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 la necessità 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
  - Maneggevole ed "espressiva"
- Occorrono:
  - Un modulo generico per la suddivisione della stringa in parti significative (*token*) → *Tokenizer*
  - Un modulo specifico per le espressioni che, per ogni parte significativa (*token*), indichi di che tipo è (operatore, numero, parentesi, ...) → *Lexer*
- Una volta costruiti i moduli, 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 (mangia) una stringa di testo e restituisce *token* (gettoni) considerati significativi
- Nel nostro caso, mangia l'espressione e restituisce gli operandi e altri simboli (operatori, parentesi, …)
- Non effettua alcuna valutazione!!

- L'idea è quella di costruire un *tokenizer* generico che si possa istruire su cosa sia significativo e cosa no
  - Istruire il *tokenizer* con un insieme di caratteri che non fanno mai parte di un token → caratteri di separazione (*ignore chars*)

7

Tokenizer – Operazioni

- Costruzione (e distruzione) del *tokenizer* data una stringa contenente un'espressione, una stringa contenente caratteri che da soli costituiscono un *token* (*token chars*), una stringa contenente caratteri di "separazione" (*ignore chars*)

- 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

8

Tokenizer – Struttura

- Il *tokenizer* può essere modellato come un ADT la cui struttura sottostante contiene:
  - Una stringa contenente l'espressione
  - Una stringa contenente i caratteri che non entreranno a far parte di alcun *token* che saranno quindi interpretati come di "separazione" (*ignore chars*)
  - Il *token* corrente – una stringa
  - L'indice di scorrimento della stringa – un intero

9

Definizioni comuni

```
#ifndef COMMON_DEFINES
#define COMMON_DEFINES

#define MAX_STACK_DIM 1024

typedef enum {false, true} boolean;

#endif
```

10

ADT Tokenizer – Definizioni

```
#define EXP_BUFFER_DIM 1024
#define CONTROL_CHAR_DIM 26
#define TOKEN_DIM 16
#ifndef SIMPLETOKENIZER
#define SIMPLETOKENIZER

typedef char ExpressionBuffer[EXP_BUFFER_DIM];
typedef char ControlChars[CONTROL_CHAR_DIM];
typedef char Token[TOKEN_DIM];
typedef struct
{
    ExpressionBuffer buffer;
    ControlChars ignoreChars;
    Token currentToken;
    int currentIdx;
} SimpleTokenizer;
#endif
```

11

ADT Tokenizer – Interfaccia

```
SimpleTokenizer* newTokenizer(char expression[],
       char ignores[]);

void destroyTokenizer(SimpleTokenizer *tokenizer);

boolean readNextTokenFromTk(SimpleTokenizer* tokenizer);

void getCurrentTokenFromTk(SimpleTokenizer* tokenizer,
       Token t);
```

12

ADT Tokenizer – Costruzione

```
SimpleTokenizer* newTokenizer(char expression[], char tokens[],
   char ignores[])
{
    SimpleTokenizer *tokenizer =
       (SimpleTokenizer*)malloc(sizeof(SimpleTokenizer));

    assert(strlen(expression) <= EXP_BUFFER_DIM - 1);
    strcpy(tokenizer->buffer, expression);
    assert(strlen(ignores) < CONTROL_CHAR_DIM);
    strcpy(tokenizer->ignoreChars, ignores);
    tokenizer->currentIdx = 0;
    strcpy(tokenizer->currentToken, "");
    return tokenizer;
}
```

13

ADT Tokenizer – Avanzamento (1)

Un po' di "nomenclatura"

- "il *token* corrente" = area di memoria nella quale inserire il *token* che si sta leggendo
- "il carattere corrente" = carattere contenuto nell'espressione puntato dall'indice di avanzamento

14

ADT Tokenizer – Avanzamento (2)

1. Inizializza il *token* corrente al *token* vuoto
2. Finché il carattere corrente non è il terminatore dell'espressione ed è un carattere di separazione
   a. Passa al carattere successivo
3. Se il carattere corrente è il terminatore dell'espressione
   a. Termina con insuccesso
4. Finché il carattere corrente non è un carattere di separazione e non è il terminatore dell'espressione
   1. Aggiungi il carattere corrente al token corrente
   2. Passa al carattere successivo
5. Termina con successo

