02393 Programming in C++ Module 12
Trees (Continued) – Novel C++ Features

Sebastian Mödersheim

November 21, 2016

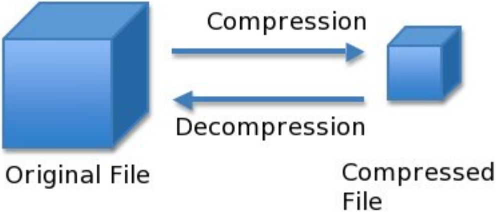
Lecture Plan

#	Date	Topic
1	29.8.	Introduction
2	5.9.	Basic C++
3	12.9.	Data Types, Pointers
4	19.9.	
5	26.9.	Libraries and Interfaces; Containers
6	3.10.	Classes and Objects I
7	10.10.	Classes and Objects II
		Efterårsferie
8	24.10.	Classes and Objects III
9	31.10.	Recursive Programming
10	7.11.	Lists
11	14.11.	Trees
12	21.11.	Trees (Cont.) Novel C++ features
13	28.11.	Summary
	5.12.	Exam

Exam

- The exam will be on the 5th of December – time and location to be announced by the student administration.
- Exam is electronic – bring your laptop with your prefered installation of tools.
- If there are technical difficulties (but please only then!), you may also submit written answers on paper.
- You are given instructions and given files electronically.
- Submission is via Campusnet.
  - ★ The files you are supposed to submit must be named $Z - library.cpp$ where Z is the exercise number.
  - ★ Each file submitted must be submitted individually (not a zip!).
- We will set up also CodeJudge, but only to help you test your code, this is not the actual submission!
  - ★ This is experimental, technical problems may occur. Do not panic if CodeJudge refuses your answer.
  - ★ When correcting the exam I will look at your answers.
  - ★ Also programs with errors can get some points!

Lossless Compression



- Shannon/Information Theory: how much information is contained in a file? What is redundant?
- Huffman's idea: we can use the fact that some symbols may occur more frequently than others.
  Example: in normal text, 'e' is more frequent than 'Y'.
- Encode the frequent ones with fewer symbols

Frequency Table

Example

Symbol	ASCII 7-bit	Frequency	Huffman Encoding
<space>	00100000	25	0
e	01100101	20	11
E	01000101	7	100
W	...	5	1011
Q	...	2	10101
Y	...	1	10100

Huffman encoding: frequent symbols have smaller code.

Question: how to get from given frequencies to a huffman code?

Frequency Table

Example

Symbol	ASCII 7-bit	Frequency	Huffman Encoding
<space>	00100000	25	0
e	01100101	20	11
E	01000101	7	100
W	...	5	1011
Q	...	2	10101
Y	...	1	10100

Huffman encoding: frequent symbols have smaller code.

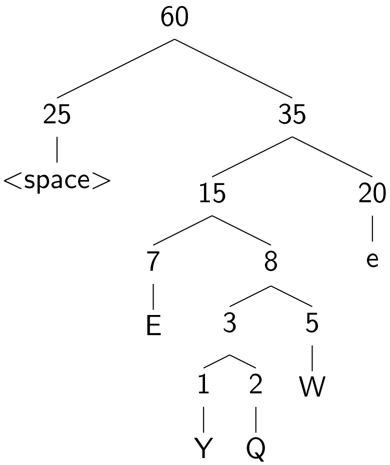
Question: how to get from given frequencies to a huffman code?

Prefix Code

The encoding of a character must never be a prefix of the encoding of another character.

Prefix Codes

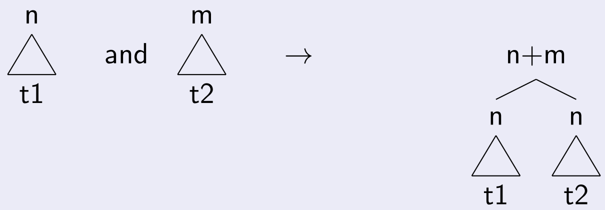
A prefix code can be regarded as a tree (0=left, 1=right):



E.g. encoding of Y is 10100 because from root node the leaf Y is reachded by: right-left-right-left-left.

