CS143
Compilers

Lecture 3

July 9, 2001

Syntax Analysis

- The specification of a source language includes rules that prescribe the syntactic structure that programs must follow. Often, this structure is hiearchical: programs consist of blocks (which may be functions or class definitions), which in turn consist of statements, which include expressions, and so on.
- Syntax is usually described using Backus-Naur Form (BNF) or some variation thereof. From such a description, it is straightforward to write a context-free grammar, which serves as the roadmap for the process of syntax analysis, or parsing.

The Role of the Parser

- The parser obtains a string of tokens from the lexical analyzer and verifies that this string of tokens forms a syntactically valid source program.
- It must report any syntax errors in the source program and provide sufficient information about the error, such as its location, so that it can easily be fixed
- It should recover from any errors and continue parsing so that other errors can be detected.

Types of Parsing Methods

- Universal parsing methods, such as the Cocke-Younger-Kasami algorithm and Earley's algorithm, can parse any grammar, but are too inefficient for use in compilers
- Top-down methods are the easiest to use when building a parser by hand, but work for a limited subclass of grammars.
- Bottom-up methods are implemented by parser generators, and work for a larger subclass of grammars than top-down methods.

Syntax Error Handling

- Due to the hierarchical nature of syntactic rules, syntax errors can often be detected long after the actual location of the error in the input. This contrasts with lexical errors, which are always detected as soon as they occur.
- It is more difficult to recover from syntax errors than lexical errors. Once an error is detected, the failure to properly recover produces "spurious" errors, which were not made by the programmer. It is helpful to consider the most common errors in developing recovery strategies.

Error-Recovery Strategies

- Panic mode recovery, one of the simplest methods, involves discarding tokens after an error is detected, until a "synchronizing token" is found that will allow the parser to continue.
- Phrase-level recovery, in which the input is "corrected" to allow parsing to continue. This does not work well if the error is detected after it occurs.
- Error productions can be used to provide useful diagnostics for common errors
- Global correction provides a set of minimal changes to the input to produce a syntactically valid program, but this is too inefficient.

Context-Free Grammars

- Context-free grammars are a very useful notation for describing the language of all valid programs of a source language. It does so by means of syntactic rules, which often have a recursive nature. Such rules cannot be described using regular expressions.
- A context-free grammar consists of two types of symbols: *terminals*, which are synonymous with tokens, and *non-terminals*, which denote sets of strings useful in defining syntactic rules. One non-terminal, the *start symbol*, denotes the set of all valid strings in the langauge denoted by the grammar.

Productions of a Grammar

- The actual syntactic rules are called the *productions* of the grammar.
- A production consists of a left side and a right side. Each side is a string of grammar symbols.
- The interpretation of the production is that the left side may be defined in terms of the right side in a syntactically valid string of grammar symbols.
- In a context-free grammar, the left side must be a single non-terminal. Context-sensitive grammars do not include this restriction.

Notational Conventions

- Lower-case letters will be used to denote terminal symbols.
- Upper-case letters will be used to denote nonterminals.
- The start symbol will be denoted by the letter S.
- Greek letters will be used to represent strings of grammar symbols, including terminals or nonterminals.

Derivations

- Given a grammar G, a *derivation* is a transformation of a string of grammar symbols α in which nonterminals in α are replaced according to the productions of G.
- α *derives* β *in one step* if β can be obtained from α by replacing a nonterminal A in α by a string γ, where $A \rightarrow \gamma$ is a production of G. We write $\alpha \Rightarrow \beta$.
- We say that α *derives* β if there exist strings $\alpha_1, \alpha_2, \ldots, \alpha_n$ such that $\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow \beta$.
- The sequence of replacements used to obtain β from α is called a derivation of β from α.

Notational Shorthands

We say that α *derives* β *in zero or more steps* (written as $\alpha \Rightarrow^* \beta$) if:

- α and β are the same string, or
- $\alpha \Rightarrow^* \gamma$ and $\gamma \Rightarrow \beta$ for some string γ. In this case, we can also say that α *derives* β *in one or more steps*, and write $\alpha \Rightarrow^+ \beta$.

