

COSC 4P75

Code Generation

- considerations
 - target machine
 - run-time support
 - from O/S
 - from support code
 - from libraries
 - memory model
 - support
 - access code
 - code for each syntactic unit

Compiler Construction 8.1

COSC 4P75

Run-time Support

- support code for program execution including:
 - storage management
 - local variables
 - objects
 - garbage collection
 - O/S interface
 - I/O
 - process management
 - standard libraries
 - e.g. mathematical routines
 - real and pseudo
- either linked in by linker or generated in-line by compiler

Compiler Construction 8.2

COSC 4P75

Storage Management & Environment Handling

- storage management
 - automatic allocation and deallocation of storage for local variables and parameters
 - allocation and deallocation routines for objects
- environment handling
 - model/code for accessing local variables and parameters
 - locals and parameters allocated as a unit called activation record (AR)
 - ARs pushed and popped from stack (FIFO)
 - accessibility based on scope rules (static nesting of blocks) not allocation (execution/stack) order
 - retain static nesting structure by a static chain (pointer from AR to the AR of the block in which it is nested)

Compiler Construction 8.3

COSC 4P75

Activation Record

- local variables
 - storage for each local variable in AR
- parameters
 - accessible within method even though may be in a block not included in the scope of the method
 - must be in AR of called method (but must be filled in from the calling block)
 - parameter transmission modes
- temporaries
 - partial computations, actual parameter values, function return results, etc.
 - only needed temporarily (i.e. not variables)
 - place on stack on top of current AR

Compiler Construction 8.4

COSC 4P75

- context information
 - restore context of calling block at return
 - context includes:
 - register values
 - execution address
 - environment information (e.g static chain)
 - save as part of AR

Compiler Construction 8.5

COSC 4P75

Activation Record Structure

- parameters
 - actual parameters for call
 - pushed onto stack by calling block (i.e. as temporaries)
 - popped from stack at return
 - variable size (but fixed for any particular method)
 - may be empty
- context
 - context information for calling block
 - created at call
 - accessed to allow return
 - fixed size

Compiler Construction 8.6

COSC 4P75

- local variables
 - storage for local variables of method
 - allocated at method entry
 - deallocated at return
 - variable size (but fixed for any particular method)
 - may be empty
- temporaries
 - actual parameter values before call, function results after call
 - partial computations at any time
 - pushed and popped dynamically
 - dynamic size
 - may be empty

Compiler Construction 8.8

COSC 4P75

- stack pointer (SP)
 - pointer to top stack word
 - modified at allocation and deallocation
- frame (AR) pointer (FP)
 - pointer to current AR (i.e. for executing method)
 - variable and parameter accessing relative to FP
 - modified at call and return

Compiler Construction 8.9

```

public void pr ( ) {
  int u, v;
  void p ( int a, int b ) {
    int w, x;
    void q ( int c, int d ) {
      int y, z;
      q(y,z);
    }; // q
    q(w,x);
  }; // p
  p(u,v);
}; // pr

```

COSC 4P75

Variable Addressing

- each block is assigned a block level (nesting) number based on the number of blocks in which it is nested (level in the nesting tree)
 - i.e. in example:

block	level
pr	1
p	2
q	3

Compiler Construction 8.11

COSC 4P75

- every variable is assigned a pair (l, o) which is the level number (l of the block in which it is declared) and offset (o from the FP within the AR for the activation of the block)
 - assume first local @FP+2 and last parameter @FP-17 then, in example:

var	(l, o)
u	(1, 2)
v	(1, 3)
a	(2, -18)
b	(2, -17)
w	(2, 2)
x	(2, 3)
c	(3, -18)
d	(3, -17)
y	(3, 2)
z	(3, 3)
 - note that these (l, o) pairs are static and can be determined at compile time

Compiler Construction 8.12

COSC 4P75

Variable Referencing

- for each reference, the difference (d) between the level of the accessing block (d) and the level of the declaring block (l (in the pair (l, o) for the variable accessed), is computed
- this difference is the length of the static chain connecting the current AR (pointed to be FP) to the declaring block's AR
- each reference (to a variable with pair (l, o)) is translated into a reference pair (d, o) where $d = c - l$

