CS143
Compilers

Lecture 1
June 27, 2001

What is a Compiler?

A compiler is a program that accepts as input a program written in one language, known as the source language, and generates as output an equivalent program written in another language, called a target language.

Typically, the source language is a high-level programming language such as Java or C++, while the target language is a lower-level language such as assembly language or machine language.

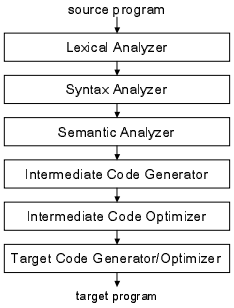
Historical Background

- When computers were invented, there were no programming languages, only machine language.
- The first FORTRAN compiler was developed in 1957 by a team led by Jim Backus and required 18 man-years of work.
- Present-day compiling techniques emerged from the difficulty of implementing early compilers.

Where compiling techniques are used

- Programming language compilers (e.g. for Java, C++ or FORTRAN)
- Internet protocol servers and clients
  - Web browsers and web servers
  - E-mail clients
- Profilers and debuggers
- Development environments

Phases of a Compiler



Analysis

- The front end of a compiler analyzes the source program text and constructs an intermediate representation of the source code that can easily be optimized and translated into target code by the back end.
- The analysis stage typically consists of the following phases:
  - Lexical analysis, or scanning
  - Syntactic analysis, or parsing
  - Semantic analysis
  - Intermediate code generation

Lexical Analysis

- A compiler is given a source program in the form of a file that contains ordinary text.
- On the other hand, the syntax of a programming language is specified using the language's tokens, such as keywords, identifiers and punctuation.
- Lexical analysis is the process of recognizing these tokens from the text of the source program. This process is analogous to that of grouping letters into words.

Syntax Analysis

- In order to translate the source program into a target language, the compiler must understand the syntactic structure of the program.
- Syntax analysis, or parsing, is the process of organizing the tokens of a source program into the constructs of the source language.
- An intuitive example of parsing is the recognition of a book as a collection of chapters, each of which contains paragraphs, which themselves contain words.

Semantic Analysis

- A syntactically correct sentence does not necessarily make any sense. This is the main reason why Mad Libs are funny.
- Semantic analysis is used to ensure that the source program is performing operations that are valid according to the language specification.
- Such analysis typically involves checking the types of operands.

Intermediate Code Generation

- A source program that is determined to be semantically valid is ready to be translated into a target language.
- By translating into an intermediate representation, one can easily build compilers for several source languages that translate to a single target language. These compilers would share a common back end.

Synthesis

- The back end of a compiler performs the actual synthesis of target code from intermediate code.
- The goal is to produce target code that correctly performs the operations described by the source program, and does so as efficiently as possible.
- Synthesis typically consists of the following phases:
  - Intermediate code optimization.
  - Target code generation.
  - Optimization.

Intermediate Code Optimization

- The target code generated by a compiler is much less efficient than the target code that one can synthesize by hand.
- Optimization is an attempt to make the target code more efficient, typically using a variety of heuristics.
- This phase can be the most expensive due to its complexity. Therefore, it can be skipped if efficient target code is not a high priority, as is the case when debugging the source program.

Target Code Generation

- Code generation is the most essential phase of synthesis: the actual translation of intermediate code into target code.
- The target language is usually machine code or assembly code.
- Often, this phase includes additional optimization that takes advantage of features that are specific to the platform on which the target program will be executed.

Issues in Compiler Design

- In designing a compiler, several issues must be considered in addition to the central task of translation to the target language. The most prevalent issues are:
  - Symbol table management
  - Error Recovery
  - Source Language Design
  - Passes

Symbol Table Management

- Often, entities in the source program correspond to entities in the target program:
  - Variables correspond to memory locations
  - Operations correspond to instructions
- Therefore, it is necessary to build a symbol table during the analysis stage and maintain it during the synthesis stage to aid in code generation and optimization.
- In compilers for programming languages, symbol tables typically store all of the identifiers in the source program, along with information about their type and scope.

Error Recovery

- Once an error has been detected in the source program, synthesis cannot be performed.
- Nevertheless, it is desirable to detect as many errors as possible, so that one may fix all of them before recompiling.
- It is also desirable to avoid reporting spurious “errors” resulting from the compiler’s inability to recover from an initial error.
- Therefore, once an error is detected, the compiler should attempt to recover and continue analysis.

