CS541 class notes

Raphael Finkel

November 24, 2020

1 Intro

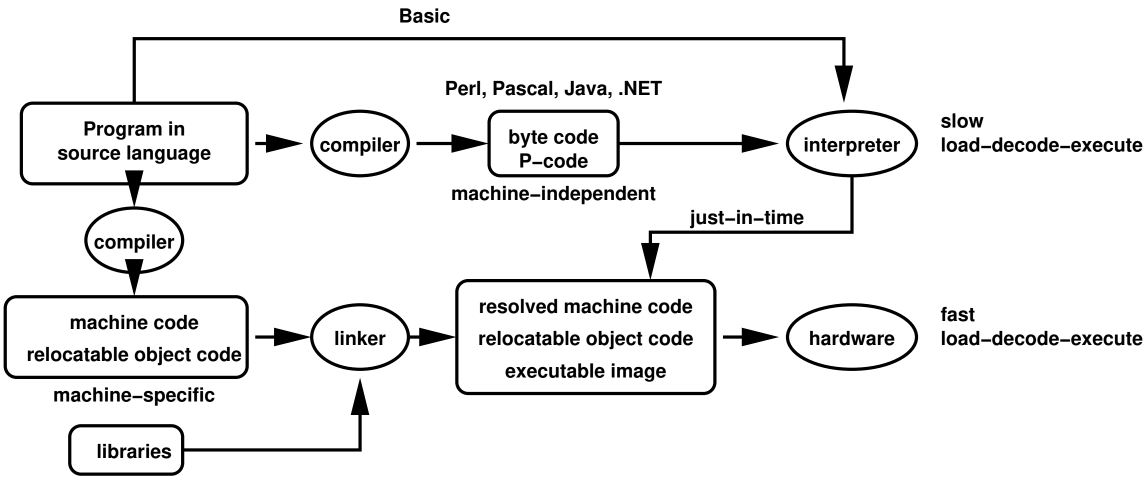
- $\boxed{\text{Class 1, 8/18/2020}}$
- Handout 1 — My names
  - Mr. / Dr. / Professor / —
  - Raphael / Rafi / Refoyl
  - Finkel / Goldstein
- Plagiarism — read aloud from handout 1
- Assignments on web. The first is very easy, the rest not, so start immediately.
- E-mail list: `cs541001@cs.uky.edu`; instructor uses to reach students.
- All students have MultiLab accounts, although you may use any computer you like to do assignments.
- Textbook — It is important that you read ahead.
- Undergraduates — Send me email; grading is 5% more lenient.

2 Overview of compilers: Chapter 1

- A compiler language is an example of a **software tool**.

$$\frac{\textbf{Use (client)}}{\textbf{Implementation}}\ \textbf{Spec} \Longrightarrow \frac{\textbf{Programmer}}{\textbf{Compiler}}\ \textbf{Language}$$

- The compiler's job.

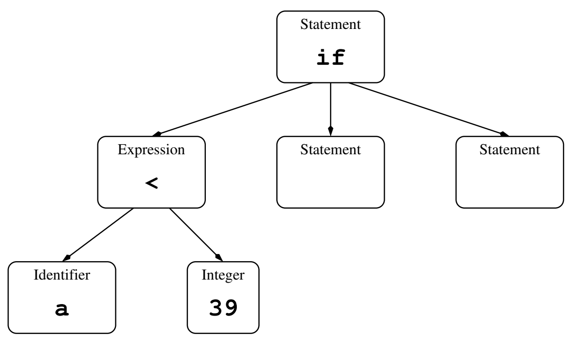


- Compiler outputs
  - Pure machine code: specific to a given architecture, no runtime linking. Example: Linux kernel.
  - Augmented machine code: specific to a given architecture and operating system. Example: C programs written for Linux, which may make OS calls.
  - Virtual machine code, interpreted or compiled on the fly during execution. Examples: Java (JVM), C# (.NET). Advantages: portability, code size **Our assignments use this output type.**
- Output representations
  - Assembler: good for cross-compilation; avoids having the compiler resolve all references. Modular compilation. **Our assignments use this output format.**
  - Relocatable binary: defers resolving external references. Modular compilation. Very common; used by Java and C.
  - Absolute binary: all references resolved.

3 The organization of a compiler

- Figure 1.4 page 15

- Scanner: reads the source program and constructs a stream of tokens, removing comments, and processing directives such as listing.
  - Example: `if (a < 39) {` is an input string of characters. The associated output tokens are **`if`**:reserved, `(`:symbol, `a`:identifier, `<`:operator, `39`:integer, `)`:symbol, `{`:symbol.
  - The scanner can discover and report errors, such as `39f`.
  - We describe **tokens** by regular expressions.
  - We recognize tokens by using a deterministic finite automaton (DFA). That automaton is built for us by a **scanner generator** tool such as *lex*, *flex*, or *jflex*. **Our assignments use *jflex*.**
- Parser: reads the token stream and creates an abstract syntax tree (AST), verifying syntax and possibly repairing syntax errors.
  - Example: given the tokens above, the tree fragment would be:



  - The parser can discover and report errors, such as `]` instead of `)` in the example.
  - We describe the syntax by a context-free grammar (CFG).
  - The table that drives the scanner is built for us by a **parser generator** tool such as *yacc*, *bison*, or *javaCUP*. **Our assignments use *javaCUP*.**
- Semantics checker: navigates through the AST and verifies that variables are declared, that types are used consistently, and that other semantic constraints (reachability, consistent use of exceptions) are met.
  - For instance, if `a` in the example is not of a numeric type, the type checker can report an error.

  - It can also modify the AST, for instance, introducing type-conversion nodes, if, for instance, `a` is a short integer, in which case it might be converted to a regular integer.

- Code generator: navigates through the AST and generates either an intermediate representation (IR) or some other representation of executable code. **Our IR will be assembler for Java bytecode.**
- Optimizer: Analyzes the IR to improve the code. There are many forms of optimization, such as simplifying expressions, moving code, re-using values, eliminating trivial arithmetic, replacing sequences of instructions. **We will not cover optimization in this class.**
- Code generator: Maps the IR to target machine code. **Our assignments use *Jasper* to generate the target machine code: Java bytecode.**

4 Programming language considerations

- $\boxed{\text{Class 2, 8/20/2020}}$
- Successful designers of programming languages often have strong backgrounds in constructing compilers. If it can't be compiled, it's not very useful.
- Many features of modern languages require special care.
  - passing by name (obsolete since Algol 60; requires thunks)
  - dynamic-sized arrays (requires runtime type descriptors)
  - nested name scopes (require static chains)
  - anonymous functions, first-class functions (as in Python, requiring closures)
  - multiple-yield iterators (as in Python, require special stack manipulation)
  - automatic reclamation of object store (requires garbage collection).

5 Computer architecture considerations

- How many registers? What operations use them? How many register classes?
- Some operations can be very expensive: virtual method dispatch, dynamic heap access, reflective programming, exceptions, threads.
- The effect of memory architecture, such as paging and caches, is difficult to present to programmers but is significant.

6 Specialty compilers

- Debugging support, including participation in an integrated development environment (IDE).
- Highly optimizing compilers.
- Retargetable compilers.

7 The ac (adding calculator) language: Chapter 2

- This is a very simple language that lets us explore the components of a compiler.
- Components
  - Types: integer and float
  - Keywords: **`f`**, **`i`**, **`p`**
  - Variables: lowercase Roman single letters, excluding keywords
- Context-free grammar (CFG), expressed in Backus-Naur Form (BNF) Figure 2.1 page 33

```
1  Prog  → Dcls Stmts $
2  Dcls  → Dcl Dcls
3        | λ
4  Dcl   → floatdcl id
5        | intdcl id
6  Stmts → Stmt Stmts
7        | λ
8  Stmt  → id assign Val Expr
9        | print id
10 Expr  → plus Val Expr
11       | minus Val Expr
12       | λ
13 Val   → id
14       | inum
15       | fnum
```

Figure 2.1: Context-free grammar for ac.

- Parse tree for **`f`** `b` **`i`** `a a = 5 b = a + 3.2` **`p`** `b $` $\boxed{\text{Figure 2.4 page 37}}$

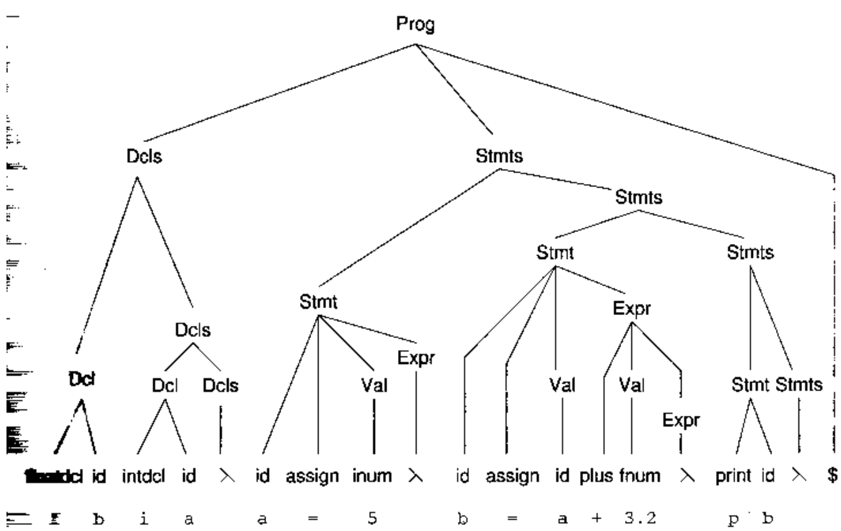


Figure 2.4: An ac program and its parse tree.

8 The scanner

- Translates a stream of characters (as above) into a stream of tokens.
- A token has a type (such as *operator* or *reserved*) and a semantic value (such as *plus* or *print*).

- It's a matter of choice whether each operator has its own type, in which case there is no need for semantic values.
- Likewise, one can choose (1) reserved words each have their own type, or (2) they are of type *reserved* with a semantic value (their spelling), or (3) that they are of type *id* with a semantic value.
- Class 3, 8/25/2020
- Hard-coded example Figure 2.5 page 40 uses `peek()` and `advance()`

```
Token scanner(Stream<char> cs) throws LexicalException {
   while (isSpace(peek(cs)) advance(cs);
   if (eof(cs)) return(eof);
   if (isDigit(peek(cs))) return(scanDigits(cs));
   char c = advance(cs);
   switch (c) {
      case {'a' .. 'z'} - {'i', 'f', 'p'}:
         return(Token(id, c));
         break;
      case 'f': return(floatDecl); break;
      case 'i': return(intDecl); break;
      case 'p': return(print); break;
      case '=': return(assign); break;
      case '+': return(plus); break;
      case '-': return(minus); break;
      default: throw LexicalException;
   } // switch
} // scanner

Token scanDigits(Stream<char>cs) {
   // the returned value is a string.
   Token answer = Token(inum, "");
   while (isDigit(peek(cs))) answer.value += advance(cs);
   if (peek(cs) != '.') return(answer);
   answer.type = fnum;
   answer.value += advance(cs);
   while (isDigit(peek(cs))) answer.value += advance(cs);
   return(answer);
} // scanDigits
```

- Production-quality scanners are constructed automatically from regular expressions. We will discuss them in the next chapter.
- This parse requires that we specify the syntax of **tokens**.
  Figure 2.3 page 36

Terminal	Regular Expression
floatdcl	"f"
intdcl	"i"
print	"p"
id	[a − e] | [g − h] | [j − o] | [q − z]
assign	"="
plus	"+"
minus	"-"
inum	$[0-9]^+$
fnum	$[0-9]^+ . [0-9]^+$
blank	$(" ")^+$

9 Formal language hierarchy

Language type	Formalism	Automaton
Regular	Regular expressions	Finite-state automaton (FSA)
Context-free	CFG (like BNF)	Push-down automaton (PDA)
Context-sensitive	CSG	Linear-bounded automaton (LBA)
Type 0	various	Turing machine

10 The parser

- Translates a stream of tokens into an **abstract syntax tree (AST)**
- The simplest method is **recursive descent**. Each nonterminal has its own procedure. By looking ahead (using `peek()`), each procedure can decide which other procedures to call.
- Parsing statements in *ac*: Figure 2.7 page 42

```
void stmt(Stream<Token> ts) throws ParserException {
   if (peek(ts) == id) {
      match(ts, id);
      match(ts, assign);
      val();
      expr();
   } else if (peek(ts) = print) {
      match(ts, print);
      match(ts, id);
   } else {
      default: throw ParserException;
   }
} // stmt
```

- One needs to discover the **predict sets** for each alternative production that has the same left-hand side. For `Stmt`, the predict set for assignment is {id}.
- One needs to discover the **follow sets** for some productions that can derive λ in order to compute the predict set for their parent productions.
- Class 4, 8/27/2020
- Given the grammar in Figure 2.1 page 33, notes p. 6, trace the parse of

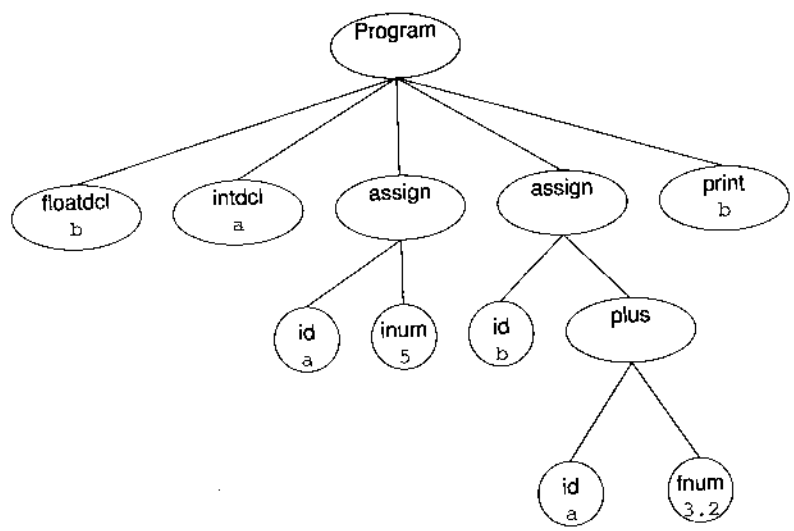
  `f` `b` **`i`** `a a = 5 b = a + 3.2` **`p`** `b`

11 What scanning and parsing cannot do

- enforce strong typing constraints
- disambiguate the meaning of some constructs, like `x.y.z` in Java, which might be package-class-field or variable-field-field or many other possibilities.
- determine the meaning of an overloaded operator.

