**CS 453: Compiler Construction Review**

**Phases of the compiler**
- lexicographical analysis, or scanning (regular expressions)
- syntactic analysis, or parsing (context free grammars)
  - building the abstract syntax tree (syntax-directed translation)
  - building the symbol table (visitor design pattern)
- semantic analysis, or type checking (visitor design pattern)
- code generation (visitor design pattern)
  - 3-address code
  - Assem(MIPS)

**How would adding floats to the MiniJava compiler affect each phase?**

---

**Structure of the MiniJava Compiler**

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Synthesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>character stream</td>
<td>IR code generation</td>
</tr>
<tr>
<td>tokens</td>
<td>Assem (MIPS)</td>
</tr>
<tr>
<td>&quot;words&quot;</td>
<td>optimization</td>
</tr>
<tr>
<td>AST</td>
<td>Assem (MIPS)</td>
</tr>
<tr>
<td>&quot;sentences&quot;</td>
<td>code gen</td>
</tr>
</tbody>
</table>

**AST and symbol table**

---

**Specifying Tokens with JFlex**

JFlex example input file:
```java
package mjparser;
import java_cup.runtime.Symbol;

KK
Kline
Kchar
Kpublic
NonVal

LETTER=[A-Za-z] DIGIT=[0-9] UNDERSCORE="_"
LETT_DIG_UND=[LETTER] {DIGIT} {UNDERSCORE} ID=[LETTER] {LETT_DIG_UND}+

"&" { return new Symbol(sym.AND, new TokenValue(yytext(), yyline, yychar)); }
"boolean" { return new Symbol(sym.BOOLEAN,...
(ID) { return new Symbol(sym.ID, new ...
```

---

**Interaction Between Scanning and Parsing**

<table>
<thead>
<tr>
<th>character stream</th>
<th>Lexical analyzer</th>
<th>Parser</th>
</tr>
</thead>
<tbody>
<tr>
<td>lexer.next_token()</td>
<td>token</td>
<td>parse tree or AST</td>
</tr>
</tbody>
</table>
Specifying Grammar with JavaCUP

Start with program;

```
package mjparser;
import java_cup.runtime.*;
import ast.node.*;

terminal AND, ASSIGN, INT;
terminal mjparser.TokenValue NUMBER;
non terminal Program program;
non terminal List<IClassDecl> class_decl_list;
non terminal MainClass main_class;
```

```
program ::= main_class:m class_decl_list:l{: RESULT = new Program(m,l); :}
exp ::= \n\{ Token token = new Token(n.text, n.line, n.pos);
RESULT = new IntegerExp( token );
\}
```

Semantic Analysis

Determine whether source is meaningful

- Check for semantic errors
- Check for type errors
- Gather type information for subsequent stages
  - Relate variable uses to their declarations

Example errors (from C)

```
function1 = 3.14159;
x = 570 + "hello, world!"
scalar[i]
```

Compiler Data Structures

Symbol Tables

- Compile-time data structure
  - Holds names, type information, and scope information for variables

Scopes

- A name space
  - In Pascal, each procedure creates a new scope
  - In C, each set of curly braces defines a new scope
  - Can create a separate symbol table for each scope
  - What are the scopes in MiniJava?

Using Symbol Tables

- For each variable declaration:
  - Check for symbol table entry
  - Add new entry; add type info
- For each variable use:
  - Check symbol table entry
Example Symbol Table

class And {
    public static void main(String[] a){
        System.out.println(new Foo().testing(42));
    }
}
class Foo {
    public int testing(int p) {
        int x;
        if (p < 10 && 2 < p) {
            x = 7;
        } else {
            x = 22;
        }
        return x;
    }
}

Properties of procedures

- Procedures/methods/functions 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 (caller or callee)
- Pop an AR on procedure exit (caller or callee)
- Why do we need a stack?

Assem intermediate representation

Assem.Instr

- "assembly language instruction without register assignments"

OPER(String assem, List<Temp> dst, List<Temp> src, List<Label> jumps)

- contains a string with holes for registers indicated by ‘d’ and ‘s’ and holes for labels indicated by ‘j’
- dst and src are lists of Temps whose register assignment should fill holes
- first entry in src is associated with ‘d0’, second with ‘s1’, etc.
- first entry in dst is associated with ‘d0’, etc.
- jumps is a list of labels for filling in label holes
Assem intermediate representation cont ...

LABEL(String assem, Label label)
- a label statement in the target code

MOVE(String assem, Temp dst, Temp src)
- similar to OPER in that assem string contains holes, but ..
  - no jumps
  - only one src and dst Temp

CJUMP(String s, Temp Temp src1, RELOP op, Temp.Temp src2, Temp.Label t, Temp.Label f)
- similar to OPER in that assem string contains holes, but ..
  - only jumps to true and false target
  - only two source Temps for comparison
  - explicit conditional operation, which enables later changes in code layout

---

x86-64 example

```plaintext
.globl foo
.type foo, @function
foo:
pushq %rbp  # %rsp = %esp-8; M[%esp ] = %rbp
movq %rsp, %rbp  # %rbp = %rsp
movl %esi, -24(%rbp)  # storing parameters to stack
movl %edi, -24(%rbp)  # accessing x
addl -28(%rbp), %eax  # adding y to x and storing in %eax
movl %eax, -4(%rbp)
leave
ret
.size foo, -.foo
```

---

Instruction selection for x86-64

Registers
- 16 64-bit registers
- RSP, the stack pointer register
- RBP, the frame pointer register
- Only jumps to true and false target
- Only two source Temps for comparison
- Explicit conditional operation, which enables later changes in code layout

Some Instructions
- Constants prefixed with ‘$’, for example $3, $4, $-5, etc
- Registers prefixed with ‘%’, for example %esp, %rbp, etc.

Example continued...

```plaintext
.globl main
.type main, @function
main:
pushq %rbp  # %rsp = %esp-8; M[%esp ] = %rbp;
pushes onto stack
movq %rsp, %rbp  # %rbp = %rsp
subq $16, %esp  # %esp = %esp - 16
movl $5, %esi  # %esi = 5
movl $4, %edi  # %edi = 4
call foo
movl %eax, -4(%rbp)  # M[%rbp-4] = %eax
movl -4(%rbp), %eax  # %eax = M[%rbp-4]
leave
ret
```

---

Example continued...