The Language Generated by a Grammar

- Given a grammar G with start symbol S, the *language generated by G*, denoted $L(G)$, is the set of all strings of terminal symbols that can be derived S.
- Any string in $L(G)$ is called a sentence of G.
- If two grammars generate the same language, they are said to be *equivalent*. Often a grammar must be converted into an equivalent grammar so that a parser may be more easily implemented.

Types of Derivations

- A leftmost derivation is a derivation in which the leftmost nonterminal is replaced during each step.
- Similarly, a rightmost derivation is a derivation in which the rightmost nonterminal is replaced during each step.
- We will see that two classes of parsing methods, top-down and bottom-up, are based on leftmost and rightmost derivations, respectively.

Parse Trees and Derivations

- A derivation may be represented graphically by a parse tree.
- Suppose that $A \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$, where A is a nonterminal. The parse tree for this derivation is constructed inductively as follows:
  - Initially, the tree consists of a single root node representing A.
  - Each step of the derivation consists of the replacement of a nonterminal X by some string α. We locate X among the leaves of the tree and create child nodes for each symbol of α, with X's node as the parent node.

Ambiguity

- A grammar that produces more than one parse tree for some sentence is *ambiguous*.
- It can be difficult to implement a parser for an ambiguous grammar, since parsing is essentially the task of building a parse tree from the leaves to the root, where each leaf is a symbol from the input. If more than one parse tree exists for the input, we will not know how to proceed with the construction of the tree.
- A grammar may be converted to an equivalent unambiguous grammar, or the ambiguity can be resolved using *disambiguating rules*.

Writing a Grammar

- Given some specification for the syntax of a source language, the first step in building a parser for the language is writing a grammar.
- When constructing a grammar, it may be desirable to use regular expressions to describe portions of the syntax, thus deferring these portions to the lexical analysis phase.
- The grammar should be written in such a way that a parser may easily be implemented. It must be possible to generate a unique parse tree for any input sentence.

Regular Expressions vs. Context-Free Grammars

Although context-free grammars can be used to describe tokens, this is not practical because:

- Regular expressions are more concise and easier to understand than grammars.
- Lexical analyzers constructed from regular expressions tend to be more efficient.
- Separating lexical rules from the remainder of the syntax provides a convenient method of isolating lexical issues within a single component.

Verifying the Language Generated by a Grammar

- Once a grammar G has been written for a source language L, it is important to be certain that the language generated by G is the set of all valid programs written in L.
- This verification consists of two basic steps:
  - Any input string w from the L must be in the language generated by G
  - Any string in the language generated by G must be a valid program written in L.

Eliminating Ambiguity

In some cases, an ambiguous grammar can be modified to produce an equivalent unambiguous grammar. One example of this is the *dangling-else* grammar:

$S \rightarrow iEtS \mid iEtSeS \mid O$

In this grammar, S represents a statement, i is the keyword "if", t is the keyword "then", e is the keyword "else", E is an expression, and O is any other type of statement.

Elimination of Left Recusion

- A grammar is said to be *left recursive* if, for some nonterminal A, $A \rightarrow A\alpha$ for some string α.
- Left-recursive grammars cannot be handled by top-down parsers because they cause infinite recursion.
- By examining non-left-recusive productions in the grammar, an equivalent grammar without left recusion may be constructed.

Immediate Left Recusion

- The grammar $A \rightarrow A\alpha \mid \beta$ contains immediate left-recursion, as A derives $A\alpha$ in one step.
- This grammar generates the language consisting of all strings than can be derived from strings of the form $\beta\alpha^*$, i.e. all strings consisting of β followed by zero or more occurrences of α.
- This same language can be generated by the equivalent productions that are not left-recursive:

$A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \varepsilon$

Algorithm for Removing all Left Recursion

Arrange the non-terminals in some order A_1, A_2, ..., A_n

For each non-terminal A

  For each non-terminal B preceding A

    For each production $A \rightarrow B\gamma$,

      For each production $B \rightarrow \delta$,

        Add the production $A \rightarrow \delta\gamma$

      Remove the production $A \rightarrow B\gamma$

  Eliminate immediate left-recursion involving A

Analysis

- The preceding algorithm eliminates left-recursion in any grammar provided that the grammar has no cycles (derivations of the form $A \Rightarrow^+ A$) and no ε-productions (productions of the form $A \rightarrow \varepsilon$).
- After using this algorithm, any productions of the form $A_i \rightarrow A_j\beta$ must have $j > i$, thus eliminating the possibility of left recusion.

