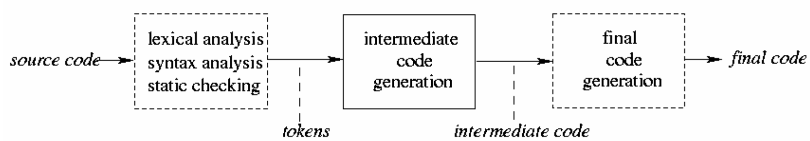
CSc 453
Intermediate Code Generation

Saumya Debray

The University of Arizona

Tucson

Overview

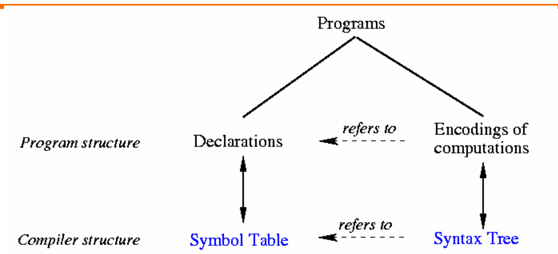


- Intermediate representations span the gap between the source and target languages:
  - closer to target language;
  - (more or less) machine independent;
  - allows many optimizations to be done in a machine-independent way.
- Implementable via syntax directed translation, so can be folded into the parsing process.

Types of Intermediate Languages

- *High Level Representations* (e.g., syntax trees):
  - closer to the source language
  - easy to generate from an input program
  - code optimizations may not be straightforward.
- *Low Level Representations* (e.g., 3-address code, RTL):
  - closer to the target machine;
  - easier for optimizations, final code generation;

Syntax Trees



A *syntax tree* shows the structure of a program by abstracting away irrelevant details from a parse tree.

- Each node represents a computation to be performed;
- The children of the node represents what that computation is performed on.

Syntax trees decouple parsing from subsequent processing.

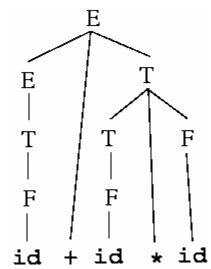
Syntax Trees: Example

Grammar :

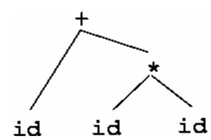
E → E + T | T
T → T * F | F
F → (E) | **id**

Input: **id + id * id**

Parse tree:

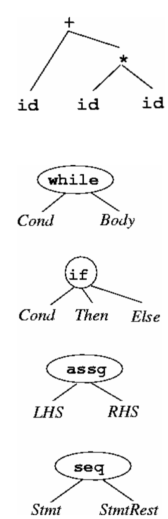


Syntax tree:



Syntax Trees: Structure

- Expressions:
  - leaves: identifiers or constants;
  - internal nodes are labeled with operators;
  - the children of a node are its operands.
- Statements:
  - a node's label indicates what kind of statement it is;
  - the children correspond to the components of the statement.



Constructing Syntax Trees

General Idea: construct bottom-up using synthesized attributes.

```
E → E + E                    { $$ = mkTree(PLUS, $1, $3); }

S → if '(' E ')' S OptElse   { $$ = mkTree(IF, $3, $5, $6); }
OptElse → else S             { $$ = $2; }
        | /* epsilon */      { $$ = NULL; }

S → while '(' E ')' S        { $$ = mkTree(WHILE, $3, $5); }
```

mkTree(NodeType, Child1, Child2, …) allocates space for the tree node and fills in its node type as well as its children.

Three Address Code

- Low-level IR
- instructions are of the form '**x = y *op* z**,' where **x**, **y**, **z** are variables, constants, or "temporaries".
- At most one operator allowed on RHS, so no 'built-up" expressions.

  Instead, expressions are computed using temporaries (compiler-generated variables).

