CSCI 3136

Program Translation to Names, Scopes, and Bindings

Zeh

Alex Safatli

Tuesday, January 9, 2013

Contents

Introduction

Overview

This course is about how to talk to your computer. The programmer's dream is to just say what you want in English. This is very expensive, is ambiguous, and has low effort on part of the programmer. We know that CPUs read code through assembly at an extremely low-level (the one on the slide is already 1 level abstraction above actual assembly at this level). This is very low-level, tedious, but unambiguous.

To bridge the gap, we develop programming languages (Java, C++, ...). We want them to be as expressive as possible at the same time as being unambiguous, expressive, and efficiently implementable (does not take much effort to become low-level). The constructs developed *have* to be able to be translated into low-level code.

Talking to the Computer

There are two distinctive parts to this.

1. Program Semantics and Translation
2. Language Features and Their Implementation

The first part is about *finding ways to specify programming languages formally.* What is syntactically correct for a given language? What does that mean? What is the computation it specifies? *This leads to the basics of compilation and interpretation.* How do we translate a syntactically correct program into (machine) code that carries out exactly the same computation?

The second part focuses on *programming paradigms.* What are different high-level ways of thinking about computation? Each paradigm is suited best for certain cases of application. Furthermore, each programming language designed to express a certain subset of these paradigms more easily. Each language has a different set of semantics associated with them.

But why is all of this useful? Knowing about these make learning new languages and understanding obscure features easier. It helps one choose the most appropriate language for the job and to evaluate trade-offs. We can learn to simulate useful features in languages that lack them, and use features of languages more effectively.

NOTE GTK is a way to implement object-oriented programming into C, a language that lacks such a feature.

Programming languages started with machine and assembly language, led to the creation of FORTRAN, and eventually leading to C, C++, Perl, Python, Java, JavaScript, PHP, and C#. But why are there so many languages? Evolution, special purpose, and personal preference are to blame.

What makes a language successful?

- Expressive power,
- ease of learning,
- ease of implementation,
- whether or not it is open-source,
- good compilers,

Talking to the Computer continued on next page...

- economics, patronage, inertia, ...

There are different *programming paradigms.*

- **Imperative Programming**: Statements changing program state. Closest to von Neumann machine (assemblers, FORTRAN, BASIC, COBOL). Structured programming (ALGOL, Pascal, C).
- **Object-Oriented Programming**: Representation of concepts as objects. A different way of expressive an imperative program. Examples: Smalltalk, C++, Java.
- **Functional Programming**: Evaluation of different mathematical functions. By definition, these do not interact with the outside world (which is what taking input in is). Examples: Lisp, Scheme, ML, Haskell.
- **Logic Programming**: Establishing a set of logical rules. A search strategy will then ensue to find the solution. How to do something rather than what to be done. Example: Prolog, VisiCalc.

Course Outline

The course will lead into *program translation*, *flow of control and information*, and then into *building abstractions.* See the slide. The required textbook will be available in the bookstore on Friday. Slides will be available online at `http://www.cs.dal.ca/~nzeh/Teaching/3136`.

Evaluation will be split into three parts: 40% for assignments (weekly), 20% for the midterm exam, and 40% for the final exam. Assignment groups are allowed for assignments: up to groups of three. A joint assignment will thus be handed in. Worst two assignments will not count.

Office hours occur on Wednesdays from 12 until 2pm, and on Friday from 1 until 3pm. Assignments will be handed out on the left-hand side column on the webpage, and will be passed in in-class unless they are programming assignments. Code will be e-mailed in those situations. Scheduling of the assignment has changed; the first assignment will come out on Monday and the last assignment will come out a week later than intended.

This course uses Java, C, C++, Python, Ruby, Perl, Scheme, and Haskell to illustrate important programming concepts. Other languages may be mentioned in passing.

Program Translation

Compilation v. Interpretation

There are essentially two parts in this course related to this question of bridging the gap between expressive languages and achieving efficiency. Efficiency concerns efficiency of translating code into machine code, and how fast the code actually runs in the end (how easy is it to implement the runtime features). The first part of this course regards *translation.*

Compilation treats turning the source program into machine code separate from running. The compiler no longer becomes involved with the target program that takes in input and produces output. **Interpretation** involves an interpreter that takes in input and a source program and produces output – machine code is produced while running.

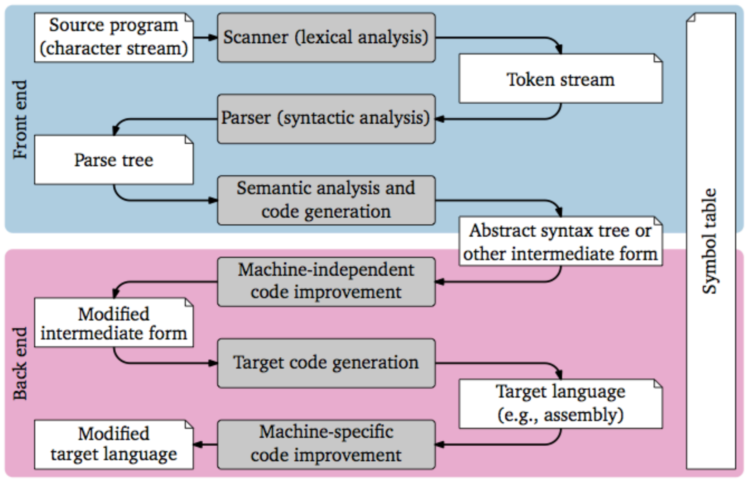
Compilation v. Interpretation continued on next page. . .

NOTE Hadoop, which is a free version of a Google project for parallelization, is written in Java. There is a reason this choice was made where the original code was taken from Python and ported.

There are good reasons to use one of the two of these. The question is — what is the trade-off? Advantages of compilation is that it creates *standalone code*, *faster code*, and *smaller code*. This is advantageous for commercial programs; otherwise, the source code has to be passed around. On the other hand, interpretation allows for faster development, more flexibility and possibly more expressive power, late binding, and dynamic features.

The boundary between compilation and interpretation is fuzzy. How do you determine if a language is one or the other? Just in time (JIT) compilation, the Perl interpreter compiles programs into intermediate code, and the Java compiler compiles the program into ”byte code” that is run by an interpreter. There are purely compiled languages, of course (C, C++).

Generally, we consider a translation program that translates source code into machine code or some intermediate code to be a compiler if it needs to perform a semantic analysis of the source to produce the intermediate code. Java is compiled, Perl is interpreted, and BASIC is interpreted.