Left Factoring

- A common parsing method is predictive parsing, which tries to determine steps in building the parse tree based on looking ahead by only one input symbol.
- This method cannot be used on a grammar for which two productions having the same left side share a common prefix.
- A technique called left factoring can be used to eliminate such productions for a grammar.

Algorithm for Left Factoring

Repeat the following:
For each nonterminal A
Find the longest common prefix α common to two or more productions having A for the left side
If α is not ε
Add the production $A \rightarrow \alpha A'$
For each production of the form $A \rightarrow \alpha B$,
Replace with the productions $A' \rightarrow B$
Until no more changes can be made

Non-Context-Free Languages

- Many languages are not context-free, and therefore cannot be described by a context-free grammar.
- For example, programming languages that require identifiers to be declared before they are used are not context-free. Also, languages that require the number and type of arguments in a function call to match the function's signature are not context-free.
- For this reason, enforcing rules such as these are deferred to the semantic analysis phase.

Top-Down Parsing

- Top-down parsing is the process of building the parse tree for a given sentence from the root of the tree to the leaves, in preorder.
- This is equivalent to computing a leftmost derivation of the input string from the start symbol of the grammar.
- We will examine top-down methods that do not backtrack through the input. Backtracking methods are rarely needed to parse according to the syntax of common programming languages.

Recursive-Descent Parsing

- Top-down parsing is often implemented using a recursive-descent parser.
- A recursive-descent parser may be implemented by writing a function for each nonterminal A. Such a function determines which production $A \rightarrow \alpha$ matches the input string. Since the right side may include A, such parsers often use recursion.
- A recursive-descent parser may backtrack if the grammar has not been left-factored. If there is left-recursion, the parser will enter an infinite loop.

Predicitive Parsers

- A predictive parser is a recursive-descent parser that requires no backtracking. It is able to construct the parse tree by looking only at the next input symbol.
- Predictive parsers only work for grammars that have been left-factored, have no left recursion, and are not ambiguous.
- In the function corresponding to each non-terminal A, the next input symbol determines which production of the form $A \rightarrow \alpha$ matches the input.

Transition Diagrams for Predictive Parsers

- A predictive parser may be designed by creating a transition diagram corresponding to the grammar.
- For each nonterminal A, create a transition diagram as follows: for each production $A \rightarrow X_1 X_2 \ldots X_n$, where each X_i is a grammar symbol, create a path from the initial state to a single accepting state with edges labeled with X_1, X_2, …, X_n.
- If the transition diagram for each nonterminal is deterministic, then the grammar is suitable for predictive parsing.

Nonrecursive Predictive Parsing

- A nonrecursive predictive parser may be built by using an explicit stack, rather than an implicit stack of function calls.
- Such a parser uses a two-dimensional table M indexed by the nonterminals and terminals of the grammar. The entry $M[X,a]$ is a production of the form $X \rightarrow a$.
- The input buffer contains the input string followed by the end-of-input marker, $
- Initially, the stack contains $ and the start symbol of the grammar on top.

repeat
  let X be the top stack symbol
  let a be the current input symbol
  if X is a terminal or $
    if $X = a$, pop X and advance to the next input symbol
    otherwise, report a syntax error
  else if $M[X,a] = X \rightarrow Y_1 Y_2 \ldots Y_k$
    pop X
    push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack in order
  else
    report a syntax error
until X = $

Predictive Parsing Tables

- The preceding algorithm for nonrecursive predictive parsing makes use of a parsing table that indicates how the parse tree is to be constructed.
- When trying to recognize a nonterminal symbol X, with a equal to the next input symbol, the table entry $M[X,a]$ indicates which production of the form $X \rightarrow \alpha$ matches the input.
- If the entry $M[X,a]$ is undefined, then a syntax error occurs whenever a appears in the input when trying to recognize X.
- We now outline a systematic method of constructing such a parsing table.

