1 Intro

- **Class 1, 8/22/2018**
- Handout 1 — My names
  - Mr. / Dr. / Professor / —
  - Raphael / Rafi / Refoyl
  - Finkel / Goldstein
- Extra 5-10 minutes on every class to correct for missed sessions?
- Plagiarism — read aloud from handout 1
- Assignments on web. The first is very easy, the rest not, so start immediately.
- E-mail list: cs541001@cs.uky.edu; instructor uses to reach students.
- All students have MultiLab accounts, although you may use any computer you like to do assignments.
- Textbook — It is essential that you read ahead.

2 Overview of compilers: Chapter 1

- A compiler language is an example of a **software tool**.
• The compiler’s job.

    | Program in source language | compiler | byte code | interpreter |
    |-----------------------------|----------|-----------|-------------|
    |                             |          | P-code    |             |
    |                             |          | machine-independent | just-in-time |
    |                             |          | resolved machine code | hardware |
    |                             |          | relocatable object code | executable image |
    |                             |          | machine-specific | libraries |

• Compiler outputs
  - Pure machine code: specific to a given architecture, no runtime linking. Example: Linux kernel.
  - Augmented machine code: specific to a given architecture and operating system. Example: C programs written for Linux, which may make OS calls.
  - Virtual machine code, interpreted or compiled on the fly during execution. Examples: Java (JVM), C# (.NET). Advantages: portability, code size Our assignments use this output type.

• Output representations
  - Assembler: good for cross-compilation; avoids having the compiler resolve all references. Modular compilation. Our assignments use this output format.
  - Relocatable binary: defers resolving external references. Modular compilation. Very common; used by Java and C.
  - Absolute binary: all references resolved.

• The organization of a compiler Figure 1.4 page 15

• Class 2, 8/24/2018
  - Scanner: reads the source program and constructs a stream of tokens, removing comments, and processing directives such as listing.
• Example: if (a < 39) { is an input string of characters. The associated output tokens are if:reserved, ( :symbol, a:identifier, < :operator, 39 :integer, ) :symbol, { :symbol.
• The scanner can discover and report errors, such as 39f.
• We describe tokens by regular expressions.
• We recognize tokens by using a deterministic finite automaton (DFA). That automaton is built for us by a scanner generator tool such as lex, flex, or jflex. Our assignments use jflex.

• Parser: reads the token stream and creates an abstract syntax tree (AST), verifying syntax and possibly repairing syntax errors.
• Example: given the tokens above, the tree fragment would be:

```
  Statement
    if
      Expression
        <
          Identifier
            a
          Integer
            39
    Statement
    Statement
```
• The parser can discover and report errors, such as ] instead of ) in the example.
• We describe the syntax by a context-free grammar (CFG).
• The table that drives the scanner is built for us by a parser generator tool such as yacc, bison, or javaCUP. Our assignments use javaCUP.

• Type checker: navigates through the AST and verifies that variables are declared and that types are used consistently.
  • For instance, if a in the example is not of a numeric type, the type checker can report an error.
  • It can also modify the AST, for instance, introducing type-conversion nodes, if, for instance, a is a short integer, in which case it might be converted to a regular integer.

• Translator: navigates through the AST and generates either an intermediate representation (IR) or some other representation
of executable code. Our IR will be assembler for Java bytecode.

- Optimizer: Analyzes the IR to improve the code. There are many forms of optimization, such as simplifying expressions, moving code, re-using values, eliminating trivial arithmetic, replacing sequences of instructions. We will not cover optimization in this class.
- Code generator: Maps the IR to target machine code. Our assignments use jasper to generate the target machine code: Java bytecode.

3 Programming language considerations

- Successful designers of programming languages often have strong backgrounds in constructing compilers. If it can’t be compiled, it’s not very useful.
- Many features of modern languages require special care.
  - passing by name (obsolete since Algol 60; requires thunks)
  - dynamic-sized arrays (requires runtime type descriptors)
  - nested name scopes (require static chains)
  - anonymous functions, first-class functions (as in Python, requiring closures)
  - multiple-yield iterators (as in Python, require special stack manipulation)
  - automatic reclamation of object store (requires garbage collection).

4 Computer architecture considerations

- How many registers? What operations use them? How many register classes?
- Some operations can be very expensive: virtual method dispatch, dynamic heap access, reflective programming, exceptions, threads
- The effect of memory architecture, such as paging and caches, is difficult to present to programmers but is significant.
5 Specialty compilers

- Debugging support, including participation in an integrated development environment (IDE).
- Highly optimizing compilers.
- Retargetable compilers.

6 The ac (adding calculator) language: Chapter 2

- Class 3, 8/27/2018
- This is a very simple language that lets us explore the components of a compiler.
- Components
  - Types: integer and float
  - Keywords: \texttt{f}, \texttt{i}, \texttt{p}
  - Variables: lowercase Roman single letters, excluding keywords
- Context-free grammar (CFG), expressed in Backus-Naur Form (BNF) \[\text{Figure 2.1 page 33}\]
- Parse tree for \[\text{Figure 2.4 page 37}\]

7 The scanner

- Translates a stream of characters (as above) into a stream of tokens.
- A token has a type (such as \texttt{operator} or \texttt{reserved}) and a semantic value (such as \texttt{plus} or \texttt{print}).
- It’s a matter of choice whether each operator has its own type, in which case there is no need for semantic values.
- Likewise, one can choose that reserved words each have their own type, or that they are of type \texttt{reserved} with a semantic value (their spelling), or that they are of type \texttt{id} with a semantic value.
- Hard-coded example \[\text{Figure 2.5 page 40}\] uses \texttt{peek()} and \texttt{advance()}
• Production-quality scanners are constructed automatically from regular expressions. We will discuss them in the next chapter.

• This parse requires that we specify the syntax of tokens.

8 Formal language hierarchy

<table>
<thead>
<tr>
<th>Language type</th>
<th>Formalism</th>
<th>Automaton</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular</td>
<td>Regular expressions</td>
<td>Finite-state automaton (FSA)</td>
</tr>
<tr>
<td>Context-free</td>
<td>CFG (like BNF)</td>
<td>Push-down automaton (PDA)</td>
</tr>
<tr>
<td>Context-sensitive</td>
<td>CSG</td>
<td>Linear-bounded automaton (LBA)</td>
</tr>
<tr>
<td>Type 0</td>
<td>various</td>
<td>Turing machine</td>
</tr>
</tbody>
</table>

9 The parser

• Class 4, 8/29/2018

• Translates a stream of tokens into an abstract syntax tree (AST)

• Simplest method is recursive descent. Each nonterminal has its own procedure. By looking ahead (using peek()), each procedure can decide which other procedures to call.