12 Abstract syntax trees

- Instead of using the parse tree, we prefer an abstraction of the parse tree: the **abstract syntax tree**.
- It omits punctuation.
- Declarations store the identifier and its type in a single node.
- It represents the order of executable statements and expressions.
- Assignment nodes have two children: the identifier (the left-hand side) and the expression (the right-hand side).
- Binary operations have two children.
- The **`print`** statement is a single node that includes the name of the identifier to be printed.
- Compare the parse tree Figure 2.4 page 37, notes p. 6 with this AST (Figure 2.9 on page 44)



- More appropriate in a Java implementation: the program has two children, Decls (declared to be a `List` of declarations) and Statements (declared to be a `List` of statements).

13 Semantic analysis

- Construct a **symbol table** for declarations and name scopes. In the case of ac, it can be very simple: an array indexed by `'a' .. 'z'`. Each element has a `type` field, initialized to `unknown`.
- Enforce type consistency.
  - Walk the tree recursively, using **visitor methods** as shown in Figure 2.12 on page 49.
  - Insert to and query the symbol table as necessary.
  - Modify the `type` field to nodes as a declaration is visited.
  - Modify the AST to introduce type conversion (in our case, **widening**) nodes.

```
class Declaration {
   Id id; Type type;
   void check() {
      Symb symb = lookup(Id.name);
      if (symb != null) error("redeclaration");
      insert(Id.name, type); // put new symb in symbol table
   } // check
} // Declaration
class Expr {
   Type type;
   abstract void check();
} // Expr
class Operation extends Expr {
   Expr op1, op2;
   void check() {
      op1.check();
      op2.check();
      if (op1.type == op2.type) {
         // no conversion
      } else if (op1.type == int) {
         op1 = new ToFloat(op1);
      } else {
         op2 = new ToFloat(op2);
      }
      type = op1.type;
   } // check
} // Operation
class Id extends Expr {
   char name;
   void check() {
      Symb symb = lookup(name);
      if (symb == null) error("undeclared␣variable");
      type = symb.type;
   } // check
} // Id
```

14 Generating code

- $\boxed{\text{Class 5, 9/1/2020}}$
- In our case, the code is calculator buttons.
  - The calculator has registers; each is a single letter, such as **a**.
  - One can load or store a register with the **L** and **S** buttons.
  - One sets the precision with the **K** button.
  - One prints with the **P** button.
- We visit the AST recursively to generate code, invoking `codeGen()` at each node. $\boxed{\text{Figure 2.14 on page 52}}$

```
class Program {
   List<Declaration> Decls;
   List<Statement> Statements;
   void codeGen() {
      for (Statement statement : Statements) {
         statement.codeGen();
      }
   } // codeGen
} // Program
class Assign extends Statement {
   Id lhs; Expr rhs;
   void codeGen() {
      rhs.codeGen();
      emit("S"); // store
      emit(lhs.name);
      emit("0␣K"); // to integer mode
   } // codeGen
} // Assign
class Operation extends Expr {
   Expr op1, op2; char operation;
   void codeGen() {
      op1.codeGen();
      op2.codeGen();
      emit(operation);
   } // codeGen
} // Operation
class Id extends Expr {
   char name;
   void codeGen() {
      emit("L"); // load
      emit name;
   } // codeGen
} // Id
class Constant extends Expr {
   String value;
   void codeGen() {
      emit(value);
   } // codeGen
} // Constant
class ToFloat extends Operation {
   Expr operand;
   void codeGen() {
      operand.codeGen();
      emit("5␣k"); // 5 significant figures
   } // codeGen
} // ToFloat
```

- Trace code generated for the AST on p. 10. See $\boxed{\text{Fig 2.15 p. 53}}$.

15 Overview of scanner: Chapter 3

- This chapter introduces a formal, systematic approach to building scanners, instead of the hard-coded version of Chapter 2.
- Short story: tokens are defined by regular expressions, which are encoded into a non-deterministic finite automaton (NDFA), which can be automatically converted to a deterministic finite automaton (DFA), which can be described as a table of state$\times$input$\rightarrow$action$\times$state, which can be executed by a simple program.
- Shorter story: write a set of regular expressions and let a **scanner generator** do the rest of the work. This method is an example of **declarative programming**.
- There are some complexities.
  - Escaped double-quote within a string literal.
  - Over-eagerness leading to error, such as `3..4` in Pascal, or `'a'` in Ada, or `DO 200 I = 1.10` in Fortran
  - The scanner needs to be very fast. Scanning tends to be the most time-consuming step of compilation, partly because of the cost of reading the source code (with all its inclusions).

16 Regular expressions

- This material should be a review.
- A regular expression defines a **language**, which is a set (possibly infinite) of strings over some **alphabet** Σ.
- A regular expression is built recursively on the following components.
  - $\varnothing$.
  - λ.
  - individual letters in Σ. Example: `a`
  - the concatenation of regular expressions. The concatenation operator is usually omitted. Example: `abca`.

  - the alternation of regular expressions. The alternation operator is written $|$.
  - closure operations: the **Kleene closure** * and the **positive closure** $^+$.
  - parentheses for grouping.
  - If you want to use a metacharacter such as $|$, *, $^+$, `(`, `)` in a regular expression, use some sort of escape character (typically \) before it.

- A regular expression generates a set of strings. That set is called the **language** generated by the regular expression.

  - $\varnothing$ generates no strings at all.
  - λ generates the empty string.
  - An individual letter generates the string containing just that letter.
  - The concatenation of two regular expressions A and B generates all two-part strings, whose first part is a string generated by A and whose second part is a string generated by B.
  - The alternation of two regular expressions A and B generates all strings generated by A and all strings generated by B.
  - The expression A^* generates the empty string and (recursively) all strings generated by AA^*. The expression A^+ generates all strings generated by AA^*.

- Useful facts

  - The set of strings generated by a regular expression is called a **regular set**. Every regular set can be generated by some regular expression.
  - Every finite set of strings is a regular set. At worst, one can just build a regular expression that enumerates them with alternations.
  - Any regular set has multiple regular expressions that generate it. For instance, $(\texttt{ab})^*$ can also be written $\lambda|\texttt{ab}|\texttt{abab}(\texttt{ab})^*$.

- Notations

  - If A is a set of characters, we use $\texttt{not}(\texttt{A})$ to denote $\Sigma - A$, the characters not in A.

- If S is a set of strings, we use $\texttt{not}(\texttt{S})$ to denote all (finite) strings except those in S. It turns out that if S is a regular set, so is $\texttt{not}(\texttt{S})$.
- If $k \geq 0$ is a constant integer and S is a set of strings, then S^k is the set of strings formed by concatenating k strings (possibly different) from S. If S is a regular set, so is S^k.

17 Useful examples

- $\boxed{\text{Class 6, 9/3/2020}}$
- a Java comment that goes to the end of the line: `//(not(`$\hookleftarrow$`))`$^*\hookleftarrow$ (Here, Σ is the set of all 16-bit Unicode characters and $\hookleftarrow$ is a line separator, which is platform-dependent.)
- a decimal literal: $D^+.D^+$ where D is shorthand for `(0|1|2|3|4|5|6|7|8|9).`
- an integer literal, optionally signed: $(+|-|\lambda)D^+$.
- a comment delimited by ## markers: $\#\#((\#|\lambda)\texttt{not}(\#))^*\#\#$
- a Fortran-like real literal, which requires digits only on one side (either one) of the decimal point: $(D^+.D^*)|(.D^+)$
- an identifier, with underscores, but not adjacent, frontal, or terminal ones: $L(L|D)^*(\_(L|D)^+)^*$, or (Daniel Michler) $L((\_|\lambda)(L|D))^*$.

18 Hashing

- Very popular data structure for searching.
- Cost of insertion and of search is $\mathcal{O}(\log\ n)$, but only because n distinct values must be $\log n$ bits long, and we need to look at the entire key. If we consider looking at the key to be $\mathcal{O}(1)$, then hashing is expected to be $\mathcal{O}(1)$.
- Java provides an interface `Map<K,V>` with several implementations: `HashMap<K,V>` (recommended), `Hashtable<K,V>` (synchronized, so more expensive) and others for specialty purposes. The key type `K` and value type `V` can be any classes, although `String` and `String` are typical.
- Idea: find the value associated with key k at `A[`$h(k)$`]`, where

  - $h()$ maps keys to integers in $0..s-1$, where s is the size of A[].
  - $h()$ is "fast". (It generally needs to look at all of k, though.)

- Example

  - k = student in class.
  - $h(k)$ = k's birthday (a value from 0 .. 365).

- Difficulty: collisions

  - Birthday paradox: Prob(no collisions with j people) = $\frac{365!}{(365-j)!365^j}$
  - This probability goes below $\frac{1}{2}$ at $j = 23$.
  - At $j = 50$, the probability is 0.029.

- Moral: One cannot in general avoid collisions. One has to deal with them.
- A good hash function

  - Desiderata

    - Uniform: Equally likely to give any value in $0..s-1$.
    - Spreading: similar inputs $\rightarrow$ dissimilar outputs, to prevent clustering. Only important for open-addressing conflict resolution.
    - Fast.

  - Several suggestions, assuming that k is a multi-word data structure, such as a string.

    - Add (or multiply) all the words of k, discarding overflow, then mod by s. It helps if $s = 2^j$, because mod is then masking with $2^j - 1$.
    - `XOR` the words of k, shifting left by 1 after each, followed by mod s.

  - Wisdom: it doesn't make much difference what hash function you choose. It is not even necessary to look at all of k. Just make sure that $h(k)$ is not constant (except for testing collision resolution).

- Dealing with collisions: open addressing

  - Overview

  - The following methods store all items in A[] and use a probe sequence. If the desired position is occupied, some other position is open to consider instead.
  - They tend to suffer from clustering.
  - Deletion is hard, because removing an element can damage unrelated searches. Deletion by **marking** is the only reasonable approach.
- Perfect hashing: if you know all n values in advance, you can look for a non-colliding hash function `h`. Finding such a function is in general quite difficult, but compiler writers do sometimes use perfect hashing to detect keywords in the language (like **begin** and **for**).
- Additional hash functions. Have a series of hash functions, $h_1(),\ h_2(), \ldots$.
  - insertion: key probing with different functions until an empty slot is found.
  - searching: probe with different functions until you find the key (success) or an empty slot (failure).
  - You need a **family** of independent hash functions.
  - The method is very expensive when A[] is almost full.
- Linear probing. Probe p is at $h(k) + p$ (mod s), for $p = 0,\ 1,\ \ldots$.
  - Terrible behavior when A[] is almost full, because chains coalesce. This problem is called "primary clustering".
- Quadratic probing. Probe p is at $h(k) + p^2$ (mod s), for $p = 0,\ 1,\ \ldots$.
  - When does this sequence hit all of A[]? Certainly it does if s is prime.
  - We still suffer "secondary clustering": if two keys have the same hash value, then the sequence of probes is the same for both.
- Add-the-hash rehash. Probe p is at $(p + 1) \cdot h(k)$ (mod s).
  - This method avoids clustering.
  - Warning: $h(k)$ must never be 0.
- Double hashing. Use two has functions, $h_1()$ and $h_2()$. Probe p is at $h_1(k) + p \cdot h_2(k)$.
  - This method avoids clustering.

- Warning: $h_2(k)$ must never be 0.

- Dealing with collisions: chaining
  - Each element in A is a pointer, initially null, to a **bucket**, which is a linked list of nodes that hash to that element; each node contains k and any other associated data.
  - insert: place k at the head of A[$h(k)$].
  - search: look through the list at A[$h(k)$].
    - optimization: When you find, promote the node to the start of its list.
  - average list length is s/n. So if we set $s \cong n$ we expect about 1 element per list, although some may be longer, some empty.
  - Instead of lists, we can use something fancier (such as 2-3 trees), but it is generally better to use a larger s.

19 Finite-state automata

- A **finite-state automaton** (FSA) is an idealization of a very simple computer, composed of
  - A finite set of **states**, usually depicted by circles.
    - One of the states is called the **start state**. It can be depicted by a circle with an arrow from nowhere pointing to it.
    - One or more of the states are called **final** (or **accepting**) states. They are usually depicted by double circles.
  - A finite **alphabet**, denoted Σ. We'll call the elements of the alphabet **letters**.
  - A set of **transitions** between states, usually depicted by labelled arrows. The labels are letters.
- Class 7, 9/8/2020
- An FSA is equivalent to a language.
  - An FSA can **recognize** strings in its language. It starts at the start state, and every time it sees the next letter, it moves to the state pointed to by an arrow from the current state that is labelled with that letter. If no such arrow exists, the string is

  not in the language. If when the string is finished, the FSA is in a final state, the string is in the language. Otherwise it is not.
  - An FSA can **generate** all the strings in a language by starting at the start state and outputting the label on each transition it takes, stopping at some final state.
  - It turns out that the language recognized or accepted by an FSA is a regular language, and every regular language can be converted to an FSA that recognizes/accepts it.

