CS412/413

Introduction to Compilers
Tim Teitelbaum

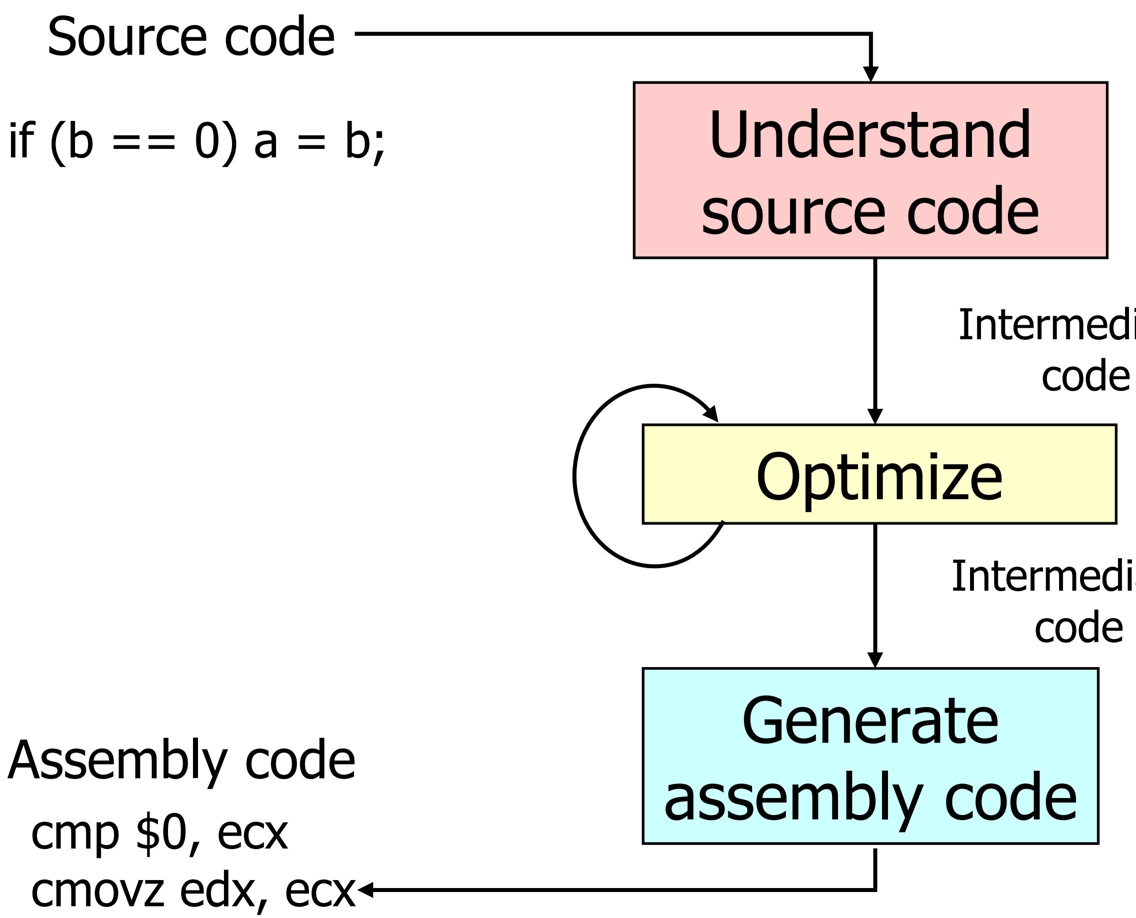
Lecture 2: Lexical Analysis
23 Jan 08

Outline

