Chapter 2 :: Programming Language Syntax

Programming Language Pragmatics

Michael L. Scott

Review

- What is a programming language ?
- Why are there so many programming languages ?
- List several of the factors that make programming languages successful
- List several of the reasons that programming languages are studied
- What are the tw broad groups of programming languages ?

Review

- What are declarative languages ?
- What are imperative languages ?
- List some languages in each group
- Describe pure compilation/interpretation
- Describe some compilation strategies
- Provide a brief overview of the compilation process

Regular Expressions

- A regular expression is one of the following:
  - A character
  - The empty string, denoted by ε
  - Two regular expressions concatenated
  - Two regular expressions separated by | (i.e., or)
  - A regular expression followed by the Kleene star (concatenation of zero or more strings)

Regular Expressions

- Numerical literals in Pascal may be generated by the following:

digit ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
unsigned_integer ⟶ *digit digit* *
unsigned_number ⟶ *unsigned_integer* ((. *unsigned_integer*) | ϵ)
(((e | E) (+ | - | ϵ) *unsigned_integer*) | ϵ)

Context-Free Grammars

- The notation for context-free grammars (CFG) is sometimes called Backus-Naur Form (BNF)
- A CFG consists of
  - A set of *terminals* *T*
  - A set of *non-terminals* *N*
  - A *start symbol* *S* (a non-terminal)
  - A set of *productions*

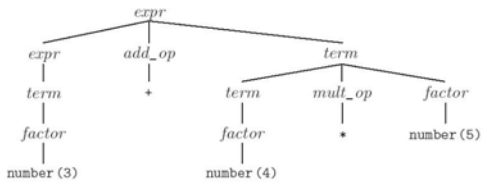
Context-Free Grammars

- Expression grammar with precedence and associativity

1. *expr* ⟶ *term* | *expr add_op term*
2. *term* ⟶ *factor* | *term mult_op factor*
3. *factor* ⟶ `id` | `number` | `-` *factor* | `(` *expr* `)`
4. *add_op* ⟶ `+` | `-`
5. *mult_op* ⟶ `*` | `/`

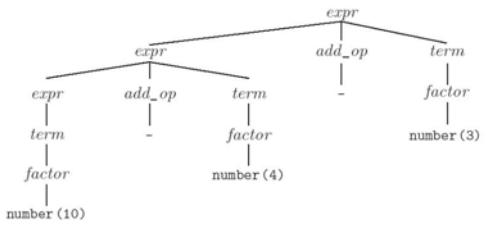
Context-Free Grammars

- Parse tree for expression grammar (with precedence) for `3 + 4 * 5`



Context-Free Grammars

- Parse tree for expression grammar (with left associativity) for `10 - 4 - 3`



Chapter 3:: Names, Scopes, and Bindings

Programming Language Pragmatics

Michael L. Scott

Name, Scope, and Binding

- A name is exactly what you think it is
  - Most names are identifiers
  - symbols (like '+') can also be names
- A binding is an association between two things, such as a name and the thing it names
- The scope of a binding is the part of the program (textually)in which the binding is active

Binding

- Binding Time is the point at which a binding is created or, more generally, the point at which any implementation decision is made
  - language design time
    - program structure, possible type
  - language implementation time
    - I/O, arithmetic overflow, type equality (if unspecified in manual)

Binding

- Implementation decisions (continued):
  - program writing time
    - algorithms, names
  - compile time
    - plan for data layout
  - link time
    - layout of whole program in memory
  - load time
    - choice of physical addresses

Binding

- Implementation decisions (continued):
  - run time
    - value/variable bindings, sizes of strings
    - subsumes
      - program start-up time
      - module entry time
      - elaboration time (point a which a declaration is first "seen")
      - procedure entry time
      - block entry time
      - statement execution time

Binding

- The terms STATIC and DYNAMIC are generally used to refer to things bound before run time and at run time, respectively
  - "static" is a coarse term; so is "dynamic"
- *IT IS DIFFICULT TO OVERSTATE THE IMPORTANCE OF BINDING TIMES IN PROGRAMMING LANGUAGES*

Binding

- In general, early binding times are associated with greater efficiency
- Later binding times are associated with greater flexibility
- Compiled languages tend to have early binding times
- Interpreted languages tend to have later binding times
- Today we talk about the binding of identifiers to the variables they name

Binding

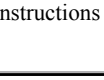
- Scope Rules - control bindings
  - Fundamental to all programming languages is the ability to name data, i.e., to refer to data using symbolic identifiers rather than addresses
  - Not all data is named! For example, dynamic storage in C or Pascal is referenced by pointers, not names

Lifetime and Storage Management

- Key events
  - creation of objects
  - creation of bindings
  - references to variables (which use bindings)
  - (temporary) deactivation of bindings
  - reactivation of bindings
  - destruction of bindings
  - destruction of objects

Lifetime and Storage Management

- The period of time from creation to destruction is called the LIFETIME of a binding
  - If object outlives binding it's garbage
  - If binding outlives object it's a dangling reference
- The textual region of the program in which the binding is *active* is its scope
- In addition to talking about the *scope of a binding*, we sometimes use the word *scope* as a noun all by itself, without an indirect object



Lifetime and Storage Management

- Storage Allocation mechanisms
  - Static
  - Stack
  - Heap
- Static allocation for
  - code
  - globals
  - static or own variables
  - explicit constants (including strings, sets, etc)
  - scalars may be stored in the instructions

Lifetime and Storage Management

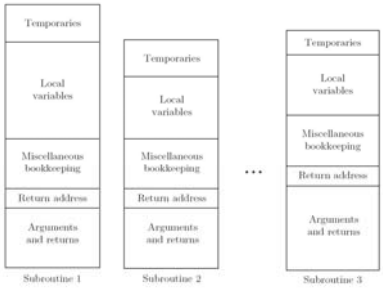


Figure 3.1: **Static allocation of space for subroutines in a language or program without recursion.**

Lifetime and Storage Management

- Central stack for
  - parameters
  - local variables
  - temporaries
- Why a stack?
  - allocate space for recursive routines (not necessary in FORTRAN – no recursion)
  - reuse space (in all programming languages)

Lifetime and Storage Management

- Contents of a stack frame (cf., Figure 3.2)
  - arguments and returns
  - local variables
  - temporaries
  - bookkeeping (saved registers, line number static link, etc.)
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time

Lifetime and Storage Management

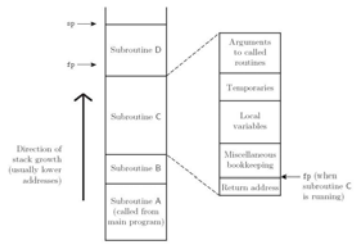


Figure 3.2: **Stack-based allocation of space for subroutines.** We assume here that subroutine A has been called by the main program, and that it then calls subroutine B. Subroutine B subsequently calls C, which in turn calls D. At any given time, the stack pointer (`sp`) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (`fp`) register points to a known location within the frame (activation record) of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.