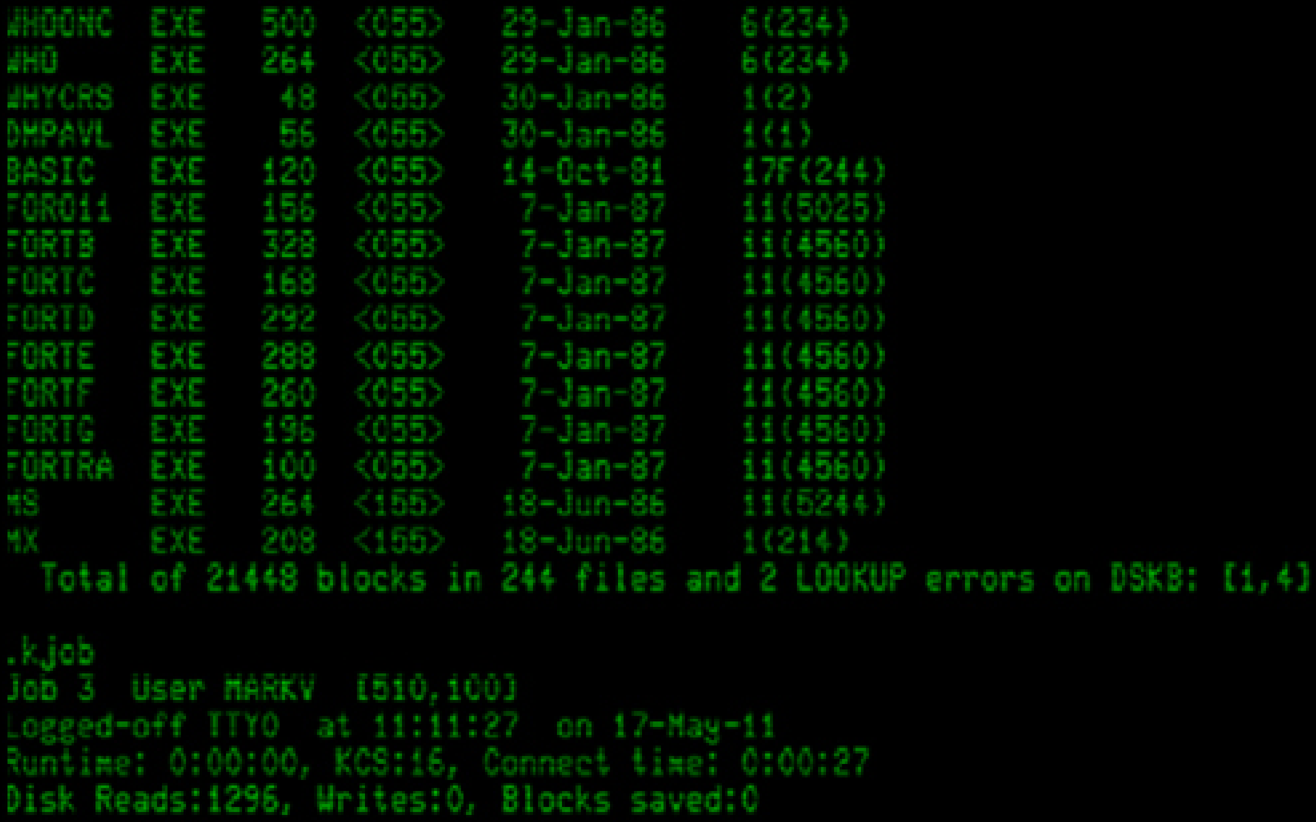
Compiler
Theory and Construction
Handbook



```
WHOONC  EXE   500  <055>   29-Jan-86    6(234)
WHO     EXE   264  <055>   29-Jan-86    6(234)
WHYCRS  EXE    48  <055>   30-Jan-86    1(2)
DMPAVL  EXE    56  <055>   30-Jan-86    1(1)
BASIC   EXE   120  <055>   14-Oct-81    17F(244)
FOR011  EXE   156  <055>    7-Jan-87    11(5025)
FORTB   EXE   328  <055>    7-Jan-87    11(4560)
FORTC   EXE   168  <055>    7-Jan-87    11(4560)
FORTD   EXE   292  <055>    7-Jan-87    11(4560)
FORTE   EXE   288  <055>    7-Jan-87    11(4560)
FORTF   EXE   260  <055>    7-Jan-87    11(4560)
FORTG   EXE   196  <055>    7-Jan-87    11(4560)
FORTRA  EXE   100  <055>    7-Jan-87    11(4560)
MS      EXE   264  <155>   18-Jun-86    11(5244)
MX      EXE   208  <155>   18-Jun-86    1(214)
  Total of 21448 blocks in 244 files and 2 LOOKUP errors on DSKB: [1,4]

.kjob
Job 3  User MARKV  [510,100]
Logged-off TTY0  at 11:11:27  on 17-May-11
Runtime: 0:00:00, KCS:16, Connect time: 0:00:27
Disk Reads:1296, Writes:0, Blocks saved:0
```

Lucius Kirkpatrick

Rohan Shepherd

First Edition, 2012

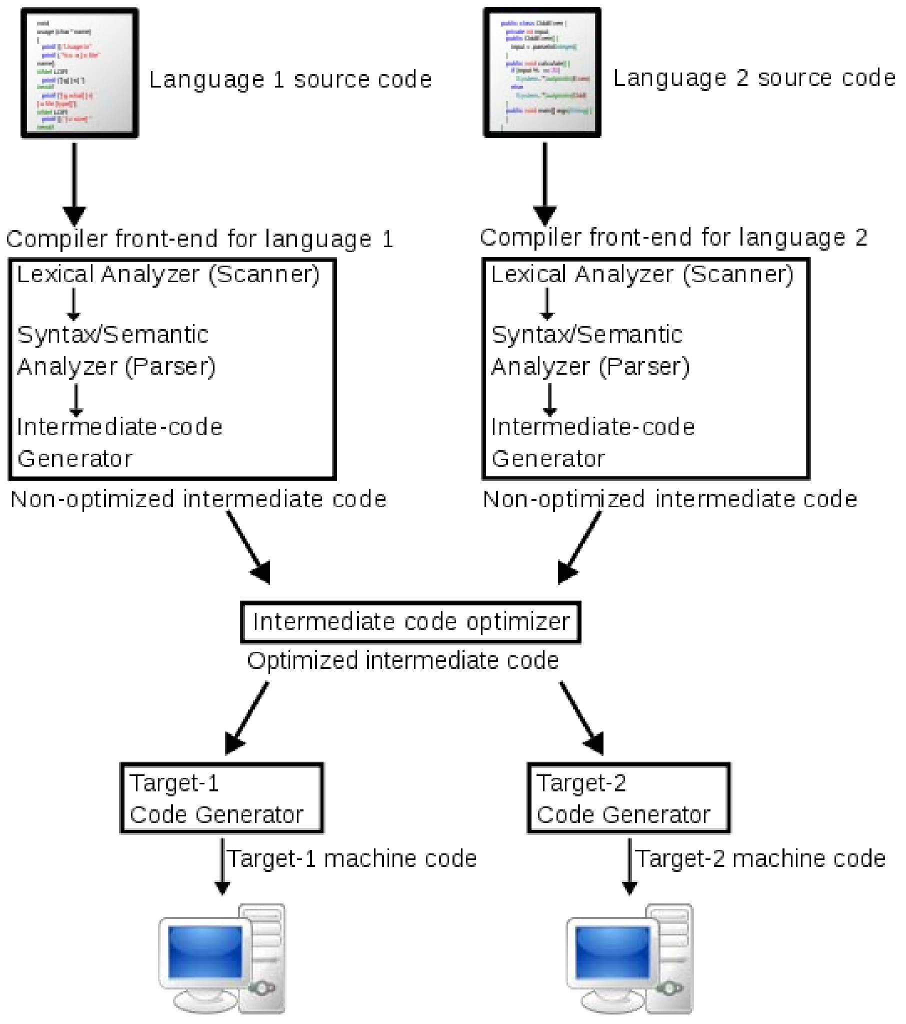
ISBN 978-81-323-1264-2

Published by:
College Publishing House
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1

Compiler


Language 1 source code
Language 2 source code
Compiler front-end for language 1
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Compiler front-end for language 2
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Non-optimized intermediate code
Non-optimized intermediate code
Intermediate code optimizer
Optimized intermediate code
Target-1 Code Generator
Target-2 Code Generator
Target-1 machine code
Target-2 machine code

A diagram of the operation of a typical multi-language, multi-target compiler

A **compiler** is a computer program (or set of programs) that transforms source code written in a programming language (the *source language*) into another computer language (the *target language*, often having a binary form known as *object code*). The most common reason for wanting to transform source code is to create an executable program.

The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code). If the compiled program can only run on a computer whose CPU or operating system is different from the one on which the compiler runs the compiler is known as a cross-compiler. A program that translates from a low level language to a higher level one is a *decompiler*. A program that translates between high-level languages is usually called a *language translator*, *source to source translator*, or *language converter*. A *language rewriter* is usually a program that translates the form of expressions without a change of language.

A compiler is likely to perform many or all of the following operations: lexical analysis, preprocessing, parsing, semantic analysis (Syntax-directed translation), code generation, and code optimization.

Program faults caused by incorrect compiler behavior can be very difficult to track down and work around and compiler implementors invest a lot of time ensuring the correctness of their software.

The term compiler-compiler is sometimes used to refer to a parser generator, a tool often used to help create the lexer and parser.

History

Software for early computers was primarily written in assembly language for many years. Higher level programming languages were not invented until the benefits of being able to reuse software on different kinds of CPUs started to become significantly greater than the cost of writing a compiler. The very limited memory capacity of early computers also created many technical problems when implementing a compiler.

Towards the end of the 1950s, machine-independent programming languages were first proposed. Subsequently, several experimental compilers were developed. The first compiler was written by Grace Hopper, in 1952, for the A-0 programming language. The FORTRAN team led by John Backus at IBM is generally credited as having introduced the first complete compiler in 1957. COBOL was an early language to be compiled on multiple architectures, in 1960.

In many application domains the idea of using a higher level language quickly caught on. Because of the expanding functionality supported by newer programming languages and the increasing complexity of computer architectures, compilers have become more and more complex.

Early compilers were written in assembly language. The first *self-hosting* compiler — capable of compiling its own source code in a high-level language — was created for Lisp by Tim Hart and Mike Levin at MIT in 1962. Since the 1970s it has become common practice to implement a compiler in the language it compiles, although both Pascal and C have been popular choices for implementation language. Building a self-hosting compiler is a bootstrapping problem—the first such compiler for a language must be compiled either by a compiler written in a different language, or (as in Hart and Levin's Lisp compiler) compiled by running the compiler in an interpreter.

Compilers in education

Compiler construction and compiler optimization are taught at universities and schools as part of the computer science curriculum. Such courses are usually supplemented with the implementation of a compiler for an educational programming language. A well-documented example is Niklaus Wirth's PL/0 compiler, which Wirth used to teach compiler construction in the 1970s. In spite of its simplicity, the PL/0 compiler introduced several influential concepts to the field:

1. Program development by stepwise refinement (also the title of a 1971 paper by Wirth)
2. The use of a recursive descent parser
3. The use of EBNF to specify the syntax of a language
4. A code generator producing portable P-code
5. The use of T-diagrams in the formal description of the bootstrapping problem

Compilation

Compilers enabled the development of programs that are machine-independent. Before the development of FORTRAN (FORmula TRANslator), the first higher-level language, in the 1950s, machine-dependent assembly language was widely used. While assembly language produces more reusable and relocatable programs than machine code on the same architecture, it has to be modified or rewritten if the program is to be executed on different hardware architecture.

With the advance of high-level programming languages soon followed after FORTRAN, such as COBOL, C, BASIC, programmers can write machine-independent source programs. A compiler translates the high-level source programs into target programs in machine languages for the specific hardwares. Once the target program is generated, the user can execute the program.

Structure of compiler

Compilers bridge source programs in high-level languages with the underlying hardwares. A compiler requires 1) to recognize legitimacy of programs, 2) to generate correct and efficient code, 3) run-time organization, 4) to format output according to assembler or linker conventions. A compiler consists of three main parts: frontend, middle-end, and backend.

Frontend checks whether the program is correctly written in terms of the programming language syntax and semantics. Here legal and illegal programs are recognized. Errors are reported, if any, in a useful way. Type checking is also performed by collecting type information. Frontend generates IR (intermediate representation) for the middle-end. Optimization of this part is almost complete so much are already automated. There are efficient algorithms typically in $O(n)$ or $O(n \log n)$.

Middle-end is where the optimizations for performance take place. Typical transformations for optimization are removal of useless or unreachable code, discovering and propagating constant values, relocation of computation to a less frequently executed place (e.g., out of a loop), or specializing a computation based on the context. Middle-end generates IR for the following backend. Most optimization efforts are focused on this part.

Backend is responsible for translation of IR into the target assembly code. The target instruction(s) are chosen for each IR instruction. Variables are also selected for the registers. Backend utilizes the hardware by figuring out how to keep parallel FUs busy, filling delay slots, and so on. Although most algorithms for optimization are in NP, heuristic techniques are well-developed.

Compiler output

One classification of compilers is by the platform on which their generated code executes. This is known as the *target platform.*

A *native* or *hosted* compiler is one whose output is intended to directly run on the same type of computer and operating system that the compiler itself runs on. The output of a cross compiler is designed to run on a different platform. Cross compilers are often used when developing software for embedded systems that are not intended to support a software development environment.

The output of a compiler that produces code for a virtual machine (VM) may or may not be executed on the same platform as the compiler that produced it. For this reason such compilers are not usually classified as native or cross compilers.

Compiled versus interpreted languages

Higher-level programming languages are generally divided for convenience into compiled languages and interpreted languages. However, in practice there is rarely anything about a language that *requires* it to be exclusively compiled or exclusively interpreted, although it is possible to design languages that rely on re-interpretation at run time. The categorization usually reflects the most popular or widespread implementations of a language — for instance, BASIC is sometimes called an interpreted language, and C a compiled one, despite the existence of BASIC compilers and C interpreters.

Modern trends toward just-in-time compilation and bytecode interpretation at times blur the traditional categorizations of compilers and interpreters.

Some language specifications spell out that implementations *must* include a compilation facility; for example, Common Lisp. However, there is nothing inherent in the definition of Common Lisp that stops it from being interpreted. Other languages have features that are very easy to implement in an interpreter, but make writing a compiler much harder; for example, APL, SNOBOL4, and many scripting languages allow programs to construct arbitrary source code at runtime with regular string operations, and then execute that code by passing it to a special evaluation function. To implement these features in a compiled language, programs must usually be shipped with a runtime library that includes a version of the compiler itself.

Hardware compilation

The output of some compilers may target hardware at a very low level, for example a Field Programmable Gate Array (FPGA) or structured Application-specific integrated circuit (ASIC). Such compilers are said to be *hardware compilers* or synthesis tools because the source code they compile effectively control the final configuration of the hardware and how it operates; the output of the compilation are not instructions that are executed in sequence - only an interconnection of transistors or lookup tables. For example, XST is the Xilinx Synthesis Tool used for configuring FPGAs. Similar tools are available from Altera, Synplicity, Synopsys and other vendors.

Compiler design

In the early days, the approach taken to compiler design used to be directly affected by the complexity of the processing, the experience of the person(s) designing it, and the resources available.

A compiler for a relatively simple language written by one person might be a single, monolithic piece of software. When the source language is large and complex, and high quality output is required the design may be split into a number of relatively independent phases. Having separate phases means development can be parceled up into small parts and given to different people. It also becomes much easier to replace a single phase by an improved one, or to insert new phases later (e.g., additional optimizations).

The division of the compilation processes into phases was championed by the Production Quality Compiler-Compiler Project (PQCC) at Carnegie Mellon University. This project introduced the terms *front end*, *middle end*, and *back end*.

All but the smallest of compilers have more than two phases. However, these phases are usually regarded as being part of the front end or the back end. The point at which these two *ends* meet is open to debate. The front end is generally considered to be where syntactic and semantic processing takes place, along with translation to a lower level of representation (than source code).

The middle end is usually designed to perform optimizations on a form other than the source code or machine code. This source code/machine code independence is intended to enable generic optimizations to be shared between versions of the compiler supporting different languages and target processors.

The back end takes the output from the middle. It may perform more analysis, transformations and optimizations that are for a particular computer. Then, it generates code for a particular processor and OS.

This front-end/middle/back-end approach makes it possible to combine front ends for different languages with back ends for different CPUs. Practical examples of this approach are the GNU Compiler Collection, LLVM, and the Amsterdam Compiler Kit, which have multiple front-ends, shared analysis and multiple back-ends.

One-pass versus multi-pass compilers

Classifying compilers by number of passes has its background in the hardware resource limitations of computers. Compiling involves performing lots of work and early computers did not have enough memory to contain one program that did all of this work. So compilers were split up into smaller programs which each made a pass over the source (or some representation of it) performing some of the required analysis and translations.

The ability to compile in a single pass has classically been seen as a benefit because it simplifies the job of writing a compiler and one pass compilers generally compile faster than multi-pass compilers. Thus, partly driven by the resource limitations of early systems, many early languages were specifically designed so that they could be compiled in a single pass (e.g., Pascal).

In some cases the design of a language feature may require a compiler to perform more than one pass over the source. For instance, consider a declaration appearing on line 20 of the source which affects the translation of a statement appearing on line 10. In this case, the first pass needs to gather information about declarations appearing after statements that they affect, with the actual translation happening during a subsequent pass.

The disadvantage of compiling in a single pass is that it is not possible to perform many of the sophisticated optimizations needed to generate high quality code. It can be difficult to count exactly how many passes an optimizing compiler makes. For instance, different phases of optimization may analyse one expression many times but only analyse another expression once.

Splitting a compiler up into small programs is a technique used by researchers interested in producing provably correct compilers. Proving the correctness of a set of small programs often requires less effort than proving the correctness of a larger, single, equivalent program.

While the typical multi-pass compiler outputs machine code from its final pass, there are several other types:

- A "source-to-source compiler" is a type of compiler that takes a high level language as its input and outputs a high level language. For example, an automatic parallelizing compiler will frequently take in a high level language program as an input and then transform the code and annotate it with parallel code annotations (e.g. OpenMP) or language constructs (e.g. Fortran's `DOALL` statements).
- Stage compiler that compiles to assembly language of a theoretical machine, like some Prolog implementations
    - This Prolog machine is also known as the Warren Abstract Machine (or WAM).
    - Bytecode compilers for Java, Python, and many more are also a subtype of this.
- Just-in-time compiler, used by Smalltalk and Java systems, and also by Microsoft .NET's Common Intermediate Language (CIL)
    - Applications are delivered in bytecode, which is compiled to native machine code just prior to execution.

Front end

The front end analyzes the source code to build an internal representation of the program, called the intermediate representation or *IR*. It also manages the symbol table, a data structure mapping each symbol in the source code to associated information such as location, type and scope. This is done over several phases, which includes some of the following:

1. **Line reconstruction**. Languages which strop their keywords or allow arbitrary spaces within identifiers require a phase before parsing, which converts the input character sequence to a canonical form ready for the parser. The top-down, recursive-descent, table-driven parsers used in the 1960s typically read the source one character at a time and did not require a separate tokenizing phase. Atlas Autocode, and Imp (and some implementations of Algol and Coral66) are examples of stropped languages whose compilers would have a *Line Reconstruction* phase.
2. Lexical analysis breaks the source code text into small pieces called *tokens*. Each token is a single atomic unit of the language, for instance a keyword, identifier or symbol name. The token syntax is typically a regular language, so a finite state automaton constructed from a regular expression can be used to recognize it. This phase is also called lexing or scanning, and the software doing lexical analysis is called a lexical analyzer or scanner.
3. Preprocessing. Some languages, e.g., C, require a preprocessing phase which supports macro substitution and conditional compilation. Typically the preprocessing phase occurs before syntactic or semantic analysis; e.g. in the case of C, the preprocessor manipulates lexical tokens rather than syntactic forms. However, some languages such as Scheme support macro substitutions based on syntactic forms.

4. Syntax analysis involves parsing the token sequence to identify the syntactic structure of the program. This phase typically builds a parse tree, which replaces the linear sequence of tokens with a tree structure built according to the rules of a formal grammar which define the language's syntax. The parse tree is often analyzed, augmented, and transformed by later phases in the compiler.
5. Semantic analysis is the phase in which the compiler adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings. Semantic analysis usually requires a complete parse tree, meaning that this phase logically follows the parsing phase, and logically precedes the code generation phase, though it is often possible to fold multiple phases into one pass over the code in a compiler implementation.

Back end

The term *back end* is sometimes confused with *code generator* because of the overlapped functionality of generating assembly code. Some literature uses *middle end* to distinguish the generic analysis and optimization phases in the back end from the machine-dependent code generators.

The main phases of the back end include the following:

1. Analysis: This is the gathering of program information from the intermediate representation derived from the input. Typical analyses are data flow analysis to build use-define chains, dependence analysis, alias analysis, pointer analysis, escape analysis etc. Accurate analysis is the basis for any compiler optimization. The call graph and control flow graph are usually also built during the analysis phase.
2. Optimization: the intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms. Popular optimizations are inline expansion, dead code elimination, constant propagation, loop transformation, register allocation and even automatic parallelization.
3. Code generation: the transformed intermediate language is translated into the output language, usually the native machine language of the system. This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions along with their associated addressing modes.

Compiler analysis is the prerequisite for any compiler optimization, and they tightly work together. For example, dependence analysis is crucial for loop transformation.

In addition, the scope of compiler analysis and optimizations vary greatly, from as small as a basic block to the procedure/function level, or even over the whole program (interprocedural optimization). Obviously, a compiler can potentially do a better job using a broader view. But that broad view is not free: large scope analysis and optimizations are very costly in terms of compilation time and memory space; this is especially true for interprocedural analysis and optimizations.

Interprocedural analysis and optimizations are common in modern commercial compilers from HP, IBM, SGI, Intel, Microsoft, and Sun Microsystems. The open source GCC was criticized for a long time for lacking powerful interprocedural optimizations, but it is changing in this respect. Another open source compiler with full analysis and optimization infrastructure is Open64, which is used by many organizations for research and commercial purposes.

Due to the extra time and space needed for compiler analysis and optimizations, some compilers skip them by default. Users have to use compilation options to explicitly tell the compiler which optimizations should be enabled.

Compiler correctness

Compiler correctness is the branch of software engineering that deals with trying to show that a compiler behaves according to its language specification. Techniques include developing the compiler using formal methods and using rigorous testing (often called compiler validation) on an existing compiler.

Related techniques

Assembly language is a type of low-level language and a program that compiles it is more commonly known as an *assembler*, with the inverse program known as a *disassembler*.

A program that translates from a low level language to a higher level one is a *decompiler*.

A program that translates between high-level languages is usually called a *language translator*, *source to source translator*, *language converter*, or *language rewriter*. The last term is usually applied to translations that do not involve a change of language.

Chapter 2

Compiler Optimization

Compiler optimization is the process of tuning the output of a compiler to minimize or maximize some attribute of an executable computer program. The most common requirement is to minimize the time taken to execute a program; a less common one is to minimize the amount of memory occupied. The growth of portable computers has created a market for minimizing the power consumed by a program. Compiler optimization is generally implemented using a sequence of *optimizing transformations*, algorithms which take a program and transform it to produce an output program that uses less resources.

It has been shown that some code optimization problems are NP-complete, or even undecidable . In practice, factors such as the programmer's willingness to wait for the compiler to complete its task place upper limits on the optimizations that a compiler implementor might provide. (Optimization is generally a very CPU- and memory-intensive process.) In the past, computer memory limitations were also a major factor in limiting which optimizations could be performed. Because of all these factors, optimization rarely produces "optimal" output in any sense, and in fact an "optimization" may impede performance in some cases; rather, they are heuristic methods for improving resource usage in typical programs.

Types of optimizations

Techniques used in optimization can be broken up among various *scopes* which can affect anything from a single statement to the entire program. Generally speaking, locally scoped techniques are easier to implement than global ones but result in smaller gains. Some examples of scopes include:

- Peephole optimizations: Usually performed late in the compilation process after machine code has been generated. This form of optimization examines a few adjacent instructions (like "looking through a peephole" at the code) to see whether they can be replaced by a single instruction or a shorter sequence of instructions. For instance, a multiplication of a value by 2 might be more efficiently executed by left-shifting the value or by adding the value to itself. (This example is also an instance of strength reduction.)

- Local optimizations: These only consider information local to a function definition. This reduces the amount of analysis that needs to be performed (saving time and reducing storage requirements) but means that worst case assumptions have to be made when function calls occur or global variables are accessed (because little information about them is available).

- *Interprocedural* or whole-program optimization: These analyze all of a program's source code. The greater quantity of information extracted means that optimizations can be more effective compared to when they only have access to local information (i.e., within a single function). This kind of optimization can also allow new techniques to be performed. For instance function inlining, where a call to a function is replaced by a copy of the function body.

- Loop optimizations: These act on the statements which make up a loop, such as a *for* loop (eg, loop-invariant code motion). Loop optimizations can have a significant impact because many programs spend a large percentage of their time inside loops.

In addition to scoped optimizations there are two further general categories of optimization:

- *Programming language-independent vs. language-dependent*: Most high-level languages share common programming constructs and abstractions — decision (if, switch, case), looping (for, while, repeat.. until, do.. while), encapsulation (structures, objects). Thus similar optimization techniques can be used across languages. However, certain language features make some kinds of optimizations difficult. For instance, the existence of pointers in C and C++ makes it difficult to optimize array accesses. However, languages such as PL/1 (that also supports pointers) nevertheless have available sophisticated optimizing compilers to achieve better performance in various other ways. Conversely, some language features make certain optimizations easier. For example, in some languages functions are not permitted to have "side effects". Therefore, if a program makes several calls to the same function with the same arguments, the compiler can immediately infer that the function's result need be computed only once.

- *Machine independent vs. machine dependent*: Many optimizations that operate on abstract programming concepts (loops, objects, structures) are independent of the machine targeted by the compiler, but many of the most effective optimizations are those that best exploit special features of the target platform.

The following is an instance of a local machine dependent optimization. To set a register to 0, the obvious way is to use the constant '0' in an instruction that sets a register value to a constant. A less obvious way is to XOR a register with itself. It is up to the compiler to know which instruction variant to use. On many RISC machines, both instructions would be equally appropriate, since they would both be the same length and take the same time. On many other microprocessors such as the Intel x86 family, it turns out that the XOR variant is shorter and probably faster, as there will be no need to decode an immediate operand, nor use the internal "immediate operand register". (A potential problem with this is that XOR may introduce a data dependency on the previous value of the register, causing a pipeline stall. However, processors often have XOR of a register with itself as a special case that doesn't cause stalls.)

Factors affecting optimization

The machine itself

Many of the choices about which optimizations can and should be done depend on the characteristics of the target machine. It is sometimes possible to parameterize some of these machine dependent factors, so that a single piece of compiler code can be used to optimize different machines just by altering the machine description parameters. GCC is a compiler which exemplifies this approach.

The architecture of the target CPU

- Number of CPU registers: To a certain extent, the more registers, the easier it is to optimize for performance. Local variables can be allocated in the registers and not on the stack. Temporary/intermediate results can be left in registers without writing to and reading back from memory.