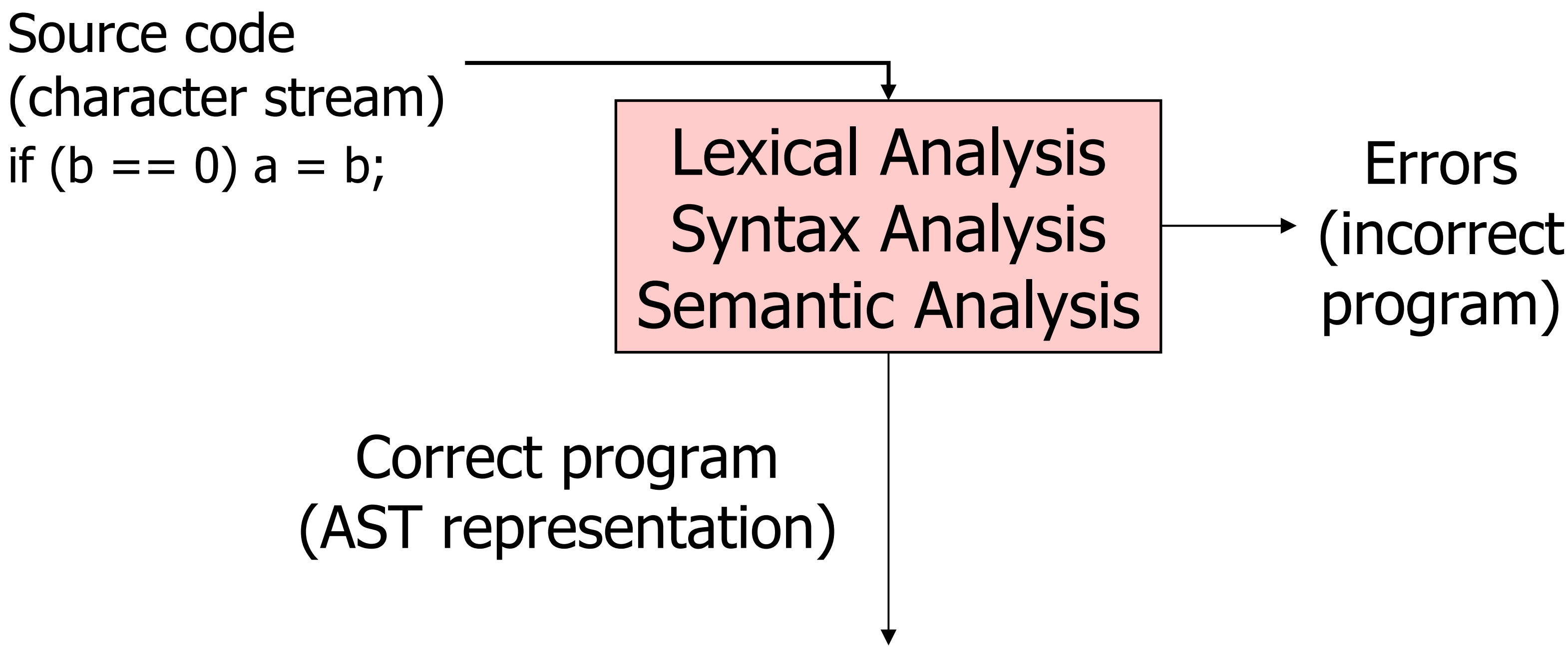
- Review compiler structure

- What is lexical analysis?
- Writing a lexer
- Specifying tokens: regular expressions

