Programming Languages

1 Overview

Types of Programming Languages.

Declarative languages focus on what a computer should do, while *imperative* languages focus on how a computer should do it.

- *Functional* languages: based on recursive definitions of functions, e.g. Lisp, ML, Haskell.
- *Dataflow* languages: model computation as the flow of information (tokens) amongst nodes, e.g. Id or Val.
- *Logical* languages: use predicate logic to find values satisfying certain specified relationships, e.g. Prolog.
- *Von Neumann* languages: computation based on variable manipulation, e.g. C, Fortran, Ada.
- *Object-oriented* languages: e.g. Java, Smalltalk, C++.
- *Scripting* languages: e.g. Python, Ruby, Perl, Lua, JavaScripts, PHP.

Compilation vs. Interpretation

Compilation	• Better performance • Thorough semantic analysis • Nontrivial translation
Interpretation	• Better diagnostics and flexibility

Steps of Compilation: Scanning, parsing, semantic analysis, intermediate code generation, machine independent code optimization, target code generation, machine dependent code improvement. Some languages like Java are compiled into another language (*source-to-source translation*) and then run via an interpreter (JVM). *Preprocessing.* Removes comments, whitespace, tokenizes characters, and may expand macros. The C preprocessor may even delete code segments for conditional compilation. *Linker.* Merges library routines into final program. *Assembler.* Translates assembly into machine language.

Bootstrapping. Refers to the development of successively more complex software. For example, compilers for new languages are developed in an existing language, then rewritten in the new language and compiled by itself (thus are *self-hosting*). *Just-in-time* compilation is when compilation occurs at runtime. *Microcode.* Technique that imposes an interpreter for the assembly-level instruction set.

Scanning & Parsing

Lexical analysis or *scanning* tokenizes characters. *Syntactic analysis* or *parsing* organizes the tokens into a parse tree according to a specified CFG. Next, *semantic analysis* occurs, during which a symbol table is built, static semantics are checked, and a simplified "abstract" parse tree is constructed. The symbol table is used to map identifiers to their information (type, scope, etc.). Semantic analysis constitutes the front-end of compilation. Target code generation constitutes the back-end. Optionally, a compiler may make code optimizations between the front and back ends.

2 Syntax

2.1 Scanning

Just review CSC 173 notes on DFAs and regular expressions. In particular, the process of converting a RE to an ϵ-NFA, to an NFA, to a DFA. Also read up on DFA minimization and implementing a DFA via a table.

2.2 LL Parsing

A *parser* calls the scanner to obtain tokens, assembles tokens into a parse tree, and then passes the parse tree (or reduced AST) to the rest of the compiler. In general, parsing if $O(n^3)$ but for LL or LR grammars we can do it in $O(n)$.

Classs	Scanning	Derivation Discovery	Parse Tree	Algorithm
LL	Left-to-right	Left-most	Top-down	Predictive
LR	Left-to-right	Right-most	Bottom-up	Shift reduce

Table 1: LL vs. LR Grammars.

Right-most derivations are sometimes called *canonical*. There are important subclasses of LR grammars, namely LALR, SLR, and 'full' LR. The number after a grammar, e.g. LL(k), denotes the fact that the grammar requires k look-ahead tokens to parse.

2.2.1 Recursive Descent

Uses subroutines that correspond 1-1 with non-terminals. A token `X` may *predict* a production of one of two reasons:

1. The RHS of the production, when recursively expanded, may yield a string beginning with `X`;
2. Or, the RHS is an epsilon production, in which case `X` may begin the yield of what *follows.*

See sec. 2.2.3.

2.2.2 LL(1) Grammars

A non-terminal A of a grammar is *left-recursive* if $A \Rightarrow^* A\alpha$ for some string of terminals α. Top-down parsing cannot manage left-recursive grammars. We may eliminate a production of the form $A \to A\alpha|\beta$ by replacing it with

$$A \to \beta A'$$
$$A' \to \alpha A'|\epsilon.$$

Another problem for top-down parsing is common prefixes, e.g. $A \to$ `abc`|`abd`. This is fixed with

$$A \to \texttt{ab}A'$$
$$A' \to \texttt{c}|\texttt{d}.$$

Note these techniques cannot always eliminate all left-recursive/common prefix non-terminals. Other heuristics may be applied, e.g. explicit end markers to handle dangling `else` statements, or pair the `else` with the nearest `if`.

