CC Lecture 10

Prepared for: 7th Sem, CE, DDU

Prepared by: Niyati J. Buch

Compiler Phases

- **Lexical analysis**: processing characters
- **Parsing**: processing the tokens and producing the syntax tree
- **Semantic analysis**: which checks whether the semantics of the programming language are satisfied by the program.
- **Intermediate code generation**
- Before **machine code generation**, **what exactly** is required for a program to execute at the time when it is put into memory and the execution begins?

What is required??

- run time arrangement
  - code generator expects that these are available at run time
- run time support
  - various parameter passing methods
  - different types of storage allocation
  - the format of activation records
  - the difference between static scope and dynamic scope
  - how to pass functions as parameters
  - heap memory management
  - garbage collection.

Run Time Support

- Interfaces between the **program** and the computer system **resources** are needed
- There is a need to **manage memory** when a program is running:-
  - Memory management must connect to the data objects of programs
  - Programs request for memory blocks and release memory blocks
  - Passing parameters to functions
- Other resources such as printers, file systems, etc., also need to be accessed (done by operating system)

Parameter Passing Methods

1. Call by value
2. Call by reference
3. Call by value result
4. Call by name

Call-by-value

- At runtime, prior to the call, the parameter is evaluated, and its actual value is put in a location private to the called procedure
- There is no way to change the actual parameters.
- C has only call-by-value method available
  - Passing pointers does not constitute call-by-reference
  - Pointers are also copied to another location
  - Hence in C, there is no way to write a function to insert a node at the front of a linked list (just after the header) without using pointers to pointers
- Found in C and C++

Call-by-Reference

- At runtime, prior to the call, the parameter is evaluated and put in a temporary location, if it is not a variable.
- The **address of the variable** (or the temporary) is passed to the called procedure.
- Thus, the actual parameter may get changed due to changes to the parameter in the called procedure.
- Found in C++ and Java

Call-by-Value-Result

- Call-by-value-result is a hybrid of Call-by-value and Call-by-reference
- Actual parameter is calculated by the calling procedure and is copied to a local location of the called procedure
- Actual parameter's value is not affected during execution of the called procedure
- At return, the value of the formal parameter is copied to the actual parameter, if the actual parameter is a variable

Call-by-Value-Result

- Becomes different from **call-by-reference** method
  - when global variables are passed as parameters to the called procedure and
  - the same global variables are also updated in another procedure invoked by the called procedure
- Found in Ada

Example: Call-by-value vs. Call-by-reference vs. Call-by-Value-Result

```
int a;
void Q() {
    a = a+1;
}
void R(int x){
    x = x+10;
    Q();
}
main(){
    a = 1;
    R(a);
    print(a);
}
```

Call by value	Call by reference	Call be value result
2	12	11

Note:
In Call-by-V-R, value of x is copied into a, when proc R returns.

Hence, a=11.

Call-by-Name

- Use of a call-by-name parameter implies a textual substitution of the formal parameter name by the actual parameter.
- Hence, we cannot evaluate the address of the actual parameter just once and use it.
- It must be recomputed every time, we reference the formal parameter within the procedure.
- A separate routine (called thunk) is used to evaluate the parameters whenever they are used.
- Found in ALGOL and functional languages

Call-by-Name

- For example, if the procedure

  void R (int X, int I);

  { I = 2; X = 5; I = 3; X = 1; }
- is called by **R(B[J*2], J)**
- this would result in (effectively) changing the body to

  { J = 2; B[J*2] = 5; J = 3; B[J*2] = 1; }
- just before executing it

- the actual parameter corresponding to *X changes whenever J changes*

Comparison

```
1. void swap (int x, int y)
2. { int temp;
3. temp = x;
4. x = y;
5. y = temp;
6. } /*swap*/
7. ...
8. { i = 1;
9. a[i] =10; /* int a[5]; */
10. print(i, a[i]);
11. swap(i, a[i]);
12. print(i, a[1]); }
```

call-by-value		**call-by-reference**		**call-by-value-result**		**call-by-name**	
1	10	1	10	1	10	1	10
1	10	10	1	10	1	Error!	

Reason for the error in the Call-by-name Example

temp = i; /* => temp = 1 */
i = a[i]; /* => i =10 since a[i] ==10 */
a[i] = temp; /* => a[10] = 1 => index out of bounds */

Code and Data Area in Memory

- Most programming languages distinguish between code and data

- Code consists of only machine instructions and normally does not have embedded data
- Code area normally does not grow or shrink in size as execution proceeds
- Unless code is loaded dynamically or code is produced dynamically (e.g. dynamic loading of classes)
- Memory area can be allocated to code statically

- Data area of a program may grow or shrink in size during execution

Static vs. Dynamic Allocation

- **Static allocation**
  - Compiler makes the decision regarding storage allocation by looking only at the program text
- **Dynamic allocation**
  - Storage allocation decisions are made only while the program is running
  - **Stack allocation**
    - Names local to a procedure are allocated space on a stack
  - **Heap allocation**
    - Used for data that may live even after a procedure call returns
    - Ex: dynamic data structures such as symbol tables
    - Requires memory manager with garbage collection

Static Storage Allocation

- In a static storage-allocation strategy, it is necessary to be able to decide at compile time exactly where each data object will reside at run time.

- In order to make such a decision, at least two criteria must be met:
  1. The size of each object must be known at compile time.
  2. Only one occurrence of each object is allowable at a given moment during program execution.

Due to these criteria, the following are **not allowed** for static allocation strategy:

- **Criterion One:**
  - Variable length strings (length cannot be determined at compile time)
  - Dynamic arrays (bounds and hence the size of data object unknown at compile time)

- **Criterion Two**:
  - Nested procedures
  - Recursive procedures

    (as which and how many times the procedure will be called is unknown at compile time)

FORTRAN

- FORTRAN typifies those languages in which a static storage-allocation policy is sufficient to handle the storage requirements of the data objects in a program.

- Because FORTRAN does not provide
  - variable-length strings
  - dynamic arrays
  - nested procedures

- Ex: FORTRAN IV and FORTRAN 77

Static storage-allocation strategy

- Very simple to implement
- During an initial pass of the source text, a symbol-table entry is created for each variable and the set of attributes
- Because the precise amount of space required by each variable is known at compile time, the object address for a variable can be assigned according to the following simple scheme.
  - The first variable is assigned some address A near the beginning of an allocated data area,
  - The second variable is assigned address A + n1 assuming the first variable requires n1 storage units (e.g., bytes),
  - The third variable is assigned address A + n1 + n2 assuming the second variable requires n2 storage units, and so on.

Part of a symbol table that would be created for the given FORTRAN program segment assuming integer values require four storage units and real values require eight.

```
REAL MAXPRN, RATE
INTEGER IND1, IND2
REAL PRIN (100), YRINT (5,100), TOTINT
        •
        •
        •
```

(*a*)

Name	Type	Dimension		Address
MAXPRN	R	0		264
RATE	R	0		272
IND1	I	0		280
IND2	I	0		284
PRIN	R	1		288
YRINT	R	2		1088
TOTINT	R	0		5088

Object Address

- **Absolute address**
  - If the compiler is written for a single-job-at-a-time environment
  - The initial address A is set such that the program and data area reside in a section of memory separate from the resident parts of the operating system.
- **Relative address**
  - If the compiler resides in a multiprogramming environment
  - a program and its data area may reside at a different set of memory locations each time the program is executed.
  - The loader reserves a set of memory locations for the program and sets a base register to the address of the first location in the data area.