

# Computer Science 160

## Translation of Programming Languages

Instructor: Christopher Kruegel  
(based on material by Tim Sherwood)

---

# Overview of Compilers

---

# Compilers

---

UC Santa Barbara

- A) Why do we need a compiler?**
  - B) What steps do we need to take to realize a compiler?
  - C) How is a compiler put together?
-

# Why Take This Class?

UC Santa Barbara

---

- Compilers are a testament to the power of computer science
    - Theory, algorithms, systems, architecture... all these things you practice in other classes are **applied** to compilers!
  - Bridge a huge mental gap between the software you know how to write and the hardware you know how to build
  - The techniques you learn in this class are applicable to many real world problems you may face “on the outside”
    - Input and output parsing (XML)
    - Application specific languages (configuration files)
    - Program analysis and understanding
-

# Compilers

UC Santa Barbara

---

- What is a compiler?
    - A program that translates a program in one language (source language) into an equivalent program in another language (target language), and it reports errors in the source program
  - A compiler typically lowers the level of abstraction of the program
    - C -> assembly code for Intel x86
    - Java -> Java bytecode
  - What is an interpreter?
    - A program that reads an executable program (one instruction at a time) and produces the results of executing these instructions
  - C is typically compiled
  - Script languages (Python, Javascript) are typically interpreted
  - Java is compiled to bytecode, which is then interpreted
-

# Why Build Compilers?

UC Santa Barbara

---

- Compilers provide an essential interface between applications and architectures
  - High level programming languages:
    - Increase programmer productivity
    - Better maintenance
    - Portable
  - Low level machine details:
    - Instruction selection
    - Addressing modes
    - Pipelines
    - Registers and cache
  - Compilers efficiently bridge the gap and shield the application developers from low level machine details
-

# Desirable Properties of Compilers

UC Santa Barbara

---

- Compiler must generate a correct executable
    - The input program and the output program must be equivalent, the compiler should preserve the meaning (semantics) of the input program
  - Output program should run fast
    - We expect the output program to be more efficient than the input program
  - Compiler itself should be fast
  - Compiler should provide good diagnostics for programming errors
  - Compiler should support separate compilation (modules, object files)
  - Compiler should work well with debuggers
  - Compiled code should be small
  - Optimizations should be consistent and predictable
  - Compile time should be proportional to code size
-

# Compiler - Example

UC Santa Barbara

- 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: MOV     total,R0
    CMP     sum,R0
    JL      L2
    GOTO    L3
```

```
L2: MOV     #10,R0
    MUL    x,R0
    ADD    sum,R0
    MOV    R0,sum
    GOTO   L1
```

```
L3: first instruction
    following the while
    statement
```

# Compilers

---

UC Santa Barbara

- A) Why do we need a compiler?
  - B) What steps do we need to take to realize a compiler?**
  - C) How is a compiler put together?
-

# What is the Input?

UC Santa Barbara

- 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=\n\tsum\b+\bx*10;\n}\n
```

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

# First Step: Lexical Analysis (Scanning)

UC Santa Barbara

- The compiler scans the input file and produces a stream of **tokens**

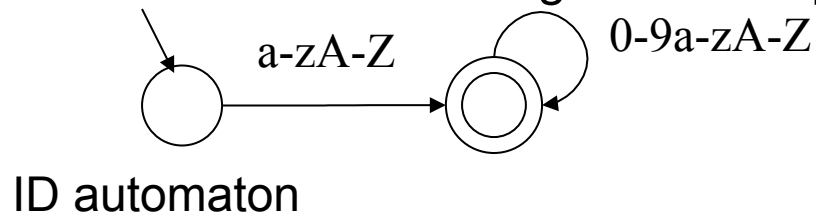
```
WHILE, LPAREN, <ID, sum>, LT, <ID, total>, RPAREN, LBRACE,  
<ID, sum>, EQ, <ID, sum>, PLUS, <ID, x>, TIMES, <NUM, 10>,  
SEMICOL, RBRACE
```

- Each token has a corresponding **lexeme**, 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)

UC Santa Barbara

- Compiler uses a set of patterns to specify valid tokens
  - tokens: LPAREN, ID, NUM, WHILE, 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



