Sequence 1.3 – Anatomy of a Compiler

P. de Oliveira Castro S. Tardieu

Anatomy of a Compiler

- A compiler takes as input a *source language* and produces an *executable* program as a binary machine assembly file.

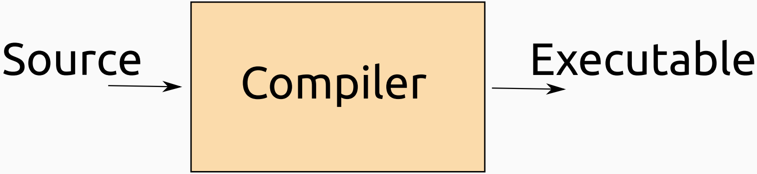


Figure 1: Source to Executable

Multiplicity of Source and Executable Languages

- How to translate from multiple source languages to multiple executable formats?
- Writing nine full compilers is intractable!

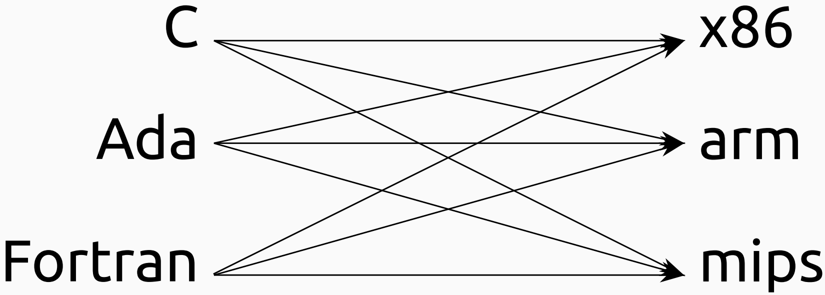


Figure 2: 9 full compilers ?

Intermediate Representation (IR)

- Introduce an *intermediate representation* to decouple the translation
- The IR is a neutral language that is agnostic both to the source language and to the executable format.

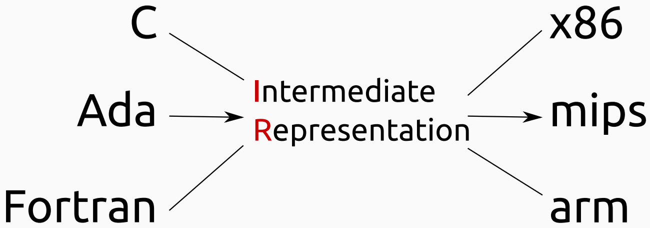


Figure 3: Intermediate Representation

Simplified Architecture of a Modern Compiler

- The IR breaks the translation into small self-contained steps:
  - More maintainable compiler
  - Each input language requires writing a single frontend
  - Each output executable format requires writing a single frontend
- Many optimization passes can be written as transformations from IR to IR

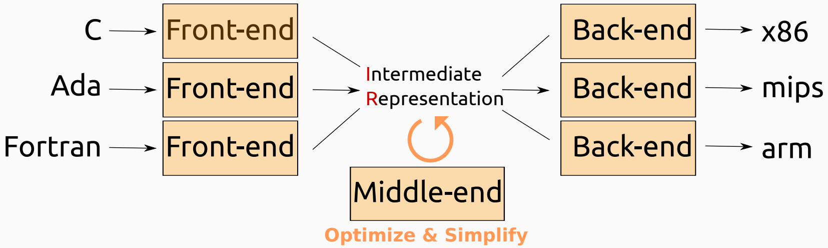
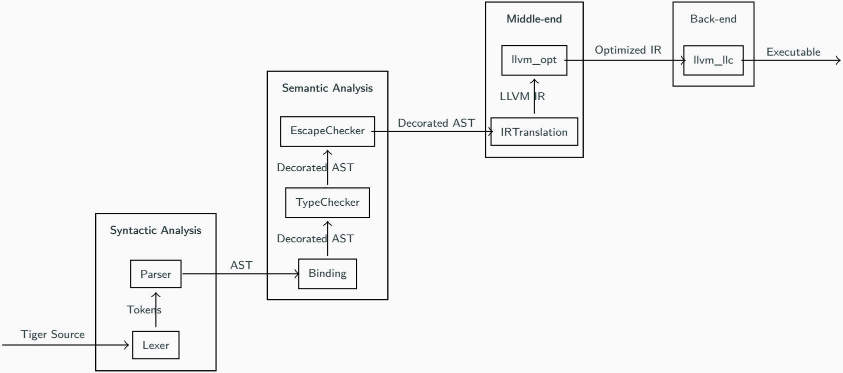


Figure 4: Architecture of a Modern Compiler

This course Compiler Architecture: The big picture

- Our compiler is going to have four steps:
    - Front-end: Syntactic and Semantic Analysis
    - Middle-end
    - Back-end



The Front-end: Syntactic Analysis

- The *Lexer* breaks the program into tokens such as “a”, “:=”, “1”, “+”, “2”
- The *Parser* analyses the grammar according to Tiger’s grammar rules. It produces an *Abstract Syntax Tree (AST)*.

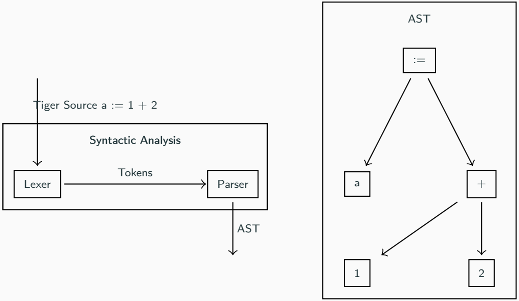


Figure 6: Syntactic Analysis and AST

The Front-end: Semantic Analysis

- Then the AST is analysed and decorated through multiple passes,
  - *Binding* pass, finds each variable or function and links it to its declaration
  - *TypeChecker* pass, checks that all the operations are correctly typed. Eg: `5 + "hello"` is illegal in Tiger
  - *EscapeChecker* pass, finds access from a nested function to variables defined in the containing outer function.

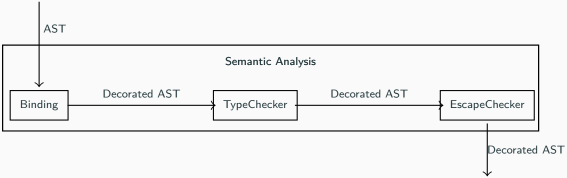


Figure 7: Semantic Analysis

The Middle-end

- *IRTranslation* transforms a decorated AST into LLVM Intermediate Representation
- *opt* is the LLVM IR optimization driver

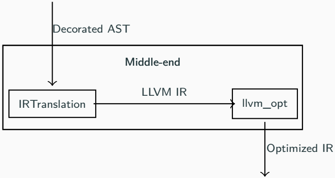


Figure 8: Middle-end

The Back-end

- *llc* is the LLVM static compiler: it takes LLVM IR and produces assembly code
- *llc* has different back-ends depending on the target architecture

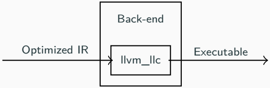


Figure 9: Back-end