Building a Scanner

CS164, Fall 2004

Administrativia

- **Extra credit for bugs in project assignments**
  - in starter kits and handouts
  - TAs are final arbiters of what's a bug
  - only the first student to report the bug gets credit

What does a lexer do?

Recall: The Structure of a Compiler

Decaf program (stream of characters)
↓
scanner
↓ stream of tokens
parser
↓ Abstract Syntax Tree (AST)
checker
↓ AST with annotations (types, declarations)
code gen
↓ maybe x86

Recall: Lexical Analysis

- The input is just a sequence of characters. Example:

```
if (i == j)
        z = 0;
else
        z = 1;
```

- More accurately, the input is string:
  \tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
- Goal: find lexemes and map them to tokens:
  1. partition input string into substrings (called lexemes), and
  2. classify them according to their role (role = token)

Continued

- Lexer input:
  \tif(i==j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
- partitioned into these *lexemes*:

\t	if	(	i	==	j	)	\n\t\t	z		=		0	;	\n\t	else	\n\t\t	z		=		1	;

- mapped to a sequence of *tokens*
  IF, LPAR, ID("i"), EQUALS, ID("j") …
- Notes:
  - whitespace lexemes are dropped, not mapped to tokens
    - is this the same fatal mistake as in FORTRAN? (see Lecture 1)
  - some tokens have attributes: the lexeme and/or line number
    - why do we need them?

What's a Token?

- A token is a syntactic category
  - In English:
    noun, verb, adjective, …
  - In a programming language:
    Identifier, Integer, Keyword, Whitespace, …
- Parser relies on the token distinctions:
  - identifiers are treated differently than keywords
  - but all identifiers are treated the same, regardless of what lexeme created them

What are lexemes?

- Webster:
  - "items in the vocabulary of a language"
- cs164:
  - same: items in the vocabulary of a language:
    - numbers, keywords, identifiers, operators, etc.
  - strings into which the input string is partitioned.

How to build a scanner for Decaf?

Writing the lexer

- Not by hand
  - tedious, repetitious, error-prone, non-maintainable
- Instead, we'll build a lexer generator
  - once we have the generator, we'll only describe the lexemes and their tokens …
    - that is, provide Decaf's lexical specification (the What)
  - … and generate code that performs the partitioning
    - generated code hides repeated code (the How)

Code generator: key benefit

- The scanner generator allows the programmer to focus on:
  - What the lexer should do,
  - rather than How it should be done.
- what: declarative programming
- how: imperative programming

Imperative scanner (in Java)

- Let's first build the scanner in Java, by hand:
  - to see how it is done, and where's the repetitious code that we want to hide
- A simple scanner will do. Only four tokens:

TOKEN	Lexeme
ID	a sequence of one or more letters or digits starting with a letter
EQUALS	"=="
PLUS	"+"
TIMES	"*"

Imperative scanner

```
c=nextChar();
if (c == '=') { c=nextChar(); if (c == '=') {return EQUALS;}}
if (c == '+') { return PLUS; }
if (c == '*') { return TIMES; }
if (c is a letter) {
   c=NextChar();
   while (c is a letter or digit) {  c=NextChar(); }
   undoNextChar(c);
   return ID;
}
```

13

Ras Bodik, CS 164, Fall 2004

Imperative scanner

- You could write your entire scanner in this style
  - and for small scanners this style is appropriate
- This code looks simple and clean, but try to add
  - tokens that start with the same string: "if" and "iffy"
  - C-style comments: /* anything here /* nested comments */ */
  - string literals with escape sequences: "…\t … \"…"
  - error handling, e.g., badly formed strings (see PA2)
- Look at StreamTokenizer.nextToken() for an example of real imperative scanner
  - in Eclipse, type Ctrl+Shift+T.
  - Enter StreamTokenizer.
  - Press F4. In the Hierarchy view, select method nextToken.