- RISC vs. CISC: CISC instruction sets often have variable instruction lengths, often have a larger number of possible instructions that can be used, and each instruction could take differing amounts of time. RISC instruction sets attempt to limit the variability in each of these: instruction sets are usually constant length, with few exceptions, there are usually fewer combinations of registers and memory operations, and the instruction issue rate (the number of instructions completed per time period, usually an integer multiple of the clock cycle) is usually constant in cases where memory latency is not a factor. There may be several ways of carrying out a certain task, with CISC usually offering more alternatives than RISC. Compilers have to know the relative costs among the various instructions and choose the best instruction sequence.
- Pipelines: A pipeline is essentially a CPU broken up into an assembly line. It allows use of parts of the CPU for different instructions by breaking up the execution of instructions into various stages: instruction decode, address decode, memory fetch, register fetch, compute, register store, etc. One instruction could be in the register store stage, while another could be in the register fetch stage. Pipeline conflicts occur when an instruction in one stage of the pipeline depends on the result of another instruction ahead of it in the pipeline but not yet completed. Pipeline conflicts can lead to pipeline stalls: where the CPU wastes cycles waiting for a conflict to resolve.

  Compilers can *schedule*, or reorder, instructions so that pipeline stalls occur less frequently.

- Number of functional units: Some CPUs have several ALUs and FPUs. This allows them to execute multiple instructions simultaneously. There may be restrictions on which instructions can pair with which other instructions ("pairing" is the simultaneous execution of two or more instructions), and which functional unit can execute which instruction. They also have issues similar to pipeline conflicts.

Here again, instructions have to be scheduled so that the various functional units are fully fed with instructions to execute.

The architecture of the machine

- Cache size (256kB-12MB) and type (direct mapped, 2-/4-/8-/16-way associative, fully associative): Techniques such as inline expansion and loop unrolling may increase the size of the generated code and reduce code locality. The program may slow down drastically if a highly utilized section of code (like inner loops in various algorithms) suddenly cannot fit in the cache. Also, caches which are not fully associative have higher chances of cache collisions even in an unfilled cache.
- Cache/Memory transfer rates: These give the compiler an indication of the penalty for cache misses. This is used mainly in specialized applications.

Intended use of the generated code

- Debugging: When a programmer is still writing an application, he or she will recompile and test often, and so compilation must be fast. This is one reason most optimizations are deliberately avoided during the test/debugging phase. Also, program code is usually "stepped through" using a symbolic debugger, and optimizing transformations, particularly those that reorder code, can make it difficult to relate the output code with the line numbers in the original source code. This can confuse both the debugging tools and the programmers using them.

- General purpose use: Prepackaged software is very often expected to be executed on a variety of machines and CPUs that may share the same instruction set, but have different timing, cache or memory characteristics. So, the code may not be tuned to any particular CPU, or may be tuned to work best on the most popular CPU and yet still work acceptably well on other CPUs.

- Special-purpose use: If the software is compiled to be used on one or a few very similar machines, with known characteristics, then the compiler can heavily tune the generated code to those specific machines (if such options are available). Important special cases include code designed for parallel and vector processors, for which special parallelizing compilers are employed.

    - Embedded systems: These are a "common" case of special-purpose use. Embedded software can be tightly tuned to an exact CPU and memory size. Also, system cost or reliability may be more important than the code's speed. So, for example, compilers for embedded software usually offer options that reduce code size at the expense of speed, because memory is the main cost of an embedded computer. The code's timing may need to be predictable, rather than "as fast as possible," so code caching might be disabled, along with compiler optimizations that require it.

Optimization techniques

Common themes

To a large extent, compiler optimization techniques have the following themes, which sometimes conflict.

Optimize the common case
: The common case may have unique properties that allow a *fast path* at the expense of a *slow path*. If the fast path is taken most often, the result is better over-all performance.

Avoid redundancy
: Reuse results that are already computed and store them for use later, instead of recomputing them.

Less code
: Remove unnecessary computations and intermediate values. Less work for the CPU, cache, and memory usually results in faster execution. Alternatively, in embedded systems, less code brings a lower product cost.

Fewer jumps by using *straight line code*, also called *branch-free code*
: Less complicated code. Jumps (conditional or unconditional branches) interfere with the prefetching of instructions, thus slowing down code. Using inlining or loop unrolling can reduce branching, at the cost of increasing binary file size by the length of the repeated code. This tends to merge several basic blocks into one. Even when it makes performance worse, programmers may eliminate certain kinds of jumps in order to defend against side-channel attacks.

Locality
: Code and data that are accessed closely together in time should be placed close together in memory to increase spatial locality of reference.

Exploit the memory hierarchy
: Accesses to memory are increasingly more expensive for each level of the memory hierarchy, so place the most commonly used items in registers first, then caches, then main memory, before going to disk.

Parallelize
: Reorder operations to allow multiple computations to happen in parallel, either at the instruction, memory, or thread level.

More precise information is better
: The more precise the information the compiler has, the better it can employ any or all of these optimization techniques.

Runtime metrics can help

Information gathered during a test run can be used in profile-guided optimization. Information gathered at runtime (ideally with minimal overhead) can be used by a JIT compiler to dynamically improve optimization.

Strength reduction

Replace complex or difficult or expensive operations with simpler ones. For example, replacing division by a constant with multiplication by its reciprocal, or using induction variable analysis to replace multiplication by a loop index with addition.

Optimization techniques

Loop optimizations

Some optimization techniques primarily designed to operate on loops include:

Induction variable analysis

Roughly, if a variable in a loop is a simple function of the index variable, such as `j:= 4*i+1`, it can be updated appropriately each time the loop variable is changed. This is a strength reduction, and also may allow the index variable's definitions to become dead code. This information is also useful for bounds-checking elimination and dependence analysis, among other things.

Loop fission or *loop distribution*

Loop fission attempts to break a loop into multiple loops over the same index range but each taking only a part of the loop's body. This can improve locality of reference, both of the data being accessed in the loop and the code in the loop's body.

Loop fusion or *loop combining*

Another technique which attempts to reduce loop overhead. When two adjacent loops would iterate the same number of times (whether or not that number is known at compile time), their bodies can be combined as long as they make no reference to each other's data.

Loop inversion

This technique changes a standard *while* loop into a *do/while* (also known as *repeat/until*) loop wrapped in an *if* conditional, reducing the number of jumps by two, for cases when the loop is executed. Doing so duplicates the condition check (increasing the size of the code) but is more efficient because jumps usually cause a pipeline stall. Additionally, if the initial condition is known at compile-time and is known to be side-effect-free, the *if* guard can be skipped.

Loop interchange

These optimizations exchange inner loops with outer loops. When the loop variables index into an array, such a transformation can improve locality of reference, depending on the array's layout.

Loop-invariant code motion

If a quantity is computed inside a loop during every iteration, and its value is the same for each iteration, it can vastly improve efficiency to hoist it outside the loop and compute its value just once before the loop begins. This is particularly important with the address-calculation expressions generated by loops over arrays. For correct implementation, this technique must be used with loop inversion, because not all code is safe to be hoisted outside the loop.

Loop nest optimization

Some pervasive algorithms such as matrix multiplication have very poor cache behavior and excessive memory accesses. Loop nest optimization increases the number of cache hits by performing the operation over small blocks and by using a loop interchange.

Loop reversal

Loop reversal reverses the order in which values are assigned to the index variable. This is a subtle optimization which can help eliminate dependencies and thus enable other optimizations.

Loop unrolling

Unrolling duplicates the body of the loop multiple times, in order to decrease the number of times the loop condition is tested and the number of jumps, which hurt performance by impairing the instruction pipeline. A "fewer jumps" optimization. Completely unrolling a loop eliminates all overhead, but requires that the number of iterations be known at compile time.

Loop splitting

Loop splitting attempts to simplify a loop or eliminate dependencies by breaking it into multiple loops which have the same bodies but iterate over different contiguous portions of the index range. A useful special case is *loop peeling*, which can simplify a loop with a problematic first iteration by performing that iteration separately before entering the loop.

Loop unswitching

Unswitching moves a conditional from inside a loop to outside the loop by duplicating the loop's body inside each of the if and else clauses of the conditional.

Software pipelining

The loop is restructured in such a way that work done in an iteration is split into several parts and done over several iterations. In a tight loop this technique hides the latency between loading and using values.

Automatic parallelization

A loop is converted into multi-threaded or vectorized (or even both) code in order to utilize multiple processors simultaneously in a shared-memory multiprocessor (SMP) machine, including multi-core machines.

Data-flow optimizations

Data flow optimizations, based on Data-flow analysis, primarily depend on how certain properties of data are propagated by control edges in the control flow graph. Some of these include:

Common subexpression elimination

In the expression "(a+b)-(a+b)/4", "common subexpression" refers to the duplicated "(a+b)". Compilers implementing this technique realize that "(a+b)" won't change, and as such, only calculate its value once.

Constant folding and propagation

replacing expressions consisting of constants (e.g. "3 + 5") with their final value ("8") at compile time, rather than doing the calculation in run-time. Used in most modern languages.

Alias classification and pointer analysis

in the presence of pointers, it is difficult to make any optimizations at all, since potentially any variable can have been changed when a memory location is assigned to. By specifying which pointers can alias which variables, unrelated pointers can be ignored.

Dead store elimination

removal of assignments to variables that are not subsequently read, either because the lifetime of the variable ends or because of a subsequent assignment that will overwrite the first value.

SSA-based optimizations

These optimizations are intended to be done after transforming the program into a special form called static single assignment, in which every variable is assigned in only one place. Although some function without SSA, they are most effective with SSA. Many optimizations listed in other sections also benefit with no special changes, such as register allocation.

Global value numbering

GVN eliminates redundancy by constructing a value graph of the program, and then determining which values are computed by equivalent expressions. GVN is able to identify some redundancy that common subexpression elimination cannot, and vice versa.

Sparse conditional constant propagation

Effectively equivalent to iteratively performing constant propagation, constant folding, and dead code elimination until there is no change, but is much more efficient. This optimization symbolically executes the program, simultaneously propagating constant values and eliminating portions of the control flow graph that this makes unreachable.

Code generator optimizations

register allocation

The most frequently used variables should be kept in processor registers for fastest access. To find which variables to put in registers an interference-graph is created. Each variable is a vertex and when two variables are used at the same time (have an intersecting liverange) they have an edge between them. This graph is colored using for example Chaitin's algorithm using the same number of colors as there are registers. If the coloring fails one variable is "spilled" to memory and the coloring is retried.

instruction selection

Most architectures, particularly CISC architectures and those with many addressing modes, offer several different ways of performing a particular operation, using entirely different sequences of instructions. The job of the instruction selector is to do a good job overall of choosing which instructions to implement which operators in the low-level intermediate representation with. For example, on many processors in the 68000 family and on the x86 architecture, complex addressing modes can be used in statements like "lea 25(a1,d5*4), a0", allowing a single instruction to perform a significant amount of arithmetic with less storage.

instruction scheduling

Instruction scheduling is an important optimization for modern pipelined processors, which avoids stalls or bubbles in the pipeline by clustering instructions with no dependencies together, while being careful to preserve the original semantics.

rematerialization

Rematerialization recalculates a value instead of loading it from memory, preventing a memory access. This is performed in tandem with register allocation to avoid spills.

code factoring

If several sequences of code are identical, or can be parameterized or reordered to be identical, they can be replaced with calls to a shared subroutine. This can often share code for subroutine set-up and sometimes tail-recursion.

trampolines

Many CPUs have smaller subroutine call instructions to access low memory. A compiler can save space by using these small calls in the main body of code. Jump instructions in low memory can access the routines at any address. This multiplies space savings from code factoring.

reordering computations

Based on integer linear programming, restructuring compilers enhance data locality and expose more parallelism by reordering computations. Space-optimizing compilers may reorder code to lengthen sequences that can be factored into subroutines.

Functional language optimizations

Although many of these also apply to non-functional languages, they either originate in, are most easily implemented in, or are particularly critical in functional languages such as Lisp and ML.

Removing recursion

Recursion is often expensive, as a function call consumes stack space and involves some overhead related to parameter passing and flushing the instruction cache. Tail recursive algorithms can be converted to iteration, which does not have call overhead and uses a constant amount of stack space, through a process called tail recursion elimination or tail call optimization. Some functional languages, e.g. Scheme, mandate that tail calls be optimized by a conforming implementation, due to their prevalence in these languages.

Data structure fusion

Because of the high level nature by which data structures are specified in functional languages such as Haskell, it is possible to combine several recursive functions which produce and consume some temporary data structure so that the data is passed directly without wasting time constructing the data structure.

Other optimizations

Please help separate and categorize these further and create detailed pages for them, especially the more complex ones, or link to one where one exists.

Bounds-checking elimination

Many languages, for example Java, enforce bounds-checking of all array accesses. This is a severe performance bottleneck on certain applications such as scientific code. Bounds-checking elimination allows the compiler to safely remove bounds-checking in many situations where it can determine that the index must fall within valid bounds, for example if it is a simple loop variable.

Branch offset optimization (machine independent)

Choose the shortest branch displacement that reaches target

Code-block reordering

Code-block reordering alters the order of the basic blocks in a program in order to reduce conditional branches and improve locality of reference.

Dead code elimination

Removes instructions that will not affect the behaviour of the program, for example definitions which have no uses, called dead code. This reduces code size and eliminates unnecessary computation.

Factoring out of invariants

If an expression is carried out both when a condition is met and is not met, it can be written just once outside of the conditional statement. Similarly, if certain types of expressions (e.g. the assignment of a constant into a variable) appear inside a loop, they can be moved out of it because their effect will be the same no matter if they're executed many times or just once. Also known as total redundancy elimination. A more powerful optimization is Partial redundancy elimination (PRE).

Inline expansion or macro expansion

When some code invokes a procedure, it is possible to directly insert the body of the procedure inside the calling code rather than transferring control to it. This saves the overhead related to procedure calls, as well as providing great opportunity for many different parameter-specific optimizations, but comes at the cost of space; the procedure body is duplicated each time the procedure is called inline. Generally, inlining is useful in performance-critical code that makes a large number of calls to small procedures. A "fewer jumps" optimization.

Jump threading

In this pass, conditional jumps in the code that branch to identical or inverse tests are detected, and can be "threaded" through a second conditional test.

Reduction of cache collisions

(e.g. by disrupting alignment within a page)

Stack height reduction

Rearrange expression tree to minimize resources needed for expression evaluation.

Test reordering

If we have two tests that are the condition for something, we can first deal with the simpler tests (e.g. comparing a variable to something) and only then with the complex tests (e.g. those that require a function call). This technique complements lazy evaluation, but can be used only when the tests are not dependent on one another. Short-circuiting semantics can make this difficult.

Interprocedural optimizations

Interprocedural optimization works on the entire program, across procedure and file boundaries. It works tightly with intraprocedural counterparts, carried out with the cooperation of a local part and global part. Typical interprocedural optimizations are: procedure inlining, interprocedural dead code elimination, interprocedural constant propagation, and procedure reordering. As usual, the compiler needs to perform interprocedural analysis before its actual optimizations. Interprocedural analyses include alias analysis, array access analysis, and the construction of a call graph.

Interprocedural optimization is common in modern commercial compilers from SGI, Intel, Microsoft, and Sun Microsystems. For a long time the open source GCC was criticized for a lack of powerful interprocedural analysis and optimizations, though this is now improving. Another good open source compiler with full analysis and optimization infrastructure is Open64, which is used by many organizations for research and for commercial purposes.

Due to the extra time and space required by interprocedural analysis, most compilers do not perform it by default. Users must use compiler options explicitly to tell the compiler to enable interprocedural analysis and other expensive optimizations.

Problems with optimization

Early in the history of compilers, compiler optimizations were not as good as hand-written ones. As compiler technologies have improved, good compilers can often generate better code than human programmers — and good post pass optimizers can improve highly hand-optimized code even further. For RISC CPU architectures, and even more so for VLIW hardware, compiler optimization is the key for obtaining efficient code, because RISC instruction sets are so compact that it is hard for a human to manually schedule or combine small instructions to get efficient results. Indeed, these architectures were designed to rely on compiler writers for adequate performance.

However, optimizing compilers are by no means perfect. There is no way that a compiler can guarantee that, for all program source code, the fastest (or smallest) possible equivalent compiled program is output; such a compiler is fundamentally impossible because it would solve the halting problem.

This may be proven by considering a call to a function, foo(). This function returns nothing and does not have side effects (no I/O, does not modify global variables and "live" data structures, etc.). The fastest possible equivalent program would be simply to eliminate the function call. However, if the function foo() in fact does *not* return, then the program with the call to foo() would be different from the program without the call; the optimizing compiler will then have to determine this by solving the halting problem.

Additionally, there are a number of other more practical issues with optimizing compiler technology:

- Optimizing compilers focus on relatively shallow "constant-factor" performance improvements and do not typically improve the algorithmic complexity of a solution. For example, a compiler will not change an implementation of bubble sort to use quicksort instead.

- Compilers usually have to support a variety of conflicting objectives, such as cost of implementation, compilation speed and quality of generated code.

- A compiler typically only deals with a part of a program at a time, often the code contained within a single file or module; the result is that it is unable to consider contextual information that can only be obtained by processing the other files.

- The overhead of compiler optimization: Any extra work takes time; whole-program optimization is time consuming for large programs.

- The often complex interaction between optimization phases makes it difficult to find an optimal sequence in which to execute the different optimization phases.

Work to improve optimization technology continues. One approach is the use of so-called "post pass" optimizers (some commercial versions of which date back to mainframe software of the late 1970s). These tools take the executable output by an "optimizing" compiler and optimize it even further. Post pass optimizers usually work on the assembly language or machine code level (contrast with compilers that optimize intermediate representations of programs). The performance of post pass compilers are limited by the fact that much of the information available in the original source code is not always available to them.

As processor performance continues to improve at a rapid pace, while memory bandwidth improves more slowly, optimizations that reduce memory bandwidth (even at the cost of making the processor execute "extra" instructions) will become more useful.

Examples of this, already mentioned above, include loop nest optimization and rematerialization.

Chapter 3

Cross Compiler and Just-In-Time Compilation

Cross compiler

A **cross compiler** is a compiler capable of creating executable code for a platform other than the one on which the compiler is run. Cross compiler tools are used to generate executables for embedded system or multiple platforms. It is used to compile for a platform upon which it is not feasible to do the compiling, like microcontrollers that don't support an operating system. It has become more common to use this tool for paravirtualization where a system may have one or more platforms in use.

Not targeted by this definition are source to source translators, which are often mistakenly called cross compilers.

Uses of cross compilers

The fundamental use of a cross compiler is to separate the build environment from target environment. This is useful in a number of situations:

- Embedded computers where a device has extremely limited resources. For example, a microwave oven will have an extremely small computer to read its touchpad and door sensor, provide output to a digital display and speaker, and to control the machinery for cooking food. This computer will not be powerful enough to run a compiler, a file system, or a development environment. Since debugging and testing may also require more resources than are available on an embedded system, cross-compilation can be less involved and less prone to errors than native compilation.
- Compiling for multiple machines. For example, a company may wish to support several different versions of an operating system or to support several different operating systems. By using a cross compiler, a single build environment can be set up to compile for each of these targets.
- Compiling on a server farm. Similar to compiling for multiple machines, a complicated build that involves many compile operations can be executed across

any machine that is free, regardless of its underlying hardware or the operating system version that it is running.
- Bootstrapping to a new platform. When developing software for a new platform, or the emulator of a future platform, one uses a cross compiler to compile necessary tools such as the operating system and a native compiler.
- Compiling native code for emulators for older now-obsolete platforms like the Commdore 64 or Apple II by enthusiasts who use cross compilers that run on a current platform (such as Aztec C's MS DOS 6502 cross compilers running under Windows XP).

Use of virtual machines (such as Java's JVM) resolves some of the reasons for which cross compilers were developed. The virtual machine paradigm allows the same compiler output to be used across multiple target systems.

Typically the hardware architecture differs (e.g. compiling a program destined for the MIPS architecture on an x86 computer) but cross-compilation is also applicable when only the operating system environment differs, as when compiling a FreeBSD program under Linux, or even just the system library, as when compiling programs with uClibc on a glibc host.

Canadian Cross

The **Canadian Cross** is a technique for building cross compilers for other machines. Given three machines A, B, and C, one uses machine A to build a cross compiler that runs on machine B to create executables for machine C. When using the Canadian Cross with GCC, there may be four compilers involved:

- The *proprietary native Compiler for machine A (1)* is used to build the *gcc native compiler for machine A (2)*.
- The *gcc native compiler for machine A (2)* is used to build the *gcc cross compiler from machine A to machine B (3)*
- The *gcc cross compiler from machine A to machine B (3)* is used to build the *gcc cross compiler from machine B to machine C (4)*

The end-result cross compiler (4) will not be able to run on your build machine A; instead you would use it on machine B to compile an application into executable code that would then be copied to machine C and executed on machine C.

For instance, NetBSD provides a POSIX Unix shell script named `build.sh` which will first build its own toolchain with the host's compiler; this, in turn, will be used to build the cross-compiler which will be used to build the whole system.

The term **Canadian Cross** came about because at the time that these issues were all being hashed out, Canada had three national political parties.

Time line of early cross compilers

- 1979 - ALGOL 68C generated ZCODE, this added on porting the compiler and other ALGOL 68 applications to alternate platforms. To compile the ALGOL 68C compiler required about 120kB of memory. With Z80 its 64kB memory is too small to actually compile the compiler. So for the Z80 the compiler itself had to be cross compiled from the larger CAP capability computer or an IBM 370 mainframe.

GCC and cross compilation

GCC, a free software collection of compilers, can be set up to cross compile. It supports many platforms and languages. However, due to limited volunteer time and the huge amount of work it takes to maintain working cross compilers, in many releases some of the cross compilers are broken.

GCC requires that a compiled copy of binutils be available for each targeted platform. Especially important is the GNU Assembler. Therefore, binutils first has to be compiled correctly with the switch `--target=some-target` sent to the configure script. GCC also has to be configured with the same `--target` option. GCC can then be run normally provided that the tools, which binutils creates, are available in the path, which can be done using the following (on UNIX-like operating systems with bash):

```
PATH=/path/to/binutils/bin:$PATH; make
```

Cross compiling GCC requires that a portion of the *target platform*'s C standard library be available on the *host platform*. At least the crt0, ... components of the library must be available. You may choose to compile the full C library, but that can be too large for many platforms. The alternative is to use newlib, which is a small C library containing only the most essential components required to compile C source code. To configure GCC with newlib, use the switch `--with-newlib`.

The GNU autotools packages (i.e. autoconf, automake, and libtool) use the notion of a *build platform*, a *host platform*, and a *target platform*. The *build platform* is where the code is actually compiled. The *host platform* is where the compiled code will execute. The *target platform* usually only applies to compilers. It represents what type of object code the package itself will produce (such as cross-compiling a cross-compiler); otherwise the *target platform* setting is irrelevant. For example, consider cross-compiling a video game that will run on a Dreamcast. The machine where the game is compiled is the *build platform* while the Dreamcast is the *host platform*.

Another method that is popularly used by embedded Linux developers is to use gcc, g++, gjc etc with scratchbox or the newer scratchbox2. These tools create a "chroot"ed sandbox where you can build up your tools, libc, and libraries without having to set extra paths. It also has facilities for tricking the runtime into thinking it is on (for example) an ARM CPU so things like configure scripts will run. The downside to scratchbox is that it is slower and you lose access to most of your tools that are on the host. The speed loss is not terrible and you can move host tools into scratchbox.

Manx Aztec C cross compilers

Manx Software Systems, of Shrewsbury, New Jersey, produced C compilers beginning in the 1980s targeted at professional developers for a variety of platforms up to and including PCs and Macs.

Manx's Aztec C programming language was available for a variety of platforms including MS DOS, Apple II DOS 3.3 and ProDOS, Commodore 64, Macintosh 68XXX and Amiga.

From the 1980s and continuing throughout the 1990s until Manx Software Systems disappeared, the MS DOS version of Aztec C was offered both as a native mode compiler or as a cross compiler for other platforms with different processors including the Commodore 64 and Apple II. Internet distributions still exist for Aztec C including their MS DOS based cross compilers. They are still in use today.

Manx's Aztec C86, their native mode 8086 MS DOS compiler, was also a cross compiler. Although it did not compile code for a different processor like their Aztec C65 6502 cross compilers for the Commodore 64 and Apple II, it created binary executables for then-legacy operating systems for the 16 bit 8086 family of processors.

When the IBM PC was first introduced it was available with a choice of operating systems, CP/M 86 and PC DOS being two of them. Aztec C86 was provided with link libraries for generating code for both IBM PC operating systems. Throughout the 1980s later versions of Aztec C86 (3.xx, 4.xx and 5.xx) added support for MS DOS "transitory" versions 1 and 2 and which were less robust than the "baseline" MS DOS version 3 and later which Aztec C86 targeted until its demise.

Finally, Aztec C86 provided C language developers with the ability to produce ROM-able "HEX" code which could then be transferred using a ROM Burner directly to an 8086 based processor. Paravirtualization may be more common today but the practice of creating low-level ROM code was more common per-capita during those years when device driver development was often done by application programmers for individual applications, and new devices amounted to a cottage industry. It was not uncommon for application programmers to interface directly with hardware without support from the manufacturer. This practice was similar to Embedded Systems Development today.

Thomas Fenwick and James Goodnow II were the two principal developers of Aztec-C. Fenwick later became notable as the author of the Microsoft Windows CE Kernel or NK ("New Kernel") as it was then called.

Microsoft C cross compilers

Early History - 1980's

Microsoft C (MSC) has a long history dating back to the 1980s. The first Microsoft C Compilers were made by the same company who made Lattice C and were rebranded by Microsoft as their own, until MSC 4 was released, which was the first version that Microsoft produced themselves.

In 1987 many developers started switching to Microsoft C, and many more would follow throughout the development of Microsoft Windows to its present state. Products like Clipper and later Clarion emerged that offered easy database application development by using cross language techniques, allowing part of their programs to be compiled with Microsoft C.

1987

C programs had long been linked with modules written in Assembly Language. C itself was usually written in Assembly Language, and most C compilers (even current compilers) offer an Assembly Language pass (that can be "tweaked" for efficiency then linked to the rest of the program after assembling).

Compilers like Aztec-C converted everything to assembly language as a distinct pass and then assembled the code in a distinct pass, and were noted for their very efficient and small code, but by 1987 the optimizer built into Microsoft C was very good, and only "mission critical" parts of a program were usually considered for rewriting. In fact, C language programming had taken over as the "lowest-level" language, with programming becoming a multi-disciplinary growth industry and projects becoming larger, with programmers writing user interfaces and database interfaces in higher-level languages, and a need had emerged for cross language development that continues to this day.

By 1987, with the release of MSC 5.1, Microsoft offered a cross language development environment for MS DOS. 16 bit binary object code written in Assembly Language (MASM) and Microsoft's other languages including Quick Basic, Pascal, and Fortran could be linked together into one program, in a process they called "Mixed Language Programming" and now "InterLanguage Calling". If BASIC was used in this mix, the main program needed to be in BASIC to support the internal runtime that compiled BASIC required for garbage collection and its other managed operations that simulated a BASIC Interpreter like QBasic in MS DOS.

The C code in particular needed to be written to pass its variables in "reverse order" on the stack and return its values on the stack rather than in a processor register. There were other programming rules to make all the languages work together, but this particular rule persisted through the cross language development that continued throughout Windows 16 and 32 bit versions and in the development of programs for OS 2, and which persists to this day. It is known as the "Pascal Calling Convention" but is so common that it is taken for granted and the term is rarely used.

Another type of cross compilation that Microsoft C was used for during this time was in retail applications that require Handheld Devices like the Symbol Technologies PDT3100 (used to take inventory), which provided a link library targeted at an 8088 based Bar Code Scanner. Comparable to what is done today for that same market using Windows Mobile by companies like Motorola who bought Symbol, the application was built on the host computer then transferred to the Handheld Device (via a serial cable) where it was run.