Front end
Source program (character stream)
Scanner (lexical analysis)
Token stream
Parser (syntactic analysis)
Parse tree
Semantic analysis and code generation
Abstract syntax tree or other intermediate form
Back end
Machine-independent code improvement
Modified intermediate form
Target code generation
Target language (e.g., assembly)
Machine-specific code improvement
Modified target language
Symbol table

The basic principles of both are very much the same. We will focus on compilation. See the slide for *phases of compilation*. There are six main phases, and they can be split into two groups of three: the front end and back end. The front end analyzes source code in its contextual form, and what you produce is an intermediate form (abstract syntax tree), which together with a symbol end, gets passed to a back end, turning this into actual machine code.

Lexical Analysis, Parsing, and Semantics

Lexical analysis involves the grouping of input characters into tokens (identifiers, keywords, numbers, ...). Extraneous characters are removed (spaces, tabs, newline characters) along with comments. Producing this grouping of characters is a fairly simple process (errors are not being looked for). This is the level of translation basic interpreters used to do. Very simple errors could be detected (such as $10a.4$).

NOTE The code for(i=0;i<10000000;i++) k++; when heavily optimized could become k = 10000000;

Parsing (a performing of *syntactic analysis* – determines whether it is syntactically correct) organizes tokens into a *parse tree* according to a grammar describing the language. The sequence of tokens are ensured to

Lexical Analysis, Parsing, and Semantics continued on next page. . .

conform to the syntax defined by the language grammar. This does not tell you if your code does what you think it will do. Some language rules are merely semantic (which cannot be caught until semantic analysis and code generation). A syntactically correct program only exists *iff* if the sequence of tokens conform to the syntax defined by the language grammar.

Semantic analysis generates *symbol tables* and *intermediate representations* of programs (e.g. syntax tree) from the parse tree.

- The symbol table maps each identifier to information known about it.
- The abstract syntax tree includes only important nodes of the parse tree but also stores annotations of these nodes (pointers from identifiers to symbol table entries).

This enforces rules are not captured by the context-free grammar (e.g. use an identifier only after it is declared). Finally, this generates code for run-time checks (e.g. array bounds) by removing irrelevant tokens. See a *sample Pascal program* on the slides and the process of scanning, parsing, and semantics. *Terminals* are tokens (characters of the language) and *non-terminals* are sentences. ϵ denotes an empty string.

Three separate processes are described here. These are typically run in parallel: your parser will ask, bit-by-bit, the next input from the scanner. The parser is what drives these two sequential processes in parallel. Furthermore, if you were to take your parse tree and turn it into an abstract syntax tree, you would not be able to do that alone (it just identifies identifiers). While you are parsing, you need to make sure information is kept around that you need later to produce the abstract syntax tree.

So... input code is a string of characters. It does not, as such, really have much information. The parser has to identify it. After scanning, you have a stream of tokens: nothing else but characters over a different alphabet. One character stream has translated into another one. Why then is there a separate process of scanning and parsing?

You can certainly create a parser without a scanner – for complicated languages, this is something you would not want to do. There is a level of expressiveness v. efficiency: you get a faster and simpler compiler when this is two different phases. In order to determine if it is a correct token or not, it has to look for the rules for creating valid floating point numbers, for instance. A smaller input (sequence of tokens) is having to be dealt with by a complicated process rather than a larger one. So, this is purely a *performance consideration.*

Lexical Analysis and Automata Theory

Formal Languages

The *motivation* behind this section of the course lies in the part of the process of compilation involved in lexical analysis.

A **formal language** L is a set of **strings** over an alphabet Σ. An alphabet Σ is a set of **characters** that can be used to form strings (letters, digits, punctuation, ...). Strings are finite sequences of characters. ϵ defines an empty string. The length of a string $|s|$ is the number of characters that string has. Σ^k is the set of strings of length k (see slide). A **Kleene star** $\Sigma^* = \Sigma^0 \cup \Sigma^1 ...$ is the set of all strings over alphabet Σ. A formal language L over alphabet Σ is a subset of the Kleene star (see slide).

Some infinite languages include $\{0\}$, $\{0,1\}^*$ (binary strings), $\{a,b,c\}^*$, the set of all positive integers, the set of all syntactically correct C programs, etc. A simple application of formal languages involves accepting valid e-mail addresses and rejecting invalid ones.

Formal Languages continued on next page...

Regular languages are those recognized (decided) by **finite automata** (very simple machines). They are useful for tokenizing program text (lexical analysis, scanner). **Context-free languages** are recognized by non-deterministic push-down automata (a finite automata that is a stack). They are useful for parsing the syntax of a program (syntactic analysis). This is why we stack our syntax check into two parts: *difference of complexity.*

We will talk about regular languages first.

Regular Languages & Expressions

A recursive definition is seen on the slide: an empty language, the empty string, and a where $a \in \Sigma$ are part of a regular language. If A and B are regular languages, then the following are also regular languages: $A \cup B$, $AB := \{ab | a \in A, b \in B\}$ (the concatenation of two strings in A and B), and A^*. See the slide for examples – important is that any finite language is regular.

NOTE If you take your finite language alphabet Σ and have n different characters in that alphabet, you can translate every given character into their own singular, trivial alphabets. If you take the union of all trivial alphabets, you can get a regular language.

Regular expressions are nothing but a more concise notation to represent regular languages. Operator precedence involves: (1) parentheses, (2) the Kleene star, (3) concatenation, and (4) union. For example, the regular expression that expresses the set of strings that do not contain 101 as a substring would be:

$$(0|\epsilon)(1|000^*)^*(0|\epsilon)$$

What are applications of regular expressions? Two common operations include searching for patterns in a text, or replace text portions matching a pattern. This is exactly what we do when we scan. It is used in text editors including emacs and vim, in system tools, and in programming languages.

In practice, there is no empty set symbol or ϵ character on the keyboard. The empty string is represented as an empty string. Recognizing the empty language is not very useful in practice. Additional notation to make it easier to expression common constructs exists. For instance: one or more repetitions of R, zero or one repetition of R, and exactly a given amount n or a range of repetitions. Some capabilities beyond regular languages include, for instance, recognition of strings of the form $\alpha\beta\alpha$ where $\alpha, \beta \in \Sigma^*$. This is most certainly not regular.

In practice, we can also employ *character classes.* Allow us to write tedious expressions such as $a|b|c|...|z$ more succinctly. "Recent" years can be written as: $199[6-9]|20(0[0-9]|1[0-3])$. There are also additional classes available in languages such as Perl (see slide).

