Programming Language Theory

- aliasing and overloading
- closures
- syntax
- lexical scanning
- parsing

Aliasing

- when two or more pointers or references refer to the same object (same memory), they are said to be **aliases** to the same object
- intentional aliasing can be useful: for example, when we pass an object into a method with the intention that the method will modify the contents of the object
  - the method parameter aliases the object reference in the calling code
- another example is a **union**, where the same block of memory is seen as different types, such as an array of bytes vs. different individual fields
- unintentional aliasing is a good source of errors
- the mere possibility of aliasing may lessen the chances for compiler optimization, because e.g. the compiler may have to keep a value in memory rather than in a register

C Unions

- another example of aliasing is a **union**
- in a union, the same block of memory is seen as different types, such as an array of bytes vs. different individual fields
- real-life example from /usr/include/elf.h:

```
typedef struct
{
  Elf64_Sxword d_tag;    /* Dynamic entry type */
  union
    {
      Elf64_Xword d_val;    /* Integer value */
      Elf64_Addr d_ptr;     /* Address value */
    } d_un;
} Elf64_Dyn;
```

- the memory for `d_un` can hold either an integer or an address

Overloading

- when two or more objects are referred to by the same name, that is **overloading**
- overloading of methods is common in strongly-typed object-oriented languages: the type of the parameters is used to disambiguate which method is intended
- overloading also includes the use of the same operator, such as +, for different operations, including integer addition, floating point addition, and (e.g. in Java) string concatenation

Closures

- a function f declared within another function g may use the variables declared in g

  e.g. in ocaml,

  ```
  let g x =
      (let f y = x + y in f);;
  ```

- this matters with higher-order functions, when the lifetime of f may be longer than the lifetime of g: the variables used by f must be kept alive (i.e., not garbage-collected) as long as f can be used
- when function f is created and returned as the result of a call to g, a hidden object called a `closure` is created by the compiler
  - the closure is hidden in the same way that instance methods in Java don't explicitly show the object parameter
- the closure makes these variables available to f when f is called, and protects the variables from garbage collection
- with dynamic scoping, the external variables may be on the stack, and their value must be copied into the closure when the closure is created, before the function g returns and makes the stack frame available for re-use

Programming Language Syntax

- **syntax** (meaning “grammar”) defines which programs are well-formed in a particular programming language
  - **semantics** define the meaning of well-formed programs
- syntax is often defined using a mathematical notation that includes

    | vertical bar to list alternatives

    * star to allow 0 or more repetitions

        + plus to allow 1 or more repetitions

    [square brackets to identify optional parts]

    " quotes around constant strings "

  - and lists of rules with named objects representing each rule
- the syntax for floating point numbers might be defined as:
  - float → [“+” | “-”] int “.” | int “.” int | “.” int [“e” int]
  - int → (“0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”) +

Two-Level Syntax

- most programming languages define the syntax for statements and expressions in terms of tokens
- and have a different set of rules to define what counts as a token
- so there is a syntax for tokens, and a separate syntax for statements and expressions

Syntax for Tokens: Regular Expressions

- tokens can usually be defined by a relatively simple set of mathematical rules that fall under the description of **regular expressions**
- regular expressions can be defined using the constructs on the previous slide without the need for recursion
  - regular expressions are common also for searches, for example in egrep

Syntax for Statements and Expressions: Context-Free Grammars

- the syntax for statements, expressions, and other language constructs is defined using the same expressions, but needs recursion to be able to accomodate, for example, a function call with a parameter value computed by a call to another function:

  f(g(x))

  - regular expressions do not support recursion
- such a syntax that allows recursion is mathematically described by **context- free grammars**

Tokens

- many programming languages have a rule for identifiers that is approximately as follows:
- id = alpha alphanum*
- this allows identifier names such as thirty3, but not 30three
- alphanum may allow additional characters, commonly including _, but in lisp also - and others
- the process of converting an input file into a list of tokens is called **scanning** or **lexical scanning**

