LANGUAGES AND COMPILER DESIGN PART II

Topics

• Semantics
• Interpreters
• Runtime Organization
• Intermediate Code Generation
• Machine Code Generation
• Optimization

Project

• Build **fab** Interpreter
• Complete **fab** Compiler for X86-64

Themes

• Mapping from high-level to low-level
• Implementing resource management
• Integration with OS and hardware environment
• Syntax-directed techniques
**Compiler Back-end Tasks**

Starting from type-checked abstract syntax for source language...

- Simplify expressions and statements into flat goto/label form
- Fix location of variables & temps in memory & registers
- Generate machine instructions
- Manage machine resources
- Interact with O/S, runtime system

...generate machine code for specific target architecture.

Or, build interpreter for higher-level language features.
a := if c > 7 and b = e
then 3 * (d + e)
else -c * (d + 2)

Source code

AST

:=

a

if

:=

and

3

*

+

d

e

-

+

c

d

2
Intermediate Code Generation

Emit intermediate code/representation ("IR") from abstract syntax or directly from parser.

Advantages:
- Keeps more of compiler machine-independent
- Facilitates some optimizations

Typical examples:
- Postfix stack code
- Trees or directed acyclic graphs (DAGs)
- Three-address code (quadruples, triples, etc.)

Abstracts key features of machine architectures
- e.g., sequential execution, explicit jumps
- but hides details
- e.g., # of registers, style of conditionals, etc.

Many possible levels of IR; some compilers use several


**“THREE-ADDRESS CODE” - A TYPICAL LINEAR IR**

Generate list of “instructions”

- Each has an operator, up to 2 args, and up to 1 result
- Instructions can be labeled
- Operands are names for locations in some abstract memory (e.g., symbol table entries)

Examples of instructions:

- \( A := B \) copy
- \( A = B \ op \ C \) binary ops
- \( A = \ op B \) unary ops
- goto L jumps
- if \( A \ relop B \) goto L conditional jumps
- param A procedure call setup
- call P,N procedure call
- return N procedure return
- \( A[I] \) array dereference
3-ADDRESS CODE EXAMPLE

- Linearized
- Nested conditionals expanded (badly)
- Temporaries for all intermediate results

if c > 7 goto L1
goto L2
L1: if b = e goto L3
L2: t1 := d + 2
t2 := -c
t3 := t1 * t2
goto L4
L3: t1 := d + e
t3 := 3 * t1
L4: a := t3
“Read” IR and generate assembly language (symbol or binary).
Must cooperate with IR to define and “enforce” runtime environment.
Must deal with idiosyncrasies of target machine,
• e.g., instruction selection
and perform resource management,
• e.g., register assignment.
Lots of case analysis, especially for complex target architectures.
Can do by hand, but hard.
Tools limited but sometimes useful; mainly based on pattern matching
SAMPLE MACHINE CODE

Assumes a global; b, c args; d, e locals.
Illustrates register conventions, condition code use, arithmetic tricks, ...

```
movl %edi, %ebx
movl %esi, %r12d
cmpl $7, %r12d
setg %dl
cmpl %ebx, %ecx
sete %al
testb %al, %dl
je L2
leal (%r13,%rcx), %eax
leal (%rax,%rax,2), %edx
jmp L1

L2:
  movl $-2, %eax
  movl %r12d, %edx
  subl %r13d, %eax
  imull %eax, %edx

L1:
  movq _a@GOTPCREL(%rip), %rax
  movl %edx, (%rax)
```
“OPTIMIZATION”

Improve (don’t perfect) code by removing inefficiencies:

- in original program
- introduced by compiler itself

Can operate on source, IR, object code.

Local Improvements

- Example: changing

```c
if c > 7 goto L1
  goto L2
L1: ...
L2: ...
```

to

```c
if c <= 7 goto L2
  L1: ...
L2: ...
```
“Global” Improvements

- Example: changing

  ```
  for (i := 0; i < 1000; i++)
  a[i] := b*c + i;
  ```

  to

  ```
  t1 := b * c;
  for (i = 0; i < 1000; i++)
  a[i] = t1 + i;
  ```

Interprocedural improvements

- Example: Inlining a function

Most of a modern compiler is devoted to optimization.
INTERPRETATION

Simulate execution of program (source, AST, or other IR) on an abstract machine.

Implement abstract machine on a real machine.

Inputs to interpreter are

- Program to be interpreted
- Input to that program

Simpler than compiling and takes no time up front, but interpreted code runs (~10X) more slowly than compiled code.

Much more portable than real machine code (as for Java).

Helps with semantic definition.