- An FSA is **deterministic** if no transitions are labelled with λ (which allows you to move to the next state without consuming input) and if no state has multiple outgoing transitions labelled with the same letter (in which case you don't know which state to go to next).
- Even stronger: Regular expressions are equivalent to deterministic FSAs. We will see later how to convert a regular expression to a deterministic FSA.
- Example: Figure 3.1 page 64
- Example: Figure 3.5 page 68

20 Implementing a deterministic FSA to recognize a language

- table-driven
  - Construct a table. Every row is a state; one might label them (arbitrarily) 1, 2, ..., with the start state labelled 1. Every column is a letter, so there are as many columns as the size of Σ.
  - For simplicity, add one extra state at the end, the **error state**, and for every (state,letter) pair for which there is no transition, add a transition to the error state.
  - Driver code

```
state := 1;
while (ch := input.advance())
   state := table[state, ch];
success := accepting(state);
```

  - The table is usually built by an automated scanner generator.

- explicit control: produced automatically or "hard-wired". Advantage: easier to read and faster, but more effort to generate and debug. Figure 3.4 page 68

21 Transducers

- A **transducer** not only recognizes strings in a regular language but also outputs some **semantic value** of the strings (tokens) it recognizes.
- Each transition can be labelled with an **action**, which can be as simple as an output letter or as complex as a function to invoke on the letter that was just recognized. The function might generally be to append to a growing string, and then to post-process that string when the input is finished, if the FSA is now in a final state.

22 Scanner generators: *lex*, *flex*, *jflex*

- Input file: defines how tokens are to be scanned and how to process them.
- Output: a program (in C, or for *jflex*, in Java), which defines a subroutine `yylex()`. (For *jflex*: a class called `Yylex` with a method called `yylex()`)
- One compiles and links this program with the rest of the components to make a functioning compiler.
- *Lex* takes care of low-level details: reading characters efficiently, matching them against token definitions.
- The best way to learn *Lex* is to use the examples for Project 2. We will follow examples in the book.
- The overall structure of a *Lex* input file is in Figure 3.8 on page 71. There are three sections separated by `%%`. *Jflex* has the same sections, but in a different order: subroutines, declarations, regular-expression rules.
- The scanner and the parser need to share token codes (typically small integers). A standard way is to use a table generated by a parser generator. *Lex* typically uses `y.tab.h`, generated by *yacc*.

JFlex uses `sym.java`, generated by *javaCUP*, which we will see later.

- Figure 3.7 page 71 shows a trivial *Lex* definition for part of *ac*, namely, the reserved words.

23 Regular expressions in *Lex*

- Class 8, 9/10/2020
- Shorthand: character classes, like D in our examples of regular expressions: `DIGIT=[0-9]`.
- Bracket syntax for range literals: Figure 3.9 page 72 and Figure 3.10 page 73.
- Escape convention for metacharacters like \.
- It is valid to quote characters or character strings with `"`, but it is not necessary to quote alphanumeric characters.
- Case is significant. Use `[pP][rR][iI][nN][tT]` or `%ignorecase`.
- One may use the usual metacharacters for regular expressions: `*`, `+`, |, and parentheses.
- The character `^` matches the beginning of an input line; `$` matches the end of an input line.
- The postfix operator `?` matches the previous expression 0 or 1 times. You can read it as “optionally”.
- See Figure 3.11 page 74 for fuller examples of regular expressions for *ac* tokens. The result that `yylex()` is meant to return is computed by the **processing code** embedded in braces. The opening brace must be on the same line as the regular expression. The rest of the code, which can comprise many statements, need not be on the same line; it finishes when a matching brace appears.
- Figure 3.12 page 74 shows the same thing with defined classes instead of range literals.
- When the scanner matches an expression, it executes the associated commands. The matched text is in a String variable `yytext` (*Jflex*:

call `yytext()`). It is overwritten by the next token the scanner matches, so save it. Usually the processing code deals with the contents of the matched string, so the rest of the scanner can ignore it.

- If two regular expressions overlap, *Lex* returns the longest possible match; if both expressions match the same string, the earlier expression wins.
- One reasonable style is to have a catch-all expression at the end to match any invalid token.
- *Lex* returns a predefined end-of-file token (integer 0) when it reaches the end of the input.
- The subroutine section of the *Lex* input file can define data structures and routines that the processing code can call.
- To avoid situations where the scanner has to back up (such as Pascal's `0..4`), one can specify **right context**: `[0-9]+/".."` means "match a string of digits, but only if looking ahead one sees two dots in a row". This expression is longer than `[0..9]` so it will win if there are two dots, but it won't match if there is only one dot. The right context portion is not consumed by this match.
- Standard symbols for *Lex*: book Figure 3.13 page 78
- There are alternatives to *lex*: *flex* (faster, GPL), *jlex* and *jflex* (for Java), *GLA* and *re2c* (generate a directly executable, not table-driven, scanner in C).

24 Practical considerations

- Identifiers
  - If the language is not block-structured, the scanner can enter identifiers into the symbol table and return a pointer to the symbol-table entry.
  - The scanner can copy the identifier into **string space** and return a pointer to that space. The parser can decide that the string is a duplicate and reclaim the space of the most recent string.
  - The scanner can copy the identifier into an identifier table (a hash table) and return an associated integer that can be used as a key.

  - If case is significant, a word like `WHILE` is most likely not reserved. If case is not significant, convert all words to a standard case before returning identifier tokens.
  - Simplest: The scanner can return a String and let the semantic checker deal with the symbol table. This alternative uses more space (redundant Strings) but avoids complexity.

- Class 9, 9/15/2020
- Literals
  - Convert numbers to internal representation. Use library routines (in C: *atoi()*; in Java `Integer.parseInt()`). You might use higher-precision methods and compare against limits to detect range errors. You can also catch `NumberFormatException` in Java.
  - Convert string literals by expanding escaped characters.
  - There is a weird ambiguity in C: `x (* y)` is a declaration, not a procedure call, if `x` has been given a meaning via **`typedef`**. The parser might keep a table of **`typedef`** identifiers, and the scanner could return a different token for such identifiers.
- Reserved words
  - One can catch reserved words by a regular expression at the expense of increased FSA size. That's how we'll do it for CSX.
  - The scanner can have processing code for identifiers that looks them up in a reserved-word table, returning a special token for such identifiers.
- Handling compiler directives
  - **file inclusion**. Keep a stack of open files, or recursively invoke the scanner.
  - **conditional compilation**. Usually handled as a separate pass before the scanner.
  - **Unicode escapes**, such as `\u05b3` in Java. Also a separate pre-scanning pass.
  - **listing**. It's hard to properly intersperse compilation diagnostics.
- End of file

  - It can simplify the parser to continue to return the `EOF` token if the scanner is invoked after it produces an `EOF` token.

- Multi-character lookahead

  - example from Fortran: `DO 10 J = 1.100`
  - general backup: buffer characters; if you enter an error state, back up until you reach an accepting state. If you back up to the start of the token, report a single-character error token and move past it.

- Speed

  - Use a scanner generator; *flex* or (better) *GLA*.
  - Hard-coded: use block operations for read, double-buffering to handle tokens that cross block boundaries; use the buffer as the token store until you decide to copy them.
  - Use a profiling tool.

- Error recovery

  - when you reach an error state, you can delete characters so far or just the first character.
  - in any case, you can return an "error token" that informs the parser that the subsequent token is unreliable.
  - runaway strings and comments: use special rules that detect, because ordinary error recovery (deleting first character) generates cascading errors. Nested comments cannot be handled by regular expressions.

25 Converting regular expressions to finite automata

- Convert the regular expression to a non-deterministic finite-state automaton (NDFA), using λ rules. $\boxed{\text{Figures 3.19-22 pages 93-94}}$
- $\boxed{\text{Class 10, 9/17/2020}}$
- Convert the NDFA to a deterministic finite-state automaton (DFA) by the **subset construction**.

- Each state of the DFA corresponds to a set of states in the NDFA.
- The start state of the DFA corresponds to the start state of the NDFA plus any NDFA states reachable by a λ transition from the start state.
- Each set of NDFA states reachable by an input string becomes a single DFA state.
- If any of the NDFA states in a DFA state accepts, then so does the DFA state.
- Iterate over all states in the DFA (this set grows) building new states.
- This method terminates, because at most there are $2^{|NDFA|}$ states in the DFA.
- See algorithm Figure 3.23 page 95
- Example: Figure 3.24 page 96
- Example: $(D^+.D^*)|(.D^+)$

26 Optimizing the resulting DFA

- Optimization is only to reduce the number of states, and hence the table; it doesn't change the speed.
- One should remove unreachable states and dead states (those from which one cannot reach an accepting state).
- All accepting states with no outgoing transitions are equivalent.
- Start by assuming all non-accepting states are equivalent.
- Repeatedly: if any character c causes transitions from a merged accepting state (possibly to a single "error state") to multiple states, split that state based on the behavior of c.
- Example: Figure 3.26 page 98

27 Converting a FSA to a regular expression

- Class 11, 9/22/2020

- not needed for compiler construction, but helps prove the equivalence of these two formalisms.
- Assume start state has no incoming transitions and there is a single accepting state with no outgoing transitions; build it if needed.
- Apply three transformations Figure 3.30 page 101 repeatedly to remove states and add regular expressions.
- Example, Figure 3.25 page 97 leads to `b`*`a (`λ`|b|ba|bb|a)`

28 Context-free grammars: Chapter 4

- A **context-free grammar** (CFG) represents a **context-free language** language (CFL) (a set of strings).
- All regular languages are also context-free, but there are some context-free languages, such as the set of balanced parentheses, that are not regular.
- Instead of an FSA, one needs a **push-down automaton** (PDA) to recognize or generate strings of a CFL. We won't be going into the theory of PDAs.
- Components of a CFG
  - a finite **terminal alphabet** Σ, composed of tokens, augmented with an EOF token. We'll use lower-case words and punctuation symbols.
  - A finite **nonterminal alphabet** N, whose symbols are like variables in the grammar. We'll use initial-capital words or single letters.
  - A **start symbol** $S \in N$.
  - A finite set of rewriting rules called **productions** of the form $A \rightarrow X_1 \ldots X_m$, where $A \in N$, $X_i \in N \cup \Sigma$. We allow $m = 0$, in which case we write λ as the right-hand side.
  - We allow | as a simplifying syntax if many rules share the same left-hand side.
- The CFG is a recipe for rewriting strings; each rewrite is a step in a **derivation** of the resulting string. We denote a step of a derivation with $\Rightarrow$. We denote possibly many steps with $\Rightarrow^*$.

- The strings we get, even if they still contain nonterminals, are called **sentential forms**.
- In a derivation, we may expand any nonterminal we wish in the next step, but there are two conventions.
  - **leftmost derivation**: always expand the first nonterminal in the current sentential form. (How would we express this order in a direction-free way? Maybe "frontmost".) Notation: $\Rightarrow_{lm}$.
    - For a leftmost derivation, we don't need to mention which nonterminal we are expanding, only what rule we are using. Example: Figure 4.1 page 116 and the derivation later on the page.
    - top-down parsers generate leftmost derivations; we say they produce a **leftmost parse**.
  - **rightmost derivation**: expand the last nonterminal ("endmost"). Notation: $\Rightarrow_{rm}$. Also called **canonical derivation**.
    - bottom-up parsers generate rightmost derivations
    - Example: book page 117
    - The parser actually discovers this parse in reverse order.

29 Parse trees

- Describes a derivation.
- The root is S; each node is either a terminal, a nonterminal, or λ.
- Interior nodes are nonterminals; together with their children, they represent the application of a production.
- Both a leftmost and a rightmost derivation give rise to the same parse tree; the tree does not show the order of derivation.
- Given a sentential form, all its symbols descended from any single internal node is a **phrase** of that sentential form.
- A **simple phrase** contains no smaller phrase; its children are leaves (but leaves might not be terminals, because the sentential form might still have nonterminals).
- The **handle** of a sentential form is its leftmost simple phrase.

- Example: Figure 4.2 page 118: sentential form `f ( v Tail)`, the simple phrases are `f` and `v Tail`, and the handle is `f`.
- Example: sentential form `Prefix ( v + v )`. The phrases are
  - `Prefix ( v + v )` (comes from `E`)
  - `v + v` (comes from `E`)
  - `+ v` (comes from `Tail`)
  - λ (simple; handle) (comes from `Tail`)

30 Grammar types

- A **regular grammar** has rules like a CFG, but the RHS is restricted to a single symbol from $\Sigma \cup \{\lambda\}$, followed optionally by a single nonterminal symbol. Regular languages are a proper subset of context-free languages.
- **Context-free grammars**, represented by BNF, can always be parsed in $\mathcal{O}(n^3)$ time. Useful subclasses of CFGs can be parsed on $\mathcal{O}(n)$ time.
- A **context-sensitive** grammar has rules like a CFG, but the LHS is allowed to have extra context both before and after the nonterminal. This context is preserved on the RHS. Context-free languages are a proper subset of context-sensitive languages.
  - We don't use context-sensitive languages because parsing them is very expensive.
  - However, they would allow us to require that variables are declared before use as part of the grammar instead of a check performed after parsing.
- An **unrestricted grammar** (also called **type-0**) lets arbitrary patterns be rewritten.