15

ADT Tokenizer – Operazioni

Servono operazioni per:

- Inizializzare il *token* corrente al *token* vuoto
  - **`initCurrentToken`**
- Verificare se il carattere corrente è il terminatore dell'espressione
  - **`isCurrentExpressionEnd`**
- Verificare se il carattere corrente è un carattere di separazione (*ignore char*)
  - **`isCurrentIgnoreChar`**
- Passare al carattere successivo
  - **`skipCurrentChar`**
- Aggiungere il carattere corrente al *token* corrente
  - **`addCurrentCharToCurrentToken`**

16

ADT Tokenizer – Avanzamento (1i)

```
boolean readNextTokenFromTk(SimpleTokenizer* tokenizer)
{
    initCurrentToken(tokenizer);
    while (!isCurrentExpressionEnd(tokenizer) &&
        isCurrentIgnoreChar(tokenizer))
    {  //via ignore chars e stop al terminatore
        skipCurrentChar(tokenizer);
    }
    if (isCurrentExpressionEnd(tokenizer))  //terminatore?
    {
        return false;
    }
    //continua…
```

17

ADT Tokenizer – Avanzamento (2i)

```
    //...continua
    do
    {
        addCurrentCharToCurrentToken(tokenizer);
        skipCurrentChar(tokenizer);
    }
    while (!isCurrentTokenChar(tokenizer) &&
        !isCurrentIgnoreChar(tokenizer) &&
        !isCurrentExpressionEnd(tokenizer));
    return true;
}
```

18

ADT Tokenizer – Operazioni (1i)

```
char getCurrentChar(SimpleTokenizer *tokenizer)
{
    return tokenizer->buffer[tokenizer->currentIdx];
}
void skipCurrentChar(SimpleTokenizer * tokenizer)
{
    tokenizer->currentIdx++;
}
void initCurrentToken(SimpleTokenizer *tokenizer)
{
    tokenizer->currentToken[0] = '\0';
}
void addCurrentCharToCurrentToken(SimpleTokenizer *tokenizer)
{
    int len = strlen(tokenizer->currentToken);
    tokenizer->currentToken[len] = getCurrentChar(tokenizer);
    tokenizer->currentToken[len + 1] = '\0';
}
boolean isCurrentExpressionEnd(SimpleTokenizer *tokenizer)
{
    return getCurrentChar(tokenizer) == '\0';
}
```

19

ADT Tokenizer – Operazioni (2i)

```
boolean isCurrentIgnoreChar(SimpleTokenizer *tokenizer)
{
    if (isCurrentExpressionEnd(tokenizer))
    {
        return false;
    }
    else
    {
        return strchr(tokenizer->ignoreChars,
            getCurrentChar(tokenizer)) != NULL;
    }
}

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

20

ADT Tokenizer – Note

- Le operazioni “aggiuntive” vengono utilizzate solamente all’interno dell’ADT
  - devono essere considerate private → non visibili all’esterno
  - Non vanno dichiarate nello header file
- E la distruzione?

21

ADT Tokenizer – Esempio 1

```
SimpleTokenizer *tok =
      newTokenizer("10 + 3   / 12.34 (  ", " ");
while (readNextTokenFromTk(tok))
{
   Token t;
   getCurrentTokenFromTk(tok, t);
   printf("%s|", t);
}
```

Deve stampare:

`10|+|3|/|12.34|(|`

Non si tratta di un’espressione valida?

Non importa, il tokenizer non interpreta!

22

ADT Tokenizer – Esempio 2

```
SimpleTokenizer *tok =
      newTokenizer("Gustavo;La Trippa#17/12/1975#", ";#");
while (readNextTokenFromTk(tok))
{
   Token t;
   getCurrentTokenFromTk(tok, t);
   printf("%s|", t);
}
```

Cosa stampa?

23

ADT Lexer

- Un *lexer* è un modulo software che prende in ingresso l'uscita di un *tokenizer* e classifica i *token* a seconda del contesto
- Nel nostro caso si avrà a che fare con un *lexer* per espressioni che dovrà classificare numeri interi e operatori (e parentesi)
- Chi usa il *lexer* non deve vedere che "sotto" c'è un *tokenizer* → il *tokenizer* è incapsulato nel *lexer*

24

ADT Lexer – Operazioni

- Costruzione e Distruzione
- Avanzamento (delega al *tokenizer* sottostante)
- Lettura del *token* corrente (delega al *tokenizer* sottostante)
- Classificazione del *token* corrente; si vogliono riconoscere:
  - Operatori (somma, sottrazione, moltiplicazione, divisione)
  - Parentesi (chiusa, aperta)
  - Numeri interi

25

ADT Lexer – Struttura

- Il lexer è composto di:
  - Un *tokenizer*
  - Un'indicazione del tipo di *token* corrente

