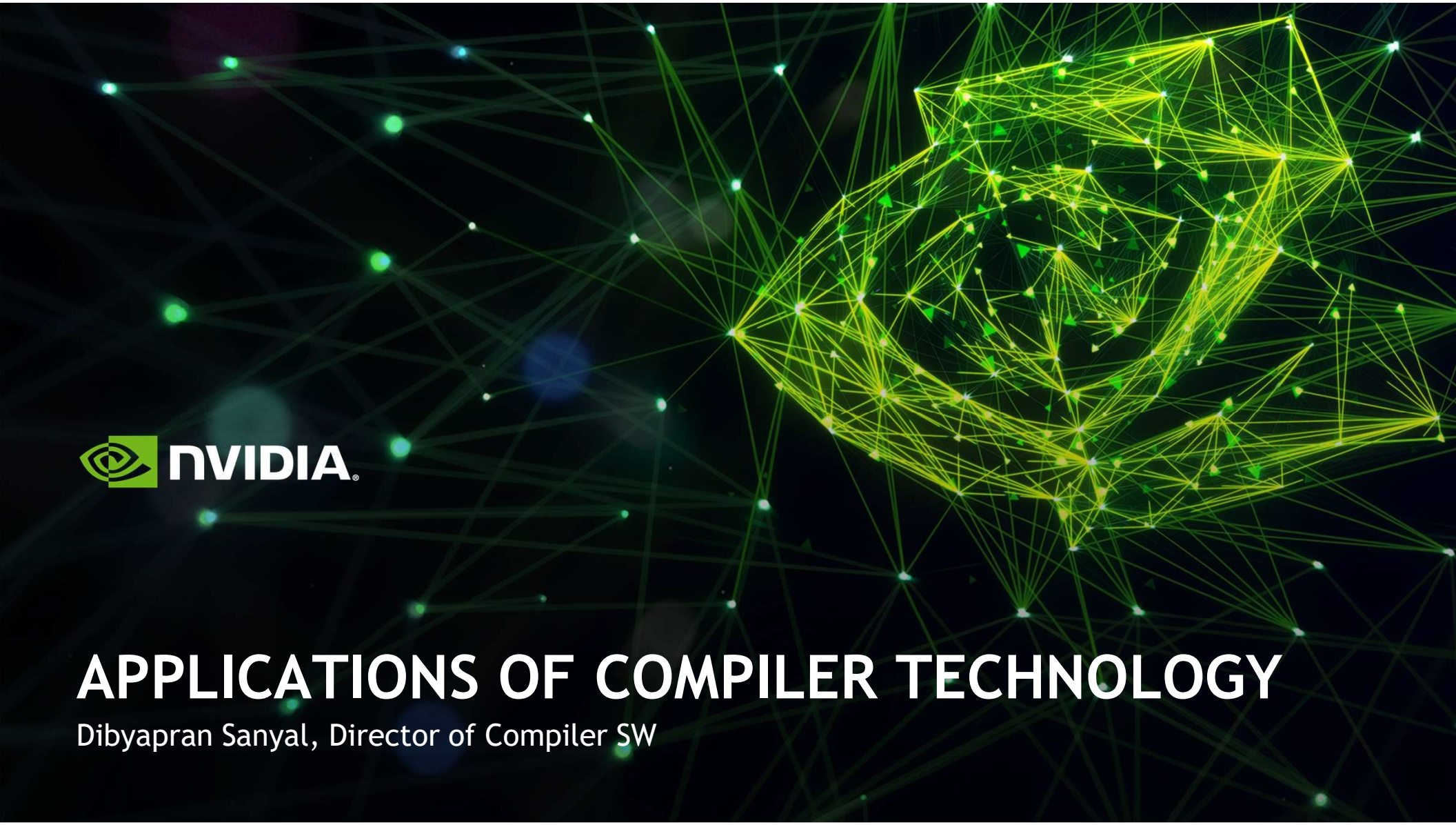


APPLICATIONS OF COMPILER TECHNOLOGY

Dibyapran Sanyal, Director of Compiler SW

AGENDA

Traditional Compilers

Compilers - Key capabilities

Static (Source) Analysis

Dynamic (Binary) Analysis

Binary Translators

Runtime Compilation

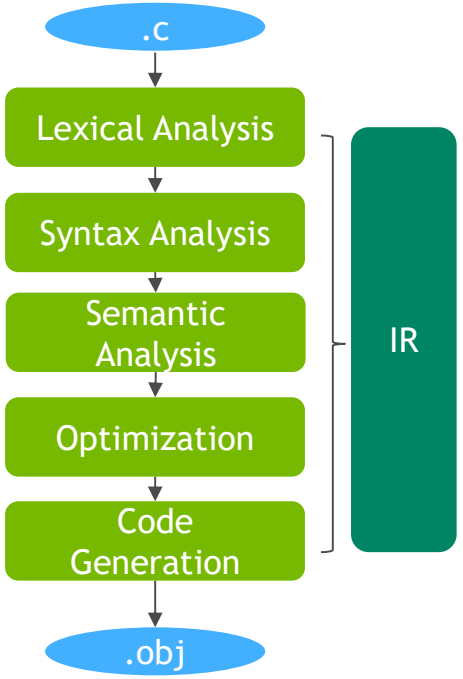
Source to Source Translators

Domain Specific Languages

TRADITIONAL COMPILER

Overview

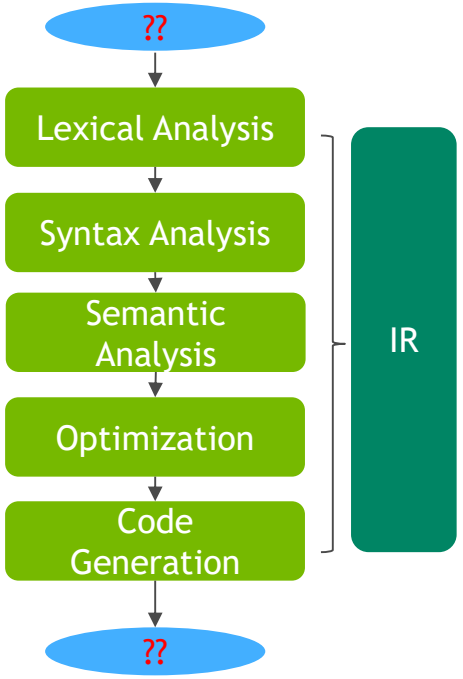
- Generate Executable Binary from High Level Language Input
- Examples: Fortran, C, C++, …
- Well known compilers : GCC, LLVM, MSVC …
- Optimizations are key area of focus
- Several Performance Metrics: Generated Code, Compile Time, Code Size, …
- Very large user base
- Very large & complex SW
- Relatively small community of compiler developers



TRADITIONAL COMPILER

Landscape

- Diverse Inputs
  - High-Level Languages: 100s of them!
  - Many Language Types/Paradigms: Imperative, Object Oriented, Functional, Parallel …