2.2.3 First/Follow & Table Driven Parsing

A table-driven top-down parser maintains an explicit stack of the symbols it expects to see in the future. The parser pops the top symbol and performs the following:

(1) If the popped symbol is a terminal, the parser will attempt to match it against the incoming token from the scanner. If the matching fails, a syntax error is thrown and error recovery is initiated.

(2) Otherwise, the parser uses the popped nonterminal and the next available token to predict a production.

Initially, the parser contains the start symbol of the grammar, and when it predicts a production, it pushes the RHS of the production onto the stack in reverse order. Predict sets for a grammar are defined in terms of First and Follow sets, where $\textsc{First}(A)$ is the set of all tokens that could start A, and $\textsc{Follow}(A)$ is the set of all tokens that may proceed A. Then we say the predict set of $A \to \beta$ is $\textsc{First}(A)$, plus $\textsc{Follow}(A)$ if $\beta \Longrightarrow^* \epsilon$.

Formally, we define these sets as follows (see Fig. 2.24)

$$\begin{aligned}
\text{EPS}(\alpha) &\equiv \text{ if } \alpha \Longrightarrow^* \epsilon \text{ then } \texttt{true} \text{ else } \texttt{false} \\
\textsc{First}(\alpha) &\equiv \{\texttt{c} : \alpha \Longrightarrow^* \texttt{c}\beta\} \\
\textsc{Follow}(A) &\equiv \{\texttt{c} : \texttt{S} \Longrightarrow^+ \alpha\texttt{Ac}\beta\} \\
\textsc{Predict}(A \to \alpha) &\equiv \textsc{First}(\alpha) \cup (\text{if EPS}(\alpha) \text{ then } \textsc{Follow}(A) \text{ else } \varnothing).
\end{aligned}$$

2.3 LR Parsing

Suppose we want to model a parse of the following LR(1) grammar.

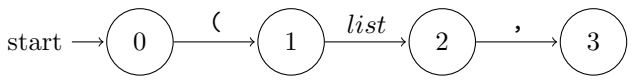
tuple ⟶ `(` *list* `)`
list ⟶ *list* `, const`
⟶ `const`

Initially, the parse stack is empty. We will augment the RHS of our productions with a • indicating the top of the parse stack. We call an augmented production, an *LR item*. Initially, we have *tuple* ⟶ • `(` *list* `)`. This constitutes the first state of our CFSM. If we match the open parenthesis, then we *shift* the item to the right, and enter a new state.

tuple ⟶ `(` • *list* `)`
list ⟶ • *list* `, const`
⟶ • `const`

Note that since *list* is a non-terminal we may be about to see the yield of that non-terminal coming up on the input. Thus we include the LR items of the *list* non-terminal and form a *closure* of all possible productions we may be in at the current state of the parse.

	State	Transition
0	*tuple* ⟶ • `(` *list* `)`	on `(` shift and goto 1
1	*tuple* ⟶ `(` • *list* `)`	on *list*, shift and goto 2
	list ⟶ • *list* `, const`	on *list*, shift and goto 2
	list ⟶ • `const`	on `const`, shift then reduce
2	*tuple* ⟶ `(` *list* • `)`	on `)`, shift then reduce
	list ⟶ *list* • `, const`	on `,`, shift and goto 3
3	*list* ⟶ *list* `,` • `const`	on `const`, shift then reduce



	list	`const`	`,`	`(`	`)`
0	-	-	-	s1	-
1	s2	sr3	-	-	-
2	-	-	s3	-	sr1
3	-	sr2	-	-	-

Table 2: s means shift, sr means shift reduce. The number afterwards either denoted the new state, or the production rule recognized.

3 Semantic Analysis

Static semantic rules are enforced at compile time, while dynamic semantic rules are checked at run time. Certain errors (array out-of-bounds, division by 0, etc.) cannot be caught at compile time so they must be enforced dynamically.

An *assertion* is a statement that a specified condition is expected to be true when execution reaches a certain point. Some languages also explicitly support *invariants, pre-* and *post-conditions*.

Attribute grammars attach semantics to the productions in a CFG (nodes in a parse tree). For example, an expression may be tagged with its type or an identifier with information from the symbol table. There are two types of semantic rules: (1) copy and (2) semantic function.

