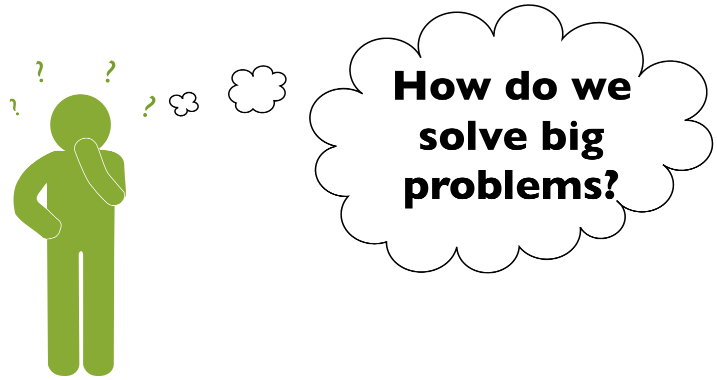
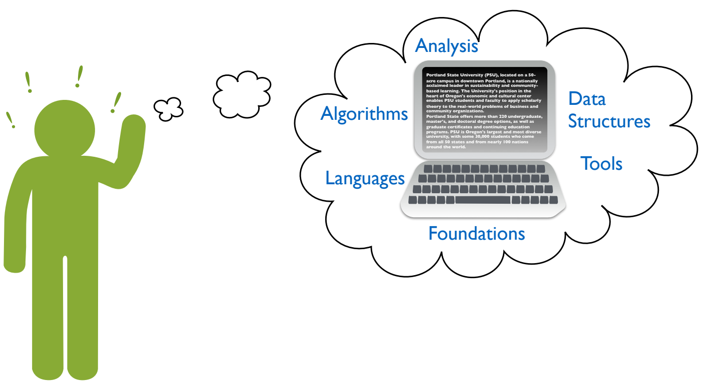
CS 320: Principles of Programming Languages

