Compilers and Code Optimization

EDOARDO FUSELLA

The course covers

- Compiler architecture
  - Front-end
  - Back-end
- LLVM
- Code optimization
- A case study: nu+

- Pre-requisite
  - Strong programming background in C, C++
  - Strong assembly language programming and computer architecture background
    - ISAs
    - RISC vs CISC
    - Assembly language coding
    - Datapath (ALU) and controller
    - Pipelining
    - Memory Hierarchies
    - Specialized Architectures
    - Out of order execution
    - …

Occupational Perspectives

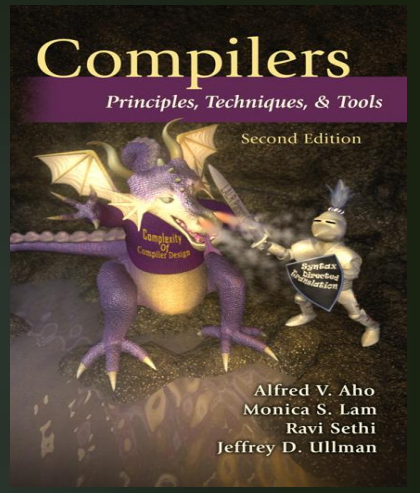
- Compiler Technologists are a scarce resource!
  - 1.5 Compiler engineers for every 1000 Software Engineers
  - 2 compiler jobs for every 100 software engineering jobs
- Typical Employers for Compiler Technologists
  - Semiconductor Industry: All major semiconductor companies have their own compiler teams; major players in Europe include STMicroelectronics (IT/FR), ARM (UK) and Sony (UK)
  - Compiler Development Industry: Specialized compiler development SMEs are another occupational option; major players in Europe include Associated Compiler Experts (NL) and Codeplay (UK)
  - Other areas: Electronic Design Automation; Security; System software

Occupational Perspectives

… also for non-specialists!

- EDA Tool Engineers
  - Most Electronic Design Automation tools are actually very similar to compilers
- Software Engineers
  - Many advanced SE techniques (automated refactoring, e.g.) require compiler techniques
- Computer Architects & Embedded Software Engineers
  - Architectural features cannot be easily exploited without compiler support
  - Understanding performance of optimized code is critical for embedded development
- Security Technologists
  - Malware analysis, reversing, and other activities typical of the security domain employ techniques not unlike those found in compilers

Resources



Book #1

- A. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques and tools*, Prentice-Hall, 2006

Book #2

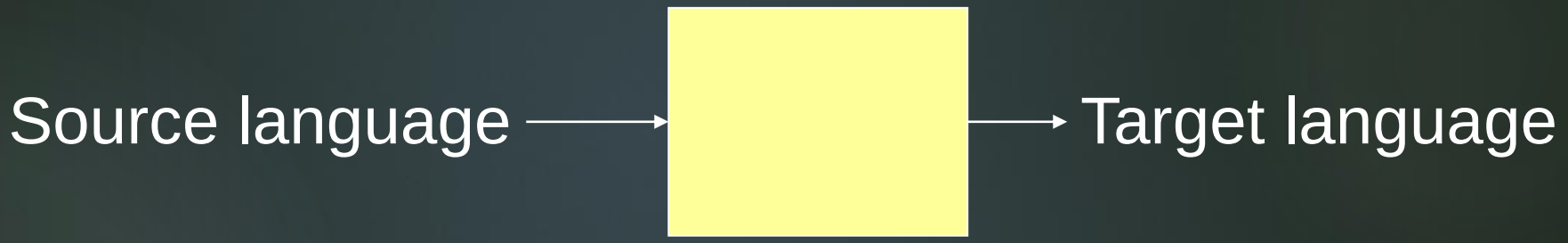
- A. Appel, *Modern compiler implementation in Java*, 2nd ed., Cambridge University Press, 2003

LLVM paper and site

- LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, by C. Lattner and V. Adve
- https://llvm.org/

What is a compiler?

A program that reads a program written in one language and translates it into another language.



- Traditionally, compilers go from high-level languages to low-level languages.
- Source program must be equivalent to target program.

Compiler Technology

- Why Compiler Technology
  - Code Transformations are critical for performance
  - Naive implementations can generate code 60% slower (or more) w.r.t. optimized code
  - Performance also reflects on resource usage (energy, availability)
- Structure of a compiler
  - Lexical analysis & parsing
  - Statement and Data Structure Lowering
  - Optimization: machine independent and machine-dependent
  - Code Generation

Aren’t compilers a solved problem?

“Optimization for scalar machines is a problem that was solved ten years ago.”

-- David Kuck, Fall 1990

- Architectures keep changing
- Languages keep changing
- Applications keep changing
- When to compile keeps changing

Changes in compilers

Role of compilers

- Bridge complexity and evolution in architectures, languages, & applications
- Help programmers with correctness, reliability, program understanding
- Compiler optimizations can significantly improve performance (up to 10x on conventional processors)
- Performance stability: one line change can dramatically alter performance

Performance Anxiety

- But does performance really matter?
  - Computers are really fast
  - Moore's law (roughly):
    hardware performance doubles every 18 months
- Real bottlenecks lie elsewhere:
  - Disk
  - Network
  - Human! (think interactive apps)
    - Human typing avg. 8 cps (max 25 cps)
    - Waste time "thinking"

Compilers Don’t Help Much

- Do compilers improve performance anyway?
  - **Proebsting’s law**
    (Todd Proebsting, Microsoft Research):
    - Difference between optimizing and non-optimizing compiler ~ 4x
    - Assume compiler technology represents 36 years of progress (actually more)
  - Compilers double program performance every 18 years!
    - Compiler optimization work makes only marginal contributions
    - Not quite Moore’s Law…

A Big BUT

- Why use high-level languages anyway?
  - Easier to write & maintain
  - Safer (think Java)
  - More convenient (think libraries, GC…)
- **But**: people will not accept massive performance hit for these gains
  - Compile with optimization!
  - Still use C and C++!!
  - Hand-optimize their code!!!
  - Even write assembler code !!!!
- Apparently performance does matter…

What qualities are important in a compiler?