# Lexical Analysis (Scanning)

UC Santa Barbara

---

- 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
-

# Next Step: Syntax Analysis (Parsing)

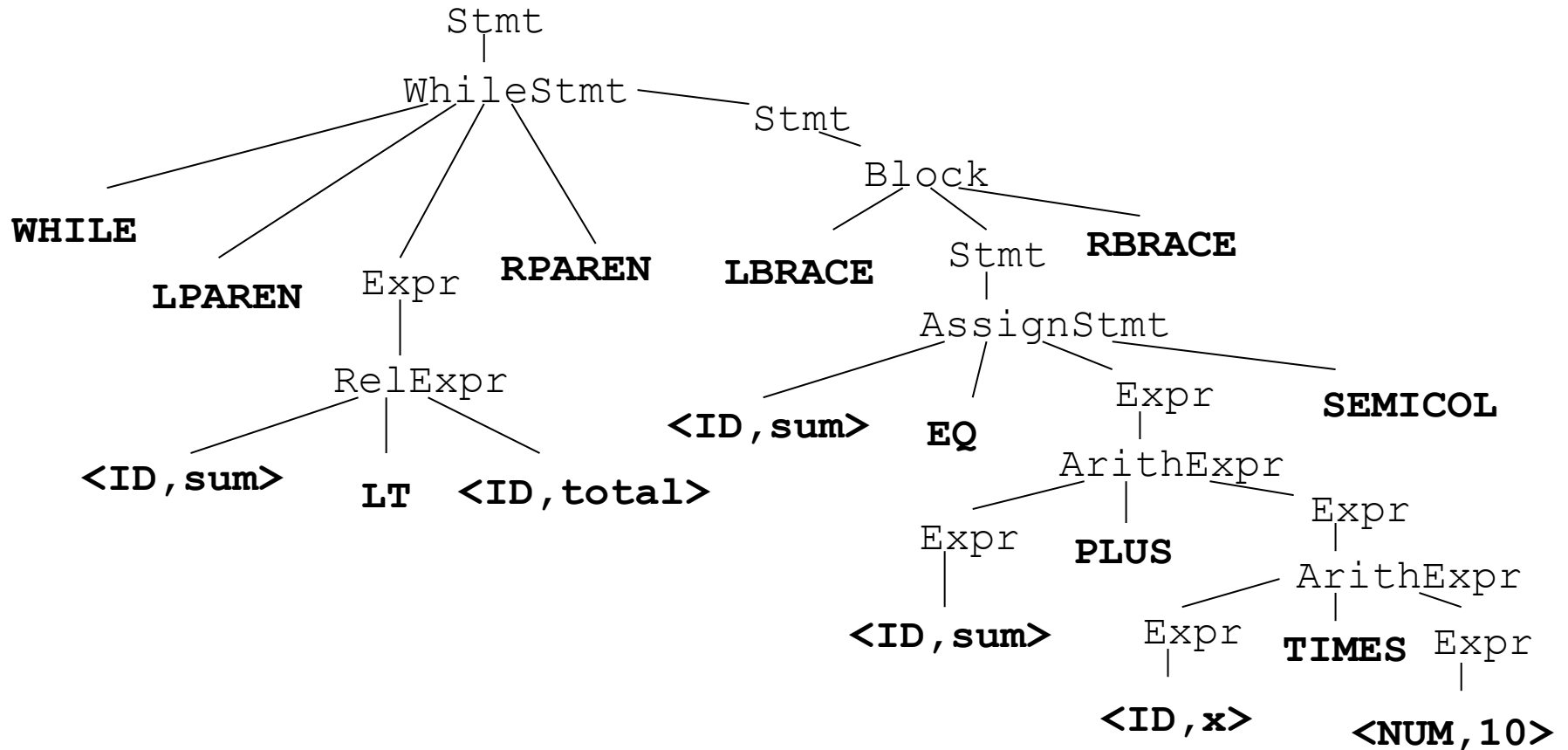
UC Santa Barbara

---

- How does the compiler recognize the structure of the program?
    - Loops, blocks, procedures, nesting?
  - Parse the stream of tokens -> parse tree
    - program will be on the leaves of the tree
-

# Syntax Analysis (Parsing)

UC Santa Barbara



# Syntax Analysis (Parsing)

UC Santa Barbara

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

`Stmt`  $\rightarrow$  `WhileStmt` | `Block` | ...

`WhileStmt`  $\rightarrow$  `WHILE LPAREN Expr RPAREN Stmt`

`Expr`  $\rightarrow$  `RelExpr` | `ArithExpr` | ...

`RelExpr`  $\rightarrow$  ...

- 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.

