CS 321: Languages and Compiler Design I

Mark P Jones, Portland State University

Winter 2014

Week 1: Basics of Compiler Structure

Why?

Ideas: High Level

- Play a game
- Display an image
- Search a database
- Visit a web page
- etc ...

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

Machines: Low Level

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

Ideas: High Level

- Play a game
- Display an image
- Search a database
- Visit a web page
- etc ...

express

Languages:

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

translate

Machines: Low Level

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

Ideas: High Level

express

Languages:

translate

Machines: Low Level

Admiral Grace Hopper
(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
(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: → express → Languages: → translate → Machines:

High Level

compiler construction

Low Level

human ingenuity required

Admiral Grace Hopper
(Photo: via Wikipedia)

Ideas: → express → Languages: → translate → Machines:

human ingenuity required

High Level

language design

compiler construction

Low Level

human ingenuity required

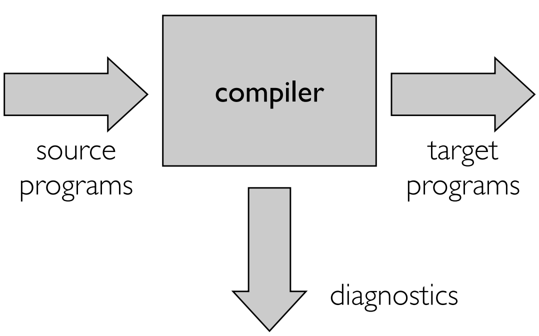
A Big Picture Vision for CS 321/322:

- Build foundations in language design & compiler construction
- Empower developers to:
  - Express their ideas more directly
  - Execute their designs on a computer
- The long term goal is to build better tools that:
  - open programming to more people and more applications
  - increase programmer productivity
  - enhance software quality (functionality, reliability, security, performance, power, ...)

Basics of Compiler Structure

What is a compiler?

Compilers are translators:



Compilers, compilers, compilers, ...

Compilers show up in many different forms:

- Translating programs in high-level languages like C, Fortran, Ada, etc… to executable machine code
- Just in time compilers: translating byte code to machine code at runtime
- Rendering an HTML web page in a browser window
- Printing a document on a Postscript printer
- Generating audio speech from written text
- Translating from English to Spanish/French/…
- …

The tools, techniques, and concepts that we cover are more general than you might have thought!

Compiler inputs and outputs

source programs

many possible source languages, from traditional, to application specific languages.

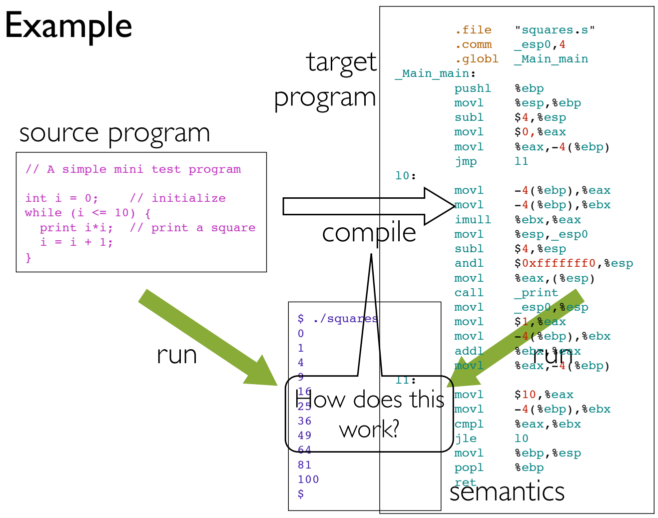
target programs

usually another programming language, often the machine language of a particular computer system.

error diagnostics

essential for serious program development.

