COS 126 Lecture 18: Systems Programming

C program

```
#include <stdio.h>
main()
   {
      printf("hello world ");
   }
```

Compile-execute abstraction

```
cc hello.c
a.out
```

Systems programs that support this abstraction

- assembler
- macro processor
- compiler (interpreter)
- loader
- linker
- "kernel"
- libraries

Programs that manipulate programs

General approach used

- does not vary much across systems
- developed as consensus of many people
- embodies significant intellectual achievements
  - though it makes sense in hindsight
- has had major impact

Assembly Language

Simplify machine language by

adding mnemonic codes for opcodes

using symbolic names for addresses

Ex:

```
.  10: B000       LA R0, 00
.  11: B11C       LA R1, DATA
.  12: B208       LA R2, SIZE
.  13: B301       LA R3, 01
.  14: 2113       S  R1, R1 - R3
.  15: B300       LA R3, 00
.  16: 9D12 LOOP  LI R5, 12
.  17: 2635       S  R6, R3, R5
.  18: 661A       BP R6, * + 2
.  19: 1305       A  R3, R0 + R5
.  1A: 7216       BC R2, LOOP
.  1B: 0000       H
.           SIZE  EQ 8
.           DATA  DS SIZE
```

Advantages over machine language

easier to add, remove instructions

working with names, not numbers

ASSEMBLER:

program to translate from

assembly language to machine language

not difficult to implement

symbol table (BST?) most difficult part

Low-level languages

Why use assembly language?

Plus: good handle on efficiency

Minus: programs too hard to write

No choice in ancient systems

resource restrictions

primitive compilers

complex abstract machines

Assembly language programming is now rare

With modern compiler technology

optimized output may be *better* than hand-coded version

Issue: is C today's assembly language?

Disadvantageous similarities

(language designer's point of view)

low level of abstraction

ability to affect bare machine

separation from high-level functionality

Advantageous similarities

(systems programmer's point of view)

low level of abstraction

ability to affect bare machine

separation from high-level functionality

Note that basic C constructs appear in virtually all languages *except* assembly...

Preprocessor (macro processor)

Ref: Kernighan and Ritchie, p. 228

"Prepare" a program for the C compiler

Certain processing functions

    not conveniently included in C

    still important and useful

Macro definition

```
#define TABSIZE 100
#define less(A, B) A < B
int table[TABSIZE]
if (less(s, t)) ...
```

File inclusion

```
#include <stdio.h>
#include "myfile.c"
```

Conditional compilation

```
#define DEBUG 1
#if DEBUG
printf(...);
#endif
```

INPUT: C program with directives

OUTPUT: C program

low-level, efficient, special purpose program

    can use awk or perl for similar tasks

Compilers

Translate programs
- from high-level language
- to "native" machine language

Three basic phases
- lexical analysis
- syntax analysis
- code generation

User program:
- specification of what user wants

Object module:
- realization on a particular machine

application of theory to a practical problem
interaction between languages and machines
nontrivial software engineering

Lexical Analysis

Convert input into stylized stream of
"tokens" for later phases

Tokens are TERMINAL SYMBOLS
"letters" in the "alphabet" of the language

Ex: (C)

if else while do for int float sizeof ...

{ } ; . -> + - * ++ -- ...

i, j, 123, 12.2, "Hello, world", ...

Lexical analyzers can be implemented with
finite-state-machines

Ex:

```
sequence of blanks/tabs/etc.  -> one blank
sequence of letters           -> "identifier"
sequence of digits            -> "number"
```

lex: Unix utility to do lexical analysis
(runs user-defined FSA)

Syntax Analysis (Parsing)

Describe language with a CONTEXT-FREE GRAMMAR
(for C, Kernighan and Ritchie, pp. 234--239)

Ex (prefix expressions):

```
...
<expr> -> * <expr> <expr> | <val>
<expr> -> + <expr> <expr> | <val>
<val>  -> identifier | constant
```

RECURSIVE DESCENT parser

- subroutine for each nonterminal
- subroutine calls for each production
- input "match" for each terminal

```
void expr()
  {
    if ( token() == ’*’ )
      { tokenptr++; term(); term(); }
    ...
```

- provides structural basis for compilation
- extends to create ABSTRACT SYNTAX TREE
- Ex: Sedgewick, Program 5.20 (see next slide)

Parsers can be described by and implemented with
pushdown automata

yacc: Unix utility to do syntax analysis

Code Generation

ABSTRACT SYNTAX TREE (parse tree)

gives structure of computation

```
.                                    *
.         +                             f
.      a                *
.                 *           +
.                    b     c     d     e
```

Traverse abstract syntax tree in POSTORDER

terminal: emit LOAD instruction

nonterminal: emit ADD or MUL instruction

```
  a  9140
  b  9241
  c  9342
  *  3423
  d  9243
  e  9344
  +  1523
  *  3645
  +  1716
  f  9145
  *  3271
```

Many details omitted (ex: register allocation)

Use intermediate abstract machine

OPTIMIZE code

Compiler-related software tools

INTERPRETER:

- High-level language simulation
- use *any* convenient abstract machine
- output is program output, not translation
- Ex:
  - TOY machine simulator (lecture 6)
  - PostScript
  - shell
  - java
  - emulator (machine-language interpreter)
- usually "slow"
  - source instruction ->
    - many interpreter instructions
  - Ex: lookup names each time they're used?

COMPILER-COMPILER

- program that produces a compiler as output
- ultimate "table-driven" compiler
  - lexical analysis: regular expression
  - syntax analysis: context-free grammar
  - code generation: machine description
- automatically convert tables into code
- useful for
  - research on new languages
  - bootstrapping (first compiler)

Loading

What is "a.out", exactly?

```
COMPILER:   C -> assembly language
ASSEMBLER:  assembly -> relocatable object code
LOADER:     relocatable object code -> executable
```

Addresses in code depend upon where it is loaded

```
B000  B000         LA R0, 00
B11C  B140         LA R1, DATA
B208  B21E         LA R2, SIZE
B301  B301         LA R3, 01
2113  2113         S  R1, R1 - R3
B300  B300         LA R3, 00
9D12  9D12   LOOP  LI R5, 12
2635  2635         S  R6, R3, R5
661A  663B         BP R6, * + 2
1305  1305         A  R3, R0 + R5
7216  7237         BC R2, LOOP
0000  0000         H
```

Compiler can’t know these values

RELOCATABLE OBJECT CODE

machine language code PLUS tables giving which addresses are RELATIVE to start

LOADER translates object code to executable code by adding start to each relative address