# Compiler Construction
Run-time environments

A compiler must accurately implement the abstractions embodied in the source-language definition. These abstractions typically include concepts such as names, scopes, data types, operators, procedures, parameters, and flow-of-control constructs. The compiler must cooperate with the operating system and other systems software to support these abstractions on the target machine.

To do so, the compiler creates and manages a run-time environment in which it assumes its target programs are being executed. This environment deals with a variety of issues, such as the layout and allocation of storage locations for the objects named in the source program, the mechanisms used by the target program to access variables, the linkages between procedures, the mechanisms for passing parameters, and the interfaces to the operating system, input/output devices, and other programs.

## 1. Storage Organization and Allocation Strategies

The run-time representation of an object program in the logical address space consists of data and program areas, as shown in Figure 1. A compiler for a language like C++ on an operating system like Linux might subdivide memory in this way.

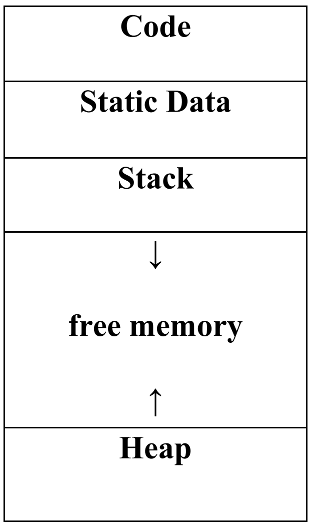

Figure 1. The subdivision of run-time memory into code and data areas

The size of the generated target code is fixed at compile time, so the compiler can place the executable target code in a statically determined area, usually in the low end of memory. Similarly, the size of some program data objects, such as global constants, and data generated by the compiler, such as information to support garbage collection, may be known at compile time, and these data objects can be placed in another statically determined area called Static. One reason for statically allocating as many data objects as possible is that the addresses of these objects can be compiled into the target code. In early versions of Fortran, all data objects could be allocated statically.
To maximize the utilization of space at run time, the other two areas, Stack and Heap, are at the opposite ends of the remainder of the address space. These areas are dynamic; their size can change as the program executes. These areas grow towards each other as needed. The stack is used to store data structures called activation records that get generated during procedure calls.

### 1.1 Static Versus Dynamic Storage Allocation

The layout and allocation of data to memory locations in the run-time environment are key issues in storage management. These issues are tricky because the same name in a program text can refer to multiple locations at run time. The two adjectives static and dynamic distinguish between compile time and run time, respectively. We say that a storage-allocation decision is static, if it can be made by the compiler looking only at the text of the program, not at what the program does when it executes. Conversely, a decision is dynamic if it can be decided only while the program is running. Many compilers use some combination of the following two strategies for dynamic storage allocation:

1. ***Stack storage:*** Names local to a procedure are allocated space on a stack. The stack supports the normal call/return policy for procedures.

2. ***Heap storage:*** Data that may outlive the call to the procedure that created it is usually allocated on a "heap" of reusable storage. The heap is an area of virtual memory that allows objects or other data elements to obtain storage when they are created and to return that storage when they are invalidated.

### 1.1.1.Stack Allocation of Space

Almost all compilers for languages that use procedures, functions, or methods as units of user-defined actions manage at least part of their run-time memory as a stack. Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.

**Activation Trees:** We can represent the activations of procedures during the running of an entire program by a tree, called an activation tree. Each node corresponds to one activation, and the root is the activation of the "main" procedure that initiates execution of the program.

**Activation Records:** Procedure calls and returns are usually managed by a run-time stack called the control stack. Each live activation has an activation record on the control stack, with the root of the activation tree at the bottom, and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides. The latter activation has its record at the top of the stack.

The contents of activation records vary with the language being implemented. Here is a list of the kinds of data that might appear in an activation record (see Figure 2 for a summary and possible order for these elements):

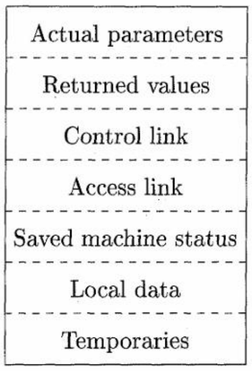

Figure 2. A general activation record

1. Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.

2. Local data belonging to the procedure whose activation record this is.
3. A saved machine status, with information about the state of the machine just before the call to the procedure. This information typically includes the return address (value of the program counter, to which the called procedure must return) and the contents of registers that were used by the calling procedure and that must be restored when the return occurs.
4. An "access link" may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record.
5. A control link, pointing to the activation record of the caller.
6. Space for the return value of the called function, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.
7. The actual parameters used by the calling procedure. Commonly, these values are not placed in the activation record but rather in registers, when possible, for greater efficiency. However, we show a space for them to be completely general.

