CSE 401 – Compilers

Lecture 4: Implementing Scanners
Michael Ringenburg
Winter 2013

Agenda

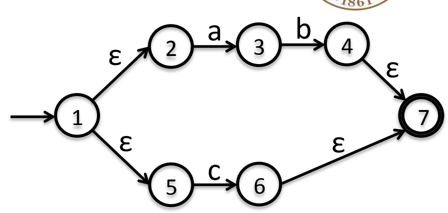
- Last week we covered regular expressions and finite automata.
- Today, we'll finish our final example (NFA to DFA conversion) and then talk about how scanners are implemented.
- Wednesday, we'll begin our discussion of parsing.

Announcements

- Part 1 of the project (the scanner) will be released tomorrow morning.
  - If you or your partner haven't emailed the course staff to let us know your team, do so TODAY.
  - If you haven't been able to find a partner, email me and I'll pair you up with someone else who hasn't.
    - You can also check the discussion board – there have been a few posts by people looking for partners.
  - We currently have an even number of students (54), so everyone should be able to have a partner.

Example

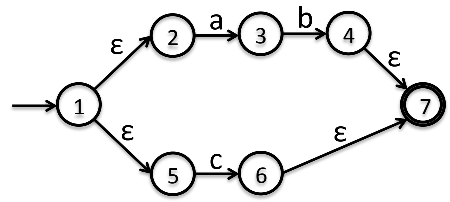
- Convert NFA to a DFA:



Step 1: Find ε closure of start state: {1,2,5}

Example

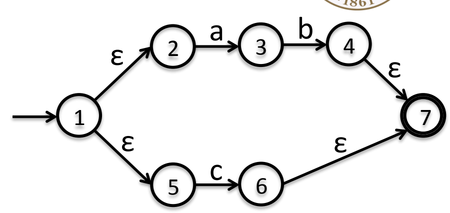
- Convert NFA to a DFA:

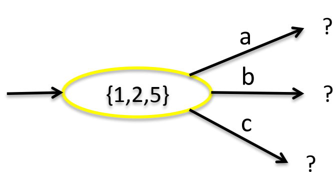


Step 2: Make a new DFA state corresponding to this ε closure. Mark it as unvisited (yellow in this diagram).

Example

- Convert NFA to a DFA:

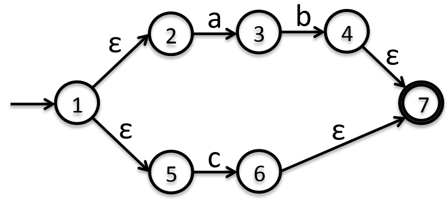


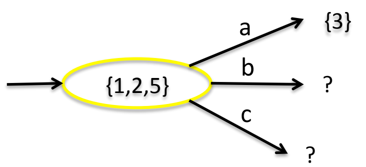


Loop: As long as there are unvisited DFA nodes, pick one. Consider transitions from its corresponding NFA states for every symbol in the alphabet.

Example

- Convert NFA to a DFA:

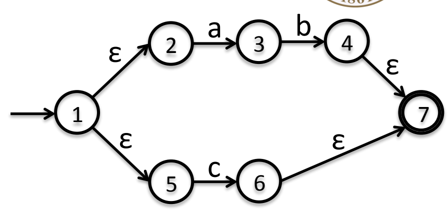


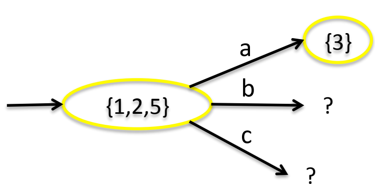


Only transition on ‘a’ from 1,2, or 5 is to 3.