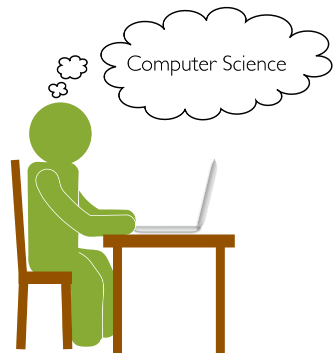
Mark P Jones, Portland State University

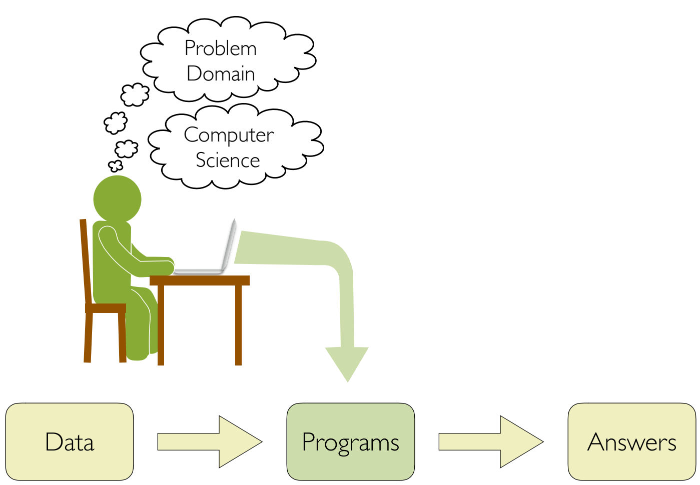
Spring 2019

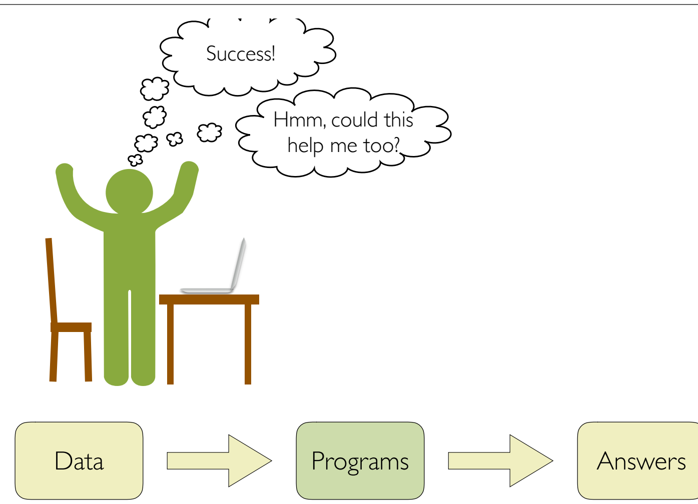
Week 3: Programs as Data

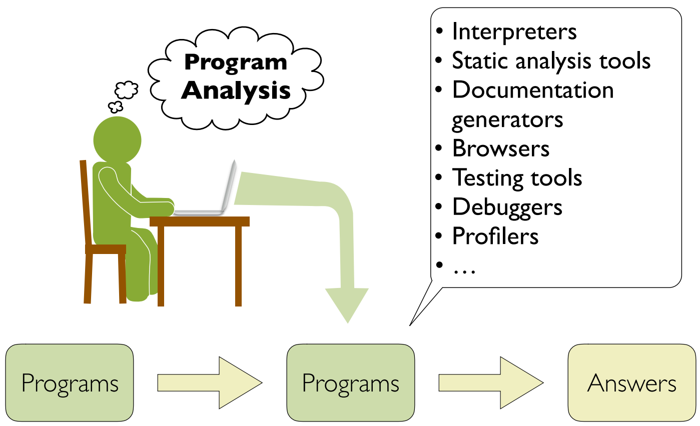




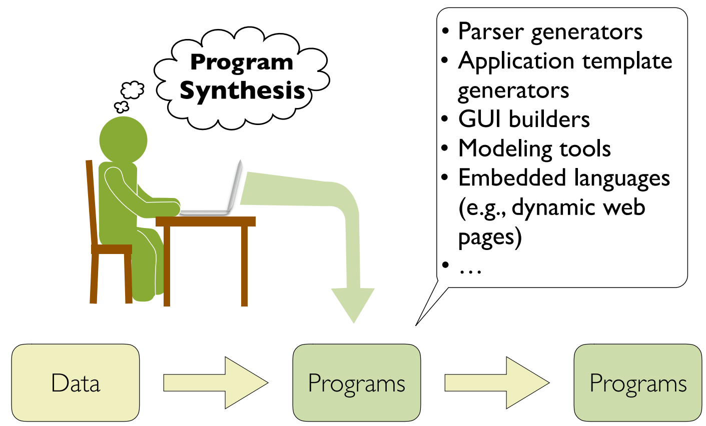




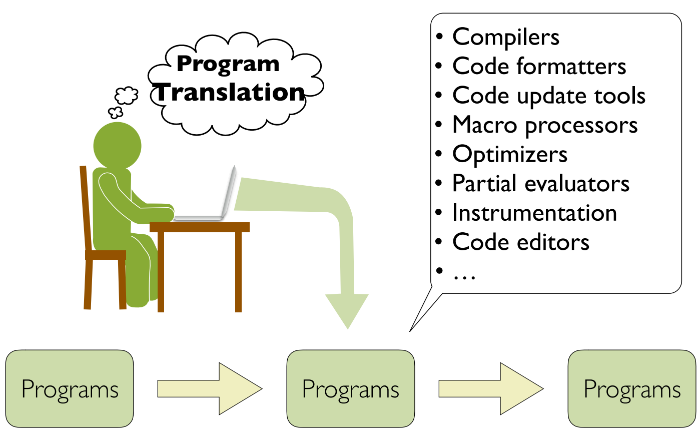




programs as inputs



programs as outputs



programs as inputs *and* outputs

General building blocks

- A **front end** reads source programs (e.g., flat text files) and captures the corresponding abstract syntax 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 flat, binary executable file) from the abstract syntax data structures of a program
- Substantial parts of these components can be shared by multiple tools
  - Example: the `g++` and `gcc` compilers (for C++ and C, resp.) use the same middle and back end components
  - Example: the `ghc` (compiler) and `ghci` (interpreter) for Haskell use the same front and middle end components

Interpreters and compilers

Interpreters and compilers

In conventional English:

- ~~**interpreter**: somebody that translates from one language to another.~~
  - ~~Example: "I need an interpreter when I'm in Japan"~~
- ~~**compiler**: somebody who collects, gathers, assembles, or organizes information or things.~~
  - ~~Latin root: compilare, "plunder or plagiarize"~~

Not how the terms are used in computer science!

Interpreters and compilers

According to my dictionary:

- **in•ter•pret•er** (noun) Computing: a program that can analyze and execute a program line by line
- **com•pile** (verb) Computing (of a computer): convert (a program) into a machine-code or lower-level form in which the program can be executed

  Derivatives: **com•pil•er** (noun)

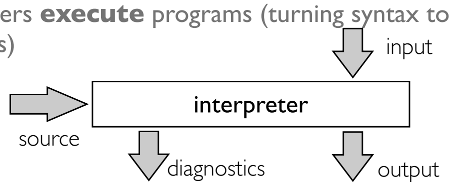
Interpreters and compilers

In computer science:

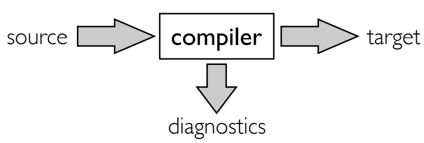
- An interpreter <u>executes</u> (or runs) programs
  - An interpreter for a language L might be thought of as a function: $\text{interp}_L : L \to M$, where M is some set of meanings of programs
- A compiler <u>translates</u> programs
  - A compiler from a language L to a language L' might be thought of as a function $\text{comp} : L \to L'$
- By "language", we mean the set of all strings that correspond to valid programs

Interpreters and compilers

- Interpreters **execute** programs (turning syntax to semantics)



- Compilers **translate** programs (turning syntax into syntax)



"Doing" vs "Thinking about doing"

