CS 536

Introduction to Programming Languages and Compilers

Charles N. Fischer

Fall 2012

http://www.cs.wisc.edu/~fischer/cs536.html

Class Meets

Mondays, Wednesdays & Fridays,
11:00 — 11:50

204 Educational Sciences

Instructor

Charles N. Fischer

5393 Computer Sciences

Telephone: 608.262.1204

E-mail: fischer@cs.wisc.edu

Office Hours:

10:30 - Noon, Tuesday & Thursday,
or by appointment

Teaching Assistant

James Paton
1309 Computer Sciences
E-mail: paton@cs.wisc.edu
Telephone: 608.262.1204
Office Hours:
Wednesday & Friday:
2:30 - 3:30
or by appointment

Key Dates

- September 19: Assignment #1
  (Identifier Cross-Reference Analysis)
- October 10: Assignment #2
  (CSX Scanner)
- November 2: Assignment #3
  (CSX Parser)
- November 9: Midterm, 11-1
  (in class)
- November 21: Assignment #4
  (CSX Type Checker)
- December 14: Assignment #5
  (CSX Code Generator)
- December 17: Final Exam
  2:45 pm - 4:45 pm

Class Text

- **Crafting a Compiler**

  Fischer, Cytron, LeBlanc

  ISBN-10: 0136067050

  ISBN-13: 9780136067054

  Publisher: Addison-Wesley
- Handouts and Web-based reading will also be used.

Reading Assignment

- Chapters 1-2 of **CaC** (as background)

Class Notes

- The lecture notes used in each lecture will be made available prior to that lecture on the class Web page (under the "Lecture Nodes" link).

Instructional Computers

Departmental Linux machines (mumble-01to mumble-40) have been assigned to CS 536. All necessary compilers and tools will be available on these machines. best-mumble will connect you to a lightly used machine.

You may also use your own computer. It will be *your* responsibility to load needed software (instructions on where to find needed software are included on the class web pages).

Academic Misconduct Policy

- You must do your assignments—no copying or sharing of solutions.
- You may discuss general concepts and Ideas.
- All cases of Misconduct *must* be reported to the Dean's office.
- Penalties may be severe.

Program & Homework Late Policy

- An assignment may be handed in up to one week late.
- Each late day will be debited 3%, up to a maximum of 21%.

Approximate Grade Weights

Program 1 - Cross-Reference Analysis 8%

Program 2 - Scanner 12%

Program 3 - Parser 12%

Program 4 - Type Checker 12%

Program 5 - Code Generator 12%

Homework #1 6%

Midterm Exam 19%

Final Exam (non-cumulative) 19%

Partnership Policy

- Program #1 and the written homework must be done individually.
- For undergraduates, programs 2 to 5 may be done individually *or* by two person teams (your choice). Graduate students must do all assignments individually.

Compilers

Compilers are fundamental to modern computing.

They act as *translators*, transforming human-oriented *programming languages* into computer-oriented *machine languages.*

To most users, a compiler can be viewed as a "black box" that performs the transformation shown below.

A compiler allows programmers to ignore the machine-dependent details of programming.

Compilers allow programs and programming skills to be *machine-independent* and *platform-independent*.

Compilers also aid in detecting and correcting programming errors (which are all too common).

Compiler techniques also help to improve computer security. For example, the Java Bytecode Verifier helps to guarantee that Java security rules are satisfied.

Compilers currently help in protection of intellectual property (using *obfuscation*) and provenance (through *watermarking*).

History of Compilers

The term *compiler* was coined in the early 1950s by Grace Murray Hopper. Translation was viewed as the "compilation" of a sequence of machine-language subprograms selected from a library.

One of the first real compilers was the FORTRAN compiler of the late 1950s. It allowed a programmer to use a problem-oriented source language.

Ambitious “optimizations” were used to produce efficient machine code, which was vital for early computers with quite limited capabilities.

Efficient use of machine resources is still an essential requirement for modern compilers.

Compilers Enable Programming Languages

Programming languages are used for much more than "ordinary" computation.

- TeX and LaTeX use compilers to translate text and formatting commands into intricate typesetting commands.
- Postscript, generated by text-formatters like LaTeX, Word, and FrameMaker, is really a programming language. It is translated and executed by laser printers and document previewers to produce a readable form of a document. A standardized document representation language allows documents to be freely interchanged, independent of how

they were created and how they will be viewed.

- Mathmatica is an interactive system that intermixes programming with mathematics; it is possible to solve intricate problems in both symbolic and numeric form. This system relies heavily on compiler techniques to handle the specification, internal representation, and solution of problems.
- Verilog and VHDL support the creation of VLSI circuits. A *silicon compiler* specifies the layout and composition of a VLSI circuit mask, using standard cell designs. Just as an ordinary compiler understands and enforces programming language rules, a silicon compiler understands and enforces the design rules that dictate the feasibility of a given circuit.

- Interactive tools often need a programming language to support automatic analysis and modification of an artifact.
  How do you *automatically* filter or change a MS Word document? You need a text-based specification that can be processed, like a program, to check validity or produce an updated version.

When do We Run a Compiler?

- **Prior to execution**
  This is standard. We compile a program once, then use it repeatedly.
- **At the start of each execution**
  We can incorporate values known at the start of the run to improve performance.
  A program may be partially complied, then completed with values set at execution-time.
- **During execution**
  Unused code need not be compiled. Active or "hot" portions of a program may be specially optimized.
- **After execution**
  We can profile a program, looking for heavily used routines, which can be specially optimized for later runs.

What do Compilers Produce?

Pure Machine Code

Compilers may generate code for a particular machine, not assuming any operating system or library routines. This is "pure code" because it includes nothing beyond the instruction set. This form is rare; it is sometimes used with system implementation languages, that define operating systems or embedded applications (like a programmable controller). Pure code can execute on bare hardware without dependence on any other software.

Augmented Machine Code

Commonly, compilers generate code for a machine architecture *augmented* with operating system routines and run-time language support routines. To use such a program, a particular operating system must be used and a collection of run-time support routines (I/O, storage allocation, mathematical functions, etc.) must be available. The combination of machine instruction and OS and run-time routines define a *virtual machine*—a computer that exists only as a hardware/ software combination.

Virtual Machine Code

Generated code can consist *entirely* of virtual instructions (no native code at all). This allows code to run on a variety of computers.

Java, with its JVM (Java Virtual Machine) is a great example of this approach.

If the virtual machine is kept simple and clean, its interpreter can be easy to write. Machine interpretation slows execution by a factor of 3:1 to perhaps 10:1 over compiled code.

A "Just in Time" (*JIT*) compiler can translate "hot" portions of virtual code into native code to speed execution.

Advantages of Virtual Instructions

Virtual instructions serve a variety of purposes.

- They simplify a compiler by providing suitable primitives (such as method calls, string manipulation, and so on).
- They aid compiler transportability.
- They may decrease in the size of generated code since instructions are designed to match a particular programming language (for example, JVM code for Java).

Almost all compilers, to a greater or lesser extent, generate code for a virtual machine, some of whose operations must be interpreted.

Formats of Translated Programs

Compilers differ in the format of the target code they generate. Target formats may be categorized as *assembly language*, *relocatable binary*, or *memory-image*.

- **Assembly Language (Symbolic) Format**

  A text file containing assembler source code is produced. A number of code generation decisions (jump targets, long vs. short address forms, and so on) can be left for the assembler. This approach is good for instructional projects.

Generating assembler code supports *cross-compilation* (running a compiler on one computer, while its target is a second computer). Generating assembly language also simplifies debugging and understanding a compiler (since you can see the generated code).

C (rather than a specific assembly language) can generated, treating C as a "universal assembly language."

C is far more machine-independent than any particular assembly language. However, some aspects of a program (such as the run-time representations of program and data) are inaccessible using C code, but readily accessible in assembly language.

- **Relocatable Binary Format**

  Target code may be generated in a *binary format* with external references and local instruction and data addresses are not yet bound. Instead, addresses are assigned relative to the beginning of the module or relative to symbolically named locations. A *linkage* step adds support libraries and other separately compiled routines and produces an absolute binary program format that is executable.

- **Memory-Image (Absolute Binary) Form**

  Compiled code may be loaded into memory and immediately executed. This is faster than going through the intermediate step of link/editing. The ability to access library and precompiled routines may be limited. The program must be recompiled for each execution. Memory-image compilers are useful for student and debugging use, where frequent changes are the rule and compilation costs far exceed execution costs.

Java is designed to use and share classes designed and implemented at a variety of sites. Rather than use a fixed copy of a class (which may be outdated), the JVM supports *dynamic linking* of externally defined classes. When first referenced, a class definition may be remotely fetched, checked, and loaded during program execution. In this way "foreign code" can be guaranteed to be up-to-date and secure.

The Structure of a Compiler

A compiler performs two major tasks:

- Analysis of the source program being compiled
- Synthesis of a target program

Almost all modern compilers are *syntax-directed:* The compilation process is driven by the syntactic structure of the source program.

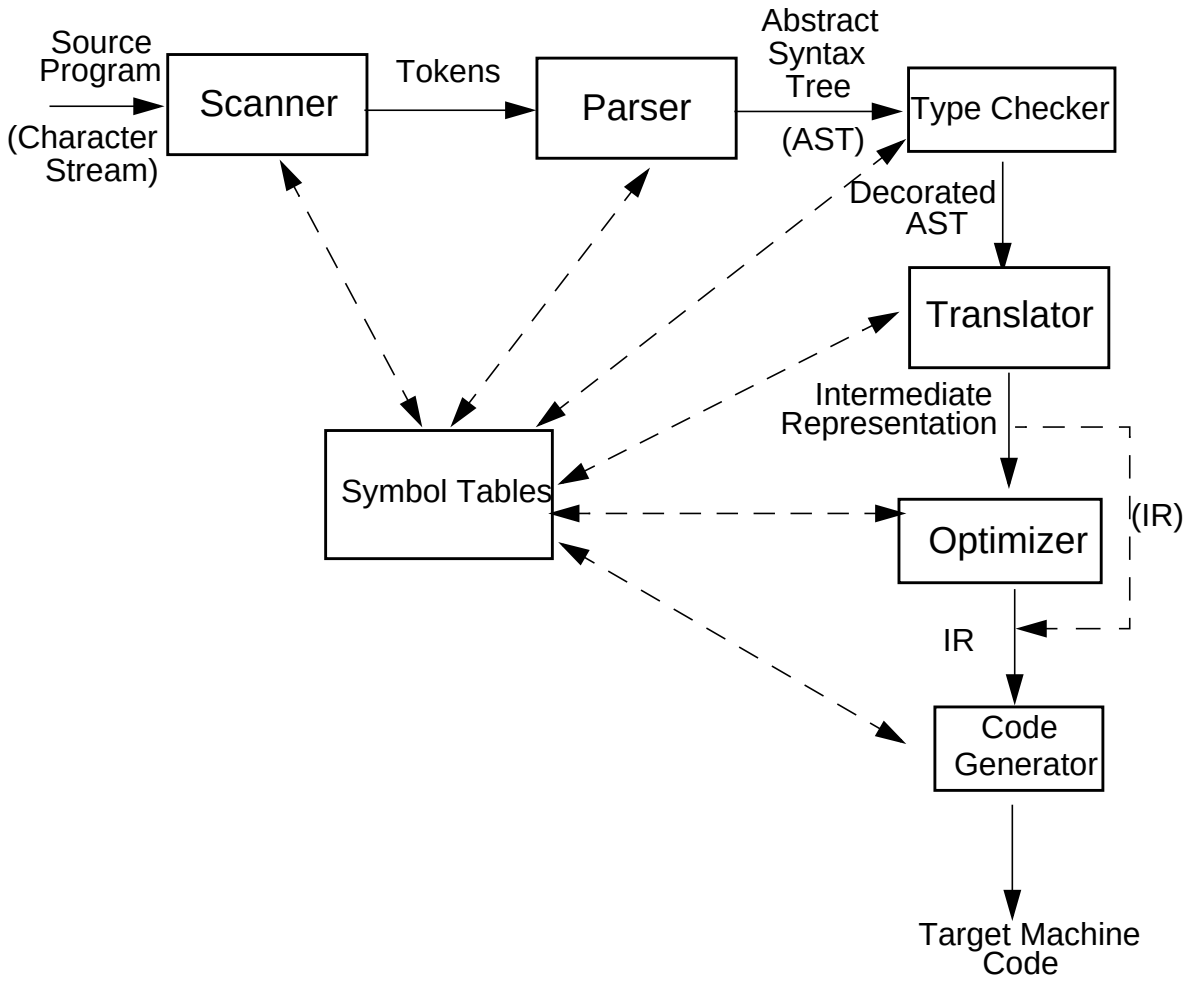
A parser builds semantic structure out of tokens, the elementary symbols of programming language syntax. Recognition of syntactic structure is a major part of the analysis task.

Semantic analysis examines the meaning (semantics) of the program. Semantic analysis plays a dual role.

It finishes the analysis task by performing a variety of correctness checks (for example, enforcing type and scope rules). Semantic analysis also begins the synthesis phase.

The synthesis phase may translate source programs into some intermediate representation (IR) or it may directly generate target code.

If an IR is generated, it then serves as input to a *code generator* component that produces the desired machine-language program. The IR may optionally be transformed by an *optimizer* so that a more efficient program may be generated.



The Structure of a Syntax-Directed Compiler

Scanner

The scanner reads the source program, character by character. It groups individual characters into tokens (identifiers, integers, reserved words, delimiters, and so on). When necessary, the actual character string comprising the token is also passed along for use by the semantic phases.

The scanner:

- Puts the program into a compact and uniform format (a stream of tokens).
- Eliminates unneeded information (such as comments).
- Sometimes enters preliminary information into symbol tables (for

example, to register the presence of a particular label or identifier).

- Optionally formats and lists the source program

Building tokens is driven by token descriptions defined using *regular expression* notation.

Regular expressions are a formal notation able to describe the tokens used in modern programming languages. Moreover, they can drive the *automatic generation* of working scanners given only a specification of the tokens. Scanner generators (like Lex, Flex and JLex) are valuable compiler-building tools.

Parser

Given a syntax specification (as a context-free grammar, CFG), the parser reads tokens and groups them into language structures.

Parsers are typically created from a CFG using a parser generator (like Yacc, Bison or Java CUP).

The parser verifies correct syntax and may issue a syntax error message.

As syntactic structure is recognized, the parser usually builds an abstract syntax tree (AST), a concise representation of program structure, which guides semantic processing.

Type Checker (Semantic Analysis)

The type checker checks the *static semantics* of each AST node. It verifies that the construct is legal and meaningful (that all identifiers involved are declared, that types are correct, and so on).

If the construct is semantically correct, the type checker "decorates" the AST node, adding type or symbol table information to it. If a semantic error is discovered, a suitable error message is issued.

Type checking is purely dependent on the semantic rules of the source language. It is independent of the compiler's target machine.

Translator (Program Synthesis)

If an AST node is semantically correct, it can be translated. Translation involves capturing the run-time "meaning" of a construct.

For example, an AST for a while loop contains two subtrees, one for the loop's control expression, and the other for the loop's body. *Nothing* in the AST shows that a while loop loops! This "meaning" is captured when a while loop's AST is translated. In the IR, the notion of testing the value of the loop control expression,

and conditionally executing the loop body becomes explicit.

The translator is dictated by the semantics of the source language. Little of the nature of the target machine need be made evident. Detailed information on the nature of the target machine (operations available, addressing, register characteristics, etc.) is reserved for the code generation phase.

In simple non-optimizing compilers (like our class project), the translator generates target code directly, without using an IR.

More elaborate compilers may first generate a high-level IR

(that is source language oriented) and then subsequently translate it into a low-level IR (that is target machine oriented). This approach allows a cleaner separation of source and target dependencies.

Optimizer

The IR code generated by the translator is analyzed and transformed into functionally equivalent but improved IR code by the optimizer.

The term optimization is misleading: we don't always produce the best possible translation of a program, even after optimization by the best of compilers.

Why?

Some optimizations are *impossible* to do in all circumstances because they involve an undecidable problem. Eliminating unreachable ("dead") code is, in general, impossible.

Other optimizations are too expensive to do in all cases. These involve NP-complete problems, believed to be inherently exponential. Assigning registers to variables is an example of an NP-complete problem.

Optimization can be complex; it may involve numerous subphases, which may need to be applied more than once.

Optimizations may be turned off to speed translation. Nonetheless, a well designed optimizer can significantly speed program execution by simplifying, moving or eliminating unneeded computations.

Code Generator

IR code produced by the translator is mapped into target machine code by the code generator. This phase uses detailed information about the target machine and includes machine-specific optimizations like *register allocation* and *code scheduling*.

Code generators can be quite complex since good target code requires consideration of many special cases.

Automatic generation of code generators is possible. The basic approach is to match a low-level IR to target instruction templates, choosing

instructions which best match each IR instruction.

A well-known compiler using automatic code generation techniques is the GNU C compiler. GCC is a heavily optimizing compiler with machine description files for over ten popular computer architectures, and at least two language front ends (C and C++).

Symbol Tables

A symbol table allows information to be associated with identifiers and shared among compiler phases. Each time an identifier is used, a symbol table provides access to the information collected about the identifier when its declaration was processed.

Example

Our source language will be *CSX*, a blend of C, C++ and Java.

Our target language will be the Java JVM, using the Jasmin assembler.

- A simple source line is

  `a = bb+abs(c-7);`

  this is a sequence of ASCII characters in a text file.

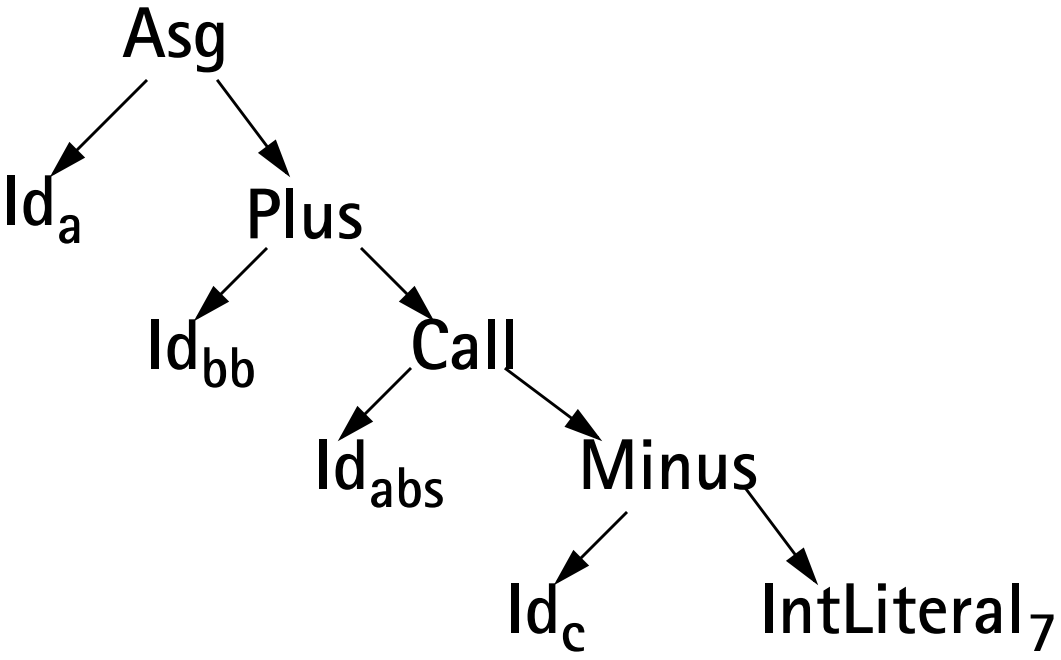
- The scanner groups characters into tokens, the basic units of a program.

  `a = bb+abs(c-7);`

  After scanning, we have the following token sequence:

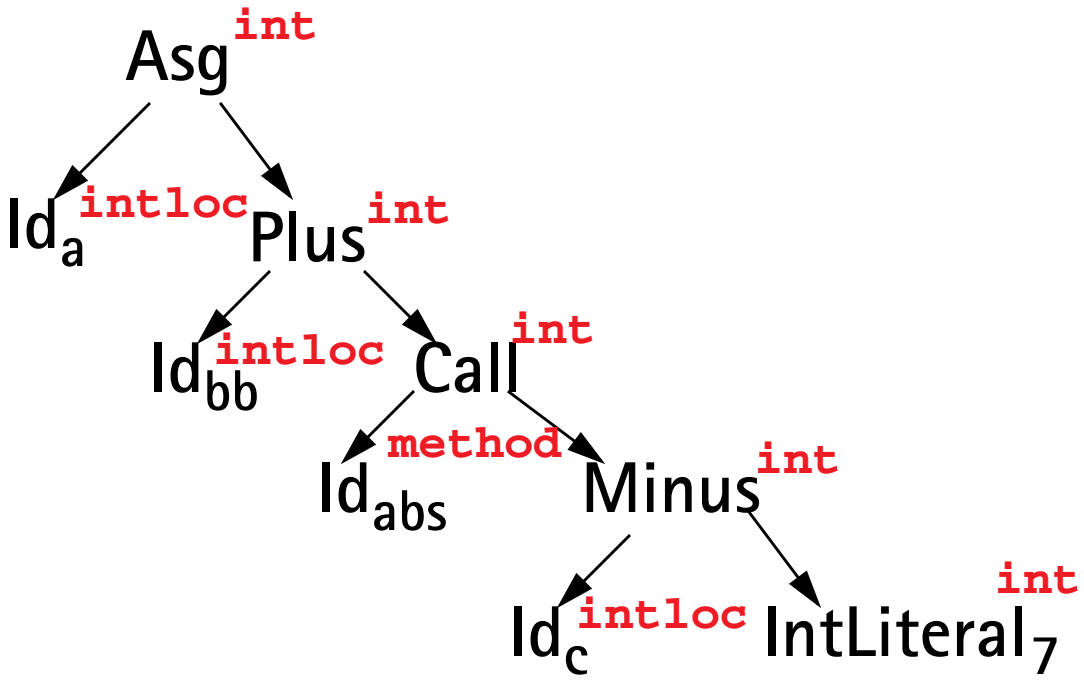
  Id_a Asg Id_{bb} Plus Id_{abs} Lparen Id_c Minus $IntLiteral_7$ Rparen Semi

- The parser groups these tokens into language constructs (expressions, statements, declarations, etc.) represented in tree form:



(What happened to the parentheses and the semicolon?)

- The type checker resolves types and binds declarations within scopes:



- Finally, JVM code is generated for each node in the tree (leaves first, then roots):

```
iload  3   ; push local 3 (bb)
iload  2   ; push local 2 (c)
ldc    7   ; Push literal 7
isub       ; compute c-7
invokestatic  java/lang/Math/
abs(I)I
iadd       ; compute bb+abs(c-7)
istore  1  ; store result into
             local 1(a)
```

Symbol Tables & Scoping

Programming languages use scopes to limit the range in which an identifier is active (and visible).

Within a scope a name may be defined only once (though overloading may be allowed).

A symbol table (or dictionary) is commonly used to collect all the definitions that appear within a scope.

At the start of a scope, the symbol table is empty. At the end of a scope, all declarations within that scope are available within the symbol table.

A language definition may or may not allow *forward references* to an identifier.

If forward references are allowed, you may use a name that is defined later in the scope (Java does this for field and method declarations within a class).

If forward references are not allowed, an identifier is visible only after its declaration. C, C++ and Java do this for variable declarations.

In CSX no forward references are allowed.

In terms of symbol tables, forward references require two passes over a scope. First all

declarations are gathered. Next, all references are resolved using the complete set of declarations stored in the symbol table.

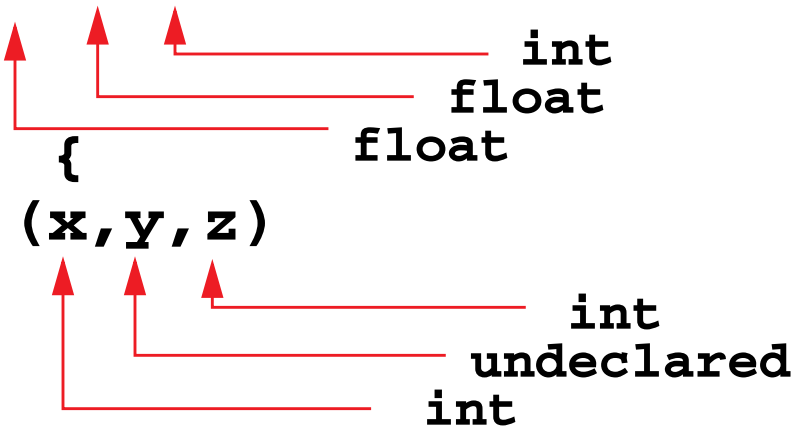
If forward references are disallowed, one pass through a scope suffices, processing declarations and uses of identifiers together.

Block Structured Languages

- Introduced by Algol 60, includes C, C++, CSX and Java.
- Identifiers may have a non-global scope. Declarations may be *local* to a class, subprogram or block.
- Scopes may *nest*, with declarations propagating to inner (contained) scopes.
- The lexically *nearest* declaration of an identifier is bound to uses of that identifier.

Example (drawn from C):

```
int x,z;
void A() {
  float x,y;
   print(x,y,z);
}
void B() {
  print (x,y,z)
}
```



Block Structure Concepts

- Nested Visibility

  No access to identifiers outside their scope.

- Nearest Declaration Applies

  Using static nesting of scopes.

- Automatic Allocation and Deallocation of Locals

  Lifetime of data objects is bound to the scope of the Identifiers that denote them.

Is Case Significant?

In some languages (C, C++, Java and many others) case *is* significant in identifiers. This means **`aa`** and **`AA`** are different symbols that may have entirely different definitions.

In other languages (Pascal, Ada, Scheme, CSX) case *is not* significant. In such languages **`aa`** and **`AA`** are two alternative spellings of the same identifier.

Data structures commonly used to implement symbol tables usually treat different cases as different symbols. This is fine when case is significant in a language. When case is insignificant, you probably will

need to *strip case* before entering or looking up identifiers.

This just means that identifiers are converted to a uniform case before they are entered or looked up. Thus if we choose to use lower case uniformly, the identifiers **`aaa`**, **`AAA`**, and **`AaA`** are all converted to **`aaa`** for purposes of insertion or lookup.

BUT, inside the symbol table the identifier is stored in the form it was declared so that programmers see the form of identifier they expect in listings, error messages, etc.

How are Symbol Tables Implemented?

There are a number of data structures that can reasonably be used to implement a symbol table:

- An Ordered List
  Symbols are stored in a linked list, sorted by the symbol's name. This is simple, but may be a bit too slow if many identifiers appear in a scope.
- A Binary Search Tree
  Lookup is much faster than in linked lists, but rebalancing may be needed. (Entering identifiers in sorted order turns a search tree into a linked list.)
- Hash Tables
  The most popular choice.

Implementing Block-Structured Symbol Tables

To implement a block structured symbol table we need to be able to efficiently open and close individual scopes, and limit insertion to the innermost current scope.

This can be done using one symbol table structure if we tag individual entries with a "scope number."

It is far easier (but more wasteful of space) to allocate one symbol table for each scope. Open scopes are stacked, pushing and popping tables as scopes are opened and closed.

Be careful though—many preprogrammed stack implementations don't allow you to "peek" at entries below the stack top. This is necessary to lookup an identifier in all open scopes.

If a suitable stack implementation (with a peek operation) isn't available, a linked list of symbol tables will suffice.

Reading Assignment

Read Chapter 3 of

Crafting a Compiler.

Scanning

A scanner transforms a character stream into a token stream.

A scanner is sometimes called a *lexical analyzer* or *lexer.*

Scanners use a formal notation (*regular expressions*) to specify the precise structure of tokens.

But why bother? Aren't tokens very simple in structure?

Token structure can be more detailed and subtle than one might expect. Consider simple quoted strings in C, C++ or Java. The body of a string can be any sequence of characters *except* a quote character (which must be escaped). But is this simple definition really correct?

Can a newline character appear in a string? In C it cannot, unless it is escaped with a backslash.

C, C++ and Java allow escaped newlines in strings, Pascal forbids them entirely. Ada forbids *all* unprintable characters.

Are null strings (zero-length) allowed? In C, C++, Java and Ada they are, but Pascal forbids them.

(In Pascal a string is a packed array of characters, and zero length arrays are disallowed.)

A precise definition of tokens can ensure that lexical rules are clearly stated and properly enforced.

Regular Expressions

Regular expressions specify simple (possibly infinite) sets of strings. Regular expressions routinely specify the tokens used in programming languages.

Regular expressions can drive a *scanner generator*.

Regular expressions are widely used in computer utilities:

- The Unix utility *grep* uses regular expressions to define search patterns in files.
- Unix shells allow regular expressions in file lists for a command.

- Most editors provide a “context search” command that specifies desired matches using regular expressions.
- The Windows Find utility allows some regular expressions.

Regular Sets

The sets of strings defined by *regular expressions* are called *regular sets.*

When scanning, a token class will be a regular set, whose structure is defined by a regular expression.

Particular instances of a token class are sometimes called *lexemes,* though we will simply call a string in a token class an *instance* of that token. Thus we call the string `abc` an identifier if it matches the regular expression that defines valid identifier tokens.

Regular expressions use a finite character set, or *vocabulary* (denoted Σ).

This vocabulary is normally the character set used by a computer. Today, the *ASCII* character set, which contains a total of 128 characters, is very widely used.

Java uses the *Unicode* character set which includes all the ASCII characters as well as a wide variety of other characters.

An empty or *null* string is allowed (denoted λ, "lambda"). Lambda represents an empty buffer in which no characters have yet been matched. It also represents optional parts of tokens. An integer literal may begin with a plus or minus, or it may begin with λ if it is unsigned.

Catenation

Strings are built from characters in the character set Σ via *catenation*.

As characters are catenated to a string, it grows in length. The string `do` is built by first catenating `d` to λ, and then catenating `o` to the string `d`. The null string, when catenated with any string s, yields s. That is, $s\,\lambda \equiv \lambda\,s \equiv s$. Catenating λ to a string is like adding 0 to an integer—nothing changes.

Catenation is extended to sets of strings:

Let P and Q be sets of strings. (The symbol $\in$ represents set membership.) If $s_1 \in P$ and $s_2 \in Q$ then string $s_1 s_2 \in (P\ Q)$.

Alternation

Small finite sets are conveniently represented by listing their elements. Parentheses delimit expressions, and |, the *alternation operator*, separates alternatives.

For example, D, the set of the ten single digits, is defined as

D = (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9).

The characters (,), ' , *, +, and | are *meta-characters* (punctuation and regular expression operators).

Meta-characters must be quoted when used as ordinary characters to avoid ambiguity.

For example the expression
`( '(' | ')' | ; | , )`
defines four single character tokens (left parenthesis, right parenthesis, semicolon and comma). The parentheses are quoted when they represent individual tokens and are not used as delimiters in a larger regular expression.

Alternation is extended to sets of strings:

Let P and Q be sets of strings.

Then string s $\in$ (P | Q) if and only if s $\in$ P or s $\in$ Q.

For example, if LC is the set of lower-case letters and UC is the set of upper-case letters, then (LC | UC) is the set of all letters (in either case).

Kleene Closure

A useful operation is *Kleene closure* represented by a postfix $*$ operator.

Let P be a set of strings. Then P^* represents all strings formed by the catenation of zero or more selections (possibly repeated) from P.

Zero selections are denoted by λ.

For example, LC^* is the set of all words composed of lower-case letters, of any length (including the zero length word, λ).

Precisely stated, a string $s \in P^*$ if and only if s can be broken into zero or more pieces: $s = s_1\ s_2\ \ldots\ s_n$ so that each $s_i \in P$ $(n \geq 0,\ 1 \leq i \leq n)$.

We allow $n = 0$, so λ is always in P.

Definition of Regular Expressions

Using catenations, alternation and Kleene closure, we can define *regular expressions* as follows:

- $\emptyset$ is a regular expression denoting the empty set (the set containing no strings). $\emptyset$ is rarely used, but is included for completeness.
- λ is a regular expression denoting the set that contains only the empty string. This set is not the same as the empty set, because it contains one element.
- A string s is a regular expression denoting a set containing the single string s.

- If A and B are regular expressions, then A | B, A B, and A^* are also regular expressions, denoting the alternation, catenation, and Kleene closure of the corresponding regular sets.

Each regular expression denotes a set of strings (a *regular set*). Any finite set of strings can be represented by a regular expression of the form (s_1 | s_2 | ... | s_k). Thus the reserved words of ANSI C can be defined as
(auto | break | case | ...).