Source Language Design

- In order to build a compiler for a given source language, the source language must be well-defined:
  - –It should be easy to ascertain the lexical and syntactic structure, and validity, of any source program.
  - –The semantics of the source language should be completely specified so that the validity of any operation can be determined.
  - –It should be easy to associate concepts of the source language with corresponding concepts of a given intermediate language.

Passes

- A pass is an iteration through a representation of an entire source program, presumably to construct a different representation.
- Conceivably, each phase of a compiler can accept the output from the previous phase as input, and generate its output for the next phase. Thus each phase requires one pass.
- In practice, several phases can be performed during a single pass. This is highly desirable for maximizing efficiency.
- Typically, analysis can be performed in one pass, while synthesis requires several passes.

Compiler Generation Tools

- The implementation of several phases of a compiler can be generated automatically by tools that employ well-established, straightforward techniques.
- These tools are themselves compilers that translate a specification for a source language into a program that implements that specification.
- Commonly used tools are:
  - Lexical analyzer generators such as lex or flex
  - Parser generators such as yacc or bison
  - Code-generator generators

A Simple Compiler

- We now illustrate the phases of a compiler by implementing a compiler for a very simple language.
- Our source language consists of only a few tokens: integers and the four basic arithmetic operations, +, -, *, and /. White space is permitted but ignored.
- Our compiler will generate textual representations of the Java virtual machine instructions for evaluating the arithmetic expressions described by the source program.

A Simple Lexical Analyzer

- Our lexical analyzer implements a simple *state transition diagram* in order to recognize the valid tokens: AddOp, MulOp, number, or EOF.
- Each token returned by the lexical analyzer to the parser is accompanied by appropriate attributes:
  - For AddOp, a numeric attribute denoting + or –
  - For MulOp, a numeric attribute denoting * or /
  - For number, the index in the symbol table where the numeric value is stored

A Simple Parser

- We define the syntax of our source language using a *context-free grammar* that enforces the proper associativity and precedence of operations:
  - Expression -> Term Expression'
  - Expression' -> AddOp Term Expression' | ∈
  - Term -> Factor Term'
  - Term' -> MulOp Factor Term' | ∈
  - Factor -> number
  - AddOp -> +|-
  - MulOp -> *|/
- We will use recursive-descent, top-down parsing.

A Simple Code Generator

- As each token is recognized, a Java instruction is generated and printed to standard output.
  - For each number, an ldc (load constant) instruction is emitted. This instruction fetches a value from a constant pool and pushes it onto the operand stack.
  - For each operation, an appropriate instruction (iadd, isub, imul or idiv) is emitted. Each of these instructions pops its operands off of the stack and pushes the result onto the stack.

Lexical Analysis

- We now examine the lexical analysis phase in depth, focusing on the following aspects:
  - The role of the lexical analyzer
  - Input buffering
  - Specification of tokens
  - Recognition of tokens
  - Lexical analyzer generators

The Role of Lexical Analysis

- The purpose of lexical analysis is to convert the text of a source program into a sequence of tokens, which in turn are passed to the second phase, syntax analysis.
- Typically, the lexical analyzer is a subroutine of a parser, invoked repeatedly in order to fetch tokens from the source program.
- Lexical analysis is often accompanied by preprocessing, such as the removal of comments, or annotation for use by later phases.

Why Lexical Analyzers?

- Techniques used in syntax analysis are more powerful than those used in lexical analysis. Why not simply implement parsers to analyze syntax directly from the text of a source program?
  - Separating lexical analysis from syntax analysis simplifies both phases. A parser that works with characters rather than tokens is considerably more complex.
  - Separation allows greater efficiency through the use of specialized input buffering techniques
  - Portability issues related to input alphabets or characters sets can be isolated from other phases

Tokens, Patterns & Lexemes

- A *token* is a conceptual component of a source language, analogous to a part of speech in a written language.
- For each token, there exist one or more *lexemes*, which are strings that are identified with the token by the lexical analyzer. Lexemes correspond to specific words in a written language.
- A lexical analyzer uses a *pattern* in order to recognize lexemes belonging to a given token.

