CSE443
Compilers
Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall
Final Exam

5/16/2022, Monday
Start @ 8:00 AM
End @ 11:00 AM

Arrive by 7:50 AM
Entry not guaranteed after 8:30 AM
Exam format

- Expect 4 short essay questions (choose from ~6). We will use BlueBooks.

- I expect you to take about 30 minutes per question (about 2 hours total).

- This leaves you with about 1 hour to proofread/edit your responses.
Possible Exam Questions

- Anything from homeworks,

...and...
Type checking
(Semantic processing)

Explain how type errors are detected. Discuss how type information is gathered, stored and checked. Pick a concrete syntactic construct that can contain a type error, and explain how type checking detects the error.
Intermediate Code Generation

- Explain how short-circuit Boolean expressions are translated into intermediate code. Discuss how jump targets can be determined during backpatching. Illustrate by showing how a concrete Boolean expression involving at least two Boolean operators is translated into intermediate code.
Describe the getReg(I) algorithm, answering the questions of what data structures it uses, when and how these structures are updated. What is meant by "spill", when does it occur, and why is it needed? Demonstrate with a concrete example.
Symbol Table Usage

Describe the structure and use of a symbol table. Explain which phases of the compiler use the table, including what data is written to or read from the table during each phase. Give a concrete code example and the corresponding ST.
Invocation Records

1. Describe a typical layout for an invocation record, detailing what information is stored in the record. Explain how variable length parameters and variable length local data can be accommodated. Discuss the location and use of the stack and top pointers. Give concrete example.
Function Calls

- Explain how a function call takes place. Include in your discussion mention of the roles of the caller and callee in setting up the invocation record: discuss both calling and return sequences, and the division of labor between caller and callee. Explain how machine state is remembered at the call and restored at return. Cover how recursive calls are handled (do NOT discuss tail-call optimization). Give concrete example.
Optimizations

- Pick an optimization and explain the benefit(s) of having the compiler apply it to code, and sketch how it works for a concrete example.

  Ex:
  - tail-call optimization
  - code motion
  - dead code elimination
LEXICAL ANALYSIS → Token stream
LEXICAL structure described by a regular grammar
FLEX
LEXICAL analyzer (C program)
NFA → DFA → min DFA
Finite control

SYNTACTIC ANALYSIS → parse tree
Context-free grammar
BISON
Lexical structure (keywords, identifiers, etc)

OUTPUT

SYMBOL TABLE
Contains all names/symbols
Stores info about those symbols:
- Kind of value it represents (e.g., for a variable, its type)
- Its scope
- Etc.

SEMICANALYSIS
Attribute grammar

ACTION GOTO
Parser
PARSER
Stack
S: $
SEMANTIC ANALYSIS

parse tree -> IR code generation

attribute grammar

Semantic rules
Semantic predicates

ACTION GOTO

FINITE CONTROL

SYMBOL TABLE
contains all names/symbols used in program info about those symbols, such as kind of value it represents: type, variable, ...
for a variable: its type
its scope ... how is this done?
"one table per scope"

assign

var := expr

expr

unary:

expr op expr

keep track of types

Global scope

To look up a name, start here and search through parent scopes to global scope.
Character stream → LEXER → token stream → PARSER → syntax tree

---

**HLL**

- `+` = `- - i`/`i / i % i`

- ` enum 1R-op`  
- ` iffalse x GOTO L2` if `x GOTO L1`
  - `GOTO L1`  
  - `GOTO L2`

- `int foo (int x, real y) {} ... 3`

---

**3-address code instructions**

1. `X = Y op Z`
2. `X = op y`
3. `X = y`
4. `GOTO []`
5. `if [x GOTO []] / iffalse [x GOTO []]`
6. `if [x rel op y] GOTO []`
7. `param x / call f, x`
8. `x = a[i]`/`a[i] = y`
9. `x = & y`  
  - `x = *y`  
  - `*x = y`

---

**IR**

- `1R`

---

**RO**

- `R0 param X`
- `R1 param Y`
- `call f, 2`
3-address code instructions

IR instructions

1. \( X = Y \) op 2
2. \( X = \text{op} \ Y \)
3. \( X = Y \)
4. \( \text{GOTO} \)
5. if \( x \) GOTO \( \text{ifFalse} \ x \) GOTO \( \)
6. if \( x \) rel op \( y \) GOTO \( \)
7. param \( x \) \( \text{call f, n} \)

invocation record

8. \( X = a[i] \) \( a[i] = y \)
9. \( x = \& y \) \( x = *y \) \( **x = y \)

Syntax tree with annotations

- \( \text{makeList}(3) \rightarrow [3] \)
- \( \text{mergeLists}([0,5], [3]) \rightarrow [0,3,5] \)
- \( \text{back patch}([0,3,5], [17]) \)
- \( \text{gen creates an IR instruction} \)

IR code arr

- \( \text{if x goto 17} \)
- \( \check \)
- \( \check \)
- \( [3] \)
- \( 4 \)
- \( 5 \)
- \( \text{goto 17} \)
- \( 6 \)

nextInstr : 18

...
```c
struct IR * instruction = ...;

A[\text{nextInstr}++] = instruction;

struct IR {
    struct SIE * dst, src1, src2;
    enum IR_op op;
    int jump-target;
    int instruction-type;
};
```
9.2.5 Live variable analysis

Useful for effective register management.

