Some Thoughts on Grad School

Goals
- learn how to learn a subject in depth
- learn how to organize a project, execute it, and write about it

Iterate through the following:
- read the background material
- try some examples
- ask lots of questions
- repeat

You will have too much to do!
- learn to prioritize
- it is not possible to read ALL of the background material
- spend 2+ hours of dedicated time EACH day on each class/project
- have fun and learn a ton!

Undergraduate Compilers Review

Announcements
- Send me email if you have set up a group so we can set up a unix group.
  Include account names for group.
- Each project is 10% of grade
- No curve

Today
- Overall structure of a compiler
- Lexical analysis (scanning)
- Syntactic analysis (parsing)
- Generating an AST
- Semantic analysis (type checking)
- Code generation
Structure of a Typical Interpreter

Analysis

- character stream
  - lexical analysis
  - tokens ("words")
  - syntactic analysis
  - AST ("sentences")
    - semantic analysis
    - annotated AST
      - interpreter

Synthesis

- IR code generation
- IR
  - optimization
  - code generation
  - target language

Structure of the MiniJava Compiler

Analysis

- character stream
  - lexical analysis
  - tokens ("words")
  - syntactic analysis
  - AST ("sentences")
    - semantic analysis
    - AST and symbol table

Synthesis

- code gen
  - MIPS
  - PA2: MiniJava and MIPS warmup
  - PA3: constant expression compiler
  - PA4: add assignments and println
  - PA5: add control flow
  - PA6: add functions
  - PA7: add arrays and classes
Lexical Analysis (Scanning)

Break character stream into tokens (“words”)
– Tokens, lexemes, and patterns
– Lexical analyzers are usually automatically generated from patterns (regular expressions) (e.g., lex)

Examples

<table>
<thead>
<tr>
<th>token</th>
<th>lexeme(s)</th>
<th>pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>const</td>
<td>const</td>
<td>const</td>
</tr>
<tr>
<td>if</td>
<td>if</td>
<td>if</td>
</tr>
<tr>
<td>relation</td>
<td>&lt;,&lt;=,=,!=,...</td>
<td>&lt;</td>
</tr>
<tr>
<td>identifier</td>
<td>foo,index</td>
<td>[a-zA-Z-]+[a-zA-Z0-9-]*</td>
</tr>
<tr>
<td>number</td>
<td>3.14159,570</td>
<td>[0-9]+</td>
</tr>
<tr>
<td>string</td>
<td>“hi’, “mom’</td>
<td>“.*”</td>
</tr>
</tbody>
</table>

const pi := 3.14159 ⇒ const, identifier(pi), assign,number(3.14159)

Specifying Tokens with JLex

JLex example input file:

```
package mjparser;
import java_cup.runtime.Symbol;

%%
%line
%char
%cup
%public

%eofval{
  return new Symbol(sym.EOF, new TokenValue(yytext(), yyline, yychar));
}

LETTER=[A-Za-z]
DIGIT=[0-9]
UNDERSCORE="_"
LETT_DIG Und={LETTER}{DIGIT}{UNDERSCORE}
ID={LETTER}{{LETT_DIGUnd}}*

"&&" { 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

Recognizing Tokens with DFAs

`if`

Ambiguity due to matching substrings
- Longest match
- Rule priority
Syntactic Analysis (Parsing)

Impose structure on token stream
- Limited to syntactic structure (⇒ high-level)
- Parser often generates abstract syntax tree (AST)
- Parsers are usually automatically generated from context-free grammars
  (e.g., yacc, bison, cup, javacc, sablecc)

Example