```
typedef SimpleTokenizer Tokenizer;

typedef enum
{
    None, Other, IntNumber, OperatorPlus, OperatorMinus,
    OperatorMul, OperatorDiv, ParOpen, ParClose,
} SymbolType;

typedef struct
{
    Tokenizer* tokenizer;
    SymbolType currentTokenType;
} ExpressionLexer;
```

26

ADT Lexer – Interfaccia

```
ExpressionLexer* newExpressionLexer(char expression[]);
void destroyExpressionLexer(ExpressionLexer *lexer);
boolean readNextTokenFromLex(ExpressionLexer* lexer);
void getCurrentTokenFromLex(ExpressionLexer* lexer, Token t);
SymbolType getCurrentTokenType(ExpressionLexer* lexer);
```

27

ADT Lexer – Costruzione

```
ExpressionLexer* newExpressionLexer(char expression[])
{
    ExpressionLexer *lexer =
       (ExpressionLexer*)malloc(sizeof(ExpressionLexer));
    lexer->tokenizer = newTokenizer(expression, " ");
    lexer->currentTokenType = None;
    return lexer;
}

void destroyExpressionLexer(ExpressionLexer *lexer)
{
    destroyTokenizer(lexer->tokenizer);
    free(lexer);
}
```

28

ADT Lexer – Avanzamento

1. Avanzare il *tokenizer* sottostante
2. Se l'avanzamento ha avuto successo
   a. Recuperare il *token* corrente
   b. Classificare il *token*

Classificare il *token*?

- Prima classificare i *token* di un solo carattere (è più semplice!)
- Poi classificare i *token* complessi → al momento solo i numeri interi!

29

Il problema dell'Operando

- Di che tipo è l'operando?
- Intero? Float? Double? Una rappresentazione strana?
- Si vuole fare qualcosa che sia "pluggabile" in modo da poter cambiare il tipo di operando in modo semplice
- La strada è la solita: un ADT che definisca:
  - Il tipo di operando
  - Le operazioni base
  - Una funzione che dica se all'interno di un token è "nascosto" un operando
  - Una funzione di trasformazione Token → Operando

30

ADT Operand

```
#ifndef OPERAND
#define OPERAND

typedef int OperandType;

OperandType doAdd(OperandType o1, OperandType o2);
OperandType doSub(OperandType o1, OperandType o2);
OperandType doMul(OperandType o1, OperandType o2);
OperandType doDiv(OperandType o1, OperandType o2);

boolean isOperand(Token t);
OperandType convertToOperand(Token t);

#endif
```

Finché l'operando è di tipo primitivo l'implementazione è ovvia...

31

ADT Operand

```
boolean isOperand(Token t)
{
    int i;
    int len = strlen(t);
    for (i = 0; i < len; i++)
    {
        if (!isdigit(t[i]))
            return false;
    }
    return true;
}

OperandType convertToOperand(Token t)
{
    int result;
    assert(isOperand(t));
    sscanf(t, "%d", &result);
    return result;
}
```

32

ADT Lexer – Classificazione

```
SymbolType classifyToken(Token t)
{
    SymbolType type = None;
    switch (t[0])
    {
        case '+': type = OperatorPlus; break;
        case '-': type = OperatorMinus; break;
        case '*': type = OperatorMul; break;
        case '/': type = OperatorDiv; break;
        case '(': type = ParOpen; break;
        case ')': type = ParClose; break;
        default:
            type = isOperand(t) ? Operand : Other;
            break;
    }
    return type;
}
```

È giusto che il lexer decida la rappresentazione degli operatori?

33

ADT Lexer – Operazioni