31 Using CFGs for describing programming-language syntax

- We don't want **unreduced** grammars, which have useless nonterminals that are never generated by any derivation of a string of terminals. Example: $\boxed{\text{book page 120}}$. Parser generators usually verify that the grammar is reduced.

- We prefer **unambiguous** grammars, where every sentence has a unique parse tree. Example of ambiguous grammar: $\boxed{\text{book page 121}}$ Unfortunately, guaranteeing that a CFG is unambiguous is undecidable. We will return to this problem later.
- We don't want CFGs that generate the "wrong" language. Deciding whether two CFGs generate the same language is undecidable.

32 Extended BNF

- Allow metacharacters '[' and ']' to surround optional symbols in the RHS.
- Allow metacharacters '{' and '}' to surround symbols that may be repeated 0 or more times. I like an extension to this notation that tells you what to place *between* each repeated symbol, typically a comma. Example:
  `Decl → [final] [static] [const] Type id {, id }`
- It's not hard to convert extended BNF into standard BNF. For every optional region, introduce a new nonterminal N with two rules, one expanding to the symbols in the region, the other to λ. For every repetition region, introduce a new nonterminal M with two rules, one expanding to the symbols of the region followed by M itself, the other to λ.

33 Recognizers and parsers

- $\boxed{\text{Class 12, 9/24/2020}}$
- A **recognizer** determines if a string is in a language.
- A **parser**, more useful to us, also builds the parse tree.
- A **top-down** parser builds a leftmost derivation, traversing the parse tree in preorder. These parsers look ahead in the input to predict the right production before applying it. Common strategy: LL(1).

- A **bottom-up** parser builds a rightmost derivation, starting at the leaves and working up to the root, traversing the tree in postorder. Children of nodes are inserted before the nodes themselves. Common strategy: LR(1).
- Extended examples: grammar book page 126, parses on previous pages.
- The names LL(1) and LR(1): the first letter indicates that input is scanned left-to-right (start to finish). The second letter indicates a leftmost (L) or rightmost (R) derivation. The notation (1) means one-token lookahead.

34 Data structures to represent a CFG

- We assume some representation for **sets** (such as Σ and N), **lists**, and **iterators** over sets or lists. In Java, classes that implement the `Collection` interface allow a **`for`** loop like this:

```java
List<Symbol> symbList = new ArrayList<Symbol>();
for (Symbol oneSymb : symbList)
   System.out.println(oneSymb.name);
```

- There is also an `Iterator` interface with a clunkier API, including `hasNext()`, `next()`, and `remove()`. You must use this interface if you plan to remove elements while iterating through the set. You get an `Iterator` from a `Collection` by `iterator()`:

```java
for (Iterator<Symbol> it = symbList.iterator();
      it.hasNext(); ) {
   Symbol oneSymb = it.next();
   System.out.println(oneSymb.name);
   if (someCondition(oneSymb)) it.remove();
}
```

- Our data structures may take advantage of these facts:
  - We won't be removing symbols from the CFG (except in rare cases of reducing the grammar).
  - Removing [. . .] and {. . . } adds symbols and productions to the grammar.

  - Given a nonterminal A, we will want to visit all rules with A on the LHS. We will also want to visit all rules that mention A in the RHS.
  - We will generally process an RHS one symbol at a time.
- Therefore, we represent a rule by its LHS symbol and a list of its RHS symbols. The list is empty if the RHS is λ.
- See page 128 for a list of utility routines we might want.

35 LL(1): recursive descent

We will build an LL(1) parser of the style called "recursive descent". To do that, we need to analyze the BNF, computing some properties:

- *RuleDerivesEmpty*(p) and *SymbolDerivesEmpty*(N).
- *First*(N).
- *Follow*(N).
- *Predict*(p).

36 Computing when a nonterminal can derive λ

- A derivation to λ may take more than one step.
- Algorithm Figure 4.7 page 129.
  - Make a list L of nonterminals that directly derive λ.
  - For each $N \in L$, find all productions where N appears on the RHS; if removing N from those productions leads to an empty RHS, add the LHS to L.
  - Keep a count of RHS lengths so the previous step can account for several nonterminals in the same RHS, all of which can derive λ.
- The result is two Boolean characteristics: *RuleDerivesEmpty*(p) and *SymbolDerivesEmpty*(N).

37 Computing *First*(α)

- *First*(α) = $\{b \in \Sigma \mid \alpha \Rightarrow^* b\beta\}$. Here, α and β are strings of symbols (terminals or nonterminals). We won't include λ in *First*(α).
- Algorithm $\boxed{\text{Figure 4.8 page 130}}$.
  - If the BNF is written in a top-down fashion, as is conventional, it's most straightforward to work from the end to the beginning.
  - Consider first character of α.
  - Easy cases: α is empty or terminal.
  - Hard case: nonterminal N.
    - For each RHS R in productions with LHS=N, recursively call algorithm on R, collecting all answers.
    - If *SymbolDerivesEmpty*(N), union in recursive result on the remaining characters of α.
- Avoid endless recursion by refusing to consider the same nonterminal twice (Boolean *visitedFirst*(N)).
- Example: $\boxed{\text{Figure 4.1 page 116}}$
- Example: $\boxed{\text{Figure 4.10 page 133}}$.

38 Computing *Follow*(N)

- *Follow*(N) is the set of terminals that can come after nonterminal N in a sentential form.
- *Follow*(N) = $\{b \in \Sigma \mid S \Rightarrow^+ \alpha N b \beta\}$. Here, α and β are strings of symbols (terminals or nonterminals).
- Algorithm $\boxed{\text{Figure 4.11 page 135}}$.
  - For each place N occurs in the RHS of some production p, union in *First*(*tail*), where *tail* represents the remaining symbols in that RHS after N.
  - If *tail* can derive empty (for instance, *tail* is itself empty), then union in the recursive result of the *Follow*(LHS(p)).
  - Avoid endless recursion by refusing to consider the same nonterminal twice (Boolean *visitedFollow*(N)).

- $\boxed{\text{Class 13, 9/29/2020}}$
- Example: $\boxed{\text{Figure 4.10 page 133}}$.
- Example: $\boxed{\text{Exercise 10 page 140}}$.

LHS	$\rightarrow$	**RHS**	**predict set**
P	$\rightarrow$	Ds Ss $	*First*(RHS) = {f, i, id, p, $}
Ds	$\rightarrow$	D Ds	*First*(RHS) = {f, i}
	|	λ	$\varnothing$ ∪ *Follow*(DS) = {id, p, $}
D	$\rightarrow$	f id	*First*(RHS) = {f}
	|	i id	*First*(RHS) = {i}
Ss	$\rightarrow$	S Ss	*First*(RHS) = {id, p}
	|	λ	*Follow*(SS) = {$}
S	$\rightarrow$	id = V E	*First*(RHS) = {id}
	|	p id	*First*(RHS) = {p}
E	$\rightarrow$	+ V E	*First*(RHS) = {+}
	|	- V E	*First*(RHS) = {-}
	|	λ	*Follow*(E) = {id, p, $}
V	$\rightarrow$	id	*First*(RHS) = {id}
	|	num	*First*(RHS) = {num}

- can derive λ: Ds, Ss, E
- *First*(V) = {id, num}
- *First*(E) = {+, −}
- *First*(S) = {id, p}
- *First*(Ss) = *First*(S) = {id, p}
- *First*(D) = {f, i}
- *First*(Ds) = *First*(D) = {f, i}
- *First*(P) = *First*(Ds) ∪ *First*(Ss) ∪ {$} = {f, i, id, p, $}
- *Follow*(P) = ∅
- *Follow*(Ds) = ∅ ∪ *First*(Ss $) = *First*(SS) ∪ {$} = {id, p, $}
- *Follow*(D) = *First*(Ds) ∪ *Follow*(Ds) = {f, i, id, p, $}
- *Follow*(Ss) = ∅ ∪ *First*($) = {$}
- *Follow*(S) = *First*(Ss) ∪ *Follow*(Ss) = {id, p, $}
- *Follow*(E) = ∅ ∪ *Follow*(S) = {id, p, $}
- *Follow*(V) = *First*(E) ∪ *Follow*(E) ∪ *Follow*(S) = {+, −, id, p, $}

39 Chapter 5: Top-down (LL) parsing (overview)

- Not as powerful as bottom-up, but simple, fast, with good error diagnostics.
- If one can build a top-down parser, the grammar is not ambiguous, although in general ambiguity is not decidable.
- Two general approaches: **recursive descent** and **table-driven**.
- Names for this style of parsing: **top-down**, **predictive**, **LL**(k), **recursive-descent**.
- The recursive-descent parser
  - Each nonterminal A has its own parsing procedure `procA`.
  - `ProcA` chooses one of A's rules by inspecting the next k (at most) tokens.
  - The constant k is the **lookahead** value. One chooses k before building the parser.
  - We'll use $k = 1$ for discussion.
  - Each of the rules for A has its own **predict set** (computed below)
- Given token `t` and all the predict sets for the current nonterminal A:
  - `t` is in none of the predict sets: Syntax error. The parser could output a useful message (saw t, expecting ...).
  - `t` is in more than one predict set: ambiguous case. The parser generator should prevent this situation.
  - `t` is in exactly one predict set. Follow the production predicted.
- To compute the predict set for a production p $\boxed{\text{Figure 5.1 page 147}}$
  - the predict set is at least *First*(RHS(p)).
  - if *RuleDerivesEmpty*(p), the predict set also has *Follow*(LHS(p)).
- Example: $\boxed{\text{Exercise 10 page 140}}$. See p. 35.
- For the CFG to be LL(1), all predict sets for a nonterminal must be disjoint.
  - It is easy to detect violations automatically.
  - If the CFG is not LL(1), it might still be LL(k) for some $k > 1$.

  - Sometimes a more powerful parsing method, like LR(k), is needed.
  - Sometimes even that fails, for instance, if the CFG is ambiguous.

40 The recursive-descent LL(1) parser

- Uses `peek()`, `advance()`, and (based on those two) `match(token)`.
- For a nonterminal A, procedure `procA()` peeks at the next token, using that value to switch among the competing productions.
- For each production p, each symbol X in the RHS(p) becomes a call.
  - If X is terminal, call `match(`X`)`.
  - If X is a nonterminal, call `procX()`.
- Example: Figure 5.7 page 151
- Class 14, 10/1/2020
- Example: Figure 5.2 page 148

41 The table-driven LL(1) parser

- Why bother? Because the code for the recursive-descent parser can be fairly long. A table can be much smaller. I think we can ignore the cost of procedure call and return.
- Data structure: a stack, with operations `push(s)`, `pop()`, and `top()`. We push terminals and nonterminals.
- Algorithm for the parser
  - if `top()` shows a terminal t, call `match(`t`); pop()`. If that terminal is not in the input, we have a syntax error.
  - if `top()` shows a non-terminal A, look up the correct production p = T[A, `peek()`].
  - if $p = 0$, syntax error.
  - otherwise, push RHS(p) onto the stack so the first element goes to the top of the stack.

- Algorithm to build the table T:
  - For each non-terminal A, for each terminal t,
    - if $t \in \mathit{Predict}(p)$, place T[$A,\ t$] = p.
    - otherwise, place T[$A,\ t$] = 0.
- Example of table: Figure 5.10 page 154
- Example of parse: Figure 5.11 page 155
- It helps to map terminals and non-terminals to small integers for the purpose of looking them up in T.

42 Full example

- Given the grammar Figure 5.2 page 148, for each nonterminal N, compute derives-λ(N), *first*(N) (in reverse order), *follow*(N) (in forward order). Then for each production p compute *predict*(p). The answer is in Figure 5.3, page 148.
- Then derive procB:

```
void procB() {
  switch (peek()) {
    case 'b':
      match('a');
      procB();
      procC();
      match('d');
      return;
    case 'q'; case 'c'; case 'd'; case '$': // $ is EOF
      return;
    default:
      error();
  } // switch
} // procB()
```

- Compute a parse table. Answer is in Figure 5.10 page 154

43 Making a CFG LL(1)

- Common prefixes shared between two productions for the same nonterminal.
  - Increasing lookahead value k doesn't help.
    Figure 5.12 page 156
  - Easy to fix by **factoring**: leave the common part in a single production, and follow it by a new nonterminal that expands in several productions to the various sequelae.
    Figure 5.14 page 157
  - Class 15, 10/6/2020
  - One may need to run this algorithm several times, removing common prefixes longest-first.
  - One can force all RHSs to begin with a terminal symbol (Greibach normal form), and then one can remove common prefixes, but the resulting grammar may still not be LL(1).
- Left recursion, which causes a unbounded recursion (in recursive-descent parsing) or unbounded pushing the RHS onto the stack (in table-driven parsing).
  - Easy to fix: if the production is $A \rightarrow A\ \alpha$, change it to $A \rightarrow X\ Y$.
  - For every other production $A \rightarrow \beta$, change it to $X \rightarrow \beta$.
  - Add production: $Y \rightarrow \alpha\ Y$ (right-recursive!)
  - Add production: $Y \rightarrow \lambda$.
  - Example: Figure 5.16 page 158

44 Review for midterm

45 midterm

Class 16, 10/8/2020

46 Review of midterm

Class 17, 10/13/2020

47 Compressing the parse table

- Class 18, 10/15/2020
- Most entries are blank, so direct representation is space-inefficient. Example (Ada): 6.5% density.
- Given a non-terminal, entries for many lookahead symbols agree. See Figure 5.10 page 154.
- Can store a *default* value for each terminal; for some terminals, it will be the empty (0) value. Then we only need to store non-default values.
- Compaction
  - Binary search: list all the non-default values in a one-dimensional table, along with their row, column indices. Search T[i, j] using binary search on those indices. For E non-default entries, the space cost is $3E$, and the time cost is $\mathcal{O}(log E)$.
  - Hash table: Search T[i, j] at $h(i, j) = (i \times j) \mod (|E| + 1)$, resolving collisions by linear resolution. There is guaranteed to be an empty cell.
  - Perfect hashing is possible, since the values are known in advance.
- Compression by double-offset indexing
  - Compute non-conflicting shift values R[i] for each row i into a large table V.
  - Find T[i, j] at *offset*=V[R[i]+j], accepting it if V.*fromrow*[*offset*] = i.
  - Example: Figure 5.24 page 169
  - Finding the best ordering of rows to add to V is NP-complete.
  - Even with an arbitrary ordering, this method works pretty well. Example (Ada): 6.8% resulting table size.
  - Time to look up an entry in the compressed table is $\mathcal{O}(1)$, and its space is no worse than original T.

