Control Flow

- Given an expression in a language, the student will be able to generate code for the expression with proper precedence and short-circuit evaluation
- Given a conditional statement, the student will be able to generate code to state the meaning of the conditional statement.
- Given an iterative construct, the student will be able to implement that construct to give its meaning.

Control Mechanisms

1. Sequencing – order statements and expressions are evaluated
2. Selection – choices among two or more statements or expressions
3. Iteration – repetitive execution of statements or expressions
4. Procedural abstraction – encapsulation of statements or expressions in subroutines subject to parameterization
5. Recursion – An expression or subroutine defined in terms of itself
6. Concurrency – two or more program fragment may be executed simultaneously
7. Exception handling – run-time error handling mechanism
8. Nondeterminism – ordering of statements or expression left unspecified

Issues in Expression Evaluation

- Operator precedence and associativity
  - How is the following expression to be evaluated?

    3 + 4 * 5 ** 3 ** 2 // ** is exponentiation in Fortran

    - What operator has the highest precedence?
    - What order should exponentiation be done? (associativity)
    - What is the value computed in Fortran?
- The language specifies both precedence and associativity.
  - How does the compiler follow these rule? (What was done in Cminus?)

Example Precedence Hierarchies (high to low)

Fortran	Pascal	C	Ada
		++, -- (post)	
**	not	++, -- (pre), +,- (u), &, * (addr) , !, ~	abs, not, **
*, /	*, /, div, mod, and	* (b), /, %	*, /, mod, rem
+, -	+, -, or	+, - (b)	+, - (u)
		<<, >>	
.eq., .ne., .lt., …	=, <>, <, …	<, <=, >, >=	=, /=, <, …
.not.		==, !=	
		&	
		^	
		|	
.and.		&&	and, or, xor
.or.		||	
.eqv., neqv.		?:	
		=, +=, -=, …	
		,	

Precedence Problems

- C is too complex
- Pascal is too simple

    var a,b,c,d : integer;

    if a < b and c < d then …

  What is the result?

Practice Problem

- Evaluate the expression 6 * 8 + 4 / 2 ** 2 ** 0 under both of the following precedence rules (high to low)

Scheme 1	Scheme 2
+	*, /
*, /	** (left associative)
** (right associative)	+

Side Effects

- A programming language construct has a side effect if it influences subsequent computation in any way other than by returning a value for use in the surrounding context. This is a product of the von Neumann model.
  - Examples
    - assignment
    - i/o
    - etc.
- Expressions always produce a value
- Statements are executed for their side effects

Assignment

- What is the meaning of assignment?
- Let's add assignment to our calc language

$$
\begin{array}{lll}
\mathit{Calc} & \rightarrow & \mathit{Calc} \,;\, \mathit{Assign} \\
 & | & \mathit{Assign} \\
\mathit{Assign} & \rightarrow & \mathit{Id} := \mathit{Expr}
\end{array}
$$

  What options do we have for giving assignment meaning?

Value Model: L-values and R-values

- In C, what is the meaning of the following?

  d = a;
  a = b + c;

  In particular, is there a difference between the reference to a on the right-hand side of the assignment and the one on the left-hand side of the assignment?
- References that denote values are called r-values. References that denote memory locations are called l-values.
  - An r-value may denote a value stored in a location.
  - An l-value denotes the value itself.
- Only references that may refer to locations can be used as l-values.
  - Which of the following are valid in C (where f is function that returns a pointer)?

```
f(a)->c = 2;
(f(a) + c) = 2;
*(f(a)+4) = 2;
```


Reference model

- In some languages, variables are not containers for values, but rather as named references to a value.
- In the reference model, all values are l-values that are dereferenced (either implicitly or explicitly) when used in a context that expects an r-value.

Code | Value Model | Reference Model

```
b = 2;
c = b;
a = b + c;
```

Reference Model

- Conceptually there is only one copy of any value.
  - What happens when assignment is added?
  - This will be discussed more in functional languages section
- In practice, most compilers use multiple copies of immutable objects.

Side Effects

- Languages with assignment allow potentially unintended effects.
  - Examples

    b = foo(a,b,update(&b),b);

    a[goo(i)] = a[goo(i)] + 10;

    c = (x – h(&y)) * (y + x);

    What is the meaning of these three assignments?
- The ambiguity related to side effects is why many prefer a purely functional style of programming (discussed later this semester).

Initialization

- What is the meaning of a reference (load from) an uninitialized variable?

  int a;
  int c = a + 1;

  - Some options
    - Undefined
    - Error
    - Zero

Selection

- Most languages allow variants of the if-then-else construct introduced in Algol 60.

  if *condition* then *statement*
  else if *condition* then *statement*
  else if *condition* then *statement*
  …
  else *statement*

  How do we implement this?