The process of evaluating attributes is called *annotation* or *decoration* of the parse tree. *Synthesized* attributes are calculated only in productions in which their symbol appears on the LHS. That is, are calculated only using values of attributes of their children. *Inherited* attributes, on the other hand, are calculated using attribute values of their parents and siblings.

An attribute grammar in which all attributes are synthesized is called *S-attributed*. The attribute flow in an S-attributed grammar is bottom-up. An *L-attributed* grammar is one in which all attributes may be evaluated in one left-to-right, depth-first traversal of the syntax tree. Attribute flows provide a partial order, for the evaluation of attributes. *Translation schemes* actually embed semantic rules in the RHS of productions, telling the compiler when to evaluate semantic functions (in a way consistent with the tree's attribute flow). The embedded semantic functions are called *action routines*.

Note bottom-up parsers are generally paired with S-attributed translation schemes (the action routine must be embedded after the productions *left-corner*, after which the production being parsed can be identified unambiguosuly), whereas top-down parsers are paired with L-attributed schemes (where the action routine may be placed arbitrarily within the production rule).

4 Names, Scopes, & Bindings

- Static Allocation. e.g. global vars, numeric/string const literals, compiler tables.
- Stack. Local variables, return address, saved registers, etc. Grows to lower addresses. Stack pointer points to top, frame pointer points to bottom of current frame, just above the return address.

- Heap. Dynamically allocated data. Internal frag: Allocates block larger than necessary. External frag: Free space, but not contiguous, too broken up by allocations.
- Garbage Collection. Mark & Sweep, Generation Collection (used in production code).

Scope: textual region of a program in which a binding is active.

- Statically scoped: binding times can be determined at compile time. Binding is to nearest lexical environment.
- Dynamically scoped: binding times determined at run-time. Binding is to most recent (time-wise) environment.

Elaboration: Process by which declarations become active when entering a new scope. The set of active bindings is called the *referencing environment.*

- *Deep Binding.* Binding to a ref. env. when a reference to a procedure is passed as parameter, assigned to var, or returned from a function, is chosen when the reference was created. Only time *closures* (subroutine with context) are needed.
- *Shallow Binding.* Binding is created until reference is actually used.

Aliasing. Multiple names for same thing. *Overloading* (see dynamic method dispatch). Same name for different things. We choose which function to execute by matching the signature with parameters being passed.

Inheritance. New classes can be defined at extensions or refinements of another. Dynamic method dispatch allows refined class to override methods of inherited class.

- First-class values: Can be passed as parameters, assigned to vars, or returned from a function.
- Second-class: Can only be passed as parameter.
- Third-class: None of the above.

E.g Anonymous function in functional languages are treated as first class values. Local objects in functional languages are given *unlimited extent*.

Static Link. Points to frame of lexically surrounding subroutine. *Dynamic Link*. Points to frame of calling subroutine.

5 Typing

Strongly typed. Language prohibits operations of objects for which the operation was not intended. *Statically typed.* Strongly typed and mostly typed checked at compile time.

- Denotational. A type is a set of values. An object has a given type if it belongs to that set.
- Structural. A type is either built-in or composite.
- Abstraction-based. A type is an interface consisting of a set of operations with well-defined, mutually consistent semantics.

Polymorphism. Code the works with multiple types.

- Subtype. Code the works on a given type also works on subtypes (subsets) of that type.
- Parametric. Type defined with type parameter. Can be done explicitly with generics, or implicitly (true polymorphism as in many functional languages).

Types of types.

- Numeric, e.g. reals, ints, rationals, booleans, chars.
- Enumeration, e.g. `type weekday = {mon, tues, wed, thurs, fri, sat, sun}`.
- Subrange, e.g. `1..100`.
- Composite, e.g. records, variant records (union), pointers, arrays, set, files, lists.

5.1 Inheritance

Single inheritance. Can have only one parent class.
Mix-in. Single parent class, unlimited interfaces.
True, Multiple Inheritance. Each class can have many parent classes (much more complicated).

6 OCaml

- `let` statements, type declarations, `match` statement (for pattern matching).
- Tail recursion is good because no new stack frame is needed.
- Lists must be homogeneous because of static typing.