• Parsing statements in ac: Figure 2.7 page 42

• One needs to discover the predict sets for each alternative production that has the same left-hand side. For Stmt, the predict set for assignment is {id}.

• One needs to discover the follow sets for some productions that can derive λ in order to compute the predict set for their parent productions.

• Given the grammar in Figure 2.1 page 33, trace the parse of

f b i a a = 5 b = a + 3.2 p b

10 Abstract syntax trees

• Syntax analysis (scanning and parsing) cannot:
  • enforce strong typing constraints
• disambiguate the meaning of some constructs, like $x.y.z$ in Java, which might be package-class-field or variable-field-field or many other possibilities.
• determine the meaning of an overloaded operator.

• Instead of using the parse tree, we prefer a simpler abstraction of the parse tree: the **abstract syntax tree**.
  • It omits punctuation.
  • Declarations store the identifier and its type in a single node.
  • It represents the order of executable statements and expressions.
  • Assignment nodes have two children: the identifier (the left-hand side) and the expression (the right-hand side).
  • Binary operations have two children.
  • The **print** statement is a single node that includes the name of the identifier to be printed.
  • Compare [Figure 2.9 on page 44](#) with [Figure 2.4 on page 37](#).

11 Semantic analysis

• Construct a **symbol table** for declarations and name scopes. In our case, it can be very simple, see [Figure 2.11 on page 48](#).
• Enforce type consistency.
  • Walk the tree bottom-up, using **visitor methods** as shown in [Figure 2.12 on page 49](#).
  • Insert to and query the symbol table as necessary.
  • Add a **type** field to nodes on the fly.
  • Modify the tree to introduce type conversion (in our case, **widening**) nodes.

12 Generating code

• In our case, the code is calculator buttons.
  • The calculator has registers; each is a single letter, such as $a$.
  • One can load or store a register with the $1$ and $s$ buttons.
• One sets the precision with the \( k \) button.
• One prints with the \( p \) button.

• We visit the AST with a visitor pattern to generate code, invoking \( \text{visit()} \) at each node. [Figure 2.14 on page 52]

13 Overview of scanner: Chapter 3

• This chapter introduces a formal, systematic approach to building scanners, instead of the hard-coded version of Chapter 2.

• Short story: tokens are defined by regular expressions, which are encoded into a non-deterministic finite automaton (NDFA), which can be automatically converted to a deterministic finite automaton (DFA), which can be described as a table of \( \text{state} \times \text{input} \rightarrow \text{action} \times \text{state} \), which can be executed by a simple program.

• Shorter story: write a set of regular expressions and let a scanner generator do the rest of the work. This method is an example of declarative programming.

• There are some complexities.
  • Escaped double-quote within a string literal.
  • Over-eagerness leading to error, such as 3..4 in Pascal, or 'a' in Ada, or \text{DO 200 I = 1.10} in Fortran
  • The scanner needs to be very fast. Even so, scanning tends to be the most time-consuming step of compilation, partly because of the cost of reading the source code (with all its inclusions).

14 Regular expressions

• This material should be a review.

• A regular expression defines a language, which is a set (possibly infinite) of strings over some alphabet \( \Sigma \).
• A regular expression is built recursively on the following components.
  • ∅.
  • λ.
  • individual letters in Σ. Example: a
  • the concatenation of regular expressions. The concatenation operator is usually omitted. Example: abca.
  • the alternation of regular expressions. The alternation operator is written |.
  • closure operations: the Kleene closure * and the positive closure +.
  • parentheses for grouping.
  • If you want to use a metacharacter such as |, *, +, (, ) in a regular expression, use some sort of escape character (typically \) before it.

• A regular expression generates a set of strings.
  • ∅ generates no strings at all.
  • λ generates the empty string.
  • An individual letter generates the string containing just that letter.
  • The concatenation of two regular expressions A and B generates all two-part strings, whose first part is a string generated by A and whose second part is a string generated by B.
  • The alternation of two regular expressions A and B generates all strings generated by A and all strings generated by B.
  • The expression A* generates the empty string and (recursively) all strings generated by AA*. The expression A+ generates all strings generated by AA*.

• Useful facts
  • The set of strings generated by a regular expression is called a regular set. Every regular set can be generated by some regular expression.
  • Every finite set of strings is a regular set. At worst, one can just build a regular expression that enumerates them with alternations.
• Any regular set has multiple regular expressions that generate it. For instance, \((ab)^*\) can also be written \(\lambda|ab|abab(ab)^*\).

• Notations

  • If \(A\) is a set of characters, we use \(\text{not}(A)\) to denote \(\Sigma - A\), the characters not in \(A\).
  • If \(S\) is a set of strings, we use \(\text{not}(S)\) to denote all (finite) strings except those in \(S\). It turns out that if \(S\) is a regular set, so is \(\text{not}(S)\).
  • If \(k \geq 0\) is a constant integer and \(S\) is a set of strings, then \(S^k\) is the set of strings formed by concatenating \(k\) strings (possibly different) from \(S\). If \(S\) is a regular set, so is \(S^k\).

15 Useful examples

• a Java comment that goes to the end of the line: 
  
  ```java
  // (not(←))∗←
  ```

  (Here, \(\Sigma\) is the set of all 16-bit Unicode characters and \(←\) is a line separator, which is platform-dependent.)

• a decimal literal: \(D^+.D^+\) where \(D\) is shorthand for \(0123456789\).

• an integer literal, optionally signed: \((+|−|\lambda)D^+\).

• a comment delimited by ## markers: 
  
  ```
  ##((#|\lambda)not(#))^∗##
  ```

• a Fortran-like real literal, which requires digits only on one side (either one) of the decimal point: \((D^+.D^+)|(0.D^+)\)

• an identifier, with underscores, but not adjacent, frontal, or terminal ones: \(L(L|D)^*(\sim(L|D)^+)\)*, or (Daniel Michler) \(L((\sim|\lambda)(L|D))^*)\).

16 Hashing

• Class 6, 9/5/2018

• Very popular data structure for searching.

• Cost of insertion and of search is \(O(\log n)\), but only because \(n\) distinct values must be \(\log n\) bits long, and we need to look at the entire key. If we consider looking at the key to be \(O(1)\), then hashing is expected to be \(O(1)\).

