattorizzazione

- Traduzione di un token in valore numerico

```
OperandType getOperandFromToken(Token t)
{
    OperandType operand;
    sscanf(t, OPERAND_FORMAT, &operand) == 1;
    return operand;
}
```

- Traduzione di un token in simbolo

```
SymbolType getSymbolFromToken(Token t)
{
    return t[0];
}
```

23

…massima fattorizzazione

- Recupero di due operandi

```
boolean getOperands(OperandStack evalStack,
      OperandType *operand1, OperandType *operand2)
{
    if (isEmptyOperandStack(evalStack))
        return false;
    *operand2 = operandPop(evalStack);
    if (isEmptyOperandStack(evalStack))
        return false;
    *operand1 = operandPop(evalStack);
    return true;
}
```

24

…massima fattorizzazione

```
OperandType evaluate(OperandType operand1,
                     OperandType operand2, SymbolType op)
{
    switch (op)
    {
    case '+':
        return operand1 + operand2;
    case '-':
        return  operand1 - operand2;
    case '*':
        return operand1 * operand2;
    case '/':
        return operand1 / operand2;
    default:
        abort();
    }
    return 0;
}
```

25

Valutazione!

```
boolean postfixEval(char expr[], OperandType *value)
{
    OperandStack evalStack = newOperandStack();
    Tokenizer *tok = newTokenizer(expr);
    Token t;
    while (readNextToken(tok))
    {
        switch (getCurrentTokenType(tok))
        {
        case Number:
            {
                OperandType number;
                getCurrentToken(tok, t);
                number = getOperandFromToken(t);
                operandPush(number, evalStack);
                break;
            }
    //continua…
```

Gli operandi finiscono sempre nello stack

26

Valutazione!

```
        case Symbol:
            {
                OperandType operand1, operand2;
                SymbolType op;
                getCurrentToken(tok, t);
                op = getSymbolFromToken(t);
                if (!getOperands(evalStack, &operand1,
                                                &operand2))
                    return false;
                operandPush(evaluate(operand1, operand2,
                                                op), evalStack);
                break;
            }
        default:
            assert(false);
        }
    }
    //…continua…
```

Gli operatori provocano l'estrazione dei relativi operandi e "forzano" la valutazione

27

Valutazione!

```
    //…continua
    if (isEmptyOperandStack(evalStack))
        return false;
    *value = operandPop(evalStack);
    return isEmptyOperandStack(evalStack);
}
```

Al termine della valutazione, nello stack deve rimanere il solo risultato

28

Peccato che…

- Noi esseri umani gradiamo le espressioni infisse
- Come si traduce un'espressione infissa in un'espressione postfissa?
- Edsger Dijkstra con il suo *Shunting yard algorithm* ha dato la soluzione
  - *Shunting yard* perché il funzionamento assomiglia al funzionamento degli scambi dei binari del treno…
  - Utilizza un solo stack, questa volta per gli operatori!

29

Shunting yard (semplificato) Pseudocodice

- Finché ci sono token da leggere
  - Leggere un token
  - Se il token è un numero, appenderlo all'espressione in uscita
  - Se il token è un operatore o_1
    - Finché esiste un operatore o_2 in cima allo stack la cui precedenza sia maggiore o uguale all'operatore o_1
      - Estrarre o_2 dallo stack ed appenderlo all'espressione in uscita
    - Inserire l'operatore o_1 nello stack degli operatori
  - Se il token è una parentesi aperta, inserirla nello stack
  - Se il token è una parentesi chiusa, estrarre tutti gli operatori dallo stack ed appenderli all'espressione in uscita finché dallo stack non viene estratta una parentesi aperta – se la parentesi aperta non è presente: errore!
- Quando non ci sono più token da leggere, estrarre dallo stack tutti gli operatori (se ne sono rimasti) ed appenderli all'espressione di uscita. Nello stack ci devono essere solo operatori (non parentesi) altrimenti errore!