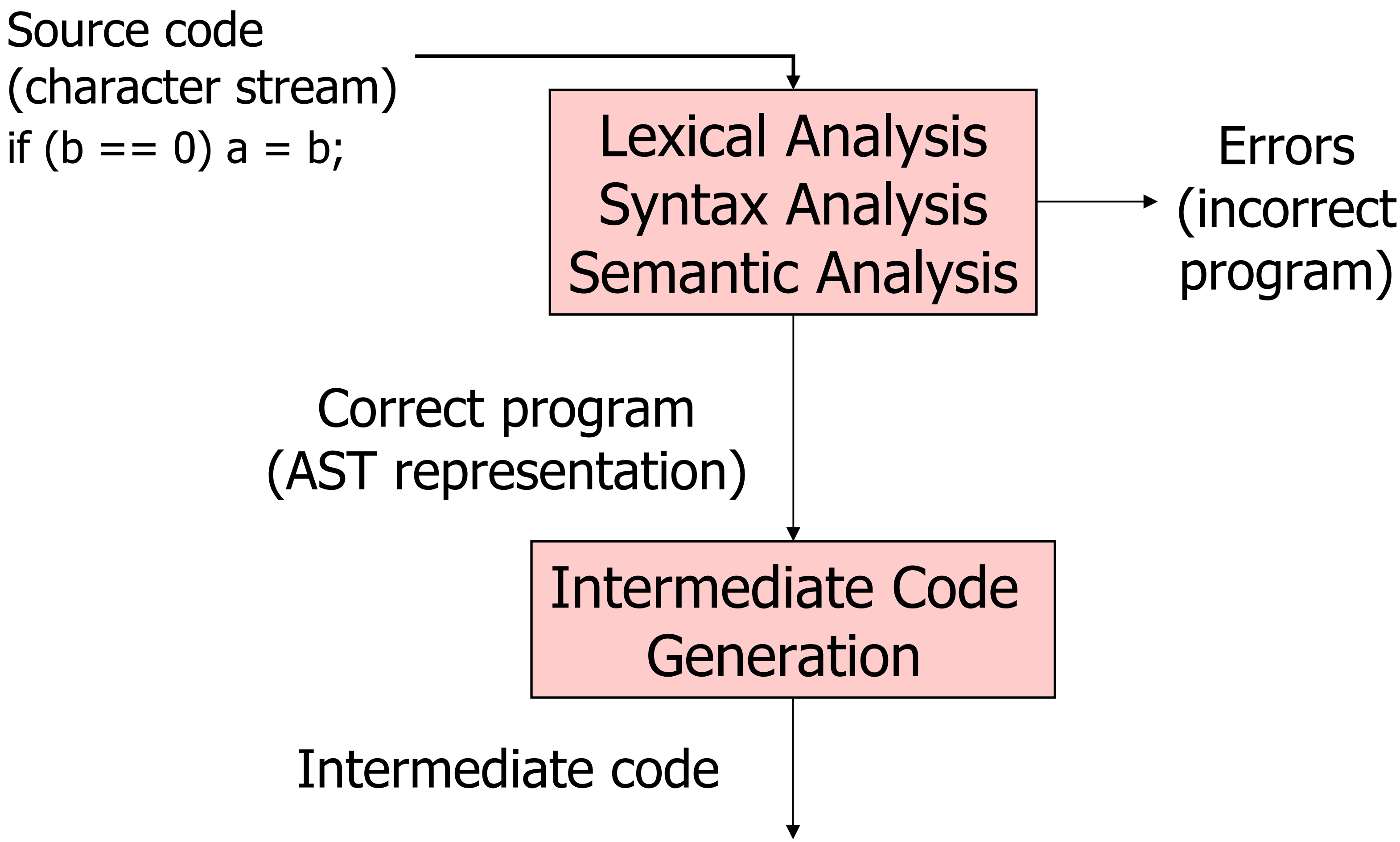
Simplified Compiler Structure



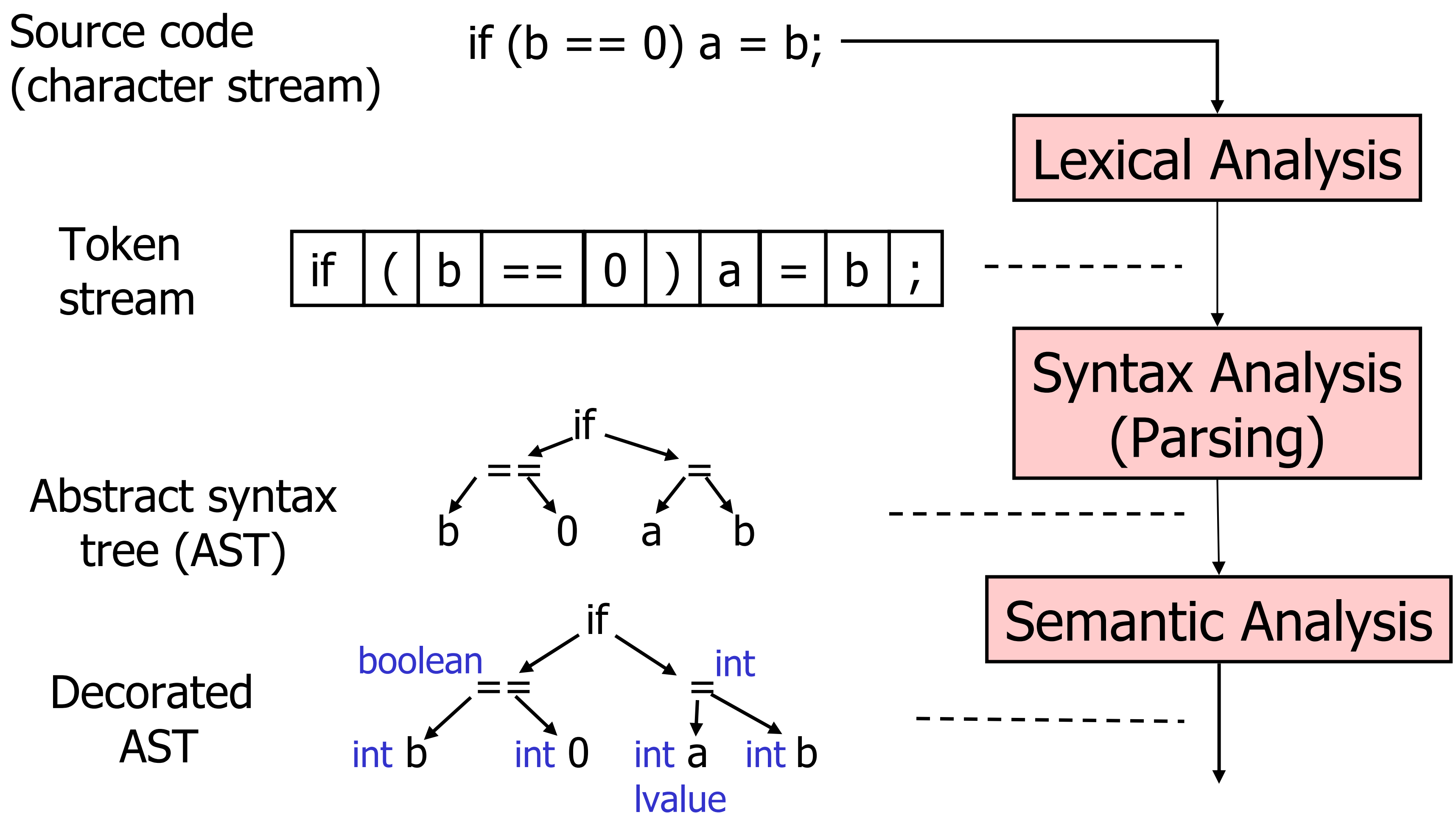
Simplified Front End Structure



More Precise Front End Structure



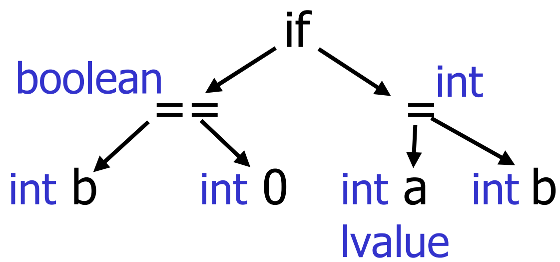
How It Works



Source code
(character stream)
if (b == 0) a = b;
Lexical Analysis
Token
stream
if (b == 0) a = b ;
Syntax Analysis
(Parsing)
Abstract syntax
tree (AST)
if
==
=
b
0
a
b
Semantic Analysis
Decorated
AST
if
boolean
==
int
=
int b
int 0
int a
lvalue
int b

How It Works, cont.