- Compilers translate programs (turning syntax to syntax)
- Interpreters run programs (turning syntax to semantics)
- Example:
  - Interpreter (Doing something):
    Use your calculator to evaluate (1+2)+(3+4):

    Answer: 10
  - Compiler (Thinking about doing something):
    Tell me what buttons to press to evaluate (1+2)+(3+4):

    Answer: 1 + 2 = M 3 + 4 + MR =

Interpreter characteristics

Common (but not universal) characteristics:

- More emphasis on interactive use:
  - Use of a read-eval-print loop (REPL)
  - Examples: language implementations designed for educational or prototyping applications
- Less emphasis on performance:
  - Interpretive overhead that could be eliminated by compilation
  - Performance of scripting code, for example, is less of an issue if the computations that are being scripted are significantly more expensive

Interpreter characteristics, continued

- Portability:
  - An interpreter is often more easily ported to multiple platforms than a compiler because it does not depend on the details of a particular target language
- Experimental platforms:
  - Specifying programming language semantics
  - More flexible language designs; some features are easier to implement in an interpreter than in a compiler

Interpreter examples

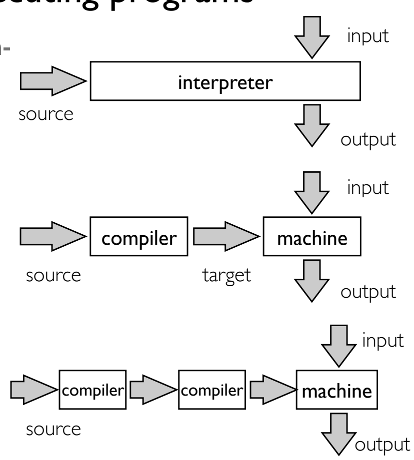
- Programming languages:
  - Scripting languages: PHP, python, ruby, perl, bash, Javascript, ...
  - Educational languages: BASIC, Logo, ...
  - Declarative languages: Lisp, Scheme, ML, Haskell, Prolog, ...
- Document description languages:
  - Postscript, HTML, ...
- ...

Interpreters and machines

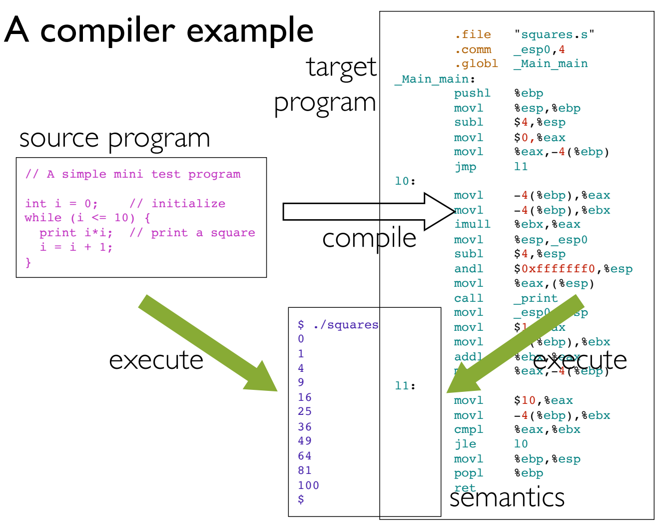
- A CPU executes (machine) programs in hardware
  - So it is a kind of interpreter too!
  - Potentially faster, but harder to change
- A virtual machine is an important kind of interpreter:
  - Executes programs written in a virtual (i.e., software-defined) instruction set
  - Example: the Java Virtual Machine (JVM) executes/interprets a language of byte code instructions (used for Java, Scala, Clojure, and more)

Strategies for executing programs

- A interpreter for a high-level source language
- Compilation to a low-level target language
- Compilation via intermediate languages



A compiler example



Compiler correctness

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

$$\begin{tikzcd}
\text{source} \arrow[r, "\text{comp}_{L\to L'}"] \arrow[d, "\text{interp}_L"'] & \text{target} \arrow[d, "\text{interp}_{L'}"] \\
\text{result}_s \arrow[r, leftrightarrow, "="] & \text{result}_t
\end{tikzcd}$$

In symbols: $\forall p.\ \text{interp}_L(p) = \text{interp}_{L'}(\text{comp}_{L\to L'}(p))$

Desirable properties of a compiler

- Performance:
  - Of compiled code: time, space, power, …
  - Of the compiler: time, space, …
- Diagnostics:
  - High quality error messages and warnings to permit early and accurate diagnosis and resolution of programming mistakes

Desirable properties, continued

- Support for large programming projects, including:
  - Separate compilation, reducing the amount of recompilation that is needed when part of a program is changed
  - Use of libraries, enabling effective software reuse
- Convenient development environment:
  - Supports program development with an IDE or a range of useful tools, for example: profiling, debugging, cross-referencing, browsing, project management (e.g., make, git, ...)