Examples

- A common token in programming languages is an identifier, typically used for the name of a variable or function.
- A commonly used pattern for describing valid identifiers is: a sequence of letters or digits, beginning with a letter. Thus “abc23” is valid, while “2a” is not.
- Using the above pattern, “name”, “value1”, and “a2b2c3” are all lexemes for the identifier token.
- Most tokens have very few lexemes. A token for relational operators has only four lexemes: “<“, “>”, “<=“, and “>=“. Each keyword, such as “if”, is a separate token with only one lexeme, the keyword itself.

Attributes for Tokens

- In many cases, a token can have more than one lexeme, or even infinitely many lexemes.
- This occurs because the lexemes, though different, are syntactically equivalent, so it simplifies the parser to represent them using a single token.
- Because the actual lexeme is still important to later phases of the compiler, a lexical analyzer needs to provide additional information about each token. This additional information is conveyed in the form of attributes.

Examples

- Consider the tokens from the previous examples: the keyword “if”, relational operators, and identifiers.
  - The “if” token requires no attributes.
  - The relational operator token needs an attribute indicating which of the 4 relational operations is to be performed. While this matters little for syntax analysis, it is essential for synthesis.
  - The identifier token usually has a symbol table entry as its attribute, so that multiple occurrences of the same identifier will have the same attribute.

Lexical Errors

- A lexical error arises when a sequence of characters in the source program cannot be recognized as any valid token of the source language.
- To recover from such an error, a lexical analyzer may try to "alter" the input, in an attempt to "fix" the error.
- Such transformations may include:
  - Deleting characters
  - Inserting a missing character
  - Replacing an invalid character with a valid one

Examples

- In the C language, @ is a character that is not part of any valid lexeme unless enclosed by quotes. Therefore its occurrence outside of quotes is a lexical error.
- To recover from such an error, a lexical analyzer might assume that this invalid character was meant to be included in a string constant, so it might insert a " before the @.
- Or, it might assume that a string constant was ended prematurely by using a " where there should have been a \", and therefore recover by searching for the next " and continuing.

Input Buffering

- Many of the tasks involved in compiling can be performed in memory.
- Therefore, a significant amount of the time spent in compiling arises from file I/O.
- For maximum efficiency, it is desirable to read characters from the source file in large segments, even though the input is scanned one character at a time.

Buffer Pairs and Sentinels

- Since a lexical analyzer may need to scan ahead several characters in order to recognize a single token, an input buffering scheme should be used.
- A common scheme is to use buffer pairs, scanning through one buffer and then loading new input into the other.
- A buffer-pair scheme can be made more efficient by adding a sentinel character, a character that cannot occur in the source program, at the end of each buffer.

Specification of Tokens

- Tokens can be most efficiently recognized by specifying patterns for each token and determining whether the input matches one of the patterns.
- A regular expression is a universal notation for specifying patterns.
- Given a set of regular expressions for each token, a lexical analyzer for a source language can be generated automatically.

Strings and Languages

- An *alphabet* or *character class* is any finite set of symbols, such as the binary alphabet {0,1} or the ASCII alphabet.
- A *string* over an alphabet is a finite sequence of symbols from that alphabet. The empty string, ε, is a string of length zero.
- A *language* is a set of strings over a fixed alphabet.
- Given two strings x and y, the *concatenation* of x and y, xy, is the string formed by appending y to x.
- The concatenation of x with itself, n times, can be written as x^n. For example, x^4 = xxxx.

Operations on Languages

- The *union* of two languages L and M is the set of all strings that are in either L or M.
- The *concatenation* of L and M is the set of all strings that are the concatenation of some string in L with some string in M.
- The *Kleene closure* of L, L^*, is the set of all strings constructed by concatenating zero or more strings of L.
- The *positive closure* of L, L^+, is the set of all strings constructed by concatenating one more strings of L. Thus, $L^* = L^+ \cup \varepsilon$.

Regular Expressions

- A regular expression over an alphabet Σ is a pattern that defines a language over Σ.
- ε is a regular expression that denotes $\{\varepsilon\}$.
- If a is a symbol in Σ, then a is a regular expression that denotes {a}.
- If r and s are regular expressions denoting the languages L(r) and L(s), then
  - r|s is a regular expression denoting $L(r) \cup L(s)$
  - rs is a regular expression denoting L(r)L(s)
  - r^* is a regular expression denoting $L(r)^*$

Examples

- We now define some regular expressions over the alphabet Σ = {a,b,c}.
  - a is the regular expression describing the language {a}.
  - a|b describes the language {a,b}.
  - a* denotes the language of all strings of zero or more a's.
  - (a|b)*c denotes the set of all strings of zero or more a's or b's, followed by a c.
  - (a*b)(a*b)* describes all strings of a's and b's that end with a b.