Furthermore, we can employ *anchors.* These are used to match some characteristic positions between characters in the string. See the slide for examples. Some languages, including Perl, provide a host of other anchors.

Another useful feature in practice are *back references.* These match previously captured sub-expressions and **thus allow us to express certain kinds of non-regular languages**. See slide.

Lexical Analysis and Automata Theory continued on next page. . .

Finite Automata

The theory of regular languages (regular sets) was introduced by Stephen Cole Kleene (1909-1994) in the "Representation of events in nerve nets and finite automata" (1956). He used mathematical notion of **regular sets** to describe models of the nervous system by McCulloch and Pitts (1940s) as small simple automata.

A **deterministic finite automaton** (DFA) is a simple machine that **accepts** or **rejects** a given input string. This defines a formal language: the set of strings accepted by the DFA. We say the DFA *recognizes* this language. We will show that a language is regular *iff* there exists a DFA that recognizes this language.

The informal definition of a DFA involves:

- A finite set of *states*,
- a designated *start state*,
- a designated *set of final states*,
- reads input one symbol at a time (new state is calculated as a function of current state and read symbol),
- and a string is accepted if and only if a final state is reached after reading the entire string.

Therefore, reading a string involves going through a path of states. Once you reach the end of a string, it is accepted *iff* a *final state* is reached. See the graphical representation on the slides; also shown in a tabular form where one state is marked as a start one, and final ones are given an asterisk. The language that the first DFA recognizes is the following regular expression: $(0|1)*1$. See the slides for further examples.

The formal definition is not much different from the informal one. We say a DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$: Q is a finite set of states, Σ is a finite set of input symbols (an alphabet), δ is a transition function, q_0 is the start state, F is the set of final or accepting states.

Examples of formal DFAs follow. The one that forbids the substring 101 is a very common operation. In order to minimize the overhead of drawing it, can remove the final part of the diagram – if a certain transition that is forbidden and it is not found on the diagram, you reject the entire string. For many of them, you can write a regular expression for them. However, for example, it would be difficult to find one to find binary numbers divisible by 3.

A DFA is **deterministic** in the sense that every input traces exactly *one path* through the automaton. A **non-deterministic finite automaton** (NFA) is identical to a DFA except there are possibly many paths traced by an input. There are two sources for non-determinism: ϵ-transitions (without consuming input), or multiple successor states for the same input symbol. The latter is not necessary.

An NFA **accepts** a string $\sigma \in \Sigma^*$ if one on of the paths traced by σ ends in an **accepting state**. These are far more corresponding with regular expressions. However, note that we cannot build NFAs, so we have to convert them to DFAs and these are what we use as scanner codes in a compiler, for instance.

A NFA is a 5-tuple, like the DFA, except for two differences: one is found in the definition of the transition function (2^Q = set of all subsets of Q), and what it means to accept a given string (see slide).

Are NFAs more powerful than DFAs? The answer depends on what we mean by this question. Is there a language L recognized by an NFA that cannot be recognized by a DFA? No (this will be proved). Is it easier to construct an NFA for a regular language than to construct a DFA for the same language? Yes (see slide for example).

Theorem: The following statements are equivalent:

Finite Automata continued on next page...

- L is a regular language.
- L is the language described by a regular expression (without extension).
- L is recognized by an NFA.
- L is recognized by a DFA.

Therefore: *a language L is regular if and only if it can be expressed using a regular expression.* How do we prove this? See slide for the only 6 forms it could be and how these are translated. Extending this, we can also say: *if a language L can be expressed using a regular expression, there exists an NFA that describes it.* See slides. And the other way?

A **generalized NFA** (GNFA) is one where edges are labelled with regular expressions. When in configuration (q_1,$\alpha\beta$ — state q_1 with input $\alpha\beta$ still to be read), we can transition into configuration (q_2,β) *iff* the edge from the first state to the second is labelled with a regular expression that matches α.

The idea of going through a proof here is then going from one NFA to a regular expression through transitions to generalized NFAs. An NFA is a GNFA. "Normalize" it by adding $\epsilon - transitions$ from its final states to a new final state f and making all original final states non-final. *From GNFA to regular expression.* Firstly, transform GNFA into an equivalent GNFA with only two states: start and final state. Transform this two-state GNFA into a regular expression.

NOTE State reduction involves picking an arbitrary non-start and non-final state of a GNFA and removing it by connecting all pairs of in- and out-neighbours of that state. This may create loops because some states may simultaneously be in- and out-neighbours of the state.

From DFA to NFA. If a language L can be recognized by a DFA, it can be recognized by an NFA. **A DFA is an NFA**. The idea of proving this is to show how to construct a DFA whose states represent the sets of states the NFA may be in at any given point in time.

- *Start state.* Before consuming any input, the NFA can be in its start state q_0 or in any state that can be reached from q_0 using a sequence of $\epsilon - transitions$. We call this the **ϵ-closure** ECLOSE(q_0) of q_0. This is the start state of the DFA.
- *Transition function and construction of more DFA states.* Assume that after reading some input, the NFA can be in any of the states in a set Q represented by a DFA state. Which states can the NFA be in after reading an input symbol a? See slide for formula. We continue to inspect all DFA state-symbol pairs until we do not discover any new states.
- *Accepting states.* A state Q of the DFA is accepting if, viewed as a set of NFA states, it contains an accepting state of the NFA.

In the worst case, the construction may turn an NFA with n states into a DFA with 2^n states (every possible subset of NFA states become a DFA state). In practice, the worst case usually does not arise.

Scanning

A **scanner** produces a token (token type, value) stream from a character stream. We can consider it to have two modes of operation:

- Complete pass produces the *token stream*, which is then passed to the parser.
- Parser calls scanner to request next token.

Scanning continued on next page...

In either case, the scanner always recognizes the longest possible token.

Implementing a scanner can be done as:

- Hand-written, ad-hoc. Usually done when speed is a concern.
- From regular expressions using a *scanner generator* (turns it into a NFA and then into a DFA). More convenient. *Result*:
  - Case statements representing transitions of the DFA.
  - Table representing the DFA's transition function plus driver code to implement the DFA.

Therefore, building a scanner usually involves: regular expression $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ minimized DFA. Typically, DFAs just give yes or no answers. Extensions to a pure DFA include:

- *Accepting a token is not enough.* Need to know which token was accepted, its value. One accepting state per token type. Return string read along the path to the accepting state.
- *Keywords are not identifiers.* Look up identifier in keyword table (e.g. hash table) to see whether it is or not a keyword.
- *"Look ahead" to distinguish tokens with common prefixes* (e.g. 100 vs. 100.5). Try to find the longest possible match by continuing to scan from an accepting state. Backtrack to last accepting state when "stuck".