Condition Evaluation

- Conditions often consist of conjunctions or disjunctions. Given the following, how should we evaluate the condition?

  if (a != 0 && c/a > 1) …

  - Options
    - Evaluate every expression (Pascal)
    - Short-circuit evaluation (C)
      - Take advantage of logical dominance
        - 0 && p = 0
        - 1 || p = 1

Meaning of (a != 0 && c/a > 1)

- Pascal

```
lw   $t0, -4($fp)
sne $t0, $t0, $zero
lw   $t1, -8($fp)
lw   $t2, -4($fp)
div $t1, $t1, $t2
li    $t2, 1
sgt $t1, $t1, $t2
and $t0, $t0, $t1
# generate if-code
```

- C

```
  lw   $t0, -4($fp)
  beq $t0, $zero, .L1
  lw   $t1, -8($fp)
  lw   $t0, -4($fp)
  div  $t1, $t1, $t0
  li     $t2, 1
  ble  $t1, $t2, .L1
  # generate if-code #
  …
.L1:    nop
```

Meaning of if-then-else

```
if condition then
   statement_1
else
   statement_2
endif
```

```
# code for condition, result in $t0
        beq     $t0, $zero, .L1
# code for statement_1
        j       .L2
.L1:    nop
# code for statement_2
.L2:    nop
```


Meaning of Case Statement

- A shorthand method to express a nested if-then-else, uses a case statement (Pascal)

  if a = 1 then *clause*$_A$
  else if a = 2 or a = 3 then *clause*$_B$
  else if a = 4 then *clause*$_C$
  else *clause*$_D$

- Case statement

  case a of
  1: *clause*$_A$
  2,3: *clause*$_B$
  4: *clause*$_C$
  else *clause*$_D$
  end;

- Implement just using same method as if-then-else

Meaning of Case (Jump Table)

- If the labels for the case clauses are dense in distribution, a jump table may yield better performance.

  case a of
  1: *clause*$_A$
  2,3: *clause*$_B$
  4: *clause*$_C$
  else *clause*$_D$
  end;

- Jump table implementation

```
T:      .word       0
        .word       .L1
        .word       .L2
        .word       .L2
        .word       .L3
        ...
# a is in $t0
        ble     $t0,$zero, .L4
        li      $t1, 4
        bgt     $t0, $t1, .L4
        la      $t1, T
        mul     $t0,$t0,4
        add     $t1, $t1, $t0
        lw      $t2, 0($t1)
        jr      $t2
.L1  # code for clauseA
        j       .L5
.L2  # code for clauseB
        j       .L5
.L3  # code for clauseC
        j       .L5
.L4  # code for clauseD
.L5     nop
```


C Switch Statement

- Give code to express the meaning of the following C switch statement

```
switch (a) {
    case 1:     clause_A
    case 2:
    case 3:     clause_B
    case 4:     clause_C
                break;
    default:    clause_D
}
```

Loops

- What is the meaning of a for-loop?

  for i = 0 to 9 by 1
      body

```
        sw      $zero, -4($fp)
.L1:    lw      $t0, -4($fp)
        li      $t1, 9
        bgt     $t0, $t1, .L2
         body
        lw      $t0, -4($fp)
        add     $t0, $t0, 1
        sw      $t0, -4($fp)
        j       .L1
.L2:    nop
```

Why do we reload i and the upper bound every time?

Loop Code Shape

Below is an alternate form for a for loop. Which is better?

```
        sw      $zero, -4($fp)
        lw      $t0, -4($fp)
        li      $t1, 9
        bgt     $t0, $t1, .L2
.L1     nop
         body
        lw      $t0, -4($fp)
        add     $t0, $t0, 1
        sw      $t0, -4($fp)
        li      $t1, 9
        ble     $t0, $t1, .L1
.L2:    nop
```


Book's Implementation

The following is the book's implementation

```
        sw      $zero, -4($fp)
        lw      $t0, -4($fp)
        li      $t1, 9
        j       .L3
.L1     nop
         body
        lw      $t0, -4($fp)
        add     $t0, $t0, 1
        sw      $t0, -4($fp)
        li      $t1, 9
.L3     ble     $t0, $t1, .L1
.L2:    nop
```

Unfortunately, jumping into the middle of the loop is a bad idea and has serious consequences in the middle end of the compiler (optimization). Never do this!!

More loops

- Give the meaning of the a while-loop and a repeat-until-loop with the following syntax.

  while (*expr*) {
  }

  repeat {

  } until (*expr*);

Structured Control Flow

- Goto is generally considered a bad idea.
- C adds continue and break statements to give structured control flow within loops, etc.

```
for () {
    // start of loop code
    break;
    // middle of loop code
    continue;
    // end of loop code
}
```

- Implementation

```
.L1 // loop header
// start of loop code
        j       .L2
// middle of loop code
        j       .L3
// end of loop code
.L3     lw      $t0, 0($fp)
        add     $t0, $t0, 1
        sw      $t0, 0($fp)
        li      $t1, 9
        ble     $t0, $t1, .L1
.L2:    nop
```

25