Example

- Convert NFA to a DFA:

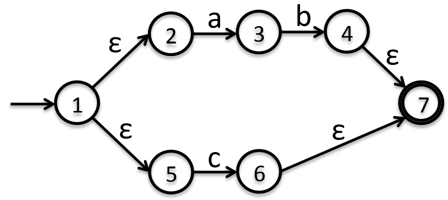


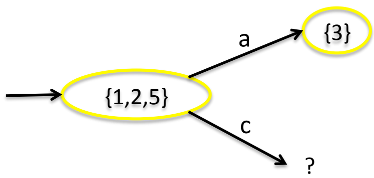


ε closure of {3} is just 3 (no ε transitions), so {1,2,5} transitions to {3} on ‘a’. This DFA state does not exist yet, so make it and mark it unvisited.

Example

- Convert NFA to a DFA:

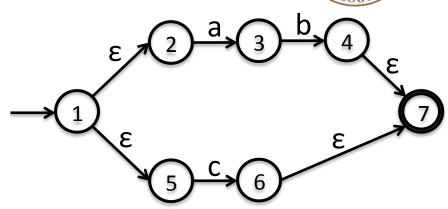


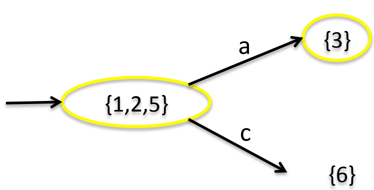


No transitions from 1, 2, or 5 on symbol 'b'.

Example

- Convert NFA to a DFA:

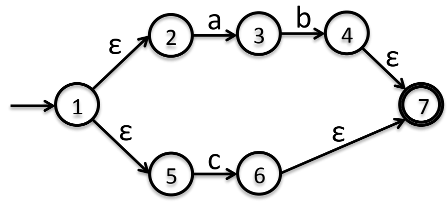


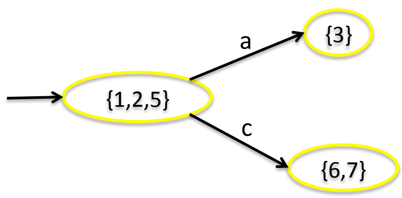


Only transition from 1, 2, or 5 on 'c' is to 6.

Example

- Convert NFA to a DFA:

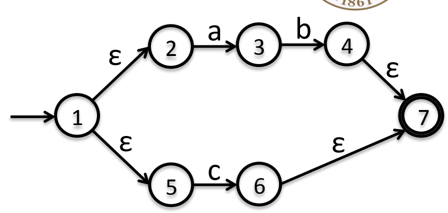


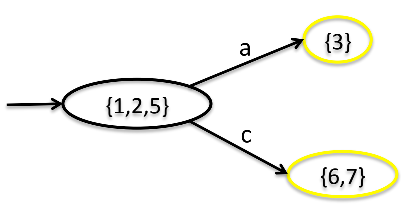


Epsilon closure of {6} is {6,7}, so {1,2,5} transitions to {6,7} on ‘c’. This doesn’t exist, so create and mark unvisited.

Example

- Convert NFA to a DFA:

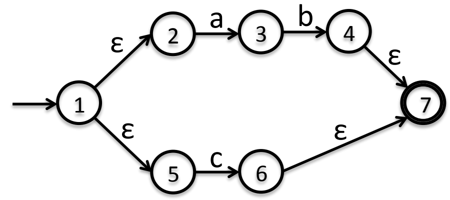


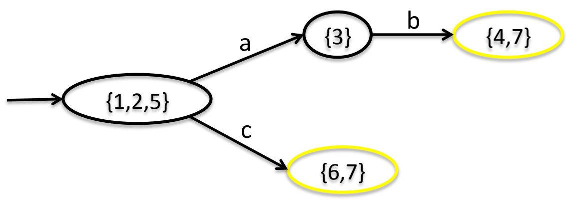


Done with {1,2,5}. Mark as visited (black in our diagram).