1. Correct code
2. Output runs fast
3. Compiler runs fast
4. Compile time proportional to program size
5. Support for parallel compilation
6. Good diagnostics for syntax errors
7. Works well with the debugger
8. Good diagnostics for flow anomalies
9. Cross language calls
10. Consistent, predictable optimization

Compilers vs interpreters

- Interpreter: Reads a statement/code line at a time, performs the stated actions
  - Short delay before starting execution
  - Interactive execution (can execute a partially written code)
  - Avoids compilation overheads if the code is not executed
- Compiler: Translates a compilation unit to machine code
  - Optimizes across different statements
  - Faster execution once the code is compiled
  - Compilation needs to be performed only once

When to compile?

- Static Compiler: Translates source code to machine code well in advance of execution -- possibly even on a different machine
  - No compilation overheads at runtime
  - No possibility to adapt the code at runtime
  - One code version needs to be generated for each target platform
- JIT (**Just-in-Time**) Compiler: Translates each function when it is first invoked
  - Only code that is actually executed is translated
  - Code is targeted for the specific architecture, possibly taking into account runtime constraints (e.g., availability of resources)
  - Code may be optimized taking into account runtime information (e.g., runtime constants)
- AOT (**Ahead-of-Time**) Compiler: Translates each function before it is first invoked
  - Attempts to mix the two above styles to reap the benefits of both
  - Popular with virtual machines (DotNet, Java)

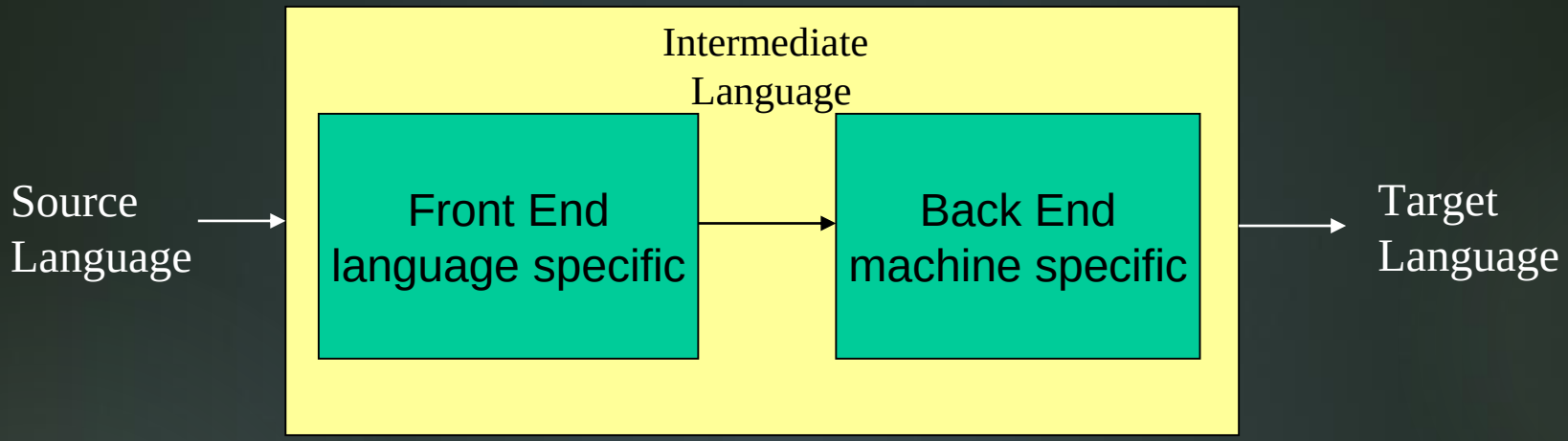
What to compile?

- Compilation units
  - Statement/Line of code: Small code regions
  - Function/Procedure: Medium sized code regions which are reusable through function call
  - Module/Source file: Large sized code regions
- Tradeoffs
  - Smaller: faster translation, may be used to provide interactive compilation and execution
  - Larger: more optimization, more complexity

Where to compile?