48 Dangling-else problem

- The dangling-else problem cannot be solved with LL(k) grammars.
  - Grammar: Figure 5.17 page 159
  - Equivalent to $\{a^j b^k \mid 0 \le k \le j\}$
  - So long as the parser sees only as, it can't decide whether to predict a matched or unmatched a. The amount of lookahead before the first b can be arbitrarily large.
  - But this language is context-free, because it has a CFG.
- To handle this problem, accept the predict conflict. Factor the grammar Figure 5.19 page 161. Get the table shown in that figure. Resolve the conflict by always choosing rule 4.

49 Properties of LL(k) parsers

- An LL(k) parser computes a correct leftmost parse.
- Grammars in the LL(k) class are unambiguous.
- Table-driven LL(1) parsers operate in $\mathcal{O}(n)$ time and space, where n is the length of the input (measured in tokens).
  - Time: each input token induces a bounded number of actions (no recursive use of the same production).
  - Space: the stack can be of length $\mathcal{O}(n)$; each input token contributes to a constant increase in stack size (no recursive use of the same production).

50 Recovering from and repairing syntax errors

- Compilers usually disable semantic analysis and code generation if there are errors. Users won't run the program in any case.
- **Correct prefix** property of LL(k) parsing: until you hit an error, there is a string of upcoming tokens that can complete a successful parse.
- **Recovery**: try to enter a configuration where the parser can continue.

- Bad recovery causes a cascade of errors.
- **Panic mode**: skip input tokens until encountering a frequently occurring delimiter, like semicolon.
- Wirth's method: pass a *termset* parameter to each procedure in the recursive-descent parser, indicating what lookahead tokens are possible for this invocation. All calls to *match*() are enhanced: If they fail, they print a message and then skip tokens until a member of the *termset* is seen. Example: Figure 5.26 page 172

- **Repair**: Modify the input to obtain an acceptable parse.
  - Given that one has parsed α and then the next token x is erroneous, followed by β, one can try
    - modifying α. Since we know that α is a correct prefix, we prefer not to modify it, except for **scope repair**, which inserts or deletes brackets to match the brackets in β.
    - inserting δ to obtain $\alpha\delta x\beta$. One must be careful not to keep inserting, or parsing never finishes. Some languages are **insert correctable**: one can always fix errors by insertion.
    - deleting x to get $\alpha\beta$. Drastic, but guaranteed to make progress.

51 Chapter 6: Bottom-up parsing

- Bottom-up parsers handle the largest class of grammars that can be parsed deterministically.
- Just as with top-down parsing, there are parser generators that build the necessary tables. *javaCUP* and *Yacc* are examples.
- A bottom-up parser starts with leaves (terminals) and decides to **reduce** them to terminals.
- Also called a **shift-reduce** parser or an LR(k) parser.
- Informal example: Figure 6.1 page 183
- Algorithm: Figure 6.3 page 185
- Example: parse `abbdc$` using the table in Figure 6.5 page 187 Squares mean "shift"; no square means "reduce".

52 LR(0) parsing

- $\boxed{\text{Class 19, 10/20/2020}}$
- Informal example: $\boxed{\text{Figure 6.4 page 186}}$
- Warning: there is actually an LR(0) parser, which uses one token of lookahead while parsing, but not while the table is built.
- Example based on $\boxed{\text{Figure 6.2 page 184}}$.
- The states in the parser are related to how far we have recognized a RHS. We call progress a **bookmark** and denote it by •.
- **Items** are production RHSs with • somewhere in them.
- A **fresh** item has • at the start of the RHS.
- If the • is at the end of the RHS, the item is **reducible**.
- If the RHS is λ, there is only one item, which is both fresh and reducible.
- See $\boxed{\text{Figure 6.8 page 192}}$ for items pertaining to one RHS.
- Each **parser state** is set of the items, called its **kernel**.
- The start state is composed of fresh items for goal productions.
- Build the **characteristic FSM**
  - work list starts with just the start state (state 0)
  - take the first state on the work list: $A \rightarrow \alpha \bullet B\beta$
  - compute its **closure**, which includes fresh items with B on the LHS. (if • is not before a nonterminal, the closure is empty), as well as fresh items with other nonterminals on the LHS for any production for B whose RHS starts with a nonterminal, recursively.
  - for each element of the closure, we get a new state whose kernel is determined by advancing the •. Put that state on the work list. The current state points to it with an arrow labelled by what we advanced the • over (it can be a terminal or nonterminal). If advancing over the same symbol applies to more than one element of the closure, the resulting items are all part of the kernel of the subsequent state.
- Example: $\boxed{\text{Figure 6.11 page 194}}$

- We can use the characteristic FSM to parse; each accepting state causes a reduction, and we start anew at the start state.
- Example: Figure 6.12 page 195
- But we don't need to start anew: We can remember the state we were in after shifting each symbol.

53 Conflicts

- Class 20, 10/22/2020
- The table may only suggest one course given a state and a lookahead: shift, reduce, error, or accept.
- We can't handle nondeterminism; NPDA is more powerful than a PDA.
- If the table has multiple suggestions, the state is **inadequate**, but a stronger parsing method might work.
- **shift-reduce conflict**: the state contains at least two items, one of which is before a terminal t, and the other is reducible.
- **reduce-reduce conflict**: the state contains two reducible items.
- Example: Figure 6.16 page 200 This grammar is in fact ambiguous, so the inadequacy cannot be resolved without changing the grammar.
- Example: Figure 6.19 page 203 This grammar is unambiguous but not LR(k); it has a reduce-reduce conflict, and no amount of lookahead can distinguish the two possibilities.
- Instead of LR(0), one can try SLR(1), LALR(1), LR(1) (in increasing order of power). LALR(1) is the method of choice.
- SLR(1): in states with a shift-reduce conflict, use $\mathit{Follow}(A)$ where A is the LHS of a reduce item to decide when to choose reduce, and *First*(tail) to decide when to choose shift. (Here, tail is whatever follows •). With luck, members of these sets do not overlap.
- Example: Figure 6.22 page 207 (grammar on page 205). To disambiguate the shift-reduce conflict in states 1 and 6:
  - $\mathit{Follow}(E) = \{\$, +\}$.

  - *First*(* n) = {*}.

- $\boxed{\text{Class 21, 10/27/2020}}$
- Example for LR(0) construction: $\boxed{\text{Example 10g page 227}}$
- LALR(1): We omit the construction, but motivate it with grammar in $\boxed{\text{Figure 6.25 page 210}}$.

54 Chapter 7: Syntax-directed translation

- **Syntax-directed translation** refers to actions taken while parsing.
- As it constructs a derivation, the parser performs **syntactic actions**.
- These actions are achieved by executing code embedded in the CFG given to the parser generator. This code is in the same language as the parser.
- The actions might associate **semantic values** with terminals and nonterminals.
  - Values that move down the tree from parent to child are called **inherited**.
  - Values that move up the tree from child to parent are called **synthesized**. Example: $\boxed{\text{Figure 7.1 page 237}}$
  - Top-down: when a production $A \to X_1 \ldots X_n$ is expanded, the value of A can be inherited by all the values of X_i. All the values of $X_i, i \leq n$ can be synthesized into the value of A and can be inherited by all $X_k, k > i$. Example: $\boxed{\text{Figure 7.2 page 238}}$ Each `A` node in the figure increments its parent's semantic value.
  - Bottom-up: when a production $A \to X_1 \ldots X_n$ is reduced, the value of A can be synthesized from the values of X_i.
  - Values of terminals are set by the scanner, and the parser can also look up and store information about identifiers in the symbol table.
- Programming semantic actions requires knowledge of parsing order.
- Programming good semantic actions may require modifying the grammar.

55 Bottom-up syntax-directed translation

- The notation to refer to a semantic value depends on the software. For *javaCUP*, *X:val* means the value of the symbol X on the right-hand side of a production; *RESULT* is the pseudo-variable that holds the value of the symbol on the left-hand side of the production. Both can have any type, including **`int`** or some **`class`**.
- The book uses an abstract notation: X_{val}.
- Along with the parse stack (syntactic stack), the software maintains a **semantic stack** so code can manipulate these values.
- Running example: Convert a string of digits into an integer value. Compilers usually do that in the scanner, but the example gives us insight into bottom-up syntax-directed translation.
- Grammar 1: Figure 7.3 page 240. Notice the left-recursive rules.
- Class 22, 10/29/2020
- Grammar 2: Change the language so we can have both octal and decimal numbers. Figure 7.4 page 241.
  - – Parsing works, but it doesn't restrict octal digits to $0 \ldots 7$.
  - – The grammar has no way of informing the bottom-up translation that an octal, not a decimal number is in progress, because the o is just shifted onto the stack, and shifts do not have semantic results.
- Modifying the grammar can fix the problem, but be careful.
  - Have a test suite of inputs and expected outputs, and test after every grammar change. Such checks are called **regression tests**.
  - Change the grammar in small steps.
  - Every bug you discover gives rise to a new regression test in the suite.
- Grammar 3: **rule cloning** of Grammar 2. Figure 7.5 page 243
  - – Inflates the grammar, which ought not to be necessary.
- Grammar 4: Force semantic actions

  - To catch a shift of a terminal t, introduce a new production $T \to t$ and associate it with a semantic action.
  - To introduce a semantic action between two symbols, introduce a nonterminal A there. Give it production $A \to \lambda$, and associate the semantic action with that production.
  - Result: $\boxed{\text{Figure 7.6 page 244}}$
  - – The actions use a global variable `base`.

- Modify the language to allow an arbitrary 1-digit base following `x`, as in `x5431$`, which means $431_5 = 116$. The new grammar is $\boxed{\text{Figure 7.7 page 245}}$.
- Global variables work, but they have drawbacks.
  - Recursive parsing means there can be unexpected interactions between semantic actions and the global variables.
  - It's necessary to make sure that every sentence leads to a parse that initializes the global variables
  - It's difficult to write accurate code that uses global variables.

- It is better to restructure the grammar so the necessary information is stored in semantic values.
  - Sketch a parse tree such that a bottom-up parse synthesizes the necessary values, as in $\boxed{\text{Figure 7.8b page 246}}$.
  - Revise the grammar to produce that tree.
  - Verify the grammar is still parseable (LALR(1), for instance) and that it passes the regression tests.
  - New grammar: $\boxed{\text{Figure 7.8a page 246}}$
  - Now the semantic value of *Digs* has two components.
  - The language is slightly different: we now accept input `x8$`.

- Puzzle: change the language to allow multi-digit bases.
  - Add a production: Setbase $\to$ x [Digs].

56 Top-down syntax-directed translation

- We'll use recursive-descent parsing, not table-driven.

- Running example: Evaluate a Lisp-like expression with operators `plus` and `prod`. Grammar: Figure 7.9 page 248
- Can pass the inherited semantic value to procedures as a parameter.
- Can return the synthesized value as a return value.
- Example Figure 7.10 page 249
- Example: My unit conversion program.

57 Method resolution in Java

- Class 23, 11/5/2020
- Java method resolution is **dynamic** in resolving which class's methods are appropriate.

```
class A { void m() {println("A");} }
class B extends A { void m() { println("B"); } }
A a = new B();
a.m(); // prints B: dynamic resolution of m
```

- Java method resolution is **static** with respect to resolving overload based on parameter number and type.

```
class X { }
class Y extends X { }
class A {
   void m(X param) { System.out.println("A-X"); }
   void m(Y param) { System.out.println("A-Y"); }
} // class A
class B extends A {
   void m(X param) { System.out.println("B-X"); }
   void m(Y param) { System.out.println("B-Y"); }
} // class B
class Foo {
   public static void main(String[] args) {
      X x = new Y();
      A a = new B();
      a.m(x); // prints B-X (B dynamic, X static)
   } // main()
} // class Foo
```

58 Abstract syntax trees

- The parser can easily build **concrete syntax trees**.
- **Abstract syntax trees** (ASTs) let a node parent any number of children, for example, to represent compound statements and parameter lists.
- Some AST nodes have a fixed number of children, such as multiplication.
- A suggested data structure Figure 7.2 page 252.
    - Each node points to its right sibling.
    - Each node points to its leftmost sibling.
    - Each node points to its parent.
    - Each node points to its leftmost child.
    - Appending to the end of the children of a node requires walking down the whole list. It would be good to also have each node point to its rightmost child, although the book doesn't suggest it.
- Functions for constructing AST nodes
    - *makeNode*(t) creates a new node. It has overloaded versions for different types of t, such as **`int`**, Symbol, Operator.
    - x.*makeSiblings*(y) appends y and all its right siblings to the end of x's right sibling list. Code is at Figure 7.13 page 253.
    - x.*adoptChildren*(y) appends y and all its right siblings to the right end of x's children. Code is at Figure 7.13 page 253.
    - *makeFamily*(op, child$_1$, ...) creates a new Operator node with the given children. If there are two children, the body is equivalent to *makeNode*(op).*adoptChildren*(child$_1$, *makeSiblings*(child$_2$)).
- The exact form of the AST can evolve as one implements the compiler.
- Desiderata of the AST structure
    - Ability to unparse into a program functionally equivalent to the original.