- Diverse Outputs
  - Many Processor Types: CPU, GPU, DSP, ASIP …
  - Many Architectural Styles: CISC, RISC, VLIW, SIMD, SIMT …
- What drives this diversity?
  - Application Needs!!



COMPILER PHASES

What do they do?

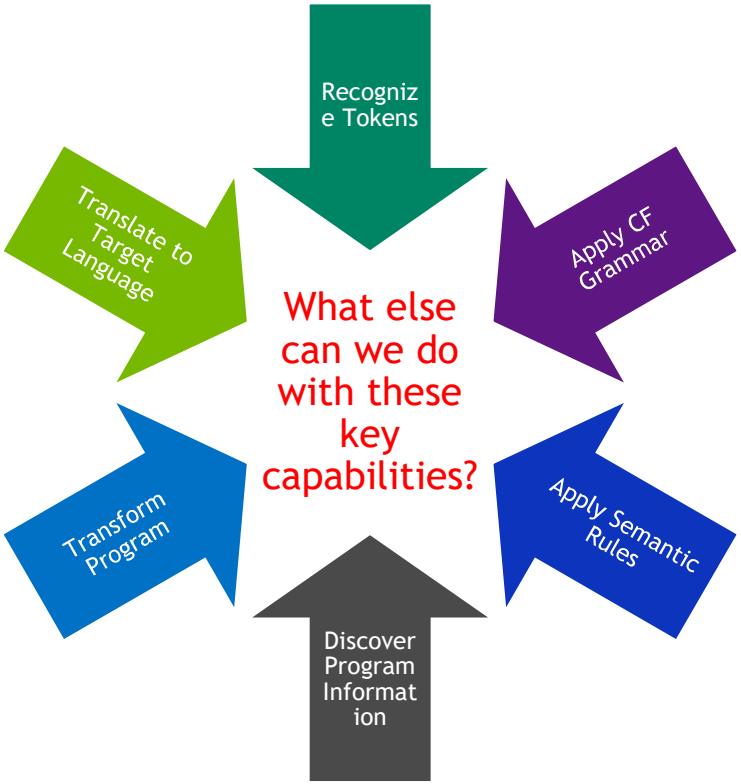
Lexical Analysis	*match lexemes by RegExp*	Token Stream	Errors e.g. illegal Identifier
Syntax Analysis	*apply CF Grammar Rules to ensure syntax tree is correct*	AST / Parse Tree	Errors e.g. illegal operator, …
Semantic Analysis	*apply Semantic Rules to evaluate syntax tree*	Legal AST / IR	Errors e.g. Type Mismatch Undeclared Variable, …
Optimization	*Program Analysis : Discover Information about program*	Program Information	Liveness, Available Expressions …
	Transform the Program into an alternate (functionally equivalent) program	Optimized Program	CSE, DCE, …
Code Generation	*Map the program to target language*	Object Code	Instruction Selection Register Allocation Instruction Scheduling