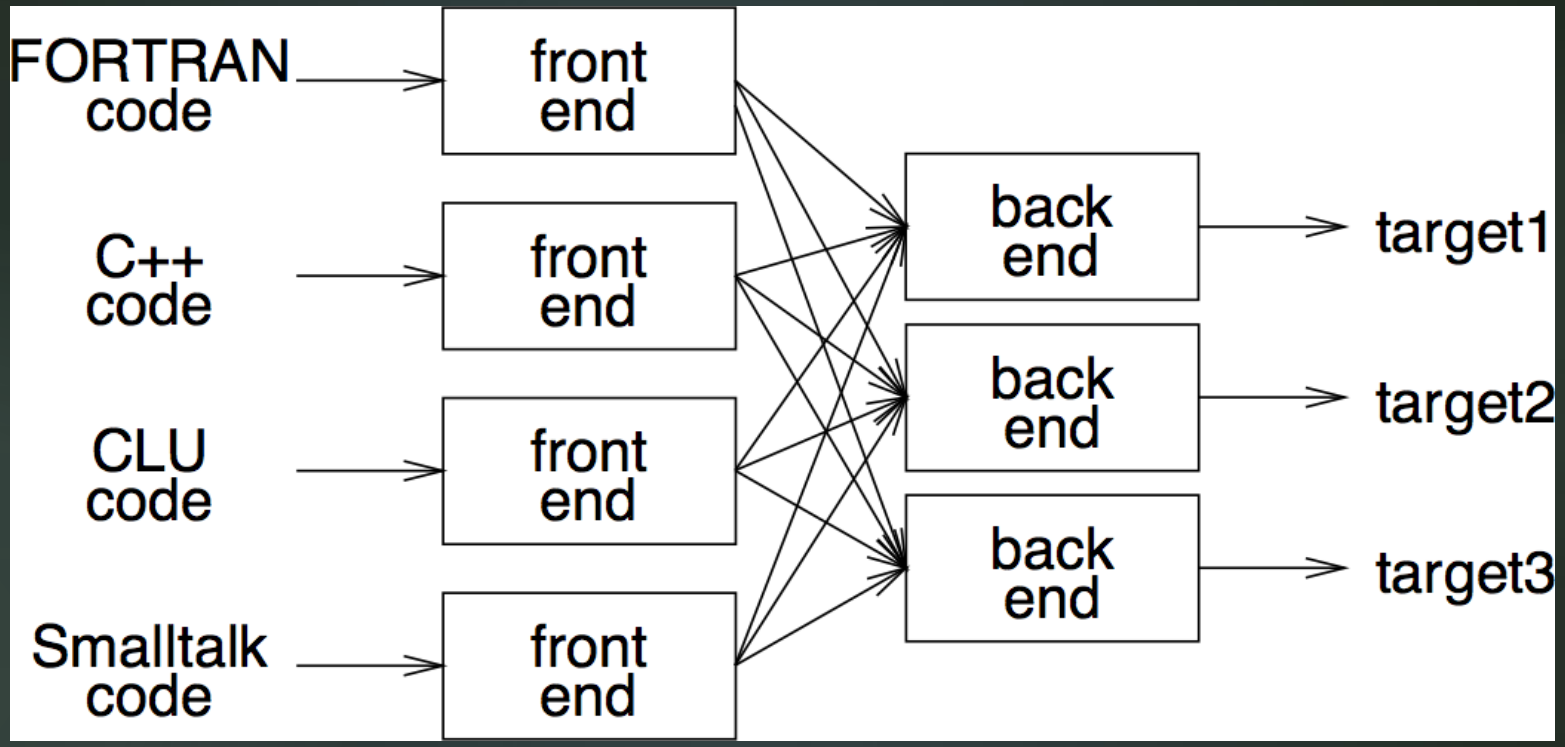
- Cross-compilation
  - Generally useful when machine T has insufficient resources for handling compilation tasks (e.g., embedded microcontrollers) or is not suited for general purpose processing (e.g., GPGPU)
  - Also useful for distributing to different platforms while having a single build machine
- Split compilation
  - Perform part of the compilation on machine H, producing an intermediate, portable code (e.g., bytecode), and part on machine T
  - Allows target-specific optimization to be performed more easily
  - Simplifies distribution of code on wide ranges of different target platforms

Overview of a compiler framework



- intermediate representation (IR)
- Separation of Concerns
  - front end maps legal code into IR
  - back end maps IR onto target machine
- simplify retargeting
- allows multiple front ends
- multiple passes = better code

A fallacy!



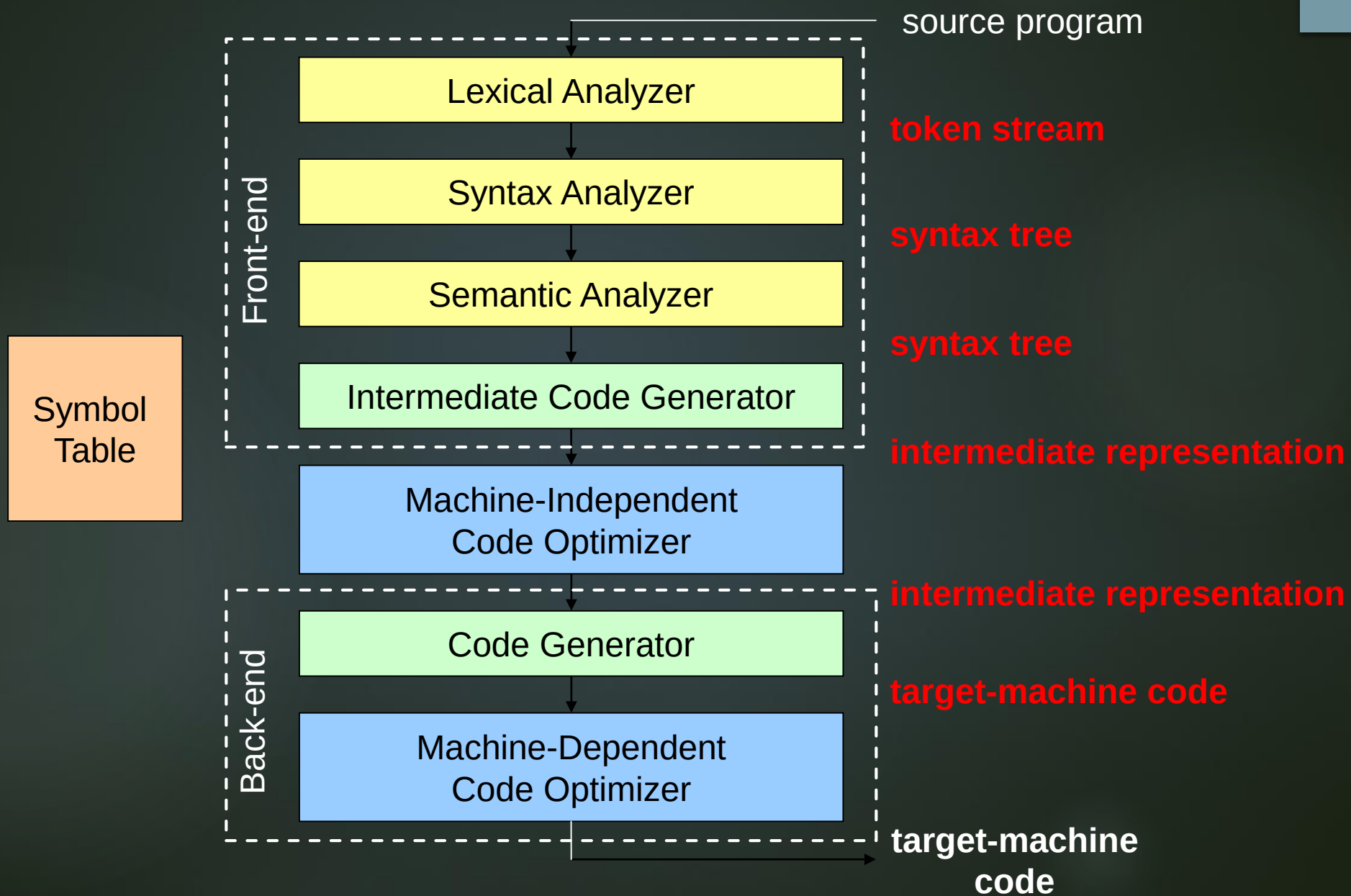
Front-end, IR and back-end must encode knowledge needed for all nxm combinations!

