Sequence 1.3 – Anatomy of a compiler

P. de Oliveira Castro S. Tardieu

Anatomy of a compiler

- A compiler translates a high-level program (the *source code*) into assembly mnemonics.

Source to assembly

Multiplicity of source and executable languages

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

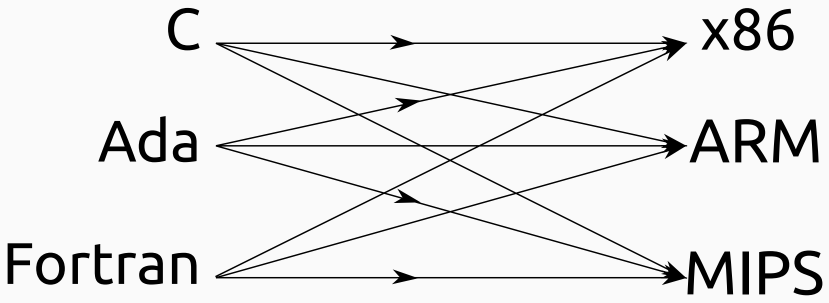


Figure 1: 9 full compilers?

Intermediate representation

- Introduce an *intermediate representation* (IR) to decouple the source language from the target.
- The IR is a neutral language that is indifferent to both the source language and the executable format.

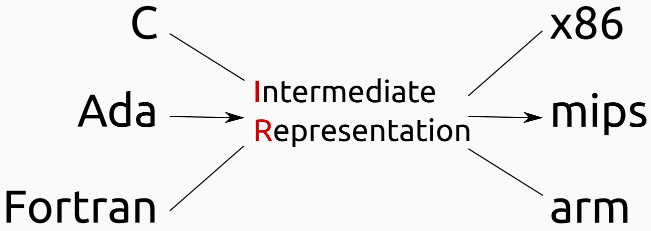


Figure 2: Intermediate representation

Simplified architecture of a modern compiler

- The IR breaks the translation into small self-contained steps, bringing:
  - a more maintainable compiler;
  - the need for a single front-end per input language;
  - the need for a single back-end per target architecture.
- Many optimization passes can be written as transformations from IR to IR, benefiting every language and target.

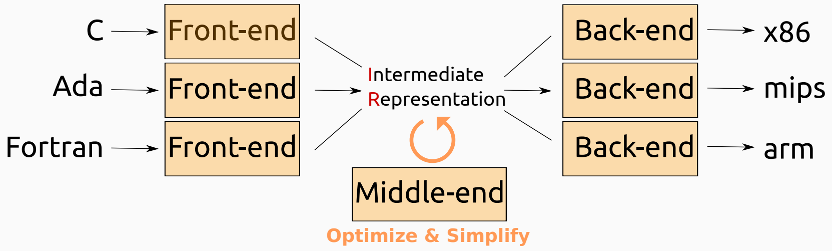


Figure 3: Architecture of a modern compiler

This course compiler architecture: the big picture

- Our compiler is going to have three steps:
  - Front-end: syntactic and semantic analyses, translation to IR;
  - Middle-end: work on the IR (optimizations);
  - Back-end: assembly code generation from the IR.

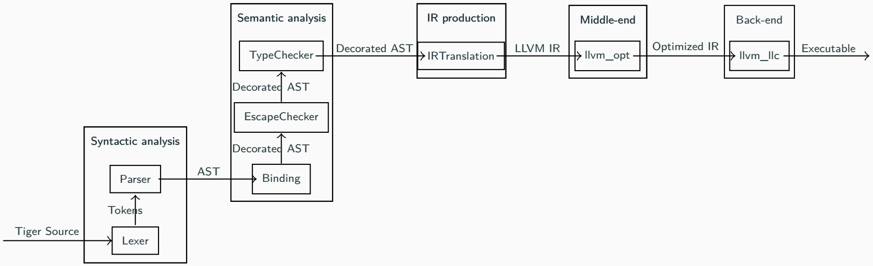
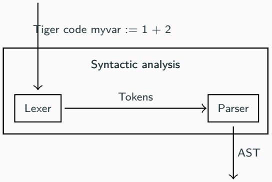


Figure 4: Tiger compiler

The front-end: syntactic analysis

- The *lexer* breaks the Tiger code “`myvar := 1 + 2`” into tokens such as “`myvar`”, “`:=`”, “`1`”, “`+`”, “`2`”.
- The *parser* analyses the grammar according to the grammar rules of Tiger. It produces an *abstract syntax tree (AST)*.



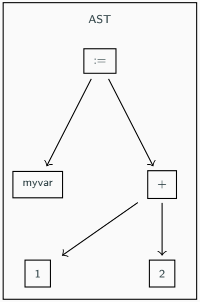


Figure 5: Syntactic analysis and AST

The front-end: semantic analysis

- Afterwards, the AST is analyzed and decorated through multiple passes:
  - The *binding* pass looks up and associates each variable or function with its declaration.
  - The *escape checker* pass records accesses to variables from another function.
  - The *type checker* pass checks that all the operations are correctly typed. For example `5 + "hello"` is illegal in Tiger.

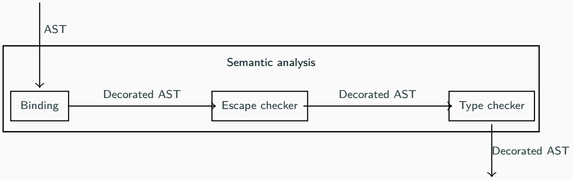


Figure 6: Semantic analysis

The front-end: IR generation

- The IR generation pass generates IR (intermediate representation) from the decorated AST.
- The AST is specific to the source language (Tiger).
- The IR is independent of the source language.

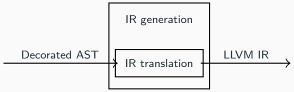


Figure 7: IR generation

The middle-end

- The IR being language and target agnostic, optimizations on the IR are generic.
- This phase is optional: the optimized IR will exhibit the same behaviour as the input IR (for a correct input).
- We will not have to implement this step: LLVM already provides an IR optimizer.

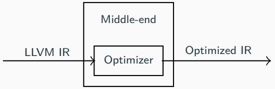


Figure 8: Middle-end

The back-end

- The back-end translates the IR into assembly code.
- LLVM provides back-ends for different architectures, which we will use in the project.

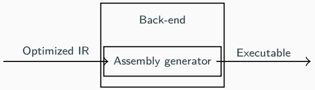


Figure 9: Back-end