Introduction to Compilers
An Overview

Compiler construction

- Compiler writing is perhaps the most pervasive topic in computer science, involving many fields:
  - Programming languages
  - Architecture
  - Theory of computation
  - Algorithms
  - Software engineering
- In this course, you will put everything you have learned together. Exciting, right??

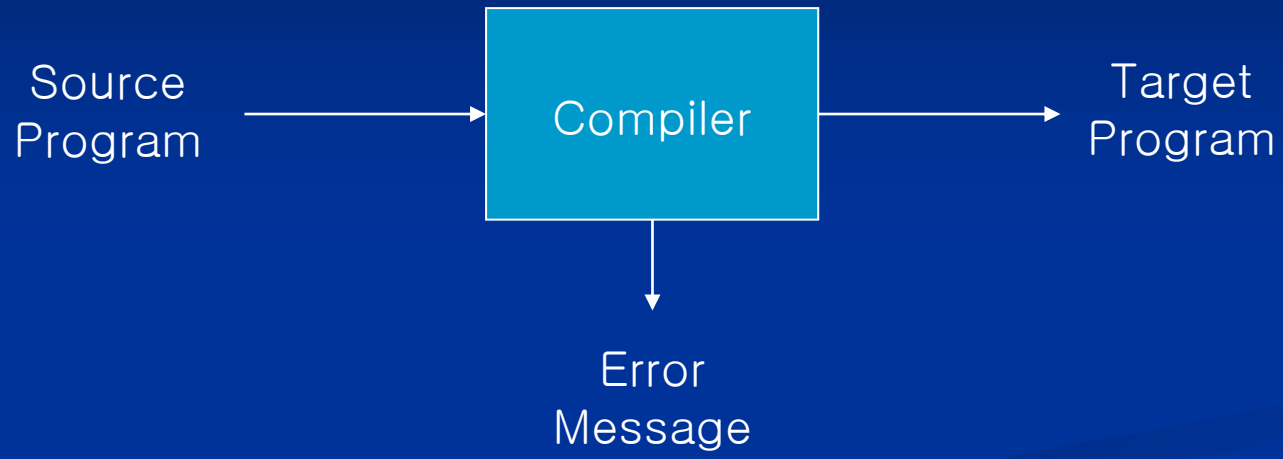
Excercise

- Consider the grammar shown below(<S> is your start symbol). Circle which of the strings shown on the below are in the language described by the grammar? There may be zero or more correct answers.
- Grammar:
  - <S> ::= <A> a <B> b
  - <A> ::= b <A> | b
  - <B> ::= <A> a | a
- Strings:

A) baab B) bbbabb C) bbaaaa D) baaabb E) bbbabab

- Compose the grammar for the language consisting of sentences of an equal number of **a**'s followed by an equal number of **b**'s. For example, **aaabbb** is in the language, **aabbb** is not, the empty string is not in the language.

What is a compiler?



- The source language might be
  - General purpose, e.g. C or Pascal
  - A “little language” for a specific domain, e.g. SIML
- The target language might be
  - Some other programming language
  - The machine language of a specific machine

Interpreter

- What is an interpreter?
  - A program that reads an *executable* program and produces the results of executing that program
- Target Machine: machine on which compiled program is to be run
- Cross-Compiler: compiler that runs on a different type of machine than is its target
- Compiler-Compiler: a tool to simplify the construction of compilers (YACC/JCUP)

Is it hard??

- In the 1950s, compiler writing took an enormous amount of effort.
  - The first FORTRAN compiler took 18 person-years
- Today, though, we have very good software tools
  - You will write your own compiler in a team of 3 in one semester!

Intrinsic interest

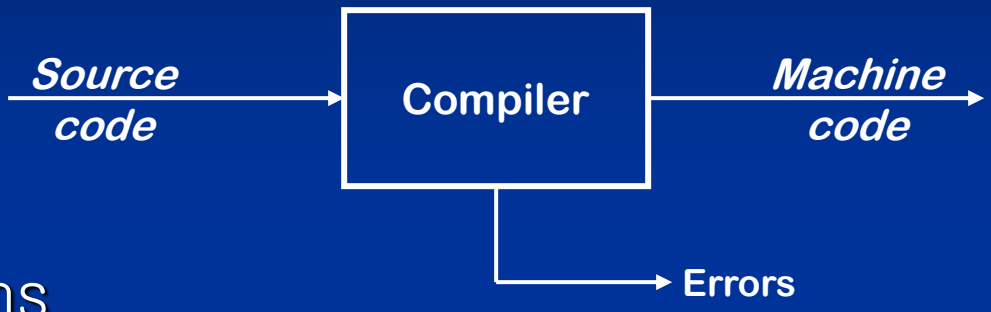
- Compiler construction involves ideas from many different parts of computer science

Artificial intelligence	Greedy algorithms Heuristic search techniques
Algorithms	Graph algorithms, union-find Dynamic programming
Theory	DFAs & PDAs, pattern matching Fixed-point algorithms
Systems	Allocation & naming, Synchronization, locality
Architecture	Pipeline & hierarchy management Instruction set use

Intrinsic merit

- Compiler construction poses challenging and interesting problems:
  - Compilers must do a lot but also run fast
  - Compilers have primary responsibility for run-time performance of target program.
  - Computer architects perpetually create new challenges for the compiler by building more complex machines
  - Compilers must hide that complexity from the programmer

