Lecture 7

A Toy code generator – 1/4

- Issues in code generation
  - Determination of evaluation order for operators in expression
  - Selection of instructions to be used in target code
  - Use of registers and handling of partial results

- Evaluation of operators depends on operator precedence

A Toy code generator – 2/4

- Choice of instruction to be used in target code depends on
  - *Type* and *length* of each operand
  - *Addressability* of each operand, i.e. where and how operand is located and accessed

A Toy code generator – 3/4

- **Operand descriptor is introduced to maintain the *type, length* and *addressability* info for each operand**

- Choice of instruction can be made by analysing an operator and descriptor of its operands

- Partial result is the value of some subexpression computed while evaluating an expression

A Toy code generator – 4/4

- **Partial results are maintained in CPU registers as far as possible**

- Important issue in code generation is when and how to move partial results between memory and CPU registers, and how to know which partial result is contained in register

- **Use a register descriptor to maintain information for this**

Operand descriptors – 1/2

- Fields
  - **Attributes** : contains type, length and miscellaneous info
  - **Addressability** : specifies where operand is located, and how it can be accessed
  - Two subfields
    - **Addressability code** : takes values ‘M’ (operand is in memory) and ‘R’ (operand is in register). Other codes ‘AR’ (address in register) and ‘AM’ (address in memory)
    - **Address** : address of COU register or memory word

Operand descriptors – 2/2

- **Operand descriptor is built for every operand participating in expression i.e. for id's, constants and partial results**
- Descriptor is built for id when id is reduced during parsing
- Partial result pr_i is result of evaluating some operator op_j
- Descriptor is built immediately after code is generated for operator op_j
- Assume all operand descriptors are stored in array called ***Operand_descriptor***
- Can designate a descriptor by its index in *Operand_descriptor*, eg. Descriptor #n is descriptor in ***Operand_descriptor[n]***

Example

- The code generated for the expression a*b is

```
MOVER    AREG, A
MULT     AREG, B
```

- Three operand descriptors are used. Assuming a, b to occupy 1 memory word

1	(int, 1)	M, addr(a)	Descriptor for a
2	(int, 1)	M, addr(b)	Descriptor for b
3	(nt, 1)	R, addr(AREG)	Descriptor for a*b

Register descriptors

- Two fields
  - **Status** : contains the code free or occupied to indicate register status
  - **Operand descriptor #** : if status=occupied, this field contains descriptor # for the operand contained in register

- Register descriptors are stored in array called ***Register_descriptor***
- **<u>One descriptor exists for each CPU register</u>**

Example

- The register descriptor for AREG after generating code for a*b would be

Occupied	#3

- This indicates that register AREG contains the operand described by descriptor #3

Saving partial results – 1/2

- If all registers are occupied (i.e. contain partial results) when operator op_i is to be evaluated, a register r is freed by copying its contents into a temporary location in memory
- r is now used to evaluate operator op_i
- Assume array *temp* is declared in target program to hold partial results

Saving partial results – 2/2

- Partial result is always stored in the next free entry of *temp*
- **When partial result is moved to temporary location, descriptor of partial result must change**
- Operand descriptor # field of the register descriptor is used to achieve this
- General rule is that partial result saved last is always used before the results saved earlier
- use LIFO data structure to organise temporary locations

Example

- Consider the expression a*b+c*d.
- After the partial result a*b is moved to a temporary location, say temp[1], the operand descriptors must become as shown:

1	(int, 1)	M, addr(a)
2	(int, 1)	M, addr(b)
3	(int, 1)	M, addr(temp[1])

- This indicates value of the operand described by operand descriptors #3 has been moved to memory location temp[1]

Example (cont…)

- Decision has to be made whether one of them exists or not in a register
- If so, generate a single instruction to perform the operation
- If none of the operands is in a register, need to move one operand into the register before performing the operation
- Hence, first free the register if it is occupied and changes the operand descriptor of the result contained previously

Triples – 1/2

- A triple is a representation of an elementary operation in the form of a psuedo-machine instruction

- Each operand of a triple is either a **variable/constant or result** of some evaluation represented by another triple