14

Ras Bodik, CS 164, Fall 2004

Maximal munch rule

- What is the need for undoNextChar()?
  - it performs look-ahead, to determine whether the ID lexeme can be grown further
- This is an example of maximal much rule:
  - this rule followed by all scanners
  - **the rule:** the input character stream is partitioned into lexemes that are as large as possible
  - **Ex.:** in Java, "iffy" is not partitioned into "if" (the IF keyword) and "fy" (ID), but into "iffy" (ID)

15

Ras Bodik, CS 164, Fall 2004

Imperative Lexer: what vs. how

```
c=nextChar();
if (c == '=') { c=nextChar(); if (c == '=') {return EQUALS;}}
if (c == '+') { return PLUS; }
if (c == '*') { return TIMES; }
if (c is a letter) {
   c=NextChar();
   while (c is a letter or digit) {  c=NextChar(); }
   undoNextChar(c);
   return ID;
}
```

☞ little logic, much plumbing

16

Ras Bodik, CS 164, Fall 2004

Identifying the plumbing (the how)

```
c=nextChar();
if (c == '=') { c=nextChar(); if (c == '=') {return EQUALS;}}
if (c == '+') { return PLUS; }
if (c == '*') { return TIMES; }
if (c is a letter) {
   c=NextChar();
   while (c is a letter or digit) {  c=NextChar(); }
   undoNextChar(c);
   return ID;
}
```

☞ characters read always the same way

17

Ras Bodik, CS 164, Fall 2004

Identifying the plumbing (the how)

```
c=nextChar();
if (c == '=') { c=nextChar(); if (c == '=') {return EQUALS;}}
if (c == '+') { return PLUS; }
if (c == '*') { return TIMES; }
if (c is a letter) {
   c=NextChar();
   while (c is a letter or digit) {  c=NextChar(); }
   undoNextChar(c);
   return ID;
}
```

☞ tokens are always return-ed

18

Ras Bodik, CS 164, Fall 2004

Identifying the plumbing (the how)

```
c=nextChar();
if (c == '=') { c=nextChar(); if (c == '=') {return EQUALS;}}
if (c == '+') { return PLUS; }
if (c == '*') { return TIMES; }
if (c is a letter) {
   c=NextChar();
   while (c is a letter or digit) {  c=NextChar(); }
   undoNextChar(c);
   return ID;
}
```

☞ the lookahead is explicit

Identifying the plumbing (the how)

```
c=nextChar();
if (c == '=') { c=nextChar(); if (c == '=') {return EQUALS;}}
if (c == '+') { return PLUS; }
if (c == '*') { return TIMES; }
if (c is a letter) {
   c=NextChar();
   while (c is a letter or digit) {  c=NextChar(); }
   undoNextChar(c);
   return ID;
}
```

☞ must build decision tree out of nested if's (yuck!)

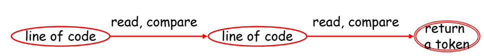
Can we hide the plumbing?

- That is, can we avoid having to
  - spell out the details of calls to the input method?
  - write the **return** in front of tokens?
  - code the look-ahead code explicitly?
  - use if's and while's to indicate the decision tree?
- In short, can we make the code look like the specification table?

TOKEN	Lexeme
ID	a sequence of one or more letters or digits starting with a letter
EQUALS	"=="
PLUS	"+"
TIMES	"*"

Separate out the how (plumbing)

- The code actually follows a simple pattern:
  - read next character and compare it with some predetermined character
  - if there is a match, jump to a different line of code
  - repeat this until you return a token.
- Is there a programming language that can encode this concisely?
  - yes, finite automata!



Separate out the what

```
c=nextChar();
if (c == '=') { c=nextChar(); if (c == '=') {return EQUALS;}}
if (c == '+') { return PLUS; }
if (c == '*') { return TIMES; }
if (c is a letter) {
   c=NextChar();
   while (c is a letter or digit) {  c=NextChar(); }
   undoNextChar(c);
   return ID;
}
```


A declarative scanner

Part 1: declarative (the what)

- describe each token as a finite automaton
  - must be supplied for each scanner, of course (it specifies the lexical properties of the input language)

Part 2: imperative (the how)

- connect these automata into a scanner automaton
  - common to all scanners (like a library)
  - responsible for the mechanics of scanning