Three Address Code: Example

- *Source*:

```
if ( x + y*z > x*y + z)
   a = 0;
```

- *Three Address Code*:

```
  tmp1 = y*z
  tmp2 = x+t1         // x + y*z
  tmp3 = x*y
  tmp4 = t3+z         // x*y + z
  if (tmp2 > tmp4) goto L
  a = 0
L:
```


An Intermediate Instruction Set

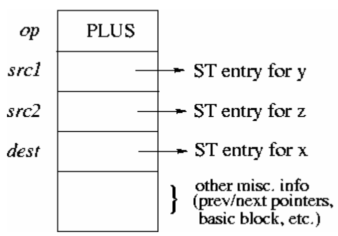
- *Assignment*:
  - x = y *op* z (*op* binary)
  - x = *op* y (*op* unary);
  - x = y
- *Jumps*:
  - if (x *op* y) goto L (L a label);
  - goto L
- *Pointer and indexed assignments*:
  - x = y[z]
  - y[z] = x
  - x = &y
  - x = *y
  - *y = x.
- *Procedure call/return*:
  - param x, k (x is the k^{th} param)
  - retval x
  - call p
  - enter p
  - leave p
  - return
  - retrieve x
- *Type Conversion*:
  - x = cvt_*A*_to_*B* y (*A*, *B* base types) e.g.: cvt_int_to_float
- *Miscellaneous*
  - label L

Three Address Code: Representation

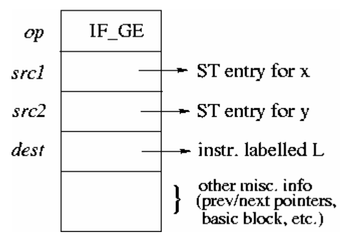
- Each instruction represented as a structure called a *quadruple* (or "*quad*"):
  - contains info about the operation, up to 3 operands.
  - for operands: use a bit to indicate whether constant or ST pointer.

E.g.:

x = y + z



if (x ≥ y) goto L



Code Generation: Approach

- function prototypes, global declarations:
  - save information in the global symbol table.
- function definitions:
  - function name, return type, argument type and number saved in global table (if not already there);
  - process formals, local declarations into local symbol table;
  - process body:
    - construct syntax tree;
    - traverse syntax tree and generate code for the function;
    - deallocate syntax tree and local symbol table.

Code Generation: Approach

Recursively traverse syntax tree:

- Node type determines action at each node;
- Code for each node is a (doubly linked) list of three-address instructions;
- Generate code for each node after processing its children

```
codeGen_stmt(synTree_node S)
{
    switch (S.nodetype) {
      case FOR:     … ; break;
      case WHILE :  … ; break;
      case IF:      … ; break;
      case '=' :    … ; break;
    …
}
```

```
codeGen_expr(synTree_node E)
{
    switch (E.nodetype) {
      case '+':   … ; break;
      case '*' :  … ; break;
      case '–':   … ; break;
      case '/' :  … ; break;
    …
}
```

recursively process the children, then generate code for this node and glue it all together.

Intermediate Code Generation

Auxiliary Routines:

- *struct symtab_entry *newtemp(typename t)*

  creates a symbol table entry for new temporary variable each time it is called, and returns a pointer to this ST entry.
- *struct instr *newlabel()*

  returns a new label instruction each time it is called.
- *struct instr *newinstr(arg$_1$, arg$_2$, …)*

  creates a new instruction, fills it in with the arguments supplied, and returns a pointer to the result.

Intermediate Code Generation…

- struct symtab_entry *newtemp(t)

```
{
  struct symtab_entry *ntmp = malloc( … );      /* check: ntmp == NULL? */
  ntmp->name = …create a new name that doesn't conflict…
  ntmp->type = t;
  ntmp->scope = LOCAL;
  return ntmp;
}
```

- struct instr *newinstr(opType, src1, src2, dest)

```
{
  struct instr *ninstr = malloc( … );           /* check: ninstr == NULL? */
  ninstr->op = opType;
  ninstr->src1 = src1; ninstr->src2 = src2; ninstr->dest = dest;
  return ninstr;
}
```


Intermediate Code for a Function

Code generated for a function *f*:

- begin with 'enter *f* ', where *f* is a pointer to the function's symbol table entry:
  - this allocates the function's activation record;
  - activation record size obtained from *f* 's symbol table information;
- this is followed by code for the function body;
  - generated using codeGen_stmt(…) [to be discussed soon]
- each return in the body (incl. any implicit return at the end of the function body) are translated to the code

```
leave f    /* clean up: f a pointer to the function's symbol table entry */
return     /* + associated return value, if any */
```

Simple Expressions

Syntax tree node for expressions augmented with the following fields:

- type: the type of the expression (or "error");
- code: a list of intermediate code instructions for evaluating the expression.
- place: the location where the value of the expression will be kept at runtime:

Simple Expressions

Syntax tree node for expressions augmented with the following fields:

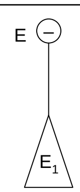
- type: the type of the expression (or "error");
- code: a list of intermediate code instructions for evaluating the expression.
- place: the location where the value of the expression will be kept at runtime:
  - When generating intermediate code, this just refers to a symbol table entry for a variable or temporary that will hold that value;
  - The variable/temporary is mapped to an actual memory location when going from intermediate to final code.

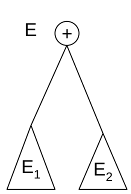
Simple Expressions 1

Syntax tree node E	***Action during intermediate code generation***
E (intcon)	codeGen_expr(E) { /* E.nodetype == INTCON; */ E.place = *newtemp*(E.type); E.code = 'E.place = intcon.val'; }
E (id)	codeGen_expr(E) { /* E.nodetype == ID; */ /* E.place is just the location of **id** (nothing more to do) */ E.code = NULL; }

Simple Expressions 2

Syntax tree node E	***Action during intermediate code generation***

	codeGen_expr(E)
	{
	/* E.nodetype == UNARY_MINUS */
	codeGen_expr(E_1); /* recursively traverse E_1, generate code for it */
	E.place = *newtemp*(E.type); /* allocate space to hold E's value */
	E.code = E_1.code $\oplus$ *newinstr*(UMINUS, E_1.place, NULL, E.place);
	}

	codeGen_expr(E)
	{
	/* E.nodetype == '+' … other binary operators are similar */
	codeGen_expr(E_1);
	codeGen_expr(E_2); /* generate code for E_1 and E_2 */
	E.place = *newtemp*(E.type); /* allocate space to hold E's value */
	E.code = E_1.code $\oplus$ E_2.code $\oplus$ *newinstr*(PLUS, E_1.place, E_2.place, E.place);
	}

Accessing Array Elements 1

- Given:
  - an array $A[lo...hi]$ that starts at address b;
  - suppose we want to access $A[\ i\ ]$.
- We can use indexed addressing in the intermediate code for this:
  - $A[\ i\ ]$ is the $(i + lo)^{th}$ array element starting from address b.
  - Code generated for $A[\ i\ ]$ is:

```
t1 = i + lo
t2 = A[ t1 ]    /* A being treated as a 0-based array at this level. */
```


Accessing Array Elements 2

- In general, address computations can't be avoided, due to pointer and record types.
- Accessing $A[\ i\ ]$ for an array $A[lo...hi]$ starting at address b, where each element is w bytes wide:

  Address of $A[\ i\ ]$ is $b + (\ i - lo\ ) * w$
  $= (b - lo * w) + i * w$
  $= k_A + i * w.$

  k_A depends only on A, and is known at compile time.
- Code generated:

```
t1 = i * w
t2 = k_A + t1    /* address of  A[ i ]  */
t3 = *t2
```

Accessing Structure Fields

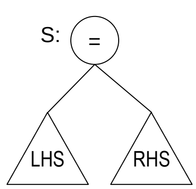
- Use the symbol table to store information about the order and type of each field within the structure.
  - Hence determine the distance from the start of a struct to each field.
  - For code generation, add the displacement to the base address of the structure to get the address of the field.
- *Example*: Given

```
struct s { … }  *p;
…
x = p→a;        /* a is at displacement δa within struct s */
```