Early 1990's

Throughout the 1990s and beginning with MSC 6 (their first ANSI C compliant compiler) Microsoft re-focused their C compilers on the emerging Windows Market, and also on OS 2 and in the development of GUI programs. Mixed Language compatibility remained through MSC 6 on the MS DOS side, but the API for Microsoft Windows 3.0 and 3.1 was written in MSC 6. MSC 6 was also extended to provide support for 32 Bit assemblies and support for the emerging Windows for Workgroups (WFW) and Windows NT which would form the foundation for Windows XP. A programming practice called a Thunk was even introduced to allow cross assembly instruction passing between 16 and 32 bit programs that took advantage of runtime binding rather than the static binding that was favoured in Monolithic 16 bit MS DOS applications. Static binding is still favoured by some native code developers but does not generally provide the degree of re-use required by newer best practices like CMM.

MS DOS support was still provided with the release of Microsoft's first C++ Compiler, MSC 7, which was backwardly compatible with the C programming language and MS DOS and supported both 16 bit and 32 bit code generation.

It is fair to say at this point that MSC took over where Aztec C86 left off. Since the market share for C compilers had turned to cross compilers which took advantage of the latest and greatest Windows features, offered C and C++ in a single bundle and still supported MS DOS systems that were already a decade old, the smaller companies that produced compilers like Aztec C could no longer compete and either turned to niche markets like embedded systems or disappeared.

MS DOS and 16 bit code generation support continued until MSC 8.00c which was bundled with Microsoft C++ and Microsoft Application Studio 1.5, the forerunner of Microsoft Visual Studio which is the cross development environment that Microsoft provide today.

Late 1990's

MSC 12 was released with Microsoft Visual Studio 6 and no longer provided support for MS DOS 16 bit binaries instead providing support for 32 bit console applications, but provided support for WIN 95 and WIN 98 code generation as well as for WIN NT. Link libraries were available for other processors that ran Microsoft Windows; a practice that Microsoft continues to this day.

MSC 13 was released with Visual Studio 2003, and MSC 14 was released with Visual Studio 2005, both which will still produce code for older systems like Windows 95, but which will produce code for several target platforms including the Mobile Market and the ARM processor.

DotNET and beyond

In 2001 Microsoft developed the Common Language Runtime (CLR), which formed the core for their DotNET (.NET) compiler in the Visual Studio IDE. This layer on the operating system which is in the API allows the mixing of development languages compiled across platforms that run the Windows operating system.

The DotNET runtime and CLR provide a mapping layer to the core routines for the processor and the devices on the target computer. The command line C compiler in Visual Studio will compile native code for a variety of processors and can be used to build the core routines themselves.

Microsoft .NET applications for target platforms like Windows Mobile on the ARM Processor cross-compile on Windows machines with a variety of processors and Microsoft also offer emulators and remote deployment environments that require very little configuration, unlike the cross compilers in days gone by or on other platforms.

Runtime libraries, such as Mono, provide compatibility for cross-compiled .NET programs to other operating systems, such as Linux.

Libraries like Qt and its predecessors including XVT Design provide source code level cross development capability with other platforms, while still using Microsoft C to build the Windows versions. Other compilers like MinGW have also become popular in this area since they are more directly compatible with the Unixes that comprise the non-Windows side of software development allowing those developers to target all platforms using a familiar build environment.

Just-in-time compilation

In computing, **just-in-time compilation** (**JIT**), also known as **dynamic translation**, is a method to improve the runtime performance of computer programs. Traditionally, computer programs had two modes of runtime operation, either interpreted or static (ahead-of-time) compilation. Interpreted code is translated from a high-level language to a machine code continuously during every execution, whereas statically compiled code is translated into machine code before execution, and only requires this translation once.

JIT compilers represent a hybrid approach, with translation occurring continuously, as with interpreters, but with caching of translated code to minimize performance degradation. It also offers other advantages over statically compiled code at development time, such as handling of late-bound data types and the ability to enforce security guarantees.

JIT builds upon two earlier ideas in run-time environments: *bytecode compilation* and *dynamic compilation*. It converts code at *runtime* prior to executing it natively, for example bytecode into native machine code.

Several modern runtime environments, such as Microsoft's .NET Framework and most implementations of Java, rely on JIT compilation for high-speed code execution.

Overview

In a bytecode-compiled system, source code is translated to an intermediate representation known as bytecode. Bytecode is not the machine code for any particular computer, and may be portable among computer architectures. The bytecode may then be interpreted by, or run on, a virtual machine. A just-in-time compiler can be used as a way to speed up execution of bytecode. At the time the bytecode is run, the just-in-time compiler will compile some or all of it to native machine code for better performance. This can be done per-file, per-function or even on any arbitrary code fragment; the code can be compiled when it is about to be executed (hence the name "just-in-time"), and then cached and reused later without needing to be recompiled.

In contrast, a traditional *interpreted virtual machine* will simply interpret the bytecode, generally with much lower performance. Some *interpreter*s even interpret source code, without the step of first compiling to bytecode, with even worse performance. *Statically compiled code* or *native code* is compiled prior to deployment. A *dynamic compilation environment* is one in which the compiler can be used during execution. For instance, most Common Lisp systems have a `compile` function which can compile new functions created during the run. This provides many of the advantages of JIT, but the programmer, rather than the runtime, is in control of what parts of the code are compiled. This can also compile dynamically generated code, which can, in many scenarios, provide substantial performance advantages over statically compiled code, as well as over most JIT systems.

A common goal of using JIT techniques is to reach or surpass the performance of static compilation, while maintaining the advantages of bytecode interpretation: Much of the "heavy lifting" of parsing the original source code and performing basic optimization is often handled at compile time, prior to deployment: compilation from bytecode to machine code is much faster than compiling from source. The deployed bytecode is portable, unlike native code. Since the runtime has control over the compilation, like interpreted bytecode, it can run in a secure sandbox. Compilers from bytecode to machine code are easier to write, because the portable bytecode compiler has already done much of the work.

JIT code generally offers far better performance than interpreters. In addition, it can in some cases offer better performance than static compilation, as many optimizations are only feasible at run-time:

1. The compilation can be optimized to the targeted CPU and the operating system model where the application runs. For example JIT can choose SSE2 CPU instructions when it detects that the CPU supports them. To obtain this level of optimization specificity with a static compiler, one must either compile a binary for each intended platform/architecture, or else include multiple versions of portions of the code within a single binary.
2. The system is able to collect statistics about how the program is actually running in the environment it is in, and it can rearrange and recompile for optimum performance. However, some static compilers can also take profile information as input.
3. The system can do global code optimizations (e.g. inlining of library functions) without losing the advantages of dynamic linking and without the overheads inherent to static compilers and linkers. Specifically, when doing global inline substitutions, a static compilation process may need run-time checks and ensure that a virtual call would occur if the actual class of the object overrides the inlined method, and boundary condition checks on array accesses may need to be processed within loops. With just-in-time compilation in many cases this processing can be moved out of loops, often giving large increases of speed.
4. Although this is possible with statically compiled garbage collected languages, a bytecode system can more easily rearrange executed code for better cache utilization.

Startup delay and optimizations

JIT typically causes a slight delay in initial execution of an application, due to the time taken to load and compile the bytecode. Sometimes this delay is called "startup time delay". In general, the more optimization JIT performs, the better the code it will generate, but the initial delay will also increase. A JIT compiler therefore has to make a trade-off between the compilation time and the quality of the code it hopes to generate. However, it seems that much of the startup time is sometimes due to IO-bound operations rather than JIT compilation (for example, the *rt.jar* class data file for the Java Virtual Machine is 40 MB and the JVM must seek a lot of data in this contextually huge file).

One possible optimization, used by Sun's HotSpot Java Virtual Machine, is to combine interpretation and JIT compilation. The application code is initially interpreted, but the JVM monitors which sequences of bytecode are frequently executed and translates them to machine code for direct execution on the hardware. For bytecode which is executed only a few times, this saves the compilation time and reduces the initial latency; for frequently executed bytecode, JIT compilation is used to run at high speed, after an initial phase of slow interpretation. Additionally, since a program spends most time executing a minority of its code, the reduced compilation time is significant. Finally, during the initial code interpretation, execution statistics can be collected before compilation, which helps to perform better optimization.

The correct tradeoff can vary due to circumstances. For example, Sun's Java Virtual Machine has two major modes—client and server. In client mode, minimal compilation and optimization is performed, to reduce startup time. In server mode, extensive compilation and optimization is performed, to maximize performance once the application is running by sacrificing startup time. Other Java just-in-time compilers have used a runtime measurement of the number of times a method has executed combined with the bytecode size of a method as a heuristic to decide when to compile. Still another uses the number of times executed combined with the detection of loops. In general, it is much harder to accurately predict which methods to optimize in short-running applications than in long-running ones.

Native Image Generator (Ngen) by Microsoft is another approach at reducing the initial delay. Ngen pre-compiles (or "pre-jits") bytecode in a Common Intermediate Language image into machine native code. As a result, no runtime compilation is needed. .NET framework 2.0 shipped with Visual Studio 2005 runs Ngen on all of the Microsoft library DLLs right after the installation. Pre-jitting provides a way to improve the startup time. However, the quality of code it generates might not be as good as the one that is jitted, for the same reasons why code compiled statically, without profile-guided optimization, cannot be as good as JIT compiled code in the extreme case: the lack of profiling data to drive, for instance, inline caching.

There also exist Java implementations that combine an AOT (ahead-of-time) compiler with either a JIT compiler (Excelsior JET) or interpreter (GNU Compiler for Java.)

History

The earliest published JIT compiler is generally attributed to work on LISP by McCarthy in 1960. In his seminal paper *Recursive functions of symbolic expressions and their computation by machine, Part I*, he mentions functions that are translated during runtime, thereby sparing the need to save the compiler output to punch cards. In 1968, Thompson presented a method to automatically compile regular expressions to machine code, which is then executed in order to perform the matching on an input text. An influential technique for deriving compiled code from interpretation was pioneered by Mitchell in 1970, which he implemented for the experimental language *LC*^{2}.

Smalltalk pioneered new aspects of JIT compilations. For example, translation to machine code was done on demand, and the result was cached for later use. When memory became sparse, the system would delete some of this code and regenerate it when it was needed again. Sun's Self language improved these techniques extensively and was at one point the fastest Smalltalk system in the world; achieving up to half the speed of optimized C but with a fully object-oriented language.

Self was abandoned by Sun, but the research went into the Java language, and currently it is used by most implementations of the Java Virtual Machine, as HotSpot builds on, and extensively uses, this research base.

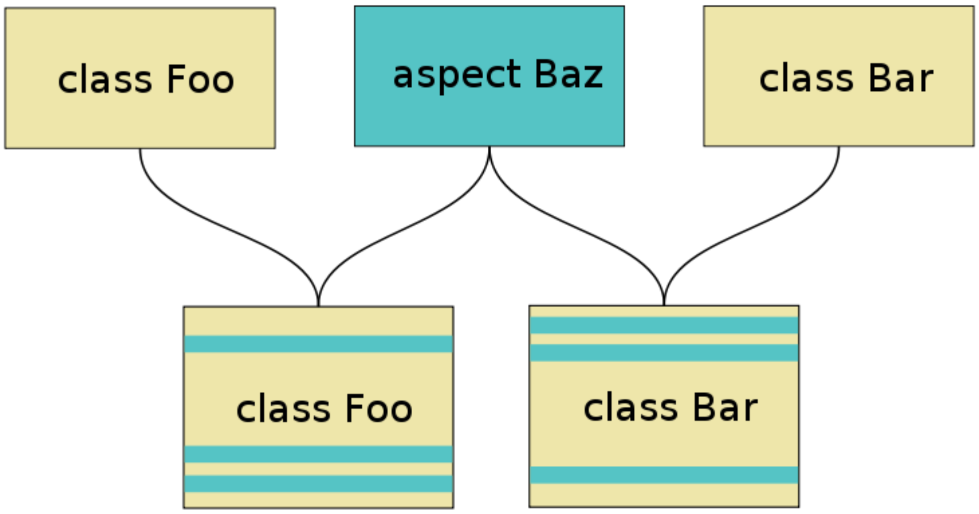
The HP project Dynamo was an experimental JIT compiler where the bytecode format and the machine code format were of the same type; the system turned HPA-8000 bytecode into HPA-8000 machine code. Counterintuitively, this resulted in speed ups, in some cases of 30% since doing this permitted optimizations at the machine code level, for example, inlining code for better cache usage and optimizations of calls to dynamic libraries and many other run-time optimizations which conventional compilers are not able to attempt.

Chapter 4

Aspect Weaver and Compiler Description Language

Aspect weaver

Aspect weaver



An aspect weaver takes information from raw classes and aspects and creates new classes with the aspect code appropriately weaved into the classes.

Usage	
Paradigm	Aspect-oriented programming
Language(s)	AspectC++, AspectJ

An **aspect weaver** is a metaprogramming utility for aspect-oriented languages designed to take instructions specified by aspects (isolated representations of a significant concepts in a program) and generate the final implementation code. The weaver integrates aspects into the locations specified by the software as a pre-compilation step. By merging aspects and classes (representations of the structure of entities in the program), the weaver generates either a woven class (which can then be compiled into bytecode) or woven bytecode directly.

Aspect weavers take instructions known as *advice* specified through the use of pointcuts and join points, special segments of code that indicate what methods should be handled by aspect code. The implementation of the aspect then specifies whether the related code should be added before, after, or throughout the related methods. By doing this, aspect weavers improve modularity, keeping code in one place that would otherwise have been interspersed throughout various, unrelated classes.

Motivation

Many programming languages are already widely accepted and understood. However, the desire to create radically different programming languages to support the aspect-oriented programming paradigm is not significant due to business-related concerns; there are risks associated with adopting new technologies. Use of an entirely new language relies on a business's ability to acquire new developers. Additionally, the existing code base of a business would need to be discarded. Finally, a business would need to acquire a new toolchain (suite of tools) for development, which is often both an expense in both money and time. Primary concerns about roadmaps for the adoption of new technologies tend to be the need to train new developers and adapt existing processes to the new technology.

To address these business concerns, an aspect weaver enables the use of widely adopted languages like Java with aspect-oriented programming through minor adaptations such as AspectJ which work with existing tools. Instead of developing an entirely new language, the aspect weaver interprets the extensions defined by AspectJ and builds "woven" Java code which can then be used by any existing Java compiler. This ensures that any existing object oriented code will still be valid aspect-oriented code and that development will feel like a natural extension of the object-oriented language. The AspectC++ programming language extends C++ through the use of an aspect weaver, offering the additional efficiency over AspectJ that is necessary for embedded systems while still retaining the benefits of aspect-oriented programming.

Implementation

Aspect weavers operate by taking instructions specified by aspects, known as *advice*, and distributing it throughout the various classes in the program automatically. The result of the weaving process is a set of classes with the same names as the original classes but with additional code injected into the classes' functions automatically. The advice specifies the exact location and functionality of the injected code.

Through this weaving process, aspect weavers allow for code which would have otherwise been duplicated across classes. By eliminating this duplication, aspect weavers promote modularity of cross-cutting concerns. Aspects define the implementation code which would have otherwise been duplicated and then use pointcuts and join points to define the advice. During weaving, the aspect weaver uses the pointcuts and join points, known as a *pointcut designator*, to identify the positions in candidate classes at which the implementation should be injected. The implementation is then injected into the classes at the points identified, thus permitting the code to be executed at the appropriate times without relying on manual duplication by the programmer.

```
aspect Logger {
   pointcut method() : execution(* *(..));
   before() : method() {
      System.out.println("Entering " +
         thisJoinPoint.getSignature().toString());
   }
   after() : method() {
      System.out.println("Leaving " +
         thisJoinPoint.getSignature().toString());
   }
}
public class Foo {
   public void bar() {
      System.out.println("Executing Foo.bar()");
   }
   public void baz() {
      System.out.println("Executing Foo.baz()");
   }
}
```

A sample aspect and class defined in the AspectJ programming language

```
public class Foo {
   public void bar() {
      System.out.println("Entering Foo.bar()");
      System.out.println("Executing Foo.bar()");
      System.out.println("Leaving Foo.bar()");
   }
   public void baz() {
      System.out.println("Entering Foo.baz()");
      System.out.println("Executing Foo.baz()");
      System.out.println("Leaving Foo.baz()");
   }
}
```

The woven class that results from executing an aspect weaver on the above sample

Weaving in AspectJ

In the programming language AspectJ, pointcuts, join points, and the modularized code are defined in an aspect block similar to that of Java classes. Classes are defined using Java syntax. The weaving process consists of executing the aspect advice to produce only a set of generated classes that have the aspect implementation code woven into it.

The example at right shows a potential implementation of an aspect which logs the entry and exit of all methods. Without an aspect weaver, this feature would necessitate duplication of code in the class for every method. Instead, the entry and exit code is defined solely within the aspect.

The aspect weaver analyzes the advice specified by the pointcut in the aspect and uses that advice to distribute the implementation code into the defined class. The code differs slightly in each method due to slight variances in requirements for the method (as the method identifier has changed). The aspect weaver determines the appropriate code to generate in each situation as defined by the implementation advice and then injects it into methods matching the specified pointcut.

Weaving to bytecode

Instead of generating a set of woven source code, some AspectJ weavers instead weave the aspects and classes together directly into bytecode, acting both as the aspect weaver and compiler. While it is expected that the performance of aspect weavers which also perform the compilation process will require more computation time due to the weaving process involved, it is also expected that this performance can be improved as aspect weavers' compiler implementations are relatively new, while traditional compilers have been thoroughly tested and optimized.

Run-time weaving

Developments in AspectJ have revealed the potential to incorporate just-in-time compilation into the execution of aspect-oriented code to address performance demands. At run-time, an aspect weaver could translate aspects in a more efficient manner than traditional, static weaving approaches. Using AspectJ on a Java Virtual Machine, dynamic weaving of aspects at run-time has been shown to improve code performance by 26%. While some implementations of just-in-time virtual machines implement this capability through a new virtual machine, some implementations can be designed to use features that already exist in current virtual machines. The requirement of a new virtual machine is contrary to one of the original design goals of AspectJ.

To accomplish just-in-time weaving, a change to the virtual machine that executes the compiled bytecode is necessary. A proposed solution for AspectJ uses a layered approach which builds upon the existing Java Virtual Machine to add support for join point management and callbacks to a *Dynamic Aspect-Oriented Programming Engine*. An alternative implementation uses a weaving engine that uses breakpoints to halt execution at the pointcut, select an appropriate method, embed it into the application, and continue. The use of breakpoints in this manner has been shown to reduce performance due to a very large number of context switches.

Performance

Aspect weavers' performance, as well as the performance of the code that they produce, has been a subject of analysis. It is preferable that the improvement in modularity supplied by aspect weaving does not impact run-time performance. Aspect weavers are able to perform aspect-specific optimizations. While traditional optimizations such as the elimination of unused special variables from aspect code can be done at compile-time, some optimizations can only be performed by the aspect weaver. For example, AspectJ contains two similar but distinct keywords, `thisJoinPoint`, which contains information about this particular instance of woven code, and `thisJoinPointStaticPart`, which contains information common to all instances of code relevant to that set of advice. The optimization of replacing `thisJoinPoint` with the more efficient and static keyword `thisJoinPointStaticPart` can only be done by the aspect weaver. By performing this replacement, the woven program avoids the creation of a join point object on every execution. Studies have shown that the unnecessary creation of join point objects in AspectJ can lead to a performance overhead of 5% at run-time, while performance degradation is only approximately 1% when this object is not created.

Compile-time performance is generally worse in aspect weavers than their traditional compiler counterparts due to the additional work necessary for locating methods which match the specified pointcuts. A study done showed that the AspectJ compiler ajc is about 34% slower than the Sun Microsystems Java 1.3 compiler and about 62% slower than the Java 1.4 compiler.

Compiler Description Language

Compiler Description Language, or CDL, is a Computer language based on affix grammars. It is very similar to Backus–Naur form(BNF) notation. It was designed for the development of compilers. It is very limited in its capabilities and control flow; and intentionally so. The benefits of these limitations are twofold. On the one hand they make possible the sophisticated data and control flow analysis used by the CDL2 optimizers resulting in extremely efficient code. The other benefit is that they foster a highly verbose naming convention. This in turn leads to programs that are to a great extent self-documenting.

The language looks a bit like Prolog (this is not surprising since both languages arose at about the same time out of work on Affix grammars). As opposed to Prolog however, control flow in CDL is deterministically based on success/failure i.e., no other alternatives are tried when the current one fails. This idea is also used in Parsing Expression Grammars.

CDL3 is the third version of the CDL language, significantly different from the previous two versions.

Short Description

The original version, designed by Cornelis H. A. Koster at the University of Nijmegen emerged in 1971 had a rather unusual concept: it had no core. A typical programming language source is translated to machine instructions or canned sequences of those instructions. Those represent the core, the most basic abstractions that the given language supports. Such primitives can be the additions of numbers, copying variables to each other and so on. CDL1 lacks such a core, it is the responsibility of the programmer to provide the primitive operations in a form that can then be turned into machine instructions by means of an assembler or a compiler for a traditional language. The CDL1 language itself has no concept of primitives, no concept of data types apart from the machine word (an abstract unit of storage - not necessarily a real machine word as such). The evaluation rules are rather similar to the Backus–Naur form syntax descriptions; in fact, writing a parser for a language described in BNF is rather simple in CDL1.

Basically, the language consists of rules. A rule can either succeed or fail. A rule consists of alternatives that are sequences of other rule invocations. A rule succeeds if any of its alternatives succeeds; these are tried in sequence. An alternative succeeds if all of its rule invocations succeed. The language provides operators to create evaluation loops without recursion (although this is not strictly necessary in CDL2 as the optimizer achieves the same effect) and some shortcuts to increase the efficiency of the otherwise recursive evaluation but the basic concept is as above. Apart from the obvious application in context-free grammar parsing, CDL is also well suited to control applications, since a lot of control applications are essentially deeply nested if-then rules.

Each CDL1 rule, while being evaluated, can act on data, which is of unspecified type. Ideally the data should not be changed unless the rule is successful (no side effects on failure). This causes problems as although this rule may succeed, the rule invoking it might still fail, in which case the data change should not take effect. It is fairly easy (albeit memory intensive) to assure the above behavior if all the data is dynamically allocated on a stack but it is rather hard when there's static data, which is often the case. The CDL2 compiler is able to flag the possible violations thanks to the requirement that the direction of parameters (input,output,input-output) and the type of rules (can fail: **test**, **predicate**; cannot fail: **function**, **action**; can have side effect: **predicate**, **action**; cannot have side effect: **test**, **function**) must be specified by the programmer.

As the rule evaluation is based on calling simpler and simpler rules, at the bottom there should be some primitive rules that do the actual work. That is where CDL1 is very surprising: it does not have those primitives. You have to provide those rules yourself. If you need addition in your program, you have to create a rule that has two input parameters and one output parameter and the output is set to be the sum of the two inputs by your code. The CDL compiler uses your code as strings (there are conventions how to refer to the input and output variables) and simply emits it as needed. If you describe your adding rule using assembly, then you will need an assembler to translate the CDL compiler's output to machine code. If you describe all the primitive rules (macros in CDL terminology) in Pascal or C, then you need a Pascal or C compiler to run after the CDL compiler. This lack of core primitives can be very painful when you have to write a snipet of code even for the simplest singe-machine-instruction operation but on the other hand it gives you very great flexibility in implementing esoteric abstract primitives acting on exotic abstract objects (the 'machine word' in CDL is more like 'unit of data storage', with no reference to the kind of data stored there). Additionally large projects made use of carefully crafted libraries of primitives. These were then replicated for each target architecture and OS allowing the production of highly efficient code for all.

To get a feel for how the language looks, here is a small code fragment adapted from the CDL2 manual:

```
ACTION quicksort + >from + >to -p -q:
  less+from+to, split+from+to+p+q,
    quicksort+from+q, quicksort+p+to;
  +.

ACTION split + >i + >j + p> + q> -m:
  make+p+i, make+q+j, add+i+j+m, halve+m,
    (again: move up+j+p+m, move down+i+q+m,
       (less+p+q, swap item+p+q, incr+p, decr+q, *again;
        less+p+m, swap item+p+m, incr+p;
        less+m+q, swap item+q+m, decr+q;
        +)).

FUNCTION move up + >j + >p> + >m:
  less+j+p;
  smaller item+m+p;
  incr+p, *.

FUNCTION move down + >i + >q> + >m:
  less+q+j;
  smaller item+q+m;
  decr+q, *.


TEST less+>a+>b:=a"<"b.
FUNCTION make+a>+>b:=a"="b.
FUNCTION add+>a+>b+sum>:=sum"="a"+"b.
FUNCTION halve+>a>:=a"/=2".
FUNCTION incr+>a>:=a"++".
FUNCTION decr+>a>:=a"--".

TEST smaller item+>i+>j:="items["i"]<items["j"]".
ACTION swap items+>i+>j-
t:=t"=items["i"];items["i"]=items["j"];items["j"]="t.
```

The primitive operations are here defined in terms of Java (or C). This is not a complete program; we must define the Java array *items* elsewhere.

CDL2, that appeared in 1976, kept the principles of CDL1 but made the language suitable for large projects. It introduced modules, enforced data-change-only-on-success and extended the capabilities of the language somewhat. The optimizers in the CDL2 compiler and especially in the CDL2 Laboratory (an IDE for CDL2) were world class and not just for their time. One feature of the CDL2 Laboratory optimizer is almost unique: it can perform optimizations across compilation units, i.e., treating the entire program as a single compilation.

CDL3 is a more recent language. It gave up the open-ended feature of the previous CDL versions and it provides primitives to basic arithmetic and storage access. The extremely puritan syntax of the earlier CDL versions (the number of keywords and symbols both run in single digits) have also been relaxed and some basic concepts are now expressed in syntax rather than explicit semantics. In addition, data types have been introduced to the language.

A book about the CDL1 / CDL2 language can be found in .

The description of CDL3 can be found in .

Programs Developed

The commercial MBP Cobol (a Cobol compiler for the PC) as well as the MProlog system (an industrial strength Prolog implementation that ran on numerous architectures (IBM mainframe, VAX, PDP-11, Intel 8086, etc) and OS-s (DOS/OS/CMS/BS2000, VMS/Unix, DOS/Windows/OS2)). The latter is in particular a testament to CDL2-s portability.

While most programs written with CDL have been compilers, there is at least one commercial GUI application that was developed and maintained in CDL. This application was a dental image acquisition application now owned by DEXIS. A dental office management system was also once developed in CDL.

Chapter 5

Context-free Grammar

In formal language theory, a **context-free grammar** (**CFG**), sometimes also called a **phrase structure grammar**, is a grammar that naturally generates a formal language in which clauses can be nested inside clauses arbitrarily deeply, but where grammatical structures are not allowed to overlap. *CFG* are expressed by Backus–Naur Form, or *BNF*. In terms of production rules, every production of a context free grammar is of the form

$$V \to w$$

where V is a single nonterminal symbol, and w is a string of terminals and/or nonterminals (w can be empty). These rewriting rules applied successively produce a parse tree, where the nonterminal symbols are nodes, the leaves are the terminal symbols, and each node expands by the production into the next level of the tree. The tree describes the nesting structure of the expression. V can therefore change while w is fixed (can't change).

In a context free grammar the left hand side of a production rule is always a single nonterminal symbol. In a general grammar, it could be a string of terminal and/or nonterminal symbols. The grammars are called *context free* because – since all rules only have a nonterminal on the left hand side – one can always replace that nonterminal symbol with what is on the right hand side of the rule. The *context* in which the symbol occurs is therefore not important.

Context-free languages are exactly those which can be understood by a finite state computer with a single infinite stack. In order to keep track of nested units, one pushes the current parsing state at the start of the unit, and one recovers it at the end.

