CMPSC 160
Translation of Programming Languages

Lecture 1: Introduction & Overview of Compilers

CMPSC 160

Instructor: Yufei Ding, Assist. Prof in CS

- Research areas: Programming systems (especially Compiler) for Machine + Quantum Computing
- Lectures: TR 12:30pm - 1:45pm @
- Office hours: Monday 2:00pm-3:00pm + By appointment.

Staff

- TAs: Pengfei Xu, Zhengyang Wang
- Discussion session: 5:00pm - 5:50p @GIRV 2108 and 6:00pm - 6:50pm @PHELP2514
- Office hours: TBD (will be posted on Piazza)

CMPSC 160

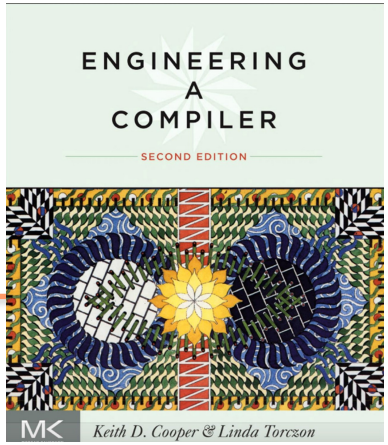
- Communication
  - Course website: http://www.cs.ucsb.edu/~yufeiding/cs160
    - (basic information, course schedule, project descriptions and due time)
  - Gradescope: https://www.gradescope.com/courses/448408
    - Submit your project here.
  - Piazza: https://piazza.com/ucsb/fall2022/cs160
    - Besides office hours, you could also post your general questions here.

- You can post questions about the class topics and projects to the piazza group. The TAs will respond as much as they can. You can also answer each other's questions BUT YOU CANNOT POST CODE.
  - If you have questions that you do not want to share with other students send e-mail to the TAs and the Instructor. ALWAYS INCLUDE BOTH THE TAs AND THE INSTRUCTOR.
  - Top 3 students that help answer most of the questions will receive 3 extra credits towards their final grade.

CMPSC 160

- Prerequisites: CMPSC 64 or EE 154, CMPSC 130A and CMPSC 138
- Prerequisites by Topic: Automata theory and formal languages; Programming in C++; Data structures, algorithms, and complexity, assembly language programming

Course Material



- We will use the following textbook:
  "Engineering a Compiler" by Keith D. Cooper and Linda Torczon
  Morgan Kaufman (Elsevier)

- There will be recommended reading material from the textbook and they will be posted on the class webpage

- **First reading assignment:** Read Chapters 1 and 2 from the textbook

- Slides will be uploaded to the class webpage.

Course Work

- The final grade will be determined according to the following weight
  - Final exams (20% based on lecture content, more from theoretical side).
  - Six Projects (75% mostly about detailed implementation).
  - Two Random in-class quizzes (5% for partial attendance check).

- For the projects, the grade breakdown is as follows:
  - 5% Project 1, 15% Project 2, 15% Project 3, 20% Project 4, 22.5% Project 5, 22.5% Project 6.

- The course work requires individual efforts.
  - We have developed several source-code plagiarism checkers.
  - The projects has been adapted from previous classes with changes.
  - Your code will be checked against both your classmates and code from previous classes.

Course Projects

- There will be 6 projects (https://sites.cs.ucsb.edu/~yufeiding/cs160/assignments.html)
  - The first two projects will be warm-up projects (project 1 is due next Thursday)
  - The goal of the last 4 projects will be to build a compiler
- Projects 3 to 6 cover all parts of the compilation process
- Compiler will work on a simple language
- You will use some well-known compiler construction tools and some C++ code that we provide

Course Goals

- To learn structure of compilers
- To construct a compiler for a small language
- To learn basic data structures used in compiler construction such as abstract syntax trees, symbol tables, three-address code, and stack machines
- To learn basic techniques used in compiler construction such as lexical analysis, top-down and bottom-up parsing, context-sensitive analysis, and intermediate code generation
- To learn software tools used in compiler construction such as lexical analyzer generators, and parser generators.

