Ahram Kim

Jim Buffenbarger

CS354 - 001

September 26, 2017

Textbook Assignment 1 : Introduction

1. Question 1.1. Errors in a computer program can be classified according to when they are detected and, if they are detected at compile time, what part of the compiler detects them. Using your favorite imperative language, give an example of each of the following.
    a. A lexical error, detected by the scanner

       : As using java, the example of a lexical error is String statement = "How are you?". A variable name cannot begin by a number.
    b. A syntax error, detected by the parser

       : As using java, syntax error is to missing ';' at the end of a statement.
    c. A static semantic error, detected by semantic analysis

       : As using java, a static semantic error is to access a method that is not declared in the class or the superclasses.
    d. A dynamic semantic error, detected by code generated by the compiler

       : As using java, the dynamic semantic error is to attempt to access an element beyond the bounds of an array. For example, int [] array = new int[10]; Array[10] = 19; This is array index out of range.

   e. An error that the compiler can neither catch nor easily generate code to catch (this should be a violation of the language definition, not just a program bug) : As using java, an error that can’t reasonably by caught is to use method name as a variable.

2. Question 1.8. The Unix make utility allows the programmer to specify dependences among the separately compiled pieces of a program. If file A depends on file B and file B is modified, make deduces that A must be recompiled, in case any of the changes to B would affect the code produced for A. How accurate is this sort of dependence management? Under what circumstances will it lead to unnecessary work? Under what circumstances will it fail to recompile something that needs to be recompiled?

   : Make depends on file modification times, maintained by the operating system. Because it works at the granularity of files, it will force recompilation of everything that depends on file A whenever anything in A or even a comment changes. It will also force recompilation if the date on the file changes for a spurious reason: e.g., due to compression, copying, etc. By the same token, if file B depends on A, and the date on B changes for a fake reason, make may fail to recognize that recompilation is needed. Because make operates independently of the compiler, and has no knowledge of language semantics, it may also fail to perform needed recompilations if the programmer makes an error in describing inter-file dependences.

3. Question 2.1. Write regular expressions to capture the following.

a. Strings in C. These are delimited by double quotes ("), and may not contain newline characters. They may contain double-quote or backslash characters if and only if those characters are "escaped" by a preceding backslash. You may find it helpful to introduce shorthand notation to represent any character that is not a member of a small specified set

   String → " (\ \ . | [not \ \ "]) * "

b. Comments in Pascal. These are delimited by (* and *) or by { and }. They are not permitted to nest.

   : " * " (. | \n) * " * "

   " { " [^ }] * " } "

c. Numeric constants in C. These are octal, decimal, or hexadecimal integers, or decimal or hexadecimal floating-point values. An octal integer begins with 0, and may contain only the digits 0–7. A hexadecimal integer begins with 0x or 0X, and may contain the digits 0–9 and a/A– f/F. A decimal floating-point value has a fractional portion(beginning with a dot) or an exponent (beginning with E or e). Unlike a decimal integer, it is allowed to start with 0. A hexadecimal floating-point value has an optional fractional portion and a mandatory exponent (beginning with P or p). In either decimal or hexadecimal, there may be digits to the left of the dot, the right of the dot, or both, and the exponent itself is given in decimal, with an optional leading + or - sign. An integer may end with an optional U or u (indicating "unsigned"), and/or L or l (indicating "long") or LL or ll (indicating "long long"). A floating point value may end with an optional F or f (indicating "float"—single precision) or L or l (indicating "long"—double precision).

: digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

alp → a | b | c | d | e |f

Up_alp → A| B | C | D | E | F

decimal → digit +

Octal → 0 [0-7]*

hexadecimal integer's → 0x|0X digit alp| Up_alp

decimal floating → digit+.digit*

Hexadecimal floating-point values → 0x|0X digit+ alp*| Up_alp*

Constant → octal| decimal| hexadecimal integer's |decimal floating| hexadecimal floating-point values

d. Floating-point constants in Ada. These match the definition of real in Example2.3,except that (1) a digit is required on both sides of the decimal point, (2) an underscore is permitted between digits, and (3) an alternative numeric base may be specified by surrounding the nonexponent part of the number with pound signs, preceded by a base in decimal (e.g., 16#6.a7#e+2). In this latter case, the letters a..f (both upper- and lowercase) are permitted as digits. Use of these letters in an inappropriate (e.g., decimal) number is an error, but need not be caught by the scanner.

: Ada_int → digit ((_ | ε) digit)*

extended_digit → digit | a | b | c | d | e | f | A | B| C| D| E | F

Ada_extended int → extended_digit ((_ | ε) extended_digit)*

Ada_FP_num → ((Ada_int ((. Ada_int | ε))

(Ada_int # Ada_extended_int

((. Ada_extended_int) | ε) #))

(((e | E) (+ | - | ε) Ada_int) | ε)

e. Inexact constants in Scheme. Scheme allows real numbers to be explicitly inexact (imprecise). A programmer who wants to express all constants using the same number of characters can use sharp signs (#) in place of any lower-significance digits whose values are not known. A base-10 constant without exponent consists of one or more digits followed by zero of more sharp signs. An optional decimal point can be placed at the beginning, the end, or anywhere in-between. (For the record, numbers in Scheme are actually a good bit more complicated than this. For the purposes of this exercise, please ignore anything you may know about sign, exponent, radix, exactness and length specifiers, and complex or rational values.)

   : digit+ # * (. # * | ε) digit*. digit+ # *

f. Financial quantities in American notation. These have a leading dollar sign ($), an optional string of asterisks (*—used on checks to discourage fraud), a string of decimal digits, and an optional fractional part consisting of a decimal point (.) and two decimal digits. The string of digits to the left of the decimal point may consist of a single zero (0). Otherwise it must not start with a zero. If there are more than three digits to the left of the decimal point, groups of three (counting from the right) must be separated by commas (,). Example: $**2,345.67. (Feel free to use "productions" to define abbreviations, so long as the language remains regular.)