"After a value is computed in a register, and presumably used within a block, it is not necessary to store that value if it is dead at the end of the block. Also, if all registers are full and we need another register, we should favor using a register with a dead value, since that value does not have to be stored." [p. 608]
9.2.5 Live variable analysis

"In live variable analysis we wish to know for variable x and point p whether the value of x at p could be used along some path in the flow graph starting at p. If so, we say x is live at p; otherwise, x is dead at p." [p. 608]

In contrast to reaching analysis, which used a forward transfer function, live variable analysis uses a backward transfer function.
9.2.5 Live variable analysis
definitions, page 609

$\text{def}_B$ is "the set of variables defined in $B$ prior to any use of that variable in $B"$

$\text{use}_B$ is "the set of variables whose values may be used in $B$ prior to any definition of the variable"
9.2.5 Live variable analysis
definitions, page 609

\[\begin{align*}
\text{IN[EXIT]} &= \emptyset \\
\text{IN[B]} &= \text{use}_B \cup (\text{OUT[B]} - \text{def}_B) \\
\text{OUT[B]} &= \bigcup_{S \text{ a successor of } B} \text{IN[S]}
\end{align*}\]
9.2.5 Live variable analysis

Algorithm [p. 610]

INPUT: A flow graph with def and use computed for each block.

OUTPUT: IN[B] and OUT[B], the set of variables live on entry and exit of each block of the flow graph.

METHOD:

\[
\text{IN[EXIT]} = \emptyset \\
\text{for (each basic block B other than EXIT)} \{ \text{IN[B]} = \emptyset \} \\
\text{while (changes to any IN occur)} \{ \\
\text{for (each basic block B other than EXIT)} \{ \\
\text{OUT[B]} = \bigcup_{S \text{ a successor of } B} \text{IN[S]} \\
\text{IN[B]} = \text{use}_B \cup (\text{OUT[B]} - \text{def}_B) \\
\} \\
\}\]
9.2.6 Available expressions

"An expression \( x+y \) is available at a point \( p \) if every path from the entry node to \( p \) evaluates to \( x+y \), and after the last such evaluation prior to reaching \( p \), there are no subsequent assignments to \( x \) or \( y \)." [p. 610]
9.2.6 Available expressions

"...a block kills expression x+y if it assigns (or may assign) x or y and does not subsequently recompute x+y." [p. 610]

"A block generates expression x+y if it definitely evaluates x+y and does not subsequently define x or y." [p. 611]
...the expression 4 * i in block B3 will be a common subexpression if 4 * i is available at the entry point of block B3.

[p 611]
"It will be available if i is not assigned a new value in block B2, …" [p 611]

Here 4 * i in B3 can be replaced by value of t1, regardless of which branch is taken.
"... or if ... 4 * i is recomputed after i is assigned in B2." [p 611]

Again, 4 * i in B3 can be replaced by value of t1, regardless of which branch is taken (since t1 contains the correct value of 4 * i in both cases)
9.2.6 Available expressions

Informally:

"If at point p set S of expressions is available, and q is the point after p, with statement \( x = y + z \) between them, then we form the set of expressions available at q by the following steps:

1. Add to S the expression \( y + z \).
2. Delete from S any expression involving variable \( x \)."

[p. 611]
**Example 9.15**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Available expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = b + c$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$b = a - d$</td>
<td>${ b + c }$</td>
</tr>
<tr>
<td>$c = b + c$</td>
<td>${ a - d }$</td>
</tr>
<tr>
<td>$d = a - d$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
9.2.6 Available expressions

"We can find available expressions in a manner reminiscent of the way reaching definitions are computed. Suppose \( U \) is the 'universal' set of all expressions appearing on the right of one or more statement of the program. For each block \( B \), let \( \text{IN}[B] \) be the set of expressions in \( U \) that are available at the point just before the beginning of \( B \). Let \( \text{OUT}[B] \) be the same for the point following the end of \( B \). Define \( \text{e}_{\text{gen}}[B] \) to be the expressions generated by \( B \) and \( \text{e}_{\text{kill}}[B] \) to be the set of expressions in \( U \) killed in \( B \). Note that \( \text{IN}, \text{OUT}, \text{e}_{\text{gen}}, \) and \( \text{e}_{\text{kill}} \) can all be represented by bit vectors." [p. 612]
9.2.6 Available expressions definitions, page 612

\[
\begin{align*}
\text{OUT[ENTRY]} &= \emptyset \\
\text{OUT[B]} &= e_{\text{genB}} \cap (\text{IN[B]} - e_{\text{killB}}) \\
\text{IN[B]} &= \bigcap_{P \text{ a predecessor of } B} \text{OUT[P]}
\end{align*}
\]
9.2.6 Available expressions
definitions, page 612

\[ \text{OUT[ENTRY]} = \emptyset \]

\[ \text{OUT[B]} = e_{\text{gen}_B} \cap (\text{IN[B]} - e_{\text{kill}_B}) \]

\[ \text{IN[B]} = \bigcap_{P \text{ a predecessor of } B} \text{OUT[P]} \]

Note use of \( \cap \) rather than \( \cup \).