Context-free grammars play a central role in the description and design of programming languages and compilers. They are also used for analyzing the syntax of natural languages. Noam Chomsky has posited that all human languages are based on context-free grammars at their core, with additional processes that can manipulate the output of the context-free component (the transformations of early Chomskyan theory).

Background

Since the time of Pāṇini, at least, linguists have described the grammars of languages in terms of their block structure, and described how sentences are recursively built up from smaller phrases, and eventually individual words or word elements.

An essential property of these block structures is that logical units never overlap. For example, the sentence: John, whose blue car was in the garage, walked to the green store.

can be logically parenthesized as follows:

(John, ((whose blue car) (was (in the garage)))), (walked (to (the green store))).

Natural human languages rarely allow overlapping constructions, such as:

[John saw (a blue car in an ad yesterday] with bright yellow headlights).

It is hard to find cases of possible overlap and many such cases can be explained in an alternative way that doesn't assume overlap.

The formalism of context-free grammars was developed in the mid-1950s by Noam Chomsky, and also their classification as a special type of formal grammar (which he called phrase-structure grammars).

A context-free grammar provides a simple and precise mechanism for describing the methods by which phrases in some natural language are built from smaller blocks, capturing the "block structure" of sentences in a natural way. Its simplicity makes the formalism amenable to rigorous mathematical study. Important features of natural language syntax such as agreement and reference are not part of the context free grammar, but the basic recursive structure of sentences, the way in which clauses nest inside other clauses, and the way in which lists of adjectives and adverbs are swallowed by nouns and verbs, is described exactly.

Block structure was introduced into computer programming languages by the Algol project, which, as a consequence, also featured a context-free grammar to describe the resulting Algol syntax. This became a standard feature of computer languages, and the notation for grammars used in concrete descriptions of computer languages came to be known as Backus-Naur Form, after two members of the Algol language design committee.

The "block structure" aspect that context-free grammars capture is so fundamental to grammar that the terms syntax and grammar are often identified with context-free grammar rules, especially in computer science. Formal constraints not captured by the grammar are then considered to be part of the "semantics" of the language.

Context-free grammars are simple enough to allow the construction of efficient parsing algorithms which, for a given string, determine whether and how it can be generated from the grammar. An Earley parser is an example of such an algorithm, while the widely used LR and LL parsers are more efficient algorithms that deal only with more restrictive subsets of context-free grammars.

Formal definitions

A context-free grammar *G* is defined by the 4-tuple:

$G = (V\,, \Sigma\,, R\,, S\,)$where

1. $V$is a finite set of *non-terminal* characters or variables. They represent different types of phrase or clause in the sentence. They are sometimes called syntactic categories. Each variable represents a language.

2. $\Sigma$is a finite set of *terminal*s, disjoint from V, which make up the actual content of the sentence.The set of terminals is the alphabet of the language defined by the grammar.

3. $R$is a relation from $V$to $(V \cup \Sigma)^*$such that $\exists\, w \in (V \cup \Sigma)^* : (S, w) \in R$. These relations are called productions or rewrite rules.

4. $S$is the start variable (or start symbol), used to represent the whole sentence (or program). It must be an element of V.

$R$is a finite set. The members of $R$are called the *rule*s or *production*s of the grammar. The asterisk represents the Kleene star operation.

Rule application

For any strings $u, v \in (V \cup \Sigma)^*$, we say $u$yields v, written as $u \Rightarrow v$, if $\exists(\alpha, \beta) \in R$and $u_1, u_2 \in (V \cup \Sigma)^*$such that $u \ = u_1 \alpha u_2$and $v \ = u_1 \beta u_2$. Thus, $v$is the result of applying the rule $(\alpha, \beta)$to u.

Repetitive rule application

For any $u, v \in (V \cup \Sigma)^*, u \overset{*}{\Rightarrow} v$(or $u \Rightarrow\Rightarrow\ v$in some textbooks), if $\exists u_1, u_2, \cdots u_k \in (V \cup \Sigma)^*, k \geq 0$such that $u \Rightarrow\ u_1 \Rightarrow\ u_2 \cdots \Rightarrow\ u_k \Rightarrow\ v$

Context-free language

The language of a grammar $G = (V\,, \Sigma\,, R\,, S\,)$is the set

$$L(G) = \{w \in \Sigma^* : S \overset{*}{\Rightarrow} w\}$$

A language $L$is said to be a context-free language (CFL), if there exists a CFG G, such that $L = L(G)$.

Proper CFGs

A context-free grammar is said to be *proper*, if it has

- no *inaccessible* symbols: $\forall N \in V : \exists \alpha, \beta \in V^* : S \overset{*}{\Rightarrow} \alpha N \beta$
- no *improductive* symbols: $\forall N \in V : \exists w \in \Sigma^* : N \overset{*}{\Rightarrow} w$
- no ε-productions: $\forall N \in V, w \in \Sigma^* : (N, w) \in R \Rightarrow w \neq$ ε
- no cycles: $\neg \exists N \in V : N \overset{*}{\Rightarrow} N$

Examples

Well-formed parentheses

The canonical example of a context free grammar is parenthesis matching, which is representative of the general case. There are two terminal symbols "(" and ")" and one nonterminal symbol S. The production rules are

S → SS
S → (S)
S → ()

The first rule allows Ss to multiply; the second rule allows Ss to become enclosed by matching parentheses; and the third rule terminates the recursion.

Starting with S, and applying the rules, one can construct:

S → SS → SSS → (S)SS → ((S))SS → ((SS))S(S)
→ ((()S))S(S) → ((()()))S(S) → ((()()))()(S)
→ ((()()))()(())

Well-formed nested parentheses and square brackets

A second canonical example is two different kinds of matching nested parentheses, described by the productions:

S → SS
S → ()
S → (S)
S → []

S → [S]

with terminal symbols [ ] ( ) and nonterminal S.

The following sequence can be derived in that grammar:

([[[()() [ ][ ]]]([ ])])

However, there is no context-free grammar for generating all sequences of two different types of parentheses, each separately balanced disregarding the other, but where the two types need not nest inside one another, for example:

[[[[(((( ]]]]))))(([ ))(([ ))([ )( ])( ])( ])

A regular grammar

S → a
S → aS
S → bS

The terminals here are *a* and *b*, while the only non-terminal is S. The language described is all nonempty strings of *a*s and *b*s that end in *a*.

This grammar is regular: no rule has more than one nonterminal in its right-hand side, and each of these nonterminals is at the same end of the right-hand side.

Every regular grammar corresponds directly to a nondeterministic finite automaton, so we know that this is a regular language.

It is common to list all right-hand sides for the same left-hand side on the same line, using | to separate them, like this:

S → a | aS | bS

So that this is the same grammar described in a terser way.

Matching pairs

In a context-free grammar, we can pair up characters the way we do with brackets. The simplest example:

S → aSb
S → ab

This grammar generates the language $\{a^n b^n : n \geq 1\}$, which is not regular (according to Pumping Lemma for regular languages).

The special character ε stands for the empty string. By changing the above grammar to

S → aSb | ε

we obtain a grammar generating the language $\{a^n b^n : n \geq 0\}$ instead. This differs only in that it contains the empty string while the original grammar did not.

Algebraic expressions

Here is a context-free grammar for syntactically correct infix algebraic expressions in the variables x, y and z:

1. S → x
2. S → y
3. S → z
4. S → S + S
5. S → S - S
6. S → S * S
7. S → S / S
8. S → (S)

This grammar can, for example, generate the string

(x + y) * x - z * y / (x + x)

as follows:

S (the start symbol)
→ S - S (by rule 5)
→ S * S - S (by rule 6, applied to the leftmost S)
→ S * S - S / S (by rule 7, applied to the rightmost S)
→ (S) * S - S / S (by rule 8, applied to the leftmost S)
→ (S) * S - S / (S) (by rule 8, applied to the rightmost S)
→ (S + S) * S - S / (S) (etc.)
→ (S + S) * S - S * S / (S)
→ (S + S) * S - S * S / (S + S)
→ (x + S) * S - S * S / (S + S)
→ (x + y) * S - S * S / (S + S)
→ (x + y) * x - S * y / (S + S)
→ (x + y) * x - S * y / (x + S)
→ (x + y) * x - z * y / (x + S)
→ (x + y) * x - z * y / (x + x)

Note that many choices were made underway which `S` was going to be rewritten next. These choices look quite arbitrary. As a matter of fact, they are.

Also, many choices were made on which rule to apply to the selected `S`. These do not look so arbitrary: they usually affect which terminal string comes out at the end.

To see this we can look at the parse tree of this derivation:

```
           S
           |
          /|\
         S - S
        /     \
       /|\    /|\
      S * S  S / S
     /    |  |    \
    /|\   x /|\   /|\
   ( S )   S * S ( S )
    /      |   |    \
   /|\     z   y   /|\
  S + S           S + S
  |   |           |   |
  x   y           x   x
```

Starting at the top, step by step, an S in the tree is expanded, until no more unexpanded `S`es (non-terminals) remain. Picking a different order of expansion will produce a different derivation, but the same parse tree. The parse tree will only change, if we pick a different rule to apply at some position in the tree.

But can a different parse tree still produce the same terminal string, which is `( x + y ) * x - z * y / ( x + x )` in this case? Yes, for this grammar, this is possible. Grammars with this property are called ambiguous.

For example, `x + y * z` can be produced with these two different parse trees:

```
       S               S
       |               |
      /|\             /|\
     S * S           S + S
    /     \         /     \
   /|\     z       x     /|\
  x + y                 y * z
```

However, the *language* described by this grammar is not inherently ambiguous: an alternative, unambiguous grammar can be given for the language, for example:

T → x
T → y
T → z
S → S + T
S → S - T

S → S * T
S → S / T
T → (S)
S → T

(once again picking S as the start symbol).

Further examples

Example 1

A context-free grammar for the language consisting of all strings over {a,b} containing an unequal number of a's and b's:

S → U | V
U → TaU | TaT
V → TbV | TbT
T → aTbT | bTaT | ε

Here, the nonterminal T can generate all strings with the same number of a's as b's, the nonterminal U generates all strings with more a's than b's and the nonterminal V generates all strings with fewer a's than b's.

Example 2

Another example of a non-regular language is $\{b^n a^m b^{2n} : n \geq 0, m \geq 0\}$. It is context-free as it can be generated by the following context-free grammar:

S → bSbb | A
A → aA | ε

Other examples

The formation rules for the terms and formulas of formal logic fit the definition of context-free grammar, except that the set of symbols may be infinite and there may be more than one start symbol.

Context-free grammars are not limited in application to mathematical ("formal") languages. For example, it has been suggested that a class of Tamil poetry called Venpa is described by a context-free grammar.

Derivations and syntax trees

There are two common ways to describe how a given string can be shown to be derivable from the start symbol of a given grammar. The simplest way is to list the set of consecutive derivations, beginning with the start symbol and ending with the string itself along with the rules applied to produce each intermediate step. However, if we introduce a strategy such as, "always replace the left-most nonterminal first"; then, for context-free grammars the list of applied grammar rules is by itself sufficient and is called the *leftmost derivation* of a string. For example, if we take the following grammar:

(1) S → S + S
(2) S → 1
(3) S → a

and the string "1 + 1 + a" then a left derivation of this string is the list [(1), (1), (2), (2), (3)], that is, the following one step derivations:

S → **S** + S → **S + S** + S → **1** + S + S → 1 + **1** + S → 1 + 1 + **a**

Analogously the *rightmost derivation* is defined as the list that we get, if we always replace the rightmost nonterminal first. In this case this could be the list [(1), (3), (1), (2), (2)].

S → S + **S** → S + **a** → **S + S** + a → S + **1** + a → **1** + 1 + a

The distinction between leftmost derivation and rightmost derivation is important because in most parsers the transformation of the input is defined by giving a piece of code for every grammar rule that is executed whenever the rule is applied. Therefore it is important to know whether the parser determines a leftmost or a rightmost derivation because this determines the order in which the pieces of code will be executed.

A derivation also imposes in some sense a hierarchical structure on the string that is derived. For example, if the string "1 + 1 + a" is derived according to the leftmost derivation:

S → S + S (1)
→ S + S + S (1)
→ 1 + S + S (2)
→ 1 + 1 + S (2)
→ 1 + 1 + a (3)

the structure of the string would be:

$\{ \{ \{ 1 \}_S + \{ 1 \}_S \}_S + \{ a \}_S \}_S$

where $\{ \ldots \}_S$ indicates a substring recognized as belonging to S. This hierarchy can also be seen as a tree:

```
   S
  /|\
 / | \
      /   |   \
     S   '+'   S
    /|\        |
   / | \       |
  S '+' S     'a'
  |     |
 '1'   '1'
```

This tree is called a *concrete syntax tree* of the string. In this case the presented leftmost and the rightmost derivations define the same syntax tree; however, there is another (leftmost) derivation of the same string

S → S + S (1)
  → 1 + S (2)
  → 1 + S + S (1)
  → 1 + 1 + S (2)
  → 1 + 1 + a (3)

and this defines the following syntax tree:

```
        S
       /|\
      / | \
     /  |  \
    S  '+'  S
    |      /|\
    |     / | \
   '1'   S '+' S
         |     |
        '1'   'a'
```

If, for certain strings in the language of the grammar, there is more than one parsing tree, then the grammar is said to be an *ambiguous grammar*. Such grammars are usually hard to parse because the parser cannot always decide which grammar rule it has to apply. Usually, ambiguity is a feature of the grammar, not the language, and an unambiguous grammar can be found that generates the same context-free language. However, there are certain languages that can only be generated by ambiguous grammars; such languages are called inherently ambiguous.

Normal forms

Every context-free grammar that does not generate the empty string can be transformed into one in which no rule has the empty string as a product [a rule with ε as a product is called an ε-production]. If it does generate the empty string, it will be necessary to include the rule $S \rightarrow \epsilon$, but there need be no other ε-rule. Every context-free grammar with no ε-production has an equivalent grammar in Chomsky normal form or Greibach normal form. "Equivalent" here means that the two grammars generate the same language.

Because of the especially simple form of production rules in Chomsky Normal Form grammars, this normal form has both theoretical and practical implications. For instance, given a context-free grammar, one can use the Chomsky Normal Form to construct a polynomial-time algorithm that decides whether a given string is in the language represented by that grammar or not (the CYK algorithm).

Undecidable problems

Some questions that are undecidable for wider classes of grammars become decidable for context-free grammars; e.g. the emptiness problem (whether the grammar generates any terminal strings at all), is undecidable for context-sensitive grammars, but decidable for context-free grammars.

Still, many problems remain undecidable. Examples:

Universality

Given a CFG, does it generate the language of all strings over the alphabet of terminal symbols used in its rules?

A reduction can be demonstrated to this problem from the well-known undecidable problem of determining whether a Turing machine accepts a particular input (the Halting problem). The reduction uses the concept of a *computation history*, a string describing an entire computation of a Turing machine. We can construct a CFG that generates all strings that are not accepting computation histories for a particular Turing machine on a particular input, and thus it will accept all strings only, if the machine doesn't accept that input.

Language equality

Given two CFGs, do they generate the same language?

The undecidability of this problem is a direct consequence of the previous: we cannot even decide whether a CFG is equivalent to the trivial CFG defining the language of all strings.

Language inclusion

Given two CFGs, can the first generate all strings that the second can generate?

Being in a lower level of the Chomsky hierarchy

Given a context-sensitive grammar, does it describe a context-free language? Given a context-free grammar, does it describe a regular language?

Each of these problems is undecidable.

Extensions

An obvious way to extend the context-free grammar formalism is to allow nonterminals to have arguments, the values of which are passed along within the rules. This allows natural language features such as agreement and reference, and programming language analogs such as the correct use and definition of identifiers, to be expressed in a natural way. E.g. we can now easily express that in English sentences, the subject and verb must agree in number.

In computer science, examples of this approach include affix grammars, attribute grammars, indexed grammars, and Van Wijngaarden two-level grammars.

Similar extensions exist in linguistics.

Another extension is to allow additional terminal symbols to appear at the left hand side of rules, constraining their application. This produces the formalism of context-sensitive grammars.

Restrictions

There are a number of important subclasses of the context free grammars:

- Deterministic grammars
- LR(*k*), Simple LR, and Look-Ahead LR grammars (based on LR, Generalized LR supports the full set of CFGs)
- LL(*k*) and LL(*) grammars

These classes are important in parsing: they allow string recognition to proceed deterministically, e.g. without backtracking.

- Simple grammars

This subclass of the LL(1) grammars is mostly interesting for its theoretical property that language equality of simple grammars is decidable, while language inclusion is not.

- Bracketed grammars

These have the property that the terminal symbols are divided into left and right bracket pairs that always match up in rules.

- Linear grammars

In linear grammars every right hand side of a rule has at most one nonterminal.

- Regular grammars

This subclass of the linear grammars describes the regular languages, i.e. they correspond to finite automata and regular expressions.

Chapter 6

Interpreter (Computing)

In computer science, an **interpreter** normally means a computer program that executes, i.e. *performs*, instructions written in a programming language. An *interpreter* may be a program that either

1. executes the source code directly
2. translates source code into some efficient intermediate representation (code) and immediately executes this
3. explicitly executes stored precompiled code made by a compiler which is part of the interpreter system

Perl, Python, MATLAB, and Ruby are examples of type 2, while UCSD Pascal and Java are type 3: Source programs are compiled ahead of time and stored as machine independent code, which is then linked at run-time and executed by an interpreter and/or compiler (for JIT systems). Some systems, such as Smalltalk, BASIC and others, may also combine 2 and 3.

While interpreting and compiling are the two main means by which programming languages are implemented, these are not fully distinct categories, one of the reasons being that most interpreting systems also perform some translation work, just like compilers. The terms "interpreted language" or "compiled language" merely mean that the canonical implementation of that language is an interpreter or a compiler; a high level language is basically an abstraction which is (ideally) independent of particular implementations.

Bytecode interpreters

There is a spectrum of possibilities between interpreting and compiling, depending on the amount of analysis performed before the program is executed. For example, Emacs Lisp is compiled to bytecode, which is a highly compressed and optimized representation of the Lisp source, but is not machine code (and therefore not tied to any particular hardware). This "compiled" code is then interpreted by a bytecode interpreter (itself written in C). The compiled code in this case is machine code for a virtual machine, which is implemented not in hardware, but in the bytecode interpreter. The same approach is used with the Forth code used in Open Firmware systems: the source language is compiled into "F code" (a bytecode), which is then interpreted by a virtual machine.

Control tables - that do not necessarily ever need to pass through a compiling phase - dictate appropriate algorithmic control flow via customized interpreters in similar fashion to bytecode interpreters.

Efficiency

The main disadvantage of interpreters is that when a program is interpreted, it typically runs more slowly than if it had been compiled. The difference in speeds could be tiny or great; often an order of magnitude and sometimes more. It generally takes longer to run a program under an interpreter than to run the compiled code but it can take less time to interpret it than the total time required to compile and run it. This is especially important when prototyping and testing code when an edit-interpret-debug cycle can often be much shorter than an edit-compile-run-debug cycle.

Interpreting code is slower than running the compiled code because the interpreter must analyze each statement in the program each time it is executed and then perform the desired action, whereas the compiled code just performs the action within a fixed context determined by the compilation. This run-time analysis is known as "interpretive overhead". Access to variables is also slower in an interpreter because the mapping of identifiers to storage locations must be done repeatedly at run-time rather than at compile time.

There are various compromises between the development speed when using an interpreter and the execution speed when using a compiler. Some systems (such as some Lisps) allow interpreted and compiled code to call each other and to share variables. This means that once a routine has been tested and debugged under the interpreter it can be compiled and thus benefit from faster execution while other routines are being developed. Many interpreters do not execute the source code as it stands but convert it into some more compact internal form. Many BASIC interpreters replace keywords with single byte tokens which can be used to find the instruction in a jump table. Others achieve even higher levels of program compaction by using a bit-orientated rather than a byte-orientated program memory structure, where commands tokens may occupy perhaps 5 bits of a 16 bit word, leaving 11 bits for their label or address operands.

An interpreter might well use the same lexical analyzer and parser as the compiler and then interpret the resulting abstract syntax tree.

Advantages and disadvantages of using interpreters

Programmers usually write programs in high level code, which the CPU cannot execute; so this source code has to be converted into machine code. This conversion is done by a compiler or an interpreter. A compiler makes the conversion just once, while an interpreter typically converts it every time a program is executed (or in some languages like early versions of BASIC, every time a single instruction is executed).

Development cycle

During program development, the programmer makes frequent changes to source code. A compiler needs to make a compilation of the altered source files, and link the whole binary code before the program can be executed. An interpreter usually just needs to translate to an intermediate representation or not translate at all, thus requiring less time before the changes can be tested.

Distribution

A compiler converts source code into binary instruction for a specific processor's architecture, thus making it less portable. This conversion is made just once, on the developer's environment, and after that the same binary can be distributed to the user's machines where it can be executed without further translation.

An interpreted program can be distributed as source code. It needs to be translated in each final machine, which takes more time but makes the program distribution independent to the machine's architecture.

Execution environment

An interpreter will make source translations during runtime. This means every line has to be converted each time the program runs. This process slows down the program execution and is a major disadvantage of interpreters over compilers. Another main disadvantage of interpreter is that it must be present on the machine as additional software to run the program.

Abstract Syntax Tree interpreters

In the spectrum between interpreting and compiling, another approach is transforming the source code into an optimized Abstract Syntax Tree (AST) then executing the program following this tree structure, or using it to generate native code Just-In-Time. In this approach, each sentence needs to be parsed just once. As an advantage over bytecode, the AST keeps the global program structure and relations between statements (which is lost in a bytecode representation), and when compressed provides a more compact representation. Thus, using AST has been proposed as a better intermediate format for Just-in-time compilers than bytecode. Also, it allows to perform better analysis during runtime.

However, for interpreters, an AST causes more overhead than a bytecode interpreter, because of nodes related to syntax performing no useful work, of a less sequential representation (requiring traversal of more pointers) and of overhead visiting the tree.

Just-in-time compilation

Further blurring the distinction between interpreters, byte-code interpreters and compilation is just-in-time compilation (or JIT), a technique in which the intermediate representation is compiled to native machine code at runtime. This confers the efficiency of running native code, at the cost of startup time and increased memory use when the bytecode or AST is first compiled. Adaptive optimization is a complementary technique in which the interpreter profiles the running program and compiles its most frequently-executed parts into native code. Both techniques are a few decades old, appearing in languages such as Smalltalk in the 1980s.

Just-in-time compilation has gained mainstream attention amongst language implementers in recent years, with Java, the .NET Framework and most modern JavaScript implementations now including JITs.

Punched card interpreter

The term "interpreter" often referred to a piece of unit record equipment that could read punched cards and print the characters in human-readable form on the card. The IBM 550 Numeric Interpreter and IBM 557 Alphabetic Interpreter are typical examples from 1930 and 1954, respectively.

Another definition of interpreter

Instead of producing a target program as a translation, an interpreter performs the operation implied by the source program. For an assignment statement, for example, an interpreter might build a tree and then carry out the operation at the node as it "walks" the tree.

Interpreters are frequently used to execute command languages, since each operator executed in command language is usually an invocation of a complex routine such as an editor or compiler.

Chapter 7

Lexical Analysis

In computer science, **lexical analysis** is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a **lexical analyzer**, **lexer** or **scanner**. A lexer often exists as a single function which is called by a parser or another function.

Lexical grammar

The specification of a programming language will often include a set of rules which defines the lexer. These rules usually consist of regular expressions and they define the set of possible character sequences that are used to form individual tokens or lexemes.

In programming languages delimiting blocks with tokens (e.g., "{" and "}") as opposed to off-side rule languages delimiting blocks with indentation, whitespace is also defined by a regular expression and influences the recognition of other tokens but does not itself contribute tokens. Whitespace is said to be *non-significant* in such languages.

Token

A **token** is a string of characters, categorized according to the rules as a symbol (e.g., IDENTIFIER, NUMBER, COMMA). The process of forming tokens from an input stream of characters is called **tokenization** and the lexer categorizes them according to a symbol type. A token can look like anything that is useful for processing an input text stream or text file.

A lexical analyzer generally does nothing with combinations of tokens, a task left for a parser. For example, a typical lexical analyzer recognizes parentheses as tokens, but does nothing to ensure that each '(' is matched with a ')'.

Consider this expression in the C programming language:

```
sum=3+2;
```

Tokenized in the following table:

Lexeme	**Token type**
sum	Identifier
=	Assignment operator
3	Number
+	Addition operator
2	Number
;	End of statement

Tokens are frequently defined by regular expressions, which are understood by a lexical analyzer generator such as lex. The lexical analyzer (either generated automatically by a tool like lex, or hand-crafted) reads in a stream of characters, identifies the lexemes in the stream, and categorizes them into tokens. This is called "tokenizing." If the lexer finds an invalid token, it will report an error.

Following tokenizing is parsing. From there, the interpreted data may be loaded into data structures for general use, interpretation, or compiling.

Scanner

The first stage, the **scanner**, is usually based on a finite-state machine (FSM). It has encoded within it information on the possible sequences of characters that can be contained within any of the tokens it handles (individual instances of these character sequences are known as lexemes). For instance, an *integer* token may contain any sequence of numerical digit characters. In many cases, the first non-whitespace character can be used to deduce the kind of token that follows and subsequent input characters are then processed one at a time until reaching a character that is not in the set of characters acceptable for that token (this is known as the maximal munch rule, or longest match rule). In some languages the lexeme creation rules are more complicated and may involve backtracking over previously read characters.

Tokenizer

Tokenization is the process of demarcating and possibly classifying sections of a string of input characters. The resulting tokens are then passed on to some other form of processing. The process can be considered a sub-task of parsing input.

Take, for example,

```
      The quick brown fox jumps over the lazy dog
```

The string isn't implicitly segmented on spaces, as an English speaker would do. The raw input, the 43 characters, must be explicitly split into the 9 tokens with a given space delimiter (i.e. matching the string `" "` or regular expression `/\s{1}/`.

The tokens could be represented in XML,

```
<sentence>
  <word>The</word>
  <word>quick</word>
  <word>brown</word>
  <word>fox</word>
  <word>jumps</word>
  <word>over</word>
  <word>the</word>
  <word>lazy</word>
  <word>dog</word>
</sentence>
```

Or an s-expression,

```
(sentence ((word The) (word quick) (word brown) (word fox) (word jumps)
(word over) (word the) (word lazy) (word dog)))
```

A lexeme, however, is only a string of characters known to be of a certain kind (e.g., a string literal, a sequence of letters). In order to construct a token, the lexical analyzer needs a second stage, the **evaluator**, which goes over the characters of the lexeme to produce a *value*. The lexeme's type combined with its value is what properly constitutes a token, which can be given to a parser. (Some tokens such as parentheses do not really have values, and so the evaluator function for these can return nothing. The evaluators for integers, identifiers, and strings can be considerably more complex. Sometimes evaluators can suppress a lexeme entirely, concealing it from the parser, which is useful for whitespace and comments.)

For example, in the source code of a computer program the string

```
      net_worth_future = (assets - liabilities);
```

might be converted (with whitespace suppressed) into the lexical token stream:

```
NAME "net_worth_future"
EQUALS
OPEN_PARENTHESIS
NAME "assets"
MINUS
NAME "liabilities"
CLOSE_PARENTHESIS
SEMICOLON
```

Though it is possible and sometimes necessary, due to licensing restrictions of existing parsers or if the list of tokens is small, to write a lexer by hand, lexers are often generated by automated tools. These tools generally accept regular expressions that describe the tokens allowed in the input stream. Each regular expression is associated with a production in the lexical grammar of the programming language that evaluates the lexemes matching the regular expression. These tools may generate source code that can be compiled and executed or construct a state table for a finite-state machine (which is plugged into template code for compilation and execution).