The generated code has the form:

```
t1 = p + δa     /* address of p→a */
x = ∗t1
```


Assignments



Code structure:

evaluate LHS

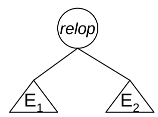
evaluate RHS

copy value of RHS into LHS

```
codeGen_stmt(S):
/* base case: S.nodetype = 'S' */
    codeGen_expr(LHS);
    codeGen_expr(RHS);
    S.code = LHS.code
               ⊕ RHS.code
               ⊕ newinstr(ASSG,
                           LHS.place,
                           RHS.place) ;
```

Logical Expressions 1

- *Syntax tree node*:



- *Naïve but Simple Code* (TRUE=1, FALSE=0):

```
        t1 = { evaluate E1
        t2 = { evaluate E2
        t3 = 1        /* TRUE */
        if ( t1 relop t2 ) goto L
        t3 = 0        /* FALSE */
    L: ...
```

- *Disadvantage*: lots of unnecessary memory references.

Logical Expressions 2

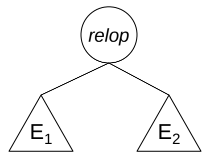
- *Observation*: Logical expressions are used mainly to direct flow of control.
- *Intuition*: "tell" the logical expression where to branch based on its truth value.
  - When generating code for *B*, use two inherited attributes, *trueDst* and *falseDst*. Each is (a pointer to) a *label* instruction.

    E.g.: for a statement **if** (*B*) S_1 **else** S_2 :

    B.trueDst = start of S_1
    B.falseDst = start of S_2
  - The code generated for *B* jumps to the appropriate label.

Logical Expressions 2: cont'd

Syntax tree:

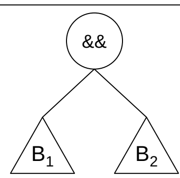


codeGen_bool(B, *trueDst*, *falseDst*):
/* base case: B.nodetype == *relop* */
B.code = E1.code
⊕ E2.code
⊕ *newinstr*(*relop*, E1.place, E2.place, *trueDst*)
⊕ *newinstr*(GOTO, *falseDst*, NULL, NULL);

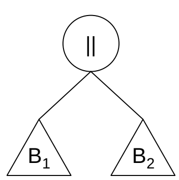
Example: $B \Rightarrow$ x+y > 2*z.
Suppose *trueDst* = Lbl1, *falseDst* = Lbl2.

E_1 ≡ x+y, E_1.place = tmp_1, E_1.code ≡ ⟨ 'tmp_1 = x + y' ⟩
E_2 ≡ 2*z, E_2.place = tmp_2, E_2.code ≡ ⟨ 'tmp_2 = 2 * z' ⟩
B.code = E_1.code ⊕ E_2.code ⊕ 'if (tmp_1 > tmp_2) goto Lbl1' ⊕ goto Lbl2
= ⟨ 'tmp_1 = x + y' , 'tmp_2 = 2 * z', 'if (tmp_1 > tmp_2) goto Lbl1' , goto Lbl2 ⟩

Short Circuit Evaluation



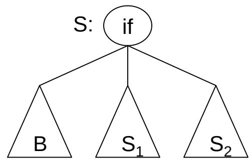
codeGen_bool (B, *trueDst*, *falseDst*):
/* recursive case 1: B.nodetype == '&&' */
L_1 = *newlabel*( );
codeGen_bool(B_1, L_1, *falseDst*);
codeGen_bool(B_2, *trueDst*, *falseDst*);
B.code = B_1.code ⊕ L_1 ⊕ B_2.code;



codeGen_bool (B, *trueDst*, *falseDst*):
/* recursive case 2: B.nodetype == '||' */
L_1 = *newlabel*( );
codeGen_bool(B_1, *trueDst*, L_1);
codeGen_bool(B_2, *trueDst*, *falseDst*);
B.code = B_1.code ⊕ L_1 ⊕ B_2.code;

