CS412/413

Introduction to
Compilers and Translators
Spring '00

Lecture 2: Lexical Analysis

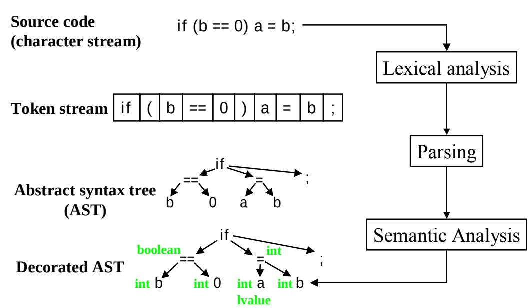
Outline

- Administration
- Compilation in a nutshell (or two)
- What is lexical analysis?
- Writing a lexer
- Specifying tokens: regular expressions
- Writing a lexer generator
  - Converting regular expressions to Non-deterministic finite automata (NFAs)
  - NFA to DFA transformation

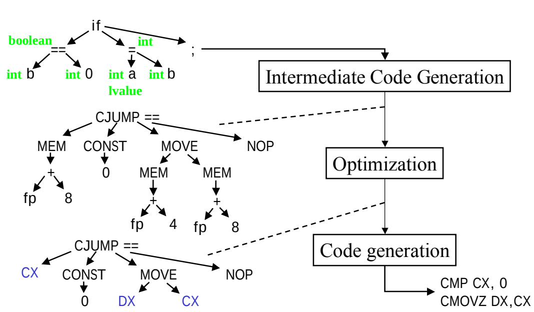
Administration

- Office hours
  Myers: Wednesday 2-3PM
  Kliger: Thursday 3-4PM
  Lin: Monday 4-5PM
  Nystrom: Friday 11-12PM
- PA1 out – due next Friday, Feb. 4
  - other handouts: advice on programming in groups
  - use this assignment as a warm-up!
- Questionnaire needed at end of class

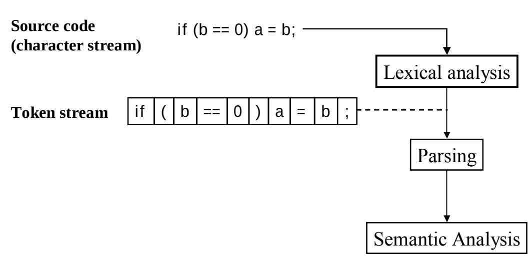
Compilation in a Nutshell 1



Compilation in a Nutshell 2



First step: lexical analysis



Tokens

- Identifiers: x y11 elsex _i00
- Keywords: if else while break
- Integers: 2 1000 -500 5L
- Floating point: 2.0 0.00020 .02 1. 1e5 0.e-10
- Symbols: + * { } ++ < << [ ] >=
- Strings: "x" "He said, \"Are you?\""
- Comments: /** comment **/

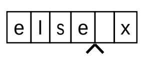
Ad-hoc lexer

- How to read identifier tokens?

```
Token readIdentifier( ) {
  String id = "";
  while (true) {
    char ch = input.read();
    if (!identifierChar(ch))
        return new Token(ID, s, lineNumber);
    s = s + String(ch);
  }
}
```


Look-ahead character

- Scan text one character at a time
- Use look-ahead character (next) to determine what kind of token to read *and* when the current token ends




```
char next;
...
while (identifierChar(next)) {
  token[len++] = next;
  next = input.read ();
}
```


Top-level loop

```
class Lexer {
  InputStream s;
  char next;
  Lexer(InputStream s_) { s = s_; next=s.read(); }
  Token nextToken( ) {
    if (identifierChar(next))
        return readIdentifier();
    if (numericChar(next))
        return readNumber();
    if (next == '\"') return readStringConst();
    ...
  }
}
```


Problems

- Don't know what kind of token we are going to read from seeing first character
  - if token begins with "i'' is it an identifier?
  - if token begins with "2" is it an integer constant?
  - interleaved tokenizer code is hard to write correctly, harder to maintain
- Need a more principled approach: *lexer generator* that generates efficient tokenizer automatically (e.g., lex, JLex)

Issues

- How to describe tokens

  2.e0 20.e-01 2.0000
- How to break text up into tokens

  if (x == 0) a = x<<1;

  iff (x == 0) a = x<1;
- How to tokenize efficiently
  - tokens may have similar prefixes
  - want to look at each character ~1 time

How to Describe Tokens

- Programming language tokens can be described as **regular expressions**
- A regular expression R describes some set of strings L(R)
  - L(**abc**) = { “**abc**” }
- *e.g.*, tokens for floating point numbers, identifiers, etc.

Regular Expression Notation

a ordinary character stands for itself

ε the empty string

R|S any string from either L(R) or L(S)

RS string from L(R) followed by one from L(S)

R* zero or more strings from L(R), concatenated

ε|R|RR|RRR|RRRR ...