```plaintext
for i = 1 to 10 do
  a[i] = x * 5;
```

Parser diagram:
```
for
 i 10 asg
 arr tms
 a i x 5
```

`for id(i) equal number(1) to number(10) do
id(a) lbracket id(i) rbracket equal id(x) times number(5) semi`

Bottom-Up Parsing: Shift-Reduce

<table>
<thead>
<tr>
<th>Grammer</th>
<th>a + b + c</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) S → E</td>
<td>S → E</td>
</tr>
<tr>
<td>(2) E → E + T</td>
<td>→ E + T</td>
</tr>
<tr>
<td>(3) E → T</td>
<td>→ E + id</td>
</tr>
<tr>
<td>(4) T → id</td>
<td>→ E + T + id</td>
</tr>
<tr>
<td></td>
<td>→ E + id + id</td>
</tr>
<tr>
<td></td>
<td>→ id + id + id</td>
</tr>
</tbody>
</table>

Rightmost derivation: expand rightmost non-terminals first
SableCC, yacc, and bison generate shift-reduce parsers:
- LALR(1): look-ahead, left-to-right, rightmost derivation in reverse, 1 symbol lookahead
- LALR is a parsing table construction method, smaller tables than canonical LR

Reference: Barbara Ryder’s 198:515 lecture notes
LR Parse Table

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>+</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>s3</td>
<td>accept</td>
</tr>
<tr>
<td>2</td>
<td>r(3)</td>
<td>r(3)</td>
</tr>
<tr>
<td>3</td>
<td>s4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>r(4)</td>
<td>r(4)</td>
</tr>
<tr>
<td>5</td>
<td>r(2)</td>
<td>r(2)</td>
</tr>
</tbody>
</table>

Look at current state and input symbol to get action
- **shift(n):** advance input, push n on stack
- **reduce(k):** pop rhs of grammar rule k, k = (lhs -> rhs)
  - look up state on top of stack and lhs for goto n
  - push n
- **accept:** stop and success
- **error:** stop and fail

Shift-Reduce Parsing Example

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ 0</td>
<td>a + b + c</td>
<td>shift 4</td>
</tr>
<tr>
<td>$ 0 a 4</td>
<td>+ b + c</td>
<td>reduce (4)</td>
</tr>
<tr>
<td>$ 0 T 2</td>
<td>+ b + c</td>
<td>reduce (3)</td>
</tr>
<tr>
<td>$ 0 E 1</td>
<td>+ b + c</td>
<td>shift</td>
</tr>
<tr>
<td>$ 0 E 1 + 3</td>
<td>b + c</td>
<td>shift</td>
</tr>
<tr>
<td>$ 0 E 1 + 3 b 4</td>
<td>+ c</td>
<td>reduce (4)</td>
</tr>
<tr>
<td>$ 0 E 1 + 3 T 5</td>
<td>+ c</td>
<td>reduce (2)</td>
</tr>
<tr>
<td>$ 0 E 1</td>
<td>+ c</td>
<td>shift</td>
</tr>
<tr>
<td>$ 0 E 1 + 3</td>
<td>c</td>
<td>shift</td>
</tr>
<tr>
<td>$ 0 E 1 + 3 c 4</td>
<td>reduce (4)</td>
<td></td>
</tr>
<tr>
<td>$ 0 E 1 + 3 T 5</td>
<td>reduce (2)</td>
<td></td>
</tr>
<tr>
<td>$ 0 E 1</td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>

**Reference:** Barbara Ryder’s 198:515 lecture notes
Subset of MiniJava Expression Grammar

Expression ::=  
Expression ( "&&" | "<" | "=" | "*" | "+" | "-" | "*" ) Expression  
| <INTEGER_LITERAL>  | "(" Expression ")"  

Full Expression Grammar

Expression ::=  
Expression ( "&&" | "<" | "+" | "-" | "*" ) Expression  
| Expression "[" Expression "]"  
| Expression "." "length"  
| Expression "." Identifier "(" ( Expression "," Expression )* ")? "")"  
| <INTEGER_LITERAL>  | "true"  | "false"  | Identifier  
| "this"  | "new" "int" "[" Expression "]"  | "new" Identifier ")"  
| "!" Expression  | "(" Expression ")"  

Expression Grammar and AST Node Hierarchy

// JavaCUP specification of part of expression grammar  
Expression ::=  
Expression ( "+" | "-" | "*" ) Expression  
| <INTEGER_LITERAL>  | "(" Expression ")"  

Node

IExp

Token

MinusExp  PlusExp  MulExp  IntegerExp
Syntax-directed Translation: AST Construction example

Grammar with production rules

\[
\begin{align*}
S & : E \quad \{ \$$ = \$1; \}; \\
E & : E \ ' + ' \ T \quad \{ \$$ = \text{new node}("+", \$1, \$3); \} \\
& \mid T \quad \{ \$$ = \$1; \} \\
; & \\
T & : \ T\_ID \quad \{ \$$ = \text{new leaf}("id", \$1); \};
\end{align*}
\]

Implicit parse tree for a+b+c

AST for a+b+c

Reference: Barbara Ryder’s 198:515 lecture notes

Using JavaCUP to specify grammar and generate AST

Show src/mjparser/mj_ast.cup
Visitor Design Pattern

Situation
– Want to perform some processing on all items in a data structure
– Will be adding many different ways to process items, different features
– Will not be changing the classes of the data structure itself much

Possibilities
– For each functionality add a method to all of the classes
  – Each new functionality is spread over multiple files
  – Sometimes can’t add methods to existing class hierarchy
– Use a large if-then-else statement in visit method
  – pro: keeps all the code for the feature in one place
  – con: can be costly and involve lots of casting
– Visitor design pattern

Borrowed SableCC Visitor Design Pattern

```java
ast_root.apply(new EvalVisitor(ast_root));
...
// in class MulExp
public void apply(Switch sw) { ((Analysis) sw).caseMulExp(this); }
...
// in class DepthFirstAdapter
public void inMulExp(MulExp node) { defaultIn(node); }
public void outMulExp(MulExp node) { defaultOut(node); }
public void caseMulExp(MulExp node) {
  inMulExp(node);
  if(node.getLExp() != null) { node.getLExp().apply(this); }
  if(node.getRExp() != null) { node.getRExp().apply(this); }
  outMulExp(node);
}
...
// in class EvalVisitor
public void outMulExp(MulExp node){
  stack.push(stack.pop()*stack.pop());
  if(node==root){ System.out.println(stack.pop()); }
}
```
SymTable, Scope, and STE classes (used in BuildSymTab)

Type implementation in the MiniJava compiler

