Compiler Technology of Programming Languages

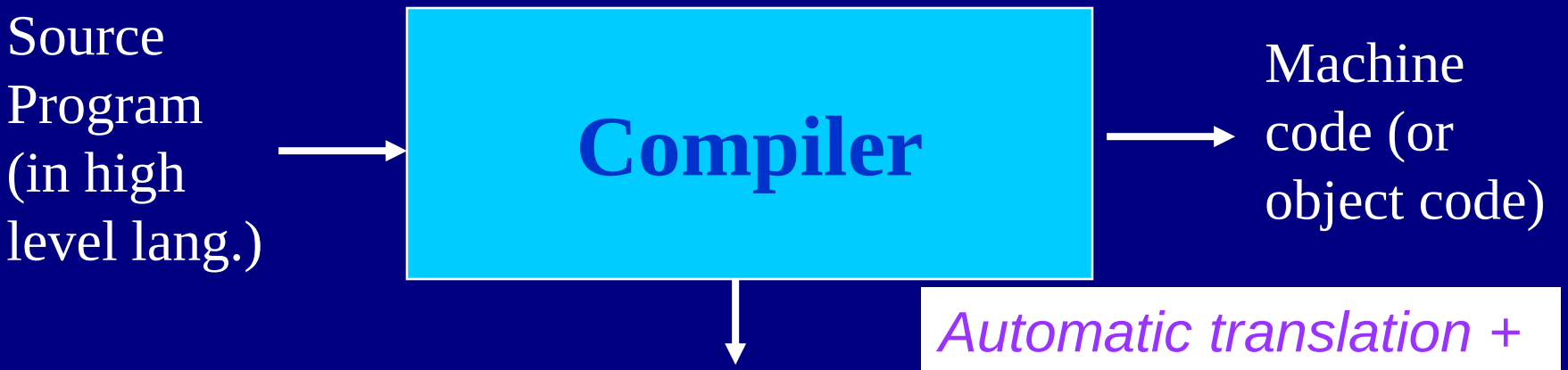
Chapter 1

Introduction

Prof. Farn Wang

What is a compiler? (original meaning)

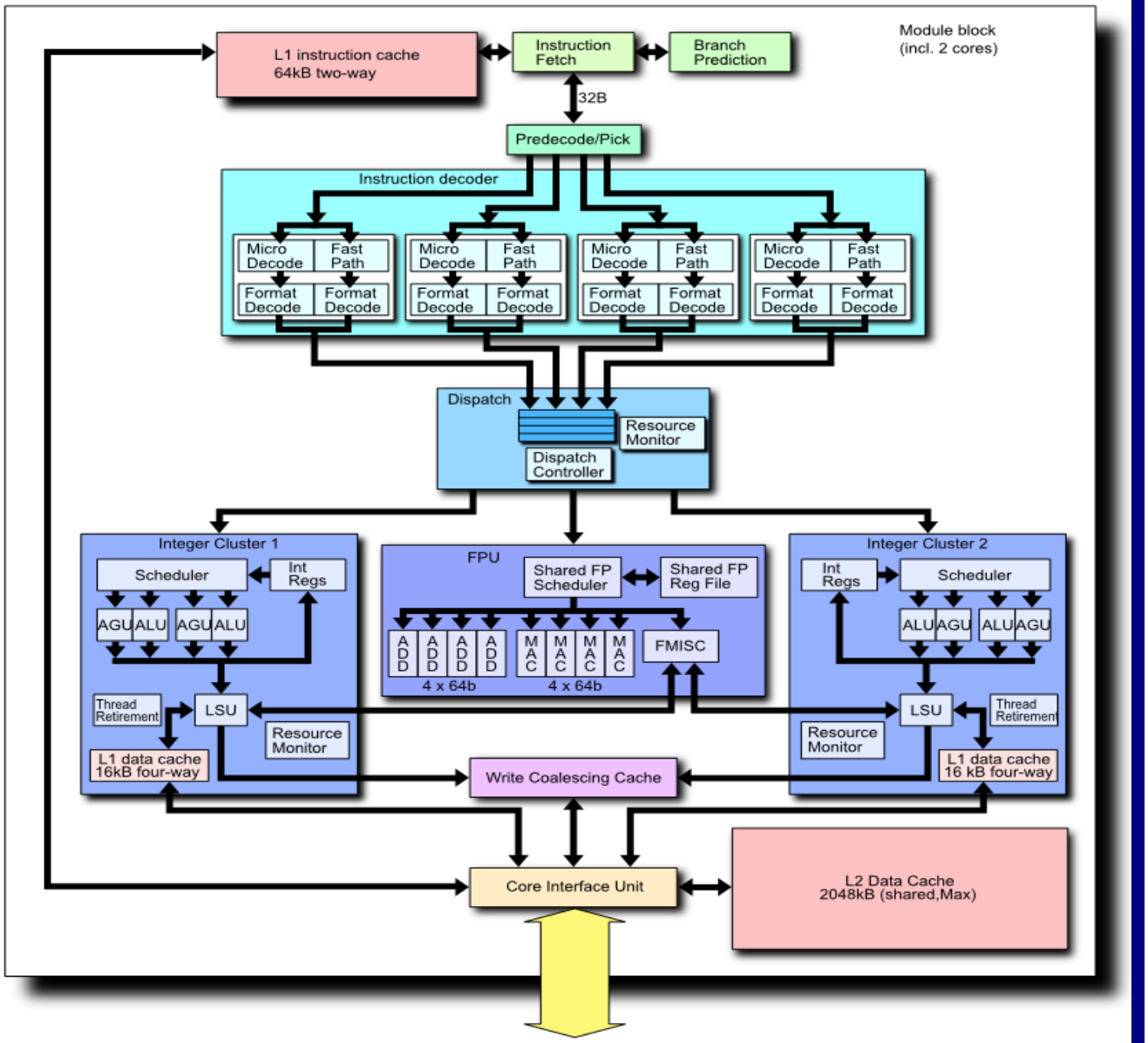
- reads a program written in the *source language* and
- translate it into an equivalent program in *machine language*.



Fallacy and Facts

Code in assembly language to obtain the highest performance

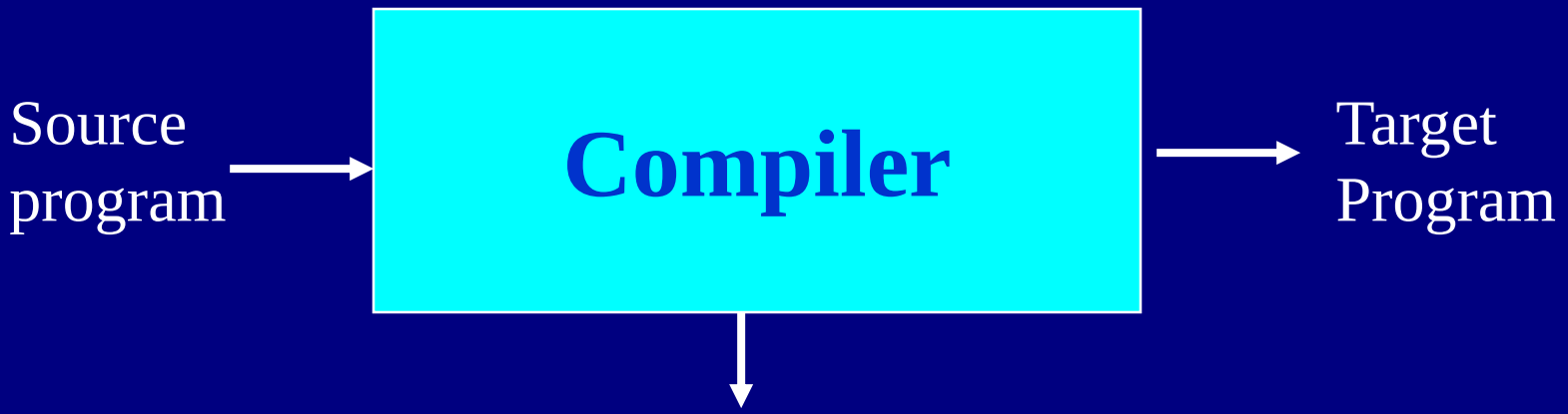
- Compilers register allocation, code scheduling, and many very sophisticated code optimization techniques.
  - Such optimizations by hand are error prone.
- Compilers usually have the better knowledge of the processor architecture.
- **Software crisis:** *High-level languages for productivity, maintainability, portability, ease of debugging, code reusability*.



- Micro-architectures are getting increasingly more complex, writing assembly code to obtain high performance is no longer a manageable task.
- Micro-architectures are company IP, for example, Intel would not give out all details of x86 micro-architectures.
- For dynamic, OOO processors, it is hard for a programmer to know how to write a high performance code. Should leave this job to experts.

What is a compiler? (broader meaning)

- reads a program written in a *source language* and
- translate it into an equivalent program in *target language*.



What Do Compilers Do?

- Source language:
  - High-level programming languages,
  - Text-formatting languages, (e.g. LaTex)
  - Logic-level specification languages (e.g. HDL)
  - Application binaries in one ISA (e.g. x86/ARM)
  - DataBase Query Languages
- Target language:
  - Machine languages,
  - Other high-level PL,
  - Typesetting commands,
  - VLSI layout
  - Other ISAs
  - DataBase Access Methods

Compiler technology can be broadly applied

Target machine code

- Pure machine code (i.e. ISA)
  - Assembly language format
  - Relocatable binary format
- Augmented machine code (augment with system calls, I/O, memory allocation, math functions, …) (i.e. ABI)
- Virtual machine code (e.g. P-code or byte code)
  - Good for transportable compilers **VISA**
  - Generated code need to be interpreted

Whatever target codes are, compilers do the same job: T*ranslation/Compilation*.

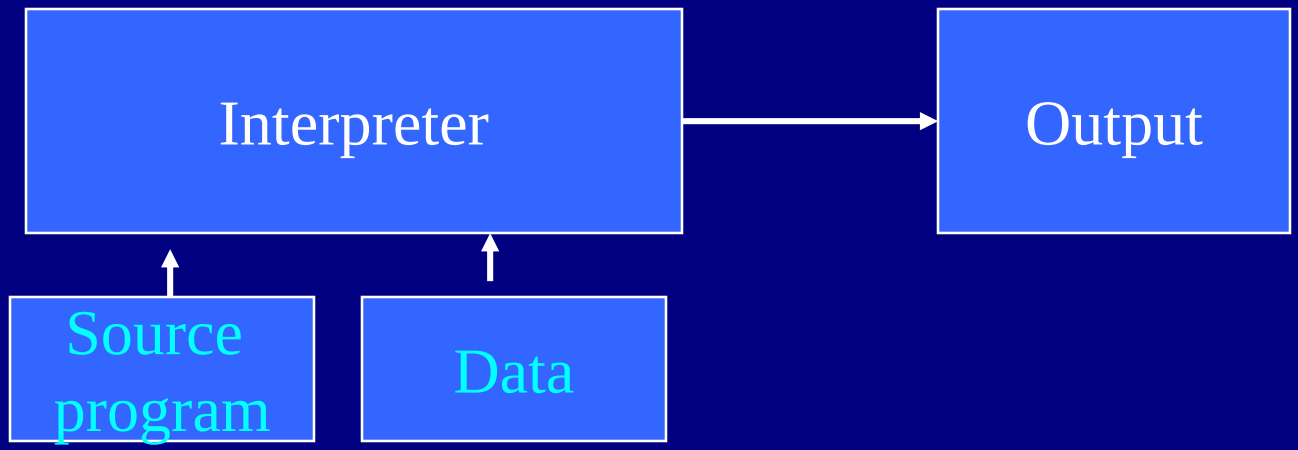
Analysis and Synthesis

Two parts to translation/compilation

- Analysis: breaking up the source program into pieces for syntax and semantic analysis, e.g. parsing.
  - widely used in many tools.
- Synthesis: constructing the desired target program, e.g. code generation.
  - can be very complex, such as the optimizer.

