Language Specification and Translation

ICOM 4036
Fall 2005
Lecture 3
Language Specification and Translation Topics

- Structure of a Compiler
- Lexical Specification and Scanning
- Syntactic Specification and Parsing
- Semantic Specification and Analysis
Syntax versus Semantics

- **Syntax** - the form or *structure* of the expressions, statements, and program units
- **Semantics** - the *meaning* of the expressions, statements, and program units
The Structure of a Compiler

1. Lexical Analysis
2. Parsing
3. Semantic Analysis
4. Optimization
5. Code Generation

The first 3, at least, can be understood by analogy to how humans comprehend English.
A Prototypical Compiler

1. **Scanner**
2. **Parser**
3. **Semantic Analysis**
4. **Optimizer**
5. **Code Generator**
6. **exe**
Introduction

• Reasons to separate compiler in phases:
  – **Simplicity** - less complex approaches can be used for lexical analysis; separating them simplifies the parser
  – **Efficiency** - separation allows optimization of the lexical analyzer
  – **Portability** - parts of the lexical analyzer may not be portable, but the parser always is portable
Lexical Analysis

• First step: recognize words.
  – Smallest unit above letters

This is a sentence.

• Note the
  – Capital “T” (start of sentence symbol)
  – Blank “ “ (word separator)
  – Period “.” (end of sentence symbol)
Lexical Analysis

- Lexical analysis is not trivial. Consider:
  ist his ase nte nce

- Plus, programming languages are typically more cryptic than English:
  \[ *p->f++ = -1.2345e-5 \]
Lexical Analysis

• Lexical analyzer divides program text into “words” or “tokens”

  if x == y then z = 1; else z = 2;

• Units:

  if, x, ==, y, then, z, =, 1, ;, else, z, =, 2, ;
Lexical Analysis

• A lexical analyzer is a pattern matcher for character strings
• A lexical analyzer is a “front-end” for the parser
• Identifies substrings of the source program that belong together - lexemes
  – Lexemes match a character pattern, which is associated with a lexical category called a token
  – sum is a lexeme; its token may be IDENT
Lexical Analysis

• The lexical analyzer is usually a function that is called by the parser when it needs the next token
• Three approaches to building a lexical analyzer:
  – Write a formal description of the tokens and use a software tool that constructs table-driven lexical analyzers given such a description (e.g. lex)
  – Design a state diagram that describes the tokens and write a program that implements the state diagram
  – Design a state diagram that describes the tokens and hand-construct a table-driven implementation of the state diagram
• We only discuss approach 2

State diagram = Finite State Machine
Pascal Scanner Finite State Diagram
Lexical Analysis

• State diagram design:
  – A naïve state diagram would have a transition from every state on every character in the source language - such a diagram would be very large!

• In many cases, transitions can be combined to simplify the state diagram
  – When recognizing an identifier, all uppercase and lowercase letters are equivalent
    • Use a character class that includes all letters
  – When recognizing an integer literal, all digits are equivalent - use a digit class
Lexical Analysis

- Reserved words and identifiers can be recognized together (rather than having a part of the diagram for each reserved word)
  - Use a table lookup to determine whether a possible identifier is in fact a reserved word
Lexical Analysis

• Convenient utility subprograms:
  – `getChar` - gets the next character of input, puts it in `nextChar`, determines its class and puts the class in `charClass`
  – `addChar` - puts the character from `nextChar` into the place the lexeme is being accumulated, `lexeme`
  – `lookup` - determines whether the string in `lexeme` is a reserved word (returns a code)
**State Diagram**

- **Start**: addChar; getChar
- **Letter**
  - addChar; getChar
  - **id**: return lookup (lexeme)
- **Digit**
  - addChar; getChar
  - **int**: return Int_Lit
- **Letter/Digit**
  - addChar; getChar
Lexical Analysis