Example

- Convert NFA to a DFA:

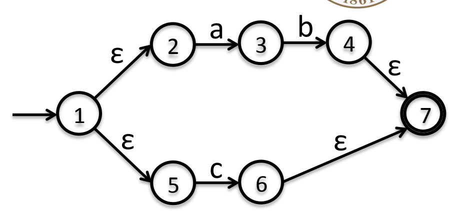


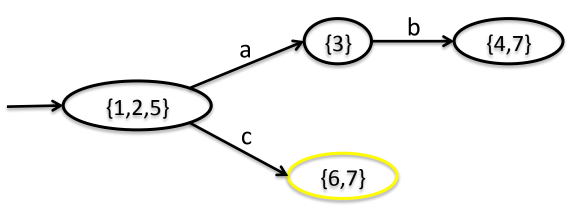


Repeat for another unvisited node ({3}). Creates {4,7}.

Example

- Convert NFA to a DFA:

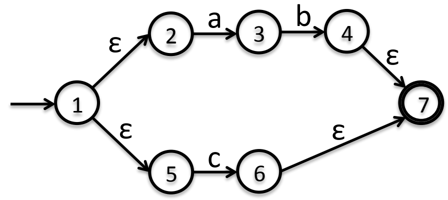


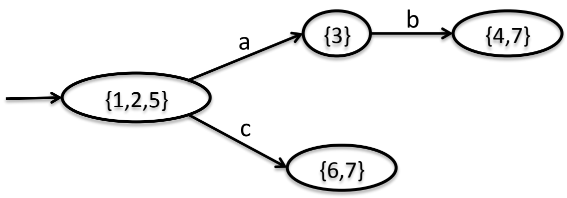


Repeat for unvisited node ({4,7}). No transitions.

Example

- Convert NFA to a DFA:

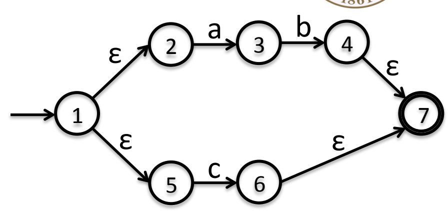


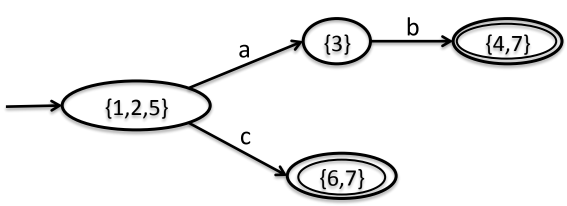


Repeat for unvisited node {6,7}. No transitions.

Example

- Convert NFA to a DFA:





No more unvisited nodes. Mark as final all states which include an NFA final state in their set.

Building a Scanner

- We've seen the theory (RE to NFA to DFA), but how is this converted to practice?
- A scanner needs to take an input stream and convert it to tokens.
  - Following the "longest match" principle – i.e., build the longest legal token starting at the current input position. Then repeat.
- General idea:
  - Create an RE for every token type. E.g., an RE for +, and RE for integers, etc.
  - Build a DFA for the union of the REs
  - Modify DFA implementation to recognize the longest matching substring (rather than only accepting the whole string).
    - This is sometimes free/unnecessary for certain DFAs
  - Repeatedly invoke (typically by the parser to obtain next token).

Scanning DFA

- How does this modified DFA work?
  - Must not just accept, but accept *and* tell us which RE generated the string (i.e., which token we found).
  - Identify the token by the final state we end in.
    - What if our DFA final state corresponds to multiple REs from the original?
    - This can happen if text matches *multiple tokens*. E.g., "for" may match the `for` keyword RE and the identifier RE. Compiler writer must define priority order (e.g., keywords > IDs).
  - Must also find longest match – may get this for free…
    - If needed, run DFA until no more transitions. If not in a final state, backtrack to last seen final state. Not always necessary.