**Calling sequences:**

- Procedures called are implemented in what is called as calling sequence, which consists of code that allocates an activation record on the stack and enters information into its fields.
- A return sequence is similar to code to restore the state of machine so the calling procedure can continue its execution after the call.
- The code in calling sequence is often divided between the calling procedure (caller) and the procedure it calls (callee).
- When designing calling sequences and the layout of activation records, the following principles are helpful:
    - Values communicated between caller and callee are generally placed at the beginning of the callee’s activation record, so they are as close as possible to the caller’s activation record.
    - Fixed length items are generally placed in the middle. Such items typically include the control link, the access link, and the machine status fields.
    - Items whose size may not be known early enough are placed at the end of the activation record. The most common example is dynamically sized array, where the value of one of the callee’s parameters determines the length of the array.
    - We must locate the top-of-stack pointer judiciously. A common approach is to have it point to the end of fixed-length fields in the activation record. Fixed-length data can then be accessed by fixed offsets, known to the intermediate-code generator, relative to the top-of-stack pointer.

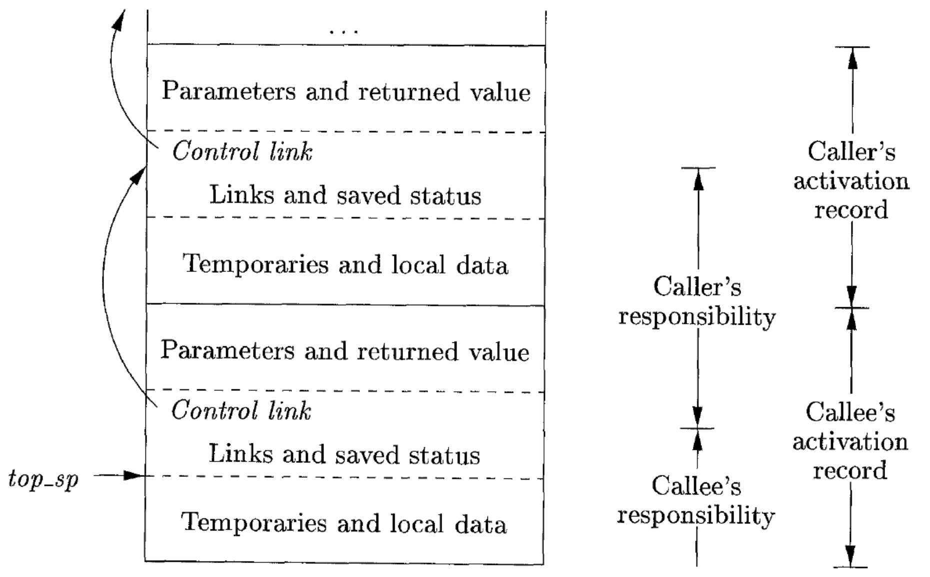
...
Parameters and returned value
Control link
Links and saved status
Temporaries and local data
Parameters and returned value
Control link
Links and saved status
Temporaries and local data
top_sp
Caller's activation record
Caller's responsibility
Callee's activation record
Callee's responsibility

Figure. 3 Division of tasks between caller and callee

- The calling sequence and its division between caller and callee are as follows.
  - The caller evaluates the actual parameters.
  - The caller stores a return address and the old value of top_sp into the callee's activation record. The caller then increments the top_sp to the respective positions.
  - The callee saves the register values and other status information.
  - The callee initializes its local data and begins execution. •
- A suitable, corresponding return sequence is:
  - The callee places the return value next to the parameters.
  - Using the information in the machine-status field, the callee restores top_sp and other registers, and then branches to the return address that the caller placed in the status field.
  - Although top_sp has been decremented, the caller knows where the return value is, relative to the current value of top_sp; the caller therefore may use that value.

**Variable length data on stack:**

- The run-time memory management system must deal frequently with the allocation of space for objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.
- The reason to prefer placing objects on the stack is that we avoid the expense of garbage collecting 7 their space.
- The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.