Implementation (assume initialization):

```c
int lex() {
    getChar();
    switch (charClass) {
        case LETTER:
            addChar();
            getChar();
            while (charClass == LETTER || charClass == DIGIT)
            {
                addChar();
                getChar();
            }
            return lookup(lexeme);
            break;
        case DIGIT:
            addChar();
            getChar();
            while (charClass == DIGIT) {
                addChar();
                getChar();
            }
            return INT_LIT;
            break;
    } /* End of switch */
} /* End of function lex */
```
Parsing

- Once words are understood, the next step is to understand sentence structure

- Parsing = Diagramming Sentences
  - The diagram is a tree
Diagramming a Sentence

This line is a longer sentence

article | noun | verb | article | adjective | noun

subject | object

sentence
Parsing Programs

- Parsing program expressions is the same
- Consider:
  
  \[
  \text{If } x \equiv y \text{ then } z = 1; \text{ else } z = 2;
  \]

- Diagrammed:
Describing Syntax

• A **sentence** is a string of characters over some alphabet
• A **language** is a set of sentences
• A **lexeme** is the lowest level syntactic unit of a language (e.g., *, sum, begin)
• A **token** is a category of lexemes (e.g., identifier)
Describing Syntax

• Formal approaches to describing syntax:
  – **Recognizers** - used in compilers (we will look at in Chapter 4)
  – **Generators** – generate the sentences of a language (what we'll study in this chapter)
Formal Methods of Describing Syntax

• Context-Free Grammars
  – Developed by Noam Chomsky in the mid-1950s
  – Language generators, meant to describe the syntax of natural languages
  – Define a class of languages called context-free languages
Formal Methods of Describing Syntax

• Backus-Naur Form (1959)
  – Invented by John Backus to describe Algol 58
  – BNF is equivalent to context-free grammars
  – A metalanguage is a language used to describe another language.
  – In BNF, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (also called nonterminal symbols)
Backus-Naur Form (1959)

```<while_stmt> → while (<logic_expr>) <stmt>```

- This is a rule; it describes the structure of a while statement
Formal Methods of Describing Syntax

- A rule has a left-hand side (LHS) and a right-hand side (RHS), and consists of terminal and nonterminal symbols.
- A grammar is a finite nonempty set of rules.
- An abstraction (or nonterminal symbol) can have more than one RHS.

\[
<\text{stmt}> \rightarrow <\text{single_stmt}> \\
\quad | \begin{align*}
& \text{begin } <\text{stmt_list}> \text{ end}
\end{align*}
\]
Formal Methods of Describing Syntax

- Syntactic lists are described using recursion
  
  <ident_list> → ident
  | ident, <ident_list>

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)
Formal Methods of Describing Syntax

• An example grammar:

\[
\begin{align*}
<\text{program}> & \rightarrow <\text{stmts}> \\
<\text{stmts}> & \rightarrow <\text{stmt}> \mid <\text{stmt}> \ ; <\text{stmts}> \\
<\text{stmt}> & \rightarrow <\var> = <\text{expr}> \\
<\var> & \rightarrow a \mid b \mid c \mid d \\
<\text{expr}> & \rightarrow <\text{term}> + <\text{term}> \mid <\text{term}> - <\text{term}> \\
<\text{term}> & \rightarrow <\var> \mid \text{const}
\end{align*}
\]
Formal Methods of Describing Syntax

• An example derivation:

\[ \text{<program>} \Rightarrow \text{<stmts>} \Rightarrow \text{<stmt>}
\]
\[ \Rightarrow \text{<var>} = \text{<expr>} \Rightarrow a = \text{<expr>}
\]
\[ \Rightarrow a = \text{<term>} + \text{<term>}
\]
\[ \Rightarrow a = \text{<var>} + \text{<term>}
\]
\[ \Rightarrow a = b + \text{<term>}
\]
\[ \Rightarrow a = b + \text{const}
\]
Derivation

• Every string of symbols in the derivation is a sentential form
• A sentence is a sentential form that has only terminal symbols
• A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded
• A derivation may be neither leftmost nor rightmost
Parse Tree

• A hierarchical representation of a derivation