7 Subroutines

7.1 Calling Sequences

The *calling sequence* (code executed immediately before/after a subroutine call), *prologue* (code executed at beginning of subroutine call), and *epilogue* are responsible for maintaining the call stack. For instance, the caller might

1) save and caller-saved registers whose values may be needed after the call
2) compute the values of any arguments and more them onto the stack/into registers
3) compute static link (it will either be the callers frame pointer or the caller will dereference its static link some number of times), and pass as hidden argument
4) jump to address of subroutine while pushing the return address onto stack.

In the prologue the callee may

1) allocate a frame by subtracting a constant from the stack pointer
2) save frame pointer onto stack and update it to point to new frame
3) save any callee-saved registers that may be overwritten.

The the epilogue could

1) restore callee-saved registers
2) restore the frame and stack pointers
3) jump to return address.

Finally, the caller

1) moves the return value to wherever it is needed

2) restores caller-saved registers.

Leaf routine makes no additional calls before returning, does not need to save return address to stack if it is already in a register. Other optimizations are possible.

In-line expansion of subroutines can save on overhead, such as space allocation, saving/restoring registers and allows for better compiler optimization. However, it increases code size and is not suitable, in general, for recursive subroutines.

7.2 Parameter Passing

Parameters in the declaration of a subroutine are *formal parameters.* Variables/expressions passed to subroutines are the *actual parameters.*

- *Call by value.* The actual parameter is assigned into a formal parameter. After this the two are independent.
- *Call by reference.* Formal parameter simply renames the actual parameter. Changes made to one, affect the other.
- *Call by result/value.* Passes parameter by value, then copies the formal parameter back at the end of the subroutine.
- *Call by sharing.* Copies actual parameter into formal, but both are references. For languages with a reference model of variables.
- *Read-Only.* Formal parameter cannot be changed by subroutine.
- `in`, `out` in Ada. `In` parameters pass info. from caller to callee; can only be read by callee. `Out` parameters pass from callee to caller; can be written but not read by callee.
- `Default parameters`. Need not be provided by caller. If not provided, then given default value.
- *Named parameters.* Association of actual and formal parameters is need not be positional.

Coroutines. Each have their own stack (Cactus stack), local variables, instruction pointer, but share global variables. Unlike, threads, there is only one coroutines running at any given time. The running coroutine must explicitly relinquish control.

8 Final Review

- Ocaml, Haskell and functional languages (A5/6/7/8).

  Understanding `let` and `match` statements. Type definitions. Variant types. Option type. Lambda functions. Iterators (true in Python/Ruby/C#, but not in Java/C++/Ocaml)

- Compilation (A8).
  - Parameter passing:
    - Pass-by-value
    - Pass-by-reference
    - Pass-by-name: Textual substitution
    - Pass-by-value/result: Pass by value, but copy back into original at end of function call.
  - Back-end is language independent: Intermediate code gen, machine independent code improvement, target code gen, machine dependent code improvement.
  - A *pass* of compilation as a phase(s) that is serialized (one must finish before the next starts). Two-pass compilers are most common.
  - Register allocation: In higher level intermediate forms, compiler uses unlimited register set. This must be mapped to the finite set specified by the ISA.
  - Translation to assembly.
  - Linking.

- Type.
  - Denotational: Type is a set of value. A value has a type if it belongs to the set.
  - Structural: Type is either a built-in type or a composite type made by a constructor.
  - Abstraction: Type is an interface consisting of operations w/ consistent semantics.
  - e.g. Numeric types (ints, floats, chars), enumerations, subranges, composite (records/structs, variant records/unions, arrays, sets, list, files, pointers)
  - Static typing: Java, Haskell, Scala, C/C++, Ocaml. Dynamic typing: Ruby, Python, Smalltalk, Perl, CommonLisp.
  - Strongly typed: Ocaml, Haskell, Python, Smalltalk, CommonLisp. Weakly typed: Perl, Pascal, C/C++.