The Analysis-Synthesis Model

There are two parts to compilation:

- Front-end: Analysis determines the operations implied by the source program which are recorded in a tree structure (machine independent)
- Back-end: Synthesis takes the tree structure and translates the operations therein into the target program (machine dependent)

Phases of a Compiler



Lexical Analyzer

- Lexical analyzer are divided into a cascade of two process.
  - Scanning
    - Consists of the simple processes that do not require tokenization of the input.
      - Deletion of comments.
      - Compaction of consecutive whitespace characters into one.
  - Lexical analysis
    - The scanner produces the sequence of tokens as output.
      - Encode constants as tokens
      - Recognize Keywords and Identifiers
      - Store identifier names in a symbol table

Lexical Analyzer

Encode constants as tokens

- For a sequence of digits, the lexical analyzer must pass to the parser a token.
  - The token consists of the terminal along with an integer-valued attribute computed from the digits.
- Example
  - 31 + 28 + 29
  - <num, 31> <+> <num, 28> <+> <num,29>

token

token attribute

Lexical Analyzer

Recognize Keywords and Identifiers

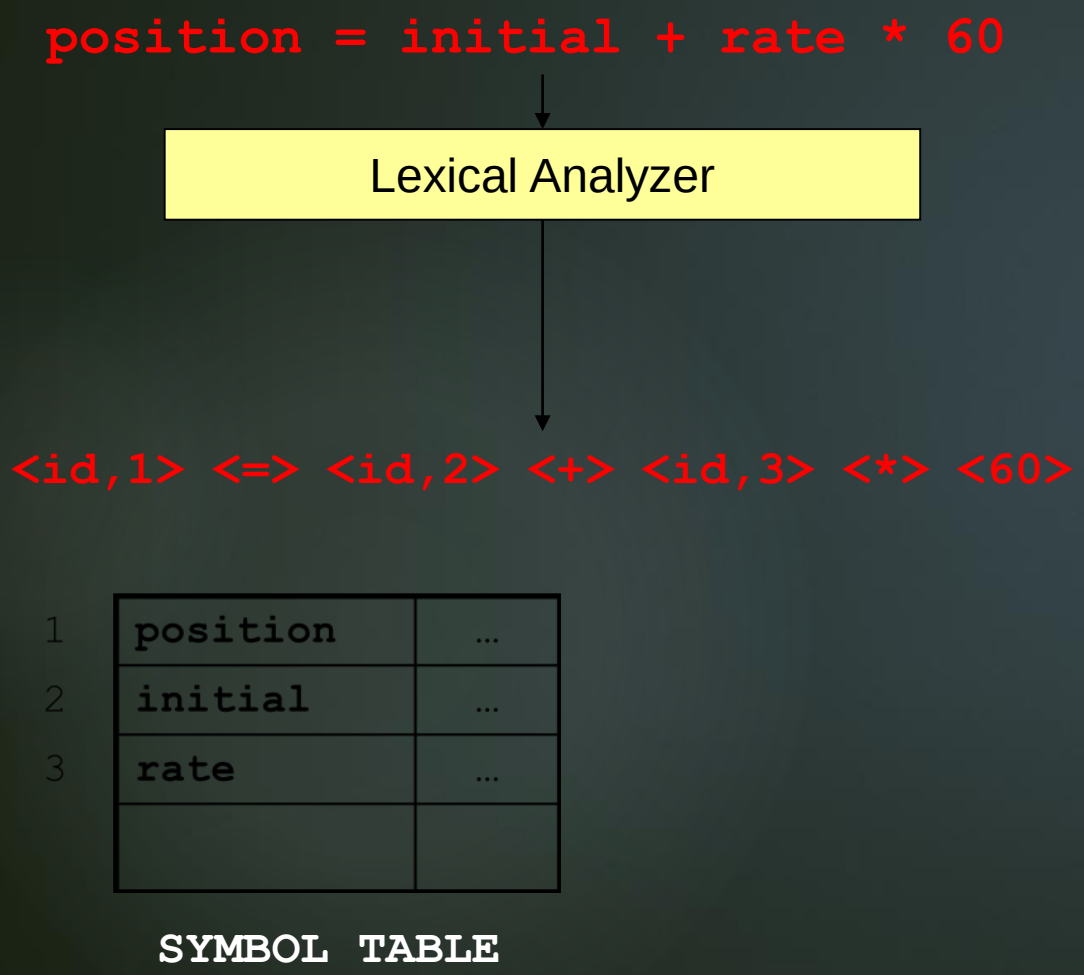
- Keyword
  - A fixed character string as punctuation marks or to identify constructs.
  - Example
    - for, while, if
- Identifier
  - Use to name variables, arrays, functions, and the like.
  - Parser treats identifiers as terminals.
  - Example
    - count = count + increment;
    - <id, ”count”> <=> <id, “count”> <+> <id, “increment”> <;>

