LECTURE 1: OVERVIEW AND HISTORY

• Evolution
  • Design considerations: What is a good or bad programming construct?
  • Early 70s: structured programming in which goto-based control flow was replaced by high-level constructs (e.g. while loops and case statements).
  • Late 80s: nested block structure gave way to object-oriented structures.

• Special Purposes
  • Many languages were designed for a specific problem domain (e.g. Scientific applications, Business applications, Artificial intelligence, Systems programming, Internet programming, etc).

• Personal Preference
  • The strength and variety of personal preference makes it unlikely that anyone will ever develop a universally accepted programming language.
• Expressive Power
  • Theoretically, all languages are equally powerful (Turing complete).
  • Language features have a huge impact on the programmer’s ability to read, write, maintain, and analyze programs.

• Ease of Use for Novice
  • Low learning curve and often interpreted, e.g. Basic and Logo.

• Ease of Implementation
  • Runs on virtually everything, e.g. Basic, Pascal, and Java.

• Open Source
  • Freely available, e.g. Java.

• Excellent Compilers and Tools
  • Supporting tools to help the programmer manage very large projects.

• Economics, Patronage, and Inertia
  • Powerful sponsor: Cobol, PL/I, Ada.
  • Some languages remain widely used long after "better" alternatives.
Classification of Programming Languages

- **Declarative**: Implicit solution. What should the computer do?
  - Functional
    - Lisp, Scheme, ML, Haskell
  - Logic
    - Prolog
  - Dataflow
    - Simulink, Scala

- **Imperative**: Explicit solution. How should the computer do it?
  - Procedural
    - Fortran, C
  - Object-Oriented
    - Smalltalk, C++, Java
Programs written in high-level languages can be run in two ways.

- *Compiled* into an executable program written in machine language for the target machine.
- Directly *interpreted* and the execution is simulated by the interpreter.

In general, which approach is more efficient?
Programs written in high-level languages can be run in two ways.

• Compiled into an executable program written in machine language for the target machine.

• Directly interpreted and the execution is simulated by the interpreter.

In general, which approach is more efficient?

Compilation is always more efficient…but interpretation leads to more flexibility.
How do you choose? Typically, most languages are implemented using a mixture of both approaches.

Practically speaking, there are two aspects that distinguish what we consider “compilation” from “interpretation”.

• Thorough Analysis
  • Compilation requires a thorough analysis of the code.

• Non-trivial Transformation
  • Compilation generates intermediate representations that typically do not resemble the source code.
Lecture 2: Compilation and Interpretation

Preprocessing

• Initial translation step.
• Slightly modifies source code to be interpreted more efficiently.
• Removing comments and whitespace, grouping characters into tokens, etc.

Linking

• Linkers merge necessary library routines to create the final executable.
LECTURE 2: COMPILATION AND INTERPRETATION

Post-Compilation Assembly

• Many compilers translate the source code into assembly rather than machine language.

• Changes in machine language won’t affect source code.

• Assembly is easier to read (for debugging purposes).

Source-to-source Translation

• Compiling source code into another high-level language.

• Early C++ programs were compiled into C, which was compiled into assembly.
LECTURE 3: COMPILER PHASES

Front-End Analysis

Source Program

- Scanner (Lexical Analysis)
- Tokens
- Parser (Syntax Analysis)
- Parse Tree
  - Semantic Analysis & Intermediate Code Generation
  - Abstract Syntax Tree

Back-End Synthesis

Abstract Syntax Tree

- Machine-Independent Code Improvement
- Modified Intermediate Form
  - Target Code Generation
  - Assembly or Object Code
    - Machine-Specific Code Improvement
    - Modified Assembly or Object Code
Lexical analysis is the process of tokenizing characters that appear in a program.

A scanner (or lexer) groups characters together into meaningful tokens which are then sent to the parser.

As the scanner reads in the characters, it produces meaningful tokens.

Tokens are typically defined using regular expressions, which are understood by a lexical analyzer generator such as lex.