The following additional operations useful. They are not strictly necessary, because their effect can be obtained using alternation, catenation, Kleene closure:

- P^+ denotes all strings consisting of *one* or more strings in P catenated together:

  $P^* = (P^+ | \lambda)$ and $P^+ = P\ P^*$.

  For example, $(0 | 1)^+$ is the set of all strings containing one or more bits.

- If A is a set of characters, Not(A) denotes $(\Sigma - A)$; that is, all *characters* in Σ *not* included in A. Since Not(A) can never be larger than Σ and Σ is finite, Not(A) must also be finite, and is therefore regular. Not(A) does not contain λ since λ is not a character (it is a zero-length string).

For example, Not(Eol) is the set of all characters excluding Eol (the end of line character, '`\n`' in Java or C).

- It is possible to extend Not to strings, rather than just Σ. That is, if S is a set of strings, we define $\overline{S}$ to be

  $(\Sigma^* - S)$; the set of all strings except those in S. Though $\overline{S}$ is usually infinite, it is also regular if S is.

- If k is a constant, the set A^k represents all strings formed by catenating k (possibly different) strings from A.

  That is, A^k = (A A A ...) (k copies).

  Thus $(0 \mid 1)^{32}$ is the set of all bit strings exactly 32 bits long.

Examples

Let D be the ten single digits and let L be the set of all 52 letters. Then

- A Java or C++ single-line comment that begins with // and ends with Eol can be defined as:

  $$\text{Comment} = // \ \text{Not}(\text{Eol})^* \ \text{Eol}$$

- A fixed decimal literal (e.g., `12.345`) can be defined as:

  $$\text{Lit} = D^+ . D^+$$

- An optionally signed integer literal can be defined as:

  $$\text{IntLiteral} = (\text{'+'} \mid - \mid \lambda) \ D^+$$

  (Why the quotes on the plus?)

- A comment delimited by ## markers, which allows single #'s within the comment body:

  Comment2 =

  $$\#\# ((\# \mid \lambda)\ \ \text{Not}(\#))^* \#\#$$

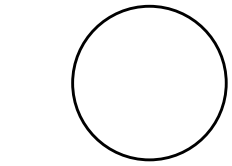
  All finite sets and many infinite sets are regular. But not all infinite sets are regular. Consider the set of balanced brackets of the form [[[...]]].
  This set is defined formally as
  $\{ [^m]^m \mid m \geq 1 \}$.
  This set is known *not* to be regular. Any regular expression that tries to define it either does not get *all* balanced nestings or it includes extra, unwanted strings.

Finite Automata and Scanners

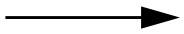
A *finite automaton* (FA) can be used to recognize the tokens specified by a regular expression. FAs are simple, idealized computers that recognize strings belonging to regular sets. An FA consists of:

- A finite set of *states*
- A set of *transitions* (or *moves*) from one state to another, labeled with characters in Σ
- A special state called the *start* state
- A subset of the states called the *accepting*, or *final*, states

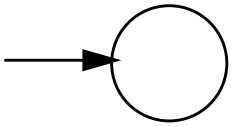
These four components of a finite automaton are often represented graphically:



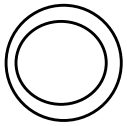
is a state



is a transition



is the start state

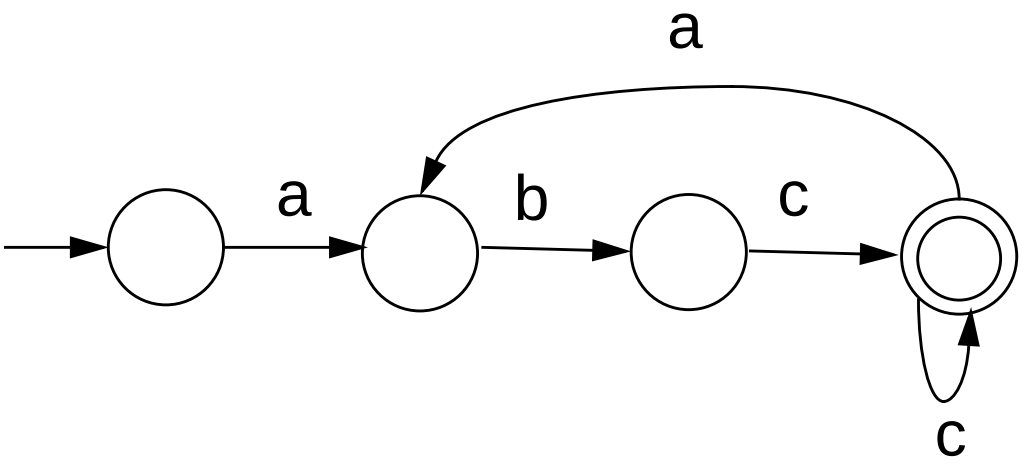


is an accepting state

Finite automata (the plural of automaton is automata) are represented graphically using *transition diagrams.* We start at the start state. If the next input character matches the label on

a transition from the current state, we go to the state it points to. If no move is possible, we stop. If we finish in an accepting state, the sequence of characters read forms a *valid* token; otherwise, we have not seen a valid token.

In this diagram, the valid tokens are the strings described by the regular expression $(a\ b\ (c)^+\ )^+$.



Deterministic Finite Automata

As an abbreviation, a transition may be labeled with more than one character (for example, Not(c)). The transition may be taken if the current input character matches any of the characters labeling the transition.

If an FA always has a *unique* transition (for a given state and character), the FA is *deterministic* (that is, a deterministic FA, or DFA). Deterministic finite automata are easy to program and often drive a scanner.

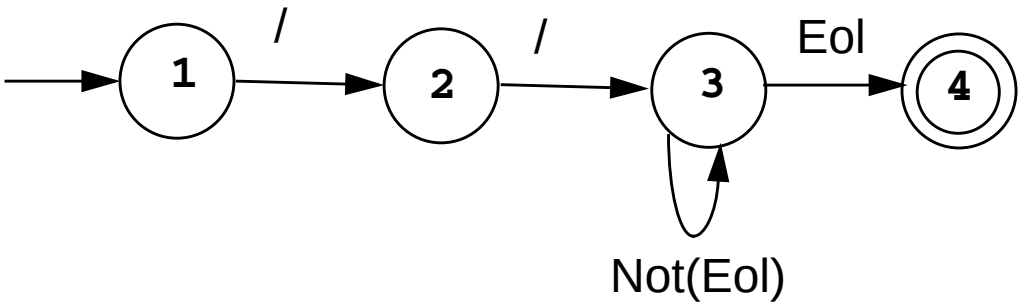
If there are transitions to more than one state for some character, then the FA is *nondeterministic* (that is, an NFA).

A DFA is conveniently represented in a computer by a *transition table.* A transition table, T, is a two dimensional array indexed by a DFA state and a vocabulary symbol.

Table entries are either a DFA state or an error flag (often represented as a blank table entry). If we are in state s, and read character c, then T[s,c] will be the next state we visit, or T[s,c] will contain an error marker indicating that c cannot extend the current token. For example, the regular expression

$$// \ \ \text{Not(Eol)}^* \ \text{Eol}$$

which defines a Java or C++ single-line comment, might be translated into



The corresponding transition table is:

State	Character				
	/	Eol	a	b	…
1	2				
2	3				
3	3	4	3	3	3
4					

A complete transition table contains one column for each character. To save space, *table compression* may be used. Only non-error entries are explicitly represented in the table, using hashing, indirection or linked structures.

All regular expressions can be translated into DFAs that accept (as valid tokens) the strings defined by the regular expressions. This translation can be done manually by a programmer or automatically using a scanner generator.

A DFA can be coded in:

- Table-driven form
- Explicit control form

In the table-driven form, the transition table that defines a DFA's actions is explicitly represented in a run-time table that is "interpreted" by a driver program.

In the direct control form, the transition table that defines a DFA's actions appears implicitly as the control logic of the program.

For example, suppose **`CurrentChar`** is the current input character. End of file is represented by a special character value, **`eof`**. Using the DFA for the Java comments shown earlier, a table-driven scanner is:

```
State = StartState
while (true){
  if (CurrentChar == eof)
      break
  NextState =
      T[State][CurrentChar]
  if(NextState == error)
      break
  State = NextState
  read(CurrentChar)
}
if (State in AcceptingStates)
     // Process valid token
else // Signal a lexical error
```

This form of scanner is produced by a scanner generator; it is definition-independent. The scanner is a driver that can scan *any* token if T contains the appropriate transition table.

Here is an explicit-control scanner for the same comment definition:

```
if (CurrentChar == '/'){
   read(CurrentChar)
   if (CurrentChar == '/')
     repeat
       read(CurrentChar)
     until (CurrentChar in
             {eol, eof})
   else //Signal lexical error
else // Signal lexical error
if (CurrentChar == eol)
  // Process valid token
else //Signal lexical error
```

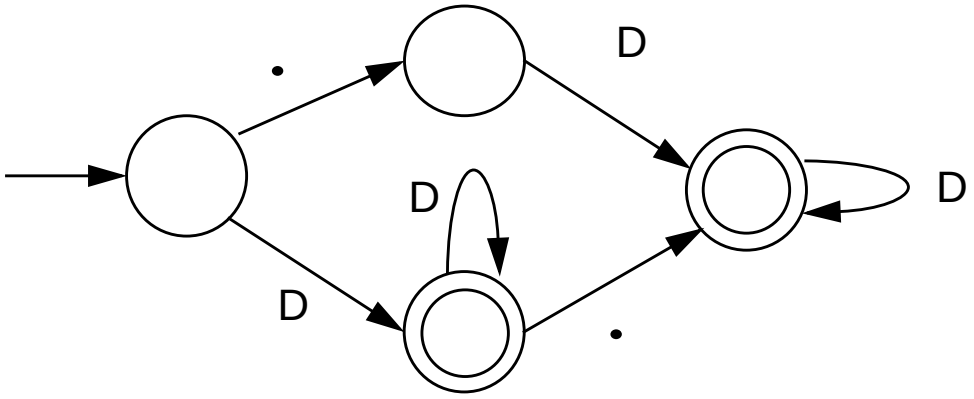
The token being scanned is "hardwired" into the logic of the code. The scanner is usually easy to read and often is more efficient, but is specific to a single token definition.

More Examples

- A FORTRAN-like real literal (which requires digits on either or both sides of a decimal point, or just a string of digits) can be defined as

  $RealLit = (D^+ (\lambda \mid .)) \mid (D^* . D^+)$

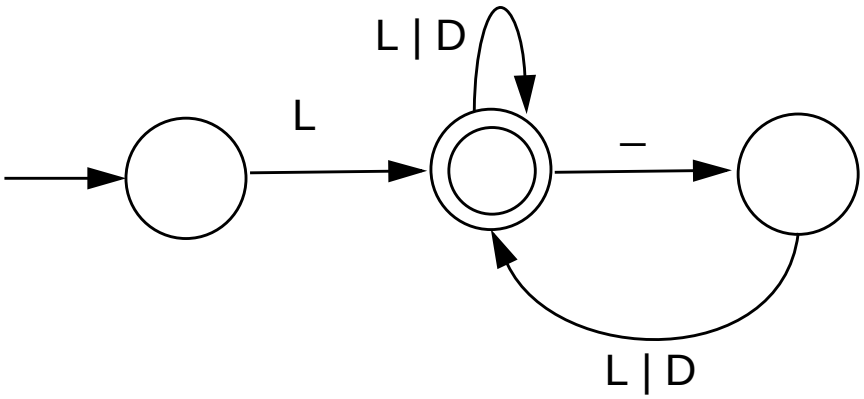
  This corresponds to the DFA



- An identifier consisting of letters, digits, and underscores, which begins with a letter and allows no adjacent or trailing underscores, may be defined as

  ID = L (L | D)* (_ (L | D)$^{+}$)*

  This definition includes identifiers like `sum` or `unit_cost`, but excludes `_one` and `two_` and `grand___total`. The DFA is:



Lex/Flex/JLex

Lex is a well-known Unix scanner generator. It builds a scanner, in C, from a set of regular expressions that define the tokens to be scanned.

Flex is a newer and faster version of Lex.

JLex is a Java version of Lex. It generates a scanner coded in Java, though its regular expression definitions are very close to those used by Lex and Flex.

Lex, Flex and JLex are largely *non-procedural*. You don't need to tell the tools *how* to scan. All you need to tell it *what* you want scanned (by giving it definitions of valid tokens).

This approach greatly simplifies building a scanner, since most of the details of scanning (I/O, buffering, character matching, etc.) are automatically handled.

JLex

JLex is coded in Java. To use it, you enter

`java JLex.Main f.jlex`

Your **`CLASSPATH`** should be set to search the directories where JLex's classes are stored.
(In build files we provide the **`CLASSPATH`** used will includ JLex's classes).

After JLex runs (assuming there are no errors in your token specifications), the Java source file
`f.jlex.java` is created. (**`f`** stands for any file name you choose. Thus **`csx.jlex`** might hold token definitions for CSX, and **`csx.jlex.java`** would hold the generated scanner).

You compile **`f.jlex.java`** just like any Java program, using your favorite Java compiler.

After compilation, the class file

`Yylex.class` is created.

It contains the methods:

- **`Token yylex()`** which is the actual scanner. The constructor for **`Yylex`** takes the file you want scanned, so **`new Yylex(System.in)`** will build a scanner that reads from **`System.in`**. **`Token`** is the token class you want returned by the scanner; you can tell JLex what class you want returned.
- **`String yytext()`** returns the character text matched by the last call to **`yylex`**.

A simple example of the use of JLex is in

~cs536-1/pubic/jlex.2012

Copy the folder to your filespace and enter

ant clean compile test

Input to JLex

There are three sections, delimited by **%%**. The general structure is:

```
User Code
%%
Jlex Directives
%%
Regular Expression rules
```

The User Code section is Java source code to be copied into the generated Java source file. It contains utility classes or return type classes you need. Thus if you want to return a class **`IntlitToken`** (for integer literals that are scanned), you include its definition in the User Code section.

JLex directives are various instructions you can give JLex to customize the scanner you generate.

These are detailed in the JLex manual. The most important are:

- **%{**
  Code copied into the Yylex class (extra fields or methods you may want)
  %}
- **%eof{**
  Java code to be executed when the end of file is reached
  %eof}
- **%type classname**
  classname is the return type you want for the scanner method, **yylex()**

Macro Definitions

In section two you may also define *macros*, that are used in section three. A macro allows you to give a name to a regular expression or character class. This allows you to reuse definitions and make regular expression rule more readable.

Macro definitions are of the form

```
name = def
```

Macros are defined one per line.

Here are some simple examples:

```
Digit=[0-9]

AnyLet=[A-Za-z]
```

In section 3, you use a macro by placing its name within `{` and `}`. Thus `{Digit}` expands to the character class defining the digits 0 to 9.

Regular Expression Rules

The third section of the JLex input file is a series of token definition rules of the form

`RegExpr {Java code}`

When a token matching the given **`RegExpr`** is matched, the corresponding Java code (enclosed in "{" and "}") is executed. JLex figures out what **`RegExpr`** applies; you need only say what the token looks like (using **`RegExpr`**) and what you want done when the token is matched (this is usually to return some token object, perhaps with some processing of the token text).

Here are some examples:

```
"+"     {return new Token(sym.Plus);}

(" ")+   {/* skip white space */}

{Digit}+ {return
   new IntToken(sym.Intlit,
    new Integer(yytext()).intValue());}
```

Regular Expressions in JLex

To define a token in JLex, the user to associates a regular expression with commands coded in Java.

When input characters that match a regular expression are read, the corresponding Java code is executed. As a user of JLex you don't need to tell it *how* to match tokens; you need only say *what* you want done when a particular token is matched.

Tokens like white space are deleted simply by having their associated command not return anything. Scanning continues until a command with a return in it is executed.

The simplest form of regular expression is a single string that matches exactly itself.

For example,

```
if    {return new Token(sym.If);}
```

If you wish, you can quote the string representing the reserved word (**"if"**), but since the string contains no delimiters or operators, quoting it is unnecessary.

For a regular expression operator, like +, quoting is necessary:

```
"+"    {return
           new Token(sym.Plus);}
```

Character Classes

Our specification of the reserved word if, as shown earlier, is incomplete. We don't (yet) handle upper or mixed-case.

To extend our definition, we'll use a very useful feature of Lex and JLex—*character classes*.

Characters often naturally fall into classes, with all characters in a class treated identically in a token definition. In our definition of identifiers all letters form a class since any of them can be used to form an identifier. Similarly, in a number, any of the ten digit characters can be used.

Character classes are delimited by `[` and `]`; individual characters are listed without any quotation or separators. However `\`, `^`, `]` and `-`, because of their special meaning in character classes, must be escaped. The character class `[xyz]` can match a single `x`, `y`, or `z`.

The character class `[\])]` can match a single `]` or `)`.

(The `]` is escaped so that it isn't misinterpreted as the end of character class.)

Ranges of characters are separated by a `-`; `[x-z]` is the same as `[xyz]`. `[0-9]` is the set of all digits and `[a-zA-Z]` is the set of all letters, upper- and lower-case. `\` is the escape character, used to represent unprintables and to escape special symbols.

Following C and Java conventions, **`\n`** is the newline (that is, end of line), **`\t`** is the tab character, **`\\`** is the backslash symbol itself, and **`\010`** is the character corresponding to octal 10.

The **`^`** symbol complements a character class (it is JLex's representation of the Not operation).

`[^xy]` is the character class that matches any single character *except* **`x`** and **`y`**. The **`^`** symbol applies to all characters that follow it in a character class definition, so **`[^0-9]`** is the set of all characters that aren't digits. **`[^]`** can be used to match all characters.

Here are some examples of character classes:

Character Class	Set of Characters Denoted
`[abc]`	Three characters: **a**, **b** and **c**
`[cba]`	Three characters: **a**, **b** and **c**
`[a-c]`	Three characters: **a**, **b** and **c**
`[aabbcc]`	Three characters: **a**, **b** and **c**
`[^abc]`	All characters except **a**, **b** and **c**
`[\^\-\]]`	Three characters: **^**, **-** and **]**
`[^]`	All characters
`"[abc]"`	Not a character class. This is one five character *string*: **[abc]**

Regular Operators in JLex

JLex provides the standard regular operators, plus some additions.

- Catenation is specified by the juxtaposition of two expressions; no explicit operator is used. Outside of character class brackets, individual letters and numbers match themselves; other characters should be quoted (to avoid misinterpretation as regular expression operators).

Regular Expr	Characters Matched
`a b cd`	Four characters: **`abcd`**
`(a)(b)(cd)`	Four characters: **`abcd`**
`[ab][cd]`	Four different strings: **`ac`** or **`ad`** or **`bc`** or **`bd`**
`while`	Five characters: **`while`**
`"while"`	Five characters: **`while`**
`[w][h][i][l][e]`	Five characters: **`while`**

Case *is* significant.

- The alternation operator is `|`. Parentheses can be used to control grouping of subexpressions.
  If we wish to match the reserved word `while` allowing any mixture of upper- and lowercase, we can use
  `(w|W)(h|H)(i|I)(l|L)(e|E)`
  or
  `[wW][hH][iI][lL][eE]`

Regular Expr	Characters Matched
`ab|cd`	Two different strings: **`ab`** or **`cd`**
`(ab)|(cd)`	Two different strings: **`ab`** or **`cd`**
`[ab]|[cd]`	Four different strings: **`a`** or **`b`** or **`c`** or **`d`**

- Postfix operators:
  ***** Kleene closure: 0 or more matches.
  (ab)* matches λ or **ab** or **abab** or **ababab** ...

  + Positive closure: 1 or more matches.
  (ab)+ matches **ab** or **abab** or **ababab** ...

  ? Optional inclusion:
  expr?
  matches `expr` zero times or once.
  expr? is equivalent to **(expr)** | λ and eliminates the need for an explicit λ symbol.

  [-+]?[0-9]+ defines an optionally signed integer literal.

- Single match:
  The character "." matches any single character (other than a newline).
- Start of line:
  The character ^ (when used outside a character class) matches the beginning of a line.
- End of line:
  The character $ matches the end of a line. Thus,

  ^A.*e$

  matches an entire line that begins with A and ends with e.

Overlapping Definitions

Regular expressions may overlap (match the same input sequence).

In the case of overlap, two rules determine which regular expression is matched:

- The *longest possible* match is performed. JLex automatically buffers characters while deciding how many characters can be matched.
- If two expressions match *exactly* the same string, the earlier expression (in the JLex specification) is preferred. Reserved words, for example, are often special cases of the pattern used for identifiers. Their definitions are therefore placed before the expression that defines an identifier token.

Often a “catch all” pattern is placed at the very end of the regular expression rules. It is used to catch characters that don’t match any of the earlier patterns and hence are probably erroneous. Recall that " ." matches any single character (other than a newline). It is useful in a catch-all pattern. However, avoid a pattern like `.*` which will consume all characters up to the next newline.

In JLex an unmatched character will cause a run-time error.

The operators and special symbols most commonly used in JLex are summarized below. Note that a symbol sometimes has one meaning in a regular expression and an *entirely different* meaning

in a character class (i.e., within a pair of brackets). If you find JLex behaving unexpectedly, it's a good idea to check this table to be sure of how the operators and symbols you've used behave. Ordinary letters and digits, and symbols not mentioned (like @) represent themselves. If you're not sure if a character is special or not, you can always escape it or make it part of a quoted string.

Symbol	Meaning in Regular Expressions	Meaning in Character Classes
(	Matches with) to group sub-expressions.	Represents itself.
)	Matches with (to group sub-expressions.	Represents itself.
[	Begins a character class.	Represents itself.
]	Represents itself.	Ends a character class.
{	Matches with } to signal macro-expansion.	Represents itself.
}	Matches with { to signal macro-expansion.	Represents itself.
"	Matches with " to delimit strings (only \ is special within strings).	Represents itself.
\	Escapes individual characters. Also used to specify a character by its octal code.	Escapes individual characters. Also used to specify a character by its octal code.
.	Matches any one character except \n.	Represents itself.
|	Alternation (or) operator.	Represents itself.

Symbol	Meaning in Regular Expressions	Meaning in Character Classes
*	Kleene closure operator (zero or more matches).	Represents itself.
+	Positive closure operator (one or more matches).	Represents itself.
?	Optional choice operator (one or zero matches).	Represents itself.
/	Context sensitive matching operator.	Represents itself.
^	Matches only at beginning of a line.	Complements remaining characters in the class.
$	Matches only at end of a line.	Represents itself.
-	Represents itself.	Range of characters operator.

Potential Problems in Using JLex

The following differences from "standard" Lex notation appear in JLex:

- Escaped characters within quoted strings are not recognized. Hence `"\n"` is *not* a new line character. Escaped characters outside of quoted strings (`\n`) and escaped characters within character classes (`[\n]`) are OK.
- A blank should not be used within a character class (i.e., `[` and `]`). You may use `\040` (which is the character code for a blank).
- A doublequote must be escaped within a character class. Use `[\"]` instead of `["]`.

- Unprintables are defined to be all characters before blank as well as the last ASCII character. Unprintables can be represented as: **`[\000-\037\177]`**

JLex Examples

A JLex scanner that looks for five letter words that begin with “P” and end with “T”.

This example is in

`~cs536-1/public/jlex.2012`

The JLex specification file is:

```
class Token {
   String text;
   Token(String t){text = t;}
}
%%
Digit=[0-9]
AnyLet=[A-Za-z]
Others=[0-9'&.]
WhiteSp=[\040\n]
// Tell JLex to have yylex() return a
Token
%type Token
// Tell JLex what to return when eof of
file is hit
%eofval{
return new Token(null);
%eofval}
%%
[Pp]{AnyLet}{AnyLet}{AnyLet}[Tt]{WhiteSp}+
     {return new Token(yytext());}

({AnyLet}|{Others})+{WhiteSp}+
     {/*skip*/}
```

The Java program that uses the scanner is:

```
import java.io.*;

class Main {

public static void main(String args[])
   throws java.io.IOException {

 Yylex lex  = new Yylex(System.in);
 Token token = lex.yylex();

 while ( token.text != null ) {
   System.out.print("\t"+token.text);
   token = lex.yylex(); //get next token
 }
}}
```

In case you care, the words that are matched include:

Pabst

paint

petit

pilot

pivot

plant

pleat

point

posit

Pratt

print

An example of CSX token specifications. This example is in

~cs536-1/public/proj2/startup/java

The JLex specification file is:

```
import java_cup.runtime.*;

/*  Expand this into your solution for
project 2 */

class CSXToken {
 int linenum;
 int colnum;
 CSXToken(int line,int col){
 linenum=line;colnum=col;};
}

class CSXIntLitToken extends CSXToken {
 int intValue;
 CSXIntLitToken(int val,int line,
   int col){
   super(line,col);intValue=val;};
}
class CSXIdentifierToken extends
CSXToken {
String identifierText;
CSXIdentifierToken(String text,int line,
  int col){
  super(line,col);identifierText=text;};
}
```

```
class CSXCharLitToken extends CSXToken {
  char charValue;
CSXCharLitToken(char val,int line,
    int col){
   super(line,col);charValue=val;};
}

class CSXStringLitToken extends CSXToken
{
   String stringText;
   CSXStringLitToken(String text,
     int line,int col){
   super(line,col);
   stringText=text; };
}
// This class is used to track line and
column numbers
// Feel free to change to extend it
class Pos {
static int  linenum = 1;
/* maintain this as line number current
   token was scanned on */
static int  colnum = 1;
  /* maintain this as column number
    current token began at */
static int  line = 1;
/* maintain this as line number after
   scanning current token  */
```

```
static int  col = 1;
  /* maintain this as column number
    after scanning current token  */
static void setpos() {
 //set starting pos for current token
   linenum = line;
   colnum = col;}
}


%%
Digit=[0-9]

// Tell JLex to have yylex() return a
 Symbol, as JavaCUP will require
%type Symbol

// Tell JLex what to return when eof of
file is hit
%eofval{
return new Symbol(sym.EOF,
          new  CSXToken(0,0));
%eofval}
```

```
%%
"+"    {Pos.setpos(); Pos.col +=1;
         return new Symbol(sym.PLUS,
           new CSXToken(Pos.linenum,
                        Pos.colnum));}
"!="     {Pos.setpos(); Pos.col +=2;
          return new Symbol(sym.NOTEQ,
            new CSXToken(Pos.linenum,
                         Pos.colnum));}
";"      {Pos.setpos(); Pos.col +=1;
            return new Symbol(sym.SEMI,
              new CSXToken(Pos.linenum,
                           Pos.colnum));}
{Digit}+   {// This def doesn't check
            // for overflow
           Pos.setpos();
           Pos.col += yytext().length();
           return new Symbol(sym.INTLIT,
             new CSXIntLitToken(
         new Integer(yytext()).intValue(),
         Pos.linenum,Pos.colnum));}

\n       {Pos.line +=1; Pos.col = 1;}
" "      {Pos.col +=1;}
```

The Java program that uses this scanner (P2) is:

```
class P2 {
 public static void main(String args[])
    throws java.io.IOException {
  if (args.length != 1) {
    System.out.println(
    "Error: Input file must be named on
      command line." );
    System.exit(-1);
  }
  java.io.FileInputStream yyin = null;
  try {
   yyin =
   new java.io.FileInputStream(args[0]);
  } catch (FileNotFoundException
           notFound){
    System.out.println(
   "Error: unable to open input file.");
    System.exit(-1);
  }

 // lex is a JLex-generated scanner that
 // will read from yyin
    Yylex lex = new Yylex(yyin);

   System.out.println(
      "Begin test of CSX scanner.");
```

```
/************************************
 You should enter code here that
 thoroughly test your scanner.

 Be sure to test extreme cases,
 like very long symbols or lines,
 illegal tokens, unrepresentable
 integers, illegals strings, etc.
 The following is only a starting point.
************************************/
Symbol token = lex.yylex();

while ( token.sym != sym.EOF ) {
 System.out.print(
   ((CSXToken) token.value).linenum
   + ":"
   + ((CSXToken) token.value).colnum
   + " ");

  switch (token.sym) {
   case sym.INTLIT:
    System.out.println(
     "\tinteger literal(" +
     ((CSXIntLitToken)
     token.value).intValue + ")");
    break;

  case sym.PLUS:
    System.out.println("\t+");
    break;
```

```
  case sym.NOTEQ:
    System.out.println("\t!=");
    break;

  default:
    throw new RuntimeException();
 }

 token = lex.yylex(); // get next token
}

System.out.println(
   "End test of CSX scanner.");
}}}
```

Other Scanner Issues

We will consider other practical issues in building real scanners for real programming languages.

Our finite automaton model sometimes needs to be augmented. Moreover, error handling must be incorporated into any practical scanner.

Identifiers vs. Reserved Words

Most programming languages contain *reserved words* like **`if`**, **`while`**, **`switch`**, etc. These tokens look like ordinary identifiers, but aren't.

It is up to the scanner to decide if what looks like an identifier is really a reserved word. This distinction is vital as reserved words have different token codes than identifiers and are parsed differently.

How can a scanner decide which tokens are identifiers and which are reserved words?

- We can scan identifiers and reserved words using the same pattern, and then look up the token in a special "reserved word" table.
- It is known that any regular expression may be *complemented* to obtain all strings not in the original regular expression. Thus $\overline{A}$, the complement of A, is regular if A is. Using complementation we can write a regular expression for nonreserved

  identifiers: $\overline{(\overline{\text{ident}}|\text{if}|\text{while}|\ldots)}$

  Since scanner generators don't usually support complementation of regular expressions, this approach is more of theoretical than practical interest.

- We can give distinct regular expression definitions for each reserved word, and for identifiers. Since the definitions overlap (**`if`** will match a reserved word *and* the general identifier pattern), we give *priority* to reserved words. Thus a token is scanned as an identifier if it matches the identifier pattern *and* does not match any reserved word pattern. This approach is commonly used in scanner generators like Lex and JLex.

Converting Token Values

For some tokens, we may need to convert from string form into numeric or binary form.

For example, for integers, we need to transform a string a digits into the internal (binary) form of integers.

We know the format of the token is valid (the scanner checked this), but:

- The string may represent an integer too large to represent in 32 or 64 bit form.
- Languages like CSX and ML use a non-standard representation for negative values (**`~123`** instead of **`-123`**)

We can safely convert from string to integer form by first converting the string to double form, checking against max and min int, and then converting to int form if the value is representable.

Thus **`d = new Double(str)`** will create an object **`d`** containing the value of **`str`** in double form. If **`str`** is too large or too small to be represented as a double, plus or minus infinity is automatically substituted.

`d.doubleValue()` will give **`d`**'s value as a Java double, which can be compared against **`Integer.MAX_VALUE`** or **`Integer.MIN_VALUE`**.

If **`d.doubleValue()`** represents a valid integer,
`(int) d.doubleValue()`
will create the appropriate integer value.

If a string representation of an integer begins with a "~" we can strip the "~", convert to a double and then negate the resulting value.

Scanner Termination

A scanner reads input characters and partitions them into tokens.

What happens when the end of the input file is reached? It may be useful to create an **`Eof`** pseudo-character when this occurs. In Java, for example, **`InputStream.read()`**, which reads a single byte, returns -1 when end of file is reached. A constant, **`EOF`**, defined as -1 can be treated as an "extended" ASCII character. This character then allows the definition of an **`Eof`** token that can be passed back to the parser.

An **`Eof`** token is useful because it allows the parser to verify that the logical end of a program corresponds to its physical end.

Most parsers *require* an end of file token.

Lex and Jlex automatically create an **`Eof`** token when the scanner they build tries to scan an **`EOF`** character (or tries to scan when **`eof()`** is true).

Multi Character Lookahead

We may allow finite automata to look beyond the next input character.

This feature is necessary to implement a scanner for FORTRAN.