Lexical Analyzer

Store identifier names in a symbol table

- The lexical analyzer uses a table to hold character strings.
  - Symbol table contain information about an identifer
    - character string (or lexeme)
    - its type
    - its position in storage
    - any other relevant information
  - A string table can be implemented by a hash table.
  - Single Representation

Lexical Analyzer

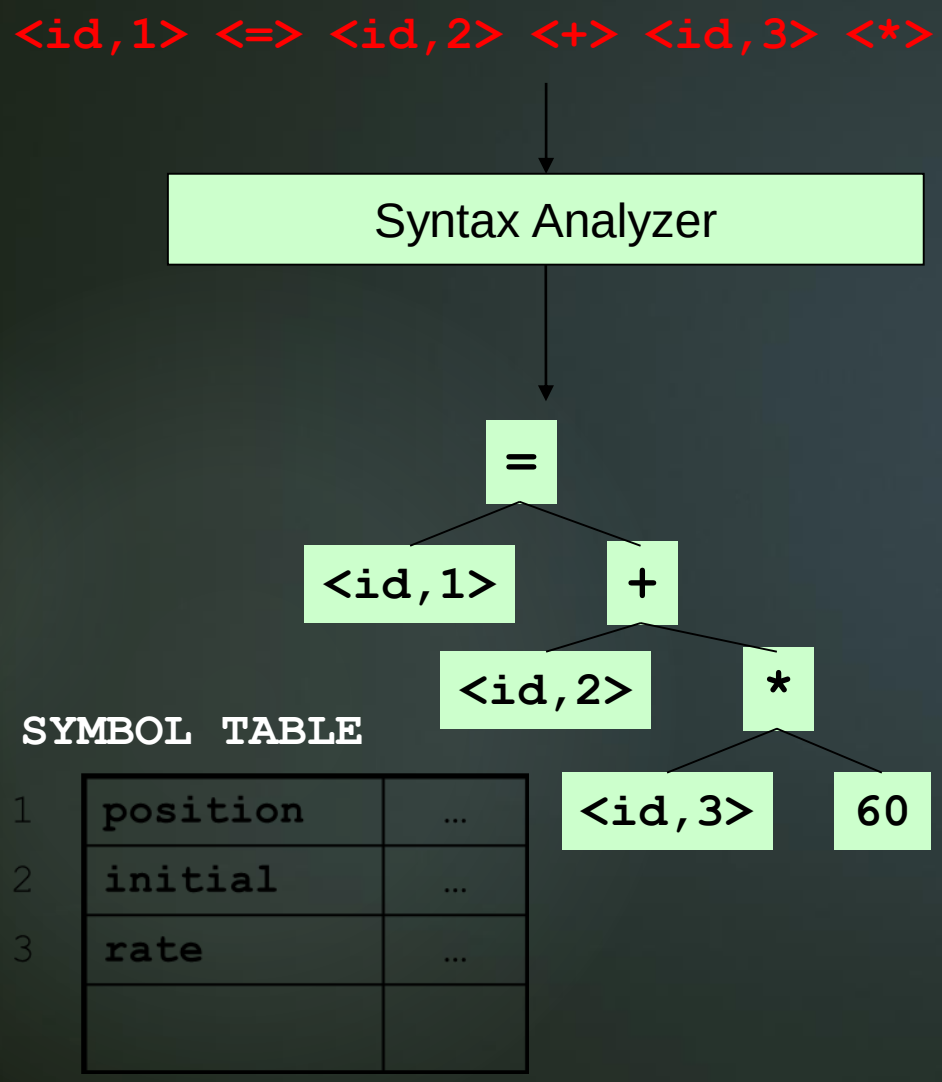


- Position is a lexeme that would be mapped into a token <id, 1>, where id is an abstract symbol standing for identifier and 1 points to the symbol-table entry for position.
- The assignment symbol = is mapped into the token <=>
- Initial is a lexeme that is mapped into the token <id, 2>, where 2 points to the symbol-table entry for initial.
- The add symbol + is mapped into the token <+>
- …

Syntax Analyzer
or parsing

- Syntax analyzer generates a parse tree (or syntax tree)
  - In a syntax tree, each interior node represents an operation and the children of the node represent the arguments of the operation.
- Token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