```
<program>
  |
<stmts>
  |
<stmt>
  |   |
<var> = <expr>
  |   |   |
a <term> + <term>
  |   |   |
<var> const
  |   |
b
```
Formal Methods of Describing Syntax

• A grammar is ambiguous iff it generates a sentential form that has two or more distinct parse trees
An Ambiguous Expression Grammar

<expr> → <expr> <op> <expr> | const
<op> → / | -

const - const / const

const - const / const
An Unambiguous Expression Grammar

• If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

<expr> → <expr> - <term> | <term>
<term> → <term> / const | const

```
<expr>  
/       
<expr>  -  <term>  
|      /     
<term>  <term> / const  
|  |         
const  const
```
Formal Methods of Describing Syntax

Derivation:

\[ \langle expr \rangle \Rightarrow \langle expr \rangle - \langle term \rangle \Rightarrow \langle term \rangle - \langle term \rangle \]
\[ \Rightarrow \text{const} - \langle term \rangle \]
\[ \Rightarrow \text{const} - \langle term \rangle / \text{const} \]
\[ \Rightarrow \text{const} - \text{const} / \text{const} \]
Formal Methods of Describing Syntax

- Operator associativity can also be indicated by a grammar

\[
\text{<expr>} \rightarrow \text{<expr>} + \text{<expr>} \mid \text{const} \quad \text{(ambiguous)}
\]

\[
\text{<expr>} \rightarrow \text{<expr>} + \text{const} \mid \text{const} \quad \text{(unambiguous)}
\]
Formal Methods of Describing Syntax

- Extended BNF (just abbreviations):
  - Optional parts are placed in brackets ([ ])

  \[<\text{proc\_call}> \rightarrow \text{ident} [ ( <\text{expr\_list}>)]\]

  - Put alternative parts of RHSs in parentheses and separate them with vertical bars

  \[<\text{term}> \rightarrow <\text{term}> (+ | -) \text{const}\]

  - Put repetitions (0 or more) in braces ({})

  \[<\text{ident}> \rightarrow \text{letter} \{\text{letter} | \text{digit}\}\]
BNF and EBNF

- **BNF:**
  
  \[
  \begin{align*}
  <expr> & \rightarrow <expr> + <term> \\
  & \quad | <expr> - <term> \\
  & \quad | <term> \\
  <term> & \rightarrow <term> * <factor> \\
  & \quad | <term> / <factor> \\
  & \quad | <factor>
  \end{align*}
  \]

- **EBNF:**
  
  \[
  \begin{align*}
  <expr> & \rightarrow <term> \{(+ \mid -) <term>\} \\
  <term> & \rightarrow <factor> \{(* \mid /) <factor>\}
  \end{align*}
  \]
The Parsing Problem

• Goals of the parser, given an input program:
  – Find all syntax errors; for each, produce an appropriate diagnostic message, and recover quickly
  – Produce the parse tree, or at least a trace of the parse tree, for the program
The Parsing Problem

• Two categories of parsers
  – Top down - produce the parse tree, beginning at the root
    • Order is that of a leftmost derivation
  – Bottom up - produce the parse tree, beginning at the leaves
    • Order is that of the reverse of a rightmost derivation

• Parsers look only one token ahead in the input
The Parsing Problem

• Top-down Parsers
  – Given a sentential form, $x A \alpha$, the parser must choose the correct $A$-rule to get the next sentential form in the leftmost derivation, using only the first token produced by $A$

• The most common top-down parsing algorithms:
  – Recursive descent - a coded implementation
  – LL parsers - table driven implementation
The Parsing Problem

• Bottom-up parsers
  – Given a right sentential form, \( \alpha \), determine what substring of \( \alpha \) is the right-hand side of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation
  – The most common bottom-up parsing algorithms are in the LR family
The Parsing Problem

• The Complexity of Parsing
