## Interpreters

Interpret :: (Program, Data) --> Output

*or (currying the inputs)* Interpret :: Program --> Data --> Output

*or (grouping the last two outputs)* Interpret :: Program --> (Data --> Output)

*or* Interpret :: Program --> Executable (Futamura's first projection, discussed below)

*"online"*

## Compilers

Compile :: Program --> Executable

Executable is in the language of the implementation of the Interpreter (e.g., assembly language for Java interpreter)

Executable :: Data --> Output

*"offline"*

## Partial evaluation

A computer program can be seen as a mapping of input data to output data:

```
prog :: Istatic x Idynamic --> Output
```

The partial evaluator (also called *specializer*) transforms (prog, Istatic) into prog* :: Idynamic -> Output

prog* is called the residual program and should run more efficiently than the original program

## Futamura Projections

Interesting case when prog is an interpreter for a programming language. Described in 1970s by Yoshihiko Futamura

If Istatic is source code designed to run inside an interpreter, then partial evaluation of the interpreter with respect to this data/program produces prog*, a version of the interpreter that only runs that source code.

This specialized version of the interpreter is written in the implementation language of the interpreter., i.e., the language in which the interpreter has been written (e.g., assembly language for the Java interpreter).

The source code does not need to be re-supplied to prog*

prog* is effectively a compiled version of Istatic

**This is the first Futamura projection**. There are three Futamura projections:

*A specializer is a partial evaluator*

1. **Futamura Projection 1**: Specializing an interpreter for given source code, yields an executable
    - **Interpret :: prog --> Executable**
    - *Specializing (partially evaluating) the Interpreter for the program*

Specialize1 :: (Interpret, prog) --> Executable

- *Currying the arguments*

  Specialize1 :: Interpret --> prog --> Executable

- *Grouping the last two outputs*

  Specialize1 :: Interpret --> (prog --> Executable)

2. **Futamura Projection 2**: Specializing the specializer for the interpreter (as applied in #1) yields a compiler

   - **Specialize1 :: Interpreter --> Compiler**

   - *Specializing the specializer for the Interpreter*

     Specialize2 :: (Specialize1, Interpreter) --> Compiler

   - *Currying the arguments*

     Specialize2 :: Specialize1 --> Interpreter --> Compiler

   - *Grouping the last two outputs*

     Specialize2 :: Specialize1 --> (Interpreter --> Compiler)

3. **Futamura Projection 3**: Specializing the specializer for itself (as applied in #2), yields a tool that can convert any interpreter to an equivalent compiler.

   - **Specialize2 :: Specialize1 --> InterpreterToCompilerConverter**

## Take-aways from Futamura's projections

- An executable is a partial-evaluation of an interpreter for a program's source code
- The process of partial-evaluation of an interpreter for a program's source code is called *compilation*
- The execution of the partially-evaluated interpreter (for the program) should ideally be faster than the execution of the original interpreter on the program.
  - In other words, the execution of the executable should be faster than the execution of the interpreter on the program.
- The quality of the compilation is determined by the speedup obtained through this *offline* partial evaluation
  - A trivial compilation is to just store the interpreter and the program's source code as a tuple; running it would involve full evaluation of the interpreter over all arguments. But this will result in slow runtime.
  - A smarter compilation will involve substituting ("constant propagating") the input program (Istatic) into the interpreter's implementation. If the interpreter is written as an evaluation loop over each statement in the source program, then compilation will also involve unrolling the loop for the number of statements in the source program:

```
Interpret(Prog, Data):
    for Statement in Prog:
        Data = Evaluate(Statement, Data)
    return Data

Executable(Data):
    Data = EvaluateStatement1(Data)
    Data = EvaluateStatement2(Data)
    Data = EvaluateStatement3(Data)
    ...
```

```
Data = EvaluateStatementN(Data)
return Data
```

Can we do better?