See *an incomplete scanner for Pascal* diagram on slides.

Minimizing the DFA and Constructing a Scanner

The idea here is that we want to group states into classes of equivalent states (accepting/non-accepting, same transitions). The procedure we follow is to:

- Initially, start with two equivalence classes: accepting and non-accepting states.
- Find an equivalence class C and a letter a such that, upon reading a, the states in C transition to states in $k > 1$ equivalence classes $C'_1, C'_2, ..., C'_k$. Partition C into subclasses $C_1, C_2, ..., C_k$ such that, upon reading a, the states in C_i transition to states in C'_i.
- Repeat until no such "partitionable" equivalence class C can be found.
- Final set of equivalence classes is the state set of the minimized DFA.

See slides for example. Once the DFA reaches q_4, the result is the same: there is a final accepting state and it will never leave this triangle of accepting states. Therefore, you can establish a *bipartition* of equivalence classes. Refinement follows.

Let us extend this to *constructing a scanner.* Given a language: strings of 0s and 1s containing an even number of 0s. The regular expression would be: $(1*01*0)*1*$. An NFA and corresponding DFA is found on the slides. A minimized DFA can be constructed – it is *unique.*

How general are regular languages? Can we use them to construct entire programming languages? For example, some properties are: if R and S are regular languages, then so are: RS, $R \cup S$, R^* by definition. Furthermore, so is the complement of R (build a DFA for it from a DFA for R by making accepting states non-accepting and vice versa), and $R^R = \sigma^R | \sigma \in R$, where σ^R is σ written backwards. See slides for more rules.

Minimizing the DFA and Constructing a Scanner continued on next page. . .

Recall, **not all languages are regular**! We want to have a way of formally proving a given language is not regular. Example: $L = \{0^n1^n | n > -1\}$ is not regular. You can assume, by contradiction, it is regular: $n =$ number of states of the DFA. Look at $0^{n+1}1^{n+1} \in L$. We will eventually go through three sets of characters: α, β, and γ where after γ you are in an accepting state. By this, it could also identify $\alpha\beta\beta\gamma$: this has more 0s than 1s.

This argument can be made general by the **Pumping Lemma**: for every regular language L, there exists a constant n such that every $\sigma \in L$ with $|\sigma| > n - 1$ can be divided into three substrings $\sigma = \alpha\beta\gamma$ with properties shown on the slides.

Proof of the Pumping Lemma. Have $D =$ DFA for L and $n =$ number of states of $D + 1$. The properties will follow out of a constructed DFA (see slides). Other examples follow.

Syntactic Analysis and Context-Free Grammars

Syntactic Analysis

A grammar for a subset of natural language is found on the slides. Note that the parser, which performs syntactic analysis, is not capable of being done by a finite automata. Sentences in the language described by this **context-free grammar** include:

- big Jim ate green cheese
- green Jim ate green cheese
- Jim ate cheese
- cheese ate Jim

Note that while all of them are syntactically true, they may not be semantically valid (see the last sentence). Note that from a parser's point of view, terminals are tokens produced by a scanner and are indivisible.

A **context-free grammar** is a 4-tuple (V, Σ, P, S) where V is a set of **non-terminals** or variables, Σ is a set of **terminals**, P is a set of rules or productions in the form $N \to (V \cup \Sigma)^*$ where $N \in V$ and $S \in V$ is the **start symbol**.

Notation includes using | to merge alternatives, show optional components with the subscript "opt", and regular expression forms (see slide). The **Backus-Naur** form allows for another way to represent $\to$ as ::=.

A **derivation** s a sequence of rewriting operations that starts with the string $\sigma = S$ and then repeats the procedure on the slides ($\sigma = \lambda X \rho \implies \sigma' = \lambda\alpha\rho$) until σ contains only terminals. Intermediate strings are called **sentential forms**. We write $S \implies^* \sigma$ to define a sequence of intermediates; every grammar G defines a language if:

$$L(G) = \{\sigma \in \Sigma^* | S \implies^* \sigma\}$$

If G is a context-free grammar, $L(G)$ is a **context-free language**. See slides for examples of such languages. Neither of the two examples are regular. **There are more context-free languages than regular ones** on the premise that all regular languages are context-free.

Every derivation can be represented by a **parse tree** where the root is S and the children of every node are the symbols (terminals and non-terminals) it is replaced with. The internal nodes are non-terminals and the leaves, the **yield** of the parse tree, are terminals.

Ambiguity

There are infinitely many context-free grammars generating a given context-free language: just add arbitrary non-terminals to the right-hand sides of productions and then add ϵ-productions for these non-terminals.

Therefore: there may be more than one parse tree for the same sentence generated by a grammar G. If this is the case, G is **ambiguous**. To allow parsing of programming languages, their grammars have to be unambiguous – these can be produced by adding more non-terminals. Some context-free languages are **inherently ambiguous** (do not have unambiguous grammars) — usually, this is not the case for programming languages.

For every sentence in a language defined by unambiguous grammar, there is only one parse tree that generates this sentence. There are *many* different derivations corresponding to this parse tree: you can do a left-most or right-most derivation (see slides).

Context-free grammars are used to formally describe the syntax of programming languages. Parsing a program involves finding the parse tree of the program. Context-free grammars for programming languages must be unambiguous and must capture the program structure. The parser should be efficient:

- Any context-free grammar can be used to derive a parser that runs in $O(n^3)$ time.
- We want linear time.

A context-free grammar is **right-linear** if all productions are of the form $A \to \sigma B$ or $A \to \sigma$ where σ is a (possibly empty) string of terminals. Similarly, it is **left-linear** if all productions are of the form $A \to B\sigma$ or $A \to \sigma$. A context-free language is **regular** if it is one of these two. **The set of languages expressed by regular grammars is exactly the set of regular languages**. **Regular grammars are too weak to express programming languages**.

Two kinds of grammars that can be parsed efficiently and are unambiguous are as follows. *Almost every programming language can be described by a LL(1)- or LR(1)-grammar (or both).*

An **LL(k)-grammar** can be scanned left-to-right and generates a leftmost derivation. If the first letter in the current sentential form is a non-terminal, k tokens look-ahead in the input suffice to decide which production to use to expand it. These are typically more intuitive.

An **LR(k)-grammar** can be scanned left-to-right and generates a rightmost derivation. The next k tokens in the input suffice to choose the next step the parser should perform. More classical; will not talk about them.