Decorated AST



Intermediate code

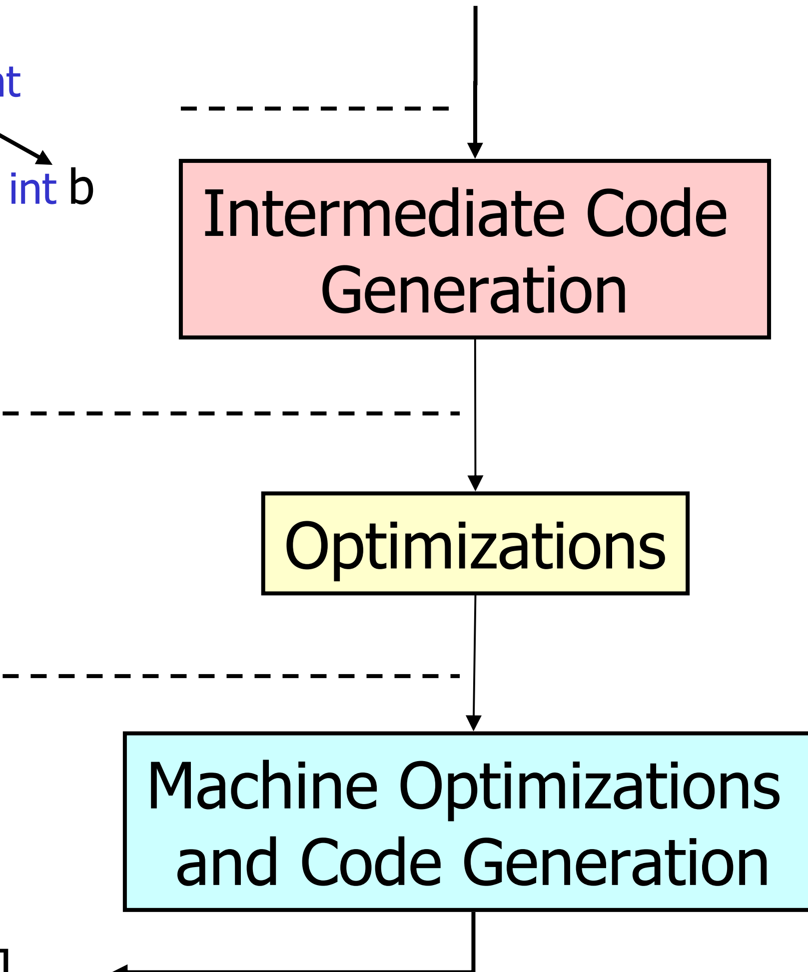
```
t = (b ==0)
jump t, L
a = b
label L
```