Compiler examples

Compilers show up in many different forms:

- Translating programs in high-level languages like C, C++, Java, etc… to executable machine code
- Just in time compilers: translating byte code to machine code at runtime
- Generating audio speech from written text
- Translating from English to Spanish/French/…
- …

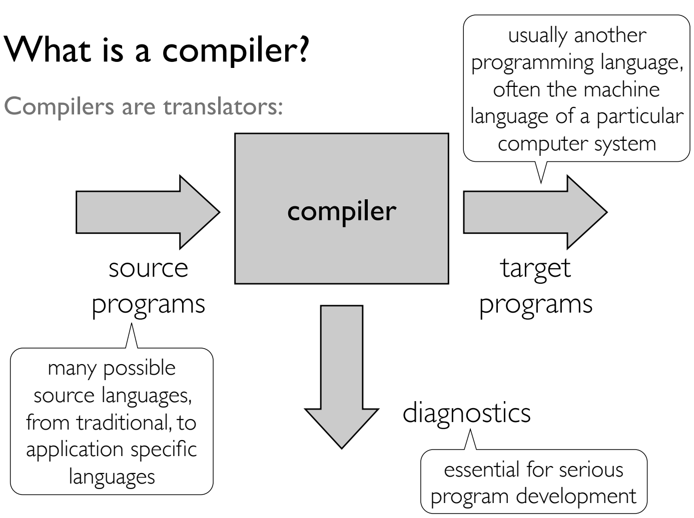
Language vs implementation

- Be very careful to distinguish between languages and their implementations
- C is a widely used language
- Haskell is an expressive language
- ML is a well-defined language
- Python is a slow language (NO: speed is a property of an implementation, not a language)
- C++ is a compiled language: (NO: "compiled" describes a property of an implementation, not a language)

Goals for Compiler Construction

What is a compiler?

Compilers are translators:



Why translation is needed

- 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]`
  - Machine language:

```
movl $0, %eax
addl 4(a), %eax
addl 8(a), %eax
addl 12(a), %eax
```

- High-level languages let us describe what is to be done without worrying about all the details
- In machine languages, every step must be carefully spelled out

Ideas:

- Search a database
- Send a message
- Create a song
- Play a game
- etc ...

High Level

How do we turn **high level ideas** into running programs on **low level machines**?

Machines:

- Read a value from memory
- Add two numbers
- Compare two numbers
- Write a value to memory
- etc ...

Low Level

Ideas:

- Search a database
- Send a message
- Create a song
- Play a game
- etc ...

High Level

express

Languages:

- Evaluate an expression
- Execute a computation multiple times
- Call a function
- Save a result in a variable
- ...

translate

Machines:

- Read a value from memory
- Add two numbers
- Compare two numbers
- Write a value to memory
- etc ...

Low Level

Ideas:

High Level

express

Languages:

translate

Machines:

Low Level

Admiral Grace Hopper (1906-1992)
(Photo: via Wikipedia)

Could we program a computer to do this?

human ingenuity required

Ideas:

High Level

express

Languages:

translate

Machines:

Low Level

Admiral Grace Hopper (1906-1992)
(Photo: via Wikipedia)

Could we program a computer to do this?

Yes! The A-0 system for UNIVAC I (1951-52): the first **compiler**

Ideas:

High Level

express

Languages:

translate

Machines:

Low Level

Admiral Grace Hopper (1906-1992)
(Photo: via Wikipedia)

compiler construction

human ingenuity required

Ideas:

High Level

human ingenuity required

express

Languages:

language design

compiler construction

translate

Machines:

human ingenuity required

Low Level

Languages and tools matter

- Language designs empower developers to:
  - Express their ideas more directly
  - Execute their designs on a computer
- Better tools (compilers, interpreters, etc.) will:
  - open programming to more people and more applications
  - increase programmer productivity
  - enhance software quality (functionality, reliability, security, performance, power, ...)

Basics of Compiler Structure

How does a compiler work?

source program

```
// A simple mini test program

int i = 0;    // initialize
while (i <= 10) {
  print i*i;  // print a square
  i = i + 1;
}
```

compile

target program

```
        .file   "squares.s"
        .comm   _esp0,4
        .globl  _Main_main
_Main_main:
        pushl   %ebp
        movl    %esp,%ebp
        subl    $4,%esp
        movl    $0,%eax
        movl    %eax,-4(%ebp)
        jmp     l1
l0:
        movl    -4(%ebp),%eax
        movl    -4(%ebp),%ebx
        imull   %ebx,%eax
        movl    %esp,_esp0
        subl    $4,%esp
        andl    $0xfffffff0,%esp
        movl    %eax,(%esp)
        call    _print
        movl    _esp0,%esp
        movl    $1,%eax
        movl    -4(%ebp),%ebx
        addl    %ebx,%eax
        movl    %eax,-4(%ebp)
l1:
        movl    $10,%eax
        movl    -4(%ebp),%ebx
        cmpl    %eax,%ebx
        jle     l0
        movl    %ebp,%esp
        popl    %ebp
        ret
```

We need to describe this process in a way that is scalable, precise, mechanical/algorithmic, ...

What is this?

