Lecture 6

ASPECTS OF COMPILATION – 1/2

- **Compiler bridges the gap between PL domain and execution domain**

- Two aspects of compilation are
  1. **Generate code to implement meaning** of source program in the execution domain
  2. **Provide diagnostics for violations of PL** semantics in a source program

ASPECTS OF COMPILATION – 2/2

- PL features that contribute to the semantic gap between a PL domain and execution domain
  - Data types
  - Data structures
  - Scope rules
  - Control structure

Data types – 1/3

- It is the specification of
  - (i) **legal values** for variables of type, and
  - (ii) **legal operations** on legal values of the type

- Legal operations of a type include an assignment operation and a set of data manipulation operations

Data types – 2/2

- Tasks to ensure this
  - **Checking legality of an operation** for the types of its operands. This ensures that a variable is subjected only to legal operations of its type
  - **Use type conversion** operations to convert values of one type to values of another type
  - Use **appropriate instruction** sequences of the target machine to implement the operations of a type

Data structures

- Arrays, stacks, records, lists
- To compile a reference to an element of a data structure, compiler must develop a **memory mapping** to access memory word(s) allocated to the element
- A record, which is heterogeneous data structure, leads to complex memory mappings
- User defined type requires mappings of different kind – those that map values into their representations in a computer and vice versa

Scope rules – 1/2

- **Determine accessibility of variables declared in different blocks of a program**
- Scope of a program entity is that part of program where entity is accessible
- Extends to an enclosed block unless the block declares a variable with identical name

Scope rules – 2/2

- Compiler performs operations
  - Scope analysis
  - Name resolution

- Determine data item designated by the use of a name in source program

- Generated code simply implements results of the analysis

Control structure

- Collection of language features for altering flow of control during execution of a program
- Conditional transfer of control, conditional execution, iteration control and procedure calls

MEMORY ALLOCATION

- Three important tasks
  - **Determine amount of memory required** to represent value of data item
  - Use appropriate **memory allocation model** to implement the lifetimes and scopes of data items
  - Determine appropriate **memory mappings** to access values in non scalar data item. Eg. values in array

- First task is implemented during semantic analysis of data declaration statements

Static & Dynamic memory allocation

- Memory binding - Association between the memory address' attribute of a data item and the address of a memory area

- **Binding ceases to exist when memory is deallocated**

- Two types of memory allocation
  – Static memory allocation
  – Dynamic memory allocation

Static memory allocation

- **Memory is allocated to a variable before execution of program begins**
- Performed during compilation
- No memory allocations or deallocations are performed during execution of a program
- **Variables remain permanently allocated**
- Allocation to variable exists even if program unit in which it is defined is not active

Dynamic memory allocation

- Memory bindings are established and destroyed during execution of a program

- Two types
  - **<u>Automatic allocation</u>** – memory binding performed at **execution init time** of program unit
  - **<u>Program controlled allocation</u>** - memory binding performed during **execution of program unit**

Automatic dynamic allocation

- Memory is allocated to variables declared in the program unit when the program unit is entered during execution and is deallocated when the program unit is exited
- **Same memory area may be used for variables of different program units**
- Also possible that different memory areas may be allocated to same variable in different activations of program unit

Program controlled dynamic allocation

- Program can allocate or deallocate memory at arbitrary points during its execution

Comparison – dynamic allocation – 1/2

- In both, address of memory area allocated to a program unit <u>cannot be determined at compilation time</u>
- Implemented using <u>stacks and heaps</u>
- <u>Slower in execution</u> than static memory allocation
- <u>Automatic dynamic allocation is implemented using stack</u> since entry and exit from program units is <u>LIFO</u> in nature
- <u>Program controlled dynamic allocation is implemented using heap</u>

Advantages – Dynamic allocation

- Dynamic allocation provides advantages
  - Recursion implemented easily
  - Allocation of separate memory area for each recursive activation
  - Support data structures whose sizes are determined dynamically

Memory allocation in block structured languages

- A block is a program unit which can contain data declarations

- Program in a block structured language of blocks

- Uses dynamic memory allocation

Scope rules – 1/4

- Data declaration using a **name** ***$name_i$*** creates a **variable** ***var_i*** and establishes a binding between *$name_i$* and *var_i*
- Represent this binding as ***($name_i$, var_i)***
- Called it the **name-var binding**
- Variable var_i is visible at a place in the program if some binding ($name_i$, var_i) is effective at that place

Scope rules – 2/4

- It is possible for data declarations in many blocks of program to use a same name, say $name_i$
- This establish many bindings of the form ($name_i$, var_k) for different values of k
- <u>Scope rules determine which of these bindings is effective at a specific place in program</u>

Scope rules – 3/4

- If variable var_i is created with the name $name_i$ in a block b
  - var_i can be accessed in any statement situated in block b
  - var_i can be accessed in any statement situated in block b' which is enclosed in b, unless b' contains a declaration using same name
- Variable declared in block b is called ***local variable*** of block b
- Variable of an enclosing block that is accessible within block b is called **nonlocal variable** of block b

Scope rules – 4/4

- To differentiate between variables created using same name in different blocks use notation
  - **name**$_{\mathbf{block\_name}}$: variable created by data declaration using name *name* in block *block_name*