Compiler Construction 8.13

COSC 4P75

- accessing code involves following static chain d times and then offsetting o from the resulting position
 - i.e., say $y, x, u, c,$ and b are referenced within c ($c \neq 3$) in example, then (where $s1$ is offset of static link from FP within an AR):

var	(l, o)	(d, o)	accessing code
y	$(3, 2)$	$(0, 2)$	$(FP)+2$
x	$(2, 3)$	$(1, 3)$	$((FP)+s1)+3$
u	$(1, 2)$	$(2, 2)$	$((FP)+s1)+s1)+2$
c	$(3, -18)$	$(0, -18)$	$(FP)-18$
b	$(2, -17)$	$(1, -17)$	$((FP)+s1)-17$

- note that d , and hence the accessing code, is static for any reference and can be determined at compile time

Compiler Construction 8.14

COSC 4P75

Static Chain

- when a new block is entered (called) the static chain must be established along with the setting of other context information
- there are 2 cases
 - block entered (called) is nested within the current (calling) block
 - entered block's static link must point to calling block's AR (i.e. is just a copy of the FP for the calling block)
 - eg. $p \rightarrow q \rightarrow p \rightarrow q$

Compiler Construction 8.16

COSC 4P75

- method called is at the same or lower nesting level than the current (calling) block
 - called block's static link must point to some ancestor of the calling block within the nesting tree
 - for the called block to be accessible to the calling block, they must share at least one ancestor
 - the desired block is the shared ancestor at the lowest level
 - the difference between the level of the calling block and the level of the called block (+1) is the distance up the static chain from the calling block's AR to the ancestor's AR
 - thus: follow static chain $c \rightarrow c_{called}$ levels and copy the static link at offset $s1$
 - eg. $p \rightarrow q \rightarrow p \rightarrow q \rightarrow p$
- note that this is static for any call and can be determined at compile time

Compiler Construction 8.18

COSC 4P75

Object Storage

- storage for instance variables
- not controlled by method call/return
 - extent from creation until freed or no more references
 - cannot allocate on stack (i.e. within an AR)
- allocated as requested (`new`) from heap
 - object record
 - referenced by a pointer (address)
 - reference variables are pointers
 - heap memory management
- `this` pointer
 - when method called, needs to know which object executing the method
 - implicit last parameter (`this`, at offset 16) is reference to object record

Compiler Construction 8.20

COSC 4P75

Object Record

- storage for instance variables
- referenced from stack or another object record
- reference to class record for class information
- instance variables referenced as offsets from beginning of object record
 - each instance variable addressed by offset (*`o`*) within object
 - class reference at offset 0
 - first instance variable at offset 1

Compiler Construction 8.21

COSC 4P75

Garbage Collection

- free object record when no longer accessible
- techniques
 - mark and gather
 - from stack, scan heap marking everything accessible
 - scan heap, freeing everything not marked
 - reference counters
 - each object has count of number of references
 - inc/dec at assignment
 - deallocate when ref count 0
 - handles
 - object references are indirect via a reference table
 - table can have size and ref counts or marks
 - easier heap reorganization

Compiler Construction 8.23

COSC 4P75

Array Storage

- attributes
 - lower bound
 - upper bound
 - element size
- static allocation
 - where attributes known at compile time
 - can allocate storage on stack or in object record
 - array variable is the array
- dynamic allocation
 - some parameters not known at compile time
 - compute size at execution time and allocate storage
 - array variable is address of array (reference)

Compiler Construction 8.24

COSC 4P75

Element Access

- access to element involves addressing array (determinable at compile time) and then address of element relative to the beginning of the array (must be computed at execution time)
 - i.e.
 - $@(a[i]) = @a + (i-lb_a) * eltsize_a$
 - eg.
 - $@(a[3]) = @a + (3-1) * 1$
 $= @a + 2$
 - with dynamic allocation, a is address of array
- $eltsize_a$ is the amount of storage occupied by an element
 - for simple types: the storage size for the type
 - for arrays: size of the array
 - $eltsize_a = (ub_a - lb_a + 1) * eltsize_e$
- array descriptors
 - dynamic allocation - don't know values
 - store values of lb , ub and/or $eltsize$ with array

Compiler Construction 8.25

COSC 4P75

Parameter Transmission

- methods
 - call-by-value
 - call-by-reference
 - call-by-value/result
 - combination of above

Compiler Construction 8.26

COSC 4P75

Call-by-value

- value of actual parameter passed to method
- formal used as local variable within method
- push value of actual onto stack
- access as normal local variable (i.e. offset from FP)
 - $@p = (FP) + o$
- advantages
 - protection of actual
 - fast access
- disadvantages
 - cost of copy
 - can't modify actual
 - space inefficient if size of object > 1 word (eg. aggregates)
- Java uses call-by-value exclusively

