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 vscode) 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 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

2.1.5 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

2.1.6 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.

2.1.7 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