Example



Critical properties of a compiler

Always:

- **Correctness**: the compiler should produce valid output for any valid input program, and the output should have the same semantics as the input

Almost Always:

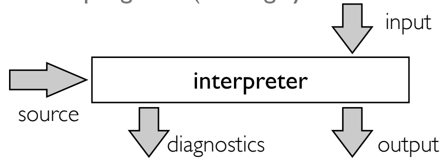
- **Performance** of compiled code: time, space, ...
- **Performance** of compiler: time, space, ...
- **Diagnostics**: to permit early and accurate diagnosis and detection of programming errors

Other desirable features, in practice

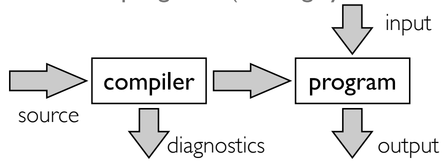
- Support for large programming projects, including:
  - **Separate compilation**, reducing the amount of recompilation that is needed when part of a program is changed
  - Use of libraries, enabling effective **software reuse**
- Convenient **development environment**:
  - Supports program development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

Interpreters vs compilers

- Interpreters run 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:
  - Use your calculator to evaluate (1+2)+(3+4):

    Answer: 10

  - Tell me what buttons to press to evaluate (1+2)+(3+4):

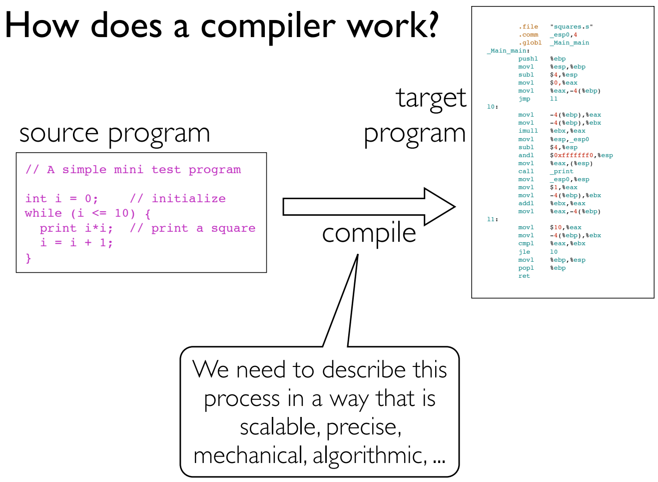
    Answer: $\boxed{1}$ $\boxed{+}$ $\boxed{2}$ $\boxed{=}$ $\boxed{M}$ $\boxed{3}$ $\boxed{+}$ $\boxed{4}$ $\boxed{+}$ $\boxed{MR}$ $\boxed{=}$

- We'll mostly focus on compilers, but will also use many of the same tools to work with interpreters.

Language vs implementation

- Be very careful to distinguish between languages and their implementations
- It doesn't make much sense to talk about a "slow language"; speed is a property of the implementation, not the language.
- It doesn't make much sense to talk about a "compiled language"; again, "compiled" is a detail of the implementation, not the language

How does a compiler work?



What is this?

41+1

Black pixels on a white background

A sequence of characters

A sequence of "tokens"

An expression

A valid expression

Meaning: 42

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 sentence must use correct grammar: ✘ parser
  my walking up left tree dog

"Compiling" English

- The sentence must make sense ✘
  This sentence is not true
- The sentence must not be ambiguous ✘
  Fruit flies like a banana
- The sentence must fit in context ✘
  This lecture is about geography

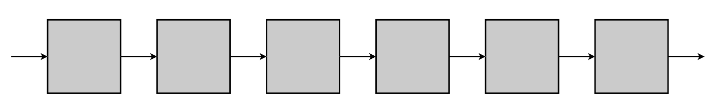
static analysis

- Finally, we are ready to translate! ✓
  Compilers are very interesting

ready for "code generation"
(i.e., for CS 322)

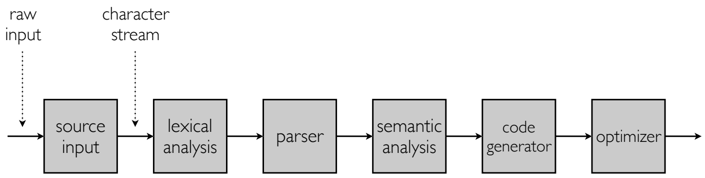
The compiler pipeline

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



Source input

(not a standard term)