• Idea: find the value associated with key \(k\) at \(A[h(k)]\), where
• $h()$ maps keys to integers in $0..s-1$, where $s$ is the size of $A[]$.
• $h()$ is “fast”. (It generally needs to look at all of $k$, though.)

• Example
  • $k =$ student in class.
  • $h(k) =$ $k$’s birthday (a value from $0 .. 365$).

• Difficulty: collisions
  • Birthday paradox: \[
    \text{Prob(no collisions with } j \text{ people)} = \frac{365!}{(365-j)!365^j}
  \]
  • This probability goes below $\frac{1}{2}$ at $j = 23$.
  • At $j = 50$, the probability is 0.029.

• Moral: One cannot in general avoid collisions. One has to deal with them.

• A good hash function

  • Desiderata
    • Uniform: Equally likely to give any value in $0..s-1$.
    • Spreading: similar inputs $\rightarrow$ dissimilar outputs, to prevent clustering. Only important for open-addressing conflict resolution.
    • Fast.

  • Several suggestions, assuming that $k$ is a multi-word data structure, such as a string.
    • Add (or multiply) all the words of $k$, discarding overflow, then mod by $s$. It helps if $s = 2^j$, because mod is then masking with $2^j - 1$.
    • $\text{XOR}$ the words of $k$, shifting left by 1 after each, followed by mod $s$.

  • Wisdom: it doesn’t make much difference what hash function you choose. It is not even necessary to look at all of $k$. Just make sure that $h(k)$ is not constant (except for testing collision resolution).

• Dealing with collisions: open addressing

  • Overview
• The following methods store all items in A[] and use a probe sequence. If the desired position is occupied, some other position is open to consider instead.
• They tend to suffer from clustering.
• Deletion is hard, because removing an element can damage unrelated searches. Deletion by marking is the only reasonable approach.

• Perfect hashing: if you know all n values in advance, you can look for a non-colliding hash function h. Finding such a function is in general quite difficult, but compiler writers do sometimes use perfect hashing to detect keywords in the language (like begin and for).

• Additional hash functions. Have a series of hash functions, h1(), h2(),……
  • insertion: key probing with different functions until an empty slot is found.
  • searching: probe with different functions until you find the key (success) or an empty slot (failure).
  • You need a family of independent hash functions.
  • The method is very expensive when A[] is almost full.

• Linear probing. Probe p is at h(k) + p (mod s), for p = 0, 1, ……
  • Terrible behavior when A[] is almost full, because chains coalesce. This problem is called “primary clustering”.

• Quadratic probing. Probe p is at h(k) + p2 (mod s), for p = 0, 1, ……
  • When does this sequence hit all of A[]? Certainly it does if s is prime.
  • We still suffer “secondary clustering”: if two keys have the same hash value, then the sequence of probes is the same for both.

• Add-the-hash rehash. Probe p is at (p + 1) · h(k) (mod s).
  • This method avoids clustering.
  • Warning: h(k) must never be 0.

• Double hashing. Use two has functions, h1() and h2(). Probe p is at h1(k) + p · h2(k).
  • This method avoids clustering.
• Warning: $h_2(k)$ must never be 0.

• Dealing with collisions: chaining
  
  • Each element in $A$ is a pointer, initially null, to a bucket, which is a linked list of nodes that hash to that element; each node contains $k$ and any other associated data.
  
  • insert: place $k$ at the head of $A[h(k)]$.
  
  • search: look through the list at $A[h(k)]$.
  
  • optimization: When you find, promote the node to the start of its list.

  • average list length is $s/n$. So if we set $s \approx n$ we expect about 1 element per list, although some may be longer, some empty.

  • Instead of lists, we can use something fancier (such as 2-3 trees), but it is generally better to use a larger $s$.

17 Finite-state automata

• A finite-state automaton (FSA) is an idealization of a very simple computer, composed of

  • A finite set of states, usually depicted by circles.
    
    • One of the states is called the start state. It can be depicted by a circle with an arrow from nowhere pointing to it.
    
    • One or more of the states are called final (or accepting) states. They are usually depicted by double circles.

  • A finite alphabet, denoted $\Sigma$. We’ll call the elements of the alphabet letters.

  • A set of transitions between states, usually depicted by labelled arrows. The labels are letters.

  • An FSA is equivalent to a language.

  • An FSA can recognize strings in its language. It starts at the start state, and every time it sees the next letter, it moves to the state pointed to by an arrow from the current state that is labelled with that letter. If no such arrow exists, the string is not in the language. If when the string is finished, the FSA is in a final state, the string is in the language. Otherwise it is not.
• An FSA can generate all the strings in a language by starting at the start state and outputting the label on each transition it takes, stopping at some final state.
• It turns out that the language recognized or accepted by an FSA is a regular language, and every regular language can be converted to an FSA that recognizes/accepts it.

• An FSA is deterministic if no transitions are labelled with λ (which allows you to move to the next state without consuming input) and if no state has multiple outgoing transitions labelled with the same letter (in which case you don’t know which state to go to next).

• Even stronger: Regular expressions are equivalent to deterministic FSAs. We will see later how to convert a regular expression to a deterministic FSA.

• Example: Figure 3.1 page 64
• Example: Figure 3.5 page 68

18 Implementing a deterministic FSA to recognize a language

• Class 7, 9/7/2018
• table-driven

• Construct a table. Every row is a state; one might label them (arbitrarily) 1, 2, …, with the start state labelled 1. Every column is a letter, so there are as many columns as the size of \( \Sigma \).
• For simplicity, add one extra state at the end, the error state, and for every (state,letter) pair for which there is no transition, add a transition to the error state.

• state ← 1;
  while (ch ← input.advance())
    state ← table[state, ch];
    success ← accepting(state)
• The table is usually built by an automated scanner generator.
explicit control: produced automatically or “hard-wired”. Advantage: easier to read and faster, but more effort to generate and debug. [Figure 3.4 page 68]

19 Transducers

- A transducer not only recognizes strings in a regular language but also outputs some semantic value of the strings (tokens) it recognizes.
- Each transition can be labelled with an action, which can be as simple as an output letter or as complex as a function to invoke on the letter that was just recognized. The function might generally be to append to a growing string, and then to post-process that string when the input is finished, if the FSA is now in a final state.