High-level View of a Compiler



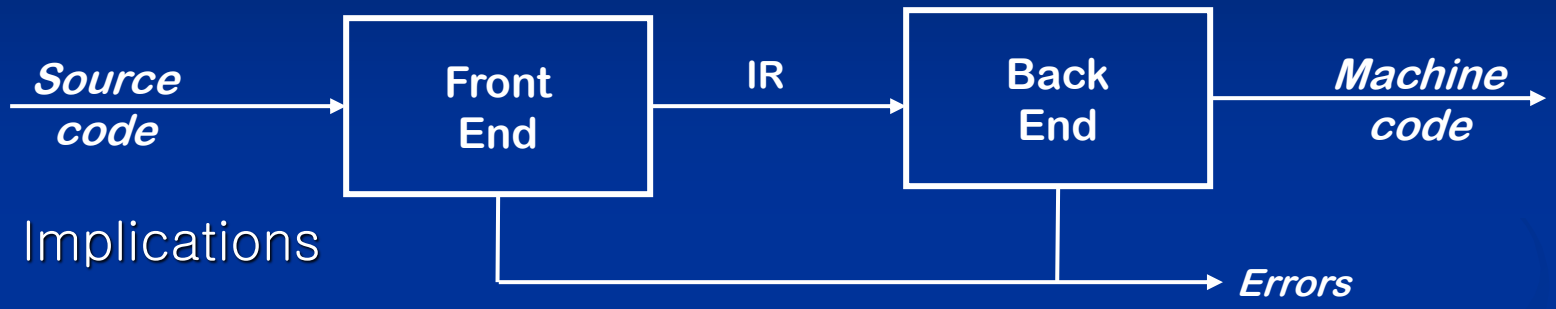
Implications

- Must recognize legal (and illegal) programs
- Must generate correct code
- Must manage storage of all variables (and code)
- Must agree with OS & linker on format for object code

Two Pass Compiler

- We break compilation into two phases:
  - ANALYSIS breaks the program into pieces and creates an intermediate representation of the source program.
  - SYNTHESIS constructs the target program from the intermediate representation.
- Sometimes we call the analysis part the FRONT END and the synthesis part the BACK END of the compiler. They can be written independently.

Traditional Two-pass Compiler

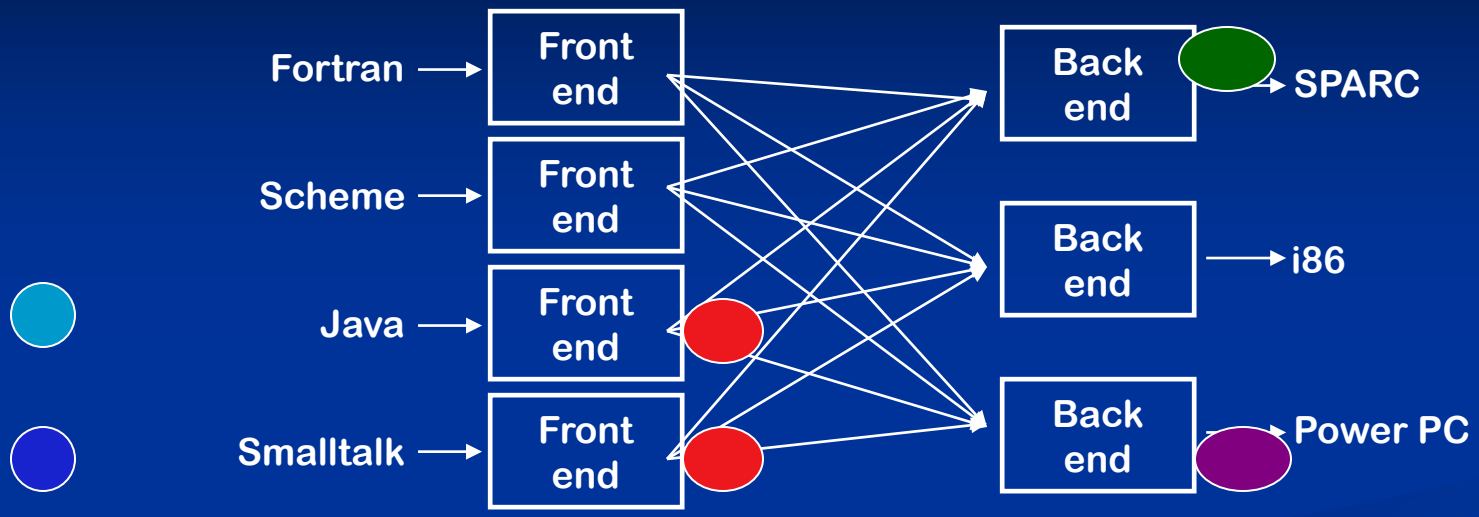


- Implications
  - Use an intermediate representation (IR)
  - Front end maps legal source code into IR
  - Back end maps IR into target machine code
  - Admits multiple front ends & multiple passes

(*better code*)

Typically, front end is O(n) or O(n log n), while back end is NP-Complete

A Common Fallacy



Can we build *n x m* compilers with *n+m* components?

