Lecture 7

COP3402 FALL 2015 – DR. MATTHEW GERBER – 9/21/2015

FROM EURIPIDES MONTAGNE, FALL 2014

Tonight

- Compilers
- Interpreters
- Hybrid Systems

Programming Languages

A *programming language* is a notation for describing computations to people and to machines.

- More precisely, it is a notation that can be:
  - *Written* by people
  - *Clearly and unambiguously followed* by machines
- Programming languages can (hopefully) also be understood by people and generated by machines, but person-to-machine instruction is the primary purpose

Machines can follow computations written in programming languages using three different methods:

- Compilation
- Interpretation
- Hybrid Compilation/Interpretation

Compilers

In the most general sense, a *compiler*:

- Takes the *source code* written in a programming language as input, and
- Produces *machine language* as output

The "machine language" can take a few different forms…

- Native executable code
- Bytecode for a virtual machine
- Assembly-language text

It's generally clearest to think of it as assembly-language text

Compilers

The process takes place in several phases:

- The *front end* translates the source code into *intermediate code* that the compiler understands
- The *back end* translates the intermediate code into the actual machine language

Compilers: Front and Back End

The front end is *itself* multiple phases.

- The *lexical analyzer*, or *scanner*, takes the source stream of characters and breaks them down into individual *lexical units* or *tokens* – the smallest meaningful units in the programming language
- The *syntax analyzer*, or *parser*, takes the tokens and constructs a *parse tree* representing the syntactic structure of the program
- The *intermediate code generator* takes the parse tree, performs *semantic analysis*, and constructs *intermediate code* – a quasi-machine language understood by the compiler

Likewise, the back end.

- The *optimizer* examines the intermediate code and makes changes to it to make it smaller and faster – eliminating unused functionality and improving its performance
- The *code generator* translates the resulting intermediate code into machine language

Front End: Lexical Analysis

The scanner converts a stream of *characters* into a stream of tokens, along with a *symbol table*. Assume in the following that *fahrenheit* and *celsius* are already defined as **real** variables:

```
fahrenheit := 32 + celsius * 1.8;
```

Then we already have the symbol table shown here, and the scanner will produce:

ID	Name	Type
1	fahrenheit	real
2	celsius	real

[SYMBOL ID 1] [:=] [INT 32] [+] [SYMBOL ID 2] [*] [REAL 1.8] [;]

Note that:

- Each *variable name* is a token that becomes a symbol ID
- Each *literal value* is a token
- Each operator, special symbol and reserved word is a token
- Comments are removed at this phase