  - Hiding of implementation details; contents accessed by getters.
  - Different APIs for different phases of compilation.

- Roadmap for designing the AST

  - Build an unambiguous grammar G for the language L. Some productions might be present specifically to disambiguate.
  - Devise an AST for L, discarding grammar details used for disambiguation and unnecessary punctuation, such as ';'.
  - Add semantic actions to G to construct the AST during parsing. This activity might require modifying G. Use the functions enumerated above.
  - Implement the phases of the compiler, modifying the AST both in structure and content as needed. Even G might need to change.

59 Extended example of AST

- The grammar G is in Figure 7.14 page 256. It needs no declarations, because all variables are integer.
- Initial design of the AST

  - Assignment statements are represented by a node with two children: variable and expression.
  - **`if`** statements: We can manage with a single structure, and treat an **`if`** without an **`else`** as a special case. So we need a node with three children: the Boolean (actually integer) predicate, the **`then`** alternative, and the **`else`** alternative. We do not need to represent the **`fi`** bracket.
  - **`while`** statements are represented by a node with two children: the Boolean predicate and the body (a statement).
  - blocks (compound statements) are represented by a node with an arbitrary list of children, one for each sub-statement.
  - The **`plus`** operator is represented by a node with two children for the left and right operands.
  - An id and a num are represented by leaf nodes.

- Semantic actions Figure 7.17 page 259

- Compare the concrete with the abstract syntax trees for a sample input: Figure 7.19 page 260.

60 Design patterns for ASTs

- Design patterns make software easier to develop and maintain.
- The discussion in the book is overly complex for our CSX project.
  - We can use a single class hierarchy of AST nodes.
  - Each phase of compilation is governed by a different method in the AST class (and subclasses).
  - You invoke that phase of compilation by calling that method in the top node of the AST.
- The book suggests this pattern.
  - Every AST node already inherits methods for manipulating the tree structure (connecting siblings, adopting children). These methods can be placed in a superclass `AbstractNode`.
  - No single class hierarchy works for all phases.
  - Code for each phase should be in a single class, not in each node type.
  - A phase f does its work by f.*visit*(`Node` n).
  - The **visitor pattern** arranges for the right instance of *visit* to invoke based on the actual type of n.
  - A single class `Visitor`, extended by each compilation phase.
  - Each node type extends `AbstractNode`. It provides its own version of *accept*(`Visitor` v), which looks identical but provides its own (typed) version of **`this`**.
- Example: Figure 7.23 page 266 Trace the **double dispatch**:
  ```
  Visitor f = new TypeChecking(...);
  AbstractNode n = new PlusNode(...);
  f.visit(n);
  ```
- Disadvantages of double dispatch
  - Repeated code in each concrete node class of the *visit*() method.

- All *visit*() methods in the phase code must exist, even if they do nothing, although they could inherit from an *EmptyVisitor* class.
- All *visit*() methods in the phase code must take node types.

- If our implementation language has **reflection** (Java does; it's also called **introspection**), we can use single dispatch. Example (don't worry about its details): Figure 7.24 page 269

61 L-values and R-values

- The L-value of an identifier (typically a variable) is the location of the identifier. This information is needed if the identifier is on the left-hand side of an assignment.
- The R-value of an identifier is the contents (current value) of the identifier. This information is needed if the identifier is on the right-hand side of an assignment or the predicate of an **`if`** or **`while`**.
- Special cases
  - Constants and **`this`** have only R-values.
  - In C, the L-value of any variable can be derived by the **referencing `&`** operator.
  - In C, any R-value can be used as an L-value by the **dereferencing `*`** operator.
- One can always derive the R-value from the L-value, but not vice-versa.
- In our AST, an identifier node will always mean its L-value.
- If we want an R-value, we will get it by explicitly adding a dereferencing node. Figure 7.22 page 263.
- Confusing example: grammar for C Figure 7.20 page 261 with its semantic actions Figure 7.21 page 262 and some sample ASTs Figure 7.22 page 263. Parse `deref addr` $\texttt{id}_a$ `=` `deref` $\texttt{id}_b$.

62 The Java Virtual Machine (JVM)

- Class 24, 11/10/2020
- The JVM runs bytecode packaged in class files.
  - The bytecode is designed to be compact: mostly 0-address, referring to an implicit stack.
  - The bytecode is designed to be safe: a bytecode verifier checks the program for type errors and stack errors before running it.
  - The *Jasmin* assembler has its own syntax, which we use here.
  - You can run `javap -c` on a class file to disassemble it.
- Types are described by strings. Figure 10.4 page 400.
- Constants are referred to by an index in a "constant pool". Entries are typed and have whatever length is required, but only use one 16-bit index.
- List of operations: `https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html`
- Arithmetic instructions: pop two elements, push one result. Versions for **`integer`**, **`long`**, **`float`**, and **`double`**. Examples: **`iadd`**, **`dmul`**. book page 401 Shorter types use wider arithmetic but lose precision during store.
- Registers: 32 bits
  - As many as needed; the compiled code for each method indicates how many it needs.
  - Allocated in the method's stack frame when the method is invoked.
  - Used for parameters (starting with register 0) and local variables (immediately following parameters).
  - In non-static methods, parameter 0 is implicitly **`this`**.
  - Registers are untyped, but the bytecode verifier makes sure that types are not misused.
  - Load (from register to stack) and store (from stack to register) instructions. Short versions exist for registers $0 \dots 9$. Examples: **`iload`**, **`iload_3`**, **`fstore`**. book page 402

  - **`double`** and **`long`** vales use two registers, which must be even-odd.
  - **`aload`** (for object references, which use one register).
  - Integer operations used for **`boolean`**, **`char`**, **`byte`**, and **`short`**.
  - loading a constant literal (int, float, literal string): **`ldc`**

- Class 25, 11/12/2020
- Type conversion: pop one element, push one result. Example: **`i2f`**.
- Getting fields (variables) of a class
  - **`getstatic`** *name type* book page 404
    - *name* is like `java/lang/System/out`
    - *type* is like `Ljava/io/PrintStream;` (the semicolon is required!)
    - The instruction is only 3 bytes: the opcode (0xb2), and a 16-bit index into the constant pool, which contains both the name and type.
    - Similar: **`putstatic`** (pops a value from stack)
  - **`getfield`** and **`putfield`** book page 405
    - pop an extra element from stack: reference to instance
- Calling a static (class) method book page 407
  - **`invokestatic`** *name*
  - *name* is like `java/lang/Math/pow(DD)D`. The instruction points to the name in the constant pool.
  - Parameters to the method are on the stack (left-most parameters are deepest); they are all popped.
  - Result value (if any) placed on the stack at completion.
- Calling a non-static (instance) method
  - **`invokevirtual`** *name*
  - Same as **`invokestatic`**, but first push the instance on the stack (before the parameters). This parameter is **`this`** in the called routine.
  - We rely on the JVM to find the right version of the routine.

- Calling constructors
  - **`invokespecial`** *name*
  - Same as **`invokevirtual`**, but a reference to an uninitialized object (usually created by **`new`**) is on the stack.
  - *name* is like `java/lang/String/<init>()Ljava/lang/String;`
  - Apparently, **`invokespecial`** is also used for calls in **`super`** and calls to **`private`** methods, but it is unclear why. In fact, it is unclear why one needs to use **`invokespecial`** for constructors.
- Branching book page 406
  - three bytes: opcode, 16-bit Δ.
  - five bytes: opcode, 32-bit Δ.
  - unconditional: **`goto`**.
  - conditional based on TOS (which is then popped): **`ifgt`**, **`ifeq`**, **`ifne`**, **`iflt`**, **`ifle`**, **`ifgt`**.
  - conditional based on two stacked values (both popped): **`if_ifcmpgt`**, **`if_ifcmplt`**, **`if_ifcmpeq`**, **`if_ifcmpne`**, **`if_ifcmpge`**, **`if_ifcmple`**.
- Stack operations
  - **`dup`** duplicates the TOS (32-bit value); **`dup2`** duplicates the TOS (64-bit value, stored in two cells).
  - **`pop`**, **`swap`**: obvious semantics.
  - **`dup_x1`**: duplicates the TOS to a position 3 below TOS; useful for multiple assignment statements.

63 Example of Jasmin code

```
.class public test ; class header
.super java/lang/Object
.field static i I ; static members of the class
.field static j I
.field static c C
.method public static main([Ljava/lang/String;)V
.limit locals 2 ; 0: param. 1 ...: locals
    ldc 100
    putstatic test/j I ; j = 100
    getstatic test/j I
    istore 1           ; localint = j
.limit stack  10     ; only really needed 4
.end method
```

64 Chapter 8: Symbol Tables

- Class 26, 11/17/2020
- After the compiler builds the AST, it can run a pass across the AST to build the **symbol table** (ST).
  - process declarations, inserting symbols in the ST
  - process references, replacing symbols with ST references
- API for the ST is exactly what we used in Project 1: *openScope*(), *closeScope*(), *enterSymbol*(), *retrieveSymbol*(), and *declaredLocally*().
- We'll assume usual **static scoping**.
  - Accessible names at any point of the program are those declared in the current scope and surrounding (**open**) scopes.
  - References are resolved by the innermost declaration.
  - New declarations always pertain to the current (innermost) scope.
  - The language may have a way of specifying specific scopes, such as **`extern`** and compilation-unit in C, qualified names and package-level scope in Java.

- Organization techniques
  - One ST per scope. Retrieval might need multiple lookups. But organization is most straightforward, and the expected depth is fairly low.
  - One unified ST, with fields:
    - *name*: pointer to string space (start index, length)
    - *type*: pointer to type information
    - *hash*: link to previous symbol hashing to same place
    - *var*: pointer to next outer declaration of same name; effectively a stack of scope declarations for this name.
    - *level*: link to next symbol in same scope, to assist in deleting all symbols when a scope is abandoned.
    - *depth*: nesting depth, useful for preventing duplicates at the same level.
  - This unified ST is accessed in two ways
    - hashing the name, linked through the *hash* field.
      - *delete*(symbol): removes symbol from its hash chain
      - *add*(symbol): adds the symbol to its hash chain
    - scope display: a stack of scopes, each of which points to the first symbol of that scope, linked through the *level* field.
- Pseudo-code Figure 8.7 page 292
- Example 8.8 page 293, using Figure 8.1a page 291

65 Dealing with language-specific concerns

- Nested structures, such as **`record`**s.
  - The structures can be arbitrarily deep, so they need a tree representation, with children of a node being its fields.
  - Instead, a structure can be a hash table, with children being its entries.
- **`typedef`** in C: an alias for a type. The scanner can check all identifiers in the ST and return a different token for a **`typedef`** identifier to make parsing possible.
- Identifier overloading

  - Method overloading is based on type signature (C++, Java)
    - Can put the type signature along with the name in the ST: `"foo(int) void"` But how do you compute the right key at the calling point?
    - Can place just the name in the ST, but pointing to a list of signatures; the AST for a procedure call refers to the entire list.
  - Operator overloading (C++, Ada): every reference to an operator must be checked in the ST.
  - Literal overloading (Ada: enumeration literals)
  - Variable semantics overloading (Pascal, Fortran): a method name refers to the return value if used in an L-context. Use two entries in the ST.
  - Identifier-kind overloading (C: same name can be used as a variable, **`struct`** name, label; Perl: same name can be used as a scalar, hash, array, and procedure).
  - Extension overloading (C++, Java: subclasses)

- Implicit declarations: (C: labels; Fortran: variables). Put them in the ST as they are encountered (to avoid duplicates, store semantic information).
- Export and import directives
  - export (Java: **`public`**; C: all methods unless **`static`**).
  - import (C, C++: header file; Java: **`import`**; Ada: **`use`**). These import directives can be used to pre-load the ST.
- Altered search rules
  - Pascal: **`with`** statement
  - forward references: might require two passes: noting the forward references, checking that they are resolved.

66 Processing declarations

- Store attributes of the declared identifier in the ST
  - For each variable, named constant, named type, method, store a (pointer to a) **type descriptor**.

  - The type descriptor has variants for different kinds of types: integer, array (holds element type, index type, bounds), record (holds fields)
  - The AST entry for an identifier can point to its ST entry, or it can simply store the relevant information, such as type.

- Using a visitor pattern (for the book's way to factor the program)
  - *TypeVisitor* extends *TopDeclVisitor* extends *SemanticsVisitor* extends *NodeVisitor* extends *Visitor*.
  - Each of these has a *visit*() method for each relevant type of AST node.
  - Figure 8.11 page 302

67 Semantics checking for AST nodes

- General steps when visiting a node
  - If appropriate, put information in the symbol table, especially for declaration AST nodes.
  - Check local semantics conditions
  - If appropriate, modify the AST structure, particularly for introducing type coercions.
  - If appropriate, store information in the AST node, particularly type information (for expression nodes) and exception information (for statement nodes). One discovers what information needs to be stored as one proceeds in writing the code.