In FORTRAN, the statement

```
DO 10 J = 1,100
```

specifies a loop, with index `J` ranging from 1 to 100.
The statement

```
DO 10 J = 1.100
```

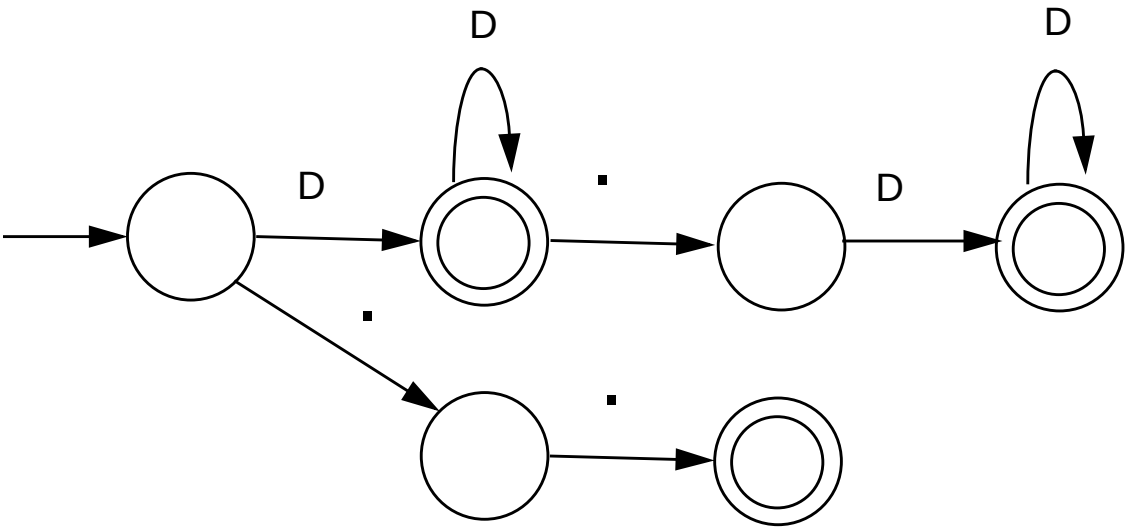
is an assignment to the variable `DO10J`. (Blanks are not significant except in strings.)

A FORTRAN scanner decides whether the `O` is the last character of a `DO` token only after reading as far as the comma (or period).

A milder form of extended lookahead problem occurs in Pascal and Ada.
The token **`10.50`** is a real literal, whereas **`10..50`** is three different tokens.

We need two-character lookahead after the **`10`** prefix to decide whether we are to return **`10`** (an integer literal) or **`10.50`** (a real literal).

Suppose we use the following FA.



Given **`10..100`** we scan three characters and stop in a non-accepting state.

Whenever we stop reading in a non-accepting state, we *back up* along accepted characters until an accepting state is found.

Characters we back up over are *rescanned* to form later tokens. If no accepting state is reached during backup, we have a lexical error.

Performance Considerations

Because scanners do so much character-level processing, they can be a real performance bottleneck in production compilers.

Speed is not a concern in our project, but let's see why scanning speed can be a concern in production compilers.

Let's assume we want to compile at a rate of 5000 lines/sec. (so that most programs compile in just a few seconds).

Assuming 30 characters/line (on average), we need to scan 150,000 char/sec.

A key to efficient scanning is to group character-level operations whenever possible. It is better to do one operation on n characters rather than n operations on single characters.

In our examples we've read input one character as a time. A subroutine call can cost hundreds or thousands of instructions to execute—far too much to spend on a single character.

We prefer routines that do block reads, putting an entire block of characters directly into a buffer.

Specialized scanner generators can produce particularly fast scanners.

The *GLA scanner generator* claims that the scanners it produces run as fast as:

```
while(c != Eof) {
   c = getchar();
}
```

Lexical Error Recovery

A character sequence that can't be scanned into any valid token is a *lexical error*.

Lexical errors are uncommon, but they still must be handled by a scanner. We won't stop compilation because of so minor an error.

Approaches to lexical error handling include:

- Delete the characters read so far and restart scanning at the next unread character.
- Delete the first character read by the scanner and resume scanning at the character following it.

Both of these approaches are reasonable.

The first is easy to do. We just reset the scanner and begin scanning anew.

The second is a bit harder but also is a bit safer (less is immediately deleted). It can be implemented using scanner backup.

Usually, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

(Why at the beginning?)

In these case, the two approaches are equivalent.

The effects of lexical error recovery might well create a later *syntax error*, handled by the parser.

Consider

```
...for$tnight...
```

The **$** terminates scanning of **for**. Since no valid token begins with **$**, it is deleted. Then **tnight** is scanned as an identifier. In effect we get

```
...for tnight...
```

which will cause a syntax error. Such “false errors” are unavoidable, though a syntactic error-repair may help.

Error Tokens

Certain lexical errors require special care. In particular, runaway strings and runaway comments ought to receive special error messages.

In Java strings may not cross line boundaries, so a runaway string is detected when an end of a line is read within the string body. Ordinary recovery rules are inappropriate for this error. In particular, deleting the first character (the double quote character) and restarting scanning is a *bad* decision.

It will almost certainly lead to a cascade of "false" errors as the string text is inappropriately scanned as ordinary input.

One way to handle runaway strings is to define an *error token*.

An error token is *not* a valid token; it is never returned to the parser. Rather, it is a *pattern* for an error condition that needs special handling. We can define an error token that represents a string terminated by an end of line rather than a double quote character.

For a valid string, in which internal double quotes and back slashes are escaped (and no other escaped characters are allowed), we can use

" (Not(" | Eol | \ ) | \" | \\ )* "

For a runaway string we use

" (Not(" | Eol | \ ) | \" | \\ )* Eol

(**Eol** is the end of line character.)

When a runaway string token is recognized, a special error message should be issued.

Further, the string may be "repaired" into a correct string by returning an ordinary string token with the closing Eol replaced by a double quote.

This repair may or may not be "correct." If the closing double quote is truly missing, the repair will be good; if it is present on a succeeding line, a cascade of inappropriate lexical and syntactic errors will follow.

Still, we have told the programmer exactly what is wrong, and that is our primary goal.

In languages like C, C++, Java and CSX, which allow multiline comments, improperly terminated (runaway) comments present a similar problem.

A runaway comment is not detected until the scanner finds a close comment symbol (possibly belonging to some other comment) or until the end of file is reached. Clearly a special, detailed error message is required.

Let's look at Pascal-style comments that begin with a `{` and end with a `}`. Comments that begin and end with a pair of characters, like `/*` and `*/` in Java, C and C++, are a bit trickier.

Correct Pascal comments are defined quite simply:

{ Not(})* }

To handle comments terminated by **Eof**, this error token can be used:

{ Not(})* Eof

We want to handle comments unexpectedly closed by a close comment belonging to another comment:

```
{... missing close comment
 ... { normal comment }...
```

We will issue a *warning* (this form of comment is lexically legal).

Any comment containing an open comment symbol in its body is most probably a missing **}** error.

We split our legal comment definition into two token definitions.

The definition that accepts an open comment in its body causes a warning message ("Possible unclosed comment") to be printed.

We now use:

{ Not({ | })* } and
{ (Not({ | })* { Not({ | })*)+ }

The first definition matches correct comments that do not contain an open comment in their body.

The second definition matches correct, but suspect, comments that contain at least one open comment in their body.

Single line comments, found in Java, CSX and C++, are terminated by Eol.

They can fall prey to a more subtle error—what if the last line has no Eol at its end?

The solution?

Another error token for single line comments:

$$// \ \text{Not(Eol)}^*$$

This rule will only be used for comments that don't end with an Eol, since scanners always match the longest rule possible.

Regular Expressions and Finite Automata

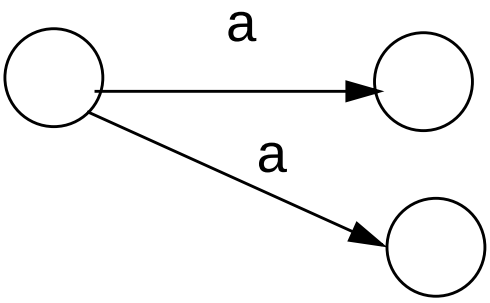
Regular expressions are fully equivalent to finite automata.

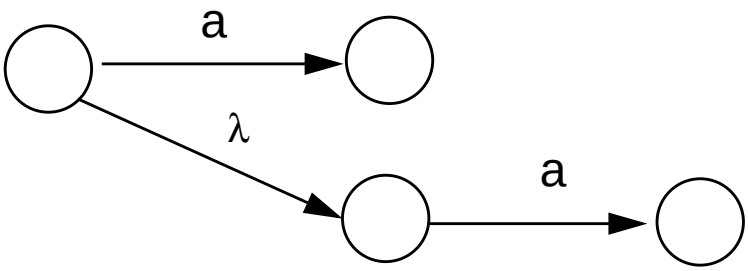
The main job of a scanner generator like JLex is to transform a regular expression definition into an equivalent finite automaton.

It first transforms a regular expression into a *nondeterministic finite automaton* (NFA).

Unlike ordinary deterministic finite automata, an NFA need not make a unique (deterministic) choice of a successor state to visit. As shown below, an NFA is allowed to have a state that has two transitions (arrows) coming out of it, labeled by the same

symbol. An NFA may also have transitions labeled with λ.





Transitions are normally labeled with individual characters in Σ, and although λ is a string (the string with no characters in it), it is definitely *not* a character. In the above example, when the automaton is in the state at the left and the next input character is a, it may choose to use the transition labeled a *or* first follow

the λ transition (you can always find λ wherever you look for it) and *then* follow an a transition. FAs that contain no λ transitions and that always have unique successor states for any symbol are *deterministic.*

Building Finite Automata From Regular Expressions

We make an FA from a regular expression in two steps:

- Transform the regular expression into an NFA.
- Transform the NFA into a deterministic FA.

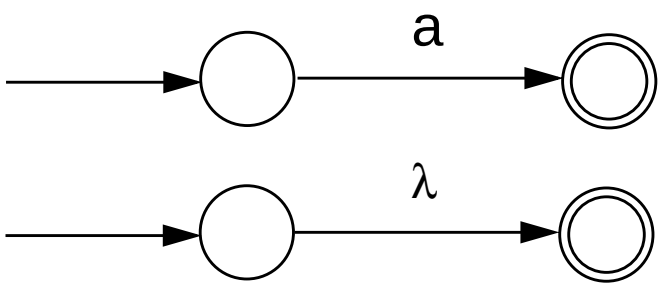
The first step is easy.

Regular expressions are all built out of the *atomic* regular expressions a (where a is a character in Σ) and λ by using the three operations

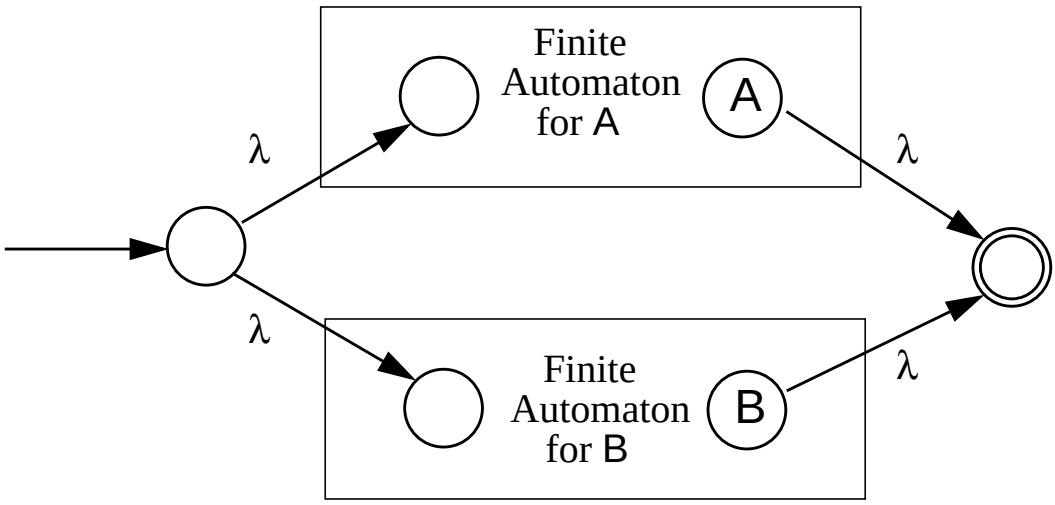
A B and A | B and A^*.

Other operations (like A^+) are just abbreviations for combinations of these.

NFAs for a and λ are trivial:



Suppose we have NFAs for A and B and want one for A | B. We construct the NFA shown below:



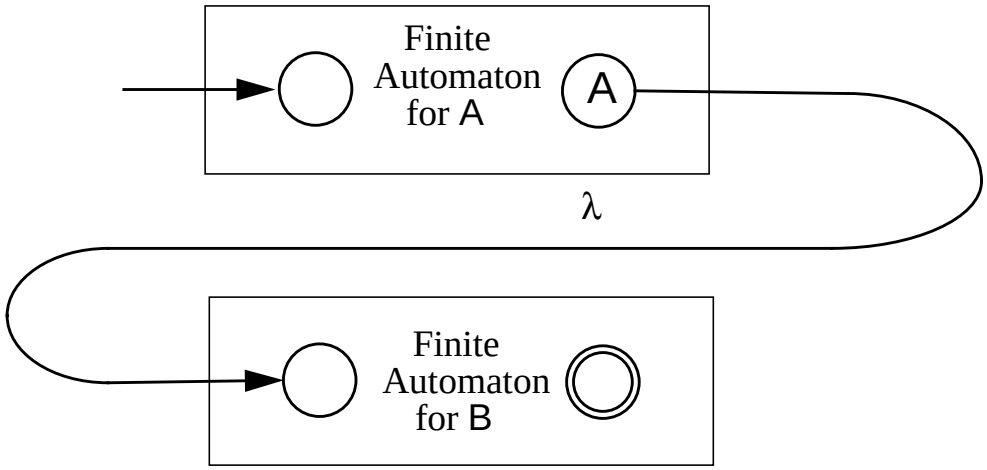
The states labeled A and B were the accepting states of the automata for A and B; we create a new accepting state for the combined automaton.

A path through the top automaton accepts strings in **A**, and a path through the bottom automation accepts strings in **B**, so the whole automaton matches **A | B**.

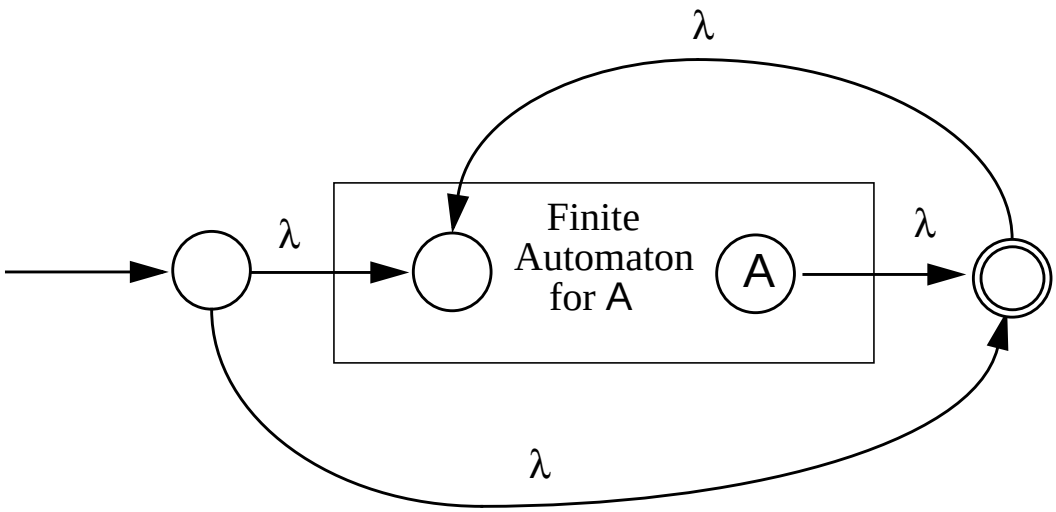
The construction for A B is even easier. The accepting state of the combined automaton is the same state that was the accepting state of B. We must follow a path through **A**'s automaton, then through **B**'s automaton, so overall **A B** is matched.

We could also just merge the accepting state of A with the initial state of B. We chose not to

only because the picture would be more difficult to draw.



Finally, let's look at the NFA for A^*. The start state reaches an accepting state via λ, so λ is accepted. Alternatively, we can follow a path through the FA for A one or more times, so zero or more strings that belong to A are matched.



Creating Deterministic Automata

The transformation from an NFA N to an equivalent DFA D works by what is sometimes called the *subset construction.*

Each state of D corresponds to a set of states of N.

The idea is that D will be in state {x, y, z} after reading a given input string if and only if N could be in *any* one of the states *x*, *y*, or *z*, depending on the transitions it chooses. Thus D keeps track of *all* the possible routes N might take and runs them simultaneously.

Because N is a *finite* automaton, it has only a finite number of states. The number of subsets of N's states is also finite, which makes

tracking various sets of states feasible.

An accepting state of D will be any set containing an accepting state of N, reflecting the convention that N accepts if there is *any* way it could get to its accepting state by choosing the "right" transitions.

The start state of D is the set of all states that N could be in without reading any input characters—that
is, the set of states reachable from the start state of N following only λ transitions. Algorithm **`close`** computes those states that can be reached following only λ transitions.

Once the start state of D is built, we begin to create successor states:

We take each state S of D, and each character c, and compute S's successor under c.

S is identified with some set of N's states, $\{n_1, n_2,...\}$.

We find all the possible successor states to $\{n_1, n_2,...\}$ under c, obtaining a set $\{m_1, m_2,...\}$.

Finally, we compute
$T = \text{CLOSE}(\{ m_1, m_2,...\})$.

T becomes a state in D, and a transition from S to T labeled with c is added to D.

We continue adding states and transitions to D until all possible successors to existing states are added.

Because each state corresponds to a finite subset of N's states, the

process of adding new states to D must eventually terminate.

Here is the algorithm for λ-closure, called **close**. It starts with a set of NFA states, S, and adds to S all states reachable from S using only λ transitions.

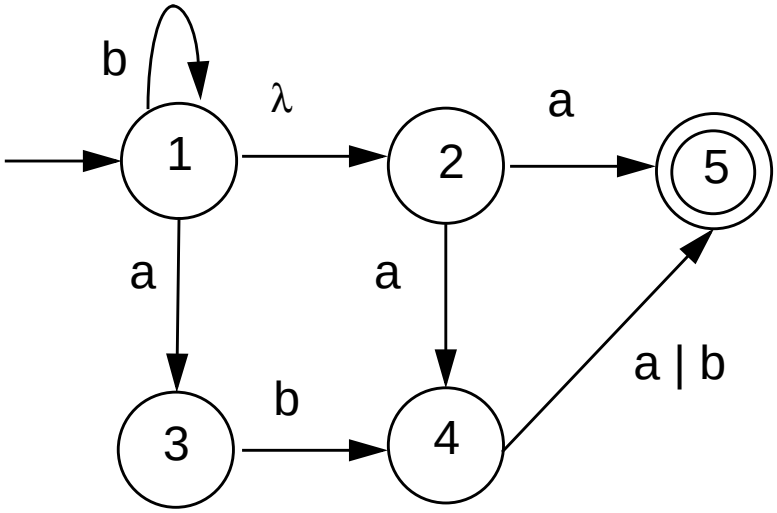
```
void close(NFASet S) {

 while (x in S and x --λ--> y
       and y notin S) {
         S = S ∪ {y}
}}
```

Using **close**, we can define the construction of a DFA, D, from an NFA, N:

```
DFA  MakeDeterministic(NFA N) {
 DFA  D ; NFASet  T
 D.StartState = { N.StartState }
 close(D.StartState)
 D.States = { D.StartState }
 while (states or transitions can be
        added to D) {
   Choose any state S in D.States
     and any character c in Alphabet
   T = {y in N.States such that
          x -c-> y for some x in S}
   close(T);
   if (T notin D.States) {
        D.States = D.States ∪ {T}
        D.Transitions =
           D.Transitions ∪
             {the transition S -c-> T}
 } }
 D.AcceptingStates =
  { S in D.States such that an
      accepting state of N in S}
}
```

Example

To see how the subset construction operates, consider the following NFA:



We start with state 1, the start state of N, and add state 2 its λ-successor.

D's start state is {1,2}.

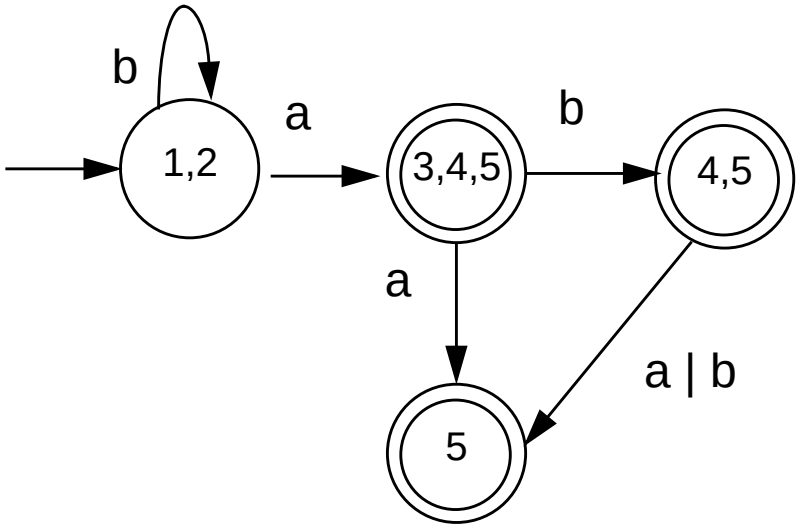
Under a, {1,2}'s successor is {3,4,5}.

State 1 has itself as a successor under b. When state 1's λ-successor, 2, is included, {1,2}'s successor is {1,2}. {3,4,5}'s successors under a and b are {5} and {4,5}.

{4,5}'s successor under b is {5}.

Accepting states of D are those state sets that contain N's accepting state which is 5.

The resulting DFA is:



It is not too difficult to establish that the DFA constructed by **`MakeDeterministic`** is equivalent to the original NFA.

The idea is that each path to an accepting state in the original NFA has a corresponding path in the DFA. Similarly, all paths through the constructed DFA correspond to paths in the original NFA.

What is less obvious is the fact that the DFA that is built can sometimes be *much larger* than the original NFA. States of the DFA are identified with *sets* of NFA states.

If the NFA has n states, there are 2^n distinct sets of NFA states, and hence the DFA may have as many as 2^n states. Certain NFAs actually

exhibit this exponential blowup in size when made deterministic.

Fortunately, the NFAs built from the kind of regular expressions used to specify programming language tokens do not exhibit this problem when they are made deterministic.

As a rule, DFAs used for scanning are simple and compact.

If creating a DFA is impractical (because of size or speed-of-generation concerns), we can scan using an NFA. Each possible path through an NFA is tracked, and reachable accepting states are identified. Scanning is slower using this approach, so it is used only when construction of a DFA is not practical.

Optimizing Finite Automata

We can improve the DFA created by **`MakeDeterministic`**.

Sometimes a DFA will have more states than necessary. For every DFA there is a unique *smallest* equivalent DFA (fewest states possible).

Some DFA's contain *unreachable states* that cannot be reached from the start state.

Other DFA's may contain *dead states* that cannot reach any accepting state.

It is clear that neither unreachable states nor dead states can participate in scanning any valid token. We therefore eliminate all such states as part of our optimization process.

We optimize a DFA by *merging together* states we know to be equivalent.

For example, two accepting states that have no transitions at all out of them are equivalent.

Why? Because they behave exactly the same way—they accept the string read so far, but will accept no additional characters.

If two states, s_1 and s_2, are equivalent, then all transitions to s_2 can be replaced with transitions to s_1. In effect, the two states are merged together into one common state.

How do we decide what states to merge together?

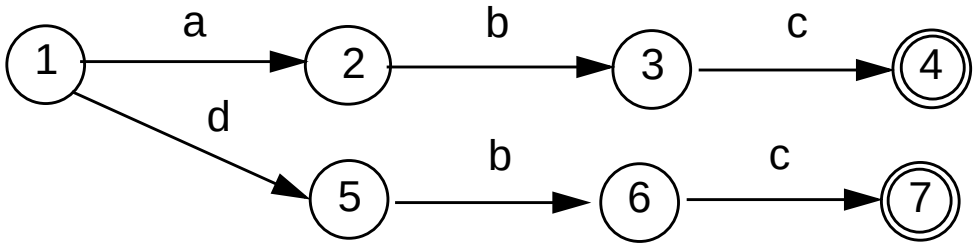
We take a *greedy* approach and try the most optimistic merger of states. By definition, accepting and non-accepting states are distinct, so we initially try to create only two states: one representing the merger of all accepting states and the other representing the merger of all non-accepting states.

This merger into only two states is almost certainly too optimistic. In particular, all the constituents of a merged state must agree on the same transition for each possible character. That is, for character c all the merged states must have no successor under c or they must all go to a single (possibly merged) state.

If all constituents of a merged state do not agree on the

transition to follow for some character, the merged state is *split* into two or more smaller states that do agree.

As an example, assume we start with the following automaton:



Initially we have a merged non-accepting state {1,2,3,5,6} and a merged accepting state {4,7}.

A merger is legal if and only if all constituent states agree on the same successor state for all characters. For example, states 3 and 6 would go to an accepting state given character c; states 1, 2, 5 would not, so a split must occur.

We will add an error state s_E to the original DFA that is the successor state under any illegal character. (Thus reaching s_E becomes equivalent to detecting an illegal token.) s_E is not a real state; rather it allows us to assume every state has a successor under every character. s_E is never merged with any real state.

Algorithm **`Split`**, shown below, splits merged states whose constituents do not agree on a common successor state for all characters. When **`Split`** terminates, we know that the states that remain merged are equivalent in that they always agree on common successors.

```
Split(FASet StateSet) {
 repeat
  for(each merged state S in StateSet) {
    Let S correspond to {s_1,…,s_n}
    for(each char c in Alphabet){
      Let t_1,…,t_n be the successor
       states to s_1,…,s_n under c
      if(t_1,…,t_n do not all belong to
          the same merged state){
           Split S into two or more new
           states such that s_i and s_j
           remain in the same merged
           state if and only if t_i and t_j
           are in the same merged state}
      }
 until no more splits are possible
}
```

Returning to our example, we initially have states {1,2,3,5,6} and {4,7}. Invoking **`Split`**, we first observe that states 3 and 6 have a common successor under c, and states 1, 2, and 5 have no successor under c (equivalently, have the error state s_E as a successor).

This forces a split, yielding {1,2,5}, {3,6} and {4,7}.

Now, for character b, states 2 and 5 would go to the merged state {3,6}, but state 1 would not, so another split occurs.

We now have: {1}, {2,5}, {3,6} and {4,7}.

At this point we are done, as all constituents of merged states agree on the same successor for each input symbol.

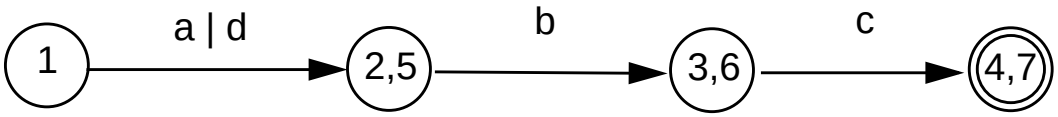
Once **`Split`** is executed, we are essentially done.

Transitions between merged states are the same as the transitions between states in the original DFA.

Thus, if there was a transition between state s_i and s_j under character c, there is now a transition under c from the merged state containing s_i to the merged state containing s_j. The start state is that merged state containing the original start state.

Accepting states are those merged states containing accepting states (recall that accepting and non-accepting states are never merged).

Returning to our example, the minimum state automaton we obtain is



Properties of Regular Expressions and Finite Automata

- Some token patterns *can't* be defined as regular expressions or finite automata. Consider the set of balanced brackets of the form [[[...]]].
  This set is defined formally as
  $\{ [^m]^m \mid m \geq 1 \}$.
  This set is *not* regular.
  No finite automaton that recognizes *exactly* this set can exist.
  Why? Consider the inputs [, [[, [[[, ...
  For two different counts (call them i and j) $[^i$ and $[^j$ must reach the same state of a given FA! (Why?)
  Once that happens, we know that if $[^i]^i$ is accepted (as it should be), the $[^j]^i$ will also be accepted (and that should not happen).

- $\overline{R} = V^* - R$ is regular if R is.
  Why?
  Build a finite automaton for R. Be careful to include transitions to an "error state" s_E for illegal characters.
  Now invert final and non-final states. What was previously accepted is now rejected, and what was rejected is now accepted. That is, $\overline{R}$ is accepted by the modified automaton.

- Not all subsets of a regular set are themselves regular. The regular expression $[^+]^+$ has a subset that isn't regular. (What is that subset?)

- Let R be a set of strings. Define R^{rev} as all strings in R, in reversed (backward) character order.
  Thus if R = {abc, def}

  then R^{rev} = {cba, fed}.

  If R is regular, then R^{rev} is too.
  Why? Build a finite automaton for R. Make sure the automaton has only one final state. Now *reverse* the direction of all transitions, and interchange the start and final states. What does the modified automation accept?

- If R_1 and R_2 are both regular, then $R_1 \cap R_2$ is also regular. We can show this two different ways:
  1. Build two finite automata, one for R1 and one for R2. Pair together states of the two automata to match R1 and R2 simultaneously. The paired-state automaton accepts only if both R1 and R2 would, so R1 $\cap$ R2 is matched.
  2. We can use the fact that R1 $\cap$ R2 is $= \overline{\overline{R_1} \cup \overline{R_2}}$ We already know union and complementation are regular.

Reading Assignment

- Read Chapter 4 of **Crafting a Compiler**

Context Free Grammars

A context-free grammar (CFG) is defined as:

- A finite terminal set V_t;
  these are the tokens produced by the scanner.
- A set of intermediate symbols, called non-terminals, V_n.
- A start symbol, a designated non-terminal, that starts all derivations.
- A set of productions (sometimes called rewriting rules) of the form
  $A \rightarrow X_1 \ldots X_m$
  X_1 to X_m may be any combination of terminals and non-terminals.
  If m =0 we have $A \rightarrow \lambda$
  which is a valid production.

Example

Prog → { Stmts }

Stmts →Stmts ; Stmt

Stmts →Stmt

Stmt →id = Expr

Expr →id

Expr →Expr + id

Often more than one production shares the same left-hand side.

Rather than repeat the left hand side, an "or notation" is used:

Prog → { Stmts }

Stmts →Stmts ; Stmt

| Stmt

Stmt →id = Expr

Expr →id

| Expr + id

Derivations

Starting with the start symbol, non-terminals are rewritten using productions until only terminals remain.

Any terminal sequence that can be generated in this manner is syntactically valid.

If a terminal sequence can't be generated using the productions of the grammar it is invalid (has syntax errors).

The set of strings derivable from the start symbol is the *language* of the grammar (sometimes denoted L(G)).

For example, starting at **Prog** we generate a terminal sequence, by repeatedly applying productions:

Prog

{ Stmts }

{ Stmts ; Stmt }

{ Stmt ; Stmt }

{ id = Expr ; Stmt }

{ id = id ; Stmt }

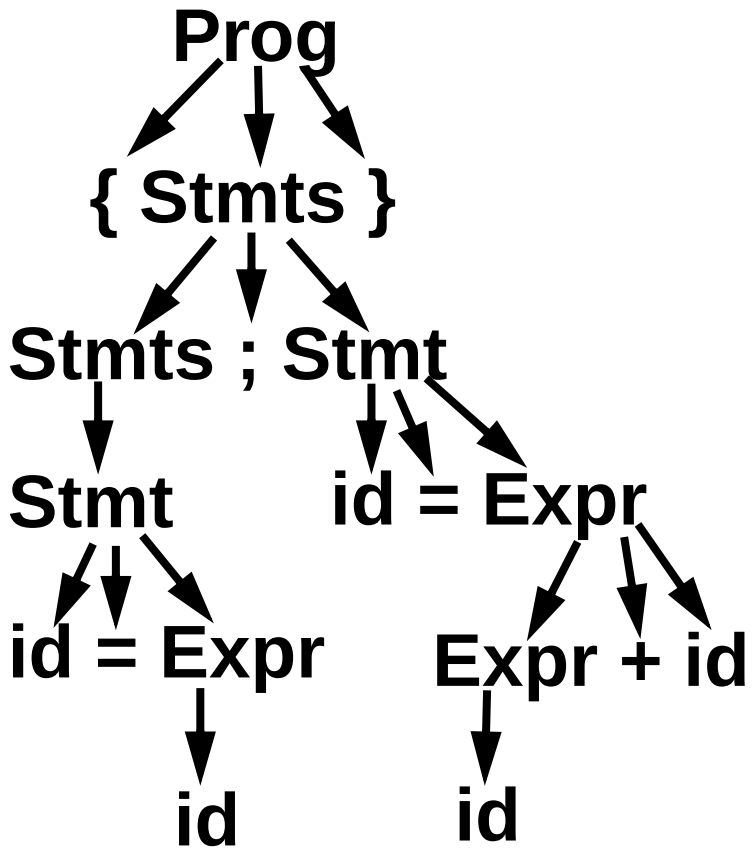
{ id = id ; id = Expr }

{ id = id ; id = Expr + id}

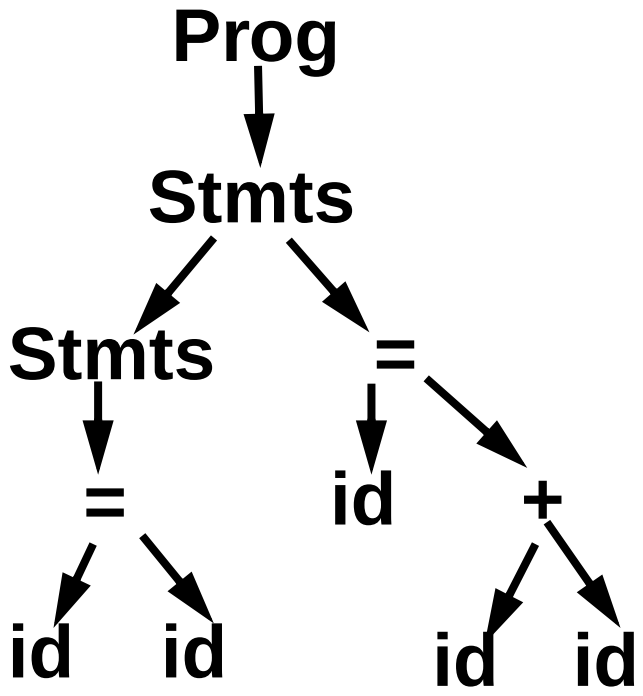
{ id = id ; id = id + id}

Parse Trees

To illustrate a derivation, we can draw a *derivation tree* (also called a *parse tree*):



An *abstract syntax tree* (AST) shows essential structure but eliminates unnecessary delimiters and intermediate symbols:



If $A \rightarrow \gamma$ is a production then
$\alpha A \beta \Rightarrow \alpha \gamma \beta$
where $\Rightarrow$ denotes a one step derivation (using production $A \rightarrow \gamma$).