Regular Definitions

- While regular expressions by themselves can be used to specify patterns, they can often be cumbersome.
- It is more convenient to assign names to regular expressions, and then use these names to write other regular expressions.
- A *regular definition* over an alphabet Σ is a sequence of definitions, where each definition is the assignment of a name to a regular expression over the union of Σ and the previously defined names.

Examples

- Suppose that valid identifiers for a programming language are all strings of letters or digits that begin with a letter. Then a regular expression describing the language of valid identifiers is:

  (a|b|…|z|A|B|…|Z)(a|b|…|z|A|B|…|Z|0|…|9)*

  which is overly cumbersome.
- Using regular definitions, this simplifies to:

  letter = a|b|…|z|A|B|…|Z

  digit = 0|1|…|9

  identifier = letter (letter | digit)*

Notational Shorthands

- Regular expressions can be written even more easily by adopting shorthands for common constructs.
  - If r is a regular expression, then $r^+ = rr^*$. r^+ denotes the positive closure of L(r), and intuitively means "one or more occurrences of r"
  - r? is equivalent to r| ε, and denotes the language $L(r) \cup \varepsilon$, or, intuitively, "zero or one occurrence of r".
  - A character class is a shorthand for a regular expression denoting the language consisting of select alphabet symbols. For example, [abc] denotes a|b|c, and [a-z] denotes a|b|…|z.

Examples

- Using notational shorthands, our description of valid identifiers simplifies even further:

  letter = [a-zA-z]

  digit = [0-9]

  identifier = letter (letter | digit)*

- Similarly, floating-point numbers can be described as follows:

  digit = [0-9]

  exp = [eE] [+-] digit*

  num = digit* (. digit*)? exp?

Non-regular Languages

- Many languages cannot be described by regular expressions. Such languages are said to be non-regular.
- For example, languages consisting of balanced or nested constructs, such as the set of all strings of balanced parentheses, are non-regular.
- Languages requiring a "memory" to recognize their strings are non-regular. One example is the set of all strings of the form wcw, where w is any string of a's and b's.

Recognition of Tokens

- Now that we know how to specify tokens, we can figure out how to recognize them.
- The basic idea is as follows:
  - First, we construct a regular expression for each token.
  - Then, we scan through the input, one character at a time, until the input is recognized as a string that matches one of the patterns described by the regular expressions.
  - Return the corresponding token, and repeat this process for the remaining input.

Transition Diagrams

- In attempting to recognize a token, we could try to match the input to each regular expression in turn until we are successful.
- This is woefully inefficient, as this forces us to pass through the input several times.
- We instead use a *transition diagram* to effectively combine the regular expressions into a single regular expression that can recognize any valid token.
- Using a transition diagram, it is rarely necessary to backtrack through the input.

Anatomy of Transition Diagrams

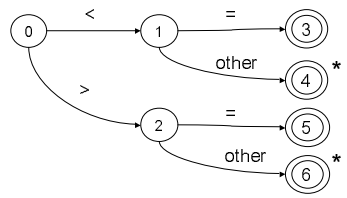
- A transition diagram consists of a set of *states* and a set of *transitions*.
- Each diagram contains a single state that is called the start state. There can be no transitions to the start state.
- Each diagram contains at least one state that is called a final or accepting state. There can be no transitions from an accepting state.

Using Transition Diagrams

- Transitions between states are guided by input symbols from a given alphabet.
- Initially, the scanner is in the start state. It then reads the first input symbol, and determines which transition to use.
- Input is read and transitions followed until an accepting state is reached. This signifies that a token has been recognized, and the string of input symbols read since the start state is the corresponding lexeme.
- Control returns to the start state to read the next token.

The following transition diagram can be used to recognize the relational operators <, <=, > and >=.

State 0 is the start state, and double circles denotes accepting states. The * next to a state indicates that the scanner should move back by one input symbol.



Implementing a Transition Diagram

- Once a transition diagram has been constructed, it is easy to write a program to implement it.
- Each state corresponds to a segment of code.
- If there are edges leaving the state, then this code must read the next input character and cause a branch to the code for the appropriate state.
- If a state is a final state, a token has been recognized.
- If a state is not a final state, and no transition to another state is available, then a lexical error has been detected.