```
boolean readNextTokenFromLex(ExpressionLexer* lexer)
{
    //Lettura token dal tokenizer
    boolean result = readNextTokenFromTk(lexer->tokenizer);
    if (result)
    {   //Recupero e classificazione del token
        Token currentToken;
        getCurrentTokenFromTk(lexer->tokenizer,
                                        currentToken);
        lexer->currentTokenType =
                        classifyToken(currentToken);
    }
    return result;
}
```

34

ADT Lexer – Operazioni

```
void getCurrentTokenFromLex(ExpressionLexer* lexer, Token t)
{
    getCurrentTokenFromTk(lexer->tokenizer, t);
}

SymbolType getCurrentTokenType(ExpressionLexer* lexer)
{
    return lexer->currentTokenType;
}
```

35

ADT Lexer – Note

- Il *tokenizer* è completamente incapsulato: chi usa il *lexer* non sa che sta usando un *tokenizer*

```
ExpressionLexer *lex =
      newExpressionLexer("10 + 3   / 12.34 (  ");
while (readNextTokenFromLex(lex))
{
   Token t;
   getCurrentTokenFromLex(lex, t);
   printf("%d - %s", getCurrentTokenType(lex), t);
}
```

Cosa stampa?

36

Stack degli operandi - Interfaccia

- Si riporta solo l'interfaccia dell'ADT OperandStack

```
OperandStack newOperandStack(void);
void pushOperand(OperandStack, OperandType);
OperandType popOperand(OperandStack);
boolean isEmptyOperandStack(OperandStack);
boolean isFullOperandStack(OperandStack);
```

37

Postfix Expression Evaluation

Pseudocodice

- Creare stack degli operandi e *lexer*
- Finché nel *lexer* ci sono *token* da leggere
  - Se il *token* corrente è un "numero"
    - Leggere il *token* corrente ed inserirlo nello stack degli operandi
  - Se il *token* corrente è un operatore:
    - Recuperare dallo stack due operandi (il primo recuperato è il secondo operando)
    - A seconda del tipo di operatore, eseguire l'operazione
    - Inserire nello stack degli operandi il risultato dell'operazione
- Quando non ci sono più token, nello stack ci deve essere solamente un operando → il risultato!

38

…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;
}
```

39

…massima fattorizzazione

- Valutazione di un'operazione

```
OperandType evaluate(OperandType operand1,
                     OperandType operand2, SymbolType op)
{
    switch (op)
    {
    case OperatorPlus:
        return doAdd(operand1, operand2);
    case OperatorMinus:
        return  doSub(operand1, operand2);
    case OperatorMul:
        return doMul(operand1, operand2);
    case OperatorDiv:
        return doDiv(operand1, operand2);
    default:
        assert(false);  //Operatore non valido
        return 0;
    }
}
```

40

Valutazione!

```
boolean postfixEval(char expr[], OperandType *value)
{
    boolean result = false, noError = true;
    OperandStack evalStack = newOperandStack();
    ExpressionLexer *lexer = newExpressionLexer(expr);
    while (readNextTokenFromLex(lexer) && noError)
    {
        SymbolType type = getCurrentTokenType(lexer);
        switch (type)
        {
        case Operand:
            {
                Token t;
                OperandType operand;
                getCurrentTokenFromLex(lexer, t);
                operand = convertToOperand(t);
                pushOperand(evalStack, operand);
                break;
            }//continua…
```

Gli operandi finiscono sempre nello stack

41

Valutazione!

```
    case OperatorPlus:
    case OperatorMinus:
    case OperatorMul:
    case OperatorDiv:
        {
            OperandType operand1, operand2;
            noError =
                getOperands(evalStack, &operand1, &operand2);
            if (noError)
            {
                pushOperand(evalStack,
                       evaluate(operand1, operand2, type));
            }
            break;
        }
    default:
        assert(false); break;
    }
} //…continua…
```

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

42

Valutazione!

```
    //…continua
    if (noError && !isEmptyOperandStack(evalStack))
    {
        *value = popOperand(evalStack);
        result = isEmptyOperandStack(evalStack);
    }
    destroyOperandStack(evalStack);
    destroyExpressionLexer(lexer);
    return result;
}
```

Al termine della valutazione, nello stack deve rimanere il risultato

43

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…
  - ...ogni token viene indirizzato nel "giusto" binario.
  - Utilizza un solo stack, questa volta per gli operatori!