- Local Optimizations: Specialize the code for EvaluateStatementI. Recall that Data is not available at compile time (Idynamic), but StatementI is available (Istatic)
- Caching the generated code (An optimization that is very effective for code with loops):
  - The evaluation of a statement involves (a) generation of machine code for the Statement and (b) execution of the generated machine code on the Data
  - If one statement, say StatementI is executed several times (typical programs execute the same statement millions to billions of times), cache the generated machine code, so we do not need to regenerate the same machine code repeatedly.
    - Static compilation, also called *Ahead-of-time* Compilation: Cache the generated machine code for all statements, irrespective of whether they execute once or multiple times.
      - *Pro*: Offline, zero compilation cost at runtime.
      - *Con*: Compile + Execute may be slower (or at best equal cost) than plain interpretation for some programs, e.g., programs which have no loops.
      - *More serious con*: Optimized compilation is expensive. For a given time budget for compilation, we do not have enough information to decide how much time to spend on optimizing which part of the program. e.g., the most executed part of the program should be optimized the most, while dead-code needs no optimization. This information is not available during ahead-of-time compilation, and can only be estimated using approximation heuristics.
    - Dynamic compilation, also called *Just-in-time* Compilation:
      - Generate machine code at runtime.
      - Maintain a cache of generated code for each program fragment.
      - Cache can be limited size. Need cache-replacement policy (e.g., Least Recently Used LRU).
      - Do not need to re-generate code on cache hit.
      - *Pro:* Can spend more effort on the program fragments that are executed most (*hot regions*).
      - *Pro:* If the generated code is much bigger than the source program, then storage requirements are smaller in JIT compilation. Real problem for Android's JVM. e.g., JIT compiled Facebook App on Android is 90MB in size, while AOT compiled Facebook App takes 250MB. For a long time, Samsung hard-codeed Facebook to be JIT compiled, while some other apps were AOT compiled.
      - *Serious con*: Pay compilation cost at runtime.
        - However, this is not a very serious con for typical applications that are relatively small but run for a long time (due to loops); typical JIT compilation times for most applications are significantly smaller than typical runtimes.
        - Often the advantages of JIT optimization outweigh the JIT costs.
        - With multi-core processors, we can use the additional processors to do the JIT compilation/optimization in parallel.
- Global optimizations: Optimizations that span multiple statements (or program fragments) are often possible.
  - Because the programmer wrote sub-optimal code. It is our job to optimize it now. e.g., the programmer wrote x=1; x=2;
  - Because the machine representation may be richer than the source code syntax. e.g., the C syntax has multiply and add as separate operations, but the machine may have a single instruction to do multiply-and-add in one go.
  - Because some optimizations enable other optimizations. e.g., constant substitution can trigger more constant substitution.
    - *Caveat*: On the other hand, it is also possible for some optimizations to *preclude* other optimizations. e.g., replacing multiplication with shift may preclude the use of

multiply-and-add instruction.
- And several other reasons...
  - Generating the optimal implementation for a given program specification is an undecidable problem in Turing's model of computation, and an NP-Hard problem in the finite model of computation.

## Why study compilers? Can't I just take them for granted and forget about them?

**If you ask me, compilers are the most exciting research area in computer systems (and perhaps in the whole of computer science) currently, and are likely to remain so for several years to come.**

Two technology trends that show the rising importance of compilers in Computer Science:

1. *Moore's law is running out of steam*
   - Gordon Moore, the co-founder of Fairchild Semiconductor and Intel, in his 1965 paper, described a doubling every year in the number of components per integrated circuit for at least the next decade.
   - In 1975 (after one decade), he revised it to doubling every two years.
   - Usually meant faster and more powerful computers, but speeds saturated in early 2000s, necessisating the need towards multi-core computing.
   - In 2015, Intel CEO, Brian Krzanich, revised the estimate to "doubling every two and a half years".
   - Physical limits to transistor scaling, such as source-to-drain leakage, limited gate metals, and limited options for channel material have been reached. While scientists are still exploring more ways to scale the transistors further (e.g., electron spintronics, tunnel junctions, etc.), it is safe to assume that all good things come to an end.
   - Moore's law meant that architects and computer system engineers did not have to work too hard
     - The process and materials engineers are already making great improvements. We are too busy catching up with that.
       - *Consequence*: Our OS and programming language stacks are quite similar to what they were in 1960s-70s. Seems absurd to imagine that this will continue as-is for the next 50 years.
     - Need innovations at the architecture level: perhaps we need innovative computing units not limited to sequential execution of simple instructions, or perhaps something else
       - If you read research papers in the architecture community, it is evident that several innovative ideas to improve performance have been proposed over the past 20 years or so.
       - Yet, these ideas have not become mainstream.
         - The primary stumbling block is: we do not know how to automatically and efficiently map program specifications written by humans to these new innovative architectural constructs.
         - Humans find it hard to directly code using these architectural constructs/abstractions, as they are not intuitive
         - Today's compilers are too fragile
           - The original designs for compilers were meant to handle simple opcodes (e.g., add, multiply, etc.), which were easily generated using similar constructs used by humans in their programs.
           - Over the years, this design the compiler has been loaded with several more requirements: vectorization, more programming constructs, more opcodes, parallel constructs and implementation, and so on...
             - Result: Over 14 million lines of code in GCC. Has increased at the rate of two million SLOC every three years. I predict that it will continue to grow at this rate (or higher) for the next decade, if the compiler design is not changed. You can call this the Compiler Complexity Law (or Sorav's Law if you will ;-).
2. *Program complexity is increasing*
   - We expected programs to just perform arithmetic computations in 1960s. Today we expect them to talk to us, write English books, and perhaps even develop a compiler!

    - This has given rise to programming in higher levels of abstraction
        - Higher levels of abstraction was about constructs like objects, automatic memory management constructs, etc. (e.g., Java, C++), in late 1980s and 1990s.
        - In 2000s, this was about abstractions for distributed data-intensive applications, e.g., Map, Reduce, Sort, Filter, etc.
        - In 2010s, this was about machine-learning constructs like Neuron and Connections, e.g., TensorFlow.
        - Several other domain-specific abstractions/languages have emerged --- e.g., image and video processing, web programming, UI programming, networking, etc.
    - Higher levels of abstraction increase the gap between the programming language and the architectural abstractions, making the search space for optimal implementations even larger. Thus compilers assume a central role in this setting.

Some more crazy ideas that seem to be going around these days

- Natural language processing can be modeled as a compilation problem? I do not believe this, but I also do not believe several other things that machine-learning people say ;-)
- Almost every problem in computer science is a compilation problem. For example, can't I just say that I need a web-server (in some very high-level syntax), and can't a compiler generate a highly optimized implementation of a webserver (using multiple machines in a data-center) for me?

## Some history

- 1954 : IBM develops the 704 (the first commercially-successful machine)
    - Found that software costs exceeded the hardware costs by a lot
- 1953 : John backus introduced something called "Speedcoding"
    - A small language that was much easier to program than directly programming the machine
    - 10-20x slower (interpreted)
    - Interpreter itself took 300 bytes of memory (which was 30\% of all available memory).
    - Did not become popular
- Most common use of computers was for evaluating arithmetic formulae. John Backus, took his Speedcoding experience, to develop a higher-level program language for Formulas, and automatically "Translate" these "Formulas", in a language called "ForTran". The compiler took three years to develop (during 1954-57).
- By 1958, 50\% of all programs were written in ForTran1 --- huge success!
- The first Fortran compiler led to a huge body of theoretical work. Modern compilers preserve the outline of Fortran1!
    - Compilers is a field that involves a mixture of theory and systems in quite an intense way.

## Outline of the Fortran1 compiler

1. Lexical Analysis
2. Parsing
3. Semantic Analysis
    - Types, scope, etc.
4. Optimization
5. Code generation (translation)

## Steps in a compiler in some more detail

Explain using an analogy to how humans understand written english.

- Understanding English: First step is to recognize words (lexical analysis)
    - Words are the smallest unit above letters
    - Example: `This is a sentence.`
        - Background computation: identify separators (white spaces), capitals, punctuation marks, etc., to identify words and sentence.