We extend $\Rightarrow$ to $\Rightarrow^+$ (derives in one or more steps), and $\Rightarrow^*$ (derives in zero or more steps).

We can show our earlier derivation as

Prog $\Rightarrow$

{ Stmts } $\Rightarrow$

{ Stmts ; Stmt } $\Rightarrow$

{ Stmt ; Stmt } $\Rightarrow$

{ id = Expr ; Stmt } $\Rightarrow$

{ id = id ; Stmt } $\Rightarrow$

{ id = id ; id = Expr } $\Rightarrow$

{ id = id ; id = Expr + id} $\Rightarrow$

{ id = id ; id = id + id}

Prog $\Rightarrow^+$ **{ id = id ; id = id + id}**

When deriving a token sequence, if more than one non-terminal is present, we have a choice of which to expand next.

We must specify, at each step, which non-terminal is expanded, and what production is applied.

For simplicity we adopt a convention on what non-terminal is expanded at each step.

We can choose the leftmost possible non-terminal at each step.

A derivation that follows this rule is a *leftmost derivation*.

If we know a derivation is leftmost, we need only specify what productions are used; the choice of non-terminal is always fixed.

To denote derivations that are leftmost,

we use $\Rightarrow_L$, $\Rightarrow_L^+$, and $\Rightarrow_L^*$

The production sequence discovered by a large class of parsers (the top-down parsers) is a leftmost derivation, hence these parsers produce a *leftmost parse*.

Prog $\Rightarrow_L$

{ Stmts } $\Rightarrow_L$

{ Stmts ; Stmt } $\Rightarrow_L$

{ Stmt ; Stmt } $\Rightarrow_L$

{ id = Expr ; Stmt } $\Rightarrow_L$

{ id = id ; Stmt } $\Rightarrow_L$

{ id = id ; id = Expr } $\Rightarrow_L$

{ id = id ; id = Expr + id} $\Rightarrow_L$

{ id = id ; id = id + id}

Prog $\Rightarrow_L^+$ **{ id = id ; id = id + id}**

Rightmost Derivations

A rightmost derivation is an alternative to a leftmost derivation. Now the rightmost non-terminal is always expanded.

This derivation sequence may seem less intuitive given our normal left-to-right bias, but it corresponds to an important class of parsers (the bottom-up parsers, including CUP).

As a bottom-up parser discovers the productions used to derive a token sequence, it discovers a rightmost derivation, but in *reverse order*.

The last production applied in a rightmost derivation is the first that is discovered. The first production used, involving the start symbol, is discovered last.

The sequence of productions recognized by a bottom-up parser is a rightmost parse.

It is the exact reverse of the production sequence that represents a rightmost derivation.

For rightmost derivations, we use the notation $\Rightarrow_R$, $\Rightarrow_R^+$, and $\Rightarrow_R^*$

Prog $\Rightarrow_R$

{ Stmts } $\Rightarrow_R$

{ Stmts ; Stmt } $\Rightarrow_R$

{ Stmts ; id = Expr } $\Rightarrow_R$

{ Stmts ; id = Expr + id } $\Rightarrow_R$

{ Stmts ; id = id + id } $\Rightarrow_R$

{ Stmt ; id = id + id } $\Rightarrow_R$

{ id = Expr ; id = id + id } $\Rightarrow_R$

{ id = id ; id = id + id}

Prog $\Rightarrow^+$ **{ id = id ; id = id + id}**

You can derive the same set of tokens using leftmost and rightmost derivations; the only difference is the order in which productions are used.

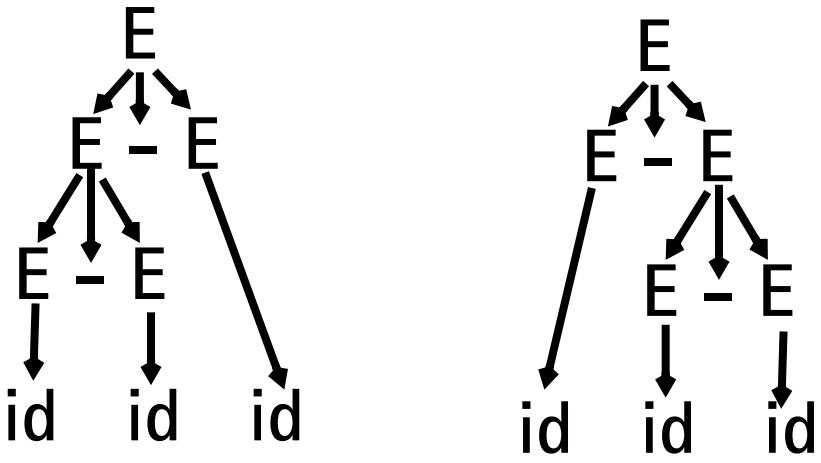
Ambiguous Grammars

Some grammars allow more than one parse tree for the same token sequence. Such grammars are *ambiguous*. Because compilers use syntactic structure to drive translation, ambiguity is undesirable—it may lead to an unexpected translation.

Consider

$$\mathbf{E} \rightarrow \mathbf{E} \text{ - } \mathbf{E}$$
$$\mid \quad \mathbf{id}$$

When parsing the input a-b-c (where a, b and c are scanned as identifiers) we can build the following two parse trees:



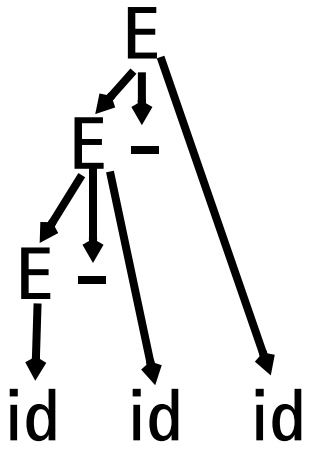
The effect is to parse a-b-c as either (a-b)-c or a-(b-c). These two groupings are certainly not equivalent.

Ambiguous grammars are usually voided in building compilers; the tools we use, like Yacc and CUP, strongly prefer unambiguous grammars.

To correct this ambiguity, we use

$$\mathbf{E \rightarrow E - id}$$
$$\mathbf{\quad\quad | \quad id}$$

Now a-b-c can only be parsed as:



Operator Precedence

Most programming languages have *operator precedence* rules that state the order in which operators are applied (in the absence of explicit parentheses). Thus in C and Java and CSX, **`a+b*c`** means compute **`b*c`**, then add in **`a`**.

These operators precedence rules can be incorporated directly into a CFG.

Consider

E $\rightarrow$ **E + T**

| **T**

T $\rightarrow$ **T * P**

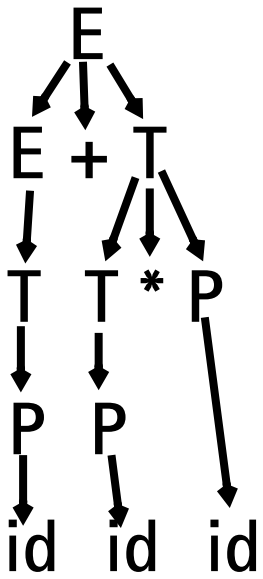
| **P**

P $\rightarrow$ **id**

| **(E)**

Does **`a+b*c`** mean **`(a+b)*c`** or **`a+(b*c)`**?

The grammar tells us! Look at the derivation tree:



The other grouping can't be obtained unless explicit parentheses are used.

(Why?)

JAVA CUP

Java CUP is a parser-generation tool, similar to Yacc.

CUP builds a Java parser for LALR(1) grammars from production rules and associated Java code fragments.

When a particular production is recognized, its associated code fragment is executed (typically to build an AST).

CUP generates a Java source file **`parser.java`**. It contains a class **`parser`**, with a method

```
Symbol parse()
```

The **`Symbol`** returned by the parser is associated with the grammar's start symbol and contains the AST for the whole source program.

The file **`sym.java`** is also built for use with a JLex-built scanner (so that both scanner and parser use the same token codes).

If an unrecovered syntax error occurs, **`Exception()`** is thrown by the parser.

CUP and Yacc accept exactly the same class of grammars—all LL(1) grammars, plus many useful non-LL(1) grammars.

CUP is called as

```
java java_cup.Main < file.cup
```

Java CUP Specifications

Java CUP specifications are of the form:

- Package and import specifications
- User code additions
- Terminal and non-terminal declarations
- A context-free grammar, augmented with Java code fragments

Package and Import Specifications

You define a package name as:

```
package name ;
```

You add imports to be used as:

```
import java_cup.runtime.*;
```

User Code Additions

You may define Java code to be included within the generated parser:

`action code {: /*java code */ :}`
This code is placed within the generated action class (which holds user-specified production actions).

`parser code {: /*java code */ :}`
This code is placed within the generated parser class .

`init with{: /*java code */ :}`
This code is used to initialize the generated parser.

`scan with{: /*java code */ :}`
This code is used to tell the generated parser how to get tokens from the scanner.

Terminal and Non-terminal Declarations

You define terminal symbols you will use as:

`terminal classname name`$_1$`, name`$_2$`, ...`

`classname` is a class used by the scanner for tokens (**`CSXToken`**, **`CSXIdentifierToken`**, etc.)

You define non-terminal symbols you will use as:

`non terminal classname name`$_1$`, name`$_2$`, ...`

`classname` is the class for the AST node associated with the non-terminal (**`stmtNode`**, **`exprNode`**, etc.)

Production Rules

Production rules are of the form

```
name ::= name1 name2 ... action ;
```

or

```
name ::= name1 name2 ...
action1
    |  name3 name4 ... action2
    |    ...
    ;
```

Names are the names of terminals or non-terminals, as declared earlier.

Actions are Java code fragments, of the form

```
{: /*java code */ :}
```

The Java object assocated with a symbol (a token or AST node) may be named by adding a **`:id`** suffix to a terminal or non-terminal in a rule.

RESULT names the left-hand side non-terminal.

The Java classes of the symbols are defined in the terminal and non-terminal declaration sections.

For example,

```
prog ::= LBRACE:l stmts:s RBRACE
    {: RESULT =
         new csxLiteNode(s,
           l.linenum,l.colnum); :}
```

This corresponds to the production

prog $\rightarrow$ **{ stmts }**

The left brace is named **l**; the stmts non-terminal is called **s**.

In the action code, a new **CSXLiteNode** is created and assigned to **prog**. It is constructed from the AST node associated with **s**. Its line and column numbers are those given to the left brace, **l** (by the scanner).

To tell CUP what non-terminal to use as the start symbol (**`prog`** in our example), we use the directive:

```
start with prog;
```

Example

Let's look at the CUP specification for CSX-lite. Recall its CFG is

program → **{ stmts }**

stmts → **stmt stmts**
| λ

stmt → **id = expr ;**
| **if (expr) stmt**

expr → **expr + id**
| **expr - id**
| **id**

The corresponding CUP specification is:

```
/***
This Is A Java CUP Specification For
CSX-lite, a Small Subset of The CSX
Language,  Used In Cs536
 ***/

/* Preliminaries to set up and use the
scanner.  */

import java_cup.runtime.*;
parser code {:
 public void syntax_error
  (Symbol cur_token){
   report_error(
    "CSX syntax error at line "+
    String.valueOf(((CSXToken)
       cur_token.value).linenum),
    null);}
:};

init with {:                :};
scan with {:
   return Scanner.next_token();
:};
```

```
/* Terminals (tokens returned by the
scanner). */
terminal CSXIdentifierToken IDENTIFIER;
terminal CSXToken SEMI, LPAREN, RPAREN,
ASG, LBRACE, RBRACE;
terminal CSXToken PLUS, MINUS, rw_IF;

/* Non terminals */
non terminal csxLiteNode prog;
non terminal stmtsNode   stmts;
non terminal stmtNode    stmt;
non terminal exprNode    exp;
non terminal nameNode    ident;


start with prog;

prog::= LBRACE:l stmts:s RBRACE
 {: RESULT=
     new csxLiteNode(s,
          l.linenum,l.colnum); :}
;

stmts::= stmt:s1  stmts:s2
 {: RESULT=
     new stmtsNode(s1,s2,
       s1.linenum,s1.colnum);
 :}
```

```
|
 {: RESULT= stmtsNode.NULL; :}
;
stmt::= ident:id ASG exp:e SEMI
 {: RESULT=
       new asgNode(id,e,
           id.linenum,id.colnum);
 :}

| rw_IF:i LPAREN exp:e RPAREN  stmt:s
 {: RESULT=new ifThenNode(e,s,
            stmtNode.NULL,
            i.linenum,i.colnum); :}
;
exp::=
 exp:leftval PLUS:op ident:rightval
 {: RESULT=new binaryOpNode(leftval,
      sym.PLUS, rightval,
      op.linenum,op.colnum); :}

| exp:leftval MINUS:op ident:rightval
 {: RESULT=new binaryOpNode(leftval,
           sym.MINUS,rightval,
           op.linenum,op.colnum); :}
| ident:i
 {: RESULT = i; :}
;
```

```
ident::= IDENTIFIER:i
 {: RESULT = new nameNode(
   new identNode(i.identifierText,
                 i.linenum,i.colnum),
   exprNode.NULL,
   i.linenum,i.colnum); :}
;
```

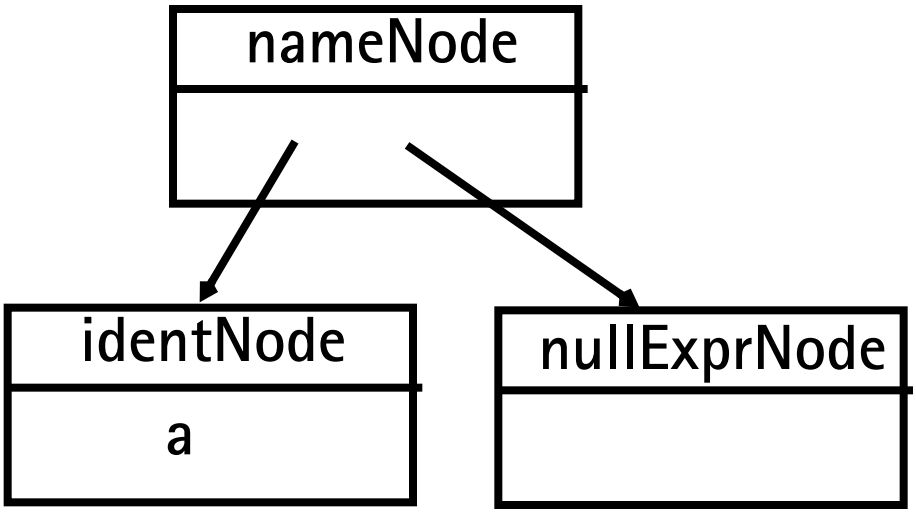
Let's parse

{ a = b ; }

First, **a** is parsed using

```
ident::= IDENTIFIER:i
 {: RESULT = new nameNode(
   new identNode(i.identifierText,
                 i.linenum,i.colnum),
   exprNode.NULL,
   i.linenum,i.colnum); :}
```

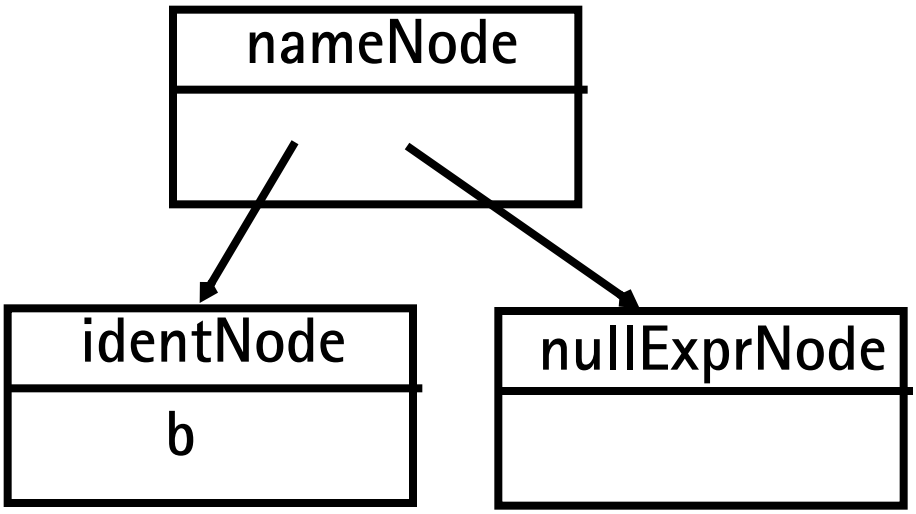
We build



Next, **b** is parsed using

```
ident::= IDENTIFIER:i
 {: RESULT = new nameNode(
   new identNode(i.identifierText,
                 i.linenum,i.colnum),
   exprNode.NULL,
   i.linenum,i.colnum); :}
```

We build



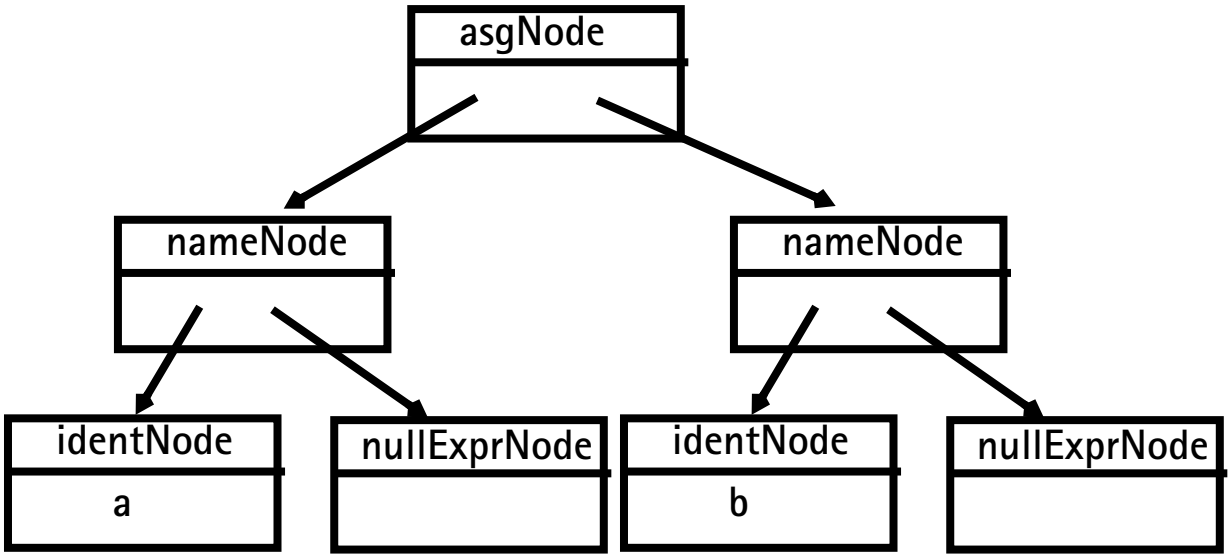
Then `b`'s subtree is recognized as an **exp**:

```
| ident:i
  {: RESULT = i; :}
```

Now the assignment statement is recognized:

```
stmt::= ident:id ASG exp:e SEMI
  {: RESULT=
        new asgNode(id,e,
            id.linenum,id.colnum);
  :}
```

We build

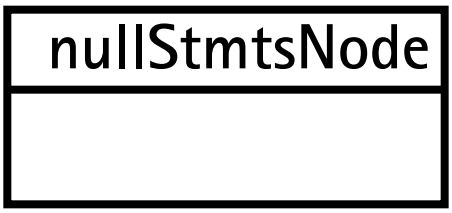


The **stmts** $\rightarrow \lambda$ production is matched (indicating that there are no more statements in the program).

CUP matches

```
stmts::=
 {: RESULT= stmtsNode.NULL; :}
```

and we build



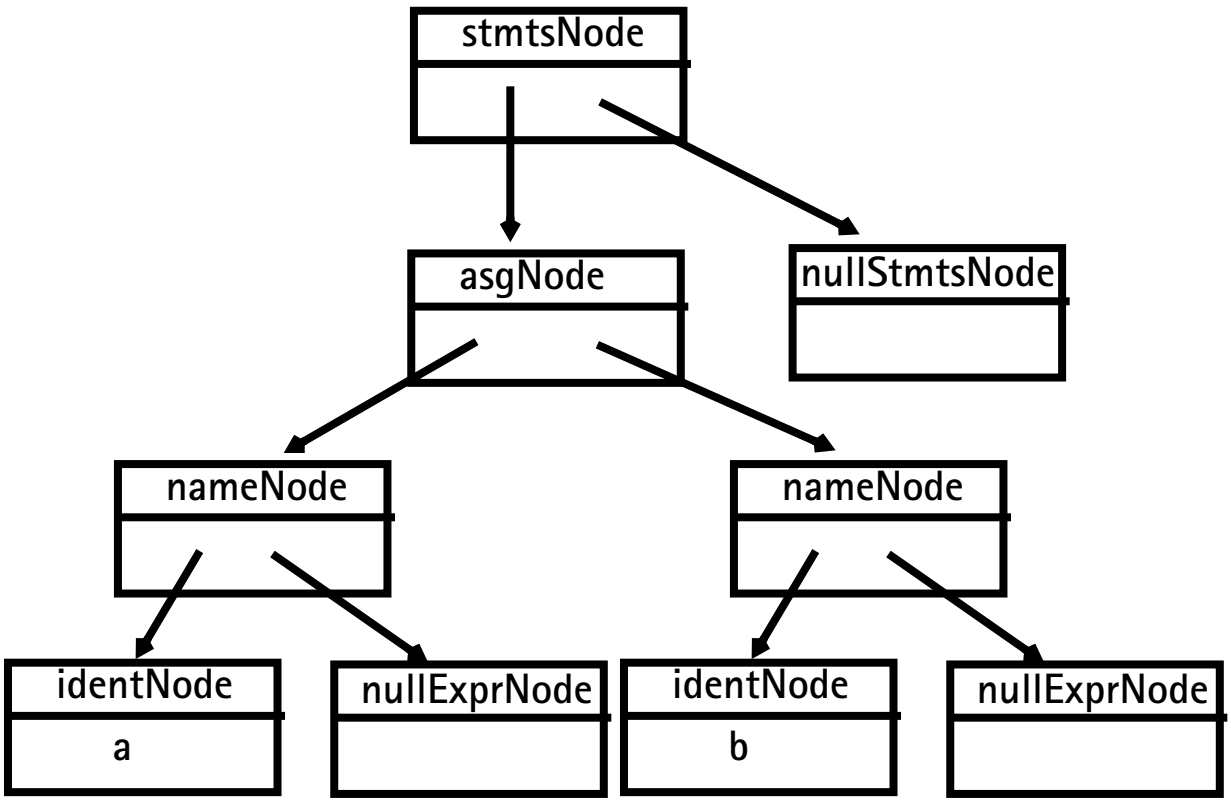
Next,

stmts $\rightarrow$ **stmt stmts**

is matched using

```
stmts::= stmt:s1  stmts:s2
 {: RESULT=
     new stmtsNode(s1,s2,
       s1.linenum,s1.colnum);

 :}
```

This builds



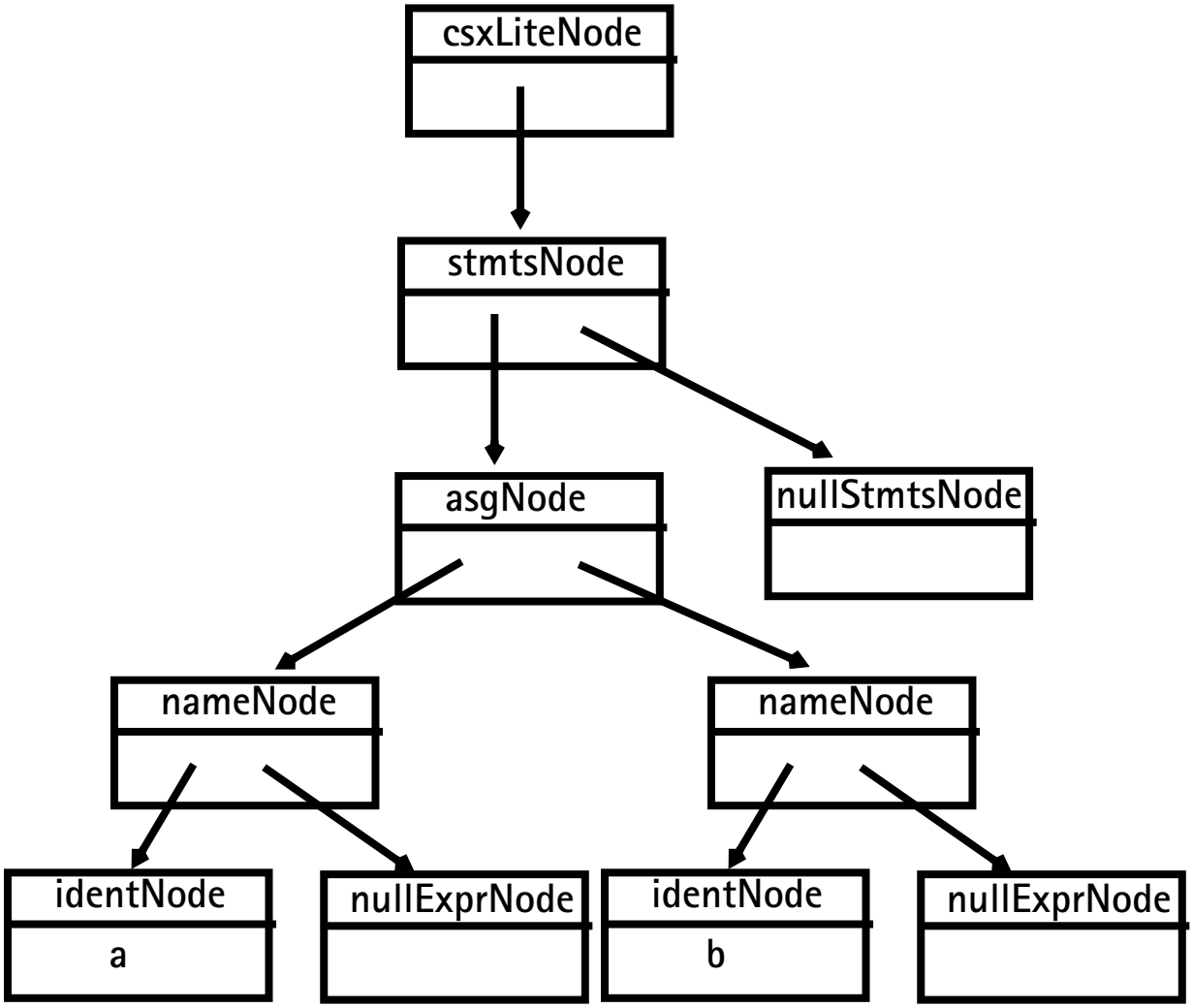
As the last step of the parse, the parser matches

program → { stmts }

using the CUP rule

```
prog::= LBRACE:l stmts:s RBRACE
 {: RESULT=
     new csxLiteNode(s,
          l.linenum,l.colnum); :}
;
```

The final AST reurned by the parser is



Errors in Context-Free Grammars

Context-free grammars can contain errors, just as programs do. Some errors are easy to detect and fix; others are more subtle.

In context-free grammars we start with the start symbol, and apply productions until a terminal string is produced.

Some context-free grammars may contain *useless* non-terminals.

Non-terminals that are unreachable (from the start symbol) or that derive no terminal string are considered useless.

Useless non-terminals (and productions that involve them) can be safely removed from a

grammar without changing the language defined by the grammar.

A grammar containing useless non-terminals is said to be *non-reduced*.

After useless non-terminals are removed, the grammar is *reduced*.

Consider

$$\mathbf{S} \rightarrow \mathbf{A}\ \ \mathbf{B}$$

$$\mid\ \ \mathbf{x}$$

$$\mathbf{B} \rightarrow \mathbf{b}$$

$$\mathbf{A} \rightarrow \mathbf{a}\ \ \mathbf{A}$$

$$\mathbf{C} \rightarrow \mathbf{d}$$

Which non-terminals are unreachable? Which derive no terminal string?

Finding Useless Non-terminals

To find non-terminals that can derive one or more terminal strings, we'll use a marking algorithm.

We iteratively mark terminals that can derive a string of terminals, until no more non-terminals can be marked. Unmarked non-terminals are useless.

(1) Mark all terminal symbols

(2) Repeat
    If all symbols on the
      righthand side of a
      production are marked
    Then mark the lefthand side
Until no more non-terminals
    can be marked

We can use a similar marking algorithm to determine which non-terminals can be reached from the start symbol:

(1) Mark the Start Symbol

(2) Repeat
    If the lefthand side of a production is marked
    Then mark all non-terminals in the righthand side
Until no more non-terminals can be marked

λ Derivations

When parsing, we'll sometimes need to know which non-terminals can derive λ. (λ is "invisible" and hence tricky to parse).

We can use the following marking algorithm to decide which non-terminals derive λ

(1) For each production $A \rightarrow \lambda$
    mark A

(2) Repeat
        If the entire righthand
        side of a production
        is marked
        Then mark the lefthand side
    Until no more non-terminals
        can be marked

As an example consider

$$\mathbf{S} \rightarrow \mathbf{A}\ \ \mathbf{B}\ \ \mathbf{C}$$

$$\mathbf{A} \rightarrow \mathbf{a}$$

$$\mathbf{B} \rightarrow \mathbf{C}\ \ \mathbf{D}$$

$$\mathbf{D} \rightarrow \mathbf{d}$$

$$\mid\ \lambda$$

$$\mathbf{C} \rightarrow \mathbf{c}$$

$$\mid\ \lambda$$

Recall that compilers prefer an unambiguous grammar because a unique parse tree structure can be guaranteed for all inputs.

Hence a unique translation, guided by the parse tree structure, will be obtained.

We would like an algorithm that checks if a grammar is ambiguous.

Unfortunately, it is undecidable whether a given CFG is ambiguous, so such an algorithm is impossible to create.

Fortunately for certain grammar classes, including those for which we can generate parsers, we can prove included grammars are unambiguous.

Potentially, the most serious flaw that a grammar might have is that it generates the “wrong language."

This is a subtle point as a grammar serves as the *definition* of a language.

For established languages (like C or Java) there is usually a suite of programs created to test and validate new compilers. An incorrect grammar will almost certainly lead to incorrect compilations of test programs, which can be automatically recognized.

For new languages, initial implementors must thoroughly test the parser to verify that inputs are scanned and parsed as expected.

Parsers and Recognizers

Given a sequence of tokens, we can ask:

"Is this input syntactically valid?"

(Is it generable from the grammar?).

A program that answers this question is a *recognizer*.

Alternatively, we can ask:

"Is this input valid and, if it is, what is its structure (parse tree)?"

A program that answers this more general question is termed a *parser*.