COMPILER COMPONENTS

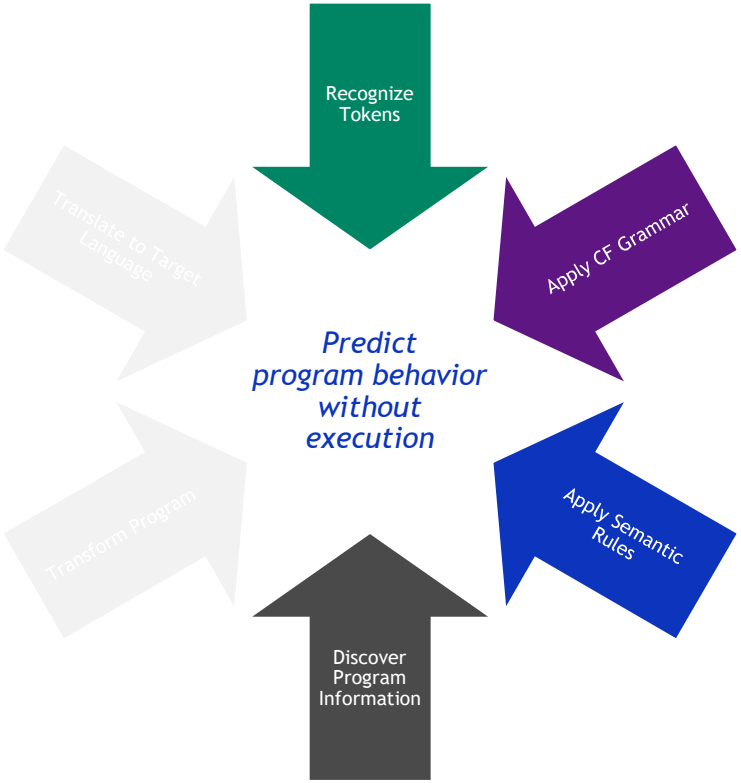
Key Capabilities

- Lexical Analysis – *Ability to recognize tokens from input*
- Syntax Analysis – *Ability to analyze conformance to language grammar*
- Semantic Analysis – *Ability to reason about meaningfulness of input program as per rules of language*
- Intermediate Representation – *Ability to represent the input program*
- Program Analysis – *Ability to discover properties about the input program*
- Optimization – *Ability to transform input program to an equivalent program*
- Code Generation – *Ability to translate input program to an alternate language*

MOTIVATING QUESTION



STATIC (SOURCE) ANALYSIS



STATIC ANALYSIS

What is it?

- Static analysis tools generate information about execution of a program without executing it
  - Not driven by test cases
- Interesting applications include
  - Detection of defects
  - Security problems
  - Performance issues
  - Enforcement of coding standards
- Relies on Lexical, Syntax, Semantic Analysis, IR construction and Program Analysis
- Input program can be Source Code or Object Code

STATIC ANALYSIS

Examples

- Buffer Overflows
- Uninitialized Variable Access
- Resource Leak
- Improper Conversions
- Dead Code
- Coding Errors
- Coding Conventions

```
buf1 =(char*)malloc(…);
strcpy(buf1, buf2);
```

```
int a, b;
...
b = ...;
return a+b;
```

```
void foo(void) {
   int * a = malloc (…);
   …
   // use of a
   …
   return;
}
```

```
int a = 0.1;
char b = a;
```

```
b = 10;
…
c = 2 * b;
…
a = b + c;
if (a < 5)
{
    foo();
}
```

```
if (x == y);
{
    return x;
}
```

```
int isNum;
int is_num;
```

STATIC ANALYSIS

How does it work?