Compiler Construction 8.27

COSC 4P75

Call-by-reference

- reference to actual parameter passed to method
- formal used as indirect reference to actual within method
- push address of actual onto stack
- indirect access from local variable (i.e. indirect from offset from FP)
 - @p = ((FP)+o)
- advantages
 - fast transmission (no copy)
 - space efficient for aggregates
 - can modify actual
- disadvantages
 - no protection of actual
 - slower access (indirect)

Compiler Construction 8.28

COSC 4P75

Generating Code

- code generation code embedded in parsing routines at appropriate places
- assume code generation helper method in SyntacticUnit:


```
protected void gen ( String opCode, String... opnds )
```

 - emits label(s) if set
 - emits opcode with 0 or more operands

Compiler Construction 8.29

COSC 4P75

Virtual Machine

- register machine
 - operands in registers
 - result in register
 - general purpose registers (fixed- or floating-point)
 - typically 2 operand instructions
 - i.e.


```
op s2,s1 s1 ← (s1) op (s2)
```
 - e.g.


```
add r2,r1 r1 ← (r1) + (r2)
```
- support for AR stack (i.e. push, pop, SP & FP)
- support for context switch (i.e. cal, rtn)
- support for variable addressing (i.e. var, val, ndx, sel)
- support for standard procedures
 - I/O (e.g. gint, pint)

Compiler Construction 8.30

COSC 4P75

Expressions

- assume helper methods in SyntacticUnit:
 - `String allocReg ()`
 - returns the register number of an unused register
 - `void freeReg (String r);`
 - sets register `r` to unused state
- expression evaluation results in a value in a register so each expression class has a method which returns register containing result:
 - `public String getReg ()`
- after operation, result in one of two operand registers, free up the other

Compiler Construction 8.31

COSC 4P75

E.g. Term

- doesn't allocate register, gets from `Factor`
- `Factor` will have generated its code to put result in specified register
- simply add code for this operation (`*` or `/`)
- special case for first `Factor`
 - i.e., generate code only after have two `Factors`
 - postfix notation
- generate code and free second register
 - check result type to select operation (fixed vs floating)
- pick up operator for next pass

Compiler Construction 8.32

COSC 4P75

Statements

- statements don't generate results
 - modify control flow
 - cause side effects
 - don't allocate register, use registers returned by expressions to modify control flow or cause side effects
 - registers become free after used by statement
- must generate code for syntactic units in the order they appear
- control structures require branch labels
 - in `SyntacticUnit`
 - `protected String newLab ()`
 - `protected void genLab (String lab)`
 - labels must be unique between all classes
 - qualify label with class name

Compiler Construction 8.33

COSC 4P75

If Statement

- for


```

if expr then
  stmt1
else
  stmt2
end;
      
```

 must generate:


```

code for expr
brf r1,l1
code for stmt1
bra l2
l1: code for stmt2
l2: ...
      
```

Compiler Construction 8.34

COSC 4P75

- for


```

if expr then
  stmt1
end;
      
```

 must generate:


```

code for expr
brf r1,l1
code for stmt1
l1: ...
      
```
- code

Compiler Construction 8.35

COSC 4P75

While Statement

- for


```

while expr do
  stmt
end;
      
```

 must generate:


```

l1: code for expr
brf r1,l2
code for stmt
bra l1
l2: :
      
```

Compiler Construction 8.36

COSC 4P75

FOR Statement

- for


```

            for id:=expr1 to expr2 do
            stmt
            end;
            must generate:
                code for id := expr1
                bra l2
            l1: code for id := id + 1
            l2: code for id <= expr2
                brf r1,l3
                code for stmt
                bra l1
            l3: ...
            
```

Compiler Construction 8.37

COSC 4P75

Assignment Statement

- AssignRest passed Name parsed by NameStmt
- Name has getDecl and getQualifier for identifier reference
- Expression has getReg returning reg containing rhs
- for:


```

            lhs := expr;
            must generate:
                code for reference to lhs variable into r1
                [code for array element access]
                code for expr into r2
                sto r2,r1
            
```
- addressing for variable reference
 - in SyntacticUnit


```

                    protected String varReference ( AVariable qualifier,
                    AVariable var )
                    
```
- pass register containing address of reference to VarRest
- free registers
- code

Compiler Construction 8.38

COSC 4P75

Addressability

- storage model
 - locals & parameters on stack
 - instance variables on heap
 - o class pointer
 - o this pointer
- local variables & parameters accessed relative to FP
- instance variables accessed relative to object reference
 - overriding?
 - o still must have superclass variables
 - o referencing handled by scope
- can determine allocations within AR and object record at compile time
 - ∴ can compute offsets at compile time