# Intermediate Representations

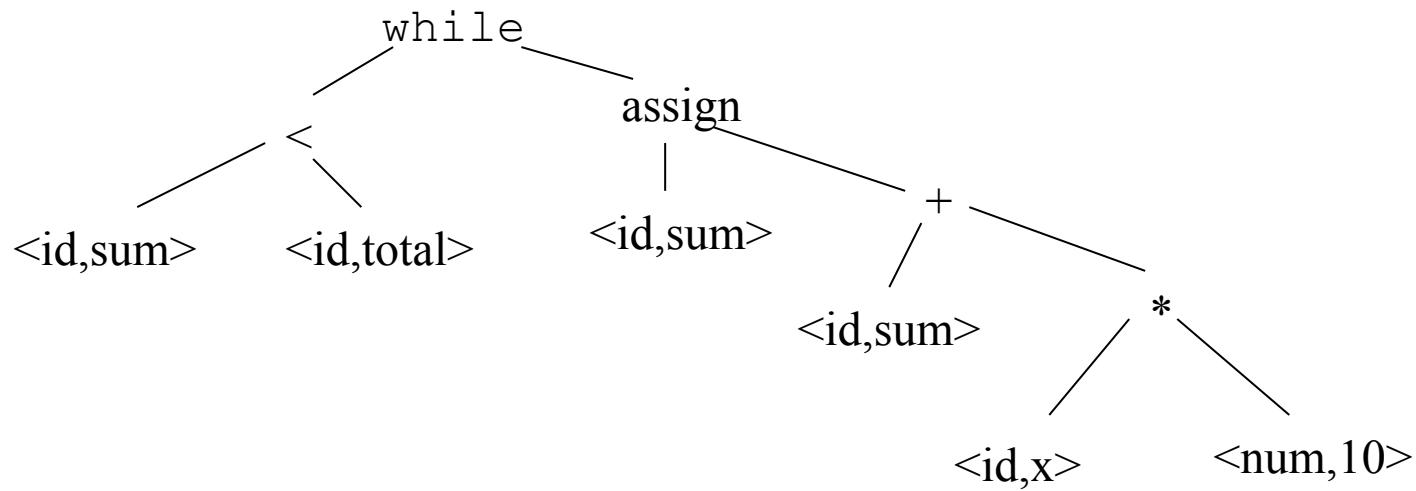
UC Santa Barbara

---

- 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

UC Santa Barbara



# Semantic (Context-Sensitive) Analysis

UC Santa Barbara

---

- Not everything that we care about is related to the *structure* of the program, in some cases we have to check the meaning (or semantics)
- Are variables declared before they are used?
  - We can find out if “`while`” is declared by looking at the symbol table
- Do variable types match?

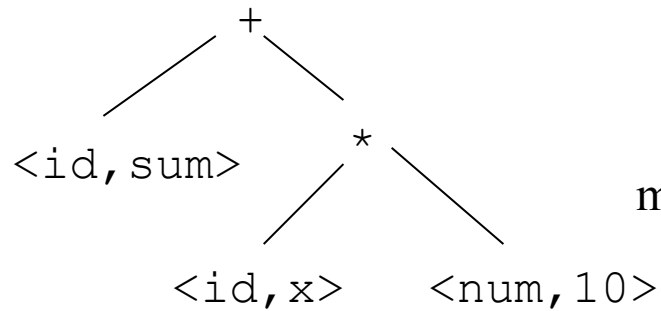
```
sum = sum + x*10;
```

---

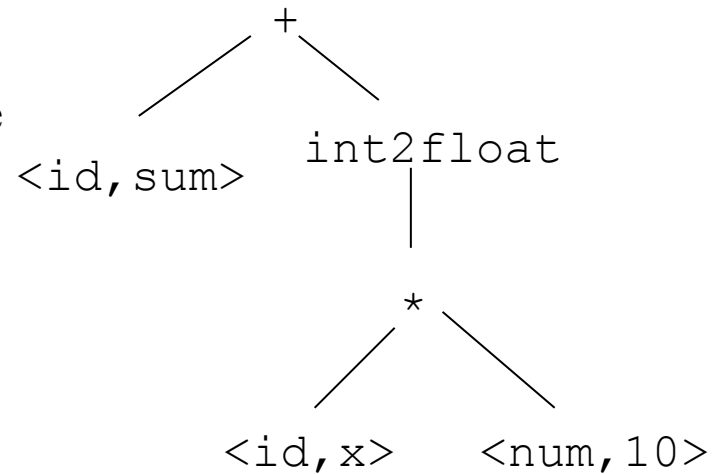
# Semantic (Context-Sensitive) Analysis

UC Santa Barbara

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



may become



Symbol  
Table

|     |       |
|-----|-------|
| sum | float |
| x   | int   |
|     |       |

# Runtime Environment

UC Santa Barbara

---

- Efficient implementation of programming language abstractions
    - Symbolic names
    - Name spaces
    - Procedures
    - Parameters
    - Control Flow
  - Bridge the gap between useful idea and practical application
-

# Code Generation

UC Santa Barbara

---

- 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 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
-

