CS 320: Principles of Programming Languages

Katie Casamento, Portland State University
Based on slides by Mark P. Jones, Portland State University, Winter 2017

Winter 2018
Week 2: Programs as Data

How do we work with programs?

Program analysis

Analyze code to produce human-usable output

- Interpreters (e.g. GHCi)
- Static analysis tools (e.g. Valgrind)
- Documentation generators (e.g. Javadoc)
- Debuggers (e.g. gdb)
- Profilers (e.g. gprof)
- …

Program synthesis

Generate programs from higher-level descriptions

- GUI builders (e.g. Android Studio Layout Editor)
- Embedded languages (e.g. PHP templates)
- Code wizards (e.g. Apple Automator)
- Modeling tools (e.g. Blender)
- …

Program translation

Convert code from one language/format to another

- Compilers (e.g. GCC)
- Code formatters (e.g. gofmt)
- Update tools (e.g. Python 2to3)
- Backport tools (e.g. Traceur)
- Macro processors (e.g. cpp)
- …

General building blocks

- A *front end* reads source programs (e.g. text files) and captures the corresponding abstract syntax tree in a collection of data structures (e.g. trees, graphs, arrays, …)
- A *middle end* analyzes and manipulates the abstract syntax data structures of a program
- A *back end* generates output (e.g. a binary executable file) from the abstract syntax data structures of a program
- Substantial parts of these components can be shared by multiple tools
    - e.g. ghc (compiler) and ghci (interpreter) both use the same front and middle end components

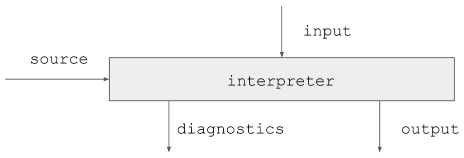
Interpreters and compilers

Interpreters and compilers

- An interpreter *executes* (or runs) programs
    - An interpreter for a language L can be seen as a function L -> M, where M is some set of meanings of programs
    - L = Prop: M = (`Env -> Bool`)
    - L = C: M = the set of possible execution traces
- A compiler *translates* programs
    - A compiler from a language L to a language L' can be seen as a function L -> L'
    - L = Prop: L' = VHDL (hardware circuit description)
    - L = C: L' = x86 assembly
- By "language", we mean *formal language*: the set of all syntactic structures that correspond to valid programs in some syntax

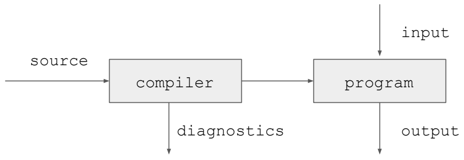
Interpreters

Interpreters *execute* programs (turning syntax to semantics)



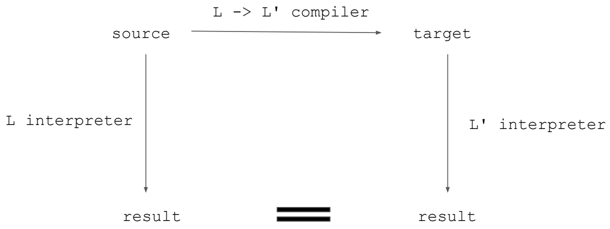
Compilers

Compilers *translate* programs (turning programs in the syntax of a *source* language into the syntax of a *target* language)



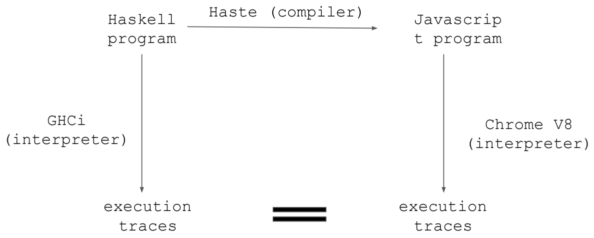
Compiler correctness

- A compiler should produce valid output for any valid input
- The output should have the same semantics as the input:

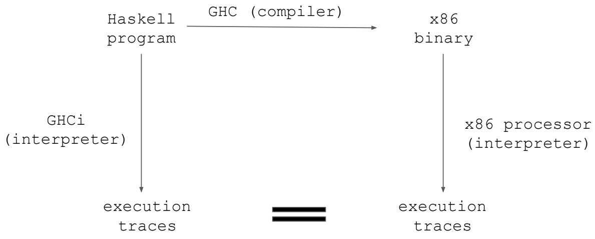


If this property holds, the compiler is *correct* with respect to the two interpreters.