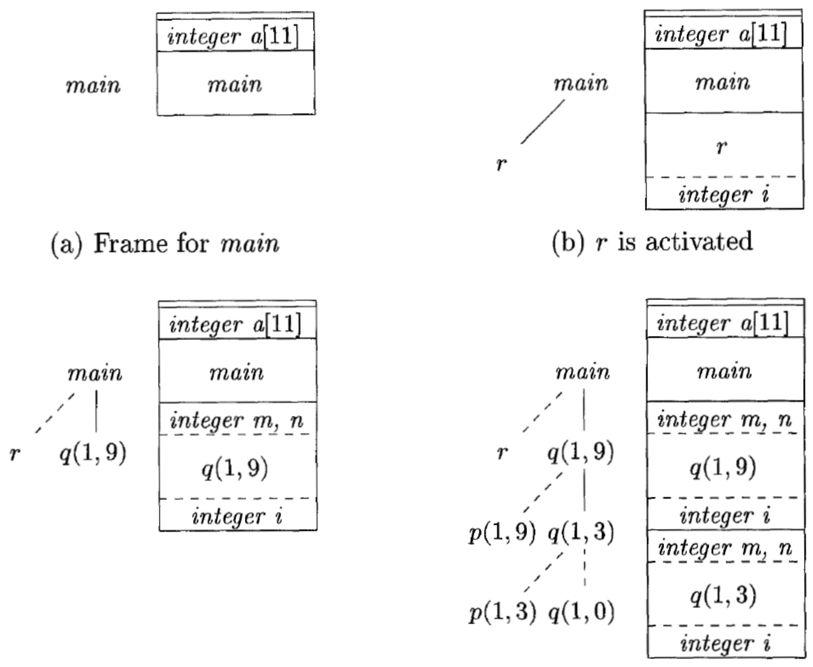

Figure. 4 Downward-growing stack of activation records

**1.1.2.Heap Management**

The heap is the portion of the store that is used for data that lives indefinitely, or until the program explicitly deletes it. While local variables typically become inaccessible when their procedures end, many languages enable us to create objects or other data whose existence is not tied to the procedure activation that creates them. For example, both C++ and Java give the programmer the ability to create objects that may be passed or pointers to them may be passed from procedure to procedure, so they continue to exist long after the procedure that created them is gone. Such objects are stored on a heap.

**The Memory Manager:**

The memory manager keeps track of all the free space in heap storage at all times. It performs two basic functions:

***Allocation:*** When a program requests memory for a variable or object, the memory manager produces a chunk of contiguous heap memory of the requested size. If possible, it satisfies an allocation request using free space in the heap; if no chunk of the needed size is available, it seeks to increase the heap storage space by getting consecutive bytes of virtual memory from the operating system. If space is exhausted, the memory manager passes that information back to the application program.

**Deallocation:** The memory manager returns deallocated space to the pool of free space, so it can reuse the space to satisfy other allocation requests. Memory managers typically do not return memory to the operating system, even if the program's heap usage drops.

Here are the properties we desire of memory managers:

***Space Efficiency:*** A memory manager should minimize the total heap space needed by a program. Doing so allows larger programs to run in a fixed virtual address space. Space efficiency is achieved by minimizing "fragmentation,"

***Program Efficiency:*** A memory manager should make good use of the memory subsystem to allow programs to run faster. The time taken to execute an instruction can vary widely depending on where objects are placed in memory. Fortunately, programs tend to exhibit "locality," a phenomenon that refers to the non-random clustered way in which typical programs access memory. By attention to the placement of objects in memory, the memory manager can make better use of space and, hopefully, make the program run faster.

***Low Overhead****:* Because memory allocations and deallocations are frequent operations in many programs, it is important that these operations be as efficient as possible. That is, we wish to minimize the overhead - the fraction of execution time spent performing allocation and deallocation. Notice that the cost of allocations is dominated by small requests; the overhead of managing large objects is less important, because it usually can be amortized over a larger amount of computation.

## 2. Parameter Passing

All programming languages have a notion of a procedure, but they can differ in how these procedures get their arguments. In this section, we shall consider how the actual parameters (the parameters used in the call of a procedure) are associated with the formal parameters (those used in the procedure definition). Which mechanism is used determines how the calling-sequence code treats parameters. The great majority of languages use either "call-by-value," or "call-by-reference," or both. We shall explain these terms, and another method known as "call-by-name," that is primarily of historical interest.

### 2.1 Call by Value

In call-by-value, the actual parameter is evaluated (if it is an expression) or copied (if it is a variable). The value is placed in the location belonging to the corresponding formal parameter of the called procedure. This method is used in C and Java, and is a common option in C++, as well as in most other languages. Call-by-value has the effect that all computation involving the formal parameters done by the called procedure is local to that procedure, and the actual parameters themselves cannot be changed.
Note, however, that in C we can pass a pointer to a variable to allow that variable to be changed by the callee. Likewise, array names passed as parameters in C, C++, or Java give the called procedure what is in effect a pointer or reference to the array itself. Thus, if a is the name of an array of the calling procedure, and it is passed by value to corresponding formal parameter x, then an assignment such as x[i] = 2 really changes the array element a[2]. The reason is that, although x gets a copy of the value of a, that value is really a pointer to the beginning of the area of the store where the array named a is located.