- If triple is a result, operand field contains **triple's no**

Triples – 2/2

- A **triple** represents an elementary in format
  Operator Operand1 Operand2

- **A table is built to contain all distinct triples in program**

- **Useful to detect identical expressions in a program**

Example

- Triples table for string a+b*c+d
- Postfix notation is abc*+ d+

	Operator	***Operand 1***	***Operand 2***
1	*	b	c
2	+	a	[1]
3	+	[2]	d

Quadruples

- A **quadruple** represents an elementary in format

  Operator Operand1 Operand2 ResultName

- Result name designates result of evaluation

- Can be used as operand of another quadruple

- **More convenient than using a number to designate subexpression**

Example

- Quadruples table for string a+b*c+d
- Postfix notation is abc*+ d+

	Operator	*Operand 1*	*Operand 2*	*Result name*
1	*	b	c	t1
2	+	a	t1	t2
3	+	t2	d	t3

COMPILATION OF CONTROL STRUCTURES

- Control structure is collection of language features which govern the sequencing of control through a program

- Consists of constructs for **control transfer, conditional execution, iteration control and procedure calls**

Control transfer, conditional execution and iterative constructs – 1/2

- Control transfers implemented through **conditional and unconditional goto's** are the **most primitive** control structure

- When target language is assembly language, a statement **goto lab** can be simply compiled as the assembly statement **BC ANY, LAB**

- Control structures like **if, for or while** cause significant semantic gap between PL domain and execution domain because control transfers are **implicit rather than explicit**

Control transfer, conditional execution and iterative constructs – 2/2

- Semantic gap is bridged in two steps
  - Control structure is mapped into equivalent program containing explicit goto's. Since destination of goto may not have a label in source program, compiler generates its own labels and puts them against appropriate statements
  - These programs are translated into assembly programs

- First step need not be carried explicitly, it could be implied in the compilation action

Function and Procedure calls – 1/4

- A function call, viz. the call on fn_1 in statement

  x := fn (y,z) + b*c

- Executes the body of function and returns its value to calling program
- Function call may also result in some side effects

- **A side effect of a function (procedure) call is a change in the value of a variable which is not local to the called function (procedure)**

Function and Procedure calls – 2/4

- While implementing a function call, compiler must ensure
  - Actual parameters are accessible in called function
  - Called function is able to produce side effects according to rules of PL
  - Control is transferred to, and is returned from, called function
  - Function value is returned to calling pgm
  - All other aspects of execution of calling pgms are unaffected by function call

Function and Procedure calls – 3/4

- Compiler uses a set of features to implement function calls
  - **Parameter list :** contains a descriptor for each actual parameter of function call. Notation D_p is used to represent descriptor corresponding to formal parameter p
  - **Save area :** called function saves contents of CPU registers in this area before beginning its execution. The register contents are restored from this area before returning from the function

Function and Procedure calls – 4/4

- **Calling conventions** : these are execution time assumptions shared by the called function and its caller(s). Conventions include
  - How parameter list is accessed
  - How save area is accessed
  - How transfers of control at call and return are implemented
  - How function value is returned to calling pgm

Calling conventions – 1/3

- Calling conventions require the addresses of the function, parameter list and save area to be contained in specific CPU registers at the time of call
- Let r_{par_list} denote register containing address of the parameter list
- Let $(d_{Dp})_{par\_list}$ denote displacement of Dp in parameter list

Calling conventions – 2/3

- $(d_{Dp})_{par\_list}$ is computed while processing formal parameter list of the function and is stored in the symbol table entry of p
- During execution, Dp has address $<r_{par\_list}> + (d_{Dp})_{par\_list}$
- Every reference to p in the function body is compiled using this address
- At return, function value may be returned in CPU register or in memory location

Calling conventions – 3/3

- In dynamic memory allocation, calling pgm constructs the parameter list and the save area on stack
- Become part of the called function's AR when its execution is initiated
- Start of the parameter list has a known displacement d_{par_list} in AR
- For every formal parameter p, displacement of Dp in AR, denoted as $(d_{Dp})_{AR}$ is computed and stored in symbol table entry of p
- During execution, Dp has the address <ARB> + $(d_{Dp})_{AR}$