The resulting tokens: int, main, (, ), {, int, i, =, getint, (, ), ....
Lecture 3: Compiler Phases

Syntax analysis is performed by a parser which takes the tokens generated by the scanner and creates a parse tree which shows how tokens fit together within a valid program.

The structure of the parse tree is dictated by the grammar of the programming language.
Semantic analysis is the process of attempting to discover whether a valid pattern of tokens is actually meaningful.

Even if we know that the sequence of tokens is valid, it may still be an incorrect program.

For example: \( a = b; \)

What if \( a \) is an int and \( b \) is a character array?

To protect against these kinds of errors, the semantic analyzer will keep track of the types of identifiers and expressions in order to ensure they are used consistently.
What kinds of errors can be caught in the lexical analysis phase?

• Invalid tokens.

What kinds of errors are caught in the syntax analysis phase?

• Syntax errors: invalid sequences of tokens.
LECTURE 3: COMPILER PHASES

• Static Semantic Checks: semantic rules that can be checked at compile time.
  • Type checking.
  • Every variable is declared before used.
  • Identifiers are used in appropriate contexts.
  • Checking function call arguments.

• Dynamic Semantic Checks: semantic rules that are checked at run time.
  • Array subscript values are within bounds.
  • Arithmetic errors, e.g. division by zero.
  • Pointers are not dereferenced unless pointing to valid object.
  • When a check fails at run time, an exception is raised.
Assuming C++, what kinds of errors are these?

- `int = @3;`
- `int = ?3;`
- `int y = 3; x = y;`
- "Hello, World!"
- `int x; double y = 2.5; x = y;`
- `void sum(int, int); sum(1,2,3);`
- `myint++`
- `z = y/x  // y is 1, x is 0`
LECTURE 3: COMPILER PHASES

• Assuming C++, what kinds of errors are these?

- `int = @3;` // Lexical
- `int = ?3;` // Syntax
- `int y = 3; x = y;` // Static semantic
- “Hello, World!” // Syntax
- `int x; double y = 2.5; x = y;` // Static semantic
- `void sum(int, int); sum(1,2,3);` // Static Semantic
- `myint++` // Syntax
- `z = y/x` // y is 1, x is 0 // Dynamic Semantic
Lecture 3: Compiler Phases

Code Optimization

- Once the AST (or alternative intermediate form) has been generated, the compiler can perform machine-independent code optimization.

- The goal is to modify the code so that it is quicker and uses resources more efficiently.

- There is an additional optimization step performed after the creation of the object code.
LECTURE 3: COMPILER PHASES

Target Code Generation

• Goal: translate the intermediate form of the code (typically, the AST) into object code.

• In the case of languages that translate into assembly language, the code generator will first pass through the symbol table, creating space for the variables.

• Next, the code generator passes through the intermediate code form, generating the appropriate assembly code.

• As stated before, the compiler makes one more pass through the object code to perform further optimization.
We know from the previous lecture that the front-end of the compiler has three main phases:

• Scanning
• Parsing
• Semantic Analysis

So how do we specify what a valid token is? Or what constitutes a valid construct?
Tokens can be constructed from regular characters using just three rules:

1. Concatenation.
2. Alternation (choice among a finite set of alternatives).

Any set of strings that can be defined by these three rules is a regular set. Regular sets are generated by regular expressions.
Formally, all of the following are valid regular expressions (let $R$ and $S$ be regular expressions and let $\Sigma$ be a finite set of symbols):

- The empty set.
- The set containing the empty string $\varepsilon$.
- The set containing a single literal character $\alpha$ from the alphabet $\Sigma$.
- Concatenation: $RS$ is the set of strings obtained by concatenation of one string from $R$ with a string from $S$.
- Alternation: $R | S$ describes the union of $R$ and $S$.
- Kleene Closure: $R^*$ is the set of strings that can be obtained by concatenating any number of strings from $R$. 
LEC T U R E 4: SYNTAX

You can either use parentheses to avoid ambiguity or assume Kleene star has the highest priority, followed by concatenation then alternation.