Intermediate code

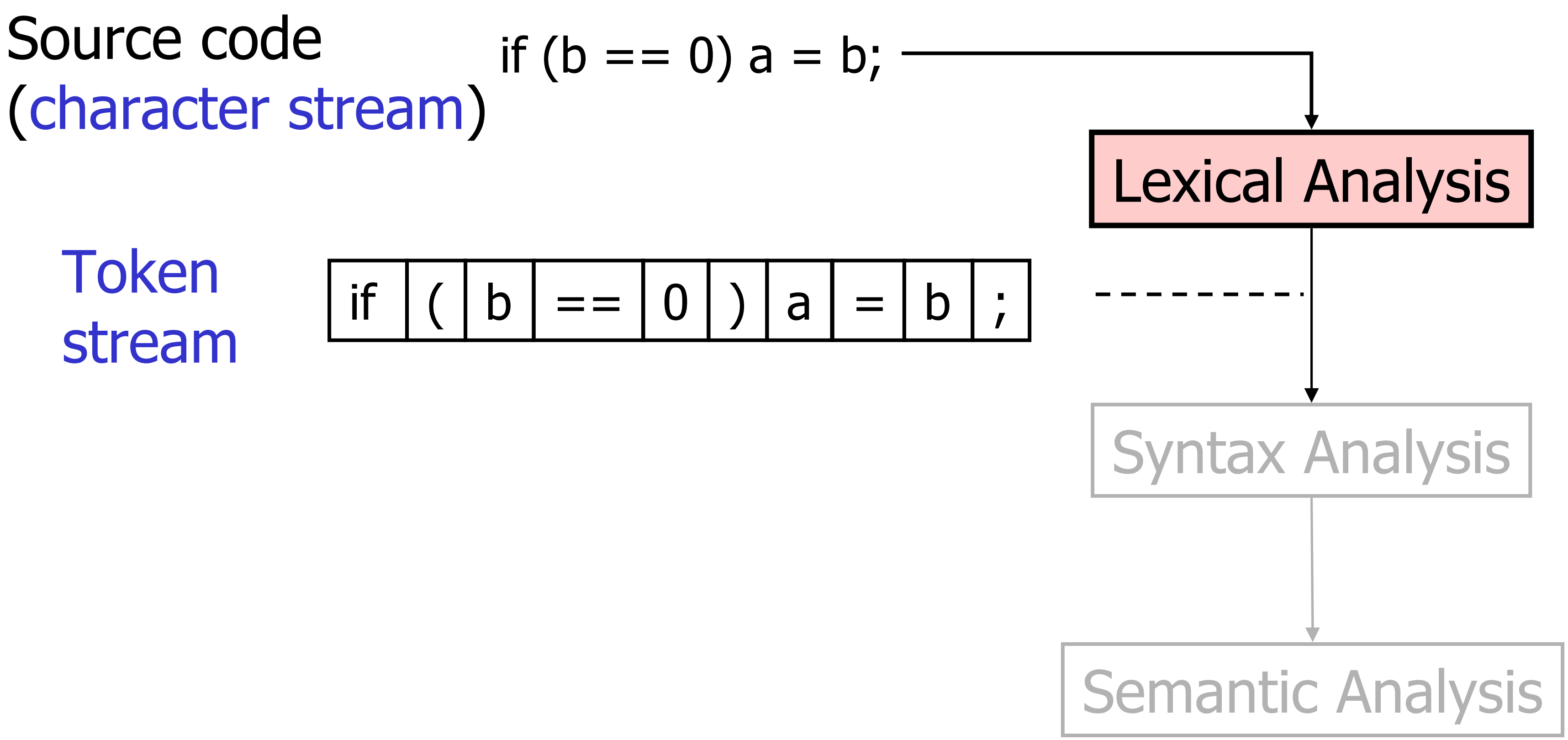
```
t = (b ==0)
jump t, L
a = 0
label L
```

Assembly code

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



First Step: Lexical Analysis



Tokens

- Identifiers: x y11 elsen _i00
- Keywords: if else while break
- Constants:
  - Integer: 2 1000 -500 5L 0x777
  - Floating-point: 2.0 0.00020 .02 1. 1e5 0.e-10
  - String: "x" "He said, \"Are you?\"\n"
  - Character: 'c' '\000'
- Symbols: + * { } ++ < << [ ] >=

- Whitespace (typically recognized and discarded):
  - Comment: /** don't change this **/
  - Space: <space>
  - Format characters: <newline> <return>

Ad-hoc Lexer

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

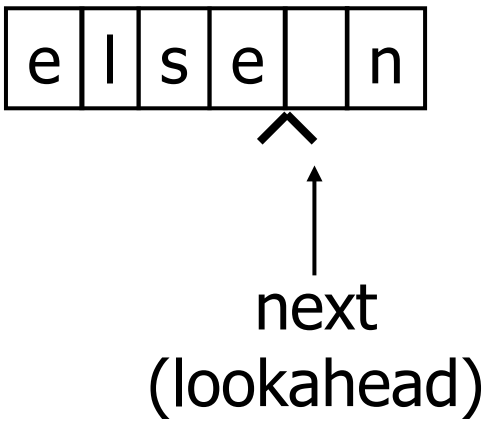
```
Token readIdentifier( ) {
    String id = "";
    while (true) {
      char c = input.read();
      if (!identifierChar(c))
                 return new Token(ID, id, lineNumber);
      id = id + String(c);
    }
}
```

- Problems
  - How to start?
  - What to do with following character?
  - How to avoid quadratic complexity of repeated concatenation?
  - How to recognize keywords?