Syntax Analyzer



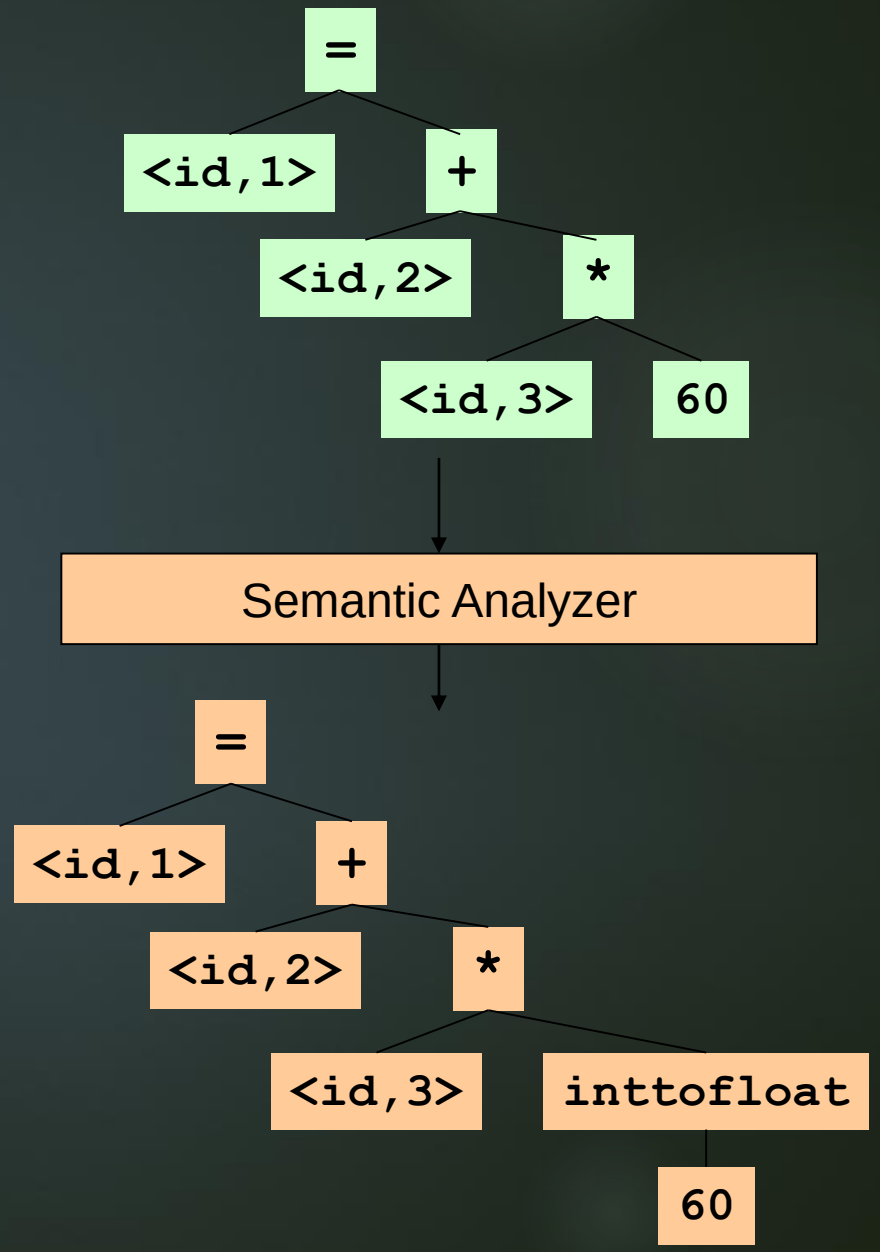
- The tree has an interior node labeled * with (id, 3) as its left child and the integer 60 as its right child. The node (id, 3) represents the identifer rate.
- The node labeled + indicates that we must add the result of this multiplication to the value of initial.
- The root of the tree, labeled =, indicates that we must store the result of this addition into the location for the identifer position.

Semantic Analyzer

- Check the source program for **semantic consistency**
- Static semantics check
  - Making sure identifiers are declared before use
  - Type checking for assignments and operators
  - Checking types and number of parameters to subroutines
  - Making sure functions contain return statements
- Simplify the structure of the parse tree (from parse tree to abstract syntax tree (AST))

Semantic Analyzer

- Suppose that position, initial, and rate have been declared to be floating-point numbers, and that the lexeme 60 by itself forms an integer.
- Semantic analyzer discovers that the operator * is applied to a floating-point number rate and an integer 60.
- The integer must be converted into a floating-point number.

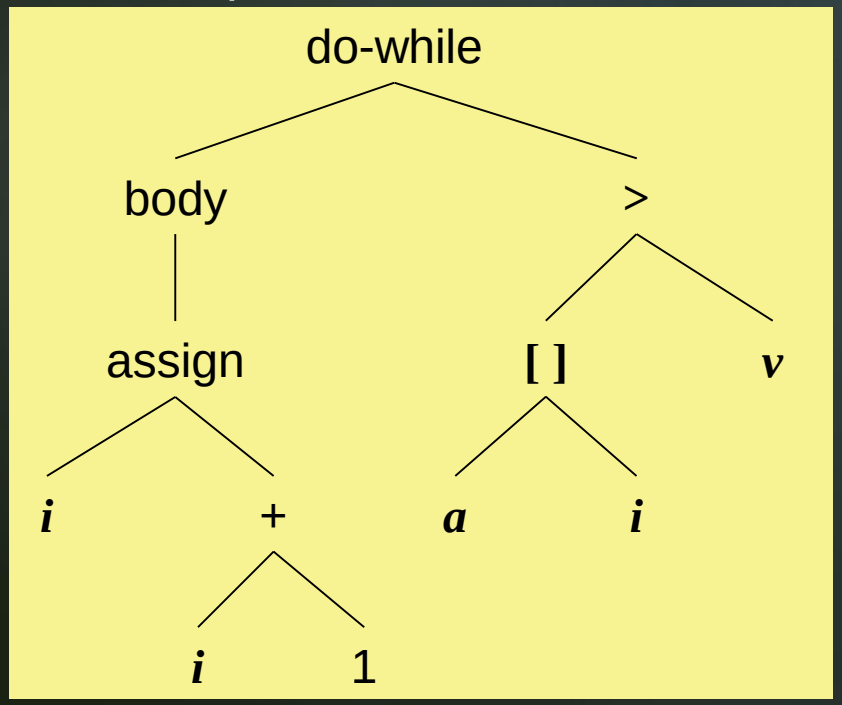


Intermediate Code Generator

- Go through the parse tree from bottom up, turning rules into code.
- e.g. A sum expression results in the code that computes the sum and saves the result
- Result: inefficient code in a machine-independent language