Memory allocation and access - 1/2

- Automatic dynamic allocation is implemented using extended stack model
- Minor variation – each record in stack has **two reserved pointers** instead of one

- Each stack record accommodates variables for one activation of a block
- Call it **activation record (AR)**

Memory allocation and access - 2/2

- During execution of a block structured program, a register called **activation record base (ARB**) always points to start address of TOS record
- Record belongs to block which contains statement being executed

- **Local variable x of this block is accessed using the address $\mathbf{d_x}$(ARB)**, where d_x is displacement of variable x from start of AR
- Address may also be written **as <ARB> + $\mathbf{d_x}$**, where <ARB> stands for words ‘contents of ARB’

Dynamic pointer

- First reserved pointer in a block’s AR
- Has the address **0(ARB)**
- **It is used for deallocating an AR**

Accessing nonlocal variables – 1/3

- A nonlocal variable *nl_var* of a block b *b_use* is a local variable of some block *b_defn* enclosing *b_use*

- A **textual ancestor** or **static ancestor** of block *b_use* is a block which **encloses block *b_use***

- The block immediately enclosing *b_use* is called its **Level 1 ancestor**
- A Level m ancestor is a block which immediately encloses the Level (m-1) ancestor

Accessing nonlocal variables – 2/3

- **The level difference between *b_use* and its Level m ancestor is m**

- If $s\_nest_{b\_use}$ represents the static level of block *b_use* in the program, *b_use* has a Level i ancestor, $\forall i < s\_nest_{b\_use}$

- **When *b_use* is in execution, *b_defn* must be active**

Accessing nonlocal variables – 3/3

- Hence AR_{b_defn} exists in the stack, and *nl_var* is to be accessed as

start address of AR_{b_defn} + d_{nl_var}

where d_{nl_var} is displacement of *nl_var* in Ar_{b_defn}

Static pointer – 1/2

- **Access to nonlocal variable** is implemented by this pointer
- It is the second reserved pointer in AR
- Has the address **1(ARB)**
- When an AR is created for a block b, its static pointer is set to point to the AR of the static ancestor of b

Static pointer – 2/2

- Code to access a nonlocal variable *nl_var* declared in a Level m ancestor of *b_use*, m>=1, is
  - r = ARB; r is some register
  - Repeat step 3 m times
  - r = 1(r); load the static pointer into r
  - Access *nl_var* using address <r> + d_{nl_var}

- Thus a nonlocal variable defined in Level m ancestor is accessed using m indirections through the static pointer

Displays – 1/3

- For large values of level difference, it is expensive to access nonlocal variables using static pointers

- **Display is an array used to improve efficiency of nonlocal accesses**

Displays – 2/3

- When a block B is in execution, the entries of Display contain the information:

Display[1] = address of level (s_nest_b – 1) ancestor of B

Display[2] = address of level (s_nest_b – 2) ancestor of B

…

Display[s_nest_b – 1] = address of level 1 ancestor of B

Display[s_nest_b] = address of AR_B

Displays – 3/3

- Let block B refer to some variable v_j defined in an ancestor block b_i
- The address of v_j is calculated as

  Display [s_nest$_{bi}$] + d$_{vj}$

- The code generated for the access would be

1. r := Display [s_nest$_{bi}$]
2. Access v_j using the address <r> + d_{vj}

Symbol table requirements – 1/6

- For dynamic allocation and access, a compiler should perform the following tasks while compiling the use of a name ***v*** in ***b_current*** (the block being compiled)

1. Determine the static nesting level of *b_current*
2. Determine the variable designated by the name *v* (scope rules)
3. Determine the static nesting level of the block in which *v* is defined (value dv)
4. Generate the access code

Symbol table requirements – 2/6

- For tasks 1,2, and 3, we use the extended stack model to organize the symbol table

- When the start of block *b_current* is encountered during compilation, a new record is pushed on the stack.

- Stack contains
  - Nesting level of *b_current*
  - Symbol table for *b_current*

Symbol table requirements – 3/6

- The reserved pointer of the new record points to the previous record in the stack
- This record contains the symbol table of the static ancestor of b_current
- Each entry in the symbol table contains a variable's name, type, length and displacement in the AR

Symbol table requirements – 4/6

- The scope rules are implemented by searching the name v referenced in b_current in the symbol table
  1. <u>Symbol table in the topmost record of the stack is searched first</u>
  2. Existence of name v implies that v is a local variable of b_current
  3. If an entry for v does not exist there, then the previous in the stack is searched

Symbol table requirements – 5/6

4. It contains the symbol table for the Level 1 ancestor of b_current
5. Existenc of v in it implies that v is a variable declared in the Level 1 ancestor block, and not redeclared in b_current
6. If v is not found there, it is searched in the previous record of the stack, i.e. in the symbol table of Level 2 ancestor, and so on

Symbol table requirements – 6/6

- When v is found in the symbol table, its displacement dv in the AR is then obtained from the first field of the stack record containing the symbol table
- Code can then be generated to implement the access to variable v

Recursion

- It includes many invocations of a procedure during the execution of a program
- A copy of the local variables of the procedure must be allocated for each invocation
- Use the stack model

Extra topics

- Limitations of stack based memory allocation (page – 176)
- Array allocation and access (page – 177)