# Improving the Code: Code Optimization

UC Santa Barbara

- 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;
}
```

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

transformation  
to more efficient  
code

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

# Code Generation: Instruction Selection

UC Santa Barbara

- Source code

a = b + c;

d = a + e;

- Target code

code for  
the first  
statement

{  
MOV b,R0  
ADD c,R0  
MOV R0,a

code for  
the second  
statement

{  
MOV a,R0  
ADD e,R0  
MOV R0,d

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

← This instruction is redundant

# Code Generation: Register Allocation

UC Santa Barbara

- 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              | three-address code   | assembly code  |
|--------------------------|--|--|
| $d = (a-b)+(a-c)+(a-c);$ | $t = a - b;$<br>$u = a - c;$<br>$v = t + u;$<br>$d = v + u;$ | MOV a,R0<br>SUB b,R0<br>MOV a,R1<br>SUB c,R1<br>ADD R1,R0<br>ADD R1,R0<br>MOV R0,d |

# Compilers

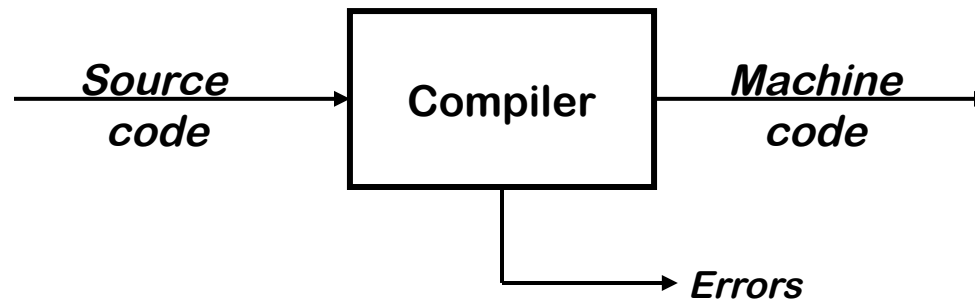
---

UC Santa Barbara

- A) Why do we need a compiler?
  - B) What Steps do we need to we need to take to realize a compiler?
  - C) How is a compiler put together?**
-

# High-level View of a Compiler

UC Santa Barbara

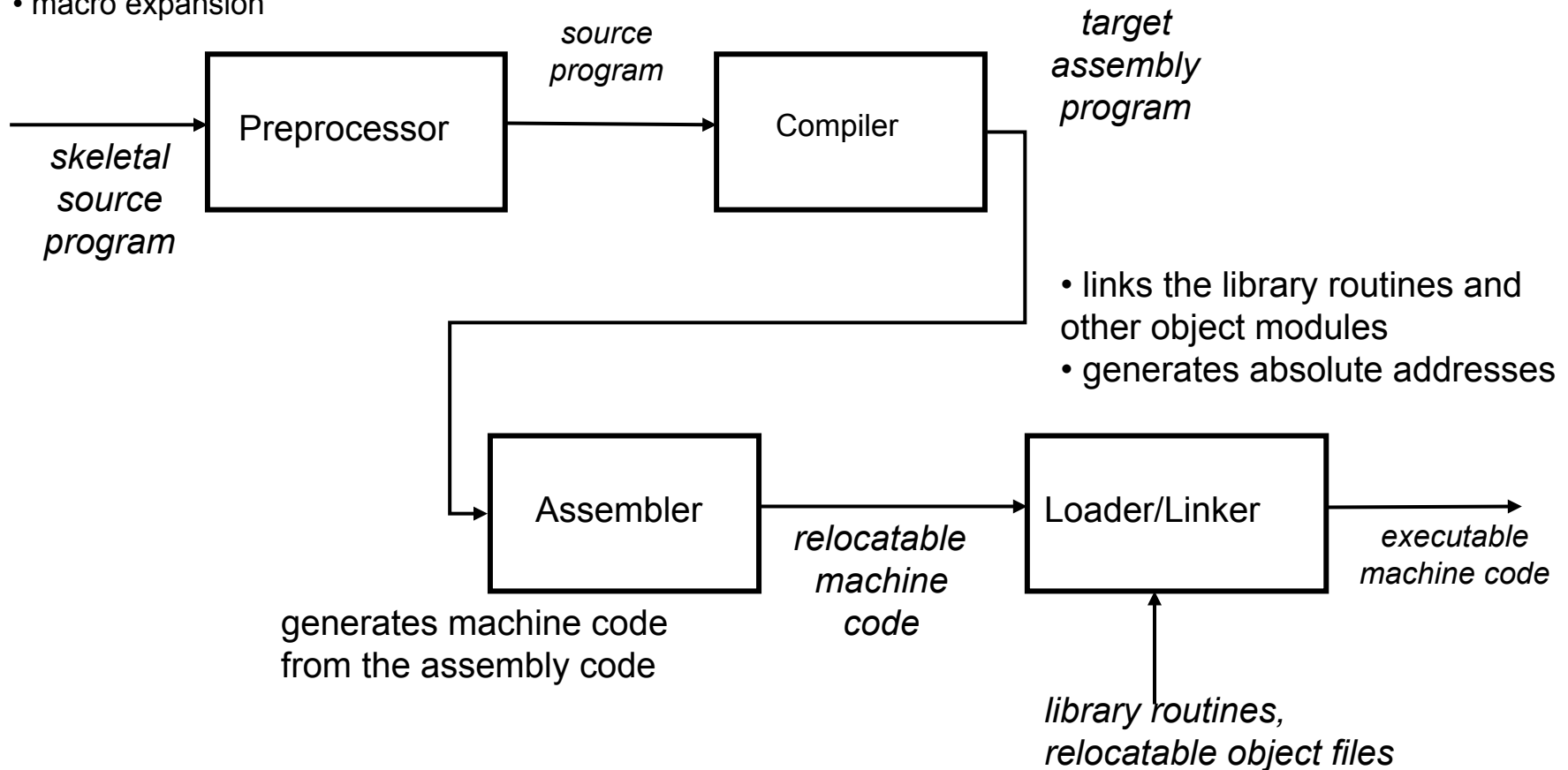


- Must recognize legal (and illegal) programs
- Must generate correct code
- Must manage storage of all variables (and code)
- Must agree with OS and linker on format for object code

# A Higher Level View: How Does the Compiler Fit In?

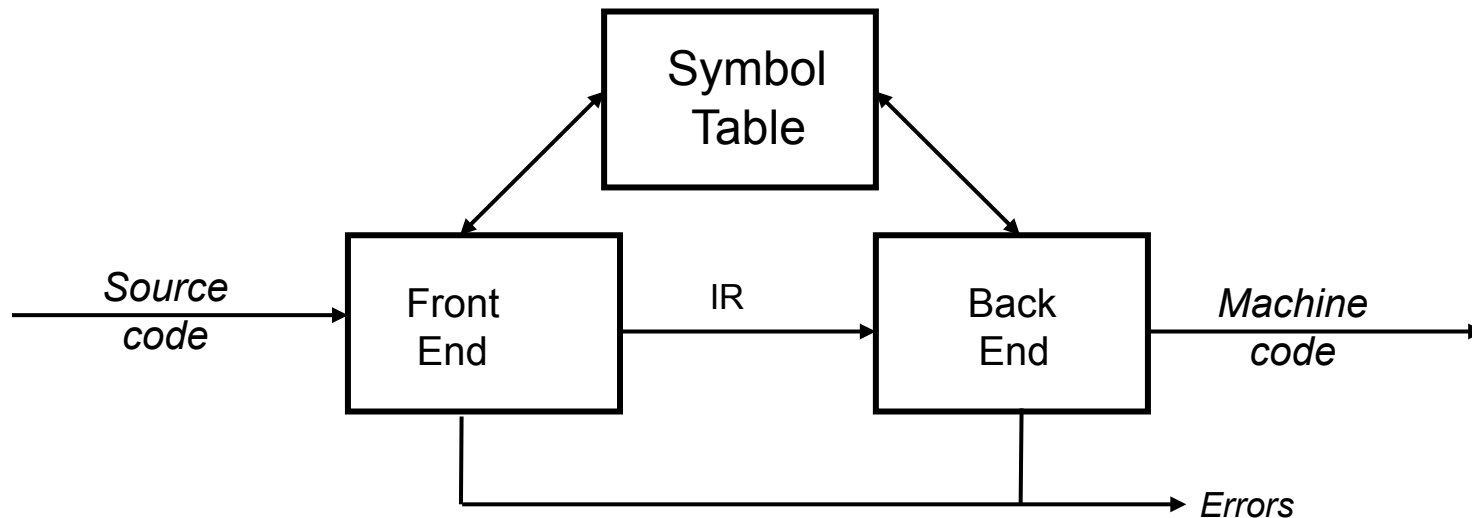
UC Santa Barbara

- collects the source program that is divided into separate files
- macro expansion



# Traditional Two-pass Compiler

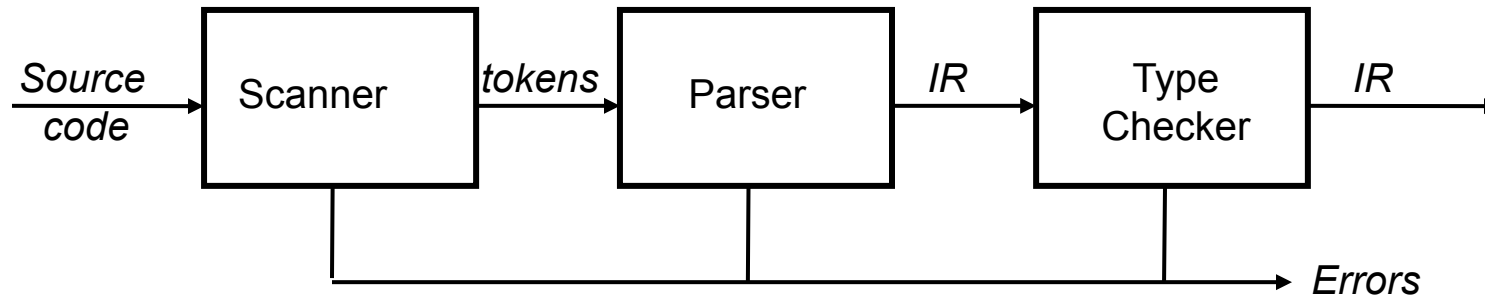
UC Santa Barbara



- 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 and multiple passes
  - Typically, front end is  $O(n)$  or  $O(n \log n)$ , back end is NP-complete
- Different phases of compiler also interact through the symbol table

