BASICS IN COMPILER DESIGN

Kunal Dey
Dr. Gokul Thanigaivasan
Dr. Santosh S Chowhan

BOOKS ARCADE

Basics in Compiler Design

Basics in Compiler Design

Kunal Dey
Dr. Gokul Thanigaivasan
Dr. Santosh S Chowhan

BOOKS ARCADE
KRISHNA NAGAR, DELHI

Basics in Compiler Design

Kunal Dey
Dr. Gokul Thanigaivasan
Dr. Santosh S Chowhan

BOOKS ARCADE

Regd. Office:
F-10/24, East Krishna Nagar, Near Vijay Chowk, Delhi-110051
Ph. No: +91-11-79669196, +91-9899073222
E-mail: info@booksarcade.co.in, booksarcade.pub@gmail.com
Website: www.booksarcade.co.in

Year of Publication 2023

International Standard Book Number-13: 978-81-19199-24-2

CONTENTS

CHAPTER 1

INTRODUCTION TO COMPILER DESIGN

Kunal Dey
Assistant Professor, Department of Data Science & Analytics, School of Sciences,
Jain (Deemed-to-be University), Bangalore-27, India
Email Id- d.kunal@jainuniversity.ac.in

Basic Concepts of Compiler Design

This chapter includes the fundamental ideas and background of compiler design. It includes compiler's structural properties, over the formal grammar and parse trees, and construct a simple recursive-descent expression compiler. However, before diving into the text, a word of encouragement about this volume and the book as a whole seems pertinent. Once you are knowledgeable about a compiler's overall structure, compiling programmers is not very challenging to master. The key issue with a compiler is not the fact any one component is difficult to grasp; rather, it is that there are so many elements and that you must first understand the majority of them before either one of them makes sense.

Properties of a Good Compiler

The main quality of a good compiler is that it is effective in medical code. A compiler that sometimes delivers flawed code is worthless; a compiler which thus produces flawed code just once a year sometimes seems helpful but it is harmful. A compiler must fully adhere to the language requirement to function. Users may even be grateful for the support when a subset, superset, or even what is sometimes mockingly kept referring to as an "extendedsubset" of the language is implemented, yet those same users will soon discover that the organizational capabilities with such a compiler are considerably less portable than those created with a fully conforming compiler. A competent compiler ought to also be able to handle programmes of almost any size, as long as memory enables. This is a quality that is sometimes underestimated. No sensible programmer, it would appear, uses more than 32 arguments in a technique or more than 128 declarations in a block, and so one might assign a predetermined amount of disk space in the compiler for each. However, it's important to remember that not only programmers create programs. Although more than 32 arguments to a procedure appear overwhelming, even for a created program, much information is generated by other programmes, and such generated software may easily have more than 128 declarations in one blockof famous last words.

The implementation of automatically developed parsers and code generators often uses extremely lengthy compared with the results, therefore any limitations on the number of scenarios in a case or switch statement are unjustified. Programs of virtually any size may be handled with the flexible memory space required at a very low-cost increase. Although not a serious problem, compilation speed is a concern. Small applications should compile on contemporary processors in under a second. Larger programming projects are often divided into several, relatively tiny compilation units, such as components, library routines, and subprograms. Although each of these compiling units may be recompiled following a programme change, this

is often limited to the updated translation units alone. Additionally, compiler developers have previously taken care to make their compilers linear in the input, which implies that the length of the input file has no consequence on how long it takes to build a programme. Given that produced programmes might be rather lengthy, this becomes especially crucial when they are being assembled. Non-linearity in compilers might come from a variety of places. First of all, all linear time parsing strategies are extremely awkward, but in the worst case, the input size for worry-free parsing techniques may be cubic. Second, as the optimum code is often only discovered by putting into consideration all feasible machine instruction combinations, many code enhancements have the potential for exponential growth in the input size. Third, careless memory use could lead to quadratic time consumption. Fortunately, all of these issues have excellent linear-time solutions or heuristics. Nowadays, with the majority of computers having gigabytes of main memory, compiler size is seldom ever a concern. However, when applications use the compiler again with run time, as in just-in-time compilation, compiler size and acting skills are crucial.

Portability and Retarget Ability

When running on several machine types with a little amount of time and effort, a programme is said to be portable. Of course, everyone possesses their definition of what "a limited and appropriate effort" is, but nowadays many applications can be migrated by just modifying the make file to accommodate the local circumstances and rearchitecting. And often, even the operation of local circumstance adaptability might be automated, for instance by using GNU's autoconf. With compilers, machine reliance may exist greater heavily in the output than it does in the programme itself. As a result, when we employ a compiler, we need to take into account one again aspect of machine independence: how easily it can be customized to produce code for other computers. This is known as the compiler optimization retarget ability, which must be separated from its portability.

Good portability may be envisaged if the compiler is built in a known for its modern language in a manner that is generally acceptable. Retargeting is accomplished by dramatically changing the back-end; as a result, the work required to build a new back-end is inversely linked to the potential to retarget. In this context, it's important to bear in mind that building a new back-end doesn't always need to start from scratch. Of course, some of the code in a back-end is machine-dependent, but the overwhelming is not. If correctly organized, certain components from other back-ends may be used, and formalized machine descriptions may be used to construct further components. This strategy may engineering materials for a huge enterprise's back end a modest job. For a competent compiler writer with the right tools, building a back-end for a new machine can take one to four programmer months. Machine descriptions might be hundreds of lines long or thousands of lines long. This completes the first section of our introduction, having focused on creating a compiler. In the remaining sections of this chapter, we address three more topics: formal grammar, closure algorithms, and the development of translators through time.

A Short History of Compiler Construction

Three periods can be distinguished in the history of compiler construction: 1945-1960, 1960-1975, and 1975–present. Of course, the years are approximate.

i. 1945–1960: Code Generation:

During this period programming languages developed relatively slowly and machines were idiosyncratic. The primary problem was how to generate code for a given machine. The problem was exacerbated by the fact that assembly programming was held in high esteem, and high-level languages and compilers were looked at with a mixture of suspicion and awe: using a compiler was often called "automatic programming". Proponents of high-level languages feared, not without reason that the idea of high-level programming would never catch on if compilers produced code that was less efficient than what assembly programmers produced by hand. The first FORTRAN compiler, written by Sheridan et al. in 1959, optimized heavily and was far ahead of its time in that respect.

ii. 1960–1975: Parsing:

The 1960s and 1970s saw a proliferation of new programming languages, and language designers began to believe that having a compiler for a new language quickly was more important than having one that generated very efficient code. This shifted the emphasis on compiler construction from back-ends to front-ends. At the same time, studies in formal languages revealed several powerful techniques that could be applied profitably in front-end construction, notably in parser generation.

iii. 1975-Present: Code Generation and Code Optimization

From 1975 to the present, both the number of new languages proposed and the number of different machine types in regular use decreased, which reduced the need for quick-and-simple/quick-and-dirty compilers for new languages and/or machines. With the greatest turmoil in language and machine design being over, people began to demand professional compilers that were reliable, and efficient, both in use and in generated code, and preferably with pleasant user interfaces. This called for more attention to the quality of the generated code, which was easier now since with the slower change in machines the expected lifetime of a code generator increased. Also, at the same time new paradigms in programming were developed, with functional, logic, and distributed programming as the most prominent examples. Almost invariably, the run-time requirements of the corresponding languages far exceeded those of the imperative languages: automatic data allocation and deallocation, list comprehensions, unification, remote procedure call, and many others, are features which require much run-time effort that corresponds to hardly any code in the program text. More and more, the emphasis shifts from "how to compile" to "what to compile".

Grammars

The fundamental formalization for expressing the structure of programmes in a programming language is grammars, or more especially context-free grammars. In theory, a language's grammar primarily explains its syntactic structure, but because a language's semantics are determined by its syntax, the grammar also plays a key role in determining the semantics. Although there are other forms of grammar than context-free grammars, we shall focus mostly on them. We will also encounter attribute grammars, which have been context-free grammars that have been expanded with parameters and code, as well as frequent grammars, which are more often referred to as "regular expressions" and are the consequence of stringent limitations on context-free grammars. Only a small part of some of the other kinds of grammars are used in compiler design. Context-free is often referred to as CF. A "grammar" is a guide for generating the components of a collection of symbol strings. When it comes to programming languages, the

symbolism serves as the language's tokens, the strings of symbols serve as the programme texts, and the collection of symbol threads serves as the language itself. Following is the string:

[BEGIN print ("Hi!") END]

Contains 6 symbols (tokens) and might be a part of the string of symbols produced by the grammar of a programming language, or, to use more common terminology, a programme in a programming language. The fact that perhaps the strings are built in an organized way, and meanings may be linked to this structure, renders this simplistic conception of a programming-language worthless.

General Fundamental Information of Compiler

The compiler is software that changes a program written in a high-level language which again is the source language to a low-level language (Object/Target/Machine Language/0's, 1's) as displayed in Figure 1.

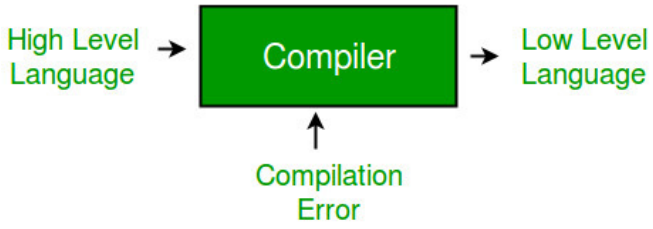


Figure 1: Illustrated the Block diagram of Compiler Design.

- A cross-compiler that performs on machine "A" and generates code for machine "B". It can manufacture code for platforms other than the one on which a compiler is an event in development.
- A source-to-source compiler, also widely recognized as a trans-compiler or compatible with older, converts source code from one programming language into the source code of another.

Language processing systems (using a Compiler)

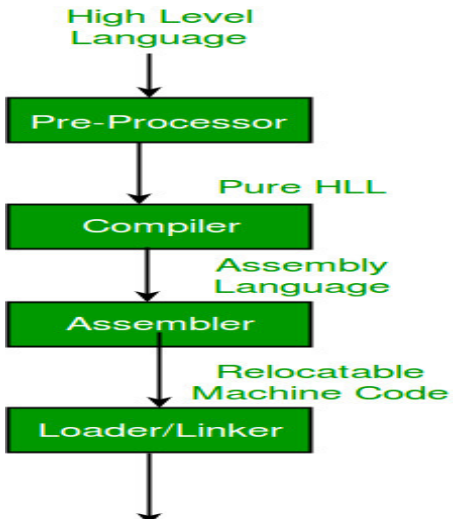


Figure 2: Illustrated the Language Processing System.

We know a computer is a logical assemblage of Software and Hardware. The hardware recognizes a language that is hard for us to grasp, subsequently, we prefer to write programs in a high-level language which is much less challenging for us to fathom and retain ideas. Now, these applications go through a series of modifications so that they should conveniently be utilized by machines. This is when language process products come in helpful.

An application that converts an input program written in one programming language into more of an equivalent program written in another language is known as a translator or language processor. A specific kind of translator called a compiler converts a program written in a high-level software package into an equivalent program written in a low-level language like machine code or assembly language.

A source program is a program written in assembly language, whereas an object (or target) program is a program translated into a low-level language. Additionally, the compiler creates an error report and tracks down the problems in the source code. No program created in a high-level language may be run before compilation. Only the machine phonetic pronunciation of the program is put into memory for execution after compilation as displayed in Figure 2. For every computer program, we have a separate compiler; nonetheless, the core functions performed by every programmer are the same.

- ***High-Level Language:***

If a program includes #define or #include directives such as #include or #define it is termed HLL. They are distant from machines but connected to people. Preprocessor directives are what are used with these (#) tags and they advise from before on what to do.

- ***Pre-Processor:***

The pre-processor destroys all #include directives via a method called file inclusion and disables all #define directives with macro expansion. It carries out file acquisition, macro processing, and file inclusion.

- ***Assembly Language:***

It is neither high-level nor in binary numbers. It is an intermediate step that is a mix of native machine code and some additional sensitive files required for execution.

- ***Assembler:***

We will have an assembler for each environment (hardware + operating system). They are not comprehensive as for each ecosystem we have one. The output of the assembler is dubbed an object file and it transforms the sequence of instructions into machine code.

- ***Interpreter:***

An interpreter turns high-level language into low-level machine language, exactly like a compiler. Nonetheless, the method that they handle the information varies. The interpretation does the same task line by line, but really the compiler receives the inputs, interprets them, and then executes the program code all at once. A compiler examines the software application and translates it for what it is into machine code while an interpreter transforms the program one statement at a time. Programs that have been interpreted often run more gradually than those that are created.

- ***Reloadable Machine Code:***

These can be loaded at any moment and can be launched. The program's address will be done in such a manner that it will truly outstanding migration.

- ***Loader/Linker:***

It attempts to execute the code after converting the reloadable code in and out of absolute code, which either results in the program operating or in an error message or sometimes both can happen.

To make a file operational, a linker must combine many object information into a single file. After that, the loader puts it into memory and runs it.

Phases of a Compiler

The compilation is broken down into three major stages, each of which comprises several components. They all communicate and use input from the conclusion of the timeframe as input.

Analysis Phase:

From the provided source code, an intermediate representation is produced:

1. Lexical Analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. Intermediate Code Generator

The program is broken up into "tokens" by the lexical analyzer, "sentences" are identified by the syntax inspector as employing the language, and the semantic analyzer examines the static interpretations of each construct. Creates "abstract" code using the Intermediate Code Generator.

Synthesis Phase:

The intermediary representation is used to construct an equivalent target program. It is divided into two sections:

A. Code Optimizer

B. Code Generator

The final Software Generator converts abstract intermediate representation into precise machine instructions once the Code Optimizer has optimized the abstract code.

Phases of Compiler

We generally have two stages of compilers, namely the Requirement analysis and Synthesis phase.

The analysis step develops an intermediate code from the provided programming language as shown in Figure 3. The synthesis step constructs an analogous activation in response to the intermediate language.

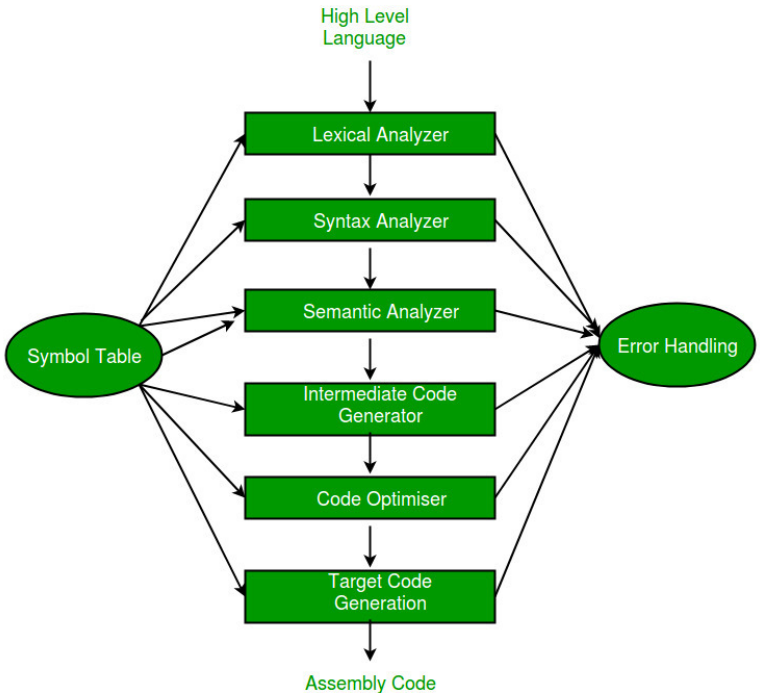


Figure 3: Illustrated the System of Phases of a Compiler.

Symbol Table:

It is a multidimensional array that the compiler uses and preserves up to date, and it contains all of the identities of identifiers together with corresponding types. It makes the compiler extremely efficient by rapidly discovering the identifiers.The study of an application code is broken down primarily into the following three steps, which have been listed below:

i. Linear Analysis:

This involves full the character stream from left to right even during the scanning step. It is then divided into the following tokens with a broader definition.

ii. Hierarchical Analysis

The tokens are categorized hierarchically into nested sections during this analysis step based on conclusions that could be drawn.

iii. Semantic Analysis:

This stage is used to determine the significant difference between the source program's components.The front endpoint and the back end are the requirements gathered that make up the compiler. The lexical analyzer, semantic analyzer, syntax analyzer, and intermediate code generating make up the frontend. The remaining are put together to construct the back end.

i. Lexical Analyzer:

Another name for it is a scanner. It accepts the preprocessor's output, which is in a pure high-level vocabulary and handles files including macro expansion, as input. It extracts the information from the c language and combines the characters into lexemes groups of character that "go together". A token is assigned to each lexeme. The lexical analyzer can comprehend sql

statements used to define tokens. It also suppresses lexical problems (e.g., erroneous consonants), comments, and white space.

ii. Syntax Analyzer:

It is sometimes spoken to as a parser. The parse tree is grown. Context-Free Grammar is used to generate the parse tree after taking each syllable one at a time.

Need of Grammar:

A few projects may adequately describe the laws of programming. The above works allow us to portray the curriculum as it truly is. The input would have to be examined to see whether it respects the appropriate methodology. The parse tree can sometimes be called the derivation tree. Parse branches are often built to examine the languages provided for ambiguity. The derivation tree is governed by a set of principles.

A. An expression is any identifier.

B. An expression may be any number.

C. Any operations carried out on the provided expression will always produce an expression. An expression is, for instance, the result of adding two expressions.

D. A syntax tree may be created by compressing the parse tree.

Symbol Table in Compiler

The compiler focuses on establishing the Symbol Table, a big information structure used to keep track of the interpretation of variables. It contains data on the scope and coupling of names, and information categorized into different things, including variable and functional names, classes, objects, etc.

i. Morphological and grammatical analysis steps are already built in.

ii. The information is obtained by the compiler's analysis stages, and it has been utilised by the synthesis of different phases to produce code.

iii. The compiler makes advantage of it to maximize compile-time usefulness.

iv. The following steps of something like the compiler make use of it:

A. ***Lexical Analysis:*** Add new table elements to the table, such as new token entries.

B. ***Syntax Analysis:*** Add details to the table about featuretype, opportunity, measurement, usage, etc.

C. ***Semantic Analysis:*** Uses the table data to determine semantics, i.e., to ensure that equations and tasks are operationally sound type checking, and to appraise the table as necessary.

D. ***Intermediate Code generation:*** Use the symbol table to add information about temporary parameters and to find out how much but what kind of run-time is granted.

E. ***Code Optimization:*** Uses data from the symbol table to improve based on the machine.

F. ***Target Code generation:*** Uses the name and address of the identifier which is included in the database to generate code.

Character Table Entries

Each item in the symbol table contains characteristics connected to it that assist the programmer at various stages.

i. Items Deposited in Symbol-Table:

A. Compiler generated temporaries

B. Strings and Literal Constants

C. Labels in source languages

D. Constants and Variable Names

E. Function Names and Procedure

ii. Information Used by the Compiler from Symbol Table:

A. For parameters, whether parameter passing by value or by reference

B. Declaring procedures

C. Offset in storage

D. If structure or record then, a pointer to structure table.

E. Name and Data Type

F. Number and type of arguments passed to function

G. Base Address

Operations of Symbol Table:

The basic operations distinct on a character in Table 1 include:

Table 1: Illustrated the Different Operations and its Function.

Operation	Function
allocate	to allocate a new empty symbol table
free	to remove all entries and free storage of symbol table
lookup	to search for a name and return pointer to its entry
insert	to insert a name in a symbol table and return a pointer to its entry
set_attribute	to associate an attribute with a given entry
get_attribute	to get an attribute associated with a given entry

Implementation of Symbol Table:

The corresponding data structures are commonly employed to construct symbol tables:

i. List:

A. The names and supporting data are stored in this technique using an array.

B. All saved records remain a reference that says "available," and new categories are added throughout the order that they are received.

C. To search for a name, we begin only at top of the list and go down to the first accessible pointer; if the moniker is not found, we get the exception "usage of the undeclared name."

D. Before creating a brand-new name, be sure it does not already exist. Otherwise, an error stating "Multiple predefined names" will appear.

E. Insertion is quick (O(1)), while searching is often sluggish (O(n) for big tables).

F. The fact that it requires little space is positive.

ii. Linked List:

A. In this construction, a linked list is used. Each document is included a link field.

B. Names are evaluated in the order suggested by the link in the link field.

C. The symbol table's initial element is referenced by the address "First," which is always present.

D. Insertion is quick O(1), but searching is often sluggish O(n) for big tables.

iii. Hash-Table:

A. The most popular method for constructing symbol tables is to manage multiple tables, a hash table and then a symbol table, as part of a scrambling scheme.

B. A search algorithm is an array with a 0 to table size-1 index range. These records serve as references to that same symbol table's names.

C. We use a hash function to look up names, which returns an integer between 0 and table size1.

D. Lookup and updating may be done incredibly quickly (1).

E. The benefit is that rapid searching is feasible, and the drawback is that scrambling is difficult to use.

iv. Binary Search Tree (BST):

A. Using a binary search tree to develop a symbol table means inserting two link fields, left and right child.

B. All names are constructed as children of something like the root node, which adheres to the data structure tree's attributes at all times.

C. Insertion and searches frequently take O(log2 n) time.

CHAPTER 2

STATIC AND DYNAMIC SCOPING

Ghouse Basha M A
Assistant Professor, Department of Data Science & Analytics, School of Sciences,
Jain (Deemed-to-be University), Bangalore-27, India
Email Id- ghouse.basha@jainuniversity.ac.in

The area of a program where a variable's designation is referenced by the usage of that variable, x. keeping variables segregated in various portions of the program is one of the significant purposes of scoping. Since there aren't many short variable names and programmers tend to adopt the same patterns when identifying variables for an array index, the same variables name will be used across different scopes in any program of reasonable size. Scopes are often divided into two main categories:

A. Static Scoping
B. Dynamic Scoping

Static Scoping

A variable consistently refers to its top-level context using static scoping, which is also known as lexical scoping. This has nothing to do about the run-time call stack and is a feature of the programmed text. Secondly, static scoping makes it considerably quicker to create modular code since a programmer can understand the scope just by reading the code. On the other hand, dynamic scope necessitates that the programmer forecast every potential dynamic situation. Variables are indeed statically or lexically scoped in the majority of scripting languages, including C, C++, and Java. This means that the binding of a variable may be controlled by program text, as shown in Figure 1, and seems to be independent of the run-time procedure call stack.

```
begin
integer m, n;

procedure hardy;
    begin
    print("in hardy -- n = ", n);
    end;

procedure laurel(n: integer);
    begin
    print("in laurel -- m = ", m);
    print("in laurel -- n = ", n);
    hardy;
    end;

m := 50;
n := 100;
print("in main program -- n = ", n);
laurel(1);
hardy;
end;
```

The output is:

```
in main program -- n = 100
in laurel -- m = 50
in laurel -- n = 1
in hardy -- n = 100    /* note that here hardy is called from laurel */
in hardy -- n = 100    /* here hardy is called from the main program */
```

Figure 1: Illustrated that the Snippets for Static Scoping.

Dynamic Scoping

Global identifiers are rare in contemporary languages and, to be used with dynamic scope, refer to the identifier corresponding with the most recognized methods. Technically speaking, this suggests that the most recent binding is checked for an identifier's existence in Figure 2 since each identifier has a permanent stack of bindings.

```
begin
integer m, n;

procedure laurel(n: integer);
    begin

    procedure hardy;
        begin
        print("in hardy -- n = ", n);
        end;

    print("in laurel -- m = ", m);
    print("in laurel -- n = ", n);
    hardy;
    end;

m := 50;
n := 100;
print("in main program -- n = ", n);
laurel(1);
/* can't call hardy from the main program any more */
end;
```

The output is:

```
in main program -- n = 100
in laurel -- m = 50
in laurel -- n = 1
in hardy -- n = 1
```

Figure 2: Illustrated the Snippets for Dynamic Scoping.

Static vs. Dynamic Scoping

Static scoping occurs predominantly in the majority of programming languages. This is attributable to the reality that static scoping is simple to reason about something and comprehend by just reading the code. Just by glancing at the text in the editor, we can discover which variables are inside the scope. Dynamic scoping is more preoccupied with how the code runs than with how it is written. The stack is pushed with a broad portfolio each time a new operation is called. Both dynamic and static scoping are supported in Perl. While the keyword "local" specifies a global variable with a dynamic scope, the keyword "my" in Perl defines a global variable with a static scope.

CHAPTER 3

ERROR DETECTION AND RECOVERY IN COMPILER

Dr. Gokul Thanigaivasan
Assistant Professor, Department of Data Science & Analytics, School of Sciences,
Jain (Deemed-to-be University), Bangalore-27, India
Email Id- t.gokul@jainuniversity.ac.in

During this stage of compilation, all potential user mistakes are found and sent to the participants in the form of error messages. The "Error Handling procedure" entails finding mistakes and informing viewers of them and the capabilities of either an error handler.

A. Detection
B. Reporting
C. Recovery

Classification of Errors

The blank columns in the symbol in Figure 1 are an error, and indeed the parser should be able to detect and notify the user of any errors inside this program. The parser can handle any failures that come up while still analyzing the remainder of the input. And although the parser is responsible for the overall management of checking for errors, accidents may still happen during the requirements phase.

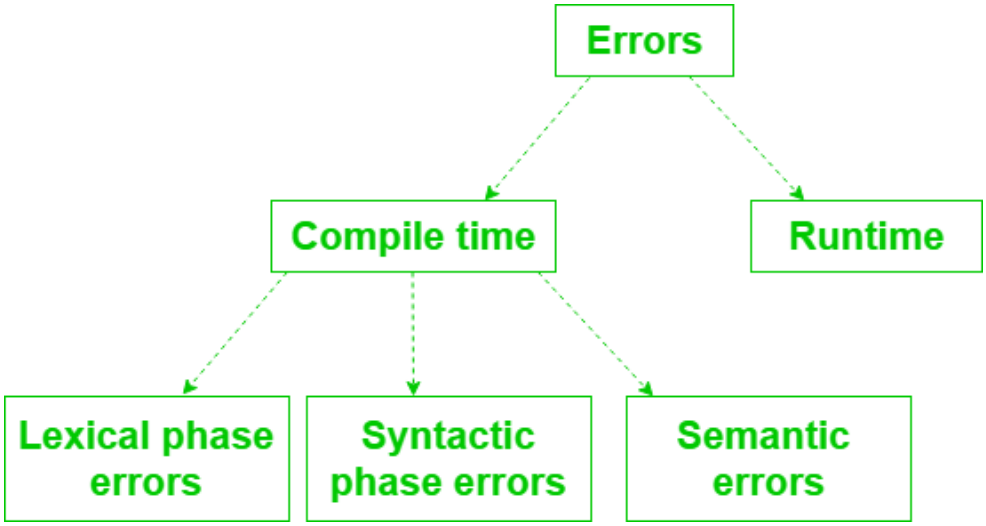


Figure 1: Illustrated the Different Types of Errors.

So, there are many types of errors and some of these are:

Types of **Sources of Error:** There are three types of error: logic, run-time and compile-time error:

i. ***Logic Errors:***

Logic Errors arise when programs execute poorly and therefore do not terminate improperly (or crash) (or crash). A logic mistake may generate significant or undesirable outputs and perhaps other behaviors, even if it is not immediately obvious.

ii. Run-time Error:

Run-time types of errors occur while a program is really being executed and are often induced by an unfavourable operating condition or incorrect input data. The availability of proper memory to operate an application or a resource conflict with another software consequent logical error is an occurrence of this. Logic errors arise when processed code does not provide the desired outcome. Logic faults are best managed by rigorous software debugging.

iii. Compile Time Errors:

Compilation-time errors appear before the programed is run, at the time of compile. Syntax mistake or missing file reference that stops the application from correctly compiling is an example given in this Figure 2.

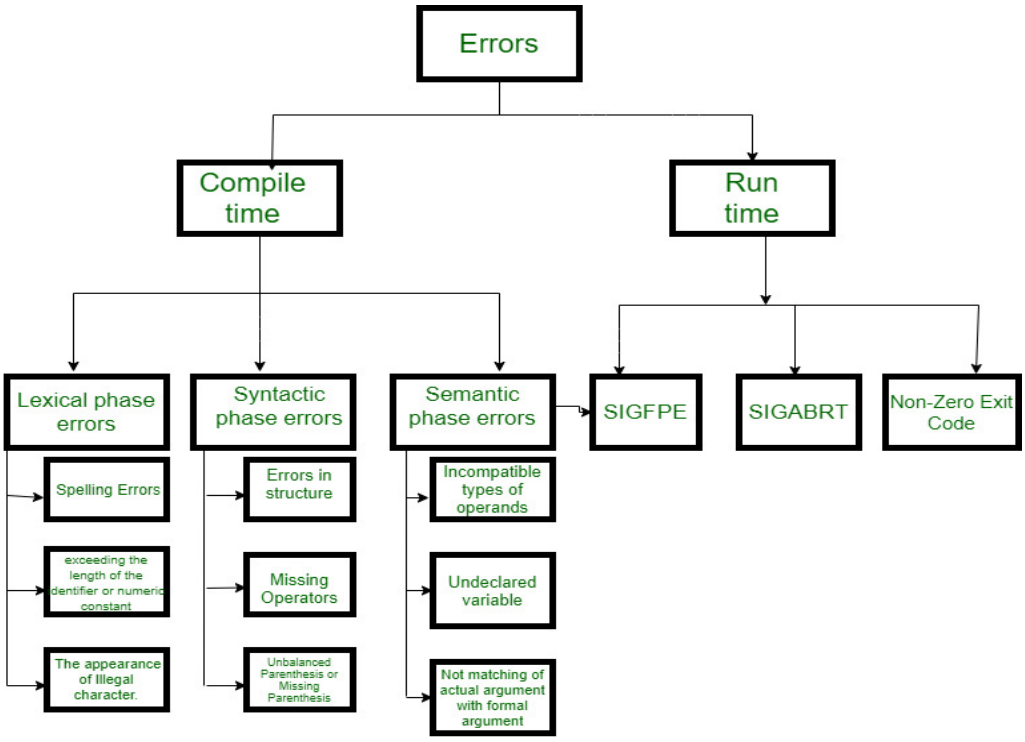


Figure 2: Illustrated the Compiling of the Errors.

- ***Panic Mode Recovery:***

This is the simplest method of errorrecovery and it stops the parser from forming infinite permutations when recovering an error. The parser discards the input emblem one at a time until one of the predefined like the end, semicolon set of synchronization tokens is often the statement or expression terminators discovered. When there aren't often repeated mistakes in a single statement, this is sufficient. Example: Take the incorrect formula (1 + + 2) + 3. Panic-mode recovery: Go ahead and go on to the next integer. Bison: To specify how much input to skip, use the special terminal error.

- ***Phase Level Recovery:***

When an error is recognized, the parser conducts local adjustments on the remaining input. In the ability to continue parsing the remainder of the statements after running through an error, a parser overcomes the disadvantage of the remaining input. You may fix the errors by removing superfluous punctuation marks, switching out commas for semicolons, or adding any that are missing. Maximum caution can be used throughout the adjustment to avoid running an endless loop. Any prefix that is recognized in the remaining input is replaced with a string each time. The parser might even go on with its execution in this manner.

- ***Error Productions:***

If the user is aware of widespread grammatical errors as well as blunders that result in erroneous formulations, they may include the error manufacturing approach.

When this method would be used, parsing may proceed even when problems are produced. Instead of writing 5*x, use 5x.

- ***Global correction:***

The parser examines the whole programming and seeks the closest approximation that is error-free to recover after incorrect input. The most accurate match was the one with the fewest token insertions, deletions, and alterations. Due to its spatial and temporal complexity, this approach is not viable.

FIRST and FOLLOW in Compiler Design

First

In the earlier essay on Orientation to Syntax Analysis, which is a fairly difficult technique to conduct, we observed the need of going back. There could have been a simpler approach to solving this issue and The compiler should make a good choice about which development rule to use if it had an understanding of the "first character of the string issued when a production rule is applied" in preparation and could equate it to the contemporary campaign or token in the participation sequence it is to see as:

S ->cAd

A ->bc|a

And the input string is “cad”.

It would have unheeded the power generation rule A->bc (because 'b' is the leading attractiveness of the string manufactured by this compression stage, not 'a'), and used the construction rule A->a (because 'a' is the paramount attractiveness of the piece of rope fashioned by this compression stage, and is the matching as the contemporary attractiveness of the comparison purposes above if it had known because after construing atmosphere 'c' in the involvement string and removing S->cAd, the next protagonist in the participation sequence.

Therefore, it is guaranteed that the compiler or parser can adequately apply the relevant construction rule to generate the accurate arrangement tree for something like the specified input string if it is aware of the first attractiveness of something like the filament that may be retrieved by smearing a development rule.

Follow

There is still a challenge for the parser and to grasp this issue, examine the grammar below.

A ->aBb

B -> c | ε

And understand the inputstring is “ab” to parse.As the first troposphere in the participation is a parser smears the rule A->aBb.

A

/ | \

a B b

The parser now looks for the upper left of the participation string, which is b. Since B is the Non-Terminal, no string issues to the attention from B may have b as the initial character. However, the Grammar has included a construction rule B ->; if this is used, B will appear and the parser will get the contribution "ab," as is seen beneath. However, the parser may only use it if it is aware that the number that follows ‘B’ in the construction rule coincides with the currently input characters.B follows Non-Terminal B in RHS of A ->aBb, i.e., FOLLOW (B) = b, and the symbol being read from the input is also b. As a response, the parser follows these rules. Additionally, it may capture the string "ab" from the inputted grammar.

A A

/ | \ / \

a B b => a b

|

ε

So, FOLLOW can style a Non-terminal disappear out if required to engender the filament from the described tree. The implication is that in order therefore for parser to correctly perform the required rule at the appropriate venue, we must identify FIRST and FOLLOW sets for a learned in order.

CHAPTER 4

CODE OPTIMIZATION IN COMPILER DESIGN

Dr. Santosh S Chowhan
Assistant Professor, Department of Data Science & Analytics, School of Sciences,
Jain(Deemed-to-be University), Bangalore-27, India
Email Id- santosh.sc@jainuniversity.ac.in

Code optimization, a programmer transformation method, is used in the synthesis phase to try to optimise the intermediate representation by making it use fewer resources (such as CPU, Memory), which should ultimately to faster-running machine code. The following goals should have been achieved through the compiler optimization process:

A. The optimization would have to be accurate and cannot in some manner alter the program's intent.
B. The software would run faster and perform better after adjustment.
C. Keep the step function reasonable.
D. The entire compilation step shouldn't be delayed by the optimization procedure.

Time to Optimize

Optimization of the code is often performed at the end of the development stage since it reduces readability and adds code that is used to increase performance.

Need of Optimize

Optimizing an algorithm is beyond the scope of the code optimization phase. So the program is optimized. And it may involve reducing the size of the code. So optimization helps to:

A. Reduce the space consumed and increases the speed of compilation.
B. Manually analyzing datasets involves a lot of time. Hence we make use of software like Tableau for data analysis. Similarly manually performing the optimization is also tedious and is better done using a code optimizer.
C. An optimized code often promotes re-usability.

Types of Code Optimization:

The optimization process can be broadly classified into two types:

i. Machine Independent Optimization:

To achieve a better target code, the intermediate code is improved during this code optimization step. There are no CPU registers or definite memory addresses used in the portion of the intermediate representation that is translated here.

ii. Machine Dependent Optimization*:*

Machine-dependent determination was carried out after the target code has been created and changed to accommodate the architecture of the specified machine. It may use absolute

therefore being rather than relative ones and incorporates CPU registers. Machine-dependent compiler optimizations attempt to use the memory hierarchies to the fullest.Code Optimization is done in the following different ways:

Compile Time Evaluation:

Compile-time function execution, also known as compile time objective function value or generic constant expresses, is the capacity of a computer to carry out a function at convert time then instead of compile it to machine code and operating it afterwards.

Variable Propagation:

One local code proposed methodology used in compiler design is the transmission. It may be referred to as the process of changing a variable's uniform distribution in an expression. To put that another way, if a value is given a known characteristic, we can just swap out the value for the constant. When a variable is utilized, the constants associated with it may be replaced and transferred across the flow diagram. Compilers do constant dissemination based on the findings of reaching description analysis, which implies that if all elements have the same constant assigned to them by their approaching definition, the variable has a uniform distribution and may be exchanged with the consistent.

Constant Propagation:

The technique of replacing known constant values in expressions is known as constant propagation. Constant propagation removes situations when values are easily assigned to another variable by copying them from one place or variable to another.

Constant Folding:

Constant folding is very much an optimization method that gets rid of expressions that compute constants that may be known in advance of something like the execution of the code. These equations or expressions often only use model parameters or constant value referencing for the variables.

Copy Propagation:

Dead code is often formed by copy propagation and might even be removed. By preventing the execution of irrelevant statements during run time, lifeless code removal increases functional recovery. Thus, in this article, a unique approach termed hash-based value enumeration is used to integrate the two strategies.

Intermediate Code Generation in Compiler Design

According to the analysis-synthesis architecture of a compiler, the whole front end converts a binary code into an independent intermediate representation, which is then used by the compiler's leading edge to create the target code that somehow a machine can understand. The use of machine-independent intermediate representation has significant capabilities:

A. Portability will be implemented due to the hardware intermediate code. For instance, if a compiler just transfers the source language to the targeting machine language without offering the opportunity of producing intermediate code, a complete native compiler will be needed for every windows computer. Because the compiler itself undoubtedly underwent some decisions that are consistent with the hardware specs.

B. Products will affect is made easier.

C. By optimizing the intermediate representation, source code enhancements that enhance source code throughput are faster and more reliable.