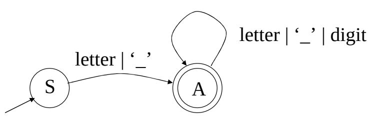
DFAs

Deterministic finite automata (DFA)

- We'll use DFA's as <u>recognizers</u>:
  - given an input string, the DFA will recognize whether the string is a valid lexeme
  - "recognize" means answer true or false
  - Example: Is "xyz" an identifier? The DFA will says yes.
- DFA's alone insufficient to build a scanner:
  - DFA's recognize if a string is, say, an identifier
  - but alone can't partition the input program into lexemes
  - so we'll need to use them in a special way

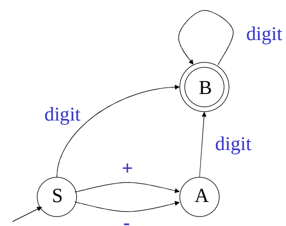
Deterministic Finite Automata

- Example: Decaf Identifiers
  - sequences of one or more letters or underscores or digits, starting with a letter or underscore:



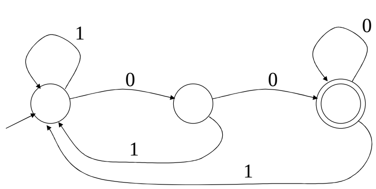
Example: Integer Literals

- DFA that accepts integer literals with an optional + or - sign:

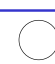
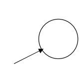
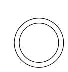
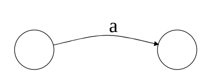


And another (more abstract) example

- Alphabet {0,1}
- What strings does this recognize?



Finite-Automata State Graphs

- A state 
- The start state 
- A final state 
- A transition 

Finite Automata

- Transition

  $s_1 \to^a s_2$

- Is read

  In state s_1 on input "a" go to state s_2

- If end of input
  - If in accepting state => *accept*
  - Otherwise => *reject*
- If no transition possible (got stuck) => reject

Formal Definition

- A finite automaton is a 5-tuple $(\Sigma, Q, \Delta, q, F)$ where:
  - An input alphabet Σ
  - A set of states Q
  - A start state q
  - A set of final states $F \subseteq Q$
  - Δ is the state transition function: $Q \times \Sigma \to Q$ (i.e., encodes transitions state $\to^{input}$ state)

Language defined by DFA

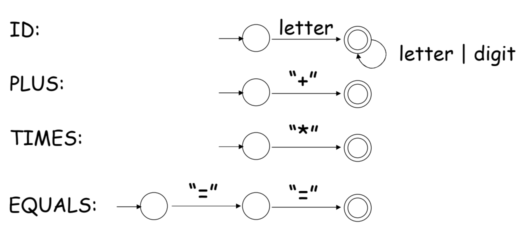
- The language defined by a DFA is the set of strings accepted by the DFA.

  - in the language of the identifier DFA shown above:
    - x, tmp2, XyZzy, position27.

  - *not* in the language of the identifier DFA shown above:
    - 123, a?, 13apples.

the declarative scanner

Part 1: create a DFA for each token



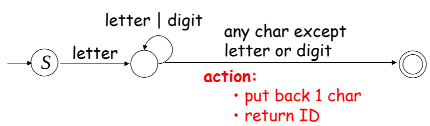
Part 2: allow actions on DFA transitions

- the action can be one of
  - "put back one character" or
  - "return token XYZ",

- such DFA is called a <u>transducer</u>
  - it translates input string to an output string

Step 2: example of extending a DFA

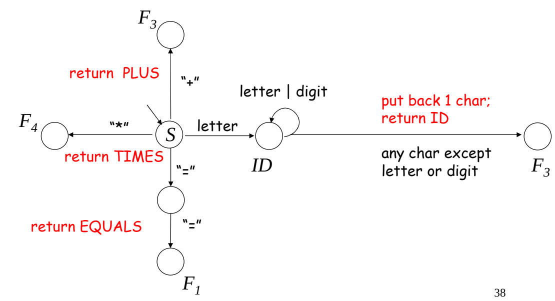
- The DFA recognizing identifiers is modified to:



- Look-ahead is added for lexemes of variable length
  - in our case, only ID needs lookahead
- A note on action "**return ID**"
  - resets the scanner back into start state S (recall that scanner is called by parser; each time, one token is returned)

Step 2: Combine the extended DFA's

The algorithm: merge start nodes of the DFA's.