  – Parsers that work for any unambiguous grammar are complex and inefficient (\(O(n^3)\), where \(n\) is the length of the input)
  – Compilers use parsers that only work for a subset of all unambiguous grammars, but do it in linear time (\(O(n)\), where \(n\) is the length of the input)
Recursive-Descent Parsing

• Recursive Descent Process
  – There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal
  – EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of nonterminals
Recursive-Descent Parsing

• A grammar for simple expressions:

\[
\begin{align*}
<\text{expr}> & \rightarrow <\text{term}> \{(+ | -) <\text{term}>\} \\
<\text{term}> & \rightarrow <\text{factor}> \{(* | /) <\text{factor}>\} \\
<\text{factor}> & \rightarrow \text{id} \mid ( <\text{expr}> )
\end{align*}
\]
Recursive-Descent Parsing

• Assume we have a lexical analyzer named \texttt{lex}, which puts the next token code in \texttt{nextToken}

• The coding process when there is only one RHS:
  – For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an error
  – For each nonterminal symbol in the RHS, call its associated parsing subprogram
Recursive-Descent Parsing

/* Function expr
   Parses strings in the language
   generated by the rule:
   \[<expr> \rightarrow <term> \{(+ | -) <term>\}\]
   */

void expr() {
    /* Parse the first term */
    term();
    /* As long as the next token is + or -, call
    lex to get the next token, and parse the
    next term */
    while (nextToken == PLUS_CODE || nextToken == MINUS_CODE){
        lex();
        term();
    }
}

- This particular routine does not detect errors
- Convention: Every parsing routine leaves the next token in nextToken
Recursive-Descent Parsing

- A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse
  - The correct RHS is chosen on the basis of the next token of input (the lookahead)
  - The next token is compared with the first token that can be generated by each RHS until a match is found
  - If no match is found, it is a syntax error
Recursive-Descent Parsing

/* Function factor
   Parses strings in the language
   generated by the rule:
   <factor> -> id | (<expr>) */

void factor() {
    /* Determine which RHS */
    if (nextToken == ID_CODE)
        /* For the RHS id, just call lex */
        lex();
    /* If the RHS is (<expr>) - call lex to pass
     over the left parenthesis, call expr, and
     check for the right parenthesis */
    else if (nextToken == LEFT_PAREN_CODE) {
        lex();
        expr();
        if (nextToken == RIGHT_PAREN_CODE)
            lex();
        else
            error();
    } /* End of else if (nextToken == ... */
    else error(); /* Neither RHS matches */
}
Recursive-Descent Parsing

- Limitations of the LL grammar classes
  - The Left Recursion Problem
    - If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser
      - A grammar can be modified to remove left recursion
  - Lack of pairwise disjointness
    - The inability to determine the correct RHS on the basis of one token of lookahead
    - Def: $\text{FIRST}(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta \}$
      (If $\alpha \Rightarrow^* \varepsilon$, $\varepsilon$ is in $\text{FIRST}(\alpha)$)
Recursive-Descent Parsing

• Pairwise Disjointness Test:
  – For each nonterminal, A, in the grammar that has more than one RHS, for each pair of rules, \( A \rightarrow \alpha_i \) and \( A \rightarrow \alpha_j \), it must be true that
    \[
    \text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset
    \]

• Examples:
  \[
  \begin{align*}
  A &\rightarrow a \mid bB \mid cAb \\
  A &\rightarrow a \mid aB