An **S-grammar** or **simple grammar** is a special case of an LL(1)-grammar. A context-free grammar is an S-grammar if:

- Every production starts with a terminal, and
- the productions for the same LHS start with different terminals.

See slides for examples. An example is an S-grammar for arithmetic expressions in *Polish notation*; sort of a stack notation in reverse. The grammar is very straight-forward. The process, using an LL-parser, used to create a parse tree and look ahead and parse is known as **recursive descent parsing**.

An equivalent example for LR parsing is *reverse Polish notation* – a simple stack language for arithmetic expressions. What happens is the parser will work backwards and make the parse tree in that fashion: productions are applied in reverse. This process is called **shift-reduce parsing**. The tree is built from left-to-right bottom-up. Two different operations are present: take the next symbol and push it on the stack or reduce elements on the stack.

Ambiguity continued on next page...

The crux with parsing deterministically using an LL(1) grammar is to decide which production to apply when the next symbol in the current sentential form is a non-terminal. There are two cases:

- $A \Longrightarrow \alpha \Longrightarrow^* a\beta$
- $A \Longrightarrow \alpha \Longrightarrow^* \epsilon$ and a derivation of A can be succeeded by a.

Intuitively (but formally not quite correctly), a terminal a is in the **predictor set** of production $A \rightarrow \alpha$ if $A\beta \Longrightarrow \alpha\beta \Longrightarrow^* a\gamma$ for some $\beta, \gamma \in \Sigma^*$. A grammar is LL(1) if the predictor sets of all productions with the same LHS are disjoint.

See slides on computing predictor sets: three kinds of sets (FIRST, FOLLOW, and PREDICT). The grammar shown is not LL(1) because the PREDICT sets are non-disjoint for A (see final PREDICT slide).

Note that there are exists context-free languages that do not have LL(1) grammars. There is no known algorithm to determine whether a language is LL(1) – but there is one to decide whether a grammar is LL(1). The "obvious" grammar for most programming languages is usually not LL(1). In many situations, a non-LL(1) grammar can be transformed into an LL(1) grammar for the same language.

Converting a grammar to LL(1). Two common reasons why a grammar is not LL(1) are "left recursion" and "common prefixes", both of which can be eliminated by modifying the grammar. See slide.

Left recursion can be replaced with **right recursion**. The side effect of this is that, often, left recursion is used intentionally to capture structure of the language (e.g., associativity of operators in arithmetic expressions). Conversions of such a form could remove this information. **Common prefixes** can be removed using left factoring: reduce to one production.

Slides following ("Converting a Grammar to LL(1): Example (1)") will show an example of arithmetic expressions. This grammar is not LL(1); repetitive PREDICT sets. When left recursion is removed ("Example (2)") by expansion into right recursion, an LL(1) grammar results.

Parsing LL(1) Languages

LL(1) languages can be parsed using efficient, easy-to-implement parsers. There are two approaches:

1. Recursive descent method.
2. Deterministic push-down automaton (DPDA).

Recursive descent parsing involves looking at each non-terminal X and writing a procedure parse-X: based on next tokens, choose a production and matches. See slides for an example. $ serves as an end-of-string token. If anything is invalid and matches are not made to the PREDICT set, then parsing fails.

For **pushed-down automata**, the motivation behind the method is that we proved that a language can be parsed by a finite automaton *iff* it is regular. We also proved that some context-free languages, including most programming languages, are *not* regular. Therefore, these finite automata are not expressive enough to parse context-free languages.

A pushed-down automaton (PDA) is an *NFA with a stack* and can solve this problem. See slide for an example of a PDA: the notation designates pushing certain elements onto a stack and checking for those elements later. PDAs *accept by empty stack*. There is a problem, however: we need to guess where the left half ends and the right half starts. These languages cannot be parsed deterministically. In particular, it is not LL(k) or LR(k) $\forall k$.

See the following slides for a formal definition of a PDA. There are two modes of acceptance:

Parsing LL(1) Languages continued on next page...

1. Accept by empty stack: accept *iff* it is possible to reach a configuration where input is completely consumed and stack is empty. State does not matter.
2. Accept by final state: accept *iff* it is possible to reach a configuration where input is completely consumed and current state is an accepting one. Stack does not matter.

The two modes of acceptance are equivalent: there exists a PDA recognizing a language L by empty stack *iff* there exists a PDA recognizing it by final state.

A language is context-free *iff* it can be recognized using a PDA.

By default, a PDA is *non-deterministic*. Multiple transitions are possible for a given combination of state, input symbol, and symbol on the top of the stack. If there is only one possible transition, for any combination of state, input symbol, and stack symbol, we call the PDA a **deterministic PDA** (DPDA).

More importantly, a language can be recognized by a DPDA *iff* it is LL(k) or LR(k). In particular, there are context-free languages that cannot be recognized by a DPDA:

$$\{\sigma\sigma^R | \sigma \in \{0,1\}^*\}$$

In the example DPDA, the starting state is the position where the cursor is ready to move on ("in-sync state"). All transitions to side states require no pushing of items onto the stack, but transitioning back does require taking the item off the stack now that it is there. Side states are remained in until we find the terminal we are looking for.

Implementation can be done in one of two ways:

1. Using nested case statements.
2. Table-driven.

Using nested case statements can be done with three levels of case statements: branch on current state, branch on current input symbol, and branch on current stack symbol. Some similarity to recursive-descent parsing. Instead of recursion, maintain explicitly.

Table-driven approach involve a loop using a 3D table mapping (state, input symbol, stack symbol) triples into strings to be pushed onto the stack. Much less space-consuming approach. Most parser generators do this.

Generating the parser is either hand-coded, or by automatic generation from grammar.

Semantic Analysis

Syntax and Semantics

Semantic analysis and code generation is the third major step in a typical compiler. Recall that **syntax** describes the form of a valid program and can be described by a context-free grammar. On the other hand, **semantics** describes the *meaning* of a program and *cannot* be described by a context-free grammar.

Some constraints that may appear syntactic are *enforced by **semantic analysis***. These include the use of an identifier only after its declaration. Semantic analysis enforces semantic rules, builds intermediate representation (e.g., an **abstract syntax tree** — AST), fills the **symbol table**, and passes its results to the intermediate code generator.

Syntax and Semantics continued on next page...

NOTE C has a single pass with both syntactic and semantic analysis to generate the entire symbol table for a given scope, and after that check you see what identifiers refer to in a later pass.