Conditionals

Syntax Tree:

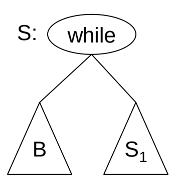


- *Code Structure*:

  code to evaluate B
  L_{then}: code for S1
  goto L_{after}
  L_{else}: code for S2
  L_{after} : …

```
codeGen_stmt(S):
/* S.nodetype == 'IF' */
   L_then = newlabel();
   L_else = newlabel();
   L_after = newlabel();
   codeGen_bool(B, L_then , L_else);
   codeGen_stmt(S_1);
   codeGen_stmt(S_2);
   S.code = B.code
            ⊕ L_then
            ⊕ S_1.code
            ⊕ newinstr(GOTO, L_after)
            ⊕ L_else
            ⊕ S_2.code
            ⊕ L_after ;
```


Loops 1

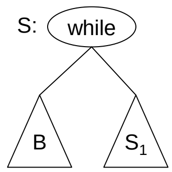


Code Structure:

L_{top} : code to evaluate B
if (!B) goto L_{after}
$L_{body:}$ code for S_1
goto L_{top}
L_{after}: …

```
codeGen_stmt(S):
/* S.nodetype == 'WHILE' */
   L_top = newlabel();
   L_body = newlabel();
   L_after = newlabel();
   codeGen_bool(B, L_body, L_after);
   codeGen_stmt(S_1);
   S.code = L_top
            ⊕ B.code
            ⊕ L_body
            ⊕ S1.code
            ⊕ newinstr(GOTO, L_top)
            ⊕ L_after ;
```

Loops 2



Code Structure:

goto L_{eval}
L_{top} :
code for S_1
L_{eval}: code to evaluate B
if (B) goto L_{top}
L_{after}:

This code executes fewer branch ops.

```
codeGen_stmt(S):
/* S.nodetype = 'WHILE' */
    L_top = newlabel();
    L_eval = newlabel();
    L_after = newlabel();
    codeGen_bool(B, L_top, L_after);
    codeGen_stmt(S_1);
    S.code =
          newinstr(GOTO, L_eval)
          ⊕ L_top
          ⊕ S_1.code
          ⊕ L_eval
          ⊕ B.code
          ⊕ L_after ;
```


Multi-way Branches: switch statements

- *Goal*:
  generate code to (efficiently) choose amongst a fixed set of alternatives based on the value of an expression.
- *Implementation Choices*:
  - *linear search*
    - best for a small number of case labels (≈ 3 or 4)
    - cost increases with no. of case labels; later cases more expensive.
  - *binary search*
    - best for a moderate number of case labels (≈ 4 – 8)
    - cost increases with no. of case labels.
  - *jump tables*
    - best for large no. of case labels (≥ 8)
    - may take a large amount of space if the labels are not well-clustered.

Background: Jump Tables

- A jump table is an array of code addresses:
  - *Tbl*[*i*] is the address of the code to execute if the expression evaluates to *i*.
  - if the set of case labels have "holes", the correspond jump table entries point to the default case.
- *Bounds checks*:
  - Before indexing into a jump table, we must check that the expression value is within the proper bounds (if not, jump to the default case).
  - The check

    lower_bound ≤ *exp_value* ≤ *upper bound*

    can be implemented using a single unsigned comparison.

Jump Tables: cont'd

- Given a ***switch*** with max. and min. case labels c_{max} and c_{min}, the jump table is accessed as follows:

Instruction	***Cost*** (cycles)
$t_0 \leftarrow$ value of expression	…
$t_0 = t_0 - c_{min}$	1
if $\neg(t_0 \leq_u c_{max} - c_{min})$ goto *DefaultCase*	4 to 6
t_1 = JmpTbl_BaseAddr	1
t_1 += $4{*}t_0$	1
jmp *t1	3 to 5
	$\sum$: 10 to 14

Jump Tables: Space Costs

- A jump table with max. and min. case labels c_{max} and c_{min} needs ≈ $c_{max} - c_{min}$ entries.