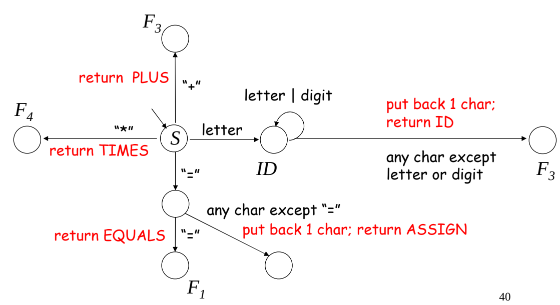


Towards a realistic scanner

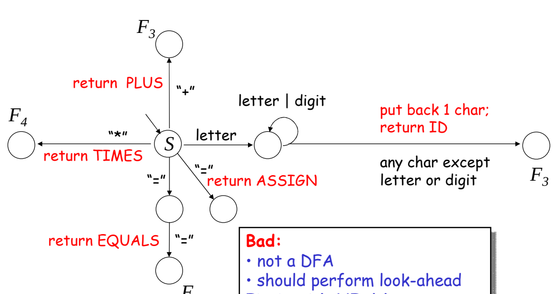
- Building a scanner out of DFA's (as shown) is simple but doesn't quite work
- Consider a fifth token type, for the assignment operator

TOKEN	Lexeme
ID	a sequence of one or more letters or digits starting with a letter
EQUALS	"=="
ASSIGN	"="
PLUS	"+"
TIMES	"*"

Correct scanner



But above algorithm produces



Bad:
- not a DFA
- should perform look-ahead

Fix it with NFA's!

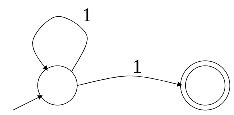
NFAs

Deterministic vs. Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - one transition per input characater per state
  - no ε-moves
- Nondeterministic Finite Automata (NFA)
  - allows multiple outgoing transitions for one input
  - can have ε-moves
- Both: finite automata have finite memory
  - Need only to encode the current state
  - NFA's can be in multiple states at once (stay tuned)

A simple NFA example

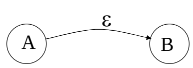
- Alphabet: { 0, 1 }



- The operation of the automaton is not completely defined by the input
  - On input "11" the automaton could be in either state

Epsilon Moves

- Another kind of transition: ε-moves



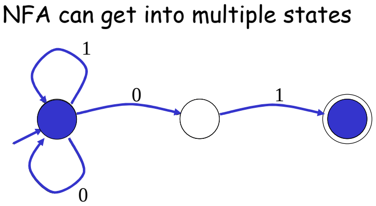
- Machine can move from state A to state B without reading input

Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input
- NFAs can choose
  - whether to make ε-moves
  - which of multiple transitions for a single input to take
  - so we think of an NFA as being in one of multiple states (see next example)

Acceptance of NFAs

- An NFA can get into multiple states



- Input: 1 0 1
- Rule: NFA accepts if it <u>can</u> get in a final state

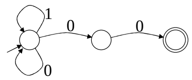
NFA vs. DFA (1)

- NFA's and DFA's are equally powerful
  - each NFA can be translated into a corresponding DFA (one that recognizes same strings)
  - formally, NFAs and DFAs recognize the same set of languages (called regular languages)
- But NFA's are more convenient
  - they allow easy merges of automata, which helps in scanner construction
- And DFAs are easier to implement
  - There are no choices to consider
- in PA2, you will use NFA's
  - we'll give you NFA recognizer code

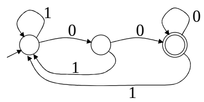
NFA vs. DFA (2)

- For a given language the NFA can be simpler than the DFA

NFA



DFA



- DFA can be exponentially larger than NFA

full declarative scanner

The algorithm

With NFA's, we can develop the scanner:

1. Construct an automaton for each lexeme (as before)
2. Merge them into the scanner automaton (as before)
3. Don't extend the automata as before, instead
   - whenever you reach a final state:
     - remember position in input (so that you can undo reads)
     - keep reading more characters, moving to other states
   - whenever you get stuck (cannot make a move on next char):
     - return to the last final state (i.e., undo the reads)
     - return the token associated with this final state

Notes

- Notice that reading past final state implements look-ahead
- This look-ahead is unbounded
  - can return any number of characters
- Can you think of two lexemes that will require the scanner to return a large amount of characters?
  - "large" means we can write an input that will make the necessary look-ahead arbitrarily large.