- Must encode all language specific knowledge in each front end
- Must encode all features in a single IR
- Must encode all target specific knowledge in each back end

Limited success in systems with very low-level IRs

Source code analysis

- Analysis is important for many applications besides compilers:
  - STRUCTURE EDITORS try to fill out syntax units as you type
  - PRETTY PRINTERS highlight comments, indent your code for you, and so on
  - STATIC CHECKERS try to find programming bugs without actually running the program
  - INTERPRETERS don't bother to produce target code, but just perform the requested operations (e.g. Matlab)

Source code analysis

- Analysis comes in three phases:
  - LINEAR ANALYSIS processes characters left-to-right and groups them into TOKENS
  - HIERARCHICAL ANALYSIS groups tokens hierarchically into nested collections of tokens
  - SEMANTIC ANALYSIS makes sure the program components fit together, e.g. variables should be declared before they are used , types are matched

Linear (lexical) analysis

The linear analysis stage is called LEXICAL ANALYSIS or SCANNING.

Example:

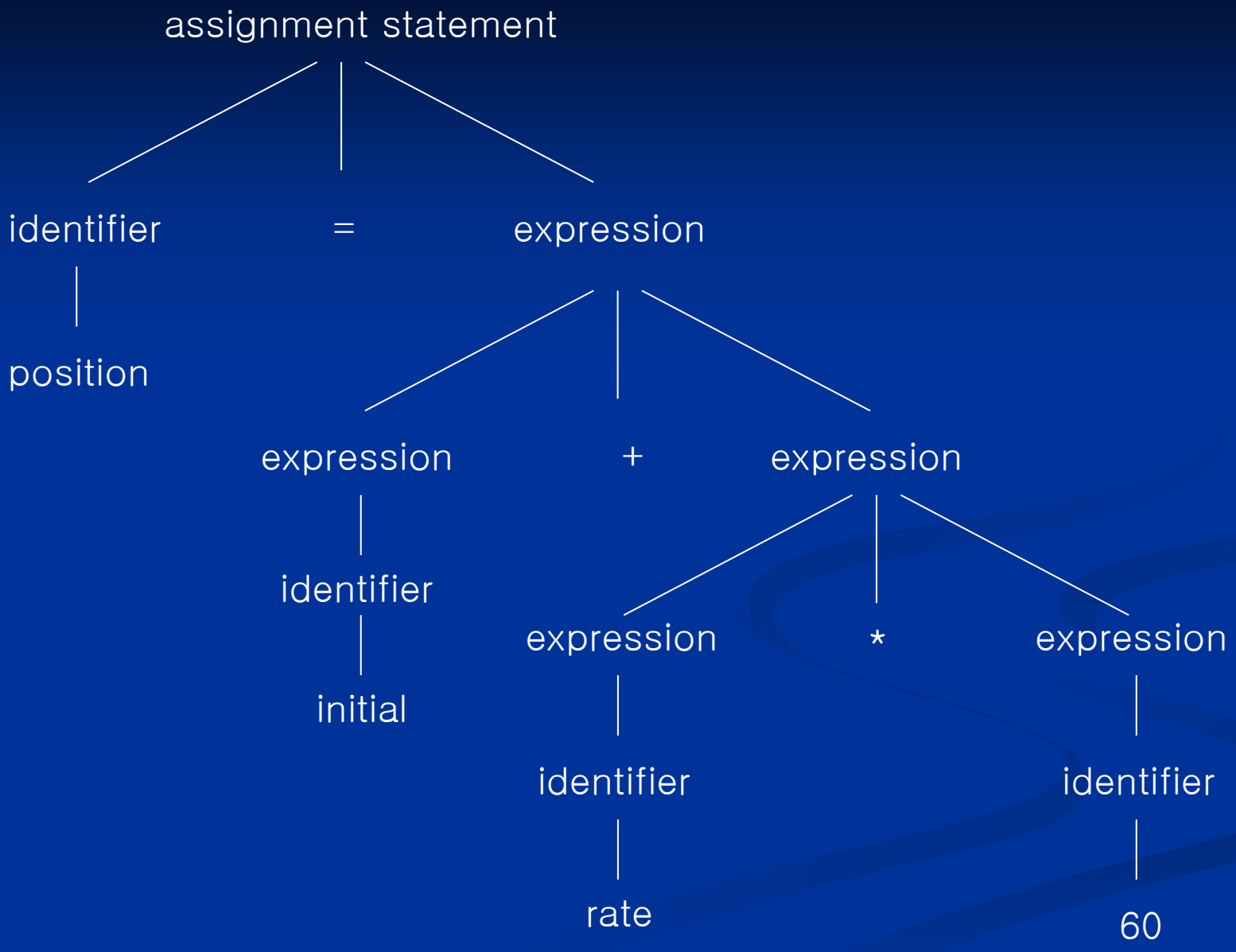
position = initial + rate * 60

gets translated as:

1. he IDENTIFIER “position”
2. The ASSIGNMENT SYMBOL “=”
3. The IDENTIFIER “initial”
4. The PLUS OPERATOR “+”
5. The IDENTIFIER “rate”
6. The MULTIPLICATION OPERATOR “*”
7. The NUMERIC LITERAL 60

Hierarchical (syntax) analysis

- The hierarchical stage is called SYNTAX ANALYSIS or PARSING.
- The hierarchical structure of the source program can be represented by a PARSE TREE, for example:


assignment statement
identifier
=
expression
position
expression
+
expression
identifier
expression
*
expression
initial
identifier
identifier
rate
60

Syntax analysis

- The hierarchical structure of the syntactic units in a programming language is normally represented by a set of recursive rules. Example for expressions:
  1. Any identifier is an expression
  2. Any number is an expression
  3. If expression1 and expression2 are expressions, so are

     expression1 + expression2

     expression1 * expression2

     (expression1)

Syntax analysis

- Example for statements:

1. If identifier1 is an identifier and expression2 is an expression, then identifier1 = expression2 is a statement.
2. If expression1 is an expression and statement2 is a statement, then the following are statements:

   while (expression1) statement2

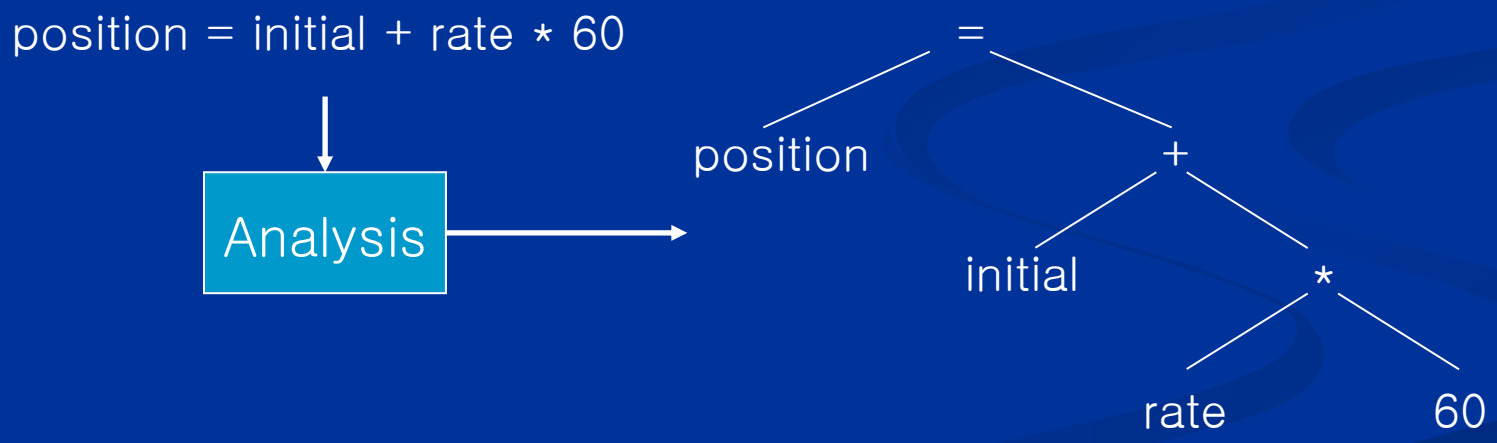
   if (expression1) statement2

Lexical vs. syntactic analysis

- Generally if a syntactic unit can be recognized in a linear scan, we convert it into a token during lexical analysis.
- More complex syntactic units, especially recursive structures, are normally processed during syntactic analysis (parsing).
- Identifiers, for example, can be recognized easily in a linear scan, so identifiers are tokenized during lexical analysis.

Source code analysis

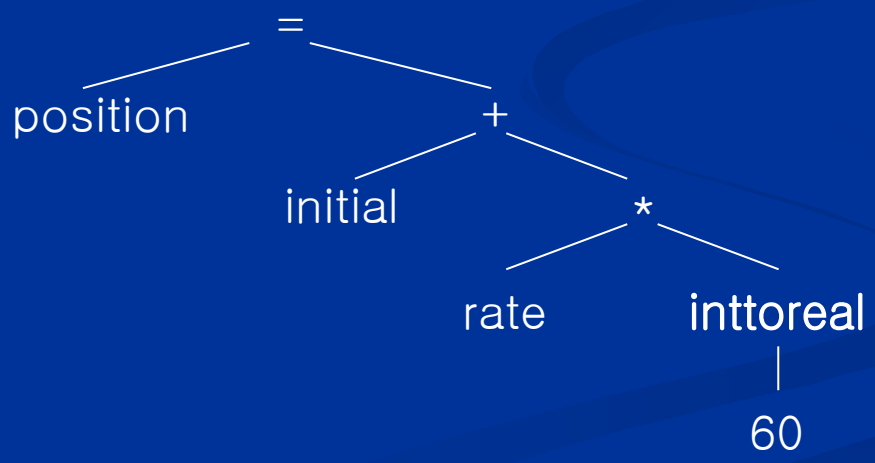
- It is common to convert complex parse trees to simpler SYNTAX TREES, with a node for each operator and children for the operands of each operator.



