Plan for Today

Logistics
- Recitation on Friday will be a review for Final Exam.
  - Review notes will be posted Wednesday night.
  - People can ask questions during recitation and/or on Piazza. Example questions with answers will receive extra credit as per syllabus.
- PA5 is due Monday. Any questions?
- PA4 Peer Review summary

Reviewing compilation covered by other professors
- (See slides on schedule or notes posted in piazza)
- LR parsing: performs a right-most derivation in reverse
- Symbol tables and scope
- Register allocation for expression trees
PA4 Peer Reviews

Specific Comments that are Useful

– "This is the cleanest compiler assignment my eyes have ever seen” …"I especially appreciated the symbol table pretty print functions, the massive amount of Java keywords that can be lexed, and the optional arguments in main for testing purposes.” Coding structure to emulate
– "Reformatting the token row/col information to be stored in a token wrapper rather than the token itself may have made the parser much easier to understand and write. This means we wouldn't have had to pass the new int arguments around and handle them in multiple cases.”

Fun jokes in README

– “I like my steak how I like my transformers. Optimus prime.”
– “A manager, a mechanical engineer, and software analyst are driving back from convention through the mountains. Suddenly, as they crest a hill, the brakes on the car go out and they fly careening down the mountain. After scraping against numerous guardrails, they come to a stop in the ditch. Everyone gets out of the car to assess the damage. The manager says, "Let's form a group to collaborate ideas on how we can solve this issue." The mechanical engineer suggests, "We should disassemble the car and analyze each part for failure.” The software analyst says, "Let's push it back up the hill and see if it does it again.”
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 → optimization
  - IR → code generation
  - target language

- IR

- Optimization
LL vs LR Parsing

LL(k) must predict which production looking ahead k:

\[ S \rightarrow SS \mid (S) \mid \varepsilon \]

is it \( S \rightarrow SS \) or \( (S) \) when I see \(((...... ?

produces the parse tree TOP DOWN, with a left to right derivation

LR(k) postpones the decision until all tokens of the rhs of a grammar production plus k more tokens have been seen. It therefore is more powerful.

It does this by parsing BOTTOM UP
Example LR parse

\[ \begin{align*}
S & \rightarrow AB \\
A & \rightarrow Aa \mid a \\
B & \rightarrow Bb \mid b \\
S' & \rightarrow S$
\end{align*} \]

\[ \text{aaabb$\leftarrow\text{Aaabb$\leftarrow\text{Aabb$\leftarrow\text{Abb$\leftarrow ABb$ \leftarrow AB$\leftarrow SS\leftarrow S'} \]

Notice that this is the rightmost derivation

\[ \begin{align*}
S' & \rightarrow S$ \rightarrow AB$ \rightarrow ABb$ \rightarrow Abb$ \rightarrow Aabb$ \rightarrow Aaabb$ \rightarrow \text{aaabb$}
\end{align*} \]

in reverse!

It does not start with the start symbol, it ends with it

LR(k) parsing scans the input Left to right and produces the Rightmost derivation (looking k tokens ahead) in reverse.
## Simplified example LR parsing engine actions

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>aaabb$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>aabb$</td>
<td>reduce: A→a</td>
</tr>
<tr>
<td>A</td>
<td>aabb$</td>
<td>shift</td>
</tr>
<tr>
<td>Aa</td>
<td>abb$</td>
<td>reduce: A→Aa</td>
</tr>
<tr>
<td>A</td>
<td>abb$</td>
<td>shift</td>
</tr>
<tr>
<td>Aa</td>
<td>bb$</td>
<td>reduce: A→Aa</td>
</tr>
<tr>
<td>A</td>
<td>bb$</td>
<td>shift</td>
</tr>
<tr>
<td>Ab</td>
<td>b$</td>
<td>reduce: B→b</td>
</tr>
<tr>
<td>AB</td>
<td>b$</td>
<td>shift</td>
</tr>
<tr>
<td>ABb</td>
<td>$</td>
<td>reduce: B→Bb</td>
</tr>
<tr>
<td>AB</td>
<td>$</td>
<td>reduce: S→AB</td>
</tr>
<tr>
<td>S</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

### Grammar Rules
- \( S \rightarrow AB \)
- \( A \rightarrow Aa | a \)
- \( B \rightarrow Bb | b \)
- \( S' \rightarrow S \$ \)
Shift reduce parsing applied to unambiguous grammars

Stack        input         action

((ID))$       shift

((ID))$       shift

((ID))$       shift

((ID ))$       reduce: S→Id

((S ))$       shift

((S ))$       reduce: S→(S)

(S )$         shift

(S )$         reduce: S→(S)

S $           accept
Static Scope Rules

Most languages have static scope rules

• **Static scope rules are based on the program text**
  – The scope of a declaration can be determined at compile time
  – Otherwise, the language is said to have dynamic scope rules
  – Macro-expansion results in dynamic scope

• **A block consists of declarations and statements**
  – Blocks are delimited by braces, `{ }`, in C, Java, ...
  – Blocks can be nested
  – Does MeggyJava have blocks?
Scope of a Declaration

How many declarations of x?

class C {
    
    int x;
    public int f(int x) {
        return x;
    }
    
    public int g(int y) {
        return x;
    }
}
Scope of a Declaration

Subscripts distinguish between roles of \( x \)

class C
{
    int \( x_1 \);
    public int f(int \( x_2 \))
    {
        return \( x_2 \);
    }
    public int g(int y)
    {
        return \( x_1 \);
    }
}
**Hole in the Scope of a Declaration**

Block $B_2$ is a hole in the scope of the declaration of $x_1$. 

```java
class C {
  int $x_1$;
  public int f(int $x_2$) {
    return $x_2$;
  }
  public int g(int $y$) {
    return $x_1$;
  }
}
```
Most Closely Nested Rule

Find the declaration of $x$ by examining blocks inside out

class C
{
    int $x_1$;
    public int $f$ (int $x_2$)
    {
        return $x_2$;
    }
    public int $g$ (int $y$)
    {
        return $x_1$;
    }
}

Block $B_1$

Block $B_2$

Block $B_3$
Symbols

What do the occurrences of \texttt{x} denote?

class Foo {
    public static void main(String[] s) {
        new E().f(3); }

class E {
    C x;
    public int f(int y) {
        C x; x = new C(); x.x = 7;
        this.initE();
        return x.x;
    }
    public C initE() { x=new C(); x.x = 9; return x; }
}
Expression Evaluation

Sethi-Ullman Register allocation

label(node)

  if node is leaf then node.label = 1
  else if node is binary
      if node.left.label == node.right.label
          then node.label = node.left.label + 1
      else
          node.label = max( node.left.label, node.right.label )
  else if node is unary
      node.label = node.child.label

Questions to understand how to answer

- How many registers are needed if not using memory (aka push/pop)?
- Order of evaluation to make this register count work?
- What if an operator has more than 2 operands?