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 (identifierFirstChar(next))
              return readIdentifier();
        if (numericFirstChar(next))
              return readNumber();
        if (next == '\"') return readStringConst();
        …
   }
}
```

Problems

- Might not 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 hard to write correctly, harder to maintain
  - in general, unbounded lookahead may be needed

Issues

- How to describe tokens unambiguously

  2.e0 20.e-01 2.0000

  “” “x” “\\” “\”\””

- How to break up text 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 ~1 time

Principled Approach

- Need a principled approach
  - lexer generator that generates efficient tokenizer automatically (e.g., lex, flex, JLex)
  - a.k.a. scanner generator
- Approach
  - Describe programming language's tokens with a set of regular expressions
  - Generate scanning automaton from that set of regular expressions

Language Theory Review

- Let Σ be a finite set
  - Σ called an alphabet
  - $a \in \Sigma$ called a symbol
- Σ^* is the set of all finite strings consisting of symbols from Σ
- A subset $L \subseteq \Sigma^*$ is called a language
- If L_1 and L_2 are languages, then $L_1 L_2$ is the concatenation of L_1 and L_2, i.e., the set of all pair-wise concatenations of strings from L_1 and L_2, respectively

Language Theory Review, ctd.

- Let $L \subseteq \Sigma^*$ be a language
- Then
  - $L^0 = \{\}$
  - $L^{n+1} = L\,L^n$ for all $n \geq 0$
- Examples
  - if $L = \{a, b\}$ then
    - $L^1 = L = \{a, b\}$
    - $L^2 = \{aa, ab, ba, bb\}$
    - $L^3 = \{aaa, aab, aba, aba, baa, bab, bba, bbb\}$
    - …

Syntax of Regular Expressions

- Set of regular expressions (RE) over alphabet Σ is defined inductively by
  - Let $a \in \Sigma$ and R, S $\in$ RE. Then:
    - $a \in$ RE
    - $\varepsilon \in$ RE
    - $\varnothing \in$ RE
    - R|S $\in$ RE
    - RS $\in$ RE
    - R* $\in$ RE
- In concrete syntactic form, precedence rules, parentheses, and abbreviations

Semantics of Regular Expressions

- Regular expression $T \in$ RE denotes the language $L(R) \subseteq \Sigma^*$ given according to the inductive structure of T:
  - $L(a) = \{a\}$ — the string "a"
  - $L(\varepsilon) = \{``"\}$ — the empty string
  - $L(\varnothing) = \{\}$ — the empty set
  - $L(R|S) = L(R) \cup L(S)$ — alternation
  - $L(RS) = L(R)\, L(S)$ — concatenation
  - $L(R^*) = \{``"\} \cup L(R) \cup L(R^2) \cup L(R^3) \cup L(R^4) \cup \ldots$ — Kleene closure

Simple Examples

- L(R) = the "language" defined by R
  - L(abc) = { abc }
  - L(hello|goodbye) = {hello, goodbye}
  - L(1(0|1)*) = all non-zero binary numerals beginning with 1

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|…|y|z)
[^ab]	anything but one of the listed chars
[^a-z]	one character not from this range
"abc"	the string "abc"
\(	the character '('
. . .	
id=R	named non-recursive regular expressions

More Examples

Regular Expression R	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”
= (-|ε)([0-9]+)((. [0-9]+)|ε)	
[a-zA-Z_][a-zA-Z0-9_]*	C identifiers
else	the keyword “else”

How To Break Up Text

elsen = 0;

1

else	n	=	0

2

elsen	=	0

- REs alone not enough: need rule(s) for disambiguation
- Most languages: longest matching token wins
- Ties in length resolved by prioritizing tokens
- Lexer definition = RE’s + priorities + longest-matching-token rule + token representation

Historical Anomalies

- PL/I
  - Keywords not reserved
    - IF IF THEN THEN ELSE ELSE;
- FORTRAN
  - Whitespace stripped out prior to scanning
    - DO 123 I = 1
    - DO 123 I = 1   ,   2
- By and large, modern language design intentionally makes scanning easier

Summary

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

Reading

- IC Language spec
- JLEX manual
- CVS manual
- Links on course web home page

Groups

- If you haven't got a full group lined up, hang around and talk to prospective group members today