Putting it together

- A scanner is a DFA that finds the next token each time it is called (and advances the input pointer to the token's end).
- Every "final" state of a DFA emits (returns) a token.
- For example:
  - **==** becomes <equal> (not <assign> <assign>)
  - **(** becomes <leftParen>
  - **private** becomes <private>
- Compiler writer (you!) choose the token names
- Also, there may be additional data associated with tokens … `\r\n` might count lines; all tokens might include line #; integer literals include value; etc.

DFA => Code, by Hand

- Option 1: One procedure per DFA state
  - Reads in a character, and uses a switch statement to determine the next state to call
  - Final states return token.
- Options 2: Single procedure for DFA, switch based on first character
  - We'll see an example of this in a few slides.
- Pros
  - Fairly straightforward to write.
  - If written well, can be faster than generated scanners (particularly option 2).
  - Can handle any weird language corner cases that don't map perfectly to the RE/NFA/DFA model.
  - Readable code (mostly).
- Cons
  - A lot of tedious work – thus, error prone.

DFA => code, automatic

- Option 1: use a tool to generate table driven scanner
  - Rows: states of DFA
  - Columns: input characters
  - Entries: action
    - Go to next state
    - Accept token, go to start state
    - Error
- Pros
  - Convenient – just feed it the token regular expressions
  - Exactly matches specification you give it, if tool correct
- Cons
  - "Magic"
  - Sometimes language constructs don't map perfectly to FA model
  - Not efficient

DFA => code, automatic

- Option 2: use tool to generate direct-coded scanner
  - Transitions embedded in the code, using conditional statements, loops, possibly goto
- Pros
  - Convenient – just feed it the REs
  - Exactly matches specification you give it, if tool correct
  - More efficient than table driven scanners
- Cons
  - "Magic"
  - Code is unreadable
  - Generates *lots* of code (but can be fairly fast)

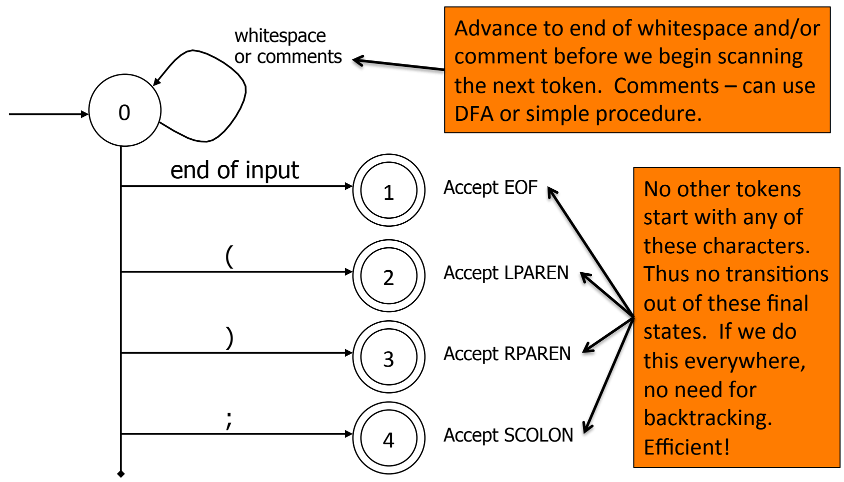
The Real World

- In commercial settings (and most gcc front ends) hand written scanners used more often than not.
  - Especially for larger languages, e.g., C++/Java.
  - Can purchase, e.g., EDG C/C++ front end (used by Cray, Intel, others).
- Why?
  - Fastest
  - Can handle language corner cases – C++ especially bad.
  - Readable/debuggable code.