- Object Orientation.
  - Simula → Smalltalk → Objective C/Eiffel. Also C++ → Java → C#.
  - Classified by inheritance, dynamic method dispatch, abstraction and information hiding.
  - Benefits of polymorphism: code reuse and modular design.
  - *Sub-type Polymorphism*: Code designed to work with values of some type, but can also work with sub-types.
  - *Parametric Polymorphism*: Code that takes type as a parameter.
    - Explicit: Generics (templates in C++).
      - Used to parameterize entire classes.
      - Serve as containers (whose operations are oblivious to the data types the objects contain)

        - Generics must always be types in Java, C#. Can also be values in C++, Ada. These languages also maintain a separate copy of the code for every instance of the generic code. Whereas in Java all instances share the same code. (If `T` is a generic type parameter in Java, then objects of class `T` are treated as instances of the `Object` class).
        - Type checking done at compile-time (at definition for Java, at instantiation for C++).
    * Implicit: Lisp, ML (compile-time). Also in Ruby, Python (run-time dynamic typing). For example, a the `min` function has type `a' -> a' -> a'`. The `a'` serve as implicit type parameters.
  - Dynamically-typed OOP languages typically use a single mechanism to implement both parametric and sub-type polymorphism.
  - Dynamic vs. Static Method Binding. Static method binding 'uses the reference type' whereas dynamic method binding 'uses the class of the object'. For example, if `student` and `professor` are both derived classes of the class `person`, all of which contain a `printId` method. Then in a language (e.g. C++, C#, Simula) the uses SMB, if `person *x = &(student) s`, then `p.printId` calls `person.printId` (this can be overridden by adding the `virtual` keyword), whereas in a language with DMB (e.g. Java, Eiffel, Ruby, Python, Smalltalk or Objective C), it calls `student.printId`.
  - True inheritance: C++, Eiffel, CLOS, OCaml, Python.

- Translation (A4/7).
- Concurrency (A9).
  - *Concurrency*: Two or more tasks may be underway (at an unpredictable point of their execution) at the same time. (Coroutines are not concurrent.)
  - *Parallel*: A concurrent system in which two tasks are active at the same time (as opposed to quasi-parallelism of random context switches on a uni-processor machine).
  - *Race condition*: two or more threads 'race' to same point in code and the output depends on which arrives first.
  - Implementing synchronization
    * Spinning/busy-waiting - threads loops until some condition becomes true.
      - Spin locks
      - `test-and-set` sets a Boolean to true and returns the previous value. `while not test-and-set(L): loop` This leads to too much communication on hardware (contention).
      - `compare-and-swap` takes a location, expected value, and new value. Checks whether location has expected value, returns whether change was made.
      - Barriers - All threads active must stop at a point until all other threads catch up.
    * Blocking - (scheduler based) threads voluntarily relinquish control.
      - *Semaphore*: A counter with two operations `P` and `V`. Calling `V` increments counter, calling `P` waits until counter is positive, then decrements.
      - *Monitors*: (abstraction of semaphore) an object with operations and condition variables.
    * Non-blocking: No dependence on other threads behavior. e.g. Michael & Scott queue: a non-blocking queue that can be used to implement, for example, a ready list (list of blocked threads which aren't currently executing).
  - Threads Creation: Co-begin, Parallel loops, Fork/Join - leads to arbitrary patterns in concurrent control flow.

  - Problem: Cache coherence between threads executing on separate cores of a machine.

- Garbage Collection.
  - Pointers are useful for (1) efficient access to elaborated objects and (2) dynamic creation of linked data structures.
  - Reference Counting: Requires strongly-typing. Don't work for circular data structures. Define usefulness as whether the an object has a pointer to it.
    * Smart pointers.
  - Tracing: Defines usefulness as whether an object is reachable from a sequence of valid pointers, starting from outside the heap.
    * *Mark & Sweep*: Mark everything useless. Beginning with pointers outside of heap, recursively explore all linked data, marking newly discovered blocks are useful. Stopping when it reaches an already useful block. Remaining useless blocks are added to freelist. (Pointer reversal: when collector explores a path, it reverses the pointers it follows back to the previously explored block. This way an explicit stack is not needed for the recursive exploration, we can just follow the flipped pointers backwards.)
    * *Generational*: Heap split into regions of young and old data. When heap space is low the young data is GC'd. If data in young section survives enough generations, it is advanced to older region.
    * *Stop & Copy*: Divide heap in half, when current is full, useful data is copied to second half (no external frag). First half is all free.