20 Lex: A scanner generator

- Input file: defines how tokens are to be scanned and how to process them.
- Output: a program (in C, or for jflex, in Java), which defines a subroutine yylex(). (For jflex: a class called Yylex with a method called yylex())
- One compiles and links this program with the rest of the components to make a functioning compiler.
- Lex takes care of low-level details: reading characters efficiently, matching them against token definitions.
- The best way to learn Lex is to use the examples for Project 2. We will follow examples in the book.
- The overall structure of a Lex input file is in [Figure 3.8 on page 71]. There are three sections separated by %%.
- The scanner and the parser need to share token codes (typically small integers). A standard way is to use a table generated by a parser generator. Lex typically uses y.tab.h, generated by yacc.
JFlex uses sym.java, generated by javaCUP, which we will see later.

- Figure 3.7 page 71 shows a trivial Lex definition for part of ac, namely, the reserved words.

21 Regular expressions in Lex

- Shorthand: character classes, like $D$ in our examples of regular expressions: DIGIT=[0-9].
- Bracket syntax for range literals: Figure 3.9 page 72 and Figure 3.10 page 73.
- Escape convention for metacharacters like \\.
- It is valid to quote characters or character strings with ", but it is not necessary to quote alphanumeric characters.
- Case is significant. Use [pP][rR][iI][nN][tT] or %ignorecase.
- One may use the usual metacharacters for regular expressions: *, +, |, and parentheses.
- The character ^ matches the beginning of an input line; $ matches the end of an input line.
- The postfix operator ? matches the previous expression 0 or 1 times. You can read it as “optionally”.
- See Figure 3.11 page 74 for fuller examples of regular expressions for ac tokens. The result that yylex() is meant to return is computed by the processing code embedded in braces. The opening brace must be on the same line as the regular expression. The rest of the code, which can comprise many statements, need not be on the same line; it finishes when a matching brace appears.
- Figure 3.12 page 74 shows the same thing with defined classes instead of range literals.
- When the scanner matches an expression, it executes the associated commands. The matched text is in a String variable yytext (Jflex: call yytext()). It is overwritten by the next token the scanner matches, so save it. Usually the processing code deals with the
contents of the matched string, so the rest of the scanner can ignore it.

- If two regular expressions overlap, \textit{Lex} returns the longest possible match; if both expressions match the same string, the earlier expression wins.

- One reasonable style is to have a catch-all expression at the end to match any invalid token.

- \textit{Lex} returns a predefined end-of-file token (integer 0) when it reaches the end of the input.

- The subroutine section of the \textit{Lex} input file can define data structures and routines that the processing code can call.

- To avoid situations where the scanner has to back up (such as Pascal’s 0..4), one can specify \textbf{right context}: \([0-9]+/\ldots\) means “match a string of digits, but only if looking ahead one sees two dots in a row”. This expression is longer than \([0..9]\) so it will win if there are two dots, but it won’t match if there is only one dot. The right context portion is not consumed by this match.

- Standard symbols for \textit{Lex}: \[\text{book Figure 3.13 page 78}\]

- There are alternatives to \textit{lex}: \textit{flex} (faster, GPL), \textit{jlex} and \textit{jflex} (for Java), \textit{GLA} and \textit{re2c} (generate a directly executable, not table-driven, scanner in C).

\section*{22 Practical considerations}

- \textbf{identifiers}

  - If the language is not block-structured, the scanner can enter identifiers into the symbol table and return a pointer to the symbol-table entry.

  - The scanner can copy the identifier into \textbf{string space} and return a pointer to that space. The parser can decide that the string is a duplicate and reclaim the space of the most recent string.

  \[\text{Class 8, 9/12/2018}\]

  - The scanner can copy the identifier into an identifier table (a hash table) and return an associated integer that can be used as a key.
• If case is significant, a word like `WHILE` is most likely not reserved. If case is not significant, convert all words to a standard case before returning identifier tokens.

• literals
  - Convert numbers to internal representation. Use library routines (in C: `atoi();` in Java `Integer.parseInt()`). You might use higher-precision methods and compare against limits to detect range errors. You can also catch `NumberFormatException` in Java.
  - Convert string literals by expanding escaped characters.
  - There is a weird ambiguity in C: `x (* y)` is a declaration, not a procedure call, if `x` has been given a meaning via `typedef`. The parser might keep a table of `typedef` identifiers, and the scanner could return a different token for such identifiers.

• reserved words
  - One can catch reserved words by a regular expression, but that increases the size of the FSA. That’s how we’ll do it for CSX.
  - The scanner can have processing code for identifiers that looks them up in a reserved-word table, returning a special token for such identifiers.

• handling compiler directives
  - `file inclusion`. Keep a stack of open files, or recursively invoke the scanner.
  - `conditional compilation`. Usually handled as a separate pass before the scanner.
  - `Unicode escapes`, such as `\u05b3` in Java. Also a separate pre-scanning pass.
  - `listing`. It’s hard to properly intersperse compilation diagnostics.

• end of file
  - It can simplify the parser to continue to return the `EOF` token if the scanner is invoked after it produces an `EOF` token.

• multi-character lookahead
• example from Fortran: \texttt{DO 10 J = 1.100}
• general backup: buffer characters; if you enter an error state, back up until you reach an accepting state. If you back up to the start of the token, report a single-character error token and move past it.

• speed
  • Use a scanner generator; \textit{flex} or (better) \textit{GLA}.
  • Hard-coded: use block operations for read, double-buffering to handle tokens that cross block boundaries; use the buffer as the token store until you decide to copy them.
  • Use a profiling tool.

• error recovery
  • when you reach an error state, you can delete characters so far or just the first character.
  • in any case, you can return an “error token” that informs the parser that the subsequent token is unreliable.
  • runaway strings and comments: use special rules that detect, because ordinary error recovery (deleting first character) generates cascading errors. Nested comments cannot be handled by regular expressions.

