Code Generation

- Two possibilities:
  - Generate intermediate level code.
    - Port a compiler to different architectures by porting only IR to code generation component.
    - Perform optimization on the intermediate level code.
  - Generate code for a target final machine.
- From IR to final code generation:
  - Macro expansion: replace each IR by a set of target machine instructions. Naive and can be inefficient. Also does not utilize the state of program to generate efficient program. Also code bloating.
  - Threaded code approach: Replace each IR instruction with a subroutine call to a support routine that implements the IR instruction.
- Our focus: some intermediate level, and then target machine. Understand what it means to generate code first.
- Types of Intermediate code:
  1. Syntax or parse tree: Similar to the parse tree that you created in your project.
  2. Three address code: closer to assembly but not yet.
     Examples:
     - Pascal P-compiler generated *P*-code for a hypothetical virtual stack machine.
     - Smalltalk and Java byte code.

Three Address Code

- Every statement contains upto three operands and upto one operator:

  `x := y` *op* `z`

  x, y, z: names, constants, or temporaries
- Example translation of expression $x + y * z$:

  ```
  t1 := y * z;
  t2 := x + t1;
  ```

  Implications: Will need to generate temporaries to store intermediate values.
  - How will they be generated?
  - Where will they be stored?
  - What will their type be?
  - Where will they be during execution?
- Design of three address language: operator set must be complete and orthogonal.
  - Easy to do final code generation.
  - small instruction set means bloated code.
- Assume the following instruction set for rest of lecture:
  1. assignment statement **(x := y op z)**. op: logical or arithmetic operator (for instance, +, * etc.);y, op: optional.
  2. unconditional jump **(goto L)**
  3. conditional jump: **if (x relop y) goto L**
  4. param x, call p, n, and return y.

     ```
     param x1
        :
     param xn
     call p, n
     ```
  5. x := y[i] and y[i] := x.

Intermediate Code Generation: Assumptions

- A local symbol table stores all symbols (identifiers) and their attributes (e.g., type)
- Two kinds of entries for variables: identifiers defined in program and temporaries generated on the fly. Assume the following:
  - `id.SymTab`: Denotes pointer to the entry into the local symbol table for identifier `id`.
    Can use this pointer to determine name of id, its attributes, etc.
  - E`.SymTab`: Denotes pointer to a symbol table entry that will store E's value. Note that each expression will evaluate to a certain value. This value will need to be stored in memory during runtime. Assume that an entry (typically a temporary variable) for this exists in symbol table.
- `Print`: a function used to construct a string from its parameters and output it.
  - If it is number or string, just output string
  - If its parameter is a pointer to an entry in symbol table, it then prints the string associated with the entry.

  Example: `Print(id1 ':=' id2 '+' id3)` produces `x := y + z`. Here `id1`, `id2`, and `id3` are pointers to symbol table entries. `Print` evaluates these entries and prints the corresponding entries in symbol table (x, y, and z respectively in this case). if id1, id2, id3 are not pointers to symbol table entries, then it prints corresponding character string.

Assumptions - cont'd.

- E.`code`: A semantic attribute;
  Stores code sequence that will evaluate string denoted by E
- `GenCode` methods associated with each non terminal. Two kinds:
  - E.`GenCode()`: a function that generates code for nonterminal E. For every nonterminal type, you will need to define a corresponding `GenCode()`.
  - E.`GenCode(label1, label2)`: a function that generates code for nonterminal E. It takes two labels `label1` and `label2` as parameters. The labels act as "inherited attributes" and are passed down in the tree as "goto" targets.
- `GenLabel()`: a function used to generate symbolic labels.
- `GenTemp()`: function that creates a temporary variable. It also puts the variable in the current symbol table (local scope).
  Returns pointer to symbol table entry.
  Ignore type of temporary for now.

Code Generation - cont'd.