False

Dark pixels on a light background

A collection of lines/strokes

A sequence of characters

A single word ("token")

An expression

A boolean expression

A truth value

One thing can be seen in many different ways

We can break a complex process into multiple (hopefully simpler) steps

"Compiling" English

- The symbols must be valid: ✘ source input
  hdk fΩfdh ksdßs dfsjf dslkjé
- The words must be valid: ✘ lexical analysis
  banana jubmod food funning
- The text must use correct grammar: ✘ parser
  my walking up left tree dog

"Compiling" English

- The phrase must make sense ✘
  This sentence is not true.
- The phrase must not be ambiguous ✘
  Look at the monkey with a telescope!
- The sentence must fit in context ✘
  My next song is about geography.

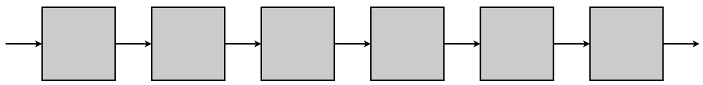
static analysis

- Finally, we have valid abstract syntax! ✓
  Languages are very interesting.

ready for "analysis"
ready for "code generation"

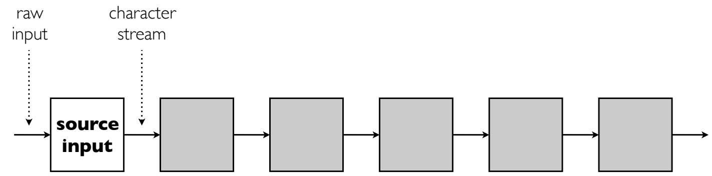
The compiler pipeline

- Traditionally, the task of compilation is broken down into several steps, or compilation phases:



Source input

(not a standard term)

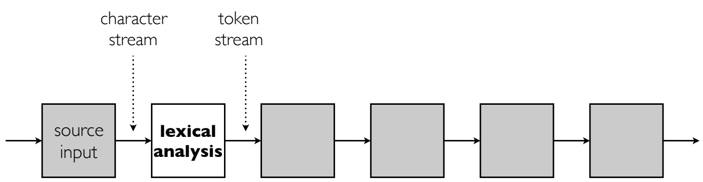


- Turn data from a raw input source into a sequence of characters or lines

  Data might come from a disk, memory, a keyboard, a network, a thumb drive, ...

  The operating system usually takes care of most of this ...

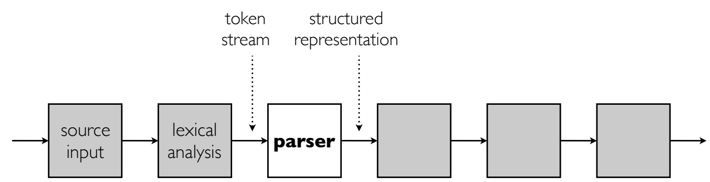
Lexical analysis



- Convert the input stream of characters into a stream of tokens
- For example, the keyword `for` is treated as a single token, and not as three separate characters
- "lexical":

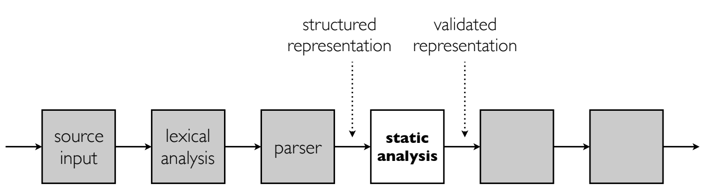
  "of or relating to the words or vocabulary of a language"

Parser



- Build data structures that capture the underlying structure (abstract syntax) of the input program
- Determines whether inputs are grammatically well-formed (and reports a syntax error when they are not)

Static analysis



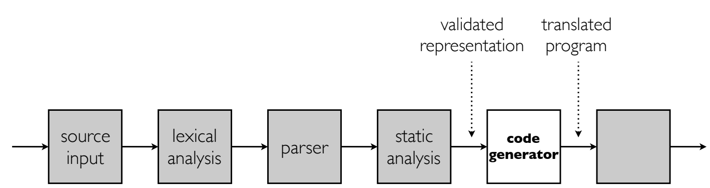
- Check that the program is reasonable:

  no references to unbound variables

  no type inconsistencies

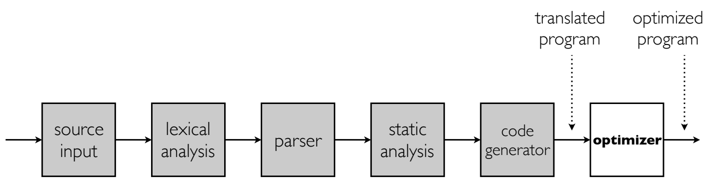
  etc...

Code generation



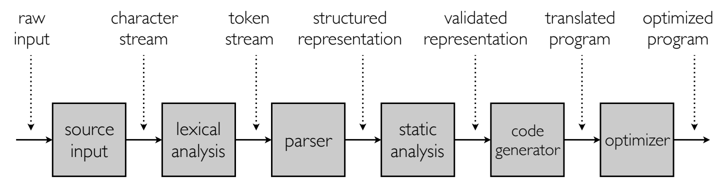
- Generate an appropriate sequence of machine instructions as output
- Different strategies are needed for different target machines

Optimization



- Look for opportunities to improve the quality of the output code:
  - There may be conflicting ways to "improve" a given program; the choice depends on the context/the user's priorities
  - Producing genuinely "optimal" code is theoretically impossible; "improved" is as good as it gets!