There are two approaches:

- interleaved with syntactic analysis,
- or as a separate phase.

The formal mechanism are **attributes grammars**. **Static semantic rules** are enforced by the compiler at compile time (e.g., do not use an undeclared variable). **Dynamic semantic rules** are rules that the compiler generates code for enforcement at run time. Examples include division by zero, array index out of bound, etc. Some compilers allow these checks to be disabled (if completely confident, makes sense to disable them — think very carefully when you disable these checks).

Attribute Grammars

Attribute grammars are *augmented* context-free grammars. Each symbol in a production has a number of *attributes*, and each production is augmented with semantic rules that:

- copy attribute values between symbols,
- evaluate attribute values using semantic functions,
- and enforce constraints on attribute values.

See slides for examples. The language below is not context-free but can be recognized by an attribute grammar.

$$\{a^n b^n c^n | n \geq 1\} = \{abc, aabbcc, aaabbbccc, ...\}$$

The way to do this is by counting the number of occurrences of different letters. **Synthesized attributes** involves attributes of LHS being computed from attributes of RHS. This is a *bottom-up approach.*

Inherited attributes is where attributes flow from left to right (from LHS to RHS and from symbols of RHS to symbols of RHS further to the right). This is a *top-down approach.* Information flows top-down and from symbols on the RHS to later ones; as well, we can allow a little bit of synthesization.

Attribute flow refers to the annotation or decoration of parse trees. It is a process of evaluating attributes. Synthesized attributes and inherited attributes are two ways of evaluating attributes (see slide for summary).

Therefore, there are two types of attribute grammars as a result of this:

1. S-attributed grammars,
2. L-attributed grammars.

S-attributed grammars are grammars where all attributes are synthesized and attributes flow bottom-up. **L-attributed grammars** involve a flow from left-to-right where for each production, every synthesized attribute depends on inherited ones and later inherited attributes depend on those before. Notice that *S-attributed grammars are a special case of L-attributed grammars* – not having any inherited attributes means you have only synthesized attributes.

Remember that the first grammar we wrote down for arithmetic expressions captured left-associativity (using left-recursion) and left-recursion is typically bad for LL(1) parsers because left-recursion grammars can never be LL(1). The given example on the slides captures left-associativity but is not LL(1).

Attribute Grammars continued on next page...

Using the trick we learned about, we eliminated left-recursion and an LL(1) grammar for the same language was formed. However, the parse tree created hangs to the right and does not really encapsulate sub-expressions (they are split over different sub-trees) or capture the concept of left-associativity. Therefore, we cannot easily synthesize attributes.

Action Routines

Action routines are instructions for ad-hoc translation interleaved with parsing. This is a more ad-hoc way of doing things; if certain productions of a rule are used, certain values are associating with symbols and an arbitrary piece of code is used to calculate them. Parser generators allow programmers to specify action routines in the grammar.

Action routines can appear anywhere in a rule (as long as the grammar is LL(1)); they can be any sort of code and can be immediately used to implement attribute grammars. Action routines are supported, for example, in *yacc* and *bison*.

Names, Scopes, and Binding

Binding

A **name** is a mnemonic character string representing *something else*; for example,

- `x`, `sin`, `f`, `prog1`, `null?` are names,
- 1,2,3, ”test” are not names
- +,-, ... may be names if they are not built-in operators.

A **binding** is an association between two entities: for instance, between name and memory location (for variables) or between name and function. Typically, a binding is between a name and the object it refers to.

A **referencing environment** is a complete set of bindings active at a certain point in a program. The **scope of a binding** is the *region* of a program (or time interval(s)) in the program's execution) during which this binding is *active*. A **scope** is a maximal region of the program where no bindings are destroyed — for instance, the body of a procedure.

When we start talking about binding, we need to know at which times they are defined. Four times include:

1. *Compile time*: map high-level language constructs to machine code, layout static data in memory.
2. *Link time*: resolve references between separately compiled modules (if you have different pieces of source code using definitions to other pieces of code, these can only be resolved here).
3. *Load time*: assign machine addresses to *static* data.
4. *Run time*: bind values to variables, allocate dynamic data and assign to variables, allocate local variables of procedures on the stack.

Early binding produces faster code and is typical in compiled languages. **Late binding** provides greater flexibility and is typical in interpreted languages – take as long as possible to commit.

Object lifetime refers to the period between the creation and destruction of a given object. For example, the time between creation and destruction of a dynamically allocated variable in C++ using `new` and `delete`.

Binding continued on next page...

Similarly, **binding lifetime** refers to the period between the creation and destruction of a *binding* (name-to-object association).

Two common mistakes include:

- *Dangling reference*: no object for a binding (e.g., a pointer refers to an object that has already been `delete`d). What is used by *garbage collection* in order to know when to deallocate memory.
- *Memory leak*: no binding for an object (preventing it from being deallocated).

An object's lifetime corresponds to a mechanism used to manage the space where the object resides – **storage allocation** – where the object is being stored.

1. *Static object*: objects stored at a fixed *absolute* address, object's lifetime spans the entire execution of the program – somewhere in the program address space.
2. *Object on stack*: object allocated on stack in connection with a subroutine call, object's lifetime spans period between invocation of the subroutine and return from the subroutine – usually the local objects of certain subroutines.
3. *Object on heap*: object stored on heap, object created/destroyed at arbitrary times (explicitly by the programmer or implicitly by a garbage collector).

An example of all of this is found in C++. See slide.

Memory Management

Static objects can include:

- **global variables**,
- static variables local to subroutines that retain their value between invocations (it gets initialized at the time it is first accessed),
- constant literals,
- tables for run-time support: debugging, type checking, etc. (*v.* C which requires a system like `gdb`),
- and spaces for subroutines, *incl.* local variables in languages that do not support recursion (e.g., early versions of FORTRAN).

Stack-based allocation is where a stack is used to allocate space for subroutines in languages that permit recursion. The **stack frame** (**activation record**) stores:

- arguments and local variables of the subroutine,
- the return value(s) of the subroutine,
- the return address, ...

The subroutine **calling sequence** maintains the stack: before a call, the caller pushes arguments and return address onto the stack; after being called (prologue), the subroutine ("callee") initializes local variables, etc.; before returning (epilogue), the subroutine cleans up local data; and after the call returns, the caller retrieves return value(s) and restores the stack to its state before the call.

Let's focus on the stack frame. Stack frames are a perfect example of the tradeoff of early binding. Stacks have a fixed structure — they have a certain list of arguments, return address, local variables, etc.

Memory Management continued on next page...