Interpreter and Compiler

- Interpreter: executes programs without explicitly performing target code generation.

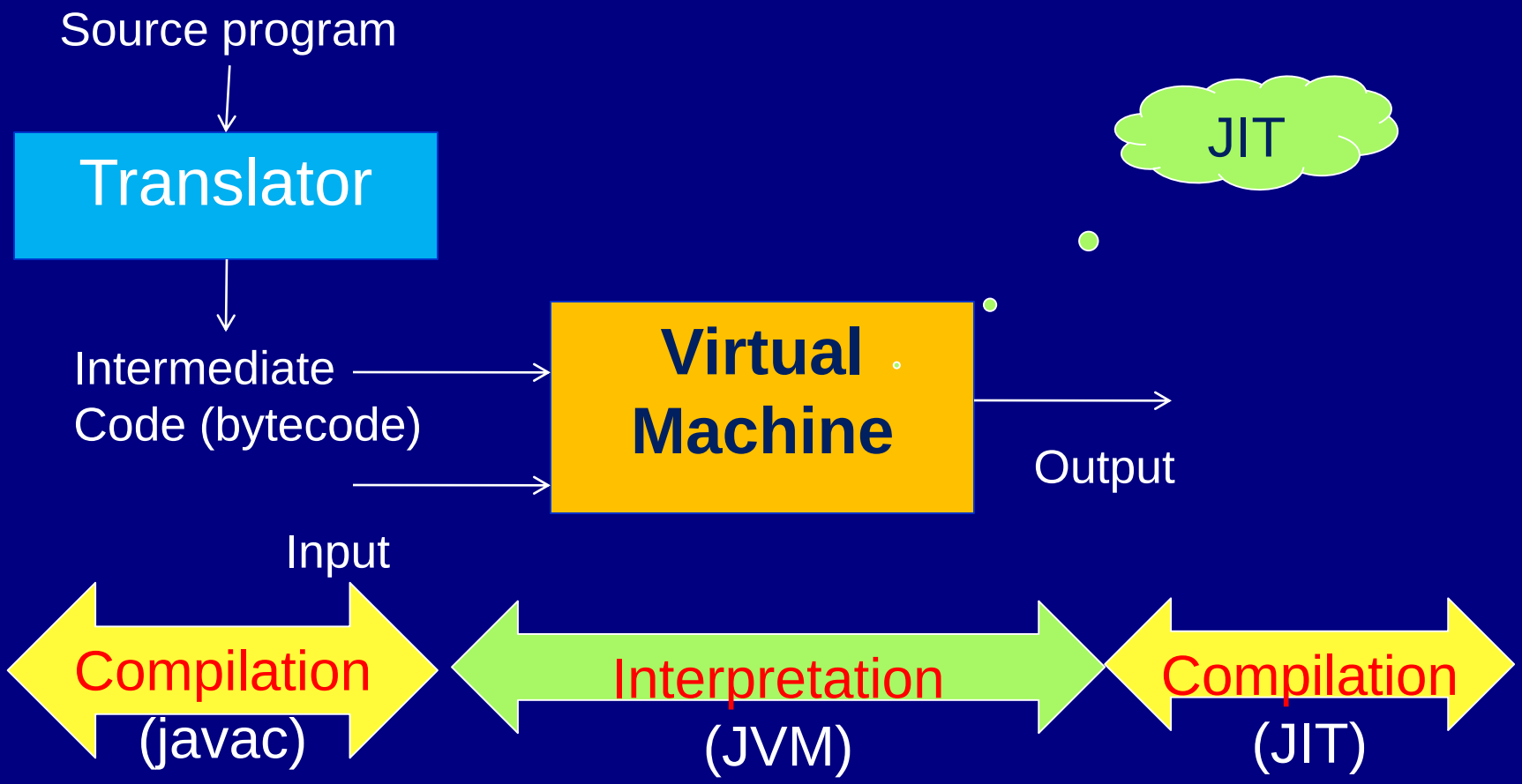


- Interpretation often involves large overhead
- Interpretation avoids the *synthesis* part.

Why Interpretation

- A higher degree of machine independence: high portability.
- Dynamic execution: modification or addition to user programs are allowed (e.g. for interactive debugging).
- Dynamic data types: type of objects may change at runtime (e.g. Javascript)
- Easier to write an interpreter – no synthesis part.
- Better diagnostics: more source text information available

Hybrid Translation



Example: A Java Virtual Machine with a JIT

Fallacy
Compilation is always static.

Runtime compilation:

- increasingly more popular, considering Java JIT, MS .net, Graphics rendering, GPGPU code generation, …
- allowing for more portable applications. (***compile then distribute*** *vs.* ***distribute then compile***)
- enableing more ***machine dependent*** code optimizations.
- enabling adaptive systems (*e.g. dispatching OpenCL kernels to a more suitable device*)

Structure of a Compiler

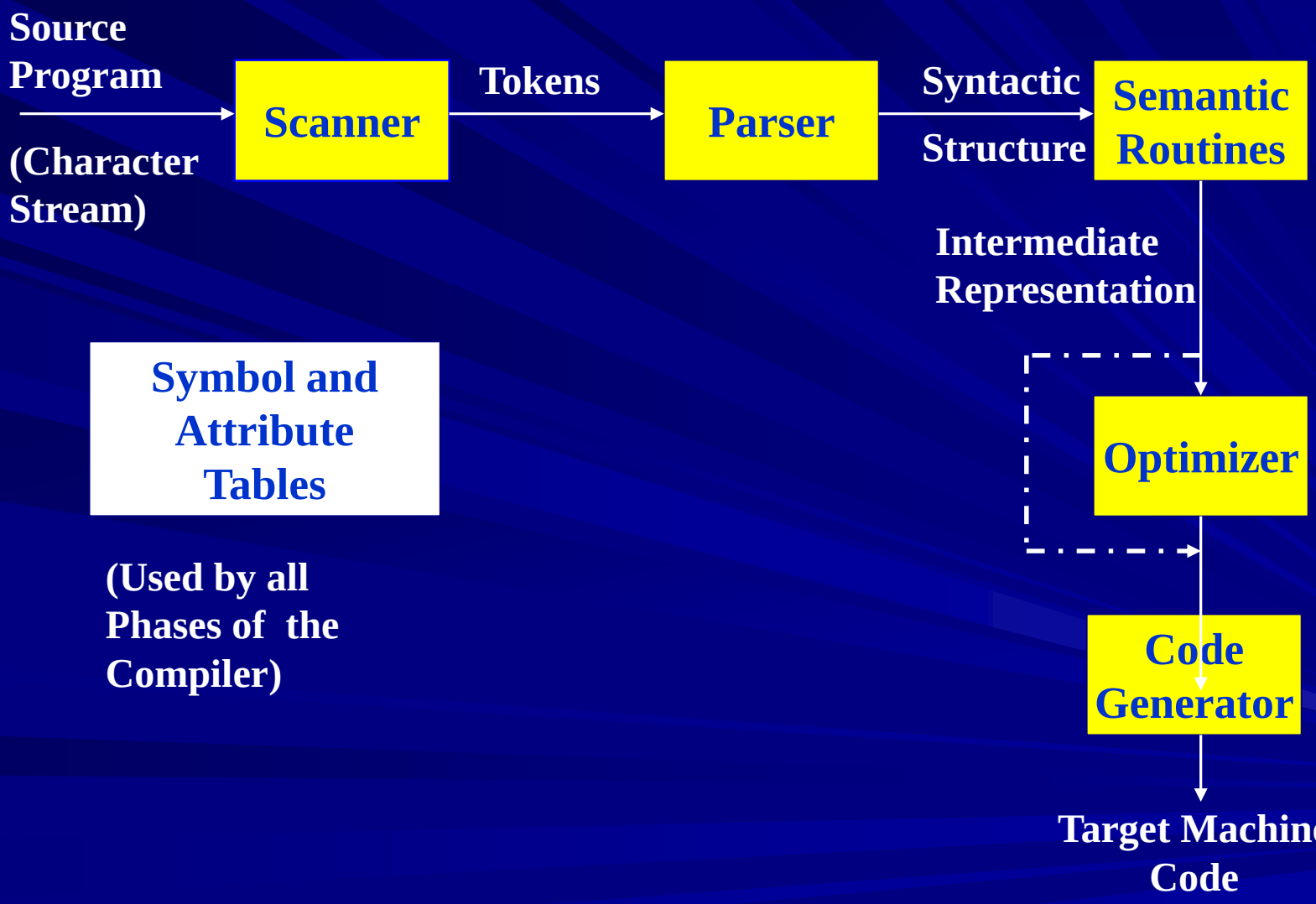
Analysis

- Lexical analysis (Linear Analysis) : stream of characters are grouped into *tokens*
- Syntax analysis (Hierarchical Analysis): tokens are grouped hierarchically with collective meaning
- Semantic Analysis: ensuring the components of a program fit together (type, scope, ..)

Synthesis

- Code Generation and Optimization

Structure of a Syntax-Directed Compiler



Syntax-Directed Translation

- Translation actions attached to each rule of grammar.
- Parsing is the driver of compilation
- Once a construct is recognized by a grammar rule, a sequence of actions are taken to perform semantic checks, code generation.