23 Converting regular expressions to finite automata

• Convert the regular expression to a non-deterministic finite-state automaton (NDFA), using \( \lambda \) rules. [Figures 3.19-22 pages 93-94]

• Convert the NDFA to a deterministic finite-state automaton (DFA) by the \textbf{subset construction}.
  • Each state of the DFA corresponds to a set of states in the NDFA.
  • The start state of the DFA corresponds to the start state of the NDFA plus any NDFA states reachable by a \( \lambda \) transition from the start state.
• Each set of NDFA states reachable by an input string becomes a single DFA state.
• If any of the NDFA states in a DFA state accepts, then so does the DFA state.
• Iterate over all states in the DFA (this set grows) building new states.
• This method terminates, because at most there are $2^{|NDFA|}$ states in the DFA.
• See algorithm Figure 3.23 page 95
• Example: Figure 3.24 page 96

24 Optimizing the resulting DFA

• Optimization is only to reduce the number of states, and hence the table; it doesn’t change the speed.
• One should remove unreachable states and dead states (those from which one cannot reach an accepting state).
• All accepting states with no outgoing transitions are equivalent.
• Start by assuming all non-accepting states are equivalent.
• Repeatedly: if any character $c$ causes multiple transitions from a merged accepting state (possibly to a single “error state”), split that state based on the behavior of $c$.
• Example: Figure 3.26 page 98

25 Converting a FSA to a regular expression

• not needed for compiler construction, but helps prove the equivalence of these two formalisms.
• Assume start state has no incoming transitions and there is a single accepting state with no outgoing transitions; build it if needed.
• Apply three transformations Figure 3.30 page 101 repeatedly to remove states and add regular expressions.
• Example, Figure 3.25 page 97 leads to $b^*a (\lambda\,|\,b\,|\,ba\,|\,bb\,|\,a)$

26 Context-free grammars: Chapter 4

• A context-free grammar (CFG) represents a context-free language (CFL) (a set of strings).

• All regular languages are also context-free, but there are some context-free languages, such as the set of balanced parentheses, that are not regular.

• Instead of an FSA, one needs a push-down automaton (PDA) to recognize or generate strings of a CFL. We won’t be going into the theory of PDAs.

• Components of a CFG

  • a finite terminal alphabet $\Sigma$, composed of tokens, augmented with an EOF token. We’ll use lower-case words and punctuation symbols.

  • A finite nonterminal alphabet $N$, whose symbols are like variables in the grammar. We’ll use initial-capital words or single letters.

  • A start symbol $S \in N$.

  • A finite set of rewriting rules called productions of the form $A \rightarrow X_1 \ldots X_m$, where $A \in N$, $X_i \in N \cup \Sigma$. We allow $m = 0$, in which case we write $\lambda$ as the right-hand side.

  • We allow $|$ as a simplifying syntax if many rules share the same left-hand side.

• The CFG is a recipe for rewriting strings; each rewrite is a step in a derivation of the resulting string. We denote a step of a derivation with $\Rightarrow$. We denote possibly many steps with $\Rightarrow^*$.

• The strings we get, even if they still contain nonterminals, are called sentential forms.

• In a derivation, we may expand any nonterminal we wish in the next step, but there are two conventions.
• **leftmost derivation**: always expand the first nonterminal in the current sentential form. (How would we express this order in a direction-free way? Maybe “frontmost”.) Notation: $\Rightarrow_{lm}$.
  - For a leftmost derivation, we don’t need to mention which nonterminal we are expanding, only what rule we are using. Example: [Figure 4.1 page 116](#) and the derivation later on the page.
  - top-down parsers generate leftmost derivations; we say they produce a **leftmost parse**.

• **rightmost derivation**: expand the last nonterminal (“endmost”). Notation: $\Rightarrow_{rm}$. Also called **canonical derivation**.
  - bottom-up parsers generate rightmost derivations
  - Example: [book page 117](#)
  - The parser actually discovers this parse in reverse order.

### 27 Parse trees

- Describes a derivation.
- The root is $S$; each node is either a terminal, a nonterminal, or $\lambda$.
- Interior nodes are nonterminals; together with their children, they represent the application of a production.
- Both a leftmost and a rightmost derivation give rise to the same parse tree; the tree does not show the order of derivation.
- Given a sentential form, all its symbols descended from any single internal node is a **phrase** of that sentential form.

- A **simple phrase** contains no smaller phrase; its children are leaves (but leaves might not be terminals, because the sentential form might still have nonterminals).
- The **handle** of a sentential form is its leftmost simple phrase.
- Example: [Figure 4.2 page 118](#): sentential form $f \ (v \ \text{Tail})$, the simple phrases are $f$ and $v \ \text{Tail}$, and the handle is $f$.
- Example: sentential form $\text{Prefix} \ (v \ + \ v)$. The phrases are
  - $\text{Prefix} \ (v \ + \ v)$ (comes from $E$)
  - $v \ + \ v$ (comes from $E$)
• $+v$ (comes from Tail)
• $\lambda$ (simple; handle) (comes from Tail)

28 Grammar types

• A **regular grammar** has rules like a CFG, but the RHS is restricted to a single symbol from $\Sigma \cup \{\lambda\}$, followed optionally by a single nonterminal symbol. Regular languages are a proper subset of context-free languages.

• **Context-free grammars**, represented by BNF, can always be parsed in $O(n^3)$ time. Useful subclasses of CFGs can be parsed on $O(n)$ time.

• A **context-sensitive** grammar has rules like a CFG, but the LHS is allowed to have extra context both before and after the nonterminal. This context is preserved on the RHS. Context-free languages are a proper subset of context-sensitive languages.
  
  • We don’t use context-sensitive languages because parsing them is very expensive.
  • However, they would allow us to require that variables are declared before use as part of the grammar instead of a check performed after parsing.

• An **unrestricted grammar** (also called type-0) lets arbitrary patterns be rewritten.

29 Using CFGs for describing programming-language syntax

• **Class 11, 9/21/2018**

• We don’t want **unreduced** grammars, which have useless nonterminals that are never generated by any derivation of a string of terminals. Example: [book page 120](#). Parser generators usually verify that the grammar is reduced.

• We prefer **unambiguous** grammars, where every sentence has a unique parse tree. Example of ambiguous grammar:
Unfortunately, guaranteeing that a CFG is unambiguous is undecidable. We will return to this problem later.

- We don’t want CFGs that generate the “wrong” language. Deciding whether two CFGs generate the same language is undecidable.

### 30 Extended BNF

- Allow metacharacters ‘[’ and ‘]’ to surround optional symbols in the RHS.
- Allow metacharacters ‘{’ and ‘}’ to surround symbols that may be repeated 0 or more times. I like an extension to this notation that tells you what to place between each repeated symbol, typically a comma. Example:
  \[\text{Decl} \to \text{[final] [static] [const] Type id \{, id \}}\]

- It’s not hard to convert extended BNF into standard BNF. For every optional region, introduce a new nonterminal \( N \) with two rules, one expanding to the symbols in the region, the other to \( \lambda \). For every repetition region, introduce a new nonterminal \( M \) with two rules, one expanding to the symbols of the region followed by \( M \) itself, the other to \( \lambda \).