        - Counterexample: `ist hisa sen tence.` is much harder to read. Hence, first step is to group letters into words (or tokens).
    - `if x == y then z = 1; else z = 2;`
        - Need to group these characters into tokens.
        - The language keywords (if, then, else) are each represented as a single token.
        - Each identifier (x, y, z) is also identified as a single token.
        - Each operator (=, ==) is identified as a single token. Now, how do we know that "==" represents a single token or is a combination of two "=" tokens? We will see this when we discuss the lexical analyzer.
        - Delimiters like white-space and semi-colon, are also recognized as separate tokens. Multiple white-space is recognized as a single token in this language, for example.
- English language: the next step is to understand sentence structure (parsing)
    - Parsing = diagramming sentences as trees
    - `This line is a longer sentence`
        - `This`: Article
        - `line`: Noun
        - `is`: Verb
        - `a`: Article
        - `longer`: Adjective
        - `sentence`: Noun
        - Next level:
            - `This line` : subject
            - `is` : verb
            - `a longer sentence` : object
        - Next level:
            - `This line is a longer sentence` : subject verb object : sentence
    - `if x == y then z = 1; else z = 2;`
        - `x==y` : relation
        - `z = 1;` : assignment
        - `z = 2;` : assignment
    - Next level
        - `x == y`: predicate
        - `z = 1`: then-statement
        - `z = 2`: else-statement
    - Next level
        - predicate then-statement else-statement : if-then-else
    - Is it a co-incidence that English parsing is exactly similar to program parsing?
- English language: the next step is to understand the "meaning" of the sentence (semantic analysis)
    - This is too hard!
    - Reduces to the equivalence checking problem in computers, which is undecidable!
    - Compilers perform limited semantic analysis to catch inconsistencies, but they do not know what the program is supposed to do.
    - Variable binding : an example of semantic analysis
        - `Jack said Jerry left his assignment at home`
            - Whom does `his` refer to? Jack or Jerry?
        - Even worse: `Jack said Jack left his assignment at home`
            - How many people are involved? Could be 1, 2, or 3.
        - These are examples of *ambiguity* in the english language syntax.
        - Such ambiguities are resolved in programming languages through strict variable binding rules. `{ int Jack = 3; { int Jack = 4; cout << Jack; } }`
            - Common rule: bind to inner-most definition (inner definition hides outer definition)
    - Type analysis: another example of semantic analysis
        - `Jack left her homework at home`
            - Assuming Jack is male, this is a type mismatch between `her` and `Jack`; we know they are different people.
- English language: optimization is perhaps a bit like editing (not a strong correlation)
    - Automatically modify programs so that they
        - Run faster

        - Use less memory
        - Reduce power
        - Reduce network messages or database queries
        - Reduce disk accesses, etc.
    - Optimization requires precise semantic modeling; several subtleties emerge in this context.
        - Can `for (unsigned i = 0; i < n + 1; i++) {}` be changed to `for (unsigned i = 0; i <= n; i++) {}`? Answer: No!
        - Can `for (int i = 0; i < n + 1; i++) {}` be changed to `for (int i = 0; i <= n; i++) {}`? Answer: Yes!
        - Can `(2*i)/2` be changed to `i`? Answer: No!
        - Can `Y*0` be changed to `0`? Answer: No for floating point!
- English language: Translation into another language (code generation)
    - Usually generate code for assembly language in compilers.
- The overall structure of almost every compiler adheres to this outline, yet proportions have changed.
    - Sorted in order of size/complexity in Fortran: Lexer, Parser, CG, Optimizations, Semantic analysis
    - Sorted in order of size/complexity in today's compilers: Optimizations (by a huge margin), Semantic analysis (deeper than before), CG (rather small), Parser, Lexer

## Economy of Programming Languages

- Why are there so many (hundreds to thousands) programming languages?
    - Application domains have distinctive/conflicting needs
        - Scientific computing
            - Need rich support for floating-point operations, arrays, and parallelism. e.g., Fortran
        - Business applications
            - Need support for persistence, report generation facilities, data analysis. e.g., SQL
        - Systems programming
            - Need support for fine-grained control over resources, real-time constraints. e.g., C/C++

      Different languages make it easier for human programmers to *efficiently* encode different types of logic, and require different types of optimization support! Somewhat related to the fragility of our current compilers.