Lexical Analysis Example

Pay := Base + Rate* 160

- Lexical analysis:

  characters are grouped into seven tokens:

  Pay, Base, Rate are identifiers

  := is assignment symbol

  + and * are operators

  160 is a number

- Error example:

  pay := base + rat@e*160

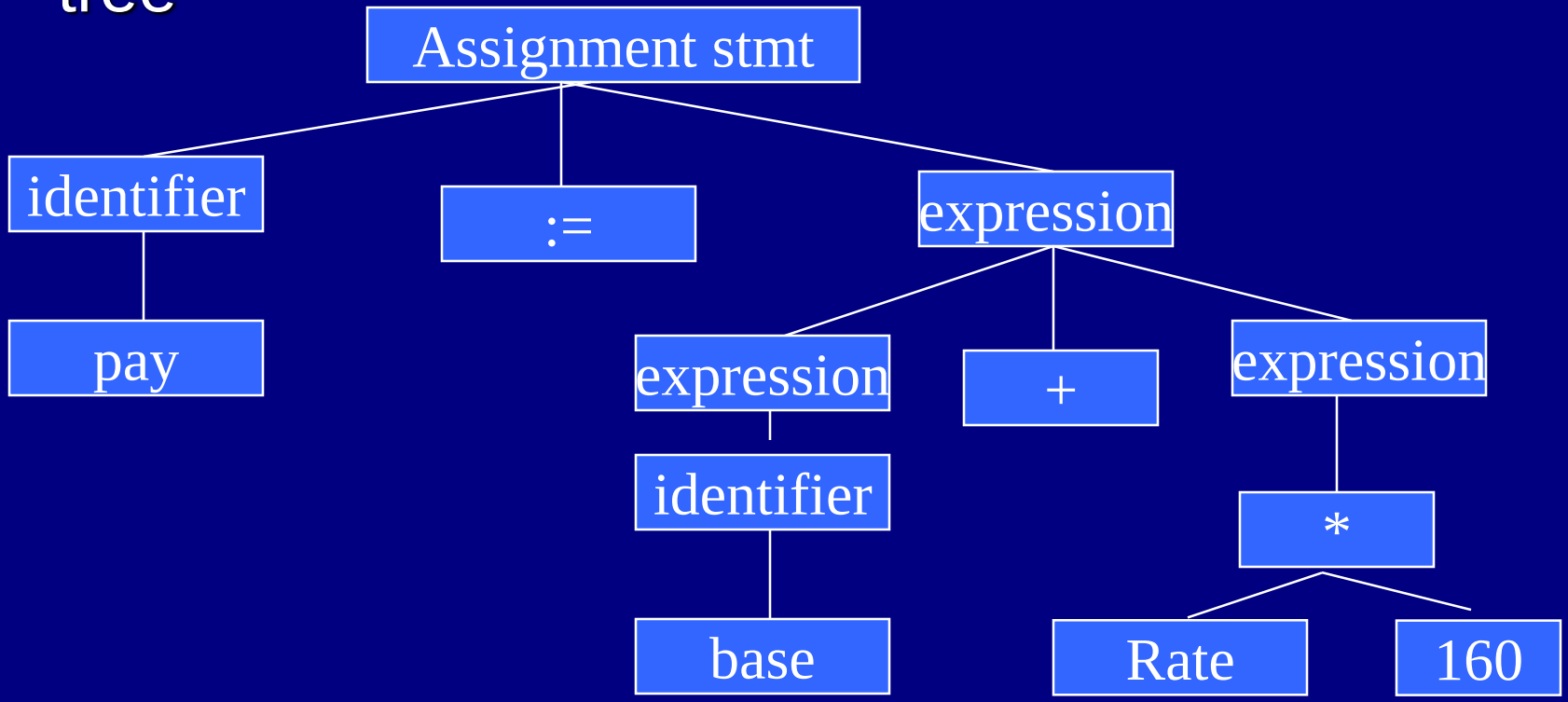
*pay := base + rat^^e*160*

→ this error (^^) will be propagated to syntax analysis and reported as a syntax error

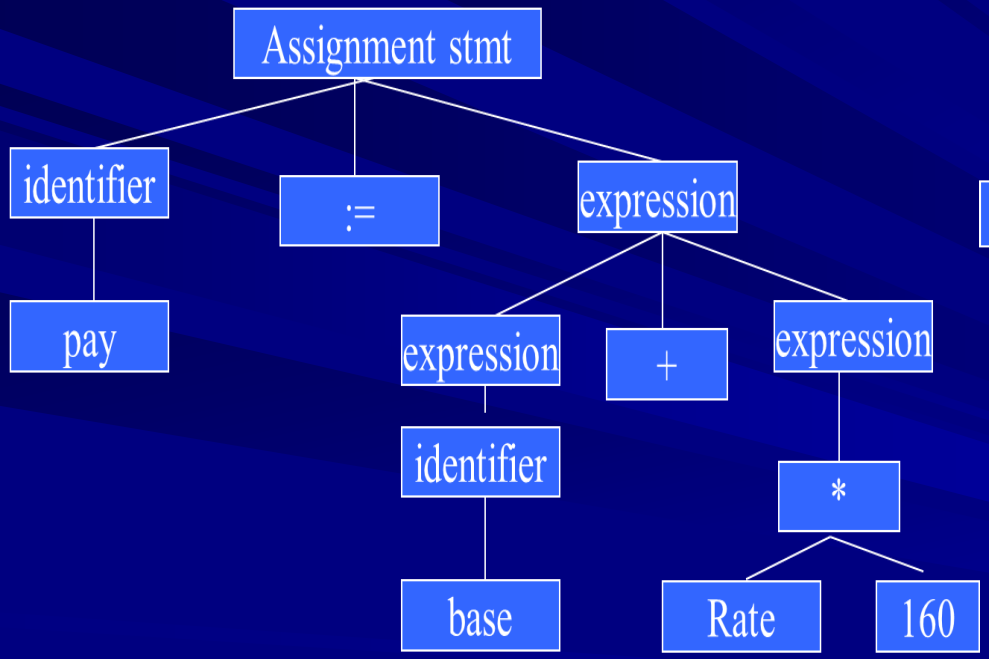
Syntax Analysis Example

Pay := Base + Rate* 160

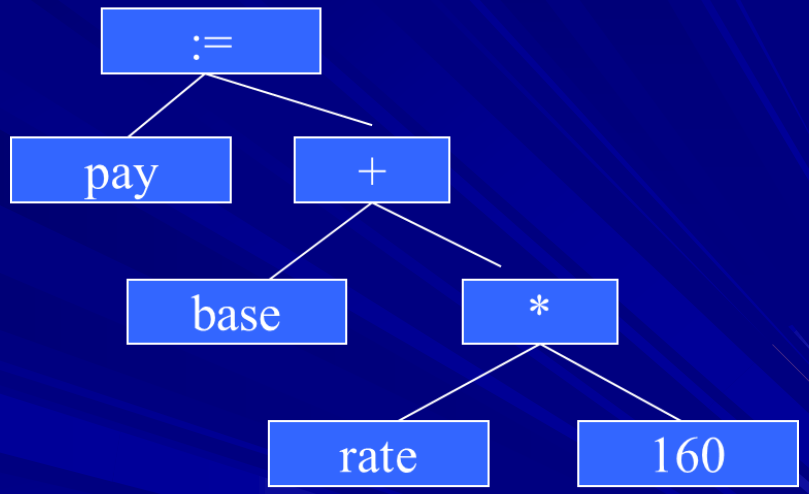
- The seven tokens are grouped into a parse tree



Transforming a Parse Tree to a Syntax Tree



Parse Tree

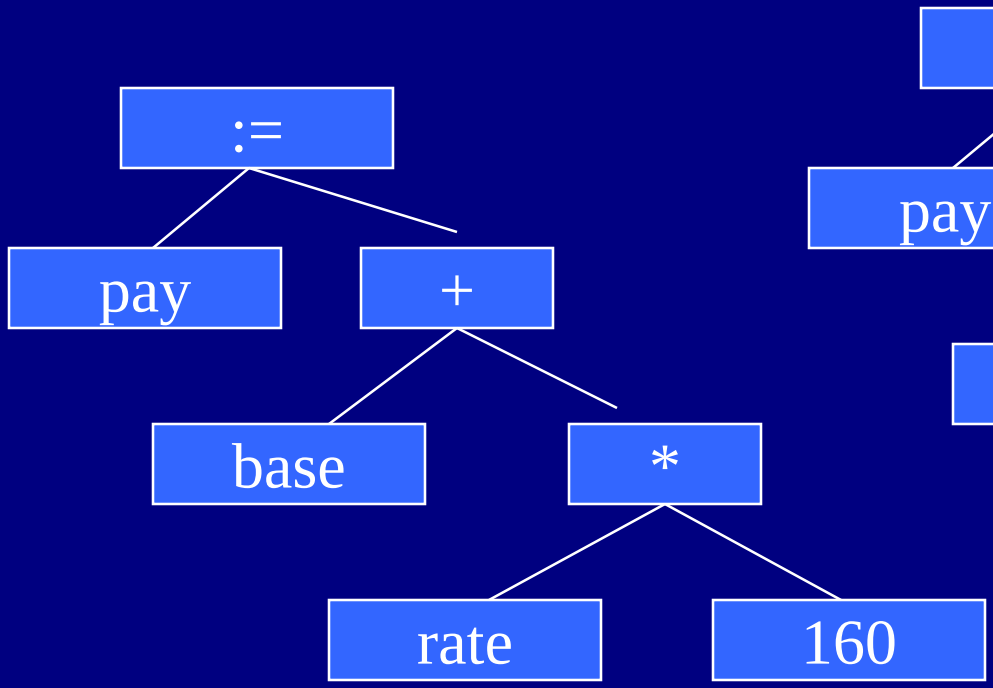


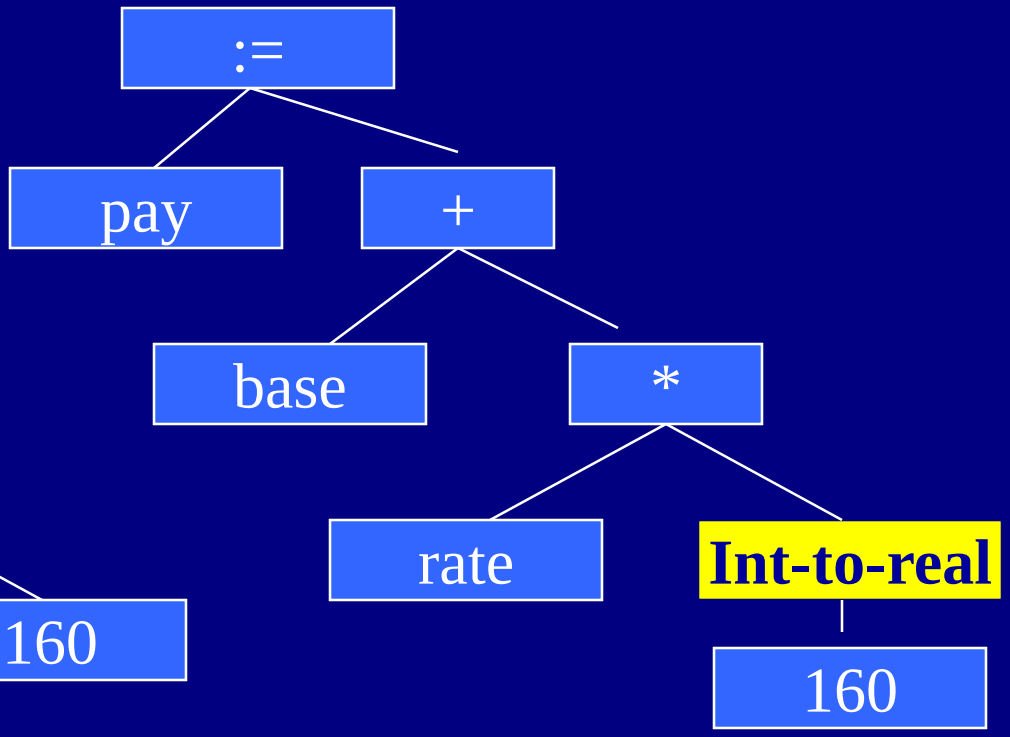
Syntax Tree
(also called
Abstract Syntax Tree
AST)