How to get a prefix-code tree from a frequency table

1. Every symbol is a single leaf node with a frequency.
2. Order this list of trees by their frequencies.
3. Combine the lowest two trees into one new binary tree; root frequency is the sum of the two subtree frequencies:



4. Insert this new tree into the sorted list
5. Continue with step (3) if more than one tree left.

Animation:

http://www.cs.pitt.edu/ kirk/cs1501/animations/Huffman.html

Front-end of an Interpreter or Compiler.

Example (Source Code)

```
result = /* z-2 */ x + 2 * y;
```

Lexer/Scanner—Tool "Lex"

Remove comments and whitespaces, split input string into Tokens.
No analysis of the structure yet.

Example (Token stream)

identifier `result`, operator=, identifier `x`, operator +, constant 2, ...

Front-end of an Interpreter or Compiler.

Example (Token stream)

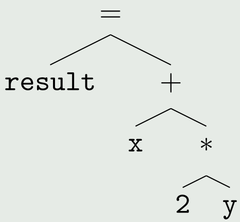
`result = /* z-2 */ x + 2 * y;`

identifier `result`, operator=, identifier `x`, operator +, constant 2, . . .

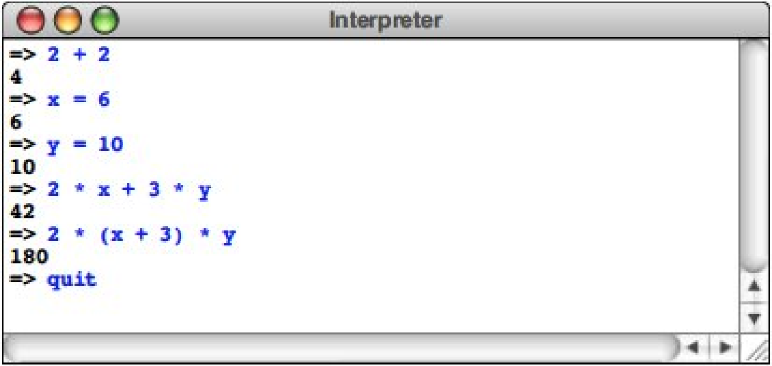
Parser—Tool “yacc” or “bison”

Identify the structure of the text into a form of tree. This does not include any analysis like type-correctness or definedness.

Example



Interpreter from the Stanford Reader



- Allows to read integer expressions
- Result can be assigned to variables
- Variables can be used in subsequent expressions

Expressions

Definition

An expression (for our interpreter) is any of the following:

- An integer constant
- A variable name
- An expression enclosed in parentheses
- A sequence of two expressions separated by an operator

This a typical example of an inductive definition:

- The first two rules give the basis: simple-most expressions
- The second two rules are the inductive step: they say how to form larger expressions from existing ones.
- Anything that cannot be constructed using these rules is not an expression.

Example: $y = 3 * (x + 1)$

Grammar

Definition

An expression (for our interpreter) is any of the following:

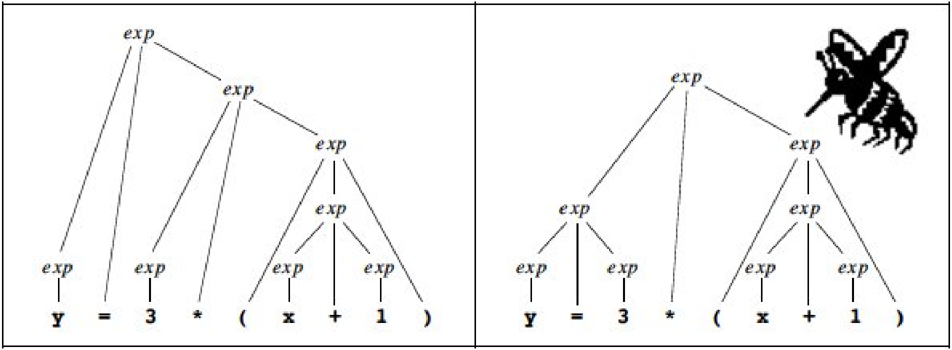
- An integer constant
- A variable name
- An expression enclosed in parentheses
- A sequence of two expressions separated by an operator

This is often expressed as a context-free grammar:

$$\mathcal{E} ::= \mathit{Const} \mid \mathit{Var} \mid (\mathcal{E}) \mid \mathcal{E}\ \mathit{op}\ \mathcal{E}$$

Tools like yacc can automatically generate the front-end of an interpreter/compiler using the grammar.