We plan to use language structure to drive compilers, so we will be especially interested in parsers.

Two general approaches to parsing exist.

The first approach is *top-down*.

A parser is top-down if it "discovers" the parse tree corresponding to a token sequence by starting at the top of the tree (the start symbol), and then expanding the tree (via predictions) in a depth-first manner.

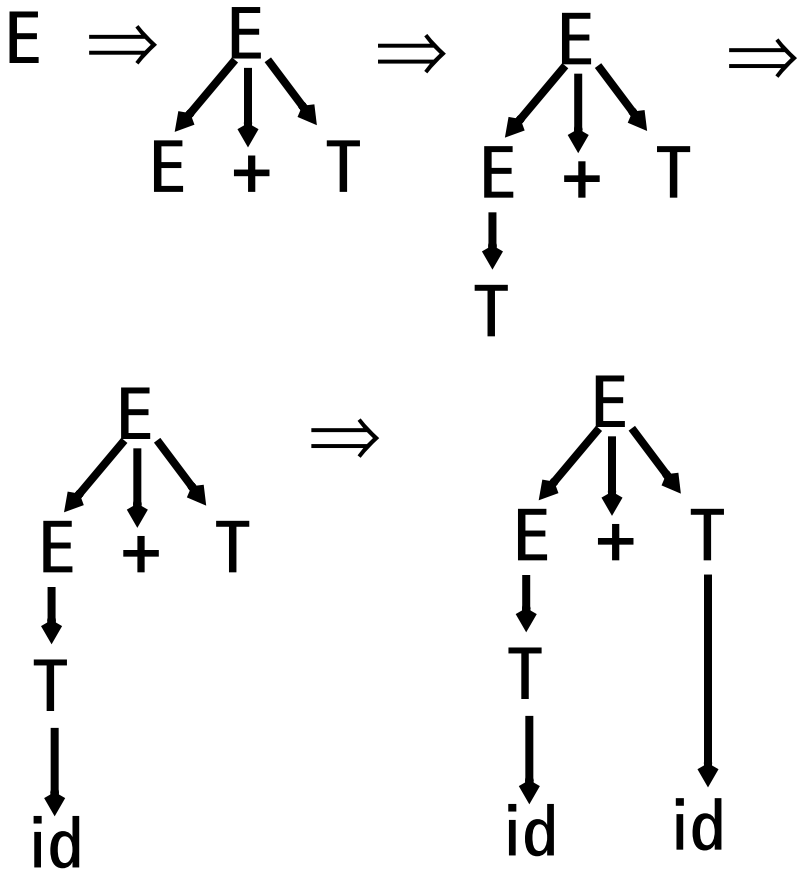
Top-down parsing techniques are *predictive* in nature because they always predict the production that is to be matched before matching actually begins.

Consider

$\mathbf{E \rightarrow E + T \quad | \quad T}$

$\mathbf{T \rightarrow T * id \quad | \quad id}$

To parse **id+id** in a top-down manner, a parser build a parse tree in the following steps:



A wide variety of parsing techniques take a different approach.

They belong to the class of *bottom-up* parsers.

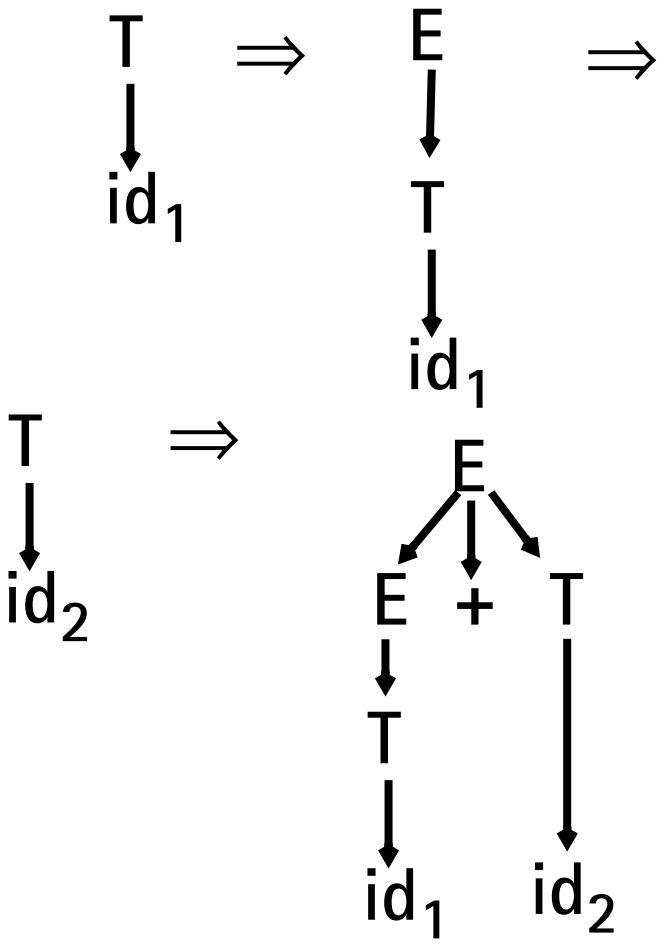
As the name suggests, bottom-up parsers discover the structure of a parse tree by beginning at its bottom (at the leaves of the tree which are terminal symbols) and determining the productions used to generate the leaves.

Then the productions used to generate the immediate parents of the leaves are discovered.

The parser continues until it reaches the production used to expand the start symbol.

At this point the entire parse tree has been determined.

A bottom-up parse of $\mathbf{id_1+id_2}$ would follow the following steps:



A Simple Top-Down Parser

We'll build a rudimentary top-down parser that simply tries each possible expansion of a non-terminal, in order of production definition.

If an expansion leads to a token sequence that doesn't match the current token being parsed, we *backup* and try the next possible production choice.

We stop when all the input tokens are correctly matched or when all possible production choices have been tried.

Example

Given the productions

S → **a**

| **(S)**

we try **a**, then **(a)**, then **((a))**, etc.

Let's next try an additional alternative:

S → **a**

| **(S)**

| **(S]**

Let's try to parse **a**, then **(a]**, then **((a]]**, etc.

We'll count the number of productions we try for each input.

- For input = **a**
  We try **S** → **a** which works.
  Matches needed = 1
- For input = **(a]**
  We try **S** → **a** which fails.
  We next try **S** → **(S)**.
  We expand the inner **S** three different ways; all fail.
  Finally, we try **S** → **(S]**.
  The inner **S** expands to **a**, which works.
  Total matches tried =
  1 + (1+3)+(1+1)= 7.
- For input = **((a]]**
  We try **S** → **a** which fails.
  We next try **S** → **(S)**.
  We match the inner **S** to **(a]** using 7 steps, then fail to match the last **]**.
  Finally, we try **S** → **(S]**.
  We match the inner **S** to **(a]** using 7

steps, then match the last **]**.
Total matches tried =
1 + (1+7)+(1+7)= 17.

- For input = **(((a]]]**
  We try **S** → **a** which fails.
  We next try **S** → **(S)**.
  We match the inner **S** to **((a]]** using 17 steps, then fail to match the last **]**.
  Finally, we try **S** → **(S]**.
  We match the inner **S** to **((a]]** using 17 steps, then match the last **]**.
  Total matches tried =
  1 + (1+17) + (1+17) = 37.

Adding one extra **(...]** pair *doubles* the number of matches we need to do the parse.

In fact to parse $\mathbf{(^i a]^i}$ takes $5*2^i-3$ matches. This is *exponential* growth!

With a more effective *dynamic programming* approach, in which results of intermediate parsing steps are cached, we can reduce the number of matches needed to $\mathbf{n}^3$ for an input with **n** tokens.

Is this acceptable?

No!

Typical source programs have at least 1000 tokens, and $1000^3 = 10^9$ is a lot of steps, even for a fast modern computer.

The solution?

—Smarter selection in the choice of productions we try.

Reading Assignment

Read Chapter 5 of

Crafting a Compiler, Second Edition.

Prediction

We want to avoid trying productions that can't possibly work.

For example, if the current token to be parsed is an identifier, it is useless to try a production that begins with an integer literal.

Before we try a production, we'll consider the set of terminals it might initially produce. If the current token is in this set, we'll try the production.

If it isn't, there is no way the production being considered could be part of the parse, so we'll ignore it.

A *predict function* tells us the set of tokens that might be initially generated from any production.

For $A \rightarrow X_1 \ldots X_n$, Predict($A \rightarrow X_1 \ldots X_n$) = Set of all initial (first) tokens derivable from $A \rightarrow X_1 \ldots X_n$

$= \{a \text{ in } V_t \mid A \rightarrow X_1 \ldots X_n \Rightarrow^* a \ldots\}$

For example, given

Stmt → Label id = Expr ;
| Label if Expr then Stmt ;
| Label read (IdList) ;
| Label id (Args) ;

Label → intlit :
| λ

Production	Predict Set
Stmt → Label id = Expr ;	**{id, intlit}**
Stmt → Label if Expr then Stmt ;	**{if, intlit}**
Stmt → Label read (IdList) ;	**{read, intlit}**
Stmt → Label id (Args) ;	**{id, intlit}**

We now will match a production p only if the next unmatched token is in p's predict set. We'll avoid trying productions that clearly won't work, so parsing will be faster.

But what is the predict set of a λ-production?

It can't be what's generated by λ (which is nothing!), so we'll define it as the tokens that can *follow* the use of a λ-production.

That is, Predict($A \rightarrow \lambda$) = Follow(A)

where (by definition)

Follow(A) = {a in V_t | $S \Rightarrow^+ \ldots Aa \ldots$}

In our example,
Follow(Label $\rightarrow \lambda$) ={ id, if, read }

(since these terminals can immediately follow uses of Label in the given productions).

Now let's parse

id (intlit) ;

Our start symbol is **Stmt** and the initial token is **id**.

id can predict

Stmt $\rightarrow$ **Label id = Expr ;**

id then predicts **Label** $\rightarrow \lambda$

The **id** is matched, but "**(**" doesn't match "**=**" so we backup and try a different production for **Stmt**.

id also predicts

Stmt $\rightarrow$ **Label id (Args) ;**

Again, **Label** $\rightarrow \lambda$ is predicted and used, and the input tokens can match the rest of the remaining production.

We had only one misprediction, which is better than before.

Now we'll rewrite the productions a bit to make predictions easier.

We remove the **Label** prefix from all the statement productions (now **intlit** won't predict all four productions).

We now have

Stmt $\rightarrow$ **Label BasicStmt**

BasicStmt $\rightarrow$ **id = Expr ;**

| **if Expr then Stmt ;**

| **read (IdList) ;**

| **id (Args) ;**

Label $\rightarrow$ **intlit :**

| λ

Now **id** predicts two different **BasicStmt** productions. If we rewrite these two productions into

BasicStmt $\rightarrow$ **id StmtSuffix**

StmtSuffix $\rightarrow$ **= Expr ;**

| **(Args) ;**

we no longer have any doubt over which production id predicts.

We now have

Production	Predict Set
Stmt → Label BasicStmt	**Not needed!**
BasicStmt → id StmtSuffix	**{id}**
BasicStmt → if Expr then Stmt ;	**{if}**
BasicStmt → read (IdList) ;	**{read}**
StmtSuffix → (Args) ;	**{ (}**
StmtSuffix → = Expr ;	**{ = }**
Label → intlit :	**{intlit}**
Label → λ	**{if, id, read}**

This grammar generates the same statements as our original grammar did, but now prediction never fails!

Whenever we must decide what production to use, the predict sets for productions with the same lefthand side are always disjoint.

Any input token will predict a unique production or no production at all (indicating a syntax error).

If we never mispredict a production, we never backup, so parsing will be fast and absolutely accurate!

LL(1) Grammars

A context-free grammar whose Predict sets are always disjoint (for the same non-terminal) is said to be *LL(1)*.

LL(1) grammars are ideally suited for top-down parsing because it is always possible to correctly predict the expansion of any non-terminal. No backup is ever needed.

Formally, let

First($X_1 ... X_n$) =

$\{a \text{ in } V_t \mid A \rightarrow X_1 ... X_n \Rightarrow^* a ...\}$

Follow(A) = $\{a \text{ in } V_t \mid S \Rightarrow^+ ... Aa ...\}$

Predict($A \rightarrow X_1 ... X_n$) =

If $X_1 ... X_n \Rightarrow^* \lambda$

Then First($X_1 ... X_n$) U Follow(A)

Else First($X_1 ... X_n$)

If some CFG, G, has the property that for all pairs of distinct productions with the same lefthand side,
$A \rightarrow X_1 ... X_n$ and $A \rightarrow Y_1 ... Y_m$
it is the case that

Predict($A \rightarrow X_1 ... X_n$) $\cap$

Predict($A \rightarrow Y_1 ... Y_m$) = ϕ

then G is LL(1).

LL(1) grammars are easy to parse in a top-down manner since predictions are always correct.

Example

Production	Predict Set
$S \rightarrow A\ a$	{b,d,a}
$A \rightarrow B\ D$	{b, d, a}
$B \rightarrow b$	{ b }
$B \rightarrow \lambda$	{d, a}
$D \rightarrow d$	{ d }
$D \rightarrow \lambda$	{ a }

Since the predict sets of both B productions and both D productions are disjoint, this grammar is LL(1).

Recursive Descent Parsers

An early implementation of top-down (LL(1)) parsing was recursive descent.

A parser was organized as a set of *parsing procedures*, one for each non-terminal. Each parsing procedure was responsible for parsing a sequence of tokens derivable from its non-terminal.

For example, a parsing procedure, A, when called, would call the scanner and match a token sequence derivable from A.

Starting with the start symbol's parsing procedure, we would then match the entire input, which must be derivable from the start symbol.

This approach is called recursive descent because the parsing procedures were typically *recursive*, and they *descended* down the input's parse tree (as top-down parsers always do).

Building A Recursive Descent Parser

We start with a procedure **Match**, that matches the current input token against a predicted token:

```
void Match(Terminal a) {
  if (a == currentToken)
     currentToken = Scanner();
  else SyntaxErrror();}
```

To build a parsing procedure for a non-terminal A, we look at all productions with A on the lefthand side:

$A \rightarrow X_1 ... X_n \mid A \rightarrow Y_1 ... Y_m \mid ...$

We use predict sets to decide which production to match (LL(1) grammars always have disjoint predict sets).

We match a production's righthand side by calling **Match** to

match terminals, and calling parsing procedures to match non-terminals.

The general form of a parsing procedure for

$A \rightarrow X_1 \ldots X_n \mid A \rightarrow Y_1 \ldots Y_m \mid \ldots$ is

```
void A() {
 if (currentToken in Predict(A→X1…Xn))
   for(i=1;i<=n;i++)
     if (X[i] is a terminal)
         Match(X[i]);
     else X[i]();
 else
 if (currentToken in Predict(A→Y1…Ym))
   for(i=1;i<=m;i++)
     if (Y[i] is a terminal)
         Match(Y[i]);
     else Y[i]();
 else
      // Handle other A →… productions
 else // No production predicted
     SyntaxError();
}
```

Usually this general form isn't used.

Instead, each production is "macro-expanded" into a sequence of **`Match`** and parsing procedure calls.

Example: CSX-Lite

Production	Predict Set
Prog → { Stmts } Eof	{
Stmts → Stmt Stmts	id if
Stmts → λ	}
Stmt → id = Expr ;	id
Stmt → if (Expr) Stmt	if
Expr → id Etail	id
Etail → + Expr	+
Etail → - Expr	-
Etail → λ	) ;

CSX-Lite Parsing Procedures

```
void Prog() {
  Match("{");
  Stmts();
  Match("}");
  Match(Eof);
}


void Stmts() {
 if (currentToken == id ||
     currentToken == if){
     Stmt();
     Stmts();
 } else {
     /* null */
}}

void Stmt() {
 if (currentToken == id){
     Match(id);
     Match("=");
     Expr();
     Match(";");
 } else {
     Match(if);
     Match("(");
     Expr();
     Match(")");
     Stmt();
}}
```

```
void Expr() {
  Match(id);
  Etail();
}


void Etail() {
 if (currentToken == "+") {
     Match("+");
     Expr();
 } else if (currentToken == "-"){
     Match("-");
     Expr();
 } else {
     /* null */
}}
```

Let's use recursive descent to parse

`{ a = b + c; } Eof`

We start by calling **`Prog()`** since this represents the start symbol.

Calls Pending	Remaining Input
`Prog()`	`{ a = b + c; } Eof`

`Match("{");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`{ a = b + c; } Eof`

`Stmts();`
`Match("}");`
`Match(Eof);`
	`a = b + c; } Eof`

`Stmt();`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`a = b + c; } Eof`

`Match(id);`
`Match("=");`
`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`a = b + c; } Eof`

Calls Pending	Remaining Input

`Match("=");`
`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`= b + c; } Eof`

`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`b + c; } Eof`

`Match(id);`
`Etail();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`b + c; } Eof`

`Etail();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`+ c; } Eof`

Calls Pending	Remaining Input

`Match("+");`
`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`+ c; } Eof`

`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`c; } Eof`

`Match(id);`
`Etail();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`c; } Eof`

`Etail();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`; } Eof`

`/* null */`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`; } Eof`

Calls Pending	Remaining Input

`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`; } Eof`

`Stmts();`
`Match("}");`
`Match(Eof);`
	`} Eof`

`/* null */`
`Match("}");`
`Match(Eof);`
	`} Eof`

`Match("}");` `Match(Eof);`	`} Eof`
`Match(Eof);`	`Eof`
Done!	All input matched

Syntax Errors in Recursive Descent Parsing

In recursive descent parsing, syntax errors are automatically detected. In fact, they are detected *as soon as possible* (as soon as the first illegal token is seen).

How? When an illegal token is seen by the parser, either it fails to predict any valid production or it fails to match an expected token in a call to **`Match`**.

Let's see how the following illegal CSX-lite program is parsed:

```
{ b + c = a; } Eof
```

(Where should the first syntax error be detected?)

Calls Pending	Remaining Input
`Prog()`	`{ b + c = a; } Eof`

`Match("{");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`{ b + c = a; } Eof`

`Stmts();`
`Match("}");`
`Match(Eof);`
	`b + c = a; } Eof`

`Stmt();`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`b + c = a; } Eof`

`Match(id);`
`Match("=");`
`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`b + c = a; } Eof`

Calls Pending	Remaining Input

`Match("=");`
`Expr();`
`Match(";");`
`Stmts();`
`Match("}");`
`Match(Eof);`
	`+ c = a; } Eof`

Call to Match fails!	`+ c = a; } Eof`

Table-Driven Top-Down Parsers

Recursive descent parsers have many attractive features. They are actual pieces of code that can be read by programmers and extended.

This makes it fairly easy to understand how parsing is done.

Parsing procedures are also convenient places to add code to build ASTs, or to do type-checking, or to generate code.

A major drawback of recursive descent is that it is quite inconvenient to change the grammar being parsed. Any change, even a minor one, may force parsing procedures to be

reprogrammed, as productions and predict sets are modified.

To a less extent, recursive descent parsing is less efficient than it might be, since subprograms are called just to match a single token or to recognize a righthand side.

An alternative to parsing procedures is to encode all prediction in a parsing table. A pre-programed driver program can use a parse table (and list of productions) to parse any LL(1) grammar.

If a grammar is changed, the parse table and list of productions will change, but the driver need not be changed.

LL(1) Parse Tables

An LL(1) parse table, T, is a two-dimensional array. Entries in T are production numbers or blank (error) entries.

T is indexed by:

- A, a non-terminal. A is the non-terminal we want to expand.
- CT, the current token that is to be matched.
- $T[A][CT] = A \rightarrow X_1 \ldots X_n$
  if CT is in Predict($A \rightarrow X_1 \ldots X_n$)
  T[A][CT] = error
  if CT predicts no production with A as its lefthand side

CSX-lite Example

	Production	Predict Set
1	Prog → { Stmts } Eof	{
2	Stmts → Stmt Stmts	id if
3	Stmts → λ	}
4	Stmt → id = Expr ;	id
5	Stmt → if (Expr) Stmt	if
6	Expr → id Etail	id
7	Etail → + Expr	+
8	Etail → - Expr	-
9	Etail → λ	) ;

	{	}	if	(	)	id	=	+	-	;	eof
Prog	1										
Stmts		3	2			2					
Stmt			5			4					
Expr						6					
Etail					9			7	8	9	

LL(1) Parser Driver

Here is the driver we'll use with the LL(1) parse table. We'll also use a *parse stack* that remembers symbols we have yet to match.

```
void LLDriver(){
  Push(StartSymbol);
  while(! stackEmpty()){
  //Let X=Top symbol on parse stack
  //Let CT = current token to match
    if (isTerminal(X)) {
      match(X); //CT is updated
      pop();    //X is updated
    } else if (T[X][CT] != Error){
      //Let T[X][CT] = X→Y1...Ym
      Replace X with
         Y1...Ym on parse stack
    } else SyntaxError(CT);
  }
}
```

Example of LL(1) Parsing

We'll again parse

```
{ a = b + c; } Eof
```

We start by placing Prog (the start symbol) on the parse stack.

Parse Stack	Remaining Input
`Prog`	`{ a = b + c; } Eof`

`{`
`Stmts`
`}`
`Eof`
	`{ a = b + c; } Eof`

`Stmts`
`}`
`Eof`
	`a = b + c; } Eof`

`Stmt`
`Stmts`
`}`
`Eof`
	`a = b + c; } Eof`

Parse Stack	Remaining Input
id = Expr ; Stmts } Eof	a = b + c; } Eof
= Expr ; Stmts } Eof	= b + c; } Eof
Expr ; Stmts } Eof	b + c; } Eof
id Etail ; Stmts } Eof	b + c; } Eof

Parse Stack	Remaining Input
Etail ; Stmts } Eof	+ c; } Eof
+ Expr ; Stmts } Eof	+ c; } Eof
Expr ; Stmts } Eof	c; } Eof
id Etail ; Stmts } Eof	c; } Eof

Parse Stack	Remaining Input
Etail ; Stmts } Eof	; } Eof
; Stmts } Eof	; } Eof
Stmts } Eof	} Eof
} Eof	} Eof
Eof	Eof
Done!	All input matched

Syntax Errors in LL(1) Parsing

In LL(1) parsing, syntax errors are automatically detected as soon as the first illegal token is seen.

How? When an illegal token is seen by the parser, either it fetches an error entry from the LL(1) parse table *or* it fails to match an expected token.

Let's see how the following illegal CSX-lite program is parsed:

```
{ b + c = a; } Eof
```

(Where should the first syntax error be detected?)

Parse Stack	Remaining Input
`Prog`	{ b + c = a; } Eof

`{`
`Stmts`
`}`
`Eof`
	{ b + c = a; } Eof

`Stmts`
`}`
`Eof`
	b + c = a; } Eof

`Stmt`
`Stmts`
`}`
`Eof`
	b + c = a; } Eof

`id`
`=`
`Expr`
`;`
`Stmts`
`}`
`Eof`
	b + c = a; } Eof

Parse Stack	Remaining Input

`=`
`Expr`
`;`
`Stmts`
`}`
`Eof`
	+ c = a; } Eof

Current token (+) fails to match expected token (=)!	+ c = a; } Eof

How do LL(1) Parsers Build Syntax Trees?

So far our LL(1) parser has acted like a recognizer. It verifies that input token are syntactically correct, but it produces no output.

Building complete (concrete) parse trees automatically is fairly easy.

As tokens and non-terminals are matched, they are pushed onto a second stack, the *semantic stack*.

At the end of each production, an action routine pops off n items from the semantic stack (where n is the length of the production's righthand side). It then builds a syntax tree whose root is the

lefthand side, and whose children are the n items just popped off.

For example, for production

Stmt → id = Expr ;

the parser would include an action symbol after the "**;**" whose actions are:

```
P4 = pop(); // Semicolon token
P3 = pop(): // Syntax tree for Expr
P2 = pop(); // Assignment token
P1 = pop(); // Identifier token
Push(new StmtNode(P1,P2,P3,P4));
```

Creating Abstract Syntax Trees

Recall that we prefer that parsers generate abstract syntax trees, since they are simpler and more concise.

Since a parser generator can't know what tree structure we want to keep, we must allow the user to define "custom" action code, just as Java CUP does.

We allow users to include "code snippets" in Java or C. We also allow labels on symbols so that we can refer to the tokens and tress we wish to access. Our production and action code will now look like this:

Stmt → id:i = Expr:e ;

```
{: RESULT = new StmtNode(i,e); :}
```

How do We Make Grammars LL(1)?

Not all grammars are LL(1); sometimes we need to modify a grammar's productions to create the disjoint Predict sets LL1) requires.

There are two common problems in grammars that make unique prediction difficult or impossible:

1. Common prefixes.
   Two or more productions with the same lefthand side begin with the same symbol(s).
   For example,

   Stmt → id = Expr ;

   Stmt → id (Args) ;

2. Left-Recursion

A production of the form

A $\rightarrow$ **A ...**

is said to be left-recursive.

When a left-recursive production is used, a non-terminal is immediately replaced by itself (with additional symbols following).

Any grammar with a left-recursive production can *never* be LL(1).

Why?

Assume a non-terminal A reaches the top of the parse stack, with CT as the current token. The LL(1) parse table entry, T[A][CT], predicts **A** $\rightarrow$ **A ...**

We expand A again, and T[A][CT], so we predict **A** $\rightarrow$ **A ...** again. We are in an infinite prediction loop!

Eliminating Common Prefixes

Assume we have two of more productions with the same lefthand side and a common prefix on their righthand sides:

$\mathbf{A} \rightarrow \alpha\,\beta \mid \alpha\,\gamma \mid \ldots \mid \alpha\,\delta$

We create a new non-terminal, **X**.

We then rewrite the above productions into:

$\mathbf{A} \rightarrow \alpha\mathbf{X} \quad \mathbf{X} \rightarrow \beta \mid \gamma \mid \ldots \mid \delta$

For example,

Stmt → id = Expr ;

Stmt → id (Args) ;

becomes

Stmt → id StmtSuffix

StmtSuffix → = Expr ;

StmtSuffix → (Args) ;

Eliminating Left Recursion

Assume we have a non-terminal that is left recursive:

$\mathbf{A} \rightarrow \mathbf{A}\alpha \quad \mathbf{A} \rightarrow \beta \mid \gamma \mid \ldots \mid \delta$

To eliminate the left recursion, we create two new non-terminals, **N** and **T**.

We then rewrite the above productions into:

$\mathbf{A} \rightarrow \mathbf{N}\ \mathbf{T} \quad \mathbf{N} \rightarrow \beta \mid \gamma \mid \ldots \mid \delta$

$\mathbf{T} \rightarrow \alpha\ \mathbf{T} \mid \lambda$

For example,

Expr $\rightarrow$ **Expr + id**

Expr $\rightarrow$ **id**

becomes

Expr $\rightarrow$ **N T**

N $\rightarrow$ **id**

T $\rightarrow$ **+ id T** | λ

This simplifies to:

Expr $\rightarrow$ **id T**

T $\rightarrow$ **+ id T** | λ

Reading Assignment

Read Sections 6.1 to 6.5.1 of Crafting a Compiler featuring Java.

How does JavaCup Work?

The main limitation of LL(1) parsing is that it must predict the correct production to use when it first starts to match the production's righthand side.

An improvement to this approach is the LALR(1) parsing method that is used in JavaCUP (and Yacc and Bison too).

The LALR(1) parser is bottom-up in approach. It tracks the portion of a righthand side already matched as tokens are scanned. It may not know immediately which is the correct production to choose, so it tracks *sets* of possible matching productions.

Configurations

We'll use the notation

$$X \rightarrow A\ B \bullet C\ D$$

to represent the fact that we are trying to match the production $X \rightarrow A\ B \bullet C\ D$ with **A** and **B** matched so far.

A production with a "•" somewhere in its righthand side is called a *configuration*.

Our goal is to reach a configuration with the "dot" at the extreme right:

$$X \rightarrow A\ B\ C\ D \bullet$$

This indicates that an entire production has just been matched.

Since we may not know which production will eventually be fully matched, we may need to track a *configuration set*. A configuration set is sometimes called a *state*.

When we predict a production, we place the "dot" at the beginning of a production:

$\mathbf{X} \rightarrow \bullet\ \mathbf{A\ B\ C\ D}$

This indicates that the production may possibly be matched, but no symbols have actually yet been matched.

We may predict a λ-production:

$\mathbf{X} \rightarrow \lambda\ \bullet$

When a λ-production is predicted, it is immediately matched, since λ can be matched at any time.

Starting the Parse

At the start of the parse, we know some production with the start symbol must be used initially. We don't yet know which one, so we predict them *all*:

S → • A B C D

S → • e F g

S → • h I

...

Closure

When we encounter a configuration with the dot to the left of a non-terminal, we know we need to try to match that non-terminal.

Thus in

X → • A B C D

we need to match some production with A as its left hand side.

Which production?

We don't know, so we predict *all* possibilities:

A → • P Q R

A → • s T

...

The newly added configurations may predict other non-terminals, forcing additional productions to be included. We continue this process until no additional configurations can be added.

This process is called *closure* (of the configuration set).

Here is the closure algorithm:

```
ConfigSet Closure(ConfigSet C){
  repeat
     if (X → a •B d is in C &&
            B is a non-terminal)
        Add all configurations of
           the form   B → •g to C)
  until (no more configurations
              can be added);
  return C;
}
```

Example of Closure

Assume we have the following grammar:

S $\rightarrow$ **A b**

A $\rightarrow$ **C D**

C $\rightarrow$ **D**

C $\rightarrow$ **c**

D $\rightarrow$ **d**

To compute Closure(**S** $\rightarrow$ • **A b**) we first include all productions that rewrite A:

A $\rightarrow$ • **C D**

Now **C** productions are included:

C $\rightarrow$ • **D**

C $\rightarrow$ • **c**

Finally, the D production is added:

D $\rightarrow$ • **d**

The complete configuration set is:

S $\rightarrow$ • **A b**

A $\rightarrow$ • **C D**

C $\rightarrow$ • **D**

C $\rightarrow$ • **c**

D $\rightarrow$ • **d**

This set tells us that if we want to match an **A**, we will need to match a **C**, and this is done by matching a **c** or **d** token.

Shift Operations

When we match a symbol (a terminal or non-terminal), we *shift* the "dot" past the symbol just matched. Configurations that don't have a dot to the left of the matched symbol are deleted (since they didn't correctly anticipate the matched symbol).

The **GoTo** function computes an updated configuration set after a symbol is shifted:

```
ConfigSet GoTo(ConfigSet C,Symbol X){
  B=ϕ;
  for each configuration f in C{
      if (f is of the form A → α•X δ)
       Add A → α X •δ to B;
   }
   return Closure(B);
}
```

For example, if **c** is

$S \rightarrow \cdot\ A\ b$

$A \rightarrow \cdot\ C\ D$

$C \rightarrow \cdot\ D$

$C \rightarrow \cdot\ c$

$D \rightarrow \cdot\ d$

and **x** is **C**, then **GoTo** returns

$A \rightarrow C \cdot D$

$D \rightarrow \cdot\ d$

Reduce Actions

When the dot in a configuration reaches the rightmost position, we have matched an entire righthand side. We are ready to replace the righthand side symbols with the lefthand side of the production. The lefthand side symbol can now be considered matched.

If a configuration set can shift a token and also reduce a production, we have a potential *shift/reduce error*.

If we can reduce more than one production, we have a potential *reduce/reduce error*.

How do we decide whether to do a shift or reduce? How do we choose among more than one reduction?

We examine the next token to see if it is consistent with the potential reduce actions.

The simplest way to do this is to use Follow sets, as we did in LL(1) parsing.

If we have a configuration

$$\mathbf{A} \rightarrow \alpha \cdot$$

we will reduce this production *only if* the current token, **CT**, is in Follow(**A**).

This makes sense since if we reduce α to **A**, we can't correctly match **CT** if **CT** can't follow **A**.

Shift/Reduce and Reduce/ Reduce Errors

If we have a parse state that contains the configurations

$\mathbf{A} \rightarrow \alpha \bullet$

$\mathbf{B} \rightarrow \beta \bullet \mathbf{a}\, \gamma$

and **a** in Follow(**A**) then there is an *unresolvable* shift/reduce conflict. This grammar can't be parsed.

Similarly, if we have a parse state that contains the configurations

$\mathbf{A} \rightarrow \alpha \bullet$

$\mathbf{B} \rightarrow \beta \bullet$

and Follow(**A**) $\cap$ Follow(**B**) $\neq \phi$, then the parser has an unresolvable reduce/reduce conflict. This grammar can't be parsed.