Semantic Analysis Example

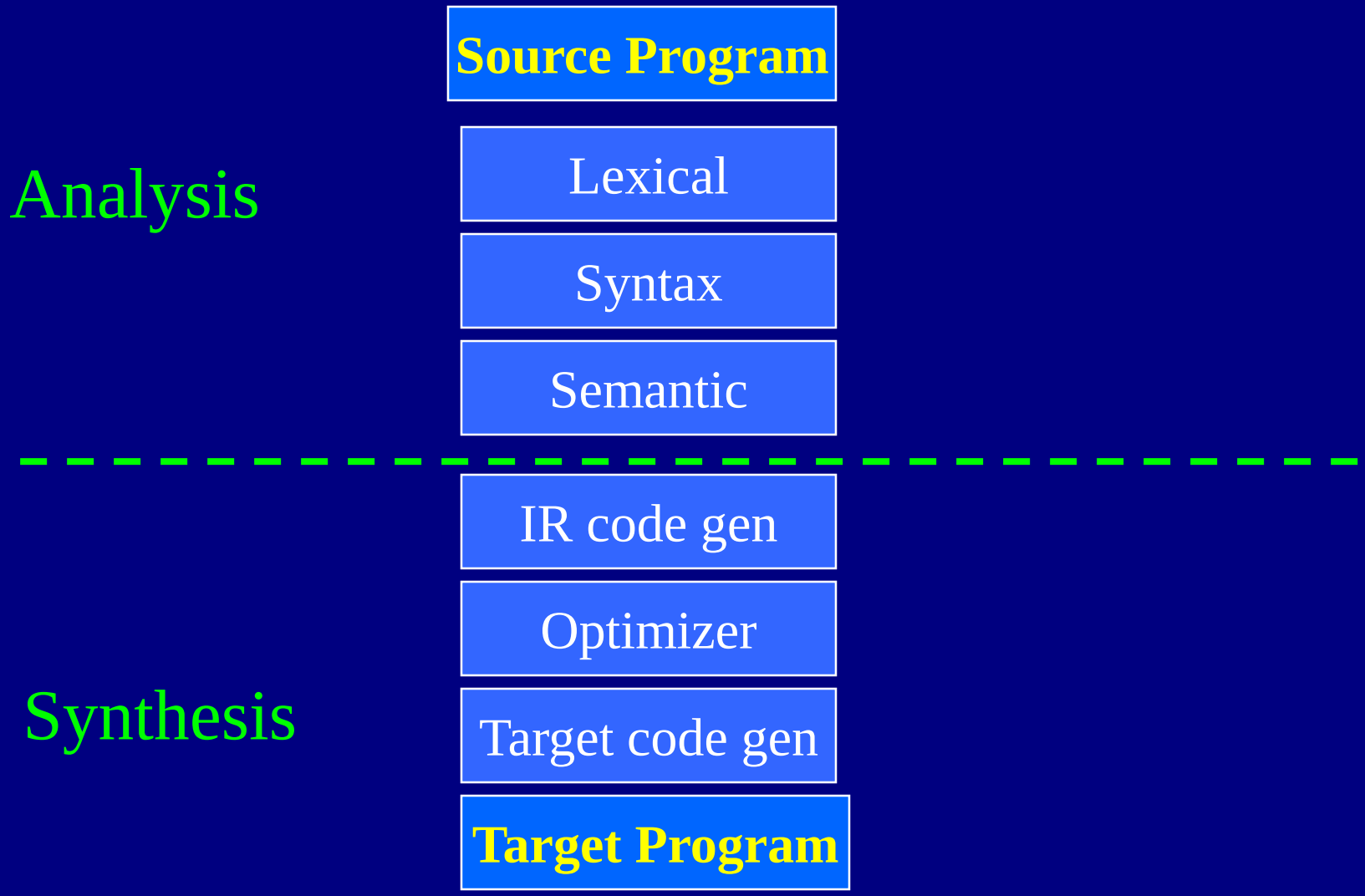
Pay := Base + Rate* 160

- Checks for semantic errors and gathers type information for code generation.