### 31 Recognizers and parsers

- A **recognizer** determines if a string is in a language.
- A **parser**, more useful to us, also builds the parse tree.
- A **top-down** parser builds a leftmost derivation, traversing the parse tree in preorder. These parsers look ahead in the input to predict the right production before applying it. Common strategy: LL(1).
- A **bottom-up** parser builds a rightmost derivation, starting at the leaves and working up to the root, traversing the tree in postorder. Children of nodes are inserted before the nodes themselves. Common strategy: LR(1).
- Extended examples: grammar parses on previous pages.
• The names LL(1) and LR(1): the first letter indicates that input is scanned left-to-right (start to finish). The second letter indicates a leftmost (L) or rightmost (R) derivation. The notation (1) means one-token lookahead.

32 Data structures to represent a CFG

• We assume some representation for sets (such as $\Sigma$ and $\mathcal{N}$), lists, and iterators over sets or lists. In Java, classes that implement the Collection interface allow a for loop like this:

```java
List<Symbol> symbList = new ArrayList<Symbol>();
...
for (Symbol oneSymb : symbList)
    System.out.println(oneSymb.name);
```

• There is also an Iterator interface with a clunkier API, including hasNext(), next(), and remove(). You must use this interface if you plan to remove elements while iterating through the set. You get an Iterator from a Collection by iterator():

```java
for (Iterator<Symbol> it = symbList.iterator();
    it.hasNext(); ) {
    Symbol oneSymb = it.next();
    System.out.println(oneSymb.name);
    if (someCondition(oneSymb)) it.remove();
}
```

• Our data structures may take advantage of these facts:

  • We won’t be removing symbols from the CFG (except in rare cases of reducing the grammar).
  • Removing […] and {...} adds symbols and productions to the grammar.
  • Given a nonterminal $A$, we will want to visit all rules with $A$ on the LHS. We will also want to visit all rules that mention $A$ in the RHS.
  • We will generally process an RHS one symbol at a time.

• Therefore, we represent a rule by its LHS symbol and a list of its RHS symbols. The list is empty if the RHS is $\lambda$.

• See page 128 for a list of utility routines we might want.
33 Computing when a nonterminal can derive $\lambda$

- A derivation to $\lambda$ may take more than one step.
- Algorithm [Figure 4.7 page 129].
  - Make a list $L$ of nonterminals that directly derive $\lambda$.
  - For each $N \in L$, find all productions where $N$ appears on the RHS; if removing $N$ from those productions leads to an empty RHS, add the LHS to $L$.
  - Keep a count of RHS lengths so the previous step can account for several nonterminals in the same RHS, all of which can derive $\lambda$.
- The result is two Boolean characteristics: $\text{ruleDerivesEmpty}(p)$ and $\text{symbolDerivesEmpty}(N)$.

34 Computing $\text{First}(\alpha)$

- We need $\text{First}(\alpha)$ in top-down parsers to decide which of several alternative rules to consider when we see a nonterminal $\alpha$.
- $\text{First}(\alpha) = \{ b \in \Sigma \mid \alpha \Rightarrow^* b\beta \}$. Here, $\alpha$ and $\beta$ are strings of symbols (terminals or nonterminals). We won’t include $\lambda$ in $\text{First}(\alpha)$.
- Algorithm [Figure 4.8 page 130].
  - Example: [Figure 4.1 page 116]
  - Consider first character of $\alpha$.
  - Easy cases: $\alpha$ is empty or terminal.
  - Hard case: nonterminal $N$.
    - For each RHS $R$ in productions with LHS=$N$, recursively call algorithm on $R$, collecting all answers.
    - If $\text{symbolDerivesEmpty}(N)$, union in recursive result on the remaining characters of $\alpha$.
  - Avoid endless recursion by refusing to consider the same nonterminal twice (Boolean $\text{visitedFirst}(N)$).
- Example: [Figure 4.10 page 133].
Computing **Follow(N)**

- We need **Follow(N)** as we generate LL parsers. It is the set of terminals that can come after nonterminal N in a sentential form.
- **Follow(N)** = \{ b ∈ Σ | S ⇒⁺ αNβ \}. Here, α and β are strings of symbols (terminals or nonterminals).
- **Algorithm** [[Figure 4.11 page 135]]
  - For each place N occurs in the RHS of some production p, union in **First(tail)**, where tail represents the remaining symbols in that RHS after N.
  - If tail can derive empty (for instance, tail is itself empty), then union in the recursive result of the **Follow(LHS(p))**.
  - Avoid endless recursion by refusing to consider the same nonterminal twice (Boolean **visitedFollow(N)**).
- **Example:** [[Figure 4.10 page 133]]
- **Example:** [[Exercise 10 page 140]]
  - can derive λ: DS, SS, E
  - **First(P)** = **First(DS)** ∪ **First(SS)** ∪ \{ $ \} = (later) \{ f, i, id, p, $ \}
  - **First(DS)** = **First(D)** = (later) \{ f, i \}
  - **First(SS)** = **First(S)** = (later) \{ id, p \}
  - **First(V)** = \{ id, num \}
  - **Follow(P)** = ∅
  - **Follow(DS)** = ∅ ∪ **First(SS)** \{ $ \} = **First(SS)** ∪ \{ $ \} = \{ id, p, $ \}
  - **Follow(D)** = **First(DS)** ∪ **Follow(DS)** = \{ f, i, id, p, $ \}
  - **Follow(SS)** = ∅ ∪ **First($)** = \{ $ \}
  - **Follow(S)** = **First(SS)** ∪ **Follow(SS)** = \{ id, p, $ \}
  - **Follow(E)** = ∅ ∪ **Follow(S)** = \{ id, p, $ \}
  - **Follow(V)** = **First(E)** ∪ **Follow(E)** ∪ **Follow(S)** = \{ +, −, id, p, $ \}
Chapter 5: Top-down (LL) parsing (overview)