Example: A hand-written DFA and scanner

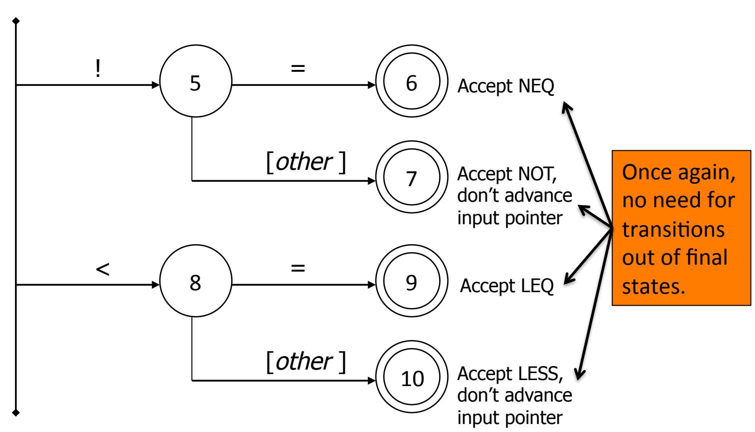
- To demonstrate, we'll show a hand-written DFA for some typical programming language constructs
  - Then use to construct a hand-written scanner
- Setting: Scanner is called whenever the parser needs a new token
  - Scanner stores current position in input
  - From there, use a DFA to recognize the longest possible input sequence that makes up a token and return that token; save updated position for next time
- Disclaimer: Example for illustration only – you'll use tools for the course project.
- Credit: Hal Perkins wrote this DFA and code.

Scanner DFA Example (1)



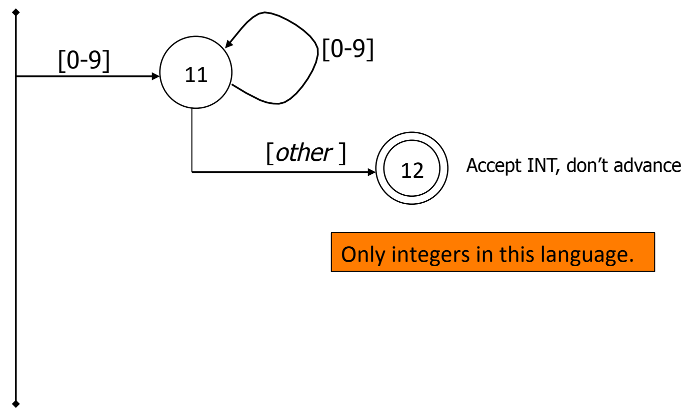
Winter 2013 UW CSE 401 (Michael Ringenburg) 25

Scanner DFA Example (2)



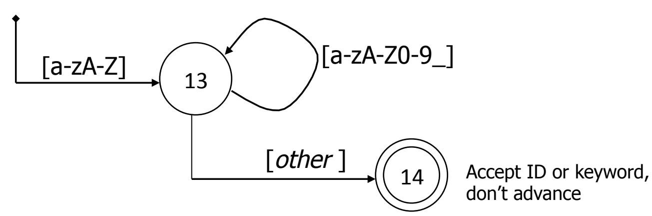
Winter 2013 UW CSE 401 (Michael Ringenburg) 26

Scanner DFA Example (3)



Only integers in this language.

Scanner DFA Example (4)



- Strategies for handling identifiers vs keywords
  - Hand-written scanner: look up identifiers in table of keywords (good application of perfect hashing–i.e., given knowledge of keys ahead of time, can ensure no collisions.)
  - Machine-generated scanner: generate DFA with appropriate transitions to recognize keywords (> priority than IDs).

Backtracking

- As we saw, backtracking is not necessary in our DFA.
  - More efficient
- In many cases, token syntax can be chosen (and DFA constructed carefully) such that backtracking is rare (or can be avoided entirely).
- Easier to ensure this happens in a hand-written scanner
  - Part of why *well-written* hand-written scanners are the most efficient.

