Lecture 15

April 2, 1996

Andrea LaPaugh

Elementary Systems Programming

Programs that manipulate programs

- Assembler
- Macro Processor
- Interpreter
- Compiler
- Loader
- Linker
- ``Kernel``

```
#include <stdio.h>

main()
  {
    printf("hello world\n");
  }
```

lcc hello.c

a.out

Assembly Language

Simplify machine language by

adding mnemonic codes for opcodes

using symbolic names for addresses

Easy to add, remove instructions

in assembly language, *not* machine language

ASSEMBLER---program to translate from

assembly language to machine language

Given ``symbol table'' (ex. binary search tree)

to associate names with numbers,

an assembler is not difficult to implement

Assembly language programming done

only when unavoidable

Example

```
  ! add up LENGTH numbers from address TABLE
  ! and store at SUM, assuming LENGTH>0

       LI    R0, 1          ! R0 <- 1, constant
       LI    R1, LENGTH     ! R1 <- LENGTH,
       SUB   R1, R1, R0     ! R1 <- R1 - 1, init. ct.
       LI    R2, TABLE      ! R2 <- TABLE, init. next val.
       LI    R3, 0          ! R3 <- 0 initial. sum
LOOP:  JLT   R1, DONE       ! loop  - done if ct < 0
       LOAD  R4, (R2+0)     ! R4 <- M(R2+0) new value
       ADD   R3, R3, R4     ! sum <- sum + R4
       ADD   R2, R2, R0     ! R2 <- R2 + 1, next value
       SUB   R1, R1, R0     ! R1 <- R1-1, decr. ct.
       JMP   LOOP           ! iterate if ct.>0
DONE:  LI    R1, SUM        ! R1 <- address for sum: SUM
       STORE R3, (R1+0)     ! store sum into SUM
       HALT
LENGTH: EQ  12              ! constant value 12
TABLE:  SPACE LENGTH        ! space for array
SUM:   SPACE 1              ! space for result
```

Preprocessor (macro processor)

Ref: Kernighan and Ritchie, p. 88-92

``Prepare`` a program for the C compiler

macro definition

file inclusion

conditional compliation

Certain processing functions

not conveniently included

in ``the language``

still important

Macro definition

```
#define TABSIZE 100
#define forever for (;;)
```

File inclusion

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

Conditional compilation

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

Assessing Macros

With macros, programmer (or reader)

has to think about program at TWO levels

instead of just one:

-- rewriting program text (macros)

-- control and data structures (program)

Many programming languages have constructs that

can do the work of macros

(see "enumeration constants" in C)

Macros often used in assembly language.

Interpreter

Simulate programs

written for one abstract machine

to be executed on another machine

Ex.

TOY machine simulator

PostScript

shell

Usually ``slow``

each source instruction

requires many interpreter instructions

(select, call subroutine, ``execute``, etc.)

translate names each time they're used?

contrast with compiler

Can be interactive

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

Lexical Analysis

Convert input into stylized stream of

``tokens`` for later phases

Examples

sequence of blanks/tabs/etc. -> one blank

sequence of letters -> ``identifier``

sequence of digits -> ``number``

(lex: Unix utility to do lexical analysis)

Syntax Analysis (Parsing)

Convert string of tokens into ``parse tree``

proves program is ``legal``

provides structural basis for compilation

Have set of rules for how strings can legally be formed

called ``context-free grammar``

example:

```
exp -> exp + exp
exp -> exp - exp
exp -> (exp)
exp -> constant
```

``exp:`` is ``nonterminal`` -- can be broken down

``+`` is terminal

``Recursive Descent`` parser

subroutine for each nonterminal

subroutine calls for each rule

``match`` input for each terminal

(yacc: Unix utility to do syntax analysis)

Parse tree

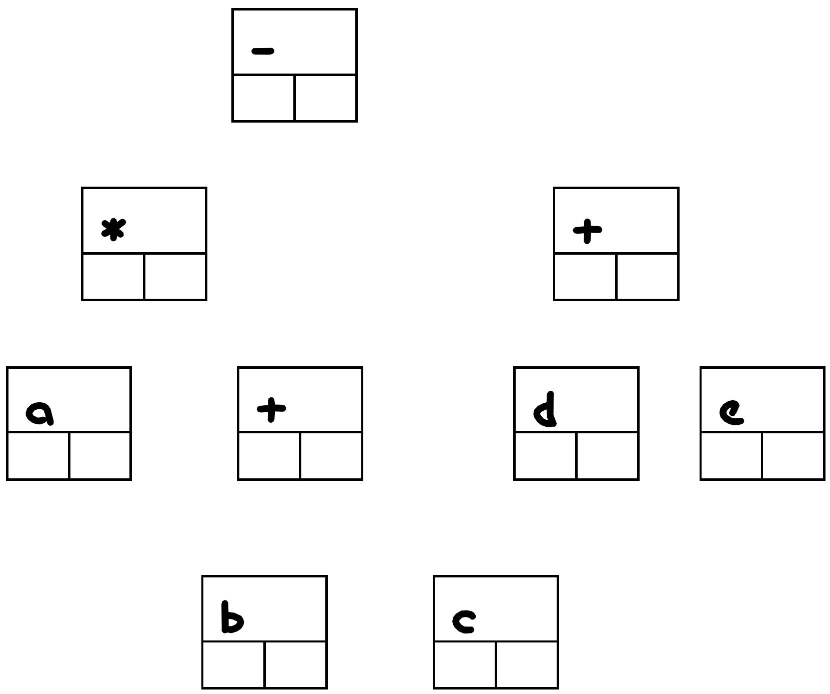
Another example of a binary tree

Each "internal node" contains a terminal symbol
that is an operator (+, − , *)

Each "leaf" contains a terminal symbol
that has a value

Example:

$(a * (b + c)) - (d + e)$



Code Generation

Traverse parse tree, emmitting code

Can use any convenient abstract machine

Ex. Stack machine for expressions

"Optimize" emitted code

Convert to native machine

optimize final code

A Simple Compiler

takes a fully parenthesized expression

using variables a,b,c,...,p and operators +,-,*

prints reverse-polish-notation version

of expression;

example:

```
% compile '(a+((b*c)+d))'
abc*d++
```

Character strings in C

But first we need to know how
to implement character strings
(K&R pgs 104 - 107)

no special "character string" type in C
- use array of characters
- "terminate" string with special character
  - "null character:" `'\0'`

can write string constants:

```
"hello, world"
```

can use

```
char *stringptr;
```

or

```
char stringArray[MAX_LENGTH];
```

(allocate different memory)

`stringptr[i]` is same as `*(stringptr + i)`

`stringArray[i]` is same as `*(stringArray + i)`

library of string functions: `#include <string.h>`

Simple compiler - preliminaries

```
#include <stdio.h>
#include <stdlib.h>
```

```
/* print error message and never come back! */
```

```
void error(char *message)
{ printf("%s, bye!\n",message);
  exit(1);
}
```

Defining parse tree

/* Data structure for parse trees;

oper can be +,-,*, or a,b,c...p

left, right are both NULL for a,b,c...

non-NULL for +,-,*

*/

```
struct tree {char oper; struct tree *left, *right; };
```

/* make a tree node and initialize it */

```
struct tree *maketree(char operation,
               struct tree *leftchild, struct tree *rightchi
{
 struct tree *t;
 t = malloc(sizeof (struct tree));
 if (t==NULL) error("out of memory");
 (*t).oper=operation;
 (*t).left=leftchild;
 (*t).right=rightchild;
 return t;
}
```

Lecture 16

Compilers, continued

April 4, 1996

Andrea LaPaugh

Trivial lexical analysis

```
char *input;  /* entire input string;
          examined only by get and look */
int pos;  /*  current position within input;
         Modified only by get, examined by look */

/* check that the next input char matches
  one of the ``options``; advance the
  input ``pos``, return the char found. */

char get(char *options)
{int i;
 for(i=0; options[i] != '\0'; i++)
   if (input[pos]==options[i])
      {pos++;
       return options[i];
      }
 error("syntax error");
}

/* tell the next character without advancing */

char look(void)
{ return input[pos]; }
```

Parsing

/* parse a fully parenthesized expression

 into a parse tree */

```
struct tree *expr(void)
{
 char paren;
 struct tree *l;
 struct tree *r;
 char oper;
 char var;

 if (look()=='(')
     { paren = get("(");
       l = expr();
       oper = get("+-*");
       r = expr();
       get(")");
       return maketree(oper,l,r);
     }
 else { var = get("abcdefghijklmnop");
        return maketree(var,NULL,NULL);
      }
}
```

Parsing, cont.

/* parse an entire formula */

```
struct tree *parse(void)
{
 struct tree *t;
 pos=0;
 t=expr();
 if (look() != '\0')
       error("garbage at end of input");
 return t;
}
```

Code Generation for Stack Machine

Reverse Polish notation

/* print a parse tree in postorder */

```
void postorder(struct tree *t)
{
 if (t != NULL)
   {postorder(t->left);
    postorder(t->right);
    printf("%c",t->oper);
   }
}
```

Code Generation for Toy

We could print Toy machine language from parse tree.

op code -- straight translation

+-*   123

registers -- register allocation

depends on how many you have

depends how many "intermediate results"

memory -- use consecutive memory locations for variables