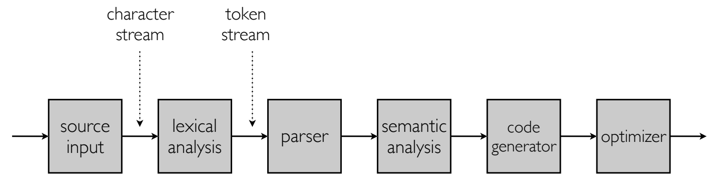


- Turn data from a raw input source into a sequence of characters or lines

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

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

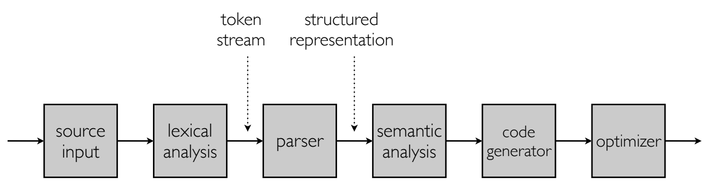
Lexical analysis



- 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":

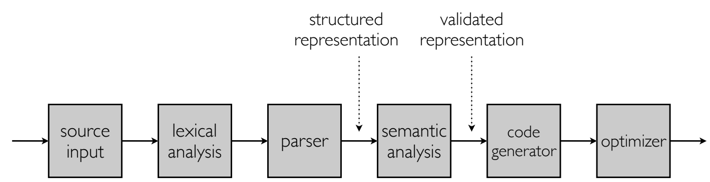
  "of or relating to the words or vocabulary of a language"

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)

Semantic analysis



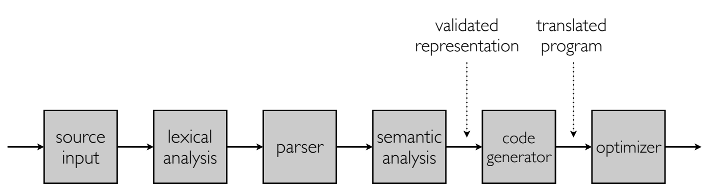
- Check that the program is reasonable:

  no references to unbound variables

  no type inconsistencies

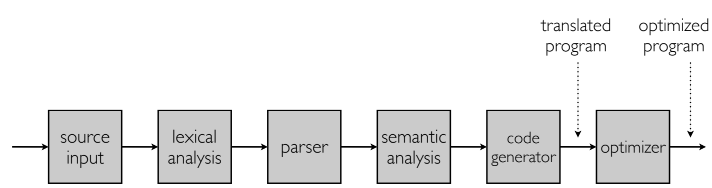
  etc...

Code generation



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

Optimization

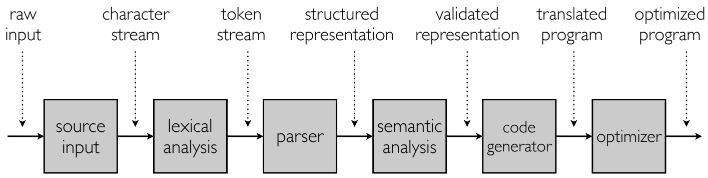


- Look for opportunities to improve the quality of the output code:

  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

Phases and passes

- A phase is a *logical* stage in a compiler pipeline
- A pass is a *physical* traversal over the representation of a program
- Several phases may be combined in one pass
- Passes may be run in sequence or in parallel
- Some languages are specifically designed so that they can be implemented in a single pass

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 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
- We'll see more about the "mini" compiler in this week's lab session

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|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|
```

```
|/|/| |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
```

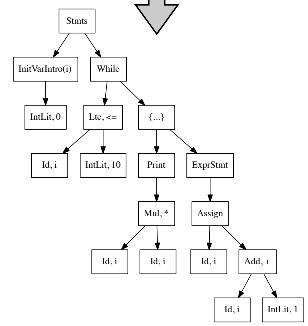
Lexical analysis