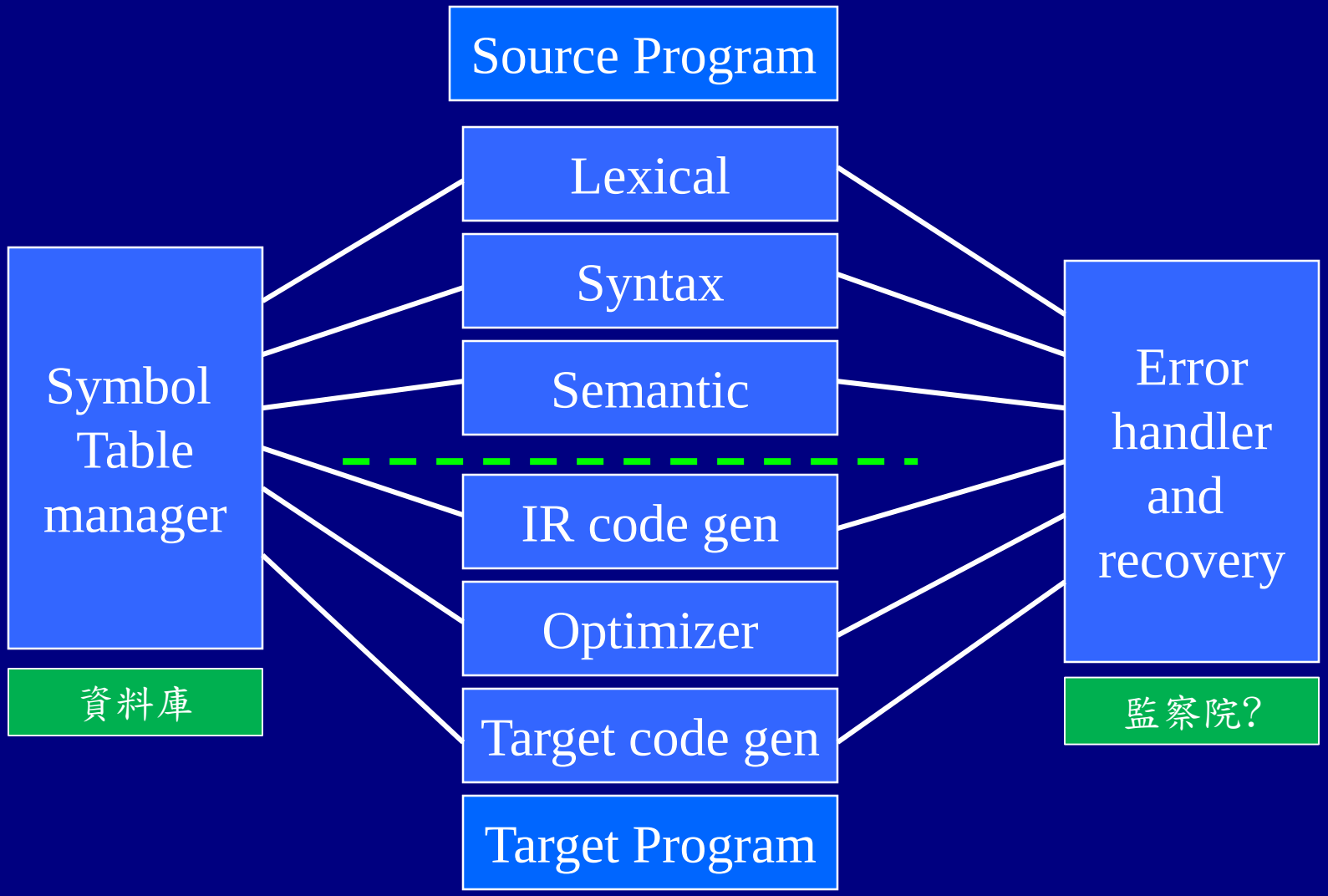




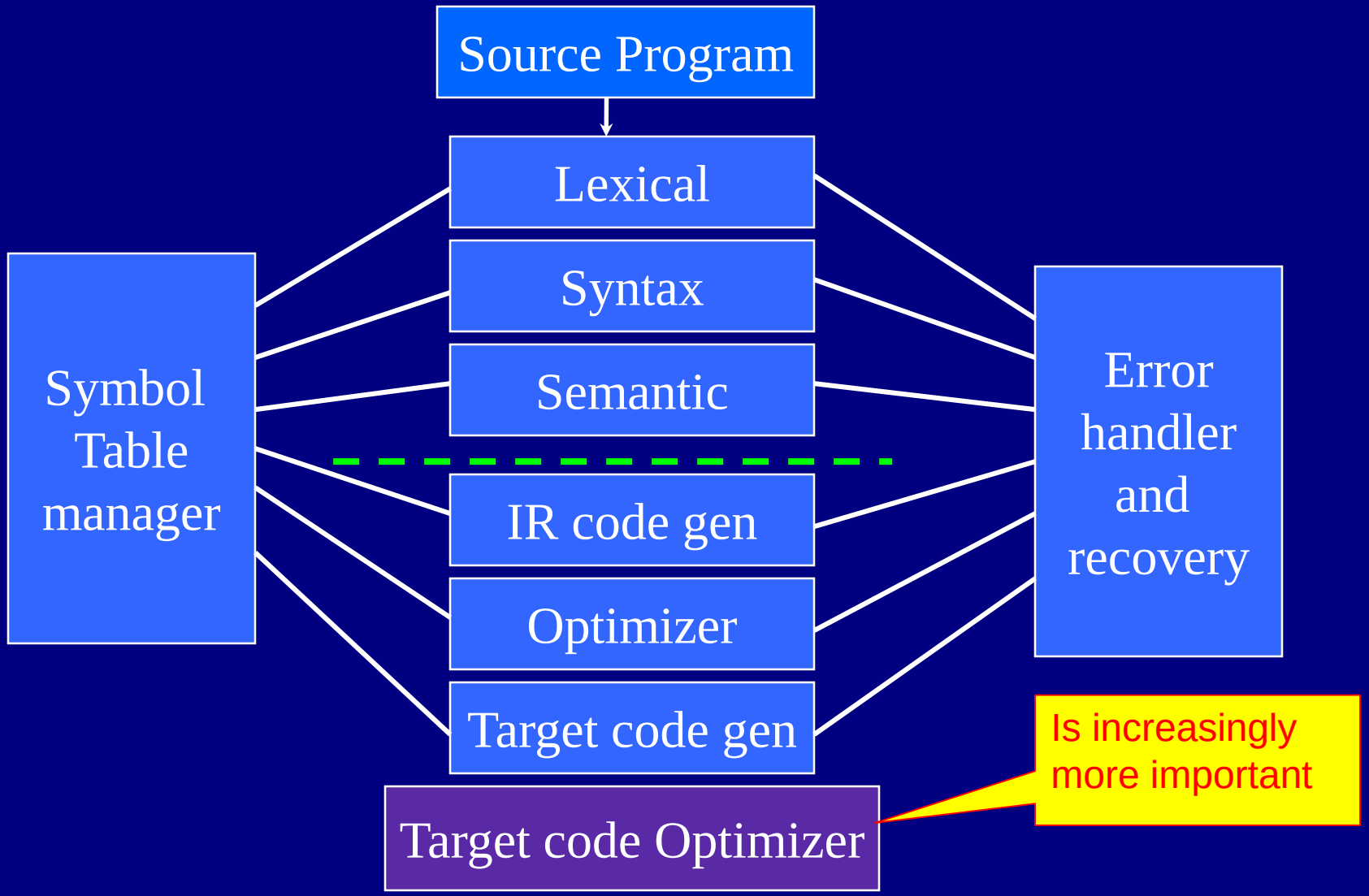
The Phases of a Compiler

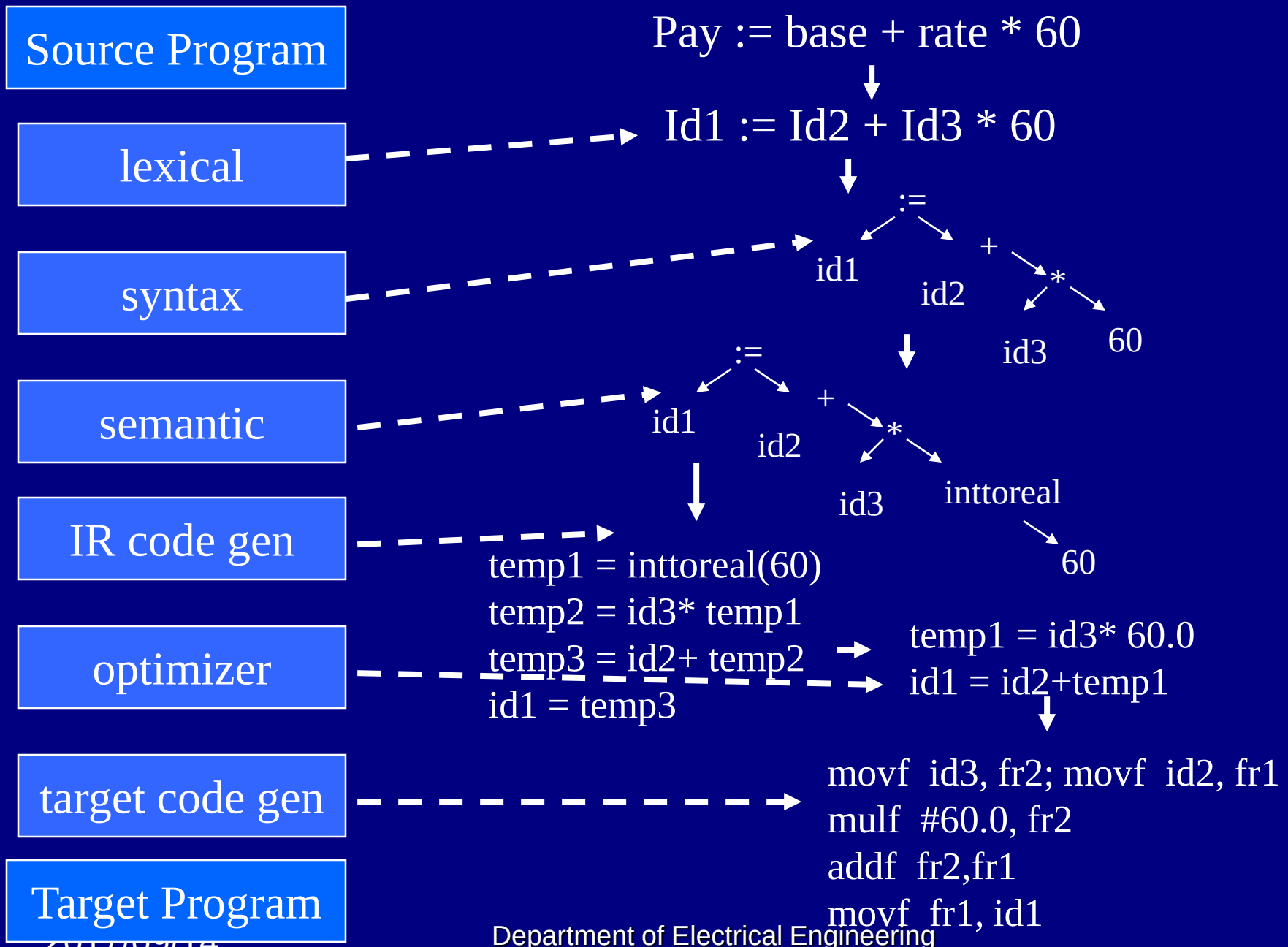


The Phases of a Compiler



The Phases of a Compiler




Source Program
lexical
syntax
semantic
IR code gen
optimizer
target code gen
Target Program
Pay := base + rate * 60
Id1 := Id2 + Id3 * 60
:=
id1
+
id2
*
id3
60
:=
id1
+
id2
*
id3
inttoreal
60
temp1 = inttoreal(60)
temp2 = id3* temp1
temp3 = id2+ temp2
id1 = temp3
temp1 = id3* 60.0
id1 = id2+temp1
movf id3, fr2; movf id2, fr1
mulf #60.0, fr2
addf fr2,fr1
movf fr1, id1

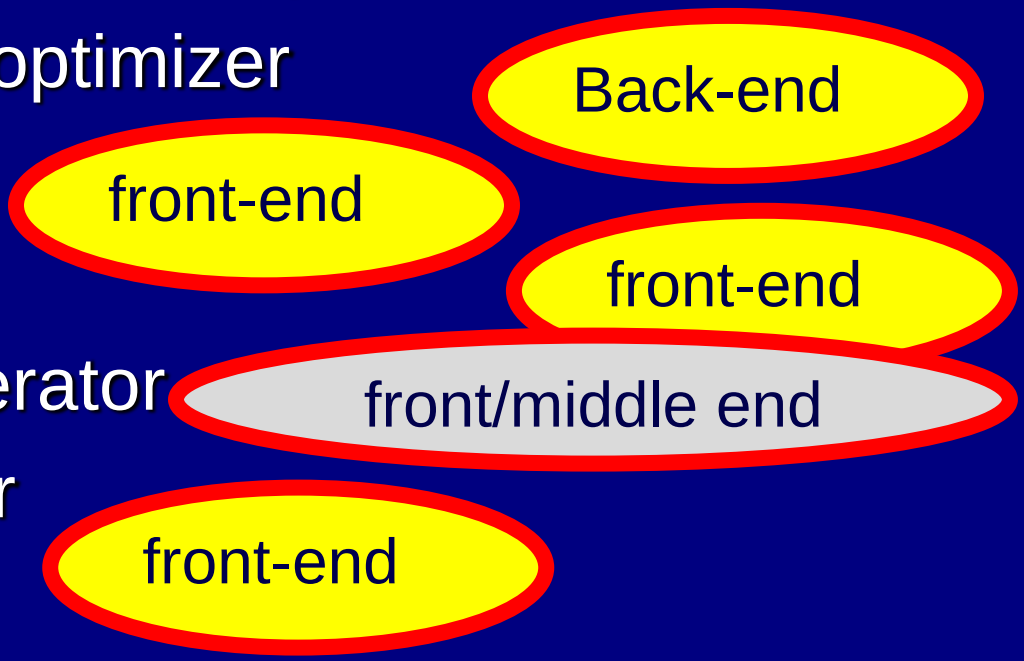
Grouping of Compiler Phases

- **Front end**
  - those phases depending on the source language but largely independent of the target machine.
- **Back end**
  - those phases that are usually target machine dependent such as optimization and code generation.

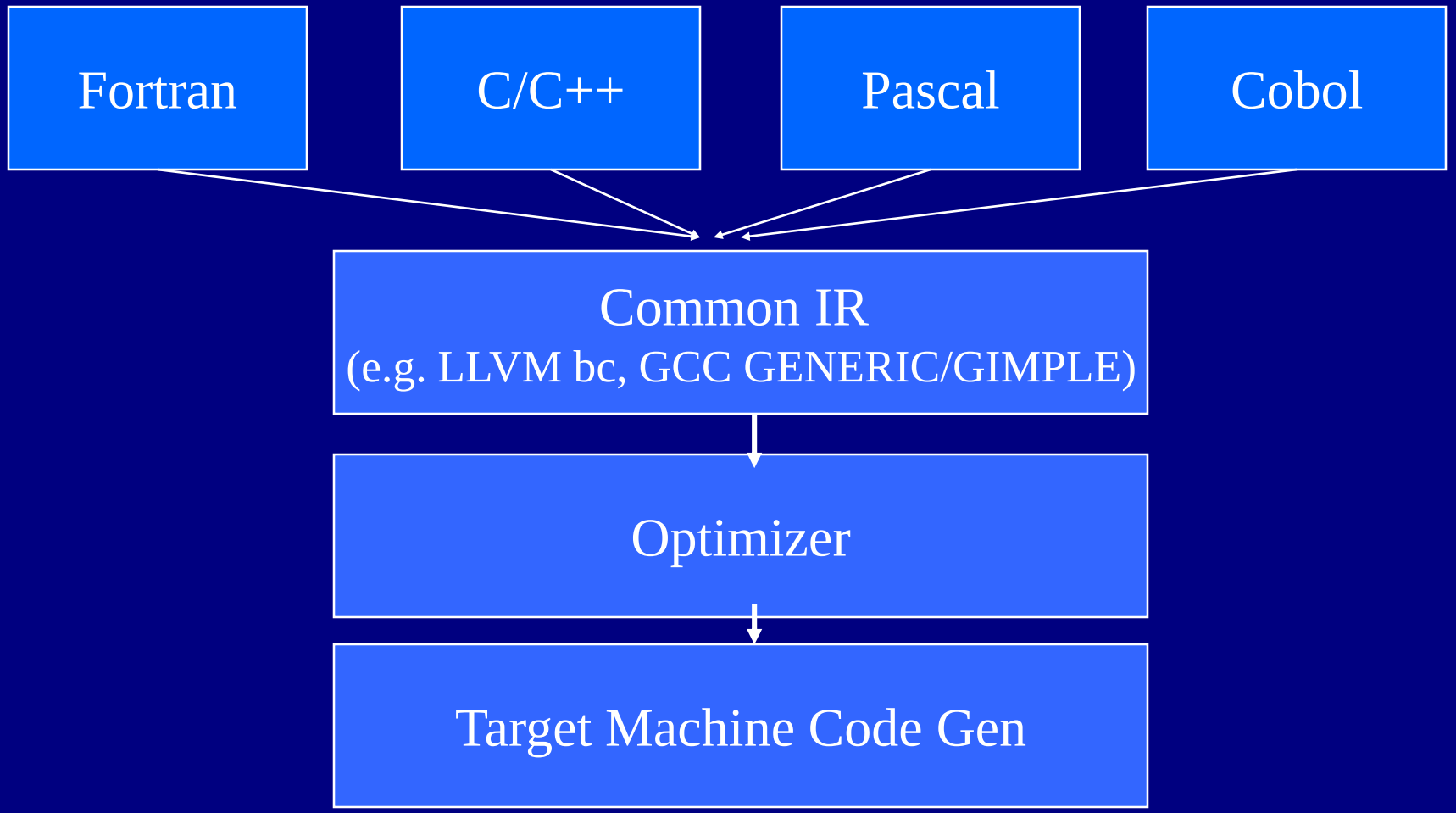
Pop quiz

Which of the following components are part of front-end, which are part of back-end ?