The full pipeline



- 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 passes)
  - additional data may be passed between phases

Front ends and back ends, revisited

- Front end: those parts of a compiler that depend most heavily on the source language
  - Source input, lexical analysis, parsing, static analysis
  - For detailed exploration of these topics, consider CS 421
- Back end: those parts of a compiler that depend most heavily on the target language
  - Code generation, optimization, assembly
  - For detailed exploration of these topics, consider CS 422

Snapshots from a "mini" compiler pipeline

Snapshots from a "mini" compiler pipeline

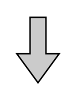
- In this section, we'll trace the results of passing the following program through a compiler for a language called "mini"
- A sample mini program:

```
// A simple mini test program

int i = 0;    // initialize
while (i <= 10) {
  print i*i;  // print a square
  i = i + 1;
}
```

- The goal here is just to get a sense of how compiler phases work together in practice; you don't need to understand all of the fine details

Source input (as numbers)

```
// A simple mini test program

int i = 0;    // initialize
while (i <= 10) {
  print i*i;  // print a square
  i = i + 1;
}
```



```
|47|47|32|65|32|115|105|109|112|108|101|32|77|105|110|105|
32|116|101|115|116|32|112|114|111|103|114|97|109|10|10|105|
110|116|32|105|32|61|32|48|59|32|32|32|32|47|47|32|105|110|
105|116|105|97|108|105|122|101|10|119|104|105|108|101|32|40|
105|32|60|61|32|49|48|41|32|123|10|32|32|112|114|105|110|
116|32|105|42|105|59|32|32|47|47|32|112|114|105|110|116|32|
97|32|115|113|117|97|114|101|10|32|32|105|32|61|32|105|32|
43|32|49|59|10|125|10|
```

Source input (as characters)

```
|47|47|32|65|32|115|105|109|112|108|101|32|109|105|110|105|
32|116|101|115|116|32|112|114|111|103|114|97|109|10|10|105|
110|116|32|105|32|61|32|48|59|32|32|32|32|47|47|32|105|110|
105|116|105|97|108|105|122|101|10|119|104|105|108|101|32|40|
105|32|60|61|32|49|48|41|32|123|10|32|32|112|114|105|110|
116|32|105|42|105|59|32|32|47|47|32|112|114|105|110|116|32|
97|32|115|113|117|97|114|101|10|32|32|105|32|61|32|105|32|
43|32|49|59|10|125|10|
```

```
|/|/| |A| |s|i|m|p|l|e| |m|i|n|i| |t|e|s|t| |p|r|o|g|r|a|m|\n
|\n
|i|n|t| |i| |=| |0|;| | | | |/|/| |i|n|i|t|i|a|l|i|z|e|\n
|w|h|i|l|e| |(|i| |<|=| |1|0|)| |{|\n
| | |p|r|i|n|t| |i|*|i|;| | |/|/| |p|r|i|n|t| |a| |s|q|u|a|r|e|\n
| | |i| |=| |i| |+| |1|;|\n
|}|\n
|\n
```

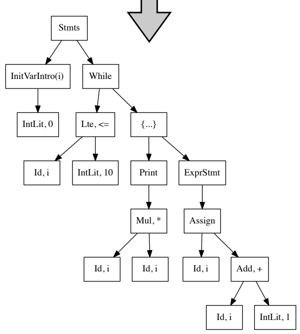

Lexical analysis

```
|/|/| |A| |s|i|m|p|l|e| |m|i|n|i| |t|e|s|t| |p|r|o|g|r|a|m|\n
|\n
|i|n|t| |i| |=| |0|;| | | | |/|/| |i|n|i|t|i|a|l|i|z|e|\n
|w|h|i|l|e| |(|i| |<|=| |1|0|)| |{|\n
| | |p|r|i|n|t| |i|*|i|;| | |/|/| |p|r|i|n|t| |a| |s|q|u|a|r|e|\n
| | |i| |=| |i| |+| |1|;|\n
|}|\n
|\n
```

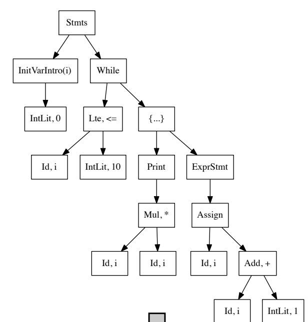
```
| INT | ID(i) | = | INTLIT(0) | Semicolon ";" | WHILE
| Open parenthesis "(" | ID(i) | <= | INTLIT(10)
| Close parenthesis ")" | Open brace "{" | PRINT | ID(i)
| * | ID(i) | Semicolon ";" | ID(i) | = | ID(i) | +
| INTLIT(1) | Semicolon ";" | Close brace "}" |
```