Today's lecture: Overview of Compiler

- Give you a big picture of compiler.
- It is a very challenging course. One of the most challenging undergraduate course:
- Even we just scratch the surface of a modern compiler, there are still lots of things to learn.
- Ask yourself whether you want to spend the time on it.

What is a Compiler?

- A compiler is a **program** that translates a program written in one language (source language) into an equivalent program in another language (target language), often along with optimizations, and meanwhile reports errors in the source program.

- Examples:
  - Domain-specific language encoded in Python/C++ to native C++
  - C to assembly code/machine code

Desirable Properties of Compilers

- Compiler must generate correct executable code
  - The input program and the output program must be *equivalent*, the compiler should preserve the meaning of the input program
- Compiler should provide good diagnostics for programming errors

- Output program should run fast
  - For optimizing compiler, we expect the output program to be more efficient than the input program
  - Optimizations should be consistent and predictable: different inputs
- Compiler itself should be fast
  - Compile time should be proportional to code size

- Compiler should support separate compilation
- Compiler should work well with debuggers
- …

Compiler vs. Interpreter

- What is an interpreter?
  - A program that reads an executable program and produces the results of executing that program

- C/C++ is typically compiled
- Python is typically interpreted
- Java is compiled to bytecodes, which are then interpreted or just-in-time compiled.

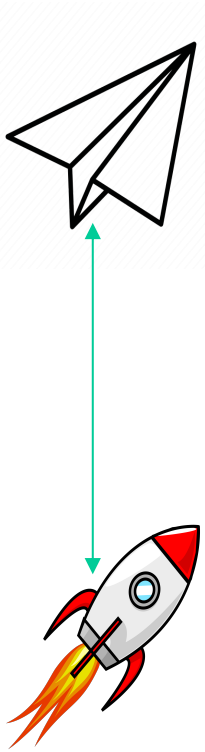
- What is their most significant functional difference of these two?

- Which one is more challenging to build? Why?

- When to use a compiler or an interpreter?

Effectiveness of A Compiler

- Performance of a matrix multiplication kernel (with n = 4,096) on Intel Xeon E5-2666 v3E, with **mostly just compiler software optimization:**



Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	+ tiling	1.79	1.70	11,772	76.782	9.184
8	Parallel divide-and-conquer	1.30	1.38	16,197	105.722	12.646
9	+ compiler vectorization	0.70	1.87	30,272	196.341	23.486
10	+ AVX intrinsics	0.39	1.76	53,292	352.408	41.677

[Charles Leiserson, MIT 6.172]

53, 292X performance difference!!!!!!!

An Alternative Way for Optimization → Library

Example:
NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

Interpreter + Library → Compiler for more in-depth optimization.

Machine Learning frameworks like TensorFlow are experiencing this path of development.