# The Front End

UC Santa Barbara

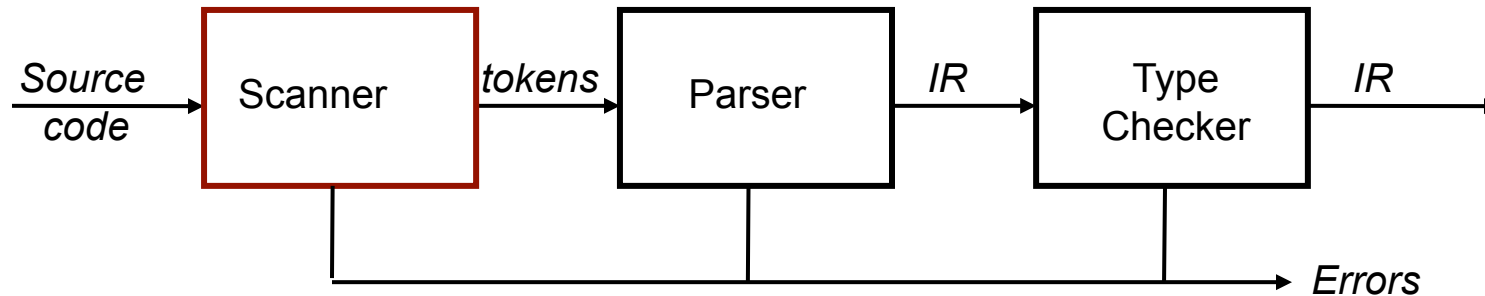


## Responsibilities

- Recognize legal programs
- Report errors for the illegal programs in a useful way
- Produce IR and construct the symbol table
- Much of front end construction can be automated

# The Front End

UC Santa Barbara

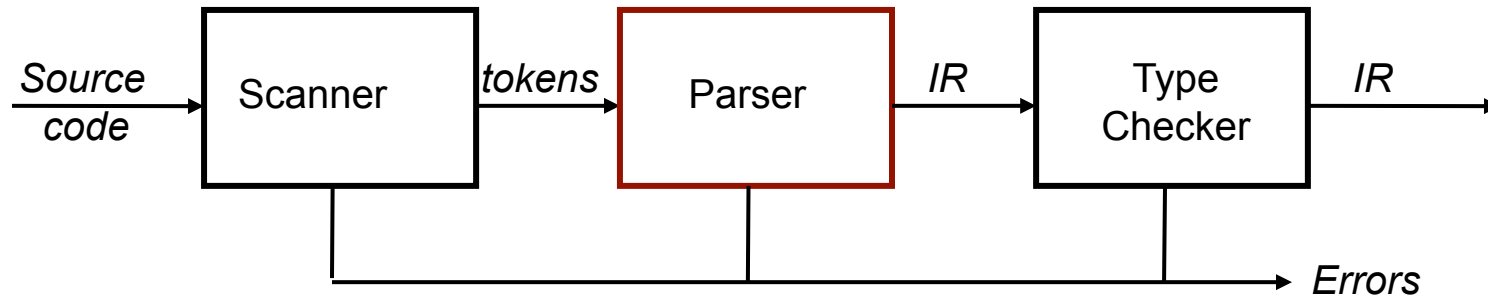


## Scanner

- Maps character stream into words—the basic unit of syntax
- Produces tokens and stores lexemes when it is necessary
  - $x = x + y ;$  becomes  
 $\langle \text{id}, x \rangle \text{ EQ } \langle \text{id}, x \rangle \text{ PLUS } \langle \text{id}, y \rangle \text{ SEMICOLON}$
  - Typical tokens include *number*, *identifier*,  $+$ ,  $-$ , *while*, *if*
- Scanner eliminates white space and comments

# The Front End

UC Santa Barbara

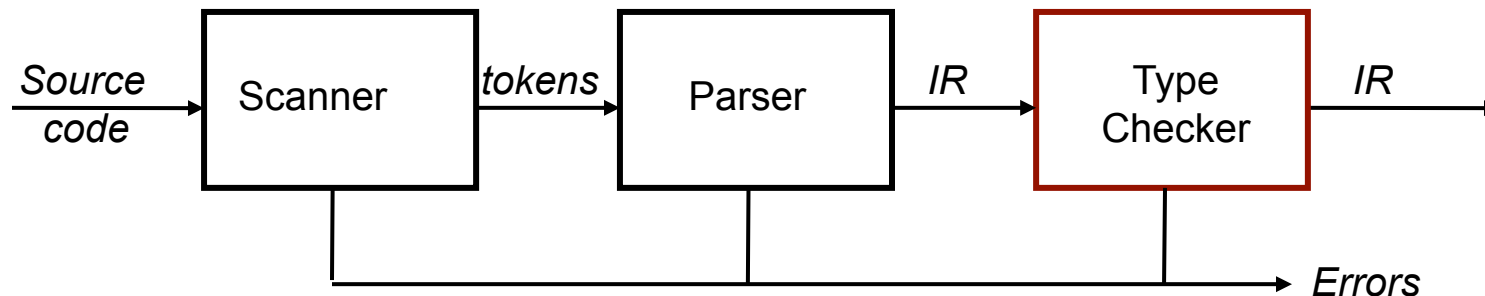


## Parser

- Uses scanner as a subroutine
- Recognizes context-free syntax and reports errors
- Guides context-sensitive analysis (type checking)
- Builds IR for source program