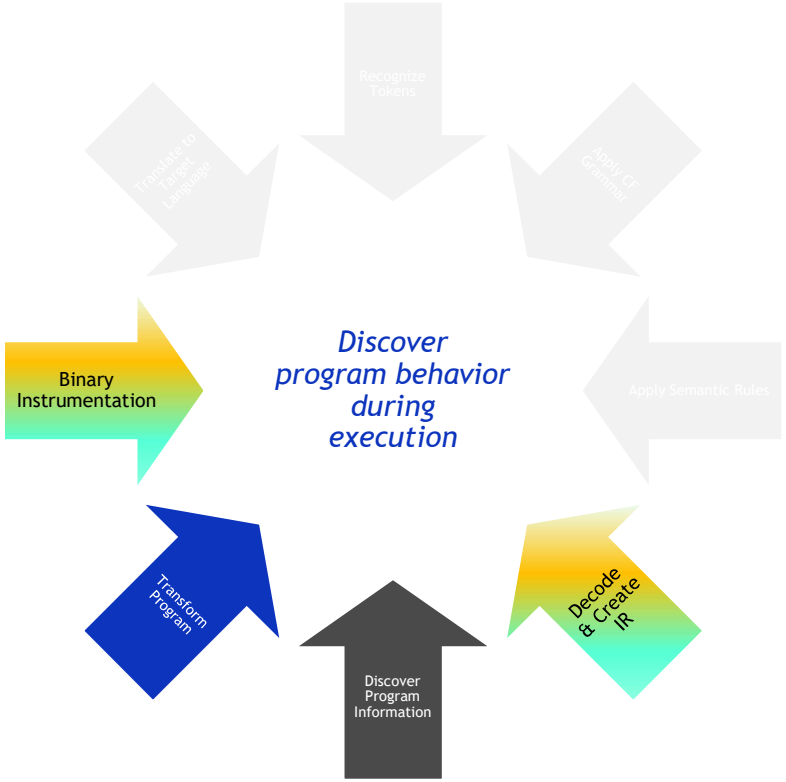
- Lexical & Syntax Analysis remain same
- Additional Semantic rule checking - Type Checks
- Program Analysis
  - Control Flow Analysis
  - Data Flow Analysis
- API aware checks
- False Positives
- Model Checking / Formal Verification

```
if (cond1)
   x = 5;
...
if (cond2) //cond2 is true when cond1 is true
{
   y = x;
}
```

STATIC ANALYSIS TOOLS

- Usually, language specific
- Fixed vs Extensible (custom checkers) Infrastructure
- Examples
  - Clang-tidy (LLVM suite)
  - Lint
  - Eclipse
  - Many other commercial & open source tools …
- Advanced Formal Verification Tools

DYNAMIC (BINARY) ANALYSIS



DYNAMIC ANALYSIS

What is it?

- Dynamic Analysis tools generate information about program execution
  - Relies on either sampling or instrumentation
  - Driven by test cases
- Interesting applications include
  - Detection of defects
  - Memory Usage
  - Profilers & Performance Tuning
- Input program is often Binary Object but can also be source code

DYNAMIC ANALYSIS

Examples

- Memory Leaks
- Memory Corruption
- Uninitialized Variable Access
- Peak & Cumulative Memory Use
- Performance Bottlenecks
  - By Instrumentation
  - By Sampling
- Race Condition Detection

```
int *a;
for (…) {
   a = malloc(...);
}
free(a);
```

```
char *str = malloc(10);
gets(str);
printf("%s\n",str);
```

```
if (cond1)
   x = 5;
...
if (cond2)
{
   y = x;
}
```

DYNAMIC ANALYSIS

How does Instrumentation work?

- Can be performed on source code or binary objects
- Binary or Source is modified to contain event probes that generate information during execution
- Analysis tool provides a runtime library/component to log and track information from event probes
- Flow for binary instrumentation
    1. Decode Binary
    2. Construction Intermediate Representation
    3. Perform Program Analysis (CFA, DFA)
    4. Generate Instrumented Binary

DYNAMIC ANALYSIS

How does Sampling work?