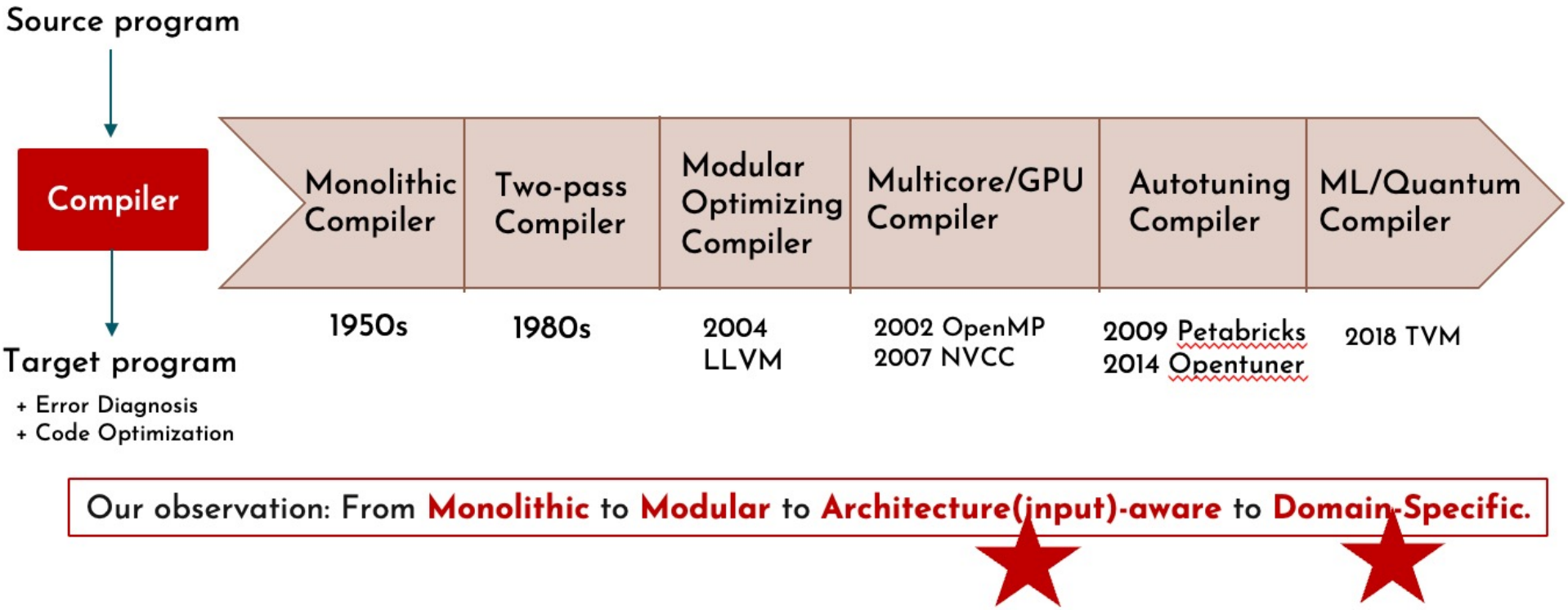
A Common Scope for Compiler Design

- Compilers provide an essential interface between applications and architectures.

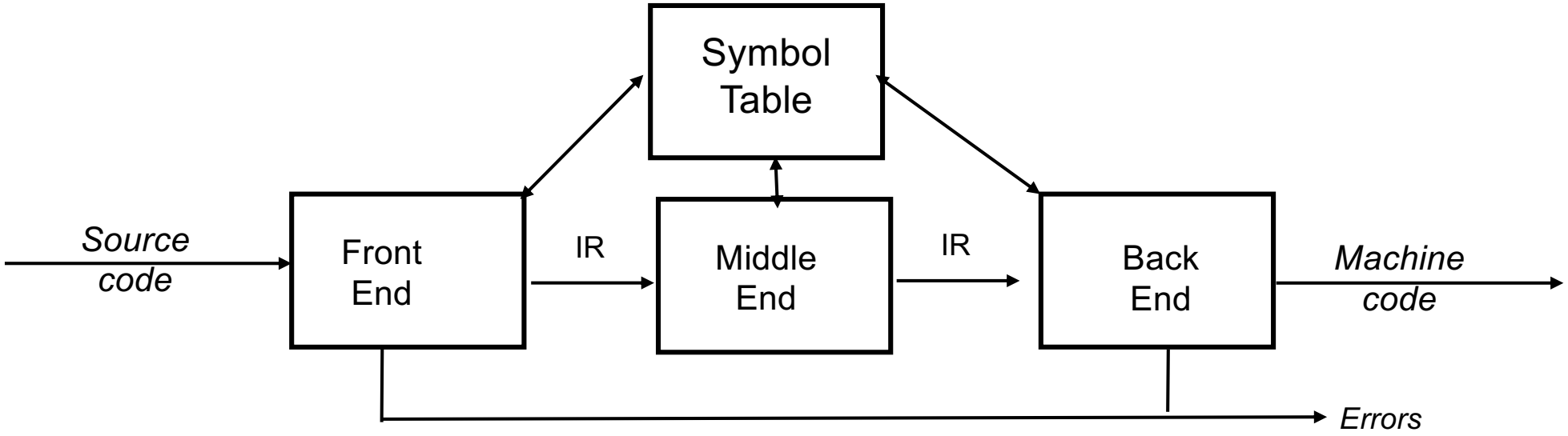
- Low level machine details:
  - Instruction Set Architecture (ISA): instruction selection
  - Memory Hierarchy: Registers and cache
  - Parallelism: Pipelines
  - Addressing mode
  - …