44

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!

45

Fattorizzazione – 1

- Appendere un token numerico all'espressione d'uscita

```
void appendToken(char expr[], Token t)
{
    int len = strlen(expr);
    if (len > 0)
    {
        strcat(expr, " ");
    }
    strcat(expr, t);
}
```

46

Fattorizzazione – 2

- Appendere un operatore all'espressione d'uscita

```
void appendOperator(char expr[], SymbolType symbol)
{
    int len = strlen(expr);
    if (len != 0)
    {
        expr[len++] = ' ';
    }
    switch (symbol)
    {
    case OperatorPlus: expr[len] = '+'; break;
    case OperatorMinus: expr[len] = '-'; break;
    case OperatorMul: expr[len] = '*'; break;
    case OperatorDiv: expr[len] = '/'; break;
    default: assert(false);  //Errore!
        break;
    }
    expr[++len] = '\0';
}
```

È giusto che il modulo di valutazione/traduzione decida la rappresentazione degli operatori?

47

Fattorizzazione – 3

- È un operatore?

```
boolean isOperator(SymbolType symbol)
{
    return symbol == OperatorPlus || symbol == OperatorMinus ||
   symbol == OperatorMul || symbol == OperatorDiv;
}
```

- Ottenere la priorità di un operatore

```
int getOperatorPriority(SymbolType op)
{
    switch (op)
    {
    case OperatorPlus:
    case OperatorMinus:
        return 1;
    case OperatorMul:
    case OperatorDiv:
        return 2;
    default:
        assert(false);  //Operatore non valido
        return 0;
    }
}
```

È giusto che il modulo di valutazione/traduzione decida la priorità degli operatori?

48

Fattorizzazione – 4

- Comparazione priorità operatori

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

È giusto che il modulo di valutazione/traduzione sappia che un valore intero alto corrisponde ad una priorità alta?

49

Shunting Yard (semplificato) – 1

```
boolean infixToPostfix(char inExpr[], char outExpr[])
{
    boolean result, noError = true;
    SymbolStack symStack = newSymbolStack();
    ExpressionLexer *lexer = newExpressionLexer(inExpr);
    Token t;
    outExpr[0] = '\0';

    while (noError && readNextTokenFromLex(lexer))
    {
        switch (getCurrentTokenType(lexer))
        {
        case Operator:
            {
                getCurrentTokenFromLex(lexer, t);
                appendToken(outExpr, t);
                break;
            }
\\continua...
```

Se è un numero finisce direttamente sull'output

50

Shunting Yard (semplificato) – 2

```
        case OperatorPlus:
        case OperatorMinus:
        case OperatorMul:
        case OperatorDiv:
        {
            SymbolType symbol =
                      getCurrentTokenType(lexer);
            while (!isEmptySymbolStack(symStack) &&
                isOperator(peekSymbol(symStack)) &&
                      compareOpPriority(symbol,
                            peekSymbol(symStack)) <= 0)
            {
                SymbolType s = popSymbol(symStack);
                appendOperator(outExpr, s);
            }
            pushSymbol(symbol, symStack);
            break;
        }
```

Se è un operatore, si mettono in output tutti gli operatori contenuti nello stack che sono meno prioritari dell'op. corrente, poi si mette l'op. corrente nello stack

Shunting Yard (semplificato) – 3

```
    case ParOpen:
        {
            SymbolType symbol = getCurrentTokenType(lexer);
            pushSymbol(symbol, symStack);
            break;
        }
    case ParClose:
        {
            SymbolType s;
            if (!isEmptySymbolStack(symStack))
            {
                s= popSymbol(symStack);
                while (noError && s != ParOpen)
                {
                    appendOperator(outExpr, s);
                    noError = !isEmptySymbolStack(symStack);
                    if (noError) s = popSymbol(symStack);
                }
            }
            break;
        }
    }    //switch
} //while
```