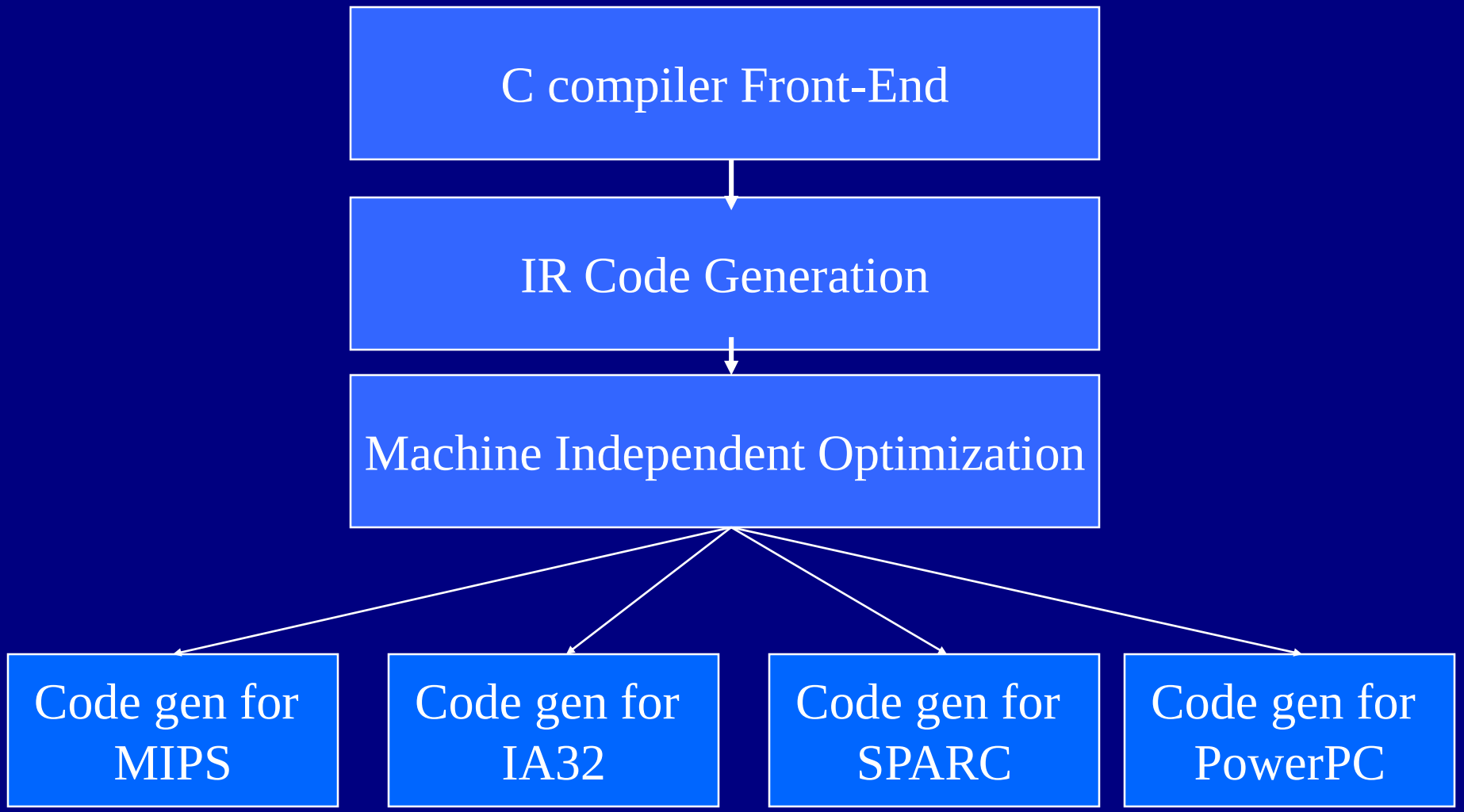
- Target code optimizer
- Parser
- Scanner
- IR code generator
- Type checker



Common Back-end Compiling System



Retargetable Compiler



Pop quiz

The HP PA-RISC compiler/optimizer is

a) a common back-end compiling system

b) a retargetable compiler

a

The ICC compiler (Intel's C Compiler) is

a) a common back-end compiling system

b) a retargetable compiler

a (C and C++)

b (x86 and IA64)