Passing parameter mechanisms

- Call by value
- Call by value-result
- Call by reference
- Call by name

Call by value – 1/2

- Values of actual parameters are passed to called function
- Values are assigned to corresponding formal parameters
- Passing of values takes place only in one direction – from calling program to called function
- If function changes value of formal parameter, change is not reflected on corresponding actual parameter

Call by value – 2/2

- Function cannot produce any side effect on its parameters
- Commonly used for built-in functions of lang
- Advantage is simplicity
- A called function may allocate memory to a formal parameter and copy value of actual parameter into this location at every call
- Formal parameter appears like an initialised local var, hence compiler can treat it as if it were local var
- Mechanism is very efficient if parameters are scalar vars

Call by value-result

- Copy values of formal parameters back into corresponding actual parameters at return
- Side effects are realized at return
- Inherits simplicity of call by value but incurs high overhead

Call by reference – 1/2

- Address of actual parameter is passed to called function
- If parameter is expression, value is computed and stored in temporary location
- Address of temporary location is passed to called function
- If parameter is array element, its address is compute at the time of call
- Parameter list is thus a list of addresses of actual parameters

Call by reference – 2/2

- Code generated to access the value of formal parameter p is
  1. r ← <ARB> + $(d_{Dp})_{AR}$ or,
     r ←<r_{par_list}>+ $(d_{Dp})_{par\_list}$
  2. Access the value using the address contained in register r
- Each access to parameter in fn body incurs overhead of step 1
- Step 2 produces instantaneous side effects if address in r is used on LHS of assignment
- Popular because it has 'cleaner' semantics than call by value-result

Call by name – 1/2

- Same effect as if every occurrence of a formal parameter in the body of the called function is replaced by the name of corresponding actual parameter
- Implemented as follows
  - Each parameter descriptor in parameter list is the address of a routine which computes the address of corresponding actual parameter

Call by name – 2/2

- Code generated for every access of formal parameter p in the function body is
  - r ←<ARB> + $(d_{Dp})_{AR}$ or
    r ← <r_{par_list}> + $(d_{Dp})_{par\_list}$
  - Call the fn whose address is contained in r
  - Use the address returned by the fn to access p
- Actual parameter corresponding to a formal parameter can change dynamically during execution of function
- Powerful , High overheads
- Less attractive in practice

Call by value, value-result, reference

```
begin
integer n;
procedure p(k: integer);
     begin
     n := n+1;
     k := k+4;
     print(n);
     end;
n := 0;
p(n);
print(n);
end;
```

Output

```
call by value:           1 1
call by value-result:    1 4
call by reference:       5 5
```

Call by value and call by name

```
begin
integer n;
procedure p(k: integer);
      begin
      print(k);
      n := n+1;
      print(k);
      end;
n := 0;
p(n+10);
end;
```

Output

```
call by value:      10  10
call by name:       10  11
```

Call by value and call by name (with evaluation errors)

```
begin
integer n;
procedure p(k: integer);
     begin
     print(n);
     end;
n := 5;
p(n/0);
end;
```

Output:

```
call by value:     divide by zero error
call by name:      5
```

Non-local references

```
procedure clam(n: integer);
begin

  procedure squid;
  begin
    print("in procedure squid -- n="); print(n);
  end;

  if n<10 then clam(n+1) else squid;

end;

clam(1);
```

Output:

```
in procedure squid -- n=10
```

CODE OPTIMIZATION

Introduction – 1/5

- **Aims at improving execution efficiency**

- Achieved in two ways
  1. Redundancies in program are eliminated
  2. Computations in program are rearranged or rewritten to make it execute efficiently

- **Code optimization must not change the meaning of the program**

Introduction – 2/5

- Concerns with scope of optimization
  - Optimization seeks to improve a program rather than algorithm used in program
  - Efficient code generation for specific target machine is beyond its scope
- Optimization techniques are independent of PL and target machine