```
|/|/| |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
```

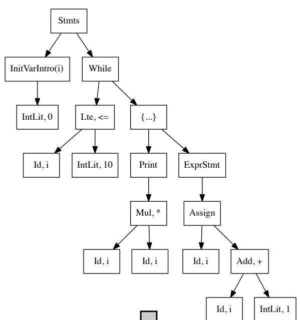
```
| INT | ID(i) | = | INTLIT(0) | Semicolon ";" | WHILE
| Open parenthesis "(" | ID(i) | <= | INTLIT(10)
| Close parenthesis ")" | Open brace "{" | PRINT | ID(i)
| * | ID(i) | Semicolon ";" | ID(i) | = | ID(i) | +
| INTLIT(1) | Semicolon ";" | Close brace "}" |
```


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 "}" |
```

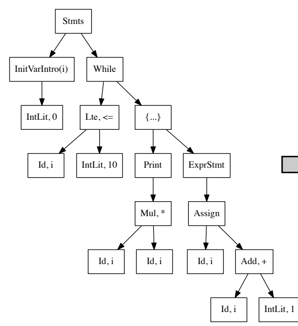


Static analysis




```
{ (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

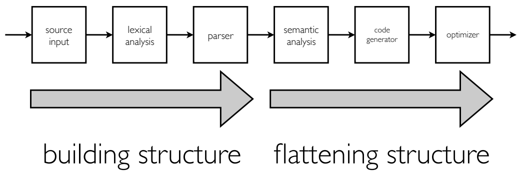
```
        .file   "squares.s"
        .comm   _esp0,4
        .globl  _Main_main
_Main_main:
        pushl   %ebp
        movl    %esp,%ebp
        subl    $4,%esp
        movl    $0,%eax
        movl    %eax,-4(%ebp)
        jmp     l1
l0:
        movl    -4(%ebp),%eax
        movl    -4(%ebp),%ebx
        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
```


Reflections

Reflections: flat vs structured

- We might prefer to think of them as texts, but the source and target programs here are really just "flat" sequences of numbers



Reflections: Syntax and semantics

- Syntax

  Written/spoken/symbolic/physical form; how things are communicated

- Semantics

  What those things mean

Syntax and Semantics

(written form) (meaning)

"41 + 1" ⇒ 42

Syntax and Semantics

(written form) (meaning)

$$\frac{A \wedge B}{A} \qquad \frac{A \wedge B}{B} \qquad \frac{A \quad B}{A \wedge B} \quad \Rightarrow$$

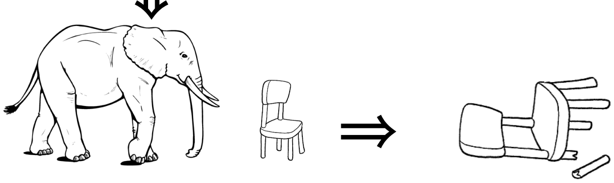
A	B	A∧B
F	F	F
F	T	F
T	F	F
T	T	T

Syntax and Semantics

(written form) (meaning)

"The elephant sat on a chair"

⇓



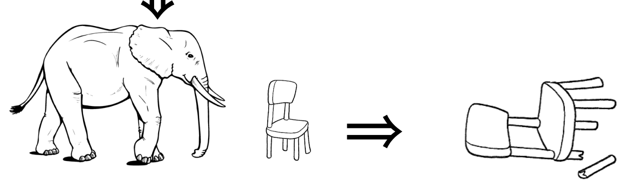
Syntax and Semantics

(spoken form) (meaning)

⇓

'The elephant sat on a chair"

⇓



Syntax and Semantics

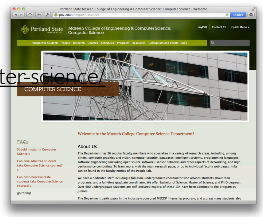
(written form) (meaning)

