Compilers Course
Lecture 12: Intermediate Representation (IR)

The purpose of the compiler's backend is to translate the AST to executable machine-specific code. However, there is a huge "semantic gap" between the source language (C, Java, etc) and the machine (MIPS, x86, etc).

An intermediate representation (IR) allows the compiler to perform the translation in smaller steps:

- First the AST is translated to the IR
- Then the IR is translated to machine-specific code

Depending on the complexity of the source-to-target translation, a series of successively simpler IRs may be used.

Using an IR can also help the compiler in other ways:

- It can support multiple target machines: each target will require a new IR-to-target translation, but the front-end of the compiler can be shared
- It can support multiple programming languages: each language requires a new parser, type checker, and AST-to-IR translator, but the back-end of the compiler can be shared

Example

```
int f(int x) {
     if ( x+g()<2 )
          return a();
     else
          return b();
}
```

RTL/quadruples

- RTL = register transfer language
- Quadruples = arithmetic operations with four parts: dst, src1, op, src2
- Sequence of simple machine-like instructions
- No nested expressions or statements
- Assumes unbounded number of temps (temporary variables == virtual registers), and that temps survive recursive calls

  + simple semantics
  + useful for global code optimization and register allocation later on
  + easy to translate to actual machine code
  - the many temps makes it not so easy to interpret by a virtual machine

```
f(x):
     t1 = x
     t2 = g()
     t3 = t1+t2
     t4 = 2
     if t3 >= t4 goto L
     t5 = a()
     return t5
L:   t6 = b()
     return t6
```

Using trees for expressions

- Sequence of assignment or control-flow statements
- Uses nested expressions for temporary values
- Must also have syntax for parameters, local variables, and global variables

+ simple AST-to-trees translation step
+ useful as initial step to eliminate high-level or ambiguous constructs
+ simple representation useful for high-level code optimization
- far from the machine, requires more translation steps
- inefficient representation for interpretation by a virtual machine

```
f(x):
     if (x + g()) >= 2 goto L
     return a()
L:   return b()
```

Using RPN

- RPN = reverse polish notation
- Sequence of instructions (push, +, if-jump, etc) for a stack machine
- Essentially just a different representation of trees

+ very easy to interpret by a virtual machine
- not useful for code optimization

```
f(x):              stack
     push x        x
     call g        x, g()
     +             x+g() [pop 2 values, compute +, push result]
     push 2         x+g(), 2
     if >= goto L   empty [pop 2 values, compare, maybe branch]
     call a         a()
     return         empty
L:   call b         b()
     return         empty
```

Summary

- Serious compilers generally use RTL as their main IR
- Trees may be used as an intermediate step between AST and RTL, especially when compiling high-level languages
- RPN is mainly used in abstract machine interpreters