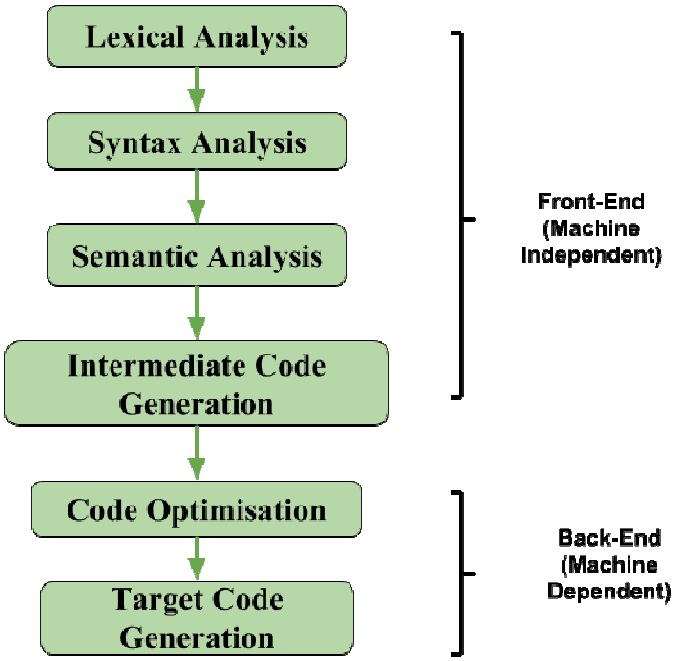


Figure 1: Illustrated the Intermediate Code Generator in Compiler Design.

There will be n code generating and optimizers for every one of the n target machines if the machine program is written straight from source code, although there will only be one compiler if the machine-independent form of technology is used as shown in Figure 1. Language-specific intermediates code, such as byte-code for Java, is however possible. Independent (three-address code). Commonly used transitional code formats also provide:

i. Postfix Notation:

Also known as suffix notation or reverse Polish notation. The postfix notation for the identical expression inserts the operator at the right end as ab +. The conventional (infix) method of expressing the sum of "a" and "b" is with an operator in the middle: a + b. In general, when "+" is applied to the values represented by "e1" and "e2," postfix notation is produced by "e1e2 +," where "e1" and "e2" are any postfix expressions and "+" is any binary operator [4]. Postfix notation does not need parentheses since there is only one possible method to decode a postfix expression given the location and number of arguments for each operator. The operator comes after the operand in postfix notation.

Example 1: The postfix representation of the expression (a + b) * c is: ab + c *

Example 2: The postfix representation of the expression (a – b) * (c + d) + (a – b) is: ab – cd + *ab -+

ii. Three-Address Code:

A three-address statement uses no more than three references two for the multiplexer and one for the result. A three-addressing code is a set of three address declarations. Three address statements are of the form x = y op z, with addresses for x, y, and z memory locations. A statement may sometimes include less than referring, but it is still referred to as a three-addressed statement.

Example: The three address codes for the expression a + b * c + d: T 1 = b * c T 2 = a + T 1 T 3 = T 2 + d T 1, T 2, T 3 are temporary variables.

There are 3 ways to represent a Three-Address Code in compiler design:

- Quadruples
- Triples
- Indirect Triples

iii. Syntax Tree:

A reduced counterpart of a parse tree is what a syntax tree is. The vertices of the graph are operators, while the connected components are operands, and the operator and keywords nodes of the parse tree are transported to their parents, replacing a chain of singular productions with a single link inside this syntax tree. Put parentheses around the expression to create a syntax tree; this makes it clear which argument should arrive first.

Example: x = (a + b * c) / (a – b * c)

Object Code in Compiler Design

Assume you have a C program and therefore provide it to the compiler, which generates the outcome in assembly code. The assembler will now receive the machine language code, and it will generate some code designated as object code for you and steps and process are mentioned in Figure 2.

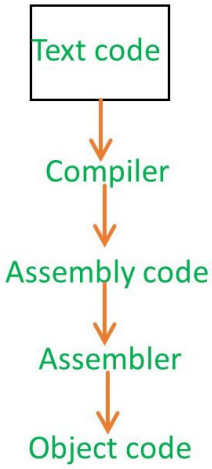


Figure 2: Illustrated that the Object Code in Compiler Design.

However, you won't utilize both compiler and the assembler when you build an application. Simply take the algorithm and provide it to the processor, and the compiler will output

programming that is ready for immediate implementation. The loader, linker, and compiler are all intertwined within the assembler. As a result, the compiler software itself preserved all the modules together. Therefore, when you use compiler collection, you call the compiler, the disassembly, the linker, and the loader within a week of finishing with the complex formation and loader. Once you activate the compiler, your object code will be present on the hard drive. This object code has many parts:

- ***Header:***

The header will list the different aspects that are present in this executable program and then link to those aspects. As a result, the header will describe where the text segment will begin, along with a link to it, as well as where the data segment will begin and where the translation information and symbol identification are located. Which is nothing more than an index; imagine a textbook where the meta-description lists the page numbers on whereby every subject is covered. Similar to all of this, the header will provide the palaces where another piece of information is situated. So that it will be convenient to jump right into such segments later onwards for other technologies.

- ***Text Segment:***It is nothing but a set of instructions.
- ***Data Segment:***

Whatever data you used will be in the data section. If you implemented a constraint, for instance, it will likely have been included in the data segment.

- ***Relocation Information:***

To define anything while writing a program, we often utilize symbolism. Presuming you have guidelines 1, 2, 3, and 4, continue reading.

Figure 3: Embellishes that the Allocation Information.

The code will now be compiled to object code and L4 will be transformed with Goto-4 if you mention anywhere Goto-L4 even if you don't write Goto statement in the high-level language, the compiler's program will write it. Now, Goto-4 for level L4 will function as anticipated as long as the application loads from address zero as displayed in Figure 3. However, the operating system will often take up the very first portion of the random access memory. Even if it is not devoted to that same operating system, there may already be another process functioning at address 0 in that case. Consequently, if the computer has to be put through into main memory, it may be imported somewhere when you load it into memory. If the new base address is 1000, all the addresses must be modified, which is referred to as reorganization in Figure 4.

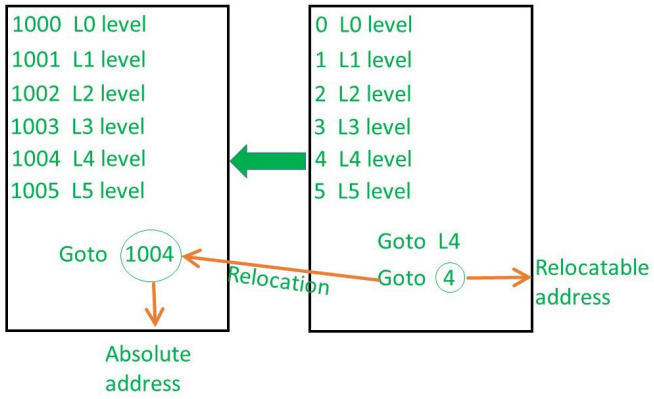


Figure 4: Illustrated the Reallocation Information.

The original address is known as the Reloadable address and the final address which we get after loading the program into the main memory is known as the Absolute address.

Pointer Analysis

For long years, pointers have been a headache for compilers. The problem illustrates how complex it is to understand pointers as things that connect to other data and their intricate interconnections. Pointer analysis has always been a crucial component of compiler design since the early 1970s, and several methods have been proposed for it everywhere in time. In this article, we'll discuss a particular approach known as "pointer-analysis," which is presently recognized as a respectable framework for developing pointer-analyzing systems but that's not truly the optimal choice.

The difficulty of Pointer Analysis

Even though pointers are dynamic, relationship analysis is a difficult task. They would either be direct or indirect, and they may be aliased pointers to the same variable. You must also be knowledgeable so that your compiler can create effective code for it. Points are created from those other pointers; for instance, if a variable "A" has one reference point and a second variable "B" has two visual references, working on "A" may benefit from one of those frames of reference, while working on "B" may not be at all relevant when coding an example. The updated following model point is essential because it will notify the user of the type of physical memory that apartment buildings the data that your source code is addressing, which is essential if this data format changes in the future. For this reason, your compiler must be ready to invent code for a pointer. Users cannot simply command the compiler to transform

interconnections into memory addresses because if they do, their applications will break anytime their data types change and, correspondingly, whenever the values at any of those places change. This is why being able to compose code in Python for connections is necessary.

Model for Pointers

In Java and C++, pointers and referencing are used. There are two different kinds of pointers in C: weak and strong. The place that the sluggish pointer links to may no longer be an issue for instance, if it has been erased from existence. Strong pointers always correspond to legitimate memory addresses. There are two distinct types of references in both languages: expensive references, also known as standards, and non-constant references, also recognized as ordinary references. These two ideas vary dramatically because constant references lose their value between calls whereas non-constant references do, but they both enable users can access the same object across various functionalities or methods without recognizing what kind of object the consumer reference.

Flow Insensitivity

A pointer may point to every kind of object, according to the principle of flow insensitivity. This characteristic, which is associated with reference equivalence, enables us to think about pointers more broadly. Using the same kind of hierarchy to describe pointers and connections is the most apparent use for this attribute. It may also be used for other things, such as encoding regular present in all cells or evaluating an expression in real-time, as when a user has an array with five components and wants to know whether there are much more than three elements in it. A pointer may refer towards another pointer, according to the second characteristic, which again is called pointer indirection. This attribute makes it possible to use pointers to construct recursive structures like trees.

Aliasing, the third attribute, means that two pointers may refer to much the same object. This characteristic makes it possible to use pointers to describe shared memory. The fourth criterion, mutability, states that a connection may be modified. This property enables us to use pointers to represent modifiable things like strings and arrays. The null pointer, which stands for something like an invalid pointer, has the fifth attribute. This is a unique identifier that cannot be referenced by all the other values and has no pointer. Empty lists, empty arrays, and other objects without so much as an element or members may all be written using it.

The Formulation in Data-log

A logic computer program called Data Log may be used to represent and address issues in a variety of fields. To examine reference operations in a program, use the Methodology for Pointer Analysis in Datalog. When we adjust pointers or references, it helped us to understand which sections of the program need updating. As per the formula, if a user owns the two application areas P1 and P2, which are included in Figure 5, then:

```
For every transition T from state A(x) => B(y).
If P1(A)  → B(y) then there must exist another transition T' such
that (P1(T')-P2(T)) > 0
If P2(A)  → B(y) then there must exist another transition T' such
that (P1(T')-P2(T)) < 0
```

Figure 5: Illustrate the Formulation of the Datalog.

The formula is useful to us because it allows us to reason about pointer manipulations in a way that is not possible with ordinary predicate logic.

Consider the following Example

The program contains two references: ref1 and ref2. They are both initialized to null, which means that neither reference has any value associated with them. We then modify ref1 so that it points to an object of type T and initialize ref2 with another reference r. We want to know what the consequences of these changes are. We can use pointer analysis formulas to find out. The first formula tells us that if we make a transition from state A (null) ⇒ B(T), then there must exist another transition from state B(T) => C. The second formula tells us that there is no way for the program to reach state C unless it has already reached state B [5].

Using Type Information

The compiler can use this information to determine whether a pointer is safe to dereference, and therefore whether it makes sense for it to be converted into an object reference (e.g., by calling malloc). It can also use type information to determine whether a pointer is valid as an argument for "**memcpy()"** or "**memcmp()"**. In both cases, if there is no dereference able object on the stack that matches what the user passed as a source parameter, then this means that whatever value the user is copying from or comparing against ends up being copied into memory somewhere else instead; there's no point doing this operation at all.

Semantic Analysis in Compiler Design

The third stage of the compiler process comprises semantic analysis. Semantic analysis verifies the grammatical accuracy of programming declarations and statements. It is a group of functions that the parser invokes when required to carry out the grammar. To verify the correctness of the provided code, a symbol table and the syntax forest from the previous step are both required. Semantic analysis includes a designated person, which helps the compiler ensure that each operator has the necessary operands.

Semantic Analyzer

To determine if the presented programming is semantically compliant with language definition, it employs a grammatical tree and symbol table. It collects type information and transmits it either in a symbol table or a semantic tree. The compiler will then utilize additional type information to generate an intermediate representation.

Semantic Errors

Errors recognized by the semantic analyzer are as follows:

A. Type mismatch
B. Undeclared variables
C. Reserved identifier misuse

Attribute Grammar

The use of additional data (attributes) to one or more context-free punctuation and grammar non-terminals in addition to giving context-sensitive knowledge is known as attribute grammar. Each attribute's range of potential values is clearly stated, and examples include integer, float, character, string, and expressions. A computer's idiomatic syntax and semantics may very well

be specified with the aid of attribute grammar, which should be a tool for giving context-free sentence construction semantics. When understood as a parse tree, attribute sentence construction may convey values as well as information amongst tree nodes.

$$E \rightarrow E + T \{E\text{-value} = E\text{-value} + T\text{-value}\}$$

The semantic rules that define how the grammatical should be understood are included in the right portion of the context-free grammar. The non-terminal E and T's values are combined in every case, and the result is transferred to the non-terminal E. When parsing values from the respective domain, semantic properties may be applied to them and evaluated when requirements are met. The characteristics may be roughly divided into two main groups: synthesized attributes as well as inherited attributes, depending according to how they acquire their values.

Synthesized attributes

These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

$S \rightarrow ABC$

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Functions of Semantic Analysis

- **Type Checking**

Ensures that data types are used in a way consistent with their definition.

- **Label checking**

A program should contain labels and references.

- **Flow Control Check**

Keeps a check that control structures are used properly(example: no break statement outside a loop).

Static and Dynamic Semantics

- **Static Semantics*:***

It is named so because these are checked at compile time. The static semantics and meaning of program during execution, are indirectly related.

- **Dynamic Semantic Analysis*:***

It defines the meaning of different units of program like expressions and statements. These are checked at runtime, unlike static semantics.

Run Time Environment

When an application is developed in source code, it is only a collection containing text (code, statements, etc.), and it needs activities to be performed on the target computer to operate effectively. Memory support is necessary for software to carry execute instructions. Procedure names, constants, and other names found in a computer must be mapped at runtime to that same correct memory address. A program that is functioning is referred to as downtime. The target machine's application program, which might contain software components, data structures, etc., is in a state where it is available to the applications that are now operating on it. A package called a runtime network of support is created mostly by the application program itself and makes it easier for frameworks to communicate with development environments. A computer program, manages physical memory and de-allocation.

Activations Tree

A service is a sequence of processes created from either a series of instructions. A procedure's requirements are carried out in order. Whatever within a procedure's start and end special characters are referred to as the procedure's body? The body of the process is composed of several finite operations and a procedure identifier. A procedure's actuation is when it is carried out. All the parameters needed to call a procedure are included in an activation record. The units listed below may be found in create dynamic: depending on the source language used.

The stack, sometimes described as the control stack, stores the excitation record for each function that is conducted. When one program calls another, the caller's execution is halted until the called process has been completed. The calling procedure's descriptive information is now kept mostly on the stack. We assume that the control signal flows sequentially, with control within one operation passing to another as it is invoked. When a triggered procedure completes its execution, management is returned to the caller.We will use the accompanying line of code as an example to thoroughly grasp this idea:

```
. . .
printf (“enter your name here”);
scanf (“%s”, username);
show data (username);
printf (“press any key to continue”);
. . .
int show data (char  *user)
{
printf  ("Your name is %s", username);
return 0;
}
. . .
```

The activation tree for the provided code is shown in Figure 6 below.

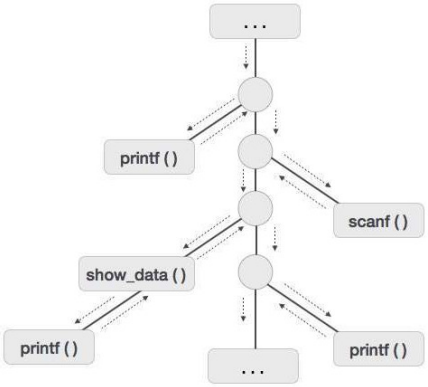


Figure 6: Illustrated the Activation tree.

Because procedures are now understood to be run depth-first, stack allocation is the most ideal kind of storage for technique activations.

Allocation of Storage

The following entities' runtime resource needs are managed by that the runtime environment:

A. **Code:** The text portion of software that is constant throughout the performance. Its memory needs are recognized at the time of compilation.

B. **Procedures:** Although each text portion is static, they are sporadically called. Stack storage is thus used to control protocol calls and activations.

C. **Variables:** Only if they are global or constant, variables are only accessible during runtime. The management of data storage and de-allocation for elements during runtime is handled by a heap dynamic memory technique.

Fixed Allocation

The compilation material in this allocation strategy is allocated to a place in memory and is not unaffected by how the program is run. Runtime implementation for ram and de-allocation is not necessary since such amount of memory needed and where it can be stored are announced in advance.

Allocation of Stack

Stack memory allocation is also used to control procedure arguments and their activations. It proceeds using the last-in, first-out (LIFO) methodology, and recursive function calls benefit enormously from this routing algorithm.

Allocating Heaps

Only during runtime seem to be variables that are local to an operation assigned and released. Memory is performed automatically to variables via heap allocation, which then claims your memory back when the objects are no longer needed. Both stacks as well as heap memory may expand and diminish dynamically and unpredictably, except maybe areas of memory that are statically allocated. As a result, they cannot be given a set volume of RAM inside the system. The text portion of the rules is allotted a set amount of capacity, as seen in the image above. At the two ends of something like the program's entire memory allotment, stack and heap resources are placed. The opposite of both contraction and proliferation.

Different Parts of Compiler

Compilers are sophisticated software because of this, they are quite often divided into several levels that require specific known as passes that interact with one another through temporary files. But again the passes themselves are only one step in the compilation process. Addition than composition, there may be other phases associated with the process of generating an executable code from a source-code file. In truth, certain operating platforms can wait until a program is loaded at run-time before really constructing an executable image. Driver applications like UNIX's cc or Microsoft C's cl, which conceal a sizable percentage of the compilation process from you, substantially complicate the matter.

These driver programmers serve as executives, managing these same numerous compiler component programmers sufficiently that you are unaware that they even exist and are utilized.

For the sake of this book, I'll refer to a "Compiler" as a computer or group of programs that convert one language's source code into perhaps another language, in this instance assembly language. The compiler does not include syntax, linker, and other tools.The graphic below Figure 7, depicts the structure of the typical four-pass compiler. The first pass is preprocessing. Preprocessors often carry out different housekeeping duties that you prefer not to bother the actual compiler with, such as macro replacement, comment cleanup from source code, and various sanitation jobs.

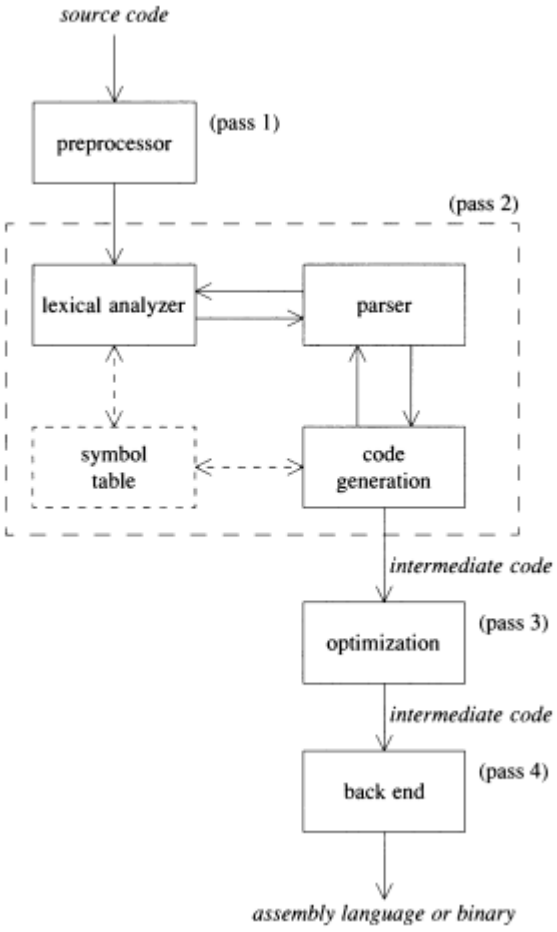


Figure 7: Represented the Structure of a Typical Four-Pass Compiler.

The second pass is where the compiler's heart is. It converts source code into an input representation that is similar to assembly language using a morphological analyzer, grammar, and code generator. The optimizer, which is employed in the third pass, enhances the quality of the created intermediate code. The back end, which is used in the daily attempt, converts the optimized code into an actual programming language or some other kind of binary, programming language. Of course, this structure is amenable to several changes. Many compilers lack preprocessors; several create assembly language in the second pass; yet others directly improve assembly language; still, others automatically generate binary instructions without first processing through an ASCII input representation like an assembler.

CHAPTER 5

THE LEXICAL ANALYZER

Dr. Thirukumaran Subbiramani
Assistant Professor, Department of Data Science & Analytics, School of Sciences,
Jain (Deemed-to-be University), Bangalore-27, India
Email Id- s.thirukumaran@jainuniversity.ac.in

The discrete job used in the synthesis process is referred to as a step. Typically, a pass is composed of several steps. The input is transformed into a form that is more useful to the rest of the compiler during the lexical analysis software phase also known as the scanner or tokenizer of the compiler. The lexical analyzer views the alpha channel as a collection of fundamental linguistic building blocks known as tokens. In other words, a token is a single lexical unit. Tokens in C include the words for while or for, >, >=, >>, and >>=, names and numbers, and so on. A lexicon is the base language that makes up a token. Note that the link between vernacular and token is not exact. While a while token necessarily corresponds to a single syllable, a name or number token, for example, may correspond to several different words.

Interwoven tokens (such as the previously used >, >=, >>, and >>=) exacerbate the problem. A lexical analyzer often locates the token that matches the longest syllable; many languages incorporate this behaviour within their language specifications. Instead of more than two tokens, a shift token is recognized after the input >> . Tokens are generated from lexemes by the lexical analyzer. Internally, tokens are often handled as distinct integers or as an enumeration type. In this example to distinguish between multiple names and number tokens, a lexicon is required in addition to the token.

The selection of a token set is one of the initial planning choices that can have an impact on the organization of the overall compiler. For example, the symbols >, >=, >>, and >>= can be handled either as four tokens or as a single comparison-operator token; Lexemes are used to separate tokens. As an alternative, you may aggregate many symbols into a single token or use a separate token for every read or write operation. The first approach may occasionally be employed to make code creation more understandable. However, if there are too many tokens, the parser will become bigger than anticipated and be more possible to establish.

No one can definitively say which is superior, but after reading through this book, you'll be aware of the design elements and be able to come to effective judgments. In general, arithmetic operators with the same precedence and corresponding characteristic as type-declaration keywords like intand char may also be combined. The lexical analyzer often functions independently of the remainder of the compiler and requires only a few subroutines, generally one or two, along with global variables. The lexical analyzer is called each time the parser needs a new token, and it returns both token and the lexeme that correlates with it. Because the actual input method is masked from the parser, changing or replacing the lexical analyzer won't have a consequence on the remainder of the compiler.

The Parser

High-level languages like C are translated into low-level languages like 8086 assemblies via compilers. He translates across languages b because of this, linguistics serves as a primary source of inspiration for most of the subject's theoretical components. An example of such a concept is parsing. When an English sentence is broken down into its constituent parts for grammatical study, it has been parsed. For example, contemplate the phrase as display in Figure 1:

Jane watches spot run and has Jane as the subject and spot as the consequent ("sees spot run"). The predicate in tum is mainly composed of the verb "sees," the direct object "spots," and the semicolon that modifies the direct object ("run"). See how a common sentence diagram, the sort you learnt to construct in sixth grade, illustrates a statement in the example below. A compiler executes the same process of disassembling a phrase into its parts during the parser step, albeit it often presents the interpreted text as a tree rather than a language diagram.

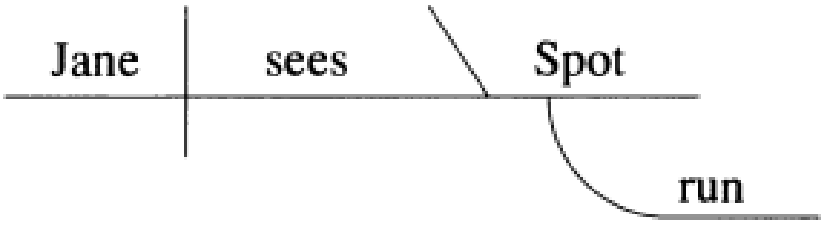


Figure 1: Represented that the Parching Phase.

Given that the sentence diagram itself highlights the syntactic connections between the phrase's individual components, this kind of graph is suitably referred to as a syntax diagram (or, if it takes the form of a tree, a syntax tree). The syntax tree may be expanded to show morphological and syntactic structures, however. A parse tree is the name of something like the second structure. The parse tree for our earlier sentence diagram is illustrated in the Figure 2 below. Because a tree structure is easier to represent in a computer program, it is used in this case rather than a language diagram.

Despite it having the same meaning as in English, "phrase" is a technical word. It is made up of several tokens that adhere to a specified grammatical framework. In the context of a compiler, the phrase often refers to a whole computer program. The resemblance is noticeable in a languages like Pascal, which imitates both English syntax and punctuation. The conclusion of a Pascal program is indicated by a time frame, exactly as it is in English. A semicolon is used as a punctuation mark to designate two full concepts, much too how an English semicolon separates two distinct sentences.

In summary, a parser, which is a group of subroutines, generates a parse tree to reflect the results by making the phrase being parsed. In other words, the parse tree represents the sentence hierarchically, beginning at the tree's root with both the fundamental information about just the phrase and moving down to the leaflets, which include the sentence's actual tokens. Although some compilers construct a physical parse tree made up of structures, pointers, and other aspects, the majority of compilers model the parse tree. Other parsing methods only keep track of their position in the tree; we'll see how this happens in a minute. They don't generate a physical tree. The parse tree itself, however, is a very helpful construct for comprehending how the interpreting process works.

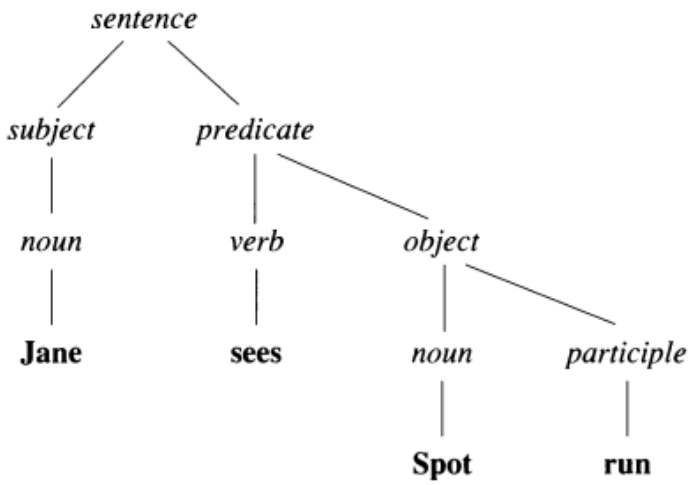


Figure 2: Represented that the Physical Tree.

The Code Generator

The last part of the compiler itself is the code generator. It is erroneous to isolate this step from the processor itself since most compilers generate code as the parse progresses. In other words, the generated code is produced by the same procedures and functions that process the input stream. But in some compilers, the parser creates a parse tree for the whole input file, which is ultimately walked by a different code generator. The creation of an interpreted language of the input by the parser, from which an optimizations pass may rebuild the syntax tree, is a third possibility. In contrast to a linear program code, a syntax tree makes certain optimizations easier. The optimizer may navigate the revised syntax tree one more time to produce the code. Although compilers may directly create object code, they often delegate code production to another program. Instead of developing machine code directly, they make a program in an intermediate representation that is translated into true machine language via a compiler's back end.

An intermediate language designed to carry out certain tasks, such as optimization, may be compared to a super assembly code. Intermediate languages come in a wide range of variations, each with a unique purpose. An approach to compiler construction using transition languages has advantages and disadvantages. The main flaw is the slowness. A parser that goes straight from syllables to binary object code will be very quick since processing intermediate code requires an additional step, which may enable the compilation time to rise by two times. However, the rewards often exceed the disadvantages in the sense of speed. They may be summed up as flexibility and adaptability.

Some parser improvements are possible, including direct continuous folding, which evaluates consistent expressions at the build phase rather than at the run time. The majority of optimizations, however, are challenging for a parser to do, if not impossible. As a consequence, the parsers provide optimizing compilers with an intermediates language that is simple to optimise in a subsequent run. You can speak whatever intermediate language you like. A single lexicalanalyzer or parser front end may be used to produce code for several different machines by providing distinct back ends that translate a commonly used measure language into machine-specific assembly language. On the other hand, after parsing numerous high-level languages, you may design a variety of front that comes to an end that all generate the same intermediate

language. Compilers for several languages may utilize a single optimizer as well as a back end when using this technique.

Another place where intermediate languages are in incremental compilers or interpreters. These program execute intermediate code directly without first converting it to binary, which decreases the time required to develop and link a real program. An interpreter may provide you with a better debugging environment since it can check for problems like out-of-bounds column indexing at run time. The output code produced by the compiler created in Chapter Six is written in an intermediate language. Although the language itself is treated in full in that chapter, I believe it requires a basic summary here since I'll be utilizing it indiscriminately for programming constructs for the remainder of this book. The majority of the directions on a typical computer often one or two are immediately translated into a limited percentage of assembly-language instructions in an intermediate language, which is only a subset of the C language. Examples include the following:

x=a+b*c+d is translated into something like this:

```
t0= _a;
tl=_b;
tl  *= _c;
t0 += tl;
 t0 +=_d;
```

The compiler allows the temporary variables "t0" and "t1" in the code above to hold the outcome of an expression that has only been partly evaluated. They are referred to as anonymous temporaries and are often referred to as just temporaries. The compiler adds an underscore to the name of a declared variable to distinguish it from variables produced by the compiler, such as 't0' and 'tl,' whose names do not. Because it's so close to C, I'm going to employ intermediate language for the time being without a conventional English meaning. Remember that the intermediate language is essentially a programming language with syntax similar to C, thus it would be of little benefit to convert outstanding C into poor C.

Representing Computer Languages

A compiler is like every other program in that some sort of design abstraction is useful when constructing the code. Flow charts, Warnier-Orr diagrams, and structure charts are examples of design abstraction. In compiler applications, the best abstraction is one that describes the language being compiled in a way that reflects the internal structure of the compiler itself.

Grammars and Parse Trees

Linguistics is also used to denote the most popular approach to formally documenting a programming language. This approach, known as formal grammar, was developed by M.I.T. K. and was created by Noam Chomsky and used by J.W. Backus as the first Fortran compiler for analyzing computer programs. The most common way to express formal grammar is in a modified Backus Naur form or BNF for short. Starting from a collection of tokens called terminal symbols and a set of definitions called non-terminal symbols, a rigorous BNF representation is constructed. Definitions provide a framework that allows the representation of any legal construct in the language. The:: = operator, which can be interpreted as "is defined as"

or "goes to", is the only supported operator. For an English sentence, the following BNF rule can serve as an introduction to grammar: The following: = subject predicate A subject and a predicate are components of a sentence. The phrase "a sentence follows a subject followed by a predicate" is another option. Each such rule is called a production. The left side of the output is represented by non-terminals to the left of ::= , and the right side by everything to the right of ::= The grammars used in this book always have a single, non-terminal symbol on the left-hand side of the production, and every non-terminal used on the right-hand side must likewise be used on the left-hand side. All symbols, including tokens in the input language that are not present on the left, are known as terminal symbols. Once each terminal symbol is defined, an actual grammar moves on to the next one. For example, the grammar might continue:

Subject=noun

Noun=JANE

Where JANE is a terminal symbol a token that matches the string "Jane" in the input. The strict BNF is usually modified to make a grammar easier to type, and I'll use a modified BNF in this book. The first modification is the addition of an OR operator, represented by a vertical bar (1). For example,

noun ::= JANE

noun ::= DICK

noun ::= SPOT

is represented as follows:

noun ::=DICK | JANE

I use the ~ in most of this book. I also consistently use italics for nonterminal symbols and boldface for terminals symbols such as + and * are also always terminals-they'll also be in boldface but sometimes it's hard to tell. There's one other important concept. Grammars must be as flexible as possible, and one of the ways to get that flexibility is to make the application of certain rules optional.

A rule like this:

noun ::= DICK | JANE | SPOT

Similarly, a → is often substituted for the::= as in:

noun → DICK | JANE

I use the → in most of this book. I also consistently use italics for nonterminal symbols and boldface for terminals (symbols such as + and * are also always terminals-they'll also be in boldface but sometimes it's hard to tell). There's one other important concept. Grammars must be as flexible as possible, and one of the ways to get that flexibility is to make the application of certain rules optional. A rule like this:

article→ THE

Says that THE is an article, and you can use that production like this:

object → article noun

In English, an object is an article followed by a noun. A rule like the foregoing requires that all nouns that comprise an object be preceded by a participle. But what if you want the article to be optional? You can do this by saying that an article can either be the noun "the" or an empty string. The following is used to do this:

article → THE | □

The □ (pronounced "epsilon") represents an empty string. If the THE token is present in the input, then the

Article → The

Production is used if it is not there, however, then the article matches an empty string, and

Article → □

So, the parser determines which of the two productions to apply by examining the next input symbol.A grammar that recognizes a limited set of English sentences is shown below:An input sentence can be recognized using this grammar, with a series of replacements, as follows:

i. Start with the topmost symbol in the grammar, the goal symbol.

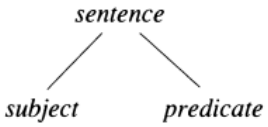
ii. Replace that symbol with one of its right-hand sides.

iii. Continue replacing nonterminals, always replacing the leftmost nonterminal with its right-hand side, until there are no more nonterminals to replace.

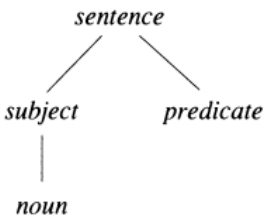
For example, the grammar can be used to recognize "Jane sees Spot run" as follows:

sentence	apply *sentence*→*subject predicate* to get:
subject predicate	apply *subject*→*noun* to get:
noun predicate	apply *noun*→**JANE** to get:
JANE *predicate*	apply *predicate*→*verb object* to get:
JANE *verb object*	apply *verb*→**SEES** to get:
JANE SEES *object*	apply *object*→**noun op_participle** to get:
JANE SEES *noun opt_participle*	apply *noun*→**SPOT** to get:
JANE SEES SPOT *opt_participle*	apply *opt_participle*→*participle* to get:
JANE SEES SPOT *participle*	apply *participle*→**RUN** to get:
JANE SEES SPOT RUN	done—there are no more nonterminals to replace

These replacements can be used to build the parse tree. For example, replacing a sentence with subject-predicate is represented in tree form like this:



The second replacement, of *subject* with *noun*, would modify the tree like this:



The evolution of the entire parse tree is pictured in the above figure. A glance at the parse tree tells you where the terms terminal and nonterminal come from. Terminal symbols are always leaves in the tree they're at the end of a branch, and nonterminal symbols are always interior nodes.

An Expression Grammar

A grammar that recognizes a list of one or more statements, each of which is an arithmetic expression followed by a semicolon. Statements are made up of a series of semicolon-delimited expressions, each comprising a series of numbers separated either by asterisks for multiplication or plus signs for addition.

Note that the grammar is recursive. For example, Production 2 has statements on Recursion in grammar. both the left- and right-hand sides. There's also third-order recursion in Production 8 since it contains an expression, but the only way to get to it is through Production 3, which has an expression on its left-hand side. This last recursion is made clear if you make a few algebraic substitutions in the grammar. You can substitute the right-hand side of Production 6 in place of the reference to the term in Production 4, yielding

Expression →factor

and then substitute the right-hand side of Production 8 in place of the factor:

expression → (expression)

Since the grammar itself is recursive, it stands to reason that recursion can also be used to parse the grammar-1'11 to show how in a moment. The recursion is also important from a structural perspective-it is the recursion that makes it possible for a finite grammar to recognize an infinite number of sentences. The strength of the foregoing grammar is that it is intuitive-its structure directly reflects the way that an expression goes together. It has a major problem, however. The leftmost symbol on the right-hand side of several of the productions is the same symbol that appears on the left-hand side. In Production 3, for example, expression appears both on the left-hand side and at the far left of the right-hand side.

The property is called left recursion, and certain parsers such as the recursive-descent parser that I'll discuss in a moment can't handle left-recursive productions. They just loop forever, repetitively replacing the leftmost symbol on the right-hand side with the entire right-hand side. You can understand the problem by considering how the parser decides to apply a particular product when it is replacing a nonterminal that has more than one right-handsideas represented in Table 1. The simple case is evident in Productions 7 and 8. The parser can choose which production to apply when it's expanding factor by looking at the next input symbol. If this symbol is a number, then the compiler applies Production 7 and replaces the factor with a number. If the next input symbol was an open parenthesis, the parser would use Production 8. The choice between Productions 5 and 6 cannot be solved in this way, however. In the case of Production 6, the right-hand side of the term starts with a factor which, in tum, starts with either a number or left parenthesis.

Consequently, the parser would like to apply Production 6 when a term is being replaced and a next input symbol is a number or left parenthesis. Production 5 the other right-hand sidestarts with a term, which can start with a factor, which can start with a number or left parenthesis, and these are the same symbols that were used to choose Production 6. To summarize, the parser must be able to choose between one of several right-hand sides by looking at the next input

symbol. It could make this decision in Productions 7 and 8, but it cannot make this decision in Productions 5 and 6, because both of the latter productions can start with the same set of terminal symbols. The previous situation, where the parser can't decide which production to apply, is called a conflict, and one of the more difficult tasks of a compiler designer is creating a grammar that has no conflicts in it. The next input symbol is called the look-ahead symbol because the parser looks ahead at it to resolve a conflict.

Table 1: Represented that the Different Statements with its Expression.

1.	*statements*	→	*expression* ;
2.		|	*expression* ; *statements*
3.	*expression*	→	*expression* + *term*
4.		|	*term*
5.	*term*	→	*term* * *factor*
6.		|	*factor*
7.	*factor*	→	**number**
8.		|	(*expression*)

Unfortunately, for reasons that are discussed in Chapter Four, you can't get rid of the recursion by swapping the first and last production element, like this so the grammar must be modified in a very counterintuitive way to build a recursive-descent parser for it. Several techniques can be used to modify grammars so that a parser can handle them. I'll use one of them now, however, without any real explanation of why it works. Take it on faith that the grammar in Table 2 recognizes the same input as the one we've been using. I'll discuss the 1- and E that appears in the grammar momentarily. The modified grammar is inferior grammar in terms of self-documentation-it is difficult to look at it and see the language that's represented. On the other hand, it works with a recursive-descent parser, and the previous grammar doesn't.