  \end{align*}
  \]
Recursive-Descent Parsing

Left factoring can resolve the problem
Replace:

\[<\text{variable}> \rightarrow \text{identifier} \mid \text{identifier} [<\text{expression}>]\]

With:

\[<\text{variable}> \rightarrow \text{identifier} \ <\text{new}>\]
\[<\text{new}> \rightarrow \epsilon \mid [<\text{expression}>]\]
Bottom-up Parsing

- The parsing problem is finding the correct RHS in a right-sentential form to reduce to get the previous right-sentential form in the derivation
Bottom-up Parsing

- The parsing problem is finding the correct RHS in a right-sentential form to reduce to get the previous right-sentential form in the derivation

- Intuition about handles:
  - Def: $\beta$ is the handle of the right sentential form $\gamma = \alpha \beta w$ if and only if $S \Rightarrow^* \alpha Aw \Rightarrow^* \alpha \beta w$
  - Def: $\beta$ is a phrase of the right sentential form $\gamma$ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1 \beta \alpha_2$
  - Def: $\beta$ is a simple phrase of the right sentential form $\gamma$ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$
A Bottom-up Parse in Detail (1)

int + (int) + (int)

\[ E \rightarrow E + ( E ) \]
\[ E \rightarrow \text{int} \]
A Bottom-up Parse in Detail (2)

\[
\begin{align*}
\text{int} & + (\text{int}) + (\text{int}) \\
E & + (\text{int}) + (\text{int})
\end{align*}
\]

\[
\begin{align*}
E & \rightarrow E + (E) \\
E & \rightarrow \text{int}
\end{align*}
\]
A Bottom-up Parse in Detail (3)

\[
\begin{align*}
\text{int} &+ (\text{int}) + (\text{int}) \\
E &+ (\text{int}) + (\text{int}) \\
E &+ (E) + (\text{int})
\end{align*}
\]

\[
E \rightarrow E + (E) \\
E \rightarrow \text{int}
\]
A Bottom-up Parse in Detail (4)

\[
\text{int} + (\text{int}) + (\text{int}) \\
E + (\text{int}) + (\text{int}) \\
E + (E) + (\text{int}) \\
E + (\text{int})
\]
A Bottom-up Parse in Detail (5)

\[
\begin{align*}
\text{int} + (\text{int}) + (\text{int}) \\
E + (\text{int}) + (\text{int}) \\
E + (E) + (\text{int}) \\
E + (\text{int}) \\
E + (E)
\end{align*}
\]
A Bottom-up Parse in Detail (6)

\[
\begin{align*}
\text{int} &+ (\text{int}) + (\text{int}) \\
E &+ (\text{int}) + (\text{int}) \\
E &+ (E) + (\text{int}) \\
E &+ (\text{int}) \\
E &+ (E) \\
E &
\end{align*}
\]

A rightmost derivation in reverse

\[
\begin{align*}
E &
\end{align*}
\]

\[
\begin{align*}
E &+ (E) \\
E &+ (\text{int}) + (\text{int}) \\
E &+ (\text{int}) + (\text{int}) \\
E &+ (\text{int}) + (\text{int}) \\
\end{align*}
\]
Bottom-up Parsing

• Advantages of LR parsers:
  – They will work for nearly all grammars that describe programming languages.
  – They work on a larger class of grammars than other bottom-up algorithms, but are as efficient as any other bottom-up parser.
  – They can detect syntax errors as soon as it is possible.
  – The LR class of grammars is a superset of the class parsable by LL parsers.
Classes of grammars
Semantic Analysis

• Once sentence structure is understood, we can try to understand “meaning”
  – But meaning is too hard for compilers

• Compilers perform limited analysis to catch inconsistencies

• Some do more analysis to improve the performance of the program
Semantic Analysis in English

• Example:
  Jack said Jerry left his assignment at home.
  What does “his” refer to? Jack or Jerry?

• Even worse:
  Jack said Jack left his assignment at home?
  How many Jacks are there?
  Which one left the assignment?
Semantic Analysis in Programming

• Programming languages define strict rules to avoid such ambiguities

• This C++ code prints “4”; the inner definition is used