Building Parse States

All the manipulations needed to build and complete configuration sets suggest that parsing may be slow—configuration sets need to be updated after each token is matched.

Fortunately, all the configuration sets we ever will need can be computed and tabled *in advance*, when a tool like Java Cup builds a parser.

The idea is simple. We first compute an initial parse state, s_0, that corresponds to predicting productions that expand the start symbol. We then just compute successor states for each token that might be scanned. A complete set of states can be computed. For typical

programming language grammars, only a few hundred states are needed.

Here is the algorithm that builds a complete set of parse states for a grammar:

```
StateSet BuildStates(){
 Let s0=Closure({S → •α, S → •β, ...});
 C={s0};
 while (not all states in C are marked){
  Choose any unmarked state, s, in C
  Mark s;
  For each X in
      terminals U nonterminals {
    if (GoTo(s,X) is not in C)
       Add GoTo(s,X) to C;
  }
 }
 return C;
}
```

Configuration Sets for CSX-Lite

State	Cofiguration Set
s_0	**Prog → •{ Stmts } Eof**
s_1	**Prog → { • Stmts } Eof** **Stmts → •Stmt Stmts** **Stmts → λ •** **Stmt → • id = Expr ;** **Stmt → • if (Expr) Stmt**
s_2	**Prog → { Stmts •} Eof**
s_3	**Stmts → Stmt • Stmts** **Stmts → •Stmt Stmts** **Stmts → λ •** **Stmt → • id = Expr ;** **Stmt → • if (Expr) Stmt**
s_4	**Stmt → id • = Expr ;**
s_5	**Stmt → if • (Expr) Stmt**

State	Cofiguration Set
s_6	**Prog → { Stmts } •Eof**
s_7	**Stmts → Stmt Stmts •**
s_8	**Stmt → id = • Expr ;** **Expr → • Expr + id** **Expr → • Expr - id** **Expr → • id**
s_9	**Stmt → if (• Expr) Stmt** **Expr → • Expr + id** **Expr → • Expr - id** **Expr → • id**
s_{10}	**Prog → { Stmts } Eof •**
s_{11}	**Stmt → id = Expr • ;** **Expr → Expr • + id** **Expr → Expr • - id**
s_{12}	**Expr → id •**
s_{13}	**Stmt → if (Expr •) Stmt** **Expr → Expr • + id** **Expr → Expr • - id**

State	Cofiguration Set
s_{14}	**Stmt → id = Expr ; •**
s_{15}	**Expr → Expr + • id**
s_{16}	**Expr → Expr - • id**
s_{17}	**Stmt → if (Expr) • Stmt** **Stmt → • id = Expr ;** **Stmt → • if (Expr) Stmt**
s_{18}	**Expr → Expr + id •**
s_{19}	**Expr → Expr - id •**
s_{20}	**Stmt → if (Expr) Stmt •**

Parser Action Table

We will table possible parser actions based on the current state (configuration set) and token.

Given configuration set C and input token T four actions are possible:

- Reduce i: The i-th production has been matched.
- Shift: Match the current token.
- Accept: Parse is correct and complete.
- Error: A syntax error has been discovered.

We will let A[C][T] represent the possible parser actions given configuration set C and input token T.

A[C][T] =
{Reduce i | i-th production is $\mathbf{A} \rightarrow \alpha$
and $\mathbf{A} \rightarrow \alpha \bullet$ is in C
and T in Follow(A) }
U (If ($\mathbf{B} \rightarrow \beta \bullet \mathbf{T}\ \gamma$ is in C)
{Shift} else ϕ)

This rule simply collects all the actions that a parser might do given C and T.

But we want parser actions to be unique so we require that the parser action always be *unique* for any C and T.

If the parser action isn't unique, then we have a shift/reduce error or reduce/reduce error. The grammar is then rejected as unparsable.

If parser actions are always unique then we will consider a shift of EOF to be an accept action.

An empty (or undefined) action for C and T will signify that token T is illegal given configuration set C.
A syntax error will be signaled.

LALR Parser Driver

Given the GoTo and parser action tables, a Shift/Reduce (LALR) parser is fairly simple:

```
void LALRDriver(){
  Push(S0);
  while(true){
  //Let S = Top state on parse stack
  //Let CT = current token to match
    switch (A[S][CT]) {
     case error:
        SyntaxError(CT);return;
     case accept:
        return;
     case shift:
        push(GoTo[S][CT]);
        CT= Scanner();
        break;
     case reduce i:
        //Let prod i = A→Y1...Ym
        pop m states;
        //Let S' = new top state
        push(GoTo[S'][A]);
        break;
} } }
```

Action Table for CSX-Lite

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
{	S																				
}		R3	S	R3				R2							R4						R5
if		S		S											R4			S			R5
(						S															
)													R8	S					R6	R7	
id		S		S					S	S					R4	S	S	S			
=					S																
+												S	R8	S					R6	R7	
-												S	R8	S					R6	R7	
;												S	R8						R6	R7	R5
eof							A														

GoTo Table for CSX-Lite

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
{	1																				
}			6																		
if		5		5														5			
(						9															
)														17							
id		4		4					12	12						18	19	4			
=					8																
+												15		15							
-												16		16							
;												14									
eof							10														
stmts		2		7																	
stmt		3		3																	
expr									11	13											

Example of LALR(1) Parsing

We'll again parse

`{ a = b + c; } Eof`

We start by pushing state 0 on the parse stack.

Parse Stack	Top State	Action	Remaining Input
0	Prog → •{ Stmts } Eof	Shift	`{ a = b + c; } Eof`
1 0	Prog → { • Stmts } Eof Stmts → • Stmt Stmts Stmts → λ • Stmt → • id = Expr ; Stmt → • if (Expr)	Shift	`a = b + c; } Eof`
4 1 0	Stmt → id • = Expr ;		`= b + c; } Eof`
8 4 1 0	Stmt → id = • Expr ; Expr → • Expr + id Expr → • Expr - id Expr → • id	Shift	`b + c; } Eof`

Parse Stack	Top State	Action	Remaining Input
12 8 4 1 0	Expr → id •	Reduce 8	+ c; } Eof
11 8 4 1 0	Stmt → id = Expr • ; Expr → Expr • + id Expr → Expr • - id	Shift	+ c; } Eof
15 11 8 4 1 0	Expr → Expr + • id	Shift	c; } Eof

Parse Stack	Top State	Action	Remaining Input
18 15 11 8 4 1 0	Expr → Expr + id •	Reduce 6	; } Eof
11 8 4 1 0	Stmt → id = Expr • ; Expr → Expr • + id Expr → Expr • - id	Shift	; } Eof
14 11 8 4 1 0	Stmt → id = Expr ; •	Reduce 4	} Eof

Parse Stack	Top State	Action	Remaining Input
3 1 0	Stmts → Stmt • Stmts Stmts → •Stmt Stmts Stmts → λ • Stmt → • id = Expr ; Stmt → • if (Expr) Stmt	Reduce 3	} Eof
7 3 1 0	Stmts → Stmt Stmts •	Reduce 2	} Eof
2 1 0	Prog → { Stmts •} Eof	Shift	} Eof
6 2 1 0	Prog → { Stmts } •Eof	Accept	Eof

Error Detection in LALR Parsers

In bottom-up, LALR parsers syntax errors are discovered when a blank (error) entry is fetched from the parser action table.

Let's again trace how the following illegal CSX-lite program is parsed:

```
{ b + c = a; } Eof
```

Parse Stack	Top State	Action	Remaining Input
0	Prog → •{ Stmts } Eof	Shift	{ b + c = a; } Eof
1 0	Prog → { • Stmts } Eof Stmts → • Stmt Stmts Stmts → λ • Stmt → • id = Expr ; Stmt → • if (Expr)	Shift	b + c = a; } Eof
4 1 0	Stmt → id • = Expr ;	Error (blank)	+ c = a; } Eof

LALR is More Powerful

Essentially all LL(1) grammars are LALR(1) plus many more. Grammar constructs that confuse LL(1) are readily handled.

- Common prefixes are no problem. Since sets of configurations are tracked, more than one prefix can be followed. For example, in

  Stmt → id = Expr ;
  Stmt → id (Args) ;

  after we match an id we have

  Stmt → id · = Expr ;
  Stmt → id · (Args) ;

  The next token will tell us which production to use.

- Left recursion is also not a problem. Since sets of configurations are tracked, we can follow a left-recursive production *and* all others it might use. For example, in

  Expr $\rightarrow \cdot$ **Expr + id**
  Expr $\rightarrow \cdot$ **id**

  we can first match an **id**:

  Expr $\rightarrow$ **id** $\cdot$

  Then the **Expr** is recognized:

  Expr $\rightarrow$ **Expr** $\cdot$ **+ id**

  The left-recursion is handled!

- But ambiguity will still block construction of an LALR parser. Some shift/reduce or reduce/ reduce conflict must appear. (Since two or more distinct parses are possible for some input). Consider our original productions for if-then and if-then-else statements:

Stmt → if (Expr) Stmt •

Stmt → if (Expr) Stmt • else Stmt

Since **else** can follow **Stmt**, we have an unresolvable shift/reduce conflict.

Grammar Engineering

Though LALR grammars are very general and inclusive, sometimes a reasonable set of productions is rejected due to shift/reduce or reduce/reduce conflicts.

In such cases, the grammar may need to be "engineered" to allow the parser to operate.

A good example of this is the definition of **MemberDecls** in CSX. A straightforward definition is

MemberDecls $\rightarrow$ **FieldDecls MethodDecls**
FieldDecls $\rightarrow$ **FieldDecl FieldDecls**
FieldDecls $\rightarrow \lambda$
MethodDecls $\rightarrow$ **MethodDecl MethodDecls**
MethodDecls $\rightarrow \lambda$
FieldDecl $\rightarrow$ **int id ;**
MethodDecl $\rightarrow$ **int id ( ) ; Body**

When we predict **MemberDecls** we get:

MemberDecls $\rightarrow$ • **FieldDecls MethodDecls**
FieldDecls $\rightarrow$ • **FieldDecl FieldDecls**
FieldDecls $\rightarrow$ λ•
FieldDecl $\rightarrow$ • **int id ;**

Now **int** follows **FieldDecls** since

MethodDecls $\Rightarrow^{+}$ **int ...**

Thus an unresolvable shift/reduce conflict exists.

The problem is that **int** is derivable from both **FieldDecls** and **MethodDecls**, so when we see an **int**, we can't tell which way to parse it (and **FieldDecls** $\rightarrow$ λ requires we make an immediate decision!).

If we rewrite the grammar so that we can delay deciding from where the int was generated, a valid LALR parser can be built:

MemberDecls → FieldDecl MemberDecls
MemberDecls → MethodDecls
MethodDecls → MethodDecl MethodDecls
MethodDecls → λ
FieldDecl → int id ;
MethodDecl → int id ( ) ; Body

When **MemberDecls** is predicted we have

MemberDecls → • FieldDecl MemberDecls
MemberDecls → • MethodDecls
MethodDecls → •MethodDecl MethodDecls
MethodDecls → λ •
FieldDecl → • int id ;
MethodDecl → • int id ( ) ; Body

Now **Follow(MethodDecls)** = **Follow(MemberDecls)** = "}", so we have no shift/reduce conflict. After **int id** is matched, the next token (a ";" or a "(") will tell us whether a **FieldDecl** or a **MethodDecl** is being matched.

Properties of LL and LALR Parsers

- Each prediction or reduce action is *guaranteed* correct. Hence the entire parse (built from LL predictions or LALR reductions) must be correct.

  This follows from the fact that LL parsers allow only one valid prediction per step. Similarly, an LALR parser never skips a reduction if it is consistent with the current token (and *all* possible reductions are tracked).

- LL and LALR parsers detect an syntax error as soon as the first invalid token is seen.

  Neither parser can match an invalid program prefix. If a token is matched it *must be* part of a valid program prefix. In fact, the prediction made or the stacked configuration sets *show* a possible derivation of the token accepted so far.

- All LL and LALR grammars are unambiguous.

  LL predictions are always unique and LALR shift/reduce or reduce/reduce conflicts are disallowed. Hence only one valid derivation of any token sequence is possible.

- All LL and LALR parsers require only linear time and space (in terms of the number of tokens parsed).

  The parsers do only fixed work per node of the concrete parse tree, and the size of this tree is linear in terms of the number of leaves in it (even with λ-productions included!).

Reading Assignment

Read Chapter 8 of **Crafting a Compiler**.

Symbol Tables in CSX

CSX is designed to make symbol tables easy to create and use.

There are three places where a new scope is opened:

- In the class that represents the program text. The scope is opened as soon as we begin processing the **`classNode`** (that roots the entire program). The scope stays open until the entire class (the whole program) is processed.
- When a **`methodDeclNode`** is processed. The name of the method is entered in the top-level (global) symbol table. Declarations of parameters and locals are placed in the method's symbol table. A method's symbol table is closed after all the statements in its body are type checked.

- When a **`blockNode`** is processed. Locals are placed in the block's symbol table. A block's symbol table is closed after all the statements in its body are type checked.

CSX Allows no Forward References

This means we can do type-checking in *one pass* over the AST. As declarations are processed, their identifiers are added to the current (innermost) symbol table. When a use of an identifier occurs, we do an ordinary block-structured lookup, always using the innermost declaration found. Hence in

```
int i = j;
int j = i;
```

the first declaration initializes `i` to the nearest non-local definition of `j`.

The second declaration initializes `j` to the current (local) definition of `i`.

Forward References Require Two Passes

If forward references are allowed, we can process declarations in two passes.

First we walk the AST to establish symbol tables entries for all local declarations. No uses (lookups) are handled in this passes.

On a second complete pass, all uses are processed, using the symbol table entries built on the first pass.

Forward references make type checking a bit trickier, as we may reference a declaration not yet fully processed.

In Java, forward references to fields within a class are allowed.

Thus in

```
class Duh {
 int i = j;
 int j = i;
}
```

a Java compiler must recognize that the initialization of **`i`** is to the **`j`** field and that the **`j`** declaration is incomplete (Java forbids uninitialized fields or variables).

Forward references do allow methods to be mutually recursive. That is, we can let method **`a`** call **`b`**, while **`b`** calls **`a`**.

In CSX this is impossible!

(Why?)

Incomplete Declarations

Some languages, like C++, allow incomplete declarations.

First, part of a declaration (usually the header of a procedure or method) is presented.

Later, the declaration is completed.

For example (in C++):

```
class C {
  int i;
 public:
  int f();
};
int C::f(){return i+1;}
```

Incomplete declarations solve potential forward reference problems, as you can declare method headers first, and bodies that use the headers later.

Headers support abstraction and separate compilation too.

In C and C++, it is common to use a **`#include`** statement to add the headers (but not bodies) of external or library routines you wish to use.

C++ also allows you to declare a class by giving its fields and method headers first, with the bodies of the methods declared later. This is good for users of the class, who don't always want to see implementation details.

Classes, Structs and Records

The fields and methods declared within a class, struct or record are stored within a individual symbol table allocated for its declarations.

Member names must be unique within the class, record or struct, but may clash with other visible declarations. This is allowed because member names are qualified by the object they occur in.

Hence the reference **`x.a`** means look up **`x`**, using normal scoping rules. Object **`x`** should have a type that includes local fields. The type of **`x`** will include a pointer to the symbol table containing the field declarations. Field **`a`** is looked up in that symbol table.

Chains of field references are no problem.

For example, in Java

```
System.out.println
```

is commonly used.

`System` is looked up and found to be a class in one of the standard Java packages (**`java.lang`**). Class **`System`** has a static member **`out`** (of type **`PrintStream`**) and **`PrintStream`** has a member **`println`**.

Internal and External Field Access

Within a class, members may be accessed without qualification. Thus in

```
class C {
  static int i;
  void subr() {
     int j = i;
  }
}
```

field `i` is accessed like an ordinary non-local variable.

To implement this, we can treat member declarations like an ordinary scope in a block-structured symbol table.

When the class definition ends, its symbol table is popped and members are referenced through the symbol table entry for the class name.

This means a simple reference to **`i`** will no longer work, but **`C.i`** will be valid.

In languages like C++ that allow incomplete declarations, symbol table references need extra care. In

```
class C {
  int i;
 public:
  int f();
};
int C::f(){return i+1;}
```

when the definition of **`f()`** is completed, we must restore **`C`**'s field definitions as a containing scope so that the reference to **`i`** in **`i+1`** is properly compiled.

Public and Private Access

C++ and Java (and most other object-oriented languages) allow members of a class to be marked **`public`** or **`private`**.

Within a class the distinction is ignored; all members may be accessed.

Outside of the class, when a qualified access like **`C.i`** is required, only **`public`** members can be accessed.

This means lookup of class members is a two-step process. First the member name is looked up in the symbol table of the class. Then, the **`public`**/**`private`** qualifier is checked. Access to **`private`** members from outside the class generates an error message.

C++ and Java also provide a **`protected`** qualifier that allows access from subclasses of the class containing the member definition.

When a subclass is defined, it "inherits" the member definitions of its ancestor classes. Local definitions may hide inherited definitions. Moreover, inherited member definitions must be **`public`** or **`protected`**; **`private`** definitions may not be directly accessed (though they are still inherited and may be indirectly accessed through other inherited definitions).

Java also allows "blank" access qualifiers which allow **`public`** access by all classes within a package (a collection of classes).

Packages and Imports

Java allows packages which group class and interface definitions into named units.

A package requires a symbol table to access members. Thus a reference

`java.util.Vector`

locates the package **`java.util`** (typically using a **`CLASSPATH`**) and looks up **`Vector`** within it.

Java supports **`import`** statements that modify symbol table lookup rules.

A single class import, like

`import java.util.Vector;`

brings the name **`Vector`** into the current symbol table (unless a

definition of **`Vector`** is already present).

An “import on demand” like

```
import java.util.*;
```

will lookup identifiers in the named packages after explicit user declarations have been checked.

Classfiles and Object Files

Class files (".class" files, produced by Java compilers) and object files (".o" files, produced by C and C++ compilers) contain internal symbol tables.

When a field or method of a Java class is accessed, the JVM uses the classfile's internal symbol table to access the symbol's value and verify that type rules are respected.

When a C or C++ object file is linked, the object file's internal symbol table is used to determine what external names are referenced, and what internally defined names will be exported.

C, C++ and Java all allow users to request that a more complete symbol table be generated for debugging purposes. This makes internal names (like local variable) visible so that a debugger can display source level information while debugging.

Overloading

A number of programming languages, including Java and C++, allow method and subprogram names to be *overloaded*.

This means several methods or subprograms may share the same name, as long as they differ in the number or types of parameters they accept. For example,

```
class C {
  int x;
  public static int sum(int v1,
                        int v2) {
     return v1 + v2;
  }
  public int sum(int v3) {
     return x + v3;
  }
}
```

For overloaded identifiers the symbol table must return a *list* of valid definitions of the identifier. Semantic analysis (type checking) then decides which definition to use.

In the above example, while checking

```
(new C()).sum(10);
```

both definitions of **`sum`** are returned when it is looked up. Since one argument is provided, the definition that uses one parameter is selected and checked.

A few languages (like Ada) allow overloading to be disambiguated on the basis of a method's result type. Algorithms that do this analysis are known, but are fairly complex.

Overloaded Operators

A few languages, like C++, allow operators to be overloaded.

This means users may add new definitions for existing operators, though they may not create new operators or alter existing precedence and associativity rules.

(Such changes would force changes to the scanner or parser.)

For example,

```
class complex{
  float re, im;
  complex operator+(complex d){
    complex ans;
    ans.re = d.re+re;
    ans.im = d.im+im;
    return ans;
} }
complex c,d; c=c+d;
```

During type checking of an operator, all visible definitions of the operator (including predefined definitions) are gathered and examined.

Only one definition should successfully pass type checks.

Thus in the above example, there may be many definitions of **+**, but only one is defined to take **`complex`** operands.

Contextual Resolution

Overloading allows multiple definitions of the same kind of object (method, procedure or operator) to co-exist.

Programming languages also sometimes allow reuse of the same name in defining different kinds of objects. Resolution is by context of use.

For example, in Java, a class name may be used for both the class and its constructor. Hence we see

```
C cvar = new C(10);
```

In Pascal, the name of a function is also used for its return value.

Java allows rather extensive reuse of an identifier, with the same identifier potentially denoting a class (type), a class constructor, a

package name, a method and a field.

For example,

```
class C {
  double v;
 C(double f) {v=f;}
}
class D {
  int C;
  double C() {return 1.0;}
 C cval = new C(C+C());
}
```

At type-checking time we examine all potential definitions and use that definition that is consistent with the context of use. Hence new `C()` must be a constructor, `+C()` must be a function call, etc.

Allowing multiple definitions to co-exist certainly makes type checking more complicated than in other languages.

Whether such reuse benefits programmers is unclear; it certainly violates Java's "keep it simple" philosophy.

Type and Kind Information in CSX

In CSX symbol table entries and in AST nodes for expressions, it is useful to store *type* and *kind* information.

This information is created and tested during type checking. In fact, most of type checking involves deciding whether the type and kind values for the current construct and its components are valid.

Possible values for **`type`** include:

- **`Integer (int)`**
- **`Boolean (bool)`**
- **`Character (char)`**
- **`String`**

- **`Void`**
  `Void` is used to represent objects that have no declared type (e.g., a label or procedure).
- **`Error`**
  `Error` is used to represent objects that should have a type, but don't (because of type errors). **`Error`** types suppress further type checking, preventing cascaded error messages.
- **`Unknown`**
  `Unknown` is used as an initial value, before the type of an object is determined.

Possible values for **`kind`** include:

- **`Var`** (a local variable or field that may be assigned to)
- **`Value`** (a value that may be read but not changed)
- **`Array`**
- **`ScalarParm`** (a by-value scalar parameter)
- **`ArrayParm`** (a by-reference array parameter)
- **`Method`** (a procedure or function)
- **`Label`** (on a **`while`** loop)

Most combinations of **`type`** and **`kind`** represent something in CSX.

Hence **`type==Boolean`** and **`kind==Value`** is a **`bool`** constant or expression.

`type==Void` and **`kind==Method`** is a procedure (a method that returns no value).

Type checking procedure and function declarations and calls requires some care.

When a method is declared, you should build a linked list of (**`type`**,**`kind`**) pairs, one for each declared parameter.

When a call is type checked you should build a second linked list of (**`type`**,**`kind`**) pairs for the actual parameters of the call.

You compare the lengths of the list of formal and actual parameters to check that the correct number of parameters has been passed.

You then compare corresponding formal and actual parameter pairs to check if each individual actual parameter correctly matches its corresponding formal parameter.

For example, given

```
p(int a, bool b[]){ …
```

and the call

```
p(1,false);
```

you create the parameter list

(**Integer**, **ScalarParm**),
(**Boolean**, **ArrayParm**)

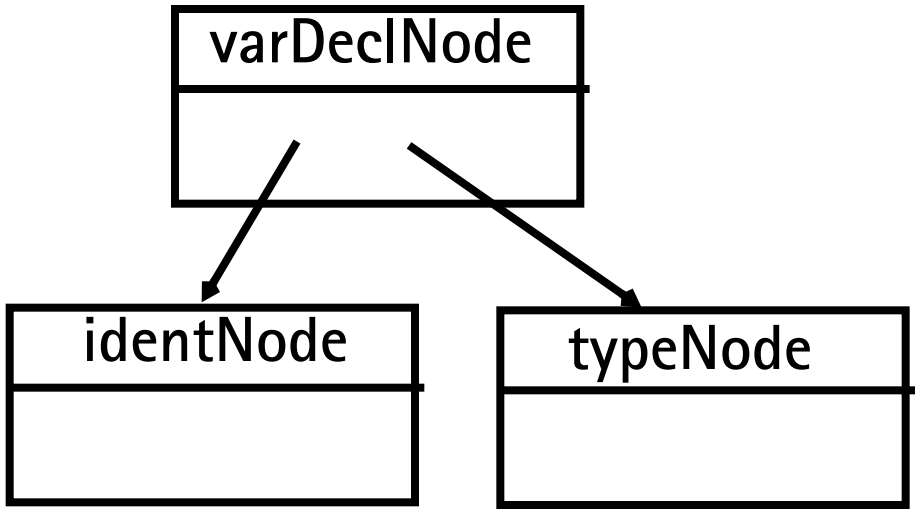
for **p**'s declaration and the parameter list

(**Integer**,**Value**),(**Boolean**, **Value**)

for **p**'s call.

Since a **`Value`** can't match an **`ArrayParm`**, you know that the second parameter in **`p`**'s call is incorrect.

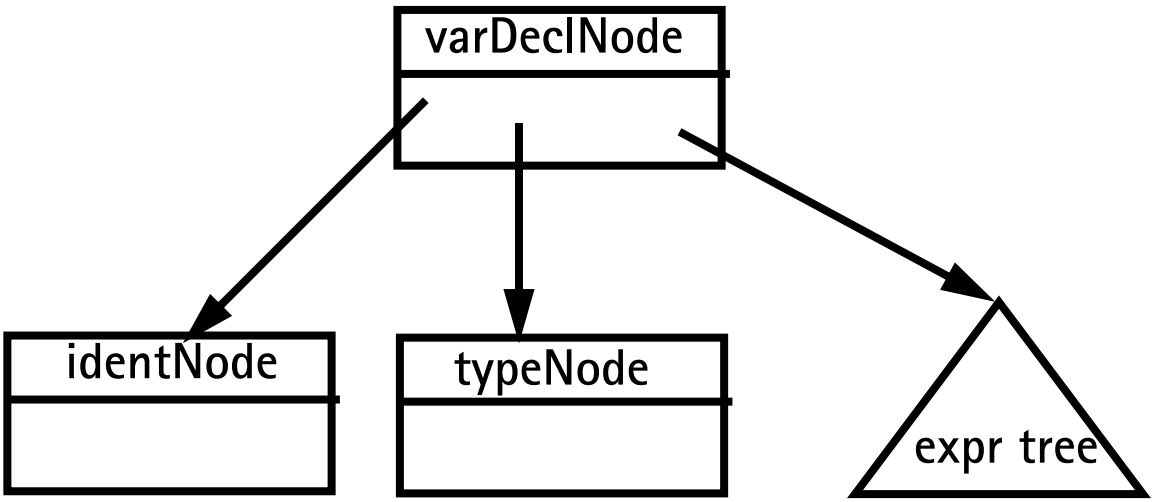
Type Checking Simple Variable Declarations



Type checking steps:

1. Check that **`identNode.idname`** is not already in the symbol table.
2. Enter **`identNode.idname`** into symbol table with
   `type = typeNode.type` and
   `kind = Variable`.

Type Checking Initialized Variable Declarations

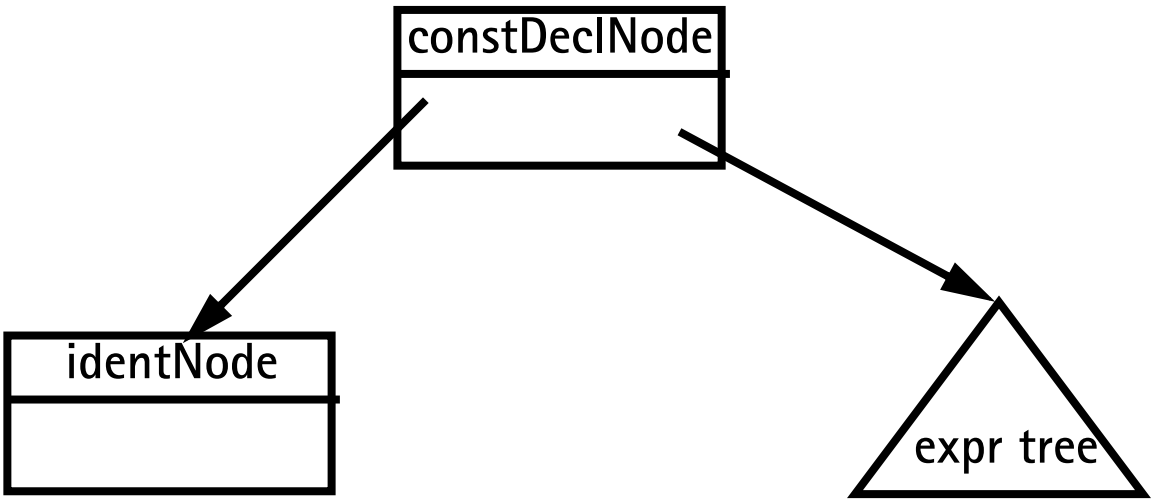


Type checking steps:

1. Check that **`identNode.idname`** is not already in the symbol table.
2. Type check initial value expression.
3. Check that the initial value's type is **`typeNode.type`**

4. Check that the initial value's kind is scalar (**`Variable`**, **`Value`** or **`ScalarParm`**).
5. Enter **`identNode.idname`** into symbol table with
   `type = typeNode.type` and
   `kind = Variable`.

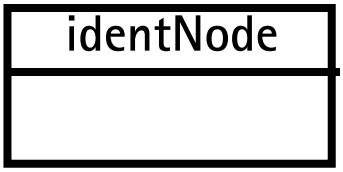
Type Checking Const Decls



Type checking steps:

1. Check that **`identNode.idname`** is not already in the symbol table.
2. Type check the const value expr.
3. Check that the const value's kind is scalar (**`Variable`**, **`Value`** or **`ScalarParm`**).
4. Enter **`identNode.idname`** into symbol table with **`type`** = **`constValue.type`** and **`kind`** = **`Value`**.

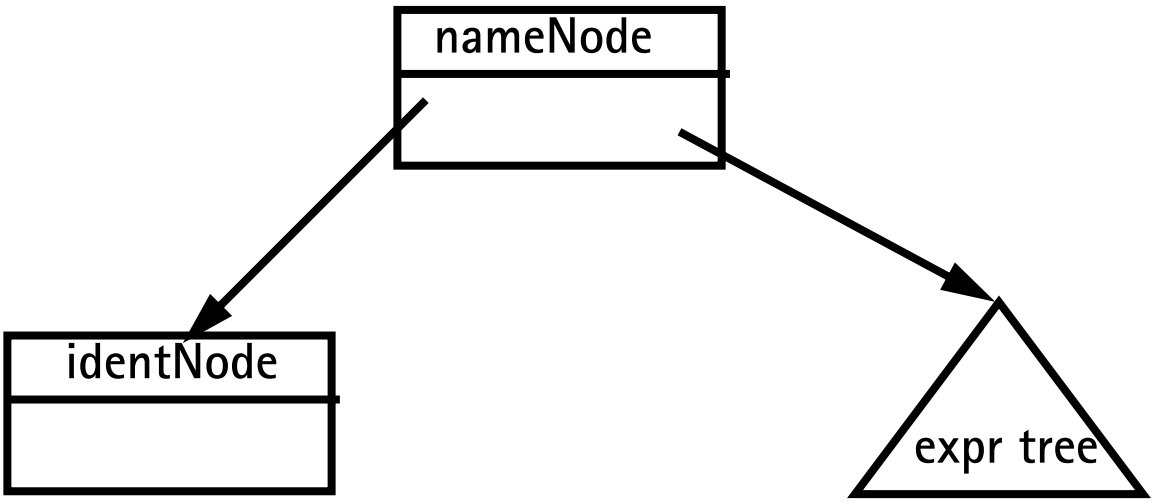
Type Checking IdentNodes



Type checking steps:

1. Lookup **`identNode.idname`** in the symbol table; error if absent.
2. Copy symbol table entry's **`type`** and **`kind`** information into the **`identNode`**.
3. Store a link to the symbol table entry in the **`identNode`** (in case we later need to access symbol table information).