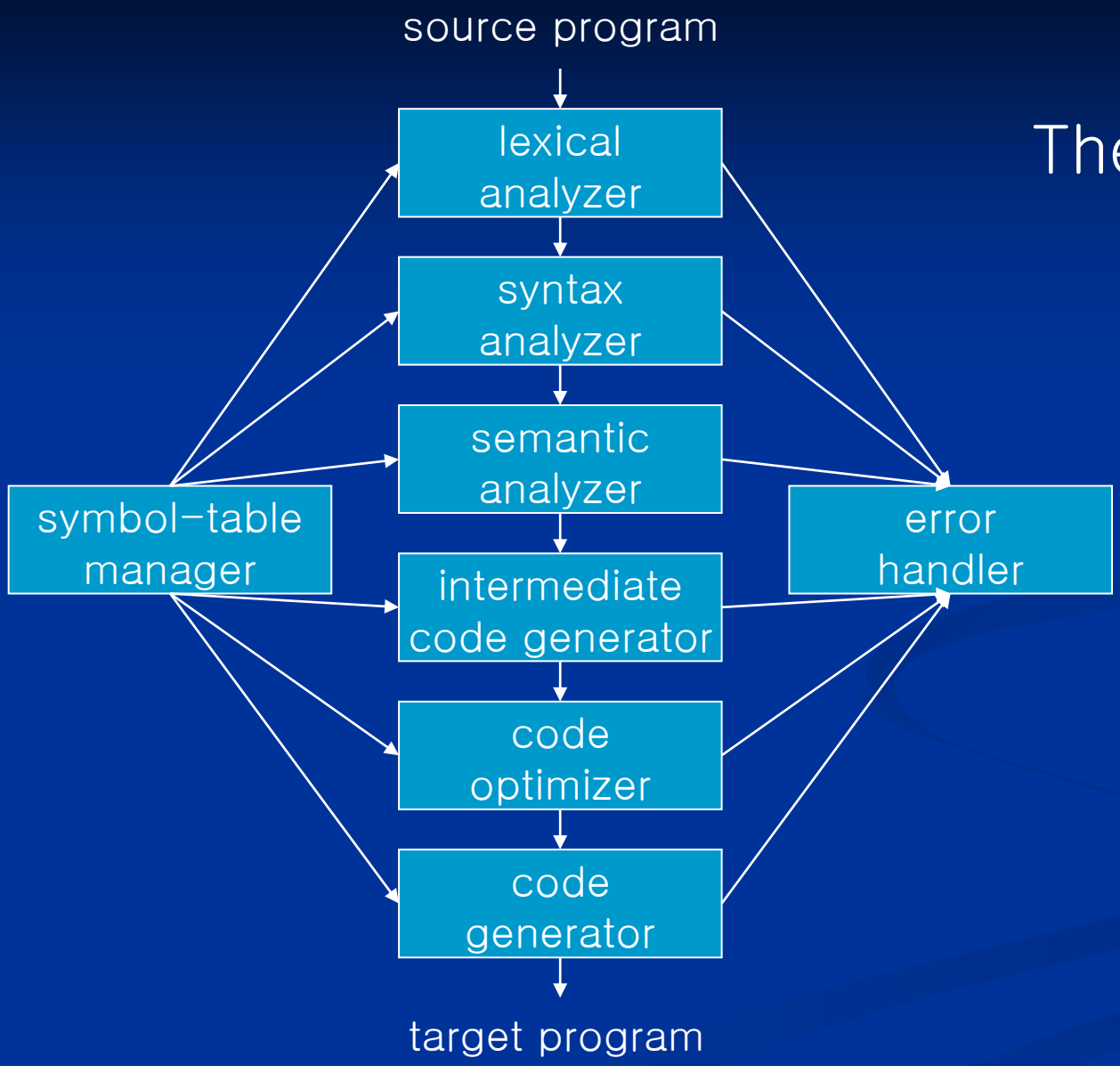
Semantic analysis

- The semantic analysis stage:
  - Checks for semantic errors, e.g. undeclared variables
  - Gathers type information
  - Determines the operators and operands of expressions

Example: if rate is a float, the integer literal 60 should be converted to a float before multiplying.



The rest of the process

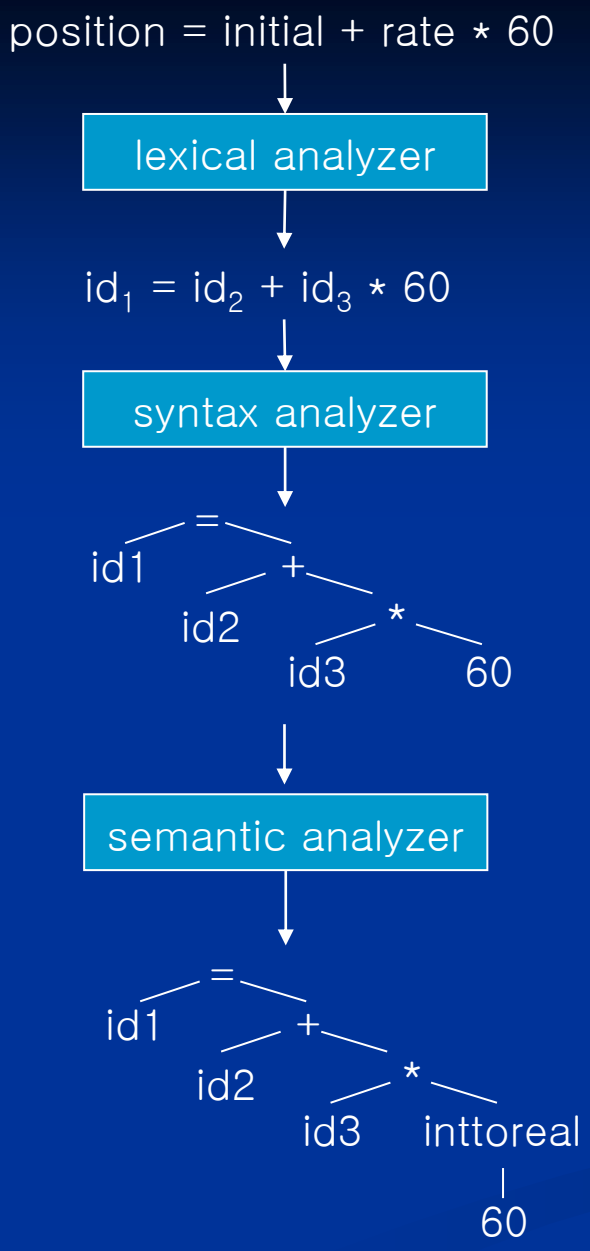


Symbol-table management

- During analysis, we record the identifiers used in the program.
- The symbol table stores each identifier with its ATTRIBUTES.
- Example attributes:
  - How much STORAGE is allocated for the id
  - The id's TYPE
  - The id's SCOPE
  - For functions, the PARAMETER PROTOCOL
- Some attributes can be determined immediately; some are delayed.