Parentesi aperta → sullo stack

Parentesi chiusa → in output il contenuto dello stack fino alla parentesi aperta

Shunting Yard (semplificato) – 3

```
    while (noError && !isEmptySymbolStack(symStack))
    {
        SymbolType op = popSymbol(symStack);
        noError = isOperator(op);
        appendOperator(outExpr, op);
    }
    result = noError && isEmptySymbolStack(symStack);
    destroySymbolStack(symStack);
    destroyExpressionLexer(lexer);
    return result;
}
```

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

53

A proposito di fattorizzazione

- È veramente tutto ben fattorizzato?
- C'è qualcosa di poco pulito...
- La rappresentazione degli operatori e la definizione delle loro priorità è sparsa all'interno di più moduli (lexer/valutatore/traduttore)
- Si può runire in un unico modulo che contenga l'ennesimo ADT?
- Come?

54

ADT Operator

- Quali operazioni?
  - Classificazione dell'operatore partendo da un *token*
    - `SymbolType convertFromToken(Token t);`
  - Conversione dell'operatore in un *token*
    - `void convertToToken(SymbolType operator, Token t);`
  - Comparazione della priorità
    - `int compareOpPriority(SymbolType op1, SymbolType op2);`

55

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!

56

Deallocazione!

- Qualsiasi cosa succeda:
  - Se c'è qualche problema nel codice (*bug*) falliscono le **`assert`** e il processo termina
    - il sistema operativo recupera lo spazio di indirizzamento del processo → deallocazione "automatica"
  - Se c'è un errore nell'espressione (es. parentesi non bilanciate)
    - l'algoritmo termina con insuccesso ed effettua correttamente la deallocazione

57

Modularità

- Il tokenizer è un componente configurabile e completamente riusabile
- Il lexer è riusabile (insieme al tokenizer) all'interno di algoritmi che analizzano espressioni
- Gli algoritmi di valutazione/traduzione sono riusabili all'interno di applicazioni più complesse

58

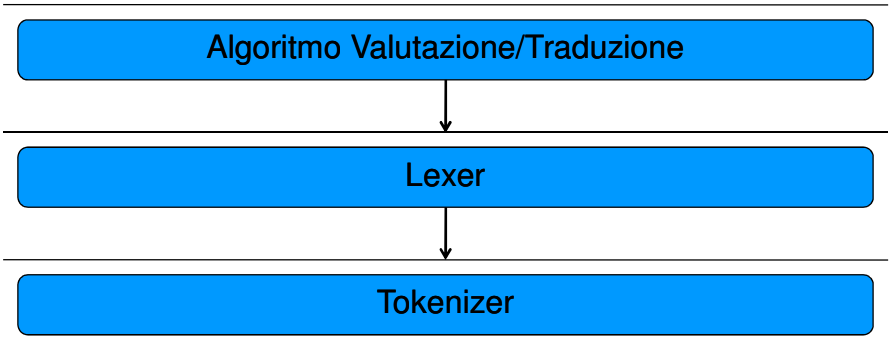
Programmazione a layer

- Ogni livello conosce direttamente solo il livello sottostante
  - Il tokenizer non conosce “nessuno”, in particolare non conosce il lexer che sta al livello superiore
  - Gli algoritmi di valutazione/traduzione conoscono direttamente solo il lexer e non il tokenizer
- Riusabilità a livelli

59

Programmazione a layer

Ogni layer conosce solo quello direttamente sottostante...



60

Programmazione a layer

- Il tokenizer è attualmente "condannato" a leggere i dati da una stringa
- Come astrarre la sorgente dati in modo che possa essere una stringa o un file o qualcos'altro?
  - Si tratta di aggiungere un layer ulteriore...
- Effetto collaterale (molto) positivo: con questa estensione sarebbe possibile usare il tokenizer per scandire il contenuto di file di testo!!!

61

Uno stream?

- Si pensi di modellare la sorgente di caratteri con un nuovo ADT: **`CharStream`**
- Operazioni:
  - Lettura/caricamento del carattere successivo; restituisce successo finché c'è qualcosa da leggere:
    - `boolean readNextChar(CharStream*);`
  - Recupero del carattere corrente: restituisce qualcosa di significativo se l'ultima invocazione a **`readNextChar`** ha avuto successo:
    - `char getCurrentChar(CharStream*);`
- Ipotesi di implementazione di CharStream basato su stringa?
- Ipotesi di implementazione di CharStream basato su file di testo?

62