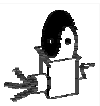


CS412/413

Introduction to Compilers
Andrew Myers

Lecture 2: Lexical Analysis
26 Jan 01

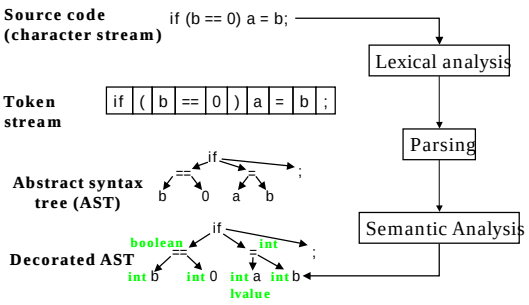
Outline

- 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

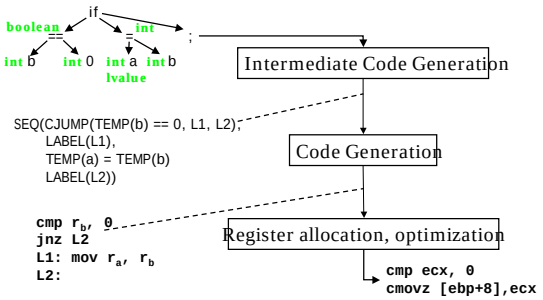
Administration

- PA1 out – due Wednesday, Feb. 7
  - use this assignment as a warm-up!
- Questionnaire needed by 5PM today
  - recommended: use web form, not paper

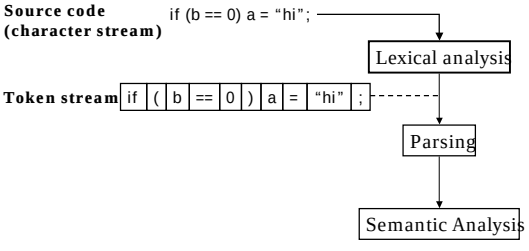
Compilation in a Nutshell 1



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: /** don't change this **/

Ad-hoc lexer

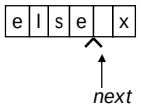
- Hand-write code to generate tokens
- How to read identifier tokens?

```
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

```
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 unambiguously
  2.e0 20.e-01 2.0000
  "" "x" "\\" "\"\'"
- 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 using **regular expressions**
- A regular expression R describes some 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
- Idea: define each kind of token using RE

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 ...

Convienent RE Shorthand

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

R? optional R: (R|ε)

[**abce**] one of the listed characters: (**a**|**b**|**c**|**e**)

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

[^**ab**] anything but one of the listed chars

[^**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*)) = -?[0-9]+(ε | (. [0-9]+))	"-1.56" "12" "1.0"
[**a**-**zA**-**Z_**][**a**-**zA**-**Z0**-**9_**]*	C identifiers

- Lexer generators support abbreviations – cannot be recursive

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
- 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: JLex

```
%%
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()); }
```


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

Groups

- If you haven't got a full group lined up, hang around and talk to prospective group members
- Send mail to cs412 if you still cannot make a full group (can also post to cornell.class.cs412)

- **Submit questionnaire in paper or web form by 5PM in any case**