CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall
hw04 / pro4

- hw04 - write one program in language: a test case
- pro4 - assembly generation with two options
  - SmallPond
  - x86-64
exercises

Assume that type checking and argument list length checking has already been accounted for in the semantic actions attached to productions:

- type checking of each argument with corresponding parameter declaration (remembering that there is no coercion allowed in either an explicit or an implicit assignment)

- checking that the number of arguments and the number of parameters is the same

How will you modify your grammar rules to generate intermediate code for function calls?

How will you modify your grammar rules to generate intermediate code for the switch/case statement?
What did you come up with?

How will you modify your grammar rules to generate intermediate code for function calls?

```plaintext
assignable
  ...
  | assignable {} ablock {}
  ...
;

ablock: L_PARENTHESIS argument_list {} R_PARENTHESIS {}
  ;

argument_list
  : {} /* empty */
  | non_empty_argument_list {}
  ;

non_empty_argument_list
  : expression {}
  | expression {} COMMA non_empty_argument_list {}
  ;
```
What did you come up with?

How will you modify your grammar rules to generate intermediate code for the switch/case statement?

```plaintext
statement
  ... |
  | SWITCH L_PARENTHESIS expression {} R_PARENTHESIS case_list {} OTHERWISE COLON sblock {} |
  ... |

case_list
  : case_statement {} |
  | case_statement {} case_list {} |
  ;

case_statement
  : CASE constant {} COLON sblock {} |
  ;
```
Previous semester exercise outcome

- Basic approach teams took was to gather up information about argument expressions in an expression list, and generate the 'param' instructions at the end of the 'assignable ablock' rule, but only if assignable is a function (as opposed to an array). After the param instructions have been generated the 'call' instruction is generated, including the arity of the function (which is determined either by looking it up in the symbol table or by counting the number of arguments supplied).
Phases of a compiler

Figure 1.6, page 5 of text
Desirable characteristics of generated code:

- correctness (this is non-negotiable)
- small execution time
- small code size
- small power consumption

Associate costs with each instruction, then "minimize" (lower) overall cost, with some balance since execution time and code size can be in conflict.
Significant tasks of code generator

- instruction selection
- register allocation and assignment
- instruction ordering
Simple generation strategy vs. code size

If we generate code for each intermediate code instruction in isolation and string the results together the result may include redundant instructions.
Consider:
\[ x = y + z \]

This might be translated as:

LD R0, y  ← load the value of y into register R0
ADD R0, R0, z  ← put into R0 the result of adding R0 and the value of z
ST x, R0  ← store the value of register R0 to x
Consider applying the same template to a larger example:
\[ a = b + c \]
\[ d = a + e \]

This might be translated as:

```
LD R0, b
ADD R0, R0, c
ST a, R0
LD R0, a
ADD R0, R0, e
ST d, R0
```
Consider applying the same template to a larger example:

\[
\begin{align*}
a &= b + c \\
d &= a + e
\end{align*}
\]

This might be translated as:

\[
\begin{align*}
&\text{LD R0, b} \\
&\text{ADD R0, R0, c} \\
&\text{ST a, R0} \\
&\text{LD R0, a} \\
&\text{ADD R0, R0, e} \\
&\text{ST d, R0}
\end{align*}
\]

This instruction is redundant: it is loading into R0 the value that is already there.
Basic blocks and flow graphs

To help us analyze the intermediate code we will group instructions from our program into "basic blocks".
Basic Block

A basic block is a "maximal sequence of consecutive three-address instructions with the properties that,

a) the flow of control can only enter the basic block through the first instruction in the block [...] 

b) control will leave the block without halting or branching, except possibly at the last instruction in the block"
Flow Graph

"The basic blocks become the nodes of a flow graph, whose edges indicate which blocks can follow which other blocks."

[p 526]
Algorithm 8.5 [p. 526]
INPUT: a sequence B of three-address instructions.
OUTPUT: a list of basic blocks for B, in which each instruction is
assigned to exactly one basic block
METHOD: First, find leaders (see below).
For each leader, its basic block consists of itself and all
instructions up to but not including the next leader, or the end
of the intermediate program." [lightly edited from original]

"The rules for finding leaders are:
1) The first three address instruction (3AI) in the intermediate
code is a leader.
2) Any instruction that is the target of a (conditional or
unconditional) jump is a leader.
3) Any instruction that immediately follows a (conditional or
unconditional) jump is a leader." [lightly edited from original]
for (i=1; i<=10; i=i+1) {
    for (j=1; j<=10; j=j+1) {
        a[i,j] = 0.0;
    }
}
for (i=1; i<=10; i=i+1) {
    a[i,i] = 1.0;
}

This code initializes a 10x10 real matrix to the identity matrix (1's along the main diagonal).

Assumptions:
matrix is of size 10x10 containing reals
a real occupies 8 bytes
matrix is stored in row-major form (see p. 382)
A possible three-address code translation of the high-level program.

1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
Identifying leaders

1) \( i = 1 \)
2) \( j = 1 \)
3) \( t_1 = 10 \times i \)
4) \( t_2 = t_1 + j \)
5) \( t_3 = 8 \times t_2 \)
6) \( t_4 = t_3 - 88 \)
7) \( a[t_4] = 0.0 \)
8) \( j = j + 1 \)
9) if \( j \leq 10 \) goto (3)
10) \( i = i + 1 \)
11) if \( i \leq 10 \) goto (2)
12) \( i = 1 \)
13) \( t_5 = i - 1 \)
14) \( t_6 = 88 \times t_5 \)
15) \( a[t_6] = 1.0 \)
16) \( i = i + 1 \)
17) if \( i \leq 10 \) goto (13)

Leaders are:
1. first instruction
2. the target of any jump
3. the instruction immediately after any jump
Each basic block is a node in the flow graph.

There is an edge between blocks B and C of the flow graph if:

1. there is a (conditional or unconditional) jump from the end of B to the start of C, or
2. C immediately follows B and B does not end with an unconditional jump.
Terminology

- B is a predecessor of C
- C is a successor of B
Flow Graph
Figure 8.9 [p. 530]

Entry and exit nodes added.

Jump targets replaced by block names.