Practical concerns

- Ambiguity
  - **problem:** scanner may reach multiple final states at once
  - **ex.:** "if" matches both ID and IF (the keyword)
  - **solution:** prioritize tokens (IF wins over ID)
- Discarding whitespace
  - **solution:** final state for white-space lexeme is special: don't return a token, but jump to (common) start state
- Error inputs
  - **problem:** discard illegal lexemes and print an error message
  - **simple solution** (discard char by char): add a lexeme that matches any character, giving it lowest priority; it will match when no other will

We have a full declarative scanner

- imperative part,
  - stored in the library (the third step of the algorithm)
  - this is the run-time: it performs look-ahead, moves, input matching, returning tokens
- declarative part
  - think of it as configuring the run-time of the scanner
  - configuration done by specifying an automaton for each token

Programming the declarative scanner

- configuring the run-time can be done by hand
  - This code creates a DFA for the EQUALS token:

```
Node start = new Node(), middle = new Node();
Node final = new FinalNode(EQUALS);
start.addEdge(middle, '='); middle.addEdge(final, '=');
```

  - this is an improvement over the imperative scanner
    - more readable, maintainable, les error-prone
  - but our goal is to avoid writing even this,
    - we'll write a code generator
    - it will translate regular expressions (our textual program) into NFA's

regular expressions

Regular Expressions

- Automaton is a good "visual" aid
  - but is not suitable as a specification (its textual description is too clumsy)
- regular expressions are a suitable *specification*
  - a compact way to define a language that can be accepted by an automaton.
- used as the input to a scanner generator
  - define each token, and also
  - define white-space, comments, etc
    - these do not correspond to tokens, but must be recognized and ignored.

Example: Pascal identifier

- Lexical specification (in English):
  - a letter, followed by zero or more letters or digits.
- Lexical specification (as a regular expression):
  - letter . (letter | digit)*

|	means "or"
.	means "followed by"
*	means zero or more instances of
()	are used for grouping

Operands of a regular expression

- Operands are same as labels on the edges of an FSM
  - single characters, or
  - the special character ε (the empty string)
- "letter" is a shorthand for
  - *a / b / c / … / z / A / … / Z*
- "digit" is a shorthand for
  - *0 / 1 / … / 9*
- sometimes we put the characters in quotes
  - necessary when denoting | . *

Precedence of | . * operators.

Regular Expression Operator	Analogous Arithmetic Operator	Precedence
|	plus	lowest
.	times	middle
*	exponentiation	highest

- Consider regular expressions:
  - letter.letter | digit*
  - letter.(letter | digit)*

More examples

- Describe (in English) the language defined by each of the following regular expressions:
  - letter (letter | digit*)
  - digit digit* "." digit digit*

Example: Integer Literals

- An integer literal with an optional sign can be defined in English as:
  - "(nothing or + or -) followed by one or more digits"
- The corresponding regular expression is:
  - (+|-|ε).(digit.digit*)
- A new convenient operator '+'
  - same precedence as '*'
  - digit.digit* is the same as
  - digit+ which means "one or more digits"

Language Defined by a Regular Expression

- Recall: language = set of strings
- Language defined by an automaton
  - the set of strings accepted by the automaton
- Launguage defined by a regular expression
  - the set of strings that match the expression.