Compiler correctness example



Compiler correctness example



Desirable properties of a compiler

- Performance:
  - Of the compiled code
  - Of the compiler
- Diagnostics:
  - High quality error messages and warnings to aid in diagnosing mistakes in programs
- Support for large programming projects
  - Separate compilation, to avoid needless recompilation
  - Library management, to enable software reuse
- Convenient development environment
  - IDE support
  - Profiling
  - Debugging

Compiler examples

Compilers show up in many forms:

- Translating programs in high-level languages like C, C++, Java, etc. to executable machine code
- Just in time (JIT) compilers: translating bytecode to machine code at runtime
- Converting a .txt file to a .pdf file
- Generating a .wav file from written text with a speech-to-text program

Interpreter characteristics

Common (but not universal) characteristics of interpreters include:

- Interactive use
  - Read-eval-print loop (REPL)
  - Often found in languages designed for educational use or with a focus on prototyping
- Less emphasis on performance
  - Overhead that could be eliminated by compilation
- Portability
  - Interpreters are often easier to port to other platforms since they don't depend on the details of a particular target language/architecture
- Experimentation
  - Often used for prototyping semantics of new languages or language features
  - More flexible for language design - some features are easier to implement in an interpreter than in a compiler (e.g. reflection)

Interpreter examples

Programming languages

- Scripting languages: PHP, Python, Ruby, Perl, Bash, Javascript, …
- Educational languages: BASIC, Logo, …
- Declarative languages: Lisp, Scheme, ML, Haskell, Prolog, …
- Virtual machines: JVM bytecode (Java/Scala), CLR bytecode (C#/F#/VB), …

Document description languages

- Postscript, HTML, …

Hardware

- A CPU interprets machine language programs

Language vs. implementation

A technicality, but often an important one: a language is not the same as an implementation of a language.

- C, Python, Haskell, etc. are languages
- GCC, CPython, GHC, etc. are implementations

Properties of languages: programming paradigm, expressivity, design patterns, …

Properties of implementations: efficiency, platform, compiled vs. interpreted, …

Goals for compiler construction

Why do we need compilers/interpreters?

We like to write programs at a higher level than the machine can execute directly:

- Spreadsheet
```
sum [A1:A3]
```
- Java:
```
a[1] + a[2] + a[3]
```
- x86 assembly:
```
movl $0,     %eax
addl 4(a),   %eax
addl 8(a),   %eax
addl 12(a),  %eax
```

How do we turn high level ideas into low level code?

Ideas:

- search a database
- send a message
- play a game

Machine code:

- read a value from memory
- add two numbers
- compare two numbers
- jump to an instruction

How do we turn high level ideas into low level code?

We can build language features (abstractions) that we can use to express ideas, and that we can then translate into machine code.

- Evaluate an expression
- Execute a computation multiple times (iteration)
- Call a function
- Save a result in a variable

How do we turn high level ideas into low level code?

Two complementary disciplines:

- Language design: building a way to express ideas
- Compiler construction: building a way to execute expressions

How do we turn high level ideas into low level code?

Historical notes:

- First program: 1842-1843, Ada Lovelace publishes a procedure[1] for calculating Bernoulli numbers on Charles Babbage's proposed Analytical Engine (mechanical computer)
- First compiler: 1951-1952, Richard Ridgeway and Margaret Harper under the management of Grace Hopper construct a compiler[2] for the A-0 arithmetic programming system on the UNIVAC 1

[1] https://commons.wikimedia.org/wiki/File:Diagram_for_the_computation_of_Bernoulli_numbers.jpg