```cpp
#include <iostream>

int main() {
    int Jack = 3;
    {
        int Jack = 4;
        std::cout << Jack;
    }
    return 0;
}
```
More Semantic Analysis

• Compilers perform many semantic checks besides variable bindings

• Example:
  
  Jack left her homework at home.

• A “type mismatch” between her and Jack; we know they are different people
  – Presumably Jack is male
Static Semantic Analysis

• Types of Checks conducted by compiler:
  1. All identifiers are declared
  2. Types
  3. Inheritance relationships
  4. Classes defined only once
  5. Methods in a class defined only once
  6. Reserved identifiers are not misused
     And others . . .

• Complex languages => Complex checks
• Algorithm: Traverse the AST produced by the parser
END OF ICOM 4036 LECTURE 3
Bottom-up Parsing

• LR parsers must be constructed with a tool
• Knuth’s insight: A bottom-up parser could use the entire history of the parse, up to the current point, to make parsing decisions
  – There were only a finite and relatively small number of different parse situations that could have occurred, so the history could be stored in a parser state, on the parse stack
Bottom-up Parsing

• An LR configuration stores the state of an LR parser

\[(S_0X_1S_1X_2S_2\ldots X_mS_m, a_ia_{i+1}\ldots a_n$)\]
Bottom-up Parsing

- LR parsers are table driven, where the table has two components, an ACTION table and a GOTO table
  - The ACTION table specifies the action of the parser, given the parser state and the next token
    - Rows are state names; columns are terminals
  - The GOTO table specifies which state to put on top of the parse stack after a reduction action is done
    - Rows are state names; columns are nonterminals
Structure of An LR Parser

Parse Stack

$S_0 \ X_1 \ S_1 \ \ldots \ X_m \ S_m$

Top

Input

$a_i \ a_{i+1} \ \ldots \ a_m \$
Bottom-up Parsing

• Initial configuration: \((S_0, a_1…a_n\$)\)

• Parser actions:
  – If \(\text{ACTION}[S_m, a_i] = \text{Shift } S\), the next configuration is:
    \((S_0X_1S_1X_2S_2…X_mS_mS, a_{i+1}…a_n\$)\)
  – If \(\text{ACTION}[S_m, a_i] = \text{Reduce } A \rightarrow \beta\) and \(S = \text{GOTO}[S_{m-r}, A]\), where \(r = \text{the length of } \beta\), the next configuration is
    \((S_0X_1S_1X_2S_2…X_mS_{m-r}AS, a_ia_{i+1}…a_n\$)\)
Bottom-up Parsing

• Parser actions (continued):
  – If $\text{ACTION}[S_m, a_i] = \text{Accept}$, the parse is complete and no errors were found.
  – If $\text{ACTION}[S_m, a_i] = \text{Error}$, the parser calls an error-handling routine.
LR Parsing Table

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S5</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>S6</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>R2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>R4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S5</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>R6</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>S5</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>S5</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>S6</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>R1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>R3</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>R5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>E</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F</td>
</tr>
</tbody>
</table>

Some parts are Copyright © 2004 Pearson Addison-Wesley. All rights reserved.
Bottom-up Parsing

• A parser table can be generated from a given grammar with a tool, e.g., **yacc**
Optimization

• No strong counterpart in English, but akin to editing

• Automatically modify programs so that they
  – Run faster
  – Use less memory
  – In general, conserve some resource

• The project has no optimization component
Optimization Example

\[ X = Y \times 0 \] is the same as \[ X = 0 \]

**NO!**

Valid for integers, but not for floating point numbers
Code Generation

• Produces assembly code (usually)

• A translation into another language
  – Analogous to human translation
Intermediate Languages

• Many compilers perform translations between successive intermediate forms
  – All but first and last are intermediate languages internal to the compiler
  – Typically there is 1 IL

• IL’s generally ordered in descending level of abstraction
  – Highest is source
  – Lowest is assembly
Intermediate Languages (Cont.)

• IL’s are useful because lower levels expose features hidden by higher levels
  – registers
  – memory layout
  – etc.

• But lower levels obscure high-level meaning
Issues

• Compiling is almost this simple, but there are many pitfalls.

• Example: How are erroneous programs handled?

• Language design has big impact on compiler
  – Determines what is easy and hard to compile
  – Course theme: many trade-offs in language design
Compilers Today

• The overall structure of almost every compiler adheres to our outline

• The proportions have changed since FORTRAN
  – Early: lexing, parsing most complex, expensive
  – Today: optimization dominates all other phases, lexing and parsing are cheap
Trends in Compilation

• Compilation for speed is less interesting. But:
  – scientific programs
  – advanced processors (Digital Signal Processors, advanced speculative architectures)

• Ideas from compilation used for improving code reliability:
  – memory safety
  – detecting concurrency errors (data races)
  – ...