Two Forms of Intermediate Code

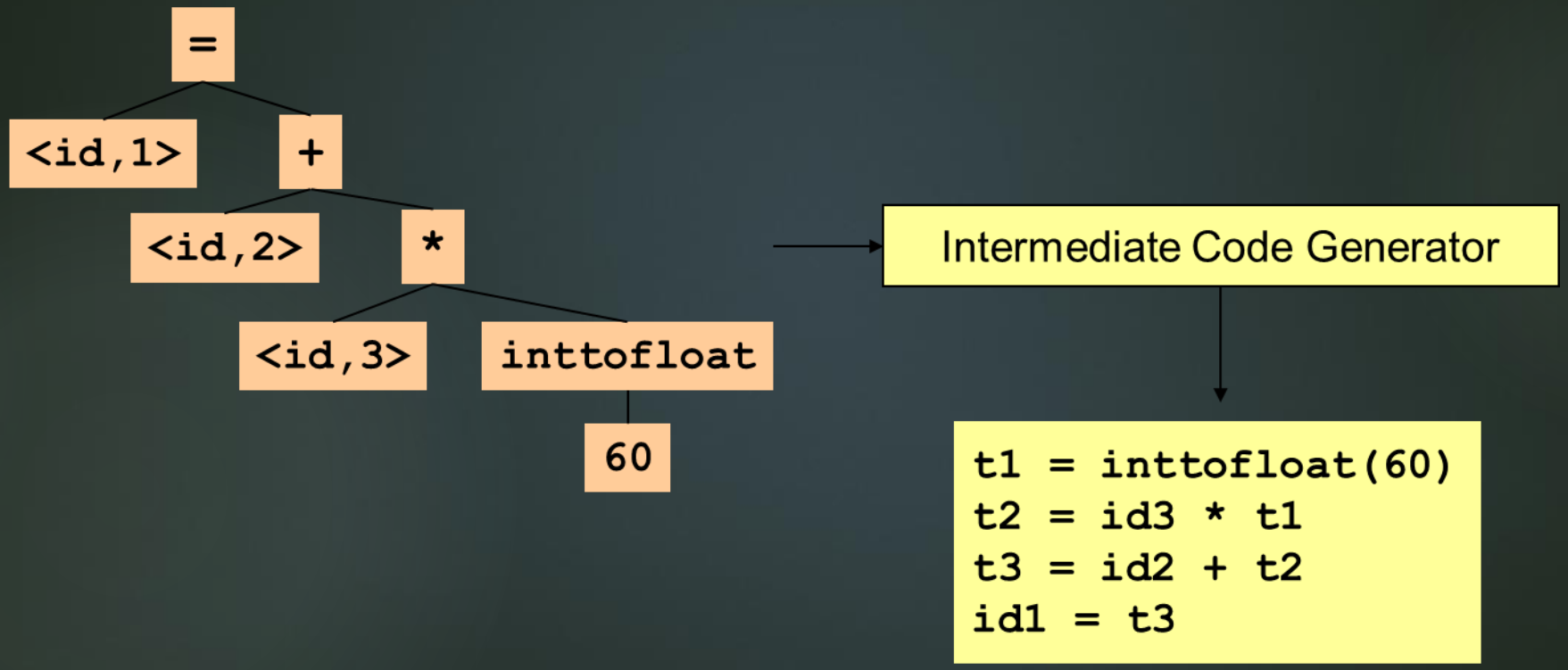
- Abstract syntax trees
  - Each interior node represents an operator
  - The children of the node represent the operands of the operator



- Tree-Address instructions
  - Each instruction has at most one operator on the right side.
  - The compiler must generate a temporary name to hold the value computed by an instruction.
  - Some instructions have fewer than three operands

```
1:   i = i + 1
2:   t1 = a [ i ]
3:   if  t1 > v  goto 1
```