[2] https://dl.acm.org/citation.cfm?id=808980

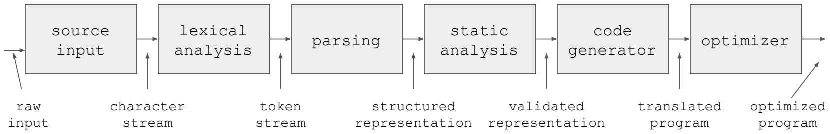
Basics of compiler structure

Analogy: "static analysis" of English sentences

- The symbols must be valid
  - "hdk fΩfdh ksdßs dfsjf dslkjé" is invalid source input
- The words must be valid
  - "Banana jubmod food funning." fails lexical analysis
- The text must use correct grammar
  - "My walking up left tree dog." fails parsing
- The phrase must make sense
  - "This sentence is not true." doesn't have any clear meaning
- The phrase must not be ambiguous
  - "They are talking." is underspecified

The compiler pipeline

Traditionally, the task of compilation is broken down into several *phases*.



source input

lexical analysis

parsing

static analysis

code generator

optimizer

raw input

character stream

token stream

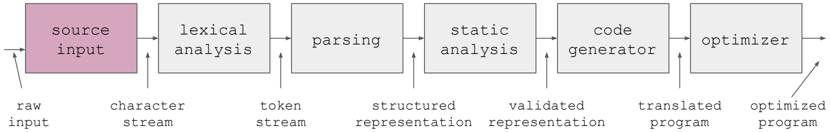
structured representation

validated representation

translated program

optimized program

Source input



source input

lexical analysis

parsing

static analysis

code generator

optimizer

raw input

character stream

token stream

structured representation

validated representation

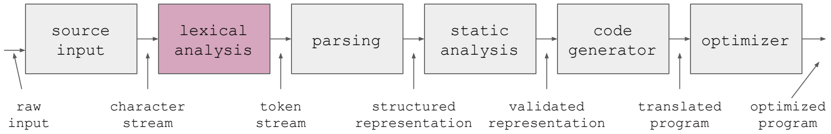
translated program

optimized program

Turn data from a raw input source into a sequence of characters or lines

- Sources: hard disk, RAM, keyboard input, …
- The operating system usually takes care of most of this

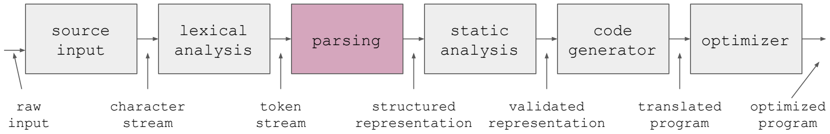
Lexical analysis



Convert the input stream of characters into a stream of tokens

- characters: "`while (i > 0)`"
- tokens: `WHILE, L_PAREN, ID(i), GREATER_THAN, NUM(0), R_PAREN`
- Analogous to spell check
- "lexical": "related to the words or vocabulary of a language"

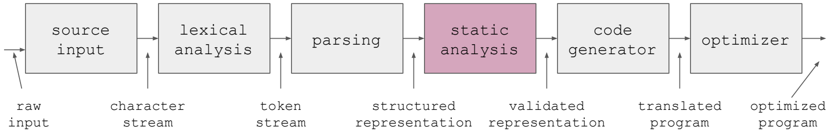
Parsing



Build data structures that capture the underlying structure (abstract syntax) of the input program

- Determines whether inputs are structurally well-formed
- Analogous to grammar check

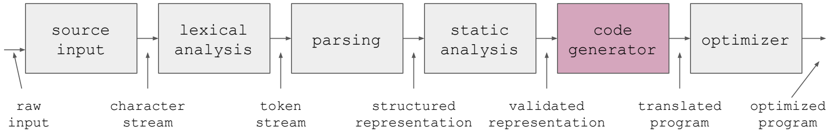
Static analysis



Check that the program is reasonable

- No references to undefined variables (scope checking)
- No type inconsistencies (type checking)
- Less commonly: bounds checking, contract checking, proof checking, …

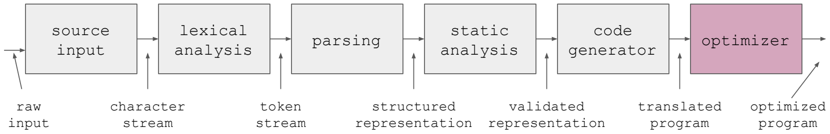
Code generation



Generate an appropriate sequence of machine instructions as output

- Different strategies are needed for different target machines
- Machines can be physical (x86, ARM) or virtual (JVM, CLR)