Regular expressions compactly represent patterns that the characters in lexemes might follow. For example, for an English-based language, a NAME token might be any English alphabetical character or an underscore, followed by any number of instances of any ASCII alphanumeric character or an underscore. This could be represented compactly by the string `[a-zA-Z_][a-zA-Z_0-9]*`. This means "any character a-z, A-Z or _, followed by 0 or more of a-z, A-Z, _ or 0-9".

Regular expressions and the finite-state machines they generate are not powerful enough to handle recursive patterns, such as "*n* opening parentheses, followed by a statement, followed by *n* closing parentheses." They are not capable of keeping count, and verifying that *n* is the same on both sides — unless you have a finite set of permissible values for *n*. It takes a full-fledged parser to recognize such patterns in their full generality. A parser can push parentheses on a stack and then try to pop them off and see if the stack is empty at the end.

The Lex programming tool and its compiler is designed to generate code for fast lexical analysers based on a formal description of the lexical syntax. It is not generally considered sufficient for applications with a complicated set of lexical rules and severe performance requirements; for instance, the GNU Compiler Collection (gcc) uses hand-written lexers.

Lexer generator

Lexical analysis can often be performed in a single pass if reading is done a character at a time. Single-pass lexers can be generated by tools such as the classic flex.

The lex/flex family of generators uses a table-driven approach which is much less efficient than the directly coded approach. With the latter approach the generator produces an engine that directly jumps to follow-up states via goto statements. Tools like re2c and Quex have proven (e.g. RE2C - A More Versatile Scanner Generator (1994)) to produce engines that are between two to three times faster than flex produced engines. It is in general difficult to hand-write analyzers that perform better than engines generated by these latter tools.

The simple utility of using a scanner generator should not be discounted, especially in the developmental phase, when a language specification might change daily. The ability to express lexical constructs as regular expressions facilitates the description of a lexical analyzer. Some tools offer the specification of pre- and post-conditions which are hard to program by hand. In that case, using a scanner generator may save a lot of development time.

Lexical analyzer generators

- ANTLR - ANTLR generates predicated-LL(k) lexers.
- Flex - Alternative variant of the classic 'lex' (C/C++).
- JFlex - a rewrite of JLex.

- Ragel - A state machine and lexical scanner generator with output support for C, C++, Objective-C, D, Java and Ruby source code.

The following lexical analysers can handle Unicode:

- JLex - A Lexical Analyzer Generator for Java.
- Quex - (or 'Queχ') A Fast Universal Lexical Analyzer Generator for C++.

Chapter 8

Regular Expression

In computing, a **regular expression**, also referred to as **regex** or **regexp**, provides a concise and flexible means for matching strings of text, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification.

The following examples illustrate a few specifications that could be expressed in a regular expression:

- The sequence of characters "car" appearing consecutively in any context, such as in "car", "cartoon", or "bicarbonate"
- The sequence of characters "car" occurring in that order with other characters between them, such as in "Icelander" or "chandler"
- The word "car" when it appears as an isolated word
- The word "car" when preceded by the word "blue" or "red"
- The word "car" when *not* preceded by the word "motor"
- A dollar sign immediately followed by one or more digits, and then optionally a period and exactly two more digits (for example, "$100" or "$245.99").

Regular expressions can be much more complex than these examples.

Regular expressions are used by many text editors, utilities, and programming languages to search and manipulate text based on patterns. Some of these languages, including Perl, Ruby, Awk, and Tcl, have fully integrated regular expressions into the syntax of the core language itself. Others like C, C++, .NET, Java, and Python instead provide access to regular expressions only through libraries. Utilities provided by Unix distributions—including the editor ed and the filter grep—were the first to popularize the concept of regular expressions.

As an example of the syntax, the regular expression `\bex` can be used to search for all instances of the string "*ex*" that occur after "word boundaries" (signified by the `\b`). Thus `\bex` will find the matching string "ex" in two possible locations, (1) at the beginning of words, and (2) between two characters in a string, where one is a word character and the other is not a word character. For instance, in the string "Texts for experts", `\bex` matches the "*ex*" in "experts" but not in "Texts" (because the "*ex*" occurs inside a word and not immediately after a word boundary).

Many modern computing systems provide wildcard characters in matching filenames from a file system. This is a core capability of many command-line shells and is also known as globbing. Wildcards differ from regular expressions in generally expressing only limited forms of patterns.

Basic concepts

A regular expression, often called a pattern, is an expression that describes a set of strings. They are usually used to give a concise description of a set, without having to list all elements. For example, the set containing the three strings "*Handel*", "*Händel*", and "*Haendel*" can be described by the pattern `H(ä|ae?)ndel` (or alternatively, it is said that the pattern *matches* each of the three strings). In most formalisms, if there is any regex that matches a particular set then there is an infinite number of such expressions. Most formalisms provide the following operations to construct regular expressions.

Boolean "or"

A vertical bar separates alternatives. For example, `gray|grey` can match "*gray*" or "*grey*".

Grouping

Parentheses are used to define the scope and precedence of the operators (among other uses). For example, `gray|grey` and `gr(a|e)y` are equivalent patterns which both describe the set of "*gray*" and "*grey*".

Quantification

A quantifier after a token (such as a character) or group specifies how often that preceding element is allowed to occur. The most common quantifiers are the question mark `?`, the asterisk `*` (derived from the Kleene star), and the plus sign `+` (Kleene cross).

- **?** The question mark indicates there is *zero or one* of the preceding element. For example, `colou?r` matches both "*color*" and "*colour*".
- ***** The asterisk indicates there are *zero or more* of the preceding element. For example, `ab*c` matches "*ac*", "*abc*", "*abbc*", "*abbbc*", and so on.
- **+** The plus sign indicates that there is *one or more* of the preceding element. For example, `ab+c` matches "*abc*", "*abbc*", "*abbbc*", and so on, but not "*ac*".

These constructions can be combined to form arbitrarily complex expressions, much like one can construct arithmetical expressions from numbers and the operations +, −, ×, and ÷. For example, `H(ae?|ä)ndel` and `H(a|ae|ä)ndel` are both valid patterns which match the same strings as the earlier example, `H(ä|ae?)ndel`.

The precise syntax for regular expressions varies among tools and with context; more detail is given in the *Syntax* section.

History

The origins of regular expressions lie in automata theory and formal language theory, both of which are part of theoretical computer science. These fields study models of computation (automata) and ways to describe and classify formal languages. In the 1950s, mathematician Stephen Cole Kleene described these models using his mathematical notation called *regular sets*. The SNOBOL language was an early implementation of pattern matching, but not identical to regular expressions. Ken Thompson built Kleene's notation into the editor QED as a means to match patterns in text files. He later added this capability to the Unix editor ed, which eventually led to the popular search tool grep's use of regular expressions ("grep" is a word derived from the command for regular expression searching in the ed editor: `g/re/p` where *re* stands for regular expression). Since that time, many variations of Thompson's original adaptation of regular expressions have been widely used in Unix and Unix-like utilities including expr, AWK, Emacs, vi, and lex.

Perl and Tcl regular expressions were derived from a regex library written by Henry Spencer, though Perl later expanded on Spencer's library to add many new features. Philip Hazel developed PCRE (Perl Compatible Regular Expressions), which attempts to closely mimic Perl's regular expression functionality and is used by many modern tools including PHP and Apache HTTP Server. Part of the effort in the design of Perl 6 is to improve Perl's regular expression integration, and to increase their scope and capabilities to allow the definition of parsing expression grammars. The result is a mini-language called Perl 6 rules, which are used to define Perl 6 grammar as well as provide a tool to programmers in the language. These rules maintain existing features of Perl 5.x regular expressions, but also allow BNF-style definition of a recursive descent parser via sub-rules.

The use of regular expressions in structured information standards for document and database modeling started in the 1960s and expanded in the 1980s when industry standards like ISO SGML (precursored by ANSI "GCA 101-1983") consolidated. The kernel of the structure specification language standards are regular expressions. Simple use is evident in the DTD element group syntax.

Formal language theory

Definition

Regular expressions describe regular languages in formal language theory. They have thus the same expressive power as regular grammars. Regular expressions consist of constants and operators that denote sets of strings and operations over these sets, respectively. The following definition is standard, and found as such in most textbooks on formal language theory. Given a finite alphabet Σ, the following constants are defined:

- (*empty set*) ∅ denoting the set ∅.

- (*empty string*) ε denoting the set containing only the "empty" string, which has no characters at all.
- (*literal character*) `a` in Σ denoting the set containing only the character *a*.

The following operations are defined:

- (*concatenation*) *RS* denoting the set { αβ | α in *R* and β in *S* }. For example {"ab", "c"}{"d", "ef"} = {"abd", "abef", "cd", "cef"}.
- (*alternation*) *R* | *S* denoting the set union of *R* and *S*. For example {"ab", "c"}|{"ab", "d", "ef"} = {"ab", "c", "d", "ef"}.
- (*Kleene star*) *R** denoting the smallest superset of *R* that contains ε and is closed under string concatenation. This is the set of all strings that can be made by concatenating any finite number (including zero) of strings from *R*. For example, {"0","1"}* is the set of all finite binary strings (including the empty string), and {"ab", "c"}* = {ε, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "abcab", ... }.

To avoid parentheses it is assumed that the Kleene star has the highest priority, then concatenation and then set union. If there is no ambiguity then parentheses may be omitted. For example, `(ab)c` can be written as `abc`, and `a|(b(c*))` can be written as `a|bc*`. Many textbooks use the symbols ∪, +, or ∨ for alternation instead of the vertical bar.

Examples:

- `a|b*` denotes {ε, *a*, *b*, *bb*, *bbb*, ...}
- `(a|b)*` denotes the set of all strings with no symbols other than *a* and *b*, including the empty string: {ε, *a*, *b*, *aa*, *ab*, *ba*, *bb*, *aaa*, ...}
- `ab*(c|ε)` denotes the set of strings starting with *a*, then zero or more *b*s and finally optionally a *c*: {*a*, *ac*, *ab*, *abc*, *abb*, *abbc*, ...}

Expressive power and compactness

The formal definition of regular expressions is purposely parsimonious and avoids defining the redundant quantifiers `?` and `+`, which can be expressed as follows: `a+` = `aa*`, and `a?` = `(a|ε)`. Sometimes the complement operator is added, to give a *generalized regular expression*; here R^c matches all strings over Σ* that do not match *R*. In principle, the complement operator is redundant, as it can always be circumscribed by using the other operators. However, the process for computing such a representation is complex, and the result may require expressions of a size that is double exponentially larger.

Regular expressions in this sense can express the regular languages, exactly the class of languages accepted by deterministic finite automata. There is, however, a significant difference in compactness. Some classes of regular languages can only be described by deterministic finite automata whose size grows exponentially in the size of the shortest equivalent regular expressions. The standard example are here the languages L_k consisting of all strings over the alphabet {*a*,*b*} whose k^{th}-last letter equals *a*. On the one hand, a regular expression describing L_4 is given by $(a \mid b)^* a(a \mid b)(a \mid b)(a \mid b)$. Generalizing this pattern to L_k gives the expression

$$(a|b)^*a\underbrace{(a|b)(a|b)\cdots(a|b)}_{k-1 \text{ times}}.$$

On the other hand, it is known that every deterministic finite automaton accepting the language L_k must have at least 2^k states. Luckily, there is a simple mapping from regular expressions to the more general nondeterministic finite automata (NFAs) that does not lead to such a blowup in size; for this reason NFAs are often used as alternative representations of regular languages. NFAs are a simple variation of the type-3 grammars of the Chomsky hierarchy.

Deciding equivalence of regular expressions

As the examples show, different regular expressions can express the same language: the formalism is redundant.

It is possible to write an algorithm which for two given regular expressions decides whether the described languages are essentially equal, reduces each expression to a minimal deterministic finite state machine, and determines whether they are isomorphic (equivalent).

To what extent can this redundancy be eliminated? Kleene star and set union are required to find an interesting subset of regular expressions that is still fully expressive, but perhaps their use can be restricted. This is a surprisingly difficult problem. As simple as the regular expressions are, there is no method to systematically rewrite them to some normal form. The lack of axiom in the past led to the star height problem. Recently, Dexter Kozen axiomatized regular expressions with Kleene algebra.

Syntax

A number of special characters or metacharacters are used to denote actions or delimit groups; but it is possible to force these special characters to be interpreted as normal characters by preceding them with a defined escape character, usually the backslash "\". For example, a dot is normally used as a "wild card" metacharacter to denote any character, but if preceded by a backslash it represents the dot character itself. The pattern `c.t` matches "cat", "cot", "cut", and non-words such as "czt" and "c.t"; but `c\.t` matches only "c.t". The backslash also escapes itself, i.e., two backslashes are interpreted as a literal backslash character.

POSIX

POSIX Basic Regular Expressions

Traditional Unix regular expression syntax followed common conventions but often differed from tool to tool. The IEEE POSIX Basic Regular Expressions (BRE) standard (released alongside an alternative flavor called Extended Regular Expressions or ERE) was designed mostly for backward compatibility with the traditional (Simple Regular Expression) syntax but provided a common standard which has since been adopted as the default syntax of many Unix regular expression tools, though there is often some variation or additional features. Many such tools also provide support for ERE syntax with command line arguments.

In the BRE syntax, most characters are treated as literals — they match only themselves (e.g., `a` matches "*a*"). The exceptions, listed below, are called metacharacters or metasequences.

Metacharacter	Description
`.`	Matches any single character (many applications exclude newlines, and exactly which characters are considered newlines is flavor, character encoding, and platform specific, but it is safe to assume that the line feed character is included). Within POSIX bracket expressions, the dot character matches a literal dot. For example, `a.c` matches "*abc*", etc., but `[a.c]` matches only "*a*", ".", or "*c*".
`[ ]`	A bracket expression. Matches a single character that is contained within the brackets. For example, `[abc]` matches "*a*", "*b*", or "*c*". `[a-z]` specifies a range which matches any lowercase letter from "*a*" to "*z*". These forms can be mixed: `[abcx-z]` matches "*a*", "*b*", "*c*", "*x*", "*y*", or "*z*", as does `[a-cx-z]`. The `-` character is treated as a literal character if it is the last or the first (after the `^`) character within the brackets: `[abc-]`, `[-abc]`. Note that backslash escapes are not allowed. The `]` character can be included in a bracket expression if it is the first (after the `^`) character: `[]abc]`.
`[^ ]`	Matches a single character that is not contained within the brackets. For example, `[^abc]` matches any character other than "*a*", "*b*", or "*c*". `[^a-z]` matches any single character that is not a lowercase letter from "*a*" to "*z*". As above, literal characters and ranges can be mixed.
`^`	Matches the starting position within the string. In line-based tools, it matches the starting position of any line.
`$`	Matches the ending position of the string or the position just before a string-ending newline. In line-based tools, it matches the ending position of any line.

BRE: `\( \)` ERE: `( )`	Defines a marked subexpression. The string matched within the parentheses can be recalled later. A marked subexpression is also called a block or capturing group.
`\`*`n`*	Matches what the *n*th marked subexpression matched, where *n* is a digit from 1 to 9. This construct is theoretically **irregular** and was not adopted in the POSIX ERE syntax. Some tools allow referencing more than nine capturing groups.
`*`	Matches the preceding element zero or more times. For example, `ab*c` matches "*ac*", "*abc*", "*abbbc*", etc. `[xyz]*` matches "", "*x*", "*y*", "*z*", "*zx*", "*zyx*", "*xyzzy*", and so on. `\(ab\)*` matches "", "*ab*", "*abab*", "*ababab*", and so on.
BRE: `\{`*`m`*`,`*`n`*`\}` ERE: `{`*`m`*`,`*`n`*`}`	Matches the preceding element at least *m* and not more than *n* times. For example, `a\{3,5\}` matches only "*aaa*", "*aaaa*", and "*aaaaa*". This is not found in a few older instances of regular expressions.

Examples:

- `.at` matches any three-character string ending with "at", including "*hat*", "*cat*", and "*bat*".
- `[hc]at` matches "*hat*" and "*cat*".
- `[^b]at` matches all strings matched by `.at` except "*bat*".
- `^[hc]at` matches "*hat*" and "*cat*", but only at the beginning of the string or line.
- `[hc]at$` matches "*hat*" and "*cat*", but only at the end of the string or line.
- `\[.\]` matches any single character surrounded by "[" and "]" since the brackets are escaped, for example: "*[a]*" and "*[b]*".

POSIX Extended Regular Expressions

The meaning of metacharacters escaped with a backslash is reversed for some characters in the POSIX Extended Regular Expression (ERE) syntax. With this syntax, a backslash causes the metacharacter to be treated as a literal character. So, for example, `\( \)` is now `( )` and `\{ \}` is now `{ }`. Additionally, support is removed for `\`*`n`* backreferences and the following metacharacters are added:

Metacharacter	Description
`?`	Matches the preceding element zero or one time. For example, `ba?` matches "*b*" or "*ba*".
`+`	Matches the preceding element one or more times. For example, `ba+` matches "*ba*", "*baa*", "*baaa*", and so on.
`|`	The choice (aka alternation or set union) operator matches either the expression before or the expression after the operator. For example, `abc|def` matches "*abc*" or "*def*".

Examples:

- `[hc]+at` matches "*hat*", "*cat*", "*hhat*", "*chat*", "*hcat*", "*ccchat*", and so on, but not "*at*".
- `[hc]?at` matches "*hat*", "*cat*", and "*at*".
- `[hc]*at` matches "*hat*", "*cat*", "*hhat*", "*chat*", "*hcat*", "*ccchat*", "*at*", and so on.
- `cat|dog` matches "*cat*" or "*dog*".

POSIX Extended Regular Expressions can often be used with modern Unix utilities by including the command line flag *-E*.

POSIX character classes

Since many ranges of characters depend on the chosen locale setting (i.e., in some settings letters are organized as *abc...zABC...Z*, while in some others as *aAbBcC...zZ*), the POSIX standard defines some classes or categories of characters as shown in the following table:

POSIX	Non-standard	Perl	ASCII	Description
`[:alnum:]`			`[A-Za-z0-9]`	Alphanumeric characters
	`[:word:]`	`\w`	`[A-Za-z0-9_]`	Alphanumeric characters plus "_"
		`\W`	`[^A-Za-z0-9_]`	Non-word characters
`[:alpha:]`			`[A-Za-z]`	Alphabetic characters
`[:blank:]`			`[ \t]`	Space and tab
		`\b`	`[(?<=\W)(?=\w)|(?<=\w)(?=\W)]`	Word boundaries
`[:cntrl:]`			`[\x00-\x1F\x7F]`	Control characters
`[:digit:]`		`\d`	`[0-9]`	Digits
		`\D`	`[^0-9]`	Non-digits
`[:graph:]`			`[\x21-\x7E]`	Visible characters
`[:lower:]`			`[a-z]`	Lowercase letters
`[:print:]`			`[\x20-\x7E]`	Visible characters and spaces

[:punct:]		[][!"#$%&'()*+,./:;<=>?@\^_`{|}~-]	Punctuation characters
[:space:]	\s	[\t\r\n\v\f]	Whitespace characters
	\S	[^ \t\r\n\v\f]	Non-whitespace characters
[:upper:]		[A-Z]	Uppercase letters
[:xdigit:]		[A-Fa-f0-9]	Hexadecimal digits

POSIX character classes can only be used within bracket expressions. For example, `[[:upper:]ab]` matches the uppercase letters and lowercase "*a*" and "*b*".

In Perl regular expressions, `[:print:]` matches `[:graph:]` union `[:space:]`. An additional non-POSIX class understood by some tools is `[:word:]`, which is usually defined as `[:alnum:]` plus underscore. This reflects the fact that in many programming languages these are the characters that may be used in identifiers. The editor Vim further distinguishes *word* and *word-head* classes (using the notation `\w` and `\h`) since in many programming languages the characters that can begin an identifier are not the same as those that can occur in other positions.

Note that what the POSIX regular expression standards call *character classes* are commonly referred to as *POSIX character classes* in other regular expression flavors which support them. With most other regular expression flavors, the term *character class* is used to describe what POSIX calls *bracket expressions*.

Perl-derivative regular expressions

Perl has a more consistent and richer syntax than the POSIX basic (BRE) and extended (ERE) regular expression standards. An example of its consistency is that `\` always escapes a non-alphanumeric character. Other examples of functionality possible with Perl but not POSIX-compliant regular expressions is the concept of lazy quantification, possessive quantifies to control backtracking, named capture groups, and recursive patterns.

Due largely to its expressive power, many other utilities and programming languages have adopted syntax similar to Perl's — for example, Java, JavaScript, PCRE, Python, Ruby, Microsoft's .NET Framework, and the W3C's XML Schema all use regular expression syntax similar to Perl's. Some languages and tools such as Boost and PHP support multiple regular expression flavors. Perl-derivative regular expression implementations are not identical, and all implement no more than a subset of Perl's features, usually those of Perl 5.0, released in 1994. With Perl 5.10, this process has come full circle with Perl incorporating syntactic extensions originally developed in Python, PCRE, and the .NET Framework.

Simple Regular Expressions

Simple Regular Expressions is a syntax that may be used by historical versions of application programs, and may be supported within some applications for the purpose of providing backward compatibility. It is deprecated.

Lazy quantification

The standard quantifiers in regular expressions are greedy, meaning they match as much as they can, only giving back as necessary to match the remainder of the regex. For example, to find the first instance of an item between < and > symbols in this example:

```
Another whale sighting occurred on <January 26>, <2004>.
```

someone new to regexes would likely come up with the pattern `<.*>` or similar. However, instead of the "<*January 26*>" that might be expected, this pattern will actually return "<*January 26>, <2004*>" because the `*` quantifier is greedy — it will consume as many characters as possible from the input, and "*January 26>, <2004*" has more characters than "*January 26*".

Though this problem can be avoided in a number of ways (e.g., by specifying the text that is *not* to be matched: `<[^>]*>`), modern regular expression tools allow a quantifier to be specified as *lazy* (also known as *non-greedy*, *reluctant*, *minimal*, or *ungreedy*) by putting a question mark after the quantifier (e.g., `<.*?>`), or by using a modifier which reverses the greediness of quantifiers (though changing the meaning of the standard quantifiers can be confusing). By using a lazy quantifier, the expression tries the minimal match first. Though in the previous example lazy matching is used to select one of many matching results, in some cases it can also be used to improve performance when greedy matching would require more backtracking.

Patterns for non-regular languages

Many features found in modern regular expression libraries provide an expressive power that far exceeds the regular languages. For example, many implementations allow grouping subexpressions with parentheses and recalling the value they match in the same expression.

The language of squares is not regular, nor is it context-free. Pattern matching with an unbounded number of back references, as supported by numerous modern tools, is NP-complete (see, Theorem 6.2).

However, many tools, libraries, and engines that provide such constructions still use the term *regular expression* for their patterns. This has led to a nomenclature where the term regular expression has different meanings in formal language theory and pattern matching. For this reason, some people have taken to using the term *regex* or simply *pattern* to describe the latter. Larry Wall, author of the Perl programming language, writes in an essay about the design of Perl 6:

> “ 'Regular expressions' [...] are only marginally related to real regular expressions. Nevertheless, the term has grown with the capabilities of our pattern matching engines, so I'm not going to try to fight linguistic necessity here. I will, however, generally call them "regexes" (or "regexen", when I'm in an Anglo-Saxon mood). ”

Implementations and running times

There are at least three different algorithms that decide if and how a given regular expression matches a string.

The oldest and fastest two rely on a result in formal language theory that allows every nondeterministic finite automaton (NFA) to be transformed into a deterministic finite automaton (DFA). The DFA can be constructed explicitly and then run on the resulting input string one symbol at a time. Constructing the DFA for a regular expression of size m has the time and memory cost of $O(2^m)$, but it can be run on a string of size n in time $O(n)$. An alternative approach is to simulate the NFA directly, essentially building each DFA state on demand and then discarding it at the next step, possibly with caching. This keeps the DFA implicit and avoids the exponential construction cost, but running cost rises to $O(nm)$. The explicit approach is called the DFA algorithm and the implicit approach the NFA algorithm. As both can be seen as different ways of executing the same DFA, they are also often called the DFA algorithm without making a distinction. These algorithms are fast, but using them for recalling grouped subexpressions, lazy quantification, and similar features is tricky.

The third algorithm is to match the pattern against the input string by backtracking. This algorithm is commonly called NFA, but this terminology can be confusing. Its running time can be exponential, which simple implementations exhibit when matching against expressions like `(a|aa)*b` that contain both alternation and unbounded quantification and force the algorithm to consider an exponentially increasing number of sub-cases. This behavior can cause a security problem called Regular expression Denial of Service - ReDoS, which might be used by hackers who want to attack a regular expression engine. More complex implementations will often identify and speed up or abort common cases where they would otherwise run slowly.

Although backtracking implementations only give an exponential guarantee in the worst case, they provide much greater flexibility and expressive power. For example, any implementation which allows the use of backreferences, or implements the various extensions introduced by Perl, must use a backtracking implementation.

Some implementations try to provide the best of both algorithms by first running a fast DFA match to see if the string matches the regular expression at all, and only in that case perform a potentially slower backtracking match.

Unicode

In theoretical terms, any token set can be matched by regular expressions as long as it is pre-defined. In terms of historical implementations, regular expressions were originally written to use ASCII characters as their token set though regular expression libraries have supported numerous other character sets. Many modern regular expression engines offer at least some support for Unicode. In most respects it makes no difference what the character set is, but some issues do arise when extending regular expressions to support Unicode.

- Supported encoding. Some regular expression libraries expect to work on some particular encoding instead of on abstract Unicode characters. Many of these require the UTF-8 encoding, while others might expect UTF-16, or UTF-32. In contrast, Perl and Java are agnostic on encodings, instead operating on decoded characters internally.
- Supported Unicode range. Many regular expression engines support only the Basic Multilingual Plane, that is, the characters which can be encoded with only 16 bits. Currently, only a few regular expression engines (e.g., Perl's and Java's) can handle the full 21-bit Unicode range.
- Extending ASCII-oriented constructs to Unicode. For example, in ASCII-based implementations, character ranges of the form `[x-y]` are valid wherever *x* and *y* are codepoints in the range [0x00,0x7F] and codepoint(x) ≤ codepoint(y). The natural extension of such character ranges to Unicode would simply change the requirement that the endpoints lie in [0x00,0x7F] to the requirement that they lie in [0,0x10FFFF]. However, in practice this is often not the case. Some implementations, such as that of gawk, do not allow character ranges to cross Unicode blocks. A range like [0x61,0x7F] is valid since both endpoints fall within the Basic Latin block, as is [0x0530,0x0560] since both endpoints fall within the Armenian block, but a range like [0x0061,0x0532] is invalid since it includes multiple Unicode blocks. Other engines, such as that of the Vim editor, allow block-crossing but limit the number of characters in a range to 128.
- Case insensitivity. Some case-insensitivity flags affect only the ASCII characters. Other flags affect all characters. Some engines have two different flags, one for ASCII, the other for Unicode. Exactly which characters belong to the POSIX classes also varies.
- Cousins of case insensitivity. As ASCII has case distinction, case insensitivity became a logical feature in text searching. Unicode introduced alphabetic scripts without case like Devanagari. For these, case sensitivity is not applicable. For scripts like Chinese, another distinction seems logical: between traditional and simplified. In Arabic scripts, insensitivity to initial, medial, final, and isolated position may be desired. In Japanese, insensitivity between hiragana and katakana is sometimes useful.