- Compilers efficiently bridge the gap and shield the application developers from low level machine details

History of Compiler Development



Traditional Three-pass Compiler



- Use an intermediate representation (IR)
- Front end maps legal source code into IR
- Middle end applies modular optimizations based on IR
- Back end maps IR into target machine code
- Admits multiple front ends and multiple passes
  - Typically, front end is O(n) or O(n log n), middle and back end are NP-complete
- Different phases of compiler also interact through the symbol table

Compiler - Example

- Source code
  - Written in a high-level programming language

```
//simple example
while (sum < total)
{
  sum = sum + x*10;
}
```

- Target code
  - Assembly language, which in turn is translated to machine code

```
L1: loadAI  R0, @sum => R1
    loadAI  R0, @total => R2
    cbr_LT  R1, R2 => L2
    jumpI   L3
L2: loadAI  R0, @sum => R1
    loadAI  R0, @x => R2
    loadI   10 => R3
    mult    R2, R3 => R2
    add     R1, R2 => R1
    storeAI R1 => R0, @sum
    jumpI   L1
L3: first instruction
    following the while
    statement
```

What is the Input?

- Input to the compiler is not

```
//simple example
while (sum < total)
{
  sum = sum + x*10;
}
```

- Input to the compiler is

```
//simple\bexample\nwhile\b(sum\b<\btotal)\b{\n\tsum\b=
\bsum\b+\bx*10;\n}\n
```

- How does the compiler recognize the keywords, identifiers, etc.?

Lexical Analysis (Scanning)

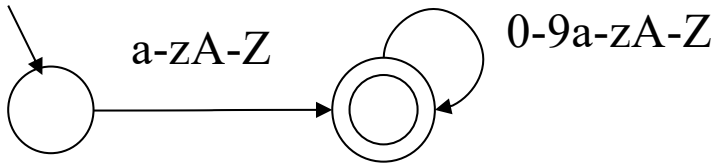
- The compiler scans the input file and produces a stream of ***tokens (categories of basic words in the langague)***

```
WHILE,LPAREN,<ID,sum>,LT,<ID,total>,RPAREN,LBRACE,
<ID,sum>,EQ,<ID,sum>,PLUS,<ID,x>,TIMES,<NUM,10>,
SEMICOL,RBRACE
```

- Each token has a corresponding ***lexeme (a particular instance for that category)***, the character string that corresponds to the token
  - For example, "`while`" is the lexeme for token `WHILE`
  - "`sum`", "`x`", "`total`" are lexemes for token `ID`

Lexical Analysis (Scanning)

- Compiler uses a set of patterns to specify valid tokens
  - tokens: LPAREN, WHILE, ID, NUM, etc.
- Each pattern is specified as a **regular expression**
  - LPAREN should match: `(`
  - WHILE should match: `while`
  - ID should match: [a-zA-Z][0-9a-zA-Z]*
- It uses **finite automata** to recognize these patterns



ID automaton

Lexical Analysis (Scanning)

- During the scan the lexical analyzer gets rid of the ***white space*** (`\b`, `\t`, `\n`, etc.) and ***comments***

- Important additional task: Error messages!
  - `Var%1` → Error! Not a token!
  - `while` → Error? It matches the identifier token.

- Natural language analogy: Tokens correspond to words and punctuation symbols in a natural language

Syntax Analysis (Parsing)

- How does the compiler recognize the structure of the program?
  - Loops, blocks, procedures, nesting?