### 2.2 Call- by-Reference

In call- b y-reference, the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter. Uses of the formal parameter in the code of the callee are implemented by following this pointer to the location indicated by the caller. Changes to the formal parameter thus appear as changes to the actual parameter.
If the actual parameter is an expression, however, then the expression is evaluated before the call, and its value stored in a location of its own. Changes to the formal parameter change this location, but can have no effect on the data of the caller.
Call-by-reference is used for "ref" parameters in C++ and is an option in many other languages. It is almost essential when the formal parameter is a large object, array, or structure. The reason is that strict call-by-value requires that the caller copy the entire actual parameter into the space belonging to the corresponding formal parameter. This copying gets expensive when the parameter is large. As we noted when discussing call-by-value, languages such as Java solve the problem of passing arrays, strings, or other objects by copying only a reference to those

objects. The effect is that Java behaves as if it used call-by-reference for anything other than a basic type such as an integer or real.

**2.3 Call- by-Name**

A third mechanism - call-by-name - was used in the early programming language Algol 60. It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the actual parameter (with renaming of local names in the called procedure, to keep them distinct). When the actual parameter is an expression rather than a variable, some unintuitive behaviors occur, which is one reason this mechanism is not favored today.

## 3. Symbol table organizations and generations

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name. These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.

The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

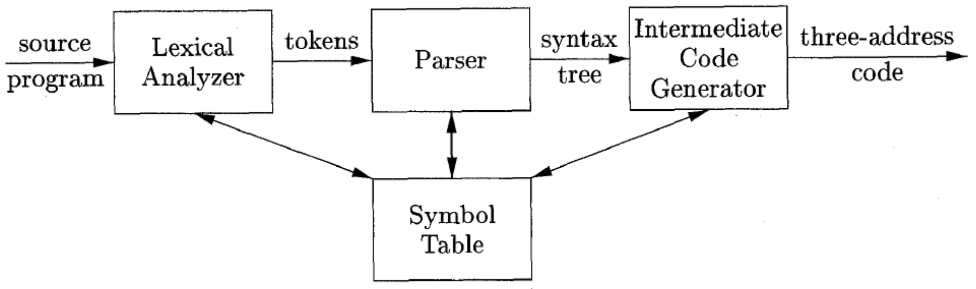

Figure 5. Roll of Symbol Table

Symbol tables are data structures that are used by compilers to hold information about source-program constructs. The information is collected incrementally by the analysis phases of a compiler and used by the synthesis phases to generate the target code. Entries in the symbol table contain information about an identifier such as its character string (or lexeme) , its type, its position in storage, and any other relevant information. Symbol tables typically need to support multiple declarations of the same identifier within a program.

The scope of a declaration is the portion of a program to which the declaration applies. We shall implement scopes by setting up a separate symbol table for each scope. A program block with declarations8 will have its own symbol table with an entry for each declaration in the block. This approach also works for other constructs that set up scopes; for example, a class would have its own table, with an entry for each field and method.

A program consists of blocks with optional declarations and "statements" consisting of single identifiers. Each such statement represents a use of the identifier. Here is a sample program in this language:

```
{ int x; char y; { bool y; x; y; } x; y; }
```

The examples of block dealt with the definitions and uses of names; the input consists solely of definitions and uses of names. The task we shall perform is to print a revised program, in which the declarations have been removed and each "statement" has its identifier followed by a colon and its type.

## 3.1 Data Structures for Symbol Table

Different data structures can be used to implement a symbol table depending on the requirements for efficiency, memory, and ease of operations like insert, search, update, and delete.

- Linear List (Array or Linked List)

Description: Identifiers are stored in a simple list, either as an array or linked list.
Operations:
Search → Linear search (O(n))
Insert → At the end (O(1))
Delete/Update → O(n) in worst case
Advantages: Easy to implement, suitable for small compilers.
Disadvantages: Inefficient for large programs because searching takes longer.

- Hash Table

Description: Uses a hash function to map identifiers (keys) to indices of a table. Collisions are handled using techniques like chaining or open addressing.
Operations:
Search → O(1) average, O(n) worst case (if many collisions)
Insert/Delete → O(1) average
Advantages: Very fast for large symbol tables.
Disadvantages: Requires a good hash function; collision handling increases complexity.

- Binary Search Tree (BST)

Description: Identifiers are stored in a BST, where left child < root < right child (lexicographic order).
Operations:
Search/Insert/Delete → O(h), where h is tree height
For balanced tree → O(log n)
Advantages: Keeps identifiers sorted, supports ordered traversal.
Disadvantages: If unbalanced, worst-case O(n).

- Balanced Binary Trees (AVL Tree, Red-Black Tree)

Description: A self-balancing BST to keep tree height = O(log n).
Operations:
Search/Insert/Delete → O(log n)
Advantages: Guarantees logarithmic performance.
Disadvantages: More complex to implement than normal BST.

- Trie (Prefix Tree)

Description: Identifiers are stored based on their prefixes, each character corresponds to a node.
Operations:
Search → O(m), where m = length of identifier
Insert → O(m)
Advantages: Efficient for string storage and retrieval; avoids repeated prefixes.
Disadvantages: Requires large memory if identifiers are long and sparse.