EECS4314 week 4

Jerry Wu

2024-01-31

Contents

Section 1

Architecture of a compiler

1.1 Abstract

- The architecture of a system can change in response to improvements in technology
- This can be seen in the way we think about compilers

1.2 Early compiler architectures (review)

1.2.1 The overall design

- In the 1970s, compilation of code was regarded as a purely sequential (batch sequential or pipeline) process.
- Text (high level programming language) gets fed into a machine that performs some operations and converts it to machine code in the following order of operations:
  - Lexical (keywords, names, etc.)
  - Syntax (brackets, parentheses, etc)
  - Semantics (variable initializations, function definitions, etc.)
  - Optimizations (improve speed, efficiency, detection of dead code, etc.)
  - Code generation (generate the assembly, bytecode, etc)

1.2.2 Problems with this design, and subsequent solition

- If one of the later steps (like optimization) wants lexical data of the compilation, it would have to go through all previous pipes in order to access it
- If we want to solve this, we can **link each filter to a separate symbol table** during lexical analysis so that any filter in the sequence can access it, i.e. optimizations wants to access data from the syntactical analysis filter step.
- In the mid 1980s, increasing attention turned to the intermediate representation of the program during compilation. This was when each filter step in the compiler was also linked to an **attributed parse tree** along with a symbol table.

1.3 Hybrid compiler architetures

- The new view accomodates various tools (e.g. syntax directed editors like vs-code) that operate on an internal representation rather than the textual form of a program
- Architectural shift to a repository style with elements of a pipeline style since the order of the execution of the process is still predetermined.
- This allows for the compiler to continuously run while code is being edited for live updates, since the parse tree and symbol table are both centralized along with edit and flow.

Section 2

Implicit invocation style

2.1 The main idea

- This style is suitable for applications that involve **loosely coupled collections of components**, each of which carries out some operation and may in the process enable other operations
- It's particularly useful for applications that must be reconfigured on the fly:
  - Changing service provider
  - Enabling or disabling features/capabilities
- Subscribers connect to publishers directly (or through a network)
- Components communicate using the event bus, not directly to each other.

2.1.1 Publish-subscribe

- Subscribers register to receive specific messages
- Publishers manage a subscription list and broadcast messages to subscribers

2.1.2 Event based

- Independent components asynchronously emit "events" communicated over an event bus/medium.

2.1.3 Components and connectors

- **Components**
  - Publishers, subscribers
  - Event generators and consumers
- **Connectors**
  - Procedure calls
  - Event bus

2.1.4 Examples of implicit invocation

- Used in **programming environments** to integrate tools
  - Debugger stops at a breakpoint and makes an announcement
  - Editors scroll to the appropriate source line and highlights it
- X, youtube, etc.

2.1.5 QA evaluation of implicit invocation

- **Performance**
  - Publish and subscribe: can it deliver thousands of messages?
  - Event based: how does it compare to repository style?
- **Availability** - Publisher needs to be replicated
- **Scalability** - Can it support thousands of users, growth in data size?
- **Modifiability** - Easily add more subscribers, chanhges in message formats affects many subscribers

2.2 Advantages of implicit invocation

- For piublish and subscribe: efficient dissemination of one way information
- Provides strong support for reuse: any component can be added by registering/subscribing for events
- Eases system evolution: components may be replaced without affecting other components in the system

2.3 Disadvantages of implicit invocation

- For publish and subscribe: needs special protocols when the number of subscribers becomes very large
- When a component announces an event:
  - It has no idea how other components will respond to the event
  - It cannot rely on the order in which the responses are invoked
  - It cannot know when responses are finished

Section 3

Layered style architecture

3.1 Overview

We can think of it like a cake with different layers as the name suggests. It is designed so that programs/functions in one layer can obtain services from a layer below it. We can break down the layers into 3 fundamental layers, but there can be more if necessary:

- High level layer (apps)
- Middle layer (drivers)
- Low level layer (hardware)

3.1.1 Specializations of layered style

- Often exceptions are made to permit non adjacent layers to communicate directly (usually done for efficiency so we can rely on abstraction)

3.1.2 Examples of layered style

- Layered communication protocols
  - Each layer provides a substrate for communication at some level of abstraction
  - Lower levels define lower levels of interactions, with the lowest level being the hardware (physical layer)
- Unix layered architecture (one layer can contain multiple adjacent blocks, i.e. files, sockets, cooked block, raw block, etc.)

3.2 Advantages of layered style

- **Design**: based on increasing levels of abstraction
- **Enhancement**: since changes to the function of one layer affects at most two other layers
- **Reuse**: since different implementations (with identical interfaces) of the same layer can be used interchangably

3.3 Disadvantages

- Not all systems are easily structures in a layered fashion
- Performance requirements may force the **coupling of high level functions to their lower level implementations**

Section 4

Client-server style

4.1 Overview

This style is suitable for applications that involve distributed data and processing across a range of components

- **Components**:
  - **Servers**: Standalone components that provide specific services such as printing, data management, etc.
- **Connector**: The network which allows clients to access remote servers

4.1.1 Examples of client server style

- File servers
  - Primitive form of data service (not a database)
  - Useful for sharing files across a network
  - Client passes request for files over the network to the file server
- Databases
  - More efficient in distributing data and is more powerful than file servers
  - Client passes SQL requests as messages to the DB server, to which results are returned
  - Query processing is done by the server
  - No need for large data transfers
  - Transaction DB servers are also available

4.2 Advantages of client server style

- **Distribution** of data is straightforward
- **Transparency** of location
- **Mix and match** heterogeneous platforms
- **Easy to add** new servers or upgrade existing servers

4.3 Disadvantages of client server style

- **No central register of names and services** - it might be hard to find out what services are available

Section 5

Process-control style

5.1 Overview

This style is suitable for applications whose purpose is to maintain specified properties of the outputs of the process at (sufficiently near) given reference values.

- **Components**
  - Process definition includes mechanisms for manipulating some process variables
  - Control algorithm for deciding how to manipulate and process variables
- **Connectors**: are the data flow relations for:
  - **Process variables**:
    - *Controlled variable* whose value the system is intended to control
    - *Input variable* that measures an input in the process
    - *Manipulated variable* whose value can be changed by the controller
  - **Set point** is the desired value for a controlled variable
  - **Sensors** to obtain the values of process variables pertinent to control

5.1.1 Examples of process control style

- Automobile anti lock brakes
- Nuclear power plants
- Automobile cruise control

Section 6

Exercise

6.1 Automated stock trading system

A customer approaches ABC Trading Solutions with the need for an architecture design for an automated stock trading system

- The system needs to take in a list of stocks related to specific sectors and buy/sell these stocks based on some predefined algorithms
- The system needs to perform well (place many orders) and scale to support many investors

Propose an architecture for the system accompanied by an informal evaluation of the advantages and disadvantages of the proposed architecture.

6.1.1 Some notes

- We will need **client server** for sure since we need to communicate with the market in order to buy and sell stocks in the first place
- **Implicit invocation** can be used for a feed subscription service to notify end users about when a stock changes price or reaches a certain threshold or when profits are optimal, etc.