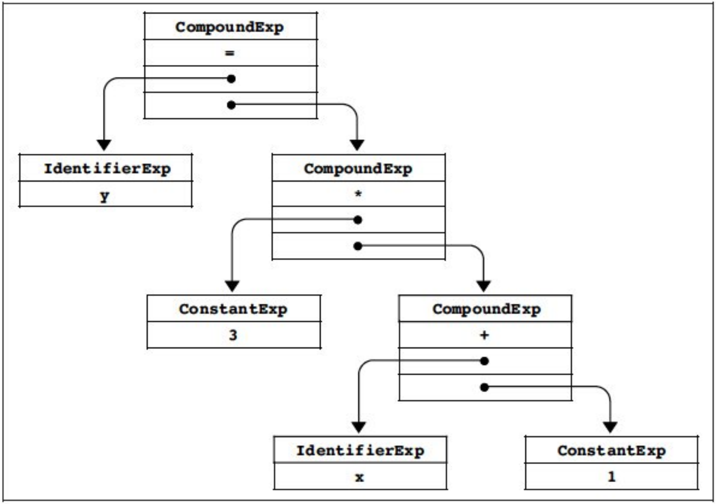
Parsing and Ambiguity



- Could be avoided for this case by restricted syntax.
- Similar ambiguities for $3 + 5 * 7$ (operator precedence).
- Tools like yacc will report conflicts between “shifts and reduce operations”.

Parse Tree

"Eval" directly evaluates expressions, but in general the parser needs to produce a parse tree. What is an appropriate data type for nodes of this tree?



Parse Tree Node: Using OO and Inheritance

```
class Expression {
public:
  Expression();
  virtual ~Expression();
  virtual int eval(Map<int> & state) = 0;
  virtual string toString() = 0;
  virtual expTypeT type() = 0;
};
```

- The different kinds of nodes/expressions (constant, variable, compound expression) are defined as subclasses of this class.
- This class is the "super-term" that can be used whenever we want to talk about *any* kind of node.
- This class contains purely virtual functions.

Parse Tree Node: Using OO and Inheritance

```
class ConstantExp: public Expression {
public:
  ConstantExp(int val);
  virtual int eval(Map<int> & state);
  virtual string toString();
  virtual expTypeT type();
  int getValue();
private:
  int value;
};
```

`ConstantExp` inherits everything from `Expression` (publicly).

Disclaimer

The following slides (also next week) contain material

- that is part of modern C++ standards (C++11 and C++14) and may not be supported by every compiler.
- will not be asked about in the exam, but the programming may be a good exercise for it!

Smart Pointers

- Problem: when a function returns a pointer to a data structure
  - ★ who owns it?
  - ★ who is responsible to delete it?
- Idea: encapsulate pointers into a special class that handles the pointer like an abstract data type and who makes clear who the owner is:
  - ★ `unique_ptr`: such pointer has only one owner
    - One cannot make copies of it. (Compiler error!)
    - Operation **release** (also when going out of scope) causes the deletion of the pointed data structure.
  - ★ `shared_ptr`: such pointer can have multiple owners
    - It is allowed to make copies.
    - The shared pointer class keeps a reference counter (number of owners).
    - Deleted when all owners have released it.

Live Programming

- We now implement a smart pointer similar to shared pointer, and test it with the `set.cpp` example from module 10.
- Automatic memory management – we can forget about `delete`?!

Live Programming

- We now implement a smart pointer similar to shared pointer, and test it with the `set.cpp` example from module 10.
- Automatic memory management – we can forget about `delete`?!
- No actually: there are some pitfalls, e.g.

  ```
  head = head->next
  ```

  can lead to double delete if there is no other reference to `head->next`. Why?