Compiler Construction 8.39

COSC 4P75

Array Access

- array variables are pointers to array
- arrays dynamic so need descriptor (length)
- storage allocation:
 - 1st word is length, rest is array
- for array element access generate:


```

access code for array variable into r1
val   r1
code for index expression into r2
ndx   r2,-1,eltsize,r1
      
```

Compiler Construction 8.47

COSC 4P75

Storage Allocation

- for parameters, done by calling method onto stack
- for locals done by call instruction
- for arrays and objects, done by Creation
- arrays:
 - allocate 1 extra word for length
 - store length
 - generate:


```

code for length expression into r1
cst   eltsize,r2
mul   r1,r2
sel   1,r2 (+1)
alc   r2,r2
sto   r1,r2 set length
          
```
 - where *eltsize* is size of element type

Compiler Construction 8.48

COSC 4P75

- objects:
 - allocate storage (`getObjectSize` in `Aclass`)
 - store class pointer and call constructor
 - generate:


```

cst    objectsize, r1
alc    r1, rj
cst    classname, r1    r1->class rec
sto    r1, rj          class pointer
push  arguments (ArgumentList)
psh    rj              this pointer
constructor call
          
```

Compiler Construction 8.49

COSC 4P75

Method Call

- required:
 - obtain address of method
 - push parameters onto stack
 - set `this` pointer
 - make call
- for function methods
 - result placed in special register (`rr`) via return instruction
 - used in resulting expression as appropriate register
 - watch `freeReg` since `rr` never allocated
- for standard procedures (I/O) generate code to access variable or value into register
 - instead of method call, generate I/O instruction, e.g.


```

pint   r1
          
```

 - input instructions put result in specified register

Compiler Construction 8.50

COSC 4P75

Class Record (Method Table)

- polymorphism
 - method called depends on object's type, not variable's type
- object record points to class record for object
- class record contains:
 - address of superclass record
 - addresses of methods for class
- method address is offset within class record
- subclass inherits superclass's methods
 - top part of subclass record is duplicate of superclass record
- method overriding
 - new method address replaces superclass's method address in subclass table
- setting method offsets
 - set in `define` in `Classes` much like variables
 - if overriding, use offset from superclass
- e.g. `Classes` (`constructor` and `define`)

Compiler Construction 8.51

COSC 4P75

Method Body

- at entry must allocate storage for locals on stack
 - amount of storage known at compile (local variable declarations)
 - assume


```
int getLocalSize()
```
- in `AMethod` returns total local storage size
- at beginning of body generate:


```
ntr mname, size
```

 - `mname` is method name qualified by class name to make it unique
 - assume


```
String qualifiedLab ( String lab )
```

in `SyntacticUnit` which prepends the class name

Compiler Construction 8.58

COSC 4P75

- at return, functions must return value and parameter list must be freed
 - on `return` statement, expression is return value, result is in register, generate:


```
rtn r1, size
```
 - `size` is the amount of parameter space, assume


```
int getSize()
```
- in `Parameters`
- at end of method body, procedure methods must return and free parameter storage, generate:


```
rtn size
```
- e.g `MethodBody`

Compiler Construction 8.59

COSC 4P75

Class Declaration

- after body, create class record (static, can be defined at compile time)
 - generate:


```
cls cname, supername
```
 - if no superclass, use 0
 - for each method (including constructor), generate an entry:


```
mth mname
```
 - must generate in method offset order
 - must accommodate overriding
 - size of table available via `getClassSize()` in `Aclass`
 - first entry is superclass, just ignore it
 - assume


```
fillMethodTable (String table[])
```
- `Classes` (`fillMethodTable`)
- `ClassDecl` dumps method table as `mth` entries (starting from element 1 (constructor) since 0 is superclass pointer)

Compiler Construction 8.60

COSC 4P75

Miscellaneous

- if class is main class (Main), generate:
 - end *size,classname,constrname*
 - *size* is object record size, *classname* is the name on the `cls` directive (e.g. `Main`) and *constrname* is qualified name of constructor (e.g. `Main_create`)
- length attribute of arrays
 - find in `Arrays` returns `AVariable` with `offset 0` and `isInstance true` for length
- string pool
 - string constants added to string pool
 - array of string values for compilation unit
 - at end of class declaration, dump string pool using:


```
str    strname, strliteral
```

 - where *strname* is a generated label and *strliteral* is the literal
 - references to string generate


```
cst    strname, ri
```

Compiler Construction © 8.61