  This can be wasteful if the entries aren't "dense enough", e.g.:

```
switch (x) {
  case 1: …
  case 1000: …
  case 1000000: …
}
```

- Define the ***density*** of a set of case labels as

  density = no. of case labels / ($c_{max} - c_{min}$)

- Compilers will not generate a jump table if density below some threshold (typically, 0.5).

Switch Statements: Overall Algorithm

- if no. of case labels is small (≤ ~ 8), use linear or binary search.
  - use no. of case labels to decide between the two.
- if density ≥ threshold (~ 0.5) :
  - generate a jump table;

  else :
  - divide the set of case labels into sub-ranges s.t. each sub-range has density ≥ threshold;
  - generate code to use binary search to choose amongst the sub-ranges;
  - handle each sub-range recursively.

Function Calls

- *Caller*:
  - evaluate actual parameters, place them where the callee expects them:
    - param x, k /* x is the k^{th} actual parameter of the call */
  - save appropriate machine state (e.g., return address) and transfer control to the callee:
    - call p
- *Callee*:
  - allocate space for activation record, save callee-saved registers as needed, update stack/frame pointers:
    - enter p

Function Returns

- *Callee*:
  - restore callee-saved registers; place return value (if any) where caller can find it; update stack/frame pointers:
    - retval x;
    - leave p
  - transfer control back to caller:
    - return
- *Caller*:
  - save value returned by callee (if any) into x:
    - retrieve x

Function Call/Return: Example

- *Source*: x = f(0, y+1) + 1;
- *Intermediate Code: Caller*:

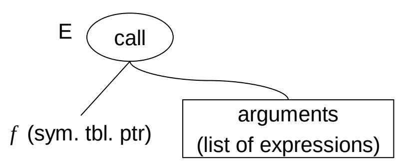
```
t1 = y+1
param t1, 2
param 0, 1
call f
retrieve t2
x = t2+1
```

- *Intermediate Code: Callee*:

```
enter f        /* set up activation record */
…              /* code for f's body */
retval t27     /* return the value of t27 */
leave f        /* clean up activation record */
return
```


Intermediate Code for Function Calls

- non-void return type:



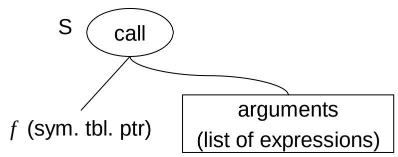
Code Structure:

```
… evaluate actuals …
param x_k  }
…          }  R-to-L
param x_1  }
call f
retrieve t0  /* t0 a temporary var */
```

codeGen_expr(E):
/* E.nodetype = FUNCALL */
  codeGen_expr_list(arguments);
  E.place = newtemp(f.returnType);
  E.code = …code to evaluate the arguments…
    ⊕ param x_k
    …
    ⊕ param x_1
    ⊕ call f, k
    ⊕ retrieve E.place;

Intermediate Code for Function Calls

- void return type:



codeGen_stmt(S):
/* S.nodetype = FUNCALL */
codeGen_expr_list(arguments);
~~E.place = newtemp(f.returnType);~~
S.code = …code to evaluate the arguments…
⊕ param x_k
…
⊕ param x_1
⊕ call *f*, k
~~⊕ retrieve E.place;~~

Code Structure:

… evaluate actuals …
param x_k } R-to-L
…
param x_1
call *f*
~~retrieve t0 /* t0 a temporary var */~~

void return type ⇒ f has no return value ⇒ no need to allocate space for one, or to retrieve any return value.

Reusing Temporaries

Storage usage can be reduced considerably by reusing space for temporaries:

- For each type T, keep a "free list" of temporaries of type T;
- *newtemp*(T) first checks the appropriate free list to see if it can reuse any temps; allocates new storage if not.
- putting temps on the free list:
  - distinguish between user variables (not freed) and compiler-generated temps (freed);
  - free a temp after the point of its last use (i.e., when its value is no longer needed).