RE Shorthand

R+ one or more strings from L(R): R(R*)

R? optional R: (R|ε)

[**a**-**z**] one character from this range: (**a**|**b**|**c**|**d**|**e**|**...**)

[^**a**-**z**] one character *not* from this range

Examples

Regular Expression	Strings in L(R)
a	“a”
ab	“ab”
a | **b**	“a” “b”
	“”
(**ab**)*	“” “ab” “abab” ...
(**a** | ε) **b**	“ab” “b”

More Examples

Regular Expression	Strings in L(R)
digit = [**0**-**9**]	“0” “1” “2” “3” ...
posint = *digit*+	“8” “412” ...
int = -? *posint*	“-42” “1024” ...
real = *int* (ε | (. *posint*))	“-1.56” “12” “1.0”
[**a**-**zA**-**Z_**][**a**-**zA**-**Z0**-**9_**]*	C, Java identifiers

How to break up text

elsex = 0;

1	else	x	=	0
2	elsex	=	0	

- REs alone not enough: need rule for choosing
- Most languages: longest matching token wins
- Exception: early FORTRAN (totally whitespace-insensitive)
- Ties resolved by prioritizing tokens
- Res + priorities + longest-matching token rule = lexer definition

Lexer Generator Spec

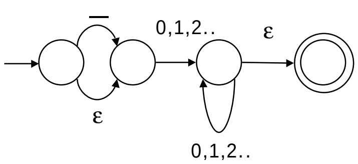
- Input to lexer generator:
  - list of regular expressions in priority order
  - associated *action* for each RE (generates appropriate kind of token, other bookkeeping)
- Output:
  - program that reads an input stream and breaks it up into tokens according to the REs. (Or reports lexical error -- “*Unexpected character*”)

How does it work?

- Regular expressions describe the languages that can be recognized by *finite automata*
- Translate each token RE into a non-deterministic finite automaton (NFA)
- Convert the NFA into an equivalent DFA
- Minimize DFA (to reduce # states)
- Emit code driven by DFA tables
- Advantage: DFAs efficient to implement
  - inner loop: look up next state using current state & look-ahead character

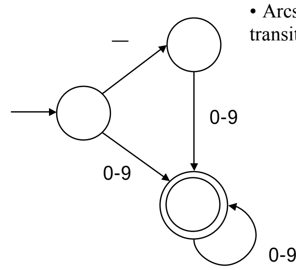
RE ⇒ NFA

-?[0-9]+ (**-**|ε) [**0**-**9**][**0**-**9**]*



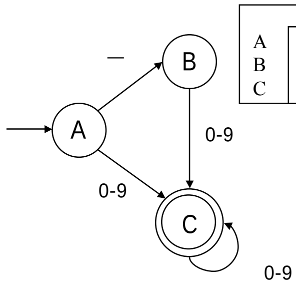
NFA: multiple arcs may have same label, ε transitions don’t eat input

NFA ⇒ minimized DFA



- Arcs may not conflict, no ε transitions

DFA ⇒ table



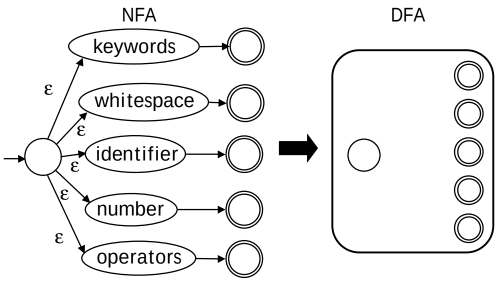
	-	0-9	other
A	B	C	ERR
B	ERR	C	ERR
C	ERR	C	**A**

Lexer Generator Output

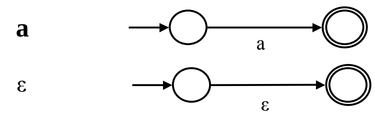
```
Token nextToken() {
  state = START; // reset the DFA
  while (nextChar != EOF) {
    nextState = table [state][nextChar];
    if (nextState == START)
      return new Token(state);
    state = nextState;
    nextChar = input.read( );
  }
}
```

+table
+user actions
in Token()

Handling multiple token REs

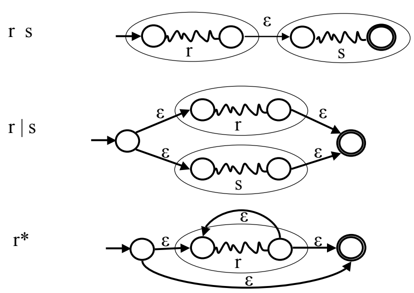


Converting an RE to an NFA



If r and s are regular expressions with the NFA's

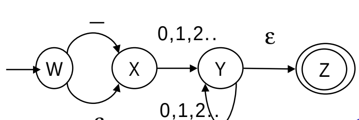
Inductive Construction



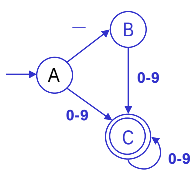
Converting NFA to DFA

- NFA inefficient to implement directly, so convert to a DFA that recognizes the same strings
- Idea:
  - NFA can be in multiple states simultaneously
  - Each DFA state corresponds to a distinct *set* of NFA states
  - n-state NFA may be 2^n state DFA in theory, not in practice (construct states lazily & minimize)

NFA states to DFA states



- Possible sets of states:
  W&X, X, Y&Z
- DFA states:
  A B C



Summary

- Lexical analyzer converts a text stream to tokens
- Tokens defined using regular expressions
- Regular expressions can be converted to a simple table-driven tokenizer by converting them to NFAs and then to DFAs.
- Have covered chapters 1-2 from Appel

Groups

- If you haven't got a full group, hang around after lecture and talk to prospective group members
- If you find a group, fill in questionnaire and submit it
- Send mail to cs412 if you still cannot make a full group
- ***Submit questionnaire*** today in any case