Lecture 19. Compilers

• The *compiler* translates a high-level language to a machine-level language

\[ \text{lcc: } \ C \rightarrow \text{SPARC assembly language} \rightarrow \ldots \rightarrow \text{SPARC machine code} \]
\[ \text{compile: } \ \text{arithmetic expressions} \rightarrow \text{TOY instructions} \]

• Most compilers have the basic phases

  Lexical Analysis  \quad \text{source code} \rightarrow \text{‘tokens’}
  Syntax Analysis  \quad \text{tokens} \rightarrow \text{abstract syntax trees}
  Code Generation  \quad \text{abstract syntax trees} \rightarrow \text{machine-level code}

• A compiler is a good example of

  Application of theoretical computer science to a practical problem
  Interaction between programming language design and computer architecture
  Building a program from independent modules — ‘software engineering’

• For *much* more

  Take COS 320, Compiler Design
  Read C. W. Fraser and D. R. Hanson, *A Retargetable C Compiler: Design and Implementation*, Addison-Wesley, 1995
Lexical Analysis

- The lexical analyzer reads the source program and emits tokens or terminal symbols: the ‘letters’ in the ‘alphabet’ of the programming language

  English:
  
  a b c d e f g h ... A B C ... ; ‘ ’ ! : — - ( ) ...

  C tokens:
  
  if else while do for int float sizeof ...
  
  { } ; . -> + - * / % ++ -- < <= == != & ^ | ~ >= > ( ) ...
  "strings" constants identifiers ...

  Simple arithmetic expressions:
  
  ( ) + - *
  
  one-letter identifiers one-digit constants

- A lexical analyzer usually discards white space: blanks, tabs, newlines, etc.

- Lexical analyzers can be described by and implemented with finite-state machines
Syntax Analysis

- A **context-free grammar** specifies how tokens can be formed into valid ‘sentences’

  Grammar rules or ‘productions’ specify how to generate all valid sentences

1. \( pgm \rightarrow expr \)
2. \( expr \rightarrow expr + expr \)
3. \( expr \rightarrow expr - expr \)
4. \( expr \rightarrow expr \ast expr \)
5. \( expr \rightarrow ( expr ) \)
6. \( expr \rightarrow identifier \)
7. \( expr \rightarrow constant \)

\( pgm \) and \( expr \) are ‘nonterminals’ — they describe **classes** of valid sentences

+ − ∗ ( ) identifier constant are terminals or tokens — the basic vocabulary

1. \( pgm \Rightarrow expr \)
   2. \( \Rightarrow expr - expr \)
   3. \( \Rightarrow ( expr ) - expr \)
   4. \( \Rightarrow ( expr \ast expr ) - expr \)
   5. \( \Rightarrow ( a \ast expr ) - expr \)
   6. \( \Rightarrow ( a \ast ( expr + expr ) ) - expr \)
   7. \( \Rightarrow ( a \ast ( b + 2 ) ) - expr \)
   5. \( \Rightarrow ( a \ast ( b + 2 ) ) - ( expr ) \)
   2. \( \Rightarrow ( a \ast ( b + 2 ) ) - ( expr + expr ) \)
   6. \( \Rightarrow ( a \ast ( b + 2 ) ) - ( c + expr ) \)
   7. \( \Rightarrow ( a \ast ( b + 2 ) ) - ( c + 9 ) \)
Parsers

- A **parser** determines if a sentence can be generated by the grammar rules
  Proves that the sentence is syntactically valid

- A parser may also build an **abstract syntax tree** to represent the sentence
  
  \[(a \times (b + 2)) - (c + 9)\]

  **Internal nodes** hold terminal symbols that denote operators: `+ - *`

  **Leaf nodes** hold terminal symbols that denote variables or constants: `a b c 2 9`

- A ‘recursive-descent’ parser has a function for each nonterminal
  ‘Matches’ terminals in input
  Calls other nonterminal functions — including itself — to apply the rules

- Parsers can be described by and implemented with **pushdown automata**
Code Generation

• A **code generator** traverses the abstract syntax tree and emits code, e.g., TAL, or TOY instructions

```
% lcc -I/u/cs126/include compile.c /u/cs126/lib/libmisc.a
% a.out 5 6 7 "(a * (b + 2)) - (c + 9)"
00: 0005
01: 0006
02: 0007
1A: 9100        R1 <- M[R0+0]
1B: 9201        R2 <- M[R0+1]
1C: B302        R3 <- 2
1D: 1223        R2 <- R2 + R3
1E: 3112        R1 <- R1 * R2
1F: 9202        R2 <- M[R0+2]
20: B309        R3 <- 9
21: 1223        R2 <- R2 + R3
22: 2112        R1 <- R1 - R2
23: 4102        print R1
24: 0000        halt
1A
```

• This compiler — and only this one — bypasses assembly, linking, and loading

```
% a.out 5 6 7 "(a * (b + 2)) - (c + 9)" /u/cs126/toy/toy
ab2+*c9+-
Toy simulator $Revision: 1.14 $
0018
```
A Simple Compiler

• Lexical analyzer: returns characters as tokens

```c
int get(char set[]) returns the next token, advances the input
int look(void) peeks at the next nonblank character
```

• Parser: returns an abstract syntax tree (AST)

```c
Tree *expr(void) parses an expr, returns its AST
Tree *pgm(char *string) initializes lexer, parses a pgm, returns its AST
```

```c
struct tree {
  int op;
  struct tree *left, *right;
};
typedef struct tree Tree;

Tree *maketree(int op, Tree *left, Tree *right) {
  Tree *t = emalloc(sizeof (Tree));
  t->op = op;
  t->left = left; t->right = right;
  return t;
}
```

• Code generator: emits TOY instructions

```c
int codegen(Tree *t, int dst, int loc) emits TOY code for AST t starting at loc
```
Lexical Analysis

• Globals hold the ‘state’ of lexical analysis: input and current input position

```c
char *input;    /* the "source code" */
int pos;       /* current position in input */

input[pos] holds the next character in the input
```

• The next token is the next non-whitespace character, which must be in set

```c
int get(char set[]) {
    while (isspace(input[pos]))
        pos++;
    if (input[pos] != '\0' && strchr(set, input[pos]) != NULL)
        return input[pos++];
    error("syntax error: expected one of '%s'\n", set);
    return 0;
}
```

• The parser must peek ahead one character to determine its next action

```c
int look(void) {
    while (isspace(input[pos]))
        pos++;
    return input[pos];
}
```
Parsing

• The parsing functions for expr and pgm echo their grammar rules

Tree *expr(void) {
    Tree *t;

    if (look() == '(') { /* expr → ( expr ) */
        get('"'); t = expr(); get('"');
    } else if (isdigit(look())) /* expr → constant */
        t = maketree(get("0123456789"), NULL, NULL);
    else /* expr → identifier */
        t = maketree(get("abcdefghijklmnopqrstuvwxyz"), NULL, NULL);
    if (look() != '\'0' && strchr("+-*", look()) != NULL) {
        int op = get("+-*"); /* expr → expr [+-*] expr */
        t = maketree(op, t, expr());
    }
    return t;
}

Tree *pgm(char *string) {
    Tree *t;

    input = string; /* initialize lexical analyzer */
    pos = 0;
    t = expr(); /* pgm → expr */
    if (look() != '\'0')
        error("expected end of input\n");
    return t;
}
Reverse Polish Notation

- A postorder traversal of the AST yields a reverse Polish rendition of the expression

```c
void postorder(Tree *t) {
    if (t != NULL) {
        postorder(t->left);
        postorder(t->right);
        fprintf(stderr, "%c", t->op);
    }
}
```

(a * (b + 2)) - (c + 9)  

- Reverse Polish can be evaluated: a stack holds operands and intermediate values

```
Stack→ R1  R2  R3
a b 2 + * c 9 + -  5  5
a b 2 + * c 9 + -  5 6  5  6
a b 2 + * c 9 + -  5 6 2  5  6  2
a b 2 + * c 9 + -  5 8  5  8
a b 2 + * c 9 + -  40 40
a b 2 + * c 9 + -  40 7  40  7
a b 2 + * c 9 + -  40 7 9  40  7  9
a b 2 + * c 9 + -  40 16  40  7  16
a b 2 + * c 9 + -  24 24
```

- Instead of evaluating the expression, generate code, using registers for the stack
Code Generation

- **codegen** emits code to evaluate an AST into register \( \text{dst} \), assuming higher numbered registers are free

```c
int codegen(Tree *t, int dst, int loc) {
    if (isalpha(t->op)) {
        int addr = t->op - 'a';
        printf("%02X: 9%X%X%X \(\text{R%d}\) <- M[\text{R%d}+%d] \n", loc++,
            dst, 0, addr, dst, 0, addr);
    } else if (isdigit(t->op))
        printf("%02X: B%X%02X \(\text{R%d}\) <- %d \n", loc++,
            dst, t->op - '0', dst, t->op - '0');
    else {
        loc = codegen(t->left, dst, loc);
        loc = codegen(t->right, dst + 1, loc);
        printf("%02X: %X%X%X \(\text{R%d}\) <- R%d %c R%d \n", loc++,
            strchr("+1-2*3", t->op)[1] - '0', dst,
            dst, dst + 1, dst, dst, t->op, dst + 1);
    }
    return loc;
}
```

Variables \( a \ldots z \) are stored in locations 0..19_{16}

\( \text{loc} \) is the location counter: the address of the next instruction emitted

**codegen** returns an updated value of \( \text{loc} \) for use by subsequent traversals
The Main Program

- The final touches

Arguments 1..argc-2 are the initial values of the corresponding variables

Argument argc-1 is the ‘source program’

Starting address is 26_{10} = 1A_{16}

```c
int main(int argc, char *argv[]) {
    Tree *e;
    int i, loc = 0;

    for (i = 1; i < argc - 1; i++)
        printf("%02X: %04X\n", loc++, atoi(argv[i]));
    if (i < argc) {
        e = pgm(argv[i]);
        postorder(e);
        fprintf(stderr, "\n");
        loc = codegen(e, 1, 26);
        printf("%02X: 4102	print R%d\n", loc++, 1);
        printf("%02X: 0000	halt\n", loc);
        printf("%02X\n", 26);
    }
    return 0;
}
```

See page 19-5 for an example of use