Examples:

- $a^*$ = {$\epsilon$, $a$, $aa$, $aaa$, $aaaa$, $aaaaa$, ...}
- $a \mid b^*$ = {$\epsilon$, $a$, $b$, $bb$, $bbb$, $bbbb$, ...}
- $(ab)^*$ = {$\epsilon$, $ab$, $abab$, $ababab$, $abababab$, ...}
- $(a \mid b)^*$ = {$\epsilon$, $a$, $b$, $aa$, $ab$, $ba$, $bb$, $aaa$, $aab$, ...}
Create regular expressions for the following examples:

• Zero or more c’s followed by a single a or a single b.
• Binary strings starting and ending with 1.
• Binary strings containing at least 3 1’s.
Create regular expressions for the following examples:

• Zero or more c’s followed by a single a or a single b.
  \( c^*(a|b) \)

• Binary strings starting and ending with 1.
  \( 1|1(0|1)^*1 \)

• Binary strings containing at least 3 1’s.
  \( 0^*10^*10^*1(0|1)^* \)
We can completely define our tokens in terms of regular expressions, but more complicated constructs necessitate recursion.

The set of strings that can be defined by adding recursion to regular expressions is known as a Context-Free Language.

Context-Free Languages are generated by Context-Free Grammars.
Context-free grammars are composed of rules known as *productions*.

Each production has left-hand side symbols known as *non-terminals*, or *variables*.

On the right-hand side, a production may contain *terminals* (tokens) or other non-terminals.

One of the non-terminals is named the *start symbol*.

\[
\begin{align*}
\text{expr} & \rightarrow \text{id} \mid \text{number} \mid - \text{expr} \mid ( \text{expr} ) \mid \text{expr op expr} \\
\text{op} & \rightarrow + \mid - \mid * \mid / 
\end{align*}
\]
So, how do we use the context-free grammar to generate syntactically valid strings of terminals (or tokens)?

1. Begin with the start symbol.
2. Choose a production with the start symbol on the left side.
3. Replace the start symbol with the right side of the chosen production.
4. Choose a non-terminal A in the resulting string.
5. Replace A with the right side of a production whose left side is A.
6. Repeat 4 and 5 until no non-terminals remain.
Write a grammar which recognizes if-statements of the form:

```plaintext
if expression
    statements
else
    statements
```

where expressions are of the form `id > num` or `id < num`. Statements can be any numbers of statements of the form `id = num` or `print id`. 
LECTURE 4: SYNTAX

program → if expr stmts else stmts
expr → id > num | id < num
stmts → stmt stmts | stmt
stmt → id = num | print id
A recognizer for a language is a program that takes a string $x$ as input and answers “yes” if $x$ is a sentence of the language and “no” otherwise.

In the context of lexical analysis, given a string and a regular expression, a recognizer of the language specified by the regular expression answers “yes” if the string is in the language.

How can we recognize a regular expression $(\text{int})$? What about $(\text{int} \mid \text{for})$?

We could, for example, write an ad hoc scanner that contained simple conditions to test, the ability to peek ahead at the next token, and loops for numerous characters of the same type.
A set of regular expressions can be compiled into a recognizer automatically by constructing a finite automaton using scanner generator tools (lex, for example).

A finite automaton is a simple idealized machine that is used to recognize patterns within some input.

- A finite automaton will accept or reject an input depending on whether the pattern defined by the finite automaton occurs in the input.

The elements of a finite automaton, given a set of input characters, are

- A finite set of states (or nodes).
- A specially-denoted start state.
- A set of final (accepting) states.
- A set of labeled transitions (or arcs) from one state to another.
Finite automata come in two flavors.

• **Deterministic**
  • Never any ambiguity.
  • For any given state and any given input, only one possible transition.

• **Non-deterministic**
  • There may be more than one transition from any given state for any given character.
  • There may be epsilon transitions – transitions labeled by the empty string.