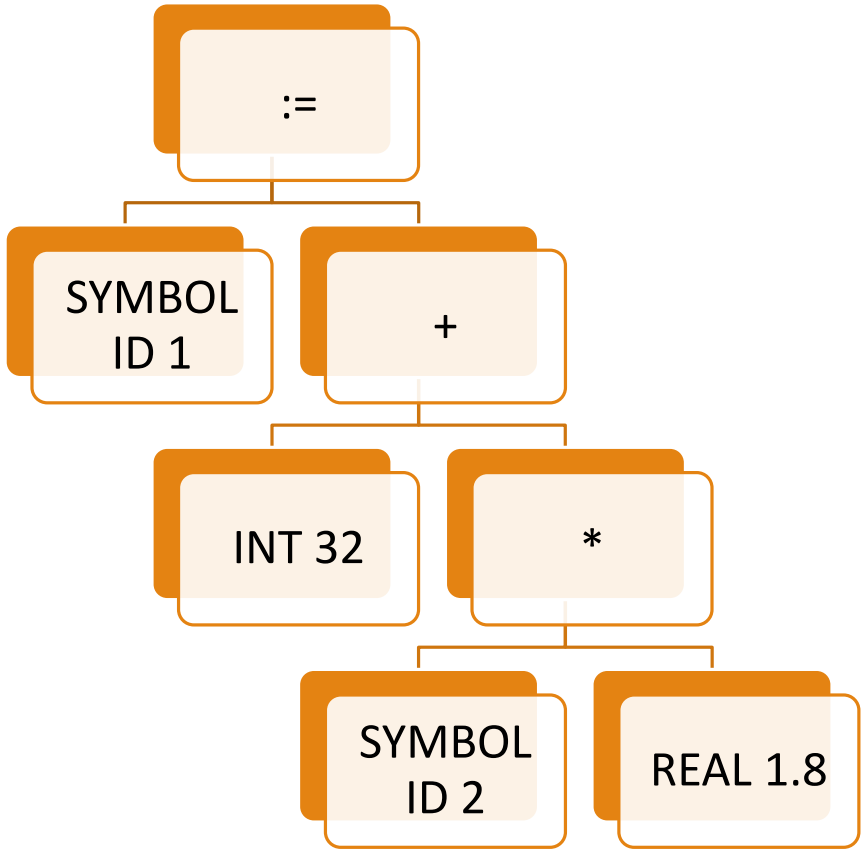
Front End: Syntax Analysis

Our lexical analyzer produced:

[SYMBOL ID 1] [:=] [INT 32] [+]
[SYMBOL ID 2] [*] [REAL 1.8] [;]

The parser will, from these tokens, create the parse tree shown to the right.

Note that the construction of the parse tree implies operator precedence.

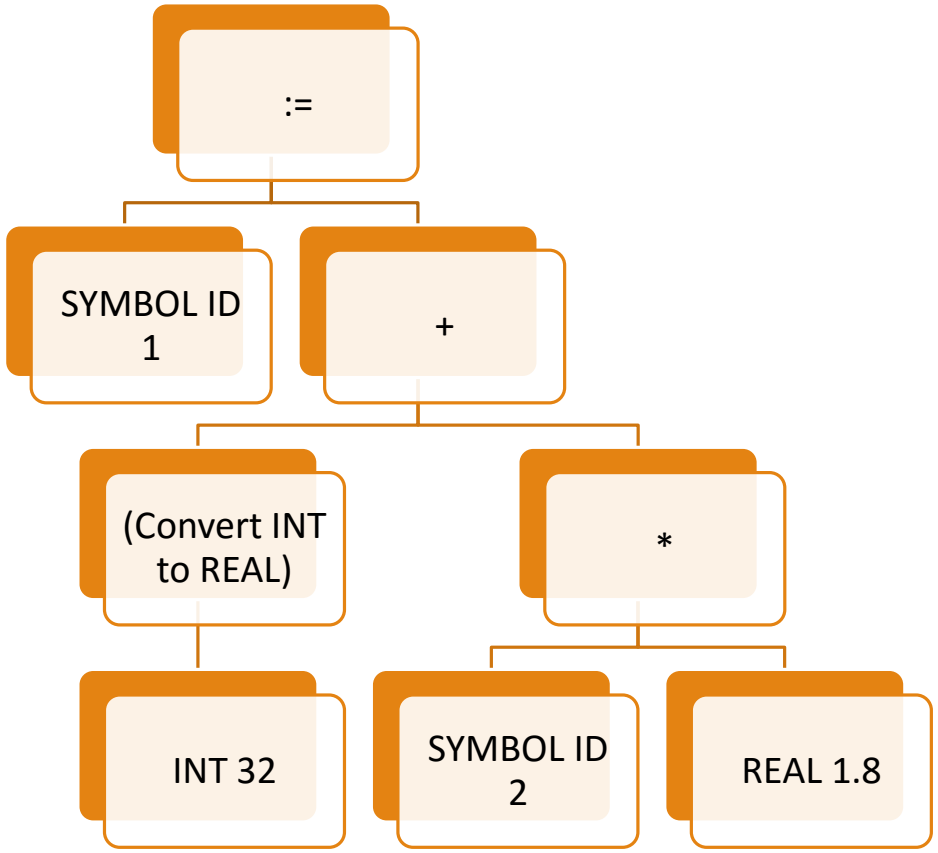


Front End: Context Analysis

As part of the syntax analysis, the parser performs *contextual* analysis.

This step handles things like automatic casts.