"...an expression is available at the beginning of a block only if it is available at the end of ALL its predecessors." [p. 612]
9.2.6 Available expressions

Algorithm [p. 614]

INPUT: A flow graph with $e_{\text{kill}}_B$ and $e_{\text{gen}}_B$ computed for each block $B$. The initial block is $B_1$.

OUTPUT: $\text{IN}[B]$ and $\text{OUT}[B]$, the set of expressions available at the entry and exit of each block of the flow graph.

METHOD:

\[
\text{OUT}[\text{ENTRY}] = \emptyset
\]

for (each basic block $B$ other than $\text{ENTRY}$) {
  \[ \text{OUT}[B] = \emptyset \]
}

while (changes to any $\text{OUT}$ occur) {
  for (each basic block $B$ other than $\text{EXIT}$) {
    \[ \text{IN}[B] = \bigcap \{ \text{OUT}[P] \mid \text{P a predecessor of } B \} \]
    \[ \text{OUT}[B] = e_{\text{gen}}_B \cap (\text{IN}[B] - e_{\text{kill}}_B) \]
  }
}
9.2.6 Available expressions

Algorithm [p. 614]

INPUT: A flow graph with $e_{\text{kill}}_B$ and $e_{\text{gen}}_B$ computed for each block $B$. The initial block is $B_1$.

OUTPUT: $\text{IN}[B]$ and $\text{OUT}[B]$, the set of expressions available at the entry and exit of each block of the flow graph.

METHOD:

\[
\text{OUT}[\text{ENTRY}] = \emptyset \\
\text{for } (\text{each basic block } B \text{ other than } \text{ENTRY}) \{ \text{OUT}[B] = U \} \\
\text{while (changes to any OUT occur)} \{ \\
\text{for } (\text{each basic block } B \text{ other than } \text{EXIT}) \{ \\
\text{IN}[B] = \bigcap \text{a predecessor of } B \text{ OUT}[?] \\
\text{OUT}[B] = e_{\text{gen}}_B \cap (\text{IN}[B] - e_{\text{kill}}_B) \\
\}\}
\]

Recall: $U$ is set of all expressions.
### 9.2 Summary

<table>
<thead>
<tr>
<th></th>
<th>Reaching definitions</th>
<th>Live variables</th>
<th>Available expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain</strong></td>
<td>sets of definitions</td>
<td>sets of variables</td>
<td>sets of expressions</td>
</tr>
<tr>
<td><strong>Direction</strong></td>
<td>forward</td>
<td>backward</td>
<td>forward</td>
</tr>
<tr>
<td><strong>Transfer</strong></td>
<td>( \text{gen}_B \cup (x - \text{kill}_B) )</td>
<td>( \text{use}_B \cup (x - \text{def}_B) )</td>
<td>( \text{e-gen}_B \cap (x - \text{e-kill}_B) )</td>
</tr>
<tr>
<td><strong>Boundary</strong></td>
<td>( \text{OUT[ENTRY]} = \emptyset )</td>
<td>( \text{IN[EXIT]} = \emptyset )</td>
<td>( \text{OUT[ENTRY]} = \emptyset )</td>
</tr>
<tr>
<td><strong>Meet ((\land))</strong></td>
<td>( \cup )</td>
<td>( \cup )</td>
<td>( \cap )</td>
</tr>
<tr>
<td><strong>Equations</strong></td>
<td>( \text{OUT}[B] = f_B(\text{IN}[B]) )</td>
<td>( \text{IN}[B] = f_B(\text{OUT}[B]) )</td>
<td>( \text{OUT}[B] = f_B(\text{IN}[B]) )</td>
</tr>
<tr>
<td></td>
<td>( \text{IN}[B] = \land_{p,\text{pred}(B)}\text{OUT}[P] )</td>
<td>( \text{OUT}[B] = \land_{s,\text{succ}(B)}\text{IN}[S] )</td>
<td>( \text{IN}[B] = \land_{p,\text{pred}(B)}\text{OUT}[P] )</td>
</tr>
<tr>
<td><strong>Initialize</strong></td>
<td>( \text{OUT}[B] = \emptyset )</td>
<td>( \text{IN}[B] = \emptyset )</td>
<td>( \text{OUT}[B] = \cup )</td>
</tr>
</tbody>
</table>
Thanks for a great semester!

Have a wonderful summer!

Congrats to everyone graduating!!