- Specify the structure of program using recursive rules: **context free grammars**
- Parse the stream of tokens based on these recursive rules to get the parse tree
  - program will be on the leaves of the tree

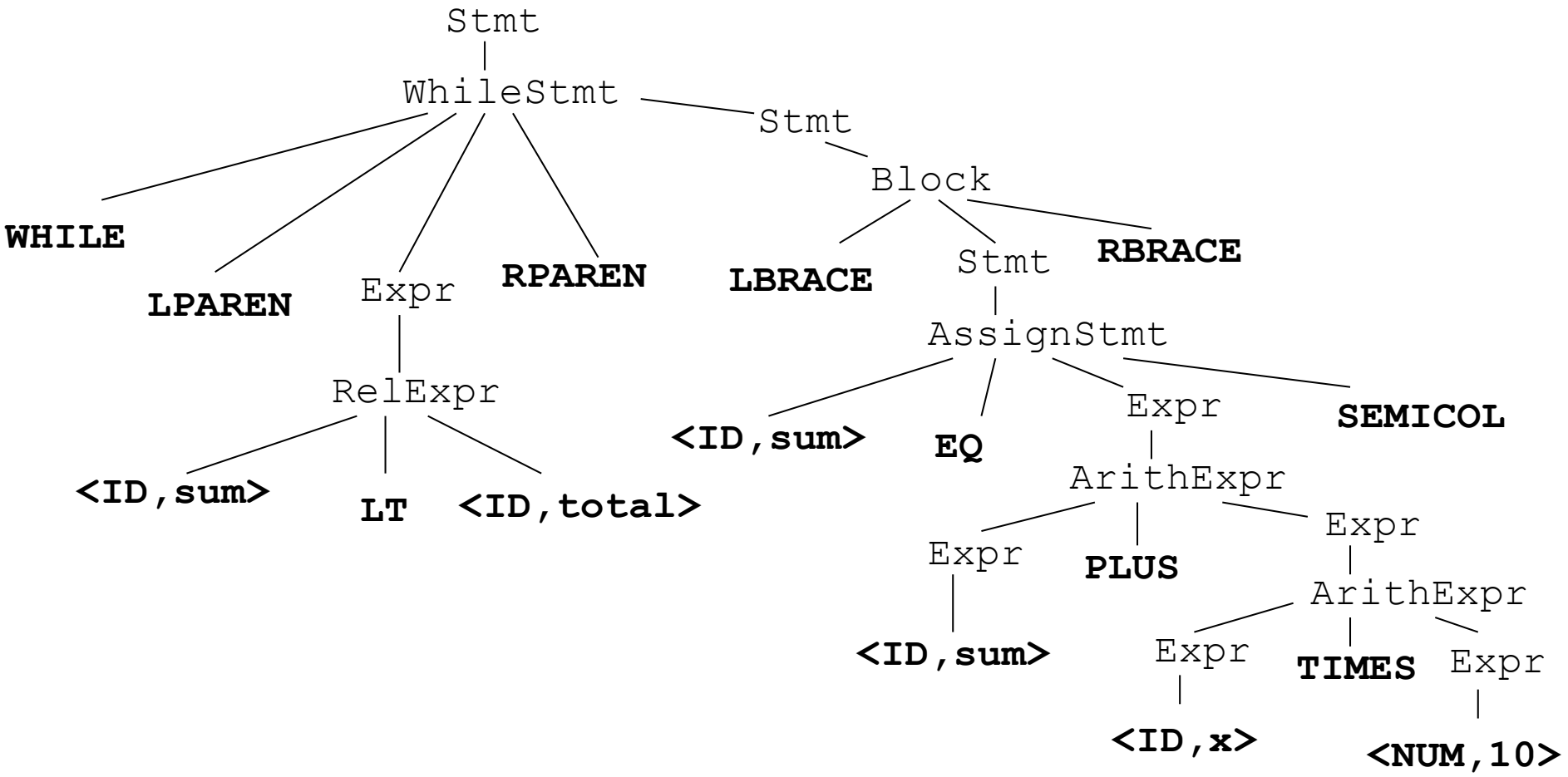
Syntax Analysis (Parsing)

- The syntax of a programming language is defined by a set of recursive rules. These sets of rules are called **context free grammars**.

```
Stmt → WhileStmt | Block | ...
WhileStmt → WHILE LPAREN Expr RPAREN Stmt
Expr →  RelExpr | ArithExpr | ...
RelExpr → ...
```

- Compilers apply these rules to produce the **parse tree**
- Again, important additional task: Error messages!
  - Missing semicolon, missing parenthesis, etc.
- Natural language analogy: It is similar to parsing English text. Paragraphs, sentences, noun-phrases, verb-phrases, verbs, prepositions, articles, nouns, etc.

