PART I:  Trivial Phase

Our ongoing programming project this semester will be an optimizer for a specific small language, a combination of pieces from multiple languages hereafter dubbed O’Neil’s Never Ever Implemented Language (ONEIL). When finished, the system will make a source-to-optimized-source conversion of small programs.

Project Overview

In order to optimize a program, a tool must be built to: understand (parse/unparse) a programming language; analyze the program to determine which potential optimizations are possible (safety analysis); and analyze the program to determine which transformations are profitable and how to apply the transformations (optimization configuration). There are several ways to resolve the problem. The one we choose the easy and flexible method of building a small parser/unparser for a C-like language.

ONEIL Grammar

Our language’s description as a context-free grammar is as follows:

```
<program> ::= title textLine <varDeclList> begin <stmtList> end
<varDeclList> ::= ε | var <varDecl> <varDeclList>
<varDecl> ::= ε | <id> | list [<number>] <id>
<stmtList> ::= ε | <stmt> <stmtList>
<stmt> ::= rem textLine | label <id> | let <id> = <expr> |
         | if (<expr> <relOp> <expr>) then goto <id> | goto <id> |
         | input <id> | input <id> [<number>] |
         | print <expr> | prompt "textLine" |
         | ε
<expr> ::= <term> <exprTail>
<exprTail> ::= ε | <mulOp> <factor> <exprTail>
<term> ::= <factor> <termTail>
<termTail> ::= ε | <addOp> <factor> <termTail>
<factor> ::= <id> | <id>[<expr>] <number> | (<expr>)
<relOp> ::= < | == | > | <= | >= | !=
<addOp> ::= + | -
<mulOp> ::= * | / | %
```

See that the first title command is simply a comment identifying the program, author and other information. It will be ignored by the compiler.

Identifiers are sequences of letters and digits starting with a letter. Uppercase letters are different from lowercase. Integer constants are sequences of decimal digits denoting a value, possibly preceded by a negative sign. Formally they are defined as follows:

```
<id> ::= <letter><idList>
<idList> ::= ε | <letter><idList> | <digit><idList>
<number> ::= <digit><idList> | -<digit><idList>
<digList> ::= ε | <digit><idList>
<letter> ::= A | ... | Z | a | ... | z
<digit> ::= 0 | ... | 9
```

Note what’s missing. There are no local variables, pointers or structure types in programs. The only data types are integer and one-dimensional integer arrays (lists). (We will later extend this language to include up to 3-dimensional arrays.) A set of test programs can be downloaded from the course web page.
Control Flow

Another thing omitted (because they are redundant) is complicated control flow structures, i.e. \texttt{for} and \texttt{while} loops. However, because they are convenient we will introduce them into our language for now.

\[
\begin{align*}
\texttt{<whileStmt>} & ::= \texttt{while \langle expr\rangle \langle relOp\rangle \langle expr\rangle \ <stmtList> } \texttt{ endwhile} \\
\texttt{<forStmt>} & ::= \texttt{for \langle id\rangle = \langle factor\rangle \texttt{ to } \langle factor\rangle \texttt{ by } \langle factor\rangle \ <stmtList> } \texttt{ endfor}
\end{align*}
\]

We can now give a simple ONEIL++ program for printing out a list of Fibonacci numbers.

\begin{verbatim}
title fibonacci by t. oneil, 10.26.2011
var
  list[32] array
  int idx
  int bound
begin
  let bound = 32
  rem initialize array
  for id0 = 0 to bound - 1 by 1
    let array[idx] = -1
  endfor
  let array0 = 1
  let array1 = 1
  rem end initialize array
  let idx = 0
  prompt "The first few digits of the Fibonacci sequence are:\n"
  while (idx < bound)
    rem fibonacci calculation
    if (array[idx] != -1) then goto target
    let array[idx] = array[idx - 1] + array[idx - 2]
label target
    rem end fibonacci calculation
    print array[idx]
    prompt " \\
    let idx = idx + 1
  endwhile
  prompt "\n"
end
\end{verbatim}

All programs on the web page are written to conform to this version of the language.

Helpful Notes

Completing this project will require you to read a text file corresponding to an ONEIL program and interpret the contents, a process known as \textbf{lexicographic analysis}. Initial lexers written in both C and Java are available on the course web page to provide you with a starting point. All these programs do is read a file and echo the contents to the screen. You will have to do extensive revisions to this code in order to complete this assignment.

It is also useful to observe certain patterns in both the grammar and the sample code above. Each line begins with a reserved word. Once this command is identified, we can predict what the rest of the line of code will contain. (A real compiler would have to verify the contents of code rather than make such assumptions. Since we are focused on optimization we will take for granted that we are starting with syntactically correct code.) Useful examples include: individual commands on separate lines (i.e. separated by end-of-line and carriage return characters); a single statement and not a compound statement to be executed if the condition in an \texttt{if-then} construct is true; \texttt{if} and \texttt{while} test conditions contained within parentheses; \texttt{then} found immediately following the closing parenthesis in an \texttt{if} test condition.