- Scanning and parsing can be grouped into one pass

# The Front End

UC Santa Barbara

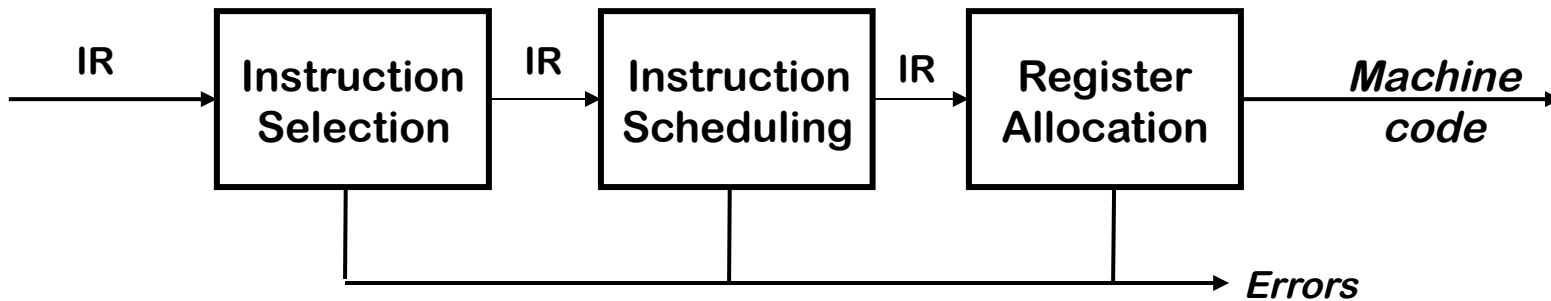


## Context Sensitive Analysis

- Check if all the variables are declared before they are used
- Type checking
  - Check type errors such as adding a procedure and an array
- Add the necessary type conversions
  - int-to-float, float-to-double, etc.

# The Back End

UC Santa Barbara



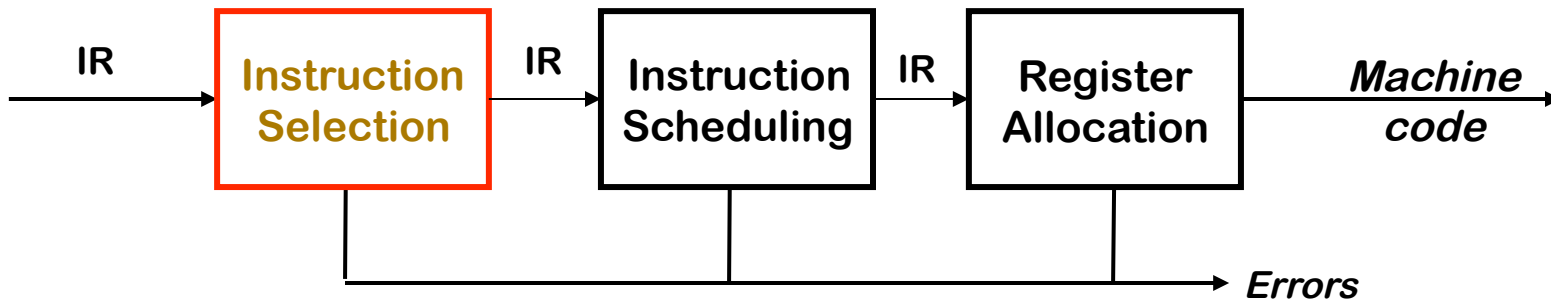
## Responsibilities

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which values to keep in registers
- Schedule the instructions for instruction pipeline

Automation has been *much less* successful in the back end

# The Back End

UC Santa Barbara

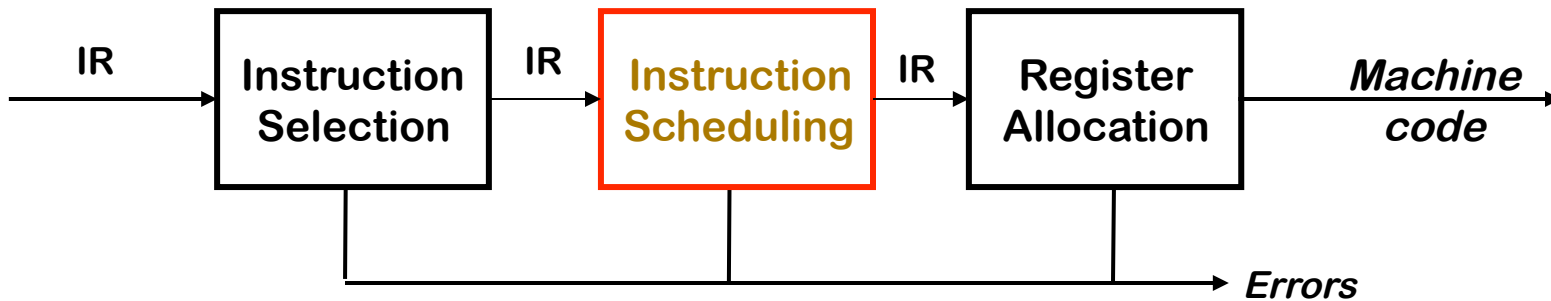


## Instruction Selection

- Produce fast, compact code
- Take advantage of target language features
  - E.g., addressing modes
- Usually viewed as a pattern matching problem
  - *Ad hoc* methods, pattern matching, dynamic programming