The GCC (GNU C Compiler) is

a) a common back-end compiling system

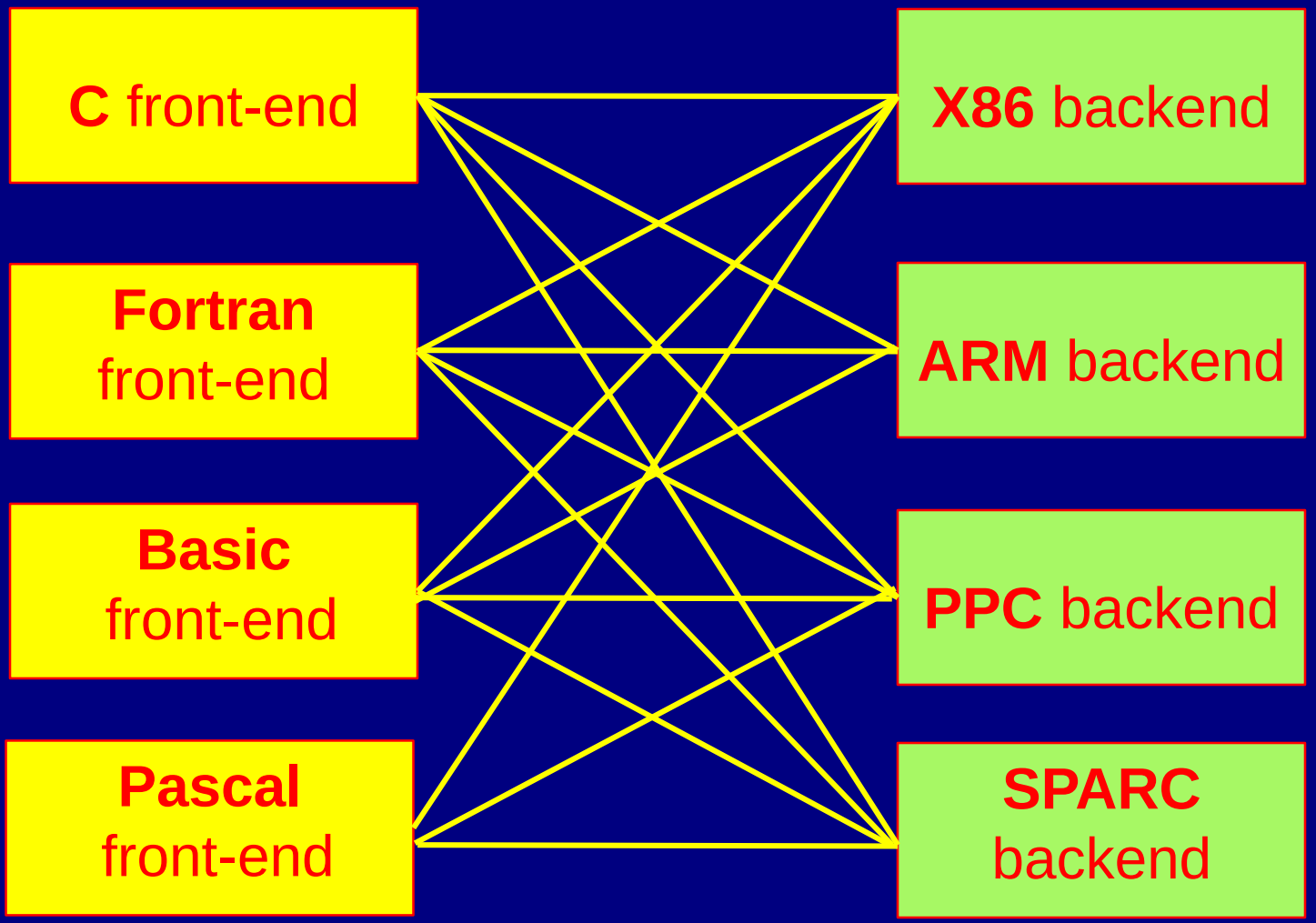
b) a retargetable compiler

b

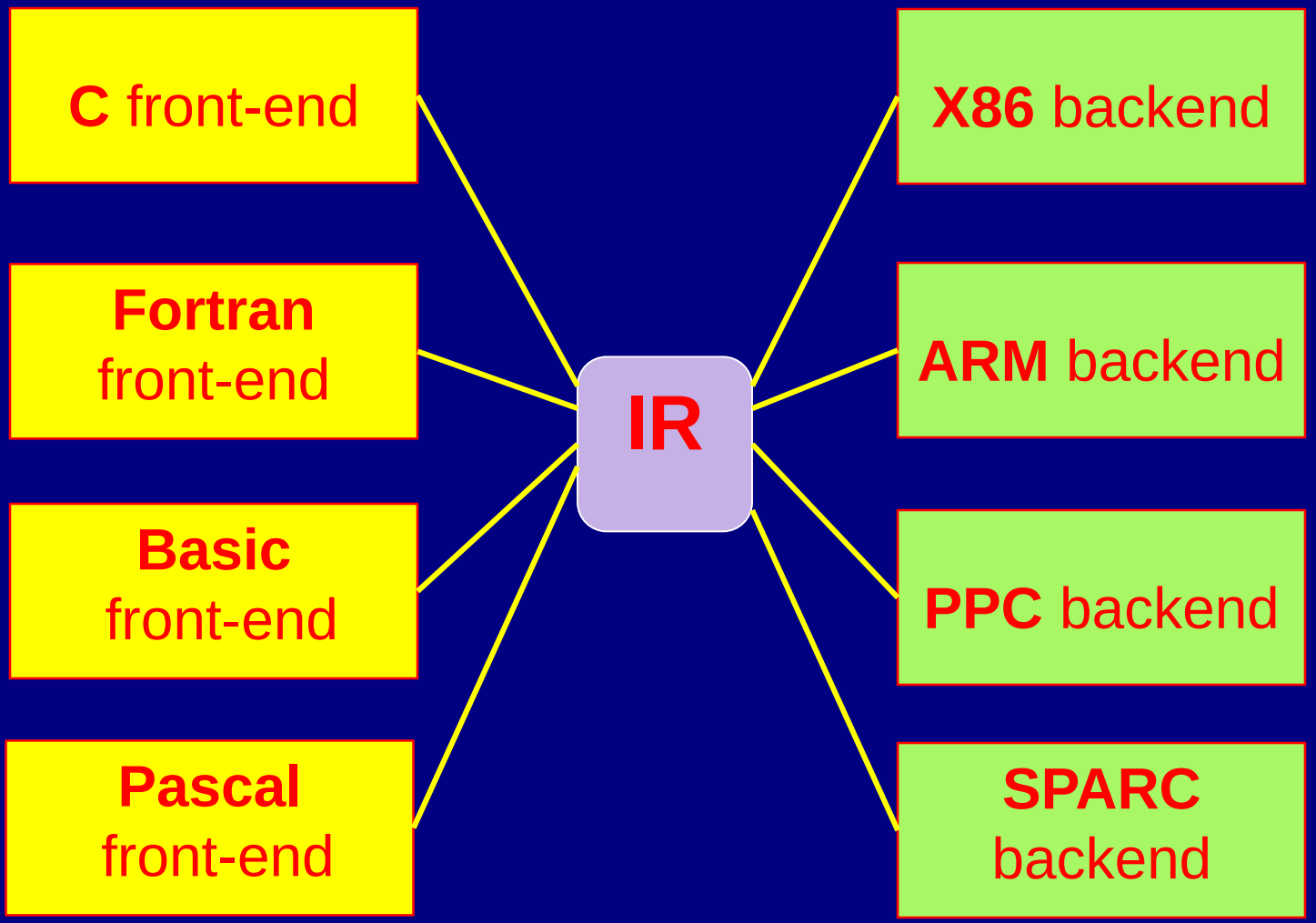
Quick Overview

- What are the two main functions of translation?
  - Analysis and Synthesis
- What are major work involved in analysis?
  - Lexical, Syntax and Semantics analysis
- What are the typical phases in a compilation?
  - lexical,syntax,semantics,IR-CG,optimization,TargetCG
- What phases belong to FE (Front-End)
  - Analysis + IR
- What phases belong to BE (Back-End)
  - Target Code Gen + Machine dependent optimization
- How is a JVM/JIT different from an ordinary compiler
  - JIT compiles ByteCode (VISA) into native code at runtime

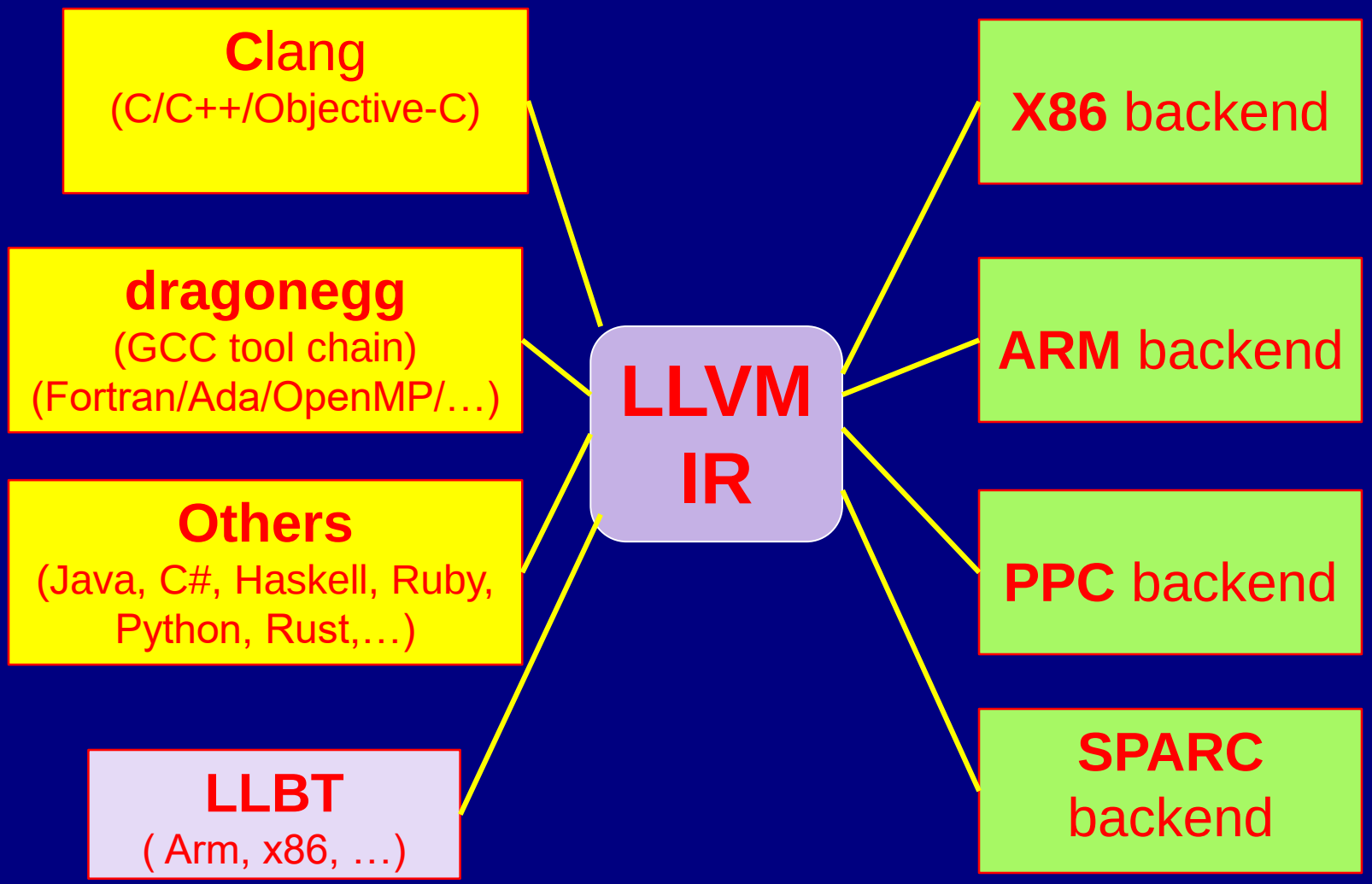
Without IR


C front-end
Fortran front-end
Basic front-end
Pascal front-end
X86 backend
ARM backend
PPC backend
SPARC backend

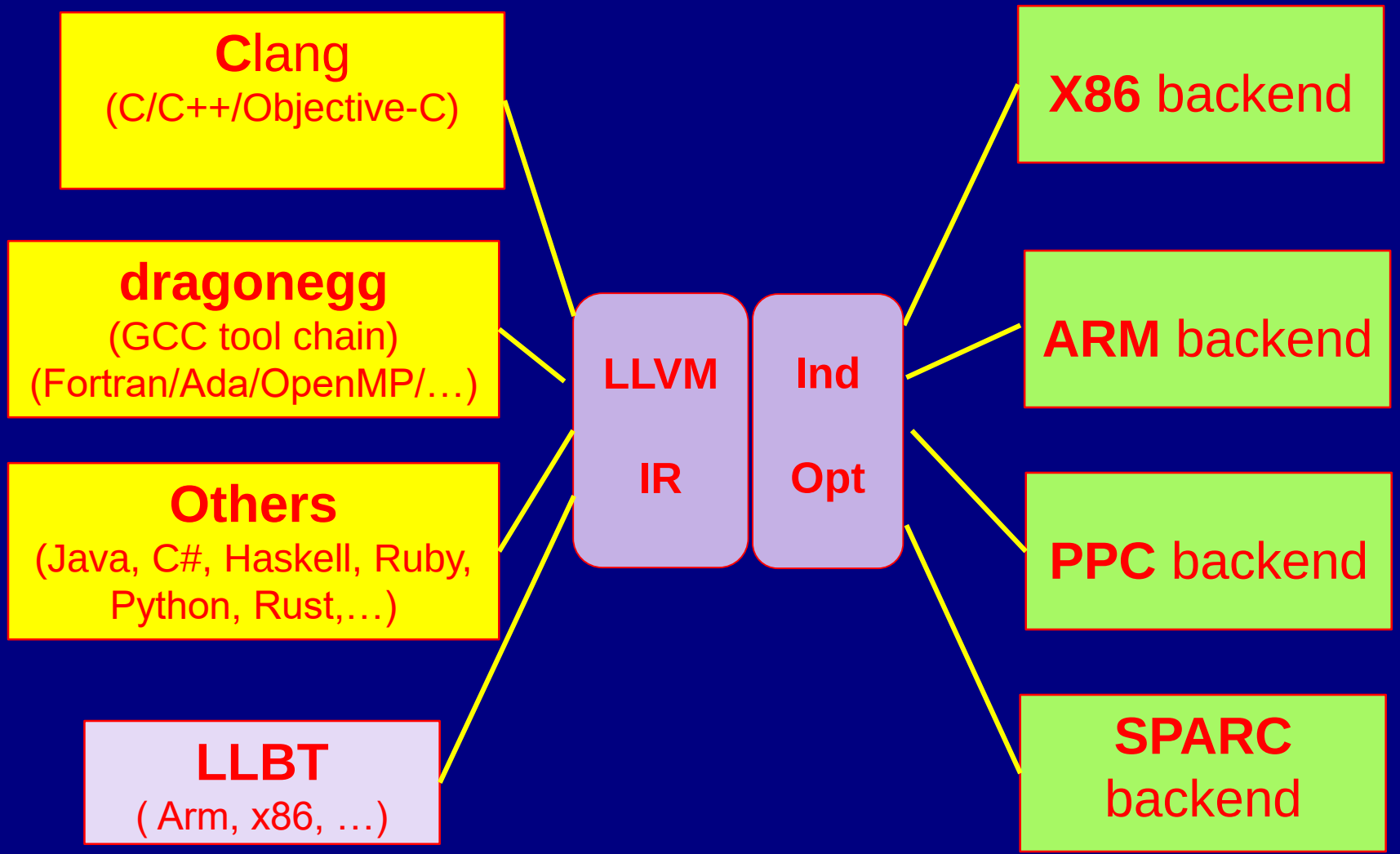
With IR


C front-end
Fortran front-end
Basic front-end
Pascal front-end
IR
X86 backend
ARM backend
PPC backend
SPARC backend

LLVM Compiler Infrastructure



LLVM Compiler Infrastructure



Syntax and Semantics

- A complete definition of a PL includes specification of its ***syntax*** (structure) and ***semantics*** (meaning)
- **Syntax**

  Syntax is usually defined by CFG (*context free* grammar)

  e.g. A := B+C; is legal, A:=B+; is not.

