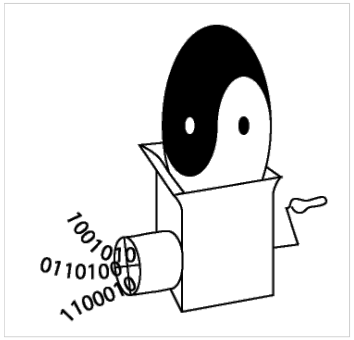


CS4120/4121

Introduction to Compilers

Andrew Myers

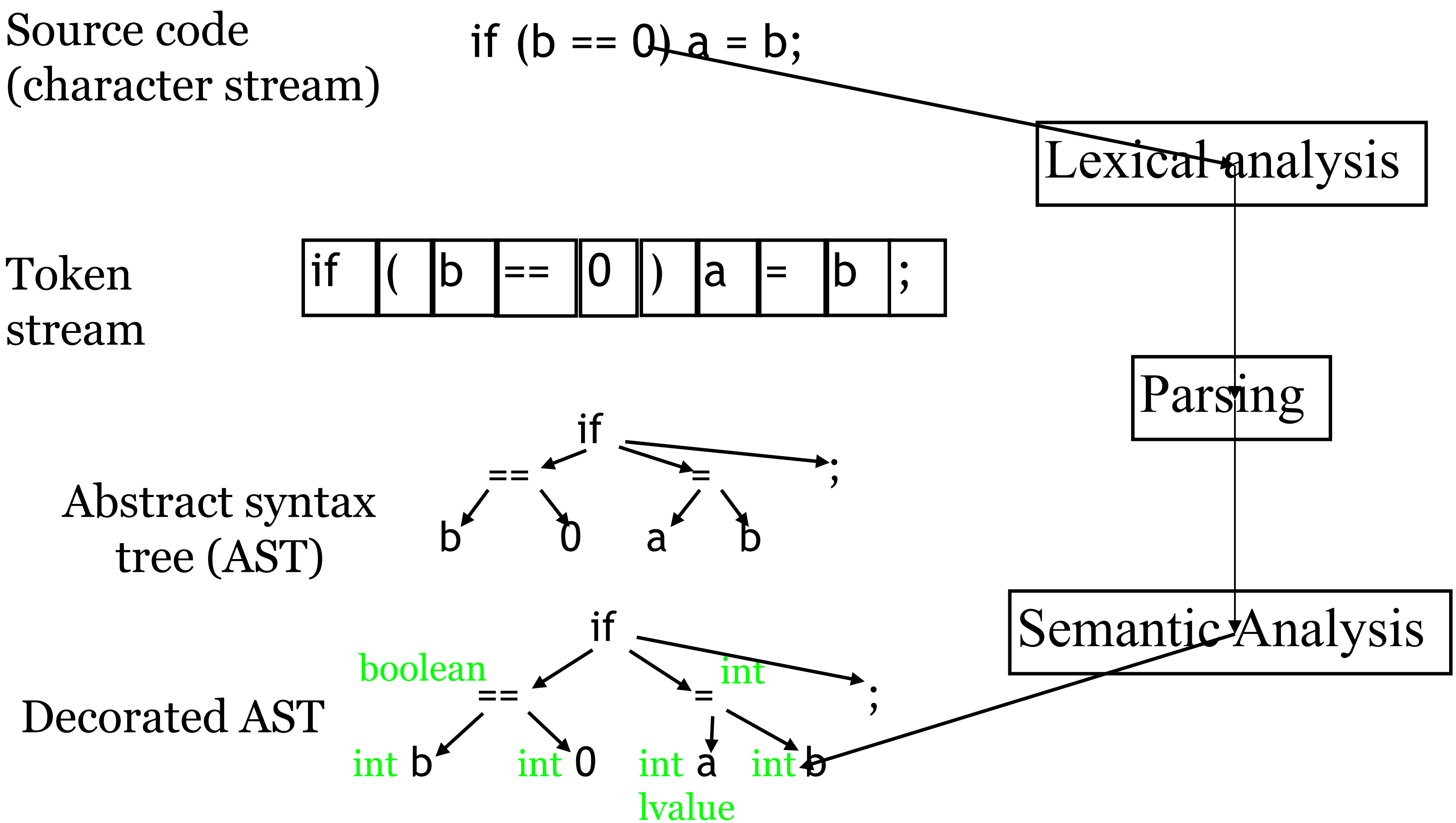
Lecture 2: Lexical Analysis

26 Jan 2018

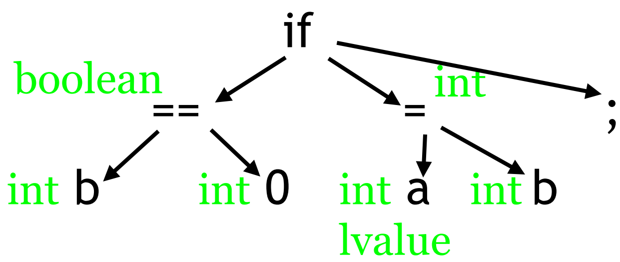
Outline

- 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

Compilation in a Nutshell 1



Compilation in a Nutshell 2



Intermediate Code Generation

SEQ(CJUMP(TEMP(b) == 0, L1, L2),
LABEL(L1),
TEMP(a) = TEMP(b)
LABEL(L2))

