Outline

- Review compiler structure
- Compilation example
- What is lexical analysis?
- Writing a lexer
-Specifying tokens: regular expressions
- Writing a lexer generator

Simplified Compiler Structure

Source code

if (b == 0) a = b;

Understand source code

Generate assembly code

Assembly code

cmp $0, ecx

cmovz $0, [ebp+8]

Intermediate code

Front End Structure

Source code (character stream)

if (b == 0) a = b;

Lexical Analysis

Syntax Analysis

Intermediate Code Generation

Correct program (AST representation)

Errors (incorrect program)

How It Works

Source code (character stream)

if (b == 0) a = b;

Token stream

if | b == 0 | a = b |

Lexical Analysis

Abstract syntax tree (AST)

if | b == 0 | a = b |

Syntax Analysis (Parsing)

Decorated AST

if | b == 0 | a = b |

Semantic Analysis

Intermediate Code Generation

Intermediate code

1 = b == 0

cmp s, L

a = b

label L

Decorated AST

Intermediate code

1 = b == 0

cmp s, L

a = 0

label L

Assembly code

cmp $0, ecx

cmovz $0, [ebp+8]

Optimizations

Machine Optimizations and Code Generation
First Step: Lexical Analysis

Source code (character stream)

Lexical Analysis

Token stream

Syntax Analysis

Semantic Analysis

Tokens

- Identifiers: x y11 elsen _i00
- Integers: 2 1000 -500 5L
- Floating point: 2.0 0.0020 .02 1.
  1e5 0.e-10
- Strings: "x" "He said, \"Are you?\""
- Comments: /** don't change this **/
- Keywords: if else while break
- Symbols: + * { } ++ < << >=

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);
    id = id + String(c);
  }
}

Look-ahead Character

- Use look-ahead character (next) to:
  - determine what kind of token to read and
  - when the current token ends

char 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

- One character look-ahead might not be enough
  - What token is it if it begins with "1"?
  - What token is it if it begins with "2"?
  - Hard to write tokenizer correctly, harder to maintain

- Need a more principled approach: lexer generator that generates efficient tokenizer automatically (e.g., lex, flex, JFlex)
Issues

• How to describe tokens unambiguously
  2.0 e 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
  - avoid scanning (parts of the) input multiple times

How to Describe Tokens?

• Solution: use regular expressions

  A regular expression \( R \) is defined inductively:
  \( a \) ordinary character stands for itself
  \( \varepsilon \) the empty string
  \( R \) | \( S \) either \( R \) or \( S \) (alternation), where \( R, S \) are REs
  \( RS \) \( R \) followed by \( S \) (concatenation), where \( R, S \) are REs
  \( R^* \) concatenate regular expression \( R \) zero or more times
  \( R^* = \varepsilon | R | RR | RR R | RRRR ... \)

Convenient RE Shorthands

\( R^+ \) one or more strings from \( L(R) \): \( R(R^*) \)
\( R? \) optional \( R \): \( (R\varepsilon) \)
\([abce]\) 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

Simple Examples

• A regular expression \( R \) describes a set of strings of characters denoted \( L(R) \)

\( L(R) = \) the language defined by \( R \)
  - \( L(abc) = \{ abc \} \)
  - \( L(\text{hello|goodbye}) = \{ \text{hello, goodbye} \} \)
  - \( L(1(0|1)^*) = \) all non-zero binary numbers

• We can define each kind of token using a regular expression

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"

• Lexer generators support such abbreviations
  - Abbreviations cannot be recursive

How To Break Up Text

• How do we tokenize "elsen = 0;"?

\( REs \) alone not enough: need rule for choosing
Most languages: longest matching token wins
Ties in length resolved by prioritizing tokens
\( RE's \) + priorities + longest-matching token rule = lexer definition
Historical Issues

- **PL/I**
  - Keywords not reserved
    
    ```
    IF IF THEN THEN = IF; ELSE ELSE = IF;
    ```

- **FORTRAN**
  - White-space insensitivity, limit identifier length:
    ```
    DO 412 I = 1,25
    DO412 I = 1,25
    INTEGER FUNCTION FOO
    ```

- By and large, modern language design intentionally make scanning easier

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