There is no obvious algorithm for converting regular expressions to DFAs.
Lecture 5: Scanning

Typically scanner generators create DFAs from regular expressions in the following way:

- Create NFA equivalent to regular expression.
- Construct DFA equivalent to NFA.
- Minimize the number of states in the DFA.
LECTURE 5: SCANNING

• Concatenation: $ab$

• Alternation: $a | b$

• Kleene Closure: $a^*$
Lecture 5: Scanning

Create NFAs for the regular expressions we created before:

• Zero or more c’s followed by a single a or a single b.
  \[ c^*(a \mid b) \]

• Binary strings starting and ending with 1.
  \[ 1 \mid 1(0 \mid 1)^*1 \]

• Binary strings containing at least 3 1’s.
  \[ 0^*10^*10^*1(0 \mid 1)^* \]
How do we take our minimized DFA and practically implement a scanner? After all, finite automata are idealized machines. We didn’t actually build a physical recognizer yet! Well, we have two options.

• Represent the DFA using goto and case (switch) statements.
  • Handwritten scanners.

• Use a table to represent states and transitions. Driver program simply indexes table.
  • Auto-generated scanners.
  • The scanner generator Lex creates a table and driver in C.
  • Some other scanner generators create only the table for use by a handwritten driver.
state = s1
token = ‘’
loop
  case state of
    s1:
      case in_char of
        ‘c’: state = s2
        else error
    s2:
      case in_char of
        ‘a’: state = s1
        ‘b’: state = s1
        ‘ ’: state = s1
        return token
        else error
  token = token + in_char
read new in_char
Longest Possible Token Rule

So, why do we need to peek ahead? Why not just accept when we pick up ‘c’ or ‘cac’?

Scanners need to accept as many tokens as they can to form a valid token.

For example, 3.14159 should be one literal token, not two (e.g. 3.14 and 159).

So when we pick up ‘4’, we peek ahead at ‘1’ to see if we can keep going or return the token as is. If we peeked ahead after ‘4’ and saw whitespace, we could return the token in its current form.

A single peek means we have a look-ahead of one character.
A driver program uses the current state and input character to index into the table. We can either

- Move to a new state.
- Return a token (and save the image).
- Raise an error (and recover gracefully).
There are two types of LL parsers: Recursive Descent Parsers and Table-Driven Top-Down Parsers.
In a table-driven parser, we have two elements:

- A driver program, which maintains a stack of symbols. (language independent)
- A parsing table, typically automatically generated. (language dependent)
Here’s the general method for performing table-driven parsing:

- We have a stack of grammar symbols. Initially, we just push the start symbol.
- We have a string of input tokens, ending with $.
- We have a parsing table $M[N, T]$.
  - We can index into $M$ using the current non-terminal at the top of the stack and the input token.

1. If top == input == ‘$’: accept.
2. If top == input: pop the top of the stack, read new input token, goto 1.
3. If top is nonterminal:
   - if $M[N, T]$ is a production: pop top of stack and replace with production, goto 1.
   - else error.
4. Else error.
Calculating an LL(1) parsing table includes calculating the *first* and *follow* sets. This is how we make decisions about which production to take based on the input.
LECTURE 8: PARSING

First Sets

Case 1: Let’s say $N \rightarrow \omega$. To figure out which input tokens will allow us to replace $N$ with $\omega$, we calculate First($\omega$) – the set of tokens which could start the string $\omega$.

- If $X$ is a terminal symbol, First($X$) = $X$.
- If $X$ is $\epsilon$, add $\epsilon$ to First($X$).
- If $X$ is a non-terminal, look at all productions where $X$ is on left-hand side. Each production will be of the form: $X \rightarrow Y_1 Y_2 \ldots Y_k$ where $Y$ is a nonterminal or terminal. Then:
  - Put First($Y_1$) - $\epsilon$ in First($X$).
  - If $\epsilon$ is in First($Y_1$), then put First($Y_2$) - $\epsilon$ in First($X$).
  - If $\epsilon$ is in First($Y_2$), then put First($Y_3$) - $\epsilon$ in First($X$).
  - ...
  - If $\epsilon$ is in $Y_1, Y_2, \ldots, Y_k$, then add $\epsilon$ to First($X$).
LECTURE 8: PARSING

If we compute First(\(X\)) for every terminal and non-terminal \(X\) in a grammar, then we can compute First(\(\omega\)), the tokens which can veritably start any string derived from \(\omega\).