The *compiler* determines the **frame pointer**: a register pointing to a known location within the current stack frame. Offsets from the frame pointer specify the location of objects within that stack frame. The absolute size of the stack frame may not be known at compile time (e.g., variable-sized arrays allocated on the stack — pointers are kept and the memory is stacked on top of these).

A **stack pointer** is a register pointing to the first unused location on the stack (used as the starting position for the next stack frame). Specified at *runtime* is the absolute location of the stack frame in memory (on the stack) and the size of the stack frame. It is not bound at compile at time as it can vary during runtime.

The opposite technique to do this is to use a dictionary: it is unnecessarily slow. On the other hand, we can bind everything as early as possible to maximize efficiency – that is what is done above.

Heap-based allocation involves the **heap**, which is a region of memory where blocks can be allocated and deallocated at arbitrary times and in arbitrary order. They can vary with time and cannot be predicted in advance.

Heap management involves a **free list** (lists of blocks of free memory; done in one linked list) and an *allocation algorithm* which searches for a block of adequate size to accommodate the allocation request.

General allocation strategy involves:

- find a free block that is at least as big as the requested amount of memory (if this is not possible, the memory allocation *fails*),
- mark requested number of bytes (plus padding) as allocated,
- and return the rest of the free block to the free list.

This leads us to two different allocation methods: first-fit and best-fit allocation.

First-fit allocation finds the *first* block large enough to accommodate the request. This is, on average, twice as fast as the next method (usually look at half of the list) **Best-fit** finds the smallest block large enough to accommodate the allocation request (traverse the entire list; has an overhead). The advantage of this method is that it does not split big blocks unless as it has to because there may be desire for such big blocks later on.

However, we have a possible problem: the **heap fragmentation** problem.

- **Internal fragmentation** is the idea that often only blocks of certain sizes (e.g. 2^k) are allocated and this may lead to part of an allocated block being unused.
- **External fragmentation** is the idea that unused space may consist of many small blocks. Therefore, although the total free space may exceed the allocation request, no block may be large enough to accommodate it.

Neither best-fit nor first-fit is guaranteed to minimize external fragmentation. Which strategy is better depends on the size distribution of the allocation requests. Both of them use this walking of a linked list. Something else can be done! So, what is the cost of allocation on a heap? If a single free list is maintained, there is linear cost to find a block to accommodate each allocation request.

However, another system can be imagined: the **buddy system**. In this system, a binary tree is conceptually built across the entire memory: blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, ...$ are built out of the memory space. Blocks are built up until it is one large whole – a conceptual binary tree. A separate free list is maintained for each individual block size.

With a given request, it is rounded to the next power of 2. If blocks of size the requested size is available, split block of size 2^{k+1}. If block of size 2^k is deallocated and its buddy is free, merge them into a block of size 2^{k+1}. The worst-case cost of this is log(memory size).

Memory Management continued on next page. . .

Using block sizes that are Fibonacci numbers (1, 1, 2, 3, 5, 8, 13, 21, 34, ...) for each given level (a ratio between each level according to *the golden ratio*, φ). This incurs less fragmentation.

Deallocation on a heap occurs can occur explicitly by the programmer (used in Pascal, C, C++, ...) which is typically efficient and may lead to bugs that are difficult to find (dangling pointers, memory leaks). It can also occur automatically by a garbage collector (used in Java, functional and logical programming languages, ...), and this can add significant runtime overhead but is safer.

Scopes

The **scope of a binding** is the region of a program or time interval(s) in the program's execution during which this binding is active; furthermore, a **scope** is a maximal region of the program where no bindings are destroyed (e.g., the body of a procedure).

Lexical (static) scoping is binding based on the nesting of blocks. It can be determined at compile time. On the other hand, **dynamic scoping** is a binding that depends on the flow of execution at runtime and can be determined only during runtime.

For lexical scope, the current binding for a name is the one encountered in the smallest enclosing lexical unit. Lexical units include:

- packages, modules, source files,
- classes,
- procedures, nested subroutines,
- blocks,
- records, and structures.

A variant to this is that the current binding for a name is the one encountered in the smallest enclosing lexical unit and preceding the current point in the program text. Examples of this is in C, Java, Prolog, Scheme, ...

NOTE In Scheme, code blocks are surrounded by (and). `(lambda(args) body)` are anonymous functions. Local variables are defined using `(let ((name exp)(name exp) body)` and only visible after that list. `let*` removes this limitation (only after the list). `letrec` allows variables to be visible regardless of the list. `define` declares variables as they are needed. `define f (lambda (...)  ...)` can be reduced to `define (f ...)`. List literals are denoted by `'(...)`.

Lexical scoping is implemented through the use of **static chains**. The stack frame of each invocation has a static link to the stack frame of the most recent invocation of the lexically surrounding subroutine. To reference a variable in some outer scope, the chain of static links is traversed, followed by adding the offset of the variable relative to the stack frame it is in.

NOTE At compile time, you can determine that a procedure is nested N levels deep inside another procedure.

Dynamic scoping differs from static scoping; the current binding for a given name is the one:

- encountered most recently during execution,
- not hidden by another binding for the same name,

Scopes continued on next page...

- and not yet destroyed by exiting its scope.

For example:

```
# static scoping
sub f {
    my $a = 1;
    print "f:$a\n"; # prints 1
&printa();
}

sub printa { print "p:$a\n"; } # prints 2

$a = 2;
&f();

# dynamic scoping
sub g {
   local $a = 1;
   print "g:$a\n"; # prints 1
   &printa();
}

sub printa { print "p:$a\n"; } # prints 1

$a = 2;
&g();
```

Dynamic scoping is both less efficient (in terms of runtime cost; have to search all frames for declarations) and more confusing. This is why it is not really used these days anymore. It is **usually a bad idea** — runtime costs could be justifiable if we could convince ourselves it allows us to write better code; but in this case, it does not.

NOTE The only way you can track variables between frames with dynamic scoping is using hash map or dictionaries. Lookups may fail, etc. repeats process down levels. Cannot avoid linearly walking. Do not know where a procedure is called, typically.

We can also make a distinction between **shallow binding** and **deep binding**. If a subroutine is passed as a parameter, when are the free variables (still need to find a binding; found inside the procedure) bound?

NOTE Scheme uses deep binding.

In shallow binding, this occurs when the routine is called. In deep binding, this occurs when the routine is first passed as a parameter. This is important using both static and dynamic scoping and is known as **the funarg problem**. Deep binding has the same advantage for static scoping: you can intuitively tell. See slides for example of the funarg problem — a fancy name for figuring out when to bind things.