Type Checking NameNodes

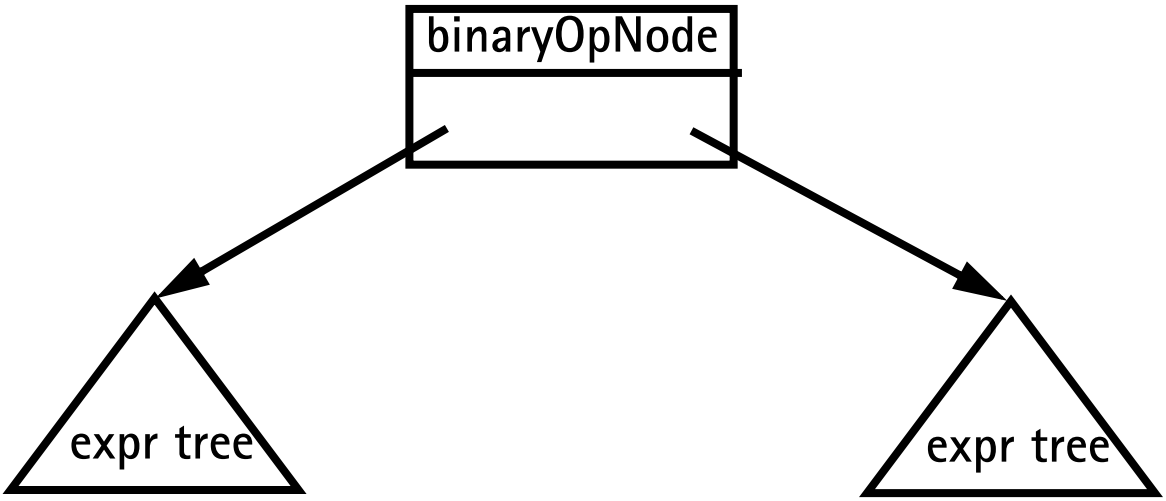


Type checking steps:

1. Type check the **`identNode`**.
2. If the **`subscriptVal`** is a null node, copy the **`identNode's type`** and **`kind`** values into the **`nameNode`** and return.
3. Type check the **`subscriptVal`**.
4. Check that **`identNode's kind`** is an array.

5. Check that **`subscriptVal's kind`** is scalar and **`type`** is integer or character.
6. Set the **`nameNode's type`** to the **`identNode's type`** and the **`nameNode's kind`** to **`Variable`**.

Type Checking Binary Operators

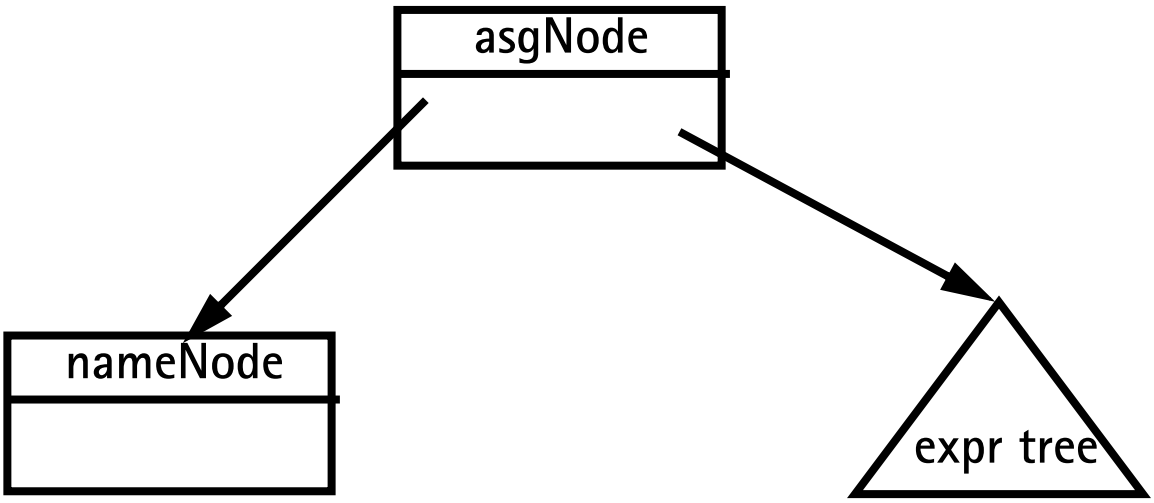


Type checking steps:

1. Type check left and right operands.
2. Check that left and right operands are both scalars.
3. **`binaryOpNode.kind` = `Value`.**

4. If **`binaryOpNode.operator`** is a plus, minus, star or slash:
   (a) Check that left and right operands have an arithmetic type (integer or character).
   (b) **`binaryOpNode.type`** = **`Integer`**
5. If **`binaryOpNode.operator`** is an and or is an or:
   (a) Check that left and right operands have a boolean type.
   (b) **`binaryOpNode.type`** = **`Boolean`**.
6. If **`binaryOpNode.operator`** is a relational operator:
   (a) Check that both left and right operands have an arithmetic type or both have a boolean type.
   (b) **`binaryOpNode.type`** = **`Boolean`**.

Type Checking Assignments

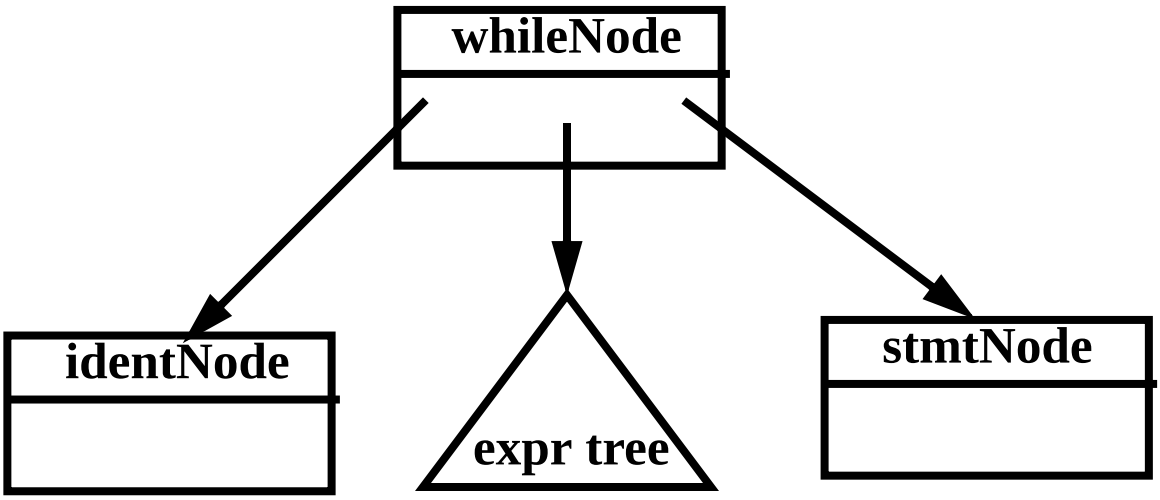


Type checking steps:

1. Type check the **`nameNode`**.
2. Type check the expression tree.
3. Check that the **`nameNode's kind`** is assignable (**`Variable`**, **`Array`**, **`ScalarParm`**, or **`ArrayParm`**).
4. If the **`nameNode's kind`** is scalar then check the expression tree's **`kind`** is also scalar and that both have the same type. Then return.

5. If the **`nameNode's`** and the expression tree's **`kinds`** are both arrays and both have the same **`type`**, check that the arrays have the same length. (Lengths of array parms are checked at run-time). Then return.
6. If the **`nameNode's kind`** is array and its **`type`** is character and the expression tree's kind is string, check that both have the same length. (Lengths of array parms are checked at run-time). Then return.
7. Otherwise, the expression may not be assigned to the **`nameNode`**.

Type Checking While Loops

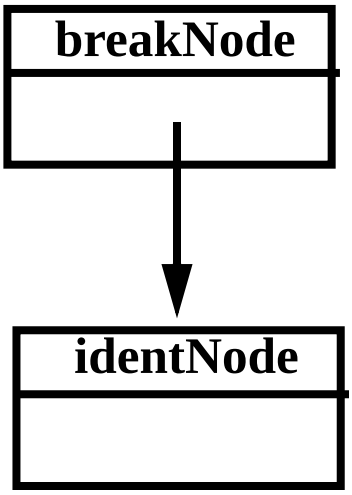


Type checking steps:

1. Type check the **`condition`** (an expr tree).
2. Check that the **`condition's type`** is **`Boolean`** and **`kind`** is scalar.
3. If the **`label`** is a null node then type check the **`stmtNode`** (the loop body) and return.

4. If there is a **`label`** (an **`identNode`**) then:
   (a) Check that the **`label`** is not already present in the symbol table.
   (b) If it isn't, enter **`label`** in the symbol table with **`kind=VisibleLabel`** and **`type= void`**.
   (c) Type check the **`stmtNode`** (the loop body).
   (d) Change the **`label's kind`** (in the symbol table) to **`HiddenLabel`**.

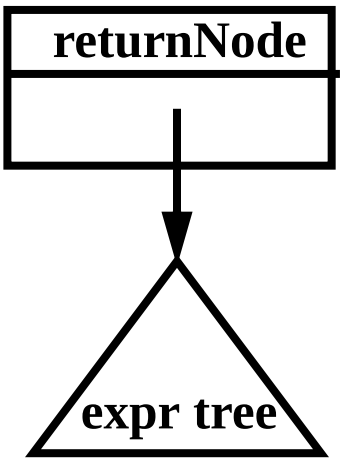
Type Checking Breaks and Continues



Type checking steps:

1. Check that the **`identNode`** is declared in the symbol table.
2. Check that **`identNode's kind`** is **`VisibleLabel`**. If **`identNode's kind`** is **`HiddenLabel`** issue a special error message.

Type Checking Returns

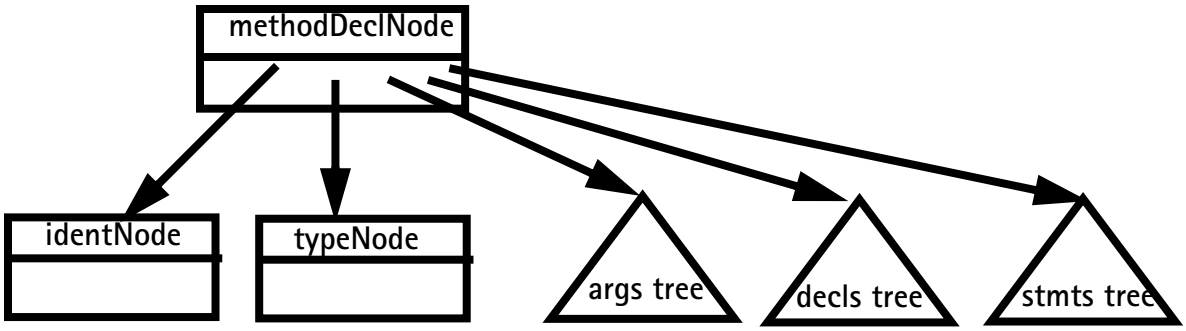


It is useful to arrange that a static field named ***currentMethod*** will always point to the **methodDeclNode** of the method we are currently checking.

Type checking steps:

1. If **returnVal** is a null node, check that **currentMethod.returnType** is **Void**.
2. If returnVal (an expr) is not null then check that returnVal's kind is scalar and returnVal's type is currentMethod.returnType.

Type Checking Method Declarations

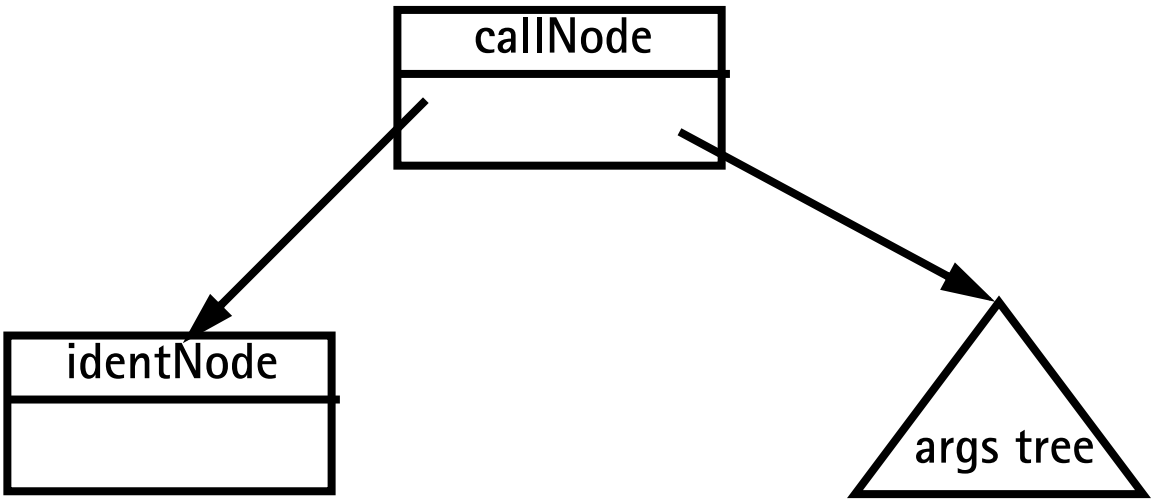


Type checking steps:

1. Create a new symbol table entry **`m`**, with **`type`** = **`typeNode.type`** and **`kind`** = **`Method`**.
2. Check that **`identNode.idname`** is not already in the symbol table; if it isn't, enter **`m`** using **`identNode.idname`**.
3. Create a new scope in the symbol table.
4. Set **`currentMethod`** = this **`methodDeclNode`**.

5. Type check the **`args`** subtree.
6. Build a list of the symbol table nodes corresponding to the **`args`** subtree; store it in **`m`**.
7. Type check the **`decls`** subtree.
8. Type check the **`stmts`** subtree.
9. Close the current scope at the top of the symbol table.

Type Checking Method Calls



We consider calls of procedures in a statement. Calls of functions in an expression are very similar.

Type checking steps:

1. Check that **`identNode.idname`** is declared in the symbol table. Its type should be **`Void`** and kind should be **`Method`**.
2. Type check the **`args`** subtree.
3. Build a list of the expression nodes found in the args subtree.

4. Get the list of parameter symbols declared for the method (stored in the method's symbol table entry).
5. Check that the arguments list and the parameter symbols list both have the same length.
6. Compare each argument node with its corresponding parameter symbol:
   (a) Both should have the same type.
   (b) A **`Variable`**, **`Value`**, or **`ScalarParm`** kind in an argument node matches a **`ScalarParm`** parameter. An **`Array`** or **`ArrayParm`** kind in an argument node matches an **`ArrayParm`** parameter.

Reading Assignment

Read Chapters 9 and 12 of
Crafting a Compiler.

Virtual Memory & Run-Time Memory Organization

The compiler decides how data and instructions are placed in memory.

It uses an *address space* provided by the hardware and operating system.

This address space is usually *virtual*—the hardware and operating system map instruction-level addresses to "actual" memory addresses.

Virtual memory allows:

- Multiple processes to run in private, protected address spaces.
- Paging can be used to extend address ranges beyond actual memory limits.

Run-Time Data Structures

Static Structures

For static structures, a fixed address is used throughout execution.

This is the oldest and simplest memory organization.

In current compilers, it is used for:

- Program code (often read-only & sharable).
- Data literals (often read-only & sharable).
- Global variables.
- Static variables.

Stack Allocation

Modern programming languages allow recursion, which requires *dynamic allocation*.

Each recursive call allocates a *new copy* of a routine's local variables.

The number of local data allocations required during program execution is not known at compile-time.

To implement recursion, all the data space required for a method is treated as a distinct data area that is called a *frame* or *activation record*.

Local data, within a frame, is accessible only while a subprogram is active.

In mainstream languages like C, C++ and Java, subprograms must return in a stack-like manner—the most recently called subprogram will be the first to return.

A frame is pushed onto a *run-time stack* when a method is called (activated).

When it returns, the frame is popped from the stack, freeing the routine's local data.

As an example, consider the following C subprogram:

```
p(int a) {
  double b;
  double c[10];
  b = c[a] * 2.51;
}
```

Procedure **p** requires space for the parameter **a** as well as the local variables **b** and **c**.

It also needs space for control information, such as the return address.

The compiler records the space requirements of a method.

The *offset* of each data item relative to the start of the frame is stored in the symbol table.

The total amount of space needed, and thus the size of the frame, is also recorded.

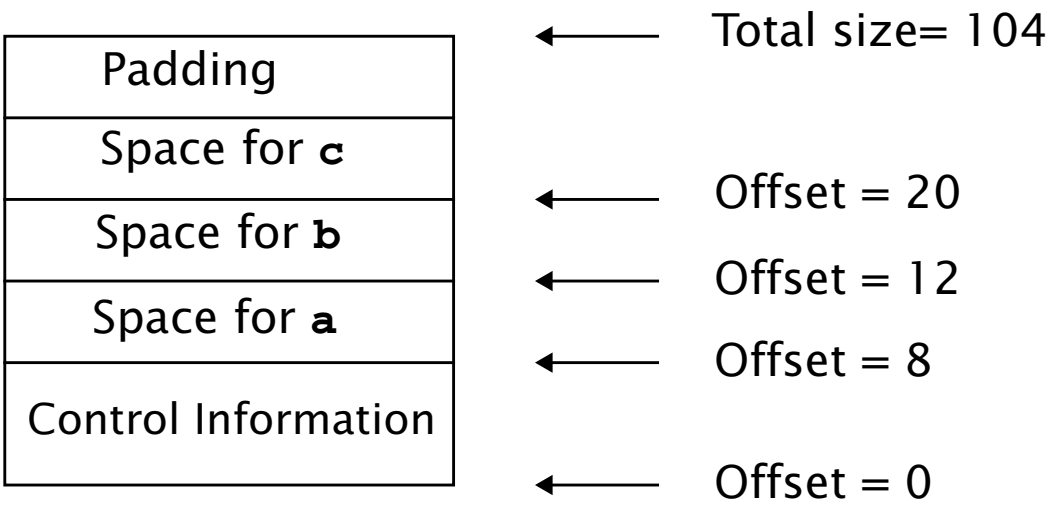
Assume **p**'s control information requires 8 bytes (this size is usually the same for all methods).

Assume parameter a requires 4 bytes, local variable **b** requires 8 bytes, and local array **c** requires 80 bytes.

Many machines require that word and doubleword data be *aligned*, so it is common to pad a frame so that its size is a multiple of 4 or 8 bytes.

This guarantees that at all times the top of the stack is properly aligned.

Here is `p`'s frame:



Within `p`, each local data object is addressed by its offset relative to the start of the frame.

This offset is a fixed constant, determined at compile-time.

We normally store the start of the frame in a register, so each piece of data can be addressed as a (Register, Offset) pair, which is a standard addressing mode in almost all computer architectures.

For example, if register R points to the beginning of `p`'s frame, variable `b` can be addressed as (R,12), with 12 actually being added to the contents of R at run-time, as memory addresses are evaluated.

Normally, the literal `2.51` of procedure `p` is *not* stored in `p`'s frame because the values of local data that are stored in a frame disappear with it at the end of a call.

It is easier and more efficient to allocate literals in a *static area*, often called a *literal pool* or *constant pool*. Java uses a constant pool to store literals, type, method and interface information as well as class and field names.

Accessing Frames at Run-Time

During execution there can be many frames on the stack. When a procedure `A` calls a procedure `B`, a frame for `B`'s local variables is pushed on the stack, covering `A`'s frame. `A`'s frame can't be popped off because `A` will resume execution after `B` returns.

For recursive routines there can be hundreds or even thousands of frames on the stack. All frames but the topmost represent suspended subroutines, waiting for a call to return.

The topmost frame is *active*; it is important to access it directly.

The active frame is at the top of the stack, so the *stack top register* could be used to access it.

The run-time stack may also be used to hold data other than frames.

It is unwise to require that the currently active frame always be at *exactly* the top of the stack.

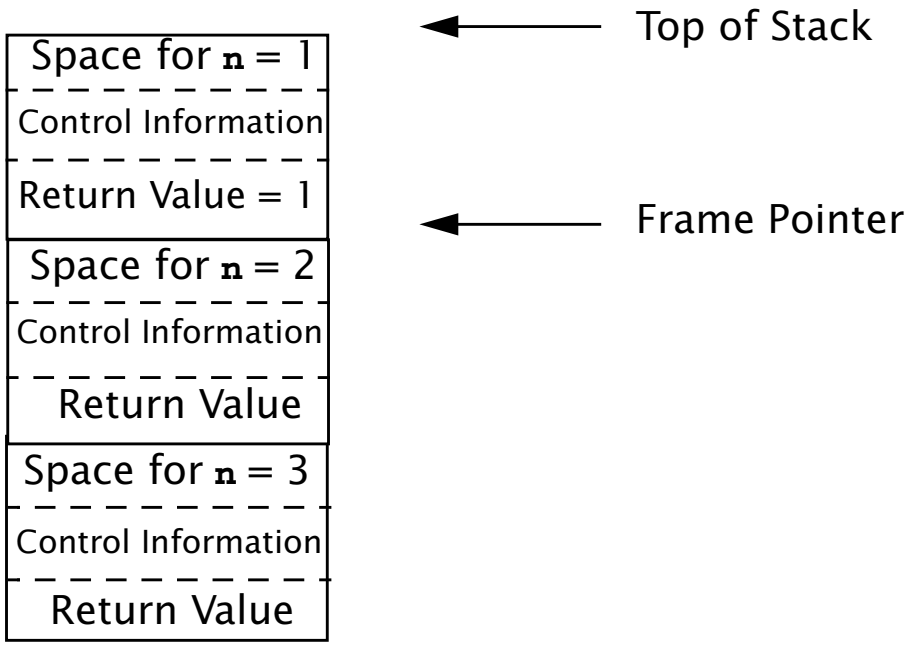
Instead a distinct register, often called the *frame pointer*, is used to access the current frame.

This allows local variables to be accessed directly as offset + frame pointer, using the indexed addressing mode found on all modern machines.

Consider the following recursive function that computes factorials.

```
int fact(int n) {
  if (n > 1)
    return n * fact(n-1);
  else
    return 1;
}
```

The run-time stack corresponding to the call **`fact(3)`** (when the call of **`fact(1)`** is about to return) is:



We place a slot for the function's return value at the very beginning of the frame.

Upon return, the return value is conveniently placed on the stack, just beyond the end of the caller's frame. Often compilers return scalar values in specially

designated registers, eliminating unnecessary loads and stores. For values too large to fit in a register (arrays or objects), the stack is used.

When a method returns, its frame is popped from the stack and the frame pointer is reset to point to the caller's frame.

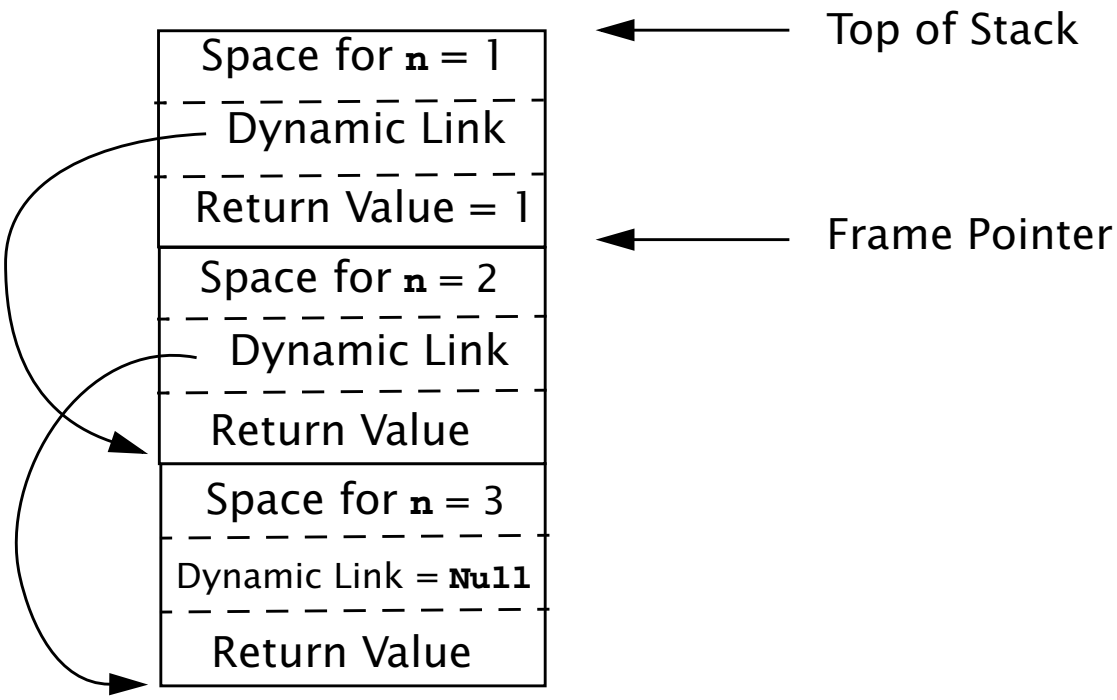
In simple cases this is done by adjusting the frame pointer by the size of the current frame.

Dynamic Links

Because the stack may contain more than just frames (e.g., function return values or registers saved across calls), it is common to save the caller's frame pointer as part of the callee's control information.

Each frame points to its caller's frame on the stack. This pointer is called a *dynamic link* because it links a frame to its dynamic (run-time) predecessor.

The run-time stack corresponding to a call of **`fact(3)`**, with dynamic links included, is:



Classes and Objects

C, C++ and Java do not allow procedures or methods to nest.

A procedure may not be declared within another procedure.

This simplifies run-time data access—all variables are either global or local.

Global variables are statically allocated. Local variables are part of a single frame, accessed through the frame pointer.

Java and C++ allow classes to have *member functions* that have direct access to instance variables.

Consider:

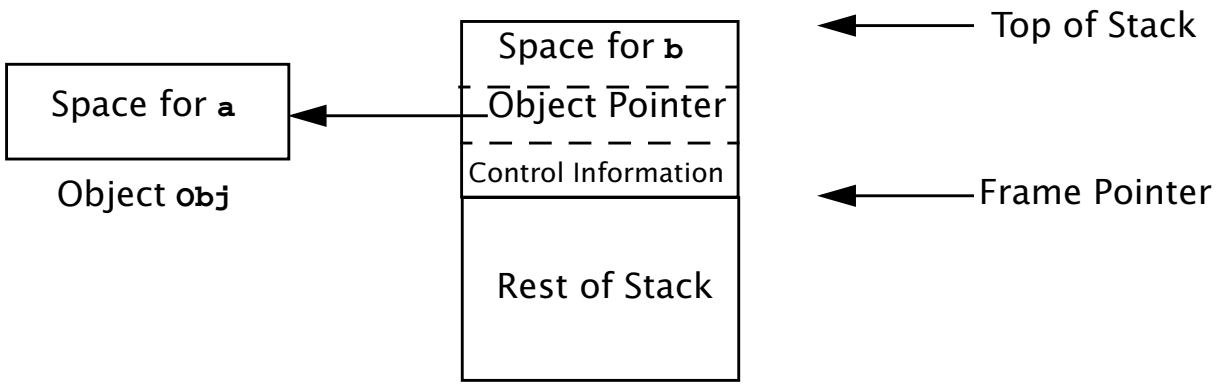
```
class K {
  int a;
  int sum(){
  int b;
  return a+b;
} }
```

Each object that is an instance of class **K** contains a member function **sum**. Only one translation of **sum** is created; it is shared by all instances of **K**.

When **sum** executes it needs *two* pointers to access local and object-level data.

Local data, as usual, resides in a frame on the run-time stack.

Data values for a particular instance of **`K`** are accessed through an object pointer (called the **`this`** pointer in Java and C++). When **`obj.sum()`** is called, it is given an extra *implicit parameter* that a pointer to **`obj`**.



When **`a+b`** is computed, **`b`**, a local variable, is accessed directly through the frame pointer. **`a`**, a member of object **`obj`**, is accessed indirectly through the object pointer that is stored in the frame (as all parameters to a method are).

C++ and Java also allow inheritance via subclassing. A new class can extend an existing class, adding new fields and adding or redefining methods.

A subclass **`D`**, of class **`C`**, maybe be used in contexts expecting an object of class **`C`** (e.g., in method calls).

This is supported rather easily—objects of class **`D`** always contain a class **`C`** object within them.

If **`C`** has a field **`F`** within it, so does **`D`**. The fields **`D`** declares are merely *appended* at the end of the allocations for **`C`**.

As a result, access to fields of **`C`** within a class **`D`** object works perfectly.

Handling Multiple Scopes

Many languages allow procedure declarations to nest. Java now allows classes to nest.

Procedure nesting can be very useful, allowing a subroutine to directly access another routine's locals and parameters.

Run-time data structures are complicated because multiple frames, corresponding to nested procedure declarations, may need to be accessed.

To see the difficulties, assume that routines *can* nest in Java or C:

```
int p(int a){
  int q(int b){
   if (b < 0)
     return q(-b);
  else
   return a+b;
  }
  return q(-10);
}
```

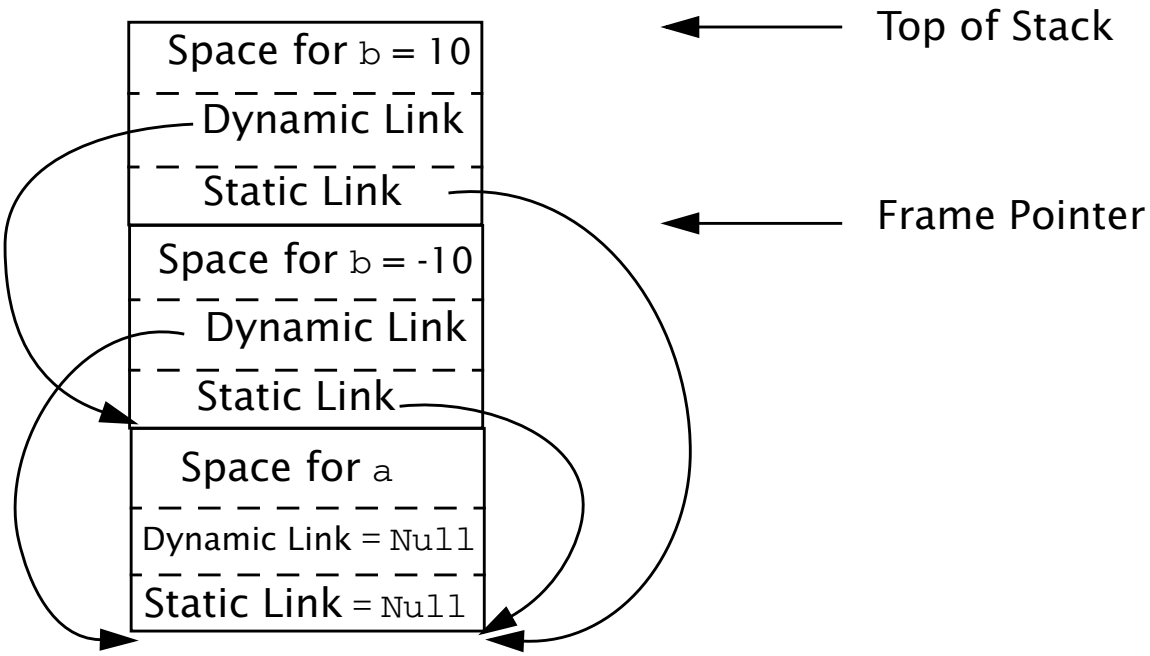
When **q** executes, it may access not only its own frame, but also that of **p**, in which it is nested.

If the depth of nesting is unlimited, so is the number of frames that must be accessible. In practice, the level of nesting actually seen is modest—usually no greater than two or three.

Static Links

Two approaches are commonly used to support access to multiple frames. One approach generalizes the idea of dynamic links introduced earlier. Along with a dynamic link, we'll also include a *static link* in the frame's control information area. The static link points to the frame of the procedure that statically encloses the current procedure. If a procedure is not nested within any other procedure, its static link is **`null`**.

The following illustrates static links:



As usual, dynamic links always point to the next frame down in the stack. Static links always point down, but they may skip past many frames. They always point to the most recent frame of the routine that statically encloses the current routine.

In our example, the static links of both of **`q`**'s frames point to **`p`**, since it is **`p`** that encloses **`q`**'s definition.

In evaluating the expression **`a+b`** that **`q`** returns, **`b`**, being local to **`q`**, is accessed directly through the frame pointer. Variable **`a`** is local to **`p`**, but also visible to **`q`** because **`q`** nests within **`p`**. **`a`** is accessed by extracting **`q`**'s static link, then using that address (plus the appropriate offset) to access **`a`**.

Displays

An alternative to using static links to access frames of enclosing routines is the use of a *display*.

A display generalizes our use of a frame pointer. Rather than maintaining a single register, we maintain a *set of registers* which comprise the display.