Table 2: Represented that the Different Grammer.

1.	*statements*	→	⊢
2.		|	*expression* ; *statements*
3.	*expression*	→	*term expression′*
4.	*expression′*	→	+ *term expression′*
5.		|	ε
6.	*term*	→	*factor term′*
7.	*term′*	→	* *factor term′*
8.		|	ε
9.	*factor*	→	**number**
10.		|	(*expression*)

The symbol is an end-of-input marker. For parsing, the end of the file is. The end input symbol (|-)is treated as an input token, and 1- represents the end of input in the grammar. In this

grammar, Production I is expanded if the current input symbol is the end of input, otherwise, Production 2 is used. Note that an explicit end-of-input marker is often omitted from a grammar, in which case 1- is implied as the rightmost symbol of the starting production the production whose left-hand side appears at the apex of the parse tree. Since eliminating the 1- symbol removes the entire right-hand side in the current grammar, you can use the following as an alternate starting production:

statements → E | expression ; statements

In English: statements can go to an empty string followed by an implied end-of-input marker. The replacement of the left-hand side by the empty string occurs whenever the current input symbol doesn't match a legallookahead symbol. In the current grammar, a term is replaced with the right-hand side *factor term' if the look-ahead symbol the next input symbol is a*· The term' is replaced withE if the next input symbol isn't a*· The process is demonstrated in Figure 3, which shows a parse of 1 + 2 using the modified grammar. The E-production stops things from going on forever.

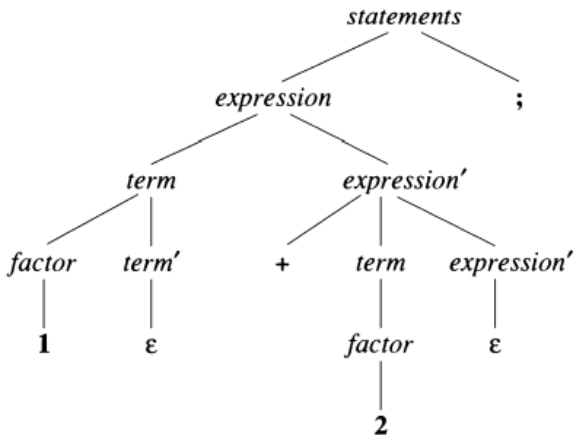


Figure 3: Represented that the Tree Production Stops.

Note that E is a terminal symbol that is not a token. It always appears at the end of a branch in the parse tree, so it is a terminal, but it does not represent a corresponding token in the input stream Just the opposite it represents the absence of a particular token in the input stream.

CHAPTER 6

SYNTAX DIAGRAMS

Dr. Uthama Kumar A
Assistant Professor, Department of Data Science & Analytics, School of Sciences,
Jain (Deemed-to-be University), Bangalore-27, India
Email Id- uthamakumar.a@jainuniversity.ac.in

You can prove to yourself that the grammar works as expected by representing it in a different way-as a syntax diagram. We saw earlier that a syntax diagram can represent the entire syntactic structure of a parse, but you can also use it in a more limited sense to represent the syntax of a single production. Syntax diagrams are useful in writing recursive-descent compilers because they translate directly into flow charts that are the main reason we're looking at them now. You can use them as a map that describes the structure of the parser more on this in a moment.

They are also somewhat more intuitive to an uninitiated reader, so they often make better documentation than a formal grammar. I'll translate our grammar into a syntax diagram in two steps. First, several of the productions can be merged into a single diagram. Figure 1 represents Productions 3, 4, and 5 of the grammar. The E production is represented by the uninterrupted line that doesn't go through a box. You can combine these two graphs by substituting the bottom graph for the reference to it in the top graph, and the same process can be applied to Productions 6, 7, and 8.

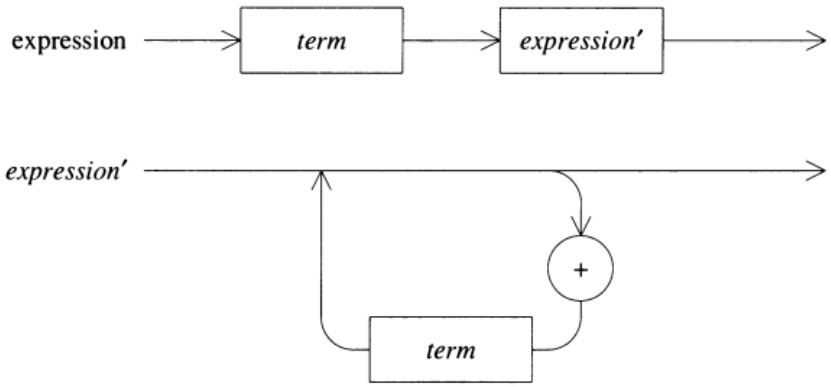


Figure 1: Represented that the Execution of Grammar.

The entire grammar is represented as a syntax diagram in the above figure. The topmost diagram, for example, defines a statement as a list of one or more semicolon-delimited expressions. The same thing is accomplished by

statements → expression ;

| expression ; statements

but the BNF form is harder to understand.

The merged diagram also demonstrates graphically how the modified grammar Diagram shows how works. Just look at it like a flow chart, where each box is a subroutine, and each circle or modified grammar works. ellipse is the symbol that must be in the input when the subroutine returns. Passing through the circled symbol removes a terminal from the input stream, and passing through a box represents a subroutine call that evaluates a nonterminal. A Recursive-Descent Expression Compiler We now know enough to build a small compiler, using the expression grammar we've been looking at. Our goal is to take simple arithmetic expressions as input and generate code that evaluates those expressions at run time. An expression like a+b*c+d is translated to the following intermediate code:

t0 =_a;

t1= _b;

t1= _c;

t0= t1;

t0= _d;

The Lexical Analyzer The first order of business is defining a token set. Except for numbers and Expression token sets. Identifiers, all the lexemes are single characters. Remember, a token is an input symbol taken as a unit, and a lexeme is the string that represents that symbol. A NUM_OR_ID token is used both for numbers and identifiers; so, they are made up of a series of contiguous characters in the range ' 0' -' 9', ' a'-' z', or 'A'-' z'. The tokens themselves are defined with the macros at the top of lex.h, Listi. The lexical analyzer translates a semicolon into a SEMI token, a series of digits into a NUM_OR_ID token, and so on.

The three external variables at the bottom of lex.hare used by the lexical analyzer to pass information to the parser. yytext points at the current lexeme, which is not ' \0' terminated; yy1eng is the number of characters in the lexeme; and yylineno is the current input line number. I've used these somewhat strange names because both lex and LEX use the same names. Usually, I try to make global-variable names begin with an upper-case letter and macro names are in all caps. This way you can distinguish these names from local-variable names, which are always made up of lowercase letters only. It seemed best to retain UNIX compatibility in the current situation, however.

Global Optimization

The global optimization phase is optional. Its purpose is simply to make the object program more efficient in space and/or time. It involves examining the sequence of atoms put out by the parser to find redundant or unnecessary instructions or inefficient code. Since it is invoked before the code generator, this phase is often called machine-independent optimization. For example, in the following program segment:

The Phases of a Compiler

stmt1

go to label1

```
stmt2
stmt3
label2: stmt4
```

stmt2 and stmt3 can never be executed. They are unreachable and can be eliminated from the object program. A second example of global optimization is shown below:

```
for (i=1; i<=100000; i++)
{ x = Math.sqrt (y); // square root method
System.out.println (x+i);
}
```

In this case, the assignment to x need not be inside the loop since y does not change as the loop repeats it is a loop invariant. In the global optimization phase, the compiler would move the assignment to x out of the loop in the object program:

```
x = Math.sqrt (y); // loop invariant
for (i=1; i<=100000; i++)
System.out.println (x+i);
```

This would eliminate 99,999 unnecessary calls to the sqrt method at run time. The reader is cautioned that global optimization can have a serious impact on run-time debugging. For example, if the value of y in the above example was negative, causing a run-time error in the sqrt function, the user would be unaware of the actual location of that portion of code which is called the sqrt function, because the compiler would have moved the offending statement usually without informing the programmer. Most compilers that perform global optimization also have a switch with which the user can turn optimization on or off. When debugging the program, the switch would be off. When the program is correct, the switch would be turned on to generate an optimized version for the user. One of the most difficult problems for the compiler writer is making sure that the compiler generates optimized and optimized object modules, from the same source module, which are equivalent.

Code Generation

Most Java compilers produce an intermediate form, known as byte code, which can be interpreted by the Java run-time environment. In this book, we will be assuming that our compiler is to produce native code for a particular machine. It is assumed that the student has had some experience with assembly language and machine language, and is aware that the computer is capable of executing only a limited number of primitive operations on operands with numeric memory addresses, all encoded as binary values. In the code generation phase, atoms or syntax trees are translated to machine language binary instructions, or to assembly language, in which case the assembler is invoked to produce the object program. Symbolic address statement labels are translated to reloadable memory addresses at this time. For target machines with several CPU registers, the code generator is responsible for register allocation. This means that the compiler must be aware of which registers are being used for particular purposes in the generated

program, and which become available as code is generated. For example, an ADD atom might be translated into three machine language instructions:

- load the first operand into a register,
- add the second operand to that register,
- store the result, as shown for the atom (ADD, a, b,temp):

```
LOD r1,a // Load an into reg. 1
ADD r1,b // Add b to reg. 1
STO r1,temp // Store reg. 1 in temp
```

Local Optimization

The local optimization phase is also optional and is needed only to make the object program more efficient. It involves examining sequences of instructions put out by the code generator to find unnecessary or redundant instructions. For this reason, local optimization is often called machine-dependent optimization. An example of a local optimization would be load/store optimization. An addition operation in the source program might result in three instructions in the object program:

i. Load one operand into a register,
ii. add the other operand to the register,
iii. store the result.

Consequently, the expression a + b + c in the source program might result in the following instructions as code generator output:

```
LOD r1,a          // Load a into register 1
ADD  r1,b          // Add b to register 1
STO r1,temp1      // Store the result in temp1*
LOD r1,temp1      // Load result into reg 1*
ADD r1,c          // Add c to register 1
STO r1,temp2      // Store the result in temp2
```

Note that some of these instructions (those marked with * in the comment) can be eliminated without changing the effect of the program, making the object program both smaller and faster:

```
LOD r1,a          // Load a into register 1
ADD r1,b          // Add b to register 1
ADD r1,c          // Add c to register 1
STO r1,temp       // Store the result in temp
```

A diagram showing the phases of compilation and the output of each phase is shown in Figure 1.4. Note that the optimization phases may be omitted that is the atoms may be passed directly from the Syntax phase to the Code Generator, and the instructions may be passed directly from the Code Generator to the compiler output file. A word needs to be said about the flow of control between phases. One way to handle this is for each phase to run from start to finish separately, writing output to a disk file. For example, lexical analysis is started and creates a file of tokens as dosplay in Figure 2.

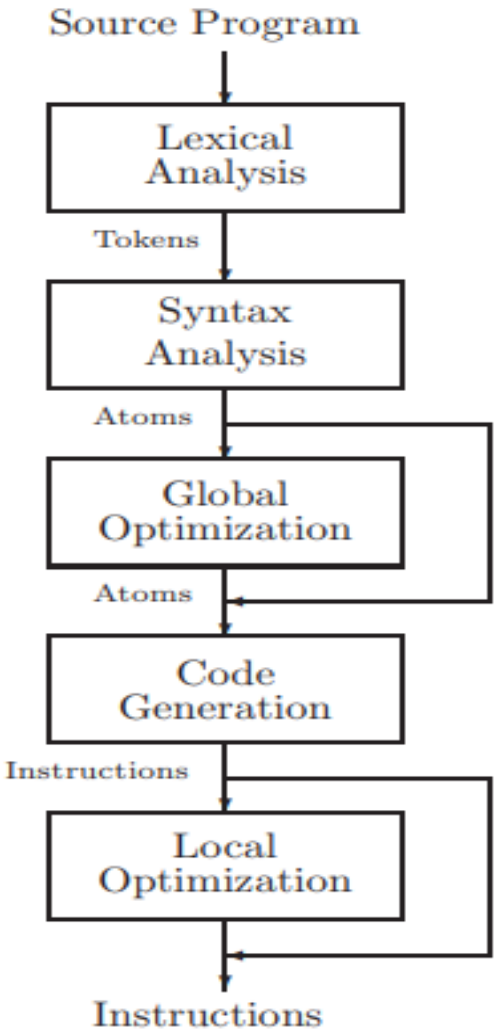


Figure 2: Represented that the Lexical Analysis.

Then, after the entire source program has been scanned, the syntax analysis phase is started, reads the entire file of tokens, and creates a file of atoms. The other phases continue in this manner; this would be a multiple-pass compiler since the input is scanned several times. Another way for the flow of control to proceed would be to start up the syntax analysis phase first. Each time it needs a token it calls the lexical analysis phase as a subroutine, which reads enough source characters to produce one token and returns it to the parser. Whenever the parser has scanned enough source code to produce an atom, the atom is converted to object code by calling the code generator a subroutine; this would be a single-pass compiler.

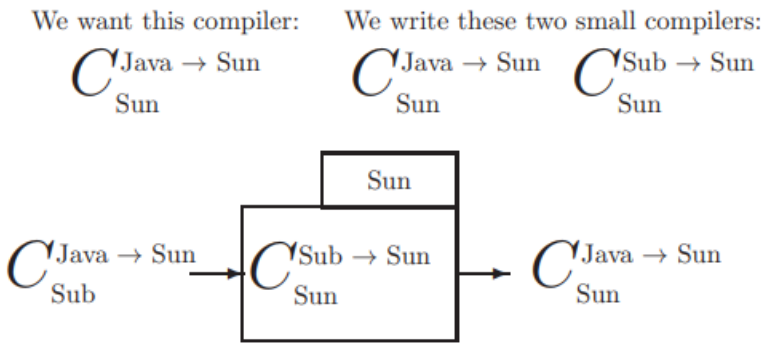
CHAPTER 7

BOOTSTRAPPING

Dr. Thirukumaran Subbiramani
Assistant Professor, Department of Data Science & Analytics, School of Sciences, Jain(Deemed-to-be University), Bangalore-27, India
Email Id- s.thirukumaran@jainuniversity.ac.in

The phrase "pull yourself up by your bootstraps" inspired the term "bootstrapping", which refers to using a curriculum as input for another program. The student may be familiar with the bootstrapping loader, which is used to initialize a computer after it is turned on, hence the phrase "to boot" a computer. On this occasion, bootstrapping a compiler is being discussed, as shown in the image below. For Sun Computer, we want to design a Java compiler. We decide to make two simple programs instead of coding the whole thing in machine or assembly code. The first is a machine assembly language compiler for a block of Java.

The second is a Java subset language-written compiler for the Java language as a whole. The Java subset language, referred to as "Sub", is simply Java minus many extraneous features such as enumerated types, unions, switch instructions, etc. The computer's memory is loaded with the first compiler, and the second is used as input. The result is the compiler we've been looking for, a Sun-based compiler that thus outputs object code in Sun machine language and supports the entire Java language. It is an iterative method that starts with a minor subset of Java and ends with some larger subset. We keep doing this until we have a compiler that can handle the entire Java language as well.

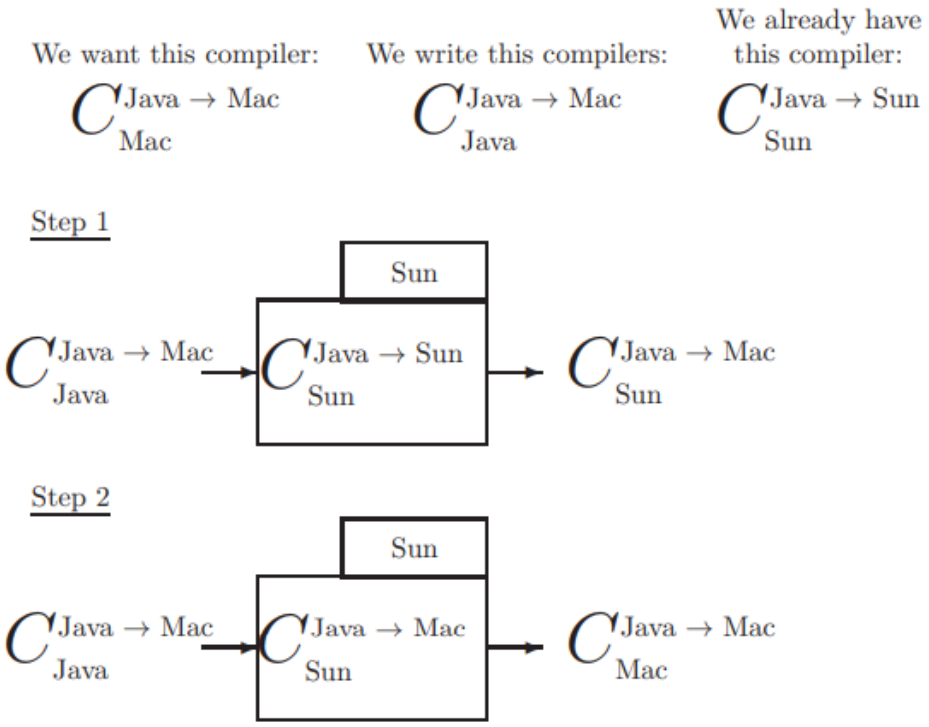


Cross Compiling

The computer industry continues to produce great machines with advanced and sometimes trivial programming languages. For every programming language currently in use, a new compiler has to be created every time a new computer is built. Cross-compiling is one such technique that is used to simplify this problem. The graphic below shows a two-step cross-generation process. Let's imagine we develop a Java compiler for Sun and a new machine called Mac. We want to

build a Java compiler for Mac as an alternative to building a compiler in machine assembly language. The first step is to use this compiler as input to the Sun Java compiler. As a result, Java is translated into Mac machine code via a compiler that runs on Sun.

The next step is to load this compiler into Sun and once again use the input from the Java-written compiler. This time, the result is the editor we planned to build a Mac-compatible Java compiler. It should be emphasized that the entire process can be completed even before a MAC is created. All we need to know is the architecture the instruction set, instruction formats, and addressing modes, of the Mac.



Compiling To Intermediate Form

It is possible to compile into an intermediate form, a language somewhere between source high-level language and machine language, as we indicated in our discussion of interpreters above. One such example of an intermediate form is the stream of atoms that the parser emits. The main advantage of this approach is that only one translator or interpreter is needed for the intermediate form on each computer and only one translator or interpreter is needed for the intermediate form from each high-level language these include Each is called a back end.

As shown in the picture below, we will need two code generators or interpreters for each computer as well as three translators for each of the three high-level languages and two machines as intermediate. Had the intermediate form not been employed, a total of six different compilers would have been required. If there were n high-level languages and m machines, we would typically need n x m compilers. We would need (n + m)/2 compilers using the intermediate form, assuming that each front end and each back end are half of a compiler as display in Figure 1.

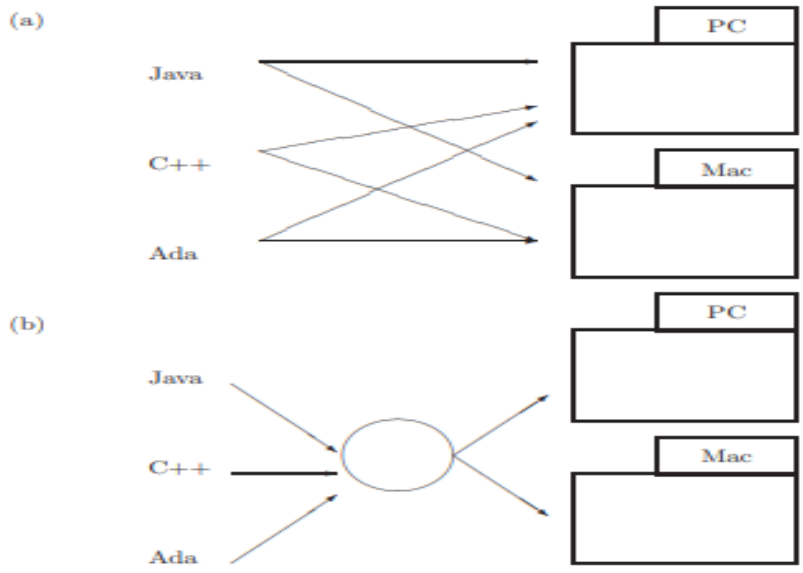


Figure 1: Represented that the Compiling To Intermediate Form.

P-code, a highly well-liked intermediate form for computers like the PDP-8 and Apple II series, was created some years ago at the University of California, San Diego. High-level languages like C are often utilized as intermediary forms nowadays. Another intermediate format that has been widely utilized on the Internet is the Java Virtual Machine also known as Java byte code.

Compiler-Compilers

At this point, a large portion of compiler design is so well-known that automation is possible. The source language and target machine requirements may be written by the compiler author so that the compiler may be created automatically and a compiler-compiler does this.

Lexical Analysis

This paragraph examines how lexical analysis is implemented by compilers. Lexical analysis is the process of identifying words in a source program, as stated in Chapter 1. Next, the compiler passes these words as tokens, each token having a class and a value. A table of identifiers, a symbol table, and a table of numeric constants are good examples of tables that can be prepared at this stage of compilation. The lexical analysis phase may also begin the development of tables that will be needed later in the compilation. However, before moving on to lexical analysis, we must ensure that the learner is familiar with the formal linguistics and automata theory concepts that are essential to the design of a lexical analyzer.

Formal Languages

The study of formal languages, which is important for understanding programming languages and compilers, is introduced in this section. A natural language is commonly used by humans, whereas a formal language is something that can be spoken carefully and is suitable for computer use. The grammar of Java is an example of a technical language, although as will be explored in the following sections, a formal language may also have no apparent significance or purpose.

Language Elements

Before we build on the language let's make sure the learner understands some basic concepts from discrete math. A set is a group of distinct objects. We usually list each element of a set only once, while it is acceptable to list an element more than once. The elements can also be written in any order. For example, the words "boy, girl, animal" are a set, yet they represent the same group as "girl, boy, animal, girl". A set can contain an infinite number of things. The empty set often referred to as the set with no elements, is still a set and is denoted either by or by. A list of characters in an alphabet is called a string.

The components of a string are not necessarily unique, and it matters how they are listed. Examples include "abc" and "cba," as well as "abb" and "ab." Even if a string does not contain any characters, it is still a string of letters from a given alphabet, and we refer to this string as the null string and identify it by the symbol. It is important to note that, for example, if we are talking about a string of zeros and ones that is, strings beginning with the letters "0, 1", then "" is a string of zeros and ones. This chapter will cover languages as well as upcoming languages. A formal language is made up of a set of letters. The learner must understand the difference between a set and a string, and in particular the difference between the empty set and the null string, to understand this. The following are examples of languages from the alphabet {0,1}:

- {0,10,1011}
- { }
- { □,0,00,000,0000,00000,... }
- The set of all strings of zeroes and ones having an even number of ones.

While the latter two cases are endless, the first two examples are finite sets. The null string is present in the latter two samples but not the first two. Four languages that may be represented by the alphabet of characters on a computer keyboard are as follows:

- {0,10,1011}
- { □ }
- Java syntax
- Italian syntax

The fourth example is a natural language in which each string is a lossless Java program while the third is the syntax of a computer language in which each string in the language is a grammatically correct Italian sentence. The second example is not the empty set.

Finite State Machines

We now face a challenge when endeavoring to exactly define the strings in an endless or very huge language. When characterizing the language in English, we are unable to be as precise about which strings are included and which are excluded from the language. Using a finite state machine, a mathematical or fictitious device is indeed a way to solve this issue. Although we won't automatically build this machine, we will characterize it in mathematical terms so that its functioning is completely evident. Automata theory is the study underlying theoretical machines, for example, the finite state machine, as an automaton is another term for a machine. An example of a finite machine of states is:

i. A finite collection of states, of those which zero or more are labelled accepted states and one is classified as the beginning state. A receptive state might alternatively be the starting point.

ii. A state transition function with a state and an inputting symbol as its two parameters from a given input alphabet. In a conclusion, it returns a state.

Here is how the gadget works. The input is composed of a string of symbols drawn from either input alphabet. When the machine initially begins, it is in its initial state. The above equation, which relies on both the input symbol and the machine's current state, illustrates how the machine changes state whenever a symbol from the input string is read. After reading the whole input string, the machine is either in either an accepted state or a non-accepting state. If the input string is found in accepting conditions, we declare that it has been accepted. If not, the incoming string has been declined because it is invalid. The finite state machine provides a precise definition of a language by constructing the language from the set of all possible input strings. Finite state machines may be represented in a variety of ways, one of which is using state diagrams. A finite state machine is shown as an example in the following Figure2.

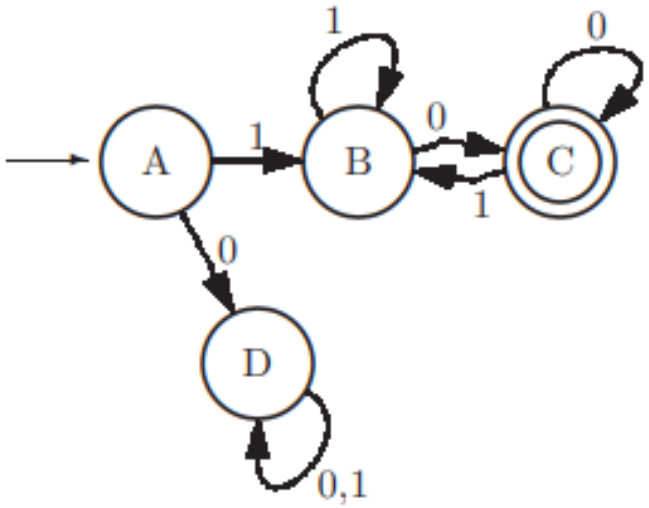


Figure 2: Represented that the Finite State Machine.

The transition function is shown by the arcs labeled with the input symbols that go from one state to the next, and each state of the machine is represented by a circle. The initial state is shown by an arc with no state at the source (tail) end, while the acceptor states are double circles. When the machine is in position B a 0 is entered, jumps to position C. When the input is 1 while the machine is in state B, the machine remains in state B. The initial state is A, and the only accepted state is C. Any string of ones and zeros that begins with a one and ends with a zero is accepted by this machine because these strings and only these will bring the machine to an acceptable position after reading the entire input string. Will bring in in the image below, another finite state system is shown in Figure 3.

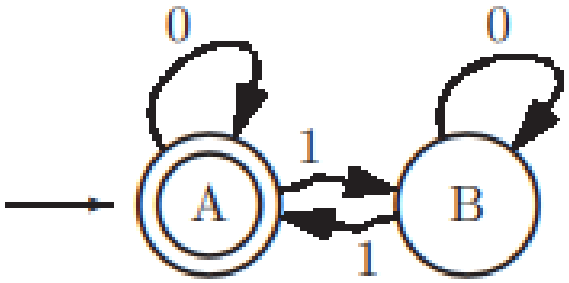


Figure 3: Reprinted that the Finite State System.

Any string of ones and zeros that also contains an even number of numbers is accepted by this machine which includes the zero string. A parity checker is a name given to such a device. The input alphabet for each of these devices is 0 and 1. It is important to note that the state transitions in each of these machines are consistent and that they are fully described. This indicates that for each state, exactly one arc identified by each possible input symbol leaves that state. These instruments are called deterministic for this reason. In our study, only deterministic finite-state machines will be used. Another example of a finite state machine is a table, in which the states are given names (A, B, C, etc.), and these names identify the rows of the table. Input symbols are used to identify columns. Each item in the table represents the future state of the machine with input and its current state. Asterisks are used to indicate acceptor states, and the first state reported in the table, as seen in the accompanying figure, is the initial state. It is simple to verify that the machine is fully described and deterministic using the table representation. However, when developing or studying finite state machines, many students find it easier to work with the state diagram model.

	0	1
A	D	B
B	C	B
*C	C	B
D	D	D

(a)

	0	1
*A	A	B
B	B	A

(b)

CHAPTER 8

REGULAR EXPRESSIONS

Kunal Dey
Assistant Professor, Department of Data Science & Analytics, School of Sciences,
Jain (Deemed-to-be University), Bangalore-27, India
Email Id- d.kunal@jainuniversity.ac.in

Another method for specifying languages is regular expressions. These are formulas or expressions consisting of three possible operations on languages: union, concatenation, and Kleene star.

i. Union Since a language is a set, this operation is the union operation as defined in set theory. The union of two sets is that set which contains all the elements in each of the two sets and nothing else. The union operation on languages is designated with a '+'. For example,

   {abc, ab, ba} + {ba, bb} = {abc, ab, ba, bb}

   Note that the union of any language with the empty set is that language:

   L + { } = L

ii. Concatenation:To define the concatenation of languages, we must first define the concatenation of strings. This operation will be designated by a raised dot whether operating on strings or languages, which may be omitted. This is simply the juxtaposition of two strings forming a new string. For example,

   abc · ba = abcba

   Note that any string concatenated with the null string is that string itself:

   s · □ = s.

In what follows, we will omit the quote marks around strings to avoid cluttering the page needlessly. The concatenation of two languages is that language is formed by concatenating each string in one language with each string in the other language. For example,

{ab, a, c} · {b, □} = {ab · b, ab · □, a · b, a · □, c · b, c · □} = {abb, ab, a, cb, c}

In this example, the string ab need not be listed twice. Note that if L1 and L2 are two languages, then L1 · L2 is not necessarily equal to L2 · L1. Also, L ·{ □ } = L, but L ·φ = φ .

iii. Kleene * This operation is a unary operation designated by a postfix asterisk and is often called closure. If L is a language, we define:

$L^0 = \{\square\}$

$L^1 = L$

$L^2 = L \cdot L$

.

.

.

$L^n = L \cdot L^{n-1}$

$L^* = L^0 + L^1 + L^2 + L^3 + L^4 + L^5 + ...$

Note that $\varphi* = \{\square\}$. Intuitively, Kleene * generates zero or more concatenations of strings from the language to which it is applied. We will use a shorthand notation in regular expressions: if x is a character in the input alphabet, then x = {'x'}; i.e., the character x represents the set consisting of one string of length 1 consisting of the character x. This simplifies some of the regular expressions we will write:

$0 + 1 = \{0\} + \{1\} = \{0, 1\}$

$0 + \square = \{0, \square\}$

A regular expression is an expression involving the above three operations and languages. Note that Kleene * is unary (postfix) and the other two operations are binary. Precedence may be specified with parentheses, but if parentheses are omitted, concatenation takes precedence over union, and Kleene * takes precedence over concatenation. If L1, L2 and L3 are languages, then:

$L1 + L2 \cdot L3 = L1 + (L2 \cdot L3)$

$L1 \cdot L2* = L1 \cdot (L2*)$

An example of a regular expression is: (0+1)* To understand what strings are in this language, let L = {0,1}. We need to find:

$L^*: L^0 = \{ \square \}$

$L^1 = \{0, 1\}$

$L^2 = L \cdot L^1 = \{00, 01, 10, 11\}$

$L^3 = L \cdot L^2 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

.

.

.

$L = \{ \square, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, ...\}$

This is the set of all strings of zeros and ones.

Another example: (0+1)*.0 = 1(0+1)*0 = {10, 100, 110, 1000, 1010, 1100, 1110, ...} This is the set of all strings of zeros and ones which begin with a 1 and end with a 0.

Note that we do not need to be concerned with the order of evaluation of several concatenations in one regular expression, since it is an associative operation. The same is true of union:

$L \cdot (L \cdot L) = (L \cdot L) \cdot L$

$L + (L + L) = (L + L) + L$

A word of explanation on nested Kleene *'s is in order. When a * operation occurs within another * operation, the two are independent. That is, in generating a sample string, each * generates 0 or more occurrences independently. For example, the regular expression (0*1)* could generate the string 0001101. The outer * repeats three times; the first time the inner * repeat three times, the second time the inner * repeat zero times, and the third time the inner * repeats once.

Lexical Tokens

The first phase of a compiler is called lexical analysis. Because this phase scans the input string without backtracking i.e. by reading each symbol once and processing it correctly, it is often called a lexical scanner. As implied by its name, lexical analysis attempts to isolate the words in an input string.

We use the word in a technical sense. A word, also known as a lexeme, a lexical item, or a lexical token, is a string of input characters which is taken as a unit and passed on to the next phase of compilation. Examples of words are:

i. while, if, else, for, ... These are words which may have a particular predefined meaning to the compiler, as opposed to identifiers which have no particular meaning. Reserved words are keywords which are not available to the programmer for use as identifiers. In most programming languages, such as Java and C, all keywords are reserved. PL/1 is an example of a language which has keywords but no reserved words.

ii. **Identifiers:** words that the programmer constructs to attach a name to a construct, usually having some indication as to the purpose or intent of the construct. Identifiers may be used to identify variables, classes, constants, functions, etc.

iii. **Operators:** symbols used for arithmetic, character, or logical operations, such as +,-,=,!=, etc. Notice that operators may consist of more than one character.

iv. **Numeric Constants:**numbers such as 124, 12.35, 0.09E-23, etc. These must be converted to a numeric format so that they can be used in arithmetic operations because the compiler initially sees all input as a string of characters. Numeric constants may be stored in a table.

v. **Character Constants:** single characters or strings of characters enclosed in quotes.

vi. **Special Characters:**characters used as delimiters such as .,(,),,,;. These are generally single-character words.

vii. **Comments:**Though comments must be detected in the lexical analysis phase, they are not put out as tokens to the next phase of compilation.

viii. **White Space:**Spaces and tabs are generally ignored by the compiler, except to serve as delimiters in most languages, and are not put out as tokens.

ix. **Newline:** In languages with the free format, newline characters should also be ignored, otherwise a newline token should be put out by the lexical scanner.

An example of Java source input, showing the word boundaries and types is given below:

while (x33 <= 2.5e+33 - total) calc (x33) ; //!

1 6 2 3 4 3 2 6 2 6 2 6 6

A symbol table is built when identifiers are disputed during lexical analysis. No of how many times an identifier appears in the source program, this data structure only saves it once. Additionally, it keeps track of the kind of identifier and the location of any associated run-time data such as the value allocated to a variable related to the identifier. For fast searching, this data structure is most often set up as a binary search tree or hash table. The processing of the symbol table is more complicated when block-structured languages like Java, C, or Algol are compiled. Both representations of the identifier must be acknowledged since the same identifier seems to have distinct declarations in other blocks or operations.

Block scopes may be specified in a single symbol table or by developed especially symbol tables for each block. The scanner might just store the identifier in a character space array and provide a reference to its first character to be able to do this at the parse or grammatical analysis stage of the compiler. It is necessary to translate arithmetic constants into the internal environment form. Consider the constant 3.4e+6 as a string of six characters which must be converted to basic operations (or fixed point integer) format for the computer to use it in the required arithmetic operations. We'll see that this is not a simple issue, hence the mass of compiler authors employ library approaches to address it.

A stream of tokens, one for each word found in the input program, is the phase's output. Each token has two components:

i. A class identifying the kind of token;

ii. A value that specifies whose class member it is.

iii. The token stream seen above may look something like this:

The above example might produce the following stream of tokens:

Token Token

Class Value

1 [code for while]

6 [code for (]

2 [ptr to symbol table entry for x33]

3 [code for <=]

4 [ptr to constant table entry for 2.5e+33]

3 [code for -]

2[ptr to symbol table entry for total] 6 [code for)]

2 [ptrto symbol table entry for calc]

6 [code for (]

2 [ptr to symbol table entry for x33]

6 [code for)] 6 [code for ;]

Keep in mind that the remark is not made public. Additionally, the valuation portion may not exist in all token types. In contrast, a left parenthesis might represent a token class despite requiring a value to be supplied. This plan may undoubtedly be modified in certain methods to improve efficiency. One assignment token could be sent out, for example, after identification and an assignment operator. A symbol table pointer for the identifier would be the token's value section.

Thus, rather than two tokens, the input string x = would be output as nothing more than a single token. Additionally, each keyword may be a different token class, which will also dramatically multiply the number of classes but might enable the syntax analysis step simpler. The scanner must support this functionality if the source language is not case-sensitive. The words then, tHeN, Then, and THEN, for example, all reflect the same keyword. The alphabetic letters might all be translated to upper or lower case using preprocessing Java respects case.

Syntax Analysis

In contrast to lexical analysis, which separates the input into separate tokens, syntax analysis, sometimes referred to as parsing, aims to reassemble these tokens. Back into something that resembles the organization of the text, not a list of characters. The syntax tree of the sentence is a kind of data structure that serves as this "something" in most cases. This building is a tree, as the title indicates. The tokens discovered by the lexical analysis are represented by the leaves of this tree, and when the leaves are scanned from left to right, the order matches that of the input text. Therefore, the syntax tree's structure and labelling of its core nodes, as well as how different leaves are joined, are both crucial. The syntax analysis must not only determine the structure of the text document but also reject incorrect texts by highlighting syntax problems. More sophisticated techniques are needed since syntax analysis has become less local compared lexical analysis.

However, we use the same fundamental technique: A low-level language suited for efficient execution is converted from a notation acceptable for human comprehension. It is known as parser generation. Context-free grammars1, a recursive notation for defining collections of strings and imposing a structure on each string individually, is the notation we employ for human manipulation. Although this notation is occasionally virtually immediately translated into recursive algorithms, it is often more practicable to build stack automata. These are comparable to the machine language used for lexical analysis, but they also have the option of using a stack, which enables symbol counting and non-local matching. We'll look at two approaches for creating these automata. The first of them, LL, is the simplest but only applies to a small subset of grammars. The SLR architecture, which we will examine in greater detail later, is more intricate but supports a larger range of grammars. Consequently, none of them applies to all context-free grammars. Some tools can handle all context-free grammars, but they may be exceedingly slow, which is why the majority of parser generators limit the class available input grammars.