Implementing a Scanner by Hand – Token Representation

- A token is a simple, tagged structure
  - (Compilers written in C/C++ often use a "tagged union" style structure)

```
public class Token {
  public int kind;   // token's lexical class
  public int intVal;// integer value if class = INT
  public String id; // actual identifier if class = ID

  // lexical classes
  public static final int EOF    = 0; // "end of file"
                                      // token
  public static final int ID     = 1; // identifier,
                                      // not keyword
  public static final int INT    = 2; // integer
  public static final int LPAREN = 4; // (
  public static final int SCOLN  = 5; // ;
  public static final int WHILEK = 6; // )
  // etc. etc. etc. …
```

Simple Scanner Example

```
// global state and methods

// next unprocessed input character
static char nextch;

// advance to next input char
void getch() { … }

// skip whitespace and comments
void skipWhitespace() { … }

// input is a letter, digit, or _
boolean isIDChar(char c);
```


Scanner getToken() method

```
// Called by parser to retrieve the next input token
public Token getToken() {
  Token result;

  skipWhiteSpace();

  if (/*no more input*/) {
    result = new Token(Token.EOF); return result;
  }

  switch(nextch) {
    case '(': result = new Token(Token.LPAREN); getch();
              return result;
    case ')': result = new Token(Token.RPAREN); getch();
              return result;
    case ';': result = new Token(Token.SCOLON); getch();
              return result;
    // Repeat for other single character tokens…
```

getToken() (2)

```
case '!': // ! or !=
  getch();
  if (nextch == '=') {
    result = new Token(Token.NEQ); getch();
    return result;
  } else {
    result = new Token(Token.NOT); return result;
  }

case '<': // < or <=
  getch();
  if (nextch == '=') {
    result = new Token(Token.LEQ); getch();
    return result;
  } else {
    result = new Token(Token.LESS); return result;
  }
```


getToken() (3)

```
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
  // integer constant
  String num = nextch;
  getch();
  while (Character.isdigit(nextch)) {
    num = num + nextch;
    getch();
  }
  result = new Token(Token.INT,
                     Integer(num).intValue());
  return result;
```

getToken() (4)

```
case 'a': … case 'z':
case 'A': … case 'Z':  // id or keyword
  string s = nextch;
  getch();
  while (isIDChar(nextch))  // letter, digit, _
  {
    s = s + nextch; getch();
  }
  if (keywordTable.isKeyword(s)) {
    result = new Token(keywordTable.getKind(s));
  } else {
    result = new Token(Token.ID, s);
  }
  return result;
```


MiniJava Scanner Generation

- We'll use the JFlex tool to automatically create a scanner from a specification file.
- We'll use the CUP tool to automatically create a parser from a specification file.
- Token class is shared by jflex and CUP. Lexical classes (token kinds) are listed in CUP's input file and it generates the token class definition.
- So you'll need to modify both specification files for the scanner portion of your project
  - Parser mods will be small.

JFlex Specification Example

- Open src/Scanner/minijava.jflex in your project starter code. You'll see that a few tokens have already been done for you, to demonstrate how it works, e.g.:

```
"+" { return symbol(sym.PLUS); }
{letter} ({letter}|{digit}|_)* {
  return symbol(sym.IDENTIFIER, yytext());
}
```

- Format is Token RE, followed by code to execute.
- Can define helper abbreviations, e.g.:

```
letter = [a-zA-Z]
digit = [0-9]
```


Specifying the tokens

- Tokens are specified in the CUP file
  - src/Parser/minijava.cup

```
/* Terminals (tokens returned by the scanner) */

/* reserved words: */
terminal DISPLAY;

/* operators: */
terminal PLUS, BECOMES;

/* delimiters: */
terminal LPAREN, RPAREN, SEMICOLON;

/* tokens with values: */
terminal String IDENTIFIER;
```

JFlex Demo!

- Your project starter code has a few tokens defined already. We'll add multiplication.

Coming Attractions

- Starting next lecture: parsing
  - Will do LR parsing first – we need this for the project, then LL (recursive-descent) parsing, which you should also know.
  - May take the rest of January – it's a big topic…
- Sections – more details about using JFlex for your project.
  - The *full* details can be found in the JFlex and CUP documentation.