- Applies to binary objects
- HW collects information about execution via Performance Monitor Unit (PMU)
- Hot Blocks, Cache Miss, Branch Prediction …
- May involve Binary Patching
- Often used by Profilers

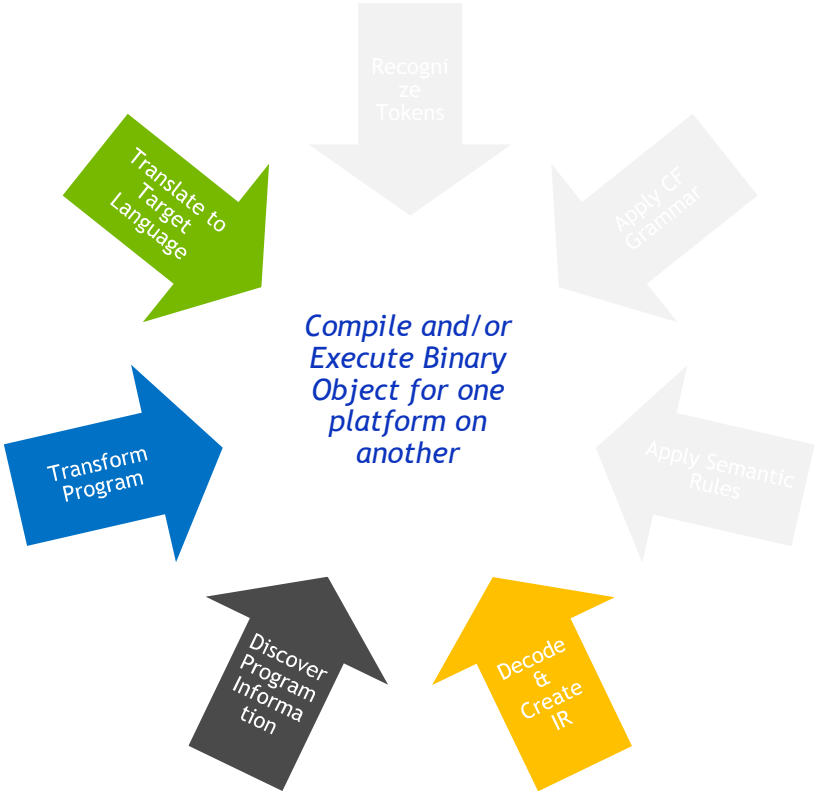
DYNAMIC ANALYSIS TOOLS

- Usually, platform specific
- Fixed vs Extensible Infrastructure
- Examples
  - Valgrind
  - Massif
  - Gprof
  - Many other commercial & open source tools …

RECAP OF ANALYSIS

Analysis	Source Code	Binary Object	Properties
Static Analysis	Discover program information by analyzing source input program.	Discover program information from binary object. E.g. security analysis of binary objects	• No execution • No test cases • May have false positives • 100% coverage
Dynamic Analysis	Discover program information during execution by instrumenting source code. E.g. Profile Guided Optimization	Discover program information during execution by instrumenting binary object.	• Execution based • Test Driven • No false positives • Coverage varies based on test assets