Lexical Scanning

- while scanning, white space, newlines, and comments are recognized as token separators, and otherwise ignored
  - python is a notable exception, it uses the whitespace at the beginning of a line to determine the nesting depth of blocks of code
- a scanned token has a type, e.g. Plus or Identifier, and may have a value
  - Using ocaml notation, we would write this along the lines of:

    ```
    type token = Plus | Minus | Id of string | Int of int
    ```
- a **parser** takes a list of tokens, either as a complete list or incrementally one token at a time, and produces a syntax tree
  - if the lexical scanning completes before parsing begins, scanning and parsing are considered separate passes over the source program, otherwise the parsing and lexing are considered a single pass

Lexical Scanners and Finite Automata

- a scanner typically reads one character at a time
- a scanner also has a state, for example recording whether the scanner is in the middle of scanning a number or an identifier
- ideally, the input character and state uniquely identify whether the new character:
  - extends the symbol currently being built, or
  - ends the symbol currently being built, possibly beginning a new symbol

  and also specifies the next scan state
- if this ideal is not met, the scanner might need to **look ahead** in the character stream to disambiguate such possibilities
  - if the symbol ends here, any characters in the look ahead must be processed later
- once an token is scanned, can determine if it is a reserved keyword such as "else"
- the scanner also generates a special symbol to identify the end of the input
- regular expressions can be mechanically compiled into scanners using **finite automata theory**

Context-Free Grammars

- regular expressions cannot specify that parentheses should match, but context-free grammars can:

  ```
  expr → id | num | - expr | ( expr ) | expr op expr
  ```

  in this example, id, num, -, (,), op are all tokens
- in a context-free grammar, the symbols on the left-hand side of a production are non-terminals, all other symbols (which we identify as tokens) are terminals
- this notation is based on formalism developed by John Backus and Peter Naur, and therefore known as Backus-Naur notation
- again using ocaml notation, the above rule can be defined as:

  ```
  type num = Int of int | Double of float;;
  type op = Plus | Minus | Times | Div | Mod;;
  type token = Op | Id of string | Num of num;;
  type expr = Id of string | Num of num | Minus of expr
            | Parenthesized of expr | Op of expr * op * expr;;
  ```

- after parsing, `(x + 3) * 3.14` would give the parse tree:

  ```
  Op (Parenthesized (Op ((Id "x"), Plus, Num (Int 3))),
      Times, Num (Double 3.14));;
  ```

- grammars can also use the symbol ε (epsilon) to allow an empty string

Parsing

- similar to the finite automata theory of lexical scanning, there has been much theoretical work around parsing
- top-down parsing creates the syntax tree by first creating the root node, then adding children based on the tokens in the input
  - top-down parsing requires selecting the root node to create at every point in the parse, so may require rewriting the grammar so that this decision can be made
    - e.g. an expression is always a term followed by a term_tail
    - the term_tail is either a +/- followed by another expression, or the empty string
- bottom-up parsing pushes tokens onto a stack until it has enough information to completely build a subtree, then builds the subtree, removes the tokens from the stack, and pushes the subtree onto the stack

Automated Parsing

- given a suitable grammar, a program can mechanically create a parser
  - `yacc` (Yet Another Compiler Compiler) was the original parser generator distributed with Unix, and lex was the lexical scanner generator
- the parser is similar to a scanner using finite automata, augmented with a stack:
  - a parser typically reads one token at a time
  - a parser has a state, for example recording whether the parser is in the middle of scanning an expression
  - a parser has a stack that may contain subtrees or tokens from the input
- ideally, the combination of input token, parse state, and the values on the stack uniquely identifies which grammar rule to apply, also determining the next state and what to pop from the stack and push onto the stack
- automated parsers can be built in both the top-down and bottom-up styles
- bottom-up is more common because it has fewer limitations on the form in which the grammar must be written