Chapter 1

Runtime organization

Up to now we have completed the front-end phase of the compiler: lexical analysis, parsing and semantic analysis. These three phases intend to enforce the language definitions. If no errors were generated during the front-end phases, the program proves to be valid in the language, and we are ready to proceed to the backend phases: optimization and code generation. However, before we can talk about the backend phases, we need to talk about runtime organization, which is essential to help us understand what we are trying to generate.

The main topics of this section include run-time resources management, correspondence static(compile-time) and dynamic (run-time)structures, and storage organization.

Exectution of a program is initially under the control of the operating system. When a program is invoked, the OS allocates space for the program, the code is loaded into part of the space, and the OS jumps to the entry point of the program, i.e. the "main" function.

The memory space allocated for a program is not necessarily contiguous. Besides the part of space storing the code, the rest of the space stores data. The compiler is responsible not only for generating the code, but also for orchestrating the data area.

1.1 Activations

We have two goals in code generation: the correctness of the code in the sense that it correctly implements the program intended by the programmer, and the speed of the program. Complications in code generation originates from the need to solve the two problems simutaneously. Over history, an elaborate framework has been developped to ensure that the two goals can be achieved together. Activation is the first topic in our discussion of the framework.

We will assume that the programming languages for which we are trying to generate code satisfy:

1. Execution is sequential. Control moves from one point in a program to

another in a well defined order. This assumption is violated if a language supports concurrency.

2. When a procedure is called, control always returns to the point immediately after the call. This assumption is violated if the language supports advanced control mechanism such as exceptions and call/cc.

An invocation of procedure P is called an **activation** of procedure P. The **lifetime** of an activation of P is all the steps to execute P, including all steps in procedures called by P. Similarly, we can define the **lifetime** of a variable x as the portion of execution in which x is defined. Note that lifetime is a dynamic/runtime concept, while scope is a static concept.

From the definitions and our assumptions, it is clear that when procedure P calls procedure Q, Q must return before P returns. Thus lifetimes of procedure activations are properly nested, which makes them suitable to be depicted as a tree, i.e. the activation tree. The activation tree depends on runtime behavior, and can be different for different inputs. Since activations are nested, we can use a stack to track currently active procedures. The procedure stack comes after the part to store code in the memory. It grows when new procedure is called, and shrinks when the current procedure returns.

The information needed to manage one procedure is called an **activation record (AR)**, or a **frame**. Activation record keeps track of the information needed to properly execute a procedure. If procedure F calls G, the G's activation record contains a mix of information about F and G. In this case, F is suspended until G completes, at which point F resumes. G's AR contains information needed to complete the execution of G, and to resume execution of F. Condiser the following Cool procedure:

```
Class Main {
  g():Int { 1 };
  f(x:Int):Int { if x=0 then g() else f(x-1) fi};
  main():Int { f(3) };
};
```

Main has no argument or local variables, and its result is never used. Thus its AR is not interesting. We will focus on the AR of f. The AR of f contains

- result of f (return value)
- argument
- control link (a pointer to the previous activation, i.e. the caller)
- return address (memory address of the instruction to jump to after f completes, i.e. where execution resumes after a procedure call finishes)

This is just one of many possible AR designs. It would also work for C, Pascal, FORTRAN, etc. The advantage of placing the return value at the 1st position in a frame is that the caller can find it at a fixed offset from its own frame. An AR design is better as long as it improves execution speed

or simplifies code generation. In practice, compilers hold as many frames in registers as possible, especially results and arguments of procedures.

The compiler must determine **at compile time** the layout of activation records and generate code that correctly accesses locations in the activation records. Thus, the AR layout and the code generator must be designed together.

1.2 Globals and heap

All references to a global variable point to the same object, thus they cannot be be stored in an activation record which is deallocated after an activation is completed. Globals are assigned a fixed address once, and we call these variables "statically allocated" because they are allocated during compile time. Depending on the language, there may be other statically allocated values.

Besides globals, a value that outlives the procedure that creates it cannot be kept in the AR either. For example, for the method `foo() {new Bar}`, the Bar value must survive the deallocation of foo's AR. Languages with with dynamically allocated data use a heap to store dynamic data.

Now we can summarize different kinds of data that a language implementation has to deal with.

- The code area contains object code. For many languages it is of fixed size and read-only.
- The static area contains data with fixed addresses, e.g. globals. This area is of fixed size, and can be read-only or writable.
- The stack contains an AR for each currently active procedure. Each AR is usually of fixed size, and contains the locals of the procedure.
- Heap contains all other data. In C, heap is managed by `malloc` and `free`, while in JAVA there is `new` for allocation and garbage collection mechanism takes care of reclaimation of heap space no longer to be used.

Both stack and heap grows. We should make sure that they do not grow into each other. A simple solution is to let them start at opposite ends of the memory and grow towards each other. We end up with the partition of the memory shown in Figure 1.1.

1.3 Alignment

Alignment is a very low-level but yet very important machine architecture detail for programmers trying to implement a compiler. Most modern machines are 32-bit or 64-bit, i.e. there are 4 or 8 bytes in a word. Machines are either byte or word addressable. A piece of data is said to be aligned if it begins at a word boundary. Most machines have some sort of alignment restrictions or performance penalties for poor alignment.

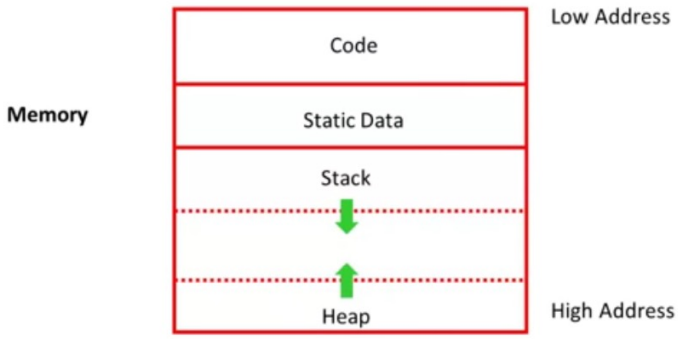

Figure 1.1: Partition of memory