Task 1. Expanding Control Flow

Having just added them, the first phase of this assignment is to write an initial optimizer that replaces \texttt{for} and \texttt{while} loops with primitive operations from ONEIL as defined. These can be replaced with \texttt{gotos} in the right combination, but forces us to insert labels at the right places. These labels must have names with no potential to conflict with identifiers chosen by the user. Because the underscore character is not approved for use in user-defined names by our grammar, insert “L_” followed by an integer as compiler-defined \texttt{goto} targets. The integer value will be a counter...
value (initially zero) that gets incremented every time a new label is inserted into the program. With this in mind, it is simple textual substitution to expand `for` and `while` loops like so:

<table>
<thead>
<tr>
<th><code>while (&lt;condition&gt;)</code></th>
<th><code>goto L_counter</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;stmtList&gt;</code></td>
<td><code>label L_(counter+1)</code></td>
</tr>
<tr>
<td><code>endwhile</code></td>
<td><code>&lt;stmtList&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>label L_counter</code></td>
</tr>
<tr>
<td></td>
<td><code>if (&lt;condition&gt;)</code></td>
</tr>
<tr>
<td></td>
<td><code>then goto L_(counter+1)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>for &lt;id&gt; = &lt;factor1&gt; to &lt;factor2&gt; by &lt;factor3&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;stmtList&gt;</code></td>
</tr>
<tr>
<td><code>endfor</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>let &lt;id&gt; = &lt;factor1&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>label L_counter</code></td>
</tr>
<tr>
<td><code>&lt;stmtList&gt;</code></td>
</tr>
<tr>
<td><code>let &lt;id&gt; = &lt;id&gt; + &lt;factor3&gt;</code></td>
</tr>
<tr>
<td><code>if (&lt;id&gt; &lt;= &lt;factor2&gt;)</code></td>
</tr>
<tr>
<td><code>then goto L_counter</code></td>
</tr>
</tbody>
</table>

Because of the potential for nested loops you should consider using recursion to implement this change. For our Fibonacci example the transformation (with changes highlighted) looks like this:

```plaintext
title fibonacci, t. oneil, 10.26.2011
var
  list[32] array
  int idx
  int bound
begin
  let bound = 32
  rem initialize array
  for idx = 0 to bound - 1 by 1
    let idx = 0
    label L_0
    let array[idx] = -1
    endfor
  let idx = idx + 1
  if (idx <= bound - 1) then goto L_0
  let array[0] = 1
  let array[1] = 1
  rem end initialize array
  prompt "The ... are:\n"
  while (idx < bound)
    rem fibonacci calculation
    if (array[idx] != -1) then goto target
    let array[idx] = array[idx - 1] + array[idx - 2]
    label target
    rem end fibonacci calculation
    print array[idx]
    prompt "\n"
    let idx = idx + 1
  endwhile
  prompt "\n"
end
```

**Task 2. Writing a Simple Compiler**

One last issue to address at this point is the fact that there is no way to compile and execute ONEIL programs to check for correctness of optimizations. Fortunately we designed our grammar for simplicity, making translation to C or Java straightforward. Since we don't have time to write a complete compiler, write a stand-alone program that creates the C equivalent for ONEIL code. We've outlined some of the things that will make this easier above. For example, if code lines are separated by end-of-line/carriage-return characters, inserting semicolons between lines of C is easy. Furthermore, once the initial reserved word is identified, textual substitution follows according to this table:
### ONEIL Construct C Code for Substitution

<table>
<thead>
<tr>
<th>ONEIL Construct</th>
<th>C Code for Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>\begin{noequivalent}ignore\end{noequivalent}</td>
<td>\begin{noequivalent}ignore\end{noequivalent}</td>
</tr>
<tr>
<td>\begin{skip}return 0; \end{skip}</td>
<td>\begin{skip}return 0; \end{skip}</td>
</tr>
<tr>
<td>\begin{goto}&lt;id&gt;\end{goto}</td>
<td>\begin{goto}&lt;id&gt;\end{goto}</td>
</tr>
<tr>
<td>\begin{input}&lt;id&gt;\end{input}</td>
<td>\begin{input}&lt;id&gt;\end{input}</td>
</tr>
<tr>
<td>\begin{int}&lt;id&gt;\end{int}</td>
<td>\begin{int}&lt;id&gt;\end{int}</td>
</tr>
<tr>
<td>\begin{let}&lt;id&gt;=\begin{expr}\end{expr};\end{let}</td>
<td>\begin{let}&lt;id&gt;=\begin{expr}\end{expr};\end{let}</td>
</tr>
<tr>
<td>\begin{list}\begin&lt;number&gt;\end{number}&lt;id&gt;\end{list}</td>
<td>\begin{list}\begin&lt;number&gt;\end{number}&lt;id&gt;\end{list}</td>
</tr>
</tbody>
</table>

(\textit{then} is highlighted to remind you to remove it during translation.) Again, our Fibonacci program translated in this manner looks like this:

```c
#include <stdio.h>

int main(void) {
    int array[32];
    int idx;
    int bound = 32;
    // initialize array
    idx = 0;
    L_0:
    array[idx] = -1;
    idx = idx + 1;
    if (idx <= bound - 1) goto L_0;
    array[0] = 1;
    array[1] = 1;
    // end initialize array
    idx = 0;
    printf("The ... are:\n"");
    goto L_1;
    L_2:
    // fibonacci calculation
    if (array[idx] != -1) goto target;
    array[idx] = array[idx - 1] + array[idx - 2];
    label target:
    // end fibonacci calculation
    printf("%d", array[idx]);
    printf(" ");
    idx = idx + 1;
    L_1:
    if (idx < bound) goto L_2;
    printf("\n");
    return 0;
}
```

### Requirements

Code for the two programs (the control-flow-expander/if-then-simplifier and the simple compiler) is to be turned into the online submission system by midnight of the due date announced in class. You should also hand in a hard copy of a short (1 – 2 pages) written report at the beginning of class on the due date. Include compilation and use instructions for your programs, a short overview of your work, a brief description of tests performed and explanations of any unexpected behavior.