Code Generation

cmp r_b, 0
jnz L2
L1: mov r_a, r_b
L2:

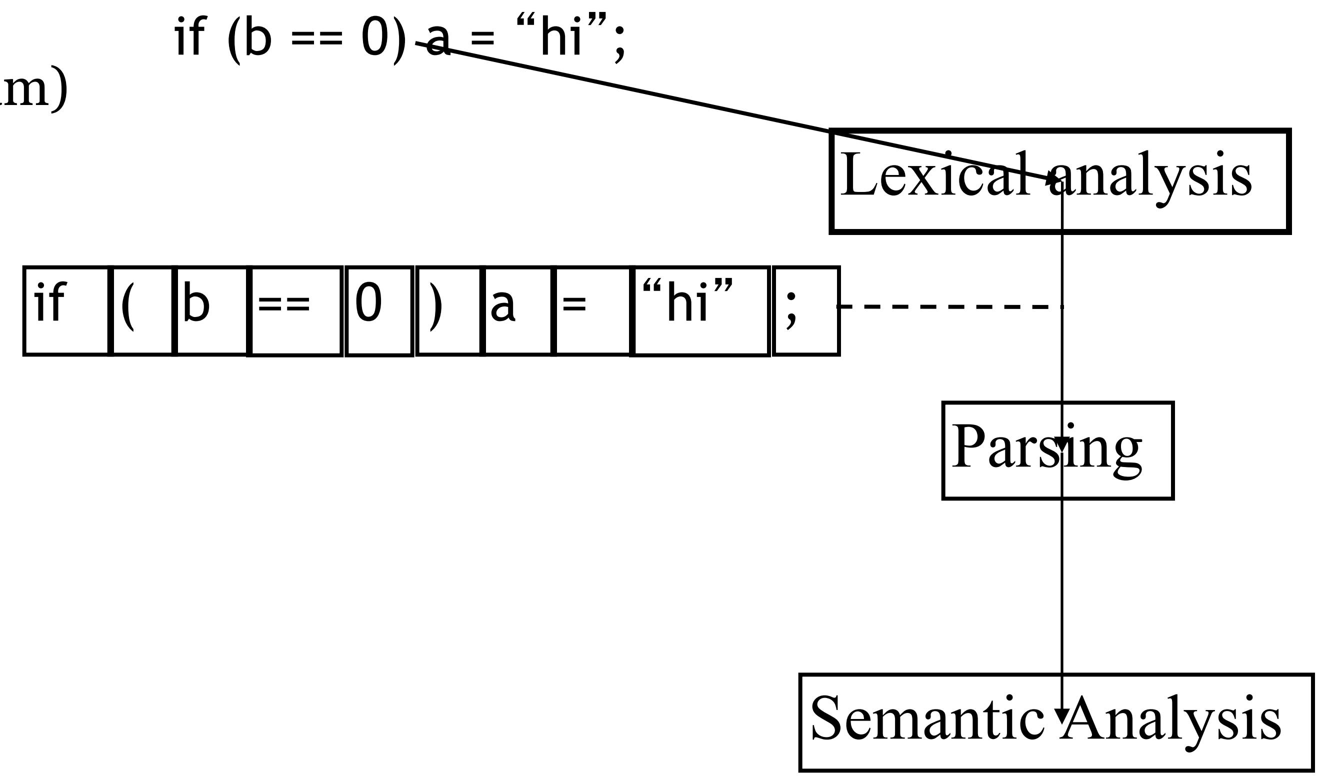
Register allocation, optimization

```
cmp ecx, 0
cmovz [ebp+8],ecx
```

First step: lexical analysis

Source code
(character stream)

Token stream



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: /** don’t change this **/

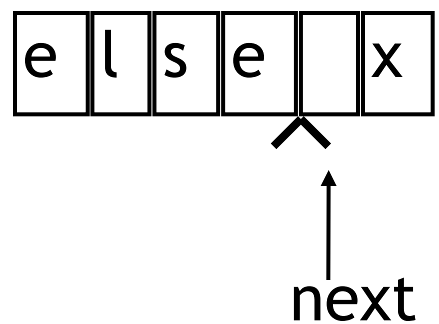
Ad-hoc lexer

- Hand-write code to generate tokens
- How to read identifier tokens?