Parsing

```
| INT | ID(i) | = | INTLIT(0) | Semicolon ";" | WHILE
| Open parenthesis "(" | ID(i) | <= | INTLIT(10)
| Close parenthesis ")" | Open brace "{" | PRINT | ID(i)
| * | ID(i) | Semicolon ";" | ID(i) | = | ID(i) | +
| INTLIT(1) | Semicolon ";" | Close brace "}" |
```

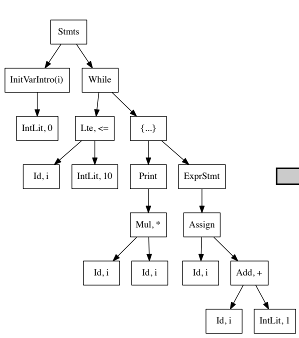


Static analysis




```
{ (i, int) }
```


Code generation




```
        .file   "squares.s"
        .comm   _esp0,4
        .globl  _Main_main
_Main_main:
        pushl   %ebp
        movl    %esp,%ebp
        subl    $4,%esp
        movl    $0,%eax
        movl    %eax,-4(%ebp)
        jmp     l1
l0:
        movl    -4(%ebp),%eax
        movl    -4(%ebp),%ebx
        imull   %ebx,%eax
        movl    %esp,_esp0
        subl    $4,%esp
        andl    $0xfffffff0,%esp
        movl    %eax,(%esp)
        call    _print
        movl    _esp0,%esp
        movl    $1,%eax
        movl    -4(%ebp),%ebx
        addl    %ebx,%eax
        movl    %eax,-4(%ebp)
l1:
        movl    $10,%eax
        movl    -4(%ebp),%ebx
        cmpl    %eax,%ebx
        jle     l0
        movl    %ebp,%esp
        popl    %ebp
        ret
```


Assembly

```
        .file   "squares.s"
        .comm   _esp0,4
        .globl  _Main_main
_Main_main:
        pushl   %ebp
        movl    %esp,%ebp
        subl    $4,%esp
        movl    $0,%eax
        movl    %eax,-4(%ebp)
        jmp     l1
l0:
        movl    -4(%ebp),%eax
        movl    -4(%ebp),%ebx
        imull   %ebx,%eax
        movl    %esp,_esp0
        subl    $4,%esp
        andl    $0xfffffff0,%esp
        movl    %eax,(%esp)
        call    _print
        movl    _esp0,%esp
        movl    $1,%eax
        movl    -4(%ebp),%ebx
        addl    %ebx,%eax
        movl    %eax,-4(%ebp)
l1:
        movl    $10,%eax
        movl    -4(%ebp),%ebx
        cmpl    %eax,%ebx
        jle     l0
        movl    %ebp,%esp
        popl    %ebp
        ret
```

```
$ od -A x -t x1 squares.o
0000000    ce  fa  ed  fe  07  00  00  00  03  00  00  00  01  00  00  00
0000010    03  00  00  00  e4  00  00  00  00  00  00  00  01  00  00  00
0000020    7c  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
0000030    00  00  00  00  00  00  00  00  87  00  00  00  00  01  00  00
0000040    87  00  00  00  07  00  00  00  07  00  00  00  01  00  00  00
0000050    00  00  00  00  5f  5f  74  65  78  74  00  00  00  00  00  00
0000060    00  00  00  00  5f  5f  54  45  58  54  00  00  00  00  00  00
0000070    00  00  00  00  00  00  00  00  87  00  00  00  00  01  00  00
0000080    00  00  00  00  88  01  00  00  08  00  00  00  00  04  00  80
0000090    00  00  00  00  00  00  00  00  02  00  00  00  18  00  00  00
00000a0    c8  01  00  00  05  00  00  00  04  02  00  00  20  00  00  00
00000b0    0b  00  00  00  50  00  00  00  00  00  00  00  02  00  00  00
00000c0    02  00  00  00  01  00  00  00  03  00  00  00  02  00  00  00
00000d0    00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
*
0000100    55  89  e5  83  ec  08  b8  00  00  00  00  89  45  fc  b8  00
0000110    00  00  00  89  45  f8  e9  3b  00  00  00  8b  45  fc  8b  5d
0000120    fc  0f  af  c3  89  25  00  00  00  00  83  ec  04  83  e4  f0
0000130    89  04  24  e8  c8  ff  ff  ff  8b  25  00  00  00  00  8b  45
0000140    fc  8b  5d  f8  01  d8  89  45  f8  b8  01  00  00  00  8b  5d
0000150    fc  01  d8  89  45  fc  b8  0a  00  00  00  8b  5d  fc  39  c3
0000160    0f  8e  b5  ff  ff  ff  8b  45  f8  89  25  00  00  00  00  83
0000170    ec  04  83  e4  f0  89  04  24  e8  83  ff  ff  ff  8b  25  00
0000180    00  00  00  89  ec  5d  c3  00  7f  00  00  00  03  00  00  0c
0000190    79  00  00  00  04  00  00  0d  6b  00  00  00  03  00  00  0c
00001a0    62  00  00  00  01  00  00  05  3a  00  00  00  03  00  00  0c
00001b0    34  00  00  00  04  00  00  0d  26  00  00  00  03  00  00  0c
00001c0    17  00  00  00  01  00  00  05  19  00  00  00  0e  01  00  00
00001d0    56  00  00  00  1c  00  00  00  0e  01  00  00  1b  00  00  00
00001e0    07  00  00  00  0f  01  00  00  00  00  00  00  01  00  00  00
00001f0    01  00  00  00  04  00  00  00  12  00  00  00  01  00  00  00
0000200    00  00  00  00  00  5f  65  73  70  30  00  5f  4d  61  69  6e
0000210    5f  6d  61  69  6e  00  5f  70  72  69  6e  74  00  6c  31  00
0000220    6c  30  00  00
0000224
```