- Not as powerful as bottom-up, but simple, fast, with good error diagnostics.
- If one can build a top-down parser, the grammar is not ambiguous, although in general ambiguity is not decidable.
- Two general approaches: recursive descent and table-driven.
- Names for this style of parsing: top-down, predictive, LL(k), recursive-descent.
- The recursive-descent parser
  - Each nonterminal $A$ has its own parsing procedure $\text{proc}_A$.
  - $\text{proc}_A$ chooses one of $A$’s rules by inspecting the next $k$ (at most) tokens.
  - The constant $k$ is the lookahead value. One chooses $k$ before building the parser.
  - We’ll use $k = 1$ for discussion.
  - Each of the rules for $A$ has its own predict set (computed below)
- Given token $t$ and all the predict sets for the current nonterminal $A$:
  - $t$ is in none of the predict sets: Syntax error. The parser could output a useful message (saw $t$, expecting ...).
  - $t$ is in more than one predict set: ambiguous case. The parser generator should prevent this situation.
  - $t$ is in exactly one predict set. Follow the production predicted.
- To compute the predict set for a production $p$
  - the predict set is at least $\text{First}$(RHS($p$)).
  - if $\text{ruleDerivesEmpty}(p)$, the predict set also has $\text{Follow}$(LHS($p$)).
- Example: Exercise 10 page 140
  - $\text{Predict}(P \rightarrow DS SS $) = \text{First}(DS SS $) = \{f, i, \text{id}, p, $\}$
  - $\text{Predict}(DS \rightarrow D DS) = \text{First}(D DS) = \{f, i\}$
  - $\text{Predict}(DS \rightarrow \lambda) = \emptyset \cup \text{Follow}(DS) = \{\text{id}, p, $\}$
  - $\text{Predict}(D \rightarrow f \text{id}) = \{f\}$
• \( \text{Predict}(D \rightarrow \text{id}) = \{\text{id}\} \)
• \( \text{Predict}(SS \rightarrow S SS) = \text{First}(S) = \{\text{id}, \text{p}\} \)
• \( \text{Predict}(SS \rightarrow \lambda) = \text{Follow}(SS) = \{\$\} \)
• \( \text{Predict}(S \rightarrow \text{id} = \text{V E}) = \{\text{id}\} \)
• \( \text{Predict}(S \rightarrow \text{p id}) = \{\text{p}\} \)
• \( \text{Predict}(E \rightarrow + \text{V E}) = \{+\} \)
• \( \text{Predict}(E \rightarrow - \text{V E}) = \{-\} \)
• \( \text{Predict}(E \rightarrow \lambda) = \text{Follow}(E) = \{\text{id, p, $}\} \)
• \( \text{Predict}(V \rightarrow \text{id}) = \{\text{id}\} \)
• \( \text{Predict}(V \rightarrow \text{num}) = \{\text{num}\} \)

• For the CFG to be LL(1), all predict sets for a nonterminal must be disjoint.
  • It is easy to detect violations automatically.
  • If the CFG is not LL(1), it might still be LL(\( k \)) for some \( k > 1 \).
  • Sometimes a more powerful parsing method, like LR(\( k \)), is needed.
  • Sometimes even that fails, for instance, if the CFG is ambiguous.

37 The recursive-descent LL(1) parser

• Uses \texttt{peek()}, \texttt{advance()}, and (based on those two) \texttt{match(token)}.
• For a nonterminal \( A \), procedure \texttt{procA()} peeks at the next token, using that value to switch among the competing productions.
• For each production \( p \), each symbol \( X \) in the RHS(\( p \)) becomes a call.
  • If \( X \) is terminal, call \texttt{match(X)}.
  • If \( X \) is a nonterminal, call \texttt{procX()}.

• Example: Figure 5.7 page 151

38 The table-driven LL(1) parser

• Class 14, 10/3/2018
• Why bother? Because the code for the recursive-descent parser can be fairly long. A table can be much smaller. I think we can ignore the cost of procedure call and return.

• Data structure: a stack, with operations \texttt{push(s)}, \texttt{pop()}, and \texttt{tos()}. We push terminals and nonterminals.

• Algorithm for the parser
  
  - if \texttt{tos()} shows a terminal \texttt{t}, call \texttt{match(t)}; \texttt{pop()}. If that terminal is not in the input, we have a syntax error.
  
  - if \texttt{tos()} shows a non-terminal \texttt{A}, look up the correct production \texttt{p} = \texttt{T[A, peek()]}.
  
  - if \texttt{p} = 0, syntax error.
  
  - otherwise, push \texttt{RHS(p)} onto the stack so the first element goes to the top of the stack.

• Algorithm to build the table \texttt{T}:
  
  - For each non-terminal \texttt{A}, for each terminal \texttt{t},
    
    - if \texttt{t} \in \texttt{Predict(p)}, place \texttt{T[A, t] = p}.
    
    - otherwise, place \texttt{T[A, t] = 0}.

• Example of table: Figure 5.10 page 154

• Example of parse: Figure 5.11 page 155

• It helps to map terminals and non-terminals to small integers for the purpose of looking them up in \texttt{T}.

39 Making a CFG LL(1)

• Common prefixes shared between two productions for the same non-terminal.
  
  - Increasing lookahead value \textit{k} doesn’t help. Figure 5.12 page 156
  
  - Easy to fix by \textbf{factoring}: leave the common part in a single production, and follow it by a new nonterminal that expands in several productions to the various sequelae. Figure 5.14 page 157
• One may need to run this algorithm several times, removing common prefixes longest-first.
• One can force all RHSs to begin with a terminal symbol (Greibach normal form), and then one can remove common prefixes, but the resulting grammar may still not be $LL(1)$.

• Left recursion, which causes a unbounded recursion (in recursive-descent parsing) or unbounded pushing the RHS onto the stack (in table-driven parsing).

• Easy to fix: if the production is $A \rightarrow A \alpha$, change it to $A \rightarrow X Y$.
• For every other production $A \rightarrow \beta$, change it to $X \rightarrow \beta$.
• Add production: $Y \rightarrow \alpha Y$ (right-recursive!)
• Add production: $Y \rightarrow \lambda$.
• Example: Figure 5.16 page 158

40 Review of algorithms

• Given the grammar Figure 5.2 page 148, for each nonterminal $N$, compute derives-$\lambda(N)$, $first(N)$ (in reverse order), $follow(N)$ (in forward order). Then for each production $p$ compute $predict(p)$. The answer is in Figure 5.3, page 148.