Why do we care about the First(\(\omega\)) sets? During parsing, suppose the top-of-stack symbol is nonterminal A and there are two productions \(A \rightarrow \alpha\) and \(A \rightarrow \beta\). Suppose also that the current token is \(a\). Well, if First(\(\alpha\)) includes \(a\), then we can predict this will be the production taken.
LECTURE 8: PARSING

Follow Sets

Follow(\(N\)) gives us the set of terminal symbols that could follow the non-terminal symbol \(N\). To calculate Follow(\(N\)), do the following:

• If \(N\) is the starting non-terminal, put EOF (or other program-ending symbol) in Follow(\(N\)).

• If \(X \rightarrow \alpha N\), where \(\alpha\) is some string of non-terminals and/or terminals, put Follow(\(X\)) in Follow(\(N\)).

• If \(X \rightarrow \alpha N\beta\) where \(\alpha, \beta\) are some string of non-terminals and/or terminals, put First(\(\beta\)) in Follow(\(N\)). If First(\(\beta\)) includes \(\epsilon\), then put Follow(\(X\)) in Follow(\(N\)).
Why do we care about the Follow(N) sets? During parsing, suppose the top-of-stack symbol is nonterminal A and there are two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$. Suppose also that the current token is $a$. What if neither First($\alpha$) nor First($\beta$) contain $a$, but they contain $\varepsilon$? We use the Follow sets to determine which production to take.
The basic outline for creating a parsing table from a LL(1) grammar is the following:

• Compute the First sets of the non-terminals.
• Compute the Follow sets of the non-terminals.
• For each production \( N \to \omega \),
  • Add \( N \to \omega \) to \( M[N, t] \) for each \( t \) in First(\( \omega \)).
  • If First(\( \omega \)) contains \( \epsilon \), add \( N \to \omega \) to \( M[N, t] \) for each \( t \) in Follow(\( N \)).
• All undefined entries represent a parsing error.
LECTURE 9: COMPUTING AN LL(1) PARSING TABLE

stmt → if expr then stmt else stmt
stmt → while expr do stmt
stmt → begin stmts end
stmts → stmt ; stmts
stmts → ε
expr → id

Compute the LL(1) parsing table for this grammar and parse the string:

while id do begin begin end ; end $
LECTURE 10: SEMANTIC ANALYSIS

We’ve discussed in previous lectures how the syntax analysis phase of compilation results in the creation of a parse tree.

Semantic analysis is performed by annotating, or decorating, the parse tree.

These annotations are known as attributes.

An attribute grammar “connects” syntax with semantics.
LECTURE 10: SEMANTIC ANALYSIS

Attribute Grammars

- Each grammar production has a semantic rule with actions (e.g. assignments) to modify values of attributes of (non)terminals.
- A (non)terminal may have any number of attributes.
- Attributes have values that hold information related to the (non)terminal.

- General form:

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;A&gt; \rightarrow &lt;B&gt; &lt;C&gt;)</td>
<td>( A.a := ...; B.a := ...; C.a := ... )</td>
</tr>
</tbody>
</table>
Some points to remember:

• A (non)terminal may have any number of attributes.

• The val attribute of a (non)terminal holds the subtotal value of the subexpression.

• Nonterminals are indexed in the attribute grammar to distinguish multiple occurrences of the nonterminal in a production – this has no bearing on the grammar itself.

• Strictly speaking, attribute grammars only contain copy rules and semantic functions.