  CFG example:

  <exp> → <exp> + identifier

  B+C, A+D, A+B+C, … are valid expressions
- **Semantics**

  Many important language features such as type compatibility and scoping rules are *context-sensitive*.

  e.g. A:=B+C; is legal when all variables are declared, are not boolean or structure type, and are type compatible

Static Semantics

- Static semantics provide a set of rules which syntactically legal programs are actually valid, such as type rules.

  e.g.
  - all identifiers are declared
  - operator and operands are type compatible
  - procedures are called with the proper parameters

- *Attribute grammars* are a popular method of formally specifying static semantics. e.g.

Exp1→Exp2+Term (Exp2.type == numeric && Term.type==numeric).

Imprecise Semantics Challenges

Example:

In Pascal, a logical expression

(I <> 0) **and** (K div I > 10)

is not well-defined – because it is not defined whether the **and** operator should be treated as ordinary binary operator (both expressions {I<>0} and {K div I} should be evaluated first) or like a short-circuit operator (i.e. expression {K div I} should not be evaluated if I==0).

Why short-circuit is not always a preferred implementation here ?

Imprecise Semantics Challenges

Another Example:

In Java, all functions must return via a ***return expr*** statement.

The following function is illegal:

```
public static int subr(int b) {
    if (b!=0) return b+100; }
```

How about the following?

```
public static int subr(int b) {
 if (b!=0) return b+100;
 else if (10*b == 0) return 1; }
```

Why Study Compilers?

- Influences on programming language design (see section 1.6)
- Influences on computer design (see section 1.7)
- Compiling techniques are useful for software development
  - Parsing techniques are often used
  - Learn practical data structures and algorithms
  - Basis for many tools such as text formatters, structure editors, silicon compilers, design verification tools,…
- So you may write more efficient code
  - Writing a compiler requires an understanding of almost many important CS subfields

Why Study Compilers?

- Influences on programming language design (see section 1.6)
- Influences on co
- Compiling techn development
  - Parsing techniq
  - Learn practical
  - Basis for many editors, silicon c
- So you may writ
  - Writing a compi many important CS subfields

Requirements for a successful language

- Easy to compile
- Has a quality compiler on a wide variety of machines
- Better code will be generated
- Fewer compiler bugs
- Compiler is smaller, faster, reliable
- Good diagnostic messages and development tools

Why Study Compilers?

- Influences on programming language design (see section 1.6)
- Influences on co
- Compiling techn
  development
  - Parsing techniq
  - Learn practical data structures and algorithms
  - Basis for many tools such as text formatters, structure editors, silicon compilers, design verification tools,…
- So you may write more efficient code
  - Writing a compiler requires an understanding of almost many important CS subfields

Many machines failed because they are hard to compile/optimize for:

Intel 860
IBM cell
Cray-2

Why Study Compilers?

- Influences on programming language design (see section 1.6)
- Influences on computer design (see section 1.7)
- Compiling techniques are useful for software development
  - Parsing techniques are often used
  - Learn practical data structures and algorithms
  - Basis for many tools such as text formatters, structure editors, silicon compilers, design verification tools,…
- So you may write more efficient code
  - Writing a compiler requires an understanding of almost ma

programming, data structures and algorithms, automata and formal language, system programming, computer organization.

Compiler and AI

- Compiler is AI, it has high intelligence:
  - It can correct your syntax errors, ensure correct semantics
  - It is an expert in generating highly efficient machine code
- Compiler is the key to “SDE” – Software Defined Everything. Retargeting a software for different devices enables SDE.
- Machine Learning (ML and DL) need domain specific compilers (such as XLA from Google) to increase performance
- Compilers need AI/ML to be more intelligent
  - Deep learning to discover most suitable optimizations
  - Efficient and effective retargeting to various devices
  - Resource allocation and management for heterogeneous systems
  - Automate the compiler generation

A Short History of Compiler Construction

- 1945—1960 Code Generation

  How to generate code for a given machine

  The goal was to match the efficiency of assembly coding.

- 1960—1975 Parsing

  Many new languages came out. Automatic parsing became more important.

- 1975—present Code Optimization

  ILP: RISC and Post-RISC machines.

  Vectorization: SIMD(SSE, GPGPU) Parallelization: Multiprocessors (SMP, MPP)

  Heterogeneous Computing: Cuda, OpenCL, HSA

A Short History of Compiler Construction

- 1945—1960 Code Generation

  How to generate code for a given machine

  The goal was to match the efficiency of assembly coding.

- 1960—1975 Parsing

  Many new languages came out. Automatic parsing became more important.

- 1975—present Code Optimization

  ILP: RISC and Post-RISC machines.

  Vectorization: SIMD(SSE, GPGPU) Parallelization: Multiprocessors (SMP, MPP)

  Heterogeneous Computing: Cuda, OpenCL,HSA

- Present to 2050 AI and Compilers

Compiler Constructions Tools

- First Fortran compiler took 18 person-years. Now with compiler construction tools, you may build a simple compiler in one semester.
- Translator writing tools:
  - Scanner generator
  - Parser generator
  - Symbol table manager
  - Attribute grammar evaluator
  - Automatic code generator
  - Data flow analyzer generator

Compiler - Compiler

Quiz

- In 2001, Intel introduced the new 64-bit architecture IA-64 or IPF (called Itanium Processor Family), and a few generations of processors, including Merced(01), McKinley (02), Madison (03), Montecito (06), and Tukwila (2010). IA-64 is not binary compatible to x86 (i.e. IA-32) machines.

Q: How to create the first C compiler on the Itanium machine?

Requirements:

1) the compiler must generates Itanium code

2) the compiler must run on Itanium machines

Quiz

- Q: How to create the first C compiler on the Itanium machine?

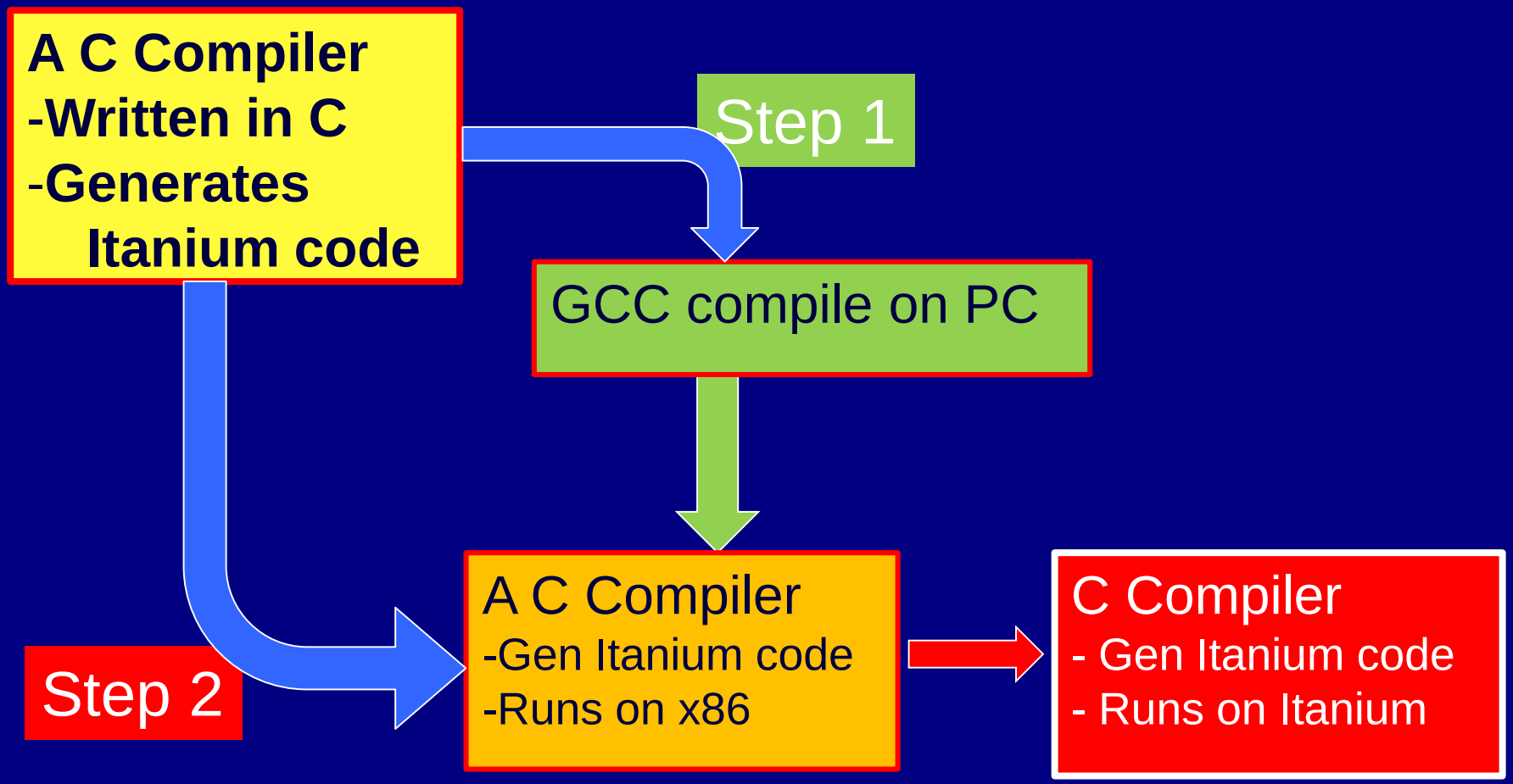
  a) Write a C compiler in C, compiled it on Itanium machines using GCC.

  b) Write a C compiler in Itanium machine code

  c) Develop a Cross-compiler on x86 PC and use it to compile *itself* into C/Itanium.

  d) Leave it to Intel/MicroSoft (i.e. evil empire)

Cross-Compilation



Cross-Compilation and Bootstrapping

- Quiz: How to create the first C compiler on the new Itanium if no cross-compilers to use?

Cross-Compilation and Bootstrapping

- Answer: *Bootstrapping*.

1. *A subset of C is selected (e.g. C--) and a simple compiler is written in assembly code, called this compiler C0.*
2. *Rewrite this subset compiler using the subset (C--), compile it with C0, get a new compiler called C1.*
3. *Write a more complete set of C in C--, compiled with C1, get a new compiler C2*
4. *Repeat the process until a complete C compiler is done*

Summary

- What is a compiler
- Why is compiling technique important
- Three phases of analysis
- Phases of a compiler
- Grouping of phases
- Compiler construction tools

Runtime Semantics

- Runtime (or execution time) semantics specifies what a program computes.
  - For example, a program state is defined, and program execution is described as in terms of changes to that state.
- Natural semantics
  - Give assertions known to be true before a construct, we may infer assertions that will hold after the evaluation of the construct.
- Axiomatic semantics
  - A set of formally specified relations or predicates that relate program variables.
  - Statements are defined by how they modify these relations
- Denotational semantics

Above methods try to enable runtime checks to be generated automatically from language semantic specification