• Then derive procB:
  
  ```c
  void procB() {
      switch (peek()) {
          case 'b':
              match('a');
              procB();
              procC();
              match('d');
              return;
          case 'q'; case 'c'; case 'd'; case '$': // $ is EOF
              return;
          default:
              error();
          } // switch
    } // procB()
  ```
• Compute a parse table. Answer is in Figure 5.10 page 154

41 Compressing the parse table

• Most entries are blank, so direct representation is space-inefficient. Example (Ada): 6.5% density.
• Given a non-terminal, entries for many lookahead symbols agree. See Figure 5.10 page 154.
• Can store a default value for each terminal; for some terminals, it will be the empty (0) value. Then we only need to store non-default values.
• Compaction
  • Binary search: list all the non-default values in a one-dimensional table, along with their row, column indices. Search $T[i, j]$ using binary search on those indices. For $E$ non-default entries, the space cost is $3E$, and the time cost is $O(\log E)$.
  • Hash table: Search $T[i, j]$ at $h(i, j) = (i \times j) \mod (|E| + 1)$, resolving collisions by linear resolution. There is guaranteed to be an empty cell.
  • Perfect hashing is possible, since the values are known in advance.
• Compression by double-offset indexing
  • Compute non-conflicting shift values $R[i]$ for each row $i$ into a large table $V$.
  • Find $T[i, j]$ at $offset=V[R[i]+j]$, accepting it if $V.fromrow[offset] = i$.
  • Example: Figure 5.24 page 169
  • Finding the best ordering of rows to add to $V$ is NP-complete.
  • Even with an arbitrary ordering, this method works pretty well. Example (Ada): 6.8% resulting table size.
  • Time to look up an entry in the compressed table is $O(1)$, and its space is no worse than original $T$. 
42 Dangling-else problem

- The dangling-else problem cannot be solved with LL($k$) grammars.
  - Grammar: Figure 5.17 page 159
  - Equivalent to $\{a^jb^k \mid 0 \leq k \leq j\}$
  - So long as the parser sees only $a$s, it can’t decide whether to predict a matched or unmatched $a$. The amount of lookahead before the first $b$ can be arbitrarily large.
  - But this language is context-free, because it has a CFG.

- To handle this problem, accept the predict conflict. Factor the grammar Figure 5.19 page 161. Get the table shown in that figure. Resolve the conflict by always choosing rule 4.

43 Properties of LL($k$) parsers

- An LL($k$) parser computes a correct leftmost parse.
- Grammars in the LL($k$) class are unambiguous.
- Table-driven LL(1) parsers operate in $O(n)$ time and space, where $n$ is the length of the input (measured in tokens).
  - Time: each input token induces a bounded number of actions (no recursive use of the same production).
  - Space: the stack can be of length $O(n)$; each input token contributes to a constant increase in stack size (no recursive use of the same production).

44 Recovering from and repairing syntax errors

- Compilers usually disable semantic analysis and code generation if there are errors. Users won’t run the program in any case.
- Correct prefix property of LL($k$) parsing: until you hit an error, there is a string of upcoming tokens that can complete a successful parse.
- Recovery: try to enter a configuration where the parser can continue.
• Bad recovery causes a cascade of errors.
• **Panic mode**: skip input tokens until encountering a frequently occurring delimiter, like semicolon.
• Wirth’s method: pass a *termset* parameter to each procedure in the recursive-descent parser, indicating what lookahead tokens are possible for this invocation. All calls to `match()` are enhanced: If they fail, they print a message and then skip tokens until a member of the *termset* is seen. Example:

![Figure 5.26 page 172](#)

• **Repair**: Modify the input to obtain an acceptable parse.
• Given that one has parsed $\alpha$ and then the next token $x$ is erroneous, followed by $\beta$, one can try
  • modifying $\alpha$. Since we know that $\alpha$ is a correct prefix, we prefer not to modify it, except for **scope repair**, which inserts or deletes brackets to match the brackets in $\beta$.
  • inserting $\delta$ to obtain $\alpha\delta x\beta$. One must be careful not to keep inserting, or parsing never finishes. Some languages are **insert correctable**: one can always fix errors by insertion.
  • deleting $x$ to get $\alpha\beta$. Drastic, but guaranteed to make progress.

45 **Chapter 6: Bottom-up parsing**

• Bottom-up parsers handle the largest class of grammars that can be parsed deterministically.
• Just as with top-down parsing, there are parser generators that build the necessary tables. *javaCUP* and *Yacc* are examples.
• A bottom-up parser starts with leaves (terminals) and decides to **reduce** them to terminals.
• Also called a **shift-reduce** parser or an LR($k$) parser.
• Informal example: ![Figure 6.1 page 183](#)
• Algorithm: ![Figure 6.3 page 185](#)
• ![Class 16, 10/8/2018](#)
• Example: parse `abbdc$` using the table in ![Figure 6.5 page 187](#)
  Squares mean “shift”; no square means “reduce”.

46 LR(0) parsing

- Warning: there is actually an LR(0) parser, which uses one token of lookahead while parsing, but not while the table is built.

- Example based on Figure 6.2 page 184.

- The states in the parser are related to how far we have recognized a RHS. We call progress a **bookmark** and denote it by •.

- **Items** are production RHSs with • somewhere in them.

- A **fresh** item has • at the start of the RHS.

- If the • is at the end of the RHS, the item is **reducible**.

- If the RHS is λ, there is only one item, which is both fresh and reducible.

- See Figure 6.8 page 192 for items pertaining to one RHS.

- Each **parser state** is set of the items, called its **kernel**.

- The start state is composed of fresh items for goal productions.

- **Build the characteristic FSM**
  
  - work list starts with just the start state (state 0)
  
  - take the first state on the work list: $A \rightarrow a \cdot B\beta$

  - compute its **closure**, which includes fresh items with $B$ on the LHS. (if • is not before a nonterminal, the closure is empty), as well as fresh items with other nonterminals on the LHS for any production for $B$ whose RHS starts with a nonterminal, recursively.

  - for each element of the closure, we get a new state whose kernel is determined by advancing the •. Put that state on the work list. The current state points to it with an arrow labelled by what we advanced the • over (it can be a terminal or nonterminal). If advancing over the same symbol applies to more than one element of the closure, the resulting items are all part of the kernel of the subsequent state.

- Example: Figure 6.11 page 194

- We can use the characteristic FSM to parse; each accepting state causes a reduction, and we start anew at the start state.
• Example: Figure 6.12 page 195
• But we don’t need to start anew: We can remember the state we were in after shifting each symbol.

47 Conflicts

• The table may only suggest one course given a state and a lookahead: shift, reduce, error, or accept.
• We can’t handle nondeterminism; NPDA is more powerful than a PDA.
• If the table has multiple suggestions, the state is inadequate, but a stronger parsing method might work.
• shift-reduce conflict: the state contains at least two items, one of which is before a terminal $t$, and the other is reducible.
• reduce-reduce conflict: the state contains two reducible items.
• Example: Figure 6.16 page 200 This grammar is in fact ambiguous, so the inadequacy cannot be resolved without changing the grammar.