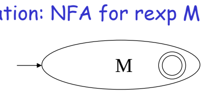
Regular Exp.	Corresponding Set of Strings
ε	{""}
a	{"a"}
a.b.c	{"abc"}
a | b | c	{"a", "b", "c"}
(a | b | c)*	{"", "a", "b", "c", "aa", "ab", ..., "bccabb" ...}

Translating RE to NFA's

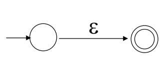
- key idea:
  - find hierarchy in the regular expression
    - find nested RE's
  - define translation for individual operation
  - compose the translations of nested RE's
- RE's as AST
  - translation via AST traversal

Regular Expressions to NFA (1)

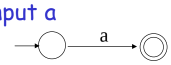
- For each kind of rexp, define an NFA
  - Notation: NFA for rexp M



- For ε

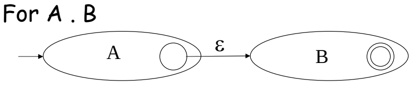


- For input a

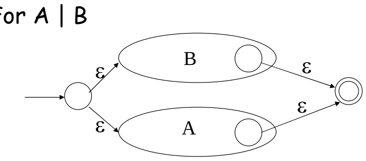


Regular Expressions to NFA (2)

- For A . B

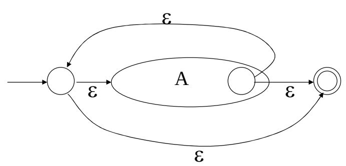


- For A | B



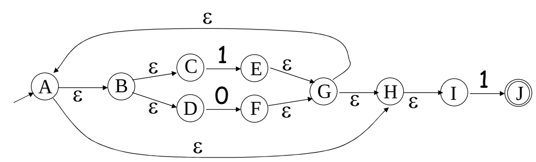
Regular Expressions to NFA (3)

- For A*



Example of RegExp -> NFA conversion

- Consider the regular expression
  (1|0)*1
- The NFA is



Automatically translating RE's to NFA's

- we need an AST that represents the RE
  - the translation is then a bottom up traversal of the AST
  - like in Java pretty printing
- options:
  - properly,
    - you write AST designed for RE operators
    - and a parser from RE syntax to the RE AST
  - in PA2, we'll reuse Java syntax and Java AST
    - overload Java operators
    - it's a hack, but it allows us to implement the RE-NFA translation quickly

odds and ends

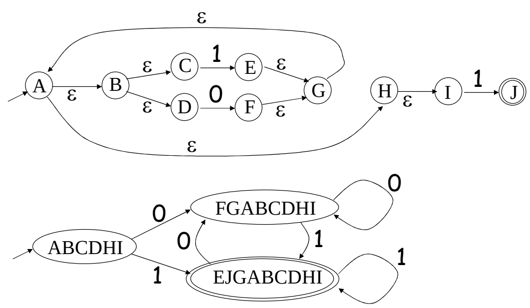
Practical issues

- PA2, we'll use NFAs
  - But DFAs are often faster
  - because they can be implemented with tables
- Next few slides
  - NFA to DFA conversion
  - table implementation of DFA's
- Feel free to implement these two in PA2
  - experiment with how much faster your scanner is than the NFA-based scanner

NFA to DFA. The Trick

- Simulate the NFA
- Each state of DFA
  = a non-empty subset of states of the NFA
- Start state
  = the set of NFA states reachable through ε-moves from NFA start state
- Add a transition $S \to^a S'$ to DFA iff
  - S' is the set of NFA states reachable from the states in S after seeing the input a
    - considering ε-moves as well

NFA -> DFA Example



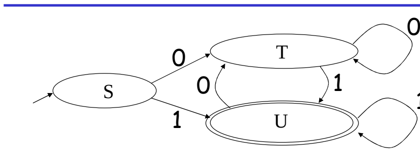
NFA to DFA. Remark

- An NFA may be in many states at any time
- How many different states ?
- If there are N states, the NFA must be in some subset of those N states
- How many non-empty subsets are there?
  - 2^N - 1 = finitely many

Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbols"
  - For every transition $S_i \to^a S_k$ define T[i,a] = k
- DFA "execution"
  - If in state S_i and input a, read T[i,a] = k and skip to state S_k
  - Very efficient

Table Implementation of a DFA



	0	1
S	T	U
T	T	U
U	T	U

Implementation (Cont.)

- NFA -> DFA conversion is at the heart of tools such as flex or jlex
- But, DFAs can be huge
- In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations

the end