Error detection

- Each compilation phase can have errors
- Normally, we want to keep processing after an error, in order to find more errors.
- Each stage has its own characteristic errors, e.g.
  - Lexical analysis: a string of characters that do not form a legal token
  - Syntax analysis: unmatched { } or missing ;
  - Semantic: trying to add a float and a pointer



symbol table

1	Position	...
2	initial	...
3	rate	...
4		

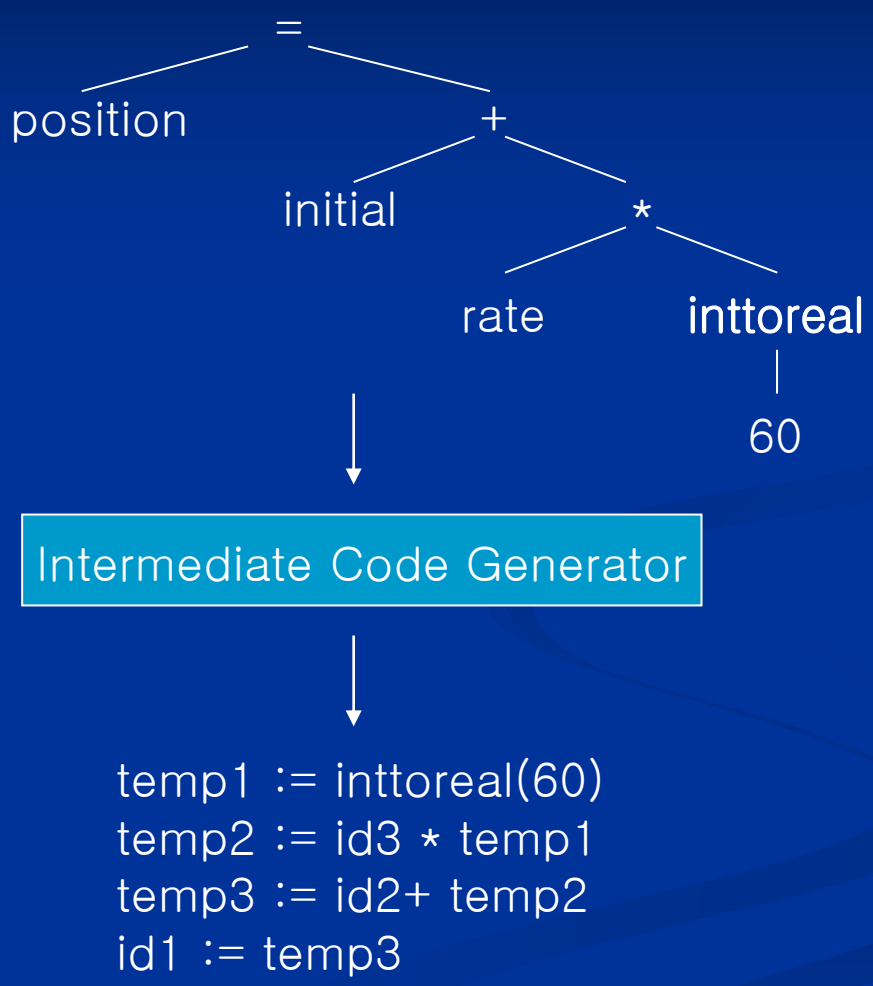
Internal Representations

Each stage of processing transforms a representation of the source code program into a new representation.

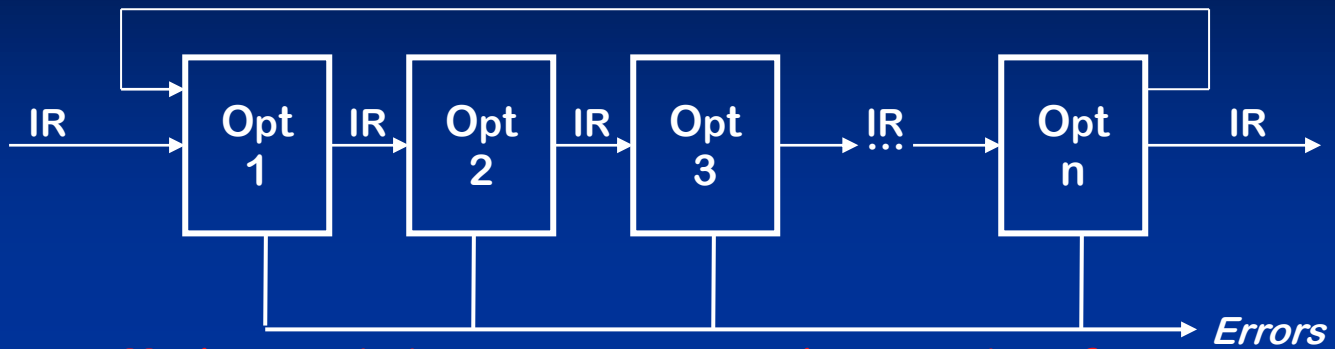
Intermediate code generation

- Some compilers explicitly create an intermediate representation of the source code program after semantic analysis.
- The representation is as a program for an abstract machine.
- Most common representation is “three-address code” in which all memory locations are treated as registers, and most instructions apply an operator to two operand registers, and store the result to a destination register.