- Especially problematic when instruction sets are complex
  - RISC architectures simplified this problem

# The Back End

UC Santa Barbara

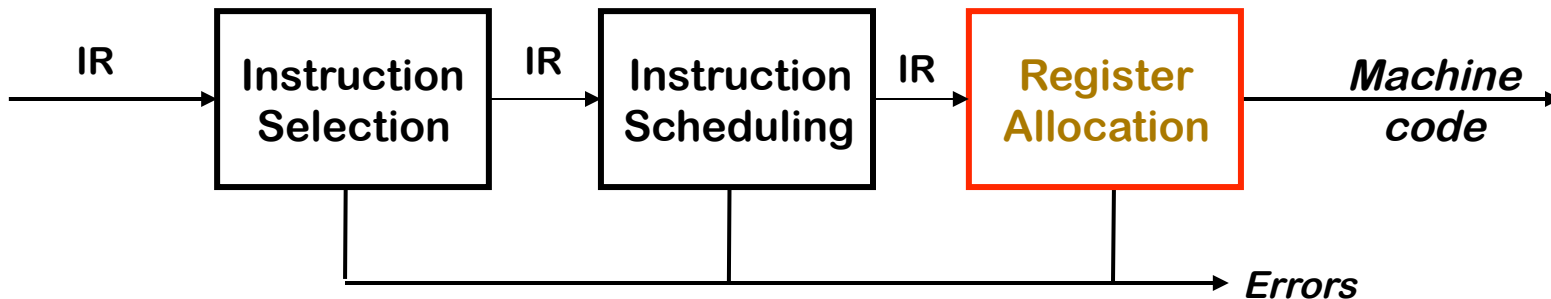


## Instruction Scheduling

- Avoid hardware stalls (keep pipeline moving)
- Use all functional units productively
- Optimal scheduling is NP-Complete

# The Back End

UC Santa Barbara



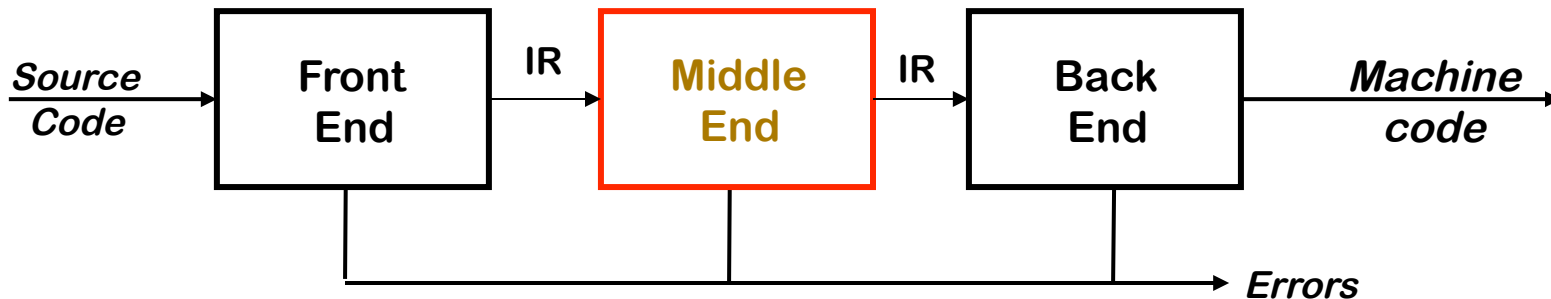
## Register Allocation

- Have each value in a register when it is used
- Manage a limited set of registers
- Can change instruction choices and insert LOADs and STOREs
- Optimal allocation is NP-Complete

Compilers approximate solutions to NP-Complete problems

# Traditional Three-pass Compiler

UC Santa Barbara

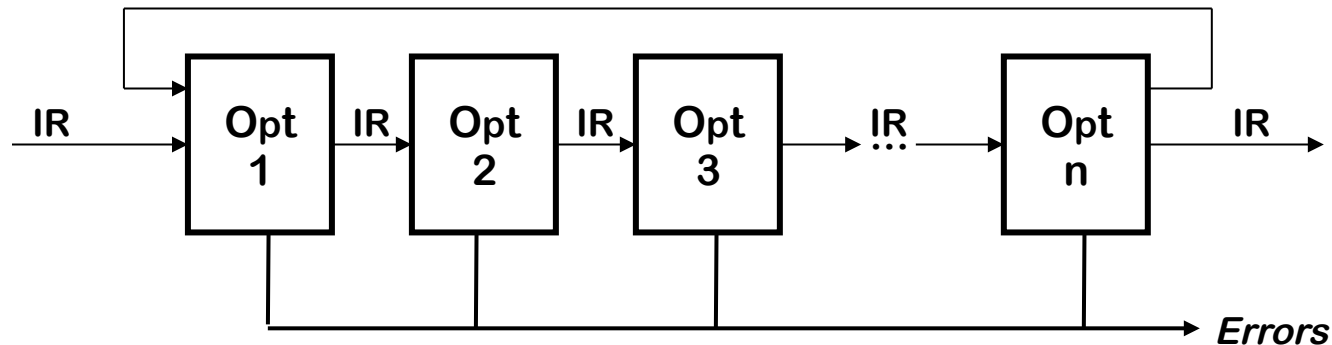


## Code Optimization

- Analyzes IR and transforms IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption (mobile computing)
- Must preserve “meaning” of the code

# The Optimizer (or Middle End)

UC Santa Barbara



*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