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

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)) {
   id = id + String(next);
   next = input.read ();
}
```

Ad-hoc lexer: top-level loop

Preloading **next.**

```
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();
        ...
   }
}
```

Cleaner: define input stream that supports lookahead automatically, e.g., cs2112.easyIO.Scanner

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?
  - tokens with common prefixes are hard to tokenize correctly. Either: complex code (hard to maintain), or backtracking (inefficient)
- More principled approach: **lexer generator** (e.g., lex, Jlex) generates efficient tokenizer *automatically* from a lexical specification.

Issues

- How to describe tokens unambiguously

  2.e0   20.e-01   2.0000

  “” “x”   “\\”   “\”\’”

- How to break text up into tokens

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

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

- How to tokenize efficiently
  - tokens may have similar prefixes
  - want to look at each character O(1) times

How to Describe Tokens

- Programming language tokens can be described using **regular expressions**
- Regular expression R describes a set of strings L(R):
  L(R) is the "language" defined by R
  - L(abc) = { abc }
  - L(hello|goodbye) = {hello, goodbye}
  - L([1-9][0-9]*) = all positive integer constants
  - L(X(Y|Z)) = L(XY|XZ) = L(XY) ∪ L(XZ)
- Idea: define each kind of token using RE

Regular expression notation

a an ordinary character stands for itself

ε the empty string (actually written as empty string in lexer specs)

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

$$L(R|S)=L(R) \cup L(S)$$

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

$$L(RS) = \{rs \mid r \in L(R) \wedge s \in L(S)\}$$

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

$$\varepsilon|R|RR|RRR|RRRR \ldots$$

Examples

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

Convenient RE Shorthand

R^+	one or more strings from L(R): = R(R*)
R?	an optional R: = (R|ε)
[abce]	one of the listed characters: (a|b|c|e)
[a–z]	one char from the range: (a|b|c|d|e|...)
[^ab]	anything but one of the listed chars
[^a–z]	one character **not** from the range
R{n}	n repetitions of R (RRRR…)
\x0A	ASCII 10 (newline)
\n	also newline

More Examples (JFlex)

Regular Expression	Strings in L(R)
digit = [0-9]	"0" "1" "2" "3" …
posint = {digit}+	"8" "412" …
int = -? {posint}	"-42" "1024" …
real = {int} (. posint)?	"-1.56" "12" "1.0"

$= (-|\ \varepsilon)(0|\dots|9)(0|\dots|9)^*(\varepsilon\ |\ (.\ (0|\dots|9)\ (0|\dots|9)^*))$

[a-zA-Z_][a-zA-Z0-9_]*	C identifiers

- Lexer generators support abbreviations – cannot be recursive. Forbidden: foo = **a**{foo}| ε

Zero-width assertions

- Not strictly regular expressions…
- Not supported by all lexer generators.

^R	matches R if preceded by newline
R$	matches R if followed by newline
\b	match a word boundary (Perl)
\A	match beginning of input (Perl)
\Z	match end of input (Perl)
R_1/R_2	matches R_1 if followed by something matching R_2 (lex)

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 – even if a shorter token is only way to parse tokens.
  - Exception: early FORTRAN (totally whitespace-insensitive)
  - Ties in length resolved by prioritizing tokens
- RE's + 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")

Example: JFlex

```
%line
%column
%%
digits = 0|[1-9][0-9]*
letter = [A-Za-z]
identifier = {letter}({letter}|[0-9_])*
whitespace = [\ \t\n\r]+
%%
{whitespace} {/* discard */}
{digits}        { return new IntegerConstant(Integer.parseInt(yytext()); }
"if"            { return new IfToken(); }
"while"         { return new WhileToken(); }
…
{identifier}    { return new IdentifierToken(yytext()); }
```

Lexer states

- Most lexer generators allow conditioning on lexer state. Helps with long tokens (strings, comments):

```
"/*"        { yybegin(COMMENT); }
<COMMENT> {
  "*/"      { yybegin(YYINITIAL); }
  .|\n      { /* ignore */ }
}
```

Summary

- Lexical analyzer converts a text stream to tokens
- Ad-hoc lexers hard to get right, maintain
- For most languages, legal tokens conveniently, precisely defined using regular expressions
- Lexer generators generate lexer code automatically from token RE's, precedence
- Next lecture: how lexer generators work