Compilation steps revisited

- Recap of steps summarized in intro:
- Read character sequence, preprocess
- Lexical analysis: produce token stream
- Syntactic analysis: produce syntax tree
- Semantic analysis: refine/augment syntax tree
- Intermediate generator: produce language independent form
- Machine-independent optimization: refine independent form
- Code generation: actual target code produced
- Machine-dependent optimization: optimize target code

Read (scan) character sequence

- Reads and preprocesses the stream of characters for the source program:
  - Standardize layout (e.g. replace all whitespace blocks with a single blank)
  - Apply supported macros to transform character sequence
  - Perform basic error checking (e.g. unsupported characters)
  - Read the resulting character sequence, divide them into recognized tokens, each with associated attributes (e.g. character sequence 324 might represent an INTEGER token with an attribute value of 324, the sequence foo might represent an IDENTIFIER token with an attribute value of “foo”)

Read (scan) character sequence

- Reads and preprocesses the stream of characters for the source program:
  - Standardize layout (e.g. replace all whitespace blocks with a single blank)
  - Apply supported macros to transform character sequence
  - Perform basic error checking (e.g. unsupported characters)
  -

Lexical analysis

- Transform the character stream into a sequence of recognized tokens, repeatedly:
  - read a sequence of character sequence until they can be recognized as a distinct unit (lexeme), assign a token type and any associated attributes
  - character sequence 324 might represent an INTEGER token with an attribute value of 324
  - character sequence foo might represent an IDENTIFIER token with an attribute value of “foo”
  - character sequence != might represent a NOTEQUALS token with no associated attributes

Syntactic analysis

- Read the token sequence from the lexical analysis, use the source language grammar rules to error check, build a syntax tree representing the structure of the program
- e.g. The original character sequence x = y ; might have produced a token sequence IDENT ASSIGN IDENT SEMICOL, from which the syntax analysis produces a tree

$$\begin{tikzcd}
 & \text{ASSIGN\_STATEMENT} \arrow[dl] \arrow[d] \arrow[dr] & & \\
\text{IDENT} & \text{ASSIGN} & \text{EXPR} \arrow[dr] & \\
 & & & \text{IDENT}
\end{tikzcd}$$

Semantic analysis

- Refine the syntax tree, augmenting with properties and rules for things like type checking/coercion, check for declare-before-use, scope checking, etc
- A symbol table is often used to store information about each symbol (e.g. variable names, constant names, etc) as the information becomes available
- As the syntax tree is built (top down) we might look up information about the identifiers (e.g. what type they were declared as) and as the tree is evaluated (bottom up) we might check those against the value determined for an expression (are the types of x and y compatible for x = y ;)

Intermediate representation

- Produce an abstract representation of the program: independent of any specific programming language
- The augmented syntax tree is an example of such a representation, possibly with further manipulations to abstract away specifics of the source language
- An abstract (platform independent) assembly language is another possible representation

Machine-independent optimization

- Some optimizations can be performed regardless of the target, e.g. removing unreachable or unused code, replacing replicated code with a subroutine call, replacing inefficient operations with more efficient ones, rearranging code segments to reduce branches/jumps (or to reduce the length of jumps)

Code generation

- Now, from the optimized independent representation we're trying to generate actual target code
- This will involve trying to match constructs in the abstract representation with specific instruction sequences from the target language (including selection of appropriate data types and addressing modes, appropriate checks in the target language to parallel those in the source, etc)

Target code optimization

- Now we're taking the target language source code we've generated and optimizing/transforming it specifically for the target architecture: this might include optimizations or techniques that were not suitable at a generic level
- Typically multiple levels of optimization are supported, some targetting completely different aspects of behaviour (e.g. minimizing code size vs minimizing run time vs minimizing page faults etc)

Front/back end stages

- Generally “front end” refers to the stages from reading the source code character stream through the production of an intermediate representation
- Generally “back end” refers to the generation and optimization of code in the target language