: nzdigit → 1 | 2 | 3 | 4 | 5 | 4 | 7 | 8 | 9

digit → 0 | nzdigit

group →, digit digit digit

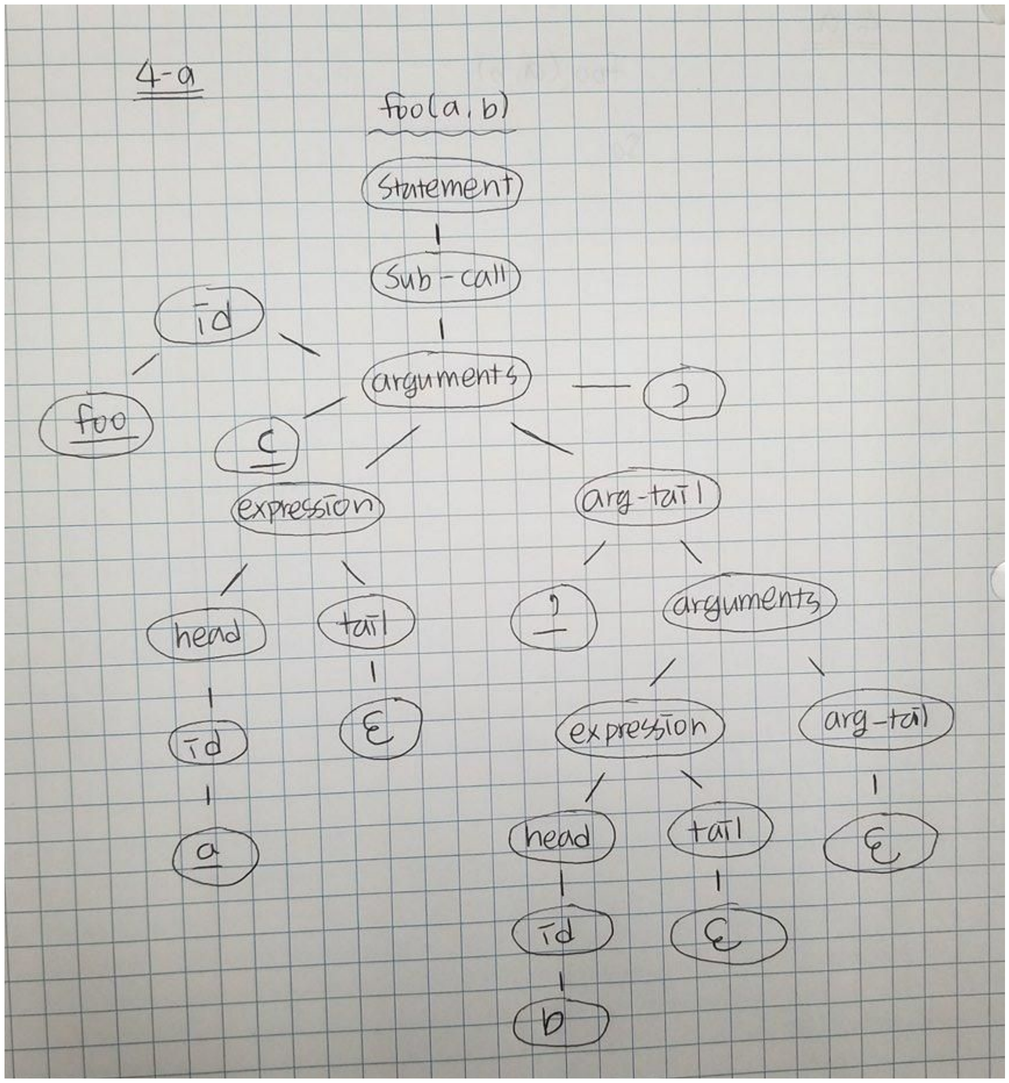
number → $ * * (0 | nzdigit (ε | digit | digit digit) group*) (ε | . digit digit)

4. Question 2.13 (a, b). Consider the following grammar:

```
stmt ⟶ assignment
     ⟶ subr_call
assignment ⟶ id := expr
subr_call ⟶ id ( arg_list )
expr ⟶ primary expr_tail
expr_tail ⟶ op expr
          ⟶ ϵ
primary ⟶ id
        ⟶ subr_call
        ⟶ ( expr )
op ⟶ + | - | * | /
arg_list ⟶ expr args_tail
args_tail ⟶ , arg_list
          ⟶ ϵ
```

a. Construct a parse tree for the input string foo(a, b)

:



b. Give a canonical (right-most) derivation of this same string.

: Stmt

Sub-call

id(arguments)

foo(expression arg-tail)

foo(head tail , arguments)

foo (id ε , expression arg-tail)

foo(a , head tail ε)

foo(a , id ε)

foo(a , b)

5. Question 2.17. Extend the grammar of Figure 2.25 to include if statements and while loops, along the lines suggested by the following examples.

```
abs := n
if n < 0 then abs := 0 - abs fi

sum := 0
read count
while count > 0 do
    read n
    sum := sum + n
    count := count - 1
od
write sum
```

Your grammar should support the six standard comparison operations in conditions, with arbitrary expressions as operands. It should also allow an arbitrary number of statements in the body of an if or while statement.

: Program → stmt_list $$

Stmt_list → stmt_list stmt

Stmt_list → id(stmt) {stmt_list}

Stmt_list → id(stmt){stmt_list}

Stmt_list → stmt

Id → if

Id → while

Stmt → id:=expr

Stmt → read id

Stmt → write expr

Expr → term

Expr → expr add_op term

Term → factor

Term → term mult_op factor

Factor → (expr)

Factor → id

Factor → number

Add_op → +

Add_op → -

Multi_op → *

Mult_op → /