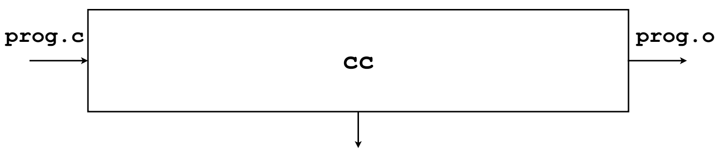
Modularity in compiler design

Modularity

- Modularity is all about building large systems from collections of smaller components
- Modular implementations can be easier to write, test, debug, understand, and maintain than monolithic implementations
- For example:
  - Components can be developed independently
  - Some components can be reused in other contexts
  - Some components may even be useful as standalone tools

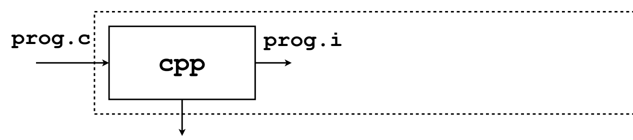
Combining compilers

- The classic Unix C compiler, `cc`, is implemented by a pipeline of compilers:



Combining compilers

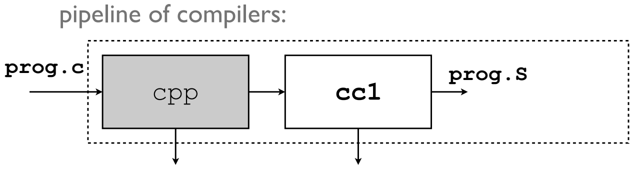
- The classic Unix C compiler, `cc`, is implemented by a pipeline of compilers:



cpp: the C preprocessor, expands the use of macros and compiler directives in the source program

Combining compilers

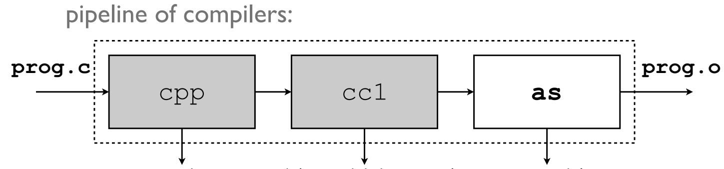
- The classic Unix C compiler, `cc`, is implemented by a pipeline of compilers:



`cc1`: the main C compiler, which translates C code to the assembly language for a particular machine

Combining compilers

- The classic Unix C compiler, `cc`, is implemented by a pipeline of compilers:



`as`: the assembler, which translates assembly language programs into machine code

Advantages of modularity

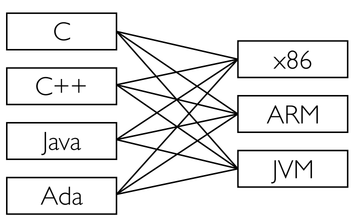
- Some components (e.g., `as`) are useful in their own right
- Some components can be reused (e.g., replace `cc1` to build a C++ compiler)
- Some components (e.g., `cpp`) are machine independent, so they do not need to be rewritten for each new machine
- Modular implementations can be easier to write, test, debug, understand, and maintain

Disadvantages of modularity?

- Performance
  - It takes extra time to write out the data produced at the end of each stage
  - It takes extra time to read it back in at the beginning of the next stage
  - Later stages may need to repeat calculations from earlier stages if the information that they need is not included in the output of those earlier stages
- But modern machines and disks are pretty fast, and compilers are often complex, so modularity usually wins!

Multiple languages and targets

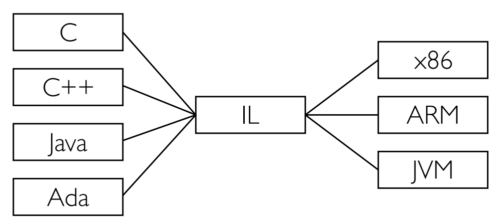
- Suppose that we want to write compilers for n different languages, with m different target platforms.



- That's n x m different compilers!

An intermediate language

- Alternatively: design a general purpose, shared "intermediate language":



- Now we only have n front ends and m back ends to write!
- The biggest challenge is to find an intermediate language that is general enough to accommodate a wide range of languages and machine types

Summary

- Basic principles
  - programs as data
  - 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