Undergraduate Compilers Review cont...

**Announcements**
- Advice for the project writeups was posted on the mailing list
- SVN log will need more than one entry for a one day extension

**Today**
- Generating intermediate representations
  - AST
  - 3-address code
  - Trees
  - Assem
- Generating MIPS assembly

Structure of a Typical Compiler

- **Analysis**
  - character stream
  - lexical analysis
  - tokens “words”
  - syntactic analysis
  - AST “sentences”
  - semantic analysis
  - annotated AST
  - interpreter

- **Synthesis**
  - IR code generation
  - IR code generation
  - optimization
  - IR code generation
  - code generation
  - target language

Program Representations

- **AST**
  - usually language dependent

**Intermediate Representation (IR)**
- Usually a language independent and target independent representation
- Examples
  - 3-address code
  - RTL used in GCC (like 3-address code)
  - LLVM used in the LLVM compiler (like 3-address code but typed)
  - Tree data structure in the MiniJava Compiler (a little different)

**IR Code Generation**

- **Goal**
  - Transforms AST into low-level *intermediate representation* (IR)

- **Simplifies the IR**
  - Removes high-level control structures: `for`, `while`, `do`, `switch`
  - Removes high-level data structures: arrays, structs, unions, enums

- **Results in assembly-like code**
  - Semantic lowering
  - Control-flow expressed in terms of “gotos”
  - Each expression is very simple (three-address code)

  e.g.,
  
  ```
  x := a * b * c
  t := a * b
  x := t * c
  ```
A Low-Level IR

Register Transfer Language (RTL)
- Linear representation
- Typically language-independent
- Nearly corresponds to machine instructions

Example operations
- Assignment \( x := y \)
- Unary op \( x := \text{op } y \)
- Binary op \( x := y \text{ op } z \)
- Address of \( p := &y \)
- Load \( x := *(p+4) \)
- Store \( *(p+4) := y \)
- Call \( x := f() \)
- Branch \( \text{goto } L1 \)
- Cbranch \( \text{if (x==3) goto } L1 \)

Example

Source code
```plaintext```
for i = 1 to 10 do
  a[i] = x * 5;
```
High-level IR (AST)
```plaintext```

Low-level IR (RTL)
```plaintext```
i := 1
loop1:
t1 := x * 5
t2 := &a
t3 := sizeof(int)
t4 := t3 * i
t5 := t2 + t4
*t5 := t1
i := i + 1
if i <= 10 goto loop1
```

Compiling Control Flow

Switch statements
- Convert `switch` into low-level IR
  ```plaintext```
  ```plaintext```
  ```plaintext```
  ```plaintext```
- Optimizations (depending on size and density of cases)
  - Create a jump table (store branch targets in table)
  - Use binary search

Compiling Arrays

Array declaration
- Store name, size, and type in symbol table

Array allocation
- Call `malloc()` or create space on the runtime stack

Array referencing
- e.g., \( A[i] \)
  ```plaintext```
  ```plaintext```
  ```plaintext```
  ```plaintext```
  ```plaintext```
MiniJava Compiler Tree Language (Array Example)

```java
a2[0] = 3;
x = a2[0];
```

MiniJava Compiler Tree Language (IF Example)

```java
if (p<3) {
  System.out.println(p);
} else {
  System.out.println(3);
}
```

Structure of a Typical Compiler

- **Analysis**
  - Character stream
  - Lexical analysis
  - Tokens: “words”
  - Syntactic analysis
  - AST: “sentences”
  - Semantic analysis
  - Annotated AST
- **Synthesis**
  - IR code generation
  - IR
  - Optimization
  - Code generation
  - Target language
  - Interpreter

Compiling Procedures

**Properties of procedures**
- Procedures define scopes
- Procedure lifetimes are nested
- Can store information related to dynamic invocation of a procedure on a call stack (activation record or AR or stack frame):
  - Space for saving registers
  - Space for passing parameters and returning values
  - Space for local variables
  - Return address of calling instruction

**Stack management**
- Push an AR on procedure entry
- Pop an AR on procedure exit
- Why do we need a stack?
Compiling Procedures (cont)

Code generation for procedures

- Emit code to manage the stack
- Are we done?

Translate procedure body

- References to local variables must be translated to refer to the current activation record
- References to non-local variables must be translated to refer to the appropriate activation record or global data space

Code Generation

Conceptually easy

- Three address code is a generic machine language
- Instruction selection converts the low-level IR to real machine instructions

The source of heroic effort on modern architectures

- Alias analysis
- Instruction scheduling for ILP
- Register allocation
- More later...

MIPS code generation in MiniJava compiler

Assem data structure

- has string with source and destination spots to represent assembly instruction
- has list of uses, defs, and jump targets

```
add rd, rs, rt
  “add ’d0, ’s0, ’s1”
beq rs, rt, label
  “beq ’s0, ’s0, ’j0”
lw rt, address
  “lw ’d0, #’(s0)”
sw rt, address
  “sw ’s0, #’(s1)”
```

Example MIPs program

```
class MultipleParams {
  public static void main(String[] args){
    System.out.println(new Foo().testing());}
}

class Foo {
  public int testing() {
    int local1;
    int local2;
    int local3;
    local1 = 1;
    local2 = 2;
    local3 = 3;
    return this.foobar(local1, local2+local3,
      local3);
  }
  public int foobar(int param1, int param2, int param3){
    return param1 - param2 * param3 ;
  }
}
```
Concepts

Representations
- AST, low-level IR (RTL)

Code Generation
- 3-address code
- IR Trees in MiniJava Compiler
  - Assumes infinite temporaries
- Mips
  - Requires mapping of all temporaries to an actual register

Next Time

Reading
- Ch 10

Lecture
- Control Flow Graphs
- Liveness Analysis