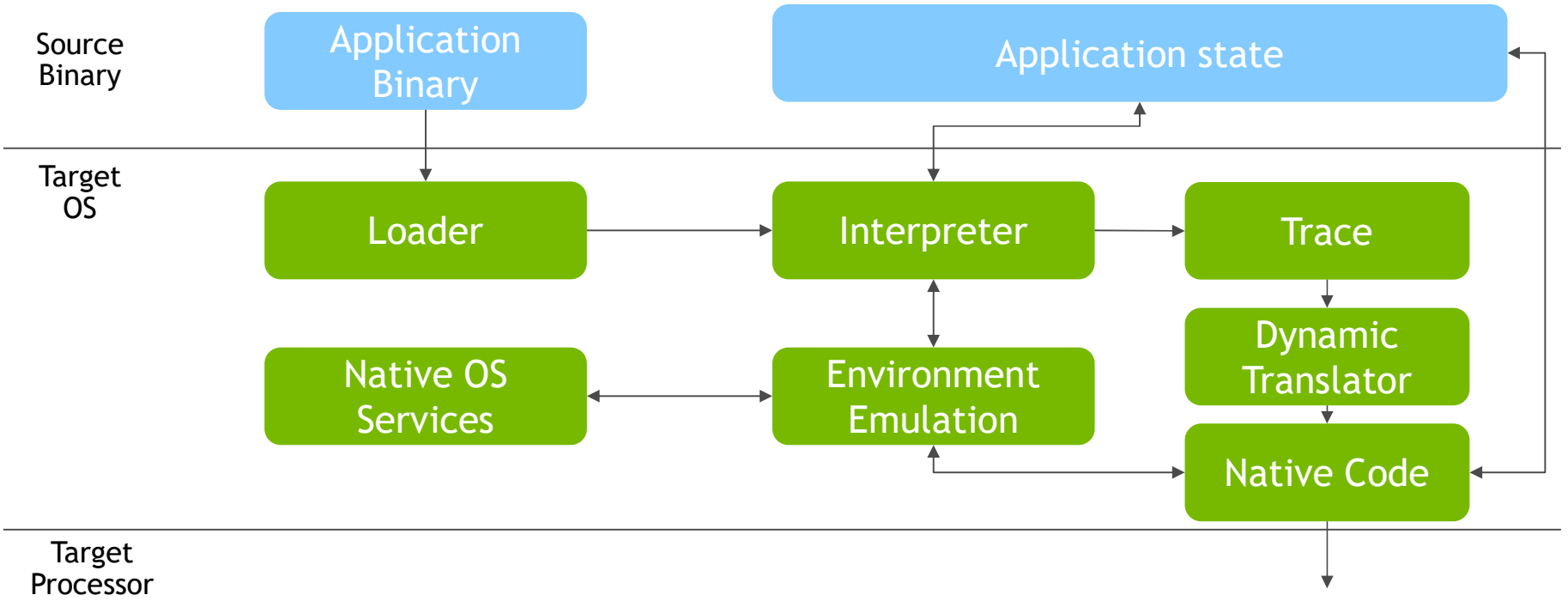
BINARY TRANSLATORS



BINARY TRANSLATORS

- Compile Executable Binary from Source Platform to Target Platform
  - Target Platform may have different Processor and/or OS
- Static Binary Translators : generate target binary without executing it
- Dynamic Binary Translators : generate target binary while executing it
- Dynamic Binary Translators depend on
  - Interpreter : to start executing source binary and identify hot blocks
  - Compiler : to translate hot blocks to native code for target platform
  - Execution Environment Emulation
- Binary optimizers are a special case where Target and Source Platforms are same