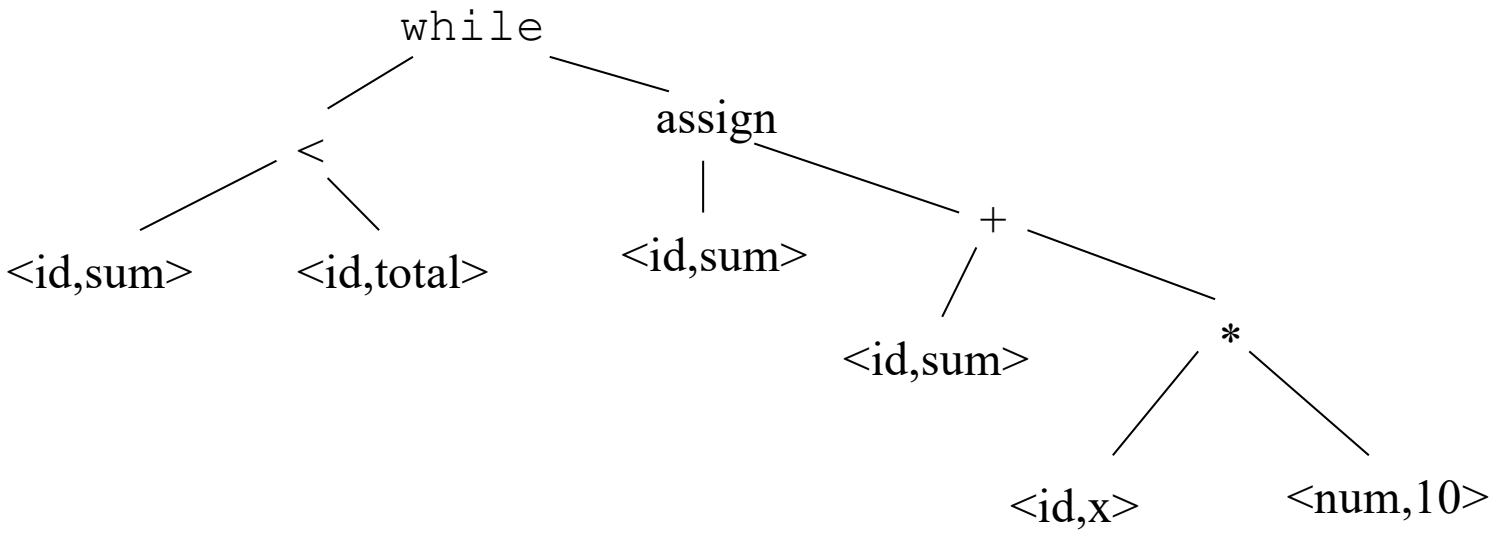
Syntax Analysis (Parsing)



Intermediate Representations

- The parse tree representation has too many details
  - `LPAREN, LBRACE, SEMICOL,` etc.
- Once the compiler understands the structure of the input program, it does not need these details (they prevent ambiguities during parsing)
- Compilers generate a more abstract representation after constructing the parse tree, which does not include the details of the derivation
- Abstract syntax trees (AST): Nodes represent operators, children represent operands

Intermediate Representations



Semantic (Context-Sensitive) Analysis

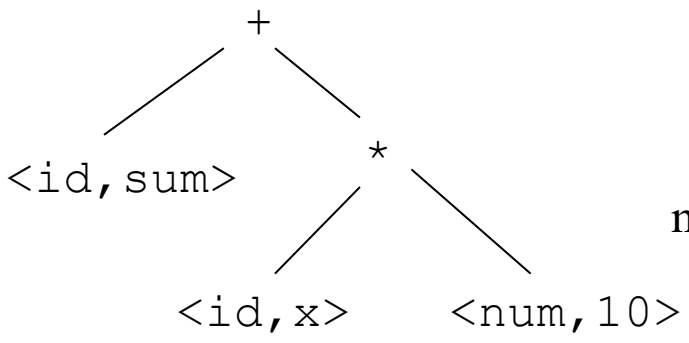
- Not everything that we care about is related to the *structure (i.e., syntax)* of the program, in some cases we have to check the meaning (i.e., semantics)