- In this case, there is a real variable coming off of the right side of the + operator
- The parser thereby knows that the left side needs to have a real variable as well
- The parser also knows a method to automatically convert an integer variable to a real variable
- If it doesn't have an automatic conversion, this is (**hopefully**) an error



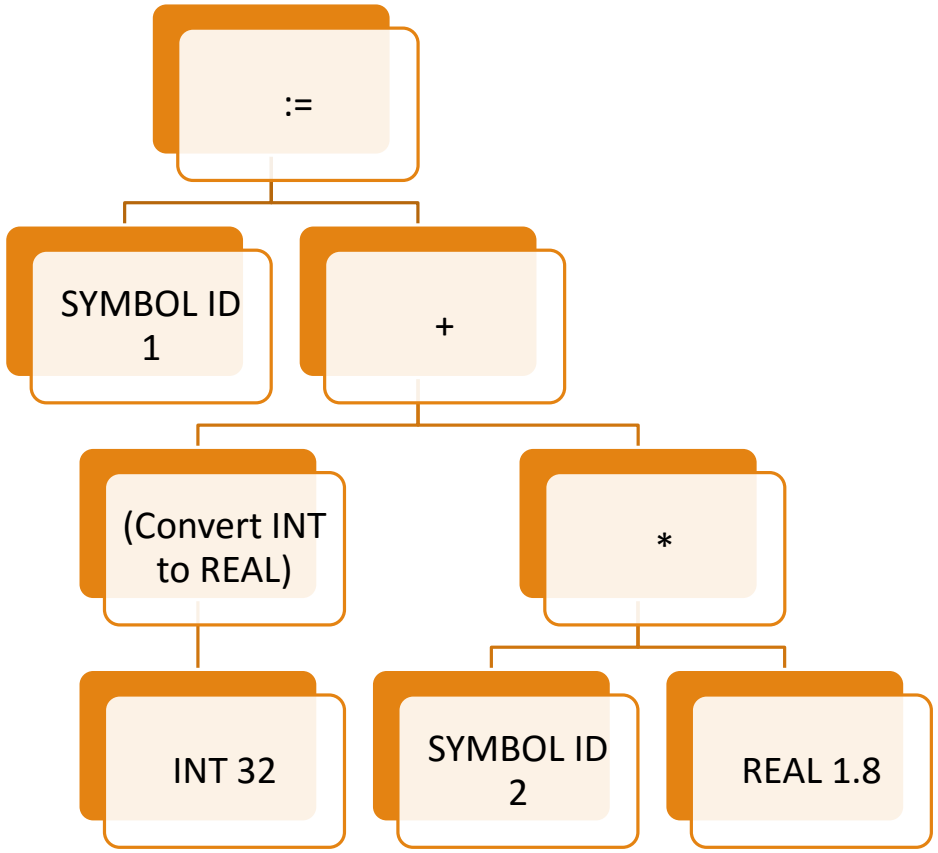
Front End: Intermediate Code Generation

Our parser generated the parse tree on the right.

Intermediate code for this tree might look like the following:

```
temp_1 := inttoreal(32)
temp_2 := [SYMBOL ID 2]
temp_2 := temp_2 * 1.8
temp_1 := temp_1 + temp_2
[SYMBOL ID 1] := temp_1
```

Note that only a single operation is performed by each step.



Back End: Optimization

Compare the two code snippets on the right.

- Clearly, they do the same thing
- Looking more closely, there is *no* difference in what they do, other than the difference between the temporary variables

The purpose of the optimizer is to get from the first to the second

- …And to do so for similar cases, all over the program
- Optimization can help you out a lot in cases of statement-level inefficiency like this
- It *cannot save you* from a bad *algorithm*

Optimization is in general beyond the scope of this course.

```
temp_1 := inttoreal(32)
temp_2 := [SYMBOL ID 2]
temp_2 := temp_2 * 1.8
temp_1 := temp_1 + temp_2
[SYMBOL ID 1] := temp_1
```

```
temp_1 := [SYMBOL ID 2]
temp_1 := temp_1 * 1.8
temp_1 := temp_1 + 32.0
[SYMBOL ID 1] := temp_1
```

Back End: Code Generation

On the right we see our optimized intermediate code, and two versions of what the machine language might look like.