DYNAMIC BINARY TRANSLATORS

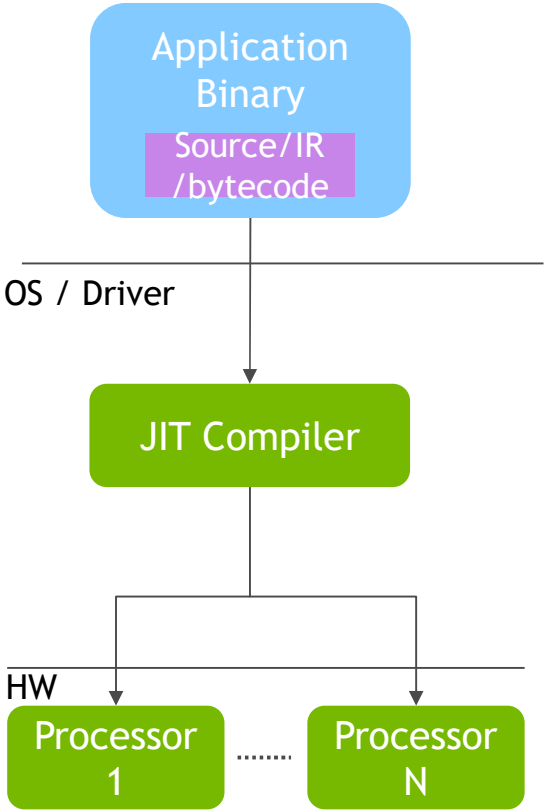


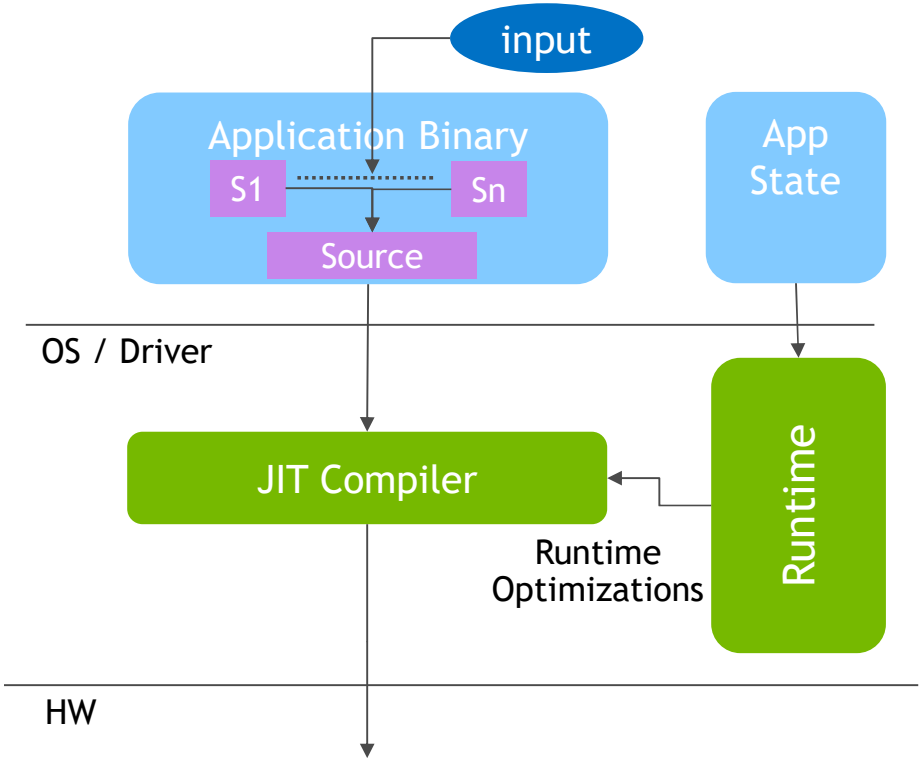
RUNTIME COMPILER

RUNTIME COMPILER

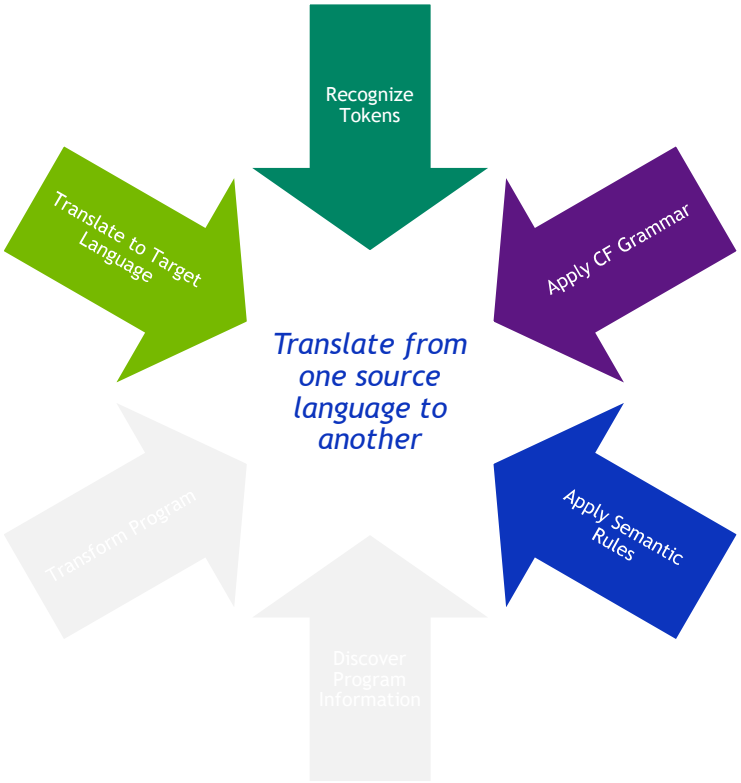
- Runtime aka JIT Compilers compile input source programs to target binary at runtime of the application
- Provide platform independence – Application can ship source or IR
  - E.g. Graphics Shaders – GLSL, HLSL
- Input driven code generation is possible for systems deploying JIT compilation
- JIT Compilers have more information from application execution state
- JIT Compilers often need sophisticated algorithms to tradeoff compile time vs generated code quality
- Virtual Machines often deploy JIT compilers for performance reasons