- Do variable types match?

```
sum = sum + x*10;
```

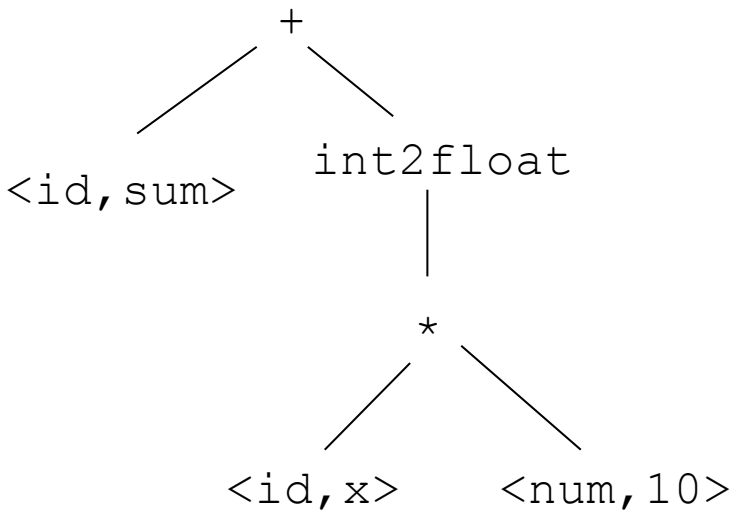
- Are variables declared before they are used?
  - We can find out if "`while`" is declared by looking at the symbol table