- Generation of code for assignment statement of form:

  ```
  var := expression
  ```

- Steps: (i) Generate code for evaluating right hand side ii) Generate code for puting value in location defined for var.
- S $\rightarrow$ id := E

```
S.GenCode() {
  E.GenCode();
  Print(id.SymTab ':=' E.SymTab);
}
```

- E $\rightarrow$ E1 op E2 where op is a binary operator:

```
E.GenCode () {
   E1.GenCode();
   E2.GenCode();
   E.SymTab := GenTemp();
   Print(E.SymTab ':=' E1.SymTab op E1.SymTab);
}
```

- E $\rightarrow$ UnaryOp E1 where UnaryOp is a unary operator

```
E.GenCode () {
   E1.GenCode();
   E.SymTab := GenTemp();
   Print(E.SymTab ':=' UnaryOp E1.SymTab);
}
```

- E $\rightarrow$ (E1)

```
E.GenCode () {
   E1.GenCode();
   E.SymTab := E1.SymTab;
}
```

- E $\rightarrow$ **id**

```
E.GenCode () {
   E.SymTab := id.SymTab
}
```

Boolean Expressions

- Expressions containing operators such as `and`, `or`, `not`, and relational operators.
- Two ways of evaluating boolean expressions:
  1. Evaluate like arithmetic expressions
  2. Evaluate incrementally: compute only required component.
     Example: For expression E1 or E2, evaluate E1. If true, do not evaluate E2. (assumption: no side effects).
     Our focus.
- For each boolean expression E, associate two labels (attributes): E`.true`, E`.false`. Generated code for E looks like the following (assume that tempE is the name of place that holds value for E):

```
Code for evaluating E;
if (tempE = true) then goto E.true;
goto E.false
```

- E $\rightarrow$ true

```
E.GenCode (label1, label2) {
   Print('goto' E.true);
}
```

- E $\rightarrow$ false

```
E.GenCode (label1, label2) {
   Print('goto' E.false);
}
```

Boolean Expressions - cont'd.

- E → E1 or E2

```
E.GenCode (E.true, E.false) {
   E1.true := E.true; E1.false := GenLabel();
   E2.true := E.true; E2.false := E.false;
   E1.GenCode(E1.true, E1.false);
   Print(E1.false ':');
   E2.GenCode(E2.true, E2.false);
}
```

- E → E1 and E2

```
E.GenCode (E.true, E.false) {
   E1.true := GenLabel(); E1.false := E.false;
   E2.true := E.true; E2.false := E.false;
   E1.GenCode(E1.true, E1.false);
   Print(E1.true ':');
   E2.GenCode(E2.true, E2.false);
}
```

- E → not E1

```
E.GenCode (E.true, E.false) {
   E1.true := E.false;
   E1.false := E.true;
   E1.GenCode(E1.true, E1.false);
}
```

- E → (E1)

```
E.GenCode (E.true.  E.false) {
   E1.true := E.true;
   E1.false := E.false;
   E1.GenCode(E1.true, E1.false);
}
```

- E → id1 relop id2

```
E.GenCode (E.true, E.false) {
   Print('if' id1.SymTab relop id2.SymTab);
   Print('goto' E.true);
   Print('goto' E.false);
}
```

Processing Control Flow Statements

- S $\rightarrow$ if E then S1 else S2

```
E.GenCode () {
    E.true := GenLabel(); E.false := GenLabel();
    S.next := GenLabel();
    E.GenCode(E.true, E.false);
    Print('E.true:');
    S1.GenCode();
    Print('goto' S.next);
    Print('E.false:');
    S2.GenCode();
    Print('S.next:');
}
```

- S $\rightarrow$ while E do S1

```
E.GenCode () {
    S.begin := GenLabel();
    E.true := GenLabel(); E.false := GenLabel();
    Print('S.begin:');
    E.GenCode(E.true, E.false);
    Print('E.true:');
    S1.GenCode();
    Print('goto' S.begin);
    Print('E.false:');
}
```

- Case statement: Evaluate selection expression. Execute code associated with the case.
  1. For small number of cases, use if and gotos. (Treat like a large if-then-else)
  2. For large cases: i) Create a table of pairs: (value, label for code) Could even use hash table by hashing value. In your case, you could simply index into the table by using the value. ii) Generate code for each case with suitable label at the end. iii) Jump to valid code through table.

Procedure Calls

- Implemented by defining a calling sequence:
  1. Sequence of actions to take just before entering a procedure
  2. Sequence of actions to take after entering a procedure
  3. Sequence of actions to take after returning from a procedure.
- Typical actions when about to call:
  1. Allocate space for activation record of called procedure
  2. Evaluate arguments of called procedure and make available to called procedure on stack
  3. Store control and access links to enable called procedure to access data in enclosed blocks.
  4. Save state of calling procedure
  5. Save return address
- Typical actions after entering:
  1. Allocate space for local variables and temps
  2. Execute program
- Typical actions while returning:
  1. Store result at a known location
  2. Restore activation record of calling procedure
  3. Jump to calling procedure's return address