RUNTIME COMPILER





SOURCE TO SOURCE TRANSLATORS



SOURCE TO SOURCE TRANSLATORS

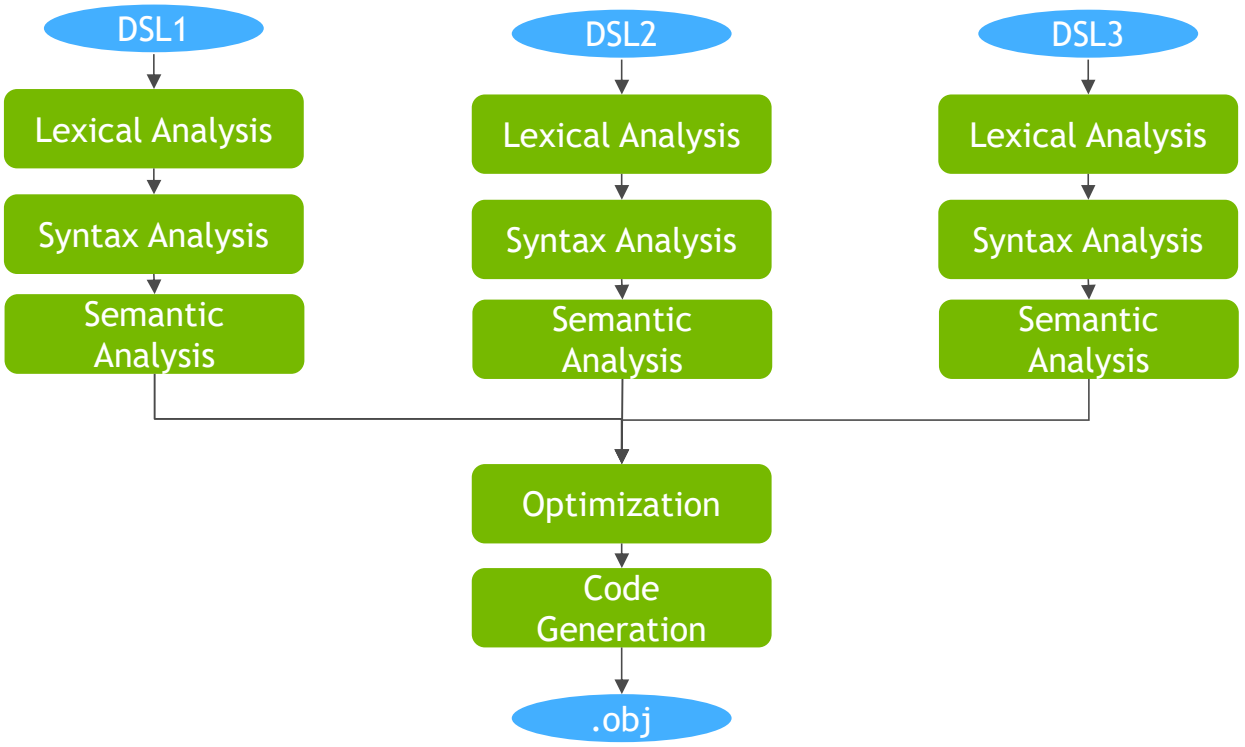
- Source to Source translators (aka Transpilers) take as input source code of one High Level Language and produce as output source code of another High Level Language
- Often, effort is made to preserve structure of source code
  - Modifications are done at level of AST
- Source to Source translators are used for
  - Compatibility
  - Porting Kits
  - Refactoring
- Examples : C++ to C, Cobol to Java, …

DOMAIN SPECIFIC LANGUAGES

- DSLs are Programming Languages custom built for a specific purpose
- Captures programmer intent more directly than a general purpose language such as C++
- DSLs represent domain specific data types, operations and programming model natively
- DSLs require custom compiler stack but benefit from re-usable compiler infrastructure that support multiple language front ends
- Examples
  - HTML for Web, SQL for Data Bases, P4 for Networking
  - Halide for Computer Vision, R for statistics
  - Many more…

DOMAIN SPECIFIC LANGUAGES

Multiple DSLs *may* share a common backend



COMPILER CAPABILITIES

In Summary

- *Compilers have ability to recognize, represent, analyze, transform and translate computer programs*
- Input to Compiler or Tools based on Compiler Technology is a program
- Output can be another program or information about the input program
- Key capabilities of a Compiler can be used in a variety of ways to process computer programs


THANK YOU
NVIDIA