FIRST and FOLLOW

- FIRST and FOLLOW are two functions that aid in constructing a predictive parsing table.
  - For any grammar symbol X, FIRST(X) is the set of all terminal symbols that can begin strings derived from X. If $X \rightarrow \varepsilon$, then ε is also in FIRST(X).
  - For any nonterminal A, FOLLOW(A) is the set of all terminal symbols that can follow A in a string of grammar symbols derived from the start symbol of the grammar, S. FOLLOW(A) may also include the end-of-input marker $ if A can be the rightmost symbol of a string derived from S.

Computing FIRST

for each terminal a, add a to FIRST(a)
for each production $A \rightarrow \varepsilon$, add ε to FIRST(A)
repeat
  for each production $A \rightarrow Y_1 Y_2 \ldots Y_k$
    for i=1,…,k,
      for each terminal a in FIRST(Y_i),
        add a to FIRST(A)
      if ε is not in FIRST(Y_i), break
    if, for each i, FIRST(Y_i) contains ε,
      for each terminal in FIRST(Y_i),
until no new symbols can be added to any FIRST set

Computing FOLLOW

add $ to FOLLOW(S)
for each production $A \rightarrow \alpha B \beta$
  for each terminal a in FIRST(β) except ε
    add a to FOLLOW(B)
  if FIRST(β) contains ε
    add FOLLOW(A) to FOLLOW(B)
for each production $A \rightarrow \alpha B$
  add FOLLOW(A) to FOLLOW(B)

Construction of Predictive Parsing Tables

- We are now ready to present a method for constructing a predictive parsing table, using the functions FIRST and FOLLOW.
- The following algorithm will add entries to the parsing table *M*.
- If, after executing the algorithm, any entry of *M* contains multiple productions, then the grammar is not suited for predictive parsing.
- Any table entry that is empty is used to detect a syntax error.

Algorithm

for each production $A \rightarrow \alpha$
  for each terminal a in FIRST(α),
    add $A \rightarrow \alpha$ to $M[A,a]$
  if ε is in FIRST(α)
    if \$ is in FOLLOW(A),
      add $A \rightarrow \alpha$ to $M[A,\$]$
    for each terminal b in FOLLOW(A),
      add $A \rightarrow \alpha$ to $M[A,b]$

LL(1) Grammars

- A grammar that is suitable for predictive parsing is said to be LL(1).
- An LL(1) grammar is a grammar *G* for which a unique parse tree can be constructed for any string in $L(G)$, in the following manner:
  - Scanning the input from left to right
  - Using a leftmost derivation
  - Using one symbol lookahead

When is a grammar LL(1)?

- To check whether a grammar *G* is LL(1), one can try to build a predictive parsing table for *G*. If any entry in the table contains multiple productions, the grammar is not LL(1).
- It can be shown that *G* is LL(1) if and only if for any two productions $A \rightarrow \alpha \mid \beta$ of *G*,
  - α and β cannot derive strings sharing a common prefix (I.e. G is left-factored)
  - At most one of α or β can derive ε
  - If $\beta \Rightarrow^* \varepsilon$, then α does not derive a string beginning with a terminal in FOLLOW(A).

Error Recovery in Predictive Parsing

- Once a syntax error is detected, one may use panic-mode recovery. This entails skipping input symbols until a token appears that belongs to a synchronizing set that will allow the parser to continue.
- Typically, a synchronizing set is assigned to each grammar symbol, and is used when an error is detected while that symbol is on top of the stack during parsing.
- The effectiveness of panic-mode recovery depends on judicious choices of symbols in synchronizing sets.
- The FIRST and FOLLOW functions can be used to construct synchronizing sets.

Synchronizing Sets

In panic-mode recover, synchronizing sets can be assigned to each grammar symbol. These sets can be constructed as follows:

- For a nonterminal A, the synchronizing set for A can include FOLLOW(A) or FIRST(A)
- If an expected terminal is not present, it can be implicitly added. Thus the synchronizing set for a token includes all other tokens.
- Synchronizing sets for lower-level constructs, such as expressions, can also include symbols that begin higher-level constructs, such as statements.
- If $A \Rightarrow^* \varepsilon$, then the production $A \rightarrow \varepsilon$ can be used as a default.

Phrase-level Recovery

- Phrase-level recover is an alternative approach to error recovery.
- It entails associating each blank entry in the parsing table with a pointer to an error-handling routine.
- Such a routine may manipulate the stack and display error messages.
- When using phrase-level recovery, one should ensure that the error handler does not cause an infinite loop.