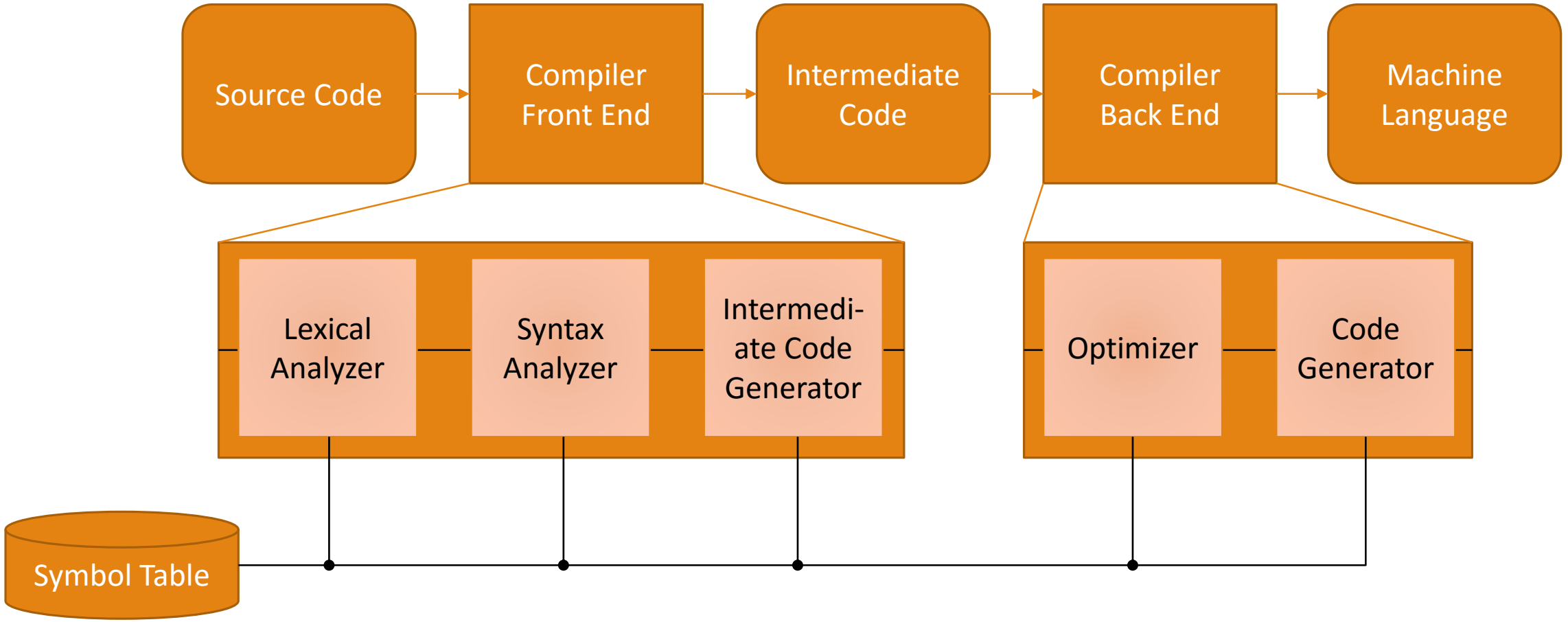
- The first version would be for a machine that allows literal operands to the MUL and ADD instructions
- The second version would be for a stricter register-file machine that does not
- If we know the characteristics of the machine, we can always translate intermediate code to machine language in straightforward fashion
  - (That's half the point of intermediate code to begin with)
- **Straightforward does not mean easy**

```
temp_1 := [SYMBOL ID 2]
temp_1 := temp_1 * 1.8
temp_1 := temp_2 + 32.0
[SYMBOL ID 1] := temp_1
```

```
mov     r1, [symbol_id_2]
mul     r1, #1.8
add     r1, #32.0
mov     [symbol_id_1], r1
```