• Semantic functions may only refer to attributes in the current production.
LECTURE 10: SEMANTIC ANALYSIS

Strictly speaking, attribute grammars only consist of copy rules and calls to semantic functions. But in practice, we can specify well-defined notation to make the semantic rules look more code-like.

\[
\begin{align*}
E_1 & \rightarrow E_2 + T \\
E_1 & \rightarrow E_2 - T \\
E & \rightarrow T \\
T_1 & \rightarrow T_2 * F \\
T_1 & \rightarrow T_2 / F \\
T & \rightarrow F \\
F_1 & \rightarrow - F_2 \\
F & \rightarrow ( E ) \\
F & \rightarrow \text{const}
\end{align*}
\]

\[
\begin{align*}
E_1.\text{val} & := E_2.\text{val} + T.\text{val} \\
E_1.\text{val} & := E_2.\text{val} - T.\text{val} \\
E.\text{val} & := T.\text{val} \\
T_1.\text{val} & := T_2.\text{val} * F.\text{val} \\
T_1.\text{val} & := T_2.\text{val}/F.\text{val} \\
T.\text{val} & := F.\text{val} \\
F_1.\text{val} & := -F_2.\text{val} \\
F.\text{val} & := E.\text{val} \\
F.\text{val} & := \text{const.\text{val}}
\end{align*}
\]
Evaluation of the attributes is called the decoration of the parse tree. Imagine we have the string \((1+3)*2\). The parse tree is shown here.

The val attribute of each symbol is shown beside it. Attribute flow is upward in this case.

The val of the overall expression is the val of the root.
Each grammar production $A \rightarrow \omega$ is associated with a set of semantic rules of the form

$$b := f(c_1, c_2, \ldots, c_k)$$

- If $b$ is an attribute associated with $A$, it is called a synthesized attribute.
- If $b$ is an attribute associated with a grammar symbol on the right side of the production (that is, in $\omega$) then $b$ is called an inherited attribute.
Synthesized attributes of a node hold values that are computed from attribute values of the child nodes in the parse tree and therefore information flows upwards.

production

\[ E_1 \rightarrow E_2 + T \]

semantic rule

\[ E_1.val := E_2.val + T.val \]
Inherited attributes of child nodes are set by the parent node or sibling nodes and therefore information flows \textit{downwards}. Consider the following attribute grammar.

\[
\begin{align*}
D &\rightarrow T L \\
T &\rightarrow \text{int} \\
T &\rightarrow \text{real} \\
L &\rightarrow L_1, id \\
L &\rightarrow id
\end{align*}
\]

\[
\begin{align*}
L.\text{in} &= T.\text{type} \\
T.\text{type} &= \text{integer} \\
T.\text{type} &= \text{real} \\
L_1.\text{in} &= L.\text{in}, \text{addtype}(\text{id}.\text{entry}, L.\text{in}) \\
\text{addtype}(\text{id}.\text{entry}, L.\text{in})
\end{align*}
\]

\text{real id1, id2, id3}
In the same way that a context-free grammar does not indicate how a string should be parsed, an attribute grammar does not specify how the attribute rules should be applied. It merely defines the set of valid decorated parse trees, not how they are constructed.

An attribute flow algorithm propagates attribute values through the parse tree by traversing the tree according to the set (write) and use (read) dependencies (an attribute must be set before it is used).
A grammar is called S-attributed if all attributes are synthesized.

A grammar is called L-attributed if the parse tree traversal to update attribute values is always left-to-right and depth-first.

- For a production $A \rightarrow X_1 X_2 X_3 \ldots X_n$
  - The attributes of $X_j$ (1 <= j <= n) only depend on:
    - The attributes of $X_1 \ X_2 \ X_3 \ldots X_{j-1}$.
    - The inherited attributes of A.

Values of inherited attributes must be passed down to children from left to right.

Semantic rules can be applied immediately during parsing and parse trees do not need to be kept in memory. This is an essential grammar property for a one-pass compiler.

An S-attributed grammar is a special case of an L-attributed grammar.