```java
public class Type {
    public static final Type ARRAY = new Type();
    public static final Type BOOL = new Type();
    public static final Type INT = new Type();

    // class type map (key: class name, value: type)
    private static final HashMap<String, Type> classTypes = new HashMap<String, Type>();
}
```

Only one instance of the type object per atomic type and class type
- to determine if types are equal just compare references
- Type class does not know about inheritance
MiniJava Types for Example

Code Generation

Conceptually easy
- The visitor that builds the symbol table (src/ast_visitors/BuildSymTable) allocates space for variables on stack or in heap-allocated object
- Visitor over AST will generate MIPS code

The source of heroic effort on modern architectures
- Instruction scheduling for ILP
- Register allocation
- Alias analysis
- More later...
### Patt & Patel Book Stackframe for MIPS (example)

```c
int foo(int x, int y, int *z) {
    int a;
    a = x * y - *z;
    return a;
}

void main() {
    int x;
    x = 2;
    printf("%d\n", foo(4, 5, &x));
}
```

```asm
.text
.main:
    addi $fp, $sp, -4
    addi $sp, $sp, -4
    # x = 2
    li $t0, 2
    sw $t0, 0($fp)
    # push $x
    addi $sp, $sp, -4
    sw $fp, 0($sp)
    # push 5
    li $t0, 5
    addi $sp, $sp, -4
    sw $t0, 0($sp)
    jal _foo
    # grab retval
    lw $t0, 0($sp)
    # pop retval & params
    addi $sp, $sp, 16
    # print $t0
    ...
    # HALT MARS
    li $v0, 10
    syscall
```

### Patt and Patel book calling convention (for MIPS)

**Calling convention (contract between caller and callee)**
- caller should push parameters right to left onto the stack
- upon callee entry, the stack pointer $sp should be pointing at the first parameter
- upon callee exit, the stack pointer $sp should be pointing at the return value, which should be followed by the first parameter
- $sp must be divisible by 4 (for MIPS)
- $sp should always be pointing at the top entry on the stack

**Standardizing the stack frame implementation for this course**
- $ra and $fp should be stored on top of the return value slot
- locals should be stored on top of $ra and $fp
- $fp should be made to point at the first local variable, so that the address for the first local is $fp+0, the address for the second local is $fp+4, ...
- The offsets for the incoming parameters will differ based on whether there is a return value. If there is a return value, then the first parameter will be at $fp-16, the second at $fp-20, etc. If there is no return value, then the first parameter will be at $fp-12, the second at $fp-16, etc.
**Structure of the MiniJava Compiler**

![Diagram of compiler structure]

**Analysis**
- character stream
- lexical analysis
- tokens: "words"
- syntactic analysis
- AST: "sentences"
- semantic analysis

**Synthesis**
- code gen
- MIPS

- PA2: MiniJava and MIPS warmup
- PA3: constant expression compiler
- PA4: add assignments and println
- PA5: add control flow
- PA6: add functions
- PA7: add arrays and classes

**Concepts**

**Compilation stages in a compiler**
- Scanning, parsing, semantic analysis, intermediate code generation, optimization, code generation

**Lexical analysis or scanning**
- Tools: SableCC, lex, flex, etc.

**Syntactic analysis or parsing**
- Tools: SableCC, yacc, bison, etc.
- Generation of AST

**Semantic Analysis**
- Symbol tables
- Using visitors over the AST

**Code Generation to MIPS**

**Parsing Terms**
- See attached slides, be familiar with these terms
Next Time

Suggested Exercises for concepts covered this time
- from book: 2.2.1, 2.2.2, 2.3.1
- follow a while loop in MiniJava through to code gen
  - what does AST look like?
  - what does IRT Tree look like?
  - what is the MIPSnoREG code?
  - how would we implement a do while loop?

Lecture
- Compiling OOP

Parsing Terms

Top-down parsing
- LL(1): left-to-right reading of tokens, leftmost derivation, 1 symbol look-ahead
- Predictive parser: an efficient non-backtracking top-down parser that can handle LL(1)
  - More generally recursive descent parsing may involve backtracking

Bottom-up Parsing
- LR(1): left-to-right reading of tokens, rightmost derivation in reverse, 1 symbol lookahead
- Shift-reduce parsers: for example, bison, yace, and SableCC generated parsers
  - Methods for producing an LR parsing table
    - SLR, simple LR
    - Canonical LR, most powerful
    - LALR(1)
 Parsing Terms (Definitely know these terms)

  **Lexical Analysis**
  – longest match and rule priority
  – regular expressions
  – tokens

  **CFG (Context-free Grammar)**
  – production rule
  – terminal
  – non-terminal

  **Syntax-directed translation**
  – inherited attributes
  – synthesized attributes