- Run-time Program Management.
  - *Run-time system*: Set of libraries on which the language implementation depends for correct operation. e.g. GC, variable number of args, exception handling, event handling, thread implementation, etc.
  - *Virtual Machine (VM)*: provides complete programming environment. Its application programming interface (API) includes everything required for correct execution of programs that run above it. Provides level of abstraction comparable to that of a computer implemented in hardware.
    * Process VM: Emulates only environment needed by single user-level process.
    * System VM: Gives user privileged instructions unlike process VM.
    * Java Virtual Machine (JVM).
    * Common Language Infrastructure (CLI). Developed by Microsoft in the '90s. Satisfied need for interoperability amongst programming languages on Windows. Like the JVM, it defined multi-threaded, stack-based support for GC, with exceptions, virtual method dispatch, mix-in inheritance (like in ruby modules). CLI has a richer type system and calling mechanism. Defined the Common Type System (CTS), Common Language Specification (CLS), and Common Intermediate Language (CIL).
  - Just-in-time Compilation: compile immediately before execution. Adds delay to start-up time. Solution was to use byte code to only recompile what would actually be needed. Still compilation must sometimes be delayed until execution (e.g. for performance optimizations based on data gathered during execution).
  - Binary translation: Recompilation of object files. Allows for already-compiled programs to run on different ISAs. Challenging due to loss of information.
  - Binary rewriting: Technique to modify existing executable programs. Used for simulating new architectures, "sandboxing" untrusted code, or auditing the quality of a compiler's optimizations, etc.

- Locality.
  - *Miss ratio*. Let for a fixed sequence of data requests, let $mr(c)$ denote the miss ratio on a cache of size c.
  - *Reuse Distance*: number of distinct data blocks that have been accessed since previous access to that block. Lower rd better locality.

    $$mr(c) = P(rd > c)$$

  - *Footprint*: $fp(x)$ is the average workspace size over all windows of length x. Smaller footprint implies better locality. Denning-HOTL conversion:

    $$mr(c) = \Delta fp(x)|_{fp(x)=c}.$$

  - Xiang's formula

    $$\lim_{n\to\infty} fp(x) = m - \sum_{i=x+1}^{\infty} (i-x)P(rt = i)$$

    where m is the number of data blocks in the trace, n is the length of the trace, x is window size, $P(rt = i)$ is the probability that an access has reuse time i.

- Call-by-Sharing: Differs from call-by-value because although we copy the actual parameter into the formal parameter, both are references (modification of object in a subroutine is reflected in the actual parameter). Also differs from call-by-reference because the called subroutine cannot make the argument refer to a different object. (e.g. Java uses call-by-sharing for user-defined class types)

- Functions as Values.
  - *Referencing Environment*: Set of active bindings.
  - The question is what to do (when should the referencing environment be created) when we create a reference to a subroutine (e.g. when passing one as a parameter).
  - *Static Scoping*: Specify that the referencing environment depends on the **lexical** nesting of a program (deep binding is default).
  - *Dynamic Scoping*: Specify that referencing environment depends on the order in which declarations are encountered at **run time** (shallow binding is default).
  - *Shallow Binding*: (Late) Referencing environment of callee is not created until it is actually called.
  - *Deep Binding*: (Early) Referencing environment is created when the reference is created.
  - *First-class Value*: Passed as parameter, returned from a function, or assigned to a variable.
  - *Second-class*: Passed as parameter.
  - *Third-class*: None.
  - Subroutines has first class status in most functional/scripting languages as well as in C/C++/C#/Fortran.
  - *Unlimited Extent*: Local objects have indefinite lifetimes (required because a reference to a subroutine may outlive the execution of the scope in which it was declared).
  - Lambda expressions are present in functional/scripting languages as well as C++, C# and Java.
  - *Static Link*: Points to most recent invocation of lexically surrounding subroutine (needed only for nested subroutines).

- Stack.
  - Stack frame contains local variables, saved registers, return address, arguments to called routines, and other miscellanea.
  - Maintenance of stack is the responsibility of calling sequence: the code executed immediately before and after a subroutine call - the prologue (code executed at the beginning) and epilogue (executed at end) of the subroutine itself.
  - Before a subroutine call, the caller pushes the parameters onto the stack, saves its `fp`, the return address, and leaves space for a return value (being sure to maintain `sp`), and jumps to the subroutine. The prologue of the subroutine allocates a new frame by setting `fp` to the current `sp` and adding space for local variables. In the epilogue, the subroutine pushes the return value onto the stack, resets `sp`, before jumping back (using the return address already on the stack). Finally, back in the caller, `fp` is reset, the return value is copied as needed, and `sp` readjusted.

References

[1] Scott, Michael. *Programming Language Pragmatics*. 4th ed. 2016.