- Normalization. Unicode introduced combining characters. Like old typewriters, plain letters can be followed by one of more non-spacing symbols (usually diacritics like accent marks) to form a single printing character. Consider a letter with both a grave and an acute accent mark. That might be written with the grave appearing before the acute, or vice versa. As a consequence, two different code sequences can result in identical character display.
- New control codes. Unicode introduced amongst others, byte order marks and text direction markers. These codes might have to be dealt with in a special way.
- Introduction of character classes for Unicode blocks, scripts, and numerous other character properties. Block properties are much less useful than script properties, because a block can have code points from several different scripts, and a script can have code points from several different blocks. In Perl and the `java.util.regex` library, properties of the form `\p{InX}` or `\p{Block=X}` match characters in block *X* and `\P{InX}` or `\P{Block=X}` matches code points not in that block. Similarly, `\p{Armenian}`, `\p{IsArmenian}`, or `\p{Script=Armenian}` matches any character in the Armenian script. In general, `\p{X}` matches any character with either the binary propery *X* or the general category *X*. For example, `\p{Lu}`, `\p{Uppercase_Letter}`, or `\p{GC=Lu}` matches any upper-case letter. Binary properties that are *not* general categories include `\p{White_Space}`, `\p{Alphabetic}`, `\p{Math}`, and `\p{Dash}`. Examples of non-binary properties are `\p{Bidi_Class=Right_to_Left}`, `\p{Word_Break=A_Letter}`, and `\p{Numeric_Value=10}`.

Uses

Regular expressions are useful in the production of syntax highlighting systems, data validation, and many other tasks.

While regular expressions would be useful on search engines such as Google, processing them across the entire database could consume excessive computer resources depending on the complexity and design of the regex. Although in many cases system administrators can run regex-based queries internally, most search engines do not offer regex support to the public. Notable exceptions: Google Code Search, Exalead.

Chapter 9

Finite-State Machine & Preprocessor

Finite-State Machine

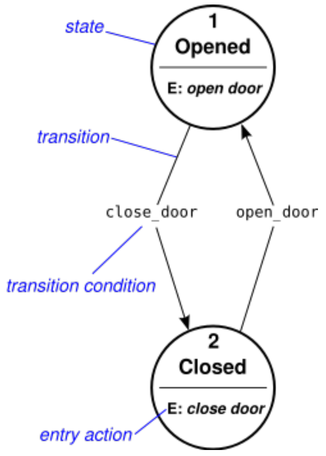


Fig. 1 Example of a simple finite state machine

A **finite-state machine** (**FSM**) or **finite-state automaton** (plural: *automata*), or simply a **state machine**, is a mathematical abstraction sometimes used to design digital logic or computer programs. It is a behavior model composed of a finite number of states, transitions between those states, and actions, similar to a flow graph in which one can inspect the way logic runs when certain conditions are met. It has finite internal memory, an input feature that reads symbols in a sequence, one at a time without going backward; and an output feature, which may be in the form of a user interface, once the model is implemented. The operation of an FSM begins from one of the states (called a *start state*), goes through transitions depending on input to different states and can end in any of those available, however only a certain set of states mark a successful flow of operation (called *accept states*).

Finite-state machines can solve a large number of problems, among which are electronic design automation, communication protocol design, parsing and other engineering applications. In biology and artificial intelligence research, state machines or hierarchies of state machines are sometimes used to describe neurological systems and in linguistics—to describe the grammars of natural languages.

Concepts and vocabulary

A current *state* is determined by past states of the system. As such, it can be said to record information about the past, i.e., it reflects the input changes from the system start to the present moment. The number and names of the states typically depend on the different possible states of the memory, e.g. if the memory is three bits long, there are 8 possible states. A *transition* indicates a state change and is described by a condition that would need to be fulfilled to enable the transition. An *action* is a description of an activity that is to be performed at a given moment. There are several action types:

Entry action
: which is performed *when entering* the state

Exit action
: which is performed *when exiting* the state

Input action
: which is performed depending on present state and input conditions

Transition action
: which is performed when performing a certain transition

An FSM can be represented using a state diagram (or state transition diagram) as in figure 1 above. Besides this, several state transition table types are used. The most common representation is shown below: the combination of current state (e.g. B) and input (e.g. Y) shows the next state (e.g. C). The complete actions information can be added only using footnotes. An FSM definition including the full actions information is possible using state tables.

State transition table

Current state → Input ↓	**State A**	**State B**	**State C**
Input X	...	...	...
Input Y	...	State C	...
Input Z	...	...	...

In addition to their use in modeling reactive systems presented here, finite state automata are significant in many different areas, including electrical engineering, linguistics, computer science, philosophy, biology, mathematics, and logic. Finite state machines are a class of automata studied in automata theory and the theory of computation. In computer science, finite state machines are widely used in modeling of application behavior, design of hardware digital systems, software engineering, compilers, network protocols, and the study of computation and languages.

Classification

There are two different groups of state machines: Acceptors/Recognizers and Transducers.

Acceptors and recognizers

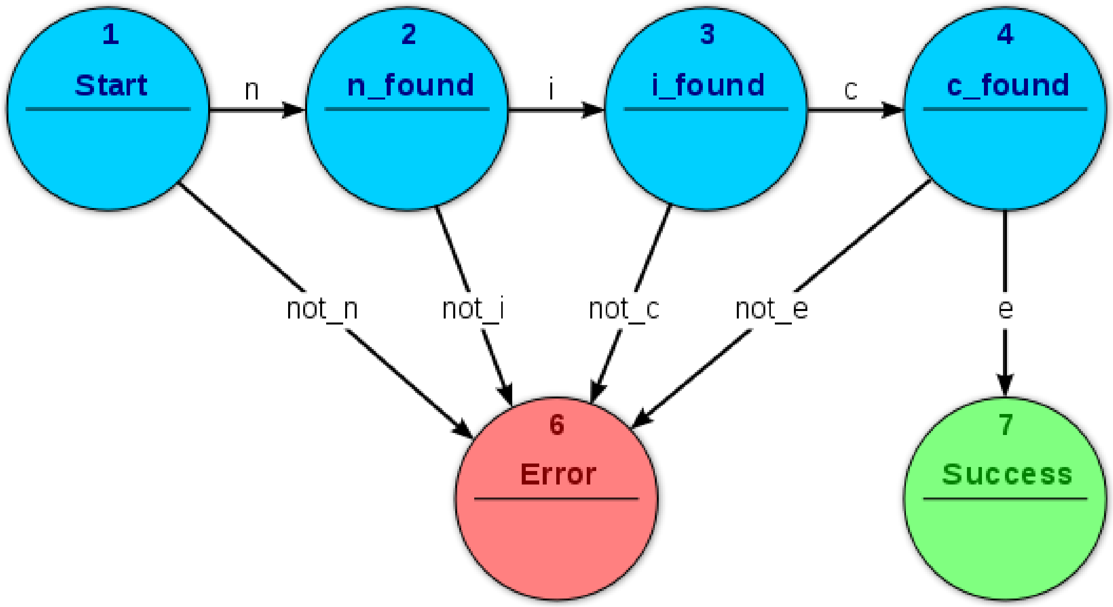


Fig. 2 Acceptor FSM: parsing the word "nice"

Acceptors and **recognizers** (also **sequence detectors**) produce a binary output, saying either *yes* or *no* to answer whether the input is accepted by the machine or not. All states of the FSM are said to be either accepting or not accepting. At the time when all input is processed, if the current state is an accepting state, the input is accepted; otherwise it is rejected. As a rule the input are symbols (characters); actions are not used. The example in figure 2 shows a finite state machine which accepts the word "nice". In this FSM the only accepting state is number 7.

The machine can also be described as defining a language, which would contain every word accepted by the machine but none of the rejected ones; we say then that the language is *accepted* by the machine. By definition, the languages accepted by FSMs are the regular languages—that is, a language is regular if there is some FSM that accepts it.

Start state

The start state is usually shown drawn with an arrow "pointing at it from any where" (Sipser (2006) p. 34).

Accept (or final) states

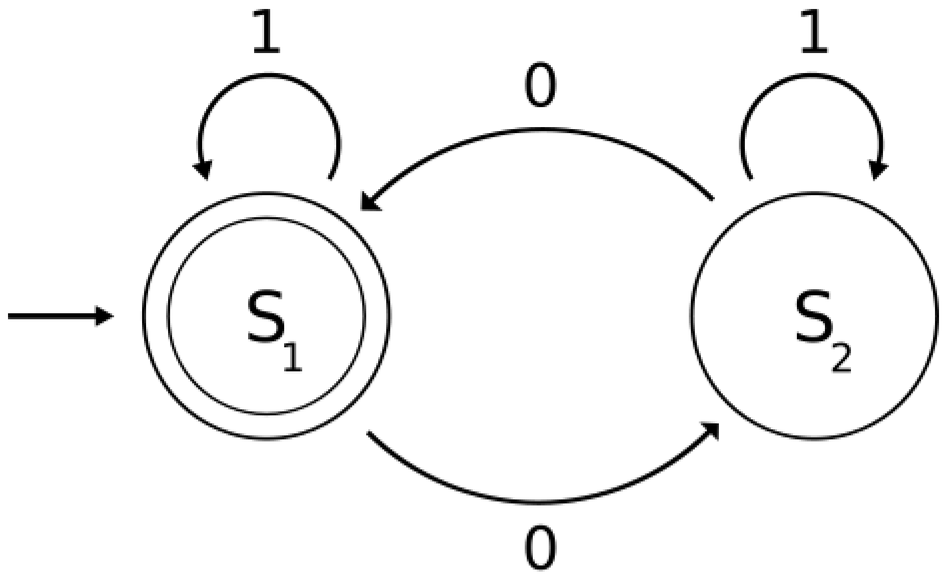


Fig. 3: Representation of a finite-state machine; this example shows one that determines whether a binary number has an odd or even number of 0's, where S_1 is an **accepting state**.

Accept states (also referred to as **accepting** or **final** states) are those at which the machine reports that the input string, as processed so far, is a member of the language it accepts. It is usually represented by a double circle.

An example of an accepting state appears in the diagram to the right: a deterministic finite automaton (DFA) that detects whether the binary input string contains an even number of 0's.

S_1 (which is also the start state) indicates the state at which an even number of 0's has been input. S_1 is therefore an accepting state. This machine will finish in an accept state, if the binary string contains an even number of 0's (including any binary string containing no 0's). Examples of strings accepted by this DFA are epsilon (the empty string), 1, 11, 11..., 00, 010, 1010, 10110, etc...

Transducers

Transducers generate output based on a given input and/or a state using actions. They are used for control applications and in the field of computational linguistics. Here two types are distinguished:

Moore machine

The FSM uses only entry actions, i.e., output depends only on the state. The advantage of the Moore model is a simplification of the behaviour. Consider an elevator door. The state machine recognizes two commands: "command_open" and "command_close" which trigger state changes. The entry action (E:) in state "Opening" starts a motor opening the door, the entry action in state "Closing" starts a motor in the other direction closing the door. States "Opened" and "Closed" stop the motor when fully opened or closed. They signal to the outside world (e.g., to other state machines) the situation: "door is open" or "door is closed".

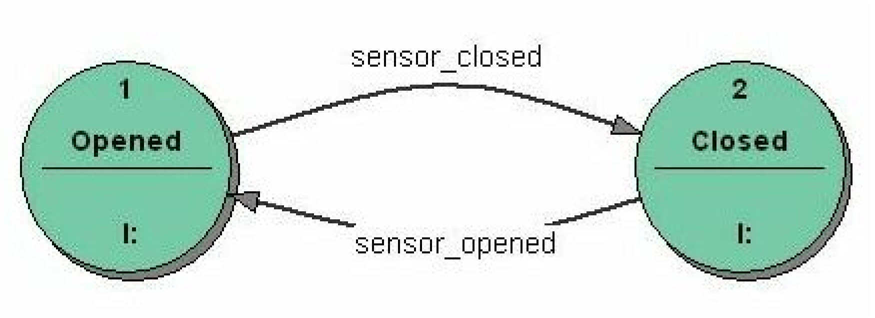


Fig. 4 Transducer FSM: Mealy model example

Mealy machine

The FSM uses only input actions, i.e., output depends on input and state. The use of a Mealy FSM leads often to a reduction of the number of states. The example in figure 4 shows a Mealy FSM implementing the same behaviour as in the Moore example (the behaviour depends on the implemented FSM execution model and will work, e.g., for virtual FSM but not for event driven FSM). There are two input actions (I:): "start motor to close the door if command_close arrives" and "start motor in the other direction to open the door if command_open arrives". The "opening" and "closing" intermediate states are not shown.

In practice mixed models are often used.

More details about the differences and usage of Moore and Mealy models, including an executable example, can be found in the external technical note "Moore or Mealy model?"

Determinism

A further distinction is between **deterministic** (DFA) and **non-deterministic** (NFA, GNFA) automata. In deterministic automata, every state has exactly one transition for each possible input. In non-deterministic automata, an input can lead to one, more than one or no transition for a given state. This distinction is relevant in practice, but not in theory, as there exists an algorithm which can transform any NFA into a more complex DFA with identical functionality.

The FSM with only one state is called a combinatorial FSM and uses only input actions. This concept is useful in cases where a number of FSM are required to work together, and where it is convenient to consider a purely combinatorial part as a form of FSM to suit the design tools.

UML state machines

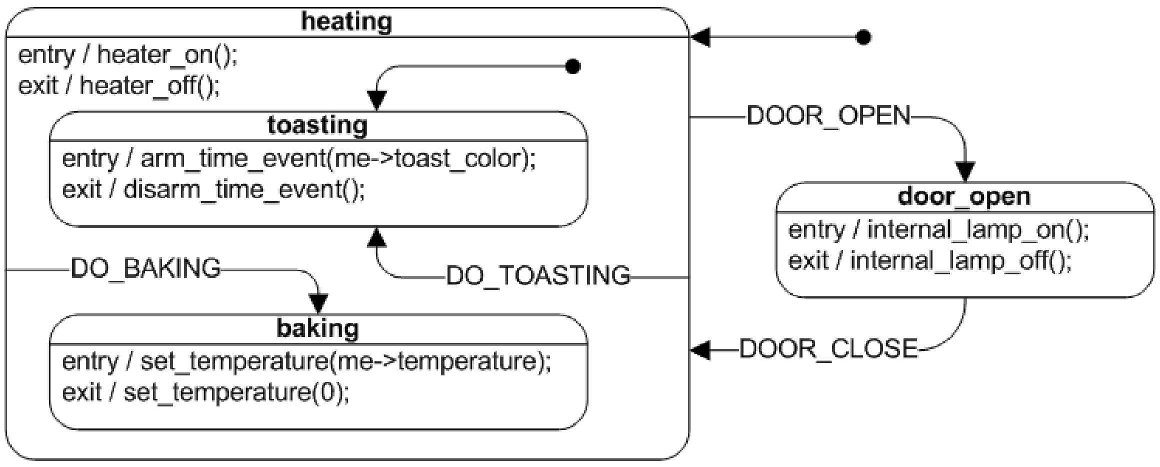

heating
entry / heater_on();
exit / heater_off();
toasting
entry / arm_time_event(me->toast_color);
exit / disarm_time_event();
DO_BAKING
DO_TOASTING
baking
entry / set_temperature(me->temperature);
exit / set_temperature(0);
DOOR_OPEN
door_open
entry / internal_lamp_on();
exit / internal_lamp_off();
DOOR_CLOSE

Fig. 5 UML state machine example (a toaster oven)

The Unified Modeling Language has a very rich semantics and notation for describing state machines. UML state machines overcome the limitations of traditional finite state machines while retaining their main benefits. UML state machines introduce the new concepts of hierarchically nested states and orthogonal regions, while extending the notion of actions. UML state machines have the characteristics of both Mealy machines and Moore machines. They support actions that depend on both the state of the system and the triggering event, as in Mealy machines, as well as entry and exit actions, which are associated with states rather than transitions, as in Moore machines.

Alternative semantics

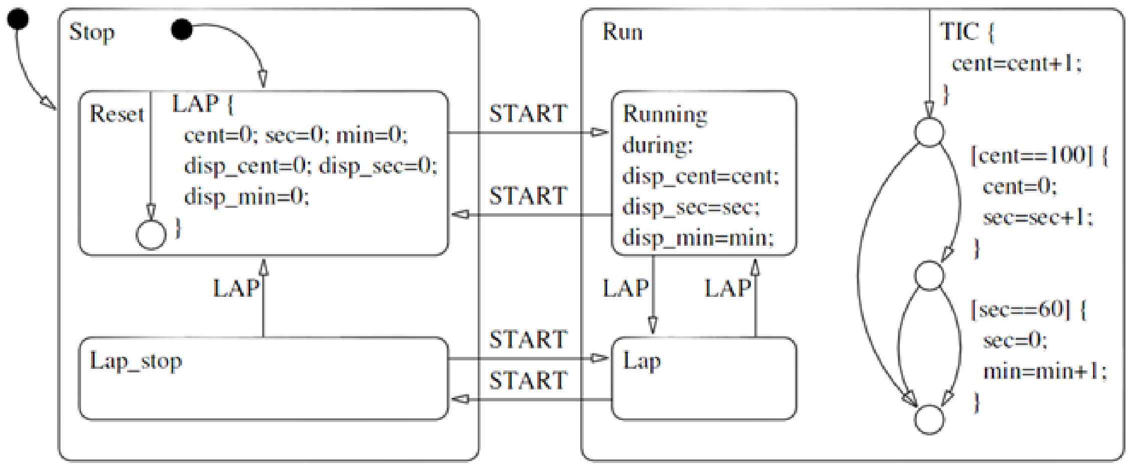


Fig. 6 Model of a simple stopwatch

There are other sets of semantics available to represent state machines. For example, there are tools for modeling and designing logic for embedded controllers. They combine hierarchical state machines, flow graphs, and truth tables into one language, resulting in a different formalism and set of semantics. Figure 6 illustrates this mix of state machines and flow graphs with a set of states to represent the state of a stopwatch and a flow graph to control the ticks of the watch. These charts, like Harel's original state machines, support hierarchically nested states, orthogonal regions, state actions, and transition actions.

FSM logic

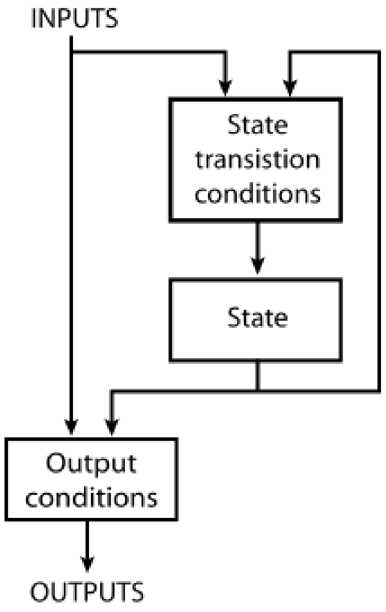


Fig. 7 FSM Logic (Mealy)

The next state and output of an FSM is a function of the input and of the current state. The FSM logic is shown in Figure 7.

Mathematical model

In accordance to the general classification, the following formal definitions are found:

- A *deterministic finite state machine* or *acceptor deterministic finite state machine* is a quintuple (Σ,S,s_0,δ,F), where:
  - Σ is the input alphabet (a finite, non-empty set of symbols).
  - S is a finite, non-empty set of states.
  - s_0 is an initial state, an element of S.
  - δ is the state-transition function: $\delta : S \times \Sigma \rightarrow S$(in a nondeterministic finite state machine it would be $\delta : S \times \Sigma \rightarrow \mathcal{P}(S)$, i.e., δ would return a set of states).
  - F is the set of final states, a (possibly empty) subset of S.

For both deterministic and non-deterministic FSMs, it is conventional to allow δ to be a partial function, i.e. $\delta(q,x)$ does not have to be defined for every combination of $q \in S$and $x \in \Sigma$. If an FSM M is in a state q, the next symbol is x and $\delta(q,x)$ is not defined, then M can announce an error (i.e. reject the input).

- A *finite state transducer* is a sextuple (Σ,Γ,S,s_0,δ,ω), where:
  - Σ is the input alphabet (a finite non empty set of symbols).
  - Γ is the output alphabet (a finite, non-empty set of symbols).
  - S is a finite, non-empty set of states.
  - s_0 is the initial state, an element of S. In a nondeterministic finite state machine, s_0 is a set of initial states.
  - δ is the state-transition function: $\delta : S \times \Sigma \rightarrow S$.
  - ω is the output function.

If the output function is a function of a state and input alphabet ($\omega : S \times \Sigma \rightarrow \Gamma$) that definition corresponds to the **Mealy model**, and can be modelled as a Mealy machine. If the output function depends only on a state ($\omega : S \rightarrow \Gamma$) that definition corresponds to the **Moore model**, and can be modelled as a Moore machine. A finite-state machine with no output function at all is known as a semiautomaton or transition system.

Optimization

Optimizing an FSM means finding the machine with the minimum number of states that performs the same function. The fastest known algorithm doing this is the Hopcroft minimization algorithm. Other techniques include using an implication table, or the Moore reduction procedure. Additionally, acyclic FSAs can be optimized using a simple bottom up algorithm.

Implementation

Hardware applications

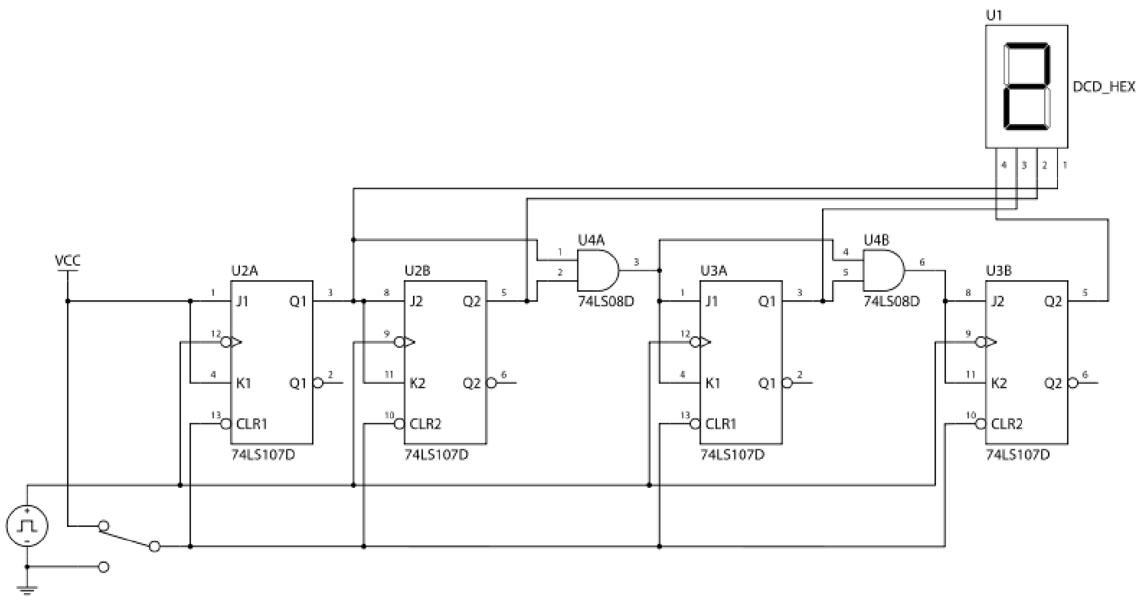


Fig. 8 The circuit diagram for a 4-bit TTL counter, a type of state machine

In a digital circuit, an FSM may be built using a programmable logic device, a programmable logic controller, logic gates and flip flops or relays. More specifically, a hardware implementation requires a register to store state variables, a block of combinational logic which determines the state transition, and a second block of combinational logic that determines the output of an FSM. One of the classic hardware implementations is the Richards controller.

Mealy and Moore machines produce logic with asynchronous output, because there is a propagation delay between the flip-flop and output. This causes slower operating frequencies in FSM. A Mealy or Moore machine can be convertible to a FSM which output is directly from a flip-flop, which makes the FSM run at higher frequencies. This kind of FSM is sometimes called Medvedev FSM. A counter is the simplest form of this kind of FSM.

Software applications

The following concepts are commonly used to build software applications with finite state machines:

- Automata-based programming
- Event driven FSM
- Virtual FSM (VFSM)

Preprocessor

In computer science, a **preprocessor** is a program that processes its input data to produce output that is used as input to another program. The output is said to be a **preprocessed** form of the input data, which is often used by some subsequent programs like compilers. The amount and kind of processing done depends on the nature of the preprocessor; some preprocessors are only capable of performing relatively simple textual substitutions and macro expansions, while others have the power of fully-fledged programming languages.

A common example from computer programming is the processing performed on source code before the next step of compilation. In some computer languages (e.g., C and PL/I) there is a phase of translation known as *preprocessing*.

Lexical preprocessors

Lexical preprocessors are the lowest-level of preprocessors, in so far as they only require lexical analysis, that is, they operate on the source text, prior to any parsing, by performing simple substitution of tokenized character sequences for other tokenized character sequences, according to user-defined rules. They typically perform macro substitution, textual inclusion of other files, and conditional compilation or inclusion.

C preprocessor

The most common example of this is the C preprocessor, which takes lines beginning with '#' as directives. Because it knows nothing about the underlying language, its use has been criticized and many of its features built directly into other languages. For example, macros replaced with aggressive inlining and templates, includes with compile-time imports (this requires the preservation of type information in the object code, making this feature impossible to retrofit into a language); conditional compilation is effectively accomplished with `if-then-else` and dead code elimination in some languages.

Other lexical preprocessors

Other lexical preprocessors include the general-purpose m4, most commonly used in cross-platform build systems such as autoconf, and GEMA, an open source macro processor which operates on patterns of context.

Syntactic preprocessors

Syntactic preprocessors were introduced with the Lisp family of languages. Their role is to transform syntax trees according to a number of user-defined rules. For some programming languages, the rules are written in the same language as the program (compile-time reflection). This is the case with Lisp and OCaml. Some other languages rely on a fully external language to define the transformations, such as the XSLT preprocessor for XML, or its statically typed counterpart CDuce.

Syntactic preprocessors are typically used to customize the syntax of a language, extend a language by adding new primitives, or embed a Domain-Specific Programming Language inside a general purpose language.

Customizing syntax

A good example of syntax customization is the existence of two different syntaxes in the Objective Caml programming language. Programs may be written indifferently using the "normal syntax" or the "revised syntax", and may be pretty-printed with either syntax on demand.

Similarly, a number of programs written in OCaml customize the syntax of the language by the addition of new operators.

Extending a language

The best examples of language extension through macros are found in the Lisp family of languages. While the languages, by themselves, are simple dynamically-typed functional cores, the standard distributions of Scheme or Common Lisp permit imperative or object-oriented programming, as well as static typing. Almost all of these features are implemented by syntactic preprocessing, although it bears noting that the "macro expansion" phase of compilation is handled by the compiler in Lisp. This can still be considered a form of preprocessing, since it takes place before other phases of compilation.

Similarly, statically-checked, type-safe regular expressions or code generation may be added to the syntax and semantics of OCaml through macros, as well as micro-threads (also known as coroutines or fibers), monads or transparent XML manipulation.

Specializing a language

One of the unusual features of the Lisp family of languages is the possibility of using macros to create an internal Domain-Specific Programming Language. Typically, in a large Lisp-based project, a module may be written in a variety of such minilanguages, one perhaps using a SQL-based dialect of Lisp, another written in a dialect specialized for GUIs or pretty-printing, etc. Common Lisp's standard library contains an example of this level of syntactic abstraction in the form of the LOOP macro, which implements an Algol-like minilanguage to describe complex iteration, while still enabling the use of standard Lisp operators.

The MetaOCaml preprocessor/language provides similar features for external Domain-Specific Programming Languages. This preprocessor takes the description of the semantics of a language (i.e. an interpreter) and, by combining compile-time interpretation and code generation, turns that definition into a compiler to the OCaml programming language—and from that language, either to bytecode or to native code.