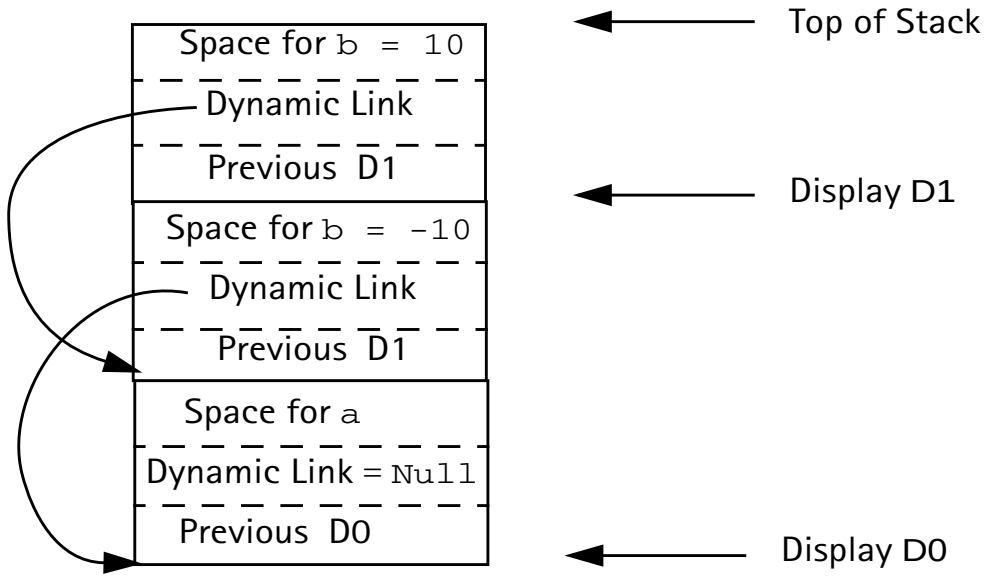
If procedure definitions nest *n* deep (this can be easily determined by examining a program's AST), we need *n+1* display registers.

Each procedure definition is tagged with a nesting level. Procedures not nested within any other routine are at level 0. Procedures nested within only one routine are at level 1, etc.

Frames for routines at level 0 are always accessed using display register D0. Those at level 1 are always accessed using register D1, etc.

Whenever a procedure **`r`** is executing, we have direct access to **`r`**'s frame plus the frames of all routines that enclose **`r`**. Each of these routines must be at a different nesting level, and hence will use a different display register.

The following illustrates the use of display registers:



Since `q` is at nesting level 1, its frame is pointed to by D1. All of **`q`**'s local variables, including **`b`**, are at a fixed offset relative to D1.

Since **`p`** is at nesting level 0, its frame and local variables are accessed via D0. Each frame's control information area contains a slot for the previous value of the frame's display register. A display register is saved when a call

begins and restored when the call ends. A dynamic link is still needed, because the previous display values doesn't always point to the caller's frame.

Not all compiler writers agree on whether static links or displays are better to use. Displays allow direct access to all frames, and thus make access to all visible variables very efficient. However, if nesting is deep, several valuable registers may need to be reserved. Static links are very flexible, allowing unlimited nesting of procedures. However, access to non-local procedure variables can be slowed by the need to extract and follow static links.

Heap Management

A very flexible storage allocation mechanism is *heap allocation.*

Any number of data objects can be allocated and freed in a memory pool, called a *heap*.

Heap allocation is enormously popular. Almost all non-trivial Java and C programs use **`new`** or **`malloc`**.

Heap Allocation

A request for heap space may be *explicit* or *implicit*.

An explicit request involves a call to a routine like **`new`** or **`malloc`**. An explicit pointer to the newly allocated space is returned.

Some languages allow the creation of data objects of unknown size. In Java, the + operator is overloaded to represent string catenation.

The expression **`Str1 + Str2`** creates a new string representing the catenation of strings **`Str1`** and **`Str2`**. There is no compile-time bound on the sizes of **`Str1`** and **`Str2`**, so heap space must be implicitly allocated to hold the newly created string.

Whether allocation is explicit or implicit, a *heap allocator* is needed. This routine takes a size parameter and examines unused heap space to find space that satisfies the request.

A *heap block* is returned. This block must be big enough to satisfy the space request, but it may well be bigger.

Heaps blocks contain a *header* field that contains the size of the block as well as bookkeeping information.

The complexity of heap allocation depends in large measure on how *deallocation* is done.

Initially, the heap is one large block of unallocated memory. Memory requests can be satisfied by simply modifying an "end of

heap" pointer, very much as a stack is pushed by modifying a stack pointer.

Things get more involved when previously allocated heap objects are deallocated and reused.

Deallocated objects are stored for future reuse on a *free space list*.

When a request for *n* bytes of heap space is received, the heap allocator must search the free space list for a block of sufficient size. There are many search strategies that might be used:

- **Best Fit**
  The free space list is searched for the free block that matches most closely the requested size. This minimizes wasted heap space, the search may be quite slow.

- **First Fit**
  The first free heap block of sufficient size is used. Unused space within the block is split off and linked as a smaller free space block. This approach is fast, but may "clutter" the beginning of the free space list with a number of blocks too small to satisfy most requests.
- **Next Fit**
  This is a variant of first fit in which succeeding searches of the free space list begin at the position where the last search ended. The idea is to "cycle through" the entire free space list rather than always revisiting free blocks at the head of the list.

- **Segregated Free Space Lists**
  There is no reason why we must have only *one* free space list. An alternative is to have several, indexed by the size of the free blocks they contain.

Deallocation Mechanisms

Allocating heap space is fairly easy. But how do we deallocate heap memory no longer in use?

Sometimes we may never need to deallocate! If heaps objects are allocated infrequently or are very long-lived, deallocation is unnecessary. We simply fill heap space with "in use" objects.

Virtual memory & paging may allow us to allocate a very large heap area.

On a 64-bit machine, if we allocate heap space at 1 MB/sec, it will take 500,000 *years* to span the entire address space! Fragmentation of a very large heap space commonly forces us to include some form of reuse of heap space.

User-controlled Deallocation

Deallocation can be manual or automatic. Manual deallocation involves explicit programmer-initiated calls to routines like **`free(p)`** or **`delete(p)`**.

The object is then added to a free-space list for subsequent reallocation.

It is the programmer's responsibility to free unneeded heap space by executing deallocation commands. The heap manager merely keeps track of freed space and makes it available for later reuse.

The really hard decision—when space should be freed—is shifted to the programmer, possibly leading to catastrophic *dangling pointer* errors.

Consider the following C program fragment

```
q = p = malloc(1000);
free(p);
/* code containing more malloc's */
q[100] = 1234;
```

After **p** is freed, **q** is a *dangling pointer*. **q** points to heap space that is no longer considered allocated.

Calls to **malloc** may reassign the space pointed to by **q**. Assignment through **q** is illegal, but this error is almost never detected.

Such an assignment may change data that is now part of another heap object, leading to very subtle errors. It may even change a header field or a free-space link, causing the heap allocator itself to fail!

Automatic Garbage Collection

The alternative to manual deallocation of heap space is *garbage collection*.

Compiler-generated code tracks pointer usage. When a heap object is no longer pointed to, it is *garbage*, and is *automatically* collected for subsequent reuse.

Many garbage collection techniques exist. Here are some of the most important approaches:

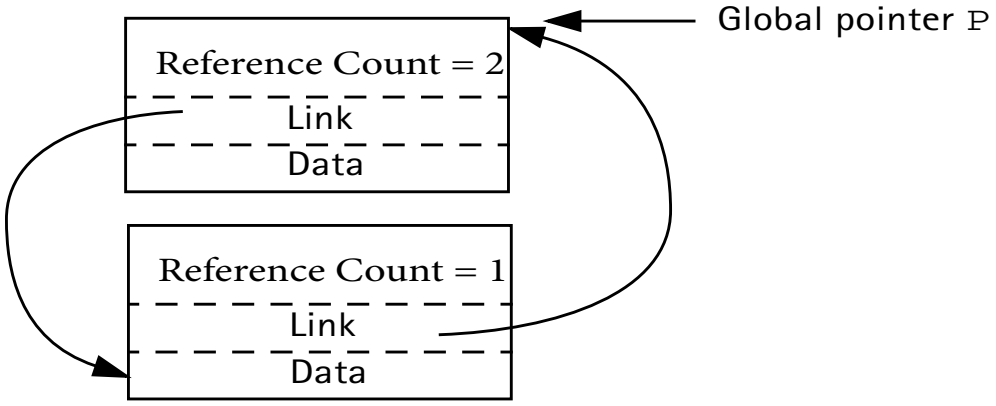
Reference Counting

This is one of the oldest and simplest garbage collection techniques.

A *reference count* field is added to each heap object. It counts how many references to the heap object exist. When an object's reference count reaches zero, it is garbage and may collected.

The reference count field is updated whenever a reference is created, copied, or destroyed. When a reference count reaches zero and an object is collected, all pointers in the collected object are also be followed and corresponding reference counts decremented.

As shown below, reference counting has difficulty with *circular structures.* If pointer **P** is



set to null, the object's reference count is reduced to 1. Both objects have a non-zero count, but neither is accessible through any external pointer. The two objects are garbage, but won't be recognized as such.

If circular structuresare common, then an auxiliary technique, like mark-sweep collection, is needed to collect garbage that reference counting misses.

Mark-Sweep Collection

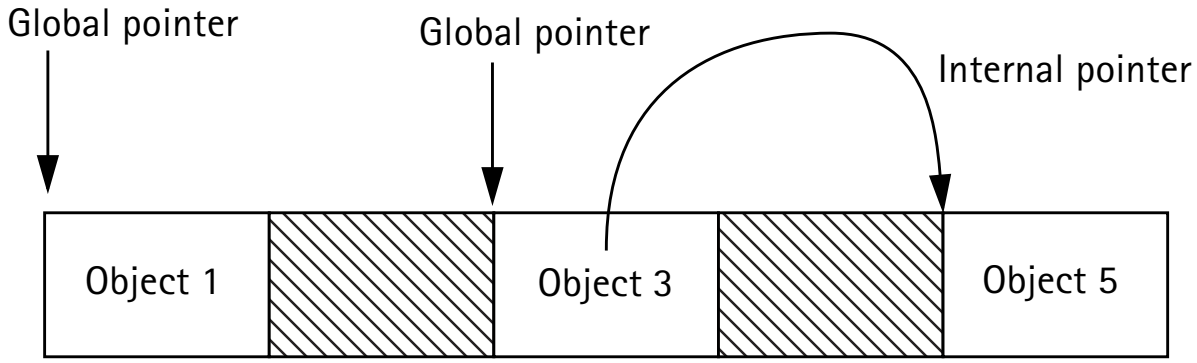
Many collectors, including mark & sweep, do nothing until heap space is nearly exhausted.

Then it executes a *marking phase* that identifies all live heap objects.

Starting with global pointers and pointers in stack frames, it marks reachable heap objects. Pointers in marked heap objects are also followed, until all live heap objects are marked.

After the marking phase, any object not marked is garbage that may be freed. We then *sweep* through the heap, collecting all unmarked objects. During the sweep phase we also clear all marks from heap objects found to be still in use.

Mark-sweep garbage collection is illustrated below.



Objects 1 and 3 are marked because they are pointed to by global pointers. Object 5 is marked because it is pointed to by object 3, which is marked. Shaded objects are not marked and will be added to the free-space list.

In any mark-sweep collector, it is vital that we mark *all* accessible heap objects. If we miss a pointer, we may fail to mark a live heap object and later incorrectly free it.

Finding all pointers is a bit tricky in languages like Java, C and C++, that have pointers mixed with other types within data structures, implicit pointers to temporaries, and so forth. Considerable information about data structures and frames must be available at run-time for this purpose. In cases where we can't be sure if a value is a pointer or not, we may need to do *conservative garbage collection*.

In mark-sweep garbage collection *all* heap objects must be swept. This is costly if most objects are dead. We'd prefer to examine *only* live objects.

Compaction

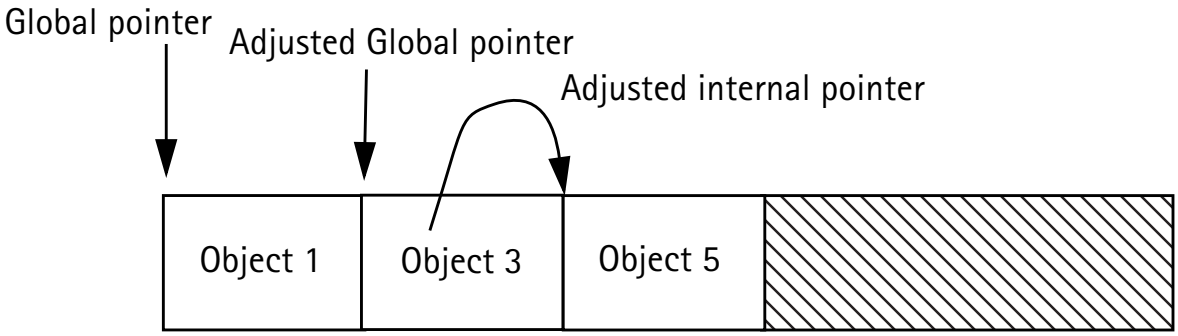
After the sweep phase, live heap objects are distributed throughout the heap space. This can lead to poor locality. If live objects span many memory pages, paging overhead may be increased. Cache locality may be degraded too.

We can add a *compaction phase* to mark-sweep garbage collection.

After live objects are identified, they are placed together at one end of the heap. This involves another tracing phase in which global, local and internal heap pointers are found and adjusted to reflect the object's new location.

Pointers are adjusted by the total size of all garbage objects

between the start of the heap and the current object. This is illustrated below:



Compaction merges together freed objects into one large block of free heap space. Fragments are no longer a problem.

Moreover, heap allocation is greatly simplified. Using an "end of heap" pointer, whenever a heap request is received, the end of heap pointer is adjusted, making heap allocation no more complex than stack allocation.

Because pointers are adjusted, compaction may not be suitable for languages like C and C++, in which it is difficult to unambiguously identify pointers.

Copying Collectors

Compaction provides many valuable benefits. Heap allocation is simple end efficient. There is no fragmentation problem, and because live objects are adjacent, paging and cache behavior is improved.

An entire family of garbage collection techniques, called *copying collectors* are designed to integrate copying with recognition of live heap objects. Copying collectors are very popular and are widely used.

Consider a simple copying collector that uses *semispaces*. We start with the heap divided into two halves—the *from* and *to spaces*.

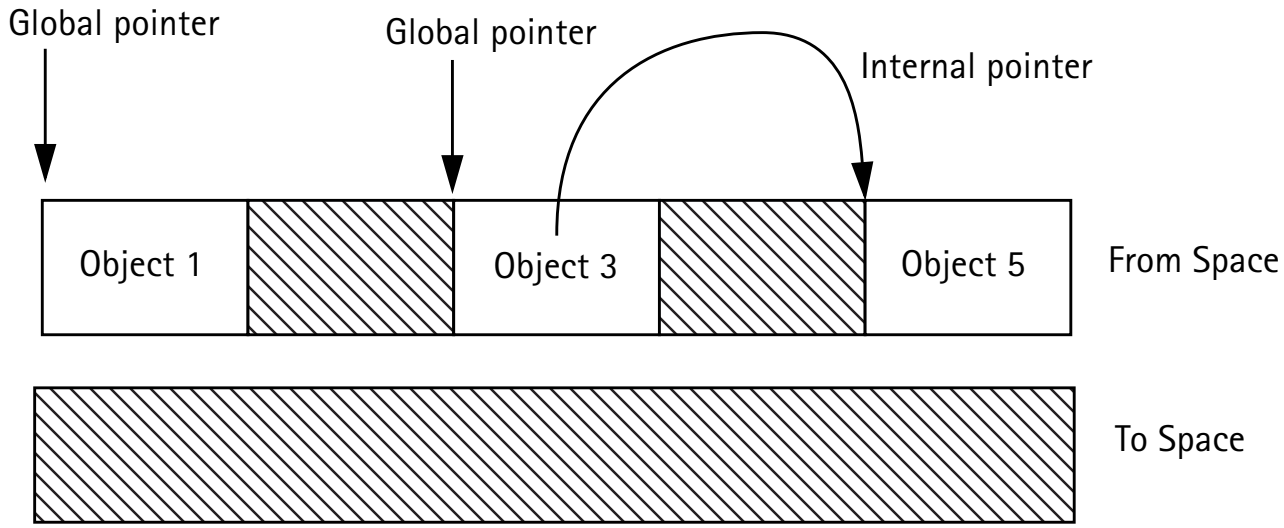
Initially, we allocate heap requests from the from space, using a simple "end of heap" pointer. When the from space is exhausted, we stop and do garbage collection.

Actually, though we *don't* collect garbage. We collect live heap objects—garbage is never touched.

We trace through global and local pointers, finding live objects. As each object is found, it is moved from its current position in the from space to the next available position in the to space.

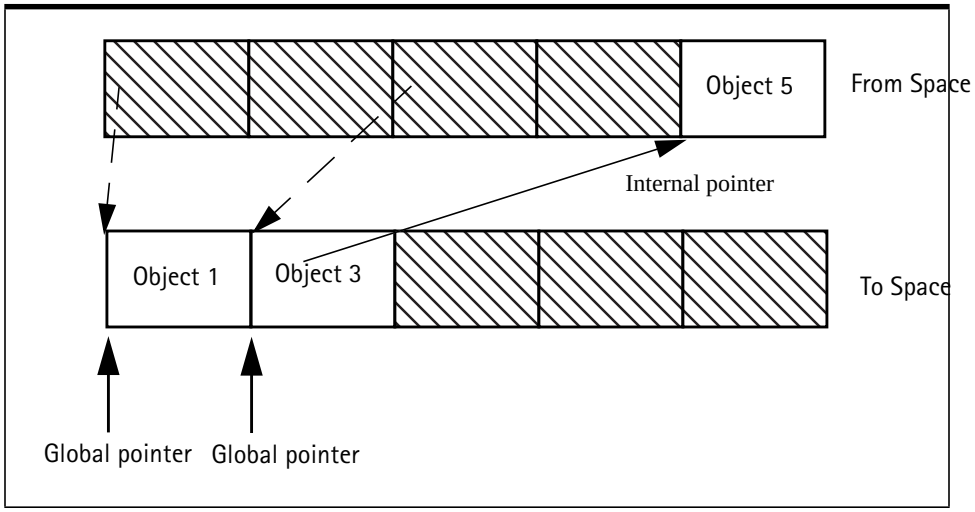
The pointer is updated to reflect the object's new location. A "forwarding pointer" is left in the object's old location in case there are multiple pointers to the same object.

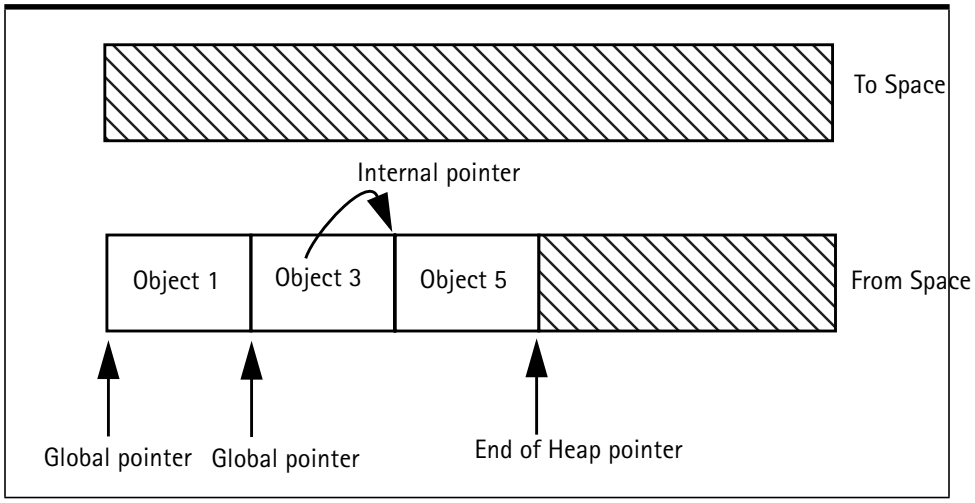
This is illustrated below:



The from space is completely filled. We trace global and local pointers, moving live objects to the to space and updating pointers. This is illustrated below. (Dashed arrows are forwarding pointers). We have yet to handle pointers internal to copied heap objects. All copied heap objects are traversed. Objects referenced are copied and internal pointers

are updated. Finally, the to and from spaces are interchanged, and heap allocation resumes just beyond the last copied object. This is illustrated in the lower figure.





The biggest advantage of copying collectors is their speed. Only live objects are copied; deallocation of dead objects is essentially free. In fact, garbage collection can be made, on average, as fast as you wish—simply make the heap bigger. As the heap gets bigger, the time between collections increases, reducing the number of times a live object must be copied. In the limit, objects are never copied, so garbage collection becomes free!

Of course, we can't increase the size of heap memory to infinity. In fact, we don't want to make the heap so large that paging is required, since swapping pages to disk is dreadfully slow. If we can make the heap large enough that the lifetime of most heap objects

is less than the time between collections, then deallocation of short-lived objects will appear to be free, though longer-lived objects will still exact a cost.

Aren't copying collectors terribly wasteful of space? After all, at most only half of the heap space is actually used. The reason for this apparent inefficiency is that *any* garbage collector that does compaction must have an area to copy live objects to. Since in the worst case *all* heap objects could be live, the target area must be as large as the heap itself. To avoid copying objects more than once, copying collectors reserve a to space as big as the from space. This is essentially a space-time trade-off, making such collectors

very fast at the expense of possibly wasted space.

If we have reason to believe that the time between garbage collections will be greater than the average lifetime of most heaps objects, we can improve our use of heap space. Assume that 50% or more of the heap will be garbage when the collector is called. We can then divide the heap into 3 segments, which we'll call A, B and C. Initially, A and B will be used as the from space, utilizing 2/3 of the heap. When we copy live objects, we'll copy them into segment C, which will be big enough if half or more of the heap objects are garbage. Then we treat C and A as the from space, using B as the to space for the next collection. If we are

unlucky and more than 1/2 the heap contains live objects, we can still get by. Excess objects are copied onto an auxiliary data space (perhaps the stack), then copied into A after all live objects in A have been moved. This slows collection down, but only rarely (if our estimate of 50% garbage per collection is sound). Of course, this idea generalizes to more than 3 segments. Thus if 2/3 of the heap were garbage (on average), we could use 3 of 4 segments as from space and the last segment as to space.

Generational Techniques

The great strength of copying collectors is that they do no work for objects that are born and die between collections. However, not all heaps objects are so short-lived. In fact, some heap objects are very long-lived. For example, many programs create a dynamic data structure at their start, and utilize that structure throughout the program. Copying collectors handle long-lived objects poorly. They are repeatedly traced and moved between semispaces without any real benefit.

Generational garbage collection techniques were developed to better handle objects with varying lifetimes. The heap is divided into two or more *generations*, each

with its own to and from space. New objects are allocated in the youngest generation, which is collected most frequently. If an object survives across one or more collections of the youngest generation, it is "promoted" to the next older generation, which is collected less often. Objects that survive one or more collections of this generation are then moved to the next older generation. This continues until very long-lived objects reach the oldest generation, which is collected very infrequently (perhaps even never).

The advantage of this approach is that long-lived objects are "filtered out," greatly reducing the cost of repeatedly processing them. Of course, some long-lived

objects will die and these will be caught when their generation is eventually collected.

An unfortunate complication of generational techniques is that although we collect older generations infrequently, we must still trace their pointers in case they reference an object in a newer generation. If we don't do this, we may mistake a live object for a dead one. When an object is promoted to an older generation, we can check to see if it contains a pointer into a younger generation. If it does, we record its address so that we can trace and update its pointer. We must also detect when an existing pointer inside an object is changed. Sometimes we can do this by checking "dirty bits" on

heap pages to see which have been updated. We then trace all objects on a page that is dirty. Otherwise, whenever we assign to a pointer that already has a value, we record the address of the pointer that is changed. This information then allows us to only trace those objects in older generations that might point to younger objects.

Experience shows that a carefully designed generational garbage collectors can be very effective. They focus on objects most likely to become garbage, and spend little overhead on long-lived objects. Generational garbage collectors are widely used in practice.

Conservative Garbage Collection

The garbage collection techniques we've studied all require that we identify pointers to heap objects accurately. In strongly typed languages like Java or ML, this can be done. We can table the addresses of all global pointers. We can include a code value in a frame (or use the return address stored in a frame) to determine the routine a frame corresponds to. This allows us to then determine what offsets in the frame contain pointers. When heap objects are allocated, we can include a type code in the object's header, again allowing us to identify pointers internal to the object.

Languages like C and C++ are weakly typed, and this makes identification of pointers much harder. Pointers may be type-cast into integers and then back into pointers. Pointer arithmetic allows pointers into the middle of an object. Pointers in frames and heap objects need not be initialized, and may contain random values. Pointers may overlay integers in unions, making the current type a dynamic property.

As a result of these complications, C and C++ have the reputation of being incompatible with garbage collection. Surprisingly, this belief is false. Using *conservative garbage collection*, C and C++ programs *can* be garbage collected.

The basic idea is simple—if we can't be sure whether a value is a pointer or not, we'll be conservative and assume it is a pointer. If what we think is a pointer isn't, we may retain an object that's really dead, but we'll find all valid pointers, and never incorrectly collect a live object. We may mistake an integer (or a floating value, or even a string) as an pointer, so compaction in any form *can't* be done. However, mark-sweep collection will work.

Garbage collectors that work with ordinary C programs have been developed. User programs need not be modified. They simply are linked to different library routines, so that **`malloc`** and **`free`** properly support the garbage collector. When new heap space is

required, dead heap objects may be automatically collected, rather than relying entirely on explicit **`free`** commands (though **`free`**s are allowed; they sometimes simplify or speed heap reuse).

With garbage collection available, C programmers need not worry about explicit heap management. This reduces programming effort and eliminates errors in which objects are prematurely freed, or perhaps never freed. In fact, experiments have shown that conservative garbage collection is very competitive in performance with application-specific manual heap management.

Jump Code

The JVM code we generate for the following if statement is quite simple and efficient.

```
if (B)
  A = 1;
else
  A = 0;
```

```
    iload 2  ; Push local #2 (B) onto stack
    ifeq L1  ; Goto L1 if B is 0 (false)
    iconst_1 ; Push literal 1 onto stack
    istore 1 ; Store stk top into local #1(A)
    goto L2  ; Skip around else part
L1: iconst_0 ; Push literal 0 onto stack
    istore 1 ; Store stk top into local #1(A)
L2:
```

In contrast, the code generated for

```
if (F == G)
  A = 1;
else
  A = 0;
```

(where F and G are local variables of type integer)

is significantly more complex:

```
  iload 4       ; Push local #4 (F) onto stack
  iload 5       ; Push local #5 (G) onto stack
  if_icmpeq L1 ; Goto L1 if F == G
  iconst_0      ; Push 0 (false) onto stack
  goto L2       ; Skip around next instruction
L1:
  iconst_1      ; Push 1 (true) onto the stack
L2:
  ifeq L3       ; Goto L3 if F==G is 0 (false)
  iconst_1      ; Push literal 1 onto stack
  istore 1      ; Store top into local #1(A)
  goto L4       ; Skip around else part
L3:
  iconst_0      ; Push literal 0 onto stack
  istore 1      ; Store top into local #1(A)
L4:
```

The problem is that in the JVM relational operators don't store a boolean value (0 or 1) onto the stack. Rather, instructions like **`if_icmpeq`** do a *conditional branch*.

So we branch to a push of 0 or 1 just so we can test the value and do a *second* conditional branch to the else part of the conditional.

Why did the JVM designers create such an odd way of evaluating relational operators?

A moment's reflection shows that we rarely actually *want* the value of a relational or logical expression. Rather, we usually

only want to do a conditional branch based on the expression's value in the context of a conditional or looping statement.

Jump code is an alternative representation of boolean values. Rather than placing a boolean value directly on the stack, we generate a conditional branch to either a **true label** or a **false label**. These labels are defined at the places where we wish execution to proceed once the boolean expression's value is known.

Returning to our previous example, we can generate **F==G** in jump code form as

```
iload 4       ; Push local #4 (F) onto stack
iload5        ; Push local #5 (G) onto stack
if_icmpne L1  ; Goto L1 if F != G
```

The label **L1** is the "false label." We branch to it if the expression **F == G** is false; otherwise, we fall through, executing the code that follows. We can then generate the then part, defining **L1** at the point where the else part is to be computed. The code we generate is

```
  iload 4       ; Push local #4 (F) onto stack
  iload5        ; Push local #5 (G) onto stack
  if_icmpne L1  ; Goto L1 if F != G
  iconst_1      ; Push literal 1 onto stack
  istore 1      ; Store top into local #1(A)
  goto L2       ; Skip around else part
L1:
  iconst_0      ; Push literal 0 onto stack
  istore 1      ; Store top into local #1(A)
L2:
```

This instruction sequence is significantly shorter (and faster) than our original translation. Jump code is routinely used in ifs, whiles and fors where we wish to alter flow-of-control rather than compute an explicit boolean value.

Jump code comes in two forms, **`JumpIfTrue`** and **`JumpIfFalse`**.

In **`JumpIfTrue`** form, the code sequence does a conditional jump (branch) if the expression is true, and "falls through" if the expression is false. Analogously, in **`JumpIfFalse`** form, the code sequence does a conditional jump (branch) if the expression is false, and "falls through" if the expression is true. We have two forms because different contexts prefer one or the other.

It is important to emphasize that even though jump code looks unusual, it is just an alternative representation of boolean values. We can convert

a boolean value on the stack to jump code by conditionally branching on its value to a true or false label.

Similarly, we convert from jump code to an explicit boolean value, by placing the jump code's true label at a load of 1 and the false label at a load of 0.

Short-Circuit Evaluation

Our translation of the && and || operators parallels that of all other binary operators: evaluate both operands onto the stack and then do an "and" or "or" operation.

But in C, C++, C#, Java (and most other languages), && and || are handled specially.

These two operators are defined to work in "short circuit" mode. That is, if the left operand is sufficient to determine the result of the operation, the right operand *isn't evaluated*.

In particular **`a&&b`** is defined as **`if a then b else false`**.

Similarly **`a||b`** is defined as **`if a then true else b`**.

The conditional evaluation of the second operand isn't just an optimization—it's essential for correctness. For example, in **`(a!=0)&&(b/a>100)`** we would perform a division by zero if the right operand were evaluated when **`a==0`**.

Jump code meshes nicely with the short-circuit definitions of `&&` and `||`, since they are already defined in terms of conditional branches.

In particular if **`exp1`** and **`exp2`** are in jump code form, then we need generate *no further code* to evaluate **`exp1&&exp2`**.

To evaluate **`&&`**, we first translate **`exp1`** into **`JumpIfFalse`** form, followed by **`exp2`**. If **`exp1`** is false, we jump out of the whole expression. If **`exp1`** is true, we fall through to **`exp2`** and evaluate it. In this way, **`exp2`** is evaluated only when necessary (when **`exp1`** is true).

Similarly, once **`exp1`** and **`exp2`** are in jump code form, **`exp1||exp2`** is easy to evaluate. We first translate **`exp1`** into **`JumpIfTrue`** form, followed by **`exp2`**. If **`exp1`** is true, we jump out of the whole expression. If **`exp1`** is false, we fall through to **`exp2`** and evaluate it. In this way, **`exp2`** is evaluated only when necessary (when **`exp1`** is false).

As an example, let's consider

```
if ((A>0)||(B<0 && C==10))
  A = 1;
else
  A = 0;
```

Assume **A**, **B** and **C** are all local integers, with indices of 1, 2 and 3 respectively.

We'll produce a **JumpIfFalse** translation, jumping to label **F** (the else part) if the expression is false and falling through to the then part if the expression is true.

Code generators for relational operators can be easily modified to produce both kinds of jump code—we can either jump if the relation holds

(**`JumpIfTrue`**) or jump if it doesn't hold (**`JumpIfFalse`**). We produce the following JVM code sequence which is quite compact and efficient.

```
    iload 1      ; Push local #1 (A) onto stack
    ifgt L1      ; Goto L1 if A > 0 is true
    iload 2      ; Push local #2 (B) onto stack
    ifge F       ; Goto F if B < 0 is false
    iload 3      ; Push local #3 (C) onto stack
    bipush 10    ; Push a byte immediate (10)
    if_icmpne F  ; Goto F if C != 10
L1:
    iconst_1     ; Push literal 1 onto stack
    istore 1     ; Store top into local #1(A)
    goto L2      ; Skip around else part
F:
    iconst_0     ; Push literal 0 onto stack
    istore 1     ; Store top into local #1(A)
L2:
```

First **`A`** is tested. If it is greater than zero, the control expression must be true, so we skip the rest of the expression and execute the then part.

Otherwise, we continue evaluating the control expression.

We next test **`B`**. If it is greater than or equal to zero, **`B<0`** is false, and so is the whole expression. We therefore branch to label **`F`** and execute the else part.

Otherwise, we finally test **`C`**. If **`C`** is not equal to 10, the control expression is false, so we branch to label **`F`** and execute the else part.

If **`C`** is equal to 10, the control expression is true, and we fall through to the then part.