Names, Scopes, and Binding continued on next page...

Closures

When using deep binding, a **closure** is a *bundle* of a **referencing environment** and a **reference to the subroutine** — a "hidden argument". Deep binding is also the default in statically scoped languages and an option in dynamically scoped languages. Dynamic scoping is not really in much use anymore.

Closures become really interesting when returning functions as the result of function calls: when the function is invoked, the scope it refers to may no longer exists (thus needs to be preserved explicitly in the closure) and this can be used to implement "poor man's objects".

Here's an example.

```
b() {
   int x;
   f() {
      g() {
         x = 1;
      }
      h(g);
   }
   f();
}

h(g) {
   j(g);
}

j(g) {
   g();
}
```

In this example, `g` would be passed along with a pointer to `f`'s stack frame.

But how about this situation?

```
b() {
   int x;
   f() {
      g() {
         x = 1;
      }
      return g;
   }
   y = f();
   h(y);
}

h(g) {
   j(g);
}

j(g) {
```

Closures continued on next page. . .

```
   g();
}
```

The stack frame associated with `f` will not exist when `h` is called. This leads us to the idea of **frames**. A **frame** is a collection of variable-object bindings. Frames can point to "parent frames" which allows the construction of a chain frames (equivalent to the idea of static frames). A **referencing environment** is a chain of frames represented by pointing to the first frame in the chain. A variable `x` in an environment is unbound if none of the frames bind it; otherwise its value is the value bound in the first frame that provides such a binding.

Therefore, a **closure** is the code of a function paired with a pointer to a referencing environment. Whenever a function is invoked, it prepends a new frame to this environment. Now we can see why closures SIMULATE OBJECTS to a degree: the nature of the closure allows for invocations of frames and environments. Note there is no inheritance, etc. unless explicitly defined.

NOTE The whole issue of closures only came about because we may want to call upon them outside of their lexical contexts – in order to do the right thing, we have to pass the lexical context along with it. We can look at three classes of objects: first-, second-, and third-class objects where there are varying requirements of returning. Only if procedures are first-class objects (can return them as the part of procedures) does this become a problem; lifetimes has to be decoupled.

In general, implementing closures has an associated cost; at least in the way Scheme treats them, the lifetime of a closure is no longer bound to the lifetime of the procedure that first created them — things that were previously stack frames before become frames that are on the heap (undefined time to live).

Modules

The motivation of **modules** is something to enable programmers in a team to work independently on different parts of a project; hiding details of different parts of a project. Requirements are these modules need to interact with each other through *well-defined interfaces*, and internals of modules should be hidden from other modules to avoid unwanted coupling.

This leads to the transition from static variables to **classes**. Static variables (in C) provide "private objects" to a single subroutine.

Modules provide the same set of "private objects" to a *group* of subroutines (essentially a single instance of a class). **Module types** can be instantiated, effectively acting like classes but without inheritance — a data type that ties data and procedures together; can create objects of these type and provoke routines of these type. Every instance of a module type or class has a separate copy of the module type's or class's variables.

Classes add inheritance to module types; therefore, modularity can still be done *without* object-oriented programming. Inheritance, the ability to specify the behaviour once and having that be inherited, has nothing to do with code reusability or modularity.

See proceeding slides for examples (e.g. of static variables, modules). The static variables persist and exist through the given function calls.

NOTE C has no support for modules, but it does have the ability to have compilation units with header files and linking. Java, C#, Perl, Python, Ada, and Haskell provide ***selectively*** **open scopes** — you can choose whatever you want; how this works depends on the syntax of the language.

Modules continued on next page...

Modules in MODULA-2 handle visibility through the use of explicit `IMPORT` (takes information out of the context upon which it is called) and `EXPORT` statements. This is an example of **closed scope** (where you need to specify what you want explicitly as a programmer), as opposed to **open scopes** where bindings from ”outside” can freely pass into the scope. Closed scope force programmers to clearly document the interface.

Constructs *similar* to modules are separate compilation units in C (judicious export of variables and functions in include files can simulate `EXPORT` lists but there is no protection against name clashes between modules), namespaces and separate compilation units in C++, packages in Java, Perl, and Ada, and clusters in Clu.

A deficiency in modules includes the example that the stack module cannot be used to provide multiple stacks to an application that requires them. Solutions include:

- possibly duplicating the code over multiple modules with the same name (*not really a solution*),
- modules that provide explicit means to create/manage/destroy multiple stacks (requiring the stack as an argument to each stack function),
- module types, or
- going all the way to classes (there is no difference, really, considering you still pass instances along as hidden parameters).

Every instance of a module type or class has a separate copy of the module type's or class's variables. Classes are *module types + inheritance*.

Aliasing and Overloading

Aliasing refers to having more than one name bound to one object (references, pointers, ...). This makes compiler optimization *more difficult if not impossible* — what a compiler may want to do could be incorrect. In the given example, if the `a` and `b` assignments were combined, there is a problem if the two pointers `p` and `q` are pointing to the same memory location.

Aliases make code more confusing and make resulting bugs hard to find. The `restrict` keyword in C99 is usable by a programmer to tell the compiler that a given pointer is the only means used to update the memory location it references. Resulting optimization may lead to even more obscure bugs if the programmer does not respect this promise.

Overloading refers to having one name bound to more than one object. Most languages have some form of overloading (e.g., arithmetic operators) — we do not think about this type of overloading normally because we think of ”doing math with numbers” and the right thing happens: there can be two different versions for integers or floating points. What is normally thought of is user-defined overloading.

This type of user-specified overloading makes a language *more powerful* and *expressive* with a potential to *increase clarity* (e.g., arithmetic operators for complex numbers *or* string concatenation). It does, however, also have the potential for tremendous confusion if the behaviour associated does not match what one would intuitively expect.

Mechanisms related to overloading include:

- **Coercion**: the compiler automatically converts an object into an object of another type when required (**type conversion**). Example: `"" + o` in Java. There are many situations where coercion is sort of a hack.
- **Polymorphism**: creating a single body of code (that does not need to be duplicated) where behaviour is customized depending on the type of argument. The given code takes a list of `a`s and turns it into

Aliasing and Overloading continued on next page. . .

an `a`. If empty, the result is 0. Otherwise, it is a summation of elements (referred to by reference and dealt with uniformly).

- **Generics**: separate *copies* of the code generated by the compiler for *each type*. Defining a template for a given class type. The advantage of using this is that the compiler can actually optimize the generated code for the determined data type.