General purpose preprocessor

Most preprocessors are specific to a particular data processing task (e.g., compiling the C language). A preprocessor may be promoted as being *general purpose*, meaning that it is not aimed at a specific usage or programming language, and is intended to be used for a wide variety of text processing tasks.

M4 is probably the most well known example of such a general purpose preprocessor, although the C preprocessor is sometimes used in a non-C specific role. Examples:

- using C preprocessor for Javascript preprocessing .
- using M4 or C preprocessor as a template engine, to HTML generation.
- imake, a make interface using the C preprocessor, used in the X Window System but now deprecated in favour of automake.
- grompp, a preprocessor for simulation input files for GROMACS (a fast, free, open-source code for some problems in computational chemistry which calls the system C preprocessor (or other preprocessor as determined by the simulation input file) to parse the topology, using mostly the #define and #include mechanisms to determine the effective topology at grompp run time.

Chapter 10

Left Recursion

In computer science, **left recursion** is a special case of recursion.

In terms of context-free grammar, a non-terminal `r` is left-recursive if the left-most symbol in any of `r`'s 'alternatives' either immediately (direct left-recursive) or through some other non-terminal definitions (indirect/hidden left-recursive) rewrites to `r` again.

Definition

"A grammar is left-recursive if we can find some non-terminal A which will eventually derive a sentential form with itself as the left-symbol."

Immediate left recursion

Immediate left recursion occurs in rules of the form

$$A \rightarrow A\alpha \mid \beta$$

where α and β are sequences of nonterminals and terminals, and β doesn't start with *A*. For example, the rule

$$Expr \rightarrow Expr + Term$$

is immediately left-recursive. The *recursive descent parser* for this rule might look like:

```
function Expr()
{
    Expr();  match('+');  Term();
}
```

and a recursive descent parser would fall into infinite recursion when trying to parse a grammar which contains this rule.

Indirect left recursion

Indirect left recursion in its simplest form could be defined as:

$$\begin{array}{l} A \to B\alpha \mid C \\ B \to A\beta \mid D, \end{array}$$

possibly giving the derivation $A \Rightarrow B\alpha \Rightarrow A\beta\alpha \Rightarrow \ldots$

More generally, for the nonterminals $A_0, A_1, \ldots, A_n$, indirect left recursion can be defined as being of the form:

$$\begin{array}{l} A_0 \to A_1\alpha_1 \mid \ldots \\ A_1 \to A_2\alpha_2 \mid \ldots \\ \cdots \\ A_n \to A_0\alpha_{n+1} \mid \ldots \end{array}$$

where $\alpha_1, \alpha_2, \ldots, \alpha_n$are sequences of nonterminals and terminals.

Accommodating left recursion in top-down parsing

A formal grammar that contains left recursion cannot be parsed by a LL(k)-parser or other naive recursive descent parser unless it is converted to a weakly equivalent right-recursive form. In contrast, left recursion is preferred for LALR parsers because it results in lower stack usage than right recursion. However, more sophisticated top-down parsers can implement general context-free grammars by use of curtailment. In 2006, Frost and Hafiz describe an algorithm which accommodates ambiguous grammars with direct left-recursive production rules. That algorithm was extended to a complete parsing algorithm to accommodate indirect as well as direct left-recursion in polynomial time, and to generate compact polynomial-size representations of the potentially-exponential number of parse trees for highly-ambiguous grammars by Frost, Hafiz and Callaghan in 2007. The authors then implemented the algorithm as a set of parser combinators written in the Haskell programming language.

Removing left recursion

Removing immediate left recursion

The general algorithm to remove immediate left recursion follows. Several improvements to this method have been made, including the ones described in "Removing Left Recursion from Context-Free Grammars", written by Robert C. Moore. For each rule of the form

$$A \to A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

where:

- A is a left-recursive nonterminal
- α is a sequence of nonterminals and terminals that is not null ($\alpha \neq \epsilon$)

- β is a sequence of nonterminals and terminals that does not start with A.

replace the A-production by the production:

$$A \rightarrow \beta_1 A' \mid \ldots \mid \beta_m A'$$

And create a new nonterminal

$$A' \rightarrow \epsilon \mid \alpha_1 A' \mid \ldots \mid \alpha_n A'$$

This newly created symbol is often called the "tail", or the "rest".

As an example, consider the rule

$$Expr \rightarrow Expr + Expr \mid Int \mid String$$

This could be rewritten to avoid left recursion as

$$Expr \rightarrow Int\, ExprRest \mid String\, ExprRest$$

$$ExprRest \rightarrow \epsilon \mid + Expr\, ExprRest$$

The last rule happens to be equivalent to the slightly shorter form

$$ExprRest \rightarrow \epsilon \mid + Expr$$

Removing indirect left recursion

If the grammar has no ε-productions (no productions of the form $A \rightarrow \ldots |\epsilon| \ldots$) and is not cyclic (no derivations of the form $A \Rightarrow \ldots \Rightarrow A$for any nonterminal A), this general algorithm may be applied to remove indirect left recursion :

Arrange the nonterminals in some (any) fixed order $A_1, \ldots A_n$.

```
for i = 1 to n {
for j = 1 to i – 1 {
```

- let the current A_j productions be

$$A_j \rightarrow \delta_1 | \ldots | \delta_k$$

- replace each production $A_i \rightarrow A_j \gamma$by

$$A_i \rightarrow \delta_1 \gamma | \ldots | \delta_k \gamma$$

- remove direct left recursion for A_i

}
}

Pitfalls

The above transformations remove left-recursion by creating a right-recursive grammar; but this changes the associativity of our rules. Left recursion makes left associativity; right recursion makes right associativity. Example : We start out with a grammar :

$$Expr \rightarrow Expr + Term \,|\, Term$$

$$Term \rightarrow Term * Factor \,|\, Factor$$

$$Factor \rightarrow (Expr) \,|\, Int$$

After having applied standard transformations to remove left-recursion, we have the following grammar :

$$Expr \rightarrow Term\ Expr'$$

$$Expr' \rightarrow +Term\ Expr' \,|\, \epsilon$$

$$Term \rightarrow Factor\ Term'$$

$$Term' \rightarrow *Factor\ Term' \,|\, \epsilon$$

$$Factor \rightarrow (Expr) \,|\, Int$$

Parsing the string 'a + a + a' with the first grammar in an LALR parser (which can recognize left-recursive grammars) would have resulted in the parse tree:

```
          Expr
        /   |   \
     Expr   +   Term
   /  |  \        \
 Expr  + Term      Factor
  |      |       |
 Term    Factor    Int
  |      |
Factor    Int
  |
 Int
```

This parse tree grows to the left, indicating that the '+' operator is left associative, representing (a + a) + a.

But now that we've changed the grammar, our parse tree looks like this :

```
    Expr ---
   /        \
 Term      Expr' --
  |       /   |    \
Factor   +  Term   Expr' ------
  |           |     |  \       \
 Int       Factor   +  Term   Expr'
              |           |      |
             Int       Factor    ε
                          |
                        Int
```

We can see that the tree grows to the right, representing a + (a + a). We have changed the associativity of our operator '+', it is now right-associative. While this isn't a problem for the associativity of addition, it would have a significantly different value if this were subtraction.

The problem is that normal arithmetic requires left associativity. Several solutions are: (a) rewrite the grammar to be left recursive, or (b) rewrite the grammar with more nonterminals to force the correct precedence/associativity, or (c) if using YACC or Bison, there are *operator declarations,* %left, %right and %nonassoc, which tell the parser generator which associativity to force.

Chapter 11

Parsing

In computer science and linguistics, **parsing**, or, more formally, **syntactic analysis**, is the process of analyzing a text, made of a sequence of tokens (for example, words), to determine its grammatical structure with respect to a given (more or less) formal grammar. Parsing can also be used as a linguistic term, especially in reference to how phrases are divided up in garden path sentences.

Parsing is also an earlier term for the diagramming of sentences of natural languages, and is still used for the diagramming of inflected languages, such as the Romance languages or Latin. The term parsing comes from Latin *pars* (*ōrātiōnis*), meaning part (of speech).

Parsing is a common term used in psycholinguistics when describing language comprehension. In this context, parsing refers to the way that human beings, rather than computers, analyze a sentence or phrase (in spoken language or text) "in terms of grammatical constituents, identifying the parts of speech, syntactic relations, etc." This term is especially common when discussing what linguistic cues help speakers to parse garden-path sentences.

Parser

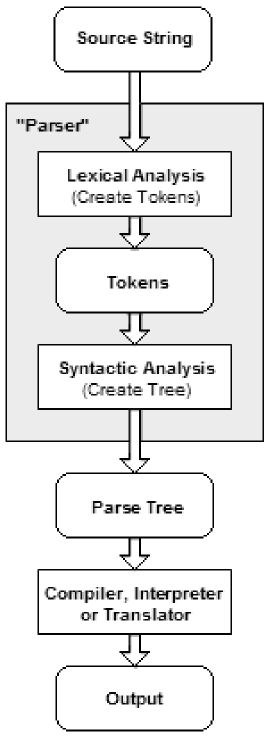
In computing, a **parser** is one of the components in an interpreter or compiler, which checks for correct syntax and builds a data structure (often some kind of parse tree, abstract syntax tree or other hierarchical structure) implicit in the input tokens. The parser often uses a separate lexical analyser to create tokens from the sequence of input characters. Parsers may be programmed by hand or may be (semi-)automatically generated (in some programming languages) by a tool.

Programming languages

The most common use of a parser is as a component of a compiler or interpreter. This parses the source code of a computer programming language to create some form of internal representation. Programming languages tend to be specified in terms of a context-free grammar because fast and efficient parsers can be written for them. Parsers are written by hand or generated by parser generators.

Context-free grammars are limited in the extent to which they can express all of the requirements of a language. Informally, the reason is that the memory of such a language is limited. The grammar cannot remember the presence of a construct over an arbitrarily long input; this is necessary for a language in which, for example, a name must be declared before it may be referenced. More powerful grammars that can express this constraint, however, cannot be parsed efficiently. Thus, it is a common strategy to create a relaxed parser for a context-free grammar which accepts a superset of the desired language constructs (that is, it accepts some invalid constructs); later, the unwanted constructs can be filtered out.

Overview of process



The following example demonstrates the common case of parsing a computer language with two levels of grammar: lexical and syntactic.

The first stage is the token generation, or lexical analysis, by which the input character stream is split into meaningful symbols defined by a grammar of regular expressions. For example, a calculator program would look at an input such as "`12*(3+4)^2`" and split it into the tokens `12`, `*`, `(`, `3`, `+`, `4`, `)`, `^`, and `2`, each of which is a meaningful symbol in the context of an arithmetic expression. The lexer would contain rules to tell it that the characters `*`, `+`, `^`, `(` and `)` mark the start of a new token, so meaningless tokens like "`12*`" or "`(3`" will not be generated.

The next stage is parsing or syntactic analysis, which is checking that the tokens form an allowable expression. This is usually done with reference to a context-free grammar which recursively defines components that can make up an expression and the order in which they must appear. However, not all rules defining programming languages can be expressed by context-free grammars alone, for example type validity and proper declaration of identifiers. These rules can be formally expressed with attribute grammars.

The final phase is semantic parsing or analysis, which is working out the implications of the expression just validated and taking the appropriate action. In the case of a calculator or interpreter, the action is to evaluate the expression or program; a compiler, on the other hand, would generate some kind of code. Attribute grammars can also be used to define these actions.

Types of parser

The *task* of the parser is essentially to determine if and how the input can be derived from the start symbol of the grammar. This can be done in essentially two ways:

- Top-down parsing- Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for parse trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules.
- Bottom-up parsing - A parser can start with the input and attempt to rewrite it to the start symbol. Intuitively, the parser attempts to locate the most basic elements, then the elements containing these, and so on. LR parsers are examples of bottom-up parsers. Another term used for this type of parser is Shift-Reduce parsing.

LL parsers and recursive-descent parser are examples of top-down parsers which cannot accommodate left recursive productions. Although it has been believed that simple implementations of top-down parsing cannot accommodate direct and indirect left-recursion and may require exponential time and space complexity while parsing ambiguous context-free grammars, more sophisticated algorithms for top-down parsing have been created by Frost, Hafiz, and Callaghan which accommodate ambiguity and left recursion in polynomial time and which generate polynomial-size representations of the potentially-exponential number of parse trees. Their algorithm is able to produce both left-most and right-most derivations of an input with regard to a given CFG.

An important distinction with regard to parsers is whether a parser generates a *leftmost derivation* or a *rightmost derivation*. LL parsers will generate a leftmost derivation and LR parsers will generate a rightmost derivation (although usually in reverse).

Examples of parsers

Top-down parsers

Some of the parsers that use top-down parsing include:

- Recursive descent parser

- LL parser (**L**eft-to-right, **L**eftmost derivation)
- Earley parser
- X-SAIGA - eXecutable SpecificAtIons of GrAmmars. Contains publications related to top-down parsing algorithm that supports left-recursion and ambiguity in polynomial time and space.

Bottom-up parsers

Some of the parsers that use bottom-up parsing include:

- Precedence parser
    - Operator-precedence parser
    - Simple precedence parser
- BC (bounded context) parsing
- LR parser (**L**eft-to-right, **R**ightmost derivation)
    - Simple LR (SLR) parser
    - LALR parser
    - Canonical LR (LR(1)) parser
    - GLR parser
- CYK parser

Parser development software

Some of the well known parser development tools include the following.

- ANTLR
- Bison
- Coco/R
- GOLD
- JavaCC
- Lemon
- Lex
- Parboiled
- ParseIT
- Ragel
- Syntax Definition Formalism
- Spirit Parser Framework
- SYNTAX
- Yacc

Chapter 12

Extended Backus–Naur Form & Backus–Naur Form

Extended Backus–Naur Form

In computer science, **Extended Backus–Naur Form (EBNF)** is a family of metasyntax notations used for expressing context-free grammars: that is, a formal way to describe computer programming languages and other formal languages. They are extensions of the basic Backus–Naur Form (BNF) metasyntax notation.

The earliest EBNF was originally developed by Niklaus Wirth. However, many variants of EBNF are in use. The International Organization for Standardization has adopted an EBNF standard (ISO/IEC 14977).

Basics

EBNF is a code that expresses the grammar of a computer language. An EBNF consists of terminal symbol and non-terminal production rules which are the restrictions governing how terminal symbols can be combined into a legal sequence. Examples of terminal symbols include alphanumeric characters, punctuation marks, and white space characters.

The EBNF defines production rules where sequences of symbols are respectively assigned to a nonterminal:

```
digit excluding zero = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" |
"9" ;
digit                = "0" | digit excluding zero ;
```

This production rule defines the nonterminal *digit* which is on the left side of the assignment. The vertical bar represents an alternative and the terminal symbols are enclosed with quotation marks followed by a semicolon as terminating character. Hence a *Digit* is a *0* or a *digit excluding zero* that can be *1* or *2* or *3* and so forth until *9*.

A production rule can also include a sequence of terminals or nonterminals, each separated by a comma:

```
twelve                          = "1" , "2" ;
two hundred one                 = "2" , "0" , "1" ;
three hundred twelve            = "3" , twelve ;
twelve thousand two hundred one = twelve , two hundred one ;
```

Expressions that may be omitted or repeated can be represented through curly braces { ... }:

```
natural number = digit excluding zero , { digit } ;
```

In this case, the strings *1*, *2*, ...,*10*,...,*12345*,... are correct expressions. To represent this, everything that is set within the curly braces may be repeated arbitrarily often, including not at all.

An option can be represented through squared brackets [...]. That is, everything that is set within the square brackets may be present just once, or not at all:

```
integer = "0" | [ "-" ] , natural number ;
```

Therefore an integer is a zero (*0*) or a natural number that may be preceded by an optional minus sign.

EBNF also provides, among other things, the syntax to describe repetitions of a specified number of times, to exclude some part of a production, or to insert comments in an EBNF grammar.

Table of symbols

The following represents a proposed standard.

Usage	Notation
definition	**=**
concatenation	**,**
termination	**;**
alternation	**|**
option	**[...]**
repetition	**{ ... }**
grouping	**(...)**
terminal string	**" ... "**
terminal string	**' ... '**
comment	**(* ... *)**
special sequence	**? ... ?**

exception	-

Advantages of EBNF over BNF

The BNF had the problem that options and repetitions could not be directly expressed. Instead, they needed the use of an intermediate rule or alternative production defined to be either nothing or the optional production for option, or either the repeated production or itself, recursively, for repetition. The same constructs can still be used in EBNF.

The BNF used the symbols (<, >, |, ::=) for itself, but did not include quotes around terminal strings. This prevented these characters from being used in the languages, and required a special symbol for the empty string. In EBNF, terminals are strictly enclosed within quotation marks ("..." or '...'). The angle brackets ("<...>") for nonterminals can be omitted.

BNF syntax can only represent a rule in one line, whereas in EBNF a terminating character, the semicolon, marks the end of a rule.

Furthermore, EBNF includes mechanisms for enhancements, defining the number of repetitions, excluding alternatives, comments, etc.

It should be noted that EBNF is not "more powerful" than the BNF: As a matter of principle, any grammar defined in EBNF can also be represented in BNF (though representations in the latter are generally more protracted).

Conventions

1. The following conventions are used:

- Each meta-identifier of Extended BNF is written as one or more words joined together by hyphens
- A meta-identifier ending with *-symbol* is the name of a terminal symbol of Extended BNF.

2. The normal character representing each operator of Extended BNF and its implied precedence is (highest precedence at the top):

```
* repetition-symbol
- except-symbol
, concatenate-symbol
| definition-separator-symbol
= defining-symbol
; terminator-symbol
```

3. The normal precedence is overridden by the following bracket pairs:

```
'  first-quote-symbol           first-quote-symbol  '
```

```
"  second-quote-symbol          second-quote-symbol  "
(* start-comment-symbol          end-comment-symbol *)
(  start-group-symbol              end-group-symbol  )
[  start-option-symbol            end-option-symbol  ]
{  start-repeat-symbol            end-repeat-symbol  }
?  special-sequence-symbol   special-sequence-symbol ?
```

The first-quote-symbol is the apostrophe as defined by ISO/IEC 646:1991, that is to say Unicode 0x0027 ('); the font used in ISO/IEC 14977:1996(E) renders it very much like the acute, Unicode 0x00B4 (´), so confusion sometimes arises. However, the ISO Extended BNF standard invokes ISO/IEC 646:1991, "ISO 7-bit coded character set for information interchange", as a normative reference and makes no mention of any other character sets, so formally, there is no confusion with Unicode characters outside the 7-bit ASCII range.

As examples, the following syntax rules illustrate the facilities for expressing repetition:

```
aa = "A";
bb = 3 * aa, "B";
cc = 3 * [aa], "C";
dd = {aa}, "D";
ee = aa, {aa}, "E";
ff = 3 * aa, 3 * [aa], "F";
gg = {3 * aa}, "G";
```

Terminal strings defined by these rules are as follows:

```
aa: A
bb: AAAB
cc: C AC AAC AAAC
dd: D AD AAD AAAD AAAAD etc.
ee: AE AAE AAAE AAAAE AAAAAE etc.
ff: AAAF AAAAF AAAAAF AAAAAAF
gg: G AAAG AAAAAAG etc.
```

EBNF extensibility

According to the ISO 14977 standard EBNF is meant to be extensible, and two facilities are mentioned. The first is part of EBNF grammar, the special sequence, which is arbitrary text enclosed with question marks. The interpretation of the text inside a special sequence is beyond the scope of the EBNF standard. For example, the space character could be defined by the following rule:

```
space = ? US-ASCII character 32 ?;
```

The second facility for extension is using the fact that parentheses cannot in EBNF be placed next to identifiers (they must be concatenated with them). The following is valid EBNF:

```
something = foo, ( bar );
```

The following is *not* valid EBNF:

```
something = foo ( bar );
```

Therefore, an extension of EBNF could use that notation. For example, in a Lisp grammar, function application could be defined by the following rule:

```
function application = list( symbol , { expression } );
```

Related work

- The W3C used a different EBNF to specify the XML syntax.
- The British Standards Institution published a standard for an EBNF: BS 6154 in 1981.
- The IETF uses Augmented BNF (ABNF), specified in RFC 5234.

Backus–Naur Form

In computer science, **BNF** (**Backus Normal Form** or **Backus–Naur Form**) is a notation technique for context-free grammars, often used to describe the syntax of languages used in computing, such as computer programming languages, document formats, instruction sets and communication protocols. It is applied wherever exact descriptions of languages are needed, for instance, in official language specifications, in manuals, and in textbooks on programming language theory.

Many extensions and variants of the original notation are used; some are exactly defined, including *Extended Backus–Naur Form* (EBNF) and *Augmented Backus–Naur Form* (ABNF).

History

The idea of describing the structure of language with rewriting rules can be traced back to at least the work of Pāṇini (about the 4th century BC), who used it in his description of Sanskrit word structure - hence, some suggest to rename BNF to **Panini–Backus Form.**. American linguists such as Leonard Bloomfield and Zellig Harris took this idea a step further by attempting to formalize language and its study in terms of formal definitions and procedures (around 1920-1960).

Meanwhile, string rewriting rules as formal, abstract systems were introduced and studied by mathematicians such as Axel Thue (in 1914), Emil Post (1920s-1940s) and Alan Turing (1936). Noam Chomsky, teaching linguistics to students of information theory at MIT, combined linguistics and mathematics, by taking what is essentially Thue's formalism as the basis for the description of the syntax of natural language; he also introduced a clear distinction between generative rules (those of context-free grammars) and transformation rules (1956).

John Backus, a programming language designer at IBM, adopted Chomsky's generative rules to describe the syntax of the new programming language IAL, known today as ALGOL 58 (1959), using the BNF notation.

Further development of ALGOL led to ALGOL 60; in its report (1963), Peter Naur named Backus's notation **Backus Normal Form**, and simplified it to minimize the character set used. However, Donald Knuth argued that BNF should rather be read as **Backus–Naur Form**, as it is "not a normal form in any sense", unlike, for instance, Chomsky Normal Form.

Introduction

A BNF specification is a set of derivation rules, written as

```
 <symbol> ::= __expression__
```

where <symbol> is a *nonterminal*, and the __expression__ consists of one or more sequences of symbols; more sequences are separated by the vertical bar, '|', indicating a choice, the whole being a possible substitution for the symbol on the left. Symbols that never appear on a left side are *terminals*. On the other hand, symbols that appear on a left side are *non-terminals* and are always enclosed between the pair <>.

Example

As an example, consider this possible BNF for a U.S. postal address:

```
 <postal-address> ::= <name-part> <street-address> <zip-part>

      <name-part> ::= <personal-part> <last-name> <opt-jr-part> <EOL>
                    | <personal-part> <name-part> <EOL>

  <personal-part> ::= <first-name> | <initial> "."

 <street-address> ::= <house-num> <street-name> <opt-apt-num> <EOL>

       <zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>

    <opt-jr-part> ::= "Sr." | "Jr." | <roman-numeral> | ""
```

This translates into English as:

- A postal address consists of a name-part, followed by a street-address part, followed by a zip-code part.
- A name-part consists of either: a personal-part followed by a last name followed by an optional suffix (Jr., Sr., or dynastic number) and end-of-

line, or a personal part followed by a name part (this rule illustrates the use of recursion in BNFs, covering the case of people who use multiple first and middle names and/or initials).
- A personal-part consists of either a first name or an initial followed by a dot.
- A street address consists of a house number, followed by a street name, followed by an optional apartment specifier, followed by an end-of-line.
- A zip-part consists of a town-name, followed by a comma, followed by a state code, followed by a ZIP-code followed by an end-of-line.
- A opt-jr-part consists of a suffix, such as "Sr.", "Jr." or a roman-numeral, or an empty string (i.e. nothing).

Note that many things (such as the format of a first-name, apartment specifier, ZIP-code, and Roman numeral) are left unspecified here. If necessary, they may be described using additional BNF rules.

Further examples

BNF's syntax itself may be represented with a BNF like the following:

```
 <syntax> ::= <rule> | <rule> <syntax>
 <rule>   ::= <opt-whitespace> "<" <rule-name> ">" <opt-whitespace>
"::="
                 <opt-whitespace> <expression> <line-end>
 <opt-whitespace> ::= " " <opt-whitespace> | ""  <!-- "" is empty
string, i.e. no whitespace -->
 <expression>     ::= <list> | <list> "|" <expression>
 <line-end>       ::= <opt-whitespace> <EOL> | <line-end> <line-end>
 <list>    ::= <term> | <term> <opt-whitespace> <list>
 <term>    ::= <literal> | "<" <rule-name> ">"
 <literal> ::= '"' <text> '"' | "'" <text> "'" <!-- actually, the
original BNF did not use quotes -->
```

This assumes that no whitespace is necessary for proper interpretation of the rule. <EOL> represents the appropriate line-end specifier (in ASCII, carriage-return and/or line-feed, depending on the operating system). <rule-name> and <text> are to be substituted with a declared rule's name/label or literal text, respectively.

In the U.S. postal address example above, the entire block-quote is a syntax. Each line or unbroken grouping of lines is a rule; for example one rule begins with "<name-part> ::=". The other part of that rule (aside from a line-end) is an expression, which consists of two lists separated by a pipe "|". These two lists consists of some terms (three terms and two terms, respectively). Each term in this particular rule is a rule-name.

Variants

There are many variants and extensions of BNF, generally either for the sake of simplicity and succinctness, or to adapt it to a specific application. One common feature of many variants is the use of regular expression repetition operators such as `*` and `+`. The Extended Backus–Naur Form (EBNF) is a common one. In fact the example above is not the pure form invented for the ALGOL 60 report. The bracket notation "[ ]" was introduced a few years later in IBM's PL/I definition but is now universally recognised. ABNF and RBNF are other extensions commonly used to describe Internet Engineering Task Force (IETF) protocols.

Parsing expression grammars build on the BNF and regular expression notations to form an alternative class of formal grammar, which is essentially analytic rather than generative in character.

Many BNF specifications found online today are intended to be human readable and are non-formal. These often include many of the following syntax rules and extensions:

- Optional items enclosed in square brackets. E.g. [<item-x>]
- Items repeating 0 or more times are enclosed in curly brackets or suffixed with an asterisk. E.g. <word> ::= <letter> {<letter>}
- Items repeating 1 or more times are followed by a '+'
- Terminals may appear in bold and NonTerminals in plain text rather than using italics and angle brackets
- Alternative choices in a production are separated by the '|' symbol. E.g., <alternative-A> | <alternative-B>
- Where items need to be grouped they are enclosed in simple parentheses

Chapter 13

Parsing Expression Grammar

A **parsing expression grammar**, or **PEG**, is a type of analytic formal grammar, i.e. it describes a formal language in terms of a set of rules for recognizing strings in the language. The formalism was introduced by Bryan Ford in 2004 and is closely related to the family of Top-down parsing languages introduced in the early 1970s. Syntactially, PEGs also look similar to context-free grammars (CFGs), but they have a different interpretation, which is closer to how string recognition tends to be done in practice, e.g. by a recursive descent parser.

Unlike CFGs, PEGs cannot be ambiguous; if a string parses, it has exactly one valid parse tree. This makes PEGs well-suited to parsing computer languages, but not natural languages.

Definition

Syntax

Formally, a parsing expression grammar consists of:

- A finite set N of *nonterminal symbols*.
- A finite set Σ of *terminal symbols* that is disjoint from N.
- A finite set P of *parsing rules*.
- An expression e_S termed the *starting expression*.