Context-free Grammars

Context-free grammars describe collections of strings, or languages, similarly to regular expressions. The structure of the strings inside this language it specifies is likewise defined by context-free grammar. An alphabet, such as the set of tokens generated by a lexer or the subset of

alphanumeric letters, is what defines a language. Terminals are the names given to the alphabetic symbols. A context-free language defines many sets of strings recursively. A name, referred to as a nonterminal, is used to identify each set. The set of non-terminals and the set of connections are not connected. A nonterminal is used to indicate the language that the grammar describes. This is considered to as the grammar's start sign. Many presentations explain the settings. Every generation provides a selection of the alternative strings that are part of the set marked by a nonterminal. A production takes this shape:

$$N \rightarrow X_1 \ldots\ldots\ldots X_n$$

X1 through Xn are zero or more symbols, every one of which is either a terminating or a nonterminal, where N is a nonterminal. The set marked by N comprises strings that were created by concatenating phrases from the sets denoted by X1... Xn, according to the notation's intended meaning. As with alphabet letters in regular expressions, a terminal here indicates a singleton set. When there is little chance of a mistake, we shall equate a nonterminal with both the collection of strings it designates.

$$A \rightarrow a$$

Declares that the one-character string is present in the set represented by the nonterminal A.

$$A \rightarrow aA$$

States that any strings created by adding an in front of a string selected from the set indicated by A are included in the set denoted by A. Together, these two productions show that A includes all non-empty sequences of as and is, thus, identical to the regular expression a+ (in the absence of further productions). The two productions may be used to construct a grammar that is equal to the regular expression a*:

$$B \rightarrow$$

$$B \rightarrow aB$$

The empty string is shown in the first production to be a component of set B. This grammar should be compared to the definition of s*. Empty productions are those that have empty right-hand sides. Sometimes, instead of leaving the right side empty, they are written with a. We haven't yet specified any sets that regular expressions couldn't have just as easily used to describe. However, it is observed that context-free grammars may describe considerably more sophisticated languages and that the language $\{an\ bn \mid n \geq 0\}$ is not regular. However, the grammar clearly explains it:

$$S \rightarrow$$

$$S \rightarrow aSb$$

For the “As” and “Bs” to appear equally often, the second production makes sure that they are coupled symmetrically around the center of the string. Only one nonterminal per grammar was utilized in the aforementioned instances. We must make it apparent which nonterminal is the start symbol when several nonterminal are used. The start symbol is typically the nonterminal on the left side of the first production (if nothing else is specified). For instance, the grammar

$$T \rightarrow R$$

$$T \rightarrow aTa$$

$$R \rightarrow b$$

$$R \rightarrow bR$$

Has "T" as the start symbol and represents the collection of strings that begin with any number of as are followed by any number of non-zero "bs," and then end with the same number of as they began. The alternate symbol (|) from regular expressions is often used to divide the right-hand sides when all the products of the same nonterminal are concatenated into a single rule. The previous grammar would appear as follows in this notation:

$$T \rightarrow R \mid aTa$$

$$R \rightarrow b \mid bR$$

There are still four productions in the grammar as explain in Table 1, even though the arrow symbol → is only used twice.

Table 1: Illustrated the Regular Expressions to Context-Free Grammars.

Sr. No.	Form of S_i	Production of N_i
1.	ε	$N_i \rightarrow$
2.	a	$N_i \rightarrow a$
3.	S_jS_k	$N_i \rightarrow N_jN_k$
4.	$S_j \mid S_k$	$N_i \rightarrow N_j$ $N_i \rightarrow N_k$
5.	S_j*	$N_i > N_jN_j$ $N_i \rightarrow$
6.	S_j+	$N_i \rightarrow N_j N_i$ $N_i \rightarrow N_j$
7.	$S_j?$	$N_i \rightarrow N_j$ Ni →

Write the Context Free Grammar

As previously said, by employing a nonterminal for each subexpression in the regular expression and one or two products for each nonterminal, a regular expression may be methodically recast

as a context-free grammar. Making a grammar for a language is simple if we can conceive of a method to represent it as a regular expression. But we also wish to characterize non-regular languages using grammars. The types of arithmetic equations used in most programming languages and on electronic calculators serve as an illustration. Grammar may be used to characterize these phrases. Here are some examples of simple expression grammar:

Exp → Exp+Exp

Exp → Exp-Exp

Exp → Exp*Exp

Exp → Exp/Exp

Exp → num

Exp → (Exp)

It should be noted that regular expressions cannot "count" the number of unmatched opening parentheses at a certain position in the text, hence they are unable to characterize matching parenthesis. However, if the language did not have parenthesis, the following regular expression might be used to describe it:

Num ((+|-|*|/)num)*

The ordinary description, however, considers the expression as a flat string rather than as having structure, thus it is useless if you want operators to have varied precedence. Here is the syntax for simple statements:

Stat → id:=Exp

Stat → Stat ;Stat

Stat → ifExpthenStat else Stat

Stat → ifExpthenStat

Context-free grammars may readily represent the majority of constructs from programming languages. This is how most contemporary languages are constructed. One often begins by categorizing the language's constructs into several syntactic groups when creating a grammar for a computer language. A sub-language that expresses a specific idea is called a syntactic category. Syntactic categories that are often used in programming languages include:

i. Expressions are used to express the calculation of values.

ii. Statements express actions that occur in a particular sequence.

iii. Declarations express properties of names used in other parts of the program.

Derivation

So far, while describing the collection of strings that a language generates, we have only relied on common sense ideas of recursion. We may anticipate using the procedures to discover the set of strings specified by a language since the productions are comparable to recursive set equations. Though these techniques, which take into account the bounds of chains of sets, theoretically extend to infinite sets, they are only applicable realistically to finite sets. Instead, we define the term "derivation" below. This method also has the benefit of being closely

connected to derivation, as we shall see later. Derivation's fundamental principle is to see productions as rewriting rules: Anytime we have a nonterminal, we may swap it out for the production's right side whenever the nonterminal comes on the left. This may be repeated until we are left with just terminals in a series of terminals and non-terminal symbols. The terminals that result are a string written in the grammar's designated language. Formally, the three rules below define the derivation relation:

i. **$\alpha N\beta \Rightarrow \alpha\gamma\beta$ if there is a production $N \rightarrow \gamma$**
ii. **$\alpha \Rightarrow \alpha$**
iii. **$\alpha \Rightarrow \gamma$ if there is a β such that $\alpha \Rightarrow \beta$ and $\beta \Rightarrow \gamma$**

Where α, β and γ possibly empty sequences of grammar are symbols terminals and non-terminals. It is a derivation step, according to the first rule, to use production as a rewrite rule anywhere in a series of grammatical symbols. The second claims that a sequence derives from itself and that the derivation connection is reflexive. The third rule explains transitivity, which states that a series of derivations is a derivation in and of itself.

Syntax Trees and Ambiguity

A derivation may be represented as a tree: Every time we rewrite a nonterminal, we add the symbols on the right-hand side of the expression that was utilised as its children. The start symbol of the grammar serves as the tree's root. The derived string is made up of the termination that makes up the tree's leaves whenever they are read from left to right. It is shown as a child of a nonterminal when it is rebuilt using an empty production. Despite its status as a leaf node though too, this one is disregarded when reading the tree's string from its leaves. The sequence of derivation is unimportant whenever writing such a syntax tree; if we deduce in the left, right, or any other manner, we still obtain the same tree. It is only important which products you use to rewrite each nonterminal. The string produces gains structure thanks to the syntax tree. In the subsequent stages of the compiler, we make use of this structure. We reverse the derivation for compilation: To create a syntax tree, we commence with a string. This procedure is known as processing or syntax analysis. The selection of products for that word or words is important while building a syntax tree even if the order of derivation is irrelevant. Naturally, various decisions would result in the derivation of distinct strings, but it is also possible for many syntax trees to be constructed for a single string. As an example, Figure 1, shows an alternative syntax tree for the same string that was derived in Figure 2.

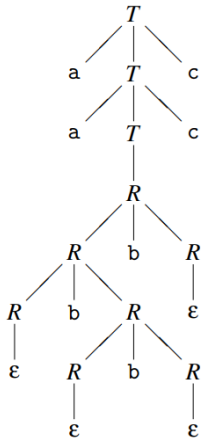


Figure 1: Represented the Syntax Tree for the String "aabbbcc" using Grammar.

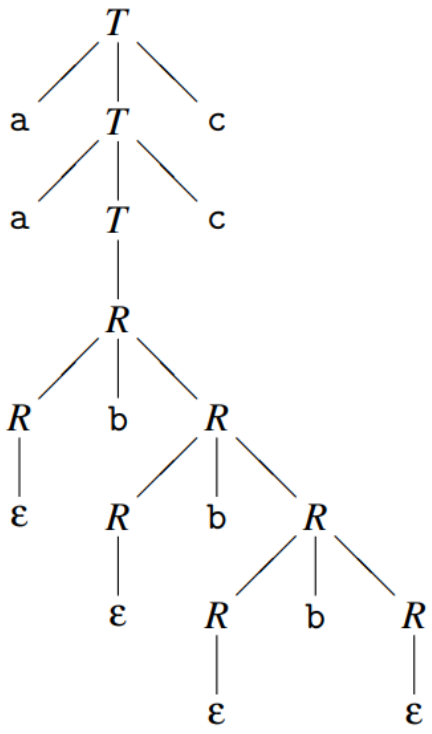


Figure 2: Alternative syntax tree for the string aabbbcc Using Grammar.

We refer to a grammar as ambiguous when it allows for very many syntax trees for certain strings. Ambiguity is not a concern if the only purpose of grammar is to describe collections of strings. However, the structure must always be the same if we wish to utilize grammar to establish structure on strings. Being unambiguous in a grammar is thus a trait that is preferred. The majority of the time, but not always, it is feasible to translate an ambiguous grammar into an unambiguous vocabulary that produces the same collection of strings, or external rules may be used to determine which of the many different syntax trees is the "correct one".

Rewriting Ambiguous Expression Grammars

If we have an ambiguous grammar

$$\mathbf{E \rightarrow E \oplus E}$$

$$\mathbf{E \rightarrow num}$$

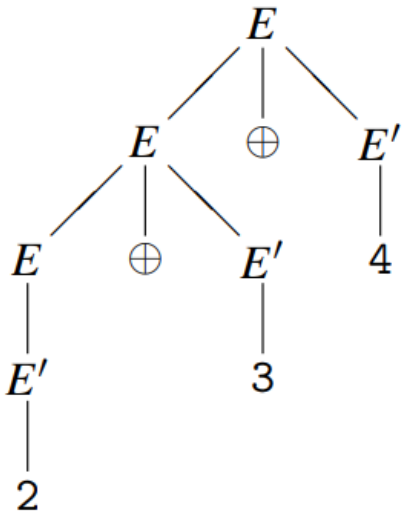
We can rewrite this to an unambiguous grammar that generates the correct structure. As this depends on the associativity ⊕, we use different rewrite rules for different associativity. If ⊕ is left-associative, we make the grammar left-recursive by having a recursive reference to the left only of the operator symbol:

$$\mathbf{E \rightarrow E \oplus E'}$$

$$\mathbf{E \rightarrow E'}$$

$$\mathbf{E \rightarrow num}$$

Now, the expression 2⊕3⊕4 can only be parsed as:



We handle right-associativity in a similar fashion: We make the offending production right-recursive:

E → E'⊕ E

E → E'

E → num

Non-associative operators are handled by non-recursive productions:

E → E'⊕ E'

E → E'

E' → num

Note that the latter transformation changes the language that the grammar generates, as it makes expressions of the form **num⊕num⊕num** illegal. So far, we have handled only cases where an operator interacts with itself. This is easily extended to the case where several operators with the same precedence and associativity interact with each other, for example, + and -:

E → E +E'

E → E –E'

E → E'

E'→ num

Operators with the same precedence must have the same associativity for this to work, as mixing left-recursive and right-recursive productions for the same nonterminal makes the grammar ambiguous.

E → E +E'

E → E'⊕ E

E → E'

E'→ num

Exp → Exp+Exp2

Exp → Exp-Exp2

Exp → Exp2

Exp2 → Exp2*Exp3

Exp2 → Exp2/Exp3

Exp2 → Exp3

Exp3 → num

Exp3 → (Exp)

As an example, the grammar seems like an obvious generalization of the principles used above, giving + and ⊕ the same precedence and different associativity. But not only is the grammar ambiguous, but it also does not even accept the intended language. For example, the string num + num⊕num is not derivable by this grammar. In general, there is no obvious way to resolve ambiguity in an expression like 1 + 2 ⊕ 3, where + is left-associative and ⊕ is right-associative or vice-versa). Hence, most programming languages and most parser generators require operators at the same precedence level to have identical associativity. We also need to handle operators with different precedence's. This is done by using a nonterminal for each precedence level. The idea is that if an expression uses an operator of a certain precedence level, then its sub-expressions cannot use operators of lower precedence unless these are inside parentheses. Hence, the productions for a nonterminal corresponding to a particular precedence level refer only to nonterminal that corresponds to the same or higher precedence levels, unless parentheses or similar bracketing constructs disambiguate the use of these.

Syntax Analysis

Using a string of tokens generated by the lexer, the syntax analysis portion of a compiler will create a syntax tree for something like the string by tracing its derivation first from grammar's start symbol. Although guessing at derivations at random isn't very productive, it may be done until the proper one is determined. However, other parsing techniques depend on "guessing" the derivations. These, however, ensure that they will constantly make the correct estimate by glancing at the string. They are referred to as predictive parsing techniques. Predictive parsers are also sometimes known as deterministic top-down parsers since they always construct the syntax tree from either the root to the leaves.

While simultaneously developing the syntax tree, other parsers scan the input text for portions that match the right-hand sides of compositions and rewrite them to the left-hand nonterminal. When the string has been recreated using an inverse derivation from the initial symbol, the syntax tree is ultimately finished. Additionally, we want to guarantee that we always choose the "correct" rewrites to achieve determinism parsing. These methods are also known as bottom-up parsing techniques. In the sections that follow, we'll first examine predictive parsing and then SLR interpreting, a bottom-up approach to parsing.

CHAPTER 9

PREDICTIVE PARSING

Ghouse Basha M A
Assistant Professor, Department of Data Science & Analytics, School of Sciences,
Jain (Deemed-to-be University), Bangalore-27, India
Email Id- ghouse.basha@jainuniversity.ac.in

If we look at the left-derivation in the above figure, we see that, to the left of the rewritten nonterminal, there are only terminals. These terminals correspond to a prefix of the string that is being parsed. In a parsing situation, this prefix will be the part of the input that has already been read. The job of the parser is now to choose the production by which the leftmost unexpanded nonterminal should be rewritten. We aim to be able to make this choice deterministically based on the next unmatched input symbol. If we look at the third line in above figure 2.3, we have already read two as and if the input string is the one shown in the bottom line the next symbol is a b. Since the right-hand side of the production:

$$\mathbf{T \rightarrow aTc}$$

Starts with an 'a', we obviously cannot use this. Hence, we can only rewrite 'T' using the production

$$\mathbf{T \rightarrow R}$$

We are not quite as lucky in the next step and none of the productions for 'R' start with a terminal symbol, so we cannot immediately choose a production based on this. As the grammar is ambiguous, it should not be a surprise that we cannot always choose uniquely. If we instead use unambiguous grammar and it can immediately choose the second production for 'R'.

When all the 'bs' are read and we are at the following 'c', we choose the empty production for R and match the remaining input with the rest of the derived string. If we can always choose a unique product based on the next input symbol, we can do predictive parsing without backtracking.

Nullable and First

In simple cases, like the above, all productions for a nonterminal start with distinct terminals except at most one product that does not start with a terminal. We chose the latter whenever the input symbol did not match any of the terminal symbols starting the other productions. We can extend the method to work also for grammars where several productions start with nonterminal. We just need to be able to select between them based on the input symbol. In other words, if the strings these productions can derive begin with symbols from disjoint sets, we check if the input symbol is in one of these sets and choose the corresponding production if it is. If not, and there is an empty production, we choose this. Otherwise, we report a syntax error message. Hence, we define the function FIRST, which gives a sequence of grammar symbols for example the right-hand side of production returns the set of symbols with which strings derived from that sequence can begin:

Nullable(ε) = true

Nullable(a) = false

Nullable(αβ) = Nullable(α) ∧Nullable(β)

Nullable(N) = Nullable(α_1) ∨...∨Nullable(α_n),

where the productions for N are

N → α_1, ... , N → α_n

Where 'a' is a terminal, 'N' is a nonterminal, 'α' and 'β' are sequences of grammar symbols and 'ε' represents the empty sequence of grammar symbols. The equations are quite natural: Any occurrence of a terminal on a right-hand side makes Nullable false for that right-hand side, but a nonterminal is nullable if any product has a nullable right-hand side. Note that this is a recursive definition since Nullable for a nonterminal is defined in terms of Nullable for its right-hand sides, which may contain that same nonterminal. We can solve this in much the same way. We have, however, now booleans instead of sets and several equations instead of one. Still, the method is essentially the same: We have a set of Boolean equations:

$$X_1 = F_1(X_1,...,X_n)$$

.

.

.

$$X_n = F_n(X_1,...,X_n)$$

At first, we suppose that all of X_1, X_2,..., X_n are untrue. The right-hand sides of the equations are then computed in any order, and the computed value is subsequently updated to the variable on the left-hand side. We keep going until every equation is met. We stipulated that the functions in grammar as given in Table 1 below must be monotonic with respect to the subset. As a consequence, we now demand that Boolean functions be monotonic with regard to truth: If we add more true inputs, the conclusion will also be truer, meaning that it may remain constant or shift from false to true but never from true to false.

Table 1: Represented that the Different Function in Grammar.

Right-hand side	**Initialization**	**Iteration 1**	**Iteration 2**	**Iteration 3**
R	False	False	True	True
aTc	False	False	False	False
ε	False	True	True	True
bR	False	False	False	False
Nonterminal				
T	False	False	True	True
R	False	True	True	True

If we look at grammar below, we get these equations for the nonterminal and right-hand sides:

Nullable (T) = Nullable(R) ∨Nullable (aTc)

Nullable(R) = Nullable (ε) ∨Nullable (bR)

Nullable(R) = Nullable(R)

Nullable (aTc) = Nullable (a) ∧Nullable (T) ∧Nullable (c)

Nullable (ε) = true

Nullable (bR) = Nullable (b) ∧Nullable (R)

In a fixed-point calculation, we initially assume that **Nullable** is false for all nonterminal and use this as a basis for calculating **Nullable** for first the right-hand sides and then the nonterminal. We repeat recalculating these until there is no change between the two iterations. As mentioned in the below Algorithm, it shows the fixed-point iteration for the above equations. In each iteration, we first evaluate the formulae for the right-hand sides and then use the results of this to evaluate the nonterminal. The rightmost column shows the final result. We can calculate **FIRST** in a similar fashion to **Nullable**:

FIRST (ε) = φ

FIRST (a) = {a}

$$\textbf{FIRST } (\alpha\ \beta) = \begin{cases} \textbf{FIRST}(\alpha) \cup \textbf{FIRST}(\beta) & \textit{if Nullable}(\alpha) \\ \textbf{FIRST}(\alpha) & \textit{if not Nullable}(\alpha) \end{cases}$$

FIRST (N) = FIRST (α_1) ∪...∪ FIRST (α_n)

Where the productions for N are

$N \rightarrow \alpha_1, ..., N \rightarrow \alpha_n$

Table 1: Represented theFixed-point iteration for the calculation of FIRST.

Right-hand side	Initialization	Iteration 1	Iteration 2	Iteration 3
R	Φ	Φ	true	true
aTc	Φ	{a}	{a}	{a}
ε	Φ	Φ	Φ	Φ
bR	Φ	{b}	{b}	{b}
Nonterminal				
T	Φ	{a}	{a,b}	{a,b}
R	Φ	{b}	{b}	{b}

Where 'a' is a terminal, 'N' is a nonterminal, α and β are sequences of grammar symbols and 'ε' represent the empty sequence of grammar symbols.The only nontrivial equation is that for αβ. Anything that can start a string derivable from α can also start a string derivable from αβ. However, if α is nullable, a derivation may proceed as αβ ⇒ β ⇒ ···, so anything in FIRST (β) is also in FIRST (αβ). The setequations are solved in the same general way as the Boolean equations for Nullable, but since we work with sets, we initially assume every set to be empty. According to above Table 1, we get the following equations:

FIRST (T) = FIRST(R) ∪ FIRST (aTc)

FIRST(R) = FIRST (ε) ∪ FIRST (bR)

FIRST(R) = FIRST(R)

FIRST (aTc) = FIRST (a)

FIRST (ε) = Φ

FIRST (bR) = FIRST (b)

When working with grammars by hand, it is usually quite easy to see for most productions if they are nullable and what their **FIRST** sets are. For example, a production is not nullable if its right-hand side has a terminal anywhere, and if the right-hand side starts with a terminal, the **FIRST** set consists of only that symbol. Sometimes, however, it is necessary to go through the motions of solving the equations. When working by hand, it is often useful to simplify the equations before the fixed-point iteration, for example, reduce **FIRST (aTc) to {a}**.

Predictive Parsing Revisited

We are now ready to construct predictive parsers for a wider class of grammars: If the right-hand sides of the productions for a nonterminal have disjoint FIRST sets, we can use the next input symbol to choose among the productions. The empty production if any on any symbol that was not in the FIRST sets of the non-empty productions for the same nonterminal. We can extend this, so we in case of no matching FIRST sets can select a product if it is Nullable. The idea is that a Nullable production can derive the empty string, so the input symbol need not be read by the production itself. But if there are several Nullable productions, we have no way of choosing between them. Hence, we do not allow more than one production for a nonterminal to be Nullable. However, this is not true with the method as stated above: We can get unique choice of production even for some ambiguous grammars, including grammar. The syntax analysis will in this case just choose one of several possible syntax trees for a given input string. In many cases, we do not consider such behaviour acceptable. We would very much like our parser construction method to tell us if we by mistake write an ambiguous grammar.

T → aTb

Even worse, the rules for predictive parsing as presented here might even for some unambiguous grammars give deterministic choice of production, but reject strings that belong to the language described by the grammar. If we, for example, change the second production in grammar 3.9 this will not change the choices made by the predictive parser for nonterminal R. However, always choosing the last production for R on a b will lead to erroneous rejection of many strings, including ab. This kind of behaviour is unacceptable. We should, at least, get a warning that this might occur, so we can rewrite the grammar or choose another syntax analysis method. Hence, we add to our construction of predictive parsers a test that will reject all ambiguous grammars

and those unambiguous grammars that can cause the parser to fail erroneously. We have so far simply chosen a nullable production if and only if no other choice is possible. This is, however, not always the right thing to do, so we must change the rule to say that we choose a production N → α on symbol c if one of the two conditions below is satisfied:

i. c ∈ FIRST(α)

ii. "α" is nullable and the sequence "Nc" can occur somewhere in a derivation starting from the start symbol of the grammar.

The first rule is self-evident; however, the second needs some explanation: It looks like a nullable production may be a viable option regardless of the following input symbol if "α" is nullable since we can build a syntax tree based on it without reading any input. When there are both character data and non-nullable manufactures for the same nonterminal, this would not only provide several acceptable production options, but it is also not necessarily wise to choose the nullable production: We always update the leftmost nonterminal "N" in the current succession of grammar symbols since predictive parsing creates a leftmost derivation. The set of grammar symbols that follow 'N' in the present sequence must thus match any input that 'N' does not match. If it's not feasible, we chose poorly when deriving "N" if it's not possible.

In particular, the next input symbol, "c," should start the reasoning of the sequence that follows "N" if "N" derives from the empty sequence. Therefore, the series of symbols that come after "N" should at the very least have a derivation that starts with "c." The sequence "Nc" should be seen throughout the derivation if we derive the symbols following "N" before "N" using a separate derivation order. In the absence of doing so, it is impossible to rewrite "N" to the empty sequence without also running into trouble when rewriting the entire sequence. We can see that if the alternative composition order becomes stuck, so will the leftmost derivation order since the derivation order does not affect the syntax tree. As a result, we can only rewrite "N" to the empty sequence if the following input symbol "c" may be used in a valid derivation with "N." We'll examine how to do this in the next section.

Keep in mind that if c ∈ FIRST (α), a nullable production N → α may be picked with validity. We may still have instances where both nullable and non-nullable products are viable options notwithstanding the limitation on selecting nullable productions. This applies to all ambiguous grammars that are not identified as being ambiguous by the old technique, where we only choose nullable products if there are no other viable options, as well as the example above with the changed grammar since Rb may occur in a derivation.

A Larger Example

The above examples of calculating FIRST and FOLLOW are rather small, so we show a somewhat more substantial example. The following grammar describes even-length strings of "as" and "bs" that are not of the form "ww" where w is any string of "as" and "bs". In other words, the strings cannot consist of two identical halves.

N → A B

N → B A

A → a

A → C A C

B → b

B → C B C

C → a

C → b

The idea is that if the string does not consist of two identical halves, there must be a point in the first string that has an "a" where the equivalent point in the second string has "a" "b" or vice-versa. The grammar states that one of these is the case. We first note that there is empty production in the grammar, so no production can be **Nullable**. So we immediately set up the equations for **FIRST** for each nonterminal and right-hand side:

FIRST (N) = FIRST (A B) ∪ FIRST (B A)

FIRST (A) = FIRST (a) ∪ FIRST (C A C)

FIRST (B) = FIRST (b) ∪ FIRST (C B C)

FIRST (C) = FIRST (a) ∪ FIRST (b)

FIRST (A B) = FIRST (A)

FIRST (B A) = FIRST (B)

FIRST (a) = {a}

FIRST (C A C) = FIRST (C)

FIRST (b) = {b}

FIRST (C B C) = FIRST (C)

Which we solve by fixed-point iteration as present in Table 2. We initially set the **FIRST** sets for the nonterminal to the empty sets, calculate the **FIRST** sets for right-hand sides and then nonterminal, repeating the last two steps until no changes occur:

Table 2: Illustrate the fixed-point iteration as present.

Right-hand side	Iteration 1	Iteration 2	Iteration 3
AB	Φ	{a}	{a, b}
BA	Φ	{b}	{a, b}
a	{a}	{a}	{a}
CAC	Φ	{a, b}	{a, b}
b	{b}	{b}	{b}
CBC	Φ	{a, b}	{a, b}

Nonterminal			
N	Φ	{a,b}	{a,b}
A	{a}	{a, b}	{a, b}
B	{b}	{a, b}	{a, b}
C	{a, b}	{a, b}	{a, b}

The next iteration does not add anything, so the fixedpoint is reached. We now add the production N'→N$ and set up the constraints for calculating FOLLOW sets as display in Table 3:

Table: 3: Represented that the Production and its Constraints.

Production	Constraints
N' → N$	{$}⊆FOLLOW(N)
N → A B	FRST(B)⊆FOLLOW(A), FOLLOW(N)⊆FOLLOW(B)
N → B A	FIRST(A)⊆FOLLOW(B), FOLLOW(N)⊆FOLLOW(A)
A → a	
A → C A C	FIRST(A)⊆FOLLOW(C),FIRST(C)⊆FOLLOW(A), FOLLOW(A)⊆FOLLOW(C)
B → b	
B → C B C	FIRST(B)⊆FOLLOW(C), FIRST(C)⊆FOLLOW(B), FOLLOW(B)⊆FOLLOW(C)
C → a	
C → b	

We first use the constraint {$} ⊆ FOLLOW (N) and constraints of the form FIRST(···) ⊆ FOLLOW(···) to get the initial sets:

FOLLOW(N) ⊆ {$}

FOLLOW(A) ⊆ {a, b}

FOLLOW(B) ⊆ {a, b}

FOLLOW(C) ⊆ {a, b}

And then use the constraints of the form FOLLOW (···) ⊆FOLLOW(···). If we do this in top-down order, we get after one iteration:

FOLLOW(N) ⊆ {$}

FOLLOW(A) ⊆ {a, b, $}

FOLLOW(B) ⊆ {a, b, $}

FOLLOW(C) ⊆ {a, b, $}

Another iteration does not add anything, so the final result is:

FOLLOW(N) = {$}

FOLLOW(A) = {a, b, $}

FOLLOW(B) = {a, b, $}

FOLLOW(C) = {a, b, $}

LL(1) parsing We have, in the previous sections, looked at how we can choose productions based on **FIRST** and **FOLLOW** sets, that is using the rule that we choose a production **N → α** on input symbol c if:

c ∈ FIRST(α),

Nullable(α) and c ∈ FOLLOW(N).

If we can always choose a production uniquely by using these rules, this is called **LL(1)** parsing- the first **L** indicates the reading direction (left-to-right), the second **L** indicates the derivation order (left) and the 1 indicates that there is one symbol look ahead. A grammar that can be parsed using **LL(1)** parsing is called an **LL(1)** grammar. In the rest of this section, we shall see how we can implement **LL(1)** parsers as programs. We look at two implementation methods: Recursive descent, where grammar structure is directly translated into the structure of a program, and a table-based approach that encodes the decision process in a table.

Recursive Descent

As the name indicates, recursive descent uses recursive functions to implement predictive parsing. The central idea is that each nonterminal in the grammar is implemented by a function in the program. Each such function looks at the next input symbol to choose one of the productions for the nonterminal. The right-hand side of the chosen product is then used for parsing in the following way:

A terminal on the right-hand side is matched against the next input symbol. If they match, we move on to the following input symbol and the next symbol on the right-hand side, otherwise, an error is reported. A nonterminal on the right-hand side is handled by calling the corresponding function and, after this call returns, continuing with the next symbol on the right-hand side. When there are no more symbols on the right-hand side, the function returns. As an example, figure 3.16 shows pseudo-code for a recursive descent parser for grammar. We have constructed this program by the following process: We have first added a production

T' → T$

and calculated FIRST and FOLLOW for all productions.

T' has only one production, so the choice is trivial. However, we have added a check on the next input symbol anyway, so we can report an error if it is not in **FIRST(T')**. This is shown in the function parseT'. For the parseT function, we look at the productions for T. As **FIRST(R) = {b}**, the production

T → R

Is chosen on the symbol b. Since R is also Nullable, we must choose this production also on symbols in **FOLLOW(T)**, i.e., c or $. **FIRST(aTc) = {a}**, so we select **T → aTc** on an a. On all other symbols, we report an error.

```
function parseT'() =
    if next = 'a' or next = 'b' or next = '$' then
parseT() ; match('$')
     else reportError()
function parseT() =
   if next = 'b' or next = 'c' or next = '$' then
parseR()
     else if next = 'a' then
       match('a') ;parseT() ; match('c')
     else reportError()
function parseR() =
     if next = 'c' or next = '$' then
       (* do nothing *)
     else if next = 'b' then
       match('b') ;parseR()
       else reportError()
```

The above code is represented that the "Recursive Descent Parser for Grammar". For parseR, we must choose the empty production on symbols in **FOLLOW(R)(c or $)**. The production **R → bR** is chosen on input b. Again, all other symbols produce an error. The function match takes as argument a symbol, which it tests for equality with the next input symbol. If they are equal, the following symbol is read into the variable next. We assume the next is initialised to the first input symbol before **'parseT'** is called. The above program only checks if the input is valid. It can easily be extended to construct a syntax tree by letting the parse functions return the sub-trees for the parts of the input that they parse.

Table-driven LL(1) Parsing

In table-driven **LL(1)** parsing, we encode the selection of productions into a table instead of in the program text. A simple non-recursive program uses this table and a stack to perform the parsing. The table is cross-indexed by nonterminal and terminal and contains for each such pair

the production (if any) that is chosen for that nonterminal when that terminal is the next input symbol. This decision is made just as for recursive descent parsing: The production **N → α** is in the table at **(N,a)** if a is in **FIRST(α)** or if both **Nullable(α)** and a are in **FOLLOW(N)**.

Table 4: Represented the Table-Driven LL(1) Parsing.

	a	**b**	**c**	**$**
T'	T'→T$	T'→T$	N/A	T'→T$
T	T→ aTc	T→ R	T→ R	T→ R
R		R→bR	R→	R→

It uses a stack, which at any time contains the part of the current derivation that has not yet been matched to the input. When this eventually becomes empty, the parse is finished. If the stack is non-empty, and the top of the stack contains a terminal, that terminal is matched against the input and popped from the stack. Otherwise, the top of the stack must be a nonterminal, which we cross-index in the table with the next input symbol. If the tableentry is empty, we report an error. If not, we pop the nonterminal from the stack and replace this with the right-hand side of the production in the table entry. The list of symbols on the right-hand side is pushed such that the first of these will be at the top of the stack. The input and stack at each step during parsing of the string "aabbbcc$" using the above Table 4. The top of the stack is to the left. It, too, can be extended to build a syntax tree. This can be done by letting each nonterminal on the stack point to its node in the partially built syntax tree. When the nonterminal is replaced by one of its right-hand sides, nodes for the symbols on the right-hand side are added as children to the node.

Conflicts

When a symbol allows several choices of production for nonterminal N we say that there is a conflict on that symbol for that nonterminal. Conflicts may be caused by ambiguous grammars indeed all ambiguous grammars will cause conflicts but there are also unambiguous grammars that cause conflicts. An example of this is the unambiguous expression grammar. We will in the next section see how we can rewrite this grammar to avoid conflicts, but it must be noted that this is not always possible: There are languages for which there exist unambiguous context-free grammars but where no grammar for the language generates a conflict-free LL(1) table. Such languages are said to be non-LL(1). It is, however, important to note the difference between a non-LL(1) language and a non-LL(1) grammar: A language may well be LL(1) even though the grammar used to describe it is not.

```
stack := empty ; push(T',stack)
  while stack <> empty do
    if top(stack) is a terminal then
  match(top(stack)) ; pop(stack)
    else if table(top(stack),next) = empty then
```

```
reportError
else
rhs := rightHandSide(table(top(stack),next)) ;
        pop(stack) ;
pushList(rhs,stack)
```

The above program are represented the Program for table-driven LL(1) parsing to note the difference between a non-LL(1) language and a non-LL(1) grammar: A language may well be LL(1) even though the grammar used to describe it is not.

input	**stack**
aabbbcc$	T'
aabbbcc$	T$
aabbbcc$	aTc$
abbbcc$	Tc$
abbbcc$	aTcc$
bbbcc$	Tcc$
bbbcc$	Rcc$
bbbcc$	bRcc$
bbcc$	Rcc$
bbcc$	bRcc$
bcc$	Rcc$
bcc$	bRcc$
cc$	Rcc$

cc$	cc$
c$	c$
$	$

Rewriting a Grammar for LL(1) Parsing

In this section, we will look at methods for rewriting grammars such that they are more palatable for LL(1) parsing. In particular, we will look at the *elimination of left recursion* and *left factorization*. It must, however, be noted that not all grammars can be rewritten to allow LL(1) parsing. In these cases, stronger parsing techniques must be used.

Eliminating left-recursion

As mentioned above, the unambiguous expression grammar is not LL(1). The reason is that all productions in Exp and Exp2 have the same FIRST sets. Overlap like this will always happen when there are left-recursive productions in the grammar, as the FIRST set of a left-recursive production will include the FIRST set of the nonterminal itself and hence be a superset of the FIRST sets of all the other productions for that nonterminal. To solve this problem, we must avoid leftrecursion in the grammar. We start by looking at direct leftrecursion. When we have a nonterminal with some left-recursive and some productions that are not, that is:

N → N α1

.

.

.

N → N αm

N → β1

.

.

.

N → βn

Where the β_i do not start with N, we observe that this generates all sequences that start with one of the β_i and continue with any number (including 0) of the α_j. In other words, the grammar is equivalent to the regular expression$(\beta_1 | ... | \beta_n)(\alpha_1 | ... | \alpha_m)^*$.

Exp → Exp2 Exp∗

Exp∗ → + Exp2 Exp∗

Exp∗ → - Exp2 Exp∗

Exp∗ →

Exp2 → Exp3 Exp2∗

Exp2∗ → * Exp3 Exp2∗

Exp2∗ → / Exp3 Exp2∗

Exp2∗ →

Exp3 → num

Exp3 → (Exp)

We saw in the below program which is denoted that the Removing left-recursion from grammar. A method for converting regular expressions into context-free grammars can generate the same set of strings. By following this procedure and simplifying a bit afterwards, we get this equivalent grammar:

$\mathbf{N \rightarrow \beta_1 N*}$

.

.

$\mathbf{N \rightarrow \beta_n N*}$

$\mathbf{N* \rightarrow \alpha_1\ N*}$

.

.

$\mathbf{N* \rightarrow \alpha_m\ N*}$

N∗ →

Where N∗ is a new nonterminal that generates a sequence of α_s.Note that, since the β_i do not start with N, there is no direct left-recursion in the first n productions. Since N∗ is a new nonterminal, the α_j cannot start with this, so the last m productions can't have direct left-recursion either. There may, however, still be indirect left-recursion if any of the αj are nullable or the β_i can derive something starting with N. We will briefly look at indirect left-recursion below. While we have eliminated direct left-recursion, we have also changed the syntax trees that are built from the strings that are parsed. Hence, after parsing, the syntax tree must be re-structured to obtain the structure that the original grammar describes.

Indirect Left Recursion

The transformation is only applicable in the simple case where there is no indirect left-recursion. Indirect left-recursion can have several faces:

i. There are mutually left-recursive productions:

$\mathbf{N_1 \rightarrow N_2 \alpha_1}$

$\mathbf{N2 \rightarrow N_3 \alpha_2}$

.

.

.

$\mathbf{N_{k-1} \rightarrow N_k \alpha_{k-1}}$

$\mathbf{N_k \rightarrow N_1 \alpha_k}$

ii. There is a production $\mathbf{N \rightarrow \alpha N \beta}$.
Where α is Nullable.

Any combination of the two. More precisely, a grammar is left-recursive if there is a non-empty derivation sequence $\mathbf{N \Rightarrow N\alpha}$, i.e., if a nonterminal derives a sequence of grammar symbols that start by that same nonterminal. If there is indirect left-recursion, we must first rewrite the grammar to make the left-recursion direct and then use the transformation above. Rewriting a grammar to turn indirect left-recursion into direct left-recursion can be done systematically, but the process is a bit complicated. We will not go into this here, as in practice most cases of left-recursion are direct left-recursion.

Left Factorization

If two productions for the same nonterminal begin with the same sequence of symbols, they have overlapping **FIRST** sets. As an example, in the below grammar the two productions for if have overlapping prefixes.