Intermediate code generation



The Optimizer (or Middle End)



Modern optimizers are structured as a series of passes

Typical Transformations

← performance, code size, power consumption etc

- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Discover a redundant computation & remove it
- Remove useless or unreachable code
- Encode an idiom in some particularly efficient form

Code optimization

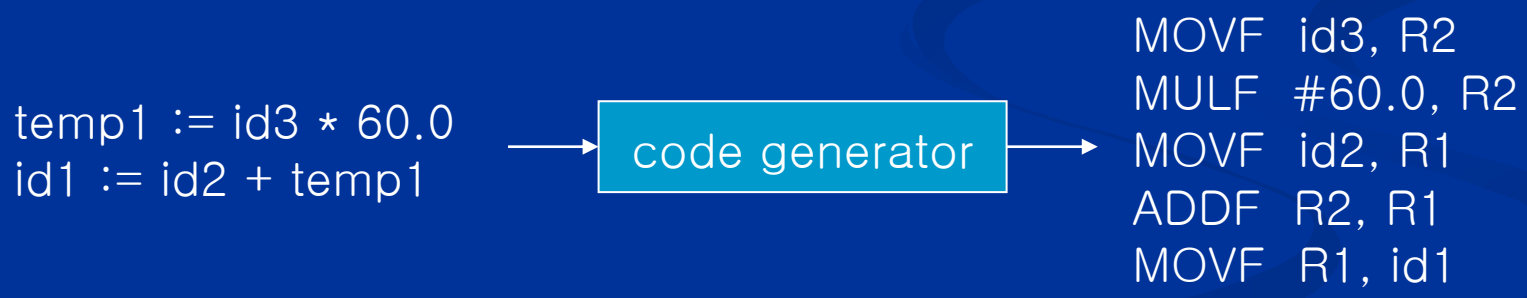
- At this stage, we improve the code to make it run faster.