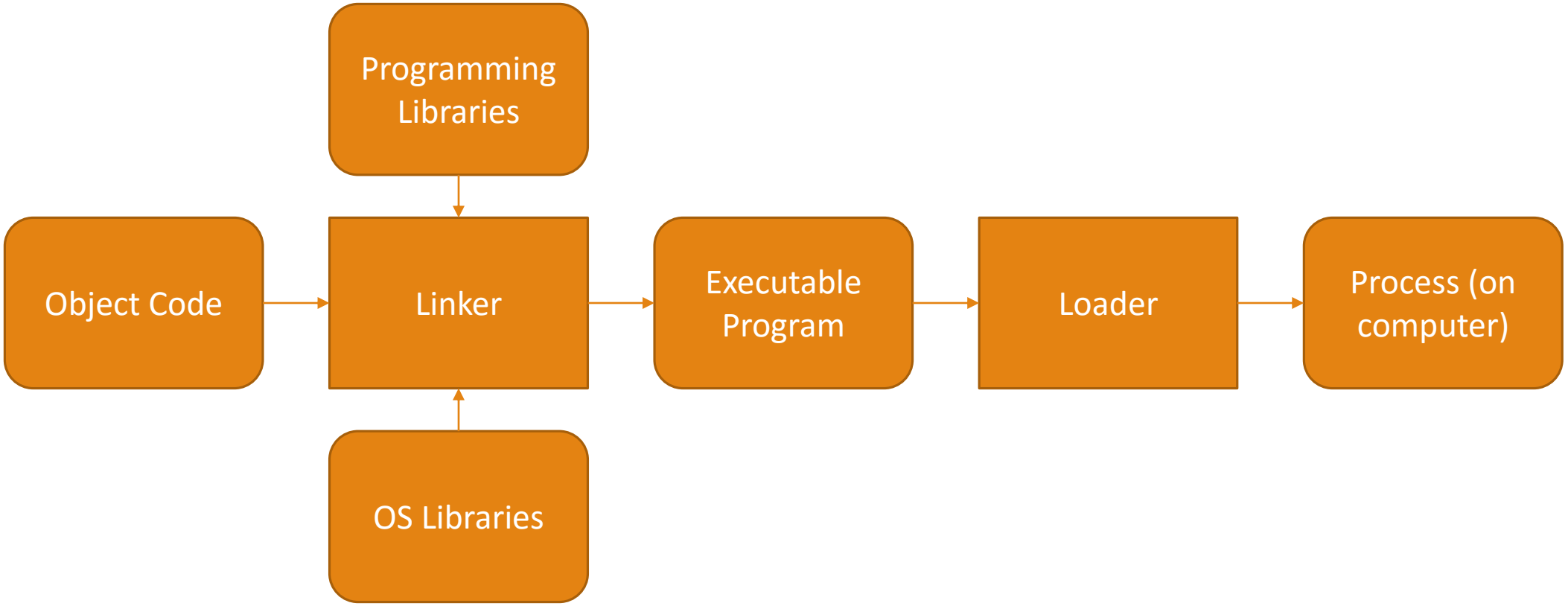
```
mov     r1, [symbol_id_2]
mov     r2, #1.8
mul     r1, r2
mov     r2, #32.0
add     r1, r2
mov     [symbol_id_1], r1
```

Putting the *Compiler* Together



Putting the *Program* Together

In a fully compiled environment the machine language either is, or becomes, object code.



Interpreters

In an *interpreted* environment, rather than being compiled to machine code and run by the computer, programs are run by *another program*.

- No compilation is necessary before running the program – which means all of the work to parse the program is done every time it is loaded
- Interpreted programs are very easy to debug – and run as much as two orders of magnitude slower than compiled programs

Examples of purely interpreted programming languages include:

- Most early versions of BASIC
- All Javascript engines prior to about 2008
- Perl prior to Perl version 6

Hybrid Compilation/Interpretation

Hybrid systems translate high-level language programs to an intermediate "bytecode" designed to allow fast interpretation by a virtual machine.

- Compilation is generally still required before running the program
- This compilation *can* be implicit

Just-in-Time compiling is used by some hybrid systems to enhance performance.

- When a method is first called, it is compiled from bytecode to native code and cached
- Reduces performance loss to the compilation plus the lack of per-instruction optimization

Examples of hybrid systems include:

- UCSD Pascal
- Java
- Perl 6
- .NET

Next Time: Compiling PL/0