Semantic (Context-Sensitive) Analysis



`sum` can be a floating point number,
`x` can be an integer

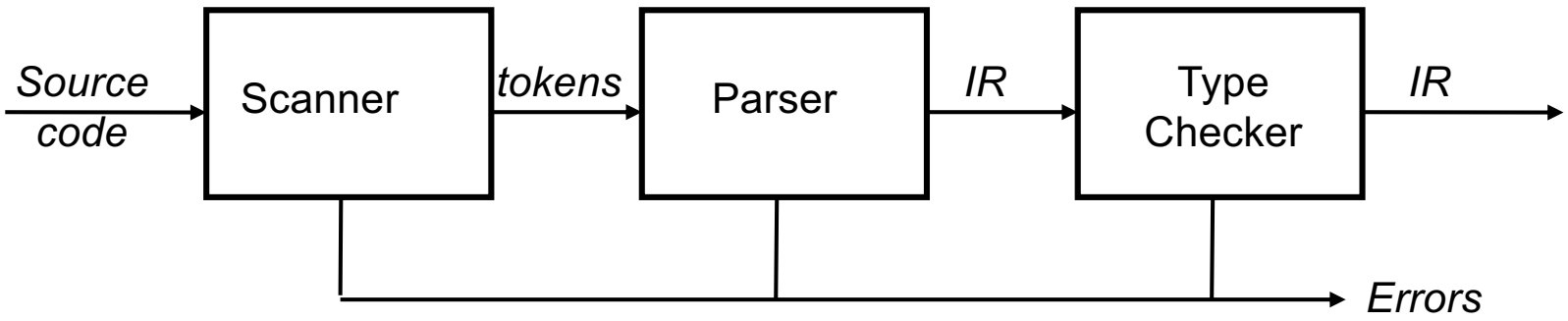
may become



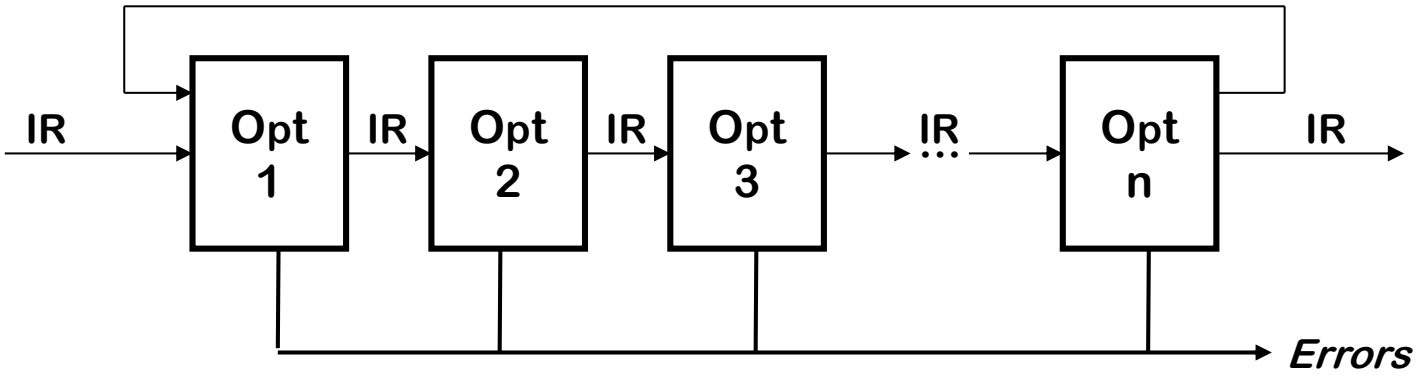
Symbol Table

sum	float
x	int

Summary of The Front End



The Optimizer (or Middle End)



Modern optimizers are structured as a series of passes

Typical Transformations

- Discover and propagate constant values (constant propagation)
- Move a computation to a less frequently executed place
- Discover a redundant computation and remove it
- Remove unreachable code

Improving the Code: Code Optimization

- Compilers can improve the quality of code by static analysis
  - Data flow analysis, dependence analysis, code transformations, dead code elimination, etc.

```
while (sum < total)
{
   sum = sum + x*10;
}
```

transformation to more efficient code

We do not need to recompute `x*10` in each iteration of the loop

```
temp = x*10;
while (sum < total)
{
   sum = sum + temp;
}
```

Code Generation - The Back End

- Abstract syntax trees are a high-level intermediate representation used in earlier phases of the compilation
- There are lower-level (i.e., closer to the machine code) intermediate representations
  - Three–address code: Every instruction has at most three operands. Very close to (MIPS, x86) assembly
  - Stack based code: Assembly language for JVM (Java Virtual Machine), an abstract stack machine.
- Intermediate code generation for these lower-level representations and machine code generation are similar

Code Generation: Instruction Selection

- Source code

```
a = b + c;
```

- Target code

```
loadAI RO, @b => R1
loadAI R0, @c => R2
add  R1, R2 => R1
storeAI  R1 => R0, @a
```

We need to map statements in the source code to instructions of the target language

Code Generation: Register Allocation

- There are a limited number of registers available on real machines
- Registers are valuable resources (keeping the values in registers prevents memory access), the compiler has to use them efficiently

source code

```
d = (a-b)+(a-c)+(a-c)+(a-c);
```

three-address code

```
t1 = a - b;
t2 = a - c;
t3 = t1 + t2;
t4 = t3 + t2;
t5 = t4 + t2;
d = t5;
```

assembly code

```
loadAI R0, @a => R1
loadAI R0, @b => R2
sub R1, R2 => R2
loadAI R0, @c => R3
sub R1, R3 => R1
add R1, R2 => R2
add R1, R2 => R2
add R1, R2 => R2
storeAI R2 => R0, @d
```

Code Generation: Optimization

- Source code

```
a = b + c;
d = a + e;
```

- Target code

If we generate code for each statement separately then we will not generate efficient code

code for the first statement

```
loadAI RO, @b => R1
loadAI R0, @c => R2
add  R1, R2 => R1
storeAI  R1 => R0, @a
```

code for the second statement

```
loadAI R0, @a => R1
loadAI R0, @e => R2
add  R1, R2 => R1
storeAI R1 => R0, @d
```


This instruction is redundant

Why Study Compilers (even if you may never need to write one)?

- Compiler construction involves several areas of computer science (theory, algorithms, systems, architecture) and demonstrates the application of techniques from these areas and the interplay among them

- By learning compilers you get an understanding of both the upper language designs and lower hardware architecture.

- Is compiler construction a solved problem?
  - No! New developments in programming languages and machine architectures present new challenges and opportunities
  - ML compiler + Quantum compiler