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2+ temp2
id1 := temp3
```

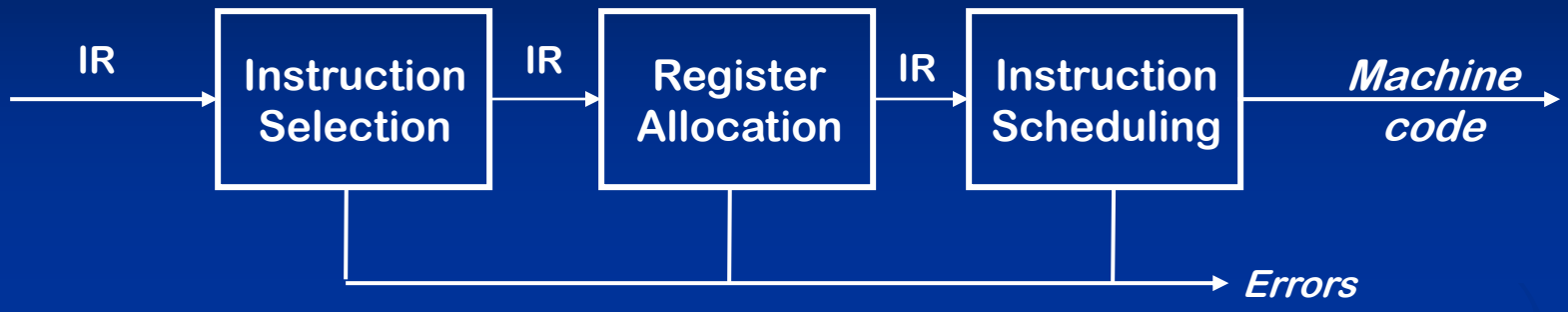

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

Code generation

- In the final stage, we take the three-address code (3AC) or other intermediate representation, and convert to the target language.
- We must pick memory locations for variables and allocate registers.



The Back End

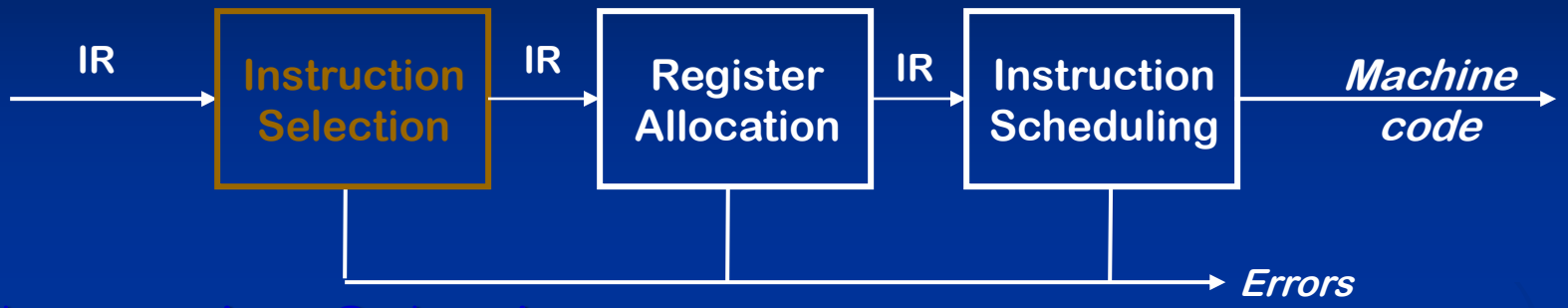


Responsibilities

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces

Automation has been *less* successful in the back end

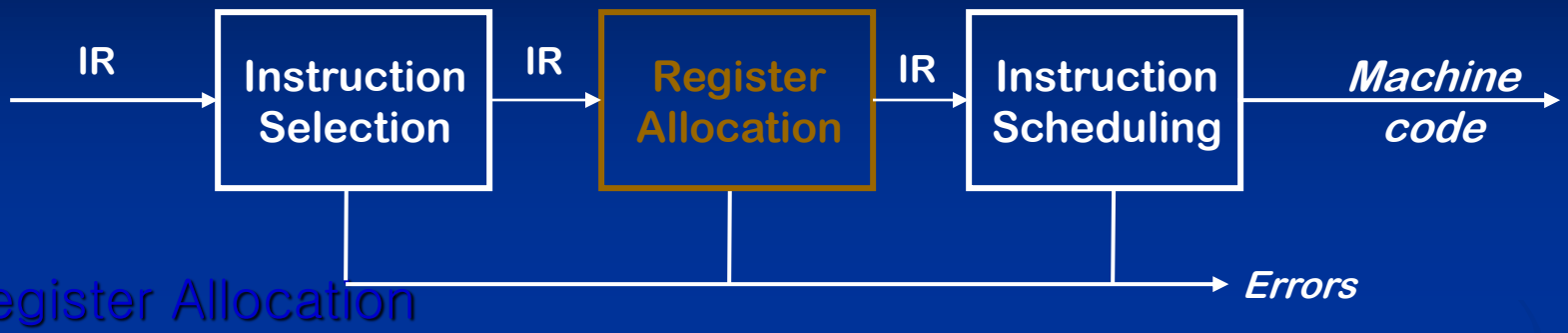
The Back End



Instruction Selection

- Produce fast, compact code
- Take advantage of target features such as addressing modes
- Usually viewed as a pattern matching problem
  - *ad hoc* methods, pattern matching, dynamic programming

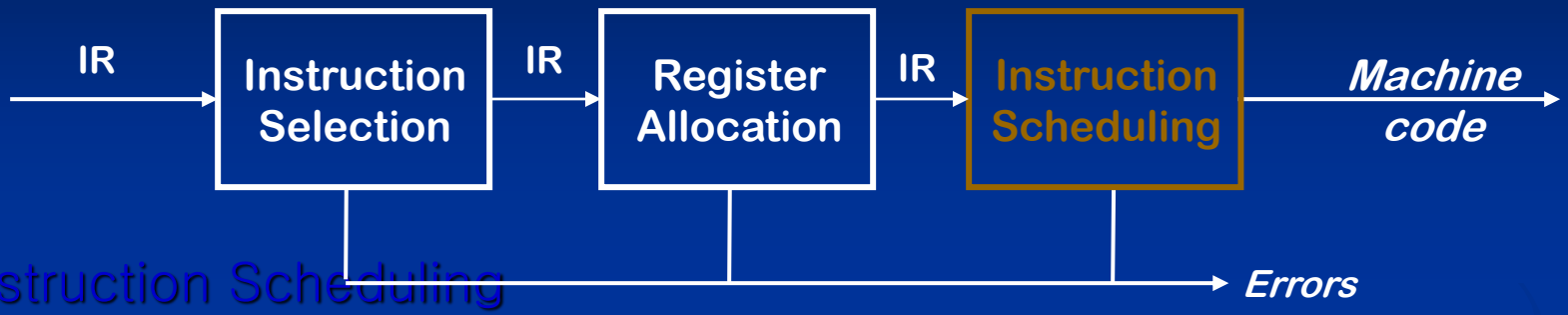
The Back End



- Have each value in a register when it is used
- Manage a limited set of resources
- Can change instruction choices & insert LOADs & STOREs
- Optimal allocation is NP-Complete (1 or *k* registers)

Compilers approximate solutions to NP-Complete problems

The Back End



- Avoid hardware stalls and interlocks
- Use all functional units productively
- Can increase lifetime of variables (changing the allocation)

Optimal scheduling is NP-Complete in nearly all cases

Heuristic techniques are well developed

Cousins of the compiler

- PREPROCESSORS take raw source code and produce the input actually read by the compiler
  - MACRO PROCESSING: macro calls need to be replaced by the correct text
    - Macros can be used to define a constant used in many places. E.g. #define BUFSIZE 100 in C
    - Also useful as shorthand for often-repeated expressions:
      #define DEG_TO_RADIANS(x) ((x)/180.0*M_PI)
      #define ARRAY(a,i,j,ncols) ((a)[(i)*(ncols)+(j)])
  - FILE INCLUSION: included files (e.g. using #include in C) need to be expanded

Cousins of the compiler

- ASSEMBLERS take assembly code and covert to machine code.
- Some compilers go directly to machine code; others produce assembly code then call a separate assembler.
- Either way, the output machine code is usually RELOCATABLE, with memory addresses starting at location 0.

Cousins of the compiler

- LOADERS take relocatable machine code and alter the addresses, putting the instructions and data in a particular location in memory.
- The LINK EDITOR (part of the loader) pieces together a complete program from several independently compiled parts.

Compiler writing tools

- We've come a long way since the 1950s.
- SCANNER GENERATORS produce lexical analyzers automatically.
  - Input: a specification of the tokens of a language (usually written as regular expressions)
  - Output: C code to break the source language into tokens.
- PARSER GENERATORS produce syntactic analyzers automatically.
  - Input: a specification of the language syntax (usually written as a context-free grammar)
  - Output: C code to build the syntax tree from the token sequence.
- There are also automated systems for code synthesis.