Stat → id:=Exp

Stat → Stat ;Stat

Stat → ifExpthenStat else Stat

Stat → ifExpthenStat

We rewrite this in such a way that the overlapping productions are made into a single product that contains the common prefix of the productions and uses a new auxiliary nonterminal for the different suffixes see the below grammar:

Stat → id:=Exp

Stat → if Exp then Stat Elsepart

Elsepart → else Stat

Elsepart →

In this grammar we can uniquely choose one of the productions for Stat based on one input token. For most grammars, combining productions with common prefix will solve the problem. However, in this particular example, the grammar still is not LL(1): We cannot uniquely choose a production for the auxiliary nonterminal Elsepart, since else is in FOLLOW(Elsepart) as well as in the FIRST set of the first production for Elsepart. This should not be a surprise to us, since, after all, the grammar is ambiguous and ambiguous grammars cannot be LL(1). The equivalent unambiguous grammar (grammar 3.13) cannot easily be rewritten to a form suitable for LL(1), so in practice grammar 3.21 is used anyway and the conflict is handled by choosing the non-empty production for Elsepart whenever the symbol else is encountered, as this gives the desired behavior of letting an else match the nearest if. We can achieve this by removing the empty production from the table entry for Elsepart/else, so only the non-empty production Elsepart → else Stat remains. Very few LL(1) conflicts caused by ambiguity can be removed in this way, however, without also changing the language recognized by the grammar. For example, operator precedence ambiguity cannot be resolved by deleting conflicting entries in the LL(1).

Construction of LL(1) Parsers Summarized

i. Eliminate ambiguity

ii. 2. Eliminate left-recursion

iii. 3. Perform left factorisation where required

iv. 4. Add an extra start production S

v. 0 → S$ to the grammar.

vi. 5. Calculate FIRST for every production and FOLLOW for every nonterminal.

vii. 6. For nonterminal N and input symbol c, choose production N → α when:

- c ∈ FIRST(α),
- Nullable(α) and c ∈ FOLLOW(N)

This choice is encoded either in a table or a recursive-descent program.

SLR parsing

The majority of grammars need significant rewriting to be put into a form that allows for a unique choice of production, which is a drawback of LL(1) parsing. There are still many grammars that cannot be mechanically converted into LL(1) grammars, even though this rewriting can be, to a significant part, automated. A family of bottom-up parsing techniques called LR parsers accepts a significantly broader range of grammars than LL(1) parsing, but not all grammars. The fundamental benefit of LR parsing is that it requires less rewriting than LL(1) parsing to bring a language into an acceptable form for LR parsing. In addition, LR parsers permit external specification of operator precedence's for resolving ambiguity rather than requiring that the grammars themselves be unambiguous. We'll examine SLR parsing, a straightforward variation of LRparsing. SLR is an acronym for "Simple," "Left," and "Right." The words "Left" and "Right" denote that the input is read from left to right and that the rightmost derivation is produced, respectively. LR parsers are bottom-up, table-driven parsers that use two different types of "actions" utilizing an input stream and a stack:

There will at some point during the parsing be no suitable actions and the parser will halt with an error message if the input text does not follow the grammatical rules. If not, the parser will read everything in and just leave one element of the grammar's start symbol on the stack. Shift-reduce parsers are another name for LR parsers. Our goal is to limit the option of action to the next input symbol and the symbol at the top of the stack, much as with LL(1). This is accomplished by building a DFA. The DFA reads the stack's contents conceptually, beginning at the bottom. The right action is a reduction by a production defined by the next input symbol and a mark on the accepting DFA state if the DFA is accepting when it reaches the top of the stack. The right response is a shift on one of the symbols for which there is an outgoing edge from the DFA state if the DFA is not accepting when it reaches the top of the stack. As a result, the DFA scans the stack from bottom to top at each step, and the next input symbol and the DFA state are used to decide what to do.

However, it is not particularly effective to have the DFA read the whole stack with each operation, so we instead save the DFA's reading status with each stack member. By doing this, we may start from the top of the stack instead of the bottom, beginning the DFA in its stored

state rather than its original state. When the DFA indicates a shift, the next step is simple: we just take the state off the top of the stack, identify the next DFA state by following the transition suggested by the next input symbol, and then store both the symbol and the new state on the stack. If the DFA showed a decrease, we remove the symbols for the production's right side from the stack. The DFA state is then read from the new stack top.

We store both the nonterminal and the state after the transition on the stack since this DFA state should have a transition on the nonterminal that is the left-hand side of the production. Because of these improvements, the DFA only has to check a terminal or nonterminal once when it is placed into the stack. It just needs to read the DFA state, which is kept at the top of the stack, at all other times. Once a transition has been performed on an input symbol or nonterminal, we don't need to keep it since no further transitions will rely on it in the future; the stored DFA state suffices. As a result, we may allow each stack element to just include the DFA state rather than the symbol plus the state. Even while the DFA is still used, it now only has to consider the next input symbol at a shift action or nonterminal and the current state stored at the top of the stack to decide what to do next. To discover one of the following actions, we cross-index a DFA state with a symbol either terminal or nonterminal and express the DFA as a table.

shift n: Read the next input symbol and push state n on the stack.

go n: Push state n on the stack.

reduce p: Reduce with the production numbered p.

accept: Parsing has been completed successfully.

error: A syntax error has been detected.

Keep in mind that the top of the stack is always in its current state. When a state and a terminal symbol are cross-indexed, shift and reduce operations are employed. When a state is cross-indexed with a nonterminal, go actions are employed. Since the destination state of a go action relies on the state at the top of the stack after the right-hand side of the reduced production is popped off, we cannot mix the go activities with the reduced actions in the table. A go in the state discovered after the stack is popped immediately follows a decrease in the current state.An example SLR table is shown in Figure 1.

	a	b	c	$	*T*	*R*
0	s3	s4	r3	r3	g1	g2
1				a		
2			r1	r1		
3	s3	s4	r3	r3	g5	g2
4		s4	r3	r3		g6
5			s7			
6			r4	r4		
7			r2	r2		

Figure 1: Represented the SLR Table for Grammar.

The actions have been abbreviated to their first letters and the error is shown as a blank entry. The algorithm for parsing a string using the table is shown in Figure 2.

```
stack := empty ; push(0,stack) ; read(next)
loop
  case table[top(stack),next] of
    shift s:  push(s,stack) ;
              read(next)

    reduce p: n := the left-hand side of production p ;
              r := the number of symbols
                        on the right-hand side of p ;
              pop r elements from the stack ;
              push(s,stack)
                  where table[top(stack),n] = go s

    accept:   terminate with success

    error:    reportError
endloop
```

Figure 2: Represented the Algorithm for SLR Parsing.

The shown algorithm just determines if a string is in the language generated by the grammar. It can, however, easily be extended to build a syntax tree: Each stack element holds in addition to the state number a portion of a syntax tree.

When performing a reduced action, a new (partial) syntax tree is built by using the nonterminal from the reduced production as root and the syntax trees stored at the popped-off stack elements as children.

The new tree and the new state are then pushed as a single stack element. Figure 3 shows an example of parsing the string aabbbcc using the table in Figure 4. The sequences of numbers in the "stack" column represent the stack contents with the stack bottom shown to the left and the stack top to the right.

At each step, we look at the next input symbol at the left end of the string in the input column and the state at the top of the stack at the right end of the sequence in the stack column. We look up the pair of input symbol and state in the table and find an action, which is shown in the action column. When the shown action is a reduce action, we also show the reduction used in parentheses and after a semicolon also the go action that is performed after the reduction.

input	stack	action
aabbbcc$	0	s3
abbbcc$	03	s3
bbbcc$	033	s4
bbcc$	0334	s4
bcc$	03344	s4
cc$	033444	r3 ($R \rightarrow$) ; g6
cc$	0334446	r4 ($R \rightarrow$ bR) ; g6
cc$	033446	r4 ($R \rightarrow$ bR) ; g6
cc$	03346	r4 ($R \rightarrow$ bR) ; g2
cc$	0332	r1 ($T \rightarrow R$) ; g5
cc$	0335	s7
c$	03357	r2 ($T \rightarrow$ aTc) ; g5
c$	035	s7
$	0357	r2 ($T \rightarrow$ aTc) ; g1
$	01	accept

Figure 3: Represented the Example SLR parsing.

$$
\begin{aligned}
0:&\quad T' \rightarrow T\\
1:&\quad T \rightarrow R\\
2:&\quad T \rightarrow \mathtt{a}T\mathtt{c}\\
3:&\quad R \rightarrow\\
4:&\quad R \rightarrow \mathtt{b}R
\end{aligned}
$$

Figure 4: Represented the Example Grammar for SLR-table Construction.

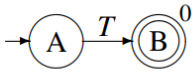
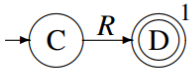
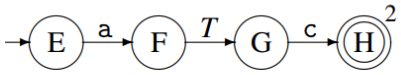
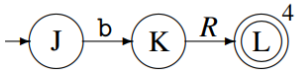
Production	NFA
$T' \rightarrow T$	
$T \rightarrow R$	
$T \rightarrow \mathtt{a}T\mathtt{c}$	
$R \rightarrow$	
$R \rightarrow \mathtt{b}R$	

Figure 5: Represented the NFAs for the productions in grammar.

The next step is to make an NFA for each production. This is done exactly like treating both terminals and nonterminals as alphabet symbols. The accepting state of each NFA is labelled with the number of the corresponding production. The result is shown in Figure 5. Note that we

have used the optimized construction for ε, which is the empty production. The NFAs in Figure 6 make transitions both on terminals and nonterminal. Transitions by terminal correspond to shift actions and transitions on nonterminal correspond to go actions. A go action happens after a reduction, whereby some elements of the stack corresponding to the right-hand side of a production are replaced by a nonterminal corresponding to the left-hand side of that production. However, before we can reduce on a nonterminal, the symbols that form the right-hand side must be on the stack. So we prepare for a later transition on a nonterminal by allowing transitions that, eventually, will leave the symbols forming a right-hand side of the production on the stack, so we can reduce these to the nonterminal and then make a transition in this.

state	epsilon-transitions
A	C, E
C	I, J
F	C, E
K	I, J

Figure 6: Represented the Epsilontransitions.

We thus permit transitions on the symbol sequences on the right-hand sides of the productions whenever a transition by a nonterminal is feasible. Epsilon transitions are added to the NFAs in Figure 3.4 to accomplish this. We add epsilon transitions from s to the starting states of all the NFAs for productions with "N" on the left-hand side whenever a nonterminal N transitions from states to state t. Epsilon transitions are noted in a separate table in Figure 3.5 rather than as arrows in Figure 3.4 since doing so would result in a considerably more crowded image. However, this is purely for display purposes: Whether they are shown in the table or the graphic of the DFA, the transitions have the same significance.

The NFAs in Figure 7 are joined by these epsilontransitions to produce a single, combined NFA. This NFA contains an accepted state for each production in the grammar as well as the beginning state "A," which serves as the starting state of the NFA for the additional start production. This NFA must now be transformed into a DFA using the subset. We create a table instead of displaying the resultant DFA visually, with transitions on terminals being represented as shift actions and transitions on non-terminals being represented as go actions.

	a	b	c	$	*T*	*R*
0	s3	s4	r3	r3	g1	g2
1				a		
2			r1	r1		
3	s3	s4	r3	r3	g5	g2
4		s4	r3	r3		g6
5			s7			
6			r4	r4		
7			r2	r2		

Figure 7: Represented the SLR Table for Grammar.

The exception that no reduce or accept actions have yet been added. The DFA was created by adding epsilontransitions to the NFA. These are required below for the addition of reduce and accept actions, but after this is complete, we may delete them from the final table since we no longer need them.To add reduce and accept actions, we first need to compute the FOLLOW sets for each nonterminal. For purpose of calculating FOLLOW, we add yet another extra start production:

T'' → T$,

To handle end-of-text conditions as described below. This gives us the following result:

FOLLOW(T') = {$}

FOLLOW(T) = {c,$}

FOLLOW(R) = {c,$}

We then add reduce actions by the following rule: If a DFA state's contains the accepting NFA state for a production

p : N → α,

We add reduce 'p' as an action to 's' on all symbols in FOLLOW(N). Reduction on production 0 (the extra start production that was added before constructing the NFA) is written as accept. Hence, we add r3 as actions at the symbols c and $ (as these are in FOLLOW(R)). State '1' contains NFA state B, which accepts production '0'. We add this at the symbol $ (FOLLOW (T')). As noted above, this is written as accept abbreviated to "a". In the same way, we add and reduce actions.

CHAPTER 10

CONFLICTS IN SLR PARSE TABLES

Dr. Gokul Thanigaivasan
Assistant Professor, Department of Data Science & Analytics, School of Sciences,
Jain (Deemed-to-be University), Bangalore-27, India
Email Id- t.gokul@jainuniversity.ac.in

When reduce actions are added to SLR parse-tables, we might add one to a place where there is already a shift action, or we may add reduce actions for several different productions to the same place. When either of these happens, we no longer have a unique choice of action that is we have a conflict. The first situation is called a shift-reduce conflict and the other case is a reduceconflict. Both may occur in the same place.

Conflicts are often caused by ambiguous grammars, but as is the case for LL-parsers even some non-ambiguous grammars may generate conflicts. If a conflict is caused by ambiguous grammar, it is usually but not always possible to find an equivalent unambiguous grammar. Alternatively, operator precedence declarations may be used to disambiguate ambiguous grammar.

1. Add the production S' → S, where S is the start symbol of the grammar.
2. Make an NFA for the right-hand side of each production.
3. If an NFA state's' has an outgoing transition on a nonterminal 'N', add epsilon transitions from 's' to the starting states of the NFAs for the right-hand sides of the productions for 'N'.
4. Convert the combined NFA to a DFA. Use the starting state of the NFA for the products added in step 1 as the starting state for the combined NFA.
5. Build a table cross-indexed by the DFA states and grammar symbols (terminals including $ and non-terminals). Add shift actions for transitions of terminals and go actions for transitions on non-terminals.
6. Calculate FOLLOW for each nonterminal. For this purpose, we add one more start production:

   S''→ S

7. When a DFA state contains an accepting NFA state marked with production number 'p', where the nonterminal for 'p' is 'N', find the symbols in FOLLOW(N) and add a reduce p action in the DFA state at all these symbols. If production 'p' is the production added in step 1, add an accept action instead of a reduce 'p' action.

But even unambiguous grammars may in some cases generate conflicts in SLRtables. In some cases, it is still possible to rewrite the grammar to get around the problem, but in a few cases, the language simply is not SLR. Rewriting unambiguous grammar to eliminate conflicts is somewhat of an art. Investigation of the NFA states that form the problematic DFA state will often help identify the exact nature of the problem, which is the first step towards solving it.

Sometimes, changing production from left-recursive to right-recursive may help, even though left-recursion, in general, is not a problem for SLR-parsers, as it is for LL(1)-parsers.SLR DFA for Grammar IN Figure 1 below.

DFA state	NFA states	Transitions				
		a	b	c	*T*	*R*
0	A, C, E, I, J	s3	s4		g1	g2
1	B					
2	D					
3	F, C, E, I, J	s3	s4		g5	g2
4	K, I, J		s4			g6
5	G			s7		
6	L					
7	H					

Figure 1: Represented the SLR DFA for Grammar.

Using precedence Rules in LR-parse Tables

The conflict arising from the dangling-else ambiguity could be removed by removing one of the entries in the LL(1) parse table. Resolving ambiguity by deleting conflicting actions can also be done in SLRtables. In general, there are more cases where this can be done successfully for SLRparsers than for LL(1)-parsers. In particular, ambiguity in expression grammars like grammar 3.2 can be eliminated this way in an SLR table, but not in an LL(1) table. Most LR-parser generators allow declarations of precedence and associativity for tokens used as infixoperators. These declarations are then used to eliminate conflicts in the parse tables.

There are several advantages to this approach:

i. Ambiguous expression grammars are more compact and easier to read than unambiguous grammars in the style.

ii. The parse tables constructed from ambiguous grammars are often smaller than tables produced from equivalent unambiguous grammars.

iii. Parsing using ambiguous grammars is (slightly) faster, as fewer reductions of the form Exp2 → Exp3 etc. are required.

Using precedence rules to eliminate conflicts is very simple. Grammar 3.2 will generate several conflicts:

i. A conflict between shifting on + and reducing the production

Exp → Exp+Exp.

ii. A conflict between shifting on + and reducing the production

Exp → Exp*Exp.

iii. A conflict between shifting on * and reducing the production

Exp → Exp+Exp.

iv. A conflict between shifting on * and reducing the production

$$Exp \rightarrow Exp*Exp.$$

And several more of similar nature involving - and /, for a total of 16 conflicts. Let us take each of the four conflicts above in turn and see how precedence rules can be used to eliminate them. We use the rules that + and * are both left-associative and that * binds more strongly than +.

i. Expressions like a+b+c cause this problem. A + is the next input symbol to be read after a+b. We now have two options: lower a+b and group around the first addition before even the second, or shift on the plus and group around the second addition before the previous by subsequently reducing b+c. We choose the first of these choices since the rules state that + is left-associative; as a result, we remove the shift-action first from the table and maintain the reduce-action.

ii. These problematic formulations have the formula a*b+c. Again, we choose reduction over shifting since multiplication binds more strongly than addition according to the rules.

iii. The rules again make multiplied bind stronger in formulations of the kind a+b*c, so we shift to avoid groups similar around the + operator and, as a result, remove the reduce-action from the table.

iv. In scenario 4, an operator that is left-associative according to the criteria disagrees with itself. Like in example 1, we deal with this by removing the shift.

In general, the elimination of conflicts by operator precedence declarations can be summarized into the following rules:

i. If there is a conflict between two operators with differing priorities, the operator with the highest priority should take precedence over the operator with the lower priority. The operator utilized in the production that is decreased is the operator linked with a reduce-action.

ii. If there is a conflict between operators with the same priority, the associativity of the operators is utilized. The shiftaction is dropped and the shifted action is kept if the operators are left-associative. In the case of right-associative operators, the shift action is kept and the reduction action is dropped. Both acts are abandoned if the operators are non-associative.

iii. iii. The last of these operators are utilized to determine the priority of the reduce-action if there are several operators with defined precedence in the production that are employed in the reduce-action.

The handling of prefix and postfix operators is comparable. Only the precedence of the prefix and postfix operators counts since associativity only applies to infix operators. Keep in mind that the aforementioned criteria only remove shift-reduce conflicts. The precedence rules of certain parser generators also provide the elimination of reduce-reduce conflicts in which case the production with the highest precedence operator is favoured, although this is less immediately beneficial than the above.

Precedence rules shouldn't always be used to resolve disagreements. You run the danger of the parser accepting just a portion of the intended language if you add precedence rules blindly until

no conflicts are noticed. Unless you have thoroughly examined the parser activities and determined that adding the precedence rules would not have any unfavourable effects, you should generally only use precedence declarations to express operator hierarchies.

Declarations and Actions

Each nonterminal and terminal is given a datatype that is defined and allocated. The values that go along with the tokens that come from the laxer, such as numeric values or names of identifiers, are stored in the datatype for a terminal. The values created for a nonterminal during the reduce-actions parsing process utilize the type. Although theoretically speaking text parsing produces a syntax tree for that string, parser generators sometimes provide the user more control over the output.

To do this, each production is assigned an action. The action is a piece of programming language that determines the value of a production that is being lowered by using the values connected with the symbols on the right-hand side. For example, parsing an expression and applying the right actions to each production may result in computing the expression's numerical value. In reality, compilers may be built such that the result produced during parsing is the compiled code of the programme. To operate on all but the most basic compilers, it is desirable to generate some kind of syntax representation during parsing and then act on this representation.

Abstract Syntax

The syntax trees are not always optimally suitable for compilation. They contain a lot of redundant information: Parentheses, keywords used for grouping purposes only, and so on. They also reflect structures in the grammar that are only introduced to eliminate ambiguity or to get the grammar accepted by a parser generator such as left-factorization or elimination of left recursion. Hence, the abstract syntax is commonly used. Abstract syntax keeps the essence of the structure of the text but omits the irrelevant details. An abstract syntax tree is a tree structure where each node corresponds to one or more nodes in the concrete syntax tree. For example, the concrete syntax tree shown in Figure 2 may be represented by the following abstract syntax tree:

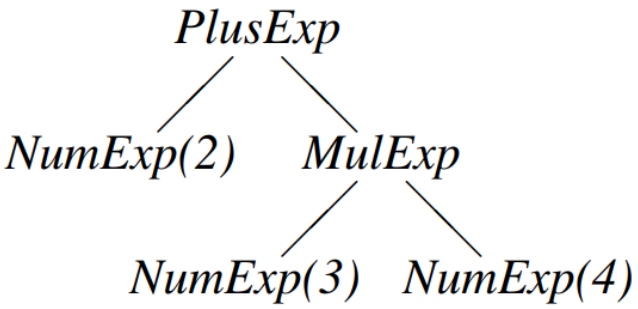


Figure 2: Represented the Abstract Syntax Tree.

Here the names ‘PlusExp’, ‘MulExp’ and ‘NumExp’ may be constructors in a datatype, they may be elements from an enumerated type used as tags in a uniontype or they may y be names of subclasses of an Exp class. The names indicate which product is chosen, so there is no need to keep the sub-trees that are implied by the choice of production, such as the sub-tree that holds the symbol +. Likewise, the sequence of nodes Exp, Exp2, and Exp3, at the left of Figure XYZ are combined into a single node NumExp(2) that includes both the choice of productions for Exp,

Exp2 and Exp3 and the value of the terminal node. In short, each node in the abstract syntax tree corresponds to one or more nodes in the concrete syntax tree.

A designer of a compiler or interpreter has much freedom in the choice of abstract syntax. Some use abstract syntax that retains all of the structure of the concrete syntax trees plus additional positioning information used for errorreporting. Others prefer abstract syntax that contains only the information necessary for compilation or interpretation, skipping parentheses and others for compilation or interpretation of irrelevant structure, as we did above. Exactly how the abstract syntax tree is represented and built depends on the parser generator used. Normally, the action assigned to production can access the values of the terminals and nonterminal on the right-hand side of production through specially named variables often called $1, $2, etc. and produces the value for the node corresponding to the left-hand-side either by assigning it to a special variable ($0) or letting it be the value of an action expression.

The data structures used for building abstract syntax trees depend on the language. Most statically typed functional languages support tree-structured datatypes with named constructors. In such languages, it is natural to represent abstract syntax by one datatype per syntactic category (e.g., Exp above) and one constructor for each instance of the syntactic category (e.g., PlusExp, NumExp and MulExp above). In Pascal, each syntactic category can be represented by a variant record type and each instance as a variant of that. In C, a syntactic category can be represented by a union of structs, each struct representing an instance of the syntactic category and the union covering all possible instances. In object-oriented languages such as Java, a syntactic category can be represented as an abstract class or interface where each instance in a syntactic category is a concrete class that implements the abstract class or interface.

In most cases, it is fairly simple to build abstract syntax using the actions for the productions in the grammar. It becomes complex only when the abstract syntax tree must have a structure that differs nontrivially from the concrete syntax tree. One example of this is if left-recursion has been eliminated to make an LL(1) parser. The preferred abstract syntax tree will in most cases be similar to the concrete syntax tree of the original left-recursive grammar rather than that of the transformed grammar. As an example, the left-recursive grammar:

E → E +num

E → num

Gets transformed by left-recursion elimination into

E → numE’

E’→ +numE’

E’ →

Which yields a completely different syntax tree. We can use the actions assigned to the productions in the transformed grammar to build an abstract syntax tree that reflects the structure in the original grammar. In the transformed grammar, E’ should return an abstract syntax tree with a *hole*. The intention is that this hole will eventually be filled by another abstract syntax tree:

- The second production for E’ returns just a hole.

- In the first production for E', the + and num terminals are used to produce a tree for a plus-expression (i.e., a PlusExp node) with a hole in place of the first subtree. This tree is used to fill the hole in the tree returned by the recursive use of E', so the abstract syntax tree is essentially built outsidein. The result is a new tree with a hole.
- In the production for E, the hole in the tree returned by the E' nonterminal is filled by a NumExp node with the number that is the value of the num terminal.

The best way of building trees with holes depends on the type of language used to implement the actions. Let us first look at the case where a functional language is used. The actions shown below for the original grammar will build an abstract syntax tree similar to the one shown at the beginning of this section.

E → E +num{ PlusExp($1,NumExp($3)) }

E → num{ NumExp($1) }

We now want to make actions for the transformed grammar that will produce the same abstract syntax trees as this will. In functional languages, an abstract syntax tree with a hole can be represented by a function. The function takes as an argument what should be put into the hole and returns a syntax tree where the hole is filled with this argument. The hole is represented by the argument variable of the function. We can write this as actions to the transformed grammar:

E → numE' { $2(NumExp($1)) }

E'→ +numE' { λx.$3(PlusExp(x,NumExp($2))) }

E→ {λx.x }

Where λx.e is a nameless function that takes x as argument and returns the value of the expression e. The empty production returns the identity function, which works like a top-level hole. The non-empty production for E' applies the function $3 returned by the E' on the right-hand side to a subtree, hence filling the hole in $3 by this subtree. The sub-tree itself has a hole x, which is filled when applying the function returned by the right-hand side. The production for E applies the function $2 returned by E' to a sub-tree that has no holes and, hence, returns a tree with no holes.

In SML, λx.e is written as fn x => e, in Haskell as \x -> e and in Scheme as (lambda (x) e).

The imperative version of the actions in the original grammar is:

E → E +num{ $0 = PlusExp($1,NumExp($3)) }

E → num{ $0 = NumExp($1) }

In this setting, NumExp and PlusExp are not constructors but functions that allocate and build nodes and return pointers to these. Unnamed functions of the kind used in the above solution for functional languages cannot be built in most imperative languages, so holes must be an explicit part of the datatype that is used to represent abstract syntax. These holes will be overwritten when the values are supplied. E' will, hence, return a record holding both an abstract syntax tree in a field named tree and a pointer to the hole that should be overwritten in a field named hole.

As actions (using C-style notation), this becomes:

```
E → numE'      { $2->hole = NumExp($1);
                 $0 = $2.tree }
E'→ +numE'     { $0.hole = makeHole();
                 $3->hole = PlusExp($0.hole,NumExp($2));
                 $0.tree = $3.tree }
E'→            { $0.hole = makeHole();
                 $0.tree = $0.hole }
```

This may look bad, but left-recursion removal is rarely needed when using LR-parser generators. An alternative approach is to let the parser build an intermediate (semi-abstract) syntax tree from the transformed grammar, and then let a separate pass restructure the intermediate syntax tree to produce the intended abstract syntax. Some LL(1) parser generators can remove leftrecursion automatically and will afterwards restructure the syntax tree so it fits the original grammar.

Conflict Handling in Parser Generators

The user of a parser generator should anticipate conflicts to be recorded when the grammar is initially provided to the parser generator for all but the simplest grammars. These conflicts may arise from ambiguity or the parsing technique's limits. In any case, disagreements may often be resolved by editing the language or adding precedence statements.Textual Representation of NFA States in Figure 3.

NFA-state	Textual representation
A	`T' -> . T`
B	`T' -> T .`
C	`T -> . R`
D	`T -> R .`
E	`T -> . aTc`
F	`T -> a . Tc`
G	`T -> aT . c`
H	`T -> aTc .`
I	`R -> .`
J	`R -> . bR`
K	`R -> b . R`
L	`R -> bR .`

Figure 3: Represented that the Textual Representation of NFA States.

Most parser generators can provide information that is useful to locate where in the grammar the problems are. When a parser generator reports conflicts, it will tell in which state in the table

these occur. This state can be written out in a barely human-readable form as a set of NFAstates. Since most parser generators rely on pure ASCII, they cannot draw the NFAs as diagrams. Instead, they rely on the fact that each state in the NFA corresponds to a position in production in grammar. If we, for example, look at the NFA states, these would be written as yyy. Note that a '.' is used to indicate the position of the state in the production. State 4 of the table will hence be written as:

R→ b . R

R → .

R → . bR

The set of NFA states, combined with information about which symbols a conflict occurs, can be used to find a remedy, e.g. by adding precedence declarations. If all efforts to get grammar through a parser generator fail, a practical solution may be to change the grammar so it accepts a larger language than the intended language and then post-process the syntax tree to reject "false positives". This elimination can be done at the same time as type-checking.

Properties of Context-free Languages:

We discussed a few regular language characteristics. Some of them, but not all, are shared by context-free languages. Deterministic finite automata for regular languages include the same group of languages as nondeterministic automata. The situation for context-free languages is different: All context-free languages are covered by nondeterministic stack automata, but only a tight subset is covered by deterministic stack automata. Deterministic context-free languages are the subset of context-free languages that can be understood by deterministic stack automata. LR parsers can recognize context-free, deterministic languages.

We have noticed that finiteness is the fundamental restriction of regular languages: The inability to count infinitely prevents a finite automaton from keeping track of matching parentheses or other features. Such counting is possible in context-free languages, which effectively use the stack for this. However, there are some restrictions: While it is feasible to express the language "a^nb^n | n≥0" by a context-free grammar, the language "$a^nb^nc^n$ | n≥0" is not a context-free language since it can only keep track of one object at a time. Further limiting the languages that may be expressed is the tight LIFO sequence in which the information is retained on the stack. The language of palindromes, or strings that read the same both forward and backwards, is easy to describe by context-free grammar, but the language of strings that can be created by repeating a string twice is not.

Context-free languages are closed under union, just like regular languages, and it is simple to create a grammar for the union of two languages given grammars for each. Additionally closed under prefixes, suffixes, subsequences, and reversal are context-free languages. A context-free language that consists just of its subsequences is regular. Languages that lack context, however, are not closed under intersection or complement. For instance, the languages "a^nb^n c^m | m,n ≥0" and "$a^mb^nc^n$ | m,n≥0" are both context-free, but their intersection "$a^nb^nc^n$ | n≥0" is not, and the language complement is not either.

CHAPTER 11

SCOPES AND SYMBOL TABLES

Dr. Santosh S Chowhan
Assistant Professor, Department of Data Science & Analytics, School of Sciences,
Jain (Deemed-to-be University), Bangalore-27, India
Email Id- santosh.sc@jainuniversity.ac.in

The ability to name things like variables, functions, and types is a key idea in programming languages. There will be a declaration for each of these named objects, in which the name is established as a synonym for the object. We refer to this as binding. Each name will also be used a variety of times to refer to the item to which it is tied. A name's declaration often only affects a small piece of the programme. Where the name may be seen. These declarations are known as local declarations, as opposed to global declarations, which make the stated name visible across the whole programme. It is possible for the same name to be defined in many nested scopes. In this situation, it is typical for that usage of the name to be defined by the declaration that is nearest to it. The term nearest in this instance refers to the program's syntax tree: A declaration's scope is a sub-tree of the syntax tree, and nested declarations result in scopes that are themselves nested sub-trees. The declaration of a name that is closest to the usage of the name is thus the one that corresponds to the smallest subtree that contains the use of the name. Take a look at this C statement block as an illustration:

```
{
        int x = 1;
        int y = 2;
        {
           double x = 3.14159265358979;
           y += (int)x;
        }
           y += x;
        }
```

Integer variables x and y are declared in the two lines that follow the initial opening brace, with scope up to the closing brace in the final line. The second opening brace establishes a new scope and declares the floating-point variable x with an initial value near to. The original x variable is hidden until the inner scope expires since this will have scope up to the first closing brace. The formula y += (int)x; adds 3 to y, making its new value 5. The initial x is restored since we have left the inner scope in the subsequent assignment, y += x;. As a result, the assignment will add 1 to y, giving it a final value of 6. The most frequent scoping rule in contemporary programming languages is static or lexical binding, which bases scoping on the structure of the syntax tree as seen in the example. =The remainder of this chapter and the rest of the book will assume that

static binding is being utilised. Dynamic binding is a feature of a few languages that determines the name's current usage based on the declaration that was most recently encountered while the programme was being executed. The methods that are described as being utilised in a compiler in the remainder of this chapter must be employed at run-time if the language supports dynamic binding since dynamic binding cannot be resolved at compile-time by nature. To ensure that each usage of a name is accurately ascribed to its declaration, a compiler will need to keep track of names and the objects with which they are associated. A symbol table is often used to do this or environment, as it is sometimes called.

Symbol Tables

A symbol table is a table that binds names to information. We need several operations on symbol tables to accomplish this:

A. We need an empty symbol table, in which no name is defined.

B. We need to be able to bind a name to a piece of information. In case the name is already defined in the symbol table, the new binding takes precedence over the old.

C. We need to be able to look up a name in a symbol table to find the information the name is bound to. If the name is not defined in the symbol table, we need to be told that.

D. We need to be able to enter a new scope.

E. We need to be able to exit a scope, reestablishing the symbol table to what it was before the scope was entered.

Implementation of Symbol Tables:

The most crucial difference between the various symbol table implementations is how scopes are handled. This may be accomplished either with an imperative or destructively-updated data structure or with a durable or functional data structure. A persistent data structure has the feature that it cannot be destroyed by an operation. The previous structure is preserved unmodified because conceptually, each time an operation modifies the data structure, a new updated copy of the data structure is created.

This implies that because the original symbol table was saved by the persistent nature of the data structure, it is simple to reconstruct it when a scope ends. In reality, when a symbol table is changed, only a little piece of the data structure is duplicated; the majority is shared with the preceding version.

Since there is only one copy of the symbol table in the imperative method, specific actions are needed to save the data necessary to roll back the symbol table to a prior state. An auxiliary stack may be used to do this. The previous binding for a name that is overwritten is stored (pushed) on the auxiliary stack whenever an update is done. An entry marker is put onto the auxiliary stack when a new scope is entered.

The bindings on the auxiliary stack (down to the marker) are used to reconstruct the previous symbol table whenever the scope is terminated.

As a result, the bindings and the marker are removed from the auxiliary stack, putting it back in the position it was in before the scope was entered. Below, we'll examine the basic

implementations of both strategies and talk about how more complex strategies might address some of the efficiency issues with simpler strategies.

Simple Persistent Symbol Tables

In functional languages like SML, Scheme or Haskell, persistent data structures are the norm rather than the exception which is why persistent data structures are sometimes called functional data structures. For example, when a new element is added to the front of a list or an element is taken off the front of the list, the old list still exists and can be used elsewhere. A list is a natural way to implement a symbol table in a functional language: A binding is a pair of a name and its associated information, and a symbol table is a list of such pairs. The operations are implemented in the following way:

- **empty:** An empty symbol table is an empty list.
- **binding:** A new binding (name/information pair) is added (consed) to the front of the list.
- **lookup:** The list is searched until a pair with a matching name is found. The information paired with the name is then returned. If the end of the list is reached, an indication that this happened is returned instead. This indication can be made by raising an exception or by letting the lookup function return a special value representing "not found". This requires a type that can hold both normal information and this special value, i.e., a sumtype.
- **enter:** The old list is remembered, i.e., a reference is made to it.
- **exit:** The old list is recalled, i.e., the above reference is used.

The latter two operations are not explicit, as the variable used to hold the symbol table before entering a new scope will still hold the same symbol table after the scope is exited. So all that is needed is a variable to hold (a reference to) the symbol table. As new bindings are added to the front of the list and the list is searched from the front to the back, bindings in inner scopes will automatically take precedence over bindings in outer scopes. Another functional approach to symbol tables is using functions: A symbol table is quite naturally seen as a function from names to information. The operations are:

- **empty:** An empty symbol table is a function that returns an error indication (or raises an exception) no matter what its argument is.
- **binding:** Adding a binding of the name n to the information i in a symbol table t is done by defining a new symbol-table function t 0 in terms t and the new binding. When t 0 is called with the name n1 as argument, it compares n1 to n. If they are equal, t 0 returns the information i. Otherwise, t 0 calls t with n1 as argument and returns the result that this call yields. In Standard ML, we can define a binding function this way:

  bind (n,i,t) = fn n1 => if n1=n then i else t n1
- **lookup:** The symbol-table function is called with the name as argument.
- **enter:**The old function is remembered referenced.
- **exit:** The old function is recalled by using a reference.

Again, the latter two operations are mostly implicit.

A Simple Imperative Symbol Table

Imperative symbol tables are natural to use if the compiler is written in an imperative language. A simple imperative symbol table can be implemented as a stack, which works in a way similar to the list-based functional implementation:

- **empty:** An empty symbol table is an empty stack.
- **binding:** A new binding (name/information pair) is pushed on top of the stack.
- **lookup:** The stack is searched top-to-bottom until a matching name is found. The information paired with the name is then returned. If the bottom of the stack is reached, we instead return an errorindication.
- **enter:** We push a marker on the top of the stack.
- **exit:** We pop bindings from the stack until a marker is found.

This is also popped from the stack. Note that since the symbol table is itself a stack, we don't need the auxiliary stack. This is not quite a persistent data structure, as leaving a scope will destroy its symbol table. For simple languages, this won't matter, as scope isn't needed again after it is exited. But language features such as classes, modules and lexical closures can require symbol tables to persist after their scope is exited. In these cases, a real persistent symbol table must be used, or the needed parts of the symbol table must be copied and stored for later retrieval before exiting a scope.

Efficiency Issues

Although the aforementioned solutions are all straightforward, they all have the same efficiency issue. Since lookup is accomplished using linear search, the worst-case lookup time is inversely correlated with the size of the symbol table. This is mostly a library-related issue: Software often makes use of libraries that define hundreds of names. Hashing is a popular method for addressing this issue: To index an array, names are hashed processed into integers. The bindings of names with the same hash code are then listed in a linear order for each member of the array.

These lists will often be quite short if the hash table is big enough, making the lookup time almost constant. Entering and leaving scopes is a little more challenging when using hash tables. Although each hash table element is a list that may be handled similarly in simple situations, doing this for every element of an array at each entry and exit adds a significant cost. Instead, imperative solutions often employ a single auxiliary stack to keep track of all table modifications so they may be undone in a period proportionate to the number of updates that were performed in the local scope. Persistent hash tables are often used in functional implementations, which solves the issue.

Shared or Separate Name Spaces

A variable and a function in the same scope may have the same name in certain languages (like Pascal) since the context of usage will indicate whether a variable or a function is utilised. Since declaring a name in one namespace doesn't impact the same name in the other, we say that functions and variables have different namespaces. Variables and functions cannot be distinguished in other languages based on context. As a result, a function defined in an outer

scope may be hidden by a local variable declaration or vice versa. The namespaces for variables and functions in these languages are similar.

For all the many names that might exist in a programme, such as variables, functions, types, exceptions, constructors, classes, field selectors, etc., namespaces may be shared or distinct. Language determines which shared name spaces exist. One symbol table per namespace may be used to construct distinct namespaces, but shared name spaces naturally use a single symbol table. Even though there are several namespaces, there are occasions when it is more convenient to utilise a single symbol table. The names may be readily changed by including name-space indications. A name-space indication may be a tag that is used in conjunction with the name or it can be a textual prefix to the name. In either scenario, a lookup in the symbol table requires that the name and name-space indication of the symbol being looked up match both those of the entry's name and name-space indicator.

CHAPTER 12

INTERPRETATION

Dr. Thirukumaran Subbiramani
Assistant Professor, Department of Data Science & Analytics, School of Sciences,
Jain (Deemed-to-be University), Bangalore-27, India
Email Id- s.thirukumaran@jainuniversity.ac.in

The abstract syntax tree of a programme exists in memory as a data structure after lexing and parsing. But we haven't yet addressed the necessity for a programme to be run. Interpretation is the most basic kind of programme execution. A programme known as an interpreter performs interpretation by taking a program's abstract syntax tree and executing it by analysing the syntax tree to determine what needs to be done. Similar to how a person might assess a mathematical phrase is as follows: The expression is evaluated piecemeal, beginning with the innermost parenthesis and working outwards until the result of the expression is obtained. The procedure may then be repeated using different values for the variables.

There are several variations, however. An interpreter will leave the formula or, rather, the abstract syntax tree of an expression unchanged and use a symbol table to keep track of the values of variables, as opposed to a human who would copy the text of the formula with variables replaced by values and then write a sequence of more and more reduced copies of the formula until it is reduced to a single value. The interpreter is a function that returns the value of the expression represented by the abstract syntax tree rather than reducing a formula. It accepts an abstract syntax tree and a symbol table as parameters. When evaluating a variable, the function may look up the value in the symbol table and can call itself recursively on various branches of the abstract syntax tree to determine the values of sub-expressions.

The underlying notion behind this procedure which may be expanded to include statements and declarations is the same. A function accepts the program's abstract syntax tree and, if necessary, any additional context data such as a symbol table or the program's input, and it returns the program's output. The interpreter may perform some input and output as unintended consequences. Since the symbol tables are presumed to be durable in this chapter, leaving an inner scope does not automatically restore the symbol table for the outer scope. While we don't need to keep the symbol tables for inner scopes once they are exited in the chapter's main content, we will need them for one of the exercises.

Structure of Interpreter

An interpreter design pattern is one of the behavioural design patterns. The interpreter pattern is used to define a grammatical representation of a language and provides an interpreter to deal with this grammar.

- This pattern involves implementing an expression interface which tells us to interpret a particular context. According to the Figure 1 this pattern is used in SQL parsing, symbol processing engines etc.
- This pattern performs upon a hierarchy of expressions. Each expression here is terminal or non-terminal.

- The tree structure of the Interpreter design pattern is somewhat similar to that defined by the composite design pattern with terminal expressions being leaf objects and non-terminal expressions being composites.
- The tree contains the expressions to be evaluated and is usually generated by a parser. The parser itself is not a part of the interpreter pattern. For Example : Here is the hierarchy of expressions for "+ – 9 8 7" :

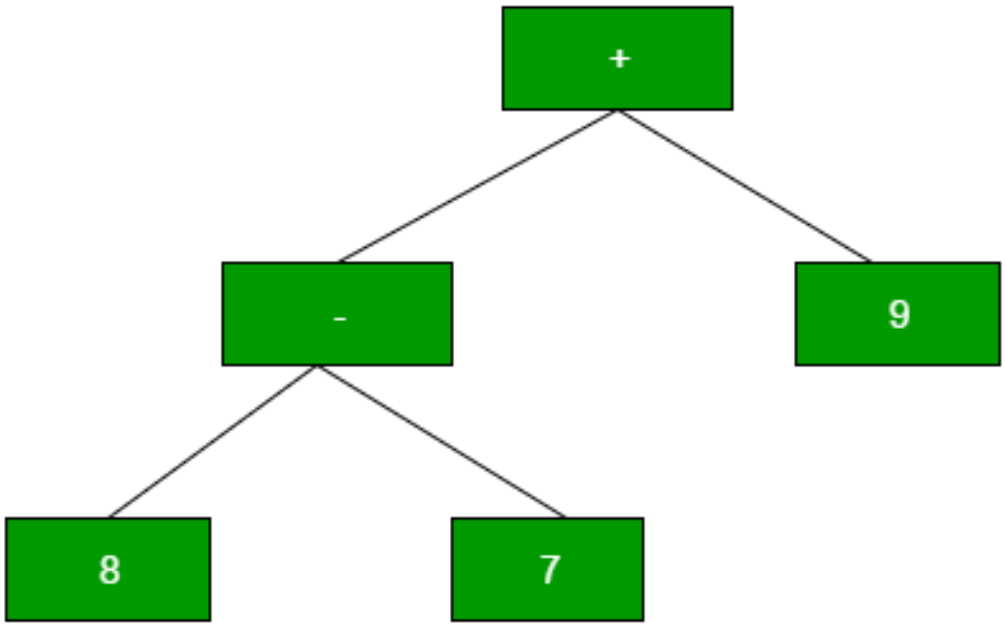


Figure 1: Represented theImplementing the Interpreter Pattern.

UML Diagram Interpreter Design Pattern

- **AbstractExpression** (Expression):

Declares an interpret () operation that all nodes (terminal and nonterminal) in the AST overrides.

- **Terminal Expression** (Number Expression):

Implements the interpret() operation for terminal expressions.

- **Nonterminal Expression**

(Addition Expression, Subtraction Expression, and Multiplication Expression):

Implements the interpret () operation for all nonterminal expressions.

- **Context** (String):

Contains information that is global to the interpreter. It is this String expression with the Postfix notation that has to be interpreted and parsed.

- **Client** (Expression Parser):

Builds or is provided the AST assembled from Terminal Expression and Non-Terminal Expression. The Client invokes the interpret () operation. In Figure 2 the UML Diagram Interpreter Design Pattern.

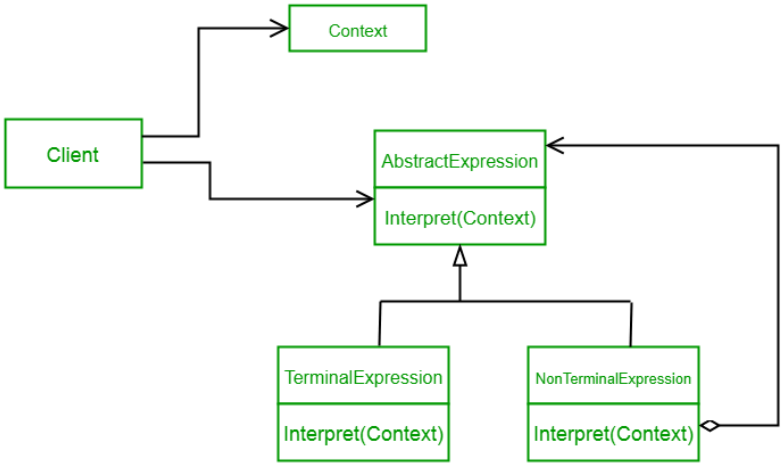


Figure 2: Represented that theUML Diagram Interpreter Design Pattern.

```
// Expression interface used to
// check the interpreter.
interface Expression
{
    booleaninterpreter(String con);
}
// TerminalExpression class implementing
// the above interface. This interpreter
// just check if the data is the same as the
// interpreter data.
class TerminalExpression implements Expression
{
    String data;

    public TerminalExpression(String data)
    {
            this.data = data;
    }
```

```
    public booleaninterpreter(String con)
    {
            if(con.contains(data))
            {
                    return true;
            }
            else
            {
                    return false;
            }
    }
}
// OrExpression class implementing
// the above interface. This interpreter
// just returns the condition of the
// data is the same as the interpreter data.
class OrExpression implements Expression
{
    Expression expr1;
    Expression expr2;

    public OrExpression(Expression expr1, Expression expr2)
    {
            this.expr1 = expr1;
            this.expr2 = expr2;
    }
    public booleaninterpreter(String con)
    {
            return expr1.interpreter(con) || expr2.interpreter(con);
    }
}
// AndExpression class implementing
// the above interface. This interpreter
```

```
// just returns the And condition of the
// data is same as the interpreter data.
class AndExpression implements Expression
{
    Expression expr1;
    Expression expr2;
    public AndExpression(Expression expr1, Expression expr2)
    {
            this.expr1 = expr1;
            this.expr2 = expr2;
    }
    public booleaninterpreter(String con)
    {
            return expr1.interpreter(con) && expr2.interpreter(con);
    }
}
// Driver class
class InterpreterPattern
{
    public static void main(String[] args)
    {
            Expression person1 = new TerminalExpression("Kushagra");
            Expression person2 = new TerminalExpression("Lokesh");
            Expression isSingle = new OrExpression(person1, person2);
            Expression vikram = new TerminalExpression("Vikram");
            Expression committed = new TerminalExpression("Committed");
            Expression isCommitted = new AndExpression(vikram, committed);
            System.out.println(isSingle.interpreter("Kushagra"));
            System.out.println(isSingle.interpreter("Lokesh"));
            System.out.println(isSingle.interpreter("Achint"));
            System.out.println(isCommitted.interpreter("Committed,
Vikram"));
            System.out.println(isCommitted.interpreter("Single, Vikram"));
```

```
    }
}
```

The output of this program is that:

```
true
true
false
true
false
```

In the above code, we are creating an interface Expression and concrete classes implementing the Expression interface. A class TerminalExpression is defined which acts as a main interpreter and other classes OrExpression, AndExpression is used to create combinational expressions.

Advantages

- It's easy to change and extend the grammar. Because the pattern uses classes to represent grammar rules, you can use inheritance to change or extend the grammar. Existing expressions can be modified incrementally, and new expressions can be defined as variations on old ones.
- Implementing the grammar is easy, too. Classes defining nodes in the abstract syntax tree have similar implementations. These classes are easy to write, and often their generation can be automated with a compiler or parser generator.

Disadvantages

- Complex grammars are hard to maintain. The Interpreter pattern defines at least one class for every rule in the grammar. Hence grammars containing many rules can be hard to manage and maintain.

Type Checking

Lexing and parsing will reject many texts as not being correct programs. However, many languages have well-formed requirements that cannot be handled exclusively by the techniques seen so far. These requirements can, for example, be static type correctness or a requirement that pattern-matching or casestatements are exhaustive. These properties are most often not context-free, i.e., they cannot be checked by the membership of a context-free language. Consequently, they are checked by a phase that conceptually comes after syntax analysis though it may be interleaved with it.

These checks may happen in a phase that does nothing else, or they may be combined with the actual execution or translation to another language. Often, the translator may exploit or depend on type information, which makes it natural to combine the calculation of types with the actual translation. We covered type-checking during execution, which is normally called dynamic typing. Will in this chapter assume that type checking and related checks are done in a phase previous to execution or translation (i.e., static typing), and similarly assume that any information gathered by this phase is available in subsequent phases.

The Design Space of Types

We have already discussed the difference between static and dynamic typing, i.e., if type checks are made before or during the execution of a program. Additionally, we can distinguish weakly and strongly typed languages. Strong typing means that the language implementation ensures that whenever an operation is performed, the arguments to the operation are of a type that the operation is defined for, so you, for example, do not try to concatenate a string and a floating-point number. This is independent of whether this is ensured statically before execution or dynamically during execution.Design Space of Types shown in Figure 3.

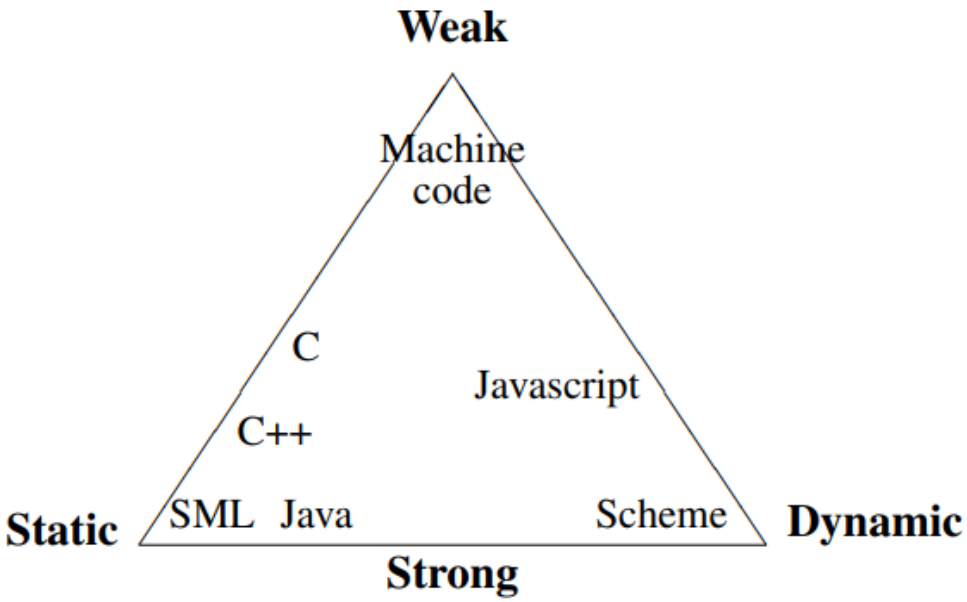


Figure 3: Represented theDesign Space of Types.

In contrast, a weakly typed language gives no guarantee that operations are performed on arguments that make sense for the operation. The archetypical weakly typed language is machine code: Operations are just performed with no checks, and if there is any concept of type at the machine level, it is fairly limited: Registers may be divided into an integer, floating point and possibly address registers, and memory is divided into only code and data. Weakly typed languages are mostly used for system programming, where you need to manipulate move, copy, encrypt or compress data without regard to what the data represents. Many languages combine both strong and weak typing or both static and dynamic typing: Some types are checked before execution and other during execution, and some types are not checked at all.

For example, C is a statically typed language since no checks are performed during execution, but not all types are checked. For example, you can store an integer in a union-typed variable and read it back as a pointer or floating-point number. Another example is JavaScript: If you try to multiply two strings, the interpreter will see if the strings contain sequences of digits and, if so, “read” the strings as numbers and multiply these. This is a kind of weak typing, as the multiplication operation is applied to arguments strings where multiplication does not make sense. But instead of, like machine code, blindly trying to multiply the machine representations of the strings as if they were numbers, JavaScript performs a dynamic check and conversion to make the values conform to the operation.

I will still call this behaviour weak typing, as there is nothing that indicates that converting strings to numbers before multiplication makes any more sense than just multiplying the machine representations of the strings. The main point is that the language, instead of reporting a possible problem, silently does something that probably makes no sense. Figure 6.3 shows a diagram of the design space of static vs. dynamic and weak vs. strong typing, placing some well-known programming languages in this design space. Note that the design space is shown as a triangle: If you never check types, you do so neither statically nor dynamically, so at the weak end of the weak vs. strong spectrum, the distinction between static and dynamic is meaningless.

Attributes

The checking phase operates on the abstract syntax tree of the program and may make several passes over this. Typically, each pass is a recursive walk over the syntax tree, gathering information or using information gathered in earlier passes. Such information is often called attributes of the syntax tree. Typically, we distinguish between two types of attributes: Synthesized attributes are passed upwards in the syntax tree, from the leaves up to the root. Inherited attributes are, conversely, passed downwards in the syntax tree. Note, however, that information that is synthesized in one sub-tree may be inherited by another sub-tree or, in a later pass, by the same sub-tree. An example of this is a symbol table: This is synthesized by a declaration and inherited by the scope of the declaration.

When declarations are recursive, the scope may be the same syntax tree as the declaration itself, in which case one pass over this tree will build the symbol table as a synthesized attribute while a second pass will use it as an inherited attribute. Typically, each syntactic category represented by a type in the data structure for the abstract syntax tree or by a group of a related nonterminal in the grammar will have its own set of attributes.

When we write a checker as a set of mutually recursive functions, there will be one or more such functions for each syntactical category. Each of these functions will take inherited attributes including the syntax tree itself as arguments and return synthesized attributes as results. We will, in this chapter, focus on type checking, and only briefly mention other properties that can be checked. The methods used for type checking can in most cases easily be modified to handle such other checks.

Environments for Type Checking

To type-check the program, we need symbol tables that bind variables and functions to their types. Since there are separate namespaces for variables and functions, we will use two symbol tables, one for variables and one for functions. A variable is bound to one of two typesint or bool. A function is bound to its type, which consists of the types of its arguments and the type of its result.

Function types are written as a parenthesized list of the argument types, an arrow and the result type, e.g., (int,bool) → int for a function taking two parameters of type int and bool, respectively and returning an integer. We will assume that symbol tables are persistent, so no explicit action is required to restore the symbol table for the outer scope when exiting an inner scope. We don't need to preserve symbol tables for inner scopes once these are exited so a stack-like behaviour is fine.

Type Checking Expressions

The symbol tables for variables and functions are inherited properties when we type check expressions. The expression's type (int or bool) is given back as a synthesised attribute. We will allow the type checker function to utilise a notation similar to the concrete syntax for pattern-matching purposes to make the presentation independent of any particular data structure for abstract syntax. However, you should continue to see it as abstract syntax since all ambiguity concerns, etc., have been fixed.

We presume that there are built-in methods for extracting characteristics from terminals'variable names and numeric constants. As a result, id contains a method called get name that returns the name of the identifier. Similar to get value, num's get value function returns the number's value. It is not necessary for static type checking to have the latter. We create one or more functions that accept an inherited attribute and an abstract syntax subtree as parameters and return the synthesised attributes for each nonterminal.

Table 1: Represented that the Different Expression.

$Check_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	
num	`int`
id	$t = lookup(vtable, getname(\textbf{id}))$ *if* $t = unbound$ *then* **error**(); `int` *else* t
Exp_1 + Exp_2	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ *if* $t_1 =$ `int` *and* $t_2 =$ `int` *then* `int` *else* **error**(); `int`
Exp_1 = Exp_2	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ *if* $t_1 = t_2$ *then* `bool` *else* **error**(); `bool`
`if` Exp_1 `then` Exp_2 `else` Exp_3	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ $t_3 = Check_{Exp}(Exp_3, vtable, ftable)$ *if* $t_1 =$ `bool` *and* $t_2 = t_3$ *then* t_2 *else* **error**(); t_2
id ($Exps$)	$t = lookup(ftable, getname(\textbf{id}))$ *if* $t = unbound$ *then* **error**(); `int` *else* $((t_1, \ldots, t_n) \rightarrow t_0) = t$ $[t'_1, \ldots, t'_m] = Check_{Exps}(Exps, vtable, ftable)$ *if* $m = n$ *and* $t_1 = t'_1, \ldots, t_n = t'_n$ *then* t_0 *else* **error**(); t_0
`let` **id** = Exp_1 `in` Exp_2	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $vtable' = bind(vtable, getname(\textbf{id}), t_1)$ $Check_{Exp}(Exp_2, vtable', ftable)$

We display the type-checking function for expressions in Table 1. CheckExp is the name of the function used to type check expressions. The parameters vtable and ftable respectively provide the symbol tables for variables and functions. A type error is reported by the function error. After the error-reporting function reports a type error, the type checker can make an educated guess as to what the type should have been and return this guess, enabling type checking to continue for the remainder of the programme. This allows the type checker to continue and report more than one error. However, if this assumption is incorrect, it may lead to the reporting of erroneous type errors in the future. Therefore, all type error messages should be regarded with a grain of salt except the first one.

- The type of a variable is found by looking its name up in the symbol table for variables. If the variable is not found in the symbol table, the lookupfunction returns the special value unbound. When this happens, an error is reported and the type checker arbitrarily guesses that the type is int. Otherwise, it returns the type returned by lookup.
- A plusexpression requires both arguments to be integers and has an integer result.
- Comparison requires that the arguments have the same type. In either case, the result is a Boolean.
- In a conditional expression, the condition must be of type bool and the two branches must have identical types. The result of a condition is the value of one of the branches, so it has the same type as these. If the branches have different types, the type checker reports an error and arbitrarily chooses the type of the then-branch as its guess for the type of the whole expression.
- At a function call, the function name is looked up in the function environment to find the number and types of the arguments as well as the return type. The number of arguments to the call must coincide with the expected number and their types must match the declared types. The resulting type is the return type of the function. If the function name is not found in ftable, an error is reported and the type checker arbitrarily guesses the result type to be int.
- A letexpression declares a new variable, the type of which is that of the expression that defines the value of the variable. The symbol table for variables is extended using the function bind, and the extended table is used for checking the bodyexpression and finding its type, which in turn is the type of the whole expression. A let-expression cannot in itself be the cause of a type error (though its parts may), so no testing is done.

Since $\text{Check}_{\text{Exp}}$ mentions the nonterminal Exps and its related type-checking function $\text{Check}_{\text{Exps}}$, we have included $\text{Check}_{\text{Exps}}$. $\text{Check}_{\text{Exps}}$ builds a list of the types of expressions in the expression list. The notation is taken from SML: A list is written in square brackets with commas between the elements. The operator :: adds an element to the front of a list.

Type Checking a Program

A program is a list of functions and is deemed type correct if all the functions are type correct, and no two function definitions are defining the same function name. Additionally, there must be a function called main with one integer argument and integer result. Since all functions are mutually recursive, each of these must be type checked using a symbol table where all functions are bound to their type. This requires two passes over the list of functions: One to build the

symbol table and one to check the function definitions using this table. Hence, we need two functions operating over Funs and two functions operating over Fun.

We have already seen one of the latter, $Check_{Fun}$. The other, Get_{Fun}, returns the pair of the function's declared name and type, which consists of the types of the arguments and the type of the result. It uses an auxiliary function Get_{Types} to find the types of the arguments. The two functions for the syntactic category Funs are Get_{Funs}, which builds the symbol table and checks for duplicate definitions, and $Check_{Funs}$, which calls $Check_{Fun}$ for all functions. These functions and the main function $Check_{Program}$, which ties the loose ends, are shown in figure 6.4. This completes type checking of our small example language.

Advanced Type Checking

Despite its simplicity, our example language does not cover every facet of type verification. Below is a list of some additional features along with short descriptions of how they may be handled.

- **Assignments:**

It is necessary to confirm that the type of something like the value matches the specified variety of the variable before assigning a value to a variable. Before assigning a value to a variable or after it has been assigned, certain compilers may check to see whether the variable could be utilised. Even if they are not strictly type errors, such performance is probably not desired. However, since it depends on non-structural information, testing for such behaviours requires a little more involved analysis than the fundamental type checking described in this chapter.

- **Data Structures:**

A value defined by a data structure could have several components such as a struct, tuple, or record, or it may be of various kinds at different times. Such constructions need to be type checked, hence the type checker has to be able to express their types. Consequently, a data structure that specifies complicated types may be required by the type checker. This could resemble the database structure used for declaring abstract syntax trees. It's important to evaluate the accuracy of operations that create or deconstruct organized data. This may be accomplished in some kind of manner similar to how function calls are verified if each operation on complex information has well-defined types for both its argument and its result.

- **Overloading:**

The term "overloading" involves the usage of the same name for many actions across several kinds. In the preceding language, when = was used to comparing both integers and Booleans, we saw a straightforward illustration of this. Arithmetic operators like + and are defined over integers, floating point numbers, and maybe other kinds of numbers throughout many different languages. All potential situations may be tested one at a moment, much as in our example, if these operators are preset and therefore only cover a limited set of circumstances.

To do this, the operator's several instances must have distinct argument types. The parameter of the text stream alone cannot be employed to choose the appropriate operator, for instance, if the function read is specified to read either integers or floating point numbers from a text stream. As a result, the type checker must transmit the anticipated type of each argument as an inherited property so that it may be used to choose the relevant instance of the overloaded operator potentially in conjunction with the varieties of the arguments. Due to a lack of information, it

may not always be able to send down an anticipated kind. If the type checker is unable to choose a unique operator due to this or another issue, it may indicate "unresolved overloading" as a type error or select a default instance.

Type Conversion

A language may have operators for converting a value of one type to a value of another type, e.g. an integer to a floating-point number. Sometimes these operators are explicit in the program and hence easy to check. However, many languages allow implicit conversion of integers to floats, such that, for example, 3 + 3.12 is well-typed with the implicit assumption that the integer is converted to a float before the addition. This can be handled as follows: If the type checker discovers that the arguments to an operator do not have the correct type, it can try to convert one or both arguments to see if this helps. If there is a small number of predefined legal conversions, this is no major problem. However, a combination of user-defined overloaded operators and user-defined types with conversions can make the type-checking process quite difficult, as the information needed to choose correctly may not be available at compiletime. This is typically the case in object-oriented languages, where method selection is often done at runtime. We will not go into details about how this can be done.

Polymorphism/Generic Types

Some languages allow a function to be polymorphic or generic, that is, to be defined over a large class of similar types, e.g. over all arrays no matter what the types of the elements are. A function can explicitly declare which parts of the type are generic/polymorphic or this can be implicit. The type checker can insert the actual types at every use of the generic/polymorphic function to create instances of the generic/polymorphic type. This mechanism is different from overloading as the instances will be related by a common generic type and because a polymorphic or generic function can be instantiated by any type, not just by a limited list of declared alternatives as is the case with overloading.

Implicit Types

Some languages require programs to be well-typed but do not require explicit type declarations for variables or functions. For such to work, a type inference algorithm is used. A type inference algorithm gathers information about the uses of functions and variables and uses this information to infer the types of these. If there are inconsistent uses of a variable, a type error is reported.

CHAPTER 13

INTERMEDIATE-CODE GENERATION

Dr. Uthama Kumar A
Assistant Professor, Department of Data Science & Analytics, School of Sciences,
Jain (Deemed-to-be University), Bangalore-27, India
Email Id- uthamakumar.a@jainuniversity.ac.in

The final goal of a compiler is to get programs written in a high-level language to run on a computer. This means that, eventually, the program will have to be expressed as machine code which can run on the computer. This does not mean that we need to translate directly from the high-level abstract syntax to machine code. Many compilers use a medium-level language as a stepping-stone between the high-level language and the very low-level machine code. Such stepping-stone languages are called intermediate codes. Apart from structuring the compiler into smaller jobs, using an intermediate language has other advantages:

A. If the compiler needs to generate code for several different machinearchitectures, only one translation to intermediate code is needed. Only the translation from intermediate code to machine language (i.e., the back-end) needs to be written in several versions.
B. If several high-level languages need to be compiled, only the translation to intermediate code needs to be written for each language. They can all share the backend, i.e., the translation from intermediate code to machine code.
C. Instead of translating the intermediate language to machine code, it can be interpreted by a small program written in machine code or a language for which a compiler or interpreter already exists.

The advantage of using an intermediate language is most obvious if many languages are to be compiled by many machines. If the translation is done directly, the number of compilers is equal to the product of the number of languages and the number of machines.If a common intermediate language is used, one front-end i.e., a compiler to intermediate code is needed for every language and one back-end interpreter or code generator is needed for each machine, making the total number of frontends and back-ends equal to the sum of the number of languages and the number of machines. If an interpreter for an intermediate language is written in a language for which there already exist implementations for the target machines, the same interpreter can be interpreted or compiled for each machine. This way, there is no need to write a separate back-end for each machine. The advantages of this approach are:

A. No actual back-end needs to be written for each new machine, as long as the machine "I" is equipped with an interpreter or compiler for the implementation language of the interpreter for the intermediate language.

B. A compiled program can be distributed in a single intermediate form for all machines, as opposed to shipping separate binaries for each machine.

C. The intermediate form may be more compact than machine code. This saves space both in distribution and on the machine that executes the programs (though the latter is somewhat offset by requiring the interpreter to be kept in memory during execution).

The disadvantage is speed: Interpreting the intermediate form will in most cases be a lot slower than executing translated code directly. Nevertheless, the approach has seen some success, e.g., with Java. Some of the speed penalty can be eliminated by translating the intermediate code to machine code immediately before or during the execution of the program. This hybrid form is called just-in-time compilation and is often used for executing the intermediate code for Java. We will this book, however, focus mainly on using the intermediate code for traditional compilation, where the intermediate form will be translated to machine code by a back-end of the compiler.

Choosing an Intermediate language

An intermediate language should, ideally, have the following properties:

A. It should be easy to translate from a high-level language to an intermediate language. This should be the case for a wide range of different source languages.

B. It should be easy to translate from the intermediate language to machine code. This should be true for a wide range of different target architectures.

C. The intermediate format should be suitable for optimizations.

The first two of these attributes may be a little challenging to combine. This should be pretty similar to the language that will serve as the source of translation from a high-level language. For more than a limited percentage of closely related languages, it could be challenging to do this. A high-level intermediate language also adds to the workload on the backends. Back-ends may be simple to create in a low-level intermediate language, but front-ends are constrained to a greater extent. Although this is normally less of an issue than the one with front-ends, since machines are typically more equivalent than high-level languages, a low-level intermediate vocabulary may not match all machines equally.

Having two intermediate levels of which one is quite a high level and the other is fairly low level for the front-ends and back-ends is a possible option that might ease the translation load but does not solve the other issues. The translation between these two discrete formats is subsequently performed by a single shared translator. It makes sense to do as much optimization on the intermediate format as feasible when it is shared by several compilers. By doing so, the work of creating effective optimizations is done just once rather than for each compiler.

The "granularity" is another factor that must be considered when selecting an intermediate language: In the intermediate language, should an operation be equated to a considerable amount of effort or little work? The first of these methods may also be used for compiling. It is often employed when the intermediate language is interpreted since it covers the costs the cost of decoding instructions across a broader quantity of labor. In this situation, each operation in the intermediate code is generally converted into a series of instructions in machine code. There are often a lot of distinct medium operations when coarse-grained intermediate code is employed.

The alternative strategy is to make each operation in the intermediatecode as brief as feasible. This indicates that numerous intermediate-code operations may be integrated into one production line or that each intermediate-code action is normally translated into a single instruction in

machinecode. Since each machine-code directive may be thought of as a series of intermediate-code instructions, the latter can, to some extent, become automated. The code generator may search for sequences that correspond to machine-code as display in Figure 1 operations after converting intermediate representation to machine code. This may be made into a combinatorial optimization challenge where the least-cost solution is obtained by giving a cost to each machine-code step.

Program → [*Instructions*]

Instructions → *Instruction*
Instructions → *Instruction* , *Instructions*

Instruction → LABEL **labelid**
Instruction → **id** := *Atom*
Instruction → **id** := **unop** *Atom*
Instruction → **id** := **id binop** *Atom*
Instruction → **id** := *M*[*Atom*]
Instruction → *M*[*Atom*] := **id**
Instruction → GOTO **labelid**
Instruction → IF **id relop** *Atom* THEN **labelid** ELSE **labelid**
Instruction → **id** := CALL **functionid**(*Args*)

Atom → **id**
Atom → **num**

Args → **id**
Args → **id** , *Args*

Figure 1: Represented the Intermediate Language.

The Intermediate Language

As it is best suited to express the methods we wish to explore, we have decided to use a somewhat low-level, fine-grained intermediate language in this chapter. We won't address function call translation until the body of a function or procedure in a real programme corresponds to a "programme" in our intermediate language for the time being. The intermediate language originally treats function calls as basic actions for the same reason. Grammar demonstrates the grammar for the intermediate language. An ordered list of instructions is a programme. The guidelines are:

A. A label. This has no effect but serves only to mark the position in the program as a target for jumps.

B. An assignment of an atomic expression that is constant or variable to a variable.

C. A unary operator is applied to an atomic expression, with the result stored in a variable. A binary operator is applied to a variable and an atomic expression, with the result stored in a variable.

D. A transfer from memory to a variable. The memory location is an atomic expression.

E. A transfer from a variable to memory. The memory location is an atomic expression.

F. A jump to a label.

G. A conditional selection between jumps to two labels. The condition is found by comparing a variable with an atomic expression by using a relational operator (=, 6=, ≤ or ≥).

H. A function call. The arguments to the function call are variables and the result is assigned to a variable. This instruction is used even if there is no actual result i.e. if a procedure is called instead of a function), in which case the result variable is a dummy variable.

Syntax Directed Translation

We will generate code using translation functions for each syntactic category, similar to the functions we used for interpretation and type checking. We generate code for a syntactic construct independently of the constructs around it, except that the parameters of a translation function may hold information about the context such as symbol tables and the result of a translation function may in addition to the generated code hold information about how the generated code interfaces with its context such as which variables it uses. Since the translation closely follows the syntactic structure of the program, it is called syntax-directed translation. Given that translation of a syntactic construct is mostly independent of the surrounding and enclosed syntactic constructs, we might miss opportunities to exploit synergies between these and, hence, generate less than optimal code. We will try to remedy this in later chapters by using various optimization techniques as mention Figure 2.

$$
\begin{array}{lcl}
Exp & \rightarrow & \textbf{num} \\
Exp & \rightarrow & \textbf{id} \\
Exp & \rightarrow & \textbf{unop}\ Exp \\
Exp & \rightarrow & Exp\ \textbf{binop}\ Exp \\
Exp & \rightarrow & \textbf{id}(Exps) \\
 & & \\
Exps & \rightarrow & Exp \\
Exps & \rightarrow & Exp\ ,\ Exps
\end{array}
$$

Figure 2: Represented the A Simple Expression Language

In syntax-directed translation, along with the grammar we associate some informal notations and these notations are called semantic rules. So we can say that,

Grammar + semantic rule = SDT (syntax directed translation)

- Depending on the kind of attribute, every non-terminal in syntactic directed translation may get one, many, or even no attributes. The semantic rules connected to the production rule assess the value of these qualities.
- In the semantic rule, an attribute is referred to as VAL, and it may include a complicated record, a string, an integer, or a memory address.
- When a construct is encountered in a programming language, Syntax Directed Translation (SDT) translates it following the semantic rules specified in that specific programming language in Table 1.

Table 1: Represented the Semantic Rules.

Sr. No.	Production	Semantic Rules
1.	E → E + T	E.val := E.val + T.val
2.	E → T	E.val := T.val
3.	T → T * F	T.val := T.val + F.val
4.	T → F	T.val := F.val
5.	F → (F)	F.val := F.val
6.	F → num	F.val := num.lexval

Syntax Directed Translation Scheme

- The grammar used by the Syntax Directed Translation system is context-free.
- To assess the order of semantic rules, the syntax-directed translation scheme is utilized.
- In the translation scheme, the productions' right side contains the semantic rules.
- Enclosed within brackets, an action's execution position is shown in Table 2. It is inscribed on the production's right side.

Table 2: Represented the Syntax Directed Translation Scheme.

Sr. No.	Production	Semantic Rules

1.	S → E $	{ printE.VAL }
2.	E → E + E	{E.VAL := E.VAL + E.VAL }
3.	E → E * E	{E.VAL := E.VAL * E.VAL }
4.	E → (E)	{E.VAL := E.VAL }
5.	E → I	{E.VAL := I.VAL }
6.	I → I digit	{I.VAL := 10 * I.VAL + LEXVAL }
7.	I → digit	{ I.VAL:= LEXVAL}

Implementation of Syntax Directed Translation

Syntax direct translation is implemented by constructing a parse tree and performing the actions in left-to-right depth-first order as mentioned in Table 3.SDT is implementing by parse the input and produce a parse tree as a result.

Table 3: Represented that the Implementation of Syntax Directed Translation.

Production	Semantic Rules
S → E $	{ printE.VAL }
E → E + E	{E.VAL := E.VAL + E.VAL }
E → E * E	{E.VAL := E.VAL * E.VAL }
E → (E)	{E.VAL := E.VAL }
E → I	{E.VAL := I.VAL }
I → I digit	{I.VAL := 10 * I.VAL + LEXVAL }
I → digit	{ I.VAL:= LEXVAL}

Intermediate Code

Intermediate code is used to translate the source code into the machine code. Intermediate code lies between the high-level language and the machine language. The block diagram is display in Figure 3.

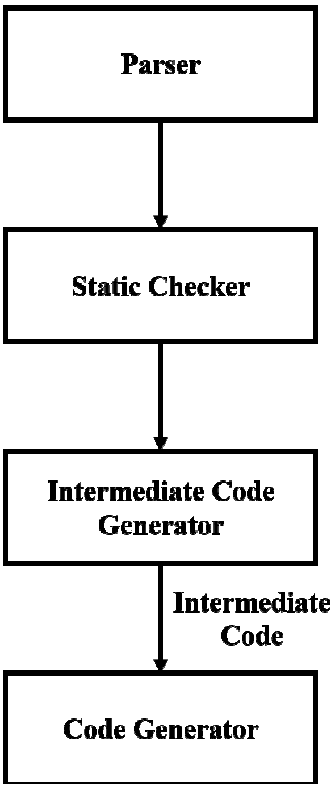


Figure 3: Represented the Position of the Intermediate Code Generator.

- A complete native compiler is needed for every new machine if the compiler directly converts source code into machine code without producing intermediate code.
- The intermediate code ensures that the analysis component is the same for all compilers, eliminating the need for a complete compiler for each system.
- The semantic analyser phase and its predecessor phase provide input to the intermediate code generator. An annotated syntax tree is the only kind of input it accepts.
- The second step of compiler generation is modified following the target machine using the intermediate code.

Postfix Notation

- If the provided language is expressions, postfix notation is a helpful type of intermediate code.
- Suffix notation and reverse polish are other names for postfix notation.
- A syntax tree is represented linearly using postfix notation.
- Any phrase may be stated clearly and without parentheses in the postfix notation.
- The standard (infix) approach to express the product of x and y is as follows: x * y. However, in the postfix notation, the operator is written as xy * at the right end.
- The operator comes after the operand in postfix notation.

Syntax Directed Translation in Compiler Design

As shown in the Figure, the parser employs a CFG (Context-free-Grammar) to check the input string and provide output for the next stage of the compiler. An abstract syntax tree or a parse tree might be the output. Syntax Directed Translation is now used to interleave semantic analysis with the compiler's syntax analysis step. We conceptually analyses the input token stream, construct the parse tree, and then traverse the tree as necessary to assess the semantic rules at the parse tree nodes, using both syntax-directed definition and translation methods. The evaluation of the semantic rules may result in the creation of code, the saving of data in a symbol table, the issuance of error messages, or any other actions. The outcome of evaluating the semantic rules is the translation of the token stream.

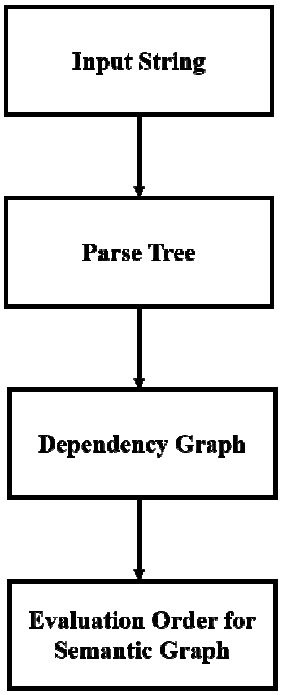


Figure 4: Represented the Syntax Directed Translation in Compiler Design

Syntax Directed Translation which display in Figure 4 has augmented rules to the grammar that facilitate semantic analysis. SDT involves passing information bottom-up and/or top-down to the parse tree in form of attributes attached to the nodes. Syntax-directed translation rules use:

i. Lexical values of nodes,

ii. Constants

iii. Attributes associated with the non-terminals in their definitions.

The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree. For example:

E → E+T | T

T → T*F | F

F → INTLIT

This is a grammar to syntactically validate an expression having additions and multiplications in it. Now, to carry out semantic analysis we will augment SDT rules to this grammar, pass some information up the parse tree and check for semantic errors, if any. In this example, we will focus on the evaluation of the given expression, as we don't have any semantic assertions to check in this very basic example.

E → E+T { E.val = E.val + T.val } PR#1

E → T { E.val = T.val } PR#2

T → T*F { T.val = T.val * F.val } PR#3

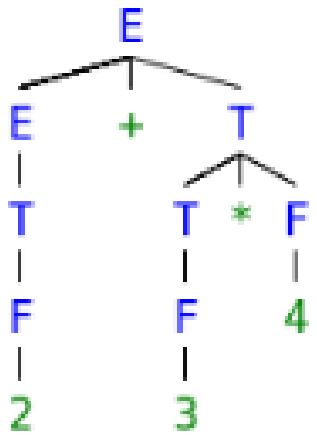
T → F { T.val = F.val } PR#4

F → INTLIT { F.val = INTLIT.lexval } PR#5

For understanding translation rules further, we take the first SDT augmented to [E -> E+T] production rule. The translation rule in consideration has "val" as an attribute for both the non-terminals E & T. Right-hand side of the translation rule corresponds to attribute values of the right-side nodes of the production rule and vice-versa. Generalizing, SDT are augmented rules to a CFG that associate:

i. Set of attributes to every node of the grammar,
ii. A set of translation rules to every production rule using attributes, constants, and lexical values.

Let's take a string to see how semantic analysis happens – S = 2+3*4. Parse tree corresponding to S would be:



To evaluate translation rules, we can employ one depth-first search traversal on the parse tree. This is possible only because SDT rules don't impose any specific order on evaluation until children's attributes are computed before parents for a grammar having all synthesized attributes. Otherwise, we would have to figure out the best-suited plan to traverse through the parse tree and evaluate all the attributes in one or more traversals. For better understanding, we

will move bottom-up in the left to right fashion for computing the translation rules of our example shown in Figure 5.

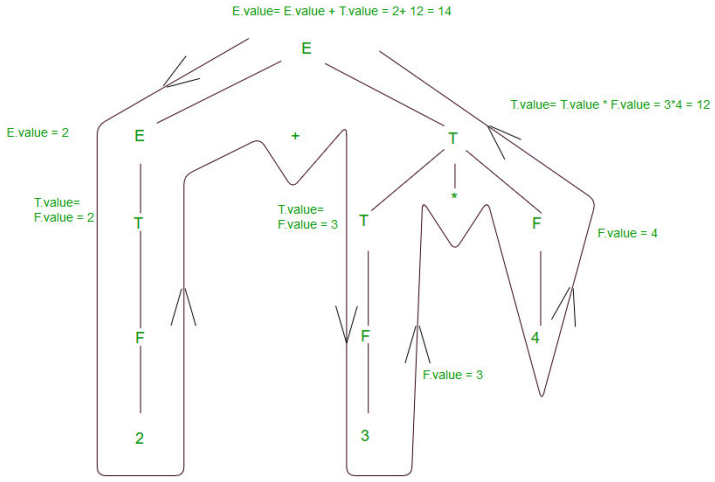


Figure 5: Represented the Semantic Analysis.

The above diagram shows how semantic analysis could happen. The flow of information happens bottom-up and all the children's attributes are computed before parents, as discussed above. Right-hand side nodes are sometimes annotated with subscript 1 to distinguish between children and parents.

Additional Information

Synthesized Attributes are such attributes that depend only on the attribute values of children nodes. Thus **[E -> E+T { E.val = E.val + T.val }]** has a synthesized attribute val corresponding to node E. If all the semantic attributes in an augmented grammar are synthesized, one depth-first search traversal in any order is sufficient for the semantic analysis phase. Inherited Attributes are such attributes that depend on parent and/or sibling's attributes. Thus **[Ep -> E+T { Ep.val = E.val + T.val, T.val = Ep.val }]**, where E & Ep are the same production symbols annotated to differentiate between parent and child, has an inherited attribute val corresponding to node T.

S-attributed and L-attributed SDTs in Syntax Directed Translation

Before coming up with S-attributed and L-attributed SDTs, here is a brief intro to Synthesized or Inherited attributes

Types of Attributes:

Attributes may be of two types – Synthesized or Inherited.

i. Synthesized Attributes:

A Synthesized attribute is an attribute of the non-terminal on the left-hand side of a production. Synthesized attributes represent information that is being passed up the parse tree. The attribute can take value only from its children (Variables in the RHS of the production).For eg. let's say A -> BC is a production of a grammar, and A's attribute is dependent on B's attributes or C's attributes then it will be a synthesized attribute.

ii. Inherited attributes:

An attribute of a nonterminal on the right-hand side of a production is called an inherited attribute. The attribute can take value either from its parent or from its siblingsvariables in the LHS or RHS of the production.For example, let's say A -> BC is a production of a grammar and B's attribute is dependent on A's attributes or C's attributes then it will be an inherited attribute.Now, let's discuss S-attributed and L-attributed SDT.

iii. S-attributed SDT :

E. If an SDT uses only synthesized attributes, it is called an S-attributed SDT.

F. S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

G. Semantic actions are placed in the rightmost place of RHS.

iii. L-attributed SDT:

B. If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attributes can inherit values from left siblings only, it is called an L-attributed SDT.

C. Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.

D. Semantic actions are placed anywhere in RHS.For example,

A -> XYZ {Y.S = A.S, Y.S = X.S, Y.S = Z.S} is not an L-attributed grammar since Y.S = A.S and Y.S = X.S are allowed but Y.S = Z.S violates the L-attributed SDT definition as attributed is inheriting the value from its right sibling, which display in Figure 6.

Note – If a definition is S-attributed, then it is also L-attributed but **NOT** vice-versa.

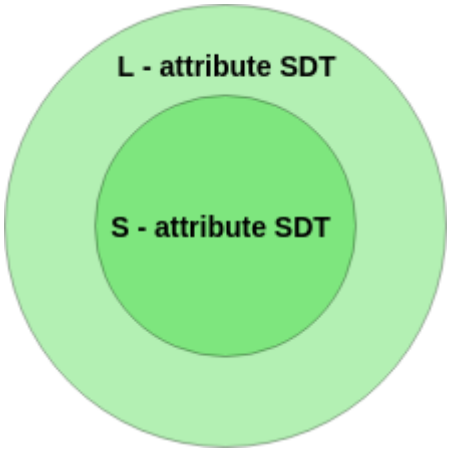


Figure 6: Display that the Grammar of Different Attributes.

Example – Consider the given below SDT.

P1: S ->MN {S.val= M.val + N.val}

P2: M ->PQ {M.val = P.val * Q.val and P.val =Q.val}

- Select the correct option.
- Both P1 and P2 are S attributed.

- P1 is S attributed and P2 is L-attributed.
- P1 is L attributed but P2 is not L-attributed.
- None of the above

Explanation:

The correct answer is option **C** as, In P1, S is a synthesized attribute and in L-attribute definition synthesized is allowed. So P1 follows the L-attributed definition. But P2 doesn't follow L-attributed definition as P is depending on Q which is RHS to it.

CHAPTER 14

RUNTIME ENVIRONMENTS IN COMPILER DESIGN

Dr. Thirukumaran Subbiramani
Assistant Professor, Department of Data Science & Analytics, School of Sciences,
Jain (Deemed-to-be University), Bangalore-27, India
Email Id- s.thirukumaran@jainuniversity.ac.in

A translation needs to relate the static source text of a program to the dynamic actions that must occur at runtime to implement the program. The program consists of names for procedures, identifiers etc., that require mapping with the actual memory location at runtime. The runtime environment is a state of the target machine, which may include software libraries, environment variables, etc., to provide services to the processes running in the system.

Source Language Issues:

A programme is made up of procedures, and in its most basic form, a procedure definition is a declaration that links a statement with a procedure name and body of the procedure. Activation of the process is what is meant by every time a procedure is carried out. The stages that are included in the execution of the operation make up the life of an activation. If "a" and "b" are two procedures, then when one is called after the other, or if they are nested, their activations won't overlap nested procedures. If a fresh activation of the same operation starts before a previous activation of the same procedure has finished, the procedure is recursive. The way control enters and exits activations is shown using an activation tree. Activation trees' characteristics include:

- Every node shows how a process has been activated.
- The root displays the primary function's activation.
- If and only if control flows from process x to procedure y, then the node for procedure "x" is the parent of node for procedure "y."

```
main() {
Int n;
readarray ();
    quicksort (1,n);
}

Quicksort (int m, int n) {
```

```
Inti= partition(m,n);
    quicksort(m,i-1);
    quicksort(i+1,n);
}
```

The activation tree for this program will be shown in Figure 1:

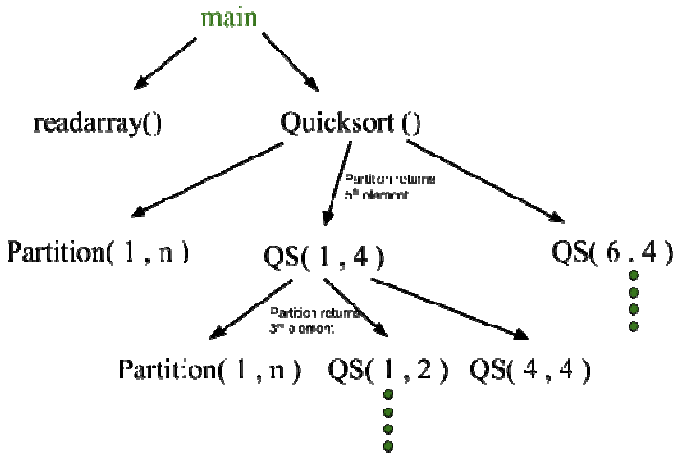


Figure 1: Represented that the Activation Tree.

First main function is root then the main calls read array and quicksort. Quicksort in turn calls partition and quicksort again. The flow of control in a program corresponds to a pre-order depth-first traversal of the activation tree which starts at the root.

Control Stack and Activation Records

Control stack or runtime stack is used to keep track of the live procedure activations i.e the procedures whose execution has not been completed. A procedure name is pushed onto the stack when it is called (activation begins) and it is popped when it returns (activation ends). Information needed by a single execution of a procedure is managed using an activation record or frame. When a procedure is called, an activation record is pushed into the stack and as soon as the control returns to the caller function the activation record is popped.

A general activation record consists of the following things:

- **Local variables:** hold the data that is local to the execution of the procedure.
- **Temporary values**: stores the values that arise in the evaluation of an expression.
- **Machine status:** holds the information about the status of the machine just before the function call.
- **Access link (optional):** refers to non-local data held in other activation records.
- **Control link (optional):** points to the activation record of caller.

- **Return value:** used by the called procedure to return a value to calling procedure
- Actual parameters

Storage Allocation Techniques

i. Static Storage Allocation

- For any program, if we create a memory at compile time, memory will be created in the static area.
- For any program, if we create a memory at compile-time only, memory is created only once.
- It doesn't support dynamic data structure i.e memory is created at compile-time and deallocated after program completion.
- The drawback with static storage allocation is recursion is not supported.
- Another drawback is the size of data should be known at compile time

  Eg- FORTRAN was designed to permit static storage allocation.

ii. Stack Storage Allocation

- Storage is organized as a stack and activation records are pushed and popped as activation begins and end respectively. Locals are contained in activation records so they are bound to fresh storage in each activation.
- Recursion is supported in stack allocation

iii. Heap Storage Allocation

- Memory allocation and deallocation can be done at any time and any place depending on the requirement of the user.
- Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.
- Recursion is supported.

iv. PARAMETER PASSING:

The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism.

v. Basic Terminology

- **R-value:** The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if it appears on the right side of the assignment operator. R-value can always be assigned to some other variable.
- **L-value:** The location of the memory(address) where the expression is stored is known as the l-value of that expression. It always appears on the left side of the assignment operator.

- **Formal Parameter:** Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function.
- **Actual Parameter:** Variables whose values and functions are passed to the called function are called actual parameters. These variables are specified in the function call as arguments.

Different ways of passing the parameters to the procedure:

- **Call by Value:** In call by value the calling procedure passes the r-value of the actual parameters and the compiler puts that into called procedure's activation record. Formal parameters hold the values passed by the calling procedure, thus any changes made in the formal parameters do not affect the actual parameters.
- **Call by Reference:** In call by reference the formal and actual parameters refers to the same memory location. The l-value of actual parameters is copied to the activation record of the called function. Thus the called function has the address of the actual parameters. If the actual parameters do not have a l-value (eg- i+3) then it is evaluated in a new temporary location and the address of the location is passed. Any changes made in the formal parameter are reflected in the actual parameters because changes are made at the address.
- **Call by Reference:** In call by reference the formal and actual parameters refers to the same memory location. The l-value of actual parameters is copied to the activation record of the called function. Thus the called function has the address of the actual parameters. If the actual parameters do not have a l-value (eg- i+3) then it is evaluated in a new temporary location and the address of the location is passed. Any changes made in the formal parameter are reflected in the actual parameters because changes are made at the address.
- **Call by Name** In call by name the actual parameters are substituted for formals in all the places formals occur in the procedure. It is also referred to as lazy evaluation because evaluation is done on parameters only when needed.

Generating Code from Expressions

We'll utilize Grammar 7.2's example of a basic expressions language as our starting point for translation. Once again, we've left the list of unary and binary operators undefined but we're assuming that it contains every operator that the expression language uses. We presum that the name of an operator in the expression language is translated into the name of the equivalent operator in the intermediate language by the function transop. The characteristics of the tokens unop and binop include the names of the actual operators, and the function getopname may access these attributes. We must choose what needs to be done at compile time and what needs to be done at run time while creating a compiler. The majority of work should be completed at compile time, but certain tasks must wait until run time since they need real variable values, etc., which are not available at compilation time.

We may use language like "the expression is evaluated and the result placed in the variable" when describing the operation of the translation functions in the sections that follow. This defines the operations that the code created at compile time does during execution. The notation used in the translation functions makes it apparent what occurs when even when the written explanation isn't always 100% clear: Run-time execution of intermediate language code occurs

while compilation of the remainder takes place. Instructions written in intermediate language may make references to values constants and register names created at compile time. Italicized operands are variables in the compiler that hold compile-time values that are put into the resulting code. These operands are used when instructions have them. For instance, the code template [place:= v] will produce the code [t14:= 42] if the place has the variable name t14 and v holds the value 42.

The main challenge when attempting to translate an expression language to an intermediate language is that the expression language is tree-structured, whereas the intermediate language is flat, necessitating the storage of each operation's result in a variable as well as each (non-constant) argument in a variable.

To create new variable names in the intermediate language, we utilise the function newvar. Every time newvar is used, a variable name that was not previously used is returned. We'll use a notation akin to the notation to illustrate how a translation function translates phrases.

The translation function has certain noticeable characteristics. The code must be sent back as a synthesized attribute. Additionally, it must translate the names of the intermediate language's equivalents of the variables and functions used in the expression language.

This may be accomplished using the vtable and ftable symbol tables, which translate the names of variables and functions in the expression language into their equivalents in the intermediate language. For the translation function, the symbol tables are supplied as inherited attributes. The translation function must make decisions about where to place the values of sub-expressions using characteristics in addition to these. There are two methods to accomplish this:

A. The parent expression, which selects a place for its value, may receive the location of a sub-expressions value as a synthesized property.

B. The parent expression may choose where to look for the values of its child's expressions and may impart this knowledge to those expressions as inherited characteristics.

These are not better than one another. When writing code for variable access, Method 1 has little benefit since it just has to return the name of the variable containing the value rather than writing any code. However, this is only effective if the variable isn't modified before the parent expression uses the value.

This is not always the case, however, as the C expression "x+(x=3)" demonstrates. Expressions may have side effects. The assignment is not a feature of our expression language, but function calls are, and they may have unintended consequences.

This issue is not present in Method 2 since no further side effects are possible between the creation of the expression's value and the execution of the assignment. When the language is eventually expanded to include assignment statements, the second approach also has a modest benefit since we can then produce code that calculates the expression result straight into the target variable rather than having to transfer it from a temporary variable. So, for our translation function TransExp, which is shown in Figure 2, we will use the second way. The intermediate-language variable that the expression's result must be saved in is identified by the inherited attribute location.

$Trans_{Exp}(Exp, vtable, ftable, place) =$ `case` Exp `of`	
num	$v = getvalue(\textbf{num})$ $[place := v]$
id	$x = lookup(vtable, getname(\textbf{id}))$ $[place := x]$
unop Exp_1	$place_1 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $op = transop(getopname(\textbf{unop}))$ $code_1 ++ [place := op\ place_1]$
Exp_1 **binop** Exp_2	$place_1 = newvar()$ $place_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, place_2)$ $op = transop(getopname(\textbf{binop}))$ $code_1 ++ code_2 ++ [place := place_1\ op\ place_2]$
id$(Exps)$	$(code_1, [a_1, \ldots, a_n])$ $= Trans_{Exps}(Exps, vtable, ftable)$ $fname = lookup(ftable, getname(\textbf{id}))$ $code_1 ++ [place :=$ `CALL` $fname(a_1, \ldots, a_n)]$

$Trans_{Exps}(Exps, vtable, ftable) =$ `case` $Exps$ `of`	
Exp	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_1, [place])$
Exp , $Exps$	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_2, args) = Trans_{Exps}(Exps, vtable, ftable)$ $code_3 = code_1 ++ code_2$ $args_1 = place :: args$ $(code_3, args_1)$

Figure 2: Represented the Translating of an Expression.

Translating Structured Data

So far, the only values we have used are integers and Booleans. However, most programming languages provide floating-point numbers and structured values like arrays, records (structs), unions, lists or treestructures. We will now look at how these can be translated. We will first look at floats, then at one-dimensional arrays, multi-dimensional arrays and finally other data structures.

Floating Point Values

Floating-point values are, in a computer, typically stored in a different set of registers than integers. Apart from this, they are treated the same way we treat integer values: We use temporary variables to store intermediate expression results and assume the intermediate language has binary operators for floating-point numbers.

The register allocator will have to make sure that the temporary variables used for floating-point values are mapped to floating-point registers.

For this reason, it may be a good idea to let the intermediate code indicate which temporary variables hold floats. This can be done by giving them special names or by using a symbol table to hold type information.

Arrays

We extend our example language with one-dimensional arrays by adding the following productions:

Exp → Index

Stat → Index := Exp

Index → id[Exp

The index is an array element, which can be used the same way as a variable, either as an expression or as the left part of an assignment statement. We will initially assume that arrays are zero-based i.e. the lowest index is 0). Arrays can be allocated statically, i.e., at compile-time, or dynamically, i.e., at run-time. In the first case, the base address of the array the address at which index 0 is stored is a compile-time constant. In the latter case, a variable will contain the base address of the array. In either case, we assume that the symbol table for variables binds an array name to the constant or variable that holds its base address.

$Trans_{Exp}(Exp, vtable, ftable, place) =$ case Exp of	
$Index$	$(code_1, address) = Trans_{Index}(Index, vtable, ftable)$ $code_1 ++ [place := M[address]]$

$Trans_{Stat}(Stat, vtable, ftable) =$ case $Stat$ of	
$Index := Exp$	$(code_1, address) = Trans_{Index}(Index, vtable, ftable)$ $t = newvar()$ $code_2 = Trans_{Exp}(Exp, vtable, ftable, t)$ $code_1 ++ code_2 ++ [M[address] := t]$

$Trans_{Index}(Index, vtable, ftable) =$ case $Index$ of	
id$[Exp]$	$base = lookup(vtable, getname(\textbf{id}))$ $t = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, t)$ $code_2 = code_1 ++ [t := t * 4, t := t + base]$ $(code_2, t)$

Figure 3: Represented the Translation for One-dimensional Arrays.

Most modern computers are byte-addressed, while integers typically are 32 or 64 bits long. This means that the index used to access array elements must be multiplied by the size of the elements measured in bytes, e.g., 4 or 8, to find the actual offset from the base address. In the translation shown in Figure 3, we use 4 for the size of integers. We show only the new parts of the translation functions for Exp and Stat.

We use a translation function TransIndex for array elements. This returns a pair consisting of the code that evaluates the address of the array element and the variable that holds this address. When an array element is used in an expression, the contents of the address are transferred to the target variable using a memory-load instruction. When an array element is used on the left-hand side of an assignment, the right-hand side is evaluated, and the value of this is stored at the address using a memory-store instruction.

The address of an array element is calculated by multiplying the index by the size of the elements and adding this to the base address of the array. Note that the base can be either a

variable or a constant depending on how the array is allocated, see below, but since both are allowed as the second operator to a binop in the intermediate language, this is no problem.

Allocating Arrays

We've just touched on how arrays are allocated so far. One option, as previously indicated, is static allocation, in which the baseaddress and indeed the array's size are known at the time of compilation. Typically, the compiler has a large address space where it may allocate things that are statically allocated. The new object is simply constructed after the completion of the already allocated objects when this occurs. There are numerous methods for dynamic allocation. One is local allocation, where an array is allocated when a method or function is called and deallocated when it is finished. Usually, this indicates that the element is allocated on a stack and removed after the process is ended. The base addresses of locally allocated arrays are predictable offsets from the stack pinnacle or the frame pointer; see chapter 10 and may be derived from this at each arraylookup if the sizes of the arrays are fixed at build time.

If the widths of these arrays are provided at runtime, however, this does not function. Here, the base addresses of each array must be stored in a variable. When the array is allocated, the address is determined and then placed in the relevant variable. This may then be used as stated in the Trans-Index section above. The variable that will store the baseaddress at runtime will be tied to the arrayname at build time in the symbol table. The array will continue to exist until the conclusion of the programme or until it is manually deallocated if dynamic allocation is used worldwide. In this scenario, a global address space must be accessible for run-time allocation. This is often handled by the operating system, which manages memory-allocation requirements from all currently executing processes. Lack of memory may cause this allocation to fail, in which case the application would either need to crash or release memory elsewhere to make space. The software itself, which first requests a significant quantity of memory from the operating system and then manages this itself, is another option for controlling allocation. As a result, array allocation may be quicker than if an operating system call were required each time an array was created. In addition, it may enable automated array reclamation via trash collection for the application.

Translating Declarations

In the translation functions used in this chapter, we have several times required that “The symbol table must contain. It is the job of the compiler to ensure that the symbol tables contain the information necessary for translation. When a name (variable, label, type, etc.) is declared, the compiler must keep in the symbol-table entry for that name the information necessary for compiling any use of that name. For scalar variables (e.g., integers), the required information is the intermediate language variable that holds the value of the variable.

For array variables, the information includes the base-address and dimensions of the array. For records, it is the offsets for each field and the total size. If a type is given a name, the symbol table must for that name provide a description of the type, such that variables that are declared to be that type can be given the information they need for their symbol-table entries.

The exact nature of the information that is put into the symbol tables will depend on the translation functions that use these tables, so it is usually a good idea to write first the translation functions for uses of names and then translation functions for their declarations.

Example: Simple Local Declarations

We extend the statement language by the following productions:

Stat → Decl ; Stat

Decl → int id

Decl → int id[num]

We can, hence, declare integer variables and one-dimensional integer arrays for use in the following statement. An integer variable should be bound to a location in the symbol table, so this declaration should add such a binding to vtable. An array should be bound to a variable containing its base address.

Furthermore, code must be generated for allocating space for the array. We assume arrays are heap-allocated and that the intermediate-code variable HP points to the first free element of the upwards-growing heap.

The Translation of Simple Declarations shown in Figure 4.

Further Reading

$Trans_{Stat}(Stat, vtable, ftable) =$ `case` *Stat* `of`	
Decl ; $Stat_1$	$(code_1, vtable_1) = Trans_{Decl}(Decl, vtable)$ $code_2 = Trans_{Stat}(Stat_1, vtable_1, ftable)$ $code_1 ++ code_2$

$Trans_{Decl}(Decl, vtable) =$ `case` *Decl* `of`	
`int` **id**	$t_1 = newvar()$ $vtable_1 = bind(vtable, getname(\textbf{id}), t_1)$ $([], vtable_1)$
`int` **id**[**num**]	$t_1 = newvar()$ $vtable_1 = bind(vtable, getname(\textbf{id}), t_1)$ $([t_1 := HP, HP := HP + (4 * getvalue(\textbf{num}))], vtable_1)$

Figure 4: Represented the Translation of Simple Declarations.

High-level intermediate languages are often used in functional and logical languages, and they are frequently converted into lower-level intermediate code before being output as machine code.The Java Virtual Machine is another high-level intermediate language. For such complicated tasks as invoking virtual methods and generating new objects, this language has a single command. JVM's high-level design was selected for several reasons:

- By breaking up typical complicated processes into single instructions, the code is made smaller, which cuts down on the amount of time it takes to transmit over the Internet.
- JVM was designed from the start with interpretation in mind, and the intricate processes also served to lessen the burden of interpretation.
- A programme in the JVM is validated, or kind of type-checked, before being interpreted or translated further. When the code is high-level, this is simpler.

Compiler Design - Code Generation

With a few exceptions, the intermediate language we used in chapter 7 is fairly low-level and comparable to the kind of machine code seen on contemporary RISC processors.

- Whereas a CPU will have a limited number of registers, we have employed an infinite amount of variables.
- To call functions, a complicated CALL instruction was employed.On most processors, the conditional jump instruction has an online target label and simply jumps to the next instruction when the condition is false. In the intermediate language, the IF-THEN-ELSE instruction has two target labels.
- Any constant may serve as an operand for arithmetic instruction, as was previously thought. RISC processors often only accept tiny constants as operands. Register allocation addresses the issue of fitting a high number of variables into a limited number of registers.

Transforming each intermediate-language instruction into one or more machine-code instructions is the easiest way to create machine code from intermediate code. But it's often feasible to combine two or more intermediate-language commands into a single machine-code instruction. We will also quickly go through various improvements.

Exploiting Complex Instructions

Most instructions in our intermediate language are atomic, in the sense that each instruction corresponds to a single operation which cannot sensibly be split into smaller steps. The exceptions to this rule are the instructions IF-THEN-ELSE, which described how to handle it, and CALL. (CISC) Complex Instruction Set Computer processors like IA-32 have composite (i.e., non-atomic) instructions in abundance. And while the philosophy behind (RISC) Reduced Instruction Set Computer processors like MIPS and ARM advocates that machine-code instructions should be simple, most RISC processors include at least a few non-atomic instructions, typically for memory-access instructions. We will in this chapter use a subset of the MIPS instruction set as an example. A description of the MIPS instruction set can be found Appendix A of [39], which is available online [27]. If you are not already familiar with the MIPS instruction set, it would be a good idea to read the description before continuing. To exploit composite instructions, several intermediate-language instructions can be grouped and translated into a single machine-code instruction. For example, the intermediate-language instruction sequence:

```
t2 := t1 +116

t3 := M[t2]
```

can be translated into a single MIPS instruction:

```
lw r3, 116(r1)
```

Where, r1 and r3 are the registers chosen for t1 and t3, respectively. However, because the combined instruction doesn't save this value anywhere, combining the two instructions can only

be done if the value of the intermediate variable t2 is not likely to be required in the future. Therefore, we will need to know if a variable's contents are needed for future usage or whether they become useless after a certain use. Most of the temporary variables that even the compiler adds during the generation of intermediate code will be single-use and may be designated as such. A single-use variable will always be used at the last time. As an alternative, last-use data may very well be discovered by doing a livens analysis on the intermediate code. We'll merely assume for the foreseeable being that the intermediate code signifies the variable's latest usage. Assuming that this is the case, the final usage of any variable in the intermediate code is denoted by last, as in the case of the variable t, which is denoted by 't' last.

The next step is to translate each machine-code instruction into one or more instructions in an intermediate language. Since the goal is to locate sequences in the intermediate code that fit the pattern and replace these sequences with instances of the replacement, we refer to the sequence of intermediate language instructions as a pattern and the accompanying machine-code instruction as its replacement.

If the same variable is used in both the pattern and the replacement, it means that the commensurate intermediate-language variable/label constant name is copied to the computer processor, where it will represent a constant, a named register, or a machine-code label. Patterns that use variables like "k," "t," or "rd" can match any approximate language constants, variables, or labels.

Two-address Instructions

In the last section, we assumed that machine code is a three-address code, meaning that an instruction's destination register might be different from its two operand registers. However, it is typical for processors to employ a two-address code, in which the destination register and the first operand register are identical. We employ pattern or replacement pairs to address this, such as these:

$r_t := r_s$	movr_t , r_s
$r_t := r_t + r_s$	add r_t ,r_s
$r_d := r_s + r_t$	move r_d, r_s add r_d,r_t

In situations when the destination register is different from the first operand, this add copy instruction is used. We'll see that by assigning 'r_d' and 'r_s' to the same register, the register allocator may often get rid of the extra copy instruction.

Similar techniques may be used by processors that partition registers into data and address registers or integer and floating-point registers: Before operations, add instructions that copy to new registers, then allow register allocation and assign them to the appropriate types of registers, removing as many movements as feasible.

Optimizations

A compiler has three possible optimization locations: the source code, the abstract syntax, the intermediate code, and the machine code. Some optimizations may be unique to the source language or the machine language, but it makes sense to focus on the intermediate language since all compilers that use the same intermediate language may benefit from the optimizations. Additionally, the work required to do optimizations is reduced since the intermediate language is often simpler than both the source language and the machine language. Although many other optimizations optimizing compilers might use, we will just briefly touch on a few of them.

Common Sub expression Elimination

In the statement,**a[i]:= a[i]+2**, the address for a[i] is calculated twice. By saving the address in a temporary variable when the address is initially computed and using this variable rather than recalculating the address, this duplicate computation may be avoided. Simple approaches for common sub-expression removal only function on basic blocks, or lines of code without labels or jumps, while more sophisticated techniques may remove redundant computations even when they cross jumps.

Code Hoisting

If part of the computation inside a loop is independent of the variables that change inside the loop, it can be moved outside the loop and only calculated once. For example, in the loop

```
while (j < k) {
    sum = sum + a[i][j];
j++;
}
```

A large part of the address calculation for a[i][j] can be done without knowing j. This part can be moved outside the loop so it will only be calculated once. Note that this optimization ca not be done on the source-code level, as the address calculations are not visible there. For the same reason, the optimized version is not shown here. If k may be less than or equal to j, the loop body may never be entered and we may, hence, unnecessarily execute the code that was moved out of the loop. This might even generate a run-time error. Hence, we can unroll the loop once to:

```
if (j < k) {
    sum = sum + a[i][j];
j++;
    while (j < k) {
        sum = sum + a[i][j];
j++;
}
}
```

The loop-independent part(s) may now without risk be calculated in the unrolled part and reused in the non-unrolled part. Again, this optimization is not shown.

Constant Propagation

A variable may, at some points in the program, have a value that is always equal to a known constant. When such a variable is used in a calculation, this calculation can often be simplified after replacing the variable with the constant that is guaranteed to be its value. Furthermore, the variable that holds the results of this computation may now also become constant, which may enable even more compile-time reduction. Constant-propagation algorithms first trace the flow of constant values through the program, and then reduce calculations. More advanced methods also look at conditions, so they can exploit that after a test on, e.g., x = 0, x is, indeed, the constant 0.

Index-Check Elimination

Some compilers insert runtime checks to catch cases when an index is outside the bounds of the array. Some of these checks can be removed by the compiler. One way of doing this is to see if the tests on the index are subsumed by earlier tests or ensured by assignments. For example, assume that, in the loop shown above, 'a' is declared to be a 'k × k' array. This means that the entry test for the loop will ensure that "j" is always less than the upper bound on the array, so this part of the index test can be eliminated. If "j" is initialized to 0 before entering the loop, we can use this to conclude that we do not need to check the lower bound either.

Register Allocation

We freely employed as many variables as we deemed practical and simply converted intermediate language variables into machine language registers one-to-one. However, as processors have a limited amount of registers, register allocation is required to resolve this problem. To fit a lot of variables into a limited number of registers is the goal of register allocation. Usually, this may be accomplished by allowing many variables to share a single register, but sometimes, the processor may not have enough registers. Some of the variables in this situation need to be temporarily saved in memory. Before creating the machine code, register allocation may either be done in the intermediate language or the machine language.

In the latter scenario, registers are first identified by symbolic names in the machine code, which are then converted to register numbers by the register allocation. The benefit of doing register allocation in the intermediate language is that several target computers may readily utilize the same register allocator it just needs to be parameterized with the set of available registers. Allocating registers later, after the machine code has been written, could have benefits. We learned in chapter 8 that many instructions may be merged into a single instruction, which may cause a variable to vanish. There is no need to assign a register to this variable, but we shall do it in the intermediate language if register allocation is used. Additionally, when an instruction in an intermediate language needs to be converted into a series of instructions in machine code, the machine code might require an additional register or two for temporary values like the register required to store the outcome of the SLT instruction when converting a jump on to MIPS code.

There must thus always be at least one extra register available for use as temporary storage, according to the register allocator.

Liveness

This idea was previously alluded to in chapter 8 when we discussed the final applications of variables. In general, two variables may share a register if they are never both active at the same time in the program. Later, we'll define it more precisely. To identify whether a variable is live, we may use the following rules:

- A variable must be live at the beginning of an instruction if it utilizes the contents of a variable.
- If a variable is given a value in an instruction but isn't used as an operand, the variable is considered dead at the beginning of the instruction since the value it now contains isn't utilized before it is overwritten.
- If a variable is active after a command but is not given a value by that instruction, the variable is active at the beginning of the instruction as well.
- A variable is alive at the termination of an instruction if it is alive at the beginning of any of the procedures that follow it straightaway.

Liveness Analysis

We can formalize the above rules as equations over sets of variables. The process of solving these equations is called livens analysis, and will at any given point in the program determine which variables are live at this point. To better speak of points in a program, we number all instructions as in Figure 5. For every instruction in the program, we have a set of successors, i.e., instructions that may immediately follow the instruction during execution. We denote the set of successors to the instruction numbered i as succ[i]. We use the following rules to find succ[i]:

```
1:  a := 0
2:  b := 1
3:  z := 0
4:  LABEL loop
5:  IF n = z THEN end ELSE body
6:  LABEL body
7:  t := a + b
8:  a := b
9:  b := t
10: n := n - 1
11: z := 0
12: GOTO loop
13: LABEL end
```

Figure 5: Represented that the Example Program for Liveness Analysis and Register Allocation.

- The instruction numbered j (if any) that is listed just after instruction number 'i' is in succ[i], unless i is a GOTO or IF-THEN-ELSE instruction. If instructions are numbered consecutively, j = i+1.
- If instruction number 'i' is of the form GOTO l, (the number of) the instruction LABEL l is in succ[i]. Note that there in a correct program will be exactly one LABEL instruction with the label used by the GOTO instruction.
- If instruction i is IF p THEN l_t ELSE l_f, (the numbers of) the instructions LABEL l_t and LABEL l_f are in succ[i].

Note that we assume that both outcomes of an IF-THEN-ELSE instruction are possible. If this happens not to be the case i.e., if the condition is always true or always false), our liveness analysis may claim that a variable is live when it is in fact dead. This is no major problem, as the worst that can happen is that we use a register for a variable that is not going to be used after all. The converse claiming a variable dead when it is, in fact, live is worse, as we may overwrite a value that could be used later on, and hence get wrong results from the program.

Precise liveness information depends on knowing exactly which paths a program may take through the code when executed, and this is not possible to compute exactly it is a formally undecidable problem, so it is quite reasonable to allow imprecise results from a liveness analysis, as long as we err on the side of safety, i.e., calling a variable live unless we can prove it to be dead For every instruction i, we have a set gen[i], which lists the variables that may be read by instruction i and, hence, are live at the start of the instruction. In other words, gen[i] is the set of variables that instruction i generates liveness for. We also have a set kill[i] that lists the variables that may be assigned a value by the instruction. Table 1 shows which variables are in gen[i] and kill[i] for the types of instruction found in intermediate code. x, y and z are (possibly identical) variables and k denotes a constant.

Table 1: Represented that the Gen and Kill Sets.

Instruction i	$gen[i]$	$kill[i]$
LABEL l	$\emptyset$	$\emptyset$
$x := y$	$\{y\}$	$\{x\}$
$x := k$	$\emptyset$	$\{x\}$
$x :=$ **unop** y	$\{y\}$	$\{x\}$
$x :=$ **unop** k	$\emptyset$	$\{x\}$
$x := y$ **binop** z	$\{y, z\}$	$\{x\}$
$x := y$ **binop** k	$\{y\}$	$\{x\}$
$x := M[y]$	$\{y\}$	$\{x\}$
$x := M[k]$	$\emptyset$	$\{x\}$
$M[x] :=$ y	$\{x, y\}$	$\emptyset$
$M[k] :=$ y	$\{y\}$	$\emptyset$
GOTO l	$\emptyset$	$\emptyset$
IF x **relop** y THEN l_t ELSE l_f	$\{x, y\}$	$\emptyset$
$x :=$ CALL $f(args)$	$args$	$\{x\}$

For each instruction 'i', we use two sets to hold the actual liveness information: in[i] holds the variables that are live at the start of 'i', and out[i] holds the variables that are live at the end of i. We define these by the following equations:

in[i] = gen[i]∪(out[i] \ kill[i])

These equations are recursive. We solve these by fixed-point iteration, as shown in appendix A: We initialize all in[i] and out[i] to be empty sets and repeatedly calculate new values for these until no changes occur. This will eventually happen since we work with sets with finite support that is a finite number of possible values and because adding elements to the sets out[i] or in[j] on the right-hand sides of the equations cannot reduce the number of elements in the sets on the left-hand sides. Since we can only add elements to a set a limited number of times, each iteration will either leave all sets untouched, in which case we are finished, or it will add items to some set. It is also clear that the sets that are produced are a solution to the equation, and thus the last iteration effectively confirms the validity of the equations.

Table 2: Represented the succ, gen and kill for the Program.

i	$succ[i]$	$gen[i]$	$kill[i]$
1	2		a
2	3		b
3	4		z
4	5		
5	6, 13	n, z	
6	7		
7	8	a, b	t
8	9	b	a
9	10	t	b
10	11	n	n
11	12		z
12	4		
13			

We create a particular case: out[i], where I have no predecessor, is defined to be the set of all variables that are live after the programme. Similarly ill-defined if succ[i] is the empty set, which is normally the case for any instruction that terminates the programme. The Nth Fibonacci number is calculated using the procedure in Table 2. (where N is given as input by initialising n to N before execution). A will contain the Nth fibonacci number when the programme terminates by getting to instruction 13, making a live variable after the programme. We set out[13] = a since instruction 13 has no predecessors (succ[13] = 0/).

Equation defines all other out sets, while equation describes all in sets. All in and out sets are initialised to the empty set before iterating until a fixed point is reached. The sequence in which we process the instructions does not affect the iteration's eventual outcome, but it may have an impact on how fast we arrive at the fixedpoint. It is wise to do the evaluation in reverse instruction sequence and to compute out[i] before in[i], since the data passes through the programme backwards. In the example, this indicates that we will compute the sets in the order shown in each iteration. are utilised when a value originates from a higher instruction number because the most recent values are used when computing the right-hand sides, respectively.

We can see that the outcome from iteration 3 is identical to that from iteration 2. We see that n is live at program startup, which is reasonable given that n is anticipated to store program input.

When a variable that is not supposed to store input is active at the beginning of a program, it may be utilized in some program executions before it has been initialized. This is often regarded as a mistake since it may result in unforeseen outcomes and even security problems. Uninitialized variable warnings may be sent by certain compilers, while others may include instructions to initialize such variables to a default value usually 0.

CHAPTER 15

AN OVERVIEW OF THE SCANNING IN COMPILER DESIGN

Dr. Uthama Kumar A
Assistant Professor, Department of Data Science & Analytics, School of Sciences,
Jain (Deemed-to-be University), Bangalore-27, India
Email Id- uthamakumar.a@jainuniversity.ac.in

Scanning is the process of identifying tokens from the raw text source code of a program. At first glance, scanning might seem trivial after all, identifying words in a natural language is as simple as looking for spaces between letters. However, identifying tokens in source code requires the language designer to clarify many fine details, so that it is clear what is permitted and what is not. Most languages will have tokens in these categories:

- **i.** Keywords are words in the language structure itself, like while or class or true. Keywords must be chosen carefully to reflect the natural structure of the language, without interfering with the likely names of variables and other identifiers.
- **ii.** Identifiers are the names of variables, functions, classes, and other code elements chosen by the programmer. Typically, identifiers are arbitrary sequences of letters and possibly numbers. Some languages require identifiers to be marked with a sentinel (like the dollar sign in Perl) to clearly distinguish identifiers from keywords.
- **iii.** Numbers could be formatted as integers, or floating point values, or fractions, or in alternate bases such as binary, octal or hexadecimal. Each format should be clearly distinguished, so that the programmer does not confuse one with the other.
- **iv.** Strings are literal character sequences that must be clearly distinguished from keywords or identifiers. Strings are typically quoted with single or double quotes, but also must have some facility for containing quotations, newlines, and unprintable characters.
- **v.** Comments and whitespace are used to format a program to make it visually clear, and in some cases (like Python) are significant to the structure of a program.

When designing a new language, or designing a compiler for an existing language, the first job is to state precisely what characters are permitted in each type of token. Initially, this could be done informally by stating,for example, “An identifier consists of a letter followed by any number of letters and numerals.”, and then assigning a symbolic constant (TOKEN IDENTIFIER) for that kind of token. As we will see, an informal approach is often ambiguous, and a more rigorous approach is needed.

A Hand-Made Scanner

To keep things simple, we only consider just a few tokens: * for multiplication,! for logical-not, != for not-equal, and sequences of letters and numbers for identifiers. The basic approach is to

read one character at a time from the input stream (fgetc(fp)) and then classify it. Some single-character tokens are easy: if the scanner reads a * character, it immediately returns TOKEN MULTIPLY, and the same would be true for addition, subtraction, and so forth. However, some characters are part of multiple tokens. If the scanner encounters! that could represent a logical-not operation by itself, or it could be the first character in the != sequence representing not-equal-to.

Upon reading !, the scanner must immediately read the next character. If the next character is =, then it has matched the sequence != and returns **TOKEN NOT EQUAL.**

```
token_t scan_token( FILE *fp ) {
    int c = fgetc(fp);
    if(c=='*') {
        return TOKEN_MULTIPLY;
    } else if(c=='!') {
        char d = fgetc(fp);
        if(d=='=') {
            return TOKEN_NOT_EQUAL;
        } else {
            ungetc(d,fp);
            return TOKEN_NOT;
        }
    } else if(isalpha(c)) {
        do {
           char d = fgetc(fp);
        } while(isalnum(d));
        ungetc(d,fp);
        return TOKEN_IDENTIFIER;
    } else if ( . . . ) {
        . . .
    }
}
```

Figure 8.1: Represented that that the A Simple Hand Made Scanner Code.

But, if the character following! is something else, then the non-matching character needs to be put back on the input stream using ungetc, because it is not part of the current token. The scanner returns TOKEN NOT and will consume the put-back character on the next call to scan token.

In a similar way, once a letter has been identified by is alpha(c), then the scanner keeps reading letters or numbers, until a non-matching character is found. The non-matching character is put back, and the scanner returns TOKEN IDENTIFIER.

(We will see this pattern come up in every stage of the compiler: an unexpected item doesn't match the current objective, so it must be put back for later. This is known more generally as backtracking.)

As you can see, a hand-made scanner is rather verbose. As more token types are added, the code can become quite convoluted, particularly if tokens share common sequences of characters. It can also be difficult for a developer to be certain that the scanner code corresponds to the desired definition of each token, which can result in unexpected behavior on complex inputs. That said, for a small language with a limited number of tokens, a hand-made scanner can be an appropriate solution.

For a complex language with a large number of tokens, we need a more formalized approach to defining and scanning tokens. A formal approach will allow us to have a greater confidence that token definitions do not conflict and the scanner is implemented correctly. Further, a formal approach will allow us to make the scanner compact and high performance surprisingly, the scanner itself can be the performance bottleneck in a compiler, since every single character must be individually considered. The formal tools of regular expressions and finite automata allow us to state very precisely what may appear in a given token type. Then, automated tools can process these definitions, find errors or ambiguities, and produce compact, high performance code.

Regular Expressions

Regular expressions (REs) are a language for expressing patterns. They were first described in the 1950s by Stephen Kleene as an element of his foundational work in automata theory and computability. Today, REs are found in slightly different forms in programming languages (Perl), standard libraries (PCRE), text editors (vi), command-line tools (grep), and many other places. We can use regular expressions as a compact and formal way of specifying the tokens accepted by the scanner of a compiler, and then automatically translate those expressions into working code. While easily explained, REs can be a bit tricky to use, and require some practice in order to achieve the desired results.Here are a few examples using just the basic rules which is mention in Table 1. (Note that a finite RE can indicate an infinite set.)

Table 1: Represented that the Basic Rules of Regular Expression.

Regular Expression s	Language L(s)
hello	{ hello }
d(oli)g	{ dog,dig }
moo*	{ mo,moo,mooo,... }
(moo)*	{ ε,moo,moomoo,moomoomoo,... }
a(bla)*a	{ aa,aaa,aba,aaaa,aaba,abaa,... }

The syntax described so far is entirely sufficient to write any regular expression. But, it is also handy to have a few helper operations built on top of the basic syntax:

- s? indicates that s is optional.
- s? can be written as (s| ε)
- s+ indicates that s is repeated one or more times.
- s+ can be written as ss*
- [a-z] indicates any character in that range.
- [a-z] can be written as (a|b|...|z)
- [^x] indicates any character except one.
- [^x] can be written as Σ – x

Regular expressions also obey several algebraic properties, which make it possible to re-arrange them as needed for efficiency or clarity:

Associativity: a|(b|c) = (a|b)|c

Commutativity: a|b = b|a

Distribution: a(b|c) = ab|ac

Idempotency: a** = a*

Using regular expressions, we can precisely state what is permitted in a given token. Suppose we have a hypothetical programming language with the following informal definitions and regular expressions. For each token type, we show examples of strings that match (and do not match) the regular expression which is mention in below box.

Informal definition:	*An identifier is a sequence of capital letters and numbers, but a number must not come first.*
Regular expression:	`[A-Z]+([A-Z]|[0-9])*`
Matches strings:	`PRINT`
	`MODE5`
Does not match:	`hello`
	`4YOU`

Informal definition:	*A number is a sequence of digits with an optional decimal point. For clarity, the decimal point must have digits on both left and right sides.*
Regular expression:	`[0-9]+(.[0-9]+)?`
Matches strings:	`123`
	`3.14`
Does not match:	`.15`
	`30.`

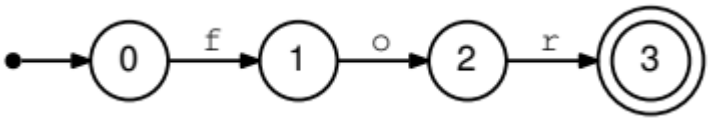
Informal definition:	*A comment is any text (except a right angle bracket) surrounded by angle brackets.*
Regular expression:	`<[^>]*>`
Matches strings:	`<tricky part>`
	`<<<<look left>`
Does not match:	`<this is an <illegal> comment>`

Finite Automata

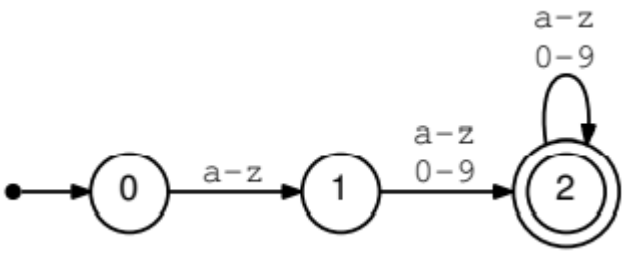
A finite automaton (FA) is an abstract machine that can be used to represent certain forms of computation. Graphically, an FA consists of a number of states (represented by numbered circles) and a number of edges (represented by labeled arrows) between those states. Each edge is labeled with one or more symbols drawn from an alphabet Σ.

The machine begins in a start state S_0. For each input symbol presented to the FA, it moves to the state indicated by the edge with the same label as the input symbol. Some states of the FA are known as accepting states and are indicated by a double circle. If the FA is in an accepting state after all input is consumed, then we say that the FA accepts the input. We say that the FA rejects the input string if it ends in a non-accepting state, or if there is no edge corresponding to the current input symbol.

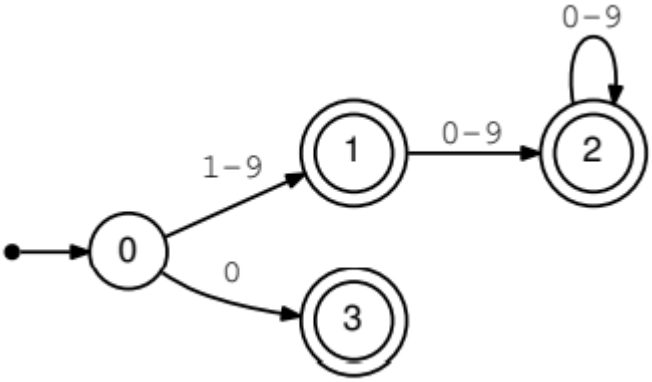
Every RE can be written as an FA, and vice versa. For a simple regular expression, one can construct an FA by hand. For example, here is an FA for the keyword for:



Here is an FA for identifiers of the form [a-z][a-z0-9]+



And here is an FA for numbers of the form ([1-9] [0-9]*) | 0



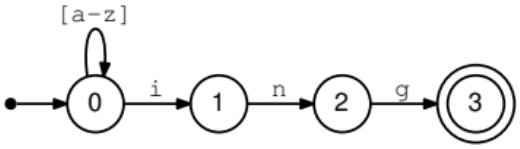
Deterministic Finite Automata

Each of these three examples is a deterministic finite automaton (DFA). A DFA is a special case of an FA where every state has no more than one outgoing edge for a given symbol. Put another way, a DFA has no ambiguity: for every combination of state and input symbol, there is exactly one choice of what to do next. Because of this property, a DFA is very easy to implement in software or hardware. One integer (c) is needed to keep track of the current state.

The transitions between states are represented by a matrix (M[s, i]) which encodes the next state, given the current state and input symbol. (If the transition is not allowed, we mark it with E to indicate an error.) For each symbol, we compute c = M[s, i] until all the input is consumed, or an error state is reached.

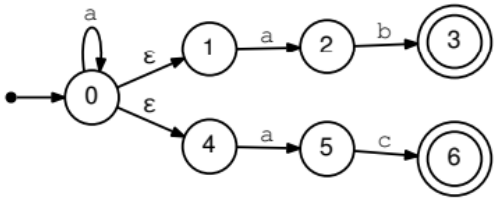
Nondeterministic Finite Automata

The alternative to a DFA is a nondeterministic finite automaton (NFA). An NFA is a perfectly valid FA, but it has an ambiguity that makes it somewhat more difficult to work with. Consider the regular expression [a-z]*ing, which represents all lowercase words ending in the suffix "ing." It can be represented with the following automaton:



Now consider how this automaton would consume the word sing. It could proceed in two different ways. One would be to move to state 0 on s, state 1 on i, state 2 on n, and state 3 on g. But the other, equally valid way would be to stay in state 0 the whole time, matching each letter to the [a-z] transition. Both ways obey the transition rules, but one results in acceptance, while the other results in rejection.

The problem here is that state 0 allows for two different transitions on the symbol i. One is to stay in state 0 matching [a-z] and the other is to move to state 1 matching i. Moreover, there is no simple rule by which we can pick one path or another. If the input is sing, the right solution is to proceed immediately from state zero to state one on i. But if the input is singing, then we should stay in state zero for the first ing and proceed to state one for the second. An NFA can also have anε(epsilon) transition, which represents the empty string. This transition can be taken without consuming any input symbols at all. For example, we could represent the regular expression a*(ablac) with this NFA:



This particular NFA presents a variety of ambiguous choices. From state zero, it could consume "a" and stay in state zero. Or, it could take a ε to state one or state four, and then consume an either way. There are two common ways to interpret this ambiguity:

- The crystal ball interpretation suggests that the NFA somehow "knows" what the best choice is, by some means external to the NFA itself. In the example above, the NFA would choose whether to proceed to state zero, one, or four before consuming the first character, and it would always make the right choice. Needless to say, this isn't possible in a real implementation.

- The many-worlds interpretation suggests that the NFA exists in all allowable states simultaneously. When the input is complete, if any of those states are accepting states, then the NFA has accepted the input. This interpretation is more useful for constructing a working NFA, or converting it to a DFA.

Let us use the many-worlds interpretation on the example above. Suppose that the input string is aaac. Initially the NFA is in state zero. Without consuming any input, it could take an epsilon transition to states one or four. So, we can consider its initial state to be all of those states simultaneously. Continuing on, the NFA would traverse these states until accepting the complete string aaac:

States	**Action**
0, 1, 4	consume a
0, 1, 2, 4, 5	consume a
0, 1, 2, 4, 5	consume a
0, 1, 2, 4, 5	consume c
6	accept

In principle, one can implement an NFA in software or hardware by simply keeping track of all of the possible states. But this is inefficient. In the worst case, we would need to evaluate all states for all characters on each input transition. A better approach is to convert the NFA into an equivalent DFA, as we show below:

Conversion Algorithms

Regular expressions and finite automata are all equally powerful. For every RE, there is an FA, and vice versa. However, a DFA is by far the most straightforward of the three to implement in software. In this section, we will show how to convert an RE into an NFA, then an NFA into a DFA, and then to optimize the size of the DFA as display in Figure 1.

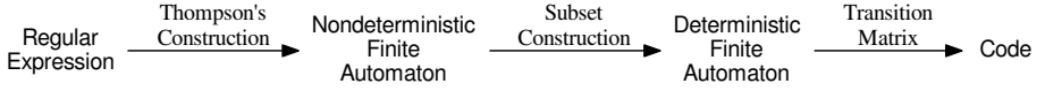
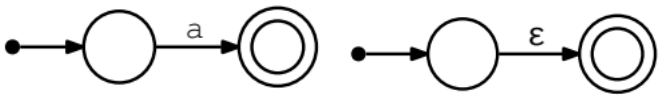


Figure 1: Display the Relationship between REs, NFAs, and DFAs.

Converting REs to NFAs

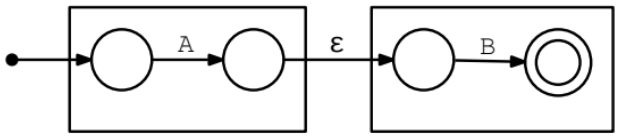
To convert a regular expression to a nondeterministic finite automaton, we can follow an algorithm given first by McNaughton and Yamada, and then by Ken Thompson. We follow the same inductive definition of regular expression as given earlier. First, we define automata corresponding to the base cases of REs:

The NFA for any character "a" is: The NFA for a ε transition is:



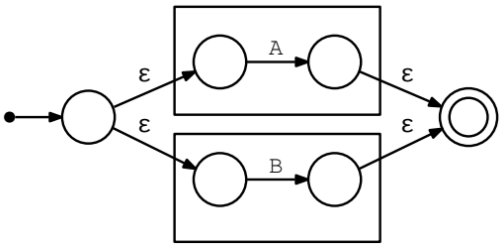
Now, suppose that we have already constructed NFAs for the regular expressions A and B, indicated below by rectangles. Both A and B have a single start state (on the left) and accepting state (on the right). If we write the concatenation of A and B as AB, then the corresponding NFA is simply A and B connected by an ε transition. The start state of A becomes the start state of the combination, and the accepting state of B becomes the accepting state of the combination:

The NFA for the concatenation AB is:



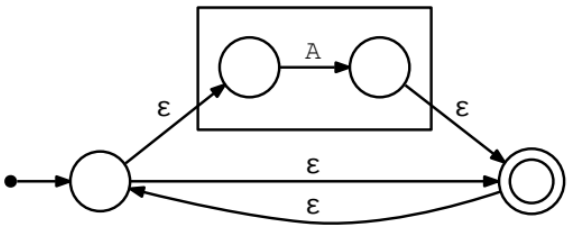
In a similar fashion, the alternation of A and B written as A|B can be ex-pressed as two automata joined by common starting and accepting nodes, all connected by ε transitions:

The NFA for the alternation A|B is:

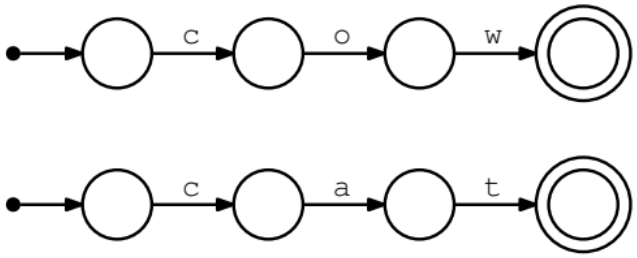


Finally, the Kleene closure A* is constructed by taking the automaton for A, adding starting and accepting nodes, then adding transitions to allow zero or more repetitions:

The NFA for the Kleene closure A* is:



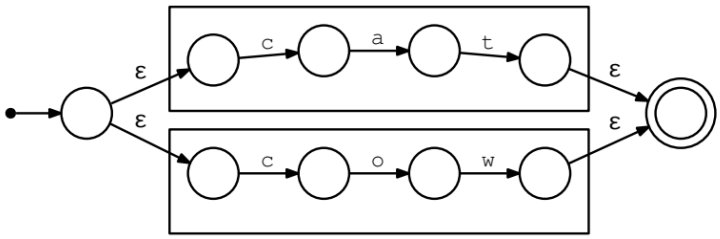
Example. Let's consider the process for an example regular expression a(cat|cow)*. First, we start with the innermost expression cat and assemble it into three transitions resulting in an accepting state. Then, do the same thing for cow, yielding these two FAs:



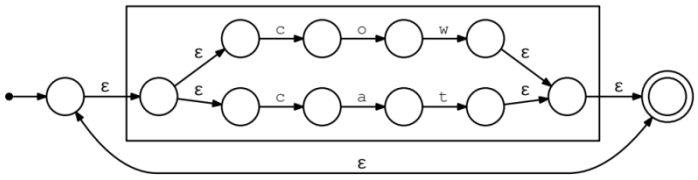
The alternation of the two expressions cat|cow is accomplished by adding a new starting and accepting node, with epsilon transitions. (The boxes are not part of the graph, but simply highlight the previous graph components carried forward.)

Conversion Algorithms

Then, the Kleene closure (cat|cow)* is accomplished by adding another starting and accepting state around the previous FA, with epsilon transitions between:



Finally, the concatenation of a (cat|cow)* is achieved by adding a single state at the beginning for a:



You can easily see that the NFA resulting from the construction algorithm, while correct, is quite complex and contains a large number of epsilon transitions. An NFA representing the tokens for a complete language could end up having thousands of states, which would be very impractical to implement. Instead, we can convert this NFA into an equivalent DFA.

Converting NFAs to DFAs

We can convert any NFA into an equivalent DFA using the technique of subset construction. The basic idea is to create a DFA such that each state in the DFA corresponds to multiple states in the NFA, according to the "many-worlds" interpretation. Suppose that we begin with an NFA consisting of states N and start state N0. We wish to construct an equivalent DFA consisting of

states D and start state D0. Each D state will correspond to multiple N states. First, we define a helper function known as the epsilon closure:

Epsilon closure.

ε–closure(n) is the set of NFA states reachable from NFA state n by zero or more ε transitions.

Now we define the subset construction algorithm. First, we create a start state D0 corresponding to the ε–closure(N0). Then, for each outgoing character c from the states in D0, we create a new state containing the epsilon closure of the state's reachable by c. More precisely:

Subset Construction Algorithm:

Given an NFA with states N and start state N_0, create an equivalent DFA with states D and start state D_0.

Let D_0 = ε–closure(N_0).

Add D_0 to a list.

While items remain on the list:

Let d be the next DFA state removed from the list.

For each character c in Σ:

Let T contain all NFA states N_k such that:

$N_j \in$ d and$N_j \rightarrow N_k$ Create new DFA state D_i = ε–closure (T)

If D_i is not already in the list, add it to the end.

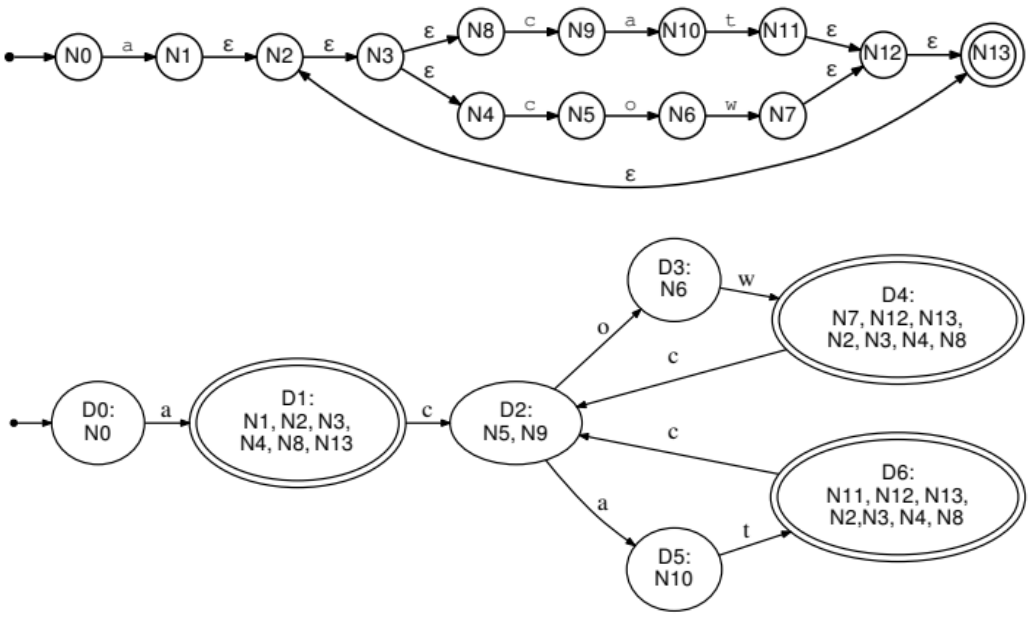


Figure 2: Representation of the Converting an NFA to a DFA via Subset Construction.

Example. Let's work out the algorithm on the NFA in Figure 2. This is the same NFA corresponding to the RE a (cat|cow)* with each of the states numbered for clarity.

1. Compute D0 which is ε–closure (N0). N0 has no ε transitions, so

   $D_0 = \{N_0\}$. Add D_0 to the work list.

2. Remove D0 from the work list. The character "a" is an outgoing transition from N0 to N1. ε –closure(N_1) = $\{N_1, N_2, N_3, N_4, N_8, N_{13}\}$ so add all of those to new state D1 and add D1 to the work list.

3. Remove D_1 from the work list. We can see that $N_4 \rightarrow N_5$ and $N_8 \rightarrow N_9$, so we create a new state $D_2 = \{N_5, N_9\}$ and add it to the worklist.

4. Remove D_2 from the work list. Both "a" and o are possible transitionsbecause of $N_5 \rightarrow N_6$ and $N_9 \rightarrow N_{10}$. So, create a new state D_3 for theo transition to N_6 and new state D5 for the "a" transition to N_{10}. Addboth D_3 and D_5 to the work list.

5. Remove D_3 from the work list. The only possible transition is $N_6 \rightarrow N_7$ so create a new state D_4 containing the ε –closure(N_7) and add itto the work list.

6. Remove D_5 from the work list. The only possible transition is $N_{10} \rightarrow N_{11}$ so create a new state D_6 containing ε–closure(N_{11}) and add it tothe work list.

7. Remove D_4 from the work list, and observe that the only outgoing transition c leads to states N_5 and N_9 which already exist as state D_2, so simply add a transition $D_4 \rightarrow D_2$.

8. Remove D_6 from the work list and, in a similar way, add $D_6 \rightarrow D_2$.

9. The work list is empty, so we are done.

Minimizing DFAs

The subset construction algorithm will definitely generate a valid DFA, but the DFA may possibly be very large (especially if we began with a complex NFA generated from an RE.) A large DFA will have a large transition matrix that will consume a lot of memory.

If it doesn't fit in L1 cache, the scanner could run very slowly. To address this problem, we can apply Hopcroft's algorithm to shrink a DFA into a smaller (but equivalent) DFA.

The general approach of the algorithm is to optimistically group together all possibly-equivalent states S into super-states T. Initially, we place all non-accepting S states into super-state T_0 and accepting states into super-state T_1.

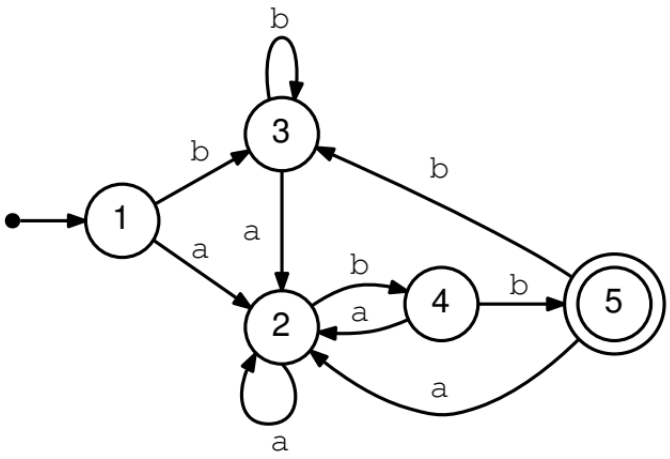
Then, we examine the outgoing edges in each state $s \in T_i$. If, a given character c has edges that begin in T_i and end in different super-states, then we consider the super-state to be inconsistent with respect to c. (Consider an impermissible transition as if it were a transition to TE, a super-state for errors.)

The super-state must then be split into multiple states that are consistent with respect to c. Repeat this process for all super-states and all characters $c \in \Sigma$ until no more splits are required.

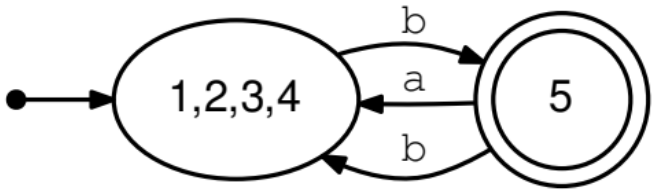
DFA Minimization Algorithm.
Given a DFA with states S, create an equivalent DFA with an equal or fewer number of states T.

First partition S into T such that:
$T_0 =$ non-accepting states of S.
$T_1 =$ accepting states of S.
Repeat:
$\forall T_i \in T$:
$\forall c \in \Sigma$:
if $T_i \xrightarrow{c}$ { more than one T state },
then split T_i into multiple T states
such that c has the same action in each.
Until no more states are split.

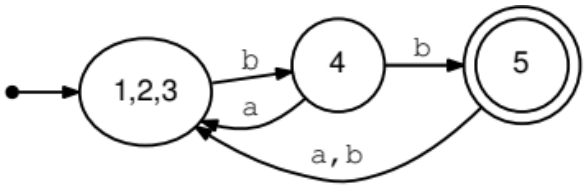
Example. Suppose we have the following non-optimized DFA and wish to reduce it to a smaller DFA:



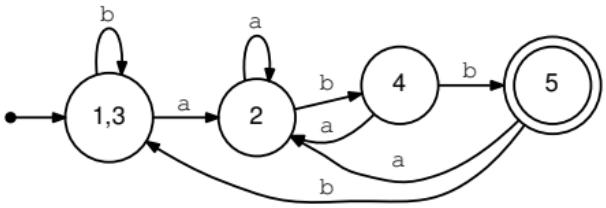
We begin by grouping all of non-accepting states 1, 2, 3, 4 into one super-state and the accepting state 5 into another super-state, like this:



Now, we ask whether this graph is consistent with respect to all possible inputs, by referring back to the original DFA. For example, we observe that, if we are in super-state (1,2,3,and 4) then an input of "a" always goes to state 2, which keeps us within the super-state. So, this DFA is consistent with respect to a. However, from super-state (1,2,3,and 4) an input of b can either stay within the super-state or go to super-state (5). So, the DFA is inconsistent with respect to b. To fix this, we try splitting out one of the inconsistent states (4) into a new super-state, taking the transitions with it:



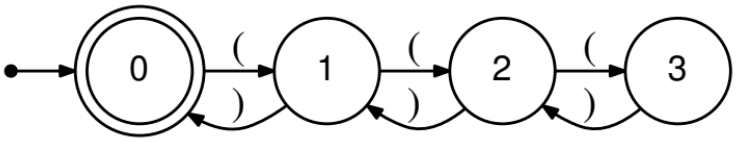
Again, we examine each super-state for consistency with respect to each input character. Again, we observe that super-state 1, 2, 3 is consistent with respect to a, but not consistent with respect to b because it can either lead to state 3 or state 4. We attempt to fix this by splitting out state 2 into its own super-state, yielding this DFA.



Again, we examine each super-state and observe that each possible input is consistent with respect to the super-state, and therefore we have the minimal DFA.

Limits of Finite Automata

Regular expressions and finite automata are powerful and effective at recognizing simple patterns in individual words or tokens, but they are not sufficient to analyze all of the structures in a problem. For example, could you use a finite automaton to match an arbitrary number of nested parentheses? It's not hard to write out an FA that could match, say, up to three pairs of nested parentheses, like this:



But the key word is arbitrary! To match any number of parentheses would require an infinite automaton, which is obviously impractical. Even if we were to apply some practical upper limit (say, 100 pairs) the automaton would still be impractically large when combined with all the other elements of a language that must be supported.

For example, a language like Python permits the nesting of parentheses () for precedence, curly brackets {} to represent dictionaries, and square brackets [] to represent lists. An automaton to match up to 100 nested pairs of each in arbitrary order would have 1,000,000 states!

So, we limit ourselves to using regular expressions and finite automata for the narrow purpose of identifying the words and symbols within a problem.

Using a Scanner Generator

Because a regular expression precisely describes all the allowable forms of a token, we can use a program to automatically transform a set of regular expressions into code for a scanner. Such a

program is known as a scanner generator. The program Lex, developed at AT&T, was one of the earliest examples of a scanner generator. Vern Paxson translated Lex into the C language to create Flex, which is distributed under the Berkeley license and is widely used in Unix-like operating systems today to generate scanners implemented in C or C++.

To use Flex, we write a specification of the scanner that is a mixture of regular expressions, fragments of C code, and some specialized directives. The Flex program itself consumes the specification and produces regular C code that can then be compiled in the normal way. The first section consists of arbitrary C code that will be placed at the beginning of scanner.c, like include files, type definitions, and similar things. Typically, this is used to include a file that contains the symbolic constants for tokens.

```
%{
    (C Preamble Code)
%}
    (Character Classes)
%%
    (Regular Expression Rules)
%%
    (Additional Code)
```

The second section declares character classes, which are symbolic shorthand's for commonly used regular expressions. For example, you might declare DIGIT [0-9]. This class can be referred to later as {DIGIT}.The third section is the most important part. It states a regular expression for each type of token that you wish to match, followed by a fragment of C code that will be executed whenever the expression is matched. In the simplest case, this code returns the type of the token, but it can also be used to extract token values, display errors, or anything else appropriate.

The fourth section is arbitrary C code that will go at the end of the scanner, typically for additional helper functions.

A peculiar requirement of Flex is that we must define a function yywrap() which returns 1 to indicate that the input is complete at the end of the file. If we wanted to continue scanning in another file, then yywrap() would open the next file and return 0.

The regular expression language accepted by Flex is very similar to that of formal regular expressions discussed above. The main difference is that characters that have special meaning with a regular expression (like parentheses, square brackets, and asterisks) must be escaped with a backslash or surrounded with double quotes. Also, a period (.) can be used to match any character at all, which is helpful for catching error conditions.

This specification describes just a few tokens: a single character addition (which must be escaped with a backslash), the while keyword, an identifier consisting of one or more letters, and a number consisting of one or more digits. As is typical in a scanner, any other type of character is an error, and returns an explicit token type for that purpose.

Flex generates the scanner code, but not a complete program, so you must write a main function to go with it. First, the main program must declare as extern the symbols it expects to use in the generated scanner code: yyin is the file from which text will be read, yylex is the function that implements the scanner, and the array yytext contains the actual text of each token discovered. Finally, we must have a consistent definition of the token types across the parts of the program, so into token.h we put an enumeration describing the new type token t. This file is included in both scanner.flex and main.c.

Scanner.flex is converted into scanner.c by invoking flex -o scanner.cscanner.flex. Then, both main.c and scanner.c are compiled to produce object files, which are linked together to produce the complete program.

Practical Considerations

Handling keywords. In many languages, keywords such as while or if would otherwise match the definitions of identifiers, unless specially handled. There are several solutions to this problem. One is to enter a regular expression for every single keyword into the Flex specification. These must precede the definition of identifiers, since Flex will accept the first expression that matches. Another is to maintain a single regular expression that matches all identifiers and keywords. The action associated with that rule can compare the token text with a separate list of keywords and return the appropriate type. Yet another approach is to treat all keywords and identifiers as a single token type, and allow the problem to be sorted out by the parser. This is necessary in languages like PL/1, where identifiers can have the same names as keywords, and are distinguished by context.

Tracking Source Locations

In later stages of the compiler, it is useful for the parser or type checker to know exactly what line and column number a token was located at, usually to print out a helpful error message. (“Undefined symbol spider at line 153.”) This is easily done by having the scanner match newline characters, and increase the line count (but not return a token) each time one is found.

Cleaning Tokens

Strings, characters, and similar token types need to be cleaned up after they are matched. For example, "hello\n" needs to have its quotes removed and the backslash-n sequence converted to a literal newline character. Internally, the compiler only cares about the actual contents of the string. Typically, this is accomplished by writing a function string clean in the post amble of the Flex specification. The function is invoked by the matching rule before returning the desired token type.

Constraining Tokens

Although regular expressions can match tokens of arbitrary length, it does not follow that a compiler must be prepared to accept them. There would be little point to accepting a 1000-letter identifier, or an integer larger than the machine’s word size. The typical approach is to set the maximum token length (YYLMAX in flex) to a very large value, then examine the token to see

if it exceeds a logical limit in the action that matches the token. This allows you to emit an error message that describes the offending token as needed.

Error Handling

The easiest approach to handling errors or invalid input is simply to print a message and exit the program. However, this is unhelpful to users of your compiler if there are multiple errors, it's (usually) better to see them all at once. A good approach is to match the minimum amount of invalid text (using the dot rule) and return an explicit token type indicating an error. The code that invokes the scanner can then emit a suitable message, and then ask for the next token.

Related Question for Practice

1. What is compiler design?
2. Enlist various types of compiler?
3. What tools are used for compiler construction?
4. What is bootstrapping in compiler design?
5. What is yacc?
6. What is Relocatable Machine Code?
7. What is Lexical analysis?
8. What is Linker?
9. What is the List of compiler construction tools?
10. What is the regular expression of identifier?
11. Write the overview of the structure of a typical compiler?
12. Briefly explain what a semantic analyzer does?

Reference of Books for Further Reading

1. Alfred V Aho and Ravi Sethi “Compilers: Principles, Techniques and Tools”.

2. Michael L Scott “Programming Language Pragmatics”.

3. Andrew W Appel “Modern Compiler Implementation in C/Java”.

4. Keith D Cooper and Linda Torczon “Engineering a Compiler”.

5. Allen I Holob “Compiler Design in C”.