Each parsing rule in P has the form $A \leftarrow e$, where A is a nonterminal symbol and e is a *parsing expression*. A parsing expression is a hierarchical expression similar to a regular expression, which is constructed in the following fashion:

1. An *atomic parsing expression* consists of:
    - any terminal symbol,
    - any nonterminal symbol, or
    - the empty string ε.
2. Given any existing parsing expressions e, e_1, and e_2, a new parsing expression can be constructed using the following operators:
    - *Sequence*: $e_1\ e_2$
    - *Ordered choice*: e_1 / e_2

- *Zero-or-more*: *e**
- *One-or-more*: *e*+
- *Optional*: *e*?
- *And-predicate*: &*e*
- *Not-predicate*: !*e*

Semantics

The fundamental difference between context-free grammars and parsing expression grammars is that the PEG's choice operator is *ordered*. If the first alternative succeeds, the second alternative is ignored. Thus ordered choice is not commutative, unlike unordered choice as in context-free grammars and regular expressions. Ordered choice is analogous to soft cut operators available in some logic programming languages.

The consequence is that if a CFG is transliterated directly to a PEG, any ambiguity in the former is resolved by deterministically picking one parse tree from the possible parses. By carefully choosing the order in which the grammar alternatives are specified, a programmer has a great deal of control over which parse tree is selected.

Like boolean context-free grammars, parsing expression grammars also add the and- and not- predicates. These facilitate further disambiguation, if reordering the alternatives cannot specify the exact parse tree desired.

Operational interpretation of parsing expressions

Each nonterminal in a parsing expression grammar essentially represents a parsing function in a recursive descent parser, and the corresponding parsing expression represents the "code" comprising the function. Each parsing function conceptually takes an input string as its argument, and yields one of the following results:

- *success*, in which the function may optionally move forward or *consume* one or more characters of the input string supplied to it, or
- *failure*, in which case no input is consumed.

A nonterminal may succeed without actually consuming any input, and this is considered an outcome distinct from failure.

An atomic parsing expression consisting of a single terminal succeeds if the first character of the input string matches that terminal, and in that case consumes the input character; otherwise the expression yields a failure result. An atomic parsing expression consisting of the empty string always trivially succeeds without consuming any input. An atomic parsing expression consisting of a nonterminal *A* represents a recursive call to the nonterminal-function *A*.

The sequence operator e_1 e_2 first invokes e_1, and if e_1 succeeds, subsequently invokes e_2 on the remainder of the input string left unconsumed by e_1, and returns the result. If either e_1 or e_2 fails, then the sequence expression e_1 e_2 fails.

The choice operator e_1 / e_2 first invokes e_1, and if e_1 succeeds, returns its result immediately. Otherwise, if e_1 fails, then the choice operator backtracks to the original input position at which it invoked e_1, but then calls e_2 instead, returning e_2's result.

The zero-or-more, one-or-more, and optional operators consume zero or more, one or more, or zero or one consecutive repetitions of their sub-expression *e*, respectively. Unlike in context-free grammars and regular expressions, however, these operators *always* behave greedily, consuming as much input as possible and never backtracking. (Regular expression matchers may start by matching greedily, but will then backtrack and try shorter matches if they fail to match.) For example, the expression a* will always consume as many a's as are consecutively available in the input string, and the expression (a* a) will always fail because the first part (a*) will never leave any a's for the second part to match.

Finally, the and-predicate and not-predicate operators implement syntactic predicates. The expression &*e* invokes the sub-expression *e*, and then succeeds if *e* succeeds and fails if *e* fails, but in either case *never consumes any input*. Conversely, the expression !*e* succeeds if *e* fails and fails if *e* succeeds, again consuming no input in either case. Because these can use an arbitrarily complex sub-expression *e* to "look ahead" into the input string without actually consuming it, they provide a powerful syntactic lookahead and disambiguation facility.

Examples

This is a PEG that recognizes mathematical formulas that apply the basic four operations to non-negative integers.

Value ← [0-9]+ / '(' *Expr* ')'
Product ← *Value* (('*' / '/') *Value*)*
Sum ← *Product* (('+' / '-') *Product*)*
Expr ← *Sum*

In the above example, the terminal symbols are characters of text, represented by characters in single quotes, such as `'('` and `')'`. The range `[0-9]` is also a shortcut for ten characters, indicating any one of the digits 0 through 9. (This range syntax is the same as the syntax used by regular expressions.) The nonterminal symbols are the ones that expand to other rules: *Value*, *Product*, *Sum*, and *Expr*.

The examples below drop quotation marks in order to be easier to read. Lowercase letters are terminal symbols, while capital letters in italics are nonterminals. Actual PEG parsers would require the lowercase letters to be in quotes.

The parsing expression **(a/b)*** matches and consumes an arbitrary-length sequence of a's and b's. The rule ***S* ← a *S*? b** describes the simple context-free "matching language" $\{a^n b^n : n \geq 1\}$. The following parsing expression grammar describes the classic non-context-free language $\{a^n b^n c^n : n \geq 1\}$:

> *S* ← &(*A* c) a+ *B* !(a/b/c)
> *A* ← a *A*? b
> *B* ← b *B*? c

The following recursive rule matches standard C-style if/then/else statements in such a way that the optional "else" clause always binds to the innermost "if", because of the implicit prioritization of the '/' operator. (In a context-free grammar, this construct yields the classic dangling else ambiguity.)

> *S* ← if *C* then *S* else *S* / if *C* then *S*

The parsing expression **foo &(bar)** matches and consumes the text "foo" but only if it is followed by the text "bar". The parsing expression **foo !(bar)** matches the text "foo" but only if it is *not* followed by the text "bar". The expression **!(a+ b) a** matches a single "a" but only if it is not the first in an arbitrarily-long sequence of a's followed by a b.

The following recursive rule matches Pascal-style nested comment syntax, (* which can (* nest *) like this *). The comment symbols appear in double quotes to distinguish them from PEG operators.

> *Begin* ← "(*"
> *End* ← "*)"
> *C* ← *Begin N* End*
> *N* ← *C* / (!*Begin* !*End Z*)
> *Z* ← *any single character*

Implementing parsers from parsing expression grammars

Any parsing expression grammar can be converted directly into a recursive descent parser. Due to the unlimited lookahead capability that the grammar formalism provides, however, the resulting parser could exhibit exponential time performance in the worst case.

It is possible to obtain better performance for any parsing expression grammar by converting its recursive descent parser into a *packrat parser*, which always runs in linear time, at the cost of substantially greater storage space requirements. A packrat parser is a form of parser similar to a recursive descent parser in construction, except that during the parsing process it memoizes the intermediate results of all invocations of the mutually recursive parsing functions, ensuring that each parsing function is only invoked at most once at a given input position. Because of this memoization, a packrat parser has the ability to parse many context-free grammars and *any* parsing expression grammar (including some that do not represent context-free languages) in linear time. Examples of memoized recursive descent parsers are known from at least as early as 1993. Note that this analysis of the performance of a packrat parser assumes that enough memory is available to hold all of the memoized results; in practice, if there were not enough memory, some parsing functions might have to be invoked more than once at the same input position, and consequently the parser could take more than linear time.

It is also possible to build LL parsers and LR parsers from parsing expression grammars, with better worst-case performance than a recursive descent parser, but the unlimited lookahead capability of the grammar formalism is then lost. Therefore, not all languages that can be expressed using parsing expression grammars can be parsed by LL or LR parsers.

Advantages

Compared to pure regular expressions (i.e. without back-references), PEGs are strictly more powerful, but require significantly more memory. For example, a regular expression inherently cannot find an arbitrary number of matched pairs of parentheses, because it is not recursive, but a PEG can. However, a PEG will require an amount of memory proportional to the length of the input, while a regular expression matcher will require only a constant amount of memory.

Any PEG can be parsed in linear time by using a packrat parser, as described above.

Parsers for languages expressed as a CFG, such as LR parsers, require a separate tokenization step to be done first, which breaks up the input based on the location of spaces, punctuation, etc. The tokenization is necessary because of the way these parsers use *lookahead* to parse CFGs that meet certain requirements in linear time. PEGs do not require tokenization to be a separate step, and tokenization rules can be written in the same way as any other grammar rule.

Many CFGs contain inherent ambiguities, even when they're intended to describe unambiguous languages. The "dangling else" problem in C, C++, and Java is one example. These problems are often resolved by applying a rule outside of the grammar. In a PEG, these ambiguities never arise, because of prioritization.

Disadvantages

Memory consumption

PEG parsing is typically carried out via *packrat parsing*, which uses memoization to eliminate redundant parsing steps. Packrat parsing requires storage proportional to the total input size, rather than the depth of the parse tree as with LR parsers. This is a significant difference in many domains: for example, hand-written source code has an effectively constant expression nesting depth independent of the length of the program—expressions nested beyond a certain depth tend to get refactored.

For some grammars and some inputs, the depth of the parse tree can be proportional to the input size, so that in asymptotic the worst-case which does not distinguish these two metrics packrat parsing will appear to be no worse than an LR parser. This is similar to a situation which arises in graph algorithms: the Bellman–Ford algorithm and Floyd–Warshall algorithm appear to have the same running time ($O(|V|^3)$) if only the number of vertices is considered. However, a more accurate analysis which accounts for the number of edges as a separate parameter assigns the Bellman–Ford algorithm a time of $O(|V| * |E|)$, which is only quadratic in the size of the input (rather than cubic).

Indirect left recursion

PEGs cannot express *left-recursive* rules where a rule refers to itself without moving forward in the string. For example, in the arithmetic grammar above, it would be tempting to move some rules around so that the precedence order of products and sums could be expressed in one line:

```
Value   ← [0-9.]+ / '(' Expr ')'
Product ← Expr (('*' / '/') Expr)*
Sum     ← Expr (('+' / '-') Expr)*
Expr    ← Product / Sum / Value
```

In this new grammar matching an `Expr` requires testing if a `Product` matches while matching a `Product` requires testing if an `Expr` matches. This circular definition cannot be resolved. However, left-recursive rules can always be rewritten to eliminate left-recursion. For example, the following left-recursive CFG rule:

```
string-of-a ← string-of-a 'a' | 'a'
```

can be rewritten in a PEG using the plus operator:

```
string-of-a ← 'a'+
```

The process of rewriting *indirectly* left-recursive rules is complex in some packrat parsers, especially when semantic actions are involved.

With some modification, traditional packrat parsing can support direct left recursion. , but doing so results in a loss of the linear-time parsing property which is generally the justification for using PEGs and Packrat Parsing in the first place. Only the OMeta parsing algorithm supports full direct and indirect left recursion without additional attendant complexity (but again, at a loss of the linear time complexity), whereas all GLR parsers support left recursion (without sacrificing linear time complexity).

Subtle grammatical errors

In order to express a grammar as a PEG, the grammar author must convert all instances of nondeterministic choice into prioritized choice. Unfortunately, this process is error prone, and often results in grammars which mis-parse certain inputs. An example and commentary can be found here.

Expressive power

Packrat parsers cannot recognize some unambiguous grammars, such as the following (example taken from)

```
S ← 'x' S 'x' | 'x'
```

In fact, neither LL(k) nor LR(k) parsing algorithms are capable of recognizing this example.

Maturity

PEGs are new and not widely implemented. In contrast, regular expressions and CFGs have been around for decades, the code to parse them has been extensively optimized, and many programmers are familiar with how to use them. There was a time when the same considerations applied to these approaches as well.

Chapter 14

LR Parser

In computer science, an **LR parser** is a parser that reads input from **L**eft to right (as it would appear if visually displayed) and produces a **R**ightmost derivation. The term **LR(*k*) parser** is also used; where the *k* refers to the number of unconsumed "look ahead" input symbols that are used in making parsing decisions. Usually *k* is 1 and the term **LR parser** is often intended to refer to this case.

The syntax of many programming languages can be defined by a grammar that is LR(1), or close to being so, and for this reason LR parsers are often used by compilers to perform syntax analysis of source code.

An LR parser can be created from a context-free grammar by a program called a parser generator or hand written by a programmer. A context-free grammar is classified as LR(*k*) if there exists an LR(*k*) parser for it, as determined by the parser generator.

An LR parser is said to perform bottom-up parsing because it attempts to deduce the top level grammar productions by building up from the **leaves**.

A deterministic context-free language is a language for which some LR(*k*) grammar exists. Every LR(*k*) grammar for $k > 1$ can be mechanically transformed into an LR(*1*) grammar for the same language, while an LR(*0*) grammar for the same language may not exist; the LR(*0*) languages are a proper subset of the deterministic ones.

An LR parser is based on an algorithm which is driven by a parser table, a data structure which contains the syntax of the computer language being parsed. So the term LR parser actually refers to a class of parser that can process almost any programming language, as long as the parser table for a programming language is supplied. The parser table is created by a program called a parser generator.

LR parsing can be generalized as arbitrary context-free language parsing without a performance penalty, even for LR(k) grammars. This is because most programming languages can be expressed with LR(k) grammars, where k is a small constant (usually 1). Note that parsing of non-LR(k) grammars is an order of magnitude slower (cubic instead of quadratic in relation to the input length).

LR parsing can handle a larger range of languages than LL parsing, and is also better at error reporting, i.e. it detects syntactic errors when the input does not conform to the grammar as soon as possible. This is in contrast to an LL(k) (or even worse, an LL(*) parser) which may defer error detection to a different branch of the grammar due to backtracking, often making errors harder to localize across disjunctions with long common prefixes.

LR parsers are difficult to produce by hand and they are usually constructed by a parser generator or a compiler-compiler. Depending on how the parsing table is generated, these parsers can be called simple LR parsers (SLR), look-ahead LR parsers (LALR), or canonical LR parsers. LALR parsers have more language recognition power than SLR parsers. Canonical LR parsers have more recognition power than LALR parsers.

Architecture of LR parsers

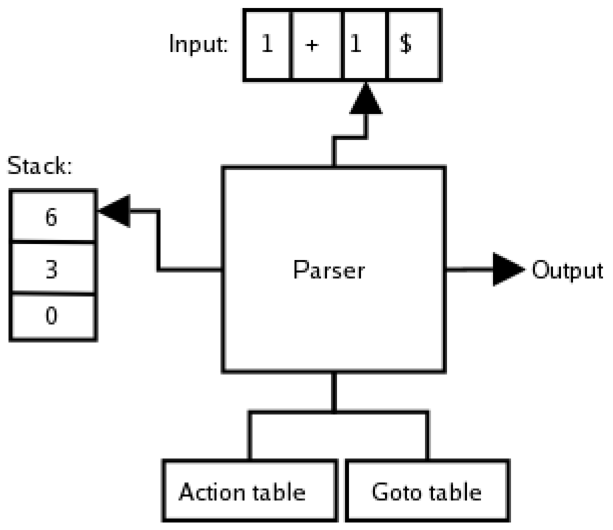


Figure 1. Architecture of a table-based bottom-up parser

Conceptually, an LR Parser is a recursive program that can be proven correct by direct computation, and can be implemented more efficiently (in time) as a recursive ascent parser, a set of mutually-recursive functions for every grammar, much like a recursive descent parser. Conventionally, however, LR parsers are presented and implemented as table-based stack machines in which the call stack of the underlying recursive program is explicitly manipulated.

A table-driven bottom-up parser can be schematically presented as in Figure 1. The following describes a rightmost derivation by this parser.

General case

The parser is a state machine. It consists of:

- an input buffer
- a stack on which is stored a list of states it has been in
- a goto table that prescribes to which new state it should move
- an action table that gives a grammar rule to apply given the current state and the current terminal in the input stream
- a set of CFL rules

Since the LR parser reads input from left to right but needs to produce a rightmost derivation, it uses reductions, instead of derivations to process input. That is, the algorithm works by creating a "leftmost reduction" of the input. The end result, when reversed, will be a rightmost derivation.

The LR parsing algorithm works as follows:

1. The stack is initialized with [**0**]. The current state will always be the state that is at the top of the stack.
2. Given the current state and the current terminal on the input stream an action is looked up in the action table. There are four cases:
    - a shift s*n*:
        - the current terminal is removed from the input stream
        - the state *n* is pushed onto the stack and becomes the current state
    - a reduce r*m*:
        - the number *m* is written to the output stream
        - for every symbol in the right-hand side of rule *m* a state is removed from the stack (i. e. if the right-hand side of rule *m* consists of 3 symbols, 3 top states are removed from the stack)
        - given the state that is then on top of the stack and the left-hand side of rule *m* a new state is looked up in the goto table and made the new current state by pushing it onto the stack
    - an accept: the string is accepted
    - no action: a syntax error is reported
3. Step 2 is repeated until either the string is accepted or a syntax error is reported.

Concrete example

To explain its workings we will use the following small grammar whose start symbol is E:

(1) E → E * B
(2) E → E + B
(3) E → B

(4) B → 0
(5) B → 1

and parse the following input:

1 + 1

Action and goto tables

The two LR(0) parsing tables for this grammar look as follows:

state	***action'***					***goto***	
	*****	**+**	**0**	**1**	**$**	**E**	**B**
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

The **action table** is indexed by a state of the parser and a terminal (including a special terminal $ that indicates the end of the input stream) and contains three types of actions:

- *shift*, which is written as 's*n*' and indicates that the next state is *n*
- *reduce*, which is written as 'r*m*' and indicates that a reduction with grammar rule *m* should be performed
- *accept*, which is written as 'acc' and indicates that the parser accepts the string in the input stream.

The **goto table** is indexed by a state of the parser and a nonterminal and simply indicates what the next state of the parser will be if it has recognized a certain nonterminal. This table is important to find out the next state after every reduction. After a reduction, the next state is found by looking up the **goto table** entry for top of the stack (i.e. current state) and the reduced rule's LHS (i.e. non-terminal).

Parsing procedure

The table below illustrates each step in the process. Here the state refers to the element at the top of the stack (the right-most element), and the next action is determined by referring to the action table above. Also note that a $ is appended to the input string to denote the end of the stream.

State	Input Stream	Output Stream	Stack	Next Action
0	1+1$		[0]	Shift 2
2	+1$		[0,2]	Reduce 5
4	+1$	5	[0,4]	Reduce 3
3	+1$	5,3	[0,3]	Shift 6
6	1$	5,3	[0,3,6]	Shift 2
2	$	5,3	[0,3,6,2]	Reduce 5
8	$	5,3,5	[0,3,6,8]	Reduce 2
3	$	5,3,5,2	[0,3]	Accept

Walkthrough

The parser starts out with the stack containing just the initial state ('0'):

[**0**]

The first symbol from the input string that the parser sees is '1'. In order to find out what the next action is (shift, reduce, accept or error), the action table is indexed with the current state (remember that the "current state" is just whatever is on the top of the stack), which in this case is 0, and the current input symbol, which is '1'. The action table specifies a shift to state 2, and so state 2 is pushed onto the stack (again, remember that all the state information is in the stack, so "shifting to state 2" is the same thing as pushing 2 onto the stack). The resulting stack is

[**0** '1' **2**]

where the top of the stack is 2. For the sake of explanation we also show the symbol (e.g., '1', B) that caused the transition to the next state, although strictly speaking it is not part of the stack.

In state 2 the action table says that regardless of what terminal we see on the input stream, we should do a reduction with grammar rule 5. If the table is correct, this means that the parser has just recognized the right-hand side of rule 5, which is indeed the case. In this case we write 5 to the output stream, pop one state from the stack (since the right-hand side of the rule has one symbol), and push on the stack the state from the cell in the goto table for state 0 and B, i.e., state 4. The resulting stack is:

[**0** B **4**]

However, in state 4 the action table says we should now do a reduction with rule 3. So we write 3 to the output stream, pop one state from the stack, and find the new state in the goto table for state 0 and E, which is state 3. The resulting stack:

[**0** E **3**]

The next terminal that the parser sees is a '+' and according to the action table it should then go to state 6:

[**0** E **3** '+' **6**]

Note that the resulting stack can be interpreted as the history of a finite state automaton that has just read a nonterminal E followed by a terminal '+'. The transition table of this automaton is defined by the shift actions in the action table and the goto actions in the goto table.

The next terminal is now '1' and this means that we perform a shift and go to state 2:

[**0** E **3** '+' **6** '1' **2**]

Just as the previous '1' this one is reduced to B giving the following stack:

[**0** E **3** '+' **6** B **8**]

Again note that the stack corresponds with a list of states of a finite automaton that has read a nonterminal E, followed by a '+' and then a nonterminal B. In state 8 we always perform a reduce with rule 2. Note that the top 3 states on the stack correspond with the 3 symbols in the right-hand side of rule 2.

[**0** E **3**]

Finally, we read a '$' from the input stream which means that according to the action table (the current state is 3) the parser accepts the input string. The rule numbers that will then have been written to the output stream will be [5, 3, 5, 2] which is indeed a rightmost derivation of the string "1 + 1" in reverse.

Constructing LR(0) parsing tables

Items

The construction of these parsing tables is based on the notion of *LR(0) items* (simply called *items* here) which are grammar rules with a special dot added somewhere in the right-hand side. For example the rule E → E + B has the following four corresponding items:

> E → • E + B
> E → E • + B
> E → E + • B
> E → E + B •

Rules of the form *A* → ε have only a single item *A* → •. The item E → E • + B, for example, indicates that the parser has recognized a string corresponding with E on the input stream and now expects to read a '+' followed by another string corresponding with B.

Item sets

It is usually not possible to characterize the state of the parser with a single item because it may not know in advance which rule it is going to use for reduction. For example if there is also a rule E → E * B then the items E → E • + B and E → E • * B will both apply after a string corresponding with E has been read. Therefore we will characterize the state of the parser by a set of items, in this case the set { E → E • + B, E → E • * B }.

Extension of Item Set by expansion of non-terminals

An item with a dot before a nonterminal, such as E → E + • B, indicates that the parser expects to parse the nonterminal B next. To ensure the item set contains all possible rules the parser may be in the midst of parsing, it must include all items describing how B itself will be parsed. This means that if there are rules such as B → 1 and B → 0 then the item set must also include the items B → • 1 and B → • 0. In general this can be formulated as follows:

> If there is an item of the form *A* → *v* • *Bw* in an item set and in the grammar there is a rule of the form *B* → *w'* then the item *B* → • *w'* should also be in the item set.

Closure of item sets

Thus, any set of items can be extended by recursively adding all the appropriate items until all nonterminals preceded by dots are accounted for. The minimal extension is called the *closure* of an item set and written as **clos**(*I*) where *I* is an item set. It is these closed item sets that we will take as the states of the parser, although only the ones that are actually reachable from the begin state will be included in the tables.

Augmented grammar

Before we start determining the transitions between the different states, the grammar is always augmented with an extra rule

(0) S → E

where S is a new start symbol and E the old start symbol. The parser will use this rule for reduction exactly when it has accepted the input string.

For our example we will take the same grammar as before and augment it:

(0) S → E
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1

It is for this augmented grammar that we will determine the item sets and the transitions between them.

Table construction

Finding the reachable item sets and the transitions between them

The first step of constructing the tables consists of determining the transitions between the closed item sets. These transitions will be determined as if we are considering a finite automaton that can read terminals as well as nonterminals. The begin state of this automaton is always the closure of the first item of the added rule: S → • E:

Item set 0
S → • E
+ E → • E * B
+ E → • E + B
+ E → • B
+ B → • 0
+ B → • 1

The boldfaced "+" in front of an item indicates the items that were added for the closure (not to be confused with the mathematical '+' operator which is a terminal). The original items without a "+" are called the *kernel* of the item set.

Starting at the begin state (S0) we will now determine all the states that can be reached from this state. The possible transitions for an item set can be found by looking at the symbols (terminals and nonterminals) we find right after the dots; in the case of item set 0 these are the terminals '0' and '1' and the nonterminal E and B. To find the item set that a symbol *x* leads to we follow the following procedure:

1. Take the set, *S*, of all items in the current item set where there is a dot in front of some symbol *x*.
2. For each item in *S*, move the dot to the right of *x*.
3. Close the resulting set of items.

For the terminal '0' this results in:

Item set 1
B → 0 •

and for the terminal '1' in:

Item set 2
B → 1 •

and for the nonterminal E in:

Item set 3
S → E •
E → E • * B
E → E • + B

and for the nonterminal B in:

Item set 4
E → B •

Note that the closure does not add new items in all cases. We continue this process until no more new item sets are found. For the item sets 1, 2, and 4 there will be no transitions since the dot is not in front of any symbol. For item set 3 we see that the dot is in front of the terminals '*' and '+'. For '*' the transition goes to:

Item set 5
E → E * • B
+ B → • 0
+ B → • 1

and for '+' the transition goes to:

Item set 6
E → E + • B
+ B → • 0
+ B → • 1

For item set 5 we have to consider the terminals '0' and '1' and the nonterminal B. For the terminals we see that the resulting closed item sets are equal to the already found item sets 1 and 2, respectively. For the nonterminal B the transition goes to:

Item set 7
E → E * B •

For item set 6 we also have to consider the terminal '0' and '1' and the nonterminal B. As before, the resulting item sets for the terminals are equal to the already found item sets 1 and 2. For the nonterminal B the transition goes to:

Item set 8
E → E + B •

These final item sets have no symbols beyond their dots so no more new item sets are added and we are finished. The finite automaton, with item sets as its states is shown below.

The transition table for the automaton now looks as follows:

Item Set	*	+	0	1	E	B
0			1	2	3	4
1						
2						
3	5	6				
4						
5			1	2		7
6			1	2		8
7						
8						

Constructing the action and goto tables

From this table and the found item sets we construct the action and goto table as follows:

1. the columns for nonterminals are copied to the goto table
2. the columns for the terminals are copied to the action table as shift actions

3. an extra column for '$' (end of input) is added to the action table that contains *acc* for every item set that contains S → E •.
4. if an item set *i* contains an item of the form *A* → *w* • and *A* → *w* is rule *m* with *m* > 0 then the row for state *i* in the action table is completely filled with the reduce action r*m*.

The reader may verify that this results indeed in the action and goto table that were presented earlier on.

A note about LR(0) versus SLR and LALR parsing

Note that only step 4 of the above procedure produces reduce actions, and so all reduce actions must occupy an entire table row, causing the reduction to occur regardless of the next symbol in the input stream. This is why these are LR(0) parse tables: they don't do any lookahead (that is, they look ahead zero symbols) before deciding which reduction to perform. A grammar that needs lookahead to disambiguate reductions would require a parse table row containing different reduce actions in different columns, and the above procedure is not capable of creating such rows.

Refinements to the **LR**(0) table construction procedure (such as SLR and LALR) are capable of constructing reduce actions that do not occupy entire rows. Therefore, they are capable of parsing more grammars than LR(0) parsers.

Conflicts in the constructed tables

The automaton is constructed in such a way that it is guaranteed to be deterministic. However, when reduce actions are added to the action table it can happen that the same cell is filled with a reduce action and a shift action (a *shift-reduce conflict*) or with two different reduce actions (a *reduce-reduce conflict*). However, it can be shown that when this happens the grammar is not an LR(0) grammar.

A small example of a non-LR(0) grammar with a shift-reduce conflict is:

(1) E → 1 E
(2) E → 1

One of the item sets we then find is:

Item set 1
E → 1 • E
E → 1 •
+ E → • 1 E
+ E → • 1

There is a shift-reduce conflict in this item set because in the cell in the action table for this item set and the terminal '1' there will be both a shift action to state 1 and a reduce action with rule 2.

A small example of a non-LR(0) grammar with a reduce-reduce conflict is:

(1) E → A 1
(2) E → B 2
(3) A → 1
(4) B → 1

In this case we obtain the following item set:

Item set 1
A → 1 •
B → 1 •

There is a reduce-reduce conflict in this item set because in the cells in the action table for this item set there will be both a reduce action for rule 3 and one for rule 4.

Both examples above can be solved by letting the parser use the follow set of a nonterminal *A* to decide if it is going to use one of *A*s rules for a reduction; it will only use the rule *A* → *w* for a reduction if the next symbol on the input stream is in the follow set of *A*. This solution results in so-called Simple LR parsers.