Intermediate Code Generator

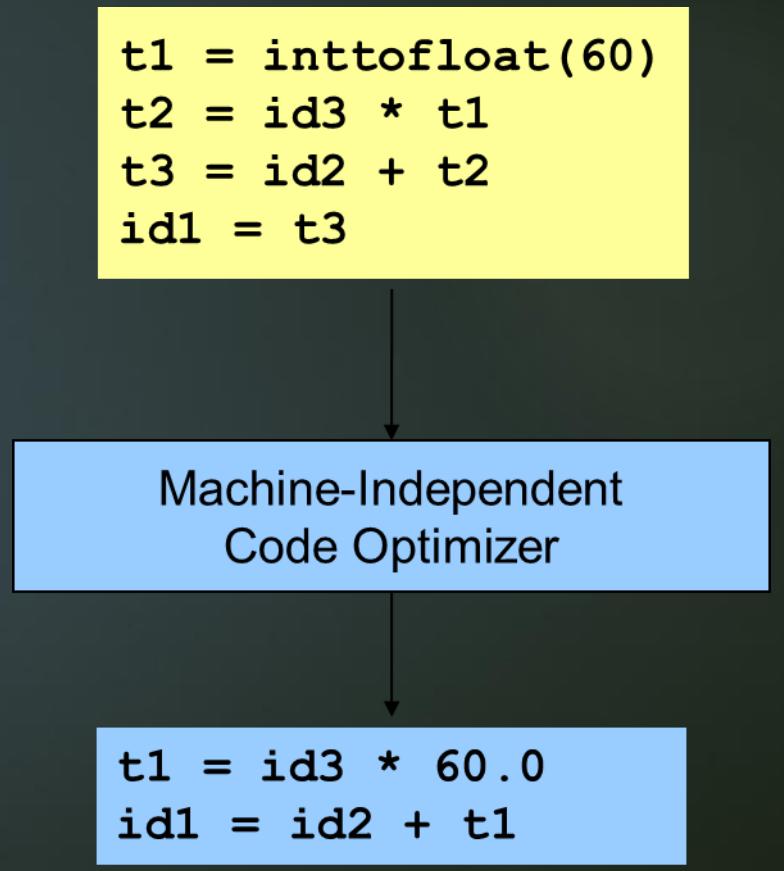


Machine-Independent Code Optimizer

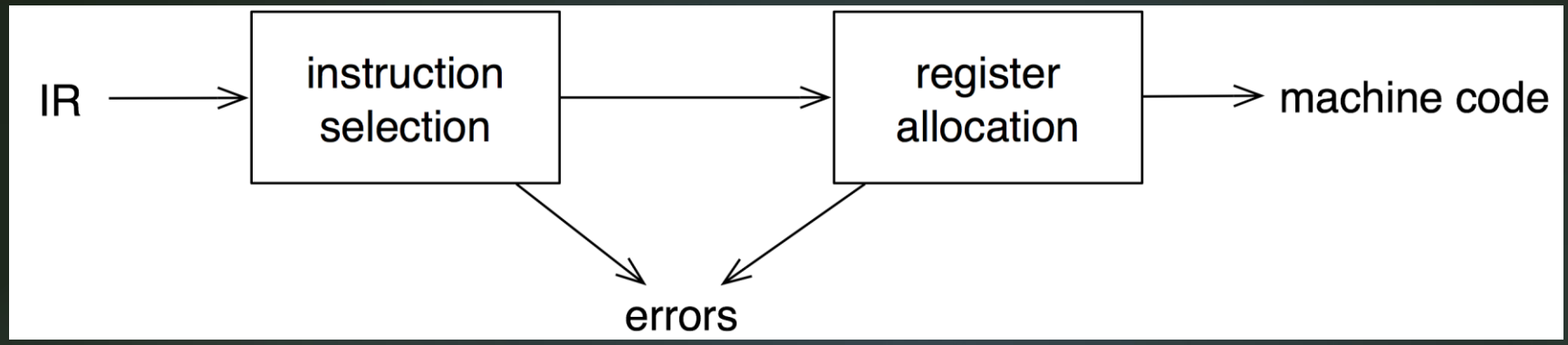
- Perform various transformations that improve the code, e.g.
  - Find and reuse common subexpressions
  - Take calculations out of loops if possible
  - Eliminate redundant operations

Machine-Independent Code Optimizer

- The inttofloat operation can be eliminated by replacing the integer 60 by the floating-point number 60.0.
- Moreover, t1 and t3 are used only once to transmit their values to t2 and id1 so the optimizer can remove them



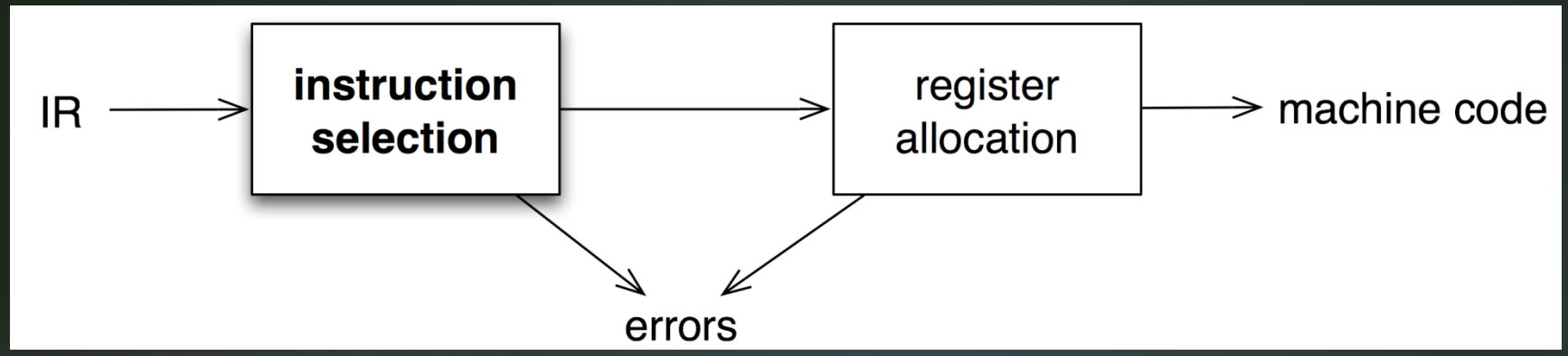
Code Generator



- Translate IR into target machine code
- Choose instructions for each IR operation
- Decide what to keep in registers at each point
- Ensure conformance with instruction set
- Make improvements that require specific knowledge of machine architecture, e.g.
  - Optimize use of available registers
  - Reorder instructions to avoid waits

Code Generator

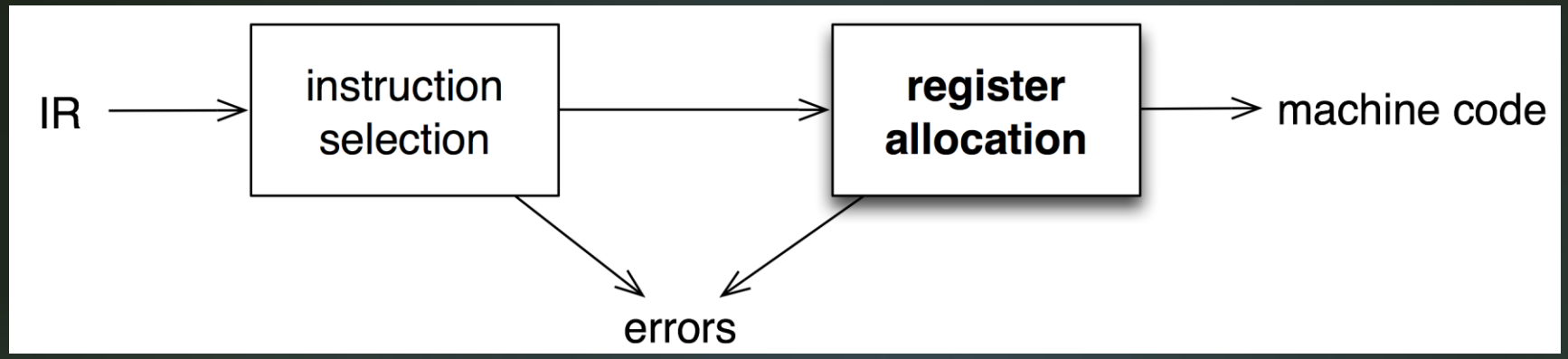
Instruction selection



- Produce compact, fast code
- Use available addressing modes
- Pattern matching problem
  - Ad hoc techniques
  - Tree pattern matching
  - String pattern matching
  - Dynamic programming

Code Generator

Register allocation



- Have value in a register when used
- Limited resources
- Changes instruction choices
- Can move loads and stores
- Optimal allocation is difficult

Code Generator

- Make improvements that require specific knowledge of machine architecture, e.g.
  - Optimize use of available registers
  - Reorder instructions to avoid waits