Title : README, Implementation of Code Generation via the Visitor Design Pattern

Author : Roy Ivey

Table of Contents

1 Goal:
Output an assembly code file based off an input file written in a subset of Turing.

2 Setup:

1. Install flex
2. Install bison
3. Run make

3 Usage:
Call make then, call the following:

```
./turing [input file] [output file]
```

4 Implementation:
The Visitor design pattern has been implemented to remove complicated and repeated decision logic associated with generating assembly code. The Visitor pattern allows for the inherently complex procedure of compilation to be tamed and carefully orchestrated. The amount of control and precision required to achieve this adds to the elegance of this design-pattern's implementation. The Visitor pattern addresses the problems that follow, each problem is paired with why the pattern simplifies the solution.

4a Problems:
1. Many Node Types implies Large switch case or long if-elseif chains
    - The preceding decision structures are removed and delegated to the double-dispatch model inherent to the Visitor pattern. Double Dispatch is described in Detail on page 3.
2. Scope changes
    - When a new scope is entered no decision is required because a Visitor simply navigates through the nested tree structure.

5 Special Considerations:
During development there were several design decisions that eased issues that arose while integrating the Visitor pattern with legacy code.

1. Backwards Compatibility with Symbol Table generation from folder P3
    a. Concrete Elements Wrap the initial NodeElement Type (tree.h)
        i. Concrete Elements inherit from NodeElement(the original Node structure)
        ii. The copy constructor of concrete Elements consume the more general NodeElement as Bison parses the codefile (mini-turing.y).
        iii. This allows the original NodeElement creation algorithm to be used with the new Visitor design.

        iv. This also allows the type checking from the P3 folder to process, unimpeded by the new design.

2. Address the if-elsif-else-structure’s forward reference problem
    a. Push dataspace memory address Labels onto the stack.
    b. Generate relevant pointers in the dataspace with previously assigned labels.
    c. Keep track of Codespace Byte-index. (NodeVisitor.cpp)
    d. Keep track of Dataspace Byte-index. (NodeVisitor.cpp)
    e. Output Dataspace before CodeSpace in output.

6 Program Limitations:

1. The code generation part of the program assumes the symbol table and parse tree are made correctly.
    a. In reality, the symbol table construction algorithm portion of this program (P3 folder) is imperfect.
    b. Given a, any declaration of a variable in a child scope will result in a segfault while parsing.
2. Variable Label and Scope Label namespace intersection
    a. There will be problems if capital letters are used to name a variable, since Capital letters are used to denote scope endings.
    b. Attempts to use 2 character representations of the labels proved troublesome, since the implication of a 2 character label in the bytecode could not to be determined by the debugger or implementor.
3. Ands, Ors, and unary(-)
    a. Ands and Ors are victims of an imperfect language definition which results in syntax errors when being used.
    b. No Support was implemented for unary(-)
4. Only Integers and Boolean Types supported in code generation

7 Potential Improvements:

After working with the problem set extensively, it is clear that a Composite-Visitor design pattern solution would have been ideal for creating a one-pass compiler implementation. Refactoring the program would produce an even more elegant solution to problem at hand.

8 Detail:

1. Double Dispatch is achieved by the following:
    a. Define an abstract class Element with a pure virtual function with the signature of: void accept(Visitor * v)
    b. Implement the Element interface in a concrete class for each important node, where an important node is one where code generation is needed. (tree.h)
    c. Define an abstract class Visitor with pure virtual functions for each implementation of Element which overload the signature of : void visit(*Element-Type* e) (NodeVisitor.h)
    d. Each concrete Element’s accept(Visitor * v) should call v->visit(*this) (tree.cpp)

e. Each concrete Visitors' overloads visit(*Element-Type* e) should implement the necessary steps for code generation associated with the Element-Type. (NodeVisitor.cpp)

9 Relevant File Manifest:

General:
- Makefile

Language and parsing definition files (P1, P2):
- mini-turing.lex
- mini-turing.y
- y.tab.h
- y.tab.c

NodeElement Definitions and Creation (P2 - P4):
- tree.h
- tree.cpp

Symbol, SymbolTable, and Symbol Table Construction (P3)
- Symbol.h

Abstract Visitor, NodeVisitor implementation (P4)
- NodeVisitor.cpp
- NodeVisitor.h

Main function (P4)
- Sym_tab.c