```
• <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML+RDFa 1.0//EN"
  "http://www.w3.org/MarkUp/DTD/xhtml-rdfa-1.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
version="XHTML+RDFa 1.0" dir="ltr"

  xmlns:content="http://purl.org/rss/1.0/modules/
content/"
  xmlns:dc="http://purl.org/dc/terms/"
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:og="http://ogp.me/ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:sioc="http://rdfs.org/sioc/ns#"
  xmlns:sioct="http://rdfs.org/sioc/types#"
  xmlns:skos="http://www.w3.org/2004/02/skos/core#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#">
...
<head profile="http://www.w3.org/1999/xhtml/vocab">
  ...
  <title>Portland State Maseeh College of Engineering
&amp; Computer Science: Computer Science | Welcome</
title>
  ...
</head>
<body class="html front not-logged-in no-sidebars page-
node page-node- page-node-1 node-type-page branding-tan-
gradient sidebar-left home" >
 ...
</body>
```

http://pdx.edu/computer-science/

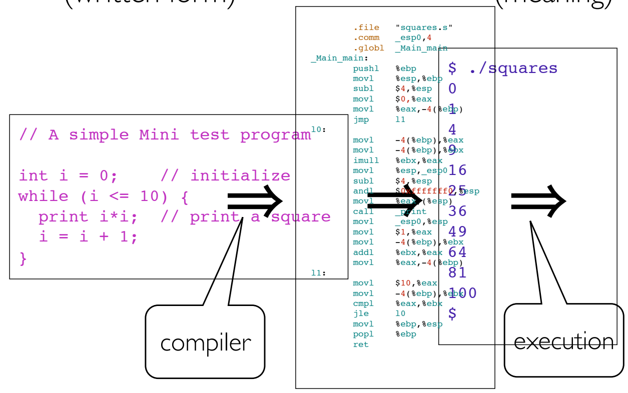
⇒



49

Syntax and Semantics

(written form) (meaning)



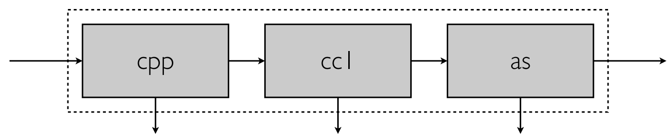
50

Modularity in compiler design

51

Combining compilers

- The standard Unix C compiler is structured as a pipeline of compilers:

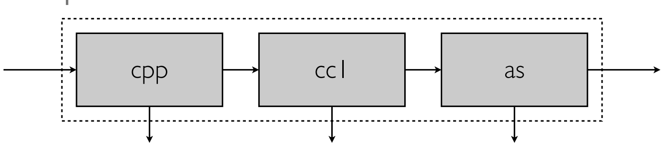


cpp: the C preprocessor, expands the use of macros and compiler directives in the source program

52

Combining compilers

- The standard Unix C compiler is structured as a pipeline of compilers:

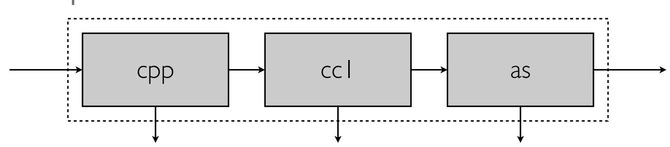


cc1: the main C compiler, which translates C code to the assembly language for a particular machine

53

Combining compilers

- The standard Unix C compiler is structured as a pipeline of compilers:



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

54

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 understand, test, debug, write, ...

Disadvantages of modularity

- Performance

  It takes extra time to write out the data produced at the end of each stage

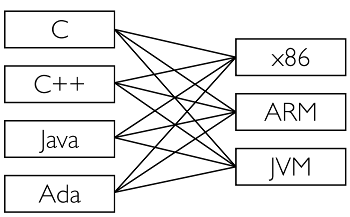
  It takes extra time to read it back in at the beginning of the next stage

  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

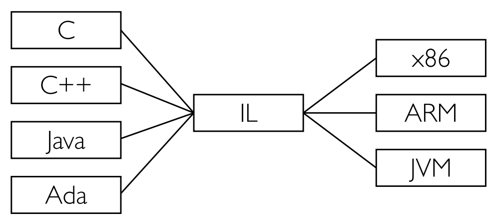
- Suppose that we want to write compilers for n different languages, with m different target platforms.



- 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!

Front ends and back ends

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

  Source input, lexical analysis, parsing, static analysis

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

  Code generation, optimization, assembly

- 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

  syntax and semantics
  correctness means preserving semantics

- The compiler pipeline

  source input, lexical analysis, parsing, static analysis, code generation, optimization

- Modularity

  Techniques for simplifying compiler construction tasks