Introduction – 3/5

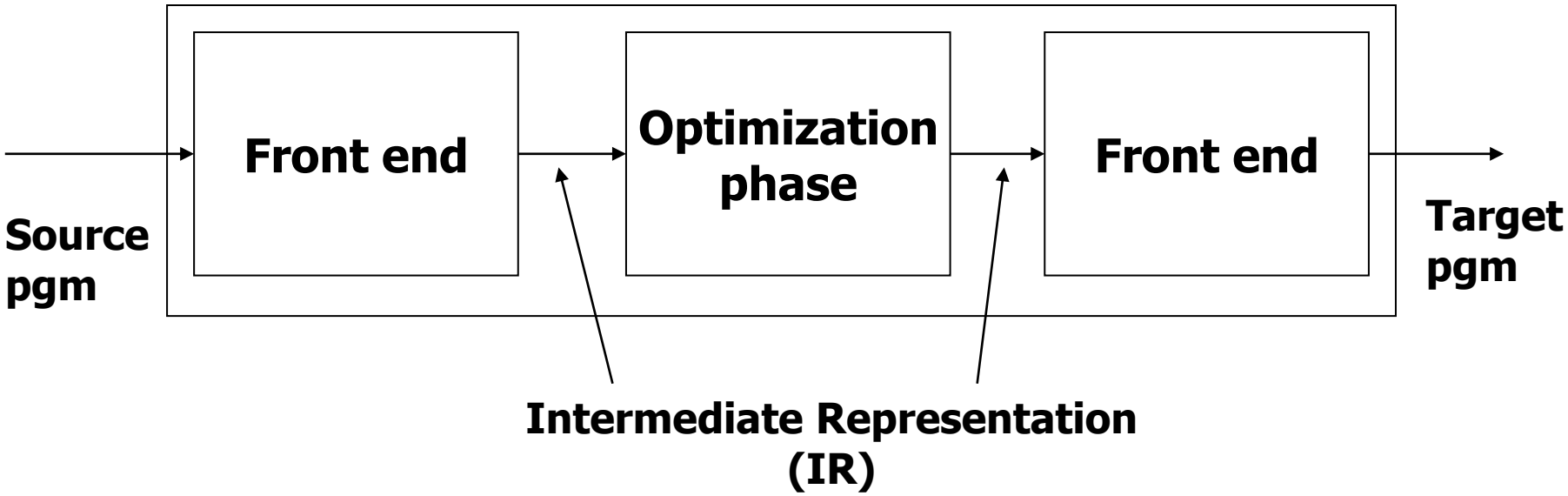

Fig: Schematic of an optimizing compiler

Introduction – 4/5

- Front end generates IR
- Transformed IR is input to the back end
- Compiler analyze a program to collect info (structure of program , manipulation manner)
- Cost and benefits of optimization depend on how a program is analyzed

Introduction – 5/5

- Some of the optimizing transformations are
  - Compile time evaluation
  - Elimination of common subexpressions
  - Dead code elimination
  - Frequency reduction
  - Strength reduction

Compile time evaluation – 1/2

- Execution efficiency can be improved by performing certain actions specified in program during compilation itself
- **Eliminates need to perform them during execution of program , reducing execution time of program**
- Constant folding is main optimization

Compile time evaluation – 2/2

- When all operands in an operation are constants, operation can be performed at compilation time

- Result of operation can (constant) can replace original evaluation in pgm
  - Eg. A : = 12/2 can be replaced by a := 6 eliminating division operation

Elimination of common subexpressions

- **Occurrences of expressions yielding the same value**
- They are called **equivalent expressions**

Example:

```
int A, B=10, C=20, D, E=5;
A = B + C;
D = B + C + E;
```

Dead code elimination

- **Code which can be omitted from a program w/o affecting its results is called a dead code**

- It is detected by checking whether a value assigned in an assignment statement is used anywhere in program

Frequency reduction

- Execution time can be reduced by moving code from a part of a program which is executed very frequently to another part of program which is executed fewer times

Eg. Transformation of loop optimization moves loop invariant code out of loop and places it prior to loop entry

Strength reduction

- Replaces occurrences of a time consuming operation by occurrence of a faster operation

  Eg. Replacement of multiplication by addition

- Very important for array access occurring within a program loop

  Eg. Array reference a[i,j] gave rise to computation i*n, n is no of rows

Extra topics

- Local and global optimization (page – 203)
- Interpreters (page – 212)