30

Fattorizzazione – 1

- Appendere un token numerico all'espressione d'uscita

```
void appendNumber(char expr[], Token t)
{
    strcat(expr, t);
    strcat(expr, " ");
}
```

- Appendere un simbolo all'espressione d'uscita

```
void appendSymbol(char expr[], SymbolType symbol)
{
    int len = strlen(expr);
    expr[len] = symbol;
    expr[len + 1] = ' ';
    expr[len + 2] = '\0';
}
```

31

Fattorizzazione – 2

- È un operatore?

```
boolean isOperator(SymbolType symbol)
{
    return symbol == '+' || symbol == '*' ||
                    symbol == '-' || symbol == '/';
}
```

- Ottenere la priorità di un operatore

```
int getOperatorPriority(SymbolType op)
{
    switch (op)
    {
    case '+':
    case '-':
        return 1;
    case '*':
    case '/':
        return 2;
    default:
        abort();
        return 0;
    }
}
```

32

Fattorizzazione – 3

- Comparazione priorità operatori

```
int compareOpPriority(SymbolType op1,
                                        SymbolType op2)
{
    return getOperatorPriority(op1) –
                              getOperatorPriority(op2);
}
```

33

Shunting Yard (semplificato) – 1

```
boolean infixToPostfix(char inExpr[], char outExpr[])
{
    SymbolStack symStack = newSymbolStack();
    Tokenizer *tok = newTokenizer(inExpr);
    Token t;
    outExpr[0] = '\0';

    while (readNextToken(tok))
    {
        switch (getCurrentTokenType(tok))
        {
        case Number:
            {
                getCurrentToken(tok, t);
                appendNumber(outExpr, t);
                break;
            }
        \\continua...
```

Se è un numero finisce direttamente sull'output

34

Shunting Yard (semplificato) – 2

```
case Symbol:
    {
        SymbolType symbol;
        getCurrentToken(tok, t);
        symbol = getSymbolFromToken(t);
        if (isOperator(symbol)) //E' un operatore
        {
            while (!isEmptySymbolStack(symStack) &&
                   isOperator(symbolPeek(symStack)) &&
                   compareOpPriority(symbol,
                          symbolPeek(symStack)) <= 0)
            {
                //Estrae l'operatore letto con peek
                SymbolType firstSymbolInStack =
                                  symbolPop(symStack);
                appendSymbol(outExpr,
                                  firstSymbolInStack);
            }
            symbolPush(symbol, symStack);
        }
```

Se è un operatore si mettono in output tutti gli operatori nello stack che sono meno prioritari – poi si mette l'operatore nello stack

Shunting Yard (semplificato) – 3

```
    else //Non è un operatore ma una parentesi
    {
        switch (symbol)
        {
        case '(':
            {
                symbolPush(symbol, symStack);
                break;
            }
        case ')':
            {
                SymbolType firstSymbolInStack;
                for (firstSymbolInStack=symbolPop(symStack);
                     firstSymbolInStack != '(';
                     firstSymbolInStack=symbolPop(symStack))
                {
                    appendSymbol(outExpr,firstSymbolInStack);
                }
                break;
            }
        }
    }
    break;      //case Symbol
}
```

Se è una parentesi aperta, finisce sullo stack

Se è una parentesi chiusa si mette in output lo stack fino alla parentesi aperta

Shunting Yard (semplificato) – 3

```
        }     //switch (getCurrentTokenType(tok))
    }   //while (readNextToken(tok))
    while (!isEmptySymbolStack(symStack) &&
                          symbolPeek(symStack) != '(')
    {
        SymbolType op = symbolPop(symStack);
        appendSymbol(outExpr, op);
    }
    return isEmptySymbolStack(symStack);
} //end function
```

Terminata l'espressione, si mettono in output tutti gli operatori rimasti nello stack. Non si devono incontrare parentesi aperte. Al termine lo stack deve essere vuoto

37

Valutatore espressioni infisse

- È "sufficiente" unire l'algoritmo *Shunting Yard* visto con l'algoritmo di valutazione della postfissa ed il gioco è fatto!
  - Al posto di mettere in output, si effettua la valutazione
  - Occorrono due stack: uno per gli operandi, uno per gli operatori
  - …il progetto completo di test è sul sito del corso!

38

Nessuno si è accorto di niente?!

- E la deallocazione della memoria occupata dalle istanze degli ADT usati?
- …sempre nel progetto scaricabile dal sito del corso…

39