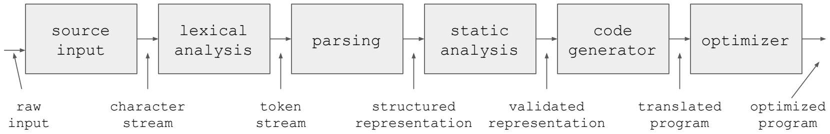
Optimization


source input
lexical analysis
parsing
static analysis
code generator
optimizer
raw input
character stream
token stream
structured representation
validated representation
translated program
optimized program

Look for opportunities to improve the quality of the code

- Eliminate dead code, reduce allocations, parallelize loops, …
- There may be conflicting ways to "improve" a given program, depending on the context and the user's priorities

Variations


source input
lexical analysis
parsing
static analysis
code generator
optimizer
raw input
character stream
token stream
structured representation
validated representation
translated program
optimized program

There are many variations on this approach that you'll see in practical compilers:

- Extra phases (e.g. preprocessing)
- Iterated phases (e.g. multiple optimization phases)
- Additional data passed between phases

Phases vs. passes

- A *phase* is a conceptual stage in a compiler pipeline
- A *pass* is a concrete traversal over the representation of a program
- Several phases may be combined into one pass
- Passes may be run in sequence or in parallel
- Some languages are specifically designed to be implemented in a single pass

Front ends and back ends, intuitively

Front end: the parts of a compiler that depend most heavily on the source language

- CS 421: source input, lexical analysis, parsing, static analysis

Back end: the parts of a compiler that depend most heavily on the target language

- CS 422: code generation, optimization, assembly

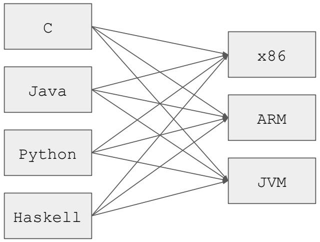
Modularity in compiler design

Modularity

- Building large systems from collections of smaller components
- Modular implementations can be easier to write, test, debug, understand, and maintain than monolithic implementations:
  - Components can be developed independently
    - **`gcc`** compiles to assembly, **`as`** assembles to machine code
  - Components can be reused in other contexts
    - **`as`** is the assembler for many different language implementations
  - Some components may be useful as a standalone tool
    - **`cpp`** (the C preprocessor) can be used on text files

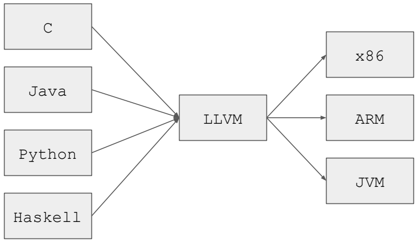
Intermediate languages

If we want to write compilers for M different languages on N different target platforms, we have M*N programs to write:



Intermediate languages

An alternative is to design a general purpose *intermediate language*:



- Now we only have M frontends + N backends to write!
- The biggest challenge is finding a general enough language to accommodate a wide variety of languages and platforms

Bootstrapping

Three different languages are relevant to a compiler:

- Source (input) language
- Target (output) language
- Implementation language

What if the implementation language is the same as the source language?

- GCC is written mostly in C++
- GHC is written mostly in Haskell
- PyPy is written mostly in Python
- These language implementations are *self-hosted*

Bootstrapping

How can you compile a self-hosting compiler?

- Use someone else's compiler for the same source language to compile your compiler
- Write a simpler compiler (e.g. without optimizations) in a different implementation language and use it to compile your compiler

This process is called *bootstrapping*, after the idiom "to pull yourself up by your bootstraps".

Summary

- Basic principles
  - Programs as data (source texts, token streams, syntactic data structures, bytecode)
  - Interpreters and compilers
  - Correctness means preserving semantics
- The compiler pipeline/phase structure
  - source input -> lexical analysis -> parsing -> static analysis -> code generation -> optimization
- Modularity
  - Techniques for simplifying compiler construction tasks