68 Particular AST nodes, extensions of StatementNode

- `IfNode`
  - three children (condition:ExpressionNode, thenPart:StatementNode, elsePart:StatementNode(can be empty)).
  - Semantic check for condition type.

  - Resulting exception information: union of exceptions thrown by the thenPart and elsePart.

- `WhileNode`
  - two children (condition:ExpressionNode, body:StatementNode).
  - Semantic check for condition type.
  - Resulting exception information: same as for body.

- `ForNode`
  - four children (initialization:StatementNode, condition:ExpressionNode, next:StatementNode, body:StatementNode).
  - In C, every statement returns a value, so the condition is actually a StatementNode.
  - If the initialization has a declaration, open a new symbol-table scope with that identifier, close it on return from this AST; don't visit the body until that scope has been opened.
  - If the condition is empty, can insert a BooleanLiteral node with value **`true`** as the condition.
  - Resulting exception information: same as for body.

- `LabelledNode`
  - Two children (label:IdentifierNode, body:StatementNode).
  - Semantic check: label not yet in local scope.
  - Action: put label in scope (kind: label); don't visit the body until the label is in the scope. (It's actually more complex: avoid conflicting name, allow **`break`**/**`continue`** to this label only within the body.)
  - Resulting exception information: same as for body.

- `ReturnNode`
  - One child (value:ExpressionNode (can be empty)).
  - Semantic check: If current function has a non-void return type, value cannot be empty and must have a compatible type. May need to add a coercion node. If current function has a void return, value must be empty.

- To find the return type: a pseudo entry in the symbol table inserted when the function is declared holding the return type.
- Resulting exception information: empty.

- `SwitchNode`
  - three children (condition:ExpressionNode, branches:List<BranchNode>, default:StatementNode(can be empty)).
    - BranchNode: two children (label:ExpressionNode, body:StatementNode).
    - Semantic check: label has a static value.
    - Resulting information: type (of the label), value (of the label), exception information from the body.
  - Semantic check: all branches have different values, all have the right type.
  - Resulting exception information: union of exceptions of all branches and the default.

69 Three kinds of semantic visitors

- type correctness: verify that he types of all components of a node conform to language rules. For instance, the condition of an **`if`** must be Boolean. *SemanticsVisitor* (perhaps better called *TypecheckVisitor*, but it also does some other semantic correctness checking).
- reachability and termination: necessary in Java and C# to determine if a construct terminates normally; optional in C and C++. *ReachabilityVisitor*.
- exception handling: if a construct can throw an exception, it must either be caught or listed in the method's throw list (Java). Each AST node containing a statement or expression has a *throwsSet* field. *ThrowsVisitor*.

70 Types

- Class 27, 11/19/2020

- When there is a type error, it is best to associate the variable with a *TypeDescriptor* called *errorType* to prevent cascading errors.
- Type equivalence (for purposes of assignment and formal-actual binding)
  - Name equivalence (Ada, Pascal): Two identifiers are of the same type if they share the same type descriptor in the ST.
  - Structural equivalence (C, C++): Two identifiers are of the same type if a traversal of the type trees is identical. One can store a digital signature (hash) of a stringification of the traversal to make static checking very easy.
- modifiers (like **`public`**, **`const`**) should be stored with identifiers in the symbol table.
- initialized variables: can introduce a new assignment node, or can make the initial value a child of the variable.
- **`array`** types: store base type, index type, length (or range), perhaps for multiple dimension.
- **`struct`** types: store a hash of fields (like a scope), each with a type and an offset from start.
- **`enum`** types: store a hash of values, each with a numeric value

71 Class declarations

- Children (name:IdentifierNode, modifiers:List<modifier>, parent:IdentifierNode, fields:Map<FieldDeclaration>, constructors, methods:Map<MethodDeclaration>)
- Semantic checks
  - name is unique. Put in current scope.
  - open a new scope.
  - Two passes: first put all fields and methods in the symbol table only as ”prototypes” (checking for unwanted duplicates), then visit them all.
  - pop the scope; it is still pointed to by parts of the AST we have visited.
- To access the current class (needed for validity checks, for instance): perhaps put **`this`** in the symbol table.

72 Method declarations

- Children (name:IdentifierNode, parameters:List<parameter>, returnType:TypeNode);
- Semantic checks
  - open a new scope.
  - put ”return” as a pseudo-identifier in the symbol table with its type (possibly **`void`**).
  - verify that all parameter names are unique. Put each in the symbol table with its type (and any modifiers, such as **`readonly`**).
  - visit the body
  - pop the scope; it is still pointed to by parts of the AST we have visited.

73 Reachability Visitor

- In Java, it is invalid to have an unreachable statement.
- Complete confidence is equivalent to solving the halting problem, so we take a conservative approach: If we are sure a statement is unreachable, it is erroneous.
- Add two Boolean fields to the ASTs for statements (S) and statement lists (SL).
  - *isReachable*: generate an error message for any statement for which this Boolean is false.
  - *terminatesNormally*: generally true, but false for **`break`**, **`return`**, **`continue`**, and loops that a conservative analysis shows can never terminate.
- Rules
  - *isReachable* is inherited by the first S of a SL; *terminatesNormally* is synthesized from the last S to its SL.
  - *isReachable* is true for the SL comprising the body of a method (or constructor or static initializer).

  - *terminatesNormally* is true for local variable declarations and for expression statements (assignments, method calls, increment or decrement), even if the statement itself is not reachable (avoiding cascading errors).
  - A null S or SL never generates an error, but its *isReachable* is propagated to its successor.
  - *terminatesNormally* is propagated from every S to *isReachable* of S's successor.
  - **`return`**, **`break`**, and **`continue`** propagate false for *isReachable*.
  - a value-returning function must end with *isReachable* false (or it didn't properly return!)
  - Example: page 346
  - Reachability visitors: Figure 9.3 page 349, plus the figures that it refers to.
  - `IfNode`: terminates normally if either branch terminates normally.
  - `WhileNode`: does not terminate normally if the condition is statically **`true`**. Body is not reachable if the condition is statically **`false`**. Otherwise synthesizes *terminatesNormally* from the body.

74 Throws visitor

- We concentrate on Java; other languages are similar.
- Exceptions are typed; they must derive from `Throwable`.
- *Checked* exceptions (descendent of `Exception`) must be caught or explicitly propagated.
- *Unchecked* exceptions (descendent of `RuntimeException` or `Error`) may be handled; if not, they terminate execution. They usually indicate errors that cannot be usefully handled.
- Basic visitors: Figure 9.4 page 351. Two purposes:
  - Discover the `throwsSet` for each node of the AST.
  - Discover any exception-related semantic errors.
- Generally, just visit children, collect their `throwsSet`, store it. (**`if`**, **`while`**, **`for`**)

- For some nodes, the `throwsSet` is empty. **`continue`**, **`break`**.
- `TryNode`
    - Three children (body:StatementNode, catches:List<CatchNode>, finally:StatementNode(can be empty)).
        - CatchNode: three children (ident:IdNode, type:TypeNode, body:StatementNode).
        - Semantic check: type is a catchable type.
        - Open a new scope, put the ident in it, before visiting the body.
        - Resulting exception information: from the body.
    - Semantic check
        - Maintain a "throws" list.
        - Initially: the exception list of the body.
        - Each catch node: the type must be in the throws list. Remove it and any subtypes from the throws list.
        - Add to the throws list the exception information from the catch node.
    - Resulting exception information: union of throws list and the exception information from the finally statement.
- `TryNode` visitors: Figure 9.28 page 374. for **try**:

```
void visit(tryNode tn) {
  visit(tn.body);
  tn.throwsSet = tn.body.throwsSet;
  for (c: tn.catches) {
    tn.throwsSet =
      reduce(tn.throwsSet, c.decl.type);
    visit(c.body);
    tn.throwsSet ∪= c.body.throwsSet;
  } // each catch block c
  visit(tn.theFinally);
  tn.throwsSet ∪= tn.theFinally.throwsSet;
} // visit
set reduce(set full, type t) {
  removed = false;
  answer = ∅;
  for (element:full) {
    if subsumes(t, element) removed = true;
    else answer ∪= {element};
  } // each element in full
  if (not removed) {
    error("catch␣block␣doesn’t␣catch␣anything");
  }
  return answer;
} // reduce
```

75 Java overloaded method resolution

- This section is taken from `https://coderanch.com/t/417622/certification/Golden-Rules-widening-boxing-varargs`.
- Rules
  - 1. Primitive Widening (W) > Boxing (B) > Varargs (V)
  - 2. Widening and Boxing (WB) not allowed.
  - 3. Boxing and Widening (BW) allowed.
  - 4. WV and BV can only be used in a mutually exclusive manner.
  - 5. Widening between wrapper classes not allowed.

- Examples

first	second	call	result	rule
foo(Integer i)	foo(long l)	foo(5)	second	1
foo(int...i)	foo(Integer i)	foo(5)	second	Rule 1
foo(Long l)	foo(int...i)	foo(5)	second	1, 2
foo(Long l)	foo(Integer...i)	foo(5)	second	1, 2
foo(Object o)	foo(Long l)	foo(5)	first	2, 3
foo(Object o)	foo(int...i)	foo(5)	first	1, 3
foo(Object o)	foo(long l)	foo(5)	second	1, 3
foo(long...l)	foo(Integer...i)	foo(5)	ambiguous	4
foo(long...l)	foo(Integer i)	foo(5)	second	1
foo(Long l)		foo(Integer(5))	error	5
foo(Long l)	foo(long...l)	foo(Integer(5))	second	1, 5

76 Semantic analysis of calls

- Class 29, 11/24/2020
- The Java compiler needs to discover the signature of any method the program calls. The JVM dynamically discovers the class that contains that method.
- To staticly resolve which method *m* is intended, the compiler finds the nearest, tightest, accessible method definition in the inheritance hierarchy whose formal parameters match the actual parameters in number and type. In Java, we don't consider the return types, although in Ada we do.
- Accumulate a list of all methods called *m* in the *starting class* and in all its superclasses.
    - For methods in the calling class, the list ignores modifiers; for methods elsewhere, it only includes **`public`** and **`protected`** methods.
    - If there is a qualifier, such as *T.m()*
        - *T*=**`super`**, the starting class is the superclass of the class in which the call is made.
        - If *T* is an expression computing an instance of a public class *C*, then *C* is the starting class.
        - If *T* is a class, then the list only includes static methods; *T* is the starting class.

  - If there is no qualifier, then the starting class is the calling class.

- Filter the list of possible methods by removing those with unacceptable parameters. The parameters are acceptable if all actuals are *bindable* to the formals.
- If no methods remain, the call is invalid.
- If several methods remain, filter the list by retaining only those in the class earliest in the list (closest to the starting class).
- If there are several methods, filter the list by removing all entries that are **loose**: all the parameters of a loose method generalize (are assignable from) the parameters of another.
- If there are still several methods, use the rules above in ”Java overloaded method resolution”.
- If there are still several, the call is ambiguous and therefore invalid.
- The resulting method's signature is then used in **`invokespecial`** or **`invokestatic`**.
- Code: Figure 9.34 page 380, and its utility routines on the previous pages.

77 Chapter 10: Intermediate representations

- Many compilers use an **intermediate representation** (IR), often formalized as an **intermediate language** (IL).
  - Reduces the cost of retargeting, unit testing, prototyping new ideas, building reference implementations.
- The Gnu compilers (such as *gcc*) use Register Transfer Language (RTL). See `gcc.gnu.org/onlinedocs/gccint/RTL.html`.
- The compilers for Microsoft's .NET use Common Intermediate Language (CIL).
- The Java Virtual Machine (JVM) (brief review)
  - Principles
    - Compact: uses mostly 0-address instructions, has short form of common instructions

  - Safe: code can be statically checked (at class-load time) to ensure that types are not mixed.
  - The **constant pool**: entries are of any length, are tagged by type, and are indexed by ordinal position, not byte offset.
  - The stack holds 32-bit quantities; two cells are needed for 64-bit quantities (**`double`** and **`long`**).

- Types: book Figure 10.4 page 400

78 Static single-assignment (SSA) form

- A form of IR that helps in program analysis and optimization.
- Variables are assigned only once; if you need a new value, introduce a new variable.
- Call the single assignment a **def** (defining instance); any other use of the variable is a **ref** (referring instance).
- Rename the defs in a program by subscripting the variable name page 411.
- In order to account for multiple paths, introduce a ϕ function when the paths join. page 411.
- Substantial example: Figure 10.5 page 412.
- Purpose: allow for optimizations. See `aggregate.org/OC/`

79 Chapter 11: Generating code for the JVM

- Because we use the JVM, we can avoid details like deferred method binding, how objects are stored, and limits on the number of registers.
- As always, we define several visitors on the AST.
  - *MethodBodyVisitor* generates code for the contents of a method, taking a label where to put the postlude code.
  - *LHSVisitor* generates code for the LHS of an assignment.

  - *SignatureVisitor* visits method definitions and method calls, developing the signature of the associated method, building such strings as `java/lang/String/parseDouble(Ljava/lang/String;)D`.

- Naming conventions for methods

  - *someNode.get*...() accesses information in the AST node.
  - *someNode.accept*(`visitor`) traverses the node with the given visitor, often **`this`**.
  - *emit*...(`variable`) causes generated code of some sort about `variable` to be placed in the output.

- The code in the visitors uses getters and setters for values in the AST to decouple the code from the specific AST representation. For instance, `methodNode` responds to a *getInstance*() and an *isVoid*() call.

80 Class declarations

- page 422
- The name of the class might be a fully qualified path name, or it might be a purely internal name (for inner classes, anonymous classes).
- If there are generic types (as in Java or Ada, or templates in C++), the code generated for one generic type must be distinguished from code generated for another.
- The list of superclasses is emitted to allow deferred method binding to work.
- Every instance variable needs information to be generated about its name, type, and permissions. (Why permissions?)
- Every static variable must have allocated storage, but just once.

81 Method declarations

- Determine the method signature, such as `main([Ljava/lang/String;)V`

- Emit the method **prelude**: name, modifiers, signature, space needed by local variables (discover by looking at the symbol table) and amount of stack needed (for Java, placed at end; determined by data-flow analysis).
- Generate a label for the **postlude** code so throwing an exception jumps to the right place.
- Generate the code for the body, perhaps passing the exit label as a parameter.
- Output postlude code, which might end with the **`return`** operation.

82 Expressions

- Overview Figure 11.2 page 421
- Constants page 425
  - Generate an immediate instruction that loads the constant into an allocated location (for Java, TOS).
  - Abstract the right instruction by *emitConstantLoad*(destination, value).
  - The proper instruction might depend on the value of the constant; a small integer value might use a shorter instruction: **`iconst_m1`** (minus 1), **`iconst_0`** .. **`iconst_5`**, **`bipush`** (byte signed integer push, up to 127), **`sipush`** (16-bit signed integer push, up to 32767), **`ldc`** (in general).
- References to local (method) variables page 426
  - Abstract the right instruction by *emitLocalLoad*(destination, source), where *destination* is an allocated destination (Java: TOS) and *source* is the location (Java: register) holding the value (determined when the local is declared, and stored in the ST).
  - For instance, **`iload`** loads an integer value in the JVM. (You can also use **`iload_0`** .. **`iload_3`**.)
- References to static (class) variables page 427

  - *emitStaticReference*(type, name), where *type* is like `Ljava/io/PrintStream;`, and *name* is like `java/lang/System/out`.

- Operations page 427
  - First generate code for all children. Their results will be in their allocated location, which could be registers, temporary storage on the stack, or (Java) the TOS.
  - Allocate space for the result, such as in a register, temporary storage on the stack, or (Java) the TOS.
  - Emit code to compute the value based on the operand results.
  - Example page 428.

83 Assignment

- Visit the LHS side first.
  - Aside: This evaluation order helps solve the pass-by-name swap riddle.
  - The LHS might contain field references and array-index computations, which themselves can contain arbitrary expressions.
  - We call a different visitor (*LHSVisitor*) to leave the location, not the value, in an allocated space.
  - This visitor might also check that the RHS value conforms to the LHS constraints (such as range limits). But to do that one first needs to know the RHS value.
  - This visitor might generate no code but just store the proper location in the LHS AST node.
- Visit the RHS expression.
- Emit code to cause the computed value to be copied to the computed location.
- Example page 430

84 LHS visitor

- Some parts of an expression on the LHS are loads, not stores. Consider: `a.b[i].c = 5`. There is one store, but there are also an object reference, a field reference, and an array reference.
- Local reference as a target
  - Generate the proper instruction (sequence) for storing into the local; this instruction is saved in the AST assignment node so the method visitor on the assignment node has access to it.
  - Example page 438
- Static reference as a target
  - Very similar, but the instruction is different.
  - Example page 439
- Field reference as a target
  - Evaluate the field instance using the method visitor.
  - Generate the proper instruction (sequence) for storing into the field.
  - Example page 439
- Array reference as a target
  - Generate code for the array reference.
  - Generate code for the index
  - Generate code to access the array at the index
  - Generate code to store into the array at that index.
  - Example page 440

85 Method calls

- page 430
- Call a *SignatureVisitor* to develop the method signature.
- Find what method matches that signature; if several, the call is erroneous. In Java, overload resolution can be complex; the result is the exact signature of the method matched.

- If the method returns a value, allocate space for that value. (Java: no need; TOS is the obvious place.)
- If the method is dynamic, compute the starting-class instance (**`this`**, or based on a variable). This instance is an implicit parameter to the call.
- Compute the values of all actual parameters.
- Call the method by emitting the appropriate (static or dynamic) method call.
- Example page 432

86 Field references

- page 432
- Like a static reference, but there is a computed object.
- Compute the object instance.
- Emit code to get the field.
- Example page 433

87 Array references

- page 433
- Compute the address of the array (Java: always a reference to an array object; C: local, global, or heap); place that address in an allocated location (Java: TOS).
- Compute the array index; place that index in an allocated location (Java: TOS).
- Emit code to check bounds (if needed; in Java, it's not needed)
- Emit code to load the array reference. (Java: specialized instruction; other: may need multiple instructions)
- Example page 434

88 Conditional statements and expressions

- page 435
- Generate two labels, L_1 for the start of the **`else`** part, L_2 for the first instruction after the conditional.
- Generate code to compute the condition.
- Generate a conditional branch (if false) to L_1.
- Generate code for the **`then`** part.
- Generate an unconditional branch to L_2.
- Emit L_1.
- Generate code for the **`else`** part.
- Emit L_2.
- Example page 435 (conditional expression)

89 `while` loops

- Generate two labels, L_1 for the start, and L_2 for the end.
- Emit L_1.
- Generate code to compute the condition.
- Generate a conditional branch (if false) to L_2.
- Generate code for the body.
- Generate an unconditional branch to L_1.
- Emit L_2.
- Example page 436

90 `switch` statement

- If the cases are few, compile the equivalent of an **`if`** statement.
- If the cases are dense, compile a jump table.
- Otherwise, compile a search method, such as a balanced binary tree or a hash table.
- Recent versions of Java allow cases to be Strings, by automatically hashing.

91 Chapter 12: Runtime organization: Allocating space for variables

- Globals: Static allocation. Can use a register to point to area.
- Per-subroutine locals: Dynamic allocation on stack.
  - May require padding to get alignment right.
  - Accessing fields in classes and structs: static offset from start.
  - The start is either on stack (struct) or in heap (object).
- Stack frames
  - One per invocation
  - Fixed control information
    - return address (in code space)
    - dynamic pointer (start of previous stack frame)
    - static pointer (start of some earlier stack frame)
    - place for return value
  - Parameters
  - Local variables
  - Temporaries (how much space is known to the compiler)
  - Subclasses
    - shared fields in same offsets as in parent class
    - additional fields at the end.
  - Multiple open scopes: nested procedures
    - static chain
  - Block-level allocation
    - **procedure-level frame allocation** is usually preferred; also called **scope flattening**.
  - Closures
    - necessary for procedures passed as parameters, first-class procedures if procedures may access non-local variables, and also for second-class labels.
    - represented by a pair: (pointer to code space, pointer to non-local referencing environment)

    - for a label, the second item is the local referencing environment.
  - Cactus stacks
    - necessary for threads sharing non-local variables.
    - usually, frames are allocated from the heap, so allocation and deallocation are more expensive than for ordinary stacks.

- Arrays
  - Static 1-dimensional arrays
    - index calculation is straightforward, based on size of elements and index, possibly lower bound.
    - Element size might need to be rounded up to a reasonable boundary, for instance, if the element is a record starting with a 64-bit field.
  - Bounds-checking code
    - Sometimes entirely compile-time, if subscript is a compile-time constant.
    - Usually, the code uses compile-time information (array bounds) and runtime information (subscript).
    - Sometimes, even the array bounds are only known dynamically. Then a **type descriptor** (often called a **dope vector**) needs to be available at runtime.
    - Types belong to scopes in Pascal; the type descriptor therefore needs to be in the frame.
  - Flex arrays: elements exist if they have been assigned into (Perl, Java `Vector`)
    - Usually allocated from the heap, recopied into larger area if needed.
  - Closures:
    Example from programming languages design: p 24
  - Multidimensional arrays
    - easiest to treat as arrays of arrays and reduce the discussion to 1-dimensional arrays.
    - usually stored in **row-major** order: The neighbor of an element is the element whose last index differs by 1.

    - if the language (Fortran) requires **column-major** order, the index computation is identical, but with the dimensions sorted backwards.

- Heap management
  - Boundary tag method
    - Each free region has a tag at front and end showing size, so it is easy to find the next region for purposes of merging.
    - Free regions are also linked together into a free list.
    - Allocation can use first fit, best fit, or circular versions of these.
  - Buddy method
    - The jth free list has free blocks of size 2^j. (The list pointers are inside these blocks).
    - Allocating n bytes always allocates $n' \lceil 2^{\log_2 n} \rceil$ bytes.
    - If free list n' is empty, break a block of larger size.
    - During deallocation, merge a block with its buddy (from which it split) if the buddy is free.
    - Linux uses the buddy method as its major allocator.
  - Slabs (called **subheaps** in the book)
    - popular sizes (as determined usually at runtime) are allocated arrays of that size with a bitmap showing which ones are free. Allocation and deallocation are therefore very fast.
  - Deallocation techniques
    - manual, by `free`(). Very error prone.
    - reference counts. Requires extra instructions on every array assignment. Cannot deallocate circular structures.
    - garbage collection
      - mark all reachable items
      - collect the rest into free space.
      - can be done concurrently.
      - compaction is sometimes possible, but all pointers must be adjusted. A level of indirection might help.
      - copying collectors: move live objects to the other space, then switch spaces.

- generational garbage collection: objects advance after surviving several collections to the next space, which is collected less frequently.

92 Chapter 13: Target-code generation

- We can assume we already have bytecode or something like it.
- Target-code generation is very target-dependent.
- We assume arithmetic is done out of registers.
- Example: Figure 13.2 page 492
- General idea
  - bytecode **`push`** ⇒ load from frame to register
  - Choose registers in increasing order; restart after statement.
  - bytecode **`pop`** ⇒ store from register to frame
  - Various optimizations are possible
    - Do arithmetic directly from values in the frame
    - Delay loading constants; maybe an immediate-operand arithmetic instruction will suffice
- Allocating objects
  - Java **`new`** ⇒ heap allocation followed by initialization. The size is known at compile time based on size of superclass and sizes of new members (both instance variables and methods).
  - Allocate on current frame if no reference **escapes** the current scope. The compiler must check that no reference is returned or assigned to a field.
  - Immutable literal objects (such as `String` literals) can be allocated statically. **Interning** strings means storing each distinct string literal only once; Java usually interns strings, which means that pointer equality (between constants!) works.
  - Array subscript calculation should often include bounds checking. Figure 13.3 page 495
- Method calls
  - The **linkage** is often determined by the target architecture.

- Need to decide how parameters are passed: Typically in registers for a few, then in a structure or on the stack for the extra ones.
- Need to decide which registers are **volatile**. These are usually saved (if in use) by the caller in the method **prologue**.
- Non-volatile registers are usually saved by the callee (if needed).
- The callee restores non-volatile register contents and restores the stack in its **epilogue**.
- Example Figure 13.5 page 497
- Dynamic resolution: to find address of method within an object, each object can have a hidden field that indexes a method table with offsets.

- Expression trees
  - The number of registers is at most the depth of the tree.
  - If two subtrees have different depths, compute the deeper one first (if semantics allow).
  - Hold the result of the first calculation in a register while computing the other one reusing the other registers. We save one register.
  - If there are inadequate registers, we **spill** registers into temporary locations (on the stack) to make them available.
  - The recursive generalization is called Sethi-Ullman numbering. Example: Figure 13.10 page 503

93 Register allocation using graph coloring

- Instead of statement-level, allocate at the procedure level.
- **Register candidates**: local and global variables, constants, parameters, return values.
- Variables that do not interfere with each other can share a register.
- **Live range** of a value: span of instructions in which that value can be accessed, from initial creation to last use.

  - For variables, the live range is from assignment (or initialization) to last use of that value; a single variable may have multiple independent live ranges, each a separate register candidate.
  - The SSA form makes it easy to compute the live ranges of variables.
  - Or each variable, parameter, or constant can be a distinct register candidate.

- The interference graph
- Two live ranges **interfere** with each other of they overlap.
- Nodes in the graph: live ranges
- Edges: interferences.
- Example: book Figure 13.15 page 510
- Find an n-coloring of the graph. Each color represents a register.
- Is there an n-coloring? NP-complete. So use a heuristic.
- If the graph is not n-colorable, where n is the number of registers, repeatedly simplify it by assigning some live range a non-register location.
  - heuristic: choose the live range with smallest cost/degree in the graph, where cost is number of accesses times 10*loop depth.
  - Example: book Figure 13.15 page 510
- There are other heuristics.

94 Code scheduling

- Purpose: avoid stalling in a pipelined architecture
- Values produced by an instruction might not be available immediately, only c cycles later. Each architecture has its own rules.
- A **basic block** is a linear sequence of instructions with no branches except at the end. We schedule code only within a basic block.
- **Postpass** code scheduler: reorder instructions after they have been generated.

- Dependency DAG: nodes are instructions, edge from A to B if A must complete before B starts. Be conservative: load through a pointer (A) must happen before any possible aliased store.
- See book Figure 13.21-24 page 520.
- Any topological sort of the DAG is an acceptable execution order.
- Pick as the next instruction one that:
  - Is not stalled by the previous instruction,
  - Preferring ones that can stall some successor,
  - Preferring ones that expose the most new nodes if generated,
  - Preferring ones that have the longest path to a leaf.

95 Peephole optimization

- Peephole optimization is a separate compiler stage.
- It can work on the AST, bytecode, or target code.
- If a known pattern is seen, replace it with a standard replacement.
- some examples book Figures 13:32-34 page 534