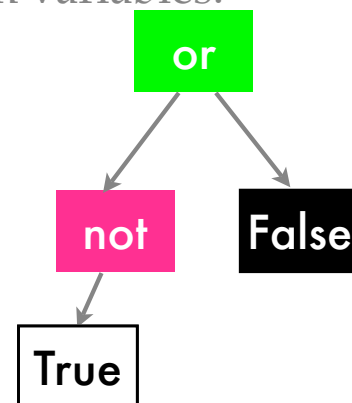


Lecture 9: Compilers

CS105: Great Insights in Computer Science
Michael L. Littman, Fall 2006

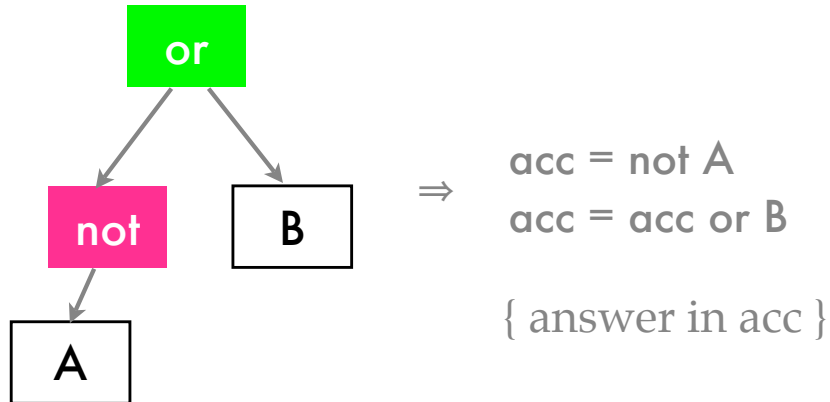
Expressions With Variables

- evaluateTree: Took an expression tree (with Trues and Falses) as input and returned its value.
- What about an expression with variables: (not A or B) ?
- If we know A and B's values, can substitute them in and use evaluateTree!
Interpreter.



Tree \Rightarrow Program

- If A and B are not known, but we still want to do something useful, we can convert the expression tree into a program that, given A and B, produces the value of the expression!



Compiler

- An interpreter takes a program as input and makes it happen.
- A compiler takes a program as input and creates a machine-language program as output.
- A program that converts a program into a program—*twisted*, but useful!

Game Plan

- We'll develop two schemes that compute the value of the expression.
 - One leaves the final value in "acc".
 - The other leaves it in a variable in memory.
- Both are given a list of variables they can use for temporary storage.
- Mutually recursive (ooh).

Code Generation: Var

- Expression: A

answer in acc

acc = A

answer in P

acc = A

P = not acc

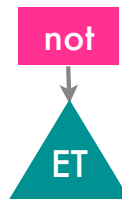
A

Code Generation: not

- Expression: not $EXPR$

answer in acc (temp N)
generate code for
 $EXPR$, answer in N
 $acc = \text{not } N$

answer in P
generate code for
 $EXPR$, answer in acc
 $P = \text{not acc}$



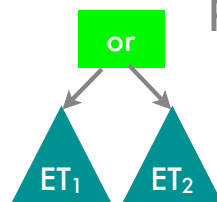
- Note: We keep a pool of temporary variables to use as needed (not just N).

Code Generation: or

- Expression: $(EXPR_1 \text{ or } EXPR_2)$

answer in acc (temp N)
generate code for
 $EXPR_1$, answer in N
generate code for
 $EXPR_2$, answer in acc
 $acc = acc \text{ or } N$

answer in P
generate code for
 $EXPR_1$, answer in P
generate code for
 $EXPR_2$, answer in acc
 $P = acc \text{ or } P$



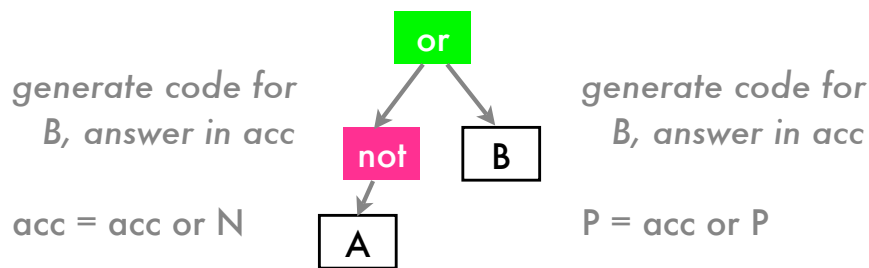
- Note: "and" is handled the same way.

Example Expression

- Expression: (not A or B)

answer in acc
generate code for
not A, answer in N

answer in P
generate code for
not A, answer in P



Assembler

- An assembler handles the last little step of translating the series of instructions to a series of numbers.

answer in acc
acc = A 32
N = not acc 125
acc = B 33
acc = acc or N 13

answer in P
acc = A 32
P = not acc 127
acc = B 33
P = acc or P 79

Complete Code

```
def compileToVar(tree,temps,target):
    if root(tree) == 'V': return [LOAD(ACC(VAR(tree)))] + [STORE(ACC(target))]
    if root(tree) == 'not':
        return compileToAcc(subtree(tree),temps+[target]) + [STORE(NOT(target))]
    if root(tree) == 'or':
        finalCmd = STORE(OR(target))
    if root(tree) == 'and':
        finalCmd = STORE(AND(target))
    return compileToVar(leftSubtree(tree), temps, target)
        + compileToAcc(rightSubtree(tree),temps) + [finalCmd]

def compileToAcc(tree,temps):
    if root(tree) == 'V': return [LOAD(ACC(VAR(tree)))]
    assert(len(temps)>0)
    if root(tree) == 'not':
        return compileToVar(subtree(tree),temps[1:], temps[0]) + [LOAD(NOT(temps[0]))]
    if root(tree) == 'or':
        finalCmd = LOAD(OR(temps[0]))
    if root(tree) == 'and':
        finalCmd = LOAD(AND(temps[0]))
    return compileToVar(leftSubtree(tree), temps[1:], temps[0])
        + compileToAcc(rightSubtree(tree),temps) + [finalCmd]
```

Inefficiency

- (not A or B)
- Automatically generated machine code:

```
answer in acc
acc = A          32
N = not acc     125
acc = B          33
acc = acc or N  13
```

- By hand:

```
answer in acc
acc = not A     48
acc = acc or B  1
```

- Often more than one way to do it!

Optimizations

- Many ways of speeding up compiled code have been developed.
- Want to minimize steps, temporary variables.
- I'll describe two important ones:
 - Shared subexpressions
 - Logical equivalence

Automatic Code (13 inst.)

- $E = ((A \text{ and } B) \text{ and } C) \text{ or } ((A \text{ and } B) \text{ and } D)$
 - $E = (A \text{ and } B) \text{ and } C$
 - $E = A \text{ and } B$
 - $\text{acc} = A$
 - $E = \text{acc}$
 - $\text{acc} = B$
 - $E = \text{acc} \text{ and } E$
 - $\text{acc} = C$
 - $E = \text{acc} \text{ and } E$
 - $\text{acc} = (A \text{ and } B) \text{ and } D$
 - $N = A \text{ and } B$
 - $\text{acc} = A$
 - $N = \text{acc}$
 - $\text{acc} = B$
 - $N = \text{acc} \text{ and } N$
 - $\text{acc} = D$
 - $\text{acc} = \text{acc} \text{ and } N$
 - $E = \text{acc} \text{ or } E$
- 1 temps, 13 instructions

Shared Subexpression

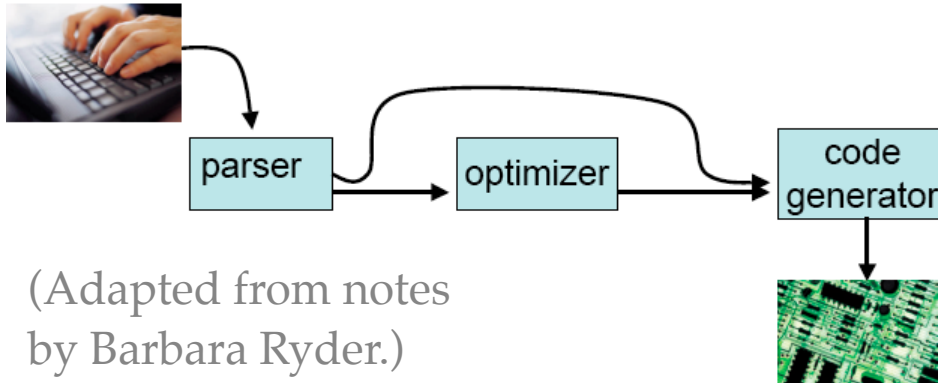
- $E = ((A \text{ and } B) \text{ and } C) \text{ or } ((A \text{ and } B) \text{ and } D)$
 - $N = A \text{ and } B$
 - $E = \text{acc}$
 - $N = A$
 - $\text{acc} = C$
 - $\text{acc} = A$
 - $E = E \text{ and } \text{acc}$
 - $N = \text{acc}$
 - $\text{acc} = N \text{ and } D$
 - $\text{acc} = B$
 - $\text{acc} = N$
 - $N = N \text{ and } \text{acc}$
 - $O = \text{acc}$
 - $E = (N \text{ and } C) \text{ or } (N \text{ and } D)$
 - $\text{acc} = D$
 - $E = N \text{ and } C$
 - $\text{acc} = \text{acc} \text{ and } O$
 - $\text{acc} = N$
 - $E = E \text{ or } \text{acc}$
- 2 temps, 13 instructions

Logical Equivalence

- $E = ((A \text{ and } B) \text{ and } C) \text{ or } ((A \text{ and } B) \text{ and } D)$
 - $N = \text{acc}$
- $E = (A \text{ and } B) \text{ and } (C \text{ or } D)$
 - $\text{acc} = D$
 - $\text{acc} = \text{acc} \text{ or } N$
 - $E = E \text{ and } \text{acc}$
 - $E = A \text{ and } B$
 - $\text{acc} = A$
 - $E = \text{acc}$
 - $\text{acc} = B$
 - $E = E \text{ and } \text{acc}$
 - $\text{acc} = C \text{ or } D$
 - $\text{acc} = C$
- 1 temps, 9 instructions

A Compiler

- A program that translates computer programs that people write into a machine language instructions for the computer to execute.



- (Adapted from notes by Barbara Ryder.)

Parser

- Programs are written in a high-level language such as Java or C++
 - A *grammar* description of the programming language describes a well-formed program
- Parsers check that a program adheres to the rules of the programming language's grammar.
- If so, parser translates the program into an internal representation used by the compiler

Code Generator

- Translates the internal representation of a program into a specific machine language.
- Has all the info it needs in the internal representation and knows the program is correct according to the rules of the grammar.
- Can change to a different computer chip with a different instruction set by changing code generators, without other changes to the compiler.

Summing Up

- Parser uses grammar rules to check expressions for correct structure -- syntax
- If correct, then builds the expression graphs
- Optimizes the graphs to find repeated subexpressions and constants that can be evaluated at compile-time
- Then generates code from the graph

Interpreters

- *Compiler* translates program to machine lang.
- *Interpreter* translates statements by executing equivalent commands
 - No real translation step
- Interpretation requires programming language have a defined meaning for its statements---*semantics*
 - Sometimes defined mathematically, sometimes in English.

How Are They Useful?

- Allow prototyping of new langs.
 - Get to test out quickly (e.g., Scheme, Prolog, Java).
- Achieves portability / universality for a PL
 - Code to be interpreted by a Virtual Machine (VM)
 - Can install the PL on a different machine (i.e., chip) merely by rewriting the VM
 - As long as spec is careful (syntax/semantics), programs should work equivalently!
 - Model for Java (e.g., JVM - Java Virtual Machine)

Java

- Language definition ~mid-1990's
- Used to write applications built out of pieces (e.g., libraries, components, middleware)
 - Built by different people, in different places, on different machines
 - Works because of VM mechanism
- Interpretation frees user from worries about machine-dependent translation details.

Some History of Langs.

- 1950's
 - Machine language programming
 - Scientific computation in Fortran with first compilers
 - LISP for non-numerical computation
- 1960's
 - First optimizing Fortran compiler (IBM)

Some History of Langs.

- 1970's
 - First program analyses enable complex optimizations
 - C language and UNIX (Linux is a form of UNIX)
 - Optimizing for space and time savings

Some History of Langs.

- 1980's
 - First widely-used object-oriented langs. - Smalltalk, C++
 - Compilers translate for parallel computers (e.g., Connection Machine, Cray)
 - Langs. allowing explicit parallelism (i.e., use of multiple processors; Ada)

Some History of Langs.

- 1990's
 - Birth of the Internet
 - PLs for explicitly distributed computation (e.g., across machines in a network)
 - Object-oriented langs. - Java (VMs)
- 2000's
 - Compiling for low power
 - Special purpose (domain specific) langs
 - Scalability, distributed computation, ubiquity

Next Time

- Subroutines, recursion.
- Hillis Chapter 5, Section 1.