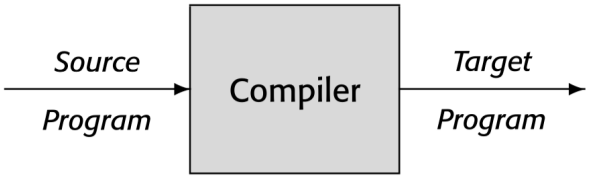
Overview of Compilation

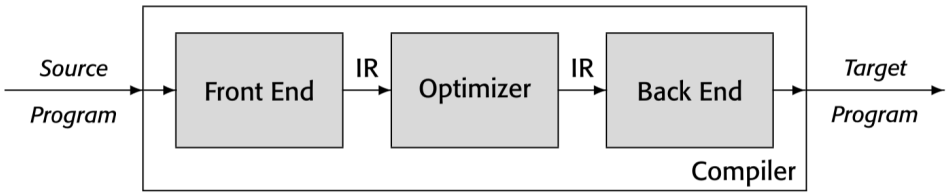
Compilers take as input a program written in some language and produce an equivalent program in another language (usually a language at a lower level of abstraction like machine code or the ISA of some processor).

- One reason this is important is that it allows us to abstract away complex tasks that would require hundreds or even thousands of lines of code. Instead, programers can write a couple of lines in a high-level language and let the compiler do rest.
  (For example, reading and writing to a file)
- Some compilers target a high-level language rather than translating the program to lower-level machine code/ISA. These are called source-to-source compilers



In contrast, an interpreter actually executes the source program it is given. However, both compilers and interpreters share many of the same phases.

In order for a compiler to accomplish this it needs to be able to understand the syntax (form/structure) and semantics (meaning) of both the source and target languages. It also needs a scheme for mapping instructions from the source to the target language that preserves the program's original meaning and improves it by some metric.

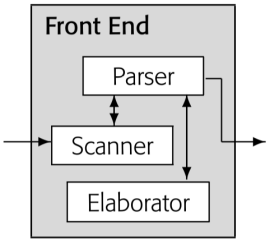


- The front-end deals with analyzing the source program's syntax and structure, determining if the source program is valid (well-written code), and building a model of the structure and meaning of the program to hand-off for translation
- The back-end uses the given model to translate the program into an equivalent program in the target language
- Connecting the front-end and back-end are intermediate representations (IR) that are usually independent from the source and target languages. They are abstract representations of the program
  - abstract syntax trees (AST)
- The optimizer is an IR-to-IR transformer that attempts to improve the IR program by making one or more passes analyzing and rewriting/restructuring the IR.

The optimizer may modify the IR in a way that may produce a faster program in the back-end. Speed is not the only improvement. The optimizer could aim to produce a smaller program or use less energy

The Front-end

In order for the front-end to determine the syntax and semantics of the source program it has to break down the actual text in the file.



First the scanner converts the stream of characters read from the source program file into a stream of words. The parser takes this stream of words and attempts to fit them to a grammer.

A grammar is a model of the syntax of a programming language (in this case a grammar for the source language). It describes a set of strings of words by defining a set of rules (or productions) over the words and syntactic variables (categories of words) in the language.

Grammar derivation example

"Compilers are engineered systems"

Scanner converts the input into a stream of classified words of the form (p, s)
p is the part of speech
s is the spelling

A real compiler may hash the string to make categorization/comparison faster instead of using the whole string like this

(noun, "Compilers"), (verb, "are"), (adjective, "engineered")
(noun, "systems"), (endmark, ".")

The parser attempts to find a derivation of the string of words using the productions of the grammar provided

1	*Sentence*	→	*Subject* verb *Object* endmark
2	*Subject*	→	noun
3	*Subject*	→	*Modifier* noun
4	*Object*	→	noun
5	*Object*	→	*Modifier* noun
6	*Modifier*	→	adjective

→

Rule	Prototype Sentence
—	*Sentence*
1	*Subject* verb *Object* endmark
2	noun verb *Object* endmark
5	noun verb *Modifier* noun endmark
6	noun verb adjective noun endmark

matches the stream of words provided by the scanner.

The sentence is grammatically correct

Generally, the parser calls the scanner as it needs words. It may also call on the elaborator to perform additional computations. These additional computations are called semantic elaborations and are necessary for detecting errors that go deeper than the grammar describes. This includes things like checking type consistancy, incorrect array references, procedure calls with wrong number of parameters, etc. The IR of the program is built up gradually alongside these computations.

To summarize, the parser determines if the input stream is a valid sentence in the source language.

The Optimizer

The optimizer takes the IR generated by the front-end, analyzes it, discovers facts about the context it will execute in, and uses that knowledge to rewrite the IR in a way that will produce more efficient code.

Efficiency could mean faster execution, a smaller generated program, reduction in the energy needed for a processor to run the program, and many other metrics.

Examples of optimization techniques include

- constant propogation
- common subexpression elimination
- loop invariant code motion
- global value numbering
- strength reduction
- scalar replacement of aggregates
- dead code elimination
- loop unrolling

The Back-end

The back-end is tasked with traversing the IR and generating an equivalent program on the target processor's ISA. To do this correctly and efficiently the back-end must solve three distinct problems

1) Instruction selection
2) Instruction scheduling
3) Register allocation

The first step is to rewrite IR operations to the operations provided by the target machine's ISA. One important detail about this step is that it uses unlimited virtual registers, relying instead on the subsequent register allocation step to map those virtual registers to real registers provided by the ISA in a way thats reasonable and efficient. This way the instruction selection can focus on being accurate and make optimizations based on the capabilities of a particular ISA (for example, some provide immediate add/multiply instruction that can be used to avoid using registers to store constants*. Another may be that addition executes faster than multiplications, making it preferable to implement multiplication as addition*)

*

```
load  r_arp, @a, r_a                load  r_arp, @a, r_a
load  2, r_2   <- maybe skip this ⇒  multI r_a, 2, r_3
mult  r_a, r_2, r_3
```

*

mult $r_a, 2, r_3$ $\Longrightarrow$ add r_a, r_a, r_3

addition may be faster in this case

The register allocator must map those virtual registers to actual registers provided by the ISA. It may rewrite the code to minimize register usage or take advantage of the properties of the existing code and its data flow. For example the register allocator could take advantage of when a value is already asigned to a register and reusing that. Or perhaps a computation is used repeatedly within a section of code and the register allocator elects to store that value in a register for reuse.

Finally, the instruction scheduler ensures that instructions are executed to improve runtime and maximize CPU usage. Some instructions require more cycles than others. What this means is that executing the instructions as-is may result in much stalling while waiting for costly instructions to execute. It's the job of the instruction scheduler to recognize how to order instructions in a way that preserves the programs original meaning but allows some instructions to be performed concurrently to minimize down time. Some instruction are dependant on the output of others and thus must wait out the cycles required, but sometimes